Spirng cloud微服务搭建

Spring cloud微服务架构，就是由多个spirng boot的微服务组成。也就是如果想搭建起springcloud，首先先创建一个可以操作数据库的spring boot程序。将这个spring boot程序作为微服务中的一个服务。

步骤如下。

第一步通过maven创建工程

【dianfaner-springcloud-parent】为父工程

【dianfaner-springcloud-api】为子模块

【dianfaner-springcloud-user】为子模块

其中【dianfaner-springcloud-parent】为父工程，负责其他几子模块共同需要的依赖管理。【dianfaner-springcloud-api】主要用于定于微服务中，各个服务所使用的entity。

【dianfaner-springcloud-user】为微服务中的一个服务，用于管理用户信息。

三个工程的pom.xml内容如下；

【dianfaner-springcloud-parent】

|  |
| --- |
| <project *xmlns*="http://maven.apache.org/POM/4.0.0" *xmlns:xsi*="http://www.w3.org/2001/XMLSchema-instance"  *xsi:schemaLocation*="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.qd.lhb</groupId>  <artifactId>dianfaner-springcloud-parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>pom</packaging>  <modules>  <module>dianfaner-springcloud-api</module>  <module>dianfaner-springcloud-user</module>  </modules>  *<!--dependencyManagement 解决maven多模块时，我们无法使用spring-boot-parent，从而无法依赖相关spring-boot的jar问题。 -->*  <dependencyManagement>  <dependencies>  *<!-- 使用此种方式后，便可以在dianfaner-web-rest-api中加载spring boot相关依赖了 -->*  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>2.0.0.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR7</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  *<!-- <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-jpa</artifactId>*  *</dependency> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-security</artifactId>*  *</dependency> -->*  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>spring-security-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.2</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

【dianfaner-springcloud-api】

|  |
| --- |
| <project *xmlns*="http://maven.apache.org/POM/4.0.0" *xmlns:xsi*="http://www.w3.org/2001/XMLSchema-instance" *xsi:schemaLocation*="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.qd.lhb</groupId>  <artifactId>dianfaner-springcloud-parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  </parent>  <artifactId>dianfaner-springcloud-api</artifactId>  </project> |

【dianfaner-springcloud-user】

|  |
| --- |
| <project *xmlns*="http://maven.apache.org/POM/4.0.0" *xmlns:xsi*="http://www.w3.org/2001/XMLSchema-instance"  *xsi:schemaLocation*="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.qd.lhb</groupId>  <artifactId>dianfaner-springcloud-parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  </parent>  <artifactId>dianfaner-springcloud-user</artifactId>  <dependencies>  <dependency>  <groupId>com.qd.lhb</groupId>  <artifactId>dianfaner-springcloud-api</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  </dependencies>  </project> |

设置好了三个工程的pom.xml文件后，我们分别为三个工程相应的package

因为我们是对数据库的user表进行操作，所以现在【dianfaner-springcloud-api】

中创建entity，我们创建包com.qd.lhb.entity,并在包下面创建UserEntity.java

|  |
| --- |
| **package** com.qd.lhb.entity;  **import** java.io.Serializable;  **public** **class** *UserEntity* **implements** Serializable{  **private** *Long* *id*;  **private** *String* *name*;  **private** *String* *dblocation*;  **public** **UserEntity**(*Long* id, *String* name, *String* dblocation) {  **super**();  **this**.*id* = id;  **this**.*name* = name;  **this**.*dblocation* = dblocation;  }  **public** *Long* **getId**() {  **return** *id*;  }  **public** **void** **setId**(*Long* id) {  **this**.*id* = id;  }  **public** *String* **getName**() {  **return** *name*;  }  **public** **void** **setName**(*String* name) {  **this**.*name* = name;  }  **public** *String* **getDblocation**() {  **return** *dblocation*;  }  **public** **void** **setDblocation**(*String* dblocation) {  **this**.*dblocation* = dblocation;  }    @Override  **public** *String* **toString**() {  **return** "UserEntity [id=" + *id* + ", name=" + *name* + ", dblocaltion=" + *dblocation* + ", getId()=" + **getId**()+ ", getName()=" + **getName**() + ", getDblocaltion()=" + **getDblocation**() + ", getClass()=" + getClass()+ ", hashCode()=" + hashCode() + ", toString()=" + **super**.toString() + "]";  }      } |

创建好了package和UserEntity.java类后，我们需要把【dianfaner-springcloud-api】

加入到【dianfaner-springcloud-user】的pom中，因为【dianfaner-springcloud-user】

中需要依赖于UserEntity.java这个类。

引入方式如下：

打开【dianfaner-springcloud-user】的pom.xml然后添加如下内容（在最前面的三个pom.xml中已经包含这些内容了。这了只是说明使用）

|  |
| --- |
| <dependency>  <groupId>com.qd.lhb</groupId>  <artifactId>dianfaner-springcloud-api</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency> |

