# 1. @RestController

## 解释：这个类要是想通过web的方式访问就要加一个 注解

# *2.* @RequestMapping

## 解释：

## 1请求的方式的GET，POST,

@RequestMapping(value = **"/hello"** ,method = RequestMethod.***GET***)

## 2同时也可以指定传输给前台的数据的文本格式是json xml 或者text 返回对象是json xml或者String是 text

@RequestMapping(value = **"/hello"** ,method = RequestMethod.***GET,produces="application/json:charst="utf-8"***)

@RequestMapping(value = **"/hello"** ,method = RequestMethod.***GET,produces=" application /xml:charst="utf-8"***)

@RequestMapping(value = **"/hello"** ,method = RequestMethod.***GET,produces="text/plain:charst="utf-8"***)

## 3.服务器端推送技术的时候，里面的格式是 produces="text/event-stream" 这是服务器端sse的支持。

# 3. @Value("${age}")

## 解释; 利用配置文件进行注入@Value("${age}") 注解进行注入





# 4. @Component,一般与@Autowired使用

# 解释:/把普通pojo实例化到spring容器中，相当于配置文件中的<bean id="" class="">

# 5. @ConfigurationProperties

## 解释：//识别配置文件的前缀prefix，可以通过一个注解ConfigurationProperties然后根据属性进行注入 @ConfigurationProperties(prefix = "person")



# 6@Autowired

## 解释：也就是不需要new这个对象了，省略 get/setter 直接进行使用



# 7.@ controller

## 解释：spring mvc中用到的控制器，可以用来跳转html和其他的页面

# 9. @PathVariable(可以帮助自动验证是不是uuid，这样就保障了，进入的id是正确的)

## 解释：用来接收参数的，一般是id，可以直接，地址/id，或者id=1;



# 10, RequestParam (+2017.5.23里面可以没有参数，比如接受uuid时候@RequestParam UUID[] id)

## 解释：获取请求参数的值（浏览器必须说明字段id=1）



## 2.设置默认值 required=false （可以不传值），默认为0



# 11. getMapping

## 解释：组合注解



# 12. @Entity

## [解释： @Entity](mailto:1.@Entity) 是表示这个类对应数据库中的一个表

# 13. [@id](mailto:2.@id) @GeneratedValue @NotEmpty

## 解释：是表示这个id是 主键自增的（稍微有点不懂 @id）



# 14. @service

## 解释：一个service服务类，使用它的时候要记得使用@Autowired





# 15. @Transactional

## 1.解释：使用数据库的时候，防止出现错误依旧存入其他数据



## 2.用法 @Transactional(readOnly = true)

### 1解释：从这一点设置的时间点开始（时间点a）到这个事务结束的过程中，其他事务所提交的数据，该事务将看不见！（查询中不会出现别人在时间点a之后提交的数据）

### 2应用场合：

1.如果你一次执行单条查询语句，则没有必要启用事务支持，数据库默认支.持SQL执行期间的读一致性；   
2 如果你一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询SQL必须保证整体的读一致性，否则，在前条SQL查询之后，后条SQL查询之前，数据被其他用户改变，则该次整体的统计查询将会出现读数据不一致的状态，此时，应该启用事务支持。

# 16 @ResponseBody

## 解释：@responseBody注解的作用是将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，可以加载方法的内部（方法名），效果是一样的

public @ResponseBody String index(){

}

## 2.通常用来返回JSON数据或者是XML，数据，

## 3.需要注意的呢，在使用此注解之后不会再走试图处理器，类名前面加入两个注释ResponseBody和controler就等于一个RestController



　@RequestMapping("/login")  
　　@ResponseBody  
　　public User login(User user){  
　　　　return user;  
　　}   
　　User字段：userName pwd  
　　那么在前台接收到的数据为：'{"userName":"xxx","pwd":"xxx"}'  
  
　　效果等同于如下代码：  
　　@RequestMapping("/login")  
　　public void login(User user, HttpServletResponse response){  
　　　　response.getWriter.write(JSONObject.fromObject(user).toString());  
　　}

@ResponseBody  
@PostMapping(**"add.json"**)  
**public int** add(@RequestBody Subdomain subdomain) {  
 **subdomianRepository**.save(subdomain);  
 **int** id = 1;  
 **return** id;  
}

