第 3 章 软件设计

3.1 软件设计概述

在软件需求分析阶段已经完全弄清楚了软件的各种需求，较好地解决了所开发的软件“做什么”的问题，并已在软件需求说明书和数据要求说明书中详尽而充分地阐明了这些需求以后，下一步就要着手对软件系统进行设计，也就是考虑应该“怎么做”的问题。软件设计（Software Design）就是根据所表示的信息域的软件需求，以及功能和性能需求，进行数据设计、系统结构设计、过程设计、界面设计等，其目标就是为了构造一个高内聚、低耦合的软件模型。

软件设计是软件开发的关键步骤，它直接影响软件的质量。在软件编码实现之前，必须先进行软件设计，否则整个系统就像没打好基石的建筑一样缺乏稳定。软件设计对系统的影响，如图 3.1 所示。

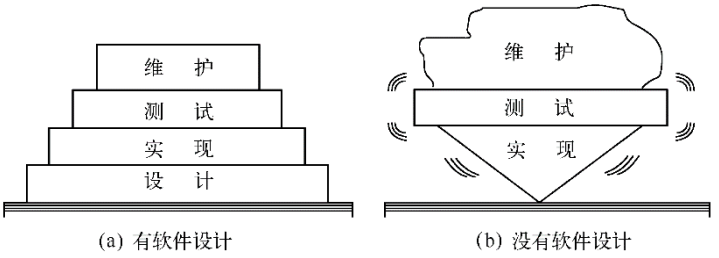
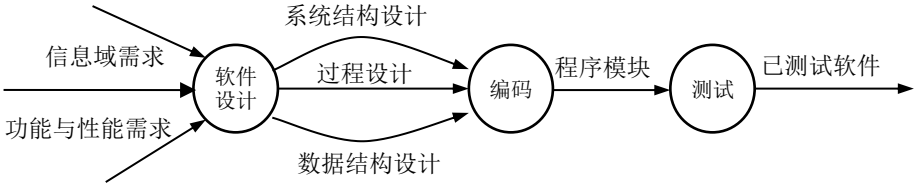


图 3.1 有无软件设计的系统对比

现代软件工程要解决的问题就是软件的质量和效率，而软件设计的好坏将直接影响软件的质量，所以，软件设计是整个系统开发过程中最为核心的部分。所有的开发工作都将根据设计的方案进行，系统的总体结构，以及数据结构也在该阶段确定，在后继的编码实现阶段，程序员可以很好地独立完成编码工作。这对软件生产实现“流水线”的传统生产方式有很大的帮助，可大大提高软件的生产效率，减小了人员间的耦合。

软件需求确定以后，进入由软件设计、编码、测试三个关联阶段构成的开发阶段。开发阶段的信息流如图 3.2 所示。在设计阶段中，根据用信息域表示的软件需求，以及功能和性能需求，采用某种设计方法进行数据设计、系统结构设计、过程设计和界面设计。数据设计侧重于软件数据结构的定义。系统结构设计用于定义软件系统的整体结构，它是软件开发的核心步骤，以建立软件主要成分之间的关系。过程设计则是把结构成分转换成软件的过程性描述。而界面设计是对系统边界的描述，是用户和系统进行交互的工具。在编码阶段中，根据这种过程性描述，生成源程序代码，然后通过测试最终得到完整有效的软件。



，

图 3.2　开发阶段信息流

3.1.1　软件设计的任务和目标

1.　软件设计的任务

软件设计阶段的主要任务是：将分析阶段获得的需求说明转换为计算机中可实现的系统，完成系统的结构设计，包括数据结构和程序结构，最后得到软件设计说明书。这是一个从现实世界到信息世界的重要抽象过程。

从工程管理的角度来看，软件设计分为总体设计（概要设计）和详细设计两个阶段，其工作流程如图 3.3 所示。

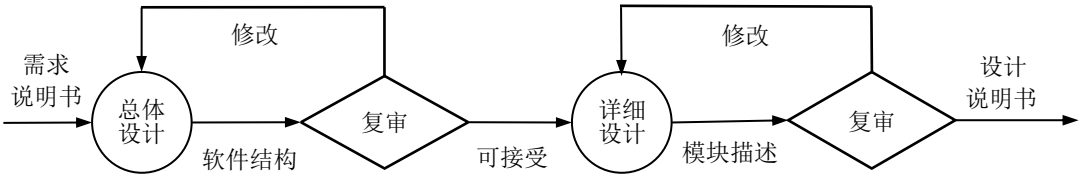


图 3.3 软件设计工作流程

首先进行总体设计，将软件需求转化为数据结构和软件的系统结构，划分出组成系统的物理元素：程序、数据库、过程、文件、类等；然后是详细设计，通过对结构表示进行细化，得到软件详细的数据结构和算法。设计阶段结束要交付的文档是设计说明书，根据设计方法的不同，有不同的设计文档。每个设计步骤完成后，都应进行复审。因此，软件设计阶段的任务又可具体分为三部分：

① 确定软件结构，划分子系统模块。好的软件结构可以使软件的开发过程流畅自如，同时也能为软件的部署带来好处。合理的模块划分可以降低软件开发的复杂度，同时也能提高软件的可重用性。

② 确定系统的数据结构。数据结构的建立对于信息系统而言尤为重要。要确定数据的类型，组织、存取方式，相关程度及处理方式等。

③ 设计用户界面。作为人机接口的用户界面起着越来越重要的作用，它直接影响到软件的寿命。

当然，在设计阶段的最后还要测试设计的正确性，只有一个正确的设计才能保证软件的质量。

2.　软件设计的目标

在设计阶段应达到的目标如图 3.4 所示。

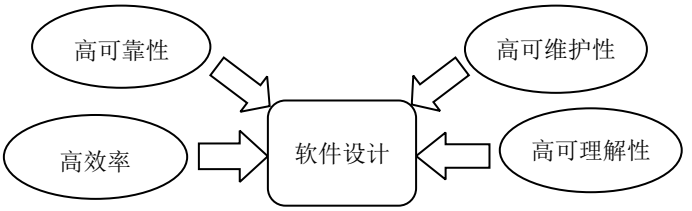


图 3.4 软件设计的目标

可根据以下准则来衡量软件设计的目标：

① 软件实体有明显的层次结构，利于软件元素间控制；

② 软件实体应该是模块化的，模块具有独立功能；

③ 软件实体与环境的界面清晰；

④ 设计规格说明清晰、简洁、完整和无二义性。

常用的设计方法有：SD 法、Jackson 法、HIPO 法、Parnas 法、Warnier 法等。现在流行的方法是面向对象的设计方法（OOD，Object-Oriented Design）。

3.1.2 软件设计过程

软件设计是一种描述。它描述了待实现的软件系统的结构，作为系统组成部分的数据、各软件构件间通信的接口/界面、使用的算法。软件设计永远不可能是一蹴而就的。设计者从不同的视角以迭代的方式，逐渐为系统添加形式化和细节内容。

软件设计的过程大致包括如下活动：

① 体系结构设计。要解决的问题是：系统有哪些子系统，它们之间的关系如何？

② 抽象说明书。对每一个子系统，编写它的服务以及受到约束的抽象说明书。

③ 接口设计。为每一个子系统设计与其他子系统通信的接口（Interface）。这些接口必须是无二义性的，并且一个子系统在使用接口时，不需要也不应该了解提供接口的子系统的实现细节。这里提到的“接口”有两个层面的含义：一是一个子系统暴露在外、可供其他子系统使用的系统功能集合，是子系统对外提供的服务的入口，在软件实现时表现为一组函数，或者是类的公有方法；二是类似于 Java 语言接口类的概念。接口类不能有数据成员，并且所有的方法必须是公有的。很明显，本小节提到的“接口”的含义是第一种。在以后的章节中，“接口”一词会多次出现，相信读者能根据上下文的描述分辨出它们的含义。

④ 构件设计。子系统提供的服务分解到构件中，为每一个构件设计接口。

⑤ 数据结构设计。

⑥ 算法设计。

当然，上述过程只是一个设计过程的概貌，而非必须遵循的步骤。例如在实际应用的设计，后两项设计过程也许会被放到软件实现而非设计阶段。

在设计过程的步骤中，最重要的就是体系结构设计。我们将在 3.2 中详细讨论。

3.2 软件体系结构设计

软件体系结构为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素

的描述、元素间的相互作用、指导元素集成的模式，以及这些模式的约束组成。软件体系结构不仅指定了系统的组织结构和拓扑结构，显示了系统需求和构成系统的元素之间的对应关系，而且提供了一些设计决策的基本原理。良好的体系结构是普遍适用的，它可以高效地处理各种各样的个体需求。

3.2.1 体系结构设计过程

体系结构设计是软件设计的第一个阶段。该阶段侧重于建立系统的基本结构性框架，即系统的宏观结构，而不关心模块的内部算法。

一般的体系结构设计过程主要包括如下几项活动：

① 系统结构设计（System Structuring）。将系统划分为一些主要的独立子系统，确定子系统间的通信方式。

② 控制建模（Control Modeling）。建立系统各部分之间的控制关系。

③ 模块分解（Modular Decomposition）。将子系统分解为模块。

以上活动通常不是按顺序而是交错进行的。在任何一个过程中，设计者都应当提供更多的细节以供决策，最终使设计满足系统的需求。体系结构设计的好坏将会直接影响系统的性能、健壮性、可分布性和可维护性。

在实际的设计中，我们其实并不需要真正地去建立一个全新的体系结构模型，而是从典型、成熟的模型中选择，大型复杂的系统可能会选择多个。当然，有经验的设计者一定会在其中做出某些变通，以适应实际系统的需求。

在下面的内容中，我们将就几种常见的通用体系结构模型进行深入讨论。其中，3.2.2-3.2.4 同属于系统结构性模型的内容。

3.2.2 仓库模型（The Repository Model）

仓库模型是一种集中式的模型。在这种结构模型当中，应用系统用一个中央数据仓库来存储各个子系统共享的数据，其他的子系统可以直接访问这些共享数据。当然，每个子系统可能会有自己的数据库。为了共享数据，所有的子系统都是紧密耦合的，并且围绕中央数据仓库，如图 3.5 所示。

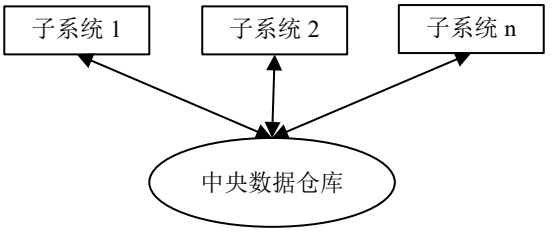


图 3.5 仓库结构

在早期，仓库模型采用的硬件平台结构是主机/终端（Host/Terminal）结构，其拓扑结构如图 3.6 所示。主机一般是一台大型甚至超级计算机，终端是一些功能有限、一般只负责数据输入输出的设备，常被称为“哑（dumb）终端”。

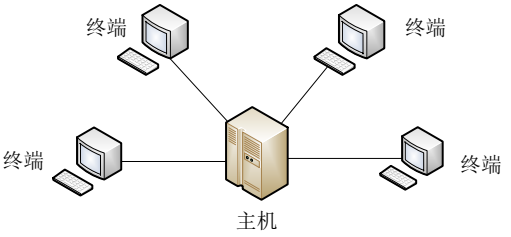


图 3.6 主机-终端系统的拓扑结构

主机/终端结构的特点是所有的计算都在主机系统中完成。这意味着，图 3.5 中的所有子系统并非运行在对应的用户终端上。实际上，用户终端的主要功能是输入和输出，除此之外，几乎不做任何数据处理，即使是光标移动这样的简单操作也交由主机处理。终端发起的任何动作都由中央主机处理，这使得主机系统处理器的压力很大。后来出现的智能终端虽然可以处理诸如光标移动、文本编辑等简单操作，但主要的计算任务仍然在主机系统中完成。为了提升主机的计算能力，多数的主机系统都配有多乃至成百上千片处理器。

仓库模型的主要优点是：

① 数据由一个子系统产生，并且被其他一些子系统共享；

② 共享数据能得到有效的管理，各子系统之间不需要通过复杂的机制来传递共享数据。

③ 一个子系统不必关心其他的子系统是如何使用它产生的数据的。

④ 所有的子系统都拥有一致的基于中央数据仓库的数据视图。如果新子系统也采用相同的规范，则将它集成于系统中是容易的。

但这种系统也有明显的缺陷：

① 虽然共享数据得到了有效的管理，但随之而来的问题是各子系统必须有一致的数据视图，以便能共享数据，换句话说，就是各子系统之间为了能共享数据必须走一条折中的路线，这不可避免地会影响整个系统的性能。