【dianfaner-springcloud-api】工程代码写好后，下面我们开始编写【dianfaner-springcloud-user】中了业务逻辑，包括与数据库的交互。

首先如果想在spring boot与数据库交互，我们选择使用mybaits作为中间件，所以首先需要添加对于mybatis的依赖，如下

|  |
| --- |
| <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.2</version>  </dependency> |

然后因为我们使用mysql，要想操作数据库，需要安装jdbc，添加依赖如下

|  |
| --- |
| <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

然后在spring boot 的application.properties文件中将连接数据的一些例如用户名密码等息息进行配置，如下

|  |
| --- |
| #mysql配置  spring.datasource.url=jdbc:mysql://127.0.0.1:3306/user  spring.datasource.username=root  spring.datasource.password=root  spring.datasource.driver-class-name=com.mysql.jdbc.Driver |

以上基本的以来配置完成后，现在就可以编写业务逻辑，操作数据库了。如下

首先建立dao层，来作为数据库的操作。

建立包com.qd.lhb.dao,在包中创建访问数据库用的接口，UserMapper.java。内容如下：

|  |
| --- |
| **package** com.qd.lhb.dao;  **import** org.apache.ibatis.annotations.Mapper;  **import** org.apache.ibatis.annotations.Select;  **import** com.qd.lhb.entity.UserEntity;  //@Mapper  **public** **interface** UserMapper {  //@Select("INSERT INTO user (id,name,dblocation) VALUES(#{id}, #{name}, #{dblocation})")  **public** **boolean** **insert**(*UserEntity* user);  **public** **boolean** **delete**(*Long* id);  **public** **boolean** **update**(*UserEntity* user);  @Select("SELECT \* FROM user WHERE id = #{1}")  **public** *UserEntity* **find**(*Long* id);  } |

注意，这里代码中的使用的是mybatis的一种写法，通过方法上直接写sql文，来代替xml文件。

@Mapper的作用是：将这个接口作为操作mybais的方法，加了这个注解，便可以自动被扫描到。如果每个上面都加这个注解，比较繁琐，所以更高级的方式是，在springboot启动类上，添加@MapperScan注解。这样便会自动将包下的类，作为操作mybaits的方法。代码如下：

|  |
| --- |
| @SpringBootApplication  @MapperScan("com.qd.lhb.dao") //告诉dao接口所在的包名  **public** **class** *ApplicationStart* {  **public** **static** **void** **main**(*String*[] args) {  *SpringApplication*.*run*(*ApplicationStart*.**class**, args);  }  } |

到这里，我们操作数据库的代码就算都完成了。

下面我们便可以编写测试类，来进行一个简单的测试了。测试方式如下，在test目录下，创建包com.qd.lhb,然后创建测试类UserTest.java

|  |
| --- |
| package com.qd.lhb;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringRunner;  import com.qd.lhb.dao.UserMapper;  import com.qd.lhb.entity.UserEntity;  @RunWith(SpringRunner.class)  @SpringBootTest  public class UserTest {  @Autowired  private UserMapper userMapper;  @Test  public void find() {  UserEntity userEntity = userMapper.find(1l);  }  @Test  public void insert() {  userMapper.insert(new UserEntity(2L,"user2", "db1"));  }  } |

然后右击，junit运行，即可。

有人会问，上面我们在dao的userMapper.java接口中，直接使用@Select注解写的sql文，虽然很方便，但是如果sql比较复杂，这种方式还是不如xml写sql的方式方便，那我们应该怎么办呢？

方法如下：我们在dao的同级目录中，添加包com.qd.lhb.mapper，然后添加与接口同名的UserMapper.xml文件，里面还是按照mybaits传统的写法编写sql文，如下

|  |
| --- |
| **<?**xml *version*="1.0" *encoding*="UTF-8"**?>**  <!DOCTYPE **mapper** **PUBLIC** **"-//mybatis.org//DTD Mapper 3.0//EN"** **"http://mybatis.org/dtd/mybatis-3-mapper.dtd"**>  <mapper *namespace*="**com.qd.lhb.dao.UserMapper**" >  <insert *id*="insert" *parameterType*="com.qd.lhb.entity.UserEntity" >  INSERT INTO  user  (id,name,dblocation)  VALUES  (#{id}, #{name}, #{dblocation})  </insert>  </mapper> |

