7 天学会 spring cloud 教程

按照官方的话说：Spring Cloud 为开发者提供了在分布式系统（如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性 Token、全局锁、决策竞选、分布式会话和集群状态）操作的开发工具。最关键的是它足够简单，一般的开发人员只需要几天时间就可以学会它的基本用法。

本 Spring Cloud 7 天系列教程，包括 7 个例子和相关短文，都是最简单的用法，也是默认最基本的用法，在实际生产环境中也可以用上，当然是初步使用。

项目开源地址：http://git.oschina.net/zhou666/spring-cloud-7simple

7 个例子包括：

1）一个基本的 spring boot 应用。

2）分布式配置管理服务端

3）分布式配置管理客户端（微服务应用）

4）服务注册服务端

5）服务注册发现客户端（微服务应用）

6）spring boot 风格的 web 前端应用

7）使用 docker 发布应用

7 天学习周期如下：

第 1 天：查看 spring boot 官方文档，实现及实验 spring boot 应用。

第 2 天：熟读 spring cloud 官方文档配置管理部分并熟悉配置管理相关概念。

第 3 天：熟悉 Git 概念，并上传配置文件到 Git 服务器，最后实现分布式配置管理。

第 4 天：熟读 spring cloud 官方文档服务注册部分，实现服务注册及发现。

第 5 天：熟读 spring cloud 官方文档剩余部分，并实现断路器。

第 6 天：深入 spring boot 相关概念，使用 angularJS 实现 web 前端应用。

第 7 天：了解 docker 概念，并结合 spring boot 搭建一个 docker 应用。

spring cloud 教程之使用 spring boot 创建一个应用

《7 天学会 spring cloud》第一天，熟悉 spring boot，并使用 spring boot 创建一个应用。

Spring Boot 是 Spring 团队推出的新框架，它所使用的核心技术还是 Spring 框架，主要是 Spring 4.x，所以如果熟悉 spring 4 的人，能够更快的接受和学会这个框架。Spring boot 可以看做是在 spring 框架基础上再包了一层，这一层包含方便开发者进行配置管理和快速开发的模块，以及提供了一些开箱即用的工具，比如监控等。

Spring Boot 官方文档有中文翻译版：

https://github.com/qibaoguang/Spring-Boot-Reference-Guide

要实现一个 spring boot 开发环境和传统的应用没有区别，这里用的是：

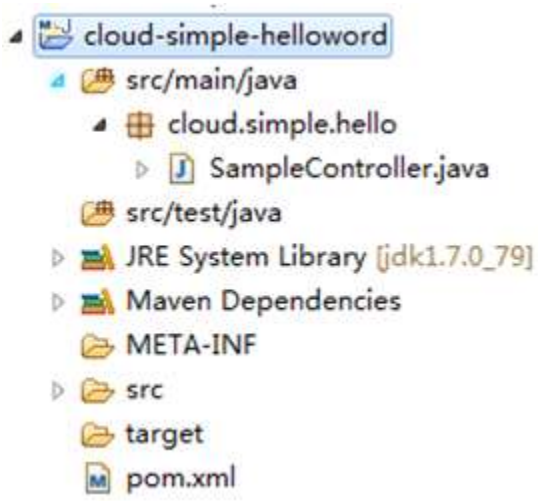
IDE：myeclipse 10

JDK：jdk1.7

WINDOWS：mvn 3

在桌面 windows 环境里需要单独安装方便我们使用命令行进行打包和操作。Eclipse 环境里也需要安装 mvn 插件，当然如果使用的是 myeclipse，那么自带的 mvn 环境就足够了。以下是建立 spring boot helloworld 应用的步骤。注意这是一个 web 应用，使用了嵌入式的 tomcat。

1） 首选建立一个最简单的 maven 工程，如下图：



这个应用只有一个类，编写代码如下：

```
1  package cloud.simple.hello;
2  import org.springframework.boot.*;
3  import org.springframework.boot.autoconfigure.*;
4  import org.springframework.stereotype.*;
5  import org.springframework.web.bind.annotation.*;
6
7  @Controller
8  @SpringBootApplication
9  public class SampleController  {
10
11     @ResponseBody
12     @RequestMapping(value = "/")
13     String home() {
14         return "Hello World!";
15     }
16
       public static void main(String[] args) throws Exception {
```

```
        SpringApplication.run(SampleController.class, args);
    }

}
```

@SpringBootApplication 相当于@Configuration、@EnableAutoConfiguration 和 @ComponentScan，你也可以同时使用这 3 个注解。其中@Configuration、@ComponentScan 是 spring 框架的语法，在 spring 3.x 就有了，用于代码方式创建配置信息和扫描包。@EnableAutoConfiguration 是 spring boot 语法，表示将使用自动配置。你如果下载了 spring boot 源码，就会看到 spring boot 实现了很多 starter 应用，这些 starter 就是一些配置信息（有点类似于 docker，一组环境一种应用的概念），spring boot 看到引入的 starter 包，就可以计算如果自动配置你的应用。

2） 配置 pom.xml

这个应用不需要配置文件，写完 class 后就可以直接配置 pom.xml 文件了，当然先配置 pom.xml 也一样。Pom 文件配置如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xs
    <modelVersion>4.0.0</modelVersion>
    <!-- spring boot 基本环境 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
    </parent>

    <groupId>spring.boot</groupId>
    <artifactId>cloud-simple-helloword</artifactId>
    <version>0.0.1</version>
    <packaging>jar</packaging>
    <name>cloud-simple-helloword</name>
    <dependencies>
        <!--web 应用基本环境配置 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
            </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

```
28
29
30
31
```

pom 文件配置完，你就可以运行应用了，点击 F11，或者在 SampleController 类右键“Run Java Application”就可以看到应用启动并运行了。

此时在浏览器输入 http://localhost:8080/，你会看到 helloworld 字样，这是一个 web 应用，使用了嵌入式的 tomcat。

在 pom 配置中我们仅仅使用了 spring-boot-starter-web 依赖，spring boot 会根据此依赖下载相关 jar 包并初始化基本的运行环境，比如说绑定端口 8080 等。

spring boot 封装所有配置信息为键值类型，你想改变默认配置，只需要向应用传入这个键值对就可以，比如我们想改变绑定端口为 8081，那么你在 main 方法里传入“—server.port=8081”即可，或者干脆使用：

```
SpringApplication.run(SampleController.class, "--server.port=8081");
```

3） 部署 spring boot 应用

要部署运行 spring boot 应用，首选要打包 spring boot 应用，你在 pom 文件中看到的 spring-boot-maven-plugin 插件就是打包 spring boot 应用的。

进入工程目录运行 mvn package，如：

```
D:\cloud-simple-helloword>mvn package
```

打包过后就可以进入 target 目录使用 java 原生命令执行这个应用了。

```
D:\cloud-simple-helloword\target>java -jar cloud-simple-helloword-0.0.1.jar --server.port=8081
```

如此，你就看到一个基于 jar 包的 web 应用启动了。

Spring boot 提供的一些开箱即用的应用非常容易使用，比如监控，你只需要在 pom 文件中引入：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

引入之后，spring boot 是默认开启监控的，运行应用你可以在浏览器中输入：

http://localhost:8080/health

就可以看到默认的监控信息了：

```
{"status":"UP","diskSpace":{"status":"UP","total":161067397120,"free":91618398208,"threshold":10485760}}
```

信息包括程序执行状态以及基本的磁盘信息。

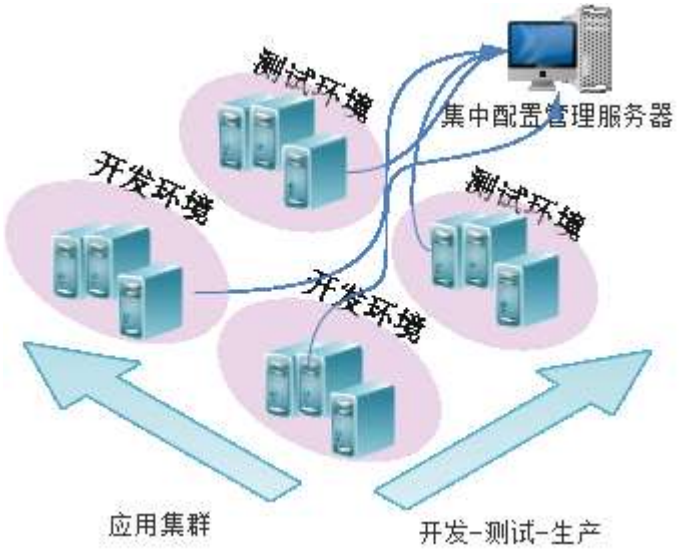
使用 spring cloud 实现分布式配置管理

《7 天学会 spring cloud 系列》之创建配置管理服务器及实现分布式配置管理应用。

本文涉及到的项目：

- 开源项目：http://git.oschina.net/zhou666/spring-cloud-7simple
- cloud-config-repo：配置文件存放的文件夹
- cloud-simple-service：一个使用 mybatis 的数据库应用

分布式配置管理应该是分布式系统和微服务应用的第一步。想象一下如果你有几十个服务或应用需要配置，而且每个服务还分为开发、测试、生产等不同维度的配置，那工作量是相当大的，而且还容易出错。如果能把各个应用的配置信息集中管理起来，使用一套机制或系统来管理，那么将极大的提高系统开发的生产效率，同时也会提高系统开发环境和生产环境运行的一致性。



在传统开发中我们往往需要自己开发“配置管理服务器”，你可以使用 redis、ldap、zookeeper、db 等来存放统一配置信息，然后开发一个管理界面来进行管理。传统的做法没什么问题，和 spring cloud 所提供的配置管理方案相比，就是前者需要自己开发，而后者直接简单使用现成的组件即可。当然还有很重要的一点，spring 配置管理模块由于是 spring boot 核心来实现的，因此做了大量的工作，**可以把一些启动参数进行外部配置**，这在传统的方案中是很难办到的，因为涉及到要改写第三方组件的问题，难度很大。比如 web 应用的绑定端口，传统应用只能在 tomcat 配置文件里改，而 spring cloud 却可以放到远程，类似的还有数据库连接、安全框架配置等。

要使用 spring cloud 分布式配置文件总体上分为 3 个大的步骤，首选你需要创建存放配置文件的仓库，然后创建一个配置文件服务器，该服务器将配置文件信息转化为 rest 接口数据，然后创建一个应用服务，该服务演示使用分布式配置文件信息。

