WEB三层架构与MVC 收藏

而我发此文的目的有二：一者，让初学者能够听到一家之言，是为解惑；二者，更希望抛砖引玉，得到专家的批判。

许多学生经常问我，MVC 到底和 WEB 三层架构有啥关系？ 开始时，我也只能给他们一些模糊的回答。时间长了，自己的良心开始受到谴责。对于一个程序员来说，这个问题显得挺学究。我在跟自己的许多程序员朋友以及同行(Java 讲师)都对 MVC 和 WEB 三层架构的关系做了探讨。现在可以说对 WEB 三层架构和 MVC 之间的关系理出了头绪。此可谓教学相长。

先说说 Web 三层架构这个古老话题。地球人都知道 web 三层架构是指:

- >用户接口层(UI Layer)
- >业务逻辑层(Bussiness Layer)
- >持久化层

关于业务逻辑和用户接口

在早期的 web 开发中，因为业务比较简单，并没有这三层的划分。用户数据的呈现及输入的接收、封装、验证、处理、以及对数据库的操作，都放在 jsp 页面中。这时的开发，好比盘古尚未开天辟地，整个 web 开发就是一片“混沌”。随着业务越来越复杂，人们开始考虑更好的利用 OOP 这把利刃来解决问题。于是有人发现把业务逻辑抽取出来并形成与显示和持久化无关的一层，能够让业务逻辑清晰，产品更便于维护。这就是 SUN 当初倡导的 JSP Model 1 开发方式。

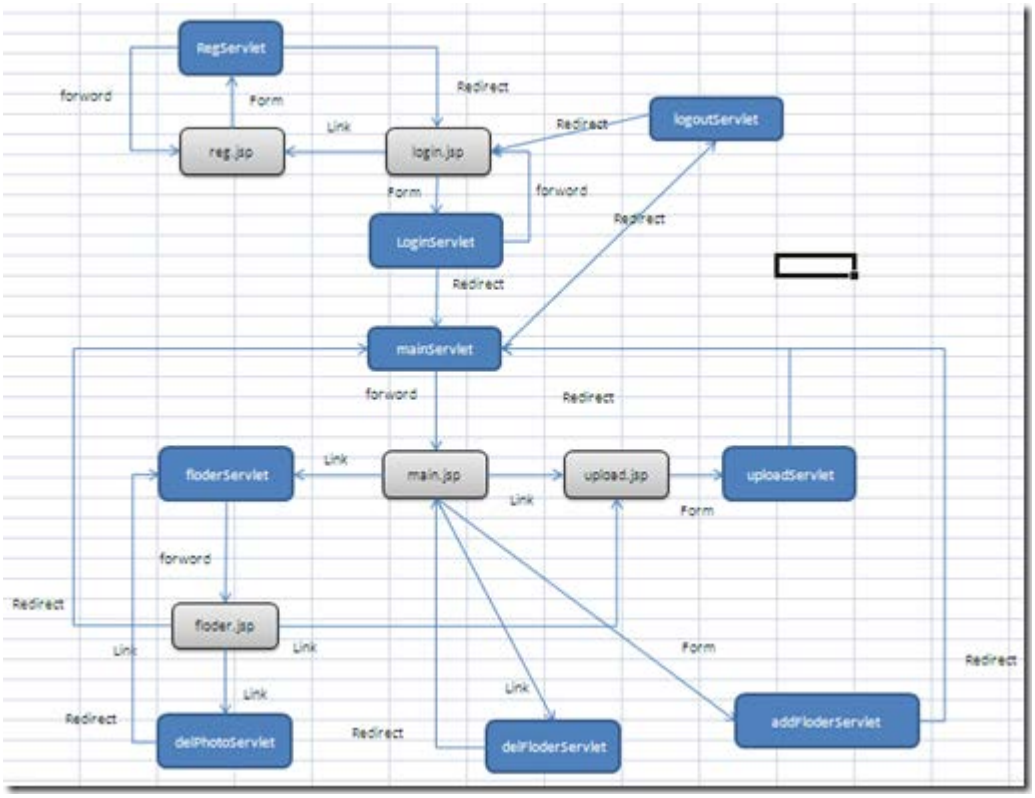
关于持久化

JSP M1 开发方式中，并没有对数据如何持久化给出建议。在许多公司中，它们的产品是以数据库为中心进行架构和设计的。在他们的产品里，虽然也有 DAO 层，但是职责不清。为什么这么说呢，因为我发现在许多人眼里，DAO 层的指责很简单——增删改查。但我认为，这样理解实际上是本末倒置了。对于简单数据的管理来说，这样理解无可厚非。但随着业务逻辑变得日益复杂。我们实在是被复杂的对象关系搞头疼了，如果这时我们还要考虑如何把数据存储起来(通常的情况下是存到关系型数据库中)，我们开始抱怨自己软件的架构太恶心，一团糟。面向对象设计思想教会我们——如果我们不想做这件事，就交给别人做吧！这时聪明的架构师们提出了一个概念——持久化。如果我们在自己的应用中添加一个新的层——专门负责对象状态的持久化保存及同步，那不就可以全心全意的“搞对象”了吗？持久化概念的产生，代表着我们对关系型数据库的依赖降低了。因此甚至有人推断——数据库已死。同时，关系型数据库这个新的概念也不断形成，并演化成理论，又由理论衍生出产品。因此一个意识良好的程序员，至少应该认同，持久化并不是产品中最重要的环节——最重要的环节是清晰正确的业务逻辑。

