# 尚硅谷网上书城

## 第一阶段：登录、注册的验证

1. 使用jQuery技术对登录中的用户名、密码进行非空验证
2. 使用jQuery技术和正则表达式对注册中的用户名、密码、确认密码、邮箱进行格式验证，对验证码进行非空验证

## 第二阶段：实现登录、注册

### 软件的三层架构

* 生活中的三层架构



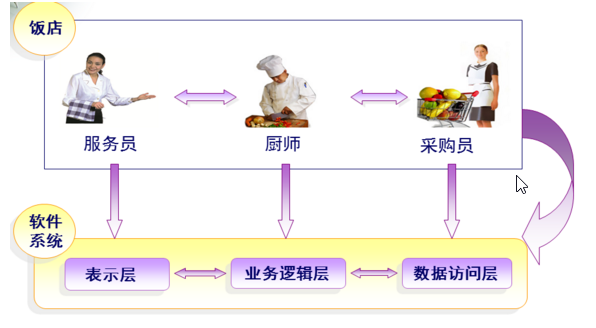
* **服务员：**只管接待客人。
* **厨师：**只管做客人点的菜。
* **采购员：**只管按客人点菜的要求采购食材。

他们各负其职，服务员不用了解厨师如何做菜，不用了解采购员如何采购食材；厨师不用知道服务员接待了哪位客人，不用知道采购员如何采购食材；同样，采购员不用知道服务员接待了哪位客人，不用知道厨师如何做菜。

* 软件中的三层架构

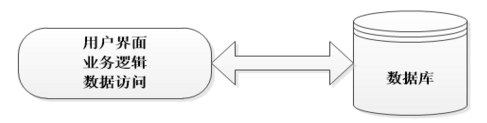


* **UI (User Interface layer表现层):**主要是指与用户交互的界面。用于接收用户输入的数据和显示处理后用户需要的数据。
* **BLL (** **Business Logic Layer业务逻辑层):**UI层和DAL层之间的桥梁。实现业务逻辑。业务逻辑具体包含：验证、计算、业务规划等等。
* **DAL (Data access layer数据访问层):**与数据库打交道。主要实现对数据的增、删、改、查。将存储在数据库中的数据提交给业务层，同时将业务层处理的数据保存到数据库。（当然这些操作都是基于UI层的，用户的需求反映给界面UI，UI反映给BLL，BLL反映给DAL，DAL进行数据的操作，操作后再一一返回，直到将用户所需数据反馈给用户）。
* 为什么使用三层架构



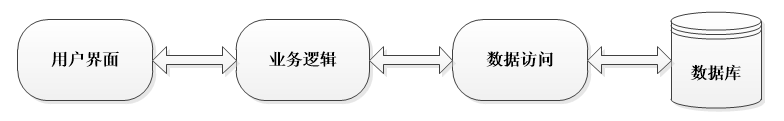
* 使用三层架构的目的：**解耦！！！**
* 同样拿上面饭店的例子来讲：
  + 服务员（UI层）辞职——另找服务员；厨师（BLL层）辞职——招聘另一个厨师；采购员（DAL）辞职——招聘另一个采购员。
  + 顾客反映：店里服务态度不好——服务员的问题，开除服务员；店里菜里有虫子——厨师的问题，换厨师。
* **任何一层发生变化都不会影响到另外一层！！！**
* 三层与一层的区别

**一层：**



* 当任何一个地方发生变化时，都需要重新开发整个系统。“多层”放在一层，分工不明确，**耦合度高**——难以适应需求变化，可维护性低、可扩展性低。

**三层：**



* 发生在哪一层的变化，只需更改该层，不需要更改整个系统。层次清晰，分工明确，每层之间**耦合度低**——提高了效率，适应需求变化，可维护性高，可扩展性高。
* 三层架构的优势和劣势
* **优势：**
  1. 结构清晰、耦合度低
  2. 可维护性高，可扩展性高
  3. 利于开发任务同步进行
  4. 容易适应需求变化
* **劣势：**

1. 降低了系统的性能。这是不言而喻的。如果不采用分层式结构，很多业务可以直接造访数据库，以此获取相应的数据，如今却必须通过中间层来完成。
2. 有时会导致级联的修改。这种修改尤其体现在自上而下的方向。如果在表示层中需要增加一个功能，为保证其设计符合分层式结构，可能需要在相应的业务逻辑层和数据访问层中都增加相应的代码。
3. 增加了代码量，增加了工作量。