# 16 @RequestBody

## 解释：RequestBody， 在这个方法中，使用@RequestParam获取参数，然后使用@RequestBody对json格式的参数转换为[Java](http://lib.csdn.net/base/javase)类型

1. @RequestMapping("doThis")
2. **public** String doThis(HttpServletRequest request,
3. @RequestParam("id") Long id, // 用户ID
4. @RequestParam("back\_url") String back\_url, // 回调地址
5. @RequestBody TestEntity json\_data // json数据，对于java实体类
6. )

@PutMapping(**"update.json"**)  
**public** String update(@RequestBody Subdomain subdomain) {  
 **subdomianRepository**.save(subdomain);  
 **return "index"**;  
}

# 17 数据表实体类的创建

## 1、@Entity注释指名这是一个实体Bean，

## 2、@Table注释指定了Entity所要映射带数据库表，其中@Table.name()用来指定映射表的表名。如果缺省@Table注释，系统默认采用类名作为映射表的表名。

**实体Bean的每个实例代表数据表中的一行数据，行中的一列对应实例中的一个属性。**

### 2,1 cataloge= 数据库名称

**name=“表名称”**

@Table(name = **"rec\_yard\_order"** cataloge=**"数据库名称"**)

## 3、@Column注释定义了将成员属性映射到关系表中的哪一列和该列的结构信息，属性如下：

#### 1）name：映射的列名。如：映射tbl\_user表的name列，可以在name属性的上面或getName方法上面加入；

@Column(name="id")

**private** Integer id;

### 2）unique：是否唯一；unique = true,

### 3）nullable：是否允许为空；nullable = true

### 4）length：对于字符型列，length属性指定列的最大字符长度；

@Column(length = 50,nullable = **true**)  
**private** String **sub**;

### 5）insertable：是否允许插入；

### 6）updatetable：是否允许更新；

### 7）columnDefinition：定义建表时创建此列的DDL；

#### 解释1，columnDefinition可以指定创建表时一些SQL语句，比如这里可以一次性指定，varchar长度128，且不能为空  当然，相同的结果可以通过nullable和length属性结合实现

##### 1 使用columnDefinition



##### 2 使用nullable 和length



#### 解释2 强制转化类型columnDefinition属性的特殊使用：  编程语言中字符串一般都用String表示，但是数据库中varcahr数值类型有长度限制，一旦需要大文本，则需要text数值类型  但是String类型默认映射的数值类型是varchar，columnDefinition可以进行额外指定



#### 解释3. 数据库中添加默认数据，而且可以添加非空数据，并且赋予默认值

*//学生审核状态*@Column(nullable=**false**,length = 1 ,columnDefinition=**"INT default 0"**)  
**private** String **stuStatus**;

### 8）secondaryTable：从表名。如果此列不建在主表上（默认是主表），该属性定义该列所在从表的名字。

## 4、@Id注释指定表的主键，它可以有多种生成方式：

#### 1、@GeneratedValue(strategy=GenerationType.AUTO)      相当于Hibernate框架的native主键生成策略 (直接写 @GeneratedValue 相当如native)

##### 解释：native

根据底层数据库的能力选择 identity、sequence 或者 hilo 中的一个; native:

##### 1.对于 oracle 采用 Sequence 方式，

##### 2.对于MySQL 和 SQL Server 采用identity（自增主键生成机制），native就是将主键的生成工作交由数据库完成，hibernate不管

#### 2、@GeneratedValue(strategy=GenerationType.IDENTITY)  相当于Hibernate框架的identity主键生成策略，适用于[MySQL](http://lib.csdn.net/base/mysql),SQL Server

##### 解释：“identity” 采用数据库提供的主键生成机制。如DB2、SQL   Server、MySQL   中的主键生成机制。，identity:，identity是自动增长列，由底层数据库生成标识符。identity是由数据库自己生成的,但这个主键必须设置为自增长,前提条件是低层数据库支持自动增长字段类型

#### 3、@GeneratedValue(strategy=GenerationType.SEQUENCE)  相当于Hibernate框架的sequence主键生成策略，适用于[Oracle](http://lib.csdn.net/base/oracle),DB2

#### 4、@GeneratedValue(strategy=GenerationType.TABLE)     使用一张单表管理主键值。

         默认情况下API规范是不支持Hibernate框架中的uuid的主键生成策略，假如我们想要使用，需要我们进行增加主键生成器，如下：

@GenericGenerator(name="hibernate-uuid",strategy="uuid")

@GeneratedValue(generator="hibernate-uuid")

## UUID

*//编号id 没有------*@Id  
@GenericGenerator(name = **"idGenerator"**, strategy = **"uuid"**)  
@GeneratedValue(generator = **"idGenerator"**)  
**private** String **id**;

