目录

[一、 Spring Tools Suite安装 3](#_Toc11242)

[1. 下载&安装 3](#_Toc343)

[2. 基本配置 3](#_Toc7379)

[3. Spring Boot工程创建 7](#_Toc23820)

[（1） 创建工程 7](#_Toc29320)

[（2） 运行 11](#_Toc17679)

[（3） 修改端口号和发布路径 13](#_Toc5173)

[4. Maven的本地安装 13](#_Toc4992)

[二、 Spring Boot 简易应用 14](#_Toc19606)

[1. Spring Boot访问规则 14](#_Toc17631)

[2. 请求需要有参数 15](#_Toc22550)

[3. 数据库的简易操作（CURD） 16](#_Toc13866)

[4. JPA的CURD 17](#_Toc28915)

[5. EntityManager的CURD 20](#_Toc11517)

[1. Spring Boot JPA 22](#_Toc12644)

[多表查询 22](#_Toc16784)

[分页查询 23](#_Toc9240)

[条件查询 24](#_Toc16841)

[2. EntityManager（待优化） 25](#_Toc23945)

[(1) 创建分页实体类 25](#_Toc7614)

[(2) 创建基本操作方法 28](#_Toc3312)

[(3) 实现分页、多表、条件查询 31](#_Toc12629)

[3. Sping Boot 定时任务 32](#_Toc10725)

[三、 细节解释 33](#_Toc7147)

[1. 注解 33](#_Toc19435)

[2. Pom.xml 详解 34](#_Toc7717)

[3. application详解 37](#_Toc25467)

[（1） 核心配置文件及两种格式 37](#_Toc30553)

[（2） 多环境配置文件 37](#_Toc2126)

[（3） 自定义注解 38](#_Toc18227)

[4. 引入jsp 38](#_Toc28394)

[(1) Pom.xml中 引入相关jar包 38](#_Toc23933)

[(2) application.properties中 配置 39](#_Toc26573)

[(3) 创建index.jsp 40](#_Toc28982)

[(4) 设置controller 40](#_Toc7328)

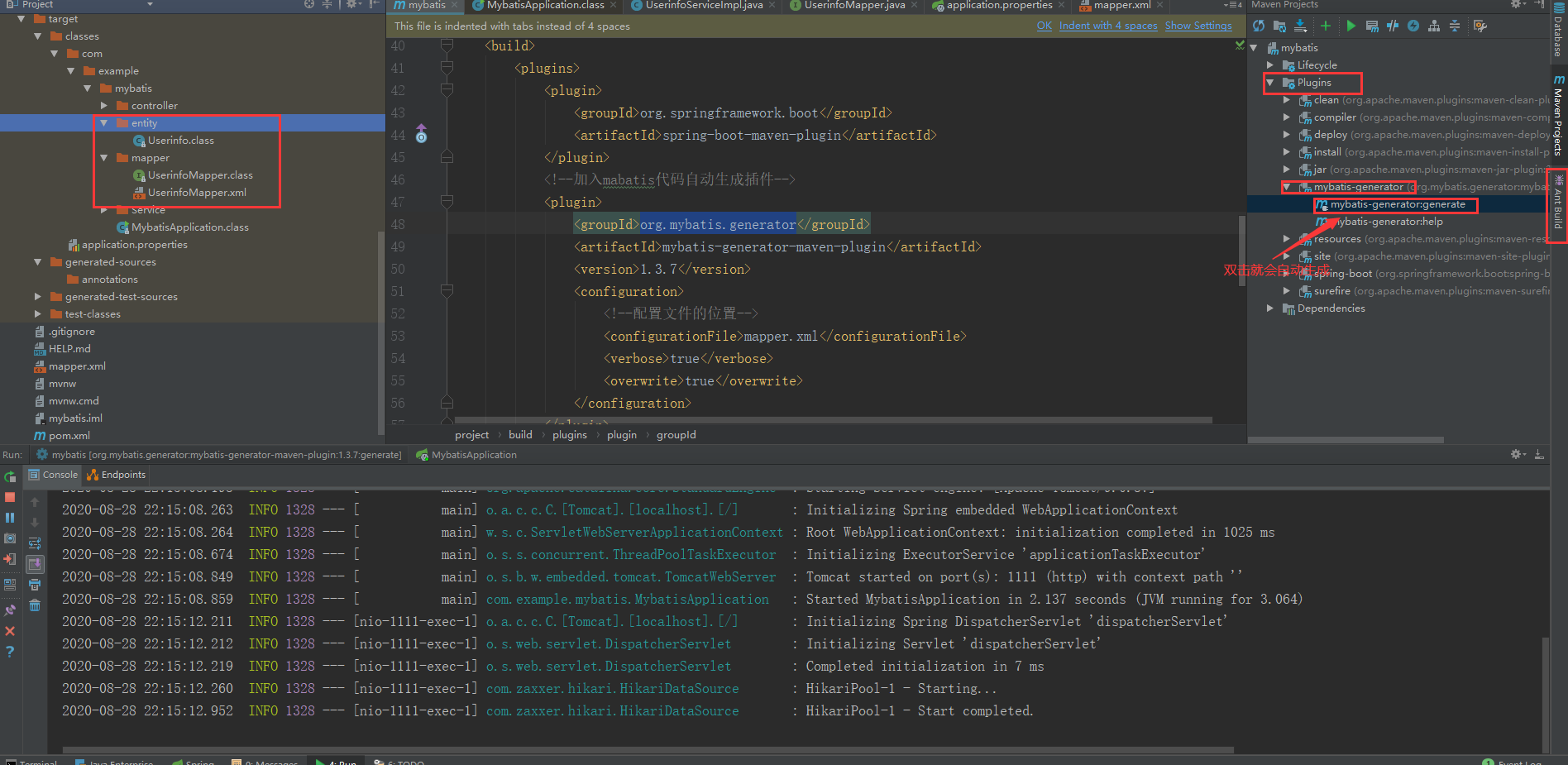
[5. 引入mybatis ★ 41](#_Toc16736)

[（1） pom.xml文件的配置 41](#_Toc9307)

[（2） application.properties文件中配置 42](#_Toc29988)

[（3） 编写mapper.xml 42](#_Toc3402)

[（4） 使用代码生成工具生成代码org.mybatis.generator 44](#_Toc32325)

[ 44](#_Toc13377)

[（5） server层及controller层的编写 45](#_Toc6134)

[（6） 加入Mapper注解即可运行 45](#_Toc14990)

[（7） Mybatis的详细使用 45](#_Toc3468)

[6. 引入事务 46](#_Toc14445)

[四、 插件的使用 46](#_Toc4011)

[1. Swagger 46](#_Toc30717)

[五、 语法进阶 48](#_Toc21462)

[1. class 及 interface 和它的实现implements 48](#_Toc9485)

[六、 可能会出现的问题 49](#_Toc22927)

[1. 如何pom.xml 报错 49](#_Toc67)

[2. 连数据库 49](#_Toc18794)

[3. pom.xml 加入jpa和mysql 49](#_Toc22621)

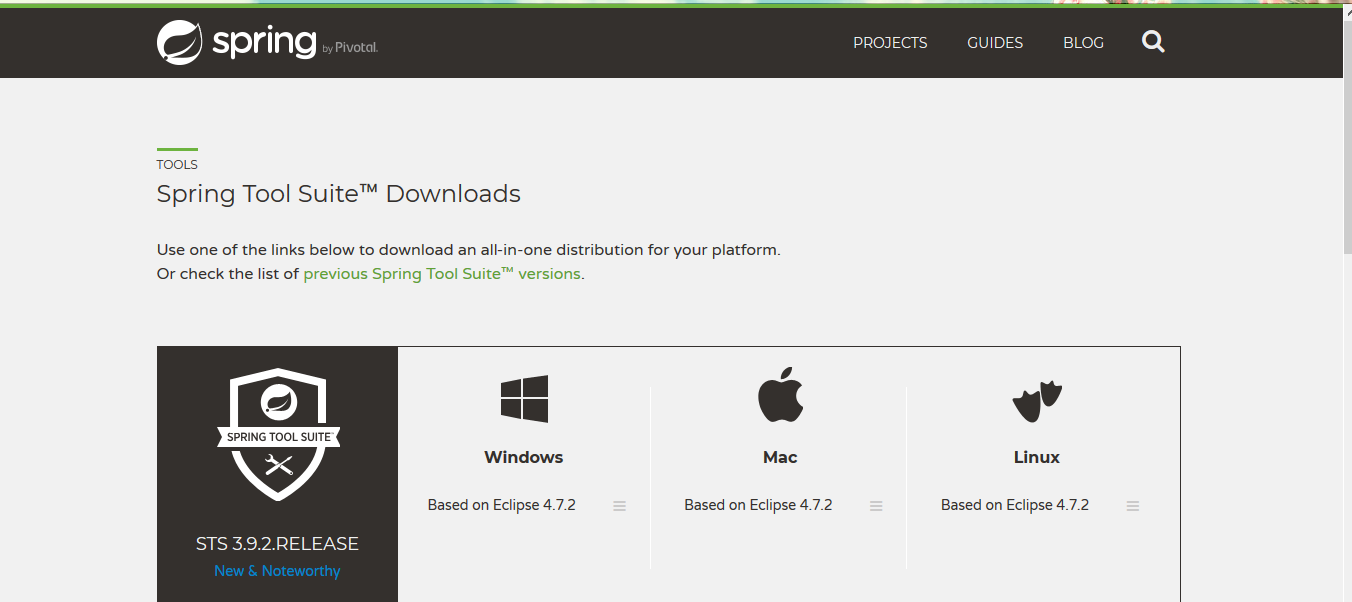
[4. 删除方法 50](#_Toc17853)

[5. 修改编码 50](#_Toc25249)

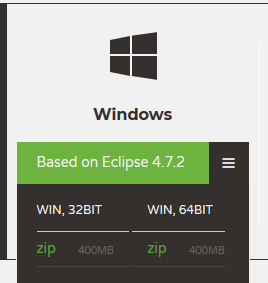
# Spring Tools Suite安装

## 下载&安装

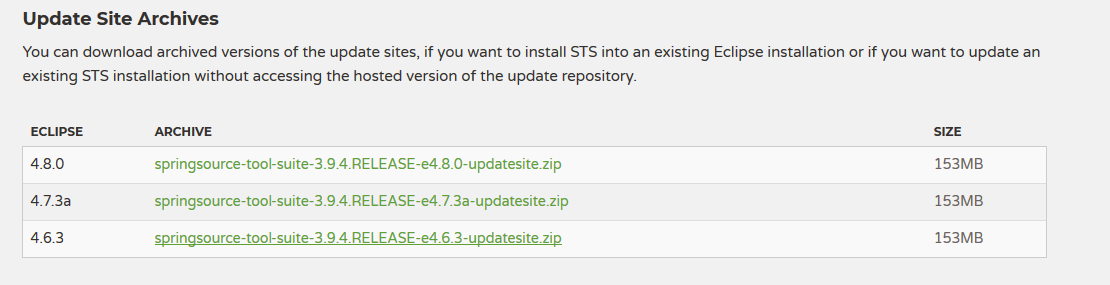
1、打开Spring Tools Suite 下载网址（http://spring.io/tools/sts/all），如下图



