序言:步履江湖观大略

——从宏观上了解 Web 应用程序

现实因素是改变一切的根源——王兴刚。

进入九十年代中后期,随着互联网的兴起，人们强烈希望应用程序的使用，能够像浏览网页一样愉快。而传统的 C/S 结构应用程序，要求每一个使用者，必须在自己的计算机上安装一个客户端软件，没有这个软件，就没有办法访问位于服务器上的应用程序。因此 C/S 结构的笨拙，与广大人民群众日以增长的简单需求之间，产生了深刻的现形矛盾，因此，也就无法适应这个时代的历练，从而，渐渐地退出了程序开发的历史舞台。

应用程序与 C/S 结构一揖而别，转而投向了 B/S 的怀抱。应用程序的开发迎来了 Web 时代。作为 Web 应用程序领头雁的 java，最开始的 web 应用，是以 Applet（小应用程序）的方式出现的。当时的应用形式是，在服务器上发布的小应用程序，当被浏览器访问时，被下载到客户的计算机上运行。这就造成了安全上的隐患。比如，如果 Applet（小应用程序）内部包含有数据库的 Url，用户名，密码等敏感信息，当 Applet 被下载到本地以后，随便找个反编译工具，很容易地就能获取到 Applet 的原代码，甚至比一些垃圾程序员书写的程序更符合编码规范。当此之时，只要不是傻子都能看明白数据库的连接信息。这就等于告诉小偷：“我刚取了 50 万银子在家里放着呢，家里没有人，门没有锁，我也没有养狗的习惯！”。如果小偷不到你家去做客，那是他人品有问题。

仔细分析 Applet 的不安全性，我们发现，原因在于 Applet 是在客户端运行的。基于安全性考虑，人们希望 java 程序能从客户端走向服务器端。程序在服务器上运行，不被下载到客户端计算机上。此时出现的第一门技术是 Servlet，它很好地满足了人们对 java 安全性的要求。但是接下来的问题是，由于 Servlet 对所有用于客户端显示的网页代码，都要在服务器端完成，也就是 Servlet 不但要完成服务器端流程的调度，还要负责页面的实现，而原本简洁的 Html 一旦在 Servlet 内部完成，就变得极其繁琐，这给开发过程带来了新的麻烦，严重影响了开发效率，从而影响了 Servlet 的大面积推广使用。

此后不久，Sun 也终于认识到自己有点太 CNN 了，于是出于简化 Servlet 应用的目的，推出了新一代 Web 应用技术规范，这就是 JSP 规范。JSP 通过在 html 内部嵌入 java 代码的方式，很好的解决了 Servlet 对页面显示的实现过于复杂这个难题，让 web 程序开发进入了一个崭新的时代。然而，倒霉的 Sun 公司又陷入了新的问题之中。什么问题哪，我们需要好好地思考软件开发过程中的“逻辑问题”。

就个人观点而言，一个 web 应用程序应该包括三个方面的逻辑，这就是：显示逻辑－－负责页面的展现；控制逻辑－－负责流程的调度，也就是调用哪个类进行数据处理，数据运算，调用哪个页面，将运算生成的数据显示出来。业务逻辑－－这里指的就是复杂繁琐的商业运算和数据处理了。

通过对 JSP 的分析，我们发现，其 Html 部分应该负责的是页面显示的内容，属于显示逻辑的范畴，而 java 脚本，主要负责的是数据运算以及数据处理，应该是属于业务逻辑的范畴。而 Jsp 没有对这两种逻辑进行很好的划分，从而造成显示逻辑与业务逻辑交织在了一起（这就是所谓的耦合度过高），给程序的测试过程以及后期的维护造成了很大的不便。于是，Sun 公司的伟大程序员们，开始伴随着使用者的抱怨和缭绕的烟雾，进行了深刻且细腻的思考。

后来有一天，当一群程序员在苹果树下放松神经的时候，忽然间，来了一阵大风，满树的苹果像冰雹一样砸到了他们的脑袋上，也砸开了他们的视野，新的应用方式诞生了，从此真正开始了 java 在 web 应用程序开发过程中的辉煌之旅，并且一直到今天，依然笑傲江湖，环球不败！那么他们思考的是什么哪？

他们发现，针对三种逻辑，JSP 的强项在于页面显示，可以用于处理显示逻辑；Servlet 的强项是流程调度，可以将 JSP 中大量用于流程调度的语句提取出来，转而交给 Servlet 来处理，这样控制逻辑的问题也很好地解决了；那么业务逻辑怎么办哪？JavaBean 除了用来封装数据，也可以有方法的啊，为什么不用 JavaBean 的方法来处理业务逻辑哪，没有理由啊！那就用呗。于是乎，业务逻辑的处理问题，也找到了合适的解决方案（后来，随着编程思想和变成艺术的提高，JavaBean 与普通 java 类之间的关系日渐模糊，人们也更提倡使用没有任何规范的普通 java 类来处理业务逻辑，并且给他起了个名字，以便更好地提倡这种思想，这就是 POJO－－普通 java 类）。于是三种逻辑，各得其所。此后，围绕三种逻辑，在不同复杂程度的业务系统中的应用，产生了两种模式：Model1（JSP＋JavaBean）和 Model2（JSP+Servlet＋JavaBean）。Model1 比较适合于小规模的系统，比如网站等。而针对复杂的企业级应用（比如金融系统，社保系统等），Sun 公司推荐使用 Model2。