*//主键 具备-------*@Id  
@GenericGenerator(strategy = **"uuid2"**,name = **"idGenerator"**)  
@GeneratedValue(generator=**"idGenerator"**)  
**private** UUID **id**;

##### 1）TABLE：容器指定用底层的数据表确保唯一；

##### 2）SEQUENCE：使用数据库德SEQUENCE列莱保证唯一（Oracle数据库通过序列来生成唯一ID）；

##### 3）IDENTITY：使用数据库的IDENTITY列莱保证唯一；

##### 4）AUTO：由容器挑选一个合适的方式来保证唯一；

##### 5）NONE：容器不负责主键的生成，由程序来完成。

@Id

@GeneratedValue(strategy = GenerationType.*AUTO*)

@Column(name="id")

**private** Integer id;

@Entity  
@Table(name=**"subdomains"**) *//没有默认就是这个实体类的名字***public class** Subdomain {  
 @Id  
 @GeneratedValue(strategy= GenerationType.***IDENTITY***)  
 **private** Integer **id**; *//id* @Column(length = 50,nullable = **true**)  
 **private** String **name**;  
 @Column(length = 50,nullable = **true**)  
 **private** String **sub**;  
 @Column(length = 255,nullable = **true**)  
 **private** String **href**;  
 @Column(nullable = **true**)  
 **private** Date **createOn**;*//创建时间* **public** Subdomain(){}  
 @Override  
 **public** String toString() {  
 **return "Subdomain{"** +  
 **"id="** + **id** +  
 **", name='"** + **name** + **'\''** +  
 **", subdomain='"** + **sub** + **'\''** +  
 **", href='"** + **href** + **'\''** +  
 **", create\_on="** + **createOn** +  
 **'}'**;  
 }  
  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public** String getSub() {  
 **return sub**;  
 }  
  
 **public void** setSub(String sub) {  
 **this**.**sub** = sub;  
 }  
  
 **public** String getHref() {  
 **return href**;  
 }  
  
 **public** Date getCreateOn() {  
 **return createOn**;  
 }  
  
 **public void** setCreateOn(Date createOn) {  
 **this**.**createOn** = createOn;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
  
 **public void** setHref(String href) {  
 **this**.**href** = href;  
 }  
  
}

# 6.范例

@Past

@DateTimeFormat(pattern="yyyy-MM-dd")

# 18. 日期

## 1. @CreationTimestamp导入数据时候自动创建日期，并且指定格式@JsonFormat ,但是版本一定要对应

@Column(nullable = **false**)  
@CreationTimestamp  
@JsonFormat(pattern = **"yyyyMMddhhmmss"**)  
**private** LocalDateTime applyDate;

*//操作时间*@Column(nullable = **false**)  
@CreationTimestamp  
@JsonFormat(pattern = **"yyyyMMdd"**)  
**private** LocalDate optDate;

## 2. @UpdateTimestamp更新数据时候自动帮我们导入时间

@Column(nullable = **true**)  
@UpdateTimestamp  
**private** LocalDateTime updateTime;

## 3.@Temporal注释用来指定java.util.Date或java.util.Calender属性与数据库类型date、time或timestamp中的那一种类型进行映射。

## @Temporal(value=TemporalType.TIME)

@Temporal(value= temporaltype.timestamp) **private** Date birthday;

 @Temporal(value=TemporalType.*DATE*)

**private** Date birthday;

### date和timestamp 的区别

     date类型是Oracle常用的日期型变量，他的时间间隔是秒。两个日期型相减得到是两个时间的间隔，注意单位是“天”。例如：查看一下当前距离伦敦奥运会开幕还有多长时间：

1 select to\_date('2012-7-28 03:12:00','yyyy-mm-dd hh24:mi:ss')-sysdate from dual

结果是：92.2472685185185天，然后你根据相应的时间换算你想要的间隔就行！这个结果可能对程序员有用，对于想直接看到结果的人，这个数字还不是很直观，所以，就引出了timestamp类型

timestamp是DATE类型的扩展，可以精确到小数秒（fractional\_seconds\_precision），可以是0 to9，缺省是６。两个timestamp相减的话，不能直接的得到天数书，而是得到，  
多少天，多少小时，多少秒等，例如：同样查看一下当前距离伦敦奥运会开幕还有多长时间.

