# Springboot

**By Brother.Lee 20171214 Shanghai**

# 第一篇：构建第一个SpringBoot工程

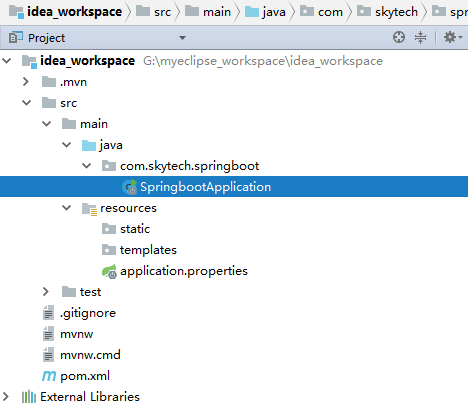
使用idea作为开发工具来开发springboot

话不多说,直接开始项目

## 一.创建项目

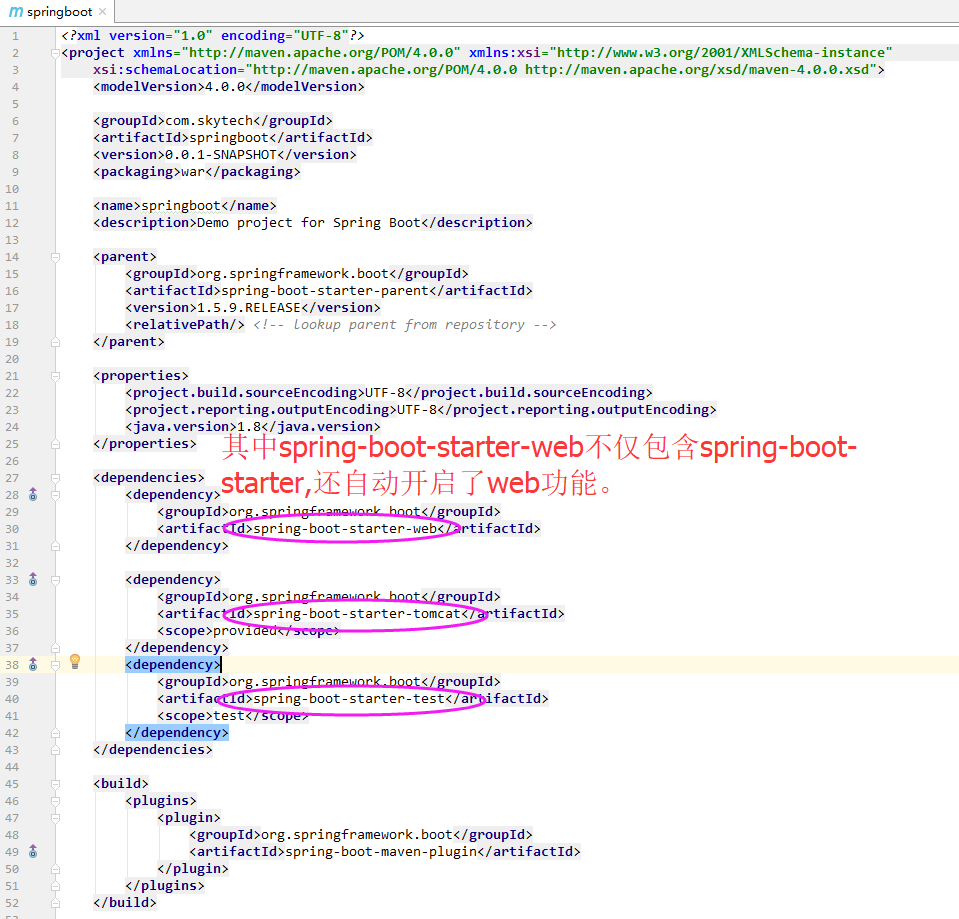
打开 Idea-> new Project ->Spring Initializer(1.8jdk) ->填写group、artifact ->钩上web(开启web功能）->点下一步就行了。

建立完成后的工程目录如下:

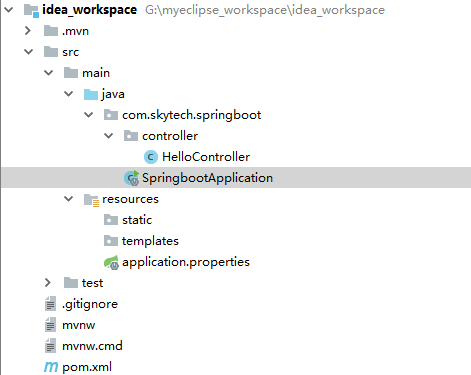


简单解释一下:

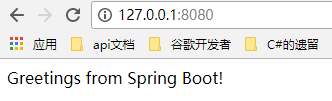
* pom文件为基本的依赖管理文件
* resouces 资源文件
* statics 静态资源
  + - templates 模板资源
    - application.yml 配置文件
* SpringbootApplication程序的入口。



建立一个HelloController



@RestController  
public class HelloController {  
  
 @RequestMapping("/")  
 public String index() {  
 return "Greetings from Spring Boot!";  
 }



神奇之处：

* 你没有做任何的web.xml配置。
* 你没有做任何的sping mvc的配置; springboot为你做了。
* 你没有配置tomcat ;springboot内嵌tomcat.

## 二.来看看springboot在启动的时候为我们注入了哪些bean

在程序入口加入：



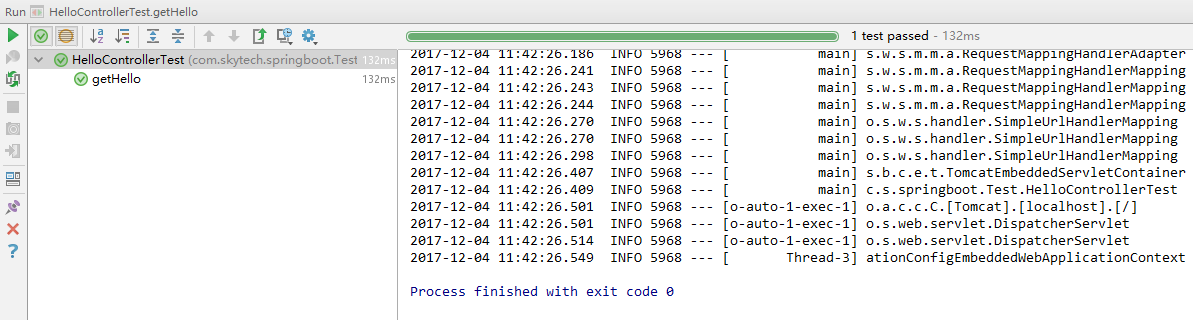
|  |
| --- |
| Let's inspect the beans provided by Spring boot:  basicErrorController  beanNameHandlerMapping  beanNameViewResolver  characterEncodingFilter  commandLineRunner  conventionErrorViewResolver  defaultServletHandlerMapping  defaultValidator  defaultViewResolver  dispatcherServlet  dispatcherServletRegistration  duplicateServerPropertiesDetector  embeddedServletContainerCustomizerBeanPostProcessor  error  errorAttributes  errorPageCustomizer  errorPageRegistrarBeanPostProcessor  faviconHandlerMapping  faviconRequestHandler  handlerExceptionResolver  helloController  hiddenHttpMethodFilter  httpPutFormContentFilter  httpRequestHandlerAdapter  jacksonObjectMapper  jacksonObjectMapperBuilder  jsonComponentModule  localeCharsetMappingsCustomizer  mappingJackson2HttpMessageConverter  mbeanExporter  mbeanServer  messageConverters  methodValidationPostProcessor  multipartConfigElement  multipartResolver  mvcContentNegotiationManager  mvcConversionService  mvcHandlerMappingIntrospector  mvcPathMatcher  mvcResourceUrlProvider  mvcUriComponentsContributor  mvcUrlPathHelper  mvcValidator  mvcViewResolver  objectNamingStrategy  org.springframework.boot.autoconfigure.AutoConfigurationPackages  org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration  org.springframework.boot.autoconfigure.condition.BeanTypeRegistry  org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration  org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration  org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration  org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory  org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration  org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$Jackson2ObjectMapperBuilderCustomizerConfiguration  org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperBuilderConfiguration  org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperConfiguration  org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration  org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration  org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration  org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletConfiguration  org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletRegistrationConfiguration  org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration  org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat  org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration  org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$DefaultErrorViewResolverConfiguration  org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration  org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration  org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration  org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration$StringHttpMessageConverterConfiguration  org.springframework.boot.autoconfigure.web.JacksonHttpMessageConvertersConfiguration  org.springframework.boot.autoconfigure.web.JacksonHttpMessageConvertersConfiguration$MappingJackson2HttpMessageConverterConfiguration  org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration  org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration  org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration  org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration$RestTemplateConfiguration  org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration  org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$EnableWebMvcConfiguration  org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter  org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter$FaviconConfiguration  org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration  org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration$TomcatWebSocketConfiguration  org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor  org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor.store  org.springframework.context.annotation.internalAutowiredAnnotationProcessor  org.springframework.context.annotation.internalCommonAnnotationProcessor  org.springframework.context.annotation.internalConfigurationAnnotationProcessor  org.springframework.context.annotation.internalRequiredAnnotationProcessor  org.springframework.context.event.internalEventListenerFactory  org.springframework.context.event.internalEventListenerProcessor  preserveErrorControllerTargetClassPostProcessor  propertySourcesPlaceholderConfigurer  requestContextFilter  requestMappingHandlerAdapter  requestMappingHandlerMapping  resourceHandlerMapping  restTemplateBuilder  serverProperties  simpleControllerHandlerAdapter  spring.http.encoding-org.springframework.boot.autoconfigure.web.HttpEncodingProperties  spring.http.multipart-org.springframework.boot.autoconfigure.web.MultipartProperties  spring.info-org.springframework.boot.autoconfigure.info.ProjectInfoProperties  spring.jackson-org.springframework.boot.autoconfigure.jackson.JacksonProperties  spring.mvc-org.springframework.boot.autoconfigure.web.WebMvcProperties  spring.resources-org.springframework.boot.autoconfigure.web.ResourceProperties  springApplicationAdminRegistrar  springbootApplication  springbootFirstApplication  standardJacksonObjectMapperBuilderCustomizer  stringHttpMessageConverter  tomcatEmbeddedServletContainerFactory  viewControllerHandlerMapping  viewResolver  websocketContainerCustomizer  welcomePageHandlerMapping |

**在程序启动的时候，springboot自动诸如注入了40-50个bean.**

## 三.单元测试

通过@RunWith() @SpringBootTest开启注解：

package com.skytech.springboot.Test;  
  
import org.junit.Before;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.context.embedded.LocalServerPort;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.boot.test.web.client.TestRestTemplate;  
import org.springframework.http.ResponseEntity;  
import org.springframework.test.context.junit4.SpringRunner;  
  
import java.net.URL;  
  
import static org.hamcrest.MatcherAssert.assertThat;  
import static org.hamcrest.Matchers.equalTo;  
  
@RunWith(SpringRunner.class)  
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.***RANDOM\_PORT***)  
public class HelloControllerTest {  
 @LocalServerPort  
 private int **port**;  
  
 private URL **base**;  
  
 @Autowired  
 private TestRestTemplate **template**;  
  
  
 @Before  
 public void setUp() throws Exception {  
 this.**base** = new URL("http://localhost:" + **port** + "/");  
 }  
  
 @Test  
 public void getHello() throws Exception {  
 ResponseEntity<String> **response** = **template**.getForEntity(**base**.toString(), String.class);  
 assertThat(**response**.getBody(), equalTo("Greetings from Spring Boot!"));  
 }  
}



测试通过!

在**SpringbootApplication加上这个**

@SpringBootApplication  
@ServletComponentScan( basePackages = "com.skytech.springboot.\*")  
@ComponentScan( basePackages = "com.skytech.springboot.\*")  
@EnableAutoConfiguration

# 第二篇：Spring Boot配置文件详解

springboot采纳了建立生产就绪Spring应用程序的观点。 Spring Boot优先于配置的惯例，旨在让您尽快启动和运行。在一般情况下，我们不需要做太多的配置就能够让spring boot正常运行。在一些特殊的情况下，我们需要做修改一些配置，或者需要有自己的配置属性。

## 一、自定义属性

当我们创建一个springboot项目的时候,系统默认会为我们在src/main/java/resource目录下创建一个application.properites.

个人习惯,我会将这个文件修改为application.yml文件,两种文件格式都支持.

在application.yml自定义一组属性:

my:

name:forezp

age:12

如果你需要读取配置文件的值只需要加@Value(“${属性名}”);

package com.skytech.springboot.controller;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class MiyaController {  
 @Value("${my.name}")  
 private String **name**;  
  
 @Value("${my.age}")  
 private int **age**;  
  
 @RequestMapping("/miya")  
 public String miya(){  
 return **name** +":"+**age**;  
 }  
}

## 二、将配置文件的属性赋给实体类

当我们有很多配置属性的时候，这时我们会把这些属性作为字段来创建一个javabean，并将属性值赋予给他们,比如：

my:  
 name: forezp  
 age: 12  
 number: ${random.int}  
 uuid : ${random.uuid}  
 max: ${random.int(10)}  
 value: ${random.value}  
 greeting: hi,i'm ${my.name}

其中配置文件中用到了${random} ，它可以用来生成各种不同类型的随机值。

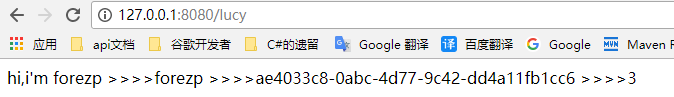
怎么讲这些属性赋于给一个javabean 呢，首先创建一个javabean ：

package com.skytech.springboot.entity;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;  
  
@ConfigurationProperties(prefix = "my")  
@Component  
public class ConfigBean {  
  
 private String **name**;  
 private int **age**;  
 private int **number**;  
 private String **uuid**;  
 private int **max**;  
 private String **value**;  
 private String **greeting**;

需要加个注解@ConfigurationProperties，并加上它的prrfix。另外@Component可加可不加。另外spring-boot-configuration-processor依赖可加可不加，具体原因不详。

另外需要在应用类或者application类，加EnableConfigurationProperties注解。

package com.skytech.springboot.controller;  
  
import com.skytech.springboot.entity.ConfigBean;  
import com.skytech.springboot.entity.User;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
@EnableConfigurationProperties({ConfigBean.class, User.class})  
public class LucyController {  
 @Autowired  
 ConfigBean **configBean**;  
  
 @RequestMapping("/lucy")  
 public String lucy() {  
 return **configBean**.getGreeting() + " >>>>" + **configBean**.getName() + " >>>>" + **configBean**.getUuid() + " >>>>" + **configBean**.getMax();  
 }  
}



## 三、自定义配置文件

上面介绍的是我们都把配置文件写到application.yml中。有时我们不愿意把配置都写到application配置文件中，这时需要我们自定义配置文件，比如test.properties:

|  |
| --- |
| **com.forezp.name**=**forezp com.forezp.age**=**12** |

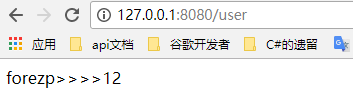
怎么将这个配置文件信息赋予给一个javabean呢？

|  |
| --- |
| package com.skytech.springboot.entity;  import org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.context.annotation.Configuration; import org.springframework.context.annotation.PropertySource;  @Configuration @PropertySource(value = "classpath:test.properties") @ConfigurationProperties(prefix = "com.forezp") public class User {   private String **name**;  private int **age**;   public String getName() {  return **name**;  }   public void setName(String name) {  this.**name** = name;  }   public int getAge() {  return **age**;  }   public void setAge(int age) {  this.**age** = age;  } } |

在最新版本的springboot，需要加这三个注解。@Configuration   
@PropertySource(value = “classpath:test.properties”)   
@ConfigurationProperties(prefix = “com.forezp”);在1.4版本需要 PropertySource加上location。

package com.skytech.springboot.controller;  
  
import com.skytech.springboot.entity.ConfigBean;  
import com.skytech.springboot.entity.User;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
@EnableConfigurationProperties({ConfigBean.class, User.class})  
public class LucyController {  
 @Autowired  
 ConfigBean **configBean**;  
  
 @RequestMapping("/lucy")  
 public String lucy() {  
 return **configBean**.getGreeting() + " >>>>" + **configBean**.getName() + " >>>>" + **configBean**.getUuid() + " >>>>" + **configBean**.getMax();  
 }  
  
 @Autowired  
 User **user**;  
  
 @RequestMapping("/user")  
 public String user() {  
 return **user**.getName() +">>>>"+ **user**.getAge();  
 }  
}

启动工程，打开localhost:8080/user;浏览器会显示：



## 四、多个环境配置文件

在现实的开发环境中，我们需要不同的配置环境；格式为application-{profile}.properties，其中{profile}对应你的环境标识，比如：

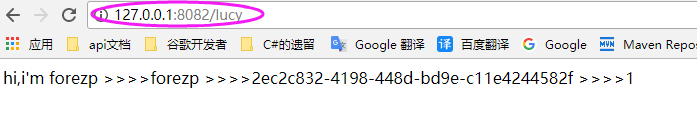
* application-test.properties：测试环境
* application-dev.properties：开发环境
* application-prod.properties：生产环境

怎么使用？只需要我们在application.yml中加：

spring:  
 profiles:  
 active: dev

其中application-dev.yml:

server:  
 port: 8082



# 第三篇：SpringBoot用JdbcTemplates访问Mysql

本文介绍springboot通过jdbc访问关系型mysql,通过spring的JdbcTemplate去访问。

准备工作

jdk 1.8

maven 3.0

idea

mysql

## 初始化mysql:

|  |
| --- |
| -- create table `account`  Use test;  DROP TABLE `account` IF EXISTS  CREATE TABLE `account` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(20) NOT NULL,  `money` double DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;  INSERT INTO `account` VALUES ('1', 'aaa', '1000');  INSERT INTO `account` VALUES ('2', 'bbb', '1000');  INSERT INTO `account` VALUES ('3', 'ccc', '1000'); |

创建工程

## 引入依赖：

在pom文件引入spring-boot-starter-jdbc的依赖：

<!-- spring-boot-starter-jdbc的依赖 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>

引入mysql连接类和连接池：

<!-- mysql连接类和连接池 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>  
  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.0.29</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

配置相关文件

在application.properties文件配置mysql的驱动类，数据库地址，数据库账号、密码信息。

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://192.168.1.37:3306/test  
spring.datasource.username=ework\_dev  
spring.datasource.password=ework\_dev@2017

