第一部分

表　述

第 1 章 分　　层

在分解复杂的软件系统时，软件设计者用得最多的技术之一就是分层。在计算机本身的架构中，可以看到：到处都有分层的例子：不同的层从包含了操作系统调用的程序设计语言，到设备驱动程序和CPU指令集，再到芯片内部的各种逻辑门。网络互联中，FTP层架构在TCP之上，TCP架构在IP之上，IP又架构在以太网之上。

当用分层的观点来考虑系统时，可以将各个子系统想像成按照“多层蛋糕”的形式来组织，每一层都依托在其下层之上。在这种组织方式下，上层使用了下层定义的各种服务，而下层对上层一无所知。另外，每一层对自己的上层隐藏其下层的细节。因此，第4层使用第3层的服务，第3层使用第2层的服务，第4层无需知道第2层的细节。（当然，并非所有的分层架构都这么隔绝，但绝大多数是不透明的，或至少是几乎不透明的。）

将系统按照层次分解有很多重要的好处：

- 在无需过多了解其他层次的基础上，可以将某一层作为一个有机整体来理解。例如，无需知道以太网的工作细节，你照样可以在TCP上构建FTP服务。
- 可以替换某层的具体实现，只要前后提供的服务相同即可。例如，FTP服务无论是在以太网、PPP上、还是网络运营商使用的任何网络上都无需改变，而且与提供传输电缆的网络运营商无关。
- 可以将层次间的依赖性减到最低。假设网络运营商改变了物理传输系统，但只要IP层不变，FTP服务就可以不改变。
- 分层有利于标准化工作。TCP和IP就是关于它们各自层次如何工作的标准。
- 一旦构建好了某一层次，就可以用它为很多上层服务提供支持。因此，TCP/IP同时被FTP、telnet、SSH和HTTP使用。否则，所有这些高层协议都必须编写它们各自的底层协议。

分层是一种重要的技术，但也有缺陷：

- 层次并不能封装所有东西。有时它会为我们带来级联修改。最经典的例子就是在一个分层设计的企业应用中，如果要增加一个在用户界面上显示的数据域，就必须在数据库中增加相应的字段，还必须在用户界面和数据库之间的每一层做相应的修改。
- 过多的层次会影响性能。在每一层，一般都会从一种表现形式转换到另一种。不过底层功能的封装通常带来比代价更大的效率提升。例如，可以优化事务控制层，提高其他各层的效率。

然而，分层架构中最困难的问题是决定建立哪些层次以及每一层的职责是什么。

1.1 企业应用中层次的演化

我虽然没有从事过早期批处理系统时期的任何工作，但我认为当时的软件工作人员不会太关注层次的概念，只要编写操作某些文件（ISAM、VSAM等）格式的程序，这就是当时的应用。它不需要层次。

20世纪90年代，随着客户/服务器系统的出现，分层的概念更明显了。这样的系统是一种两个层次的系统：客户端包括用户界面和其他应用代码，服务器端通常是关系型数据库。常见的客户端工具如VB、PowerBuilder和Delphi。这些工具使得构建数据密集型应用非常容易。因为它们的用户界面控件通常都是SQL感知的。因此，可以通过将控件拖拽到“设计区域”来建立界面，然后再使用属性表单把控件连接到后台数据库。

如果应用仅仅包括关系数据的简单显示和修改，那么这种客户/服务器系统的工作方式非常合适。问题来自领域逻辑：如业务规则、验证、计算等。通常，人们会把它们写在客户端，但是这样很笨拙，并且往往把领域逻辑直接嵌入到用户界面。随着领域逻辑的不断复杂化，这些代码将越来越难以使用。而且，这样做很容易产生冗余代码，这意味着简单的变化都会导致要在很多界面中寻找相似代码。

另外一种办法是把这些领域逻辑放到数据库端，作为存储过程。但是，存储过程只提供有限的结构化机制，这将再次导致笨拙的代码。而且，很多人喜欢关系型数据库的原因之一是SQL是一个标准，允许他们更换数据库厂商。尽管真正更换数据库厂商的用户寥寥无几，但还是有很多人希望拥有这种选择，并且没有太大的附加代价。由于存储过程都是数据库厂商私有的，因此普通用户被剥夺了这种选择权。