② 一个子系统发生了改变，它产生的数据结构也可能发生改变。为了其他共享的目的，数据翻译系统会被用到。但这种翻译的代价是很高的，并且有时是不可能完成的。

③ 中央数据仓库和各子系统拥有的数据库必须有相同的关于备份、安全、访问控制和恢复的策略，这可能会影响子系统的效率。

④ 集中式的控制使数据和子系统的分布变得非常困难甚至成为不可能。这里分布一词指的是将数据或子系统分散到不同的机器上。

仓库模型的特点决定了它的应用范围。一般来说，银行系统、命令控制系统、CAD 系统等常采用这种结构。

3.2.3 层次模型（The Layered Model）

层次结构模型将系统划分为若干层次，每个层次提供相应的服务，并且下层的服务只向它的直接上层提供。这种结构非常适合增量的软件开发，新增加的部分将位于原有的系统之上或将原系统包裹起来，进而扩展了原系统的功能。

层次结构的一个经典例子就是 ISO 的 OSI 七层网络参考模型。在 OSI 模型中，网络服务被划分成七层，如图 3.7 所示。当一个网络实体 A 向另一个网络实体 B 发送数据时，数据不是从 A 的应用层直接发送到 B 的应用层，而是首先在本方的层次结构中从上到下利用下层提供的服务接口向下传递，直至物理层，然后物理层再将信号在传输介质上传送至 B 的物理层，

此后数据从下到上依次传送到 B 的应用层。不过从逻辑上来看，我们可以认为 OSI 的每一层都可以对等通信。

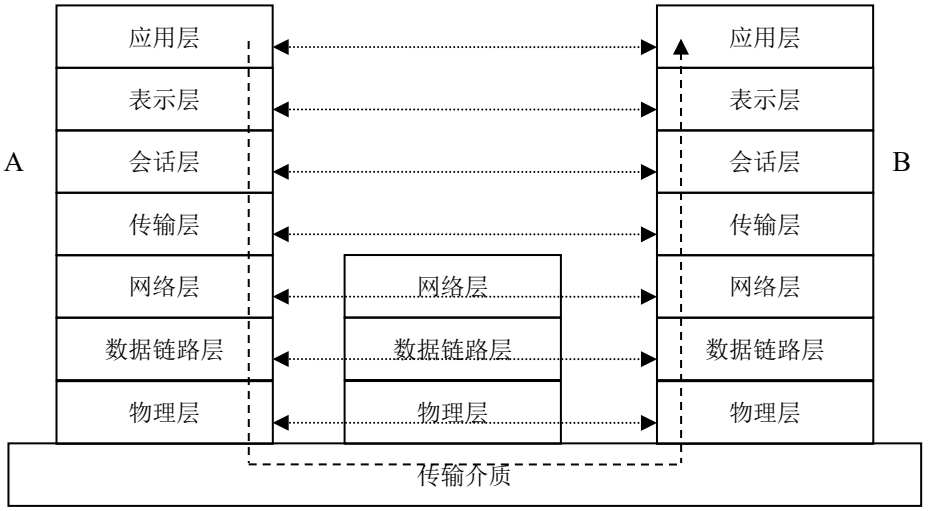


图 3.7 OSI 七层参考模型

3.2.4 分布式系统模型（The Distributed System Model）

仓库模型集中处理的特点决定了主机的计算能力是系统扩展的瓶颈。主机系统高昂的价格也使普通用户望而却步。因此，必须找到一种廉价的、能使普通用户受惠的计算模式。

随着网络技术的飞速发展，以及个人计算机和廉价 PC 服务器的迅速普及，将计算分散到网络中的不同性能的计算机上成为一种高效且廉价的解决方案。在这种方案中，每个连接在网络上的计算节点（计算机）都会根据各自的计算能力负责力所能及的计算工作，并通过这样的分工合作得到整体结果，从而提高整个系统的能力和效率。这就是分布式系统结构（Distributed System Architecture）的基本思想。

分布式结构中的节点可以是同构的（即具有相同的软/硬系统），也可以是异构的，只要系统的所有节点采用相同的网络协议，能够无障碍地交换数据就能达到要求。图 3.8 是分布式系统的网络拓扑结构。

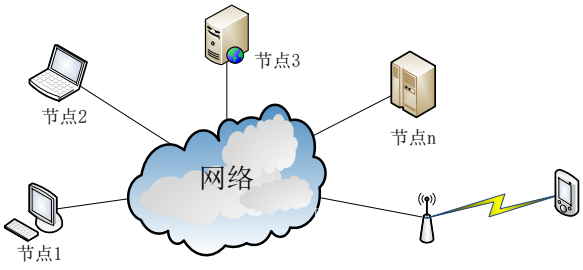


图 3.8 分布式系统的网络拓扑结构

图中的节点可能包括 PC 机、服务器、大型主机和移动平台等。

分布式结构有如下一些优势：

① 资源共享：系统中每个服务结点上的资源都可以被系统中的其他结点访问。

② 开放性高：系统可以方便地增删不同软、硬件结构的结点。
③ 可伸缩性好：系统可以方便地增删新的服务资源以满足需要。
④ 容错能力强：分布式系统中的信息冗余可以容忍一定程度的软、硬件故障。
⑤ 透明性高：系统中的结点一般只需知道服务的位置而不必清楚系统的结构。

但分布式结构也存在如下一些不足：

① 复杂性：分布式系统比集中式系统要复杂的多。集中式系统的性能主要依赖于主机的处理器能力，而分布式系统的性能则还会依赖于网络的带宽，这让情况变得更加复杂。
② 安全性：网络环境随时面临着各种威胁，如病毒、恶意代码、非法访问等，如何保证安全性是一个让人头疼的问题。
③ 可管理性：分布式系统的开放性使得系统常常是异构的。显而易见，管理异构的系统比管理主机系统要困难得多。
④ 不可预知性：这主要指系统的响应时间。网络环境本身的特点决定了网络负载会明显地影响整个系统的响应时间。

下面主要讨论几种最常见和常用的分布式结构。

1. 客户/服务器（Client/Server，C/S）模型

在客户/服务器模型这个术语中，客户和服务器都是软件层面的概念。

典型的 C/S 结构的系统包括两个主要的组成部分：

① 服务器（Server）：多个独立的服务器为系统提供诸如 Web、文件共享、打印、存储等服务。
② 客户（Client）：多个并发客户应用访问多个服务器提供的服务，每个客户应用都是独立的，同样的客户应用可以同时有多个实例。

网络是 C/S 的基础设施，客户和服务器通过网络连接在一起。有时客户应用和服务器应用会在同一台机器上运行，但两个应用还是要通过本机的网络协议进行通信，其效果就像在不同的机器上运行一样。

C/S 结构的应用系统由多个负责不同任务的子系统构成。为了降低这些子系统之间的耦合度，可以根据子系统的逻辑功能，将它们规约到三个相对独立的逻辑层中：

① 用户界面层（User Interface Layer，UI）：实现与用户交互；
② 业务逻辑层（Business Logical Layer，BLL）：进行具体运算和数据处理；
③ 数据访问层（Data Access Layer，DAL）：完成数据查询、修改、更新等任务。

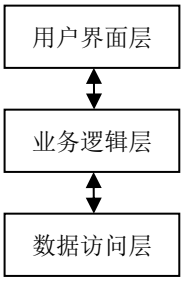


图 3.9 应用的三层模型

以上三个逻辑层之间的关系如图 3.9 所示。这种三层模型往往被称为应用系统的“三层架构”。

根据业务逻辑层所处的位置，C/S 结构的应用系统常可以分为两层结构、三层/多层应用结构。下面讨论这几种结构的特点。

(1) 两层客户-服务器模型（Two Tier Client/Server Architectural Model）

在两层 C/S 结构中，应用系统由两个典型的应用组成，其中一个是主要负责用户界面部分的客户端，另一个是主要负责数据访问的服务器，两者通过网络进行数据交换。其结构如图 3.10 所示。

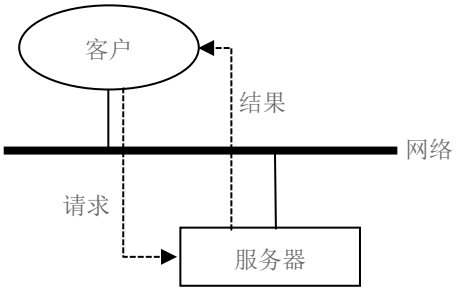


图 3.10　两层 Client/Server 结构

前面提到过，应用系统的逻辑分为三层。这样，在两层的 C/S 结构中，一个必须解决的问题就是将业务逻辑层映射到两个端系统上。有三种映射方式：

- 映射到客户端。这样的结构称为“胖客户端（Fat-Client）结构”。早起的 MIS 系统采用此结构。这种结构可能带来管理上的问题。
- 映射到服务器端。这样的结构称为“瘦客户端（Thin-Client）结构”。后面提到的 B/S 结构就属于此类。这种结构可能有伸缩性和性能上的问题。
- 在客户端和服务器端均有分布。这样的结构可以达到一定程度的平衡，但也可能使系统变得更加复杂。

以上三种方式都有广泛的实际应用。

这里举数据库应用的例子来说明两层 C/S 结构的工作方式。

客户应用根据需求向数据库服务器发出数据访问请求，数据库服务器会响应这个请求，查询、更新数据，然后将结果返回给客户端。这是典型的“请求-响应-得结果”模式。当然，不是所有的请求都需要返回结果。

C/S 客户端往往包含一些数据查询和更新等操作。常用的一种设计就是将 SQL 语句嵌入到客户端代码中，然后发送到服务器端执行。这其实是一种不好的设计：客户端需要更多的了解服务器端的数据库结构，并把它们暴露在不安全的环境中。一个改进的方法是将数据操纵语句转移到服务器端，并组织在一些存储过程中，客户端通过远程过程调用（Remote Procedure Call，RPC）获得结果。

C/S 结构中的客户和服务器之间的关系不一定是一对一的。网络上可能会存在多个提供不同资源的服务器应用，提供诸如数据服务、Web 服务、打印服务、文件服务等；同时也可以拥有多个客户应用，这些应用根据自己的需要访问不同的服务器应用。

由于两层 C/S 架构将数据表示和处理逻辑分开，因此客户端和服务器的功能相对来说就比较单一，两端的维护和升级也比集中式结构简单。但 C/S 架构也存在着明显的缺陷：由于业务逻辑和两端之一是紧耦合的，因此当一端发生改变时，这种改变极有可能反射到另一端。因此，C/S 架构不适合用在多用户、多数据库、非安全的网络环境中。另外，客户端应用程序越来越大，对使用者的要求也越来越高。

(2) 三层/多层应用模型（Three/Multi-Tier Model）

多层模型是两层 C/S 模型的扩展。在这种模型当中，为了弥补两层 C/S 结构的缺陷，业务逻辑部分不是被映射到两个端应用中，而是被分离出来成为单独的一层（中间层）；甚至为了满足应用的需要，被分离成多层。这些中间层将业务规则提取出来，交由一些完成业务处理功能的分布式对象组件来完成。这样，客户端和服务器将会只聚焦在自己所负责的用户界面和数据访问工作上，从而变得更加的单纯。而应用中最复杂的业务逻辑处理部分，将由中间的多个业务逻辑层负责，它们完成具体的业务处理，其中隐含了分布处理、负载平衡、事务逻辑、持久性和安全性等技术。

在图 3.11 所示的多层模型中，为了有效地管理那些完成业务逻辑的组件，中间层会用到应用服务，包括事务服务、消息服务等。常见的事务服务器有 Microsoft Transaction Server，消息服务器有 Microsoft Message Queue。

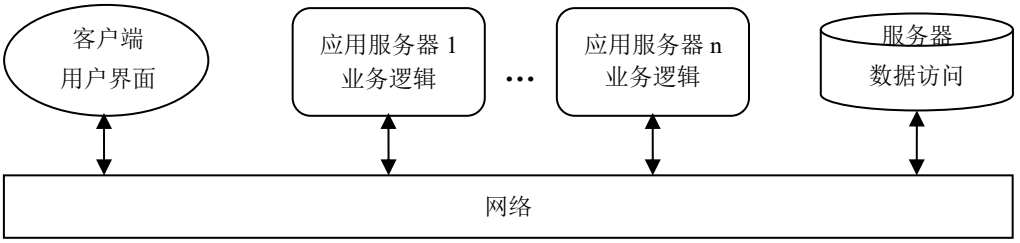


图 3.11 多层模型

在多层模型中，应用服务器的数目可以有多个，这正体现了多层的含义。不过，这并不意味着每个应用服务器都必须运行在独立的硬件上。

多层应用模型的优点相当的明显：

- 客户端的功能单一，变得更“瘦”。
- 每一层可以被单独改变，而无须其他层的改变。
- 降低了部署与维护的开销，提高了灵活性、可伸缩性。
- 应用程序各部分之间松散耦合，从而使应用程序各部分的更新相互独立。
- 业务逻辑集中放在服务器上由所有用户共享，使得系统的维护和更新变得简单，也更安全。