灰色地带

是的，从理论上看，web 三层架构很美了。但在实际开发产品的时候，我们发现了很多问题。主要问题就是用 ***UI 层***和**业务层**之间有许多灰色地带。这些灰色地带业务逻辑层不想管，UI 层也不想管。让我们举一些例子：

例子 1， 难以管理的页面跳转关系



上图是我在讲 JSP 课程时，一个简单案例的页面跳转关系图。这是一个十分简单的例子，但页面跳转关系已经挺复杂了。试想，如果你正在做一个有上百张表，十几个核心模块，几百个页面的产品时，这张图将变得多么复杂！而问题是，这些页面跳转关系分散在 JSP 和 Servlet 中，非常难以管理。

例子 2，表单数据的验证及封装：

假设我们正在做一个简单的表单提交，我们希望对用户数据的数据进行验证和封装，最终交给业务逻辑层一个实体对象。从三层架构分析，我们想要做的事情是这样的：

但是该把验证和封装数据的工作交给谁来做呢？UI 层还是业务逻辑层？都不太合适！

例子 3，国际化:

如果我们想为不同国家和地区的人提供不同的语言，无疑需要国际化的支持。那么，我们需要在 JSP 页面上根据用户的配置或请求信息判断应该为该用户呈现哪国文字。而这些判断和显示的逻辑应该划分到业务逻辑层还是 UI 层呢？

用 MVC 的思路解决问题

对于纠缠不清的问题，我们总要想办法将其分解。MVC 是一种设计思想。这种思想强调实现模型(Model)、视图(View)和控制器的分离。这种思想是如何作用于 web 的呢？实际上，我们在 web 开发中引入 MVC 思想，想要达到的目的是：实现 UI 层和业务逻辑层分离——控制器是为了实现上述目的而存在的！

在解决了持久化的问题后，我们发现，我们的所说的业务逻辑层和 MVC 中的 Model 指的是一回事，我们所说的 UI 层和 MVC 中的 View 是一回事。MVC 提供了让模型和视图相分离的思路——引入控制器。我们把页面跳转关系管理、表单数据的封装及验证、国际化等任务交给控制器处理。因此，也不难理解为什么流行的 MVC 框架都具有管理页面跳转关系、表单数据的封装及验证、国际化等特性。

总结

在 Java web 开发中，MVC 框架充当了 UI 层和业务逻辑层的适配器的作用。MVC 框架实现了 UI 层和业务逻辑层最大程度的分离。

使用 mvc 的好处，MVC 使用规则，java 三层架构设计思想

java 开发 web 应用

MVC 使用规则

为了提供可重用的设计及代码，M-V-C 之间的交互应该很好地定义，以及它们相互间地依赖关系要尽量最小。

使用 MVC 模式的其中一个目的就是，使一个单一的模型能与多个视图及控制器联合起来。MVC 模型保证了视图能与模型同步。当控制器从用户输入中接受到一个有效的命令后，它将调用模型上的相应方法。模型将确认该操作是否与当前的状态一致，然后再执行它，并相应地修改视图的状态。而视图，作为一个观察者，将根据得到的模型状态改变来更新它的显示。

依赖关系保持最小

为了使一个模型能在多个视图及控制器中使用，它们之间的依赖关系必须保持最小。要做到这些，必须遵守一下规则（如图 10-03）：

注意：A 依赖于 B，表示 A 的代码中需要与 B 相关的信息，如调用 B 的方法或使用 B 的属性。

1、 模型必须与视图及控制器没有任何依赖关系。

2、 视图依赖于与它相关的模型，它必须知道模型状态结构，这样才能把模型显示出来。

3、 视图不能依赖与控制器，这样的话，几个不同的控制器可以关联相同视图。

4、 控制器依赖于相关的模型及视图，模型定义了控制器能调用的方法，而视图定义上下关系，通过它控制器可以解释用户输入信息。这使得控制器能紧紧地跟视图联系在一起。

图 10-03 MVC 模式中允许的依赖关系

交互必须保持最小

另外一个需要使用多视图及多控制器的前提条件就是保存最小的交互。特别是，控制器一定不要直接影响与它相关联的视图的显示。而是在用户输入产生的影响在视图中可见之前，控制器必须与模型进行一个完整的往返交互，这样能保证一个状态修改能更新所有的视图，以及视图能保持与模型同步。使用单一控制器的实现通常会违反这种规则，因为一些不够严谨的思维：“我已经知道这个状态修改要发生，所以不需要模型来告诉我这个”。但这是错的，原因有三个：