1 select to\_timestamp('2012-7-28 03:12:00','yyyy-mm-dd hh24:mi:ss')-systimestamp from dual

     结果是：+000000092 05:51:24.032000000，稍加截取，就可以得到92天5小时，51分钟，24秒，这样用户看起来比较直观一些！但是这个数字对程序员来说不是很直观了，如果想要具体的时间长度的话，并且精度不要求到毫秒的话，可以将timestamp类型转成date类型，然后直接相减即可。

# [17.2 @Type](mailto:17.2@Type) 使用jsonb 字段设置为json（有文档的详细说明）

@Entity  
@Table(name = **"rec\_career\_fair\_entry"**)  
@TypeDefs({@TypeDef(name = **"StringJsonObject"**, typeClass = StringJsonUserType.**class**)})  
**public class** CarreFairEntry **implements** Serializable {

*//发票信息json*@Type(type = **"StringJsonObject"**)  
@Column(nullable = **true**)  
**private** String **invoiceInfo**;

之后数据库中就是jsonb的字段



# 19.JavaBean的valiation

<http://www.cnblogs.com/yangzhilong/p/3724967.html> 讲的非常好，

1. @Null   被注释的元素必须为 **null**
2. @NotNull    被注释的元素必须不为 **null**
3. @AssertTrue     被注释的元素必须为 **true**
4. @AssertFalse    被注释的元素必须为 **false**
5. @Min(value)     被注释的元素必须是一个数字，其值必须大于等于指定的最小值
6. @Max(value)     被注释的元素必须是一个数字，其值必须小于等于指定的最大值
7. @DecimalMin(value)  被注释的元素必须是一个数字，其值必须大于等于指定的最小值
8. @DecimalMax(value)  被注释的元素必须是一个数字，其值必须小于等于指定的最大值
9. @Size(max=, min=)   被注释的元素的大小必须在指定的范围内
10. @Digits (integer, fraction)     被注释的元素必须是一个数字，其值必须在可接受的范围内
11. @Past   被注释的元素必须是一个过去的日期
12. @Future     被注释的元素必须是一个将来的日期
13. @Pattern(regex=,flag=)  被注释的元素必须符合指定的正则表达式
15. Hibernate Validator 附加的 constraint
16. @NotBlank(message =)   验证字符串非**null**，且长度必须大于0
17. @Email  被注释的元素必须是电子邮箱地址
18. @Length(min=,max=)  被注释的字符串的大小必须在指定的范围内
19. @NotEmpty   被注释的字符串的必须非空
20. @Range(min=,max=,message=)  被注释的元素必须在合适的范围内

@DateTimeFormat(pattern="yyyy-MM-dd")

# [20.@SessionAttributes](mailto:20.@SessionAttributes) 防止重复刷新，使用redirect重定向，进行转发时候，不能保存到request中，所以使用session

@Controller

@RequestMapping("/user")

//将ModelMap中名为user的属性放到Session中，以便这个属性可以跨请求访问

@SessionAttributes("user")

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

//处理登录请求

@RequestMapping(value="/login")

**public** String login(Users user,ModelMap model){

//将user对象以user为键放入到model中

model.put("user", user);

**return** "success";

}

//@RequestMapping(value="/toReg")

//@RequestMapping(value="/reg",method=RequestMethod.GET)

@RequestMapping(params="method=toReg")

//获取"爱好"下拉列表数据，显示注册页面

**public** String toReg(ModelMap model){

model.put("hobbyList", **new** String[]{"Swimming","Running"});

**return** "register";

}

//日期格式转换

@InitBinder

**public** **void** initBinder(WebDataBinder binder){

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(**new** SimpleDateFormat("yyyy-MM-dd"), **true**));

}

//处理注册请求

//@RequestMapping(value="/doReg")

//@RequestMapping(value="/reg",method=RequestMethod.POST)

@RequestMapping(params="method=doReg") //使用参数之后就是 user.do?method=doReg

**public** String doReg(Users user, ModelMap model){

//将user对象以user为键放入到model中

model.put("user", user);

//这样重定向是访问不到user 的所以上面我们添加了了 @SessionAttributes("user")

**return** "redirect:/show.jsp";

}

# 21. @Repository

## 1.在接口继承操作数据库中可以进行注释

@Repository  
**public interface** LargeOrderRepository **extends** CrudRepository<LargeOrder, UUID> {  
 LargeOrder findById(UUID id);  
 *//查找全部信息* List<LargeOrder> findAll();  
  
  
}

# 22 @Modifying@Query

## 解释 更新，增添，或者删除的时候用到modify，查询不需要

*//更新审核状态 根据 id*@Modifying  
@Query(value = **"update LargeOrder set auditState = ?2 ,auditTime = ?3,auditPerson=?4 where id = ?1"**)  
**int** updateLargeOrderAudit(String id, String auditState, Date auditTime, String auditPerson);