注意：绿色namespace处，一定是我们对应的dao中的接口的包名+接口名。

有了xml后，剩下的就是，我们需要在spring boot的application.propertires文件中添加一些设置：

|  |
| --- |
| #mybatis 配置  mybatis.mapper-locations=classpath:com/qd/lhb/mapper/\*.xml |
| 添加这个配置，便可以让mybatis找到mapper.xml的路径，然后将sql问的内容注入到框架中，这样便可以在调用dao层的接口是，直接映射到xml，实现数据库的操作。 |

以上测试通过后，便可以通过restful的方式使用controller和service来调用dao，然后返回json串。将json串作为服务提供者的接口发布出去，这样我们便顺利的完成了一个微服务的一个服务提供者的实现。消费者只要通过http请求调用即可。

Controller的代码如下：

|  |
| --- |
| **package** com.qd.lhb.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.qd.lhb.service.IUserService;  @RestController  @RequestMapping("/user")  **public** **class** *UserController* {  @Autowired  **private** IUserService *userService*;  @RequestMapping("/find/{id}")  **public** *Object* **findUser**(@PathVariable ("id") *Long* id){  **return** *userService*.find(id);  }  } |

Servi的代码如下

|  |
| --- |
| **package** com.qd.lhb.service;  **import** com.qd.lhb.entity.UserEntity;  **public** **interface** IUserService {  **public** **boolean** **insert**(*UserEntity* user);  **public** **boolean** **delete**(*Long* id);  **public** **boolean** **update**(*UserEntity* user);  **public** *UserEntity* **find**(*Long* id);  } |

|  |
| --- |
| **package** com.qd.lhb.service.impl;  **import** org.apache.catalina.User;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  **import** com.qd.lhb.dao.UserMapper;  **import** com.qd.lhb.entity.UserEntity;  **import** com.qd.lhb.service.IUserService;  @Service  **public** **class** *UserServiceImpl* **implements** IUserService {  @Autowired  **private** UserMapper *userMapper*;  **public** **boolean** **insert**(*UserEntity* user) {  *userMapper*.insert(user);  **return** **false**;  }  **public** **boolean** **delete**(*Long* id) {  *userMapper*.delete(id);  **return** **false**;  }  **public** **boolean** **update**(*UserEntity* user) {  *userMapper*.update(user);  **return** **false**;  }  **public** *UserEntity* **find**(*Long* id) {  **return** *userMapper*.find(id);  }  } |

基本一个restful的微服务开发完成，但是我们还需要安全方面的考虑，否则无法防止恶意进攻，那么我们需要加上权限等安全访问机制，这事我们考虑使用spring security。

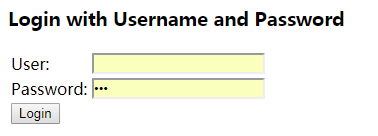
那么我们手下添加依赖到【dianfaner-springloud-parent】的pom.xml中

|  |
| --- |
| <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>spring-security-web</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>spring-security-config</artifactId>  <version>5.0.3.RELEASE</version>  </dependency> |

当添加依赖后，我们再次重启服务，会发现控制台中出现了一段文字：

Using generated security password: 1081d520-9fd1-4081-becd-421a1609e1a4

，同时访问我们的restful接口，会发现弹出了输入密码的页面，如下：



这就说明我们的spring security开始起作用了。

那么既然出现了需要输入用户名和密码的页面，那么肯定就需要一个地方法来定义用户名和密码的。打开【dianfaner-springcloud-user】的application.properties文件，然后添加下面内容

|  |
| --- |
| #spring security  # Default user name.  spring.security.user.name=user  # Password for the default user name. A random password is logged on startup by default.  spring.security.user.password=123456 |

然后我们重启服务，输入这里设置的用户名和密码，我们发现，成功登录，然后可以调用我们的接口了。

Tip：然而我们发现，每次重新开启浏览器，访问我们接口的时候，都会要求输入用户名和密码，那么有没有什么方式可以直接访问，而无需填写用户名密码呢。可以使用这样的方式：

<http://user:123456@localhost:8080/user/find/1>

到这里位置我们是通过application.properties来设置security的用户名和密码。

但是在实际项目中，用户账号和密码都是通过数据库表读取的，所以下面我们来模拟怎样通过代码中设置用户名密码这样的访问权限。

首先在【dianfaner-springcloud-user】中新建一个包com.qd.lhb.configure，然后创建一个配置类SecurityConfigure，来设置认证和权限相关的内容。代码如下

|  |
| --- |
| package com.qd.lhb.configure;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  import org.springframework.security.config.annotation.web.builders.HttpSecurity;  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  @EnableWebSecurity  public class SecurityConfig extends WebSecurityConfigurerAdapter {  @Override  protected void configure(HttpSecurity http) throws Exception {  http .authorizeRequests().anyRequest().authenticated().and().formLogin().and().httpBasic();  }  @Autowired  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  //spring security5.0版本后,密码的设置有所变更，方式如下。{noop}表示一种加密方式，后面的123456才是真正的密码。界面还是输入123456，不用加{noop}  auth.inMemoryAuthentication().withUser("user1").password("{noop}123456").roles("USER")  .and().withUser("admin").password("{noop}admin").roles("ADMIN", "USER");  }  } |