2、点击 Based Eclipse，选择适合自己的版本下载

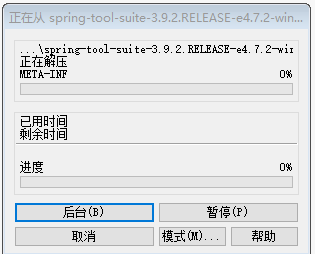


1. 下边的也可以直接下载了用

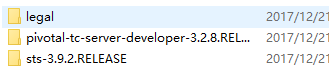


## 基本配置

1、打开下载好的ZIP文件，把里面的文件全部解压到自己习惯的位置上（例如: D:\spring-tool-suite，以下均以该路径说明）



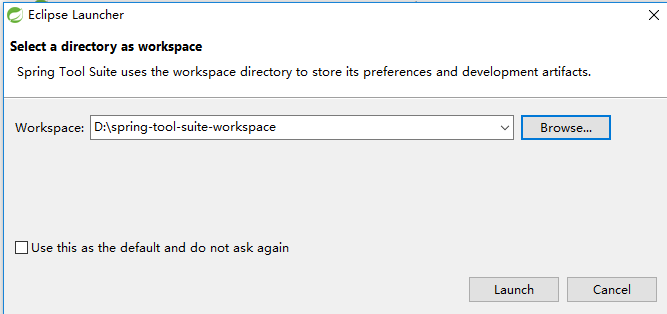
2、打开刚解压到的路径下面，有三个文件（如下面）legal是授权信息，pivotal-tc-server-developer-3.2.8.RELEASE是集成的服务器pivotal-tc-server（Spring Tools Suites集成的是开发版，是免费的，另外还有两个版本标准版和Spring版本不是免费）



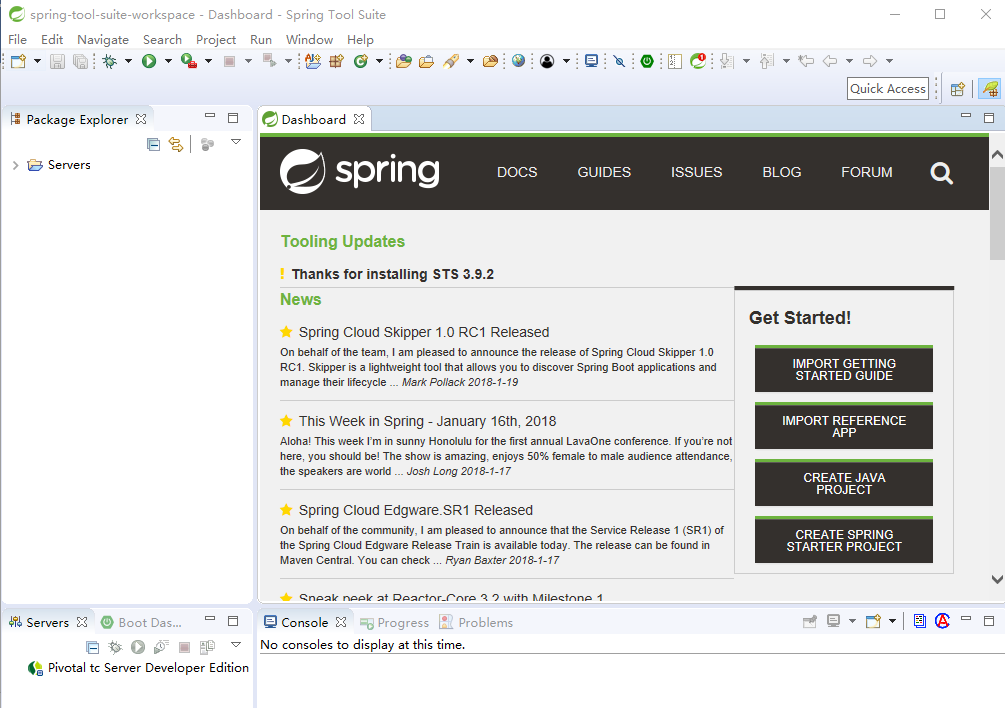
3、打开sts-3.9.2.RELEASE目录（如下图），STS.exe为主要执行程序，我们把它创建一个快捷方式（右键->发送到->桌面快捷方式）

![C:\Users\kds\AppData\Roaming\Tencent\Users\490707478\TIM\WinTemp\RichOle\HNDR%D%{PH9KF5HFW6G0](7.png](data:image/png;base64,)

4、执行STS.exe或者自己前面创建的快捷方式，选择工作空间位置（如下面图一，例如：D:\ spring-tool-suite-workspace），然后点击启动（Launch）,等待加载完成（加载完成界面如下面图二）

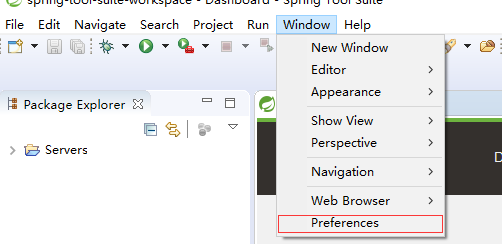


（图一）

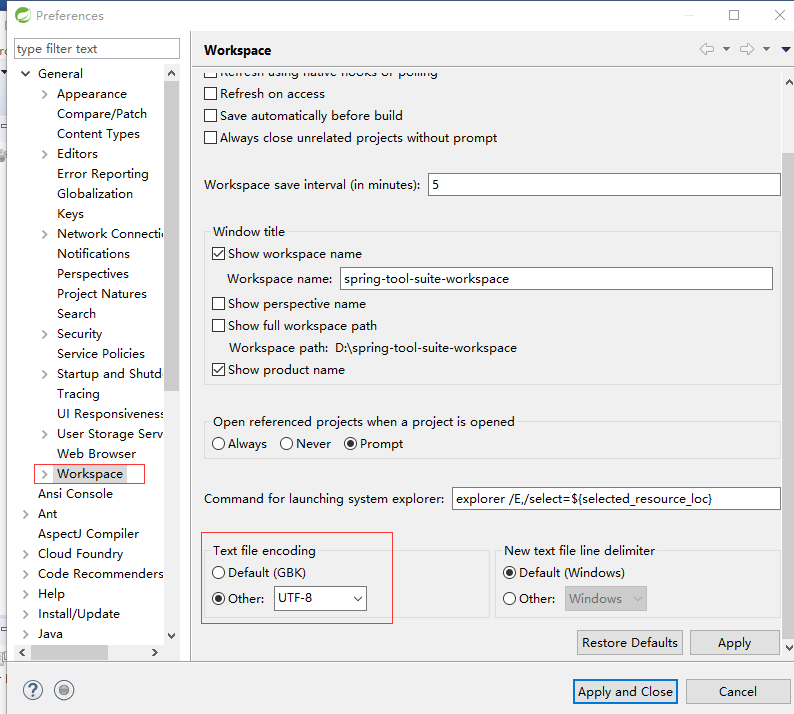


（图二）

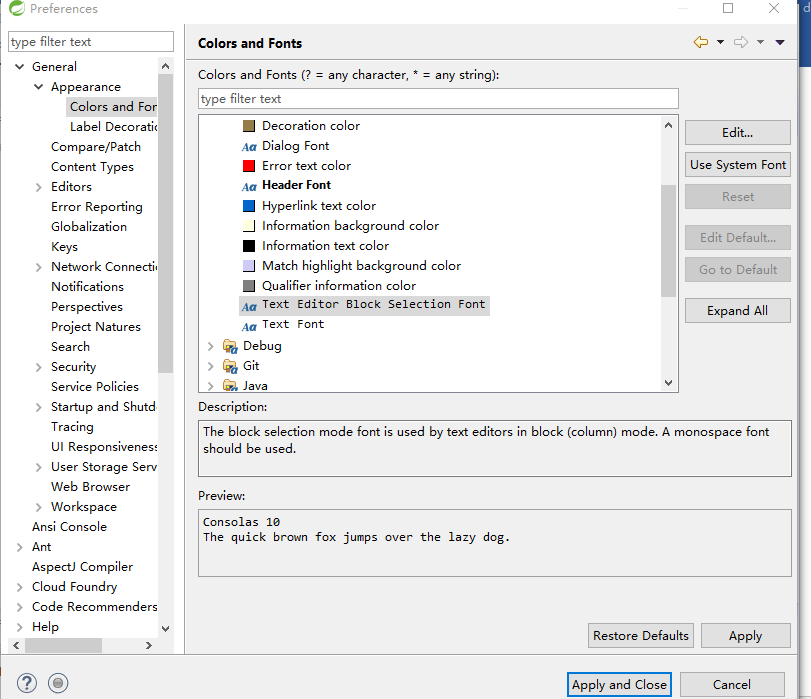
4、修改工作区编码和字体大小，点击菜单栏的Window->Preferences（如下图图一），在左边栏展开General,找到WorkSpace（不展开），把右边text file encoding改为Other(UTF-8,如下图二所示)，字体大小在Appearance(展开后)下面的Colors and Fonts。（这里示范修改了Basic的Text font和Text Editor Block Selection Font为13号字体字如下图三所示）



（图一）



(图二)

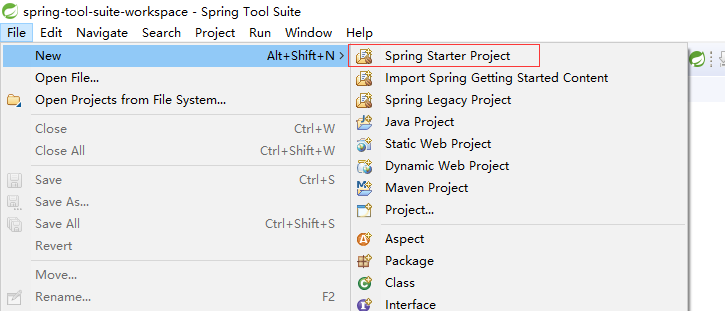


(图三)