通过引入这些依赖和配置一些基本信息，springboot就可以访问数据库类。

具体编码

## 实体类

package com.skytech.springboot.entity;  
  
public class Account {  
 private int **id**;  
 private String **name**;  
 private double **money**;

## dao层

package com.skytech.springboot.dao;  
  
import com.skytech.springboot.entity.Account;  
  
import java.util.List;  
  
public interface IAccountDAO {  
 int add(Account account);  
  
 int update(Account account);  
  
 int delete(int id);  
  
 Account findAccountById(int id);  
  
 List<Account> findAccountList();  
}

## dao具体的实现类：

package com.skytech.springboot.dao.impl;  
  
import com.skytech.springboot.dao.IAccountDAO;  
import com.skytech.springboot.entity.Account;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.BeanPropertyRowMapper;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.stereotype.Repository;  
  
import java.util.List;  
  
@Repository  
public class AccountDaoImpl implements IAccountDAO {  
  
 @Autowired  
 private JdbcTemplate **jdbcTemplate**;  
  
 @Override  
 public int add(Account account) {  
 return **jdbcTemplate**.update("insert into account(name,money) values(?,?)", account.getName(), account.getMoney());  
 }  
  
 @Override  
 public int update(Account account) {  
 return **jdbcTemplate**.update("update account SET name = ? , money = ? where id = ?", account.getName(), account.getMoney(), account.getId());  
 }  
  
 @Override  
 public int delete(int id) {  
 return **jdbcTemplate**.update("DELETE from account where id = ?", id);  
 }  
  
 @Override  
 public Account findAccountById(int id) {  
 List<Account> **list** = **jdbcTemplate**.query("select *\** from account where id=?", new Object[]{id}, new BeanPropertyRowMapper(Account.class));  
 if (**list** != null && **list**.size() > 0) {  
 Account **account** = **list**.get(0);  
 return **account**;  
 } else {  
 return null;  
 }  
 }  
  
 @Override  
 public List<Account> findAccountList() {  
 List<Account> **list** = **jdbcTemplate**.query("select *\** from account", new Object[]{}, new BeanPropertyRowMapper(Account.class));  
 if (**list** != null && **list**.size() > 0) {  
 return **list**;  
 } else {  
 return null;  
 }  
 }  
}

## service层

package com.skytech.springboot.service;  
  
import com.skytech.springboot.entity.Account;  
  
import java.util.List;  
  
public interface IAccountService {  
 int add(Account account);  
  
 int update(Account account);  
  
 int delete(int id);  
  
 Account findAccountById(int id);  
  
 List<Account> findAccountList();  
}

## service具体实现类：

package com.skytech.springboot.service.impl;  
  
import com.skytech.springboot.dao.IAccountDAO;  
import com.skytech.springboot.entity.Account;  
import com.skytech.springboot.service.IAccountService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class AccountService implements IAccountService {  
  
 @Autowired  
 IAccountDAO **accountDao**;  
  
 @Override  
 public int add(Account account) {  
 return **accountDao**.add(account);  
 }  
  
 @Override  
 public int update(Account account) {  
 return **accountDao**.update(account);  
 }  
  
 @Override  
 public int delete(int id) {  
 return **accountDao**.delete(id);  
 }  
  
 @Override  
 public Account findAccountById(int id) {  
 return **accountDao**.findAccountById(id);  
 }  
  
 @Override  
 public List<Account> findAccountList() {  
 return **accountDao**.findAccountList();  
 }  
}

## 构建一组restful api来展示

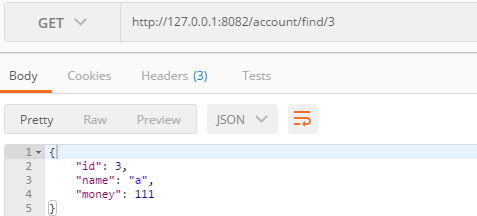
package com.skytech.springboot.controller;  
  
import com.skytech.springboot.entity.Account;  
import com.skytech.springboot.service.IAccountService;  
import org.apache.log4j.Logger;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/account")  
public class AccountController {  
  
 private static Logger **logger** = Logger.getLogger(AccountController.class);  
  
 @Autowired  
 IAccountService **accountService**;  
  
  
 @RequestMapping(value = "/list", method = RequestMethod.***GET***)  
 public List<Account> getAccounts() {  
 return **accountService**.findAccountList();  
 }  
  
  
 @RequestMapping(value = "/find/{id}", method = RequestMethod.***GET***)  
 public Account getAccountById(@PathVariable("id") int id) {  
 return **accountService**.findAccountById(id);  
 }  
  
  
 @RequestMapping(value = "/delete/{id}", method = RequestMethod.***DELETE***)  
 public int delete(@PathVariable("id") int id) {  
 return **accountService**.delete(id);  
 }  
  
  
 @RequestMapping(value = "/update/{id}", method = RequestMethod.***PUT***)  
 public String updateAccount(@PathVariable("id") int id,  
 @RequestParam(value = "name", required = true) String name,  
 @RequestParam(value = "money", required = true) double money) {  
 Account **account** = new Account();  
 **account**.setMoney(money);  
 **account**.setName(name);  
 **account**.setId(id);  
 int **t** = **accountService**.update(**account**);  
 if (**t** == 1) {  
 return **account**.toString();  
 } else {  
 return "fail";  
 }  
 }  
  
 @RequestMapping(value = "/add", method = RequestMethod.***POST***)  
 public String postAccount(@RequestParam(value = "name") String name,  
 @RequestParam(value = "money") Double money) {  
 Account **account** = new Account();  
 **account**.setMoney(money);  
 **account**.setName(name);  
 int **t** = **accountService**.add(**account**);  
 if (**t** == 1) {  
 return **account**.toString();  
 } else {  
 return "fail";  
 }  
 }  
}

## postman测试:

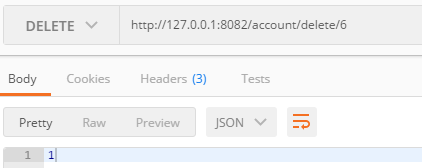
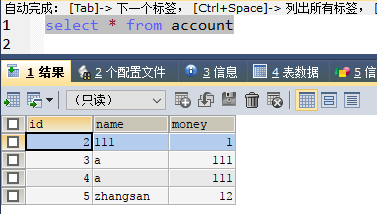
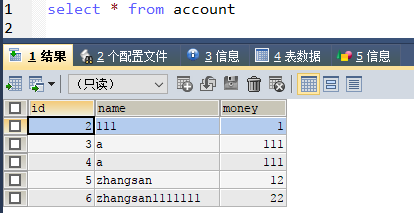
### http://127.0.0.1:8082/account/list



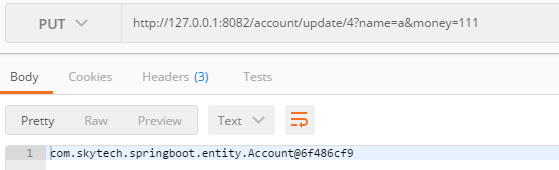
### http://127.0.0.1:8082/account/find/3



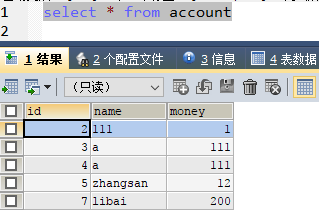
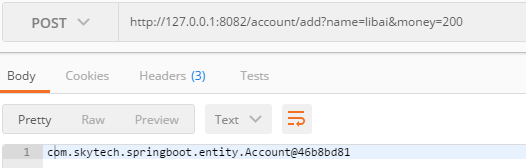
### http://127.0.0.1:8082/account/delete/6



### http://127.0.0.1:8082/account/update/4?name=a&money=111



### http://127.0.0.1:8082/account/add?name=libai&money=200



# 第四篇：SpringBoot 整合JPA

JPA全称Java Persistence API.JPA通过JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体对象持久化到数据库中。

JPA 的目标之一是制定一个可以由很多供应商实现的API，并且开发人员可以编码来实现该API，而不是使用私有供应商特有的API。

JPA是需要Provider来实现其功能的，Hibernate就是JPA Provider中很强的一个，应该说无人能出其右。从功能上来说，JPA就是Hibernate功能的一个子集。

## 添加相关依赖

添加spring-boot-starter-jdbc依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
</**dependency**>

添加mysql连接类和连接池类：

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

添加版本附加依赖(hibernate更新缺失的文件)

<!-- https://mvnrepository.com/artifact/org.antlr/antlr-complete -->  
<**dependency**>  
 <**groupId**>org.antlr</**groupId**>  
 <**artifactId**>antlr-complete</**artifactId**>  
 <**version**>3.5.2</**version**>  
</**dependency**>

配置数据源，在application.properties文件配置：

**spring:  
 datasource:  
 driver-class-name: com**.**mysql**.**jdbc**.**Driver  
 url:** jdbc:mysql://192.168.1.37:3306/test  
 **username:** ework\_dev  
 **password:** ework\_dev@2017  
  
 **jpa:  
 hibernate:  
 ddl-auto:** update  
 **show-sql: true**

注意，如果通过jpa在数据库中建表，将jpa.hibernate,ddl-auto改为create，建完表之后，要改为update,要不然每次重启工程会删除表并新建。

## 创建实体类

通过@Entity 表明是一个映射的实体类， @Id表明id， @GeneratedValue 字段自动生成

**package com.skytech.springbootjpa.web**;  
  
**import javax.persistence.**Entity;  
**import javax.persistence.**GeneratedValue;  
**import javax.persistence.**Id;  
  
@Entity  
**public class Account** {  
 @Id  
 @GeneratedValue  
 **private int** *id*;  
 **private String** *name*;  
 **private double** *money*;  
  
 **public int** getId() {  
 **return** *id*;  
 }  
  
 **public void** setId(**int** id) {  
 **this**.*id* = id;  
 }  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public double** getMoney() {  
 **return** *money*;  
 }  
  
 **public void** setMoney(**double** money) {  
 **this**.*money* = money;  
 }  
}

## Dao层

数据访问层，通过编写一个继承自 JpaRepository 的接口就能完成数据访问,其中包含了几本的单表查询的方法，非常的方便。值得注意的是，这个Account 对象名，而不是具体的表名，另外Interger是主键的类型，一般为Integer或者Long

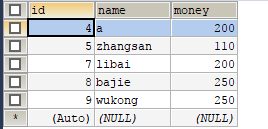
**package com.skytech.springbootjpa.web**;  
  
**import org.springframework.data.jpa.repository.JpaRepository**;  
  
**public interface AccountDao extends JpaRepository**<**Account**, **Integer**> {  
}

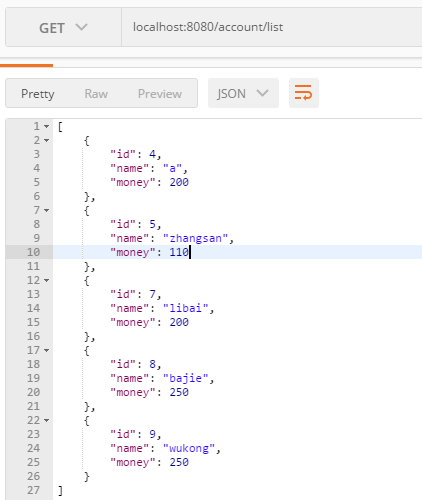
## Web层

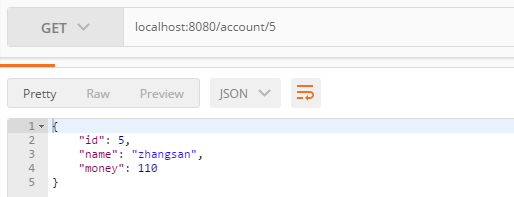
在这个栗子中我简略了service层的书写，在实际开发中，不可省略。新写一个controller，写几个restful api来测试数据的访问。

**package com.skytech.springbootjpa.web**;  
  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.web.bind.annotation.**\*;  
  
**import java.util.List**;  
  
@RestController  
@RequestMapping("/account")  
**public class AccountController** {  
  
 @Autowired  
 **AccountDao** *accountDao*;  
  
 @RequestMapping(value = "/list", method = **RequestMethod**.***GET***)  
 **public List**<**Account**> getAccounts() {  
 **return** *accountDao*.findAll();  
 }  
  
 @RequestMapping(value = "/{id}", method = **RequestMethod**.***GET***)  
 **public Account** getAccountById(@PathVariable("id") **int** id) {  
 **return** *accountDao*.findOne(id);  
 }  
  
 @RequestMapping(value = "/{id}", method = **RequestMethod**.***PUT***)  
 **public String** updateAccount(@PathVariable("id") **int** id, @RequestParam(value = "name", required = **true**) **String** name,  
 @RequestParam(value = "money", required = **true**) **double** money) {  
 **Account** account = **new** Account();  
 account.setMoney(money);  
 account.setName(name);  
 account.setId(id);  
 **Account** account1 = *accountDao*.saveAndFlush(account);  
  
 **return** account1.toString();  
  
 }  
  
 @RequestMapping(value = "", method = **RequestMethod**.***POST***)  
 **public String** postAccount(@RequestParam(value = "name") **String** name,  
 @RequestParam(value = "money") **double** money) {  
 **Account** account = **new** Account();  
 account.setMoney(money);  
 account.setName(name);  
 **Account** account1 = *accountDao*.save(account);  
 **return** account1.toString();  
  
 }  
  
  
}

通过postman请求测试，代码已经全部通过测试。







## 参考资料

[accessing-data-jpa](https://spring.io/guides/gs/accessing-data-jpa/)

# 第六篇：springboot整合mybatis

本文主要讲解如何在springboot下整合mybatis，并访问数据库。由于mybatis这个框架太过于流行，所以我就不讲解了。

## 引入依赖

在pom文件引入mybatis-spring-boot-starter的依赖：

<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.0</version>  
</dependency>

引入数据库连接依赖：

<!-- mysql连接类和连接池 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>  
  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.0.29</version>  
</dependency>

引入数据源

application.properties配置文件中引入数据源：

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://192.168.1.37:3306/test  
spring.datasource.username=ework\_dev  
spring.datasource.password=ework\_dev@2017

这样，springboot就可以访问数据了。

## 创建数据库表

建表语句：

|  |
| --- |
| -- create table `account`  Use test;  DROP TABLE `account` IF EXISTS  CREATE TABLE `account` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(20) NOT NULL,  `money` double DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;  INSERT INTO `account` VALUES ('1', 'aaa', '1000');  INSERT INTO `account` VALUES ('2', 'bbb', '1000');  INSERT INTO `account` VALUES ('3', 'ccc', '1000'); |

具体实现

这篇文篇通过注解的形式实现。

## 创建实体：

package com.skytech.springboot.entity;  
  
public class Account {  
  
 private int **id**;  
 private String **name**;  
 private double **money**;  
  
 public int getId() {  
 return **id**;  
 }  
  
 public void setId(int id) {  
 this.**id** = id;  
 }  
  
 public String getName() {  
 return **name**;  
 }  
  
 public void setName(String name) {  
 this.**name** = name;  
 }  
  
 public double getMoney() {  
 return **money**;  
 }  
  
 public void setMoney(double money) {  
 this.**money** = money;  
 }  
}

## dao接口(没有实现层)

package com.skytech.springboot.dao;  
  
import com.skytech.springboot.entity.Account;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
  
@Mapper  
public interface IAccountMybatisDAO {  
  
 @Insert("insert into account(name,money) values(#{name},#{money})")  
 int add(@Param("name") String name, @Param("money") double money);  
  
 @Update("update account set name=#{name},money=#{money} where id=#{id}")  
 int update(@Param("name") String name, @Param("money") double money, @Param("id") int id);  
  
 @Delete("delete from account where id =#{id}")  
 int delete(int id);  
  
 @Select("select id,name,money from account where id = #{id}")  
 Account findAccountById(@Param("id") int id);  
  
 @Select("select \* from account")  
 List<Account> findAccountList();  
}

## service层(没有接口层)

package com.skytech.springboot.service.impl;  
  
import com.skytech.springboot.dao.IAccountDAO;  
import com.skytech.springboot.dao.IAccountMybatisDAO;  
import com.skytech.springboot.entity.Account;  
import org.apache.ibatis.annotations.\*;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class AccountMybatisService {  
  
 @Autowired  
 IAccountMybatisDAO **accountMybatisDao**;  
  
 public int add(String name, double money) {  
 return **accountMybatisDao**.add(name, money);  
 }  
  
 public int update(String name, double money, int id) {  
 return **accountMybatisDao**.update(name, money, id);  
 }  
  
 public int delete(int id) {  
 return **accountMybatisDao**.delete(id);  
 }  
  
 public Account findAccountById(int id) {  
 return **accountMybatisDao**.findAccountById(id);  
 }  
  
 public List<Account> findAccountList() {  
 return **accountMybatisDao**.findAccountList();  
 }  
}

## controller层，构建restful API

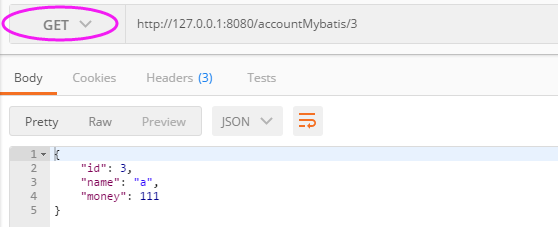
package com.skytech.springboot.controller;  
  
import com.skytech.springboot.entity.Account;  
import com.skytech.springboot.service.impl.AccountMybatisService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/accountMybatis")  
public class AccountMybatisController {  
 @Autowired  
 AccountMybatisService **accountMybatisService**;  
  
  
 @RequestMapping(value = "/list", method = RequestMethod.***GET***)  
 public List<Account> getAccounts() {  
 return **accountMybatisService**.findAccountList();  
 }  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.***GET***)  
 public Account getAccountById(@PathVariable("id") int id) {  
 return **accountMybatisService**.findAccountById(id);  
 }  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.***PUT***)  
 public String updateAccount(@PathVariable("id") int id,  
 @RequestParam(value = "name", required = true) String name,  
 @RequestParam(value = "money", required = true) double money) {  
 int **t** = **accountMybatisService**.update(name, money, id);  
 if (**t** == 1) {  
 return "success";  
 } else {  
 return "fail";  
 }  
  
 }  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.***DELETE***)  
 public String delete(@PathVariable(value = "id") int id) {  
 int **t** = **accountMybatisService**.delete(id);  
 if (**t** == 1) {  
 return "success";  
 } else {  
 return "fail";  
 }  
  
 }  
  
 @RequestMapping(value = "", method = RequestMethod.***POST***)  
 public String postAccount(@RequestParam(value = "name") String name,  
 @RequestParam(value = "money") double money) {  
  
 int **t** = **accountMybatisService**.add(name, money);  
 if (**t** == 1) {  
 return "success";  
 } else {  
 return "fail";  
 }  
 }  
}