因此，现在越来越多的应用采用了多层结构，以适应不断变化的用户需求。

2. 浏览器/服务器（Browser/Server，B/S）模型

B/S 模型实际上是一种典型的瘦客户结构。此结构中，客户端简单到只是一个浏览器；服务器是能够传送 HTML 页面的 Web 服务器。

B/S 结构应用系统的工作原理是：用户通过浏览器向 Web 服务器发出页面请求；服务器响应请求，并将指定的页面文件传送给客户浏览器；浏览器在接受到页面后，解释页面的内容，并将解释的结果显示在浏览器的窗口之中。

在 B/S 结构中，由于页面文件是驻留在服务器端的，因此相对于传统的 C/S 应用，B/S 应用的重新部署非常方便：C/S 应用在更新客户端时需要用户下载新版本或者更新程序，然后重新安装客户端应用或者打补丁，这可能极大地增加重新部署的成本，甚至使重新部署不可能；而 B/S 应用则不需要这样的过程，因为一旦完成服务器端的页面更新，那么用户下载的页面

就总是最新的。

早期的 B/S 应用只由一些 HTML 静态页面构成。页面文件中一般含有用于指定页面样式的 CSS 文档的链接，以及用于改善用户交互体验的 JavaScript 脚本。而现在的 B/S 应用都比较复杂，完成系统功能的应用程序可分为两大类：一是在服务器端运行的服务器端应用，由一组用脚本语言（例如 ASP.NET、JSP、PHP 等）编写的应用程序组成；二是呈现数据的页面模板，或者是由脚本动态生成的页面。服务器在接收到客户端浏览器发出的页面请求后，运行页面关联的脚本，将结果数据与指定页面模板绑定在一起，然后将生成的最终页面传送到浏览器，此后由浏览器解释页面文件，并显示出页面内容。

服务器端脚本常常用于数据访问。因此，这样的 B/S 应用实际上是一种三层结构，如图 3.12 所示。

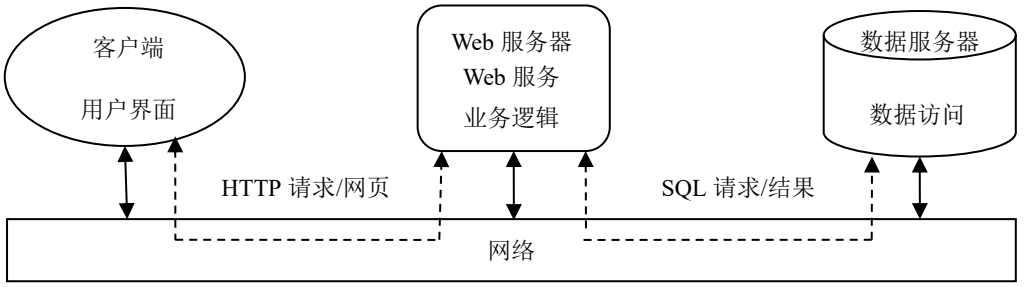


图 3.12　三层 B/S 模型

从图中可以看到，业务逻辑处理子系统是以组件对象的形式集成到 Web 服务器中，这无疑加重了 Web 服务器的负担，而这种负担本来不该由它来承担。所以，更好的结构是将业务逻辑从 Web 服务器中分离出来，把原来的三层结构变成每一层的工作都很单纯的多层结构。

B/S 结构的主要优点是：

- 客户端很“瘦”，易于采用通用的软件系统完成，例如流行的浏览器等；
- 应用的更新和重新部署非常方便。

其主要缺点是：

- 客户端采用的主要技术是 HTML，而这种技术的交互能力较弱。虽然应用 JavaScript 可以进行一定程度的改善，但还是不适合处理有大量数据输入的工作；
- 最新的 HTML 技术虽然支持更多的界面元素，例如用于绘图的<canvas>、用于多媒体的<audio>、<video>等，但仍然不适合进行大量复杂图形界面的处理；
- 基于安全等因素的考虑，浏览器访问本地资源的能力较弱，或者受到限制，而 C/S 应用则很容易实现这一点。

因此，在一些现代的应用系统中，采用了结合 C/S 和 B/S 的混合结构，以吸取了二者的优点。对于那些客户端相对稳定，并且需要处理复杂业务逻辑、图形图像处理、大量数据输入的场合，采用 C/S 结构；而对那些需要频繁更新、发布信息的场合，采用 B/S 结构。一些常见的桌面型网络游戏就是采用这种结构：游戏客户端主体采用 C/S 结构，但在其中嵌入一个浏览器来整合二者的优点。

3. 云计算（Cloud Computing）模型

云计算模型是对分布式处理、并行处理和网格计算及分布式数据库的改进和商业化处理，

其前身是利用并行计算解决大型问题的网格计算和将计算资源作为可计量的服务提供的公用计算，在宽带技术和虚拟化技术高速发展成熟后云计算技术得以萌生。云计算描述了一种基于互联网的新的 IT 服务增加、使用和交付模式：所有服务采用租赁的方式提供给用户，用户无需购买。这可能从根本上改变将来软件和服务发布的模式。图 3.13 示意了云计算模式的拓扑结构。

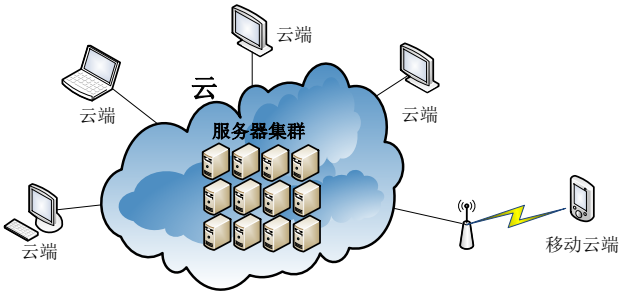


图 3.13 云计算模型的拓扑结构

为了支持大量用户，“云”中设置了服务器集群。集群可能由数量庞大的 PC 服务器组成，并采用虚拟化技术向用户推送服务。虚拟化是为某些对象创造的虚拟化（相对于真实）版本，比如操作系统、计算机系统、存储设备和网络资源等。它是表示计算机资源的抽象方法，通过虚拟化可以用与访问抽象前资源一致的方法访问抽象后的资源，从而隐藏属性和操作之间的差异，并允许通过一种通用的方式来查看和维护资源。

对于用户而言，他们并不需要了解“云”中的细节，不必具有相应的专业知识，也无需直接进行控制。用户只需通过互联网连上“云”，就能获得所需的服务。基于此，用户端系统（云端）得到简化。一个超级简化的例子是：云端甚至连操作系统都不需要，只要通过高速宽带连接连上云，那么就能在云端设备上显示桌面并获得相应服务。

从这个角度来看，云计算似乎是对集中处理模式的一种回归。然而，这两种模式有着巨大的区别：

- 集中系统中，终端通过终端线直接连接在主机上；而云系统中，云端设备通过网络连接到云上；
- 云端设备除了可以标准的云终端外，还可以是一个标准的桌面型计算机，或者任何形式的移动终端，它们都远比哑终端“聪明”；
- 云系统中，负责事务处理的不是大型主机，而是对用户透明的、庞大的服务器集群。集群中的服务器是同构的，且数量庞大。商用的云系统中的服务器少至数百台，而多至数百万台。

① 计算的体系结构

美国国家标准和技术研究院（NIST）定义的云计算体系结构可以用图 3.14 描述。

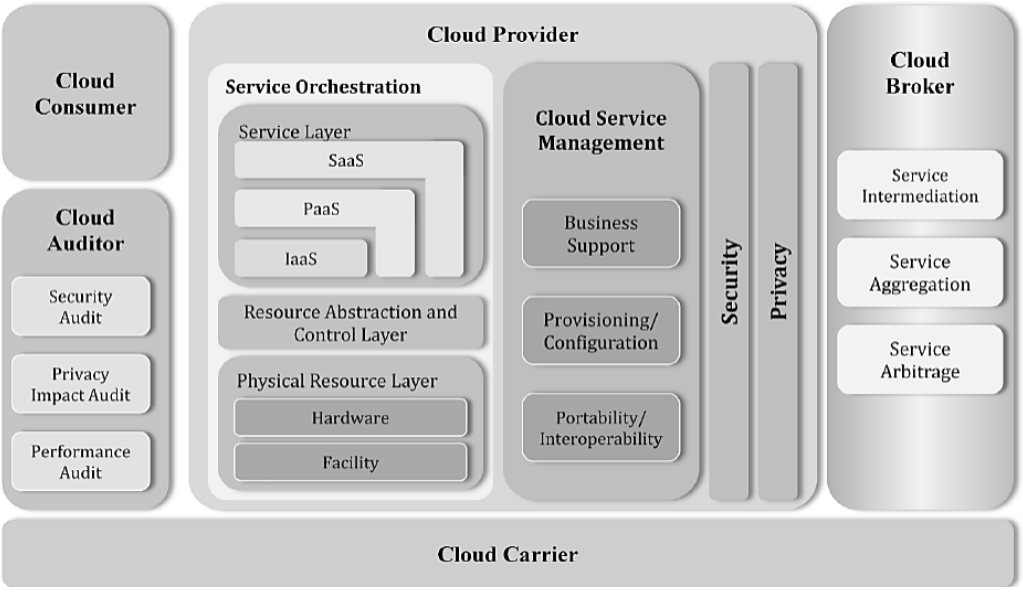


图 3.14 NIST 的云计算参考模型

② 基本特征

NIST 定义的云计算服务应该具备以下几条特征：

- 随需应变自助服务。
- 随时随地用任何网络设备访问。
- 多人共享资源池。
- 快速重新部署灵活度。
- 可被监控与量测的服务。

③ 服务模式

NIST 的云计算定义中明确了三种服务模式：

- 软件即服务（SaaS，Software as a Service）：用户使用应用程序，但不掌控操作系统、硬件或网络基础架构。SaaS 是一种服务观念的基础，用户以租赁而非购买的方式获取软件服务供应商提供的服务。
- 平台即服务（PaaS，Platform as a Service）：用户掌控运行应用程序的环境甚至部分主机掌控权，但并不掌控操作系统、硬件或网络基础架构。
- 基础架构即服务（IaaS，Infrastructure as a Service）：用户使用“基础计算资源”，如处理能力、存储空间、网络组件或中间件，能掌控操作系统、存储空间、已部署的应用程序及网络组件（如防火墙、负载平衡器等），但并不掌控云基础架构。

④ 部署模型

- 公用云（Public Cloud）

公用云服务可通过网络及第三方服务供应者，开放给用户使用。“公用”并不一定代表“免费”（但也可能代表免费或相当廉价），也不表示用户数据可供任何人随意查看，公用云供应者通常会对用户实施使用访问控制机制。公用云作为解决方案，既有弹性，又具备成本效益。

- 私有云（Private Cloud）

私有云具备许多公用云环境的优点。两者的差别在于私有云服务中，数据与程序皆在组

织内管理，因此受到网络带宽、安全、法规限制等因素影响较小。此外，私有云服务让供应者及用户更能掌控云基础架构、改善安全与弹性。

- 混合云（Hybrid Cloud）

混合云结合了公用云及私有云的特点。这个模式中，用户通常将非企业关键信息外包，并在公用云上处理，但同时掌控企业关键服务及数据。

⑤ 云安全

云计算因其特质，故可以在教育、社交、政务、存储、物联方面得到广泛应用。然而在用户大量参与的情况下，不可避免的出现了隐私问题。用户为了使用云服务，一般需要在云平台上共享信息。实际上，云计算的核心特征之一就是数据的储存和安全完全由云计算提供商负责，这可以降低了组织内部和个人成本。但是，数据共享就意味着需要承担隐私外泄的风险。因此，许多用户担心自己的隐私权会受到侵犯，其私密的信息会被泄露和滥用。云计算的隐私安全问题主要包括：

- 在未经授权的情况下，他人以不正当的方式进行数据侵入，获得用户数据。
- 政府部门或其他权利机构为达到目的对云计算平台上的信息进行检查，获取相应的资料已到达监管和控制的目的。
- 云计算提供商为获取商业利益对用户信息进行收集和处理。