1）创建配置文件存放仓库

Spring cloud 使用 git 或 svn 存放配置文件，默认情况下使用 git，因此你需要安装 git 私服或者直接使用互联网上的 github 或者 git.oschina，这里推荐使用 git.oschina。本文示例使用的是 git.oschina，创建好 git 工程后，也就是文章开头所提到的工程，在此工程再创建一个文件夹 cloud-config-repo 来存放配置文件。然后创建两个配置文件：

- cloud-config-dev.properties
- cloud-config-test.properties

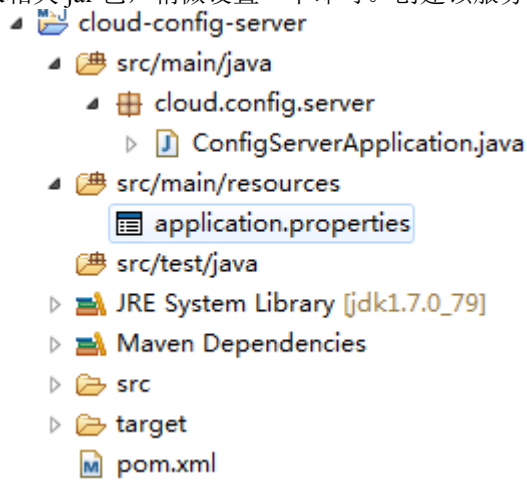
这两个文件分别对应开发环境和测试环境所需要的配置信息，配置信息如下：

```
mysqldb.datasource.url=jdbc\:mysql\://10.0.12.170\:3306/test?useUnicode\=true&characterEncoding\=utf-8
mysqldb.datasource.username=csst
mysqldb.datasource.password=csst
logging.level.org.springframework.web:DEBUG
```

配置信息提供了数据库连接参数等，这是因为后面的应用服务中使用到了数据库。

2）创建 spring cloud 配置服务器

配置文件仓库创建好了后，就需要创建配置管理服务器，如前所述该服务器只是将配置文件转换为 rest 接口服务，不做其它用途。这个服务器的功能也是 spring cloud 提供的，所以我们只需要引入相关 jar 包，稍微设置一下即可。创建该服务应用，你需要首选创建一个空的 maven 工程：



然后在这个工程中增加一个类，命名为：ConfigServerApplication，代码如下：

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
  public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

可以看到，我们只需要用@EnableConfigServer 激活该应用为配置文件服务器即可。如此以来该应用启动后就会完成前面提到的功能，即：读取远程配置文件，转换为 rest 接口服务。

当然，需要配置远程配置文件读取路径，在 application.properties 中：

```
server.port=8888
spring.cloud.config.server.git.uri=https://git.oschina.net/zhou666/spring-cloud-7simple.git
spring.cloud.config.server.git.searchPaths=cloud-config-repo
```

其中 server.port 是配置当前 web 应用绑定 8888 端口，git.uri 指定配置文件所在的 git 工程路径，searchPaths 表示将搜索该文件夹下的配置文件（我们的配置文件放在 spring-cloud-7simple 这个工程的 cloud-config-repo 文件夹下）。

最后，还需要在 pom 文件中增加配置服务器的相关依赖：

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

如此以来，配置文件服务器就建立好了，可以直接启动了，服务端口是 8888，应用只需要绑定改服务器的 uri 和端口号就可以拿到配置信息了。

3） 创建一个服务使用该远程配置

现在可以创建一个服务使用该远程配置了，你可以在远程配置中定义一个简单的自定义信息，比如：

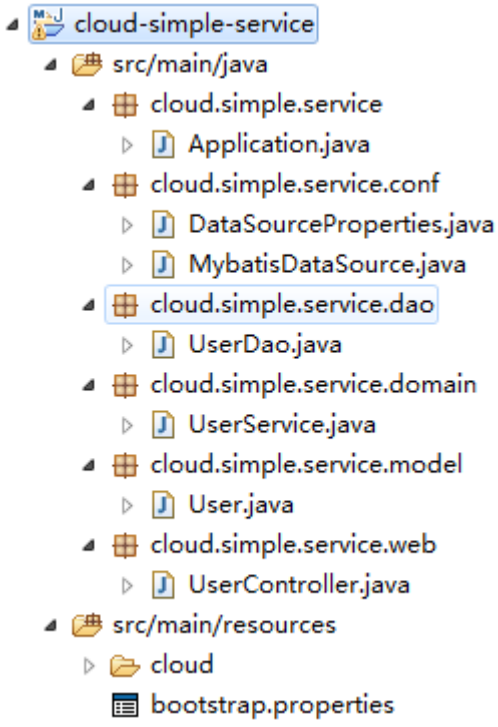
```
my.message=helloword
```

然后使用前面我们提到的 spring boot helloworld 应用来读取这个信息。当然，限于篇幅我们直接使用比较复杂的一个服务来演示这个配置管理器的使用，这个服务需要用到数据库访问，数据库访问层我们使用的是 mybaits，数据表只有一个，DDL 如下：

```
CREATE TABLE `user` (
```

```
`id` varchar(50) NOT NULL DEFAULT '',
`username` varchar(50) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

创建好数据表后，回到我们的应用服务：



该服务使用 DataSourceProperties 封装了 mybatis 加载配置信息。要拿到远程配置信息，需要设置配置管理服务器地址，该配置设置在：

bootstrap.properties

该配置文件信息如下：

```
spring.cloud.config.uri=http://127.0.0.1:${config.port:8888}
spring.cloud.config.name=cloud-config
spring.cloud.config.profile=${config.profile:dev}
```

其中 config.uri 指定远程加载配置信息的地址，就是前面我们刚建立的配置管理服务器的地址，绑定端口 8888，其中 config.port:8888，表示如果在命令行提供了 config.port 参数，我们就用这个端口，否则就用 8888 端口。config.name 表示配置文件名称，查看我们前面创建配置文件，是这个名称：

cloud-config-dev.properties

可以分成两部分: {application}- {profile}.properties

所以我们配置 config.name 为 cloud-config，config.profile 为 dev，其中 dev 表示开发配置文件，配置文件仓库里还有一个测试环境的配置文件，切换该配置文件只需要将 dev 改为 test 即可，当然这个参数也可以由启动时命令行传入，如：

```
java -jar cloud-simple-service-1.0.0.jar --config.profile =test
```

此时应用就会加载测试环境下的配置信息。

Spring cloud 实现服务注册及发现

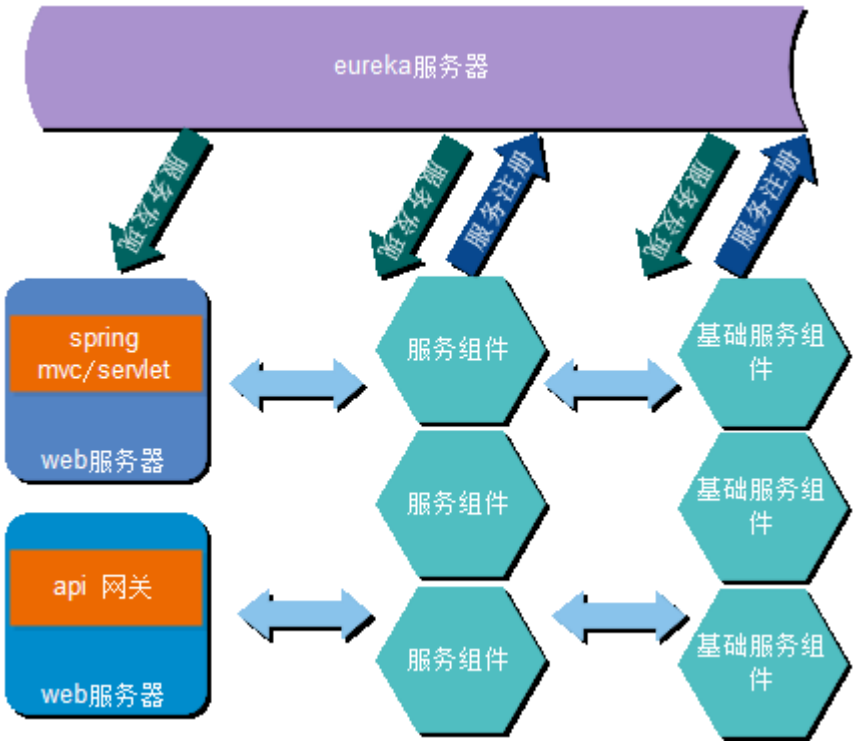
服务注册与发现对于微服务系统来说非常重要。有了服务发现与注册，你就不需要整天改服务调用的配置文件了，你只需要使用服务的标识符，就可以访问到服务。

本文属于《7 天学会 spring cloud 系列》之四，关注服务注册与发现，本文涉及到的项目：

- 开源项目：http://git.oschina.net/zhou666/spring-cloud-7simple

- cloud-eureka-server：eureka 注册服务器
- cloud-simple-service：一个使用 mybatis 的数据库应用，服务端

服务注册管理器原理如下图所示：

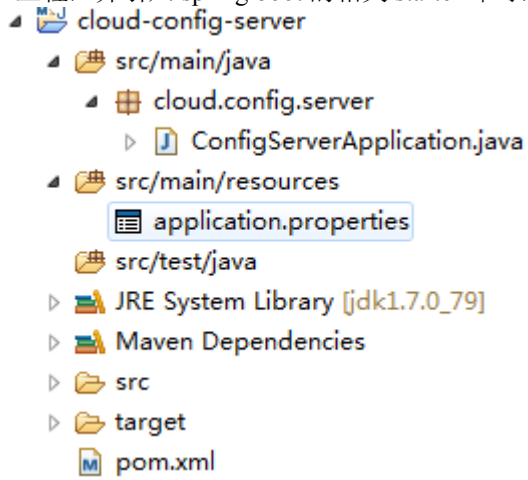


所有的服务端及访问服务的客户端都需要连接到注册管理器（eureka 服务器）。服务在启动时会自动注册自己到 eureka 服务器，每一个服务都有一个名字，这个名字会被注册到 eureka 服务器。使用服务的一方只需要使用该名字加上方法名就可以调用到服务。

Spring cloud 的服务注册及发现，不仅仅只有 eureka，还支持 Zookeeper 和 Consul。默认情况下是 eureka，spring 封装了 eureka，使其非常简单易用，只需要比传统应用增加一行代码就可以使用了，这一行代码就是一个注解。我们按以下步骤实现服务注册和发现功能。

1）首选需要建立 eureka 服务器

创建 spring cloud eureka 服务器和创建之前那个配置文件服务器类似，你只需要创建一个空的 maven 工程，并引入 spring boot 的相关 starter 即可，然后创建一个近乎空的执行类，工程如下图：



在 EurekaServer 类中我们加入如下代码：

@SpringBootApplication