通过这种方式后，便可以吧application.properties中的security的设置删除了。

然后启动服务，输入用户名密码，便可以访问了。

基本的服务做好了，现在开始制作我们的注册中心，新建一个maven module，名字为【dianfaner-springcloud-eureka】,它主要用来作为服务注册中心。

spring cloud中使用的是eureka来作为注册中心，而不是zookeeper。这点要注意。由于spring boot2.0还没有发realease版，在之前测试过程中出现过问题。调查发现spring boot2.0如果和spring cloud合用，那么spring cloud使用Finchley.M7比较好。具体pom配置修改如下，将【dianfaner-springcloud-user】的pom.xml文件dependencyManagement内容改为如下版本。

|  |
| --- |
| <dependencyManagement>  <dependencies>  *<!-- 使用此种方式后，便可以在dianfaner-web-rest-api中加载spring boot相关依赖了 -->*  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>2.0.0.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Finchley.M7</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  *<!-- 注意： 这里必须要添加， 否者各种依赖有问题 -->*  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/libs-milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

Pom修改后，我们便在【dianfaner-springcloud-eureka】的pom中添加如下依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  </dependency>  </dependencies> |

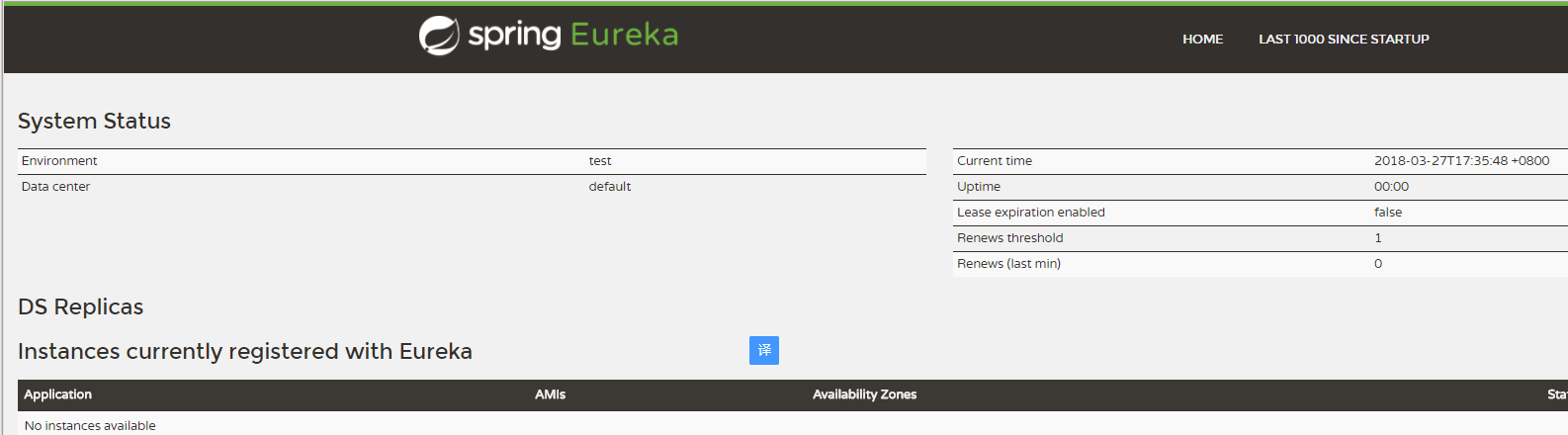
然后创建spring boot启动类*EurekaApplication，代码如下*

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  **public** **class** *EurekaApplication* {  **public** **static** **void** **main**(*String*[] args) {  *SpringApplication*.*run*(*EurekaApplication*.**class**, args);  }  } |

然后编写eureka的配置文件，application.yml

|  |
| --- |
| server:  port: 9871  eureka:  instance:  hostname: localhost  client:  registerWithEureka: false  fetchRegistry: false  serviceUrl:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ |

上面都配置完后，启动服务，访问<http://localhost:9871/>。出现一个下面的页面表示注册中心创建成功了。图如下：



|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |