# 1今天任务

springboot框架的学习

spring-data：访问数据库

spring-boot-web:springmvc,接受请求和响应

# 2配置sts开发工具

Sys 基于eclipse工具的： 是针对spring框架提供的：

需要配置：

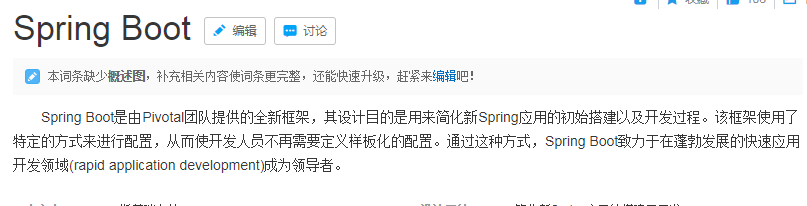
配置java--->jre

配置maven (maven的安装路径，加载maven的setttings.xml)

配置workspace编码

# 3介绍Springboot

## 3.1百度百科的介绍：



总结：

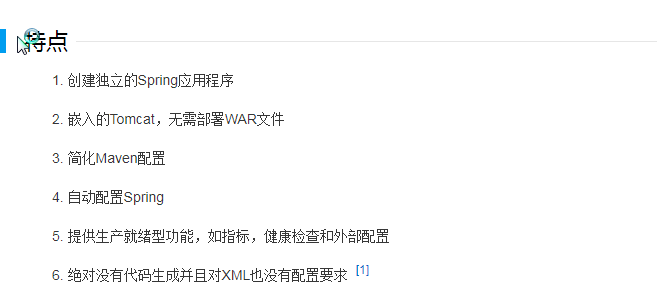
简化传统的spring项目在开发过程中的大量配置，0配置

实现好多开源的框架的集成，实现应用程序的快速开发

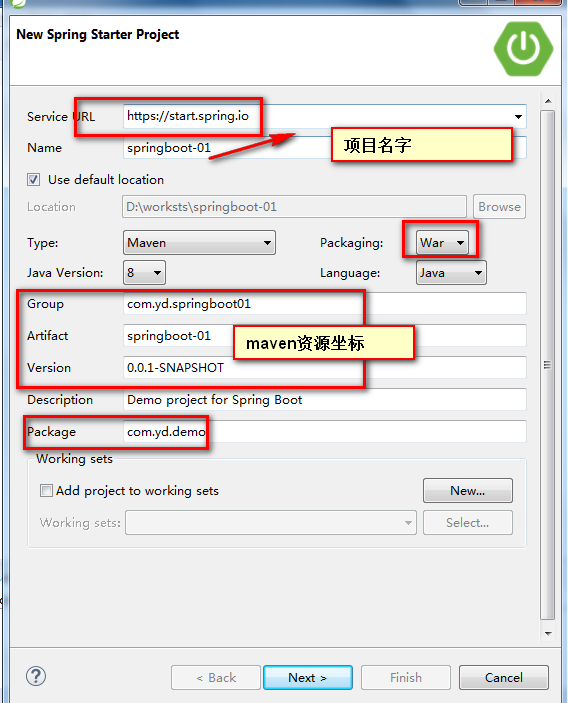
实现了maven配置的简化，自动完成spring配置

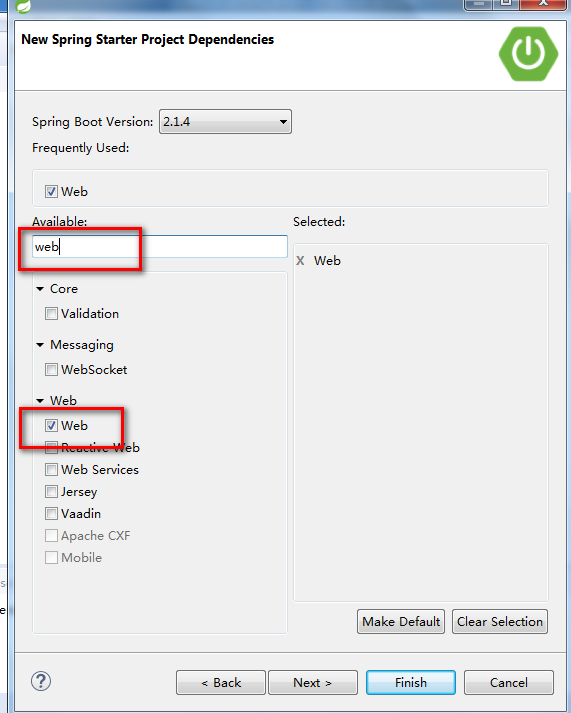
## 3.2特点：

从百度百科查询springboot的特点：

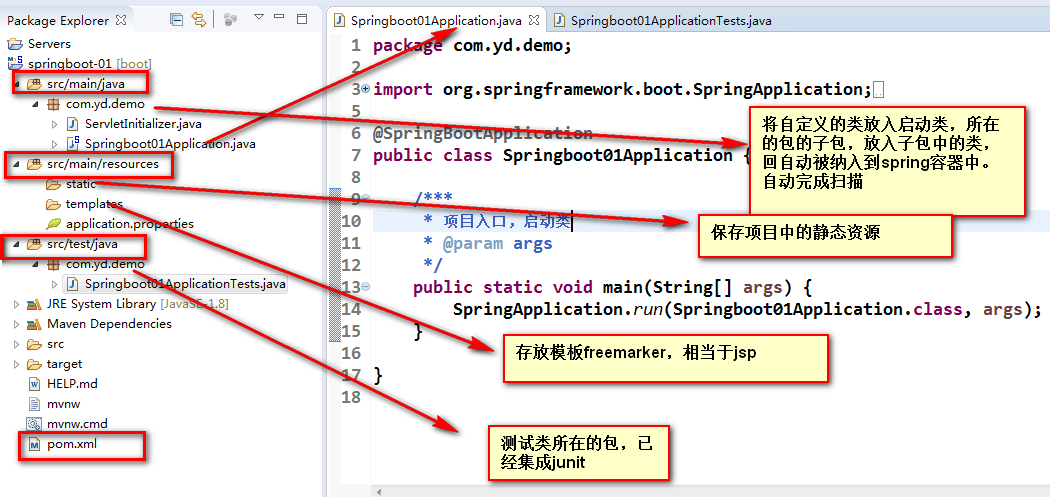


# 4第一个springboot应用





## 4.1Springboot项目的目录结构



## 4.2创建HelloController

|  |
| --- |
| **package** com.yd.demo.controller;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  @Controller //将Controller纳入到spring容器,不需要配置包扫描  **public** **class** HelloController {  /\*\*\*  \* 接收前台的请求，完成异步响应  \* **@param** name  \* **@return 返回的字符串不是跳转的页面： @ResponseBody的作用，完成一个异步的响应： 默认情况springboot不支持jsp， 需要进行单独的配置才能支持。**  \*/  @RequestMapping("/helloSpringboot")  @ResponseBody  **public** String sayHello(String name) {  **return** "hello "+name;  }  } |

## 4.3创建HelloController02

|  |
| --- |