1、 模型可能因为某些原因否决该操作，然后这个操作就不会发生了。

2、 其他控制器可能同时调用了该模型的操作，有些操作可能也会影响视图的显示，或者使操作不合法。

3、 另外，将来使用其他控制器来扩展这个实现也是可能的。

MVC 模式是"Model-View-Controller"的缩写，中文翻译为"模式-视图-控制器"。MVC 应用程序总是由这三个部分组成。Event(事件)导致 Controller 改变 Model 或 View，或者同时改变两者。只要 Controller 改变了 Models 的数据或者属性，所有依赖的 View 都会自动更新。类似的，只要 Controller 改变了 View，View 会从潜在的 Model 中获取数据来刷新自己。MVC 模式最早是 smalltalk 语言研究团提出的，应用于用户交互应用程序中。smalltalk 语言和 java 语言有很多相似性,都是面向对象语言，很自然的 SUN 在 petstore(宠物店)事例应用程序中就推荐 MVC 模式作为开发 Web 应用的架构模式。MVC 模式是一种架构模式，其实需要其他模式协作完成。在 J2EE 模式目录中，通常采用 service to worker 模式实现，而 service to worker 模式可由集中控制器模式，派遣器模式和 Page Helper 模式组成。而 Struts 只实现了 MVC 的 View 和 Controller 两个部分，Model 部分需要开发者自己来实现，Struts 提供了抽象类 Action 使开发者能将 Model 应用于 Struts 框架中。

MVC 模式是一个复杂的架构模式，其实现也显得非常复杂。但是，我们已经终结出了很多可靠的设计模式，多种设计模式结合在一起，使 MVC 模式的实现变得相对简单易行。Views 可以看作一棵树，显然可以用 Composite Pattern 来实现。Views 和 Models 之间的关系可以用 Observer Pattern 体现。Controller 控制 Views 的显示，可以用 Strategy Pattern 实现。Model 通常是一个调停者，可采用 Mediator Pattern 来实现。

现在让我们来了解一下 MVC 三个部分在 J2EE 架构中处于什么位置，这样有助于我们理解 MVC 模式的实现。MVC 与 J2EE 架构的对应关系是:View 处于 Web Tier 或者说是 Client Tier，通常是 JSP/Servlet，即页面显示部分。Controller 也处于 Web Tier，通常用 Servlet 来实现，即页面显示的逻辑部分实现。Model 处于 Middle Tier，通常用服务端的 javaBean 或者 EJB 实现，即业务逻辑部分的实现。

一、MVC 设计思想

MVC 英文即 Model-View-Controller，即把一个应用的输入、处理、输出流程按照 Model、View、Controller 的方式进行分离，这样一个应用被分成三个层——模型层、视图层、控制层。

视图(View)代表用户交互界面，对于 Web 应用来说，可以概括为 HTML 界面，但有可能为 XHTML、XML 和 Applet。随着应用的复杂性和规模性，界面的处理也变得具有挑战性。一个应用可能有很多不同的视图，MVC 设计模式对于视图的处理仅限于视图上数据的采集和处理，以及用户的请求，而不包括在视图上的业务流程的处理。业务流程的处理交予模型(Model)处理。比如一个订单的视图只接受来自模型的数据并显示给用户，以及将用户界面的输入数据和请求传递给控制和模型。

模型(Model)：就是业务流程/状态的处理以及业务规则的制定。业务流程的处理过程对其它层来说是黑箱操作，模型接受视图请求的数据，并返回最终的处理结果。业务模型的设计可以说是 MVC 最主要的核心。目前流行的 EJB 模型就是一个典型的应用例子，它从应用技术实现的角度对模型做了进一步的划分，以便充分利用现有的组件，但它不能作为应用设计模型的框架。它仅仅告诉你按这种模型设计就可以利用某些技术组件，从而减少了技术上的困难。对一个开发者来说，就可以专注于业务模型的设计。MVC 设计模式告诉我们，把应用的模型按一定的规则抽取出来，抽取的层次很重要，这也是判断开发人员是否优秀的设计依据。抽象与具体不能隔得太远，也不能太近。MVC 并没有提供模型的设计方法，而只告诉你应该组织管理这些模型，以便于模型的重构和提高重用性。我们

可以用对象编程来做比喻，MVC 定义了一个顶级类，告诉它的子类你只能做这些，但没法限制你能做这些。这点对编程的开发人员非常重要。

业务模型还有一个很重要的模型那就是数据模型。数据模型主要指实体对象的数据 保存（持续化）。比如将一张订单保存到数据库，从数据库获取订单。我们可以将这个模型单独列出，所有有关数据库的操作只限制在该模型中。

控制(Controller)可以理解为从用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。划分控制层的作用也很明显，它清楚地告诉你，它就是一个分发器，选择什么样的模型，选择什么样的视图，可以完成什么样的用户请求。控制层并不做任何的数据处理。例如，用户点击一个连接，控制层接受请求后，并不处理业务信息，它只把用户的信息传递给模型，告诉模型做什么，选择符合要求的视图返回给用户。因此，一个模型可能对应多个视图，一个视图可能对应多个模型。