## postman测试

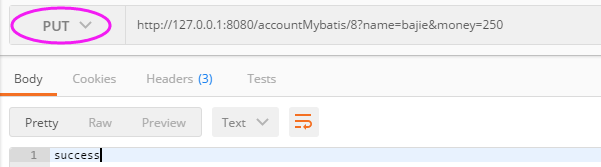
### http://127.0.0.1:8080/accountMybatis/list



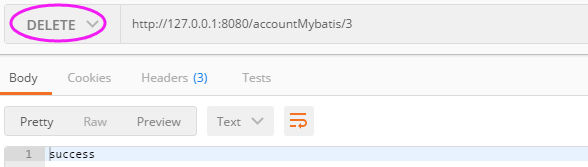
### http://127.0.0.1:8080/accountMybatis/3



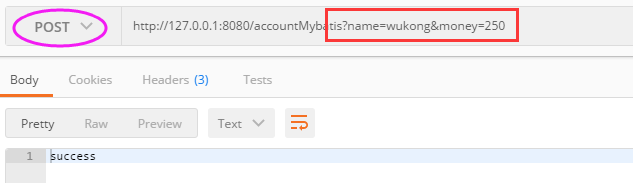
### http://127.0.0.1:8080/accountMybatis/8?name=bajie&money=250



### http://127.0.0.1:8080/accountMybatis/3



### http://127.0.0.1:8080/accountMybatis?name=wukong&money=250



# 第七篇：springboot开启声明式事务

**springboot开启事务很简单，只需要一个注解@Transactional 就可以了**。因为在springboot中已经默认对jpa、jdbc、mybatis开启了事事务，引入它们依赖的时候，事物就默认开启。当然，如果你需要用其他的orm，比如beatlsql，就需要自己配置相关的事物管理器。

## 准备阶段

以上一篇文章的代码为例子，即springboot整合mybatis，上一篇文章是基于注解来实现mybatis的数据访问层，这篇文章基于xml的来实现，并开启声明式事务。

## 环境依赖

在pom文件中引入mybatis启动依赖：

<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.0</version>  
</dependency>

引入mysql 依赖

<!-- mysql连接类和连接池 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>  
  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.0.29</version>  
</dependency>

初始化数据库脚本

|  |
| --- |
| -- create table `account`  Use test;  DROP TABLE `account` IF EXISTS  CREATE TABLE `account` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(20) NOT NULL,  `money` double DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;  INSERT INTO `account` VALUES ('1', 'aaa', '1000');  INSERT INTO `account` VALUES ('2', 'bbb', '1000');  INSERT INTO `account` VALUES ('3', 'ccc', '1000'); |

## 配置数据源

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://192.168.1.37:3306/test  
spring.datasource.username=ework\_dev  
spring.datasource.password=ework\_dev@2017  
mybatis.mapper-locations=classpath\*:mybatis/\*Mapper.xml  
mybatis.type-aliases-package=com.skytech.springboot.entity

通过配置mybatis.mapper-locations来指明mapper的xml文件存放位置，我是放在resources/mybatis文件下的。mybatis.type-aliases-package来指明和数据库映射的实体的所在包。

经过以上步骤，springboot就可以通过mybatis访问数据库来。

## 创建实体类

package com.skytech.springboot.entity;  
  
public class Account {  
  
 private int **id**;  
 private String **name**;  
 private double **money**;  
  
  
 public int getId() {  
 return **id**;  
 }  
  
 public void setId(int id) {  
 this.**id** = id;  
 }  
  
 public String getName() {  
 return **name**;  
 }  
  
 public void setName(String name) {  
 this.**name** = name;  
 }  
  
 public double getMoney() {  
 return **money**;  
 }  
  
 public void setMoney(double money) {  
 this.**money** = money;  
 }  
}

## 数据访问dao 层接口：

package com.skytech.springboot.dao;  
  
import com.skytech.springboot.entity.Account;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
  
@Mapper  
public interface IAccountMybatisDAO {  
 //测试声明式事务用的  
 void updateTransfer( @Param("money") double money, @Param("id") int id);  
}

## mapper:

*<?*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.skytech.springboot.dao.IAccountMybatisDAO">  
 <update id="updateTransfer">  
 UPDATE account set money=#{money} WHERE id = #{id}  
 </update>  
</mapper>

## service层

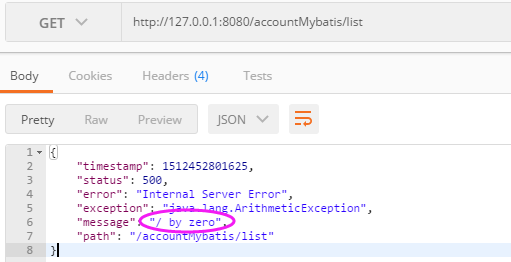
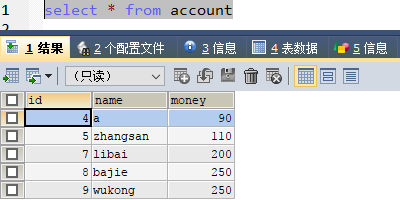
package com.skytech.springboot.service.impl;  
  
import com.skytech.springboot.dao.IAccountDAO;  
import com.skytech.springboot.dao.IAccountMybatisDAO;  
import com.skytech.springboot.entity.Account;  
import org.apache.ibatis.annotations.\*;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
import java.util.List;  
  
@Service  
public class AccountMybatisService {  
  
 @Autowired  
 IAccountMybatisDAO **accountMybatisDao**;  
  
 //测试声明式事务用的  
 @Transactional  
 public void transfer() {  
 try{  
 **accountMybatisDao**.updateTransfer(2,4);  
 int **i** = 1/0;  
 **accountMybatisDao**.updateTransfer(2,5);  
 }catch(Exception ex){  
 ex.printStackTrace();  
 }  
 }  
}

@Transactional，声明事务，并设计一个转账方法，用户1减10块，用户2加10块。在用户1减10 ，之后，抛出异常，即用户2加10块钱不能执行，当加注解@Transactional之后，两个人的钱都没有增减。当不加@Transactional，用户1减了10，用户2没有增加，即没有操作用户2 的数据。可见@Transactional注解开启了事物。

结语

springboot 开启事物很简单，只需要加一行注解就可以了，前提你用的是jdbctemplate, jpa, mybatis，这种常见的orm。

## 测试



数据库数据没做任何改变.

# 第八篇：springboot整合mongodb

这篇文章主要介绍springboot如何整合mongodb。

准备工作

* 安装 MongoDB
* jdk 1.8
* maven 3.0
* idea

## 环境依赖

在pom文件引入spring-boot-starter-data-mongodb依赖：

<!--mongodb的依赖-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-mongodb</artifactId>  
</dependency>

数据源配置

如果mongodb端口是默认端口，并且没有设置密码，可不配置，sprinboot会开启默认的。

spring.data.mongodb.uri=mongodb://localhost:27017/hj

mongodb设置了密码，这样配置：

spring.data.mongodb.uri=mongodb://name:pass@localhost:27017/hj

## 定义一个简单的实体

package com.skytech.springboot.entity;  
  
import org.springframework.data.annotation.Id;  
  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
  
public class Customer {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***AUTO***)  
 private String **id**;  
  
 private String **name**;  
 private int **age**;  
  
  
 public **Customer**() {  
 }  
  
 public **Customer**(String id, String name, int age) {  
 this.**id** = id;  
 this.**name** = name;  
 this.**age** = age;  
 }  
  
  
 public String getId() {  
 return **id**;  
 }  
  
 public void setId(String id) {  
 this.**id** = id;  
 }  
  
 public String getName() {  
 return **name**;  
 }  
  
 public void setName(String name) {  
 this.**name** = name;  
 }  
  
 public int getAge() {  
 return **age**;  
 }  
  
 public void setAge(int age) {  
 this.**age** = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Customer{" +  
 "id=" + **id** +  
 ", name='" + **name** + '\'' +  
 ", age=" + **age** +  
 '}';  
 }  
}

## dao

package com.skytech.springboot.dao;  
  
import com.skytech.springboot.entity.Customer;  
import org.springframework.data.mongodb.repository.MongoRepository;  
  
public interface CustomerRepository extends MongoRepository<Customer, String> {  
 Customer findByName(String name);  
}

## service

package com.skytech.springboot.service;  
  
import com.skytech.springboot.entity.Customer;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface ICustomerService {  
 void save(Customer customer);  
  
 Customer findByName(String name);  
  
}

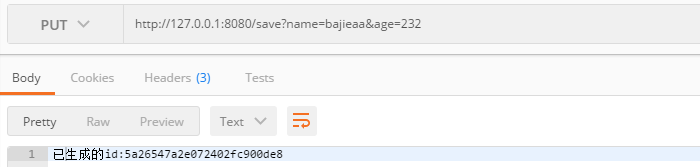
## service实现

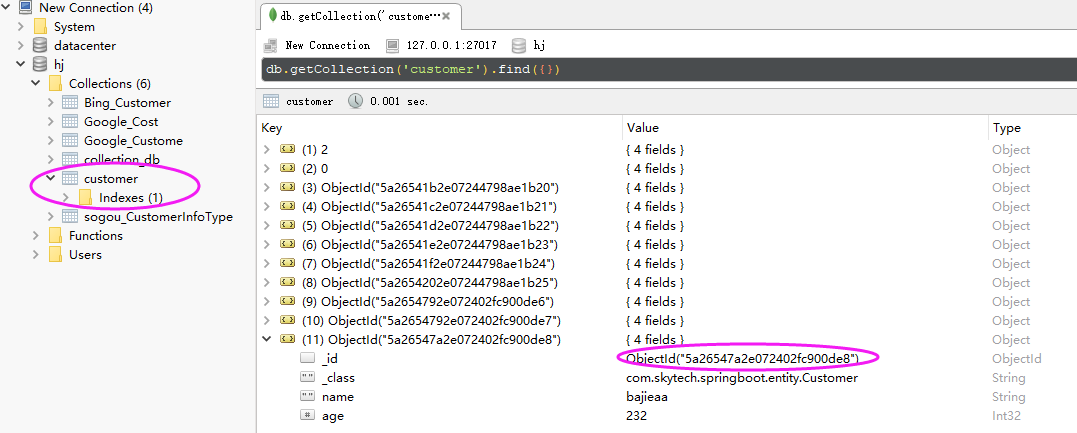
package com.skytech.springboot.service.impl;  
  
import com.skytech.springboot.dao.CustomerRepository;  
import com.skytech.springboot.entity.Customer;  
import com.skytech.springboot.service.ICustomerService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class CustomerServiceImpl implements ICustomerService {  
  
 @Autowired  
 private CustomerRepository **customerRepository**;  
  
  
 @Override  
 public void save(Customer customer) {  
 **customerRepository**.save(customer);  
 }  
  
 @Override  
 public Customer findByName(String name) {  
 return this.**customerRepository**.findByName(name);  
 }  
}

## Controller

package com.skytech.springboot.controller;  
  
import com.skytech.springboot.entity.Customer;  
import com.skytech.springboot.service.ICustomerService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.mongodb.core.MongoTemplate;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
public class CustomerController {  
 @Autowired  
 private ICustomerService **customerService**;  
  
 @Autowired  
 private MongoTemplate **mongoTemplate**;  
  
  
 @RequestMapping(value = "/save", method = RequestMethod.***PUT***)  
 public String save(@RequestParam(value = "name") String name,  
 @RequestParam(value = "age") int age) {  
 Customer **customer** = new Customer();  
 **customer**.setName(name);  
 **customer**.setAge(age);  
 **mongoTemplate**.save(**customer**);  
  
 return "已生成的id:" + **customer**.getId();  
 }  
  
 @GetMapping("/find")  
 public List<Customer> find() {  
 List<Customer> **customerList** = **mongoTemplate**.findAll(Customer.class);  
 return **customerList**;  
 }  
  
  
 @GetMapping("/findByName")  
 public Customer findByName(@RequestParam("name") String name) {  
 Customer **customer** = **customerService**.findByName(name);  
 return **customer**;  
 }  
}

## 测试





# 第九篇： springboot整合Redis

这篇文章主要介绍springboot整合redis，至于没有接触过redis的同学可以看下这篇文章：[5分钟带你入门Redis](http://blog.csdn.net/forezp/article/details/61471712)。

## 引入依赖：

在pom文件中添加redis依赖：

<!--springboot redis依赖-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

## Mysql依赖

<!-- mysql连接类和连接池 -->  
<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**scope**>runtime</**scope**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
 <**version**>1.0.29</**version**>  
</**dependency**>

Mybatis依赖

<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.1</**version**>  
</**dependency**>

## 配置数据源

#数据源  
**spring.datasource.driver-class-name**=com.mysql.jdbc.Driver  
**spring.datasource.url**=jdbc:mysql://192.168.1.37:3306/test  
**spring.datasource.username**=ework\_dev  
**spring.datasource.password**=ework\_dev@2017

Redis配置

#整合redis单机版,使用redis作为缓存  
# Redis数据库索引（默认为0）  
**spring.redis.database**=0  
# Redis服务器地址  
**spring.redis.host**=192.168.1.37  
# Redis服务器连接端口  
**spring.redis.port**=6379  
# Redis服务器连接密码（默认为空）  
**spring.redis.password**=root@redis

Mybatis配置

#加载mybatis配置文件  
**mybatis.mapper-locations**=classpath\*:mapper/\*Mapper.xml  
#定义别名  
**mybatis.type-aliases-package**=com.skytech.springbootdemo.domain

## 实体

**package com.skytech.springbootdemo.domain**;  
  
**import java.util.Date**;  
  
**public class User** {  
  
 **private int** *id*;  
 **private String** *name*;  
 **private double** *money*;  
  
 **public int** getId() {  
 **return** *id*;  
 }  
  
 **public void** setId(**int** id) {  
 **this**.*id* = id;  
 }  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public double** getMoney() {  
 **return** *money*;  
 }  
  
 **public void** setMoney(**double** money) {  
 **this**.*money* = money;  
 }  
}

## mapper类

**package com.skytech.springbootdemo.mapper**;  
  
**import com.skytech.springbootdemo.domain.User**;  
**import org.apache.ibatis.annotations.**Mapper;  
  
**import java.util.List**;  
  
@Mapper  
**public interface UserMapper** {  
 **public List**<**User**> findAll();  
}

## mapper.xml文件

*<?***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.skytech.springbootdemo.mapper.UserMapper">  
  
  
 <**select id**="findAll" **resultType**="user">  
 select *\** from account;  
 </**select**>  
</**mapper**>

## Service

**package com.skytech.springbootdemo.service**;  
  
**import com.skytech.springbootdemo.domain.User**;  
**import com.skytech.springbootdemo.mapper.UserMapper**;  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.cache.annotation.**Cacheable;  
**import org.springframework.stereotype.**Service;  
  
**import java.util.List**;  
  
@Service  
**public class UserService** {  
  
 @Autowired  
 **private UserMapper** *userMapper*;  
  
 /\*\*  
 \* 测试redis缓存  
 \* @return  
 \* @Cacheable(value = "findAll")这里是什么意思呢?  
 \* 表示在缓存区域开启了一个名叫findAll的缓存,findAll里缓存的是刚才查询的数据,  
 \* 下次可以根据这个名称findAll去寻找,也就是说给缓存取了一个唯一标识.我们称之为查询缓存  
 \*/  
 @Cacheable(value = "findAll")  
 **public List**<**User**> findAll() {  
 **System**.***out***.println("如果第二次没有打印此文字,说明缓存存在,没有执行此方法!");  
 **List**<**User**> list = *userMapper*.findAll();  
 **return** list;  
 }  
}

## controller

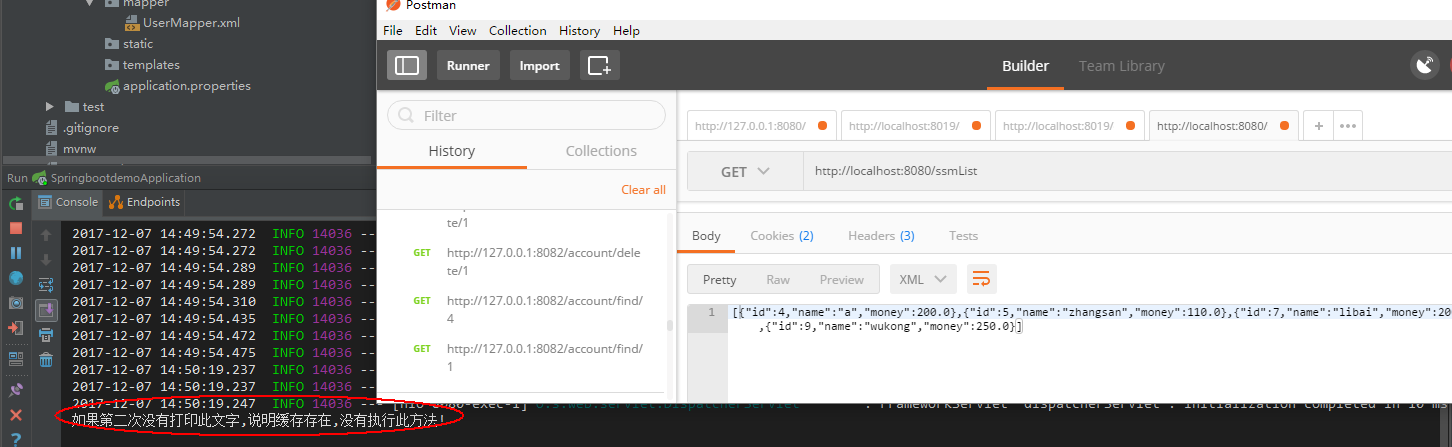
**package com.skytech.springbootdemo.controller**;  
  
**import com.skytech.springbootdemo.domain.User**;  
**import com.skytech.springbootdemo.service.UserService**;  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.web.bind.annotation.**RequestMapping;  
**import org.springframework.web.bind.annotation.**RestController;  
  
**import java.util.List**;  
  
@RestController  
**public class UserController** {  
  
 //service对象  
 @Autowired  
 **private UserService** *userService*;  
  
  
 /\*\*  
 \* 整合的ssm  
 \*  
 \* @return  
 \*/  
 @RequestMapping("/ssmList")  
 **public List**<**User**> findAll() {  
 **List**<**User**> list = *userService*.findAll();  
 **return** list;  
 }  
}