### 书城的三层架构

* 表现层
* HTML、Servlet
* 接受用户的请求，调用业务逻辑层处理用户请求，显示处理结果
* 业务逻辑层
* Service
* 调用数据访问层处理业务逻辑
* 采用面向接口编程的思想，先定义接口，再创建实现类
* 数据访问层
* Dao
* 用来操作数据库，对数据库进行增删改查
* 采用面向接口编程的思想，先定义接口，再创建实现类

### 书城的三层架构具体实现

## 第三阶段：动态书城的开始及局部优化

### 将所有HTML页面改为JSP页面

* 复制一份BookStore02，重命名为BookStore03，仅此还不够！还需要在BookStore03工程上右键→Properties→搜索Web→单机Web Project Settings→修改Context root为BookStore03。
* 在每一个HTML页面的第一行添加JSP的page指令：

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"* pageEncoding=*"UTF-8"*%>，然后将HTML页面的文件扩展名改为jsp，即将.html改为.jsp。**注意：一定要先在HTML页面中添加jsp指令再修改扩展名**，否则页面中会出现乱码现象。

* 动态获取base标签

<base href=*"http://*<%=request.getServerName() %>*:*<%=request.getServerPort() %><%=request.getContextPath() %>*/"*>

* 获取IP地址：**request.getServerName()**
* 获取端口号：**request.getServerPort()**
* 获取项目虚拟路径：**request.getContextPath()**
* 将每个页面都使用的那些重复的部分使用JSP指令中的include指令进行静态包含。
* 创建一个base.jsp页面，删除除page指令之外的所有内容；将base标签和link标签复制到base.jsp页面中
* 为了不让用户能够直接访问base.jsp页面，在WEB-INF下创建一个include目录，将base.jsp页面放入该目录中。

### 显示错误消息及表单回显

* 登录、注册失败时我们分别转发到了登录、注册页面，但是并没有给用户一个错误消息，用户体验差，现在我们分别为登录、注册失败时提供一个错误消息并在页面显示：
  + 登录、注册失败时我们将错误消息放到request域中，通过**转发**将错误消息带到登录、注册页面，然后在页面中通过request域对象获取错误消息。
* 登录、注册失败时我们输入的用户名等信息会被清空，为了让用户有更好的体验，让用户看到之前输入的内容，需要对表单进行回显：
  + 给需要回显的表单项设置value属性，value值通过**request.getParameter("属性值")**来获取。

### 局部优化

* 优化一（整合Servlet）：现在我们的项目是一个请求对应一个Servlet，**我们能不能将多个请求对应一个Servlet呢？**比如用户登录、注册的LoginServlet、RegistServlet使用一个UserServlet代替：
* 在用户发送请求的URL地址后面添加一个method请求参数，在UserServlet中添加两个方法login()和regist()。**但是**，两个方法的格式与doGet和doPost要保持一致，可以复制一个doGet或doPost方法，将方法名改为login或regist。
* 登录表单提交到**UserServlet?method=login**，注册表单提交到**UserServlet?method=regist**。在UserServlet的doGet方法中获取请求参数，在doPost方法中调用doGet方法；在doGet方法中使用if else语句判断method的值，如果是login就调用login方法，如果是regist就调用regist方法。
* 在UserServlet中创建UserService对象，将LoginServlet、RegistServlet的doPost方法中的内容分别复制到UserServlet的login方法和regist方法中即可实现之前的登录注册功能。
* 优化二（动态调用方法）：在优化一中实现了多个请求对应一个Servlet的需求，但是我们在UserServlet中每添加一个方法就要在doGet方法中添加一个 else if语句！而且我们发现我们获取的请求参数的值正好是UserServlet中的方法名**，那么我们能不能通过请求参数动态的调用对应的方法呢？**
  + 通过**反射**的形式动态获取方法对象，这样我们再添加新的方法时就不需要再写if else语句了，但是新添加的方法的格式仍需要与doGet或者doPost的格式一致。
    - 根据请求方式获取方法名
      * String **methodName** = request.getParameter("method");
    - 根据方法名获取方法的对象，**getDeclaredMethod()**方法用来获取当前类中的某一个方法；第一个参数是方法名，第二个参数是方法的参数类型
      * Method method = this.getClass().getDeclaredMethod(**methodName**, HttpServletRequest.class,HttpServletResponse.class);
* 设置访问权限
* method.setAccessible(true);
* 调用方法。invoke()方法用来调用一个方法；第一个参数是要调用哪个对象的方法，第二个参数是调用方法需要传的参数
  + method.invoke(this, request,response);
* 优化三（创建BaseServlet）：在优化二中我们实现了动态调用对应方法的需求，但是以后每次创建Servlet的时候都需要将以上代码重写一遍，那么我们为何不把以上代码单独放到一个Servlet中，我们给它命名为BaseServlet，以后再创建Servlet时直接让它继承BaseServlet即可。
  + 创建一个专门用来被其他Servlet继承的BaseServlet
  + **但是不可以重写BaseServlet中的doGet和doPost方法**