模型、视图与控制器的分离，使得一个模型可以具有多个显示视图。如果用户通过某个视图的控制器改变了模型的数据，所有其它依赖于这些数据的视图都应反映到这些变化。因此，无论何时发生了何种数据变化，控制器都会将变化通知所有的视图，导致显示的更新。这实际上是一种模型的变化-传播机制。模型、视图、控制器三者之间的关系和各自的主要功能，如图 1 所示。

二、MVC 设计模式的实现

ASP.NET 提供了一个很好的实现这种经典设计模式的类似环境。开发者通过在 ASPX 页面中开发用户接口来实现视图；控制器的功能在逻辑功能代码(.cs)中实现；模型通常对应应用系统的业务部分。在 ASP.NET 中实现这种设计而提供的一个多层系统，较经典的 ASP 结构实现的系统来说有明显的优点。将用户显示（视图）从动作（控制器）中分离出来,提高了代码的重用性。将数据（模型）从对其操作的动作（控制器）分离出来可以让你设计一个与后台存储数据无关的系统。就 MVC 结构的本质而言，它是一种解决耦合系统问题的方法。

2.1 视图

视图是模型的表示，它提供用户交互界面。使用多个包含单显示页面的用户部件，复杂的 Web 页面可以展示来自多个数据源的内容，并且网页人员，美工能独自参与这些 Web 页面的开发和维护。

在 ASP.NET 下，视图的实现很简单。可以像开发 WINDOWS 界面一样直接在集成开发环境下通过拖动控件来完成页面开发本。本文中介绍每一个页面都采用复合视图的形式即：一个页面由多个子视图(用户部件)组成；子视图可以是最简单 HTML 控件、服务器控件或多个控件嵌套构而成的 Web 自定义控件。页面都由模板定义，模板定义了页面的布局，用户部件的标签和数目，用户指定一个模板，平台根据这些信息自动创建页面。针对静态的模板内容，如页面上的站点导航，菜单，友好链接，这些使用缺省的模板内容配置；针对动态的模板内容(主要是业务内容)，由于用户的请求不同，只能使用后期绑定，并且针对用户的不同，用户部件的显示内容进行过滤。使用由用户部件根据模板配置组成的组合页面，它增强了可重用性，并原型化了站点的布局。

视图部分大致处理流程如下：首先，页面模板定义了页面的布局；页面配置文件定义视图标签的具体内容（用户部件）；然后，由页面布局策略类初始化并加载页面；每个用户部件根据它自己的配置进行初始化，加载校验器并设置参数，以及事件的委托等；用户提交后，通过了表示层的校验，用户部件把数据自动提交给业务实体即模型。

这一部分主要定义了 WEB 页面基类 PageBase；页面布局策略类 PageLayout，完成页面布局，用于加载用户部件到页面；用户部件基类 UserControlBase 即用户部件框架，用于动态加载检验部件，以及实现用户部件的个性化。为了实现 WEB 应用的灵活性，视图部分也用到了许多配置文件例如：置文件有模板配置、页面配置、路径配置、验证配置等。

2.2 控制器

为了能够控制和协调每个用户跨越多个请求的处理，控制机制应该以集中的方式进行管理。因此，为了达到集中管理的目的引入了控制器。应用程序的控制器集中从客户端接收请求（典型情况下是一个运行浏览器的用户），决定执行什么商业逻辑功能，然后将产生下一步用户界面的责任委派给一个适当的视图组件。

用控制器提供一个控制和处理请求的集中入口点，它负责接收、截取并处理用户请求；并将请求委托给分发者类，根据当前状态和业务操作的结果决定向客户呈现的视图。在这一部分主要定义了 HttpReqDispatcher(分发者类)、HttpCapture(请求捕获者类)、Controller(控制器类)等，它们相互配合来完成控制

器的功能。请求捕获者类捕获 HTTP 请求并转发给控制器类。控制器类是系统中处理所有请求的最初入口点。控制器完成一些必要的处理后把请求委托给分发者类；分发者类分发者负责视图的管理和导航，它管理将选择哪个视图提供给用户，并提供给分发资源控制。在这一部分分别采用了分发者、策略、工厂方法、适配器等设计模式。

为了使请求捕获者类自动捕获用户请求并进行处理，ASP.NET 提供低级别的请求/响应 API，使开发人员能够使用 .NET 框架类为传入的 HTTP 请求提供服务。为此，必须创作支持 System.Web.IHTTPHandler 接口和实现 ProcessRequest() 方法的类即：请求捕获者类，并在 web.config 的 ＜httphandlers＞ 节中添加类。ASP.NET 收到的每个传入 HTTP 请求最终由实现 IHTTPHandler 的类的特定实例来处理。IHttpHandlerFactory 提供了处理 IHttpHandler 实例 URL 请求的实际解析的结构。HTTP 处理程序和工厂在 ASP.NET 配置中声明为 web.config 文件的一部分。ASP.NET 定义了一个 ＜httphandlers＞ 配置节，在其中可以添加和移除处理程序和工厂。子目录继承 HttpHandlerFactory 和 HttpHandler 的设置。 HTTP 处理程序和工厂是 ASP.NET 页框架的主体。工厂将每个请求分配给一个处理程序，后者处理该请求。 例如，在全局 machine.config 文件中，ASP.NET 将所有对 ASPx 文件的请求映射到 HttpCapture 类：

```
＜httphandlers＞
...
...
＜/httphandlers＞
```