| **package** com.yd.demo.controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController //@Controller+@ResponseBody 组合注解  **public** **class** HelloController2 {  /\*\*\*\*  \* httpclient=http+json  \*/  /\*\*\*  \* 完成异步响应  \* **@param** name  \* **@return 获得数据，并不是页面的跳转**  \*/  @RequestMapping("/helloSpringboot02")  **public** String sayHeollo02(String name) {  **return** "hello "+name;  }  } |

## 4.4配置application.properties

在启动项目的时候默认加载：

Properties文件在默认情况下， 不支持中文， 默认文件的编码是iso： 修改成utf-8就能支持中文：

|  |
| --- |
| #配置tomcat端口号  server.port=8081  #配置项目的上下文  server.servlet.context-path=/jd  server.context-path=/jd |

## 4.5启动springboot的项目

启动程序的入口：

# 5springboot默认加载application.yml

application.yml文件： 或者是application.yaml

启动项目的时候默认加载

是通过结构化的方式进行数据的配置

该文件中的内容，被赋值给某些类的属性：这些实体类可以是自定义可以是系统类。

## 5.1创建User实体类

|  |
| --- |
| **package** com.yd.demo.entity;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  /\*\*\*  \* 封装在application.yml配置文件中配置的用户信息  \* **@author** Administrator  \*  \*/  @Component //将user纳入到spring容器当中  //启动项目实例化User类，并且将applicatoin.yml中配置的信息封装进来  @ConfigurationProperties(prefix="user")  **public** **class** User {    **private** Integer uid;  **private** String uname;  **private** String pwd;  **public** Integer getUid() {  **return** uid;  }  **public** **void** setUid(Integer uid) {  **this**.uid = uid;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  **public** String getPwd() {  **return** pwd;  }  **public** **void** setPwd(String pwd) {  **this**.pwd = pwd;  }      } |

## 5.2创建UserController类

|  |
| --- |
| **package** com.yd.demo.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.yd.demo.entity.User;  @RestController  **public** **class** UserController {  //注入User类的对象  @Autowired  **private** User user;  @RequestMapping("/showUser")  **public** String showUser() {  **return** user.getUid()+"----"+user.getUname()+"-----"+user.getPwd();  }  } |

## 5.3创建一个application.yml文件

Server当中的内容的属性信息会封装到系统提供的类当中；

User 配置的信息会封装到自定义的实体当中；

|  |
| --- |
| server:  port: 90 #tomcat的端口号  servlet:  context-path: / #项目访问的上下文    user:  uid: 1000  uname: zhangsan  pwd: 909090 |

## 5.4：启动项目访问

# 6RestFul风格API的设计

RestFul是一个软件架构的设计风格,指导服务器端怎么设计一套API(远程方法)

用于访问特定资源API。简单，便于维护，方便访问消费者的调用。

发布服务

method()

## 6.1Http动词

表明http请求的不同方式。不同请求方式对应着不同的数据库操作

post insert

get select

delete delete

put update

## 6.2RestFul的API设计原则

使用一个url(名词)，访问服务器的资源，而且可以实现资源的不同操作

/users post insert

/users get select

/users delete delete

/users put update

使用同一个url(名词)，对该url表示的服务器上的资源进行访问，用http动词进行资源状态转化

理解转换： 查询是记录 转换成对象：：

添加是将对象转换成记录：：

优点：减少了url的个数：

方便客户端的调用

## 6.3设置用户数据访问的API

### 6.3.1创建User实体类

|  |
| --- |
| **package** com.yd.demo.entity;    **public** **class** User {    **private** Integer uid;  **private** String uname;  **private** String pwd;  **public** Integer getUid() {  **return** uid;  }  **public** **void** setUid(Integer uid) {  **this**.uid = uid;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  **public** String getPwd() {  **return** pwd;  }  **public** **void** setPwd(String pwd) {  **this**.pwd = pwd;  }      } |

### 6.3.2创建UserController

在Controller中发布远程http服务（rpc的服务端）

|  |
| --- |
| **package** com.yd.demo.controller;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.springframework.web.bind.annotation.DeleteMapping;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.PostMapping;  **import** org.springframework.web.bind.annotation.PutMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.yd.demo.entity.User;  @RestController  **public** **class** UesrController {  /\*\*\*  \* 处理用户的添加请求，该方法只能接受post请求  \*/  @PostMapping("/users")  **public** User addUsers(User user) {  System.***out***.println("--------添加用户------");  **return** user;  }    /\*\*\*  \* 查询某个用户信息,该方法只能接受get请求  \* users?uid=1000  \* /users/100  \*/  @GetMapping("/users/{id}")  **public** User loadUser(@PathVariable Integer id) {  User user=**new** User();  user.setUid(id);  user.setUname("888888");  user.setPwd("909090");  **return** user;  }  /\*\*\*\*  \* 查询所有用户信息  \*/  @GetMapping("/users")  **public** List<User> loadUserList(){  User user1=**new** User();  user1.setUid(345);  user1.setUname("888888");  user1.setPwd("909090");    User user2=**new** User();  user2.setUid(678);  user2.setUname("1111");  user2.setPwd("333333");    List<User> list=**new** ArrayList<>();  list.add(user1);  list.add(user2);    **return** list;    }    /\*\*\*  \* 更新用户信息  \*/  @PutMapping("/users")  **public** User updateUser(User user) {  System.***out***.println("---------更新用户信息-----------");  **return** user;    }    /\*\*\*  \* 删除用户信息  \*/  @DeleteMapping("/users/{id}")  **public** User deleteUser(@PathVariable Integer id) {  System.***out***.println("---------删除用户信息-----------");  User user2=**new** User();  user2.setUid(id);  user2.setUname("被删除的用户");  user2.setPwd("333333");  **return** user2;    }  } |

## 6.4启动项目测试

、

# 7Api文档的开发（面试重点）

为什么要API开发？

第一： 给消费者提供一份文档， 供调用者使用。

第二：前后分离的开发，同一个公司开后分离的开发，由前端的团队提供api文档.

同一个公司， 不同的团队，在开发之前， 先定义文档。定义一些开发的规范

如果是不同的公司 由服务的提供方法提供文档。

如果是不同的公司的项目相互调用，有服务提供端开发该文件给消费者

后台服务

Controller

前

APP,PC(UI),平板

文档：先定义一般都是前端人员写。

|  |  |
| --- | --- |
| 功能 | 实现用户的注册 |
| url | /users |
| params | Uid =123  Uname=zhangsan  Pwd=456 |
| Method | Post |
| Return | 成功  {  Status：200  Message : "注册成功"  }  失败  {  Status：-1  Message : "注册失败"  } |

演示微信公众号平台开发的API文档：

<https://mp.weixin.qq.com/>

点击订阅号：开发文档 ， 开始开发，token获得 用户列表等功能模块都有相应的案例。

# 8Springboot中的拦截器

## 8.1自定义拦截器

|  |
| --- |
| **package** com.yd.demo.interceptor;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.stereotype.Component;  **import** org.springframework.util.StringUtils;  **import** org.springframework.web.servlet.HandlerInterceptor;  /\*\*\*  \* 自定义拦截器，实现HandlerInterceptor接口  \* **@author** Administrator  \*  \*/  @Component 定义的拦截器需要交给spring管理。 放在spring容器当中  **public** **class** PrimessionInterceptor **implements** HandlerInterceptor {  /\*\*\*  \* 拦截器需要实现的业务逻辑  \*/  @Override  **public** **boolean** preHandle(HttpServletRequest request,  HttpServletResponse response,  Object handler)  **throws** Exception {  //获得消费者提价到生产者的token  String token=request.getParameter("token");    **if**(!StringUtils.*isEmpty*(token)) {  **if**(token.equals("123456")) {  //有权限，访问远程的http服务  **return** **true**;//拦截器放行，调用Controller的方法  }  }  response.setContentType("text/html;charset=UTF-8");  response.getWriter().write("权限不足.......");  **return** **false**; //不能放行  }    } |

总结： 实现的接口并没有要求重写接口当中的方法， 因为JDK1.8 对接口的功能进行增强。 方法可以有方法体。 子类对象当中只需要重写需要的方法就ok。

## 8.2注册拦截器

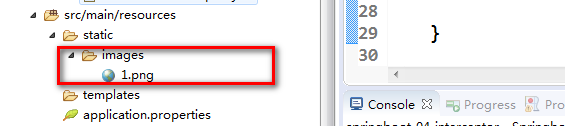
|  |
| --- |
| **package** com.yd.demo.config;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  **import** com.yd.demo.interceptor.PrimessionInterceptor;  /\*\*\*  \* 完成拦截器的配置  \* **@author** Administrator  \* WebConfig=springmvc.xml  \*/  @Configuration  **public** **class** WebConfig **implements** WebMvcConfigurer {  //注入自定义的拦截器对象  @Autowired  **private** PrimessionInterceptor primessionInterceptor;    @Override  **public** **void** addInterceptors(InterceptorRegistry registry) {  // **TODO** Auto-generated method stub  registry.addInterceptor(primessionInterceptor);  }    } |

总结：； springmvc当中注册拦截器， 第一需要创建拦截器， 第二需要在springmvc.xml当中进行配置。 springboot当中讲究0配置， 将配置文件写成了java类来替代。 在该类当中打上一个注解： @ Configuration 表明该类是一个配置类， 该类在springboot启动的时候，就会自动加载。

## 8.3PostMan测试拦截器是否生效

# 9Springboot静态资源的处理

## 9.1静态资源的默认保存位置



拦截器会静态资源进行拦截.当前端控制器配置\*.do的时候， 静态资源就不再拦截。

## 9.2拦截器放行静态资源

|  |
| --- |
| **package** com.yd.demo.config;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  **import** com.yd.demo.interceptor.PrimessionInterceptor;  /\*\*\*  \* 完成拦截器的配置  \* **@author** Administrator  \* WebConfig=springmvc.xml  \*/  @Configuration  **public** **class** WebConfig **implements** WebMvcConfigurer {  //注入自定义的拦截器对象  @Autowired  **private** PrimessionInterceptor primessionInterceptor;    @Override  **public** **void** addInterceptors(InterceptorRegistry registry) {  // 这是一个链式编程 默认链接所有/\*\*  registry.addInterceptor(primessionInterceptor)  .excludePathPatterns("/images/\*\*") //不想要拦截的请求  .addPathPatterns("/\*\*"); //需要拦截的请求    }    } |

# 10Springboot对jsp的支持

补充： 释放window8080 的端口号：



Springboot 默认不支持jsp。 直接创建一个index.jsp 文件， 会将index.jsp文件下载。

主要原因是springboot不能解析.jsp文件，不识别： 添加一个依赖就可以解决该问题。

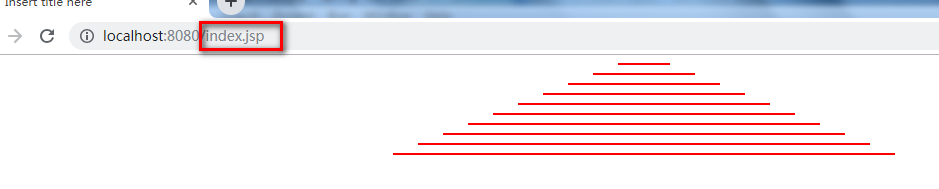
## 10.1添加jsp解析器依赖

|  |
| --- |
| <!-- 添加tomcat对jsp的支持，解析jsp -->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  </dependency> |

## 10.2创建index.jsp

|  |
| --- |
| <body>  <%  **for**(**int** x=1;x<=10;x++){  out.print("<hr/ color='red' width='"+(x\*50)+"'>");  }  %>  </body> |

## 10.3启动springboot访问jsp



此时开发jsp的时候，我们经常使用jstl， 但是缺少jstl的jar包， 需要再添加一个jstl依赖。

<!-- https://mvnrepository.com/artifact/javax.servlet.jsp.jstl/jstl -->

<dependency>

<groupId>javax.servlet.jsp.jstl</groupId>

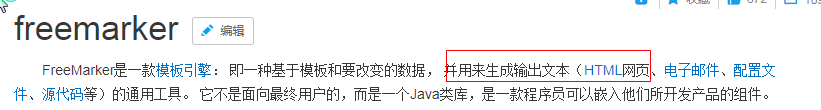
<artifactId>jstl</artifactId>

<version>1.2</version>

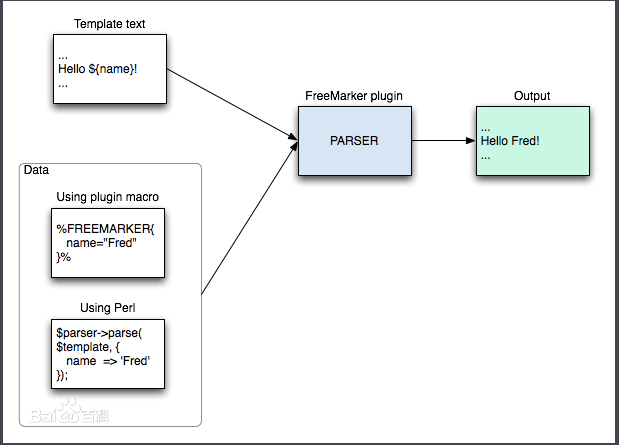
</dependency>

# 11springboot对freemaker的支持

freemaker是模板引擎，实现用户需要的UI.

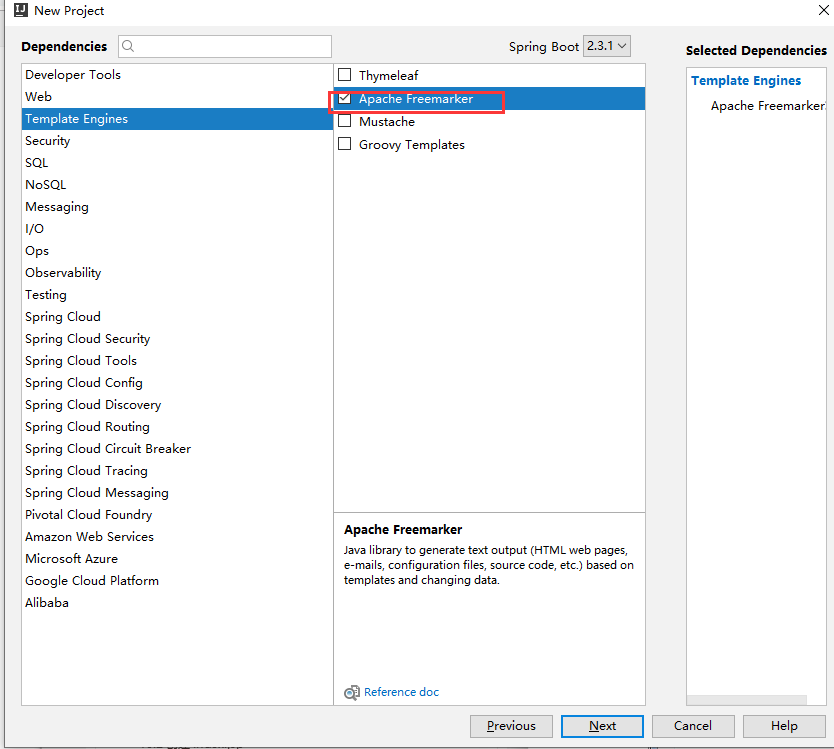


工作原理：



模板+ 数据 --->使用freemark引擎 就会 生成一个HTML页面。

## 11.1创建springboot-06-freemarker项目



创建成功之后： pom文件当中会添加freemarker的依赖：

*<!--引入了freemarker依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-freemarker</**artifactId**>  
</**dependency**>

## 11.2创建User实体类

|  |
| --- |
| **package** com.yd.demo.entity;    **public** **class** User {    **private** Integer uid;  **private** String uname;  **private** String pwd;  **public** Integer getUid() {  **return** uid;  }  **public** **void** setUid(Integer uid) {  **this**.uid = uid;  }  **public** String getUname() {  **return** uname;  }  **public** **void** setUname(String uname) {  **this**.uname = uname;  }  **public** String getPwd() {  **return** pwd;  }  **public** **void** setPwd(String pwd) {  **this**.pwd = pwd;  }      } |

## 11.3创建UserController

|  |
| --- |
| **package** com.yd.demo.controller;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.ui.Model;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** com.yd.demo.entity.User;  @Controller  **public** **class** UserController {  /\*\*\*  \* 加载用户列表信息  \*/  @RequestMapping("/loadUserList")  **public** String loadUsersList(Model model) {    User u1=**new** User();  u1.setUid(111);  u1.setUname("zhangsan");  u1.setPwd("909090");    User u2=**new** User();  u2.setUid(2222);  u2.setUname("wangwu");  u2.setPwd("909090");    List<User> lts=**new** ArrayList<>();  lts.add(u1);  lts.add(u2);    //将lts集合保存到model  model.addAttribute("lts", lts);    model.addAttribute("sex", 1);  model.addAttribute("username", "wangwu");    **return** "index";  }  } |

## 11.4创建index.ftlh文件 在resources/templates下

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset=UTF-8">  <title>Insert title here</title>  </head>  <body>    用户名:${username}  <hr/>    <#if sex==1>  男  <#else>  女  </#if>    <hr/>    <#list lts as u>  ${u.uname}-----${u.pwd}-----${u.uid}  <hr/>  </#list>  </body>  </html> |

# 12 springboot 整合Mybatis框架

## 12.1 加入Mybatis的依赖+Mysql 的依赖

*<!--引入web层的启动器-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>  
*<!--引入Mybatis的依赖-->*<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>2.1.0</**version**>  
</**dependency**>  
*<!--引入Mysql的驱动：高版本-->*<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**scope**>runtime</**scope**>  
</**dependency**>  
*<!--引入了单元测试的依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
</**dependency**>

## 12.2 加入数据库的链接信息

可以使用Properties文件， 可以使用yml文件

**spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver  
spring.datasource.url**=**jdbc:mysql://localhost:3306/mybatis?useUnicode=true&useSSL=true&zeroDateTimeBehavior=convertToNull&serverTimezone=UTC  
spring.datasource.username**=**root  
spring.datasource.password**=**root**

**Spring.datasource.type=指定链接池的具体限定类名（可以不指定，默认）**

总结： 高版本的Mysql，必须加入url后的时区， 否则报错

## 12.3 创建一个user表

CREATE TABLE `user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`username` varchar(32) NOT NULL COMMENT '用户名称',

`birthday` date DEFAULT NULL COMMENT '生日',

`sex` char(1) DEFAULT NULL COMMENT '性别',

`address` varchar(256) DEFAULT NULL COMMENT '地址',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=66 DEFAULT CHARSET=utf8;

插入数据

insert into `user`(`id`,`username`,`birthday`,`sex`,`address`)

values (4,'老登','2019-04-24','3','燕郊'),(45,'tongtong','2019-04-24','1','yanjiao'),

(49,'花儿','2019-04-24',NULL,'北京'),(51,'老谭','2019-04-25','1','华科'),

(55,'老谭','2019-04-25','3','华科2'),(56,'老谭','2019-04-25','3','华科2'),

(57,'老谭4','2019-04-25','3','华科2'),(60,'小peng','2019-07-04','1','新疆'),

(61,'小peng','2019-07-04','1','新疆'),(62,'xx',NULL,NULL,NULL),

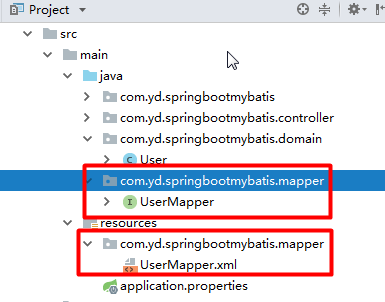
(63,'yy',NULL,NULL,NULL),(65,'小peng2','2019-07-04','1','新疆');

## 12.4 创建实体bean

**package** com.yd.springbootmybatis.domain;  
  
**import** java.util.Date;  
  
**public class** User {  
 **private** Integer **id**;  
 **private** String **username**;  
 **private** Date **birthday**;  
 **private** String **sex**;  
 **private** String **address**;  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getUsername() {  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
 **this**.**username** = username;  
 }  
  
 **public** Date getBirthday() {  
 **return birthday**;  
 }  
  
 **public void** setBirthday(Date birthday) {  
 **this**.**birthday** = birthday;  
 }  
  
 **public** String getSex() {  
 **return sex**;  
 }  
  
 **public void** setSex(String sex) {  
 **this**.**sex** = sex;  
 }  
  
 **public** String getAddress() {  
 **return address**;  
 }  
  
 **public void** setAddress(String address) {  
 **this**.**address** = address;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "User{"** +  
 **"id="** + **id** +  
 **", username='"** + **username** + **'\''** +  
 **", birthday="** + **birthday** +  
 **", sex='"** + **sex** + **'\''** +  
 **", address='"** + **address** + **'\''** +  
 **'}'**;  
 }  
}

## 12.5 编写mapper

在src\main\resource 下创建一个目录： 创建的目录必须是mapper文件存放在同一个目录，并且他们的名称要保持一致。



创建mapper接口

**package** com.yd.springbootmybatis.mapper;  
  
**import** com.yd.springbootmybatis.domain.User;  
**import** org.apache.ibatis.annotations.Mapper;  
  
**import** java.util.List;  
  
*//@Mapper 标记类是一个Mybatis的的mapper接口， 可以在springboot自动扫描到spring上下文环境当中*@Mapper  
**public interface** UserMapper {  
 */\*\*  
 \* 查询用户列表  
 \** ***@return*** *\*/* List<User> finAll();  
  
}

创建的具体的mapper.xml文件如下：

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>***<**mapper namespace="com.yd.springbootmybatis.mapper.UserMapper"**>  
 <**select id="finAll" resultType="user"**>  
 select *\** from user  
 </**select**>  
</**mapper**>

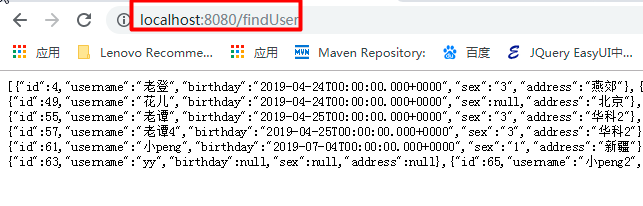
## 12.6 在application.properties中添加Mybatis的信息

*#配置Mybatis的链接信息：  
#配置别名扫描***mybatis.type-aliases-package**=**com.yd.springbootmybatis.domain***#加载Mybatis的映射文件：***mybatis.mapper-locations**=**classpath:com/yd/springbootmybatis/mapper/\*Mapper.xml**

## 12.7 编写Controller进行测试

**package** com.yd.springbootmybatis.controller;  
  
**import** com.yd.springbootmybatis.domain.User;  
**import** com.yd.springbootmybatis.mapper.UserMapper;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Component;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RestController;  
  
**import** java.util.List;  
  
*//此注解加入spring容器当中： 并且响应是异步的*@RestController  
**public class** UserController {  
  
 @Autowired  
 **private** UserMapper **userMapper**;  
  
 @RequestMapping(**"/findUser"**)  
 **public** List<User> showUser(){  
 List<User> users = **userMapper**.finAll();  
 **return** users;  
 }  
}