# 23@Embeddable内嵌表集成Entry

## 1.编写内嵌表

@Embeddable  
**public class** Audit {  
 *//审核状态（0 等待审核，1 通过审核，2 未通过审核）* @Column(length = 1,nullable = **false**)  
 **private char auditStatus**;  
 *//审核时间* @Column(nullable = **true**)  
 **private** LocalDateTime **auditTime**;  
 *//审核人* @Column(length = 32,nullable = **true**)  
 **private** String **auditPerson**;  
 *//审核失败原因* @Column(length = 100,nullable = **true**)  
 **private** String **auditFailReason**;

## 2. @Embedded加入内嵌表

### <http://blog.csdn.net/lmy86263/article/details/52108130>

*//审核相关*@Embedded  
@AttributeOverrides(  
 { name是类中字段名字   
 @AttributeOverride(name = **"auditStatus"**, column = @Column(name = **"audit\_status"**, nullable = **false**)),  
 @AttributeOverride(name = **"auditTime"**, column = @Column(name = **"audit\_time"**, nullable = **true**)),   
 @AttributeOverride(name = **"auditPerson"**, column = @Column(name = **"audit\_person"**, nullable = **true**)),  
 @AttributeOverride(name = **"auditFailReason"**, column = @Column(name = **"audit\_fail\_reason"**, nullable = **true**))  
 }  
)

## 3.解释：（只是用Embedded 和只是用@Embedded 是一个效果，可以以多层嵌套）

### <http://blog.csdn.net/lmy86263/article/details/52108130>

## 4. 覆盖@Embeddable类中字段的列属性

这里就要使用另外的两个注解@AttributeOverrides和@AttributeOverride，这两个注解是用来覆盖@Embeddable类中字段的属性的。

#### @AttributeOverrides：里面只包含了@AttributeOverride类型数组；

#### @AttributeOverride：包含要覆盖的@Embeddable类中字段名name和新增的@Column字段的属性；

### 1.Person类如下：

@Entity

public class Person implements Serializable{

private static final long serialVersionUID = 8849870114127659929L;

@Id

@GeneratedValue

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false)

private Integer age;

@Embedded

@AttributeOverrides({@AttributeOverride(name="country", column=@Column(name = "person\_country", length = 25, nullable = false)),

@AttributeOverride(name="city", column = @Column(name = "person\_city", length = 15))})

private Address address;

//setter、getter

### 2.Address类如下：

@Embeddable

public class Address implements Serializable{

private static final long serialVersionUID = 8849870114128959929L;

@Column(nullable = false)

private String country;

@Column(length = 30)

private String province;

@Column(unique = true)

private String city;

@Column(length = 50)

private String detail;

//setter、getter

}

### 3.生成的数据表如下：



可以看出我们的@AttributeOverrides和@AttributeOverride两个注解起作用了。

# 24. 多个注释整理

## 1.@Component @Service @Repository @Controller 是等效的，区分是

### @Component 没有明确的角色

### @Service 业务逻辑层，Service层

### @Repository 数据访问层 Dao层

### @Controller 展现层，Mvc Spring mvc 使用

## 2. @Autowired @Inject @Resourse 是等效的,可以注解到set方法，或者属性上。一般都是属性上

## 3.Column 修饰数据库表中的字段，可以注解到get方法上或者属性上

# [25.@Configuration](mailto:25.@Configuration) 申明当前类是一个配置类，相当于一个spring的xml文件，springboot会自动识别

# @ComponentScan 自动扫描包下所有使用@Component @Service @Repository @Controller 的类 ，并且注册为Bean 见： P20

@Configuration 拦截器的配置时候用到过  
**public class** MyWebAppConfigurer **extends** WebMvcConfigurerAdapter {

# 26. @InitBinder

## 1.解释，下文中先将得到的字符串转为Date ，再将得到的Date转为LocalDateTime

@PutMapping(**"/largeOrder/updateAudit"**)  
**public** String updateAudit(String id, String auditState, Date auditTime, String auditPerson ) {  
 **int** updateStatus = **largeOrderService**.updateLargeOrderAudit(id,auditState,**this**.DateToLocalDateTime(auditTime), auditPerson);  
 **if**(updateStatus!=0){  
 **return "ok"**;  
 }  
 **return null**;  
}  
*//将url字符串日期转化为Date 类型*@InitBinder  
**public void** initBinder(ServletRequestDataBinder binder){  
 binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(**new** SimpleDateFormat(**"yyyy-MM-dd"**), **true**));  
}  
*//将Date转化为LocalDateTime***public** LocalDateTime DateToLocalDateTime(Date auditTime){  
 Instant instant = Instant.*ofEpochMilli*(auditTime.getTime());  
 LocalDateTime res = LocalDateTime.*ofInstant*(instant, ZoneId.*systemDefault*());  
 **return** res;  
}