2.3 模型

MVC 系统中的模型从概念上可以分为两类——系统的内部状态和改变系统状态的动作。模型是你所有的商业逻辑代码片段所在。本文为模型提供了业务实体对象和业务处理对象：所有的业务处理对象都是从 ProcessBase 类派生的子类。业务处理对象封装了具体的处理逻辑，调用业务逻辑模型，并且把响应提交到合适的视图组件以产生响应。业务实体对象可以通过定义属性描述客户端表单数据。所有业务实体对象都 EntityBase 派生子类对象，业务处理对象可以直接对它进行读写，而不再需要和 request、response 对象进行数据交互。通过业务实体对象实现了对视图和模型之间交互的支持。实现时把"做什么"（业务处理）和"如何做"（业务实体）分离。这样可以实现业务逻辑的重用。由于各个应用的具体业务是不同的，这里不再列举其具体代码实例。

三、MVC 设计模式的扩展

通过在 ASP.NET 中的 MVC 模式编写的，具有极其良好的可扩展性。它可以轻松实现以下功能：

①实现一个模型的多个视图；

②采用多个控制器；

③当模型改变时，所有视图将自动刷新；

④所有的控制器将相互独立工作。

这就是 MVC 模式的好处，只需在以前的程序上稍作修改或增加新的类，即可轻松增加许多程序功能。以前开发的许多类可以重用，而程序结构根本不再需要改变，各类之间相互独立，便于团体开发，提高开发效率。下面讨论如何实现一个模型、两个视图和一个控制器的程序。其中模型类及视图类根本不需要改变，与前面的完全一样，这就是面向对象编程的好处。对于控制器中的类，只需要增加另一个视图，并与模型发生关联即可。该模式下视图、控制器、模型三者之间的示意图如图 2 所示。

同样也可以实现其它形式的 MVC 例如：一个模型、两个视图和两个控制器。从上面可以看出，通过 MVC 模式实现的应用程序具有极其良好的可扩展性，是 ASP.NET 面向对象编程的未来方向。

四、MVC 的优点

大部分用过程语言比如 ASP、PHP 开发出来的 Web 应用，初始的开发模板就是混合层的数据编程。例如，直接向数据库发送请求并用 HTML 显示,开发速度往往比较快,但由于数据页面的分离不是很直接,因而很难体现出业务模型的样子或者模型的重用性。产品设计弹性力度很小，很难满足用户的变化性需求。MVC 要求对应用分层，虽然要花费额外的工作，但产品的结构清晰，产品的应用通过模型可以得到更好地体现。

首先，最重要的是应该有多个视图对应一个模型的能力。在目前用户需求的快速变化下，可能有多种方式访问应用的要求。例如，订单模型可能有本系统的订单，也有网上订单，或者其他系统的订单，但对于订单的处理都是一样，也就是说订单的处理是一致的。按 MVC 设计模式，一个订单模型以及多个视图即可解决问题。这样减少了代码的复制，即减少了代码的维护量，一旦模型发生改变，也易于维护。 其次，由于模型返回的数据不带任何显示格式，因而这些模型也可直接应用于接口的使用。

再次，由于一个应用被分离为三层，因此有时改变其中的一层就能满足应用的改变。一个应用的业务流程或者业务规则的改变只需改动 MVC 的模型层。

控制层的概念也很有效，由于它把不同的模型和不同的视图组合在一起完成不同的请求，因此，控制层可以说是包含了用户请求权限的概念。

最后，它还有利于软件工程化管理。由于不同的层各司其职，每一层不同的应用具有某些相同的特征，有利于通过工程化、工具化产生管理程序代码。

五、MVC 的不足

MVC 的不足体现在以下几个方面：

（1）增加了系统结构和实现的复杂性。对于简单的界面，严格遵循 MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率。

（2）视图与控制器间的过于紧密的连接。视图与控制器是相互分离，但确实联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

（3）视图对模型数据的低效率访问。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

（4） 目前，一般高级的界面工具或构造器不支持 MVC 模式。改造这些工具以适应 MVC 需要和建立分离的部件的代价是很高的，从而造成使用 MVC 的困难。

自己理解的 J2EE 三层架构（与软件设计模式的联系区别）

(2009-04-13 09:37:50)