## 12.8 测试的结果如下图



# 13 springboot整合Junit

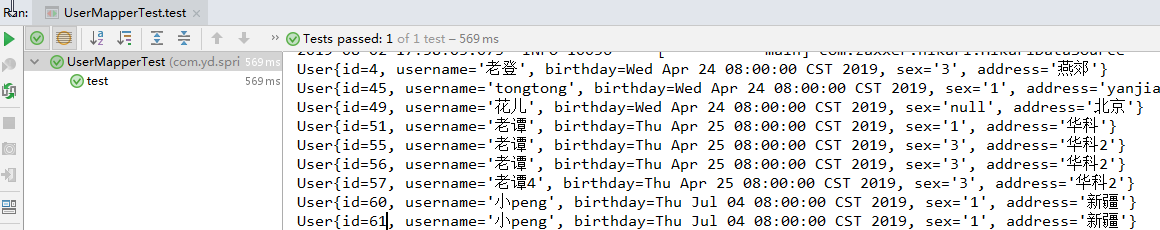
## 13.1 添加Junit的依赖

*<!--引入了单元测试的依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
</**dependency**>

## 13.2 编写测试类

**package** com.yd.springbootmybatis.test;  
  
**import** com.yd.springbootmybatis.SpringbootMybatisApplication;  
**import** com.yd.springbootmybatis.domain.User;  
**import** com.yd.springbootmybatis.mapper.UserMapper;  
**import** org.junit.Test;  
**import** org.junit.runner.RunWith;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.boot.test.context.SpringBootTest;  
**import** org.springframework.test.context.junit4.SpringRunner;  
  
**import** java.util.List;  
  
*/\*\*  
 \* springboot 整合junit  
 \*/*@RunWith(SpringRunner.**class**)  
@SpringBootTest(classes = SpringbootMybatisApplication.**class**)  
**public class** UserMapperTest {  
  
 @Autowired  
 **private** UserMapper **userMapper**;  
  
 @Test  
 **public void** test(){  
 List<User> users = **userMapper**.finAll();  
 **for** (User user: users ) {  
 System.***out***.println(user);  
 }  
  
 }  
}

## 13.3 运行检查测试结果



## 13.4 注意

在测试类上加入了RunWith 引入的SpringRunner.class 继承与SpringJunit4ClassRunner，使用哪一个spring提供的测试引擎都可以。



@SpringBootTest 注解的属性指定的是引导类的字节码对象。

代码如下：

引入的依赖：

*<!--springboot 引入test的依赖-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
  
</**dependency**>

测试代码：

@SpringBootTest(classes = DemoApplication.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)*//指定了spring 整合junit的类：***class** DemoApplicationTests {  
  
 @Autowired  
 **private** UserMapper **userMapper**;  
 @Test  
 **void** contextLoads() {  
 List<User> allUsers = **userMapper**.findAllUsers();  
 System.***out***.println(allUsers);  
 }  
  
}

# 14 springboot整合Redis（重点）

## 14.1 添加Redis的依赖

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

## 14.2 在application.xml配置Redis的连接信息

Redis的服务器必须启动，可以是本地window系统的，可以是远程Linux系统的

*#Redis配置信息***spring.redis.host**=**192.168.153.10  
spring.redis.port**=**6379**

## 14.3 注入RedisTemplate测试Redis

Redis应用场景：

Redis作为一个缓存数据库使用。 当我们service层需要的数据，不直接从dao层获得，先是从redis缓存当中获得，如果redis当中有数据，直接返回数据。 如果redis当中没有数据，然后调用dao查询数据，查询的数据返回。并且把查询到的数据存放到redis当中一份。 那么使用该数据的时候，redis当中就会有数据。

@RunWith(SpringRunner.**class**)  
@SpringBootTest(classes = SpringbootMybatisApplication.**class**)  
**public class** RedisTest {  
  
 *//注入Mybatis的UserMapper* @Autowired  
 **private** UserMapper **userMapper**;  
  
 *//注入Redis 模板类对象：* @Autowired  
 **private** RedisTemplate<String,String> **redisTemplate**;  
  
 @Test  
 **public void** test() **throws** Exception{  
  
 *//从缓存当中获得执行的数据：* String name = **redisTemplate**.boundValueOps(**"name"**).get();  
  
 *//判断从redis当中是否获得数据：* **if** (name== **null**){  
 *//查询数据库获得数据：* List<User> users = **userMapper**.finAll();  
 *//将users转换成json格式字符串* ObjectMapper om = **new** ObjectMapper();  
 name = om.writeValueAsString(users);  
  
 *//将写好的json存在redis当中一份* **redisTemplate**.boundValueOps(**"name"**).set(name);  
 System.***out***.println(**"=========从数据库当中获得数据==========="**);  
 }**else**{  
 System.***out***.println(**"========Redis当中获得数据=========="**);  
 }  
 System.***out***.println(name);  
  
 }  
}