上述问题都是云计算提供商需要迫切解决的问题。但无论如何，云计算是一种先进的体系结构，代表这未来发展的方向，是目前业界的一个大热点，是大数据处理不可或缺的基础。

3.2.5 控制模型

系统的结构性模型主要考虑如何将系统分解成若干子系统，但其中不需要也不应该包括控制信息。然而，系统结构肯定并且应当按照某些控制模型来组织子系统。这里所指的控制模型属于体系结构层面的内容，它控制子系统之间的工作流程。

常用的控制模型主要包括：

① 集中控制（Centralized Control）模型。一个子系统能够在总体上控制、启动、停止其他子系统。它也可以将控制移交给其他子系统，但必须能回收控制。

② 事件驱动控制（Event-driven Control）模型。子系统响应外部事件。这些事件可能来自于其他子系统或者系统使用环境。

控制模型支持结构性模型。所有的结构性模型都会用到以上两种控制模型。

1. 集中控制模型

在此模型中，一个子系统被赋予了系统控制器的角色，它能管理其他子系统的运行。根据子系统的执行情况，该模型又分为两类：

(1) 调用—返回模型（The Call-Return Model）

调用—返回模型实际上是一种自顶向下的层次模型。上层的子系统控制下层子系统的运行。图 3.15 是这种模型的示意，其中的主程序扮演了系统控制器的角色。这种模型只适用于顺序执行系统。我们编写的 C 语言程序都是这样的。

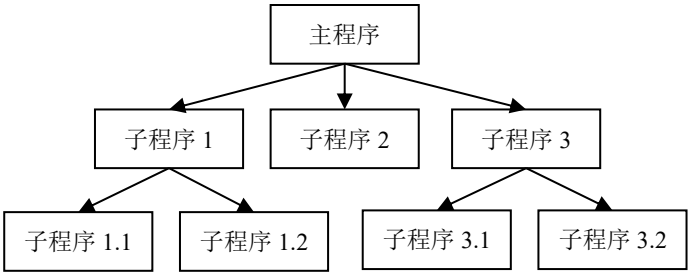


图 3.15 调用—返回模型

(2) 管理者模型（The Manager Model）

这种模型多由并行系统实现。一个系统组件被设计成为系统的管理者（系统控制器），它控制系统中其他进程的启动、终止和协调工作。这里提到的进程系指能够与其他进程并行执行的子系统或者模块。图 3.16 是这种模型的示意图。

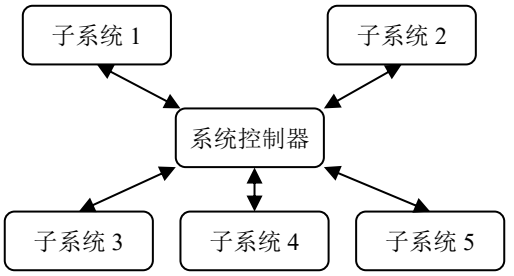


图 3.16 管着者模型

2. 事件驱动控制模型

事件驱动控制模型由外部事件驱动。事件（Event）由系统的外部环境产生，如用户操作可以触发一个事件。事件与普通输入不同：普通输入在子系统的时间控制序列之内，而事件则不受子系统的控制而来自于外部。事件往往与一个消息（Message）相关联。这里的消息系指由携带了有用信息的信号（Signal）。当一个事件被触发，于此相关联的消息会被发送到应用系统，然后由应用系统决定是否处理这个事件/消息。此外，系统的各子系统也可以互相发送消息，但不会触发事件。

典型的使用事件驱动控制模型的实例是窗口系统。当用户在窗口中发起一个动作，例如点击鼠标左键，那么就会触发“left button click”事件，该事件相关的消息被转发。该消息除了包含事件类型信息外，还可能包含点击处的坐标、Shift/Ctrl/Alt 键的状态等信息。这个消息会传递给被点击的窗口，它会响应这个点击事件并给出反馈信息。

根据系统处理事件的方式，该模型可以分为两类：

(1) 广播模型（Broadcast Model）

此模型中，在系统级别层面上设置了事件/消息监听器，它负责监听事件是否发生以及收集事件的相关信息；在应用层面上，需要处理特定事件的子系统会注册一系列事件处理器（Event Handler）。当系统的事件处理器接收到一个事件后，采用广播模式向各子系统发布事件。任何注册了该事件处理器的子系统将被激活并处理这个事件。很多的窗口系统采用了此模型。图 3.17 示意了这种工作方式。

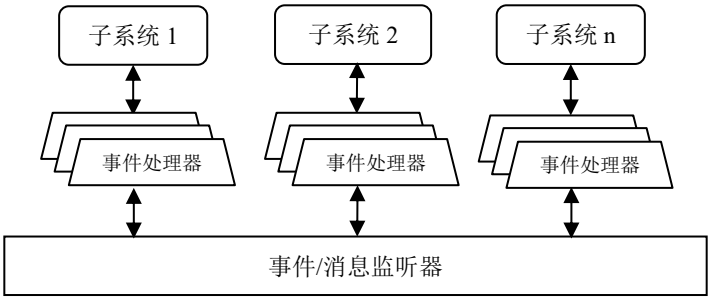


图 3.17 广播模型

(2) 中断驱动模型（Interrupt-driven Model）

这种模型多用于实时系统当中。在该模型中，一系列与特定中断事件绑定的中断处理器（Interrupt Handler）始终处于监听状态。一旦中断事件发生，该中断处理器能接受到该事件，并将事件传递给相应的进程。图 3.18 示意了这种模型的结构。图中，术语“中断向量（Interrupt Vector）”可以简单地理解为由中断处理器入口地址组成的线性结构。

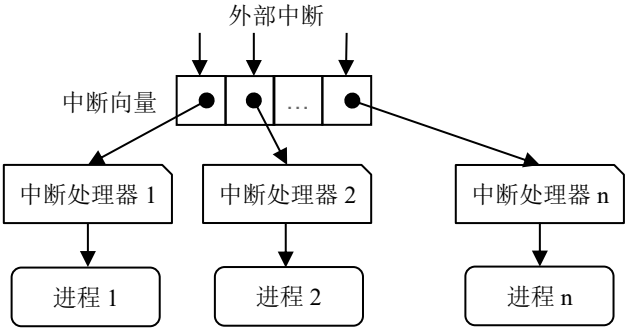


图 3.18 中断模型

在设计过程的此阶段，设计者应该根据系统的需求以及系统运行的环境，选择一种控制结构，并在其中做出必要的适应性变更。

3.2.6 模块分解

一旦前面的工作完成后，接下来需要做的工作就是进行系统的模块分解（Modular Decomposition）。

模块（Module）是命名的程序对象的集合，如过程、函数、类等。模块是构成软件系统结构的基本元素。一个大的系统总是由若干稍小的功能模块聚合而成的，而这些子模块可能又由更小的模块构成。如何对系统进行合理的分解是个值得考虑的问题，因为模块划分好坏将会决定整个系统的质量。

前面内容中提到的“子系统（Subsystem）”是一个与模块类似的概念，因此这里不对它们做特别的区分。

1. 模块分解的目的

模块分解的目的是将系统“分而治之”，以降低问题的复杂性，使软件结构清晰，便于阅读和理解，易于测试和调试，因而也有助于提高软件的可靠性。

下面对模块分解能够降低软件复杂度进行简单证明。

令 $C(X)$表示问题 X 的复杂度函数；$E(X)$为解决问题X所需工作量的复杂度函数；

若有问题 P_1 和 P_2 ，$C(P_1)>C(P_2)$，显然 $E(P_1)>E(P_2)$。

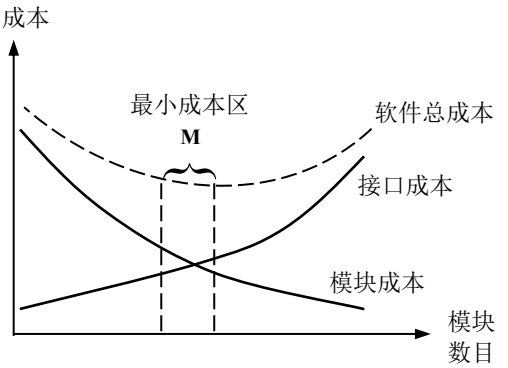


图 3.19 模块分解与软件成本的关系

由经验 $C(P_1+P_2)>C(P_1)+C(P_2)$

于是 $E(P_1+P_2)>E(P_1)+E(P_2)$

即将问题（P_1+P_2）划分为两个问题 P_1 和 P_2 后，其工作量和复杂度都降低。

但并非模块分得越小越好，因为模块数越多，模块之间接口的复杂度和工作量将增加。显然，每个软件系统都有一个分解的最佳模块数 M，注意选择分解的最佳模块数，可以在降低问题复杂度的同时获得较低的成本。图 3.19 描述了模块分解与软件成本的关系。

2. 模块的独立性

模块具有如下三个基本属性：

① 功能：指该模块实现什么功能，做什么事情。必须注意，这里所说的模块功能，应是该模块本身的功能加上它所调用的所有子模块的功能。

② 逻辑：描述模块内部怎么做。

③ 状态：该模块使用时的环境和条件。

所谓模块的独立性，是指软件系统中每个模块只涉及软件要求的具体的子功能，而和软件系统中其他的模块的接口是简单的。即独立性的模块应具有专一功能，模块之间无过多的相互作用的模块。

这种类型的模块可以并行开发，模块独立性越强，并行开发越容易。独立性强的模块，还能减少错误的影响，使模块容易组合、修改及测试。

模块独立性的度量标准是两个定性准则，即耦合性和内聚性。

⑴ 耦合性（Coupling）

耦合性是指软件结构中模块之间相互连接的紧密程度，是模块间相互连接性的度量。模块分解的一个目标是使模块之间的联系尽可能少。如图 3.20 所示，模块间联系可从三个方面衡量：

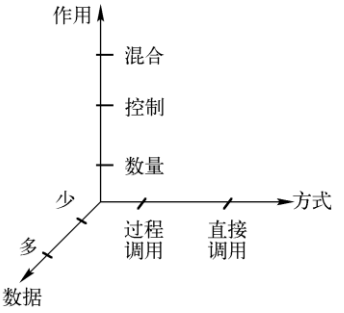


图 3.20　模块间联系

① 方式。模块间联系方式有“直接引用”或“过程语句调用”。显然直接引用方式模块间联系紧密。

② 作用。模块间传送的公用信息（参数）类型，可为“数据型”、“控制型”或“混合型”（数据/控制型），控制型信息使模块间联系紧密。

③ 数量。模块间传送的公用信息的数量越大，模块间联系越紧密。模块间公用的信息（如参数等）应尽量少。

模块分解的一个目标是使块间联系尽可能少。为达到这个目标可采取以下措施：每个模块应采用过程语句（或函数）等间接调用等方式调用其他模块。模块间传送的参数为数据型。

按照耦合性的高低，耦合性的类型如图 3.21 所示。

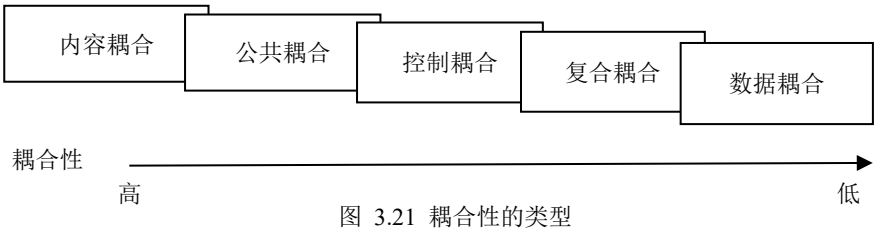


图 3.21 耦合性的类型

- 内容耦合的模块常具有以下特征：一个模块直接访问另一个模块的内部数据；一个模块不通过正常入口转到另一个模块的内部；一个模块有多个入口或者两个模块有部分代码重叠。
- 若干模块访问一个公共的数据环境，则它们之间的耦合称为公共耦合。公共环境可为全局数据结构、共享的通信区、内存的公共覆盖区等。显然，公共数据区的变化，将影响所有公共耦合模块，严重影响模块的可靠性和可适应性，降低软件的可读性。
- 控制耦合：是一个模块传递给另一个模块的信息是用于控制该模块内部逻辑的控制信号。显然，对被控制模块的任何修改，都会影响控制模块。
- 复合耦合：是一个模块传送给另一个模块的参数是一个复合的数据结构，例如，包含几个数据单项的记录。
- 数据耦合：一个模块传送给另一个模块的参数是一个单个的数据项或者单个数据项组成的数组。