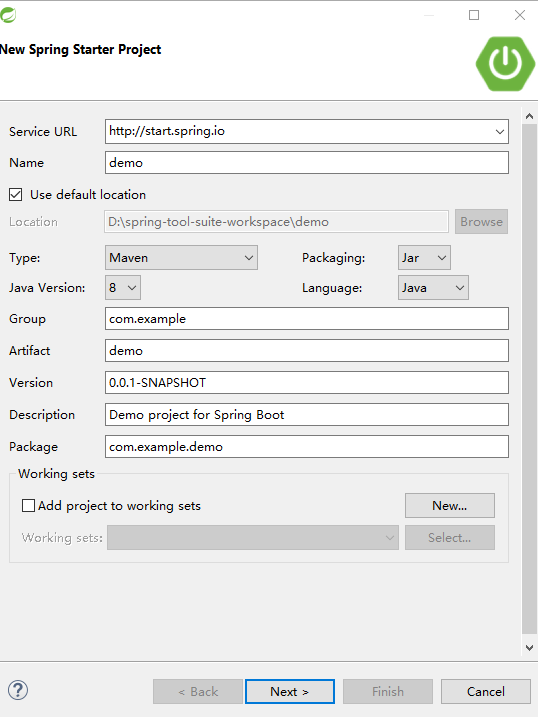
## Spring Boot工程创建

### 创建工程

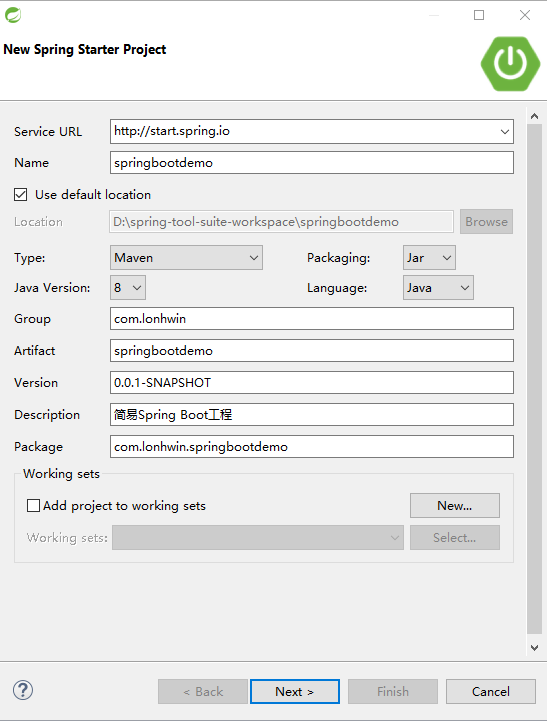
1、点击菜单栏的File->New->Spring Starter Project(如下面一所示)，会弹出如图二所示界面，其中Service URL为模板服务器地址，此处保留默认即可，Name为工程名称，Type是依赖包管理方式（这里有Gradle和Maven，这里推荐Maven），Packaging是打包的文件格式（JAR集成Tomcat可以直接运行，WAR包需要发布到服务器运行软件才能运行，这里说明以JAR），JAVA Version为JDK编译版本号（这里推荐JDK版本为8（内部版本为1.8））language为开发语言（这里有三个选项JAVA，Kotlin（JAVA衍生版本，可以编译为.class和.js），Groovy（JAVA衍生版本是动态语言）），我们在这里选择JAVA。Group为组（这里统一为com.lonwhin）,Artifact为工程目录名（这里推荐与工程名称保持一致）Version为本项目版本号，Description为工程描述信息，Package根据Group和Artifact组合而成（如下图三所示），修改完成后点击下一步



(图一)

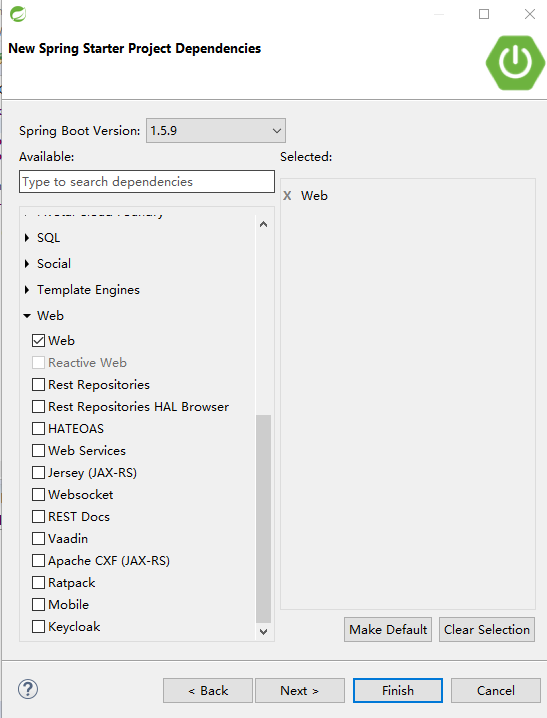


(图二)

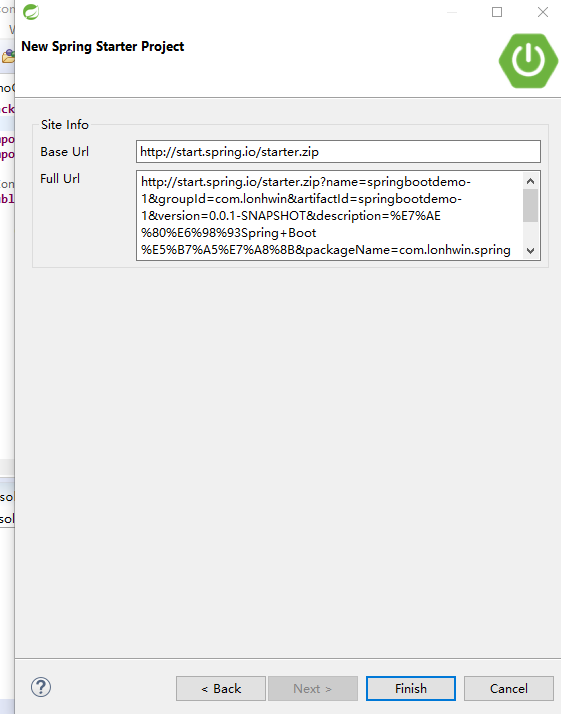


(图三)

2、选择需要的插件(如下图一所示)，Sping boot Version是Spring Boot版本号（这里以1.5.9做说明），这里只是简易创建工程，在左边Web里面选择Web然后点击点击Next（如下图二所示），然后点击Finish,等待下载完成

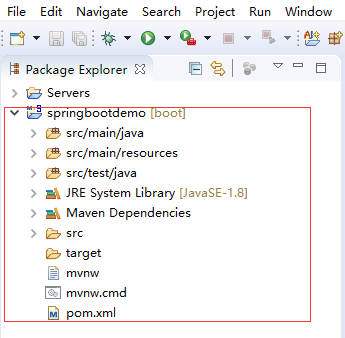


(图一)



(图二)

3、下载完成以后，左边工作区会多出一个自己创建的工程

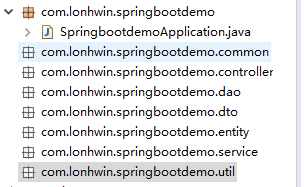


（这里简易说明一下：src/main/java是主要代码位置，src/main/resources为静态资源文件位置，src/test/java为测试代码位置）

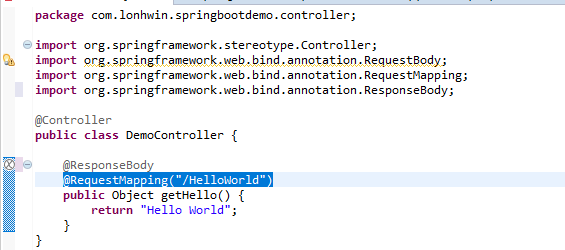
开始写方法

1、展开src/main/java,展开自己创建的包（com.lonhwin.springbootdemo）依次在下面建立几个包common、controller、entity、util、service、dao、dto(如下图所示)。

注意：SpringbootdemoApplication.java必须处com.lonhwin.springbootdemo下面，不能在下级子包里面（SpringBoot注解扫描只会扫描本级及其下级）



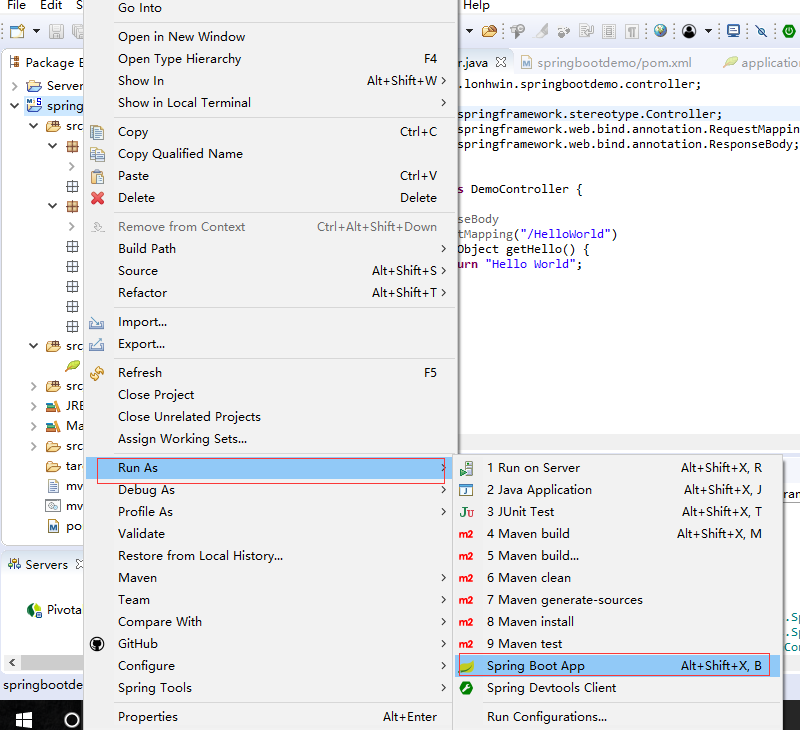
2、在controller下面建立一个类DemoController，在这个类上面使用注解@Controller，并在这个类里面写一个方法，并在这个方法上面使用注解@ResponseBody（返回JSON）和@RequestMapping("/HelloWorld")（绑定的访问地址）如下图所示。



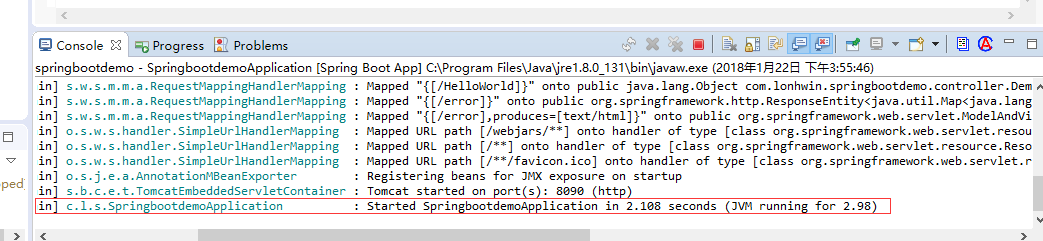
### 运行

1、一个基本的方法就写完，就可以启动起来看下，右键项目Run As->Spring Boot App（如图一）启动即可（Spring Boot默认为8080端口，发现项目路径为根路径，下面会讲述如何修改），等待加载完成以后用（如图二，出现

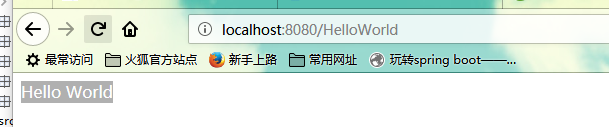
Started SpringbootdemoApplication in 2.108 seconds (JVM running for 2.98)就表示启动完成了）浏览器访问<http://localhost:8080/HelloWorld>即可看到返回的字符Hello World，如图三



