### 1.创建maven工程，配置相关的依赖。

### 2.数据库设计然后设计对应的实体类。

实体类遇到的知识点：

可以通过@Cache注解来提供缓存。Hibernate提供以下击中缓存策略:

@Cache(usage = CacheConcurrencyStrategy.NONE) // 不使用缓存

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY) //只读模式

@Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)//读写模式，加锁模式

@Cache(usage = CacheConcurrencyStrategy.NONSTRICT\_READ\_WRITE)//不加锁模式读写

@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)//事务模式，当事务回滚时，缓存也回滚。

### 3.对于ManytoMany，需要设置中间表

然后在实体类配置如下：

1）拥有方上定义

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE}, mappedBy = "manBoards", fetch = FetchType.LAZY)

private Set<User> users = new HashSet<User>();

cascade(级联)

级联在编写触发器时经常用到，触发器的作用是当主控表信息改变时,用来保证其关联表中数据同步更新。若对触发器来修改或删除关联表相记录，必须要删除对应的关联表信息，否则，会存有脏数据。所以，适当的做法是，删除主表的同时，关联表的信息也要同时删除，在hibernate中，只需设置cascade属性值即可。

CascadeType.PERSIST：级联新增（又称级联保存）：对一个对象保存时也对另一个里的对象也会保存。对应EntityManager的presist方法。

CascadeType.MERGE：级联合并（级联更新）：若A属性修改了那么B对象保存时同时修改A里的对象。对应EntityManager的merge方法 。

CascadeType.REMOVE：级联删除：对B对象删除也对A里的对象也会删除。对应EntityManager的remove方法。

CascadeType.REFRESH：级联刷新：获取B对象里也同时也重新获取最新的A时的对象。对应EntityManager的refresh(object)方法有效。即会重新查询数据库里的最新数据。

CascadeType.ALL：以上四种都是。

mappedBy = "manBoards"//在拥有方定义的，正是由于拥有方的关联被拥有方的字段存在，拥有方才拥有了被拥有方。mappedBy这方定义JoinColumn/JoinTable总是失效的，不会建立对应的字段或者表。

FetchType.LAZY：懒加载，加载一个实体时，定义懒加载的属性不会马上从数据库中加载，即从数据库读到内存。

FetchType.EAGER：急加载，加载一个实体时，定义急加载的属性会立即从数据库中加载。也可以说成表示关联关系的从类在主类加载时同时加载。

2）被拥有方上定义：

@ManyToMany(cascade={CascadeType.PERSIST,CascadeType.MERGE},fetch= FetchType.EAGER)

@JoinTable(name = "t\_board\_manager", joinColumns = {@JoinColumn(name ="user\_id" )}, inverseJoinColumns = {@JoinColumn(name = "board\_id") })

通过表关联的方式来映射一对多或者多对多的关系时，要使用@JoinTable这个标记。

**@JoinTable支持的属性**

| **属性** | **是否必须** | **说明** |
| --- | --- | --- |
| name | 否 | 指定该连接表的表名 |
| JoinColumns | 否 | 该属性值可接受多个@JoinColumn，用于配置连接表中外键列的信息，这些外键列参照当前实体对应表的主键列 |
| inverseJoinColumns | 否 | 该属性值可接受多个@JoinColumn，用于配置连接表中外键列的信息，这些外键列参照当前实体的关联实体对应表的主键列 |
| targetEntity | 否 | 该属性指定关联实体的类名。在默认情况下，Hibernate将通过反射来判断关联实体的类名 |
| catalog | 否 | 设置将该连接表放入指定的catalog中。如果没有指定该属性，连接表将放入默认的catalog |
| schema | 否 | 设置将该连接表放入指定的schema中。如果没有指定该属性，连接表将放入默认的schema |
| uniqueConstraints | 否 | 该属性用于为连接表增加唯一约束 |
| indexes | 否 | 该属性值为@Index注解数组，用于为该连接表定义多个索引 |

注解来继承类

@Inheritance(strategy = InheritanceType.*SINGLE\_TABLE*)  
@DiscriminatorColumn(name = "post\_type",discriminatorType = DiscriminatorType.*STRING*)  
@DiscriminatorValue("1")