在客户/服务器方式逐渐大众化的同时，面向对象方式开始崛起。面向对象为领域逻辑的问题找到了答案：转到三层架构的系统。在这种方式下，在表现层实现用户界面，在领域层实现领域逻辑，在数据源层存取数据。这种方式使你可以将复杂的领域逻辑从界面代码中抽取出来，单独放到中间层，用对象加以建模和组织。

尽管有这些优势，但一开始面向对象的进展并不大。当时的实际情况是：大多数系统并不特别复杂，或者至少在构建之初没有那么复杂。因此，当系统比较简单时，相对于三层架构的优势，强有力的客户/服务器工具的竞争力非常大。但客户/服务器工具很难甚至无法应用于三层架构系统的配置。

我认为真正巨大的冲击来自Web的兴起。人们忽然想在Web浏览器上部署这些客户/服务器应用。然而，如果所有的领域逻辑都是写在“胖”客户中，则所有这些都必须在Web界面中重写。对于设计良好的三层系统来说，只需要增加一个新的表现层，就可以了。另外，Java的出现使得面向对象语言无所顾忌地向当时的主流技术发起冲击。用于构建Web页面的工具对SQL的绑定也没有那么紧密了，这也使得它们比较容易适应三层结构。

当人们讨论分层时，常常不容易区分*layer*和*tier*。这两个词汇经常被用作同义词，但是很多人还是认为*tier*意味着物理上的分离。客户/服务器系统常常被称为“two-tier system”，其分离是物理上的分离：客户端是一台台式机，而服务器端是一台服务器。我使用*layer*，旨在强调无需把不同的层次放在不同的计算机上运行。独立出来的领域逻辑层，既可以运行在台式计算机上，

也可以运行在数据库服务器上。在这种情形下，有两个节点，但是有三个层次。如果数据库也在本地，还可以在一台笔记本电脑上运行三层软件，当然，仍旧存在三个截然不同的层次。

1.2 三个基本层次

本书主要就三个基本层次的架构展开讨论：表现层、领域层和数据源层（这里的命名取自文献[Brown et al.]）。表1-1总结了这些层次。

表1-1 三个基本层次

层 次	职 责
表现层	提供服务，显示信息（例如在Windows或HTML页面中，处理用户请求（鼠标点击，键盘敲击等），HTTP请求，命令行调用，批处理API）
领域层	逻辑，系统中真正的核心
数据源层	与数据库、消息系统、事务管理器及其他软件包通信

表现逻辑处理用户与软件间的交互。可能简单到只是命令行或基于文本的菜单系统，但是当前的客户界面往往是功能完善的胖客户图形界面，或者是基于HTML的浏览器界面（本书中的“胖客户”是指Windows/Swing/fat-client用户界面，不包括HTML浏览器）。表现层的主要职责是向用户显示信息并把从用户那里获取的信息解释成领域层或数据源层上的各种动作。

数据源逻辑主要关注与其他系统的交互，这些系统将代表应用完成相关的任务。它们可以是事务监控器、其他应用、消息系统等。对于大多数企业应用来说，最主要的数据源逻辑就是数据库，它的主要责任是存储持久数据。

最后一部分就是领域逻辑，也称为业务逻辑。它就是应用必须做的所有领域相关工作：包括根据输入数据或已有数据进行计算，对从表现层输入的数据进行验证，以及根据从表现层接收的命令来确定应该调度哪些数据源逻辑。

有时，层次组织成领域层对表现层完全隐藏了数据源层。但更多的时候，是表现层直接对数据存储进行操作。虽然这样做并不纯粹，但是在实践中往往运行良好。表现层可能解释来自用户的命令，通过数据源层将相关数据从数据库中提取出来，然后让领域逻辑层在向用户显示相关数据之前先处理这些相关数据。

一个单独的企业应用，可能在上述的三个层次上都包含多个软件包。如果某个应用不仅要支持用户通过胖客户机界面访问，还要支持用户通过命令行形式访问，则它需要两个表现层：一个支持胖客户机界面，另一个支持命令行。对于不同的数据库，通常也需要多个数据源组件，特别是在与已有的软件包通信时。即便是领域逻辑，也有可能被分割成相互独立的不同部分，特定的数据源包只能由特定的领域包使用。