⑵内聚性（Cohesion）

内聚性用来表示一个模块内部各种数据和各种处理之间联系的紧密程度，它是从功能的角度来度量模块内的联系的。显然，模块内部联系越紧，即内聚性越强，模块独立性越好。

块内部联系的类型如图 3.22 所示。

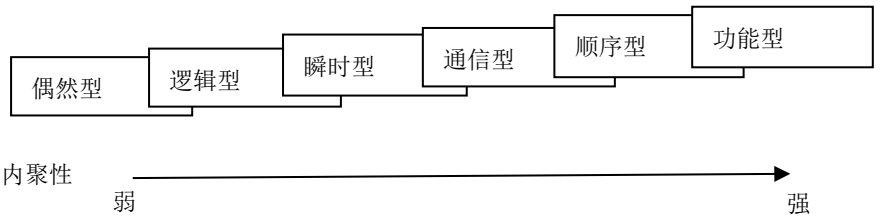


图 3.22 块内联系的类型

- 偶然型：又称为巧合型，为了节约空间，将毫无关系（或联系不多）的各成分放在一个模块中，这样的模块显然不易理解，不易修改。
- 逻辑型：将几个逻辑上相似的功能放在一个模块中，调用时由调用模块所传递的参数确定执行的功能。由于要进行控制参数的传递，必然要影响模块的内聚性。
- 瞬时型（Temporal Cohesion）：将需要同时执行的成分放在一个模块中，因为模块中的各功能与时间有关，因此又称为时间内聚或经典内聚。例如，初始化模块、中止模块等，这类模块内部结构较简单，一般较少判定，因此比逻辑内聚强，但是由于将多个功能放在一起，将给修改和维护造成困难。
- 通信型：模块中的成分引用共同的输入数据，或者产生相同的输出数据，则称为是通信内聚模块。通信型模块比瞬时型模块的内聚性强，因为模块中包含了许多独立的功能，但却引用相同数据。通信模块一般可以通过数据流图来定义。
- 顺序型：模块中某个成分的输出是另一成分的输入。由于这类模块无论是数据还是在执行顺序上，模块中的一部分都依赖于另外一部分，因此具有较好的内聚性。
- 功能型：一个模块包括而且仅包括完成某一具体功能所必须的所有成分，或者说，模块的所有成分都是为完成该功能而协同工作、紧密联系、不可分割的，则称该模块是功能型的。

模块分解的总则是：降低块间联系，提高块内联系。

3. 分解模型

模块分解的模型有两种模型：

⑴面向对象模型（Object-Oriented Model）

在此模型中，系统被分解为若干能互相通信的、松耦合的对象，这些对象都具有精心设计的接口。一个对象通过调用其他对象的接口获得后者提供的服务。在实现的系统中，会采用某种控制结构来协调对象之间的运作。

【例 3-1】考虑一个简化的成绩管理系统的模块分解。根据常识，该系统中的活跃对象有教师、学生、课程、成绩单。这四类对象的扼要关系是：

(1) 教师任课
(2) 学生选课
(3) 教师评分出学生课程成绩

据此，图 3.23 示意了该系统的面向对象结构模型。其中，虚线表示依赖关系，实现表示引用关系。请读者注意，这不是一个精确的 UML 类图，而只是一个类示意图。

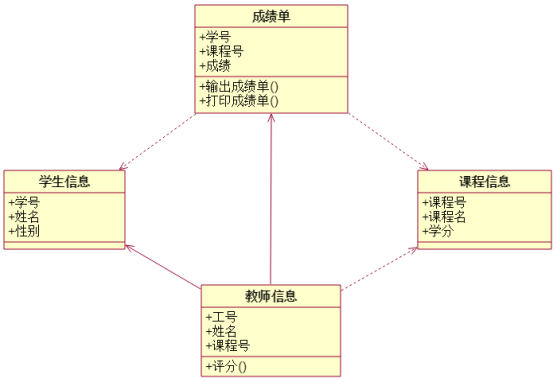


图 3.23 成绩管理系统的面向对象结构模型

关于对象设计的细节将在 3.3 中讨论。

面向对象模型的有点是显而易见的。其一，对象可以视为是对客观世界中实体的一对一映像，因此对象的设计是很容易理解的；其二，因为对象之间是松耦合的，所以只要对象的接口不变，那么对象内部细节的修改不会影响到其他对象；其三，对象是高度可重用的。

此模型的缺点是：对象在获得服务时必须显式引用其他对象的名字和接口。一旦接口为了满足系统需求而发生改变，那么引用该接口的对象不得不跟着改变。这种改变的成本也许会变得很高。所以，这要求设计者必须精心设计对象及其接口，以保持长时间的不变性和稳定性。

⑵数据流模型（Data-flow Model）

在模型中，一种称为函数转换（Functional Transformation）的单元处理数据的输入和输出。数据流从一个单元流向下一个单元时被处理转换，最终从输入被转换为输出。函数转换单元的执行可以是顺序的，也可以是并行的，这依赖于系统的运作方式。

图 3.24 示意了上面提到的成绩管理系统的数据流模型。

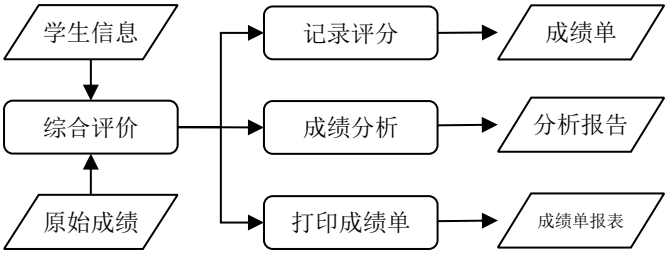


图 3.24 成绩管理系统的数据流模型

数据流模型的优点是：

- 支持转换单元的可重用；
- 数据从输入到输出的过程是自然的，符合人们的思考习惯；
- 在系统中添加转换单元是直接的；
- 可以方便地用顺序或者并行方式实现。

它的主要缺点是：数据在两个转换单元间传递时必须是两个单元都可理解的，因此要使用通用的数据格式。这无疑会增加系统的管理成本，并且很难将不规范的数据格式集成到系统中。

3.3 面向对象设计

面向对象的系统开发分为三个阶段：

(1) 面向对象分析（Object-Oriented Analysis，OOA）；

(2) 面向对象设计（Object-Oriented Design，OOD）；

(3) 面向对象程序设计（Object-Oriented Programming，OOP）。

从一个阶段过渡到下一个阶段应该是平滑无缝的。过渡中，可能涉及对已有设计的改进，例如添加一些细节。由于信息是被封装的，因此与信息呈现相关的设计细节的决策可以推迟到实现阶段。这意味着，系统设计者可以不被系统的实现细节束缚住，他们更应该关注的是如何设计出在不同环境下都能运行的系统。

在三个阶段中，OOD 是面向对象方法（OO）的核心阶段。按照描述 OO 方法的“喷泉模型”，软件生命期的各阶段交叠回溯，整个生命期的概念、术语、描述方式具有一致性，因此从分析到设计无须表示方式的转换，只是分析和设计的任务分工与侧重不同。

OOA 建立的是应用领域面向对象的模型，而 OOD 建立的则是软件系统的模型。与 OOA 的模型比较，OOD 模型的抽象层次较低，因为它包含了与具体实现有关的细节，但是建模的原则和方法是相同的。

3.3.1 面向对象设计的准则和基本任务

1. 设计准则

建立 OOD 模型，可以看作是按照设计的准则，对分析模型进行细化。虽然这些设计准则并非为面向对象的系统独用，但对面向对象设计起着重要的支持作用。面向对象的设计准则有：

⑴ 抽象

抽象是指强调实体的本质及内在的属性，而忽略了一些无关紧要的属性。在系统开发中，分析阶段使用抽象仅仅涉及应用域的概念，在理解问题域以前不考虑设计与实现。而在面向对象的设计阶段，抽象概念不仅用于子系统，而且还用于对象设计中。对象具有极强的抽象表达能力，类实现了对象的数据和行为的抽象。

⑵信息隐蔽

信息隐蔽在面向对象的方法中的具体体现即是“封装性”，是保证软件部件具有优良的模块性的基础。封装性将对象的属性及操作（服务）结合为一个整体，尽可能屏蔽对象的内部细节，软件部件外部对内部的访问通过接口实现。

在面向对象的语言中，类（class）是实现抽象和信息隐蔽的封装部件。类的定义将其说明（用户可见的外部接口）与实现（用户内部实现）分开，而对其内部的实现按照具体定义的作用域提供保护。

⑶弱耦合

按照抽象与封装性，弱耦合是指子系统之间的联系应该尽量的少。子系统应具有良好的

接口，子系统通过接口与系统的其他部分联系。

⑷强内聚

强内聚指子系统内部由一些关系密切的类构成，除了少数的“通信类”外，子系统中的类应该只与该子系统中的其他类协作，构成具有强内聚性的子系统。

⑸可重用

只有构建独立性强（弱耦合、强内聚）的子系统和类，才能够有效地提高所设计的部件的可重用性。

2. 基本任务

面向对象的设计是面向对象方法在软件设计阶段应用与扩展的结果，是将 OOA 所创建的分析模型转换为设计模型，解决“如何做”的问题。面向对象设计的主要目标是提高生产效率，提高质量和可维护性。

OOA 主要考虑系统做什么，而不关心系统如何实现的问题。在 OOD 中为了实现系统，需要以 OOA 模型为基础，重新定义或补充一些新的类，或在原有类中补充或修改一些属性及操作。因此，OOD 的目标是产生一个满足用户需求的可实现的 OOD 模型。

OOD 的主要任务是对象设计。对象是一种封装了其状态以及改变状态的一系列操作的实体。在面向对象技术中，状态用一系列对象属性（Attribute）来表示；操作常称为方法（Method），用来描述对象可以提供的服务。对象由其关联的类创建。

对象设计确定解空间中的类、关联、接口形式及实现服务的算法。

3.3.2 设计过程

在 Ian Sommerville 的论述中提到，常用 OOD 设计过程包含如下过程：

1. 理解和定义系统上下文及用例模型

设计的第一个阶段是设计出一个容易理解的、描述系统和外部应用环境关系的模型。

(1) 系统上下文

系统上下文（System Context）是一种静态模型，它描述系统结构的概貌。我们可以用一个框图（关联模型，Association Model）来刻画系统上下文。

【例 3-2】一个报表系统的扼要功能模型（图 3.25）

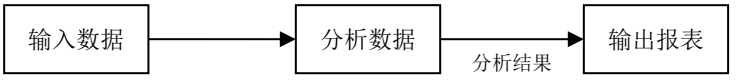


图 3.25 简易报表系统的功能模型

上述框图可以扩展成 UML 包图。这里不再详细描述扩展过程。

(2) 用例模型

用例模型（Use-case Model）是一种动态模型，它描述了系统如何与环境交互。可以用 UML 用例图来刻画此模型。

【例 3-3】报表系统的用例图（图 3.26）

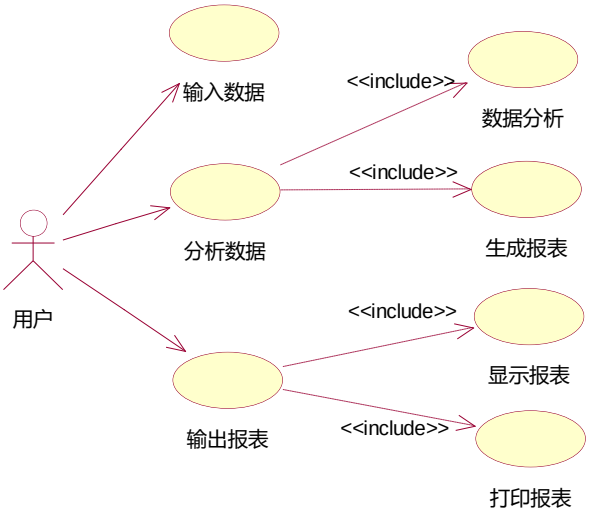


图 3.26 用例图

2. 设计系统体系结构

一旦环境与系统的交互方式定下来，那么我们就可以据此进行系统体系结构设计。我们可以用 UML 包图来展示设计结果。

【例 3-4】报表系统的结构（图 3.27）

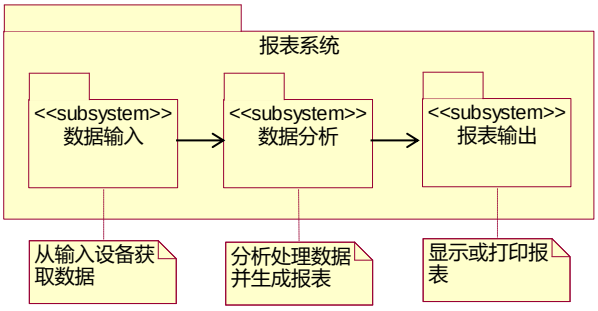


图 3.27 报表系统的包图

3. 标识系统中主要的对象

接下来的过程是标识系统中主要的对象。这里所说的对象实际上系指对象所属的类。

在实践中，有多种方法可以在系统中发现对象和对象所属的类。例如基于场景的分析可以帮助设计者发现在每个场景中活跃的对象，并且能很容易地标识出这些对象的属性和方法；在后期的设计中，更多的细节会被添加到类中。