（图一）、



（图二）

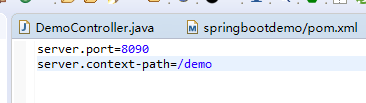


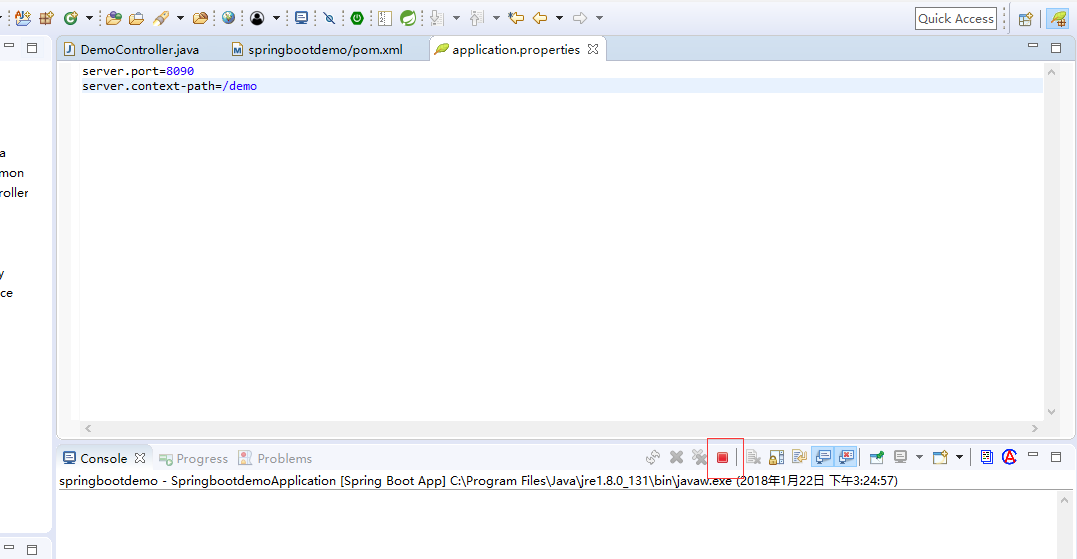
（图三）

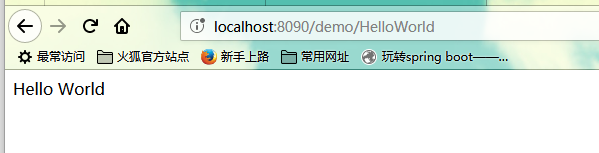
### 修改端口号和发布路径

SpringBoot的默认配置文件为application.properties（这个文件在src/main/resources下面），需要发布端口和发布路径都需要在这个文件进行配置(server.port为发布端口，server.context-path为发布路径，常用的配置会在后面附录说明)，按下图设置这两个参数

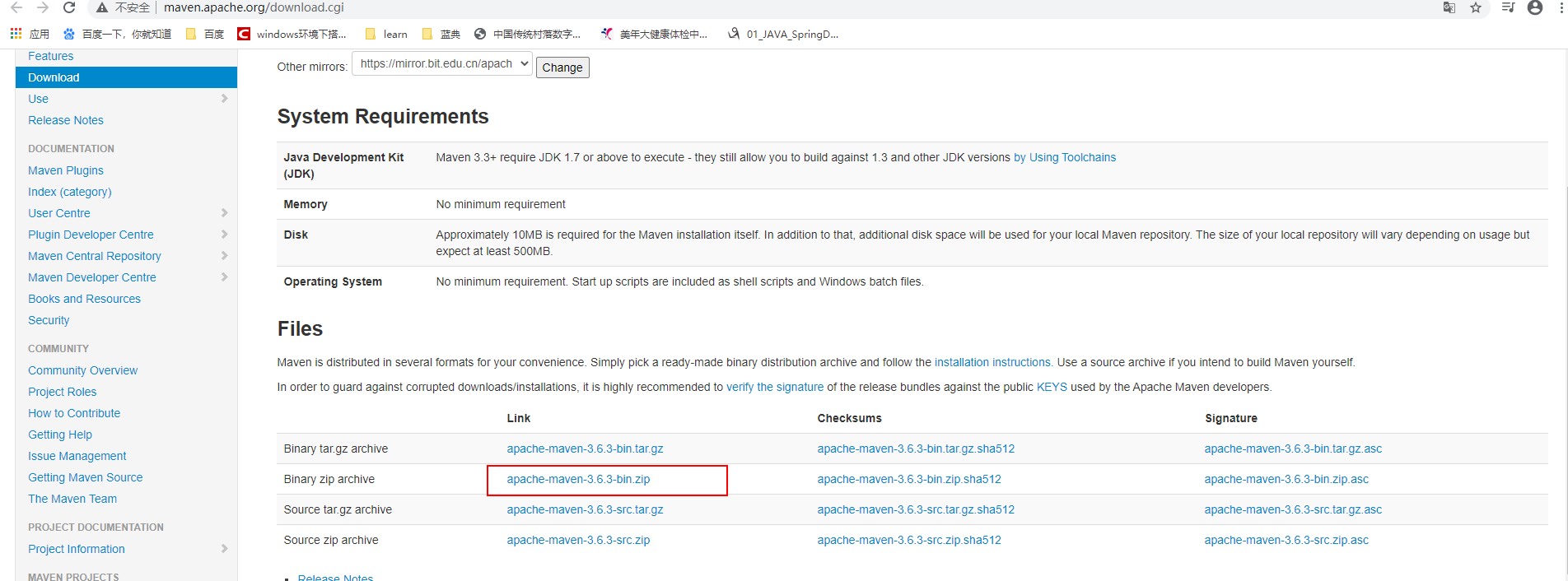
server.servlet.context-path=/demo （第二个的新写法）



然后点击右下角的红色方框停止当前运行的SpringBoot

再次右键项目Run As->Spring Boot App，等待加载完成，用浏览器访问<http://localhost:8090/demo/HelloWorld>即可看到和前面同意结果

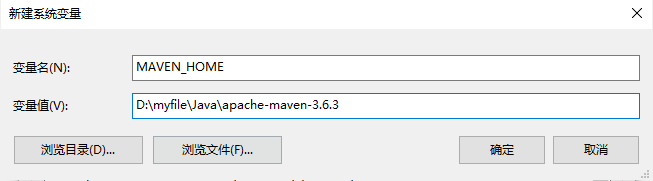
## Maven的本地安装

下载地址：<http://maven.apache.org/download.cgi>  


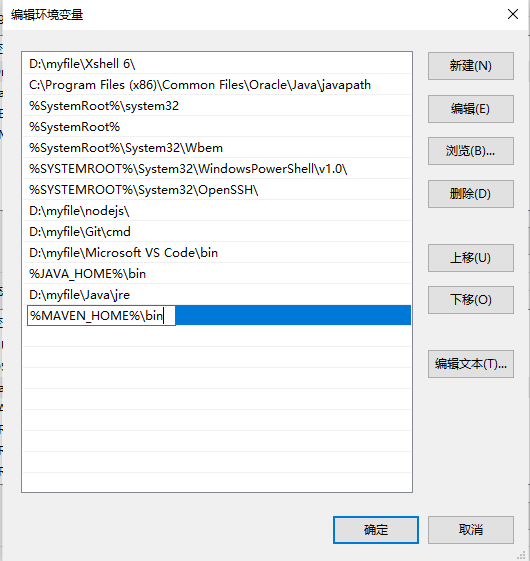
1. 下载并解压

解压即安装，这个不需要加入注册表

1. 检查jdk是否安装成功 java -version
2. 设置环境变量MAVEN\_HOME



1. 加到path里边 %MAVEN\_HOME%\bin



1. 重启cmd弹框，重新使用mvn -v查看

# Spring Boot 简易应用

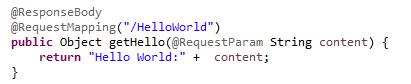
### Spring Boot访问规则

SpringBoot一个访问路径是由发布路径 + 类上面@RequestMapping的值（没有则为空） + 方法上面@RequestMapping的值来共同决定的，如同上面DemoController一样，由于在类上面并没有@RequestMapping所以访问路径为

Localhost + 8090（设定的端口号）+ /demo（项目发布名称）+ “”（类上面的RequestMapping的值） + /HelloWorld（方法上面的RequestMapping的值）?content=a(方法所需参数)

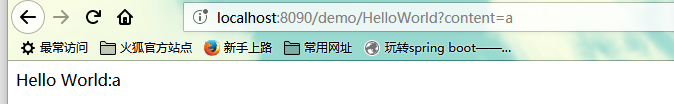
### 请求需要有参数

1、打开前面创建DemoController,在gethello这个方法增加一个String content并在这个参数前面加入@RequestParam并把return那一行进行修改return "Hello World:" + content; 如下图所示



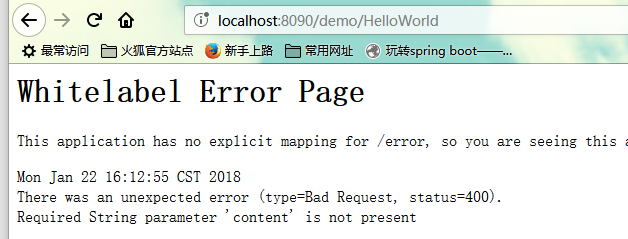
我们再次停止并重新启动这个项目，用浏览器访问

<http://localhost:8090/demo/HelloWorld?content=a>即可看到修改结果

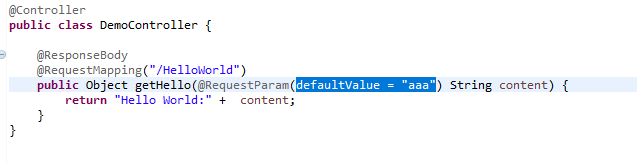


2、但是有些时候我们并不需要一定有content，这个时候如果没有这个参数，Spring Boot

会抛出缺少参数错误,那应该如何解决呢？

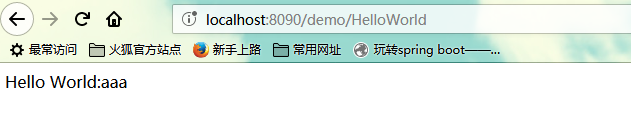


其实方法很简单修改@RequestParam String content为@RequestParam(defaultValue = "aaa") String content，RequestParam这个注解如果没加defaultValue，默认是必要参数且不能为空



然后再次重新启动项目，浏览器再次访问<http://localhost:8090/demo/HelloWorld>

就会发现没有参数也能正常访问



### 数据库的简易操作（CURD）

1、先打开工程根目录的pom.xml，在<dependencies> </dependencies>中间加入两个引用（这里数据库以Mysql为例）

*<!—JPA工具 -->*

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

*<!—数据库驱动 -->*

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

<!-- 添加Tomcat-JDBC依赖 新的spring boot需要這個 -->  
<dependency>  
 <groupId>org.apache.tomcat</groupId>  
 <artifactId>tomcat-jdbc</artifactId>  
</dependency>

2、打开application.properties配置以下参数

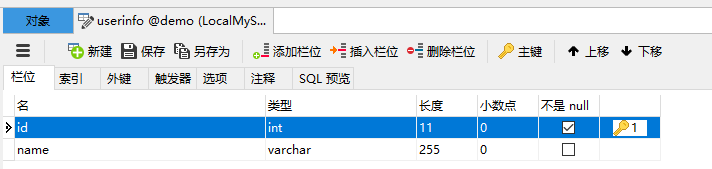
*# 数据库访问配置***spring.datasource.type**=**org.apache.tomcat.jdbc.pool.DataSource  
spring.datasource.url**=**jdbc:mysql://127.0.0.1:3306/demo  
spring.datasource.username**= **root  
spring.datasource.password**= **123456  
spring.datasource.driverClassName** = **com.mysql.jdbc.Driver**