```
@EnableEurekaServer
public class EurekaServer {
    public static void main(String[] args) {
    SpringApplication.run(EurekaServer.class, args);
  }
}
```

可以看到只需要使用@EnableEurekaServer 注解就可以让应用变为 Eureka 服务器，这是因为 spring boot 封装了 Eureka Server，让你可以嵌入到应用中直接使用。至于真正的 EurekaServer 是 Netflix 公司的开源项目，也是可以单独下载使用的。

在 application.properties 配置文件中使用如下配置：

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

其中 server.port 配置 eureka 服务器端口号。Eureka 的配置属性都在开源项目 spring-cloud-netflix-master 中定义（spring boot 连文档都没有，只能看源码了），在这个项目中有两个类 EurekaInstanceConfigBean 和 EurekaClientConfigBean，分别含有 eureka.instance 和 eureka.client 相关属性的解释和定义。从中可以看到，registerWithEureka 表示是否注册自身到 eureka 服务器，因为当前这个应用就是 eureka 服务器，没必要注册自身，所以这里是 false。fetchRegistry 表示是否从 eureka 服务器获取注册信息，同上，这里不需要。defaultZone 就比较重要了，是设置 eureka 服务器所在的地址，查询服务和注册服务都需要依赖这个地址。

做完这些后当然，还要改一下 pom 文件，增加 eureka-server 的 starter 即可：

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

如此 eureka 服务器就完成了，在命令行启动就可以了。

2）让服务使用 eureka 服务器

让服务使用 eureka 服务器，只需添加@EnableDiscoveryClient 注解就可以了。回到我们在上篇文章中实现的 cloud-simple-service 微服务应用。在 main 方法所在的 Application 类中，添加 @EnableDiscoveryClient 注解。然后在配置文件中添加：

```
eureka.client.serviceUrl.defaultZone=http\://localhost\:8761/eureka/
spring.application.name=cloud-simple-service
```

其中 defaultZone 是指定 eureka 服务器的地址，无论是注册还是发现服务都需要这个地址。application.name 是指定进行服务注册时该服务的名称。这个名称就是后面调用服务时的服务标识符（这是服务发现的功能，我们在后面章节具体介绍）。当然，pom 文件也需要增加：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

如此以来该服务启动后会自动注册到 eureka 服务器。如果在该服务中还需要调用别的服务，那么直接使用那个服务的服务名称加方法名构成的 url 即可，具体我们将在下章结合 ui 端的应用具体介绍。

综合使用 spring cloud 技术实现微服务应用

在之前的章节，我们已经实现了配置服务器、注册服务器、微服务服务端，实现了服务注册与发现。这一章将实现微服务的客户端，以及联调、实现整个 spring cloud 框架核心应用。

本文属于《7 天学会 spring cloud 系列》之五，涉及到的项目包括：

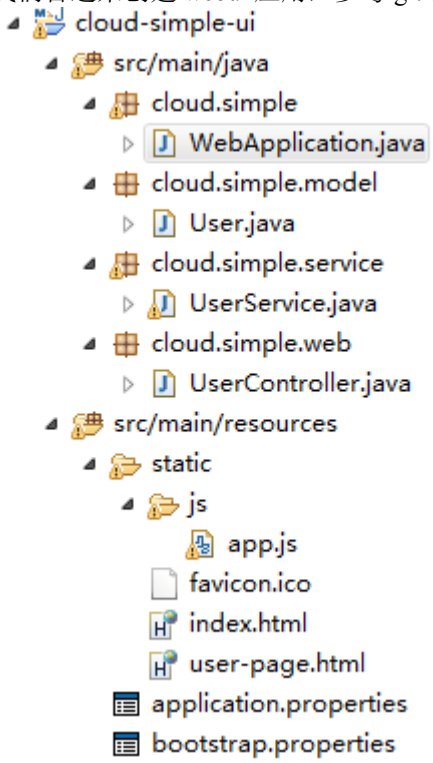
开源项目：http://git.oschina.net/zhou666/spring-cloud-7simple

- cloud-config-server：配置服务器
- cloud-eureka-server：eureka 注册服务器

- cloud-simple-service：一个使用 mybatis 的数据库应用，服务端
- cloud-simple-ui：webui 客户端

我们先来看看如何实现 webui 客户端。在 spring boot 中，已经不推荐使用 jsp，所以你如果使用 jsp 来实现 webui 端，将会很麻烦。这可能跟现在的开发主流偏重移动端有关，跟微服务有关，跟整个时代当前的技术需求有关。单纯以 html 来作为客户端，有很多好处，比如更利于使用高速缓存；使后台服务无状态话，更利于处理高并发；更利于页面作为服务，小服务组合成大服务等。

我们首选来创建 webui 应用，参考 git cloud-simple-ui 工程：



这个应用包括前端 html 页面，还包括一个后台 controller 浅层。这是一个前端应用，为什么要包括 controller 浅层？这是因为这个 controller 浅层是用来做服务组合的，因为一个页面可能涉及到调用多个服务，而这些服务又有可能涉及事务和并发问题，所以还是需要靠 java 来完成。当然，也可以单独编写一个 api gateway 来实现这一层，可以使用 go、node.js 语言等，也可以使用 java/netty（主流还是 java，因为开发部署效率高）。

Controller 层的 WebApplication 就是这个应用的入口，代码如下：

```
@SpringBootApplication
@EnableEurekaClient
@EnableHystrix
public class WebApplication {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(WebApplication.class, args);
    }
}
```

这些注解大都在前面提到过用法：

@SpringBootApplication 相当于@Configuration、@EnableAutoConfiguration、@ComponentScan 三个注解合用。

@EnableEurekaClient 配置本应用将使用服务注册和服务发现，注意：注册和发现用这一个注解。

@EnableHystrix 表示启用断路器，断路器依赖于服务注册和发现。

Controller 层的 service 包封装了服务接口访问，UserService 类代码如下：

```
@Service
public class UserService {
      @Autowired
      RestTemplate restTemplate;
      final String SERVICE_NAME="cloud-simple-service";
       @HystrixCommand(fallbackMethod = "fallbackSearchAll")
      public List<User> searchAll() {
        return restTemplate.getForObject("http://"+SERVICE_NAME+"/user", List.class);
      }
      private List<User> fallbackSearchAll() {
      System.out.println("HystrixCommand fallbackMethod handle!");
      List<User> ls = new ArrayList<User>();
      User user = new User();
      user.setUsername("TestHystrix");
      ls.add(user);
      return ls;
       }
}
```

这里使用了断路器，就是@HystrixCommand 注解。断路器的基本作用就是@HystrixCommand 注解的方法失败后，系统将自动切换到 fallbackMethod 方法执行。断路器 Hystrix 具备回退机制、请求缓存和请求打包以及监控和配置等功能，在这里我们只是使用了它的核心功能：回退机制，使用该功能允许你快速失败并迅速恢复或者回退并优雅降级。

这里使用 restTemplate 进行服务调用，因为使用了服务注册和发现，所以我们只需要传入服务名称 SERVICE_NAME 作为 url 的根路径，如此 restTemplate 就会去 EurekaServer 查找服务名称所代表的服务并调用。而这个服务名称就是在服务提供端 cloud-simple-service 中 spring.application.name 所配置的名字，这个名字在服务启动时连同它的 IP 和端口号都注册到了 EurekaServer。

封装好服务调用后，你只需要在 Controller 类里面注入这个服务即可：

```
@RestController
public class UserController {
@Autowired
UserService userService;
@RequestMapping(value="/users")
public ResponseEntity<List<User>> readUserInfo(){
     List<User> users=userService.searchAll();
     return new ResponseEntity<List<User>>(users,HttpStatus.OK);
}
}
```

至此这个 webui 应用的 java 端就开发完了，我们再来看页面模块。

可以看到，页面文件都放在 resources\static 目录下面，这是 spring boot 默认的页面文件主文件夹，你如果在里面放一个 index.html，那么直接访问根路径 http://localhost:8080/会直接定位到这个页面。所以这个 static 就相当于传统 web 项目里面的 webapp 文件夹。

在这里包括两个页面 index.html 和 user-page.html，用户访问首页时将会加载 user-page.html 子页面。

首页 index.html 页面核心代码如下：

```
<html ng-app="users">
<head>
      <script src="js/app.js"></script>
</head>
<body>
<div ng-view class="container">
</div>
</body>
</html>
```

页面 user-page.html 代码如下：

```
<div>
<ul ng-controller="userCtr">
  <li ng-repeat="item in userList">
{{item.username}}
</li>
</ul>
</div>
```

app.js 页面代码如下：

```
angular.module('users', ['ngRoute']).config(function ($routeProvider) {
$routeProvider.when('/', {
templateUrl: 'user-page.html',
controller: 'userCtr'
})
}).controller('userCtr', function ($scope, $http) {
$http.get('users').success(function (data) {
   //alert(data+"");
$scope.userList = data;
});
});
```

这里使用了 angularJS 库。Angular 可以绑定模型数据到页面变量，在 user-page.html 看到的{{item.username}}就是它的绑定语法，这个语法根 jsp 或 freemarker 这些模板技术都类似，只不过 Angular 是用在客户端的，而前者是用在服务端的。

至此这个 webui 应用就开发完了，改一下配置文件和 pom 文件就可以运行了。配置文件包括配置管理服务器地址，注册服务器地址等，在 bootstrap.properties 文件中：