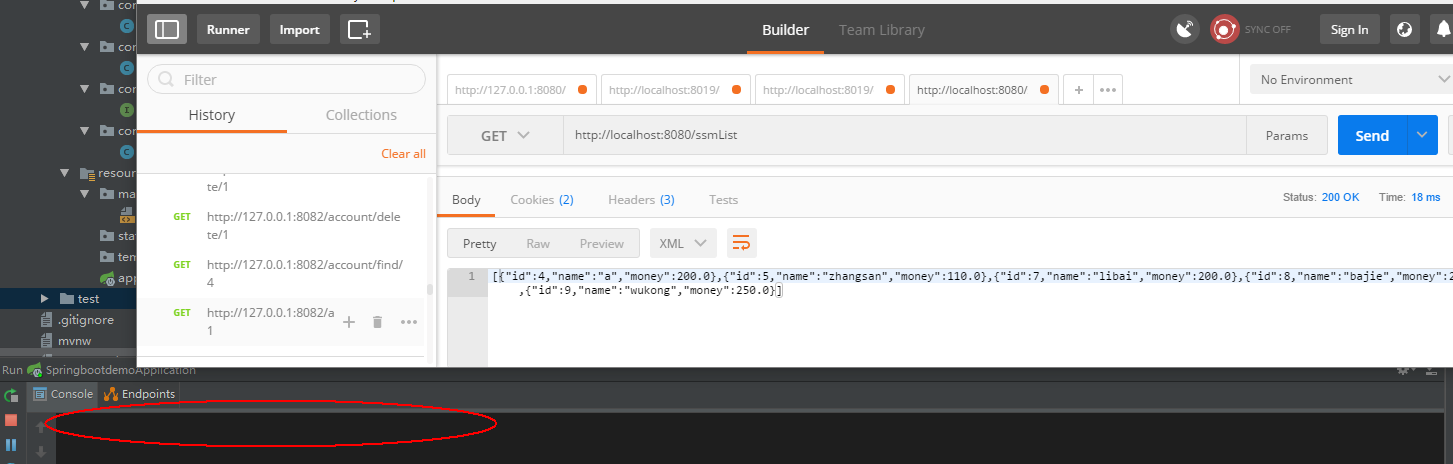
## springboot初始化程序开启缓存注解

**package com.skytech.springbootdemo**;  
  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**EnableAutoConfiguration;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.boot.web.servlet.**ServletComponentScan;  
**import org.springframework.cache.annotation.**EnableCaching;  
**import org.springframework.context.annotation.**ComponentScan;  
  
@SpringBootApplication  
@ServletComponentScan(basePackages = "com.skytech.springbootdemo.\*")  
@ComponentScan(basePackages = "com.skytech.springbootdemo.\*")  
@EnableAutoConfiguration  
**@EnableCaching**  
**public class SpringbootdemoApplication** {  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**SpringbootdemoApplication**.**class**, args);  
 }  
}

## 测试

### http://localhost:8080/ssmList





说明使用的是缓存

# 第十篇： 用spring Restdocs创建API文档

这篇文章将带你了解如何用spring官方推荐的restdoc去生成api文档。本文创建一个简单的springboot工程，将http接口通过Api文档暴露出来。只需要通过 JUnit单元测试和Spring的MockMVC就可以生成文档。

准备工作

你需要15min

Jdk 1.8

maven 3.0+

idea

## 创建工程

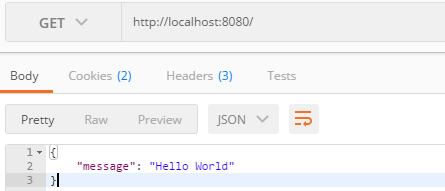
引入依赖，其pom文件：

<**dependency**>  
 <**groupId**>org.springframework.restdocs</**groupId**>  
 <**artifactId**>spring-restdocs-mockmvc</**artifactId**>  
 <**scope**>test</**scope**>  
</**dependency**>

## 在springboot通常创建一个controller:

**package com.skytech.springbootrestdocs.web**;  
  
**import org.springframework.web.bind.annotation.**GetMapping;  
**import org.springframework.web.bind.annotation.**RestController;  
  
**import java.util.Collections**;  
**import java.util.Map**;  
  
@RestController  
**public class HomeController** {  
  
 @GetMapping("/")  
 **public Map**<**String**, **Object**> greeting() {  
 **return Collections**.singletonMap("message", "Hello World");  
 }  
  
}

启动工程，访问localhost:8080，浏览器显示：



证明接口已经写好了，但是如何通过restdoc生存api文档呢

restdocs是通过单元测试生存snippets文件，然后snippets根据插件生成htm文档的。

## 建一个单元测试类：

**package com.skytech.springbootrestdocs**;  
  
**import com.skytech.springbootrestdocs.web.HomeController**;  
**import org.junit.**Test;  
**import org.junit.runner.**RunWith;  
  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.boot.test.autoconfigure.restdocs.**AutoConfigureRestDocs;  
**import org.springframework.boot.test.autoconfigure.web.servlet.**WebMvcTest;  
**import org.springframework.test.context.junit4.SpringRunner**;  
**import org.springframework.test.web.servlet.MockMvc**;  
  
**import static org.hamcrest.Matchers**.containsString;  
**import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation**.document;  
**import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders**.get;  
**import static org.springframework.test.web.servlet.result.MockMvcResultHandlers**.print;  
**import static org.springframework.test.web.servlet.result.MockMvcResultMatchers**.content;  
**import static org.springframework.test.web.servlet.result.MockMvcResultMatchers**.status;  
  
@RunWith(**SpringRunner**.**class**)  
@WebMvcTest(**HomeController**.**class**)  
@AutoConfigureRestDocs(outputDir = "target/snippets")  
**public class WebLayerTest** {  
  
 @Autowired  
 **private MockMvc** *mockMvc*;  
  
 @Test  
 **public void** shouldReturnDefaultMessage() **throws Exception** {  
 **this**.*mockMvc*.perform(get("/")).andDo(print()).andExpect(status().isOk())  
 .andExpect(content().string(containsString("Hello World")))  
 .andDo(document("home"));  
 }  
}

其中，@ AutoConfigureRestDocs注解开启了生成snippets文件，并指定了存放位置。

启动单元测试，测试通过，你会发现在target文件下生成了一个snippets文件夹，其目录结构如下：

└── target

└── snippets

└── home

└── httpie-request.adoc

└── curl-request.adoc

└── http-request.adoc

└── http-response.adoc

默认情况下，snippets是Asciidoctor格式的文件，包括request和reponse，另外其他两种httpie和curl两种流行的命令行的http请求模式。

到目前为止，只生成了Snippets文件，需要用Snippets文件生成文档。

怎么用Snippets

## 创建一个新文件src/main/asciidoc/index.adoc ：

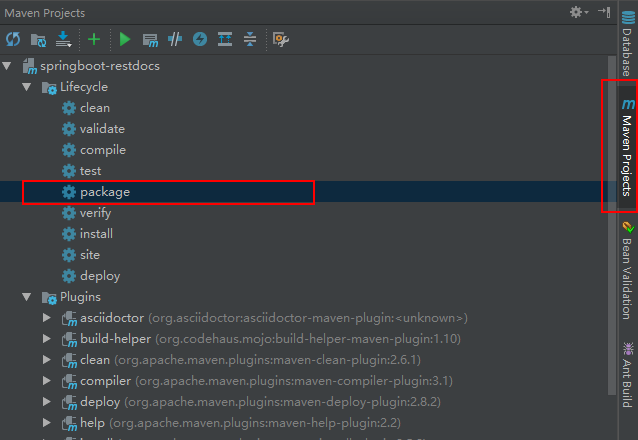
= 用 Spring REST Docs 构建文档  
  
This is an example output for a service running at http://localhost:8080:  
  
.request  
include::{snippets}/home/http-request.adoc[]  
  
.response  
include::{snippets}/home/http-response.adoc[]  
  
这个例子非常简单，通过单元测试和一些简单的配置就能够得到api文档了。

## 需要使用asciidoctor-maven-plugin插件，在其pom文件加上：

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
  
 <**plugin**>  
 <**groupId**>org.asciidoctor</**groupId**>  
 <**artifactId**>asciidoctor-maven-plugin</**artifactId**>  
 <**executions**>  
 <**execution**>  
 <**id**>generate-docs</**id**>  
 <**phase**>prepare-package</**phase**>  
 <**goals**>  
 <**goal**>process-asciidoc</**goal**>  
 </**goals**>  
 <**configuration**>  
 <**sourceDocumentName**>index.adoc</**sourceDocumentName**>  
 <**backend**>html</**backend**>  
 <**attributes**>  
 <**snippets**>${project.build.directory}/snippets</**snippets**>  
 </**attributes**>  
 </**configuration**>  
 </**execution**>  
 </**executions**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

## 这时只需要通过mvn package命令就可以生成文档了。

在/target/generated-docs下有个index.html，打开这个html,显示如下，界面还算简洁：







## 结语

通过单元测试，生存adoc文件，再用adoc文件生存html，只需要简单的几步就可以生成一个api文档的html文件，这个html文件你可以通网站发布出去。整个过程很简单，对代码无任何影响。

# 第十一篇：springboot集成swagger2，构建优雅的Restful API

swagger,中文“拽”的意思。它是一个功能强大的api框架，它的集成非常简单，不仅提供了在线文档的查阅，而且还提供了在线文档的测试。另外swagger很容易构建restful风格的api，简单优雅帅气，正如它的名字。

## 一、引入依赖

<!--swagger2-->  
<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger2</**artifactId**>  
 <**version**>2.6.1</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger-ui</**artifactId**>  
 <**version**>2.6.1</**version**>  
</**dependency**>

## 二、写配置类

**package com.skytech.springbootrestdocs.conf**;  
  
**import org.springframework.context.annotation.**Configuration;  
**import springfox.documentation.builders.ApiInfoBuilder**;  
**import springfox.documentation.builders.PathSelectors**;  
**import springfox.documentation.builders.RequestHandlerSelectors**;  
**import springfox.documentation.service.ApiInfo**;  
**import springfox.documentation.spi.DocumentationType**;  
**import springfox.documentation.spring.web.plugins.Docket**;  
**import springfox.documentation.swagger2.annotations.**EnableSwagger2;  
  
@Configuration  
@EnableSwagger2  
**public class Swagger2** {  
  
 **public Docket** createRestApi() {  
 **return new** Docket(**DocumentationType**.***SWAGGER\_2***)  
 .apiInfo(apiInfo())  
 .select()  
 .apis(**RequestHandlerSelectors**.basePackage("com.skytech.springbootrestdocs.controller"))  
 .paths(**PathSelectors**.any())  
 .build();  
 }  
  
 **private ApiInfo** apiInfo() {  
 **return new** ApiInfoBuilder()  
 .title("springboot利用swagger构建api文档")  
 .description("简单优雅的restful风格,http://www.runoob.com/w3cnote/restful-architecture.html")  
 .termsOfServiceUrl("http://www.runoob.com/w3cnote/restful-architecture.html")  
 .version("1.0")  
 .build();  
 }  
}

通过@Configuration注解，表明它是一个配置类，@EnableSwagger2开启swagger2。apiINfo()配置一些基本的信息。apis()指定扫描的包会生成文档。

## 三、写生产文档的注解

swagger通过注解表明该接口会生成文档，包括接口名、请求方法、参数、返回信息的等等。

* @Api：修饰整个类，描述Controller的作用
* @ApiOperation：描述一个类的一个方法，或者说一个接口
* @ApiParam：单个参数描述
* @ApiModel：用对象来接收参数
* @ApiProperty：用对象接收参数时，描述对象的一个字段
* @ApiResponse：HTTP响应其中1个描述
* @ApiResponses：HTTP响应整体描述
* @ApiIgnore：使用该注解忽略这个API
* @ApiError ：发生错误返回的信息
* @ApiParamImplicitL：一个请求参数
* @ApiParamsImplicit 多个请求参数

## Entity

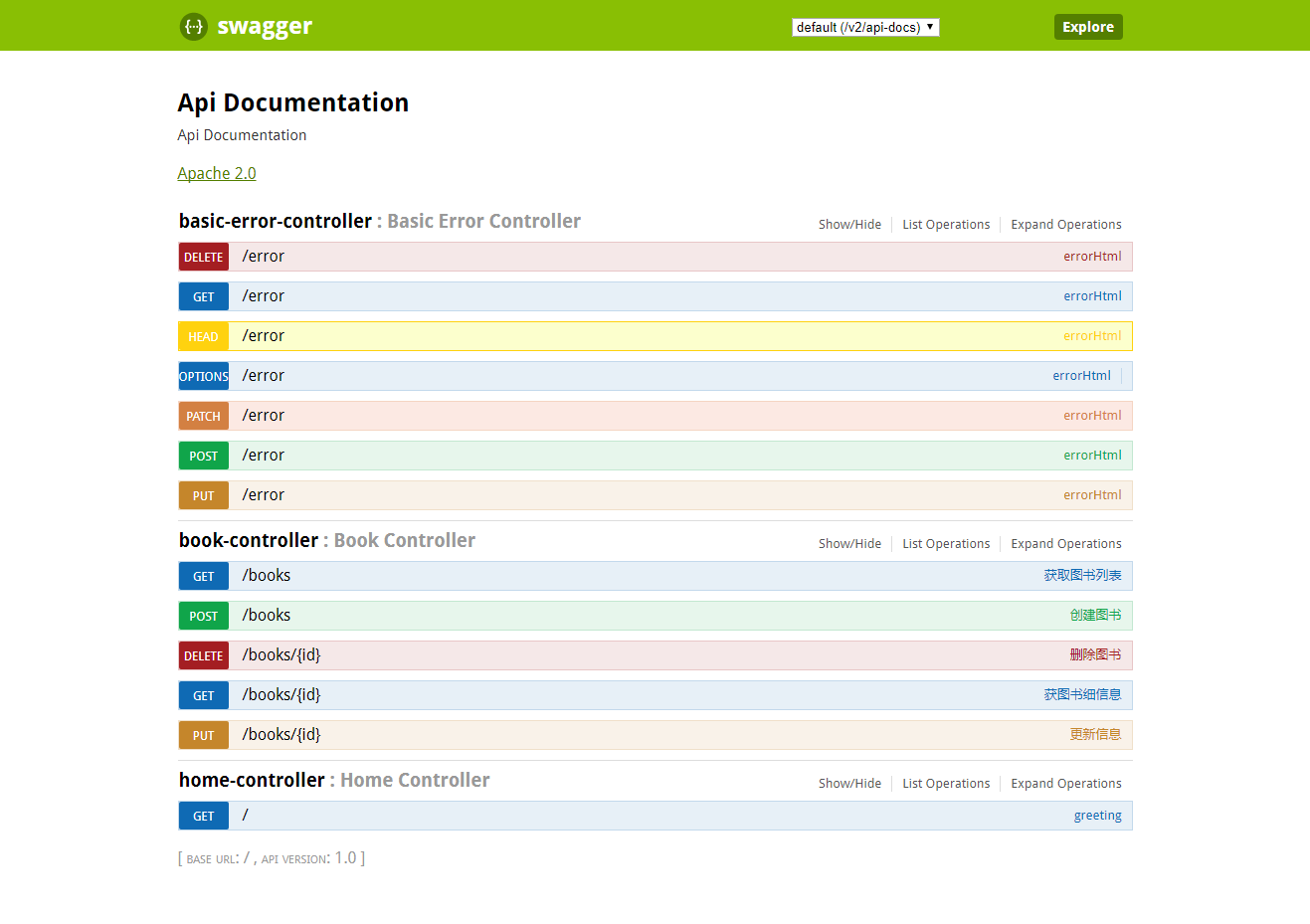
**package com.skytech.springbootrestdocs.entity**;  
  
**public class Book** {  
 **private long** *id*;  
 **private String** *name*;  
 **private double** *price*;  
  
 **public long** getId() {  
 **return** *id*;  
 }  
  
 **public void** setId(**long** id) {  
 **this**.*id* = id;  
 }  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public double** getPrice() {  
 **return** *price*;  
 }  
  
 **public void** setPrice(**double** price) {  
 **this**.*price* = price;  
 }  
}

## 现在通过一个栗子来说明：

**package com.skytech.springbootrestdocs.controller**;  
  
**import com.skytech.springbootrestdocs.entity.Book**;  
**import io.swagger.annotations.**ApiImplicitParam;  
**import io.swagger.annotations.**ApiImplicitParams;  
**import io.swagger.annotations.**ApiOperation;  
**import org.springframework.web.bind.annotation.**\*;  
**import springfox.documentation.annotations.**ApiIgnore;  
  
**import java.util.**\*;  
  
@RestController  
@RequestMapping(value = "/books")  
**public class BookController** {  
  
 **Map**<**Long**, **Book**> *books* = **Collections**.synchronizedMap(**new** HashMap<**Long**, **Book**>());  
  
  
 @ApiOperation(value = "获取图书列表", notes = "获取图书列表")  
 @RequestMapping(value = {""}, method = **RequestMethod**.***GET***)  
 **public List**<**Book**> getBook() {  
 **List**<**Book**> book = **new** ArrayList<>(*books*.values());  
 **return** book;  
 }  
  
 @ApiOperation(value = "创建图书", notes = "创建图书")  
 @ApiImplicitParam(name = "book", value = "图书详细实体", required = **true**, dataType = "Book")  
 @RequestMapping(value = "", method = **RequestMethod**.***POST***)  
 **public String** postBook(@RequestBody **Book** book) {  
 *books*.put(book.getId(), book);  
 **return** "success";  
 }  
  
 @ApiOperation(value = "获图书细信息", notes = "根据url的id来获取详细信息")  
 @ApiImplicitParam(name = "id", value = "ID", required = **true**, dataType = "Long", paramType = "path")  
 @RequestMapping(value = "/{id}", method = **RequestMethod**.***GET***)  
 **public Book** getBook(@PathVariable **Long** id) {  
 **return** *books*.get(id);  
 }  
  
 @ApiOperation(value = "更新信息", notes = "根据url的id来指定更新图书信息")  
 @ApiImplicitParams({  
 @ApiImplicitParam(name = "id", value = "图书ID", required = **true**, dataType = "Long", paramType = "path"),  
 @ApiImplicitParam(name = "book", value = "图书实体book", required = **true**, dataType = "Book")  
 })  
  
  
 @RequestMapping(value = "/{id}", method = **RequestMethod**.***PUT***)  
 **public String** putUser(@PathVariable **Long** id, @RequestBody **Book** book) {  
 **Book** book1 = *books*.get(id);  
 book1.setName(book.getName());  
 book1.setPrice(book.getPrice());  
 *books*.put(id, book1);  
 **return** "success";  
 }  
  