*# 数据源配置***spring.datasource.initialSize**=**5  
spring.datasource.minIdle**=**5  
spring.datasource.maxActive**=**20****spring.datasource.maxWait**=**60000****spring.datasource.timeBetweenEvictionRunsMillis**=**60000****spring.datasource.minEvictableIdleTimeMillis**=**300000  
spring.datasource.validationQuery**=**select count(\*) from information\_schema.tables  
spring.datasource.testWhileIdle**=**true  
spring.datasource.testOnBorrow**=**false  
spring.datasource.testOnReturn**=**false**

*#JPA设置*

**spring.jpa.database**=*mysql***spring.jpa.show-sql**=**false  
spring.jpa.generate-ddl**=**true****spring.jpa.hibernate.naming\_strategy**=**org.hibernate.cfg.ImprovedNamingStrategy****spring.jpa.properties.hibernate.dialect**=**org.hibernate.dialect.MySQL5InnoDBDialect**

3、打开Mysql创建一个名称为demo（这里就不详细讲述demo建立过程），并在下面建立一张表userinfo，字段如下，并把id设置为自增。



4、在com.lonhwin.springbootdemo.entity上新建类Userinfo,具体内容如下

@Entity

**public** **class** Userinfo {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** **int** id;

**private** String name;

**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;

}

}

注:此处因为是自增主键，Userinfo实体类里面这句话@GeneratedValue(strategy = GenerationType.IDENTITY)(此处代表主键是自动生成)必须要，否则在新增保存的时候主键不会返回到实体类。下面会介绍数据库2种操作方法，选择其一即可

### JPA的CURD

1、在com.lonhwin.springbootdemo.dao下面建立接口类UserinfoRepository 并且继承JpaRepository,具体代码如下，Userinfo为前面的实体类,后面Integer为主键类型，因为前面表设置int所以这地方Integer或者int都可以

**public** **interface** UserinfoRepository **extends** JpaRepository<Userinfo, Integer>{

}

2、在com.lonhwin.springbootdemo.service下面建立UserinfoService，在这个类里面写数据库的CURD，具体代码如下

@Service

**public** **class** UserinfoService {

@Autowired

**private** UserinfoRepository userinfoRepository;

//增

**public** Userinfo add(Userinfo userinfo) {

**return** userinfoRepository.save(userinfo);

}

//删

**public** **void** delete(**int** id) {

userinfoRepository.deleteById(id);

}

//查

**public** List<Userinfo> findAll(){

**return** userinfoRepository.findAll();

}

//改

**public** Userinfo modity(Userinfo userinfo) {

**return** userinfoRepository.save(userinfo);

}

}

3、在com.lonhwin.springbootdemo.controller下面建立UserController用于http请求，具体代码如下

@Controller

@RequestMapping("/user")

**public** **class** UserController {

@Autowired

**private** UserinfoService userinfoservice;

@RequestMapping("/add")

@ResponseBody

**public** Object add(@RequestParam String name){

Userinfo userinfo = **new** Userinfo();

userinfo.setName(name);

**return** userinfoservice.add(userinfo);

}

@RequestMapping("/find")

@ResponseBody

**public** Object find() {

**return** userinfoservice.findAll();

}

@RequestMapping("/delete")

@ResponseBody

**public** Object delete(@RequestParam **int** id) {

userinfoservice.delete(id);

**return** "success";

}

@RequestMapping("/modify")

@ResponseBody

**public** Object modify(@RequestParam **int** id, @RequestParam String name) {

Userinfo userinfo = **new** Userinfo();

userinfo.setId(id);

userinfo.setName(name);

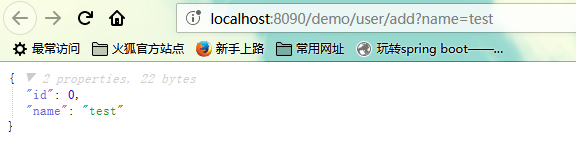
**return** userinfoservice.modity(userinfo);

}

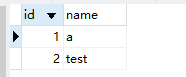
}

4、把项目启动 起来，打开浏览器和数据库连接工具查看操作结果是否成功

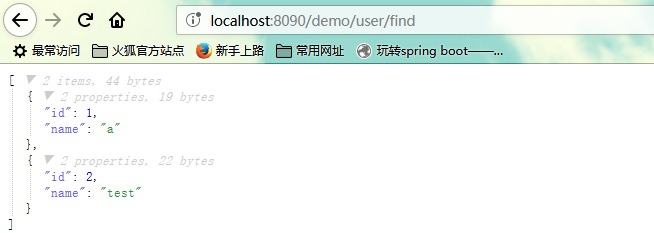
1)新增，访问[http://localhost:8090/demo/user/a](http://localhost:8090/demo/user/find)dd?name=test 可以看到如下返回



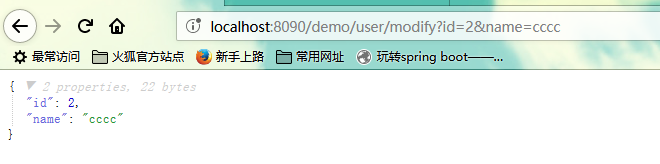
我们查看数据库表数据如图下，表示已经新成功



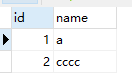
2)查：访问<http://localhost:8090/demo/user/find>，可以看到返回结果如下



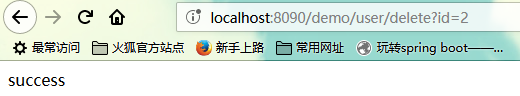
3)改；访问<http://localhost:8090/demo/user/modify?id=2&name=cccc>（原纪录id为2的name应该是test）返回如下图所示已经返回新的name了



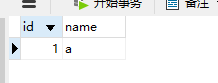
再打开数据看下记录是否正确，如下图所示记录已经正确修改



4)删：访问localhost:8090/demo/user/delete?id=2，可以看到返回success字样



对照数据，数据已经成功被删除



### EntityManager的CURD

1、在com.lonhwin.springbootdemo.dao下面建立UserinfoDao,并写入简易的数据操作方法，具体代码如下

@Repository

**public** **class** UserinfoDao {

@Autowired

**private** EntityManager em;

// 增

@Transactional //事务管理 保存、修改、删除时候必须开启事务管理

**public** Userinfo add(Userinfo userinfo) {

em.persist(userinfo);

**return** userinfo;

}

// 删

@Transactional //事务管理

**public** **void** delete(**int** id) {

Userinfo userinfo = em.find(Userinfo.**class**, id);

**if**(userinfo != **null**) {

em.remove(userinfo);

}

}

// 查

**public** List<Userinfo> findAll() {

String jpql = "select ui from Userinfo ui";

Query query = em.createQuery(jpql);

List<Userinfo> result = query.getResultList();

**return** result;

}

// 改

@Transactional //事务管理

**public** Userinfo modity(Userinfo userinfo) {

em.merge(userinfo);

**return** userinfo;

}

}

2、在com.lonhwin.springbootdemo.service下面建立UserinfoEmService，在这个类里面写数据库的CURD，具体代码如下

@Service

**public** **class** UserinfoEmService {

@Autowired

**private** UserinfoDao dao;

// 增

**public** Userinfo add(Userinfo userinfo) {

**return** dao.add(userinfo);

}

// 删

**public** **boolean** delete(**int** id) {

dao.delete(id);

**return** **true**;

}

// 查

**public** List<Userinfo> findAll() {

**return** dao.findAll();

}

// 改

**public** Userinfo modity(Userinfo userinfo) {

**return** dao.modity(userinfo);

}

}

3、在com.lonhwin.springbootdemo.controller下面建立UserEmController用于http请求，具体代码如下

@Controller

@RequestMapping("/userem")

**public** **class** UserEmController {

@Autowired

**private** UserinfoEmService userinfoemservice;

@RequestMapping("/add")

@ResponseBody

**public** Object add(@RequestParam String name){

Userinfo userinfo = **new** Userinfo();

userinfo.setName(name);

**return** userinfoemservice.add(userinfo);

}

@RequestMapping("/find")

@ResponseBody

**public** Object find() {

**return** userinfoemservice.findAll();

}

@RequestMapping("/delete")

@ResponseBody

**public** Object delete(@RequestParam **int** id) {

userinfoemservice.delete(id);

**return** "success";

}

@RequestMapping("/modify")

@ResponseBody

**public** Object modify(@RequestParam **int** id, @RequestParam String name) {

Userinfo userinfo = **new** Userinfo();

userinfo.setId(id);

userinfo.setName(name);

**return** userinfoemservice.modity(userinfo);

}

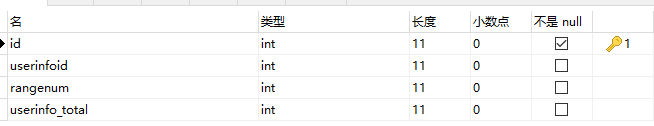
}

4、和上面JPA的一样，只是访问路径user变更为userem,就可以看到一样可以对数据看进行操作。

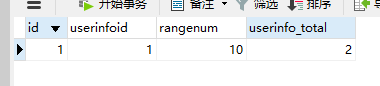
## Spring Boot JPA

### 多表查询

这里为了多表查询，在前面的demo数据库中，新建了一张userbasicinfo，具体字段如下



Id依然为自增类型，向其中插入1条数



1、在UserinfoRepository这个类里面加入下面一个方法，

@Query(value = "select ui.\*,ub.rangenum,ub.userinfo\_total from userinfo ui left join userbasicinfo ub on ub.userinfoid = ui.id", nativeQuery = **true**)

**public** List findListMultiTable();

其中NativeQuery是指是否用原生SQL查询

2、在UserinfoService这个类里面加入下面一个方法

**public** List findListMap(){

**return** userinfoRepository.findListMultiTable();

}

3、在UserController里面加入下面一个方法

@RequestMapping("/findListMap")

@ResponseBody

**public** Object findListMap() {

**return** userinfoservice.findListMap();

}

然后启动Spring Boot项目，访问<http://localhost:8090/demo/user/findListMap>就可以看到查询结果了

### 分页查询

1、在UserinfoService这个类里面加入下面一个方法（注：这里页面从0算起）

**public** Page findPage(**int** Num) {

Pageable page = **new** PageRequest(Num, 5);

**return** userinfoRepository.findAll(page);

}

2、在UserController里面加入下面一个方法

@RequestMapping("/findPage")