```
spring.cloud.config.uri=http://127.0.0.1:${config.port:8888}
spring.cloud.config.name=cloud-config
spring.cloud.config.profile=${config.profile:dev}
eureka.client.serviceUrl.defaultZone=http\://localhost\:8761/eureka/
spring.application.name=simple-ui-phone
```

这些配置我们在上一章已经提到过了，这里不再解释。因为使用了断路器，所以相比上一章的 cloud-simple-service 项目，需要加入 hystrix 依赖，只有一个 starter 依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

Spring 把事情做的如此简单，不管是服务注册还是断路器，连同注解加配置文件再加上依赖总共不超过十行代码。如此简单就可以使用一个分布式应用，也难怪国内外业内人员都高度重视这个框架，在简单和速度就是硬道理的今天，这个框架有着重要的意义。

剩下最重要的事情就是进行部署了，我们以此使用命令启动这些服务及应用：

1） java -jar cloud-config-server-0.0.1.jar，启动配置服务器，固定绑定端口 8888；

2） java -jar cloud-eureka-server-1.0.0.jar，启动注册服务器，固定绑定端口 8671；

3） java -jar cloud-simple-service-1.0.0.jar --server.port=8081 >log8081.log，启动后台服务，绑定端口 8081

4） java -jar cloud-simple-service-1.0.0.jar --server.port=8082 >log8082.log，启动后台服务，绑定端口 8082

5） java -jar cloud-simple-ui-1.0.0.jar --server.port=8080 >log8080.log，启动前端 ui 应用，绑定端口 8080

运行 http://localhost:8080/即可看到运行的结果。其中“>log8080.log”表示输出日志到文件。

这里运行了两个 cloud-simple-service 实例，主要是为了演示 ribbon 负载均衡。默认情况下使用 ribbon 不需要再作任何配置，不过它依赖于注册服务器。当然也可以对 ribbon 进行一些自定义设置，比如配置它的超时时间、重试次数等。启用了负载均衡后，当我们关掉前端页面上次指向的服务时（从日志中看），比如我们刷新页面看到它调用的是 8081 服务，那么我们关掉这个服务。关

掉后再刷新会发现执行了断路器，过几秒再刷新，它已经切换到了 8082 这台服务器，这说明负载均衡起作用了。

使用 docker 发布 spring cloud 应用

本文涉及到的项目：

cloud-simple-docker：一个简单的 spring boot 应用

Docker 是一种虚拟机技术，准确的说是在 linux 虚拟机技术 LXC 基础上又封装了一层，可以看成是基于 LXC 的容器技术。可以把容器看做是一个简易版的 Linux 环境（包括 root 用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。容器是用来装东西的，Docker 可以装载应用本身及其运行环境进容器，这是一个很小的文件，然后把这个文件扔到任何兼容的服务器上就可以运行，也是基于这一点，Docker 可以同时让应用的部署、测试和分发都变得**前所未有**的高效和轻松！

下面例子参考“Spring Boot with Docker”官方例子。

1）建立一个简单的应用，只有一个类，包含 main 方法，代码如下：

```
@SpringBootApplication
@RestController
public class Application {
@RequestMapping("/")
public String home() {
 return "Hello Docker World";
}
public static void main(String[] args) {
 SpringApplication.run(Application.class, args);
}
}
```

2）建立 Dockerfile

```
# 基于那个镜像
FROM daocloud.io/java:8
# 将本地文件夹挂载到当前容器（tomcat 使用）
VOLUME /tmp
# 拷贝文件到容器
ADD cloud-simple-docker-1.0.0.jar /app.jar
# 打开服务端口
EXPOSE 8080
# 配置容器启动后执行的命令
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

这里特别要注意，这里的 FROM 采用国内的 docker 镜像，如果连国外的 docker 镜像下载，基本是不太可能下载下来的，原因大家都知道。

有了 Dockerfile，就可以部署 docker 了。

3）部署 docker 示例

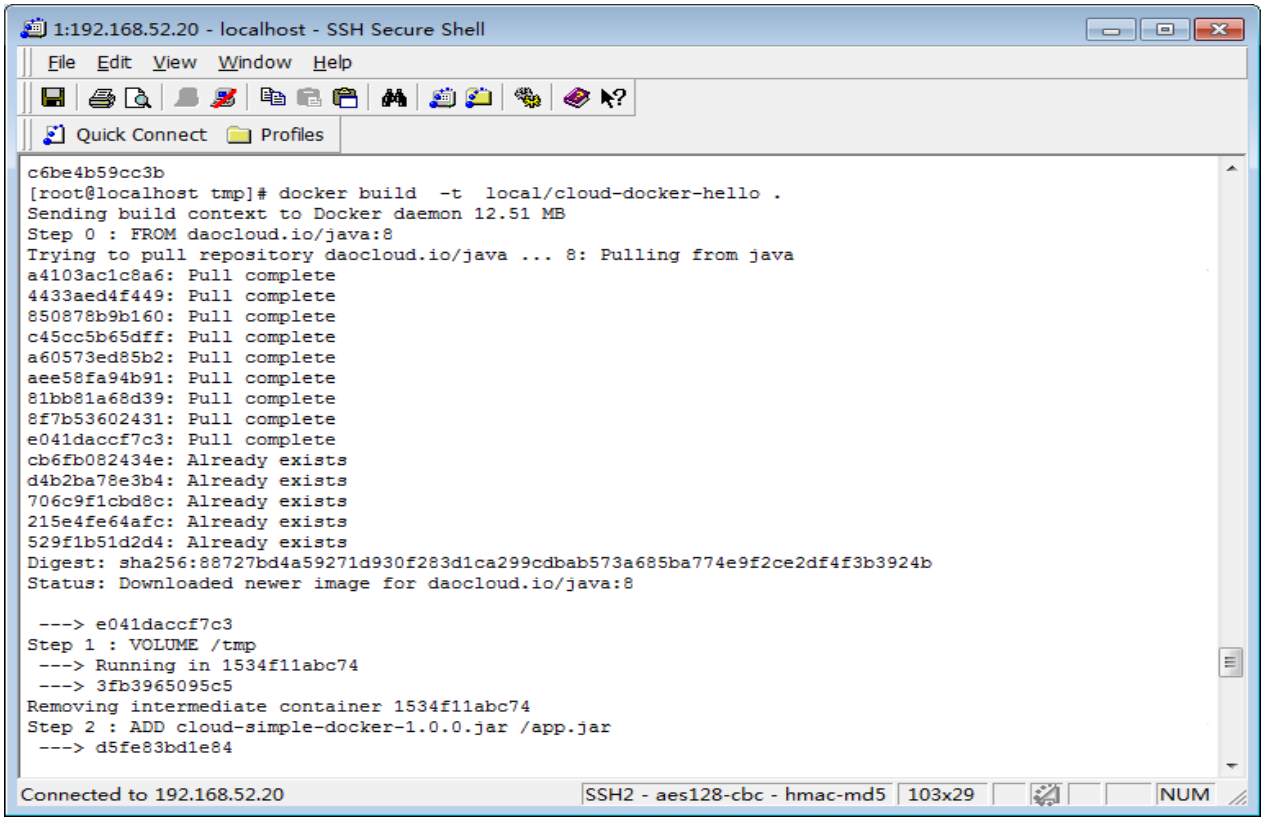
部署分为 2 步，分别是创建镜像、运行。

- 创建镜像

将编译后的 jar 文件考到服务器某个目录，这里是 tmp 目录。然后将 Dockerfile 也考到该目录，最后进入到该目录下运行命令：

docker build -t local/cloud-docker-hello .

别掉了后面的“.”符号，这个符号表示目录，这个命令执行成功，你会看到以下界面：



运行成功后，就创建了一个镜像，可以使用 docker images 来查看该镜像。

- 运行镜像

有了镜像就可以运行了，使用下面命令运行：

docker run -p 8080:8080 –t local/cloud-simple-docker

其中 8080:8080 表示本机端口映射到 Docker 实例端口。如果本机端口没有打开，还需要打开该端口，打开端口在 centos 7 中使用 firewall-cmd 命令：

firewall-cmd --zone=public --add-port=8080/tcp --permanent

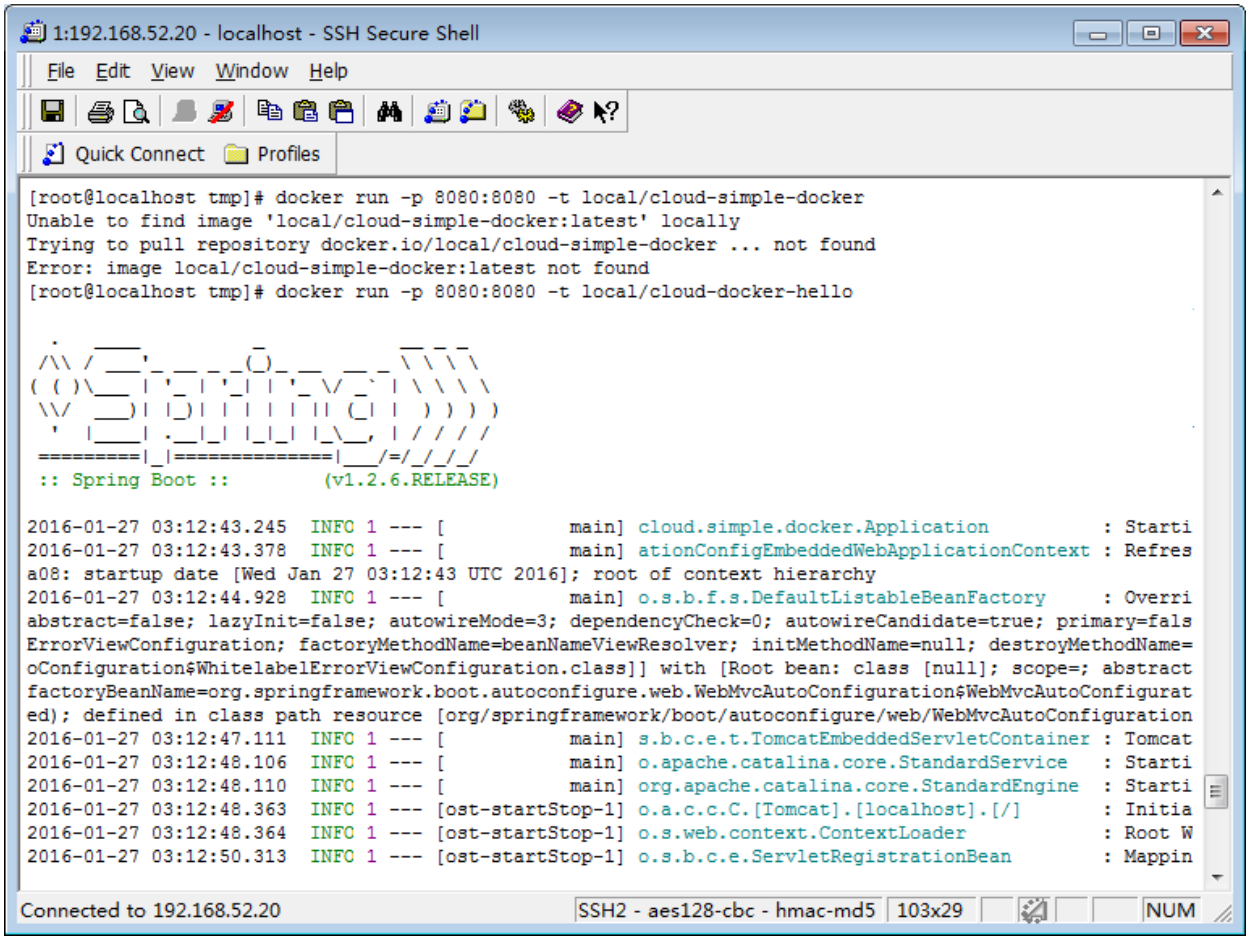
firewall-cmd –reload

成功后可以使用 docker ps –a 查看镜像运行情况：

local/cloud-docker-hello | latest | 3ef51d55eb27 | 22 minutes ago | 667.2 MB

可以看到这个包括了 java 运行环境的 web 应用镜像是 667MB。

该步骤运行结果如下：



最后使用 curl 命令或者直接在浏览器打开，可以直接测试该应用。使用 curl 命令如下：

curl http://localhost:8080

返回：Hello Docker World

docker 应用，水比较深，这里只是举一个 hello world 级别的例子。Spring Boot with Docker 例子还提供了一个 maven 插件 docker-maven-plugin，这个插件可以直接将项目编译为 docker 镜像。另外这个插件还可以让你结合 jenkins 完成项目的持续集成和自动部署。具体来说就是 jenkins 自动通过 git 或 svn 拉到工程代码，然后使用 maven 进行编译，再调用项目中的 Dockerfile 生成镜像。如此以来不管是什么应用，只需要部署 docker 就可以了，那些配置，那些乱七八糟的环境搭建都不需要了。