到目前为止，我们一直都在讨论用户。这很自然地会引出一个问题：如果驱动软件的不是人，情况又怎么样呢？比如说驱动者可能是时髦的Web Service或是一个老土但实用的批处理程序。对于后者，用户将是一个客户程序。这样，很明显，表现层就有可能与数据源层出现某些相似之处，因为它们都是系统与外界的接口。这就是Alistair Cockburn的*Hexagonal Architecture*模式[wiki]背后的逻辑，它将任何系统都视为由到外部系统的接口所围绕的一个核心。在

*Hexagonal Architecture*中，所有外部的东西都被视为外部接口。因此，从这种意义上说，它是一种对称视图，而不是本书中的非对称分层视图。

然而，我认为这种非对称性是有益的。因为，为别人提供服务的接口与使用别人服务的接口存在较大的差别，需要明确区分。这就是表现层和数据源层相对于核心的本质差别。表现层是系统对外提供服务的外部接口，不管外面是复杂的人类还是一个简单的远端程序。数据源层是系统使用外部服务的接口。这样区分的好处是：客户的不同将改变你对服务的看法。

对每个企业应用，尽管我们能够区分出其中的主要的表现层、领域层和数据源层，但是具体如何分离要取决于应用的复杂程度。从数据库中读取数据并把它显示在Web页面上的简单脚本，可能全部在一个过程中。我将仍然尽量保持三层架构的风格，不过在这里可能只是把每个层的行为放到三个不同的子程序中。一旦系统再复杂一点，就可以将三个层次的工作分解成不同的类。如果复杂度继续增加，则把类分配到不同的包中。我的总体建议就是根据不同的问题，选择一种适合的分离方式，但是切记一定要进行某种形式的分离——至少在子程序级别。

伴随着分离，还有一条关于依赖性的普遍原则：领域层和数据源层绝对不要依赖于表现层。也就是说，在领域层和数据源层的代码中，不要出现调用表现层代码的情况。这条规则将简化在相同的基础上替换表现层的代价，也使得表现层的修改所带来的连锁反应尽可能小。领域层与数据源层的关系更复杂，其取决于数据源层的架构模式。

使用领域逻辑时，其中一个最困难的部分就是区分什么是领域逻辑，什么是其他逻辑。一种不太正规的测试办法就是：假想向系统中增加一个完全不同的新层，例如为Web应用增加一个命令行界面层。如果在这个过程中，发现需要重复实现某些功能，则说明可能有一些本应该在领域层实现的逻辑，现在在表现层实现了。类似地，你也可以假想一下，将后台数据库更换成XML文件格式，看看情况又会如何？

举一个例子。我所知道的一个系统有一张产品列表，其中，当月销售量比上月销售量大10%的产品需要用红色显示。为实现这个功能，开发者在表现层逻辑中比较当月和上月的销售量，然后将差别大于10%的产品显示为红色。

这样做的麻烦就是将领域逻辑放到了表现层中。为了进行适当的分离，需要在领域层中定义一个方法，用来指示该产品的销售量是否较上月有较大提高。该方法完成销售量的比较，返回一个布尔值。表现层则只需要简单地调用一下这个布尔方法，如果返回值为真，则用红色突出显示这个产品。这样，该过程就分解成两部分：确定需不需要突出显示，选择如何突出显示。

当然，我担心这样也许有些太教条主义了。Alan Knight在审阅本书时评论说：他自己“很头大，将部分领域逻辑混入表现层到底是滑向地狱的第一步呢，还是只有少数纯粹主义者才会抱怨的小问题？”我们之所以担心，正是因为这种做法两者兼备。

1.3 为各层选择运行环境

本书绝大多数篇幅讨论的都是逻辑层次——将系统中各部分分离，以降低不同部分之间的耦合程度。即使是都运行在同一台计算机上，不同层次间的分离也是非常重要的。当然，系统物理结构的不同会有所影响。

对于大多数信息系统来说，主要的决定就是在哪里运行处理工作，是在客户机上，还是在

台式机上，又或是在服务器上？

通常，最简单的情况是将所有东西都运行在服务器上。在这种情况下，一个使用Web浏览器的HTML前端是一个好方法。这样做最大的好处是所有的东西都在有限的环境内，很容易修改维护。无需考虑将它们分发到不同的客户端并维护与服务器的同步等问题。也不必考虑与客户机上其他软件的兼容性问题。