Model2 出现以后，人们于实际应用过程中发现，其中的 jsp 关注的是页面显示，Servlet 关注的是流程控制，而 JavaBean 关注的是业务处理。这与传统的 MVC 非常类似，传统的 MVC 包括了三个方面的内容，模型，视图，控制器。模型：关注数据处理；视图：关注数据显示和报表处理;控制器:负责协调模型和视图。因此，人们开始认为，Model2 是基于 MVC 的。就个人观点而言，如果只是进行网站的开发，还是 Model1 方便。原因是什么哪？Model2 有些复杂。

随着现代社会的发展，商业规则的复杂程度，已经超过了人们的想象。而刁瞒的客户是不管这些的，他们给我们的时间总量是固定的。而软件公司在与客户相处的过程中始终处于弱势地位，是没有太多发言权的，原因很简单，因为公司要盈利，我们要赚客户的钱！

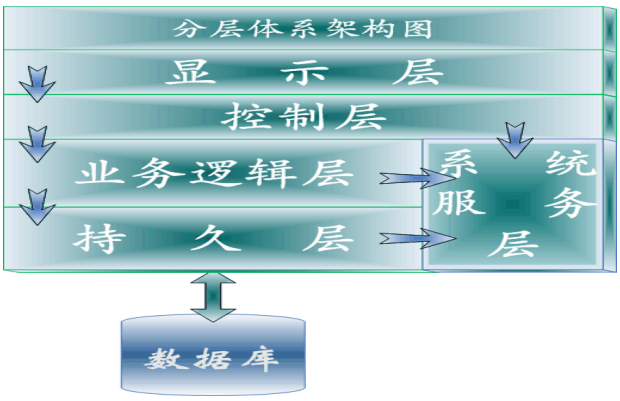
客户限定了时间，也就限定了项目的周期。针对一个复杂的系统，在总量一定的情况下，我们总是在寻求技术时间与业务设计时间的最佳配比。在项目周期固定的情况下，技术时间与业务设计时间是成反比的。两者此消彼长，如果用于技术处理的时间长了，那么业务设计的时间必然就要缩短，同理，如果用于业务处理的时间要增加，就必须缩短技术处理的时间。而在两者的选择过程中，我们不得不选择缩短技术时间，原因很简单，业务规则越来越麻烦，客户并不跟你摆事实，讲道理。而 Model2，虽然很好地解决了三种逻辑的划分，可以应用于企业应用程序开发，但是过于繁琐，实现的手法过于粗略。好多基础性的服务没有为我们提供。比如，请求的分发，请求过程中数据的获取等等。于是人们希望应用新的方式替换 Model2，或是简化 Model2 的应用。基于此，大量基于 Model2 的 web 框架，如雨后春笋般诞生出来。其中最优秀的就是 Struts1 和 WebWork。后来，两者实现了新的融合，形成了 Struts2。由于他们都是基于 Model2 的，而 Model2 是基于 MVC 的，于是人们认为，他们都是基于 MVC 思想而产生的。

Struts1 是第一个成熟的基于 MVC 的 web 框架。他的出现使 Model2 的应用得到了大大的简化，我们将在此后的章节中学习它，并且会发现他的简洁和优美。Struts1 简化了 Model2 的应用，但是只是解决了企业级应用系统的 Web 层问题。那么企业应用系统还有那些层哪，为什么要有这些层哪，这些层又是用什么方式解决更好哪？

MVC 的出现，到今天，已经有 40 年的历史了，面对复杂的现代商业规则，他已经有些苍老，而用它来解决今天的问题，就好像把秦始皇从地下扣出来，然后给他一个遥控器，说一句：“哥们，把电视机闭了！”。估计这哥们肯定龙颜大怒，回敬你一句：“拖出去，毙了！”。那么既然 MVC 苍老了，谁还年富力强哪，答案就是分层体系架构！

下面我们来看看贯穿后继章节的分层思想。

一个好的应用系统，一般都是基于分层体系结构来完成的。分层体系结构大致如下图所示：



其中的显示层，对应于 MVC 的视图（V），控制层对应的是 MVC 的控制器（C），在后继的章节中，我们将其合并到一起，统称 Web 层（也有的书上称为展示层）。而 Struts 只是解决了这里的 WEB 层的技术实现。其它层又是些什么哪，应用那些技术来实现哪？