转载

标签：

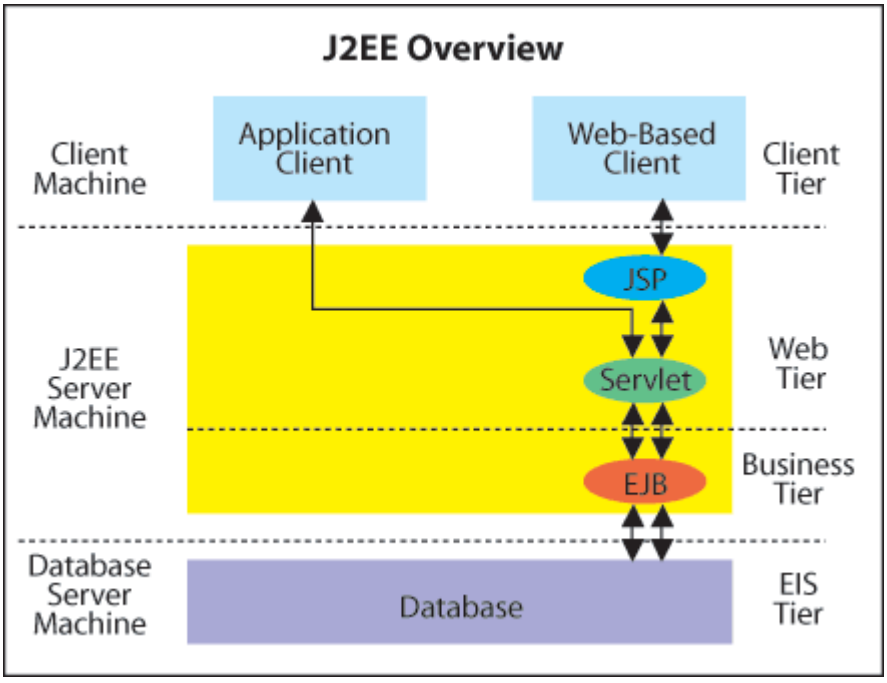
软件设计模式

dao

j2ee

表示层

杂谈



如上图

1.J2EE 分 3 层：
服务器端业务逻辑（有业务逻辑层和持久化数据层,Businness Tier 和 EIS Tier）、服务器端表示层(Web Tier)及客户端表示层（Client Tier）

可以将 J2EE 设计模式归纳到 6 个类别
（1）表示层体系结构模式（服务器端表示层）
a.前端控制器模式
b.MVC 模式

c.装饰器模式

（2）表示层高级体系结构模式
a.复合视图模式（在服务器端表示层）
b.视图助手模式
c.服务工作者模式

（3）表示层伸缩性模式（服务器端表示层）
a.异步页面模式
b.缓存过滤器模式
c.资源池模式

（4）业务层模式（服务器端业务逻辑层）
a.复合实体模式
b.域对象模式（业务数据层）

（5）数据传递模式（用于业务逻辑层和表示层之间）
a.DTO 模式
b.DTO 传递散列模式
c.数据库行集合 DTO 模式

（6）数据库模式（服务器持久化数据层）
a.DAO 模式
b.DAO 工厂模式

2.前端控制器模式
常用的应用为，一个 servlet 作为一个前端控制器，它负责根据页面的请求，然后转发到页面控制器。页面控制器也是一个 servlet，这个 servlet 电泳关于这个请求所应该执行的业务逻辑，并根据业务逻辑的结果返回到具体的现实页面。简化的使用是前端控制器 servlet 利用“命令模式”将页面控制器转化成一个个更细粒度的类。

AcitonServlet 是前端控制器,部分代码

```
RequestUtils.selecetModule(request,getServletContext());
getRequestProcessor(getModuleConfig(request)).process(request,response);
```

RequestProcessor 的 process 方法处理公共任务，部分代码

```
if(!processPreprocess(request,response)){
 return;
}
```

processXXX 方法都在处理公共的动作

RequestProcessor 的 processActionPerform 方法实现命令模式，该方式将具体请求的动作分配到 Action。
部分代码

```
return(action.execute(mapping,form,request,response));
```

前端控制器选择页面控制器解析用户的请求，实现具体业务逻辑，并根据结果转发到页面试图

3.装饰器模式

（1）设计模式的的装饰器模式

装饰器模式为客户端提供一个透明扩充某实例功能的方法，该方法的返回类型就是这个实例，在客户端不断调用这个方法的同时，该实例的内部表象已经随着功能改变完全不同。

（2）J2EE 设计模式中的装饰器模式

Servlet 支持装饰器模式，装饰一个 request 请求，利用装饰器类截获所有发送给目标对象的请求，并执行需要的工作，完成之后再把请求转发到下一个装饰器，知道没有了装饰器，最后将请求发送到 Servlet

Servlet 利过滤器来拦截请求和响应，在请求到达 Servlet 前，为这个请求做一个些额外处理。处理器可以被看成一个链，每个过滤器之间都能互相传递，以下是过滤器 Filter 接口的 doFilter 有三个参数，ServletRRequest,ServletResponse,FilterChain，利用 FilterChain 的 doFilter() 激活下一个相关的过滤器

```
public void doFilter(ServletRequest request,ServletResponse response,FilterChain
chain)throws          IOException,ServletException{
//用secletEncoding方法设置字符编码
 if(ingnore || request.getCharacterEncoding() == null){
  String encoding = selectEncoding(request);
  if(encoding != null)
   request.setCharacterEncoding(encoding);
 }
 chain.doFilter(request,response);
}
```

4.复合视图模式

设计模式中的“合成模式”