【例 3-5】报表系统中主要的类。图 3.28 示意了系统中主要的类，没有设计其它的辅助类或者接口。

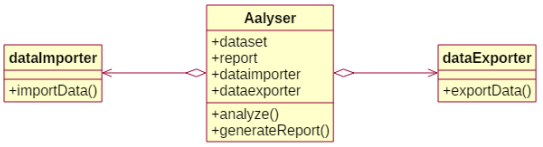


图 3.28 报表系统中的类

这里需要说明的是，图 3.28 所示的类并不完整。这里只简单地展示出将用到的类的概貌，没有太多的细节。此外，上述类设计是有缺陷的，我们会在后续的讨论中认识到它的不足。

4. 形成设计模型

设计模型（Design Model）展示了对象（类）之间的关系，是桥接系统需求与实现的关键。

设计模型应当是抽象的，不包含不必要的细节。但是另一方面，模型的细节又必须是足够的，以方便程序员的程序设计。

模型设计中重要的一步是决定采用哪种设计模型，以及模型必须的细节。这在很大程度上取决于系统的类型，不同类型的系统应该有不同的设计。

有两种设计模型可共选择：

(1) 静态模型：描述系统中对象静态关系的结构；

(2) 动态模型：描述系统中对象交互的动态结构。

在前面内容中提到的用例图、包图等用来展示系统的静态结构。常用来展示动态结构的 UML 图是顺序图，它可以清晰地展示系统中各部件的交互情况。

【例 3-5】报表系统的动态模型：顺序图（图 3.29）。

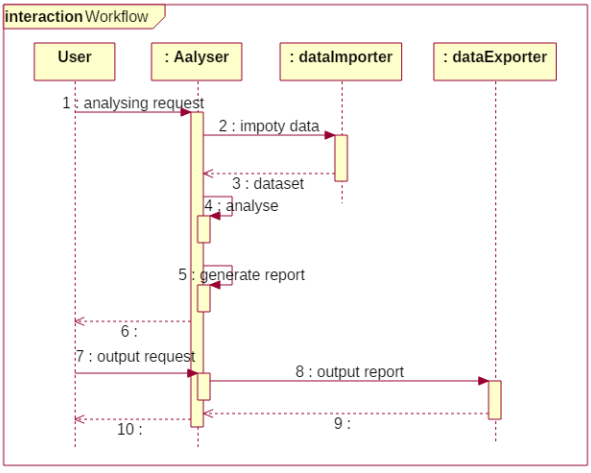


图 3.29 报表系统的顺序图

5. 说明对象接口

对象的接口（对外暴露的方法）是各对象之间交互的界面。一旦接口定下来，那么这些接口就应该有相当的稳定性，并且其他对象的开发者应该确信这些接口能够实现。

接口设计者应当避免设计的接口暴露内部信息，信息应当是隐藏的。如果信息隐藏做得很成功，那么类的内部信息的改变就不会影响到其他使用该类的对象。

【例 3-6】类 Analyser 的接口说明（类 Java 语法）

```
class Analyser
{
   public dataset;        //数据集
   public report;         //报表
   public dataimporter; //数据输入器
   public dataexporter; //报表输出器
   public analyze();      //数据分析器
   public generateReport(); //报表生成器
};
```

以上设计沿袭了一种非常自然的设计思路。但实际上，这种思路是有缺陷的，因为这不符合多条对象设计原则。请读者参阅后续的内容思考修改方案。

3.3.3 对象设计

在面向对象的系统中，模块、数据结构及接口等都集中地体现在对象和对象层次结构中，系统开发的全过程都与对象层次结构直接相关，是面向对象系统的基础和核心。面向对象的设计通过对象的认定和对象层次结构的组织，确定解空间中应存在的对象和对象层次结构，并确定外部接口和主要的数据结构。

对象设计实际上指的是对象所属类（class）的设计，即为每个类的属性和操作进行详细设计，包括属性和操作的数据结构、实现算法，以及类之间的关联。在进行对象设计的同时也要进行消息设计，即设计连接类与它的协作者之间的消息规约（Specification of The Messages）。

这里我们展示一些常用于类设计的原则。

1. 单一职责原则（SRP，The Single Responsibility Principle）

一个类要改变的理由永远不能多于一个，或者说，一个类应该负担最少的责任。

如果修改一个类的动机多于一个，那么这个类所的责任就不止一个。而这样的类是“不好”的。类的设计应当本着“各人自扫门前雪”的态度进行。

在实际应用中，要将类的责任分得很清楚并不容易，因为一些想法总显得自然而然。要设计出良好的类需要程序员大量的经验累积。

2. 开放封闭原则（OCR，The Open-Closed Principle）

类对扩展是开放的，而对修改是封闭的，即在扩展一个类时，不应该修改该类。

一个类符合 OCR 原则的表现为：

① 类的行为是可以扩展的。

② 类的源码是不可侵犯的，即在扩展时，不允许对该类的源码进行修改。

要实现以上两点，抽象是关键。抽象机制（例如 C++的虚函数）允许程序在这些抽象类的子类中覆盖那些抽象行为，进而扩展了类的行为。当然，我们还可以通过更多的抽象继承来获得更多的扩展。

3. 依赖倒置原则（DIP，The Dependency Inversion Principle）

高层模块不应该依赖于低层模块。二者都应该依赖于抽象；抽象不依赖于细节，细节依赖于抽象。

从程序设计的角度来看，如果一个高层类依赖于一个具体实现类，而当应用的需求改变，高层类需要另外的实现类支撑时，那么高层类不得不重新编码。更好的设计是：增加一个中间抽象类，使高层类依赖于这个抽象类，而其他实现类是这个抽象类的子类，那么增加、更改一个实现就变得非常容易了，高层类和抽象类都不会有变动。

4. 接口分离原则（ISP，The Interface Segregation Principle）

客户代码不应该依赖于那些它们用不到的接口。

这条原则说明，不要试图去设计一个大而全的接口/基类，它们的设计应该是小而紧凑的，在功能上相对独立的，子类可以通过多继承（或者编程语言支持的其他方式）获取所需的功能。反过来说，如果一个子类实现/继承自一个接口/基类，但必须实现一些不需要的功能，那么在设计上，那个祖先接口/基类就应该被拆分成完成不同功能的多个接口/基类。

5. 里氏替换原则（LSP，Liskov Substitution Principle）

使用父类指针或者引用作为形式参数的函数，能够使用父类的子类对象作为实际参数，并且不应该试图去了解子类。

这条原则的转述为：在使用父类指针或引用的场合，子类对象可以完全替换父类对象，并且程序实体并不能察觉这种替换。

为了让多态正确地发生，那么父类应该尽量定义成抽象的（如 C++的抽象类、Java 的接口）；子类尽量从抽象类而不是实类继承。

6. 组合/聚合复用原则（CARP ，Composite/Aggregate Reuse Principle）

在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分，新对象通过向这些对象的委派（Delegation）发消息达到复用已有功能的目的。

换言之，就是优先使用组合/聚合而非继承；只有以下条件同时满足时，才可以使用继承：

① 子类是父类的一个特殊种类，而不是父类的一个角色。要正确区分“Has-A”和“Is-A”这两种关系：只有“Is-A”关系才符合继承关系，“Has-A”关系应当用聚合来描述。

② 永远不会出现需要将子类换成另外一个类的子类的情况。如果不能肯定将来是否会出现这种情况，就不要使用继承。

③ 子类具有扩展父类的责任，而不是具有覆盖（Override）或抵消（Nullify）父类的责任。如果一个子类需要大量覆盖父类的行为，那么这个类就不应该是这个父类的子类。

④ 只有在分类学角度上有意义时，才可以使用继承。

7. 最少知识原则（LKP ，Least Knowledge Principle）

一个对象应当对其他对象有尽可能少的了解。

LKP 原则又称为迪米特法则（Law of Demete），其含义是：每一个软件实体对其他的实体都只有最少的知识，而且局限于那些本实体密切相关的软件实体里。就是说，如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用；如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

无论如何，上述设计原则都不是死的教条。设计者应该在对象/类设计过程中，灵活运用设计规则，以期设计出结构精良和高度可重用的对象/类。

3.4 详细设计描述工具

在总体设计阶段，完成了对系统的体系结构的描述。还必须对系统结构做进一步的细化。

详细设计阶段的任务是开发一个可以直接转换为程序的软件表示，即对系统中每个模块的内部过程进行设计和描述。

常用的描述工具有：传统的流程图、结构化流程图（N-S 图）、问题分析图（PAD 图）、PDL 语言等。

1. 流程图

图 3.30 描述了一个具有多种结构的流程图。有关流程图的画法及使用，这里不再讨论。

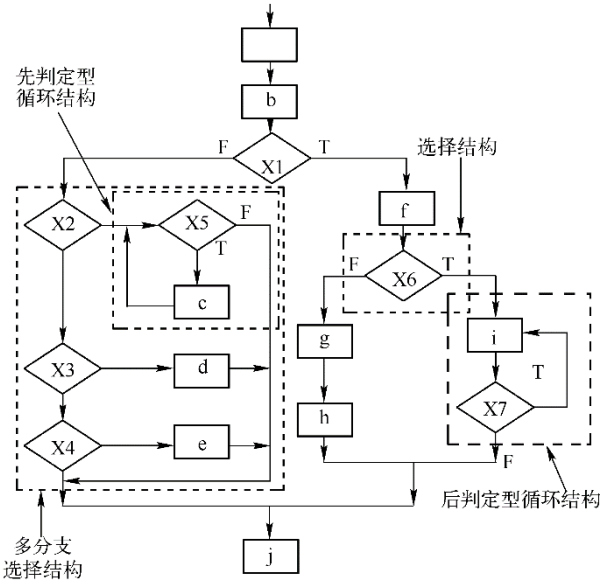


图 3.30 传统的流程图

2. N-S 图

N-S 图，又称为盒图，是一种结构化的流程图，由而且仅由顺序、选择、循环三种基本结构组成。其基本图例如图 3.31 所示。

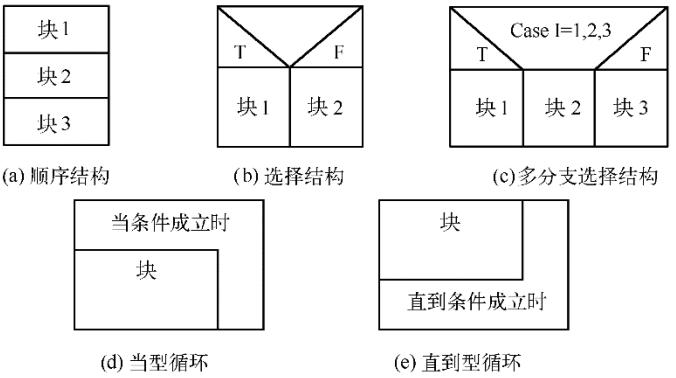


图 3.31　N-S 图

3. PAD 图

PAD 图，即问题分析图（Problem Analysis Diagram），是一种结构化的图，其基本控制结构如图 3.32 所示，也由三种基本结构组成。其中选择结构分为两分支和多分支，循环结构分为 WHILE 型循环和 UNTIL 型循环两类。