使用 spring boot 和 thrift、zookeeper 建立微服务

Spring cloud 适应于云端服务，也适用于企业信息化 SOA 建设。spring boot 也是 restful 微服务开发的利器。但对于内网服务，即服务与服务之间的调用，spring 并没有去刻意封装，也许他们认为已经没有必要了，因为已经有了 thrift、ice 等强大的框架。

如果是用 spring boot 本身提供的 restful 服务作为服务与服务之间的调用，效率低很多，thrift 的效率大概是 restful 的 100-1000 倍左右。本篇既是基于 spring boot 框架，结合 thrift 和 zookeeper 实现的一个简单微服务框架，服务与服务之间使用 thrift 通信（thrift 既是通信方式也是数据压缩方式）。

本 demo 一共包括三个工程：

cloud-thrift-server：服务提供方

cloud-thrift-interface：接口及传输对象定义

cloud-thrift-client：服务调用方

开源代码地址：http://git.oschina.net/zhou666/spring-cloud-7simple

1）建立 thrift 接口定义文档

```
namespace java cloud.simple.service
struct UserDto {
  1: i32 id
  2: string username
}
 service UserService {
  UserDto getUser()
}
```

接口定义完后，使用 thrift 命令生成对应的 java 文件，主要生成两个文件，分别是 UserService.java 和 UserDto.java，把这两个文件放入 cloud-thrift-interface 工程，因为客户端也需要这个接口定义。

2）实现 thrift 服务注册

在服务的提供端需要实现接口，并且还要把实现类注册到 thrift 服务器。

```
UserService.Processor processor = new UserService.Processor(
                new UserServiceImpl());
    TServer server = new TThreadPoolServer(new TThreadPoolServer.Args(
                tServerTransport()).processor(processor));
```

UserServiceImpl 就是接口实现类，将其注册金 Tserver。

注册完服务后，需要启动 Tserver，很显然这个需要在线程里启动。

```
executor.execute(new Runnable() {
        @Override
        public void run() {
```

```
                tServer().serve();
            }
    });
```

3） 使用 zookeeper 进行服务名称注册

上面是注册具体的服务执行类，这一步是将服务的实例注册进 zookeeper，这样才能实现负载均衡。让客户端可以根据服务实例列表选择服务来执行。当然这里只需要注册服务所在服务器的 IP 即可，因为客户端只要知道 IP，也就知道访问那个 IP 下的该服务。

```
String servicePath = "/"+serviceName ;// 根节点路径
    ZkClient zkClient = new ZkClient(serverList);
    boolean rootExists = zkClient.exists(servicePath);
    if (!rootExists) {
        zkClient.createPersistent(servicePath);
    }
    InetAddress addr = null;
    try {
        addr = InetAddress.getLocalHost();
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
    String ip = addr.getHostAddress().toString();
    String serviceInstance = System.nanoTime() +"-"+ ip;
    // 注册当前服务
    zkClient.createEphemeral(servicePath + "/" + serviceInstance);
    System.out.println("提供的服务为：" + servicePath + "/" + serviceInstance);
```

要注意这里使用 zkClient.createEphemeral 建立临时节点，如果这台服务器宕机，这个临时节点是会被清除的，这样客户端在访问时就不会再选择该服务器上的服务。

4） 客户端更新服务列表

客户端需要能及时的监听服务列表的变化并作出负载均衡，我们用如下方式监听服务列表的变化：

```
// 注册事件监听
    zkClient.subscribeChildChanges(servicePath, new IZkChildListener() {
        // @Override
        public void handleChildChange(String parentPath,
                List<String> currentChilds) throws Exception {
```

```
                // 实例(path)列表:当某个服务实例宕机，实例列表内会减去该实例
                for (String instanceName : currentChilds) {
                    // 没有该服务，建立该服务
                    if (!serviceMap.containsKey(instanceName)) {
                        serviceMap.put(instanceName,createUserService(instanceName));
                    }
                }
                for (Map.Entry<String, UserService.Client> entry : serviceMap.entrySet()) {
                    // 该服务已被移除
                    if (!currentChilds.contains(entry.getKey())) {
                        serviceMap.remove(entry.getKey());
                    }
                }
                System.out.println(parentPath + "事件触发");
            }
        });
```

有了服务列表，客户端在调用服务的时候就可以采用负载均衡的方式了，在这里使用最简单的随机方式：

```
    public UserService.Client getBalanceUserService(){
    Map<String, UserService.Client> serviceMap =ZooKeeperConfig.serviceMap;
    //以负载均衡的方式获取服务实例
    for (Map.Entry<String, UserService.Client> entry : serviceMap.entrySet()) {
        System.out.println("可供选择服务:"+entry.getKey());
    }
    int rand=new Random().nextInt(serviceMap.size());
    String[] mkeys = serviceMap.keySet().toArray(new String[serviceMap.size()]);
    return serviceMap.get(mkeys[rand]);
 }
```

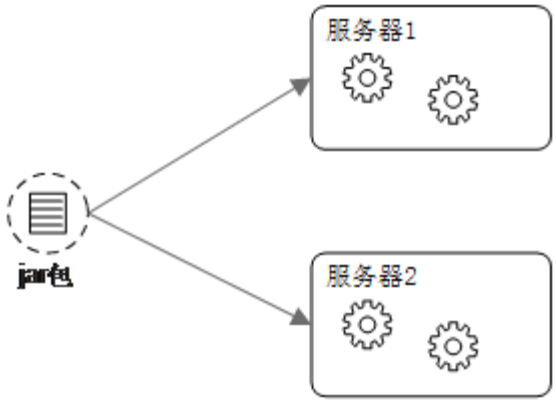
本文结束，具体参见代码，另外，之前还不了解 thrift 和 zookeeper 的朋友，不要被他们吓到，其实他们是很轻量级的技术，很容易上手，这也许就是他们流行的原因。

spring boot 自动部署方案

现在主流的自动部署方案大都是基于 Docker 的了，但传统的自动部署方案比较适合中小型公司，下面的方案就是比较传统的自动部署方案。

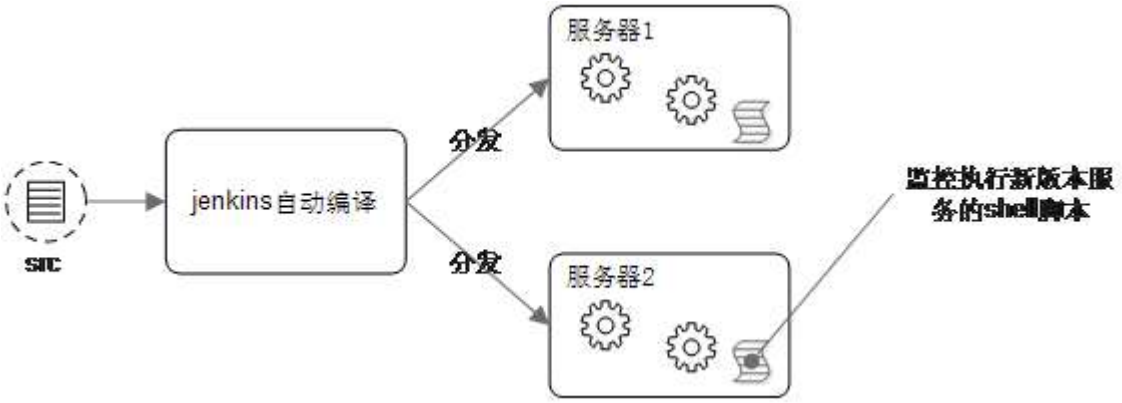
1、为什么需要自动部署

基于微服务的架构，自动部署显得非常重要。因为每一个服务都需要部署。如果是手动部署，那么有 M 个服务，那么至少需要部署 M 次，如果每个同样的服务部署 N 个实例，那么就需要部署 M*N 次。所以自动部署对于微服务架构几乎是必须的，这一点不同于传统应用。

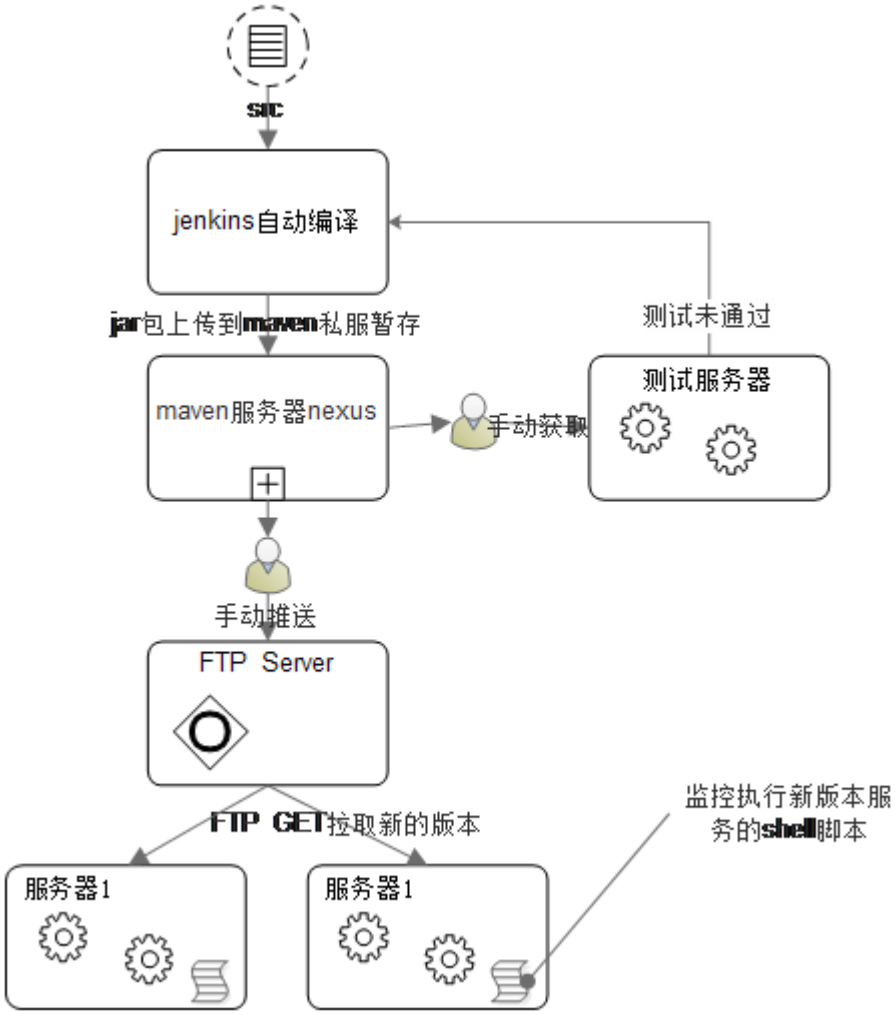


2、如何实现自动部署

自动部署一般都是通过以下步骤进行的。首选由持续性集成工具进行自动编译产生项目的输出，对于我们来说也就是 jar 包。然后该 jar 经过测试就可以分发到各个服务器，各个服务器的监控脚本监控到该新版本，自动停止旧实例重新运行新实例。