### 使用EL替换项目中的所有JSP表达式

## 第四阶段：图书的增删改查

### 管理端图书的增删改查

* 创建图书类Book



* private Integer id;
* private String title; //书名
* private String author; //作者
* private Double price; //价格
* private Integer sales; //销量
* private Integer stock; //库存
* private String imgPath; //封面图片的路径
  + 指定一个默认值 static/img/default.jpg
* 创建图书表books
* 创建BookDao接口及实现类
* public void saveBook(Book book); 向数据库中插入一本图书
* public void deleteBook(String bookId); 根据图书的ID删除一本图书
* public void updateBook(Book book); 更新一本图书
* public Book getBookById(String bookId); 根据图书的ID查询一本图书
* public List<Book> getBookList(); 获取所有的图书
* 创建BookService接口及实现类
* public void saveBook(Book book); 向数据库中插入一本图书
* public void deleteBook(String bookId); 根据图书的ID删除一本图书
* public void updateBook(Book book); 更新一本图书
* public Book getBookById(String bookId); 根据图书的ID查询一本图书
* public List<Book> getBookList(); 获取所有的图书
* 创建BookManagerServlet
* bookList() → 获取图书列表的方法
* ~~addBook() → 添加图书的方法~~
* deleteBook() → 删除图书的方法
* editBook() → 编辑图书的方法
* saveOrUpdate() → 保存或者更新图书的方法
* 修改图书操作
  + 当我们点击修改图书时，首先我们需要根据当前图书的ID去数据库中将该图书的信息查询出来显示到编辑图书的页面book\_edit.jsp。
    - editBook() → 通过ID从数据库中查询到该图书显示到book\_edit.jsp页面。
* 当修改完图书之后我们需要把修改之后的图书信息提交到一个更新图书的方法中去，不过这个时候我们添加图书和修改图书的页面都是book\_edit.jsp，**那么我们怎么判断我们是在添加图书还是在修改图书呢？**
  + **方式一：**我们可以将添加图书和修改图书分离，再创建一个jsp页面
  + **方式二：**添加图书没有ID，但是修改图书有ID，所有我们可以通过是否有ID来判断当前是添加图书的操作还是修改图书的操作，不过我们需要把图书的ID放到一个隐藏域中。
    - **将之前的addBook()方法改为saveOrUpdate()方法**，在该方法中通过判断获取的图书ID是否为null来决定是调用BookService的saveBook方法还是update()方法。
* 至此，我们管理端图书的增删改查已经完成，但是我们在获取图书列表的方法中是将数据库中的所有图书都查询了出来，如果数据库中的图书过多在页面显示时就会出现滚动条，不但用户体验不好而且一次性从数据库中查询这么多条记录性能也差。所以**我们需要对图书进行分页显示**：
  + 在sql语句中使用limit关键字进行分页
  + 我们知道我们从数据库中查询出来的图书放到了一个List集合中，但是如果添加分页操作我们需要知道总页数、当前页码、总记录数等信息，但是List中并没有这些信息，所有我们需要**创建一个类来封装分页的信息**。



* Page<T>类
* private List<T> list; //从数据库中查询的集合
* public static final int ***PAGE\_SIZE*** = 5; //每页显示多少条记录
* private int pageNo; //当前页
* private int totalPageNo; //总页数
* private int totalRecord; //总记录数
* 修改BookDao：添加一个分页的方法
  + Page<Book> getPageBook(Page<Book> page) → 分页查找图书的方法
    - 该方法由BookService调用，传过来一个带有pageNo的Page<Book> 类型的page参数，通过该方法从数据库中先获取总的记录数，然后通过page的(pageNo-1)\*PAGE\_SIZE和PAGE\_SIZE获取一个带分页的List集合。给page设置了总的记录数和List后再将page返回。
* 修改BookService：同样添加一个分页的方法
  + Page<Book> getPageBook(String pageNo) → 分页查找图书的方法
    - 该方法由Servlet调用，接受的请求参数都是String类型的，所以传入的页码pageNo设置为了字符串类型。实现该方法是首先要创建一个Page<Book>对象，然后设置pageNo属性，不过这里需要将获取的字符串类型的pageNo转换为int类型。**注意：有可能会出现转换异常（例如用户输入的是字符串abc）**
* 在BookManagerServlet中添加getPageBook()方法
* **这时我们发现我们原来的bookList()已经没用了……**
  + 就会导致原来添加图书、修改图书、删除图书出现bug。
  + 改完bug之后考虑一个问题：目前我们执行添加图书、删除图书、修改图书之后都回到了图书列表的首页，**我们能不能让它回到执行之前的那个页面呢？（提示：我们可以通过请求头中的Referer获取之前的地址）**
* 关于分页页码的问题
  + 目前我们是将所有的页面都显示了出来，如果页面过多则用户体验不好，**我们能不能让它只显示5个页面呢？**
  + **[ [1](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=1)]  [2](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=2)  [3](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=3)  [4](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=4)  [5](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=5)**
  + **[1](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=1)  [[2](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=2)]  [3](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=3)  [4](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=4)  [5](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=5)**
  + **[1](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=1)  [2](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=2)  [[3](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=3)]  [4](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=4)  [5](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=5)**
  + **[2](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=2)  [3](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=3)  [[4](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=4)]  [5](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=5)  [6](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=6)**
  + **[3](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=3)  [4](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=4)  [[5](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=5)]  [6](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=6)  [7](http://localhost:8080/BookStore04/manager/BookManagerServlet?method=getPageBook&pageNo=7)**
  + **综上可知：一共有三种情况**
    1. **总页数小于5时**
    2. **总页数等于5，当前页小于等于3时**
    3. **总页数大于5，当前页大于3时**