合成模式：提供一个树状的对象结构，树枝类与树叶类都实现同一个接口，以便客户端在调用任何对象时都只要需要调用该树状结构的根接口就可以了。

J2EE 设计模式中的“复合视图模式”

将视图的布局从中抽离出来，形成由一系列通用组件的模板。可以利用 XML 来描述视图的组成

5.视图助手模式

jsp 页面的标签库和 struts 的标签库，一个标签应该继承 javax.servlet.jsp.tagext.TagSupport，并给出 doStarTag 和 doEndTag 两个方法的实现。

doStarTag 实现业务逻辑

doEndTag 控制输出

6.服务工作者模式
将页面流转、前端控制器模式，视图助手模式合在一起使用，表示“请求-转发-视图”的一整套流程。该模式也是 MVC 模式的实现标准，struts 也基于这个模式实现

7.异步页面模式
当远程数据发生变化时，将其缓存下来，称为“发布者-订阅者-模型”。在 J2EE 的功能是，利用一个订阅者角色，在一个的时间间隔或数据发生变化时，接受来自发布者角色的数据，订阅者角色同时会利用模式曾来更新数据库。这样的工作累世于软件设计模式中的“观察者模式”。常见的应用为，当发布服务器需要显示最新信息的 HTML 页面时，会利用一个订阅者角色来负责。例如 ActionServlet

8.缓存过滤器模式
这个模式用来椽村动态产生的页面，尽可能减少重复生成的页面。在 J2EE 的功能是，利用一个缓存过滤器截获请求，判断该请求所返回的页面是否有缓存。缓存过滤器应该放在“装饰过滤器”和工作 Servlet 之前。缓存过滤器是装饰过滤器的一个变体。对 HTTPServletResponse 对象进行装饰来保存请求处理的结果。

9.资源池模式
客户端在需要 JDBC 连接时，应该从一个池中去取得。如果池中有可用的 JDBC 连接，则返回这一对象资源。如果没有任何可用资源，但池中还有容量，就使用工厂生成一个新的实例。如果池中没有任何可用资源且池的容量已经沾满，那就必须等待，知道其他客户端还回至少一个对象。

10.复合实体模式
该模式可以降低工作环境中的复杂性和通信开销。一个复合实体将来自各种不同来源的数据集中到一个单独的对象中。应用为 EJB 环境的集中对象。

11.域对象模式
将一张数据库中的表结构对象化，例如 hibernate 中的表转化成对象，使用在持久层框架理论中

12.DTO 模式
struts 的 ActionForm 就是一个 DTO 模式的实现，从页面视图得到数据传递给模型层，模型层通过业务逻辑的调用后将返还数据给 ActionForm 用作视图层的数据显示
ActionForm 仅仅被用在从视图层传递到模型层。

13.DTO 散列模式
struts 的 DynaActionForm 就是一个 DTO 散列模式，让程序员设置某个数据的主键，而后在 Acton 中可以通过该主键得到页面数据

14.数据库行集合 DTO 模式
将 JDBC 的 ResultSet 发送到客户端显示，使用 ResultSet 接口的一个独立类作为 DTO 发送到客户端显示。

15.DAO 模式
数据访问对象模式被认为是持久层的一种基本模式
DAO 模式有反问数据和处理业务逻辑的功能，现在不再流行

16.DAO 工厂模式

用户只对被创建的产品感兴趣，而这些被创建的产品在创建之前所做的许多额外的工作被封装到工厂接口的子类中，而不适用具体产品类的构造函数，达到隐式的使用

改模式只有和数据库连接的功能，没有实现访问数据的功能

17.J2EE 设计模式与设计模式的区别

（1）软件设计模式是设计，J2EE 设计模式关注的就是构架

（2）软件设计模式为了解决各种软件世界中常见问题提炼的一种最佳时间，是许多经验丰富的软件设计者不断成功和失败的总结。

J2EE 模式为了解决企业级应用的构架问题。

（3）软件设计模式的主要目的是解耦，可以在 J2EE 模型的任何层中出现软件设计模式。J2EE 设计模式对于 J2EE 模型中的分层进行解耦。软件设计模式关注的是微观的方法学，J2EE 设计模式则是宏观的方法学。

18.

（1）前端控制器模式、MVC 模式、服务工作者模式被整合成了表示层框架，如 struts 框架，webwork 框架

（2）复合视图模式有 Tiles 框架实现

（3）视图助手模式的主要实现是标签库，JSTL 标签库

（4）DAO 工厂模式随着中间层框架 Spring 的依赖注入，已经不一定需要实现。

下面摘自网友的文章：

一、MVC 架构

Struts 是一个不错的 MVC 架构，我一直以来都用它，通过简单的配置即可将 view,controler,和 Model 结合起来。View 主要以 JSP 来实现，因为它是面向标签的，所以对于网页设计人员提供了很好的接口。FormBean 是介于 JSP 和 Action 之间的中间数据载体，它肩负着数据从 JSP 到 ACTION 的传递过程。Action 是流程的中转站，不同的业务在不同的 Action 中以不同的 Model 调用来实现。Model 就是实现具体业务的操作过程，不过这种过程是一种在较高水平上的实现。