上面只是一个大概的步骤，真实的实现还需要更详细的步骤，如下：



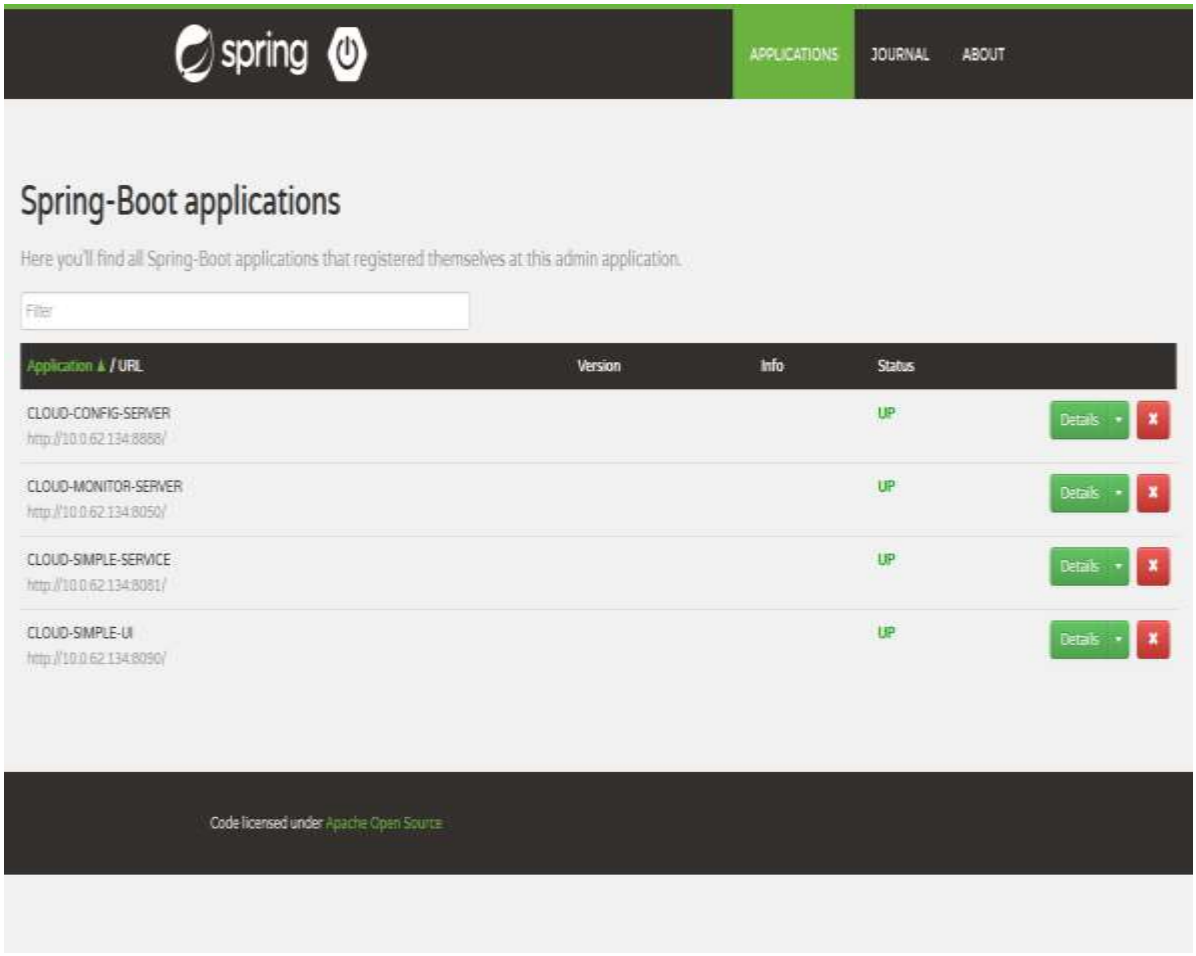
Jenkins 编译的结果需要暂时存放，以便于测试人员拉取进行测试。这里存放在 maven 库中。测试通过后也需要手动推送到生产环境，因为不可能每个版本都推送到生产环境。生产环境需要一台 FTP 或 GIT、SVN Server 作为中转机，暂存打包的应用，然后生产的服务器通过脚本轮询该中转机获得新的版本。获得新的版本后，自动停止旧的版本，运行新的版本。