# 27. @ControllerAdvice 这个本身就是一个controller

## 1.处理局部异常（Controller内）

### 1.处理局部异常 代码

@Controller  
**public class** LocalException {  
 @ExceptionHandler  
 **public** ModelAndView exceptionHandler(Exception ex){  
 ModelAndView mv = **new** ModelAndView(**"error"**);  
 mv.addObject(**"exception"**, ex);  
 System.***out***.println(**"in testExceptionHandler"**);  
 **return** mv;  
 }  
 @RequestMapping(**"/error"**)  
 **public** String error(){  
 **int** i = 5/0;  
 **return "hello"**;  
 }  
}

### 2.测试http://localhost:8080/error





## 2.处理全局异常（所有Controller）

### 1.代码

@ControllerAdvice  
**public class** GlobalException {  
 @ExceptionHandler  
 **public** ModelAndView exceptionHandler(Exception ex){  
 ModelAndView mv = **new** ModelAndView(**"error"**);  
 mv.addObject(**"exception"**, ex);  
 System.***out***.println(**"in testControllerAdvice"**);  
 **return** mv;  
 }  
 @RequestMapping(**"/globalException"**)  
 **public** String error(){  
 **int** i = 5/0;  
 **return "hello"**;  
 }  
}

## 2.测试，和局部异常结果一样

# 38 @ModerAttributu

<http://www.cnblogs.com/HD/p/4127915.html>

## 1. @ModelAttribute放在方法的注解上时，代表的是Controller的所有方法在调用前，先执行此@ModelAttribute方法。

### 1.代码

*/\*\*  
 \* ModelAttribute 没有参数，采用 map保存，相当于model也是一样  
 \* Created by HealerJean on 2017/3/29.  
 \*/*@Controller  
**public class** TestMa1Controller {  
  
 @ModelAttribute  
 **public void** getUser(Map<String, Object> map){  
 User user=**new** User( 23, **"张宇晋"**);  
 map.put(**"user"**, user);  
 }  
 @RequestMapping(**"test1"**)  
 **public** String testModelAttribute(User user){  
 System.***out***.println(**"修改："**+user);  
 **return "index"**;  
 }

### 2.测试



## 2. 1有User返回值，2两个ModelAttribute

### 1.代码

*/\*\*  
 \* ModelAttribute 不管有没有user参数 都可以返回值 ，  
 \* 解释;这种方法其实也是放到了model里面  
 \* 注意：有了返回值 User才能生效，仅仅是void不能生效  
  
 \*/*@Controller  
**public class** TestMa2Controller {  
 *//这种不带参数也是可以的  
 //@ModelAttribute* @ModelAttribute(**"user"**)  
 **public** User getUser(Map<String, Object> map){  
 User user=**new** User( 23, **"张宇晋"**);  
 **return** user;  
 }  
 @RequestMapping(**"test2"**)  
 *// @ModelAttribute("user") 整体 可有可没有  
 //public String testModelAttribute(@ModelAttribute("user") User user){* **public** String testModelAttribute(User user){  
 System.***out***.println(**"修改："**+user);  
 **return "index"**;  
 }

### 2.测试



## 3.自动匹配url

### 1.代码

*/\*\*  
 \* 地址栏输入 http://localhost:8080/test3?age=23&name=zhangyujin  
 \* Created by HealerJean on 2017/3/29.  
 \*/*@Controller  
**public class** TestMa3Controller {  
  
 @RequestMapping(**"test3"**)  
 **public** String testModelAttribute(@ModelAttribute(**"user"**) User user){  
 System.***out***.println(**"修改："**+user);  
 **return "index"**;  
 }

### 2.测试



# 39 @ControllerAdvice @ExceptionHandler

## 1.处理全局异常

### 1.先编写处理全局异常的类

@ControllerAdvice  
**public class** GlobalException {  
 @ExceptionHandler  
 **public** ModelAndView exceptionHandler(Exception ex){  
 ModelAndView mv = **new** ModelAndView(**"globalerror"**);  
 mv.addObject(**"exception"**, ex);  
 System.***out***.println(**"in testControllerAdvice"**);  
 **return** mv;   
 }  
}