 @ApiOperation(value = "删除图书", notes = "根据url的id来指定删除图书")  
 @ApiImplicitParam(name = "id", value = "图书ID", required = **true**, dataType = "Long", paramType = "path")  
 @RequestMapping(value = "/{id}", method = **RequestMethod**.***DELETE***)  
 **public String** deleteUser(@PathVariable **Long** id) {  
 *books*.remove(id);  
 **return** "success";  
 }  
  
 @ApiIgnore//使用该注解忽略这个API  
 @RequestMapping(value = "/hi", method = **RequestMethod**.***GET***)  
 **public String** jsonTest() {  
 **return** " hi you!";  
 }  
  
}

通过相关注解，就可以让swagger2生成相应的文档。如果你不需要某接口生成文档，只需要在加@ApiIgnore注解即可。需要说明的是，如果请求参数在url上，@ApiImplicitParam 上加paramType = “path” 。

## 启动工程，访问：<http://localhost:8080/swagger-ui.html> ，就看到swagger-ui:



整个集成过程非常简单，但是我看了相关的资料，swagger没有做安全方面的防护，可能需要我们自己做相关的工作。

## 四、参考资料

[swagger.io](http://swagger.io/docs/)

[Spring Boot中使用Swagger2构建强大的RESTful API文档](http://blog.didispace.com/springbootswagger2/)

# 第十三篇：springboot集成spring cache

本文介绍如何在springboot中使用默认的spring cache

## 声明式缓存

Spring 定义 CacheManager 和 Cache 接口用来统一不同的缓存技术。例如 JCache、 EhCache、 Hazelcast、 Guava、 Redis 等。在使用 Spring 集成 Cache 的时候，我们需要注册实现的 CacheManager 的 Bean。

Spring Boot 为我们自动配置了 JcacheCacheConfiguration、 EhCacheCacheConfiguration、HazelcastCacheConfiguration、GuavaCacheConfiguration、RedisCacheConfiguration、SimpleCacheConfiguration 等。

## 默认使用 ConcurrenMapCacheManager

在我们不使用其他第三方缓存依赖的时候，springboot自动采用ConcurrenMapCacheManager作为缓存管理器。

## 环境依赖

在pom文件引入spring-boot-starter-cache环境依赖：

<!--spring cache-->  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-cache</**artifactId**>  
</**dependency**>

创建一个book数据访问层

## 先创建一个实体类

**package com.skytech.springbootrestdocs.entity**;  
  
**public class Book** {  
 **private String** *isbn*;  
 **private String** *title*;  
  
  
 **public String** getIsbn() {  
 **return** *isbn*;  
 }  
  
 **public void** setIsbn(**String** isbn) {  
 **this**.*isbn* = isbn;  
 }  
  
 **public String** getTitle() {  
 **return** *title*;  
 }  
  
 **public void** setTitle(**String** title) {  
 **this**.*title* = title;  
 }  
  
 **public Book**(**String** isbn, **String** title) {  
 **this**.*isbn* = isbn;  
 **this**.*title* = title;  
 }  
}

## 创建一个数据访问接口

**package com.skytech.springbootrestdocs.dao**;  
  
**import com.skytech.springbootrestdocs.entity.Book**;  
  
**public interface BookRepository** {  
 **Book** getByIsbn(**String** isbn);  
}

这个你可以写一个很复杂的数据查询操作，比如操作mysql、nosql等等。为了演示这个栗子，我只做了一下线程的延迟操作，当作是查询数据库的时间。

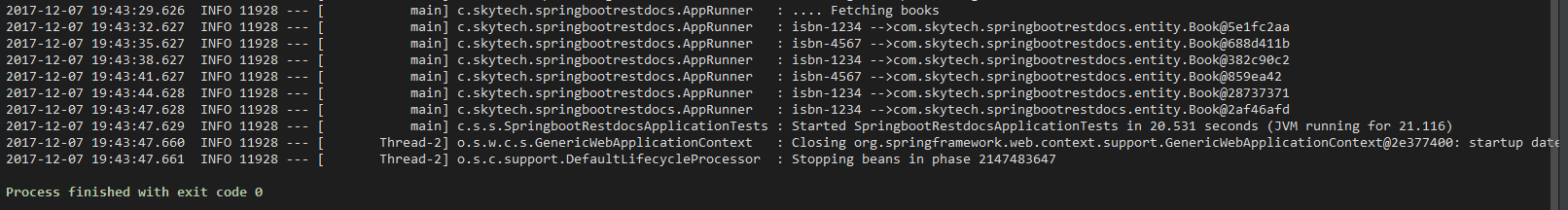
## 实现接口类：

**package com.skytech.springbootrestdocs.service**;  
  
**import com.skytech.springbootrestdocs.dao.BookRepository**;  
**import com.skytech.springbootrestdocs.entity.Book**;  
**import org.springframework.stereotype.**Component;  
  
@Component  
**public class SimpleBookRepository implements BookRepository** {  
  
  
 @Override  
// @Cacheable("books")  
 **public Book** getByIsbn(**String** isbn) {  
 simulateSlowService();  
 **return new** Book(isbn, "Some book");  
 }  
  
  
 **public void** simulateSlowService() {  
 **try** {  
 **long** time = **3000L**;  
 **Thread**.sleep(time);  
 } **catch** (**InterruptedException** e) {  
 **throw new** IllegalStateException(e);  
 }  
 }  
}

## 测试类

**package com.skytech.springbootrestdocs**;  
  
**import com.skytech.springbootrestdocs.dao.BookRepository**;  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.boot.CommandLineRunner**;  
**import org.springframework.stereotype.**Component;  
  
@Component  
**public class AppRunner implements CommandLineRunner** {  
  
  
 **private static final Logger *logger*** = **LoggerFactory**.getLogger(**AppRunner**.**class**);  
  
 **private final BookRepository** *bookRepository*;  
  
 **public AppRunner**(**BookRepository** bookRepository) {  
 **this**.*bookRepository* = bookRepository;  
 }  
  
 @Override  
 **public void** run(**String**... args) **throws Exception** {  
 ***logger***.info(".... Fetching books");  
 ***logger***.info("isbn-1234 -->" + *bookRepository*.getByIsbn("isbn-1234"));  
 ***logger***.info("isbn-4567 -->" + *bookRepository*.getByIsbn("isbn-4567"));  
 ***logger***.info("isbn-1234 -->" + *bookRepository*.getByIsbn("isbn-1234"));  
 ***logger***.info("isbn-4567 -->" + *bookRepository*.getByIsbn("isbn-4567"));  
 ***logger***.info("isbn-1234 -->" + *bookRepository*.getByIsbn("isbn-1234"));  
 ***logger***.info("isbn-1234 -->" + *bookRepository*.getByIsbn("isbn-1234"));  
 }  
}

启动程序，你会发现程序在控制台依次打印了：



你会发现程序依次3s打印一行日志。这时还没开启缓存技术。

## 开启缓存技术

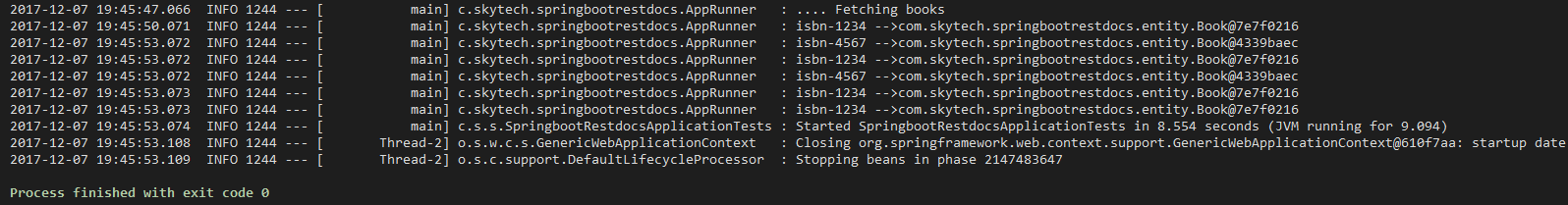
在程序的入口中加入@ EnableCaching开启缓存技术：

**package com.skytech.springbootrestdocs**;  
  
**import org.junit.**Test;  
**import org.junit.runner.**RunWith;  
**import org.springframework.boot.test.context.**SpringBootTest;  
**import org.springframework.cache.annotation.**EnableCaching;  
**import org.springframework.test.context.junit4.SpringRunner**;  
  
@RunWith(**SpringRunner**.**class**)  
@SpringBootTest  
@EnableCaching  
**public class SpringbootRestdocsApplicationTests** {  
  
 @Test  
 **public void** contextLoads() {  
 }  
  
}

在需要缓存的地方加入@Cacheable注解，比如在getByIsbn（）方法上加入@Cacheable(“books”)，这个方法就开启了缓存策略，当缓存有这个数据的时候，会直接返回数据，不会等待去查询数据库。

**package com.skytech.springbootrestdocs.service**;  
  
**import com.skytech.springbootrestdocs.dao.BookRepository**;  
**import com.skytech.springbootrestdocs.entity.Book**;  
**import org.springframework.cache.annotation.**Cacheable;  
**import org.springframework.stereotype.**Component;  
  
@Component  
**public class SimpleBookRepository implements BookRepository** {  
 @Override  
 @Cacheable("books")  
 **public Book** getByIsbn(**String** isbn) {  
 simulateSlowService();  
 **return new** Book(isbn, "Some book");  
 }  
  
  
 **public void** simulateSlowService() {  
 **try** {  
 **long** time = **3000L**;  
 **Thread**.sleep(time);  
 } **catch** (**InterruptedException** e) {  
 **throw new** IllegalStateException(e);  
 }  
 }  
}

这时再启动程序，你会发现程序打印：



应该看不出说明区别.如果你不实际操作的话.

只有打印前面2个数据，程序等了3s，之后的数据瞬间打印在控制台上了，这说明缓存起了作用。

源码下载：<https://github.com/forezp/SpringBootLearning>

# 第十四篇：在springboot中用redis实现消息队列

这篇文章主要讲述如何在springboot中用reids实现消息队列。

## 准备阶段

安装redis,可参考我的另一篇文章，[5分钟带你入门Redis](http://blog.csdn.net/forezp/article/details/61471712)。

java 1.8

maven 3.0

idea

## 环境依赖

创建一个新的springboot工程，在其pom文件,加入spring-boot-starter-data-redis依赖：

<!--redis-->  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
</**dependency**>

## 创建一个消息接收者

REcevier类，它是一个普通的类，需要注入到springboot中。

**package com.skytech.springbootredismessage.message**;  
  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.beans.factory.annotation.**Autowired;  
  
**import java.util.concurrent.CountDownLatch**;  
  
**public class Receiver** {  
 **private static final Logger *LOGGER*** = **LoggerFactory**.getLogger(**Receiver**.**class**);  
  
 **private CountDownLatch** *latch*;  
  
 @Autowired  
 **public Receiver**(**CountDownLatch** latch) {  
 **this**.*latch* = latch;  
 }  
  
 **public void** receiveMessage(**String** message) {  
 ***LOGGER***.info("Received <" + message + ">");  
 *latch*.countDown();  
 }  
}

## redis配置

#整合redis单机版,使用redis作为缓存  
# Redis数据库索引（默认为0）  
**spring.redis.database**=1  
# Redis服务器地址  
**spring.redis.host**=192.168.1.37  
# Redis服务器连接端口  
**spring.redis.port**=6379  
# Redis服务器连接密码（默认为空）  
**spring.redis.password**=root@redis

## 注入消息监听容器

在spring data redis中，利用redis发送一条消息和接受一条消息，需要三样东西：

一个连接工厂

一个消息监听容器

Redis template

上述1、3步已经完成，所以只需注入消息监听容器即可：

**package com.skytech.springbootredismessage**;  
  
**import com.skytech.springbootredismessage.message.Receiver**;  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.context.ApplicationContext**;  
**import org.springframework.context.annotation.**Bean;  
**import org.springframework.data.redis.connection.RedisConnectionFactory**;  
**import org.springframework.data.redis.core.StringRedisTemplate**;  
**import org.springframework.data.redis.listener.PatternTopic**;  
**import org.springframework.data.redis.listener.RedisMessageListenerContainer**;  
**import org.springframework.data.redis.listener.adapter.MessageListenerAdapter**;  
  
**import java.util.concurrent.CountDownLatch**;  
  
@SpringBootApplication  
**public class SpringbootRedisMessageApplication** {  
 **private static final Logger *LOGGER*** = **LoggerFactory**.getLogger(**SpringbootRedisMessageApplication**.**class**);  
  
 @Bean  
 **RedisMessageListenerContainer** container(**RedisConnectionFactory** connectionFactory,  
 **MessageListenerAdapter** listenerAdapter) {  
  
 **RedisMessageListenerContainer** container = **new** RedisMessageListenerContainer();  
 container.setConnectionFactory(connectionFactory);  
 container.addMessageListener(listenerAdapter, **new** PatternTopic("chat"));  
  
 **return** container;  
 }  
  
 @Bean  
 **MessageListenerAdapter** listenerAdapter(**Receiver** receiver) {  
 **return new** MessageListenerAdapter(receiver, "receiveMessage");  
 }  
  
 @Bean  
 **Receiver** receiver(**CountDownLatch** latch) {  
 **return new** Receiver(latch);  
 }  
  
 @Bean  
 **CountDownLatch** latch() {  
 **return new** CountDownLatch(**1**);  
 }  
  
 @Bean  
 **StringRedisTemplate** template(**RedisConnectionFactory** connectionFactory) {  
 **return new** StringRedisTemplate(connectionFactory);  
 }  
  
 **public static void** main(**String**[] args) **throws Exception** {  
 **ApplicationContext** ctx = **SpringApplication**.run(**SpringbootRedisMessageApplication**.**class**, args);  
  
 **StringRedisTemplate** template = ctx.getBean(**StringRedisTemplate**.**class**);  
 **CountDownLatch** latch = ctx.getBean(**CountDownLatch**.**class**);  
  
 ***LOGGER***.info("Sending message...");  
 template.convertAndSend("chat", "Hello from Redis!");  
  
 latch.await();  
  
 **System**.exit(**0**);  
 }  
}

## 测试

先用redisTemplate发送一条消息，接收者接收到后，打印出来。启动springboot程序，控制台打印：



# 第十五篇：Springboot整合RabbitMQ

这篇文章带你了解怎么整合RabbitMQ服务器，并且通过它怎么去发送和接收消息。我将构建一个springboot工程，通过RabbitTemplate去通过MessageListenerAdapter去订阅一个POJO类型的消息。

## 准备工作

15min

IDEA

maven 3.0

在开始构建项目之前，机器需要安装rabbitmq，你可以去官网下载，<http://www.rabbitmq.com/download.html>

首先在命令行输入：rabbitmq-service stop，

接着输入rabbitmq-service remove，

再接着输入rabbitmq-service install，

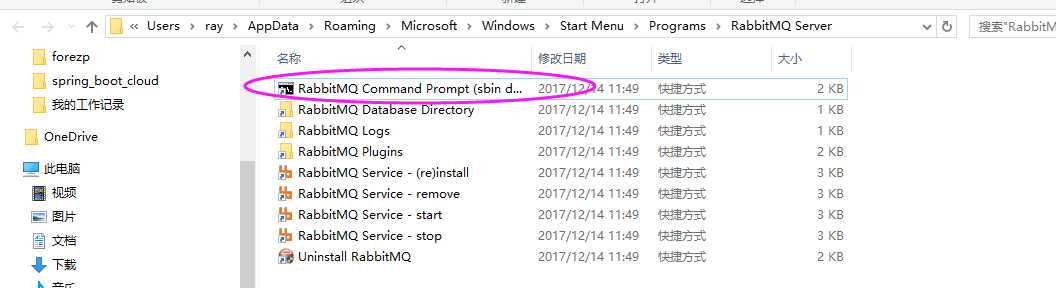
接着输入rabbitmq-service start，

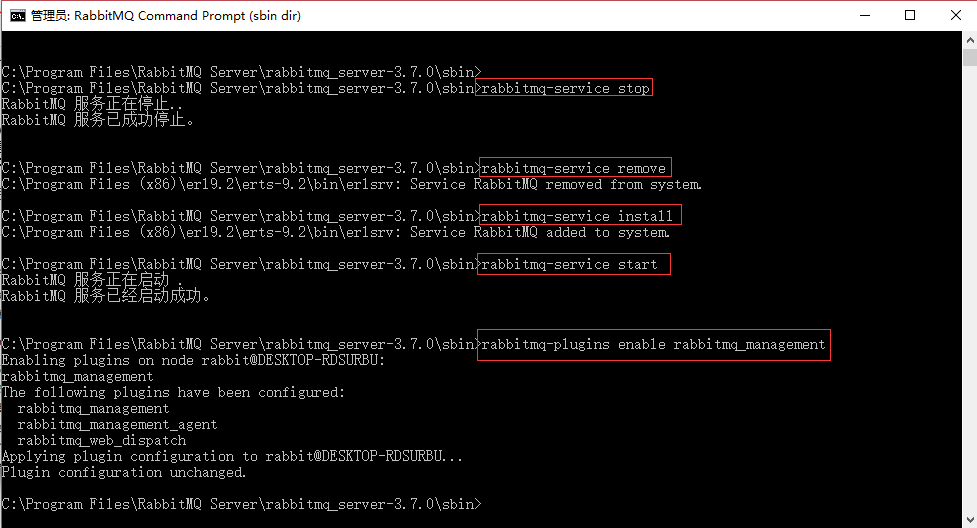
最后重新输入rabbitmq-plugins enable rabbitmq\_management试试，我是这样解决的；