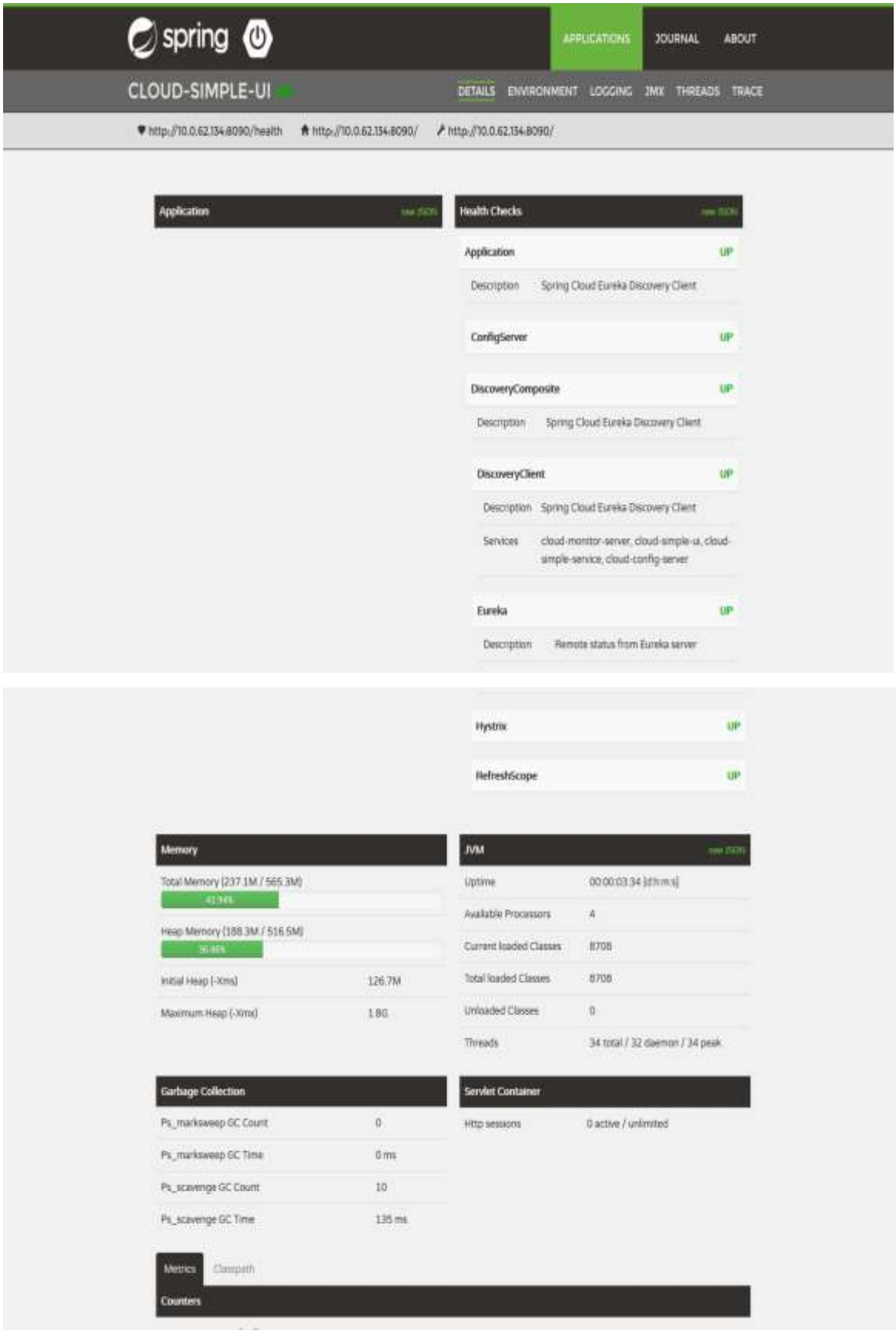
spring boot/cloud 应用监控

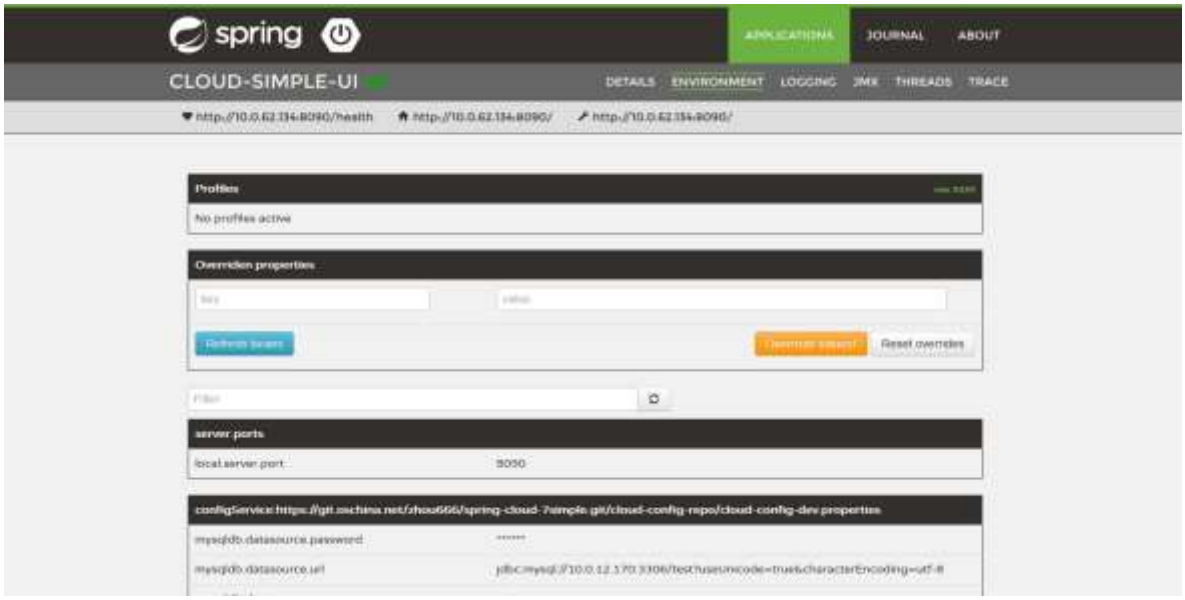
应用的监控功能，对于分布式系统非常重要。如果把分布式系统比作整个社会系统。那么各个服务对应社会中具体服务机构，比如银行、学校、超市等，那么监控就类似于警察局和医院，所以其重要性显而易见。这里说的，监控服务的部署及运行情况，和断路器监控不一样，这里主要是监控服务及服务器的各项指标。该项目是使用了开源 spring boot 监控项目 spring-boot-admin，开源项目地址：

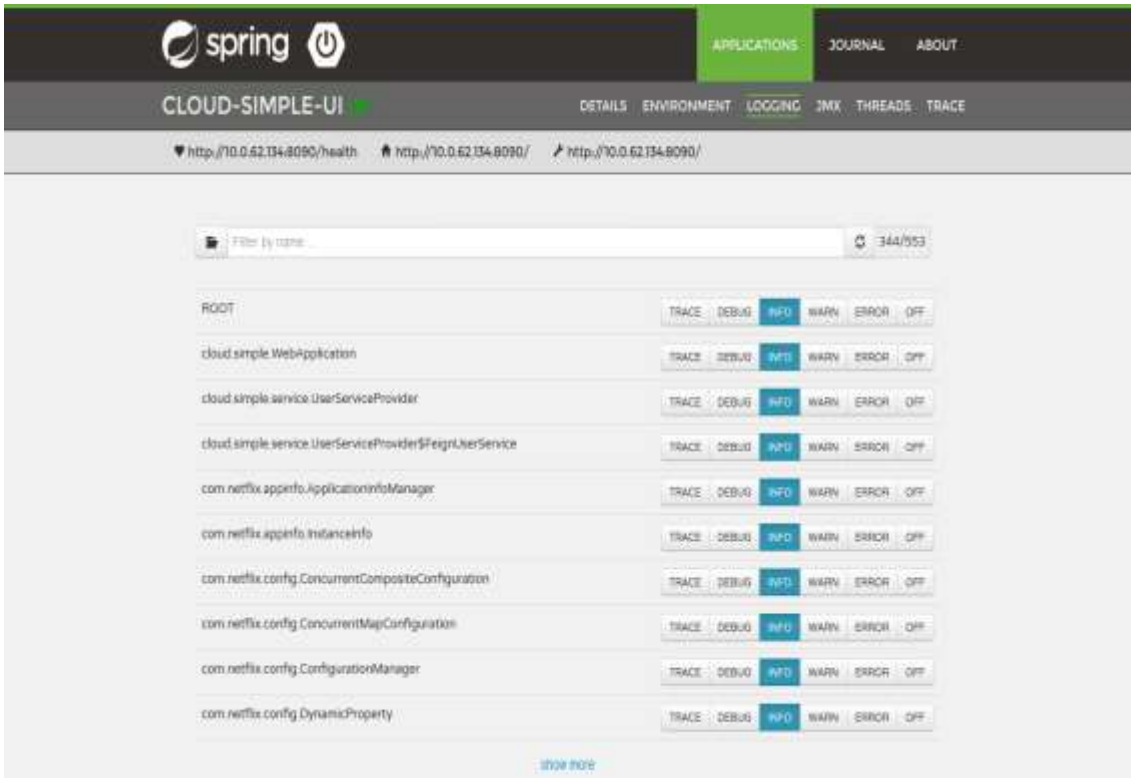
spring-boot-admin：https://github.com/codecentric/spring-boot-admin
demo 地址：http://git.oschina.net/zhou666/spring-cloud-7simple/tree/master/cloud-monitor-server

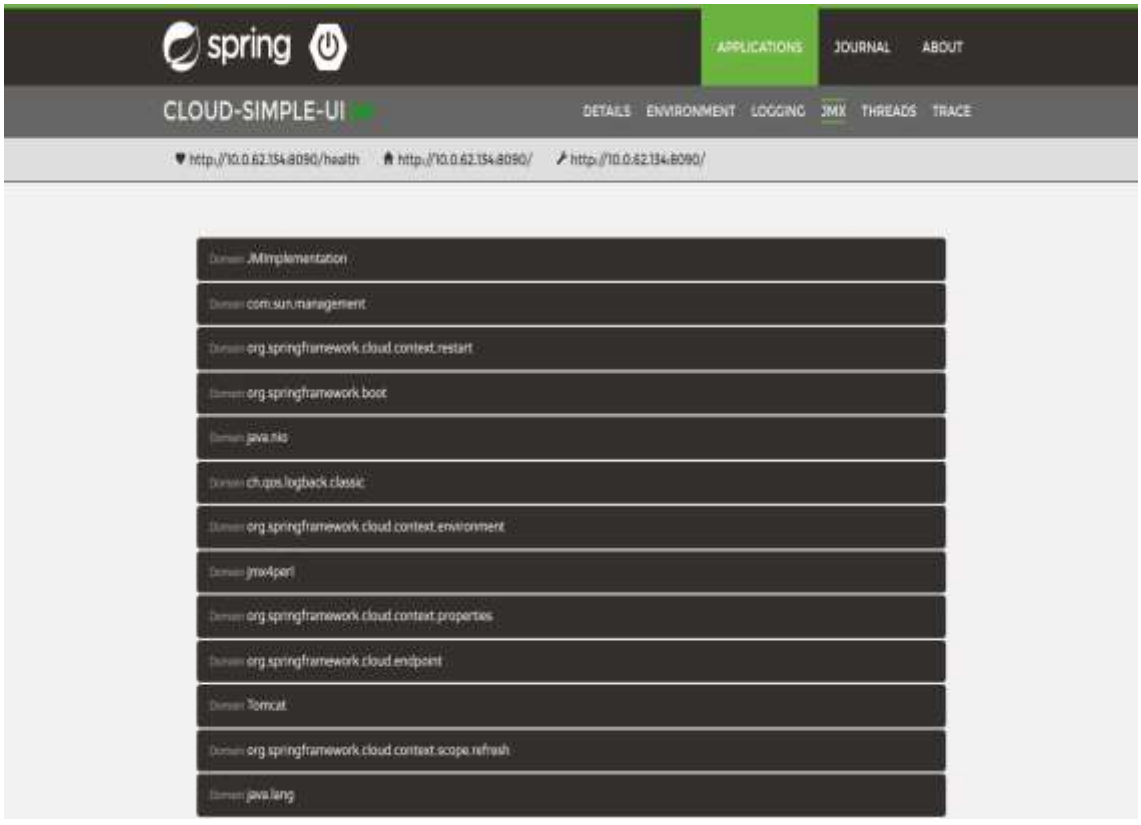
spring-boot-admin 不光是监控还可以动态改变配置参数，废话不多说，直接上图好了，：

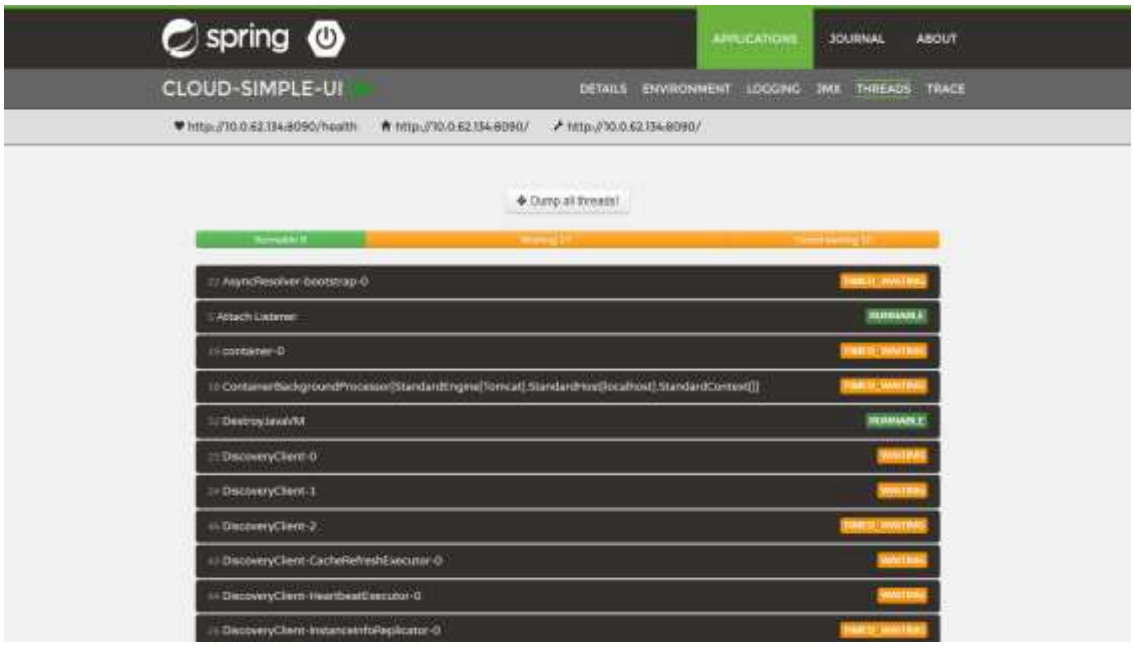

spring
APPLICATIONS
JOURNAL
ABOUT
Spring-Boot applications
Here you'll find all Spring-Boot applications that registered themselves at this admin application.
Filter
Application ▲ / URL
Version
Info
Status
CLOUD-CONFIG-SERVER
http://10.0.62.134:8888/
UP
Details
CLOUD-MONITOR-SERVER
http://10.0.62.134:8050/
UP
Details
CLOUD-SIMPLE-SERVICE
http://10.0.62.134:8081/
UP
Details
CLOUD-SIMPLE-UI
http://10.0.62.134:8090/
UP
Details
Code licensed under Apache Open Source