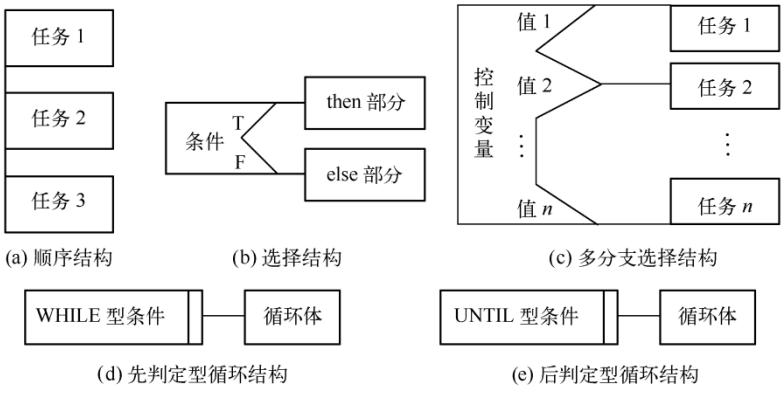


图 3.32 PAD 图基本结构

3.5 用户界面设计

随着各种应用软件的面市，作为人机接口的用户界面具有越来越重要的作用，用户界面是否友好将直接影响到软件的寿命与竞争力。因此，对用户界面的设计必须予以足够的重视。本节将对用户界面设计中的主要问题进行讨论。

3.5.1 用户界面设计的特性与设计任务

1. 用户界面设计的特性

一个好的用户界面应具有以下特性：

⑴可使用性

① 使用简单，用户对界面的学习周期应该较短。
② 用户界面中所用术语应该标准化，采用用户熟悉的标准系列；同时术语应该具有一致性，在系统任何地方出现的相同概念的术语都是一致的。
③ 提供 Help 功能，以便用户在需要时获得指导。
④ 系统响应速度要尽可能的快，不要让用户产生系统停止运行的错觉；系统成本也应该控制在低水平。
⑤ 具有容错能力，就是当用户输入了错误的数据时系统应该具有处理这种错误的能力，而不是简单地退出甚至崩溃。

⑵灵活性

① 考虑用户的特点、能力、知识水平，提供不同的指导、帮助信息和快捷方式，且提供的信息不能太专业化，否则会使用户难于理解。
② 提供不同的系统响应信息。根据用户操作的熟练程度，系统提供的信息应该有繁简之分。
③ 提供根据用户需求定制和修改界面功能。初学者、熟练用户和专家用户对界面的繁简程度有不同的要求，界面的定制功能可以适应这种要求。

⑶界面的复杂性与可靠性

复杂性指界面规模及组织的复杂程度。应该越简单越好。

可靠性指无故障使用的时间间隔。用户界面应该能够保证用户正确、可靠地使用系统，以及程序、数据的安全。

2. 用户界面设计的任务

这部分工作应该与软件需求分析同步进行。包括以下内容：

⑴用户特性分析——建立用户模型

了解所有用户的技能和经验，针对用户能力设计或更改界面。可从以下方面分析：用户类型，通常分为外行型、初学型、熟练型和专家型；用户特性度量，它与用户使用模式和用户群体能力有关，包括用户使用频度、用户用机能力、用户的知识、思维能力等。

⑵用户界面的任务分析——建立任务模型（DFD 图）

这是对系统内部活动的分解，不仅要进行功能分解（用 DFD 图描述），还要包括与人相关的活动。每个加工即为一个功能或任务。

⑶确定用户界面类型

应用程序的界面一般分为三种：

- 字符界面。例如，在 Windows 命令提示符下运行的很多应用都采用字符界面；
- GUI 界面。例如，我们常用的办公软件都有漂亮的 GUI 界面；
- 无交互界面。很多的系统级服务应用都是没有交互界面的。

软件的开发者应根据应用的需要来确定使用哪种界面。

3.5.2 用户界面设计的基本原则

软件的设计者往往会不自觉地根据自己的常识和喜好来安排界面、操作、颜色搭配等。另外，还常常按照自己的思路来设计程序的流程，也就是试图控制用户。这些都是不好的设计习惯，应该努力避免。

用户界面设计的一条总原则就是：以人为本，以用户的体验为准。Ben Shneiderman 提出了如下 8 条界面设计的黄金原则：

① 争取保持一致性。在类似的环境中应要求一致的动作序列；在提示、菜单和帮助屏幕中应使用相同的术语；应始终使用一致的颜色、布局、大写和字体等。异常情况，如要求确认、删除和密码而没有回显，应是可理解的，并且数量有限。

② 满足普遍可用性的需求。认识到不同用户和可塑性设计的要求，可使内容的转换更便捷。应意识到新用户和专家用户之间的区别，并在设计中体现这些区别。

③ 提供信息反馈。对每个用户动作都应提供系统犯规。常用和较少用动作应提供适当的响应，而对少用和主要动作则应提供更多的响应。

④ 设计对话框已产生结束信息。应把动作序列分组，每组有开始、中间和结束 3 个阶段。每组动作完成后提供信息反馈，这能给操作者完成任务的满足感、轻松感。

⑤ 预防错误。要尽可能地设计用户不易犯严重错误的系统。如果用户犯错，界面应检测到错误并提供简单、用建设性和具体的说明来恢复。错误的动作不应引起系统状态的变化，或者界面应给出恢复状态的说明。

⑥ 允许动作回退。应尽可能允许动作回退。这个特性能减轻焦虑，因为用户知道错误动作能够撤销，而且鼓励探索不熟悉的选项。

⑦ 支持内部控制点。有经验的用户一般都有一种他们能掌控界面且界面响应他们动作的强烈渴望。他们不希望熟悉的行为发生意外或者改变，并且会因不能得到想要的结果而感到郁闷。

⑧ 减轻短期记忆负担。由于人类利用短期记忆进行信息处理的能力有限，因此要求设计的界面中避免要求用户必须记住某个屏幕上的信息，然后在另一个屏幕上使用这些信息。

当然，以上原则不应教条地生搬硬套，必须根据具体的环境去解释、改进和扩充。

3.5.3 用户界面的基本元素

现在的应用系统一般都具有漂亮的 GUI(Graphic User Interface)界面。GUI 界面易学易用，并且显示的信息量大，是进行界面设计的首选。这里我们也主要以 GUI 界面来进行讨论。

GUI 界面的主要元素有：窗口（Window）、图标（Icon）、菜单（Menu）、图像（Graphics）、指点（Pointing）。

⑴ 窗口

一般桌面系统下的窗口包含这些部件：标题杠（Title Bar）、菜单条（Menu Bar）、工具栏（Tool Bar）、状态栏（Status Bar）、用户工作区（Client Area）。

移动终端因其屏幕尺寸的限制，其上的窗口布局与桌面系统的窗口布局有很大的不同：它们更加简洁，重点更加突出。

⑵图标

图标是窗口中的一个小图像，它是一种快捷方式，往往代表了一个程序、一个文件/文件夹，或者一项操作。用户通过单击或双击图标就可以启动响应的动作。

⑶菜单

按照显示方式可以分为：正文菜单、图标菜单、正文和图标混合菜单等；按屏幕位置和操作风格又可以分为：弹出式、下拉式、嵌入式等。

移动终端因其屏幕较小，因此其上的菜单布局与桌面系统有很大的不同，往往采用列表形式，显得紧凑而重点突出。

⑷图像

在用户界面中，适当地加入图形图像，将能够更加形象地为用户提供有用的信息。在这种界面中，图像常做这样的处理：隐蔽和再现、滚动/滑动、动画等。这些操作可以为界面增加动态效果。不过需要注意的是，人的眼睛对运动的事物非常的敏感，所以太多的动画或闪烁不但起不到帮助的作用，相反，会使用户的注意力分散，从而使界面的可用性降低。

⑸指点

GUI 界面往往会用到鼠标这样的指点设备，同时它也是一种选择设备。用户使用指点设备来定位光标、选择菜单或者激活屏幕上的目标，这比使用键盘这样的设备要方便和快捷。

在移动终端上，要求系统能够识别如手指轻触，以及像滑动、转动等这样的手势操作。

3.5.4 用户交互

一个系统的界面是用户与系统进行交互的门户。良好的交互风格将会给用户以正面的引导和激励，给他们以掌控感和成就感，使他们能够始终保持使用系统的欲望。与之相反，糟糕的交互风格可能会因其繁琐的操作、不友好的信息呈现或者权威性的强制模式，极大地打击用户的热情，从而使用户产生烦躁情绪，甚至放弃对系统的使用。因此，仔细选择交互风格是设计过程中非常重要的一个环节。

1. 交互风格

在界面设计中，常被设计者纳入考虑中的交互风格有 5 种：直接操纵、菜单选择、表格填充、命令语言和自然语言。下面我们只对前三种进行讨论。

⑴直接操纵

直接操纵能够使用户以非常直观的方式操作对象，其动作是可见的。例如，把文档图标直接拖拽到回收站（垃圾桶）图标上，或者选中文档后直接按 DEL 键就能删除该文档。这种

操作显然是相当鼓舞人心的，因为用户不必去键入命令。要知道，命令对非专家用户来说，有时是繁琐和难以记忆的。

直接操纵被广泛地应用在软件设计当中。著名的“所见即所得（WYSIWYG）”技术就是其中一项。这项技术在应用软件系统，如 Word、电子表格、游戏、计算机辅助设计（CAD）、体感游戏、3D 游戏、虚拟现实和增强现实中得到广泛应用等中被广泛采用。

直接操纵模式在现在流行的移动设备上更是显示出强大的威力。

⑵菜单选择、表格填充和对话框

当设计者不能创建适当的直接操纵策略时，菜单选择和表格填充就成为非常有吸引力的选择。

① 菜单选择

合理编制的菜单可以使用户方便地通过指点或按键来选择操作，而不是强迫用户记忆并回忆命令的语法。用户的选择可以立即得到相应和反馈。这对非专业用户来说是非常有效的。

菜单的使用有如下几种主要的方式：

- 单菜单

常见的拥有单个/两个按钮（一般是确定/取消）的对话框、单选按钮和复选按钮都是单菜单的实例。

- 下拉式、弹出式和丝带菜单

在常用的应用软件（如 Word）中我们能看到这些菜单的样式和布局。

- 长列表菜单

长列表菜单的主要表现形式为下拉列表框、组合框、滑动杆、摇杆等。长列表菜单将用户的可能选择全部列举出来，或者限制了用户的选择范围，从而使用户输入出错的概率降到最低，也能减轻了用户的记忆负担。

- 二维菜单

对于那些含有大量选项的菜单，可以选择二维菜单来进行组织。这可以给用户良好的选项概览，同时也减少了所需动作的次数，并允许做出快速选择。

- 嵌入式菜单和热链接

嵌入式菜单将可选择的内容嵌入到正常的文本当中，并用突出的方式显示已引起用户的注意。实际上，超文本中的热链接就是嵌入式菜单的典型应用。合理地使用嵌入式菜单可以在不分散用户注意力的情况下提供用户关注的信息，同时还可以节约屏幕空间。

② 表格填充

当用户的输入项较多时，表格填充显然比菜单选择要合适。

表格设计的元素包括如下主要内容：

- 有意义的标题
- 可理解的说明书
- 数据域的分组和排序
- 具有吸引力的布局
- 一致的术语和缩写
- 方便的光标移动
- 尽可能的错误预防

- 不可接受值得出错信息
- 必选域的标记

有时我们需要用户输入特定格式的数据，例如：日期、IP 地址、email 地址等。在这种情况下，最好将输入域与某种特定的、可以限制格式的输入组件绑定在一起。例如：在输入日期时，可以使用日历组件，让用户在其中选择而不是直接输入。

③ 对话框

对话框常用于对用户操作的确认、信息反馈等场合。弹出的对话框可能会带来一些小麻烦，例如：要遮挡一部分屏幕信息。因此，对话框的设计不应太大，并且要仔细选择它弹出的位置。

设计对话框应把握以下主要原则：

- 有意义的标题，一致的风格和布局
- 标准的按钮（如确定、取消）
- 使用更直接操纵的错误预防
- 平滑地出现和消失
- 清楚地表明如何完成或撤销

2. 其他考虑因素

⑴ 声频的使用

当用户的手眼都很忙碌的时候，或者在为有视力障碍的用户设计界面时，声频的使用显得尤为重要。一个典型的场景就是用户在驾驶时使用车载导航设备。导航设备应当给出清晰准确的声频信息，以引导用户做出正确的选择。

有时用户需要使用语音与系统进行交互。这就需要系统有相应的语音识别能力。

⑵小尺寸屏幕

小尺寸屏幕主要应用的手机这样的移动终端上。由于屏幕尺寸的限制，大多数用于桌面系统屏幕设计变得不实用了。因此，需要为这类设备设计特别的界面。

为小屏幕系统设计界面需要注意如下事项：

① 说明目标域：说明界面为何种操作设置

② 专用设备意味着专用界面

③ 适当分配功能：考虑使用频率和重要性

④ 简化设计：关注主要功能

⑤ 为响应性设计：为中断做计划，提供持续的反馈

需要注意的是：小屏幕并不意味着低分辨率。事实恰恰相反，常用智能手机都拥有非常高的分辨率。这意味着设计者在选择界面元素尺寸、字体大小时，不能沿用做桌面设计的经验。另一个需要考虑的因素是流量问题。这要求设计者不能使用高分辨率或者大尺寸的图片，并且应当有意识地减小文档的大小。