@Entity  
@Cache(usage = CacheConcurrencyStrategy.*READ\_WRITE*)  
@Inheritance(strategy = InheritanceType.*SINGLE\_TABLE*)  
@DiscriminatorColumn(name = "post\_type", discriminatorType = DiscriminatorType.*STRING*)  
@DiscriminatorValue("2")  
public class MainPost extends Post{  
}

一、SINGLE\_TABLE

SINGLE\_TABLE 是将父类和其所有的子类集合在一块，存在一张表中，并创建一个新的字段来判断对象的类型。

Person.[**Java**](http://lib.csdn.net/base/java)：

1: @Entity

2: @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

3: @DiscriminatorColumn(name="discriminator",discriminatorType=DiscriminatorType.STRING)

4: @DiscriminatorValue("person")

5: public class Person {

6: @Id

7: @GeneratedValue

8: private int id;

9: private String name;

10: //Getters and setters omitted here...

11: }

@Inheritance 的 strategy 属性是指定继承关系的生成策略，@DiscriminatorColumn 注解作用是指定生成的新的判断对象类型的字段的名称和类型，@DiscriminatorValue 注解是确定此类（Person）的标示，即 DiscriminatorColumn 的值。

Student.java：

1: @Entity

2: @DiscriminatorValue("student")

3: public class Student extends Person{

4: private int score;

5: //Getters and setters omitted here...

6: }

Teacher.java：

1: @Entity

2: @DiscriminatorValue("teacher")

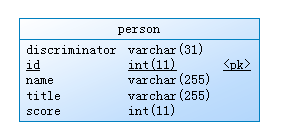
3: public class Teacher extends Person{

4: private String title;

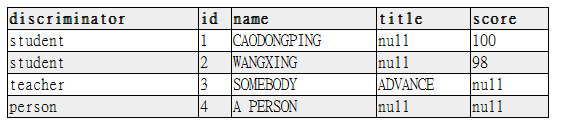
5: //Getters and setters omitted here...

6: }

生成的[**数据库**](http://lib.csdn.net/base/mysql)表如下：

[](http://hi.csdn.net/attachment/201102/18/0_1298031556kctL.gif)

存入数据后结构如下：

[](http://hi.csdn.net/attachment/201102/18/0_1298031557h2m2.gif)

二、TABLE\_PER\_CLASS

TABLE\_PER\_CLASS 是为每一个类创建一个表，这些表是相互独立的。

Person.java：

1: @Entity

2: @Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

3: public class Person {

4: @Id

5: private int id;

6: private String name;

7: //Getters and setters omitted here...

8: }

Student.java：

1: @Entity

2: public class Student extends Person{

3: private int score;

4: //Getters and setters omitted here...

5: }

Teacher.java：

1: @Entity

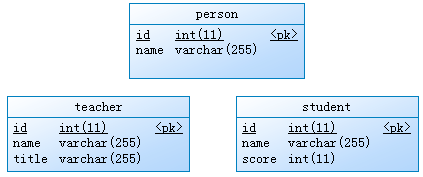
2: public class Teacher extends Person{

3: private String title;

4: //Getters and setters omitted here...

5: }

创建的数据库表如下：

[](http://hi.csdn.net/attachment/201102/18/0_1298031557xRq5.gif)

三、JOINED

JOINED 是将父类、子类分别存放在不同的表中，并且建立相应的外键，以确定相互之间的关系。

Person.java：

1: @Entity

2: @Inheritance(strategy=InheritanceType.JOINED)

3: public class Person {

4: @Id

5: @GeneratedValue

6: private int id;

7: private String name;

8: //Getters and setters omitted here...

9: }

子类中只需声明与父类不同的域即可：

Student.java：

1: @Entity

2: public class Student extends Person{

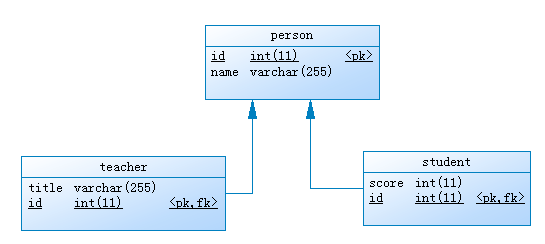
3: private int coat;

4: //Getters and setters omitted here...