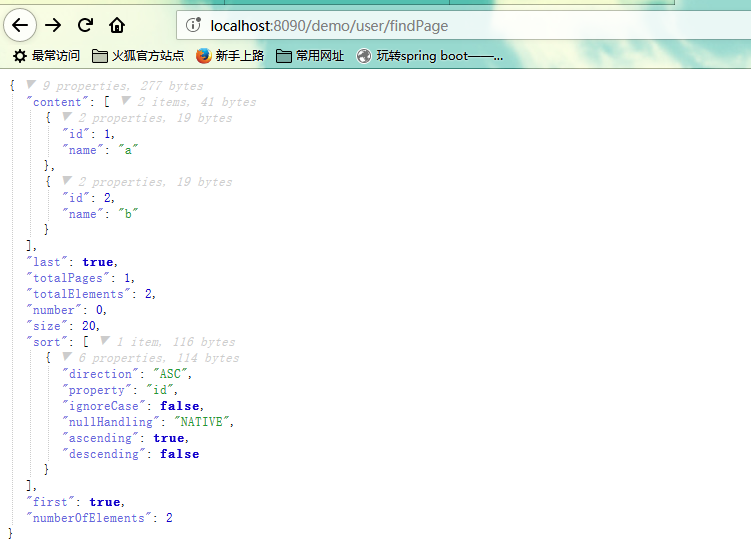
@ResponseBody

**public** Object findPage() {

**return** userinfoservice.findPage(1);

}

3、然后启动Spring Boot项目，访问http://localhost:8090/demo/user/findPage就可以看到查询结果了



### 条件查询

JPA条件查询由两个方法

方法一

在UserinfoRepository这样写即可实现参数方法

@Query(value="select ui from Userinfo ui where ui.name like %:name%")

**public** List<Userinfo> findByNameLike(@Param("name") String name);

在UserinfoService中

public List findByNameLike(String name) {  
 return userinfoRepository.findByNameLike(name);  
}

在Controller中

@ApiOperation("自定义SQL语句查列表带参数")  
@GetMapping("/findByNameLike")  
@ResponseBody  
public Object findByNameLike(@RequestParam String name) {  
 return userinfoservice.findByNameLike(name);  
}

方法二

//查

**public** List<Userinfo> findByLike(String name){

//创建查询条件数据对象

Userinfo userinfo = **new** Userinfo();

userinfo.setName("a");

//创建匹配器，即如何使用查询条件

ExampleMatcher matcher = ExampleMatcher.*matching*() //构建对象

.withMatcher("name", GenericPropertyMatchers.*contains*()); //姓名采用“包含匹配”的方式查询

//创建实例

Example<Userinfo> ex = Example.*of*(userinfo, matcher);

**return** userinfoRepository.findAll(ex);

}

## EntityManager（待优化）

### 创建分页实体类

在com.lonhwin.springbootdemo.dto建立PageList,代码如下

**public** **class** PageList<T> {

/\*\*

\* 记录总数

\*/

**private** **int** recordNumber;

/\*\*

\* 分页总数

\*/

**private** **int** pageNumber = 1;

/\*\*

\* 当前分页（从第1页开始）

\*/

**private** **int** pageIndex = 1;

/\*\*

\* 分页记录数

\*/

**private** **int** pageSize = 0;

/\*\*

\* 结果集

\*/

**private** List<?> dataList = **new** ArrayList<T>();

/\*\*

\* 获得当前分页

\*

\* **@return** 当前分页

\*/

**public** **int** getPageIndex() {

**return** **this**.pageIndex;

}

/\*\*

\* 设置当前分页

\*

\* **@param** pageIndex

\* 当前分页

\*/

**public** **void** setPageIndex(**int** pageIndex) {

**this**.pageIndex = pageIndex;

}

/\*\*

\* 获得当前分页记录数

\*

\* **@return** 当前分页记录数

\*/

**public** **int** getPageSize() {

**return** **this**.pageSize;

}

/\*\*

\* 设置当前分页记录数

\*

\* **@param** pageSize

\* 分页记录数

\*/

**public** **void** setPageSize(**int** pageSize) {

**this**.pageSize = pageSize;

}

/\*\*

\* 获得记录总数

\*

\* **@return** 记录总数

\*/

**public** **int** getRecordNumber() {

**return** **this**.recordNumber;

}

/\*\*

\* 设置记录总数

\*

\* **@param** recordNumber

\* 记录总数

\*/

**public** **void** setRecordNumber(**int** recordNumber) {

**this**.recordNumber = recordNumber;

}

/\*\*

\* 获得分页总数

\*

\* **@return** 分页总数

\*/

**public** **int** getPageNumber() {

**return** **this**.pageNumber;

}

/\*\*

\* 设置分页总数

\*

\* **@param** pageNumber

\* 分页总数

\*/

**public** **void** setPageNumber(**int** pageNumber) {

**this**.pageNumber = pageNumber;

}

/\*\*

\* 获得结果集

\*

\* **@return** 结果集

\*/

**public** List<?> getDataList() {

**return** dataList;

}

/\*\*

\* 设置结果集

\*

\* **@param** dataList

\* 结果集

\*/

**public** **void** setDataList(List<?> dataList) {

**this**.dataList = dataList;

}

}

### 创建基本操作方法

在com.lonhwin.springbootdemo.dao建立BaseDao，写入代码

**import** com.lonhwin.springbootdemo.dto.PageList;

**public** **class** BaseDao {

**public** PageList findPageData(EntityManager nem, Integer pageIndex,

Integer pageSize, String JPQL, Object... perpertys) {

**if** (nem == **null**) **return** **null**;

// nem = getEm();

Integer RecordNum = 0;

Integer PageNum = 0;

PageList<?> page = **new** PageList();

List datas = findPageList(nem, JPQL, pageSize, pageIndex, perpertys);

RecordNum = recordsNum(nem, JPQL, perpertys);

PageNum = RecordNum % pageSize == 0 ? RecordNum / pageSize : RecordNum

/ pageSize + 1;

page.setDataList(datas);

page.setPageNumber(PageNum);

page.setPageIndex(pageIndex);

page.setPageSize(pageSize);

page.setRecordNumber(RecordNum);

**return** page;

}

**public** List<?> findPageList(EntityManager nem, String JPQL, **int** max,

**int** whichpage, Object... perpertys) {

**if** (nem == **null**) **return** **null**;

// nem = getEm();

Query query = nem.createQuery(JPQL);

**if**(perpertys != **null**){

**for** (**int** i = 0; i < perpertys.length; i++) {

**if** (perpertys[i] **instanceof** Date) {

query.setParameter(i + 1, (Date) perpertys[i],

TemporalType.***TIMESTAMP***);

} **else** {

query.setParameter(i + 1, perpertys[i]);

}

}

}

**int** index = 0;

**if** (whichpage <= 1)

index = 0;

**else**

index = (whichpage - 1) \* max;

List<?> list = query.setMaxResults(max).// 设置分页最大记录数

setFirstResult(index).// 设置分页查询开始序号

getResultList();

nem.clear();// 分离内存中受EntityManager管理的实体bean，让VM进行垃圾回收

**return** list;

}

**public** Integer recordsNum(EntityManager nem, String JPQL,

Object... perpertys) {

**if** (nem == **null**) **return** **null**;

// nem = getEm();

Integer num = 0;

StringBuffer sb = **new** StringBuffer("select count(\*) ");

**if** (JPQL.indexOf(" from ") != -1)

sb.append(JPQL.substring(JPQL.indexOf(" from ")));

**else** **if** (JPQL.indexOf(" FROM ") != -1)

sb.append(JPQL.substring(JPQL.indexOf(" FROM ")));

JPQL = sb.toString();

**if** (JPQL.indexOf("order by") != -1)

JPQL = JPQL.substring(0, JPQL.indexOf("order by"));

**else** **if** (JPQL.indexOf("ORDER BY") != -1)

JPQL = JPQL.substring(0, JPQL.indexOf("ORDER BY"));

**try** {

Object objnum = findOneData(nem, JPQL, perpertys);

num = Integer.*valueOf*(objnum.toString());

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** num;

}

**public** Object findOneData(EntityManager nem, String JPQL,Object... perpertys) {

**if** (nem == **null**) **return** **null**;

// nem = getEm();

Query query = nem.createQuery(JPQL);

**if**(perpertys != **null**){

**for** (**int** i = 0; i < perpertys.length; i++) {

**if** (perpertys[i] **instanceof** Date) {

query.setParameter(i + 1, (Date) perpertys[i],

TemporalType.***TIMESTAMP***);

} **else** {

query.setParameter(i + 1, perpertys[i]);

}

}

}

List result=query.setMaxResults(1).getResultList();

**if**(result==**null** || result.size()==0) **return** **null**;

**else** **return** result.get(0);

}

}

public List findListDataNativeMap(EntityManager nem, String sql, int num,Object... perpertys){

if (nem == null)return null;

// nem = getEm();

Query query = nem.createNativeQuery(sql);

if (null == query)

return null;

//设置放回格式为MAP

query.unwrap(SQLQuery.class).setResultTransformer(Transformers.ALIAS\_TO\_ENTITY\_MAP);

if(perpertys != null){

for (int i = 0; i < perpertys.length; i++) {

if (perpertys[i] instanceof Date) {

query.setParameter(i + 1, (Date) perpertys[i],

TemporalType.TIMESTAMP);

} else {

query.setParameter(i + 1, perpertys[i]);

}

}

}

if (num <= 0)

return query.getResultList();

return query.setMaxResults(num).getResultList();

}

### 实现分页、多表、条件查询

1、修改UserinfoDao 继承 BaseDao类似 public class UserinfoDao extends BaseDao

2、然后在UserinfoDao就可以写以下来实现分页，多表等

//分页

**public** PageList findPage(**int** Page, **int** PageSize) {

String jpql = "select ui from Userinfo ui";

**return** findPageData(em, Page, PageSize, jpql);

}

//多表

**public** List findListMap() {

String sql = "select ui.\*,ub.rangenum,ub.userinfo\_total from userinfo ui "

+ "left join userbasicinfo ub on ub.userinfoid = ui.id";

**return** findListDataNativeMap(em, sql, 0);

}

//参数查询

**public** List findListByName(String name) {

String jpql = "select ui from Userinfo ui where ui.name like ?";

Query query = em.createQuery(jpql);

query.setParameter(0, name + "%");

List<Userinfo> result = query.getResultList();

**return** result;

}

3、然后依次在Service和Controller写对应方法后即可用Http访问

注:

此处方法是直接调用EntityManager来创建SQL，需要一定Hibernate基础知识

优点是可以不局限类，也可以不继承任何类

缺点是性能局限EntityManager提供的方法，需要自己制定优化。

## Sping Boot 定时任务

定时任务非常简单只需要SpringbootdemoApplication加入

@EnableScheduling *//此注解必须要 才能开始定时任务*

然后在需要执行定时任务的类上面@Component，然后在执行的方法上面@Scheduled(cron=**"0 0/5 \* \* \* \*"**)既可实现定时任务



注：

定时说明

*//@Scheduled(cron="0 10 \* \* \* \*")  
/\*\*  
 \* 定时任务并发处理(第一个到最后1个)  
 \* Seconds : 可出现", - \* /"四个字符，有效范围为0-59的整数  
 \* Minutes : 可出现", - \* /"四个字符，有效范围为0-59的整数  
 \* Hours : 可出现", - \* /"四个字符，有效范围为0-23的整数  
 \* DayofMonth : 可出现", - \* / ? L W C"八个字符，有效范围为0-31的整数  
 \* Month : 可出现", - \* /"四个字符，有效范围为1-12的整数或JAN-DEc  
 \* DayofWeek : 可出现", - \* / ? L C #"四个字符，有效范围为1-7的整数或SUN-SAT两个范围。1表示星期天，2表示星期一， 依次类推  
 \* Year : 可出现", - \* /"四个字符，有效范围为1970-2099年  
 \*/*