在客户机上运行应用程序的好处是系统的响应性好，或者在网络断开的情况下也能正常工作。任何运行在服务器上的逻辑在响应客户请求时，都需要一个来回的通信开销。如果用户仅仅是为了试试系统的即时反馈，这个通信来回也无法避免。它还需要在网络连接保持的状态下运行。当然，网络的分布可能会无所不在，但是至少在我写本书的时候，31000英尺高的地方还没有。也许在不久的将来，就会到处都有了，但有不少人需要立即工作，不必等待网络连接。断接操作带来特别的挑战性，我不想在本书中过多讨论。

有了这些约束，我们就可以逐层分析了。数据源层一般都是运行在服务器上。例外情况是：当需要断接操作时，可以将服务器的功能复制到一台功能强大的客户机上。在这种情况下，在离线的客户机上对数据源的任何修改，都需要同步到服务器上。正如我前面提到的那样，关于这方面的讨论，我想留到以后某个时候或留给另一位作者。

在何处运行表现层主要取决于用户界面的种类。如果运行的是一个胖客户，则意味着表现层运行在客户端。如果运行的是一个Web界面，则意味着表现层运行在服务器端。当然，也有例外——例如，客户软件（如Unix中的X servers）的远程操作在台式机上运行了一个Web服务器——当然，这是极少数情况。

如果要建立一个B2C系统，就没什么选择了。无论是谁都可能访问你的服务器，你也不想因为客户用的是TRS-80系统，就把他拒之门外。在这种情况下，可以在服务器上完成所有工作，并提供一个HTML界面给浏览器。这样做的缺点就是每个操作都必须要一个来回的通信开销，可能会影响响应时间。可以通过可下载的applet或浏览器脚本来缓解问题，但是它们同时会带来浏览器兼容性等其他问题。使用的HTML越纯粹，事情就会变得越简单。

即使你们的每一台台式机都是由你们公司的信息系统部门手工精心搭建的，这种简单性仍然非常诱人。因为即使是非常简单的胖客户系统，也会遇到维护客户端一致性以及避免各种软件不兼容等问题。

人们希望有胖客户表现层的主要原因是：有些任务对于用户而言太复杂了，为了有一个可用的应用系统，它们的需要超出了Web GUI的能力。当然，人们已经在逐渐习惯于采用使Web前端更可用的各种方法，这些方法减少了对胖客户方式的需求。因此，我非常赞同只要有可能就用Web表现方式，只有在必需的情况下才使用胖客户方式。

剩下来的就是领域逻辑了。领域逻辑可以全都运行于服务器端，也可以全都运行于客户端，也可以一分为二。再有，全部运行在服务器端有助于系统维护，向客户端转移是为了响应时间及断接使用的需要。

如果你必须在客户端运行某种逻辑，可以考虑将所有逻辑都运行在客户端——这样至少保证了相关的东西都在一起。这样，胖客户端和Web服务器联合部署在客户机上，对响应性的改善不会太大，但是它可以作为一种处理断接操作的办法。在这种情况下，可以通过不同的模块将

领域逻辑与表现层分开，可以使用事务脚本或领域模型。将所有的领域逻辑放在客户端的问题是升级和维护代价高。

将领域逻辑分割在客户端和服务器端，应该是最差的选择，因为这样做无法确定任意一块逻辑到底在哪。采用这种方式的主要原因是：只有一小部分领域逻辑需要在客户端完成。其中的诀窍就是将这一小部分独立出来成为自包含的模块，使得它不依赖于系统的任意其他部分。这样，就可以在客户端或服务器端运行它了。这将需要采用一些烦人的小技巧，但它的确是一个解决问题的好办法。

一旦选择了处理节点，接下来就应该尽可能使所有代码保持在单一进程内完成（可能是在同一个节点上，也可能拷贝在集群中的多个节点上）。除非不得已，否则不要把层次放在多个进程中完成。因为那样做不但损失性能，而且增加复杂性，因为必须增加类似下面的模式，如远程外观和数据传输对象。

以下因素被Jens Coldewey称为复杂性增压器（complexity booster）：分布、显式多线程、范型差异（例如对象/关系）、多平台开发以及极限性能要求（如每秒100个事务以上）。所有这些因素都会带来很大的代价。当然，有时我们无法回避它们，但是要切记：这里罗列的每一项都会为开发和运行维护阶段带来开销。