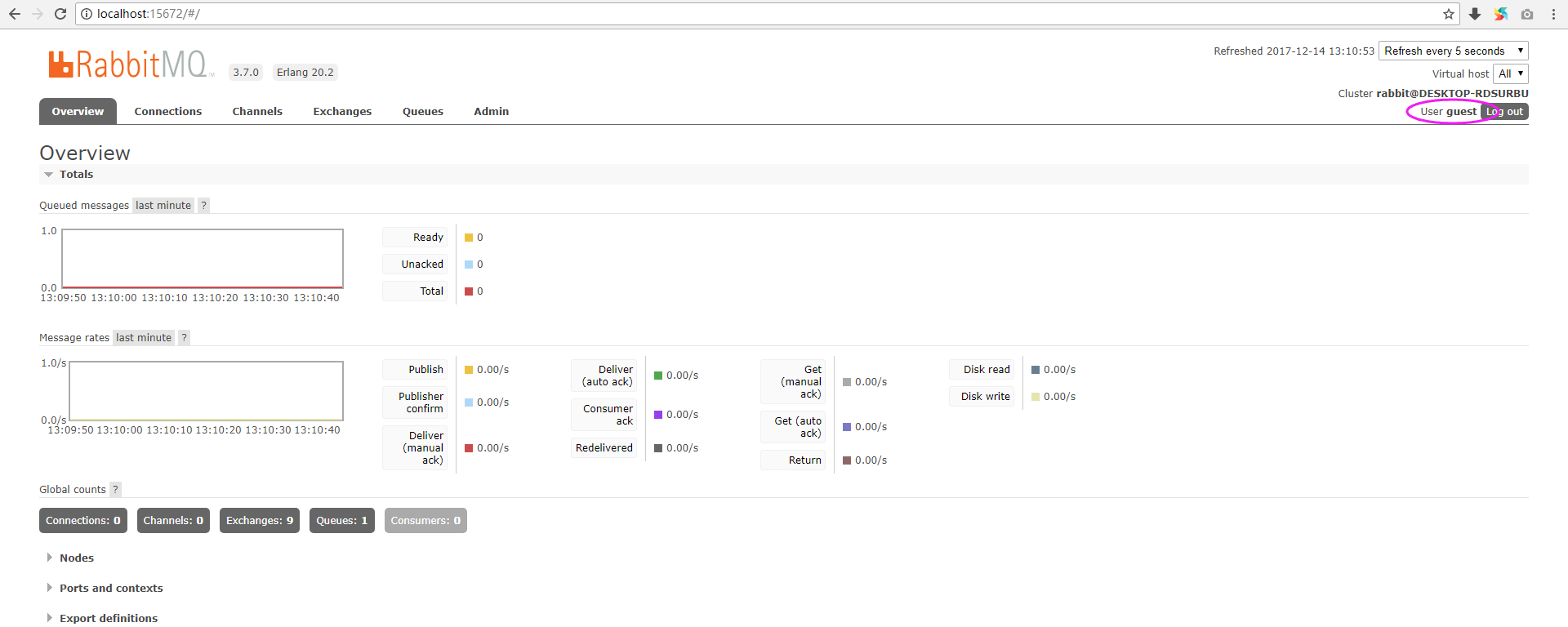
插件安装完之后，在浏览器输入http://localhost:15672进行验证，

你会看到下面界面，输入用户名：guest，密码：guest你就可以进入管理界面，当然用户名密码你都可以变的；

安装完成后开启服务器：







## 构建工程

构架一个SpringBoot工程，其pom文件依赖加上spring-boot-starter-amqp的起步依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
</**dependency**>

## 创建消息接收者

在任何的消息队列程序中，你需要创建一个消息接收者，用于响应发送的消息。

**package com.skytech.springbootrabbitmq.web**;  
  
  
**import org.springframework.stereotype.**Component;  
  
**import java.util.concurrent.CountDownLatch**;  
  
@Component  
**public class Receiver** {  
  
 **private CountDownLatch** *latch* = **new** CountDownLatch(**1**);  
  
 **public void** receiveMessage(**String** message) {  
 **System**.***out***.println("Received <" + message + ">");  
 *latch*.countDown();  
 }  
  
 **public CountDownLatch** getLatch() {  
 **return** *latch*;  
 }  
}

消息接收者是一个简单的POJO类，它定义了一个方法去接收消息，当你注册它去接收消息，你可以给它取任何的名字。其中，它有CountDownLatch这样的一个类，它是用于告诉发送者消息已经收到了，你不需要在应用程序中具体实现它，只需要latch.countDown()就行了。

## 创建消息监听，并发送一条消息

在spring程序中，RabbitTemplate提供了发送消息和接收消息的所有方法。你只需简单的配置下就行了：

* 需要一个消息监听容器
* 声明一个quene,一个exchange,并且绑定它们
* 一个组件去发送消息

代码清单如下：

**package com.skytech.springbootrabbitmq**;  
  
**import com.skytech.springbootrabbitmq.web.Receiver**;  
**import org.springframework.amqp.core.Binding**;  
**import org.springframework.amqp.core.BindingBuilder**;  
**import org.springframework.amqp.core.Queue**;  
**import org.springframework.amqp.core.TopicExchange**;  
**import org.springframework.amqp.rabbit.connection.ConnectionFactory**;  
**import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer**;  
**import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter**;  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.context.annotation.**Bean;  
  
@SpringBootApplication  
**public class SpringbootRabbitmqApplication** {  
  
  
 **public final static String *queueName*** = "spring-boot";  
  
 @Bean  
 **Queue** queue() {  
 **return new** Queue(***queueName***, **false**);  
 }  
  
 @Bean  
 **TopicExchange** exchange() {  
 **return new** TopicExchange("spring-boot-exchange");  
 }  
  
 @Bean  
 **Binding** binding(**Queue** queue, **TopicExchange** exchange) {  
 **return BindingBuilder**.bind(queue).to(exchange).with(***queueName***);  
 }  
  
 @Bean  
 **SimpleMessageListenerContainer** container(**ConnectionFactory** connectionFactory,  
 **MessageListenerAdapter** listenerAdapter) {  
 **SimpleMessageListenerContainer** container = **new** SimpleMessageListenerContainer();  
 container.setConnectionFactory(connectionFactory);  
 container.setQueueNames(***queueName***);  
 container.setMessageListener(listenerAdapter);  
 **return** container;  
 }  
  
 @Bean  
 **MessageListenerAdapter** listenerAdapter(**Receiver** receiver) {  
 **return new** MessageListenerAdapter(receiver, "receiveMessage");  
 }  
  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**SpringbootRabbitmqApplication**.**class**, args);  
 }  
}

创建一个测试方法：

**package com.skytech.springbootrabbitmq.web**;  
  
**import com.skytech.springbootrabbitmq.SpringbootRabbitmqApplication**;  
**import org.springframework.amqp.rabbit.core.RabbitTemplate**;  
**import org.springframework.boot.CommandLineRunner**;  
**import org.springframework.context.ConfigurableApplicationContext**;  
**import org.springframework.stereotype.**Component;  
  
**import java.util.concurrent.TimeUnit**;  
  
@Component  
**public class Runner implements CommandLineRunner** {  
  
 **private final RabbitTemplate** *rabbitTemplate*;  
 **private final Receiver** *receiver*;  
 **private final ConfigurableApplicationContext** *context*;  
  
 **public Runner**(**Receiver** receiver, **RabbitTemplate** rabbitTemplate,  
 **ConfigurableApplicationContext** context) {  
 **this**.*receiver* = receiver;  
 **this**.*rabbitTemplate* = rabbitTemplate;  
 **this**.*context* = context;  
 }  
  
 @Override  
 **public void** run(**String**... args) **throws Exception** {  
 **System**.***out***.println("Sending message...");  
 *rabbitTemplate*.convertAndSend(**SpringbootRabbitmqApplication**.***queueName***, "Hello from RabbitMQ!");  
 *receiver*.getLatch().await(**10000**, **TimeUnit**.***MILLISECONDS***);  
 *context*.close();  
 }  
  
}

启动程序，你会发现控制台打印：



## 总结

恭喜！你刚才已经学会了如何通过spring raabitmq去构建一个消息发送和订阅的程序。 这仅仅是一个好的开始，你可以通过spring-rabbitmq做更多的事，[点击这里](http://docs.spring.io/spring-amqp/reference/html/_introduction.html#quick-tour)。

## 参考资料

<https://spring.io/guides/gs/messaging-rabbitmq/>

# 第十六篇：用restTemplate消费服务

这篇文章主要介绍怎么用消费一个 Restful的web服务。我将用restTemplate去消费一个服务： <http://gturnquist-quoters.cfapps.io/api/random>.

## 构架工程

创建一个springboot工程，去消费RESTFUL的服务。这个服务是 [http:///gturnquist-quoters.cfapps.io/api/random](http://gturnquist-quoters.cfapps.io/api/random) ，它会随机返回Json字符串。   
在Spring项目中，它提供了一个非常简便的类，叫RestTemplate，它可以很简便的消费服务。

## 消费服务

通过RestTemplate消费服务，需要先context中注册一个RestTemplate bean。代码如下：

**package com.skytech.springbootredismessage**;  
  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.boot.CommandLineRunner**;  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.boot.web.client.RestTemplateBuilder**;  
**import org.springframework.context.annotation.**Bean;  
**import org.springframework.web.client.RestTemplate**;  
  
@SpringBootApplication  
**public class SpringbootResttemplateApplication** {  
  
  
 **private static final Logger *log*** = **LoggerFactory**.getLogger(**SpringbootResttemplateApplication**.**class**);  
  
 **public static void** main(**String** args[]) {  
 **SpringApplication**.run(**SpringbootResttemplateApplication**.**class**);  
 }  
  
 @Bean  
 **public RestTemplate** restTemplate(**RestTemplateBuilder** builder) {  
 **return** builder.build();  
 }  
  
 @Bean  
 **public CommandLineRunner** run(**RestTemplate** restTemplate) **throws Exception** {  
 **return** args -> {  
 **String** quote = **restTemplate**.getForObject(  
 "http://gturnquist-quoters.cfapps.io/api/random", **String**.**class**);  
 ***log***.info(quote.toString());  
 };  
 }  
}

运行程序，控制台打印：

2017-12-08 10:18:12.013 INFO 7568 --- [ main] c.s.s.SpringbootResttemplateApplication : {"type":"success","value":{"id":5,"quote":"Spring Boot solves this problem. It gets rid of XML and wires up common components for me, so I don't have to spend hours scratching my head just to figure out how it's all pieced together."}}

## 参考资料

<https://spring.io/guides/gs/consuming-rest/>

# 第十七篇：上传文件

这篇文章主要介绍，如何在springboot工程作为服务器，去接收通过http 上传的multi-file的文件。

## 构建工程

为例创建一个springmvc工程你需要spring-boot-starter-thymeleaf和 spring-boot-starter-web的起步依赖。为例能够上传文件在服务器，你需要在web.xml中加入标签做相关的配置，但在sringboot 工程中，它已经为你自动做了，所以不需要你做任何的配置。

<!--thymeleaf-->  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>  
</**dependency**>

## 创建文件上传controller

**package com.skytech.springbootuploadfile.controller**;  
  
**import com.skytech.springbootuploadfile.storage.StorageService**;  
**import com.skytech.springbootuploadfile.storage.StorageFileNotFoundException**;  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.core.io.Resource**;  
**import org.springframework.http.HttpHeaders**;  
**import org.springframework.http.ResponseEntity**;  
**import org.springframework.stereotype.**Controller;  
**import org.springframework.ui.Model**;  
**import org.springframework.web.bind.annotation.**\*;  
**import org.springframework.web.multipart.MultipartFile**;  
**import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder**;  
**import org.springframework.web.servlet.mvc.support.RedirectAttributes**;  
  
**import java.io.IOException**;  
**import java.util.stream.Collectors**;  
  
@Controller  
**public class FileUploadController** {  
  
  
 **private final StorageService** *storageService*;  
  
  
 @Autowired  
 **public FileUploadController**(**StorageService** storageService) {  
 **this**.*storageService* = storageService;  
 }  
  
 @RequestMapping("/")  
 **public String** listUploadFiles(**Model** model) **throws IOException** {  
 model.addAttribute("files", *storageService* .loadAll()  
 .map(path ->  
 **MvcUriComponentsBuilder** .fromMethodName(**FileUploadController**.**class**, "serveFile", path.getFileName().toString())  
 .build().toString()  
 ).collect(**Collectors**.toList()));  
 **return** "uploadForm";  
 }  
  
 @GetMapping("/files/{filename:.+}")  
 @ResponseBody  
 **public ResponseEntity**<**Resource**> serveFile(@PathVariable **String** filename) {  
  
 **Resource** file = *storageService*.loadAsResource(filename);  
 **return ResponseEntity** .ok()  
 .header(**HttpHeaders**.***CONTENT\_DISPOSITION***, "attachment; filename=**\"**" + file.getFilename() + "**\"**")  
 .body(file);  
 }  
  
  
 @PostMapping("/")  
 **public String** handleFileUpload(@RequestParam("file") **MultipartFile** file, **RedirectAttributes** redirectAttributes) {  
 *storageService*.store(file);  
 redirectAttributes.addAttribute("message", "you successfully uploaded" + file.getOriginalFilename() + "!");  
 **return** "redirect:/";  
 }  
  
  
 @ExceptionHandler(**StorageFileNotFoundException**.**class**)  
 **public ResponseEntity** handleStorageFileNotFound(**StorageFileNotFoundException** ex) {  
 **return ResponseEntity**.notFound().build();  
 }  
  
  
}

## SpringbootUploadFileApplication

**package com.skytech.springbootuploadfile**;  
  
**import com.skytech.springbootuploadfile.storage.StorageProperties**;  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**EnableAutoConfiguration;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.boot.context.properties.**EnableConfigurationProperties;  
**import org.springframework.boot.web.servlet.**ServletComponentScan;  
**import org.springframework.context.annotation.**ComponentScan;  
  
@SpringBootApplication  
@EnableConfigurationProperties(**StorageProperties**.**class**)  
@ServletComponentScan(basePackages = "com.skytech.springbootuploadfile.\*")  
@ComponentScan(basePackages = "com.skytech.springbootuploadfile.\*")  
@EnableAutoConfiguration  
**public class SpringbootUploadFileApplication** {  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**SpringbootUploadFileApplication**.**class**, args);  
 }  
}

## com.skytech.springbootuploadfile.storage.StorageService

**package com.skytech.springbootuploadfile.storage**;  
  
**import org.springframework.core.io.Resource**;  
**import org.springframework.web.multipart.MultipartFile**;  
  
**import java.nio.file.Path**;  
**import java.util.stream.Stream**;  
  
**public interface StorageService** {  
  
 **void** init();  
  
 **void** store(**MultipartFile** file);  
  
 **Stream**<**Path**> loadAll();  
  
 **Path** load(**String** filename);  
  
 **Resource** loadAsResource(**String** filename);  
  
 **void** deleteAll();  
  
}

## com.skytech.springbootuploadfile.storage.FileSystemStorageService

**package com.skytech.springbootuploadfile.storage**;  
  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.core.io.Resource**;  
**import org.springframework.core.io.UrlResource**;  
**import org.springframework.stereotype.**Service;  
**import org.springframework.util.FileSystemUtils**;  
**import org.springframework.web.multipart.MultipartFile**;  
  
**import java.io.IOException**;  
**import java.net.MalformedURLException**;  
**import java.nio.file.Files**;  
**import java.nio.file.Path**;  
**import java.nio.file.Paths**;  
**import java.util.stream.Stream**;  
  
@Service  
**public class FileSystemStorageService implements StorageService** {  
  
 **private final Path** *rootLocation*;  
  
 @Autowired  
 **public FileSystemStorageService**(**StorageProperties** properties) {  
 **this**.*rootLocation* = **Paths**.get(properties.getLocation());  
 }  
  
 @Override  
 **public void** store(**MultipartFile** file) {  
 **try** {  
 **if** (file.isEmpty()) {  
 **throw new** StorageException("Failed to store empty file " + file.getOriginalFilename());  
 }  
 **Files**.copy(file.getInputStream(), **this**.*rootLocation*.resolve(file.getOriginalFilename()));  
 } **catch** (**IOException** e) {  
 **throw new** StorageException("Failed to store file " + file.getOriginalFilename(), e);  
 }  
 }  
  
 @Override  
 **public Stream**<**Path**> loadAll() {  
 **try** {  
 **return Files**.walk(**this**.*rootLocation*, **1**)  
 .filter(path -> !path.equals(**this**.*rootLocation*))  
 .map(path -> **this**.*rootLocation*.relativize(path));  
 } **catch** (**IOException** e) {  
 **throw new** StorageException("Failed to read stored files", e);  
 }  
  
 }  
  
 @Override  
 **public Path** load(**String** filename) {  
 **return** *rootLocation*.resolve(filename);  
 }  
  
 @Override  
 **public Resource** loadAsResource(**String** filename) {  
 **try** {  
 **Path** file = load(filename);  
 **Resource** resource = **new** UrlResource(file.toUri());  
 **if** (resource.exists() || resource.isReadable()) {  
 **return** resource;  
 } **else** {  
 **throw new** StorageFileNotFoundException("Could not read file: " + filename);  
  
 }  
 } **catch** (**MalformedURLException** e) {  
 **throw new** StorageFileNotFoundException("Could not read file: " + filename, e);  
 }  
 }  
  
 @Override  
 **public void** deleteAll() {  
 **FileSystemUtils**.deleteRecursively(*rootLocation*.toFile());  
 }  
  
 @Override  
 **public void** init() {  
 **try** {  
 **Files**.createDirectory(*rootLocation*);  
 } **catch** (**IOException** e) {  
 **throw new** StorageException("Could not initialize storage", e);  
 }  
 }  
}

## com.skytech.springbootuploadfile.storage.StorageException

**package com.skytech.springbootuploadfile.storage**;  
  
**public class StorageException extends RuntimeException** {  
  
  
 **public StorageException**(**String** message) {  
 **super**(message);  
 }  
  
 **public StorageException**(**String** message, **Throwable** cause) {  
 **super**(message, cause);  
 }  
}

## com.skytech.springbootuploadfile.storage.StorageFileNotFoundException

**package com.skytech.springbootuploadfile.storage**;  
  
**public class StorageFileNotFoundException extends StorageException** {  
 **public StorageFileNotFoundException**(**String** message) {  
 **super**(message);  
 }  
  
  
 **public StorageFileNotFoundException**(**String** message, **Throwable** cause) {  
 **super**(message, cause);  
 }  
}

## com.skytech.springbootuploadfile.storage.StorageProperties

**package com.skytech.springbootuploadfile.storage**;  
  
**import org.springframework.boot.context.properties.**ConfigurationProperties;  
  
@ConfigurationProperties("storage")  
**public class StorageProperties** {  
  
 /\*\*  
 \* Folder location for storing files  
 \*/  
 **private String** *location* = "src/upload-dir";  
  
 **public String** getLocation() {  
 **return** *location*;  
 }  
  
 **public void** setLocation(**String** location) {  
 **this**.*location* = location;  
 }  
  
}

这个类通过@Controller注解，表明自己上一个Spring mvc的c。每个方法通过   
@GetMapping 或者@PostMapping注解表明自己的 http方法。

* GET / 获取已经上传的文件列表
* GET /files/{filename} 下载已经存在于服务器的文件
* POST / 上传文件给服务器

## 创建一个简单的 html模板

为了展示上传文件的过程，我们做一个界面：   
在src/main/resources/templates/uploadForm.html

<**html xmlns:***th***=**"http://www.thymeleaf.org">  
<**body**>  
  
<**div** *th***:if=**"${message}">  
 <**h2** *th***:text=**"${message}"/>  
</**div**>  
  