*//@Scheduled(fixedRate="1000\*1")  
//该属性的含义是上一个调用开始后再次调用的延时（不用等待上一次调用完成）*

*//@Scheduled(fixedDelay="1000\*1")  
//该属性的功效与上面的fixedRate则是相反的，配置了该属性后会等到方法执行完成后延迟配置的时间再次执行该方法。*

*//@Scheduled(initialDelay="1000\*10" fixedDelay="1000\*1")  
//该属性的作用是第一次执行延迟时间，只是做延迟的设定，并不会控制其他逻辑，所以要配合fixedDelay或者fixedRate来使用*

# 细节解释

## 注解

@SpringBootApplication

简单理解：主执行类上默认的

这个注解是Spring Boot最核心的注解，用在 Spring Boot的主类上，标识这是一个 Spring Boot 应用，用来开启 Spring Boot 的各项能力。实际上这个注解是@Configuration,@EnableAutoConfiguration,@ComponentScan三个注解的组合。由于这些注解一般都是一起使用，所以Spring Boot提供了一个统一的注解@SpringBootApplication。

@Controller

简单理解：申明是controller层

用于标记在一个类上，使用它标记的类就是一个SpringMvc Controller对象，分发处理器会扫描使用该注解的类的方法，并检测该方法是否使用了@RequestMapping注解

[@RestController](http://www.baidu.com/link?url=E4T9l9RE7JqLRbkSw7JivlmHWbveUNI6T4065ugauEMW5bpG5EKf3HEPs0js_EeXFz4kHftzjk6_ojM-zlTgn3rpuP2AgsczonfunpKjHdG" \t "https://www.baidu.com/_blank)（以后用这个即可）

在Spring中@RestController的作用等同于@Controller + @ResponseBody。

　　所以想要理解@RestController注解就要先了解@Controller和@ResponseBody注解。

@ResponseBody

简单理解：返回json格式数据

将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据。  
注意：在使用此注解之后不会再走视图处理器，而是直接将数据写入到输入流中，他的效果等同于通过response对象输出指定格式的数据。

@ResponseBody是作用在方法上的，@ResponseBody 表示该方法的返回结果直接写入 HTTP response body 中，一般在异步获取数据时使用【也就是AJAX】。  
注意：在使用 @RequestMapping后，返回值通常解析为跳转路径，但是加上 @ResponseBody 后返回结果不会被解析为跳转路径，而是直接写入 HTTP response body 中。 比如异步获取 json 数据，加上 @ResponseBody 后，会直接返回 json 数据。@RequestBody 将 HTTP 请求正文插入方法中，使用适合的 HttpMessageConverter 将请求体写入某个对象

@RequestMapping("/HelloWorld")

简单理解：请求地址

在对 Spring MVC 进行的配置的时候, 你需要指定请求与处理方法之间的映射关系。

@GetMapping

等于@RequestMapping(method = RequestMethod.GET)

@PostMapping

等于@RequestMapping(method = RequestMethod.POST)

@PutMapping

@DeleteMapping

@PatchMapping

## Pom.xml 详解

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <!--继承springboot的父级项目的依赖-->  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.3.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如com.winner.trade，maven会将该项目打成的jar包放本地路径：/com/winner/trade -->  
 <groupId>com.example</groupId>  
 <!-- 本项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->  
 <artifactId>demo2</artifactId>  
 <!-- 本项目目前所处的版本号 -->  
 <version>0.0.1-SNAPSHOT</version>  
 <name>demo2</name>  
 <description>Demo project for Spring Boot</description>  
 <!-- 为pom定义一些常量，属性配置 -->  
 <properties>  
 <java.version>1.8</java.version>  
 <!--<file.encoding>UTF-8</file.encoding>-->  
 <!--<java.source.version>1.8</java.source.version>-->  
 <!--<java.target.version>1.5</java.target.version>-->  
 </properties>  
 <!-- 定义本项目的依赖关系 -->  
 <dependencies>  
 <!-- 每个dependency都对应这一个jar包 -->  
 <dependency>  
 <!--一般情况下，maven是通过groupId、artifactId、version这三个元素值（俗称坐标）来检索该构件， 然后引入你的工程。如果别人想引用你现在开发的这个项目（前提是已开发完毕并发布到了远程仓库），-->  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 <!--这个没有版本号是因为已经继承了父级的版本号-->  
  
 <!--scope包含如下的取值：compile（编译范围）、provided（已提供范围）、runtime（运行时范围）、test（测试范围）、system（系统范围） -->  
 <!--<scope>test</scope> -->  
  
 <!-- 设置指依赖是否可选，默认为false,即子项目默认都继承:为true,则子项目必需显示的引入，与dependencyManagement里定义的依赖类似 -->  
 <!--<optional>false</optional>-->  
 </dependency>  
  
 <!--这个是一个测试的起步依赖-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <!--springboot提供的编译打包插件-->  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

application.properties常用配置

*# 发布端口*

**server.port**=**8080**

*# 发布路径*

server.context-path   
*# 数据库访问配置*

*# 数据源类型***spring.datasource.type   
spring.datasource.url**

**spring.datasource.username   
spring.datasource.password   
spring.datasource.driverClassName***# 下面为连接池的补充设置，应用到上面所有数据源中  
# 初始化大小，最小，最大***spring.datasource.initialSize  
spring.datasource.minIdle  
spring.datasource.maxActive***# 配置获取连接等待超时的时间***spring.datasource.maxWait***# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒***spring.datasource.timeBetweenEvictionRunsMillis***# 配置一个连接在池中最小生存的时间，单位是毫秒***spring.datasource.minEvictableIdleTimeMillis  
spring.datasource.validationQuery   
spring.datasource.testWhileIdle   
spring.datasource.testOnBorrow   
spring.datasource.testOnReturn***# 打开PSCache，并且指定每个连接上PSCache的大小***spring.datasource.poolPreparedStatements   
spring.datasource.maxPoolPreparedStatementPerConnectionSize***# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙***spring.datasource.filters**

*# 通过connectProperties属性来打开mergeSql功能；慢SQL记录***spring.datasource.connectionProperties****spring.datasource.useGlobalDataSourceStat** *#JPA配置:***spring.jpa.database****spring.jpa.show-sql  
spring.jpa.generate-ddl****spring.jpa.hibernate.naming\_strategy****spring.jpa.properties.hibernate.dialect***#thymeleaf配置***spring.thymeleaf.mode   
spring.thymeleaf.encoding  
spring.thymeleaf.content-type   
spring.thymeleaf.prefix**

*#freemarker配置***:  
spring.freemarker.suffix:** .html  
**spring.freemarker.charset:** utf-8  
**spring.freemarker.content-type:** text/html;charset=utf-8  
**spring.freemarker.expose-session-attributes: true  
spring.freemarker.expose-request-attributes: true  
spring.freemarker.expose-spring-macro-helpers: true  
spring.freemarker.request-context-attribute:** request

*#开发时关闭缓存,不然没法看到实时页面***spring.thymeleaf.cache**  *#Redis配置***spring.redis.host  
spring.redis.port  
spring.redis.password**  
**spring.redis.database  
spring.redis.timeout  
spring.redis.pool.max-active  
spring.redis.pool.max-wait  
spring.redis.pool.max-idle  
spring.redis.pool.min-idle**

## application详解

### 核心配置文件及两种格式

application.properties 这个为正常格式，

例如server.port=8090

application.yml

server:  
 port: 8090

两者同时存在，执行application.properties

### 多环境配置文件

创建三个文件

application-dev.properties

application-test.properties

application-online.properties

application.properties文件中

spring.profiles.active=dev

这个代表使用dev环境，使用哪个=号后边是-后边的名称

如果application.properties中配置了端口，激活文件中也配置了端口，那application.properties中的端口无论放在哪里都是无效的，如果激活文件中没有，那application.properties中的有效

### 自定义注解

application.properties中定义变量，不需要引号一类

boot.name=大家好啊啊啊

@Value("${boot.name}")  
private String aa;

方法中将自定义的值赋值在新定义的变量中，后边可直接使用这个变量

## 引入jsp

### Pom.xml中 引入相关jar包

<!--servlet以来的jar包-->  
<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>javax.servlet-api</artifactId>  
</dependency>  
<!--jsp依赖的jar包-->  
<dependency>  
 <groupId>javax.servlet.jsp</groupId>  
 <artifactId>javax.servlet.jsp-api</artifactId>  
 <version>2.3.1</version>  
</dependency>  
<!--jstl标签-->  
<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
</dependency>

如果访问jsp文件会变成下载

<!--访问jsp文件会变成下载-->  
<dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-jasper</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.apache.tomcat</groupId>  
 <artifactId>tomcat-jsp-api</artifactId>  
</dependency>

还需要在build 里边加resources

<build>  
 <plugins>  
 <!--springboot提供的编译打包插件-->  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 <resource>  
 <directory>src/main/resources</directory>  
 <includes>  
 <include>\*\*/\*.</include>  
 </includes>  
 </resource>  
 <resource>  
 <directory>src/main/webapp</directory>  
 <targetPath>META-INF/resoureces</targetPath>  
 <includes>  
 <include>\*\*/\*.</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

### application.properties中 配置

前端视图展示为jsp

spring.mvc.view.prefix=/  
spring.mvc.view.suffix=.jsp

如果乱码

server.tomcat.uri-encoding=utf-8  
server.servlet.encoding.force=**true**server.servlet.encoding.enabled=**true**server.servlet.encoding.charset=utf-8

### 创建index.jsp

与resources同层级，建立webapp文件夹，里边创建index.jsp

<%@ **page** contentType="text/html;charset=utf-8"%>  
<%--上边这句可以解决页面乱码--%>  
<html>  
<meta charset="utf-8" />  
<head>  
  
 <title>第一个 JSP 程序</title>  
</head>  
<body>  
**${**msg**}**<h3>一行 Java 代码</h3>  
<p>  
 今天的日期是: **<%=**(new java.util.Date())**%>**</p>  
<h3>多行 Java 代码</h3>  
<p>  
 你的 IP 地址是：  
 **<%** out.println("Your IP address is " + request.getRemoteAddr()+"</br>");  
 out.println("一段代码 ");  
 **%>**</p>  
</body>  
</html>

### 设置controller

Controller文件夹下创建文件jspController

@Controller  
public class jspController {  
 @GetMapping("/haha")  
 public String index (Model model) {  
 model.addAttribute("msg","spring boot 集成jsp");  
 return "index";  
 }  
}

## 引入mybatis ★

### pom.xml文件的配置

引入mybatis ，对应的数据库，如果之前没有引入，还需要spring-boot-starter-web

<!--引入mybatis-->  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.1</version>  
</dependency>  
<!--引入mysql-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

<!--引入web包-->

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

加入mybatis代码自动生成插件

<build>  
 <plugins>  
 <!--加入mabatis代码自动生成插件-->  
 <plugin>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-maven-plugin</artifactId>  
 <version>1.3.7</version>  
 <configuration>  
 <!--配置文件的位置 todo-->  
 <configurationFile>mapper.xml</configurationFile>  
 <verbose>true</verbose>  
 <overwrite>true</overwrite>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