**<!-- 设置两个变量 begin 和 end -->**

**<!-- 注意：当end大于总页数时这种情况 -->**

<c:choose>

<c:when test=*"*${page.totalPageNo < 5 }*"*>

<c:set var=*"begin"* value=*"1"*></c:set>

<c:set var=*"end"* value=*"*${page.totalPageNo }*"*></c:set>

</c:when>

<c:when test=*"*${page.pageNo <=3 }*"*>

<c:set var=*"begin"* value=*"1"*></c:set>

<c:set var=*"end"* value=*"5"*></c:set>

</c:when>

<c:otherwise>

<c:set var=*"begin"* value=*"*${page.pageNo - 2 }*"*></c:set>

<c:set var=*"end"* value=*"*${page.pageNo + 2 }*"*></c:set>

<c:if test=*"*${end > page.totalPageNo }*"*>

<c:set var=*"begin"* value=*"*${page.totalPageNo - 4 }*"*></c:set>

<c:set var=*"end"* value=*"*${page.totalPageNo}*"*></c:set>

</c:if>

</c:otherwise>

</c:choose>

* **分页完成之后如何复用分页**
  + 使页码超链接的请求地址动态显示
    - 获取请求地址
    - 获取查询字符串
    - 将请求地址与查询字符串使用?拼接
    - 给Page类添加一个path属性，将拼接好的地址设置到page对象的path属性中

### 客户端图书的显示

* 创建BookClientServlet
  + 添加getPageBook()方法获取分页图书信息。
* 根据用户输入的价格查询图书
  + 在BookDao中添加一个根据价格查询图书记录的方法
    - Page<Book> getPageBookByPrice(Page<Book> page,double min,double max);
* 在BookService中添加一个根据价格查询图书记录方法
  + Page<Book> getPageBookByPrice(String pageNo,String min,String max);
* 在BookClientServlet中添加getPageBookByPrice()方法
* 完成了查询按钮之后我们发现一个问题：
  + **点击下一页时查询条件丢失，如何实现带查询条件的分页呢：**
    - 由于我们查询操作的表单使用的是Post请求，请求参数在请求体中，所以获取请求参数时获取不到最低价格和最高价格。
    - 如果我们将post请求改为**get请求**，那么我们提交表单时action属性中的method请求参数将被覆盖，即method=getPageBookByPrice将丢失，所有我们决定将**getPageBookByPrice请求参数放到一个隐藏域中**
* **修改index.jsp页面**
* 将首页里的内容复制到另一个页面中
  + 删除首页的内容使用<jsp:forward>标签转发到获取图书列表的servlet

## 第五阶段：登录、登出 、验证码、购物车

### 登录、登出

* 登录
  + 登录成功后将用户保存到Session域中，根据Session域中是否含有用户信息来判断用户是否登录。
* 登出
  + 在UserServlet中添加一个logout方法，在该方法中获取Session对象然后强制Session失效即可。

### 验证码

* 使用Google提供的第三方jar包
  + 导入kaptcha-2.3.2.jar包
  + 在web.xml文件中注册Servlet
  + 在Session对象中获取验证码，**获取之后不要忘记移除**

### 购物车

* 购物车实现的三种方式：
  + 基于Cookie的，将购物信息都放入到Cookie中，由浏览器保存。
  + 基于Session的，将购物信息都放入到Session中，由服务器保存。√
  + 基于数据库表的，将购物信息都放入数据库中，由数据库保存。