随着商业规则复杂度的增加，数据处理也日渐繁琐，而数据库技术的诞生很好地解决了这个问题。可以说，今天的应用程序开发，基本上是离不开数据库的，无论是各种运算，处理，最后都要把数据存储到数据库中，以便于将来查询，或是进一步对数据进行分析和加工。这样程序设计的最终目的，也就转换为对数据库的检索，分析和更新。也就是最终都要把内存中的数据保存到某种数据库中，以便长久存储。这就是所谓的数据持久化。而数据持久层就是解决持久化的一个专用的系统逻辑层。目前来说，应用最为广泛的是 Hibernate，他是事实上的数据持久化标准。还有，后继兴起的 EJB3 技术，也属于此范畴，其前途也是不可限量的。

当持久层技术出现以后，它与原先的模型是什么关系，在模型内部，它处于什么地位，怎么应用持久化技术，这些在原始的 MVC 中都没有给出明确的定义。这也是 MVC 思想过时的一个原因，它的模型概念过粗，过于笼统。而分层体系架构，对此作出了明确的定义。持久层，只负责解决数据的持久化（数据持久化对象的增删改查）问题，而为了更好地应用持久层，分层思想引入了业务逻辑层。这一层，一般而言就是普通的 java 类（POJO），它负责组合多个持久化组件，完成复杂的商业处理（比如利息计算，帐户结转，银行转帐等）。

在一个软件系统中，经常会有一些提供基础服务的类，比如 GUID 码的生成，MD5 加密，四舍五入的处理，电子邮件的发送等等。我们发现他们为具体的业务处理提供了基础性的服务，但是，又不专署于任何一种特定的业务逻辑。为此，分层体系结构将其单独抽象出来，形成了系统服务层。这一层，一般而言，我们也应用普通 Java 类（POJO）来进行处理。

基于上述分析，我们可以看到，分层体系结构中，一般而言包括四个宏观的逻辑层次，分别是 Web 层，业务逻辑层，持久层和系统服务层。同时，分层体系结构还定义了它们的强

制性调度关系：“同层和相邻层可以相互访问，不允许夸层访问”。比如，控制层，不能跨越业务逻辑层，直接访问持久层。

当分层体系思想出现以后，软件开发系统中的各个组件都有了合适的位置。我们可以高枕无忧地大步前行了。是不是真的这样哪？实践是检验一切的唯一标准，现实的因素决定一切技术和思维的生命周期！

在实际应用过程中，新的问题又出现了。

在完成复杂功能的同时，我们一直在追求着已有功能实现（组件）的复用。而复用的最佳途径是通过已有组件的组合，产生新的功能点。这就不可避免地涉及到组件之间的调用方式问题。传统的组件调用方式，是由应用程序自身管理组件之间的调用和组合，从而造成了组件之间的高耦合，给组件的独立测试和整个系统的后期维护带来了很大麻烦，使软件质量的提高一如镜花水月，无法真正地从技术上实现。同时，在面向对象思想的应用过程中，伟大的程序员们发现，面向对象只是解决了事物之间纵向的一脉相承，而对于诸如系统日志，安全支撑，事务管理等贯穿于整个系统所有功能点的横向公共性功能，没有给出很好的解决方案。为了解决上述问题，一个新的框架应运而生了，它就是 Spring。在 Spring 中，通过 DI（依赖注入）思想，很好地解决了组件之间的调度问题。同时，引入 AOP（面向切面编程）很好地解决了公共性功能的处理问题。从而给软件开发，带来了一场深刻的，触及灵魂的革命，软件行业，迎来了真正的春天。

在本书的知识体系中，对于分层体系架构的实现，WEB 层将采用 Struts1.x,持久层将采用 Hibernate。而对于整个架构各个组件之间的宏观调度和事务处理，将采用 Spring2.x 来完成。这就是目前比较流行的 SSH 整合架构设计。

上述对于数据持久化，分层体系结构，依赖注入等的论述，都只是简单的介绍，目的是让读者能从宏观上对分层体系结构及其 SSH 实现有个粗浅的了解，在后继的章节中，本书将会进行详细的阐述。不必着急，那样吃不了热豆腐，坚持下去，一切都会好起来。毕竟，人，活着就有希望！

事物都在经历一个从无到有，从弱到强的过程，受实践因素的影响，我们的认识总是停留在一定的局限上的，随着实践的深入，人们的编程思想和编程技术在不断成熟和发展，新的应用技术和方法也会逐步产生，从这个角度来说，浩茫广宇，存在着无数的新思想和新技术等待我们去发现和实现。而程序员的乐趣也正在于此：可以不断地接触新奇的世界，新奇的思想。胸含广宇，气象万千！

最后，演绎林锐先生的一段话，作为本篇的结束：

让我们高举 JAVA 主义、软件工程思想的伟大旗帜，紧密团结在以 Sun 公司为核心的软件巨头周围，沿着比尔·盖茨先生的生财之道，不分白天黑夜地吐血编程，把建设有中国特色软件产业的伟大事业，全面推向 21 世纪。构建和谐社会，享受大康人生！