spring
APPLICATIONS
JOURNAL
ABOUT
CLOUD-SIMPLE-UI
DETAILS
ENVIRONMENT
LOGGING
JMX
THREADS
TRACE
http://10.0.62.134:8090/health
http://10.0.62.134:8090/
http://10.0.62.134:8090/
Application
Health Checks
Application
UP
Description
Spring Cloud Eureka Discovery Client
ConfigServer
UP
DiscoveryComposite
UP
Description
Spring Cloud Eureka Discovery Client
DiscoveryClient
UP
Description
Spring Cloud Eureka Discovery Client
Services
cloud-monitor-server, cloud-simple-ui, cloud-simple-service, cloud-config-server
Eureka
UP
Description
Remote status from Eureka server
Hystrix
UP
RefreshScope
UP
Memory
Total Memory (237.1M / 565.3M)
41.94%
Heap Memory (188.3M / 516.5M)
36.46%
Initial Heap (-Xms)
126.7M
Maximum Heap (-Xmx)
1.8G
JVM
Uptime
00:00:03.34 [d:h:m:s]
Available Processors
4
Current loaded Classes
8708
Total loaded Classes
8708
Unloaded Classes
0
Threads
34 total / 32 daemon / 34 peak
Garbage Collection
Ps_marksweep GC Count
0
Ps_marksweep GC Time
0 ms
Ps_scavenge GC Count
10
Ps_scavenge GC Time
135 ms
Servlet Container
Http sessions
0 active / unlimited
Metrics
Classpath
Counters


spring
APPLICATIONS
JOURNAL
ABOUT
CLOUD-SIMPLE-UI
DETAILS
ENVIRONMENT
LOGGING
JMX
THREADS
TRACE
http://10.0.62.134:8090/health
http://10.0.62.134:8090/
http://10.0.62.134:8090/
Profiles
No profiles active
Overriden properties
Reset overrides
server ports
local.server.port
8090
mysqldb.datasource.password
mysqldb.datasource.url


spring
APPLICATIONS
JOURNAL
ABOUT
CLOUD-SIMPLE-UI
DETAILS
ENVIRONMENT
LOGGING
JMX
THREADS
TRACE
http://10.0.62.134:8090/health
http://10.0.62.134:8090/
http://10.0.62.134:8090/
344/553
ROOT
cloud.simple.WebApplication
cloud.simple.service.UserServiceProvider
cloud.simple.service.UserServiceProvider$FeignUserService
com.netflix.appinfo.ApplicationInfoManager
com.netflix.appinfo.InstanceInfo
com.netflix.config.ConcurrentCompositeConfiguration
com.netflix.config.ConcurrentMapConfiguration
com.netflix.config.ConfigurationManager
com.netflix.config.DynamicProperty
TRACE
DEBUG
INFO
WARN
ERROR
OFF
show more



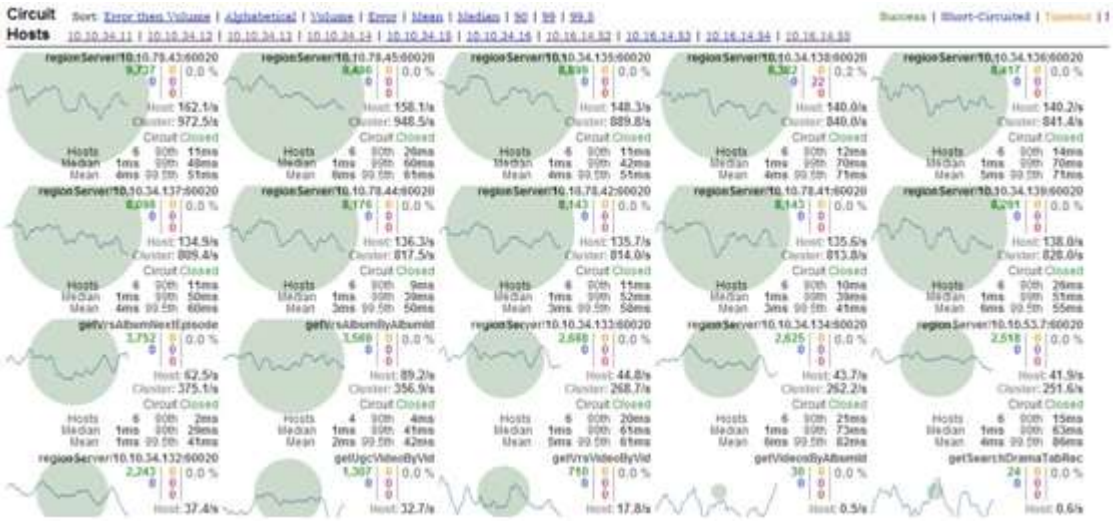


hystrix-turbine 监控的使用

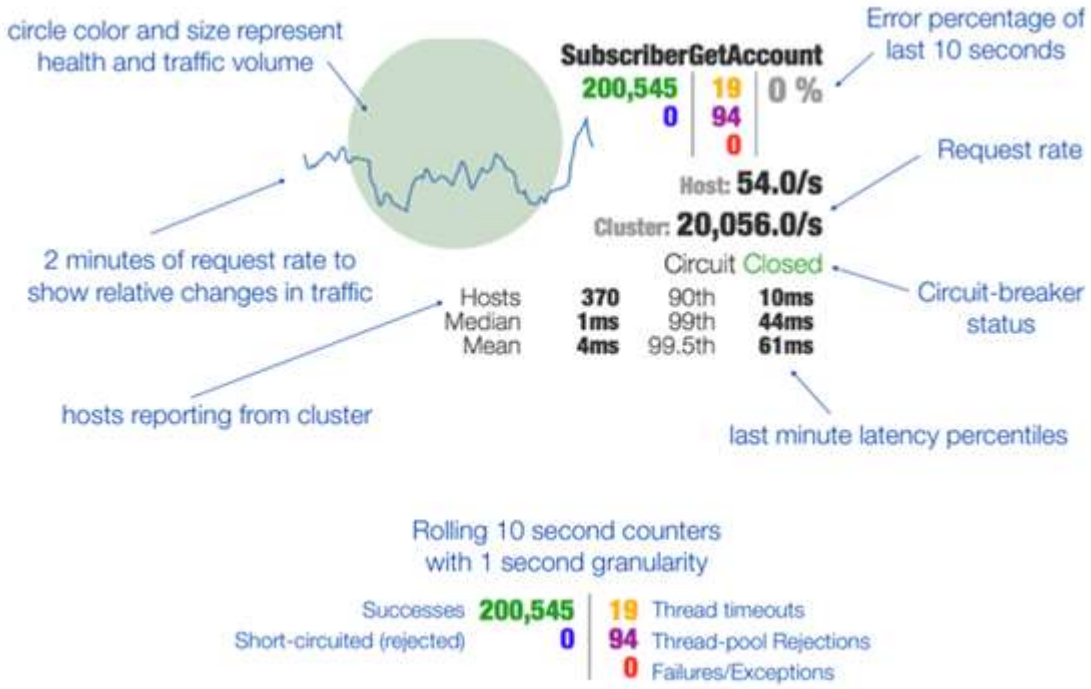
1. 概述

Demo 地址：http://git.oschina.net/zhou666/spring-cloud-7simple/tree/master/cloud-hystrix-turbine

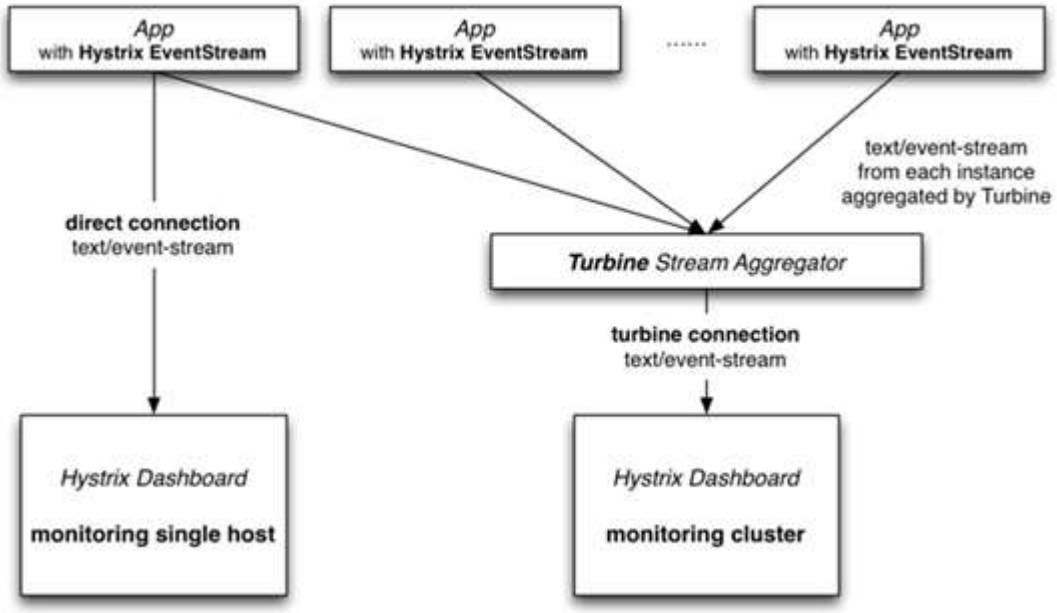
hystrix-turbine 集成了 hystrix 看板和 turbine，用来监控实现了 hystrix 的工程项目：



每一个监控项目的具体解释：



原本的 hystrix 看板只能监控一台服务器上的服务调用情况，使用了 turbine 后就可以监控多台服务器的情况。Turbine 原理如下：



2. 主要配置文件

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<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>
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-turbine</artifactId>
  </dependency>
```

配置文件：

```
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10 #心跳间隔
  client:
    registerWithEureka: true #注册本工程为服务
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/ #注册服务器地址
turbine:
  aggregator:
    clusterConfig: CLOUD-SIMPLE-SERVICE  #turbine 监控的服务名称，可以多个
  appConfig: cloud-simple-service  #turbine 监控的服务，可以有多个
  clusterNameExpression: metadata['cluster']
```

3. 启动与调试

启动应用输入 http://localhost:8989/hystrix 会看到 hystrix 面板，在这个面板里的监控 url 输入，http://localhost:8989/turbine.stream??cluster=CLOUD-SIMPLE-SERVICE，其中 cluster 对应配置文件中 clusterConfig 中的名称。