下边这段是解决idea里运行不能生成.xml文件等的，在这个方法里很奇怪，如果加入resources会报错

有时候加下边这些运行mybatis代码自动生成工具会报错，可以暂时去掉，但运行项目的时候，不加又不能生成.xml文件，又得加上

<resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>

!--加入这个会报错-->  
<!--<resource>-->  
 <!--<directory>src/main/resources</directory>-->  
 <!--<includes>-->  
 <!--<include>\*\*/\*.</include>-->  
 <!--</includes>-->  
<!--</resource>-->  
 <resource>  
 <directory>src/main/webapp</directory>  
 <targetPath>META-INF/resoureces</targetPath>  
 <includes>  
 <include>\*\*/\*.</include>  
 </includes>  
 </resource>  
</resources>

### application.properties文件中配置

配置mybatis的Mapper.xml文件的位置

配置数据库连接

#配置mapper文件  
mybatis.mapper-locations=classpath:com/example/demo2/mapper/\*.xml  
  
# 数据库配置  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.datasource.url=jdbc:mysql://49.4.88.81:13306/test?useSSL=false  
spring.datasource.username= root  
spring.datasource.password= 123456

### 编写mapper.xml

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE generatorConfiguration  
 PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  
 "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  
<generatorConfiguration>  
 <properties resource="application.properties"></properties>  
 <!-- 数据库驱动:选择你的本地硬盘上面的数据库驱动包的地址 要改-->  
 <classPathEntry location="D:\myapp\jar\mysql-connector-java-8.0.12.jar"/>  
 <!--配置demotable表信息内容体，targetRuntime指定采用MyBatis3版本-->  
 <context id="demotable" targetRuntime="MyBatis3">  
 <!--注释生成器-->  
 <commentGenerator>  
 <!--阻止生成的注释包含时间戳，默认为false-->  
 <property name="suppressDate" value="true"/>  
 <!-- 是否去除自动生成的注释 true：是 ： false:否，因为注释是英文的，一般选true -->  
 <property name="suppressAllComments" value="true"/>  
 </commentGenerator>  
 <!--配置数据库连接信息 *todo*-->  
 <jdbcConnection  
 driverClass="com.mysql.cj.jdbc.Driver"  
 connectionURL="${spring.datasource.url}"  
 userId="${spring.datasource.username}"  
 password="${spring.datasource.password}">  
 </jdbcConnection>  
 <!--java类型解析器 非必需，类型处理器，在数据库类型和java类型之间的转换控制-->  
 <javaTypeResolver>  
 <property name="forceBigDecimals" value="false"/>  
 </javaTypeResolver>  
 <!-- 生成实体类的包名和位置 targetPackage指定model类的包名 *todo*-->  
 <javaModelGenerator  
 targetPackage="com.example.mybatis.entity"  
 targetProject="src/main/java">  
 <property name="enableSubPackages" value="true"/>  
 <property name="trimStrings" value="true"/>  
 </javaModelGenerator>  
 <!-- 生成sql映射文件的包名和位置 targetPackage指定mapper.xml文件的包名 targetProject指定mapper.xml放在哪个位置 *todo*-->  
 <sqlMapGenerator  
 targetPackage="com.example.mybatis.mapper"  
 targetProject="src/main/java">  
 <property name="enableSubPackages" value="true"/>  
 </sqlMapGenerator>  
 <!-- 生成mapper接口的包名和位置 要改-->  
 <javaClientGenerator  
 type="XMLMAPPER"  
 targetPackage="com.example.mybatis.mapper"  
 targetProject="src/main/java">  
 <property name="enableSubPackages" value="true"/>  
 </javaClientGenerator>  
  
 <!-- 要生成的表 tableName是数据库中的表名或视图名 domainObjectName是实体类名 要改-->  
 <table  
 tableName="userinfo"  
 domainObjectName="Userinfo"  
 enableCountByExample="false"  
 enableUpdateByExample="false"  
 enableDeleteByExample="false"  
 enableSelectByExample="false"  
 selectByExampleQueryId="false">  
  
 </table>  
 <!--<table-->  
 <!--tableName="meeting\_details\_tab"-->  
 <!--domainObjectName="MeetingDetailsTab"-->  
 <!--enableCountByExample="false"-->  
 <!--enableUpdateByExample="false"-->  
 <!--enableDeleteByExample="false"-->  
 <!--enableSelectByExample="false"-->  
 <!--selectByExampleQueryId="false">-->  
  
 <!--</table>-->  
 </context>  
</generatorConfiguration>

mysql-connector-java

下载地址

<https://repo1.maven.org/maven2/mysql/mysql-connector-java/>

各自有个版本对应，mysql8 就应8以上版本 mysql-connector-java-8.0.12

### 使用代码生成工具生成代码org.mybatis.generator

### 

双击就会自动生成实体类，mapper文件，xml

如果配置了mvn ，也可以直接运行 mvn mybatis-generator:generate

也就相当于生成了dao层和entity层

添加一个查询所有数据的方法

1.在生成的Mapper 接口中，定义一个方法

List<Userinfo> selectAll();

1. Mapper.xml中实现这个方法

<select id="selectAll" resultMap="BaseResultMap">  
 select  
 <include refid="Base\_Column\_List" />  
 from userinfo  
</select>

### server层及controller层的编写

server

@Service  
public class UserService {  
 @Autowired  
 private UserinfoMapper userinfoMapper;  
 public List<Userinfo> haha() {  
 return userinfoMapper.selectAll();  
 }  
}

contorller

@RestController  
public class MybatisController {  
 @Autowired  
 private UserService userService;  
 @GetMapping("/getAllUserinfo")  
 public Object userinfo () {  
 return userService.haha();  
 }  
}

### 加入Mapper注解即可运行

在mapper接口中添加@Mapper注解

[或者在运行的主类上添加@MapperScan(“com.example.demo2.mapper”)](mailto:或者在运行的主类上添加@MapperScan(\“com.example.demo2.mapper\”))

### Mybatis的详细使用

## 引入事务

在主方法上开始springboot事务支持

@EnableTransactionManagement

service层的具体方法上

@Transactional

如果加入了事务，当出现错误的时候会回调，返回之前的状态

# 插件的使用

## Swagger

【application.properties 文件 添加maven 依赖】

<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.9.2</version>  
</dependency>  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>2.9.2</version>  
</dependency>

【添加swagger配置类】

创建package config，在里边新建类 SwaggerConfig

注意里边的basePackage必须是当前项目的

package com.example.demo.config;  
  
import org.springframework.context.annotation.Bean;  
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.Contact;  
import springfox.documentation.spi.DocumentationType;  
import springfox.documentation.spring.web.plugins.Docket;  
import springfox.documentation.swagger2.annotations.EnableSwagger2;  
  
@Configuration  
@EnableSwagger2  
public class SwaggerConfig {  
 @Bean  
 public Docket createRestApi() {  
 return new Docket(DocumentationType.SWAGGER\_2)  
 .pathMapping("/")  
 .select()  
 .apis(RequestHandlerSelectors.basePackage("com.example.demo"))   
 .paths(PathSelectors.any())  
 .build().apiInfo(new ApiInfoBuilder()  
 .title("SpringBoot整合Swagger")  
 .description("SpringBoot整合Swagger，详细信息......")  
 .version("9.0")  
 .contact(new Contact("啊啊啊啊","blog.csdn.net","aaa@gmail.com"))  
 .license("The Apache License")  
 .licenseUrl("http://www.baidu.com")  
 .build());  
 }  
}

【运行】

<http://localhost:8080/swagger-ui.html>

【注解】

| **作用范围** | **API** | **使用位置** |
| --- | --- | --- |
| 对象属性 | @ApiModelProperty | 用在出入参数对象的字段上 |
| 协议集描述 | @Api | 用于controller类上 |
| 协议描述 | @ApiOperation | 用在controller的方法上 |
| Response集 | @ApiResponses | 用在controller的方法上 |
| Response | @ApiResponse | 用在 @ApiResponses里边 |
| 非对象参数集 | @ApiImplicitParams | 用在controller的方法上 |
| 非对象参数描述 | @ApiImplicitParam | 用在@ApiImplicitParams的方法里边 |
| 描述返回对象的意义 | @ApiModel | 用在返回对象类上 |

@Api注解可以用来标记当前Controller的功能。class上

@ApiOperation注解用来标记一个方法的作用。方法上

@ApiImplicitParam注解用来描述一个参数，可以配置参数的中文含义，也可以给参数设置默认值，这样在接口测试的时候可以避免手动输入。

如果有多个参数，则需要使用多个@ApiImplicitParam注解来描述，多个@ApiImplicitParam注解需要放在一个@ApiImplicitParams注解中。

需要注意的是，@ApiImplicitParam注解中虽然可以指定参数是必填的，但是却不能代替@RequestParam(required = true)，前者的必填只是在Swagger2框架内必填，抛弃了Swagger2，这个限制就没用了，所以假如开发者需要指定一个参数必填，@RequestParam(required = true)注解还是不能省略。

如果参数是一个对象（例如上文的更新接口），对于参数的描述也可以放在实体类中。例如下面一段代码：

@ApiModel

简单解释：用在实体类前边

@ApiModelProperty(value = "用户id")

简单解释：用在实体类的字段前边，解释这个字段

# 语法进阶

## class 及 interface 和它的实现implements

Class 原本写法

public class UserService {  
 @Autowired  
 private UserinfoMapper userinfoMapper;  
 public List<Userinfo> haha() {  
 return userinfoMapper.selectAll();  
 }  
}

接口类

public interface UserinfoService {  
 public List<Userinfo> getAllUserinfo();  
 public Userinfo selectByPrimaryKey();  
}

接口类的实现

public class UserinfoServiceImpl implements UserinfoService {  
  
 @Autowired  
 private UserinfoMapper userinfoMapper;  
  
 @Override  
 public List<Userinfo> getAllUserinfo() {  
 return userinfoMapper.selectAll();  
 }  
  
 public Userinfo selectByPrimaryKey() {  
 return userinfoMapper.selectByPrimaryKey(2);  
 }  
}

# 可能会出现的问题

## 如何pom.xml 报错

项目-右键-Maven-Update Project-Force Update of Snapshots/Releases

更新依赖包

## 连数据库

**spring.datasource.type**=**com.zaxxer.hikari.HikariDataSource  
spring.datasource.url**=**jdbc:mysql://127.0.0.1:3306/world?useSSL=true  
spring.datasource.username**= **root  
spring.datasource.password**= **root  
spring.datasource.driverClassName** = **com.mysql.jdbc.Driver  
  
spring.jpa.database=mysql**

**spring.jpa.generate-ddl=false**

**spring.data.jpa.repositories.enabled=true**

当mysql版本大于5.6的时候 加上 **?useSSL=true**

## pom.xml 加入jpa和mysql

Springboot 2的这个版本 jpa 弄2.0.0

，在<parent> </parent>

<version>2.0.0.RELEASE</version>

，在<dependencies> </dependencies>中间加入两个引用

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

## 删除方法

public void delete(int id) {

userinfoRepository.delete(id);

}

改为

userinfoRepository.deleteById(id);

## 修改编码

软件左上角file - settings