<**div**>  
 <**form method=**"POST" **enctype=**"multipart/form-data" **action=**"/">  
 <**table**>  
 <**tr**>  
 <**td**>File to upload:</**td**>  
 <**td**><**input type=**"file" **name=**"file"/></**td**>  
 </**tr**>  
 <**tr**>  
 <**td**></**td**>  
 <**td**><**input type=**"submit" **value=**"Upload"/></**td**>  
 </**tr**>  
 </**table**>  
 </**form**>  
</**div**>  
  
<**div**>  
 <**ul**>  
 <**li** *th***:each=**"file : ${files}">  
 <**a** *th***:href=**"${file}" *th***:text=**"${file}"/>  
 </**li**>  
 </**ul**>  
</**div**>  
  
</**body**>  
</**html**>

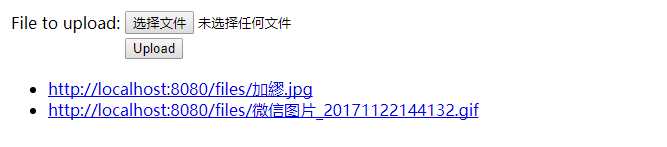
## 上传文件大小限制

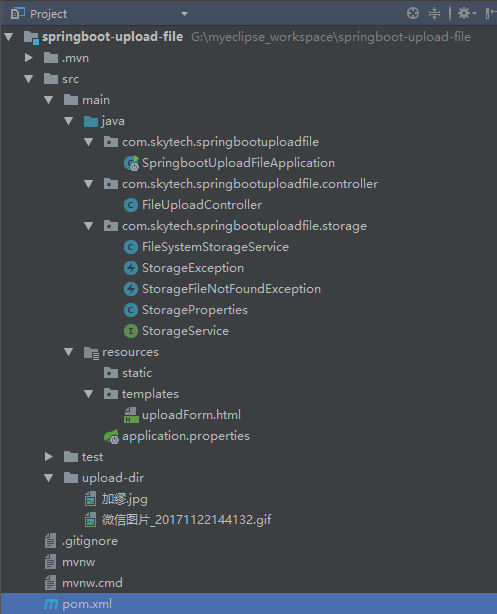
如果需要限制上传文件的大小也很简单，只需要在springboot 工程的src/main/resources/application.properties 加入以下：

**spring.http.multipart.max-file-size**=128MB  
**spring.http.multipart.max-request-size**=128MB

## 测试

测试情况如图：





# 第十八篇： 定时任务（Scheduling Tasks）

这篇文章将介绍怎么通过spring去做调度任务。

## 构建工程

创建一个Springboot工程，在它的程序入口加上@EnableScheduling,开启调度任务。

**package com.skytech.springbootschedulingtasks**;  
  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.scheduling.annotation.**EnableScheduling;  
  
@SpringBootApplication  
@EnableScheduling  
**public class SpringbootSchedulingTasksApplication** {  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**SpringbootSchedulingTasksApplication**.**class**, args);  
 }  
}

创建定时任务

创建一个定时任务，每过5s在控制台打印当前时间。

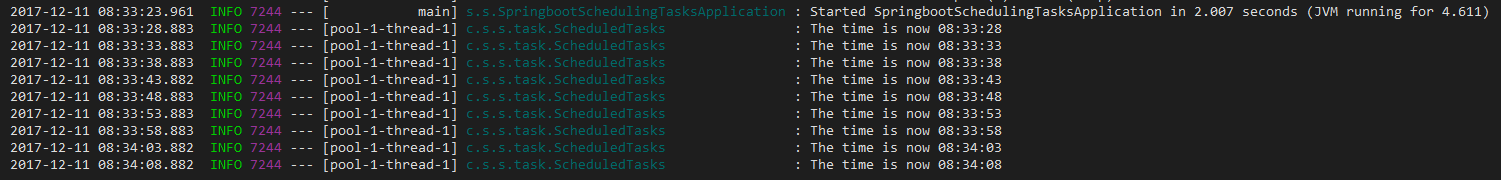
**package com.skytech.springbootschedulingtasks.task**;  
  
  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.scheduling.annotation.**Scheduled;  
**import org.springframework.stereotype.**Component;  
  
**import java.text.SimpleDateFormat**;  
**import java.util.Date**;  
  
@Component  
**public class ScheduledTasks** {  
  
 **private static final Logger *log*** = **LoggerFactory**.getLogger(**ScheduledTasks**.**class**);  
  
 **private static final SimpleDateFormat *dateFormat*** = **new** SimpleDateFormat("HH:mm:ss");  
  
 @Scheduled(fixedRate = **5000**)  
 **public void** reportCurrentTime() {  
 ***log***.info("The time is now {}", ***dateFormat***.format(**new** Date()));  
 }  
  
}

通过在方法上加@Scheduled注解，表明该方法是一个调度任务。

* @Scheduled(fixedRate = 5000) ：上一次开始执行时间点之后5秒再执行
* @Scheduled(fixedDelay = 5000) ：上一次执行完毕时间点之后5秒再执行
* @Scheduled(initialDelay=1000, fixedRate=5000) ：第一次延迟1秒后执行，之后按fixedRate的规则每5秒执行一次
* @Scheduled(cron=” /5 “) ：通过cron表达式定义规则，什么是cro表达式，自行搜索引擎。

## 测试

启动springboot工程，控制台没过5s就打印出了当前的时间。



## 总结

在springboot创建定时任务比较简单，只需2步：

1.在程序的入口加上@EnableScheduling注解。

2.在定时方法上加@Scheduled注解。

## 参考资料

<https://spring.io/guides/gs/scheduling-tasks/>

# 第十九篇： 验证表单信息

这篇文篇主要简述如何在springboot中验证表单信息。在springmvc工程中，需要检查表单信息，表单信息验证主要通过注解的形式。

## 构建工程

创建一个springboot工程，由于用到了 web 、thymeleaf、validator、el，引入相应的起步依赖和依赖，代码清单如下：

<!--web 、thymeleaf、validator、el-->  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-validator</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.tomcat.embed</**groupId**>  
 <**artifactId**>tomcat-embed-el</**artifactId**>  
</**dependency**>

创建一个PresonForm的Object类

**package com.skytech.springbootschedulingtasks.entity**;  
  
**import javax.validation.constraints.**Min;  
**import javax.validation.constraints.**NotNull;  
**import javax.validation.constraints.**Size;  
  
**public class PersonForm** {  
  
 @NotNull  
 @Size(min = **2**, max = **30**)  
 **private String** *name*;  
  
  
 @NotNull  
 @Min(**18**)  
 **private Integer** *age*;  
  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public Integer** getAge() {  
 **return** *age*;  
 }  
  
 **public void** setAge(**Integer** age) {  
 **this**.*age* = age;  
 }  
  
  
 @Override  
 **public String** toString() {  
 **return** "PersonForm{" +  
 "name='" + *name* + '**\'**' +  
 ", age=" + *age* +  
 '}';  
 }  
}

这个实体类，在2个属性:name,age.它们各自有验证的注解：

* @Size(min=2, max=30) name的长度为2-30个字符
* @NotNull 不为空
* @Min(18)age不能小于18

## 创建 web Controller

**package com.skytech.springbootschedulingtasks.controller**;  
  
**import com.skytech.springbootschedulingtasks.entity.PersonForm**;  
**import org.springframework.stereotype.**Controller;  
**import org.springframework.validation.BindingResult**;  
**import org.springframework.web.bind.annotation.**GetMapping;  
**import org.springframework.web.bind.annotation.**PostMapping;  
**import org.springframework.web.servlet.config.annotation.ViewControllerRegistry**;  
**import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter**;  
  
**import javax.validation.**Valid;  
  
@Controller  
**public class WebController extends WebMvcConfigurerAdapter** {  
  
  
 **public void** addViewController(**ViewControllerRegistry** registry) {  
 registry.addViewController("/results").setViewName("results");  
 }  
  
  
 @GetMapping("/")  
 **public String** showForm(**PersonForm** personForm) {  
 **return** "form";  
 }  
  
 @PostMapping("/")  
 **public String** checkPersonInfo(@Valid **PersonForm** personForm, **BindingResult** bindingResult) {  
 **if** (bindingResult.hasErrors()) {  
 **return** "form";  
 }  
 **return** "redirect:/results";  
 }  
}

## 创建form表单

src/main/resources/templates/form.html:

<**html**>  
<**body**>  
<**form action=**"#" *th***:action=**"@{/}" *th***:object=**"${personForm}" **method=**"post">  
 <**table**>  
 <**tr**>  
 <**td**>Name:</**td**>  
 <**td**><**input type=**"text" *th***:field=**"\*{name}"/></**td**>  
 <**td** *th***:if=**"${#fields.hasErrors('name')}" *th***:errors=**"\*{name}">Name Error</**td**>  
 </**tr**>  
 <**tr**>  
 <**td**>Age:</**td**>  
 <**td**><**input type=**"text" *th***:field=**"\*{age}"/></**td**>  
 <**td** *th***:if=**"${#fields.hasErrors('age')}" *th***:errors=**"\*{age}">Age Error</**td**>  
 </**tr**>  
 <**tr**>  
 <**td**>  
 <**button type=**"submit">Submit</**button**>  
 </**td**>  
 </**tr**>  
 </**table**>  
</**form**>  
</**body**>  
</**html**>

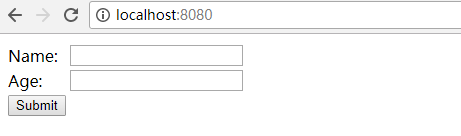
## 注册成功的页面

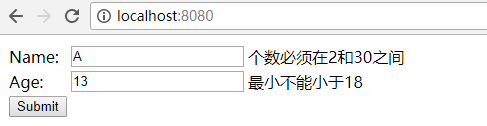
src/main/resources/templates/results.html:

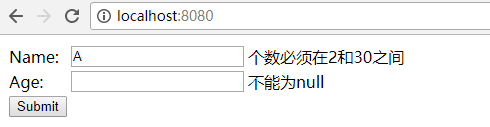
<html>  
<**body**>  
Congratulations! You are old enough to sign up for this site.  
</**body**>  
</**html**>

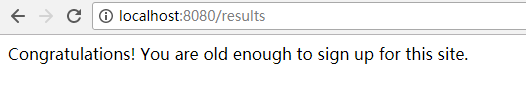
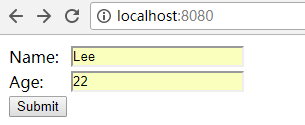
## 演示

启动工程，访问<http://localhost:8080/>：





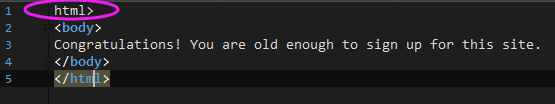




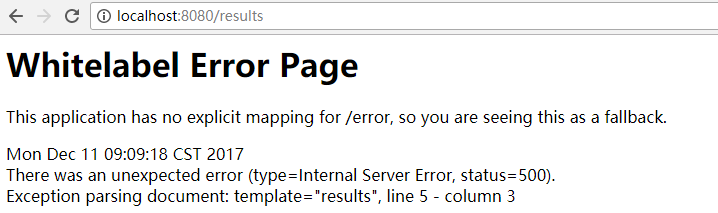
## 参考资料

<https://spring.io/guides/gs/validating-form-input/>

遇到的问题:



.html少写一个标签,会出现一个错误.



在我没有补齐这个html标签的情况下,我通过其他办法,可以修复这个bug.

1. 添加依赖文件

<!-- <dependency>  
 <groupId>net.sourceforge.nekohtml</groupId>  
 <artifactId>nekohtml</artifactId>  
 <version>1.9.22</version>  
 </dependency>-->

2. 更改application.properties属性

#spring.thymeleaf.mode=LEGACYHTML5

# 第二十篇： 处理表单提交

这篇文件主要介绍通过springboot 去创建和提交一个表单。

## 创建工程

涉及了 web，加上spring-boot-starter-web和spring-boot-starter-thymeleaf的起步依赖。

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>  
</**dependency**>

## 创建实体

代码清单如下：

**package com.skytech.springbootschedulingtasks.entity**;  
  
**public class Greeting** {  
  
 **private long** *id*;  
 **private String** *content*;  
  
 **public long** getId() {  
 **return** *id*;  
 }  
  
 **public void** setId(**long** id) {  
 **this**.*id* = id;  
 }  
  
 **public String** getContent() {  
 **return** *content*;  
 }  
  
 **public void** setContent(**String** content) {  
 **this**.*content* = content;  
 }  
}

## 创建Controller

**package com.skytech.springbootschedulingtasks.controller**;  
  
**import com.skytech.springbootschedulingtasks.entity.Greeting**;  
**import org.springframework.stereotype.**Controller;  
**import org.springframework.ui.Model**;  
**import org.springframework.web.bind.annotation.**GetMapping;  
**import org.springframework.web.bind.annotation.**ModelAttribute;  
**import org.springframework.web.bind.annotation.**PostMapping;  
  
@Controller  
**public class GreetingController** {  
  
  
 @GetMapping("/greeting")  
 **public String** greetingForm(**Model** model) {  
 model.addAttribute("greeting", **new** Greeting());  
 **return** "greeting";  
 }  
  
 @PostMapping("/greeting")  
 **public String** greetingSubmit(@ModelAttribute **Greeting** greeting) {  
 **return** "result";  
 }  
  
}

## 页面展示层

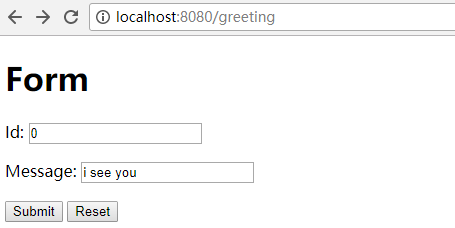
src/main/resources/templates/greeting.html

<!DOCTYPE **HTML**>  
<**html xmlns:***th***=**"http://www.thymeleaf.org">  
<**head**>  
 <**title**>Getting Started: Handling Form Submission</**title**>  
 <**meta http-equiv=**"Content-Type" **content=**"text/html; charset=UTF-8"/>  
</**head**>  
<**body**>  
<**h1**>Form</**h1**>  
<**form action=**"#" *th***:action=**"@{/greeting}" *th***:object=**"${greeting}" **method=**"post">  
 <**p**>Id: <**input type=**"text" *th***:field=**"\*{id}"/></**p**>  
 <**p**>Message: <**input type=**"text" *th***:field=**"\*{content}"/></**p**>  
 <**p**><**input type=**"submit" **value=**"Submit"/> <**input type=**"reset" **value=**"Reset"/></**p**>  
</**form**>  
</**body**>  
</**html**>

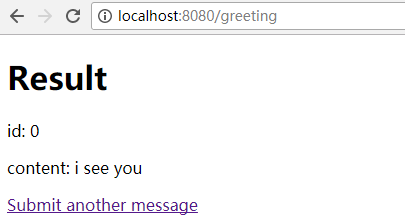
src/main/resources/templates/result.html

<!DOCTYPE **HTML**>  
<**html xmlns:***th***=**"http://www.thymeleaf.org">  
<**head**>  
 <**title**>Getting Started: Handling Form Submission</**title**>  
 <**meta http-equiv=**"Content-Type" **content=**"text/html; charset=UTF-8"/>  
</**head**>  
<**body**>  
<**h1**>Result</**h1**>  
<**p** *th***:text=**"'id: ' + ${greeting.id}"/>  
<**p** *th***:text=**"'content: ' + ${greeting.content}"/>  
<**a href=**"/greeting">Submit another message</**a**>  
</**body**>  
</**html**>

启动工程，访问ttp://localhost:8080/greeting:



点击submit:



## 参考资料

<https://spring.io/guides/gs/handling-form-submission/>

# 第二十一篇： springboot集成JMS

springboot对JMS提供了很好的支持，对其做了起步依赖。

## 构架工程

创建一个springboot工程，在其pom文件加入：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-mail</**artifactId**>  
</**dependency**>

## 添加配置

**spring.mail.host**=smtp.163.com  
**spring.mail.username**=zhuanshen27@163.com  
**spring.mail.password**=  
**spring.mail.port**=25  
**spring.mail.protocol**=smtp  
**spring.mail.default-encoding**=UTF-8

在password 中填写自己的邮箱密码。

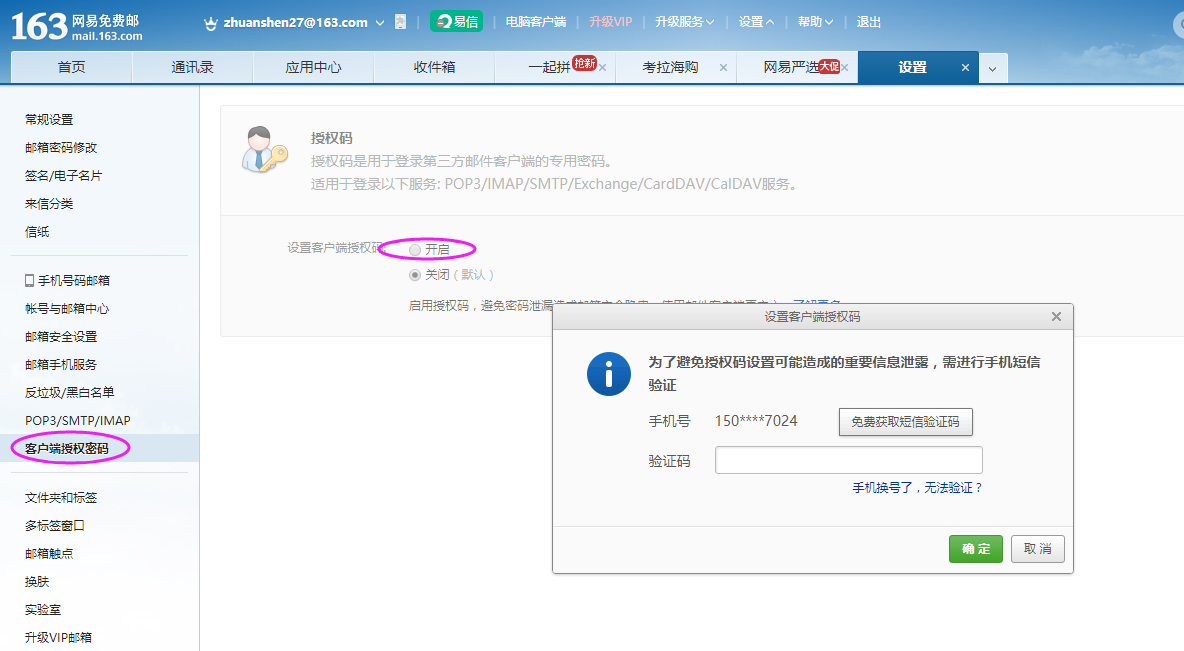
## 测试发邮件

测试代码清单如下：

**package com.skytech.springbootjms**;  
  
  
**import org.junit.**Test;  
**import org.junit.runner.**RunWith;  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.beans.factory.annotation.**Value;  
**import org.springframework.boot.test.context.**SpringBootTest;  
**import org.springframework.core.io.FileSystemResource**;  
**import org.springframework.mail.SimpleMailMessage**;  
**import org.springframework.mail.javamail.JavaMailSenderImpl**;  
**import org.springframework.mail.javamail.MimeMessageHelper**;  
**import org.springframework.test.context.junit4.SpringRunner**;  
  