### 2.编写测试异常的类

@Controller  
**public class** GlobalExceptionController {  
 @RequestMapping(**"/globalException"**)  
 **public** String error(){  
 **int** i = 5/0;  
 **return "hello"**;  
 }  
}

### 3.异常html

<**body**>  
全局异常  
<**input name ="id" th:value="${exception}" readonly="true"**/>  
<**br**/>  
局部异常  
<**input name ="id" th:value="${localexception}" readonly="true"**/>  
  
</**body**>

### 4.测试结果



## [2.局部异常 @ExceptionHandler](mailto:2.局部异常@ExceptionHandler) control内

### 1.测试类

@Controller  
**public class** LocalExceptionController {  
 @ExceptionHandler  
 **public** ModelAndView exceptionHandler(Exception ex){  
 ModelAndView mv = **new** ModelAndView(**"globalerror"**);  
 mv.addObject(**"localexception"**, ex);  
 System.***out***.println(**"in testExceptionHandler"**);  
 **return** mv;  
 }  
 @RequestMapping(**"/localException"**)  
 **public** String error(){  
 **int** i = 5/0;  
 **return "hello"**;  
 }  
  
}

### 2.测试成功



# 40。@SpringBootAppliction

## 解释：

## 1注解它的表示一个启动类，里面包含main方法

## 2. 使用下面这个三个的注解和它是一样的 @Configuration @EnableAutoConfiguration

#### 解释：让springboot 根据类的路径中的jar包依赖为当前项目进行自动配置，比如，添加了spring-boot-start-web，会自动添加Tomact和spring MVC 的依赖。这样就会对二者进行自动配置）

**@ComponentScan**

## 3.spring boot会自动扫描 springBootApplication 所在类的同级别的包，所以这个启动类要放到组合包下

# 41. @NamedQuery 查询语句 可以视情况变为List<Person>

## 1.实体类

@Entity *//1*@NamedQuery(name = **"Person.withNameAndAddressNamedQuery"**,  
query = **"select p from Person p where p.name=?1 and address=?2"**)  
**public class** Person {  
 @Id *//2* @GeneratedValue *//3* **private** Long **id**;  
   
 **private** String **name**;  
   
 **private** Integer **age**;  
   
 **private** String **address**;

## 2.jpa接口层

Person withNameAndAddressNamedQuery(String name,String address);

# 42@UniqueConstraint

## 1.唯一约束，stuNo 为实体类中的变量

@Table(name = **"stu\_information"**,uniqueConstraints = {@UniqueConstraint(columnNames=**"stuNo"**)})

## 2. 如果是联合约束就用下面这种

@Table(name="tbl\_sky",  
  uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})})

# 43@JsonIgnore （DTO）

## 解释：向前台返回数据，时候不返回而是进行忽略。这里使用字码表来返回真实数据，而不返回代码

*//申请审核状态（1 待审核 ，2 院系同意，3 院系不同意，4 学校同意 ，5 学校不同意）***private** String **auditStatus**;  
*//申请审核状态对应名称***private** String **auditStatusName**;

**public** String getAuditStatusName() {  
 **return** PubCodeUtil.*getName*(**"totalAuditStatus"**,**this**.getAuditStatus());  
}

**var** status = $(e)[0].auditStatusName;

# 44. @Transient

## 解释：实体类中如果加上这个注解将不会再数据库表中映射字段，例如下面的字码表中的使用，（js中）

*//状态status ? 最终审核状态*@Column(length = 1,nullable = **false**)  
**private** String **status**;  
@Transient  
**private** String **statusName**;

**public** String getStuStatusName() {  
 **return** PubCodeUtil.*getName*(**"auditStatus"**,**this**.**stuStatus**);  
}  
  
**public** String getStatusName() {  
 **return** PubCodeUtil.*getName*(**"auditStatus"**,**this**.**status**);  
}

## 2.举例，这样就会对应出相应的子码代码的名称

**$**.each(obj.**content**, **function**(i,e) {  
 inner +=  
 **'<tr>'** +  
 **'<td>'**+**$**(e)[0].stuStatusName+**'</td>'** +  
 **'<td>'**+**$**(e)[0].statusName +**'</td>'** +  
 **'<td>'**+**$**(e)[0].studentInfo.**stuNo**+**'</td>'** +  
 **'<td>'**+**$**(e)[0].studentInfo.**name**+**'</td>'** +  
 **'<td>'**+**$**(e)[0].studentInfo.majorCodeName+**'</td>'** +  
 **'<td>'**+**$**(e)[0].studentInfo.eduModeName+**'</td>'** +  
 **'<td>'**+**$**(e)[0].**recName**+**'</td>'** +  
 **'<td>'**+**$**(e)[0].**recAddress**+**'</td>'** +  
 **'<td><a>详情</a></td>'** +  
 **'</tr>'**;