5: }

Teacher.java 类似。

生成的数据库表如下：

[](http://hi.csdn.net/attachment/201102/18/0_129803155840V0.gif)

### 4.Dao Bean和service的装配

1）开启扫描：扫描dao层、service层

2）引入jdbc配置文件

3）配置数据源

4）配置sessionFactory：包括属性packagetoscan配置到实体类

5）配置事务管理器、开启事务注解

一、spring mvc处理请求的流程。

客户端发出http请求，如果匹配到DispatcherServlet的请求映射路径，则web容器将请求转交给DispatcherServlet处理。

DispatcherServlet接受到请求后，根据请求的信息和HandlerMapping的配置，找到处理请求的处理器(Handler)。HandlerMapping类是路由控制器。

得到相对应的Handler后，通过HandlerAdapter对Handler进行封装，再以统一的适配器接口调用Handler。

处理完业务后，返回ModelAndView给DispatcherServlet。

通过ViewResolver完成ModelAndView逻辑视图到真实视图的解析工作。

二、在web中配置DispatcherServlet

1.配置<context-param>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-base.xml,classpath:spring-another.xml</param-value>

</context-param>

通过contextConfigLoaction在web.xml里配置需要加载的spring配置文件

2.配置<listener>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

在web.xml里配置Listener，本Listener会去加载第一步中指定的配置文件，如果指定了要加载的文件,则会去加载相应的xml,而不会去加载/WEB-INF/下的applicationContext.xml。(指定加载即是第1步的配置)如果没有指定的话，默认会去/WEB-INF/下加载applicationContext.xml。

3.配置DispatcherServlet

<servlet>

<servlet-name>Servlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Servlet</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

配置了名为Servlet的DispatcherServlet，它默认自动加载/WEB-INF/Servlet-servlet.xml的spring配置文件，启动web层的Spring容器。

可以通过<servlet>的<init-param>指定：

属性namespace：对于DispatcherServlet的命名空间，默认为<servlet-name>-servlet。用于构造spring配置文件的路径，指定该属性，可以指定spring配置文件。

属性contextConfigLocation：如果DispatcherServlet对应上下文有多个spring配置文件，则可以使用该属性指定。

三、注解驱动的控制器

1.使用@RequestMapping映射请求

/\*\*

\* 接收GET请求

\* @return

\*/

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

public String get(){

System.out.println("get");

return "get";

}

/\*\*

\* 接收POST请求

\* @return

\*/

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

public String post(){

System.out.println("post");

return "post";

}

/\*\*

\* 只接收 name 参数

\* @return

\*/

@RequestMapping(value="/delete",params="name")

public String params(String name){

System.out.println("hello "+name);

return "helloworld";

}

/\*\*

\* 只接收请求头中 Content-Type 为 text/html;charset=UTF-8的请求

\* @return

\*/

@RequestMapping(value="/delete",headers="Content-Type:text/html;charset=UTF-8")

public String headers(){

System.out.println("headers");

return "helloworld";

}

2.请求处理方法签名

Spring MVC通过分析处理方法的签名，把http请求信息绑定到处理方法的相应入参中，然后再调用处理方法得到返回值。以下是四种绑定参数的方式：

// ①请求参数按名称匹配的方式绑定到方法入参中，方法返回对应的字符串代表逻辑视图名

@RequestMapping(value = "/handle1")

public String handle1(@RequestParam("userName") String userName,

@RequestParam("password") String password,

@RequestParam("realName") String realName) {

return "success";

}

// ②将Cooke值及报文头属性绑定到入参中、方法返回ModelAndView

@RequestMapping(value = "/handle2")

public ModelAndView handle2(@CookieValue("JSESSIONID") String sessionId,

@RequestHeader("Accept-Language") String accpetLanguage) {

ModelAndView mav = new ModelAndView();

mav.setViewName("success");

mav.addObject("user", new User());

return mav;

}

// ③请求参数按名称匹配的方式绑定到user的属性中、方法返回对应的字符串代表逻辑视图名

@RequestMapping(value = "/handle3")

public String handle3(User user) {

return "success";

}

// ④直接将HTTP请求对象传递给处理方法、方法返回对应的字符串代表逻辑视图名

@RequestMapping(value = "/handle4")

public String handle4(HttpServletRequest request) {

return "success";