**import javax.mail.internet.MimeMessage**;  
**import java.io.File**;  
  
@RunWith(**SpringRunner**.**class**)  
@SpringBootTest  
**public class SpringbootJmsApplicationTests** {  
  
 // @Test  
 **public void** contextLoads() {  
 }  
  
  
 @Autowired  
 **private JavaMailSenderImpl** *mailSender*;  
  
 //读取配置文件中的参数  
 @Value("${spring.mail.username}")  
 **private String** *SenderName*;  
  
 /\*\*  
 \* 发送包含简单文本的邮件  
 \*/  
 @Test  
 **public void** sendTxtMail() **throws Exception** {  
 **SimpleMailMessage** simpleMailMessage = **new** SimpleMailMessage();  
 // 设置收件人，寄件人  
 simpleMailMessage.setTo(**new** String[]{*SenderName*});  
 simpleMailMessage.setFrom(*SenderName*);  
 simpleMailMessage.setSubject(*SenderName*);  
 simpleMailMessage.setText(*SenderName*);  
 // 发送邮件  
 *mailSender*.send(simpleMailMessage);  
  
 **System**.***out***.println("邮件已发送");  
 }  
  
 /\*\*  
 \* 发送包含HTML文本的邮件  
 \*  
 \* @throws **Exception** \*/  
 @Test  
 **public void** sendHtmlMail() **throws Exception** {  
 **MimeMessage** mimeMessage = *mailSender*.createMimeMessage();  
 **MimeMessageHelper** mimeMessageHelper = **new** MimeMessageHelper(mimeMessage);  
 mimeMessageHelper.setTo(*SenderName*);  
 mimeMessageHelper.setFrom(*SenderName*);  
 mimeMessageHelper.setSubject("Spring Boot Mail 邮件测试【HTML】");  
  
 **StringBuilder** sb = **new** StringBuilder();  
 sb.append("<html><head></head>");  
 sb.append("<body><h1>spring 邮件测试</h1><p>hello!this is spring mail test。</p></body>");  
 sb.append("</html>");  
  
 // 启用html  
 mimeMessageHelper.setText(sb.toString(), **true**);  
 // 发送邮件  
 *mailSender*.send(mimeMessage);  
  
 **System**.***out***.println("邮件已发送");  
  
 }  
  
 /\*\*  
 \* 发送包含内嵌图片的邮件  
 \*  
 \* @throws **Exception** \*/  
 @Test  
 **public void** sendAttachedImageMail() **throws Exception** {  
 **MimeMessage** mimeMessage = *mailSender*.createMimeMessage();  
 // multipart模式  
 **MimeMessageHelper** mimeMessageHelper = **new** MimeMessageHelper(mimeMessage, **true**);  
 mimeMessageHelper.setTo(*SenderName*);  
 mimeMessageHelper.setFrom(*SenderName*);  
 mimeMessageHelper.setSubject("Spring Boot Mail 邮件测试【图片】");  
  
 **StringBuilder** sb = **new** StringBuilder();  
 sb.append("<html><head></head>");  
 sb.append("<body><h1>spring 邮件测试</h1><p>hello!this is spring mail test。</p>");  
 // cid为固定写法，imageId指定一个标识  
 sb.append("<img src=**\"**cid:imageId**\"**/></body>");  
 sb.append("</html>");  
  
 // 启用html  
 mimeMessageHelper.setText(sb.toString(), **true**);  
  
 // 设置imageId  
 **FileSystemResource** img = **new** FileSystemResource(**new** File("E:/1.jpg"));  
 mimeMessageHelper.addInline("imageId", img);  
  
 // 发送邮件  
 *mailSender*.send(mimeMessage);  
  
 **System**.***out***.println("邮件已发送");  
 }  
  
 /\*\*  
 \* 发送包含附件的邮件  
 \*  
 \* @throws **Exception** \*/  
 @Test  
 **public void** sendAttendedFileMail() **throws Exception** {  
 **MimeMessage** mimeMessage = *mailSender*.createMimeMessage();  
 // multipart模式  
 **MimeMessageHelper** mimeMessageHelper = **new** MimeMessageHelper(mimeMessage, **true**, "utf-8");  
 mimeMessageHelper.setTo(*SenderName*);  
 mimeMessageHelper.setFrom(*SenderName*);  
 mimeMessageHelper.setSubject("Spring Boot Mail 邮件测试【附件】");  
  
 **StringBuilder** sb = **new** StringBuilder();  
 sb.append("<html><head></head>");  
 sb.append("<body><h1>spring 邮件测试</h1><p>hello!this is spring mail test。</p></body>");  
 sb.append("</html>");  
  
 // 启用html  
 mimeMessageHelper.setText(sb.toString(), **true**);  
 // 设置附件  
 **FileSystemResource** img = **new** FileSystemResource(**new** File("E:/1.jpg"));  
 mimeMessageHelper.addAttachment("image.jpg", img);  
  
 // 发送邮件  
 *mailSender*.send(mimeMessage);  
  
 **System**.***out***.println("邮件已发送");  
 }  
}

## 出现的问题:



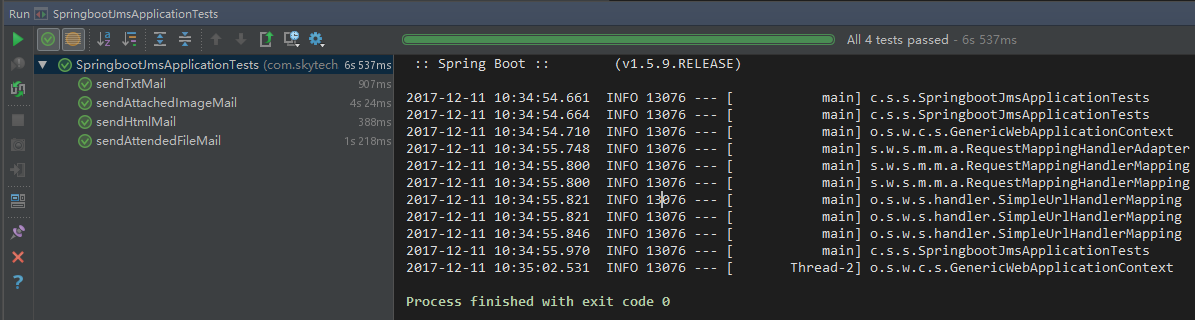


**设置完毕后，在代码中用使用客户端授权密码代替原始的邮箱密码，这样就可以正确的发送邮件了。**

发送的内容不可以出现敏感内容,不然会报554错误



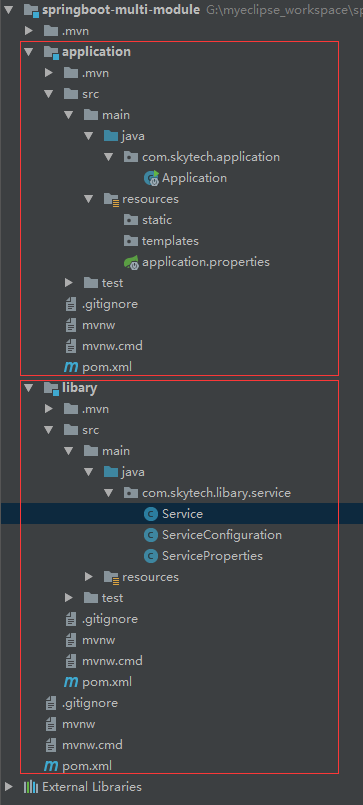
## 最后结果:





# 第二十二篇： 创建含有多module的springboot工程

这篇文章主要介绍如何在springboot中如何创建含有多个module的工程，栗子中含有两个 module，一个作为libarary. 工程，另外一个是主工程，调用libary .其中libary jar有一个服务，main工程调用这个服务。



## 创建根工程

创建一个maven 工程,其pom文件为：

*<?***xml version**="1.0" **encoding**="UTF-8"*?>*<**project xmlns**="http://maven.apache.org/POM/4.0.0" **xmlns:***xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 *xsi***:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.skytech</**groupId**>  
 <**artifactId**>springboot-multi-module</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 <**packaging**>pom</**packaging**>  
 <**name**>springboot-multi-module</**name**>  
 <**description**>Demo project for Spring Boot</**description**>  
  
  
</**project**>

需要注意的是packaging标签为pom 属性。

## 创建libary工程

libary工程为maven工程，其pom文件的packaging标签为jar 属性。创建一个service组件,它读取配置文件的 service.message属性。

**package com.skytech.libary.service**;  
  
**import org.springframework.boot.context.properties.**ConfigurationProperties;  
  
@ConfigurationProperties("service")  
**public class ServiceProperties** {  
  
 **private String** *message*;  
  
 **public String** getMessage() {  
 **return** *message*;  
 }  
  
 **public void** setMessage(**String** message) {  
 **this**.*message* = message;  
 }  
}

提供一个对外暴露的方法：

**package com.skytech.libary.service**;  
  
**import org.springframework.boot.context.properties.**EnableConfigurationProperties;  
**import org.springframework.context.annotation.**Bean;  
**import org.springframework.context.annotation.**Configuration;  
  
@Configuration  
@EnableConfigurationProperties(**ServiceProperties**.**class**)  
**public class ServiceConfiguration** {  
 @Bean  
 **public Service** service(**ServiceProperties** properties) {  
 **return new** Service(properties.getMessage());  
 }  
}

service

**package com.skytech.libary.service**;  
  
**import org.springframework.stereotype.**Component;  
  
@Component  
**public class Service** {  
  
 **private final String** *message*;  
  
 **public Service**(**String** message) {  
 **this**.*message* = message;  
 }  
  
 **public String** message() {  
 **return this**.*message*;  
 }  
}

## 创建一个springbot modeul工程

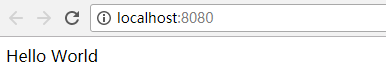
引入相应的依赖,创建一个web服务：

**package com.skytech.application**;  
  
**import com.skytech.libary.service.Service**;  
**import com.skytech.libary.service.ServiceConfiguration**;  
**import org.springframework.beans.factory.annotation.**Autowired;  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.context.annotation.**Import;  
**import org.springframework.web.bind.annotation.**GetMapping;  
**import org.springframework.web.bind.annotation.**RestController;  
  
@SpringBootApplication  
@Import(**ServiceConfiguration**.**class**)  
@RestController  
**public class Application** {  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**Application**.**class**, args);  
 }  
  
 **private final Service** *service*;  
  
 @Autowired  
 **public Application**(**Service** service) {  
 **this**.*service* = service;  
 }  
  
 @GetMapping("/")  
 **public String** home() {  
 **return** *service*.message();  
 }  
}

在配置文件application.properties中加入：

**service.message**=Hello World

打开浏览器访问：<http://localhost:8080/>;浏览器显示：



说明确实引用了libary中的方法。

## 参考资料

<https://spring.io/guides/gs/multi-module/>

# 第二十三篇： 异步方法

这篇文章主要介绍在springboot 使用异步方法，去请求github api.

## 创建工程

在pom文件引入相关依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter</**artifactId**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-web</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.fasterxml.jackson.core</**groupId**>  
 <**artifactId**>jackson-databind</**artifactId**>  
</**dependency**>

## 创建一个接收数据的实体：

**package com.skytech.springbootasynchronousmethods.entity**;  
  
  
**import com.fasterxml.jackson.annotation.**JsonIgnoreProperties;  
  
@JsonIgnoreProperties(ignoreUnknown = **true**)  
**public class User** {  
  
 **private String** *name*;  
 **private String** *blog*;  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public String** getBlog() {  
 **return** *blog*;  
 }  
  
 **public void** setBlog(**String** blog) {  
 **this**.*blog* = blog;  
 }  
  
 @Override  
 **public String** toString() {  
 **return** "User{" +  
 "name='" + *name* + '**\'**' +  
 ", blog='" + *blog* + '**\'**' +  
 '}';  
 }  
}

## 创建一个请求的　githib的service:

**package com.skytech.springbootasynchronousmethods.service**;  
  
**import com.skytech.springbootasynchronousmethods.entity.User**;  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.boot.web.client.RestTemplateBuilder**;  
**import org.springframework.scheduling.annotation.**Async;  
**import org.springframework.scheduling.annotation.AsyncResult**;  
**import org.springframework.stereotype.**Service;  
**import org.springframework.web.client.RestTemplate**;  
  
**import java.util.concurrent.Future**;  
  
  
@Service  
**public class GitHubLookupService** {  
 **private static final Logger *logger*** = **LoggerFactory**.getLogger(**GitHubLookupService**.**class**);  
  
 **private final RestTemplate** *restTemplate*;  
  
 **public GitHubLookupService**(**RestTemplateBuilder** restTemplateBuilder) {  
 **this**.*restTemplate* = restTemplateBuilder.build();  
 }  
  
// @Async  
 **public Future**<**User**> findUser(**String** user) **throws InterruptedException** {  
 ***logger***.info("Looking up " + user);  
 **String** url = **String**.format("https://api.github.com/users/%s", user);  
 **User** results = *restTemplate*.getForObject(url, **User**.**class**);  
 // Artificial delay of 1s for demonstration purposes  
 **Thread**.sleep(**1000L**);  
 **return new** AsyncResult<>(results);  
 }  
}

通过，RestTemplate去请求，另外加上类@Async 表明是一个异步任务。

## 开启异步任务：

**package com.skytech.springbootasynchronousmethods**;  
  
**import org.springframework.boot.SpringApplication**;  
**import org.springframework.boot.autoconfigure.**SpringBootApplication;  
**import org.springframework.scheduling.annotation.AsyncConfigurerSupport**;  
**import org.springframework.scheduling.annotation.**EnableAsync;  
**import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor**;  
  
**import java.util.concurrent.Executor**;  
  
  
@SpringBootApplication  
@EnableAsync  
**public class SpringbootAsynchronousMethodsApplication extends AsyncConfigurerSupport** {  
  
 **public static void** main(**String**[] args) {  
 **SpringApplication**.run(**SpringbootAsynchronousMethodsApplication**.**class**, args);  
 }  
  
 @Override  
 **public Executor** getAsyncExecutor() {  
 **ThreadPoolTaskExecutor** executor = **new** ThreadPoolTaskExecutor();  
 executor.setCorePoolSize(**2**);  
 executor.setMaxPoolSize(**2**);  
 executor.setQueueCapacity(**500**);  
 executor.setThreadNamePrefix("GithubLookup-");  
 executor.initialize();  
 **return** executor;  
 }  
}

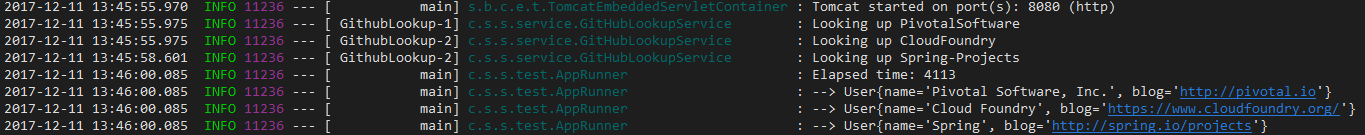
通过@EnableAsync开启异步任务；并且配置AsyncConfigurerSupport，比如最大的线程池为2.

## 测试

测试代码如下：

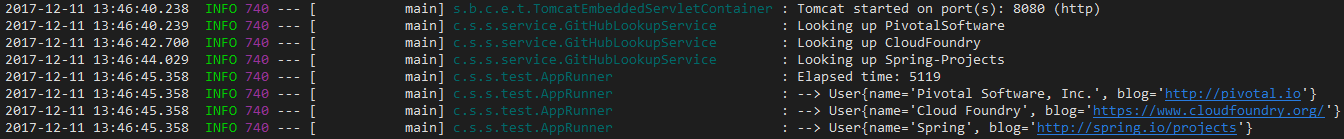
**package com.skytech.springbootasynchronousmethods.test**;  
  
  
**import com.skytech.springbootasynchronousmethods.entity.User**;  
**import com.skytech.springbootasynchronousmethods.service.GitHubLookupService**;  
**import org.slf4j.Logger**;  
**import org.slf4j.LoggerFactory**;  
**import org.springframework.boot.CommandLineRunner**;  
**import org.springframework.stereotype.**Component;  
  
**import java.util.concurrent.Future**;  
  
@Component  
**public class AppRunner implements CommandLineRunner** {  
  
 **private static final Logger *logger*** = **LoggerFactory**.getLogger(**AppRunner**.**class**);  
  
 **private final GitHubLookupService** *gitHubLookupService*;  
  
 **public AppRunner**(**GitHubLookupService** gitHubLookupService) {  
 **this**.*gitHubLookupService* = gitHubLookupService;  
 }  
  
 @Override  
 **public void** run(**String**... strings) **throws Exception** {  
 **long** start = **System**.currentTimeMillis();  
  
 **Future**<**User**> page1 = *gitHubLookupService*.findUser("PivotalSoftware");  
 **Future**<**User**> page2 = *gitHubLookupService*.findUser("CloudFoundry");  
 **Future**<**User**> page3 = *gitHubLookupService*.findUser("Spring-Projects");  
  
 **while** (!(page1.isDone() && page2.isDone() && page3.isDone())) {  
 **Thread**.sleep(**10**);  
 }  
  
 ***logger***.info("Elapsed time: " + (**System**.currentTimeMillis() - start));  
 ***logger***.info("--> " + page1.get());  
 ***logger***.info("--> " + page2.get());  
 ***logger***.info("--> " + page3.get());  
 }  
}

启动程序，控制台会打印：



分析：可以卡的前面2个方法分别在GithubLookup-1 和GithubLookup-2执行，第三个在GithubLookup-2执行，注意因为在配置线程池的时候最大线程为2.如果你把线程池的个数为3的时候，耗时减少。

如果去掉@Async，你会发现，执行这三个方法都在main线程中执行。耗时总结，如下：



通过这一个小的栗子，你应该对异步任务有了一定的了解。

## 参考资料

<https://spring.io/guides/gs/async-method/>