# 45. @PrimaryKeyJoinColumn

## 解释：一对一关系，让主键相同 下面是就业去向和 学生表的主键是相同的

@Entity  
@Table(name = **"graduate\_destination"**)  
**public class** GraduateDestination {  
 *// 编号，主键UUID* @Id  
 @GenericGenerator(name = **"myForeignGenerator"**, strategy = **"foreign"**,  
 parameters = @Parameter(name=**"property"**,value=**"studentInfo"**))  
 @GeneratedValue(generator = **"myForeignGenerator"**)  
 **private** UUID **id**;  
 *// 关联的学生信息* @OneToOne(cascade={CascadeType.***ALL***}, optional = **false**)  
 @PrimaryKeyJoinColumn  
 **private** StudentInformation **studentInfo**;

# 46. @Test (单元测试)

## 1.依赖

*<!--Test测试-->* <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 </**dependency**>

## 2.测试方法

**public class** Test {  
 @org.junit.Test  
 **public void** testHello(){  
 TestService testService = **new** TestService();  
 testService.hello();  
 }

**public class** TestService { //另外一个类  
 **public void** hello(){  
 System.***out***.println(**"测试"**);  
 }  
}

# 47@param



# 48. @Deprecated（多个程序员开发时候，为了项目中改动）

## 解释：  这个注释是一个标记注释。所谓标记注释，就是在源程序中加入这个标记后，并不影响程序的编译，但有时编译器会显示一些警告信息。      那么Deprecated注释是什么意思呢？如果你经常使用eclipse等IDE编写java程序时，可能会经常在属性或方法提示中看到这个词。如果某个类成员的提示中出现了个词，就表示这个并不建议使用这个类成员。因为这个类成员在未来的JDK版本中可能被删除。之所以在现在还保留，是因为给那些已经使用了这些类成员的程序一个缓冲期。如果现在就去了，那么这些程序就无法在新的编译器中编译了。

举例：    Deprecated注释还有一个作用。就是如果一个类从另外一个类继承，并且override被继承类的Deprecated方法，在编译时将会出现一个警告。如test.java的内容如下：

class Class1 { @Deprecated public void myMethod(){} } class Class2 extends Class1 { public void myMethod(){} }

运行javac test.java 出现如下警告：   
  
    注意：test.java 使用或覆盖了已过时的 API。   
    注意：要了解详细信息，请使用 -Xlint:deprecation 重新编译   
    使用-Xlint:deprecation显示更详细的警告信息：   
  
    test.java:4: 警告：[deprecation] Class1 中的 myMethod() 已过时   
  
    public void myMethod()   
    ^   
    1 警告   
  
    这些警告并不会影响编译，只是提醒你一下尽量不要用myMethod方法。

# 49.@CookieValue(value="name",required=false

## 解释：从cookie数组中获取相应的cookie，,required=false，可以为空

# 50.${random.value}，使用随机数

## 解释：yml页面中使用，这种方式可以在设置应用端口等场景，以避免在本地调试中出现端口冲突的问题

## 1、random.value 随机字符串

## 2、random.int 随机int

## 3、random.long 随机long

## 4、random.int(10) 10以内的随机数

## 5、randomo.int(10,20) 10到20以内的随机数

# 52. @AutoConfigureBefore(ThymeleafAutoConfiguration.class)

## 解释：在加载某个类之前率先加载本类

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | /\*\* | |  | \* 在Thymeleaf的自动配置前先将附加的Thymeleaf自定义方言注入进去 | |  | \*/ | |  | @Configuration | |  | @AutoConfigureBefore(ThymeleafAutoConfiguration.class) | |  | public class ThymeleafAdditionalDialect {  @ConditionalOnMissingBean注解判断是否执行初始化代码，即如果用户已经创建了bean，则相关的初始化代码不再执行 | |  | @Bean | |  | @ConditionalOnMissingBean | |  | public PubCodeDialect pubcodeDialect() { | |  | return new PubCodeDialect(); | |  | } | |  |  | |  | @Bean | |  | @ConditionalOnMissingBean | |  | public ShiroDialect shiroDialect() { | |  | return new ShiroDialect(); | |  | } | |  | } | |