}

3.使用HttpMessageConverter

1）概念：HttpMessageConverter<T>负责把请求信息转换为一个对象，将对象输出作为响应信息。

2）使用：通过@RequestBody和@ResponseBody

@RequestMapping(value = "/handle41")

public String handle41(@RequestBody String body) {//把请求报文体转换为字符串绑定到RequestBody中

System.out.println(body);

return "success";

}

@ResponseBody//把数据输出到响应流中。

@RequestMapping(value = "/handle42/{imageId}")

public byte[] handle42(@PathVariable("imageId") String imageId)

throws IOException {

System.out.println("load image of " + imageId);

Resource res = new ClassPathResource("/image.jpg");

byte[] fileData = FileCopyUtils.copyToByteArray(res.getInputStream());

return fileData;

}

@RequestBody

作用：

i) 该注解用于读取Request请求的body部分数据，使用系统默认配置的HttpMessageConverter进行解析，然后把相应的数据绑定到要返回的对象上；

ii) 再把HttpMessageConverter返回的对象数据绑定到 controller中方法的参数上。

@ResponseBody

作用：

该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

使用时机：

返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

@RestController

作用：

相当于@Controller+@ResponseBody

4.处理模型数据

spring mvc在调用处理方法前会创建一个隐含的模型对象，如果处理方法的入参为Map或者Model类型，则spring mvc会把隐形模型的引用传递给这些入参，一般是在方法参数写上Model model，然后通过model.addAttribute(key,value)来添加数据。

@RequestMapping(value = "/handle63")

public String handle63(ModelMap modelMap) {

User user = (User) modelMap.get("user");//springmvcv会把隐含模型对象传递给modelMap，所以可以访问模型的数据

user.setUserName("tom");

modelMap.addAttribute("testAttr", "value1");

return "/user/showUser";

}

四、处理方法的数据绑定

spring mvc通过反射机制对目标处理函数的签名进行分析，将请求消息绑定到处理方法的入参中，核心部件是DataBinder。

1.数据格式化

spring使用转换器进行源类型对象到目标类型对象的转换。spring提供两个内建的实现类，分别支持数字及日期类型的注解驱动格式化。

@NumberFormat和@DateTimeFormat注解

<bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<list>

<bean class="com.smart.domain.StringToUserConverter"/>

</list>

</property>

</bean>

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

private Date birthday;

@NumberFormat(pattern="#,###.##")

private long salary;

form.add("birthday", "1980-01-01");

form.add("salary", "4,500.00");

2.数据校验

<mvc:annotation-driven/>会默认装配一个LocalValidatorFactoryBean，通过处理方法的入参上标注的@Valid注解，即可让spring mvc在完成数据绑定后执行数据校验工作。

1）在model设置检验条件

@Pattern(regexp="w{4,30}")//通过正则校验，匹配4-30个包含数字、字母、下划线的字符

private String userName;

@Pattern(regexp="S{6,30}")//通过正则校验，匹配6-30个非空白的字符

private String password;

@Length(min=2,max=100)//长度限制在2-100

private String realName;

@Past //时间值必须是个过去的时间

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

private Date birthday;

@DecimalMin(value="1000.00")//数据必须在1000.00-100000.00之间

@DecimalMax(value="100000.00")

@NumberFormat(pattern="#,###.##")

private long salary;

2）然后在处理函数中处理

@RequestMapping(value = "/handle91")

public String handle91(@Valid @ModelAttribute("user") User user,

BindingResult bindingResult, ModelMap mm) {

if (bindingResult.hasErrors()) {

return "/user/register3";

} else {

return "/user/showUser";

}

}

在入参对象前添加了@Valid注解，同时在其后声明了一个BindingResult入参。然后通过BindingResult就可以进行判断，处理了。BindingResult除了hasErrors()外还有以下几个接口：getFieldError(String field)获取属性名对应的校验错误、getFiledErrors()获取所有属性校验错误、getFiledValue(String field)获取属性值、getErrorCount()获取错误数量。

五、视图和视图解析器

请求处理方法执行完成后，最终返回一个ModelAndView对象，spring mvc借助视图解析器得到最终的视图对象View。视图对象是一个bean，由视图解析器负责实例化。

在xml中配置试图解析器

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"

p:order="100" p:viewClass="org.springframework.web.servlet.view.JstlView"

p:prefix="/WEB-INF/views/" p:suffix=".jsp"/>