购物项

购物车

* 创建购物项类CartItem
  + private Book book; //图书信息
  + private int count; //图书的数量
  + private double amount; // 购物项中图书的金额，通过计算得到
* 创建购物车类Cart
  + private Map<String, CartItem> map = new LinkedHashMap<String, CartItem>();
    - 保存购物项的Map，key是bookId，value是CartItem
    - 使用LinkedHashMap是为了保证购物车中添加图书的顺序
* private int totalCount;
  + 购物车中商品的总数量，通过计算得到
* private double totalAmount;
  + 购物车中商品的总金额，通过计算得到
* **另外还需要在购物车类中添加一些方法**：
  + public List<CartItem> getCartItems();
    - 获取购物车中所有的购物项，以便以后遍历
* public void addBook2Cart(Book book);
  + 向购物车中添加图书
    - **注意：需要判断购物车中是否已经存在该图书：如果存在，让数量加一；如果不存在，首先需要创建一个购物项，然后在购物项中setBook，设置数量为1，最后将该购物项添加到购物车的map中。**
* public void delCartItem(String bookId);
  + 删除一个购物项
* public void updateCartItem(String bookId, String countStr);
  + 更新购物项
    - 根据图书ID和用户输入的数量更新购物项中图书的数量
* public void emptyCart();
  + 清空购物车
* 处理购物车请求的CartServlet
  + 里面需要四个方法：添加图书、更新购物项、删除购物项、清空购物车

## 第六阶段：结账、添加事务、使用Ajax

### 结账

* 结账就是把我们购物车中的东西转换成订单，我们之前将购物车中的东西都放在了Session对象中，但是订单中的东西我们就需要**保存在数据库中**了，所以我们就需要创建类和表。
* 和购物车类似，我们需要创建订单类和订单项类
* 订单类Order
  + private String id; // 订单号
  + private Date orderTime; // 生成订单的时间
  + private int totalCount; // 商品总数量
  + private double totalAmount; // 商品总金额
  + private int state; // 订单状态 0：未发货 1：已发货 2：交易完成
  + private int userId; // 订单所属的用户
  + 对于订单号的要求：
    - 不能重复，唯一
    - 尽量少体现出订单信息
  + 订单和用户的关系
    - 多对一的关系：一个用户可以有多个订单，一个订单只能属于一个用户
* 订单项类OrderItem
  + private Integer id; // 订单项id
  + private int count; // 买了多少本图书
  + private double amount; // 买了count本图书花费的钱数
  + private String title; // 所买图书的书名
  + private String author; // 图书的作者
  + private double price; // 图书的价格
  + private String imgPath; // 图书封面
  + private String orderId; // 订单项所属的订单
  + 订单项和订单的关系
    - 多对一的关系：一个订单可以包含多个订单项，一个订单项属于一个订单
* 操作订单的接口OrderDao
  + public void saveOrder(Order order); // 向数据库中添加订单
  + public List<Order> getOrders(); //获取所有订单
  + public List<Order> getOrdersByUserId(int userId); //根据用户id查询用户的所有订单
  + public void updateOrderState(String orderId, int state); //根据用户id和状态码更新订单状态
* 操作订单项的接口OrderItemDao
  + public void saveOrderItem(OrderItem orderItem); //向数据库中添加订单项
  + public List<OrderItem> getOrderItemsByOrderId(String orderId); //根据订单号查询所有订单项
* 处理订单相关业务的接口OrderService
  + public String createOrder(Cart cart, User user); // 生成订单，返回订单号
    - **注意：这个方法里面的业务逻辑比较多**
      1. **根据当前时间和用户Id拼接订单号**
      2. **创建订单对象，将购物车中的信息封装到订单对象中并将订单保存到数据库中**
      3. **创建订单项对象，将购物车中购物项的信息封装到订单项中，其中不要忘记更新图书的销量和库存，然后将订单项保存到数据库中**
      4. **最后还需要清空购物车**
* public List<Order> getOrders(); //获取所有的订单
* public List<Order> getOrdersByUserId(int userId); //根据用户ID获取该用户的订单
* public List<OrderItem> getOrderItemsByOrderId(String orderId); //根据订单号获取所有订单项
* public void sendOrder(String orderId); //管理员发货
* public void takeOrder(String orderId); //用户确认收货
* **不管是发货还是收货，本质就是更新订单的状态**
* 客户端订单管理
  + 结账、我的订单、订单详情、确认收货
    - 以上操作都需要判断用户是否登录
* 管理端订单管理
  + 订单管理、发货