总之，MVC 架构实现了三层结构中的两层，即表现层和业务层，另外还有一层被称之为持久化层。

二、三层架构

正如以上所说的，三层架构即“表现层”，“业务层”，“持久化层”。表现层实现的代表作品是 Struts 框架，业务层实现的代表作品是 Spring，持久层实现的代表作品是 Hibernate。不过在我的开发中 Spring 被省掉了，因为我认为业务过于简单，还不至于用 Spring 来实现。下面我将具体的来说说我的三层实现。

1、三种 Bean

在我的实现中，总共有三种 Bean,它们都是为保存对象状态而存在的。持久层中，这种 Bean 就是 POJO 对象。它是面向数据库的，或者说多少得照顾到数据库的实现，因为我习惯于用 PowerDesigner 来设计数据库，做出来的结构也是比较中规中矩的，而如果直接用 Hibernate 来实数据库的话，就不那么干净了。所以 POJO 对象的设计就不能完全面向对象。业务层中的 Bean 我把它称之为 Entity,这是一个完全面向程序逻辑的 Bean。它可能是好几个 POJO 的集合。对于业务逻辑来，这种被称之为 Entity 的 Bean 做到了“拿来就用”，很便于业务的进

行。在显示层中的 Bean 就是 FormBean 了，主要用于传入数据所用。
POJO 仅生存于持久化层，到了业务层就将数据交给 Entity 了。而 Entity 不仅生存于业务层，它还被传到了 JSP 中，用于显示。
2、Pojo 与 Dao
我认为这是数据与操作的关系，Dao 只在呼操作，而 Pojo 仅用于保存数据。下面是我 Dao 接口的实个现。

```
public interface Dao {
 /
  public void resetPO(Pojo pojo) {
    this.userPo = (UserPo)pojo;
  }


  public String save() {
   String oid = null;
   try{
     //Session session = HibernateUtil.currentSession()  已经被提至构造函数中初始化了
     Transaction tx = session.beginTransaction();
     if (userPo.getId() == null) {
       oid = (String)session.save(userPo);//如果这是一个新的pojo则进行insert操作。
       session.flush();
       tx.commit();
     }else{
       session.update(userPo,userPo.getId());//如果该pojo已被初始化过则进行update操作
       session.flush();
       tx.commit();
     }
   }catch(HibernateException ex){
     System.out.println("//\n UserDao.save()出现异常！ ");
     log.error("UserDao.save()出现异常！ ",ex);
   }
   return oid;
 }


 public Pojo load() {
   UserPo tmp = new UserPo();
   try{
     //Session session = HibernateUtil.currentSession()  已经被提至构造函数中初始化了
     tmp = (UserPo) session.get(UserPo.class, userPo.getId()); //用确切存在的ID值获得对象
   }catch(HibernateException hbe){
     hbe.printStackTrace();
   }
   if (tmp != null) {
```

```
    userPo = tmp;
    return userPo;
   }
   else
    return null;
  }

  public List find(short by) {
   //Session session = HibernateUtil.currentSession()  已经被提至构造函数中初始化了
   Query query = null;
   String where=null;
   switch(by){
    case 0 :
     where = "";
     break;
    case 1 :
     where = " where us.name='"+userPo.getName()+"'";
     break;
    case 2 :
     where = " where us.account='"+userPo.getAccount()+"'";
     break;
    case 3 :
     where = " where us.password='"+userPo.getPassword()+"'";
     break;
    case 4 :
     where = " where us.account='"+userPo.getAccount()+"' and
us.password='"+userPo.getPassword()+"'";
   }
   query = session.createQuery("from UserPo us"+where);
   return query.list();
  }
  public void close(){
   HibernateUtil.closeSession();
  }
}
```

其中HibernateUtil是一个Hibernate方法类，它提供线程安全的Session。和关闭这个Session的方法（虽然关闭Session过于简单）。每个Dao是由DaoFactory来产生的。DaoFactory也有一个公共的接口。如下：

```
public interface DaoFactory {

  Dao getDocumentDAO();
  Dao getDocHeadDAO();
```

```
  Dao getDocAccessoryDAO();
  Dao getUserDao(User u);
}
```

下面是一个基于 Hibernate 的 Dao 的实现。

```
public class HbDaoFactory implements DaoFactory {
  public HbDaoFactory() {
  }

  public Dao getDocumentDAO() {

    return new com.cecs.dao.DocumentDAO();
  }

  public Dao getDocHeadDAO() {

    throw new java.lang.UnsupportedOperationException("Method getDocHeadDAO() not yet
implemented.");
  }

  public Dao getDocAccessoryDAO() {

    throw new java.lang.UnsupportedOperationException("Method getDocAccessoryDAO() not
yet implemented.");
  }

  public Dao getUserDao(User u) {
    return new UserDao(u);
  }
}
```

这样一但不用 Hibernate 来实现持久层，也可以很方便改为其它的 Dao 而不用修改业务层。