3.5.5 功能和时尚的平衡

当今的用户对界面的要求越来越高。他们在操作软件界面时，希望获得良好的个人体验，例如：便捷人性的操作方式、容易阅读且易懂的信息呈现、时尚大方的界面布局、令人赏心

悦目的颜色搭配等等。因此，设计者必须以最终用户的眼光看待界面设计，而不总是以专家的角度诠释用户需求，需要在界面的功能和时尚间做出精准的平衡。

这里我们就影响平衡的一些主要的因素做出讨论。

1. 出错信息

出错信息是友好界面设计策略中的关键部分，因为它会直接影响到用户的使用体验。糟糕的信息会极大地挫伤用户的积极性，甚至导致弃用系统。为尽可能避免此类事件的发生，设计者应注意如下事项：

① 尽量具体、精准，不要使用会让用户感到疑惑不解的专业术语、缩写等，不过用户的行话可能会有必要。

② 尽可能使用主动语态而不是被动语态；肯定的语气比否定的语气容易让人接受；用一种客气的但不过分的语气。

③ 给出必要的提示或建议。

④ 需要考虑用户的经验甚至文化、地域背景。

⑤ 如果有可能应提供声音提示。

举个例子：当用户在登录界面中输入的信息有误时，良好的反馈信息最好类似于：

您输入的用户信息有误，请检查您输入的账号或者密码是否正确

对比之下，一种不好的信息设计是这样的：

无效的用户 ID 或口令

2. 显示设计

显示设计首先应该了解对信息显示的要求，然后选择适当的显示内容和显示形式。

⑴显示内容选择的原则

显示内容只选择必需的信息；联系紧密的信息应该一起显示；显示的信息应与用户执行的任务有关；每一屏信息的数量，包括标题、工具栏、数据等不超过整个屏幕可显示区域的30%。

⑵安排显示结构的规则

按照某种逻辑结构分组，还可以根据使用频率、操作顺序或者功能来分组。信息安排要方便用户使用，要提供明了的提示帮助信息。关键词、识别符应安排在窗口的左上角。显示的信息应该便于用户理解，尽量少使用代码和缩写等。

⑶信息的表示方法

这里讨论文字信息和数值信息的显示方法。

在显示文字信息时，要选择合适的字体，并且注意字体颜色和背景颜色的对比。比如，Times New Roman 字体会让数字 1、英文小写字母 l 不容易分清楚，因此在显示重要的信息时，一定要注意这些因素。

在显示数值信息时，可以根据这些信息是要定量还是要定性表达来选择显示方式。

- 定量信息要求准确清晰，所以最好采用文字的方式来显示。不过，文字的方式有一些缺陷，比如，不容易引起人的注意，不适合显示高频变化的值等。解决的方法是采

用文字图形相结合的方式。

- 定性信息不要求精确，只是一个量级的概念，比如软件安装过程中的时间耗费。对于这样的信息，模拟（图形）的方式是最好的。

除了在前面讨论过的几种图形化显示控件外，还可以使用进度条（Progress Bar）、图表（Chart）等。

对于海量信息，如天气数据、分子结构数据等，最好的呈现方法就是可视化（Visualization）。可视化技术有二维和三维之分，并且都已经比较成熟，可视化不属于本书讨论内容。

3. 颜色的使用

在几乎所有的交互式系统中，都会用颜色来表达一定的信息。颜色可以增加用户界面的可用度，同时在一定程度上有助于用户理解界面。但是，颜色也很容易被滥用来表达信息。然而遗憾的是，没有一组简单的规则去规定设计者怎样使用颜色。设计者很容易依据自身的审美观点去设计颜色方案。这是可以理解的，但也许不能使更多的用户满意。

Shneiderman 在其论述中给出了界面中颜色使用的 14 条指南，其中最重要的几条是：

① 限制使用的颜色数目并且用一种保守的方式使用它们。一般地，单个显示中不应该出现 4 种以上的不同的颜色，整个显示序列中不应该超过 7 种。界面设计者应该仔细地选择所用的颜色，它们应该能贴切地表达设计者的意图。

② 认识到颜色也是一种编码技术。彩色编码可以加快很多任务的识别程度，但在使用时，颜色编码要一致，整个系统使用相同的颜色编码规则；否则，用户会误读信息从而采取错误的举动。此外，不同文化、教育和职业背景的人对相同的颜色有着不同的解释，这会造成用户对颜色编码的不同解读，从而可能导致错误。

③ 使用颜色的改变来表示系统状态的改变。颜色的改变最好意味着一些比较“重大”的系统事件发生了。这种方式在界面上有大量的图形元素显示的时候特别有效，它会引起用户的警觉，从而采取相应的措施。

④ 考虑颜色的生理作用。高饱和度和高亮度的颜色（特别是在色块面积较大时）以及某些颜色组合会让用户感到不适，因此界面应尽多使用不饱和及低亮度的颜色以及合理的颜色搭配。

⑤ 尽量不要用颜色来表示信息。我们都知道，有大约 10%的人是色盲，他们不能区分有些或全部颜色，如红色和绿色。

3.6 MVC 设计模式

MVC 是“Model-View-Controller”的缩写，意思是“模型-视图-控制器”。这是一种非常流行的设计模型，由 Trygve Reenskaug 提出，首先被应用在 SmallTalk-80 环境中，是许多交互界面系统的构成基础。

3.6.1 MVC 的概念

MVC 设计模式是一种非常有效的支持信息多种表示的方法。用户可以用最适合的方式与每种信息表示交互。在此设计模型中，软件被分成三类不同的部件：

- 模型（Model）：是软件所处理问题的逻辑在独立于外在显示内容和形式情况下的

内在抽象，封装了问题的核心数据、逻辑和功能的计算关系，它独立于具体的界面表达和 I/O 操作。模型可以直接访问数据，例如对数据库的访问。模型不依赖视图和控制器，也就是说，它不关心数据如何显示或是如何操作。模型总是与一些视图相关联，模型中数据的变化会通过一种更新机制反射到这些视图上，以使视图发生改变。

- 视图（View）：视图把表示模型数据及逻辑关系和状态的信息及特定形式展示给用户。它从模型获得显示信息，对于相同的信息可以有多个不同的显示形式或视图。每个模型对象都可以关联一系列不同的视图对象。视图中一般没有程序上的逻辑。
- 控制器（Controller）：每个视图对象拥有一个控制器对象。控制器用来处理用户与软件的交互操作的，其职责是控制模型中任何变化的传播，确保用户界面与模型间的对应联系；它接受用户的输入，将输入反馈给模型，进而实现对模型的计算控制，是使模型和视图协调工作的部件。

图 3.33 示意了 MVC 模式的工作模式。

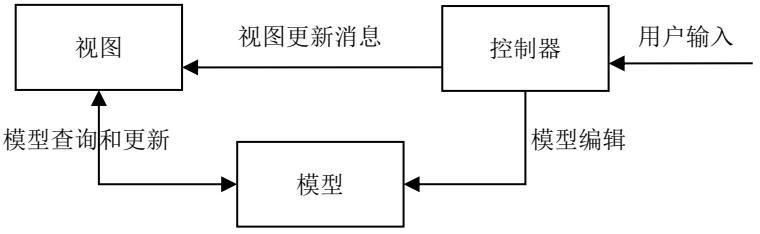


图 3.33　MVC 模式

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，所有其他依赖于这些数据的视图都应反映出这些变化。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。

MVC 设计模式的优点表现在以下几个方面：

① 可以为一个模型在运行时同时建立和使用多个视图。MVC 机制确保所有相关的视图及时得到模型数据变化通知，从而使所有关联的视图和控制器做到行为同步。

② 视图与控制器的可接插性。允许更换视图和控制器对象。而且可以根据需求动态的打开或关闭，甚至在运行期间进行对象替换。

③ 模型的可移植性。因为模型是独立于视图的，所以可以把一个模型独立地移植到新的平台工作。需要做的只是在新平台上对视图和控制器进行新的修改。

④ 潜在的框架结构。可以基于此模型建立应用程序框架，不仅仅是用在界面的设计中。

MVC 的不足体现在以下几个方面：

① MVC 模式的缺点是由于它没有明确的定义，因此完全理解 MVC 模式并不是很容易。使用 MVC 模式需要精心的计划，由于它的内部原理比较复杂，所以需要花费一些时间去思考。

② 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。

③ 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的，但却是联系紧密

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，这样就妨碍了它们的独立重用。

④ 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。

无论如何，对于开发存在大量用户界面，并且逻辑复杂的大型应用程序，MVC 将会使软件在健壮性、代码重用和结构方面上一个新的台阶。尽管在最初构建 MVC 模式框架时会花费一定的工作量，但从长远的角度来看，它会大大提高后期软件开发的效率。

目前，很多的开发语言/平台都提供了 MVC 框架。例如：ASP.NET、Java Swing、Android、Ruby on Rails、Python Django、JavaScriptMVC 等。

是否采用 MVC 设计模式可以简单地用应用的规模来界定：如果应用规模较小，那么采用 MVC 并不是一种好的选择；反之，MVC 会为中大型应用打下坚实的可扩展、可维护的基础。

3.6.2 MVC 的工作流程

MVC 的一般工作流程是：

① 建立系统中的 Model、View、Controller 类；

② 确立三者之间的关系。一般地，View 和 Controller 总是成对的，它们只和一个 Model 相关联；反之，一个 Model 可以关联多个 View-Controller 对；

③ 根据用户动作触发某个 Controller，它处理用户输入，然后启动与之关联的 View 的更新动作；View 为了更新显示，要从与之关联的 Model 中查询数据，然后将这个数据集格式化后展示给用户。

为了能使上述流程更加流畅，在系统实现时一般会使用路由(Route)机制。该机制可以扼要描述为：系统设置一个路由器（Router，这借用了硬件的概念，一般由类来实现），并维护一张路由表，其中写明了 View-Controller 对与 Model 的关联关系；当用户发起动作时，路由器根据用户的选择触发对应的 Controller，随后完成后续的工作。

3.6.3 MVC 与三层架构的区别

在讨论体系结构的内容中，我们曾提到了应用的三个相对独立的逻辑层：

① 顶层：用户界面层（UI）

② 中间层：业务逻辑层（BLL）

③ 底层：数据访问层（DAL）

在实际应用中，很容易让人迷惑的就是 MVC 和三层架构的关系，或者异同。一些设计者会做出这样的简单映射：View→UI，Controller→BLL，Model→DAL。这是一种很自然的思维方式，但却是一种不好的设计。为澄清三层架构和 MVC 的关系，有必要讨论一下二者的异同。

① 二者是在不同层面上的概念。三层架构属于体系结构的范畴，是一个更宏观的概念；而 MVC 是一种设计模式。换句话说，就是一种体系结构在实现时可以采用不同的设计模式；

② 三层架构是一种层次模型；而 MVC 没有划分层次，其体系由三个互相作用的部件构成，三个部件之间的关系可以视为是一种网状模型。此外，MVC 中的 View 和 Controller 总是成对出现的，并且一个 Model 可以关联多个 View 和 Controller 对；

③ 二者要达到的目标同样是为了降低各子系统间的耦合度，但侧重点不同：三层架构侧重于系统业务的宏观划分；MVC 侧重于信息表现时的职责划分；

④ 如果一定要确定二者之间的映射关系，那么可以粗略地认为：

（View，Controller）→UI

Model→（BLL，DAL）

在这个意义上，为了更精确地体现三层架构的意图，在实际应用中，Model 可能还会被划分出多个层次以对应/实现 BLL 和 DAL。

无论如何，设计者应该根据应用的需要去决定是否采用三层架构，或者 MVC 设计模式。

3.7 软件设计实例

【例 3-7】针对于表 3.1 的销售数据，设计一个应用系统，它可以用表格、柱状图、折线图等方式显示结果。

表 3.1 销售量表

	Q1	Q2	Q3	Q4
2012	10	9	15	6
2013	12	10	20	10
2014	16	12	23	11

可以看出，这是一个非常简单的应用系统。但是，我们还是按部就班地做出系统设计。这里，我们采用 OOD 方式进行。

3.7.1 用例模型

本系统的使用环境非常简单：用户使用系统，选择图表类型，系统则根据用户的选择显示相应的图表。图 3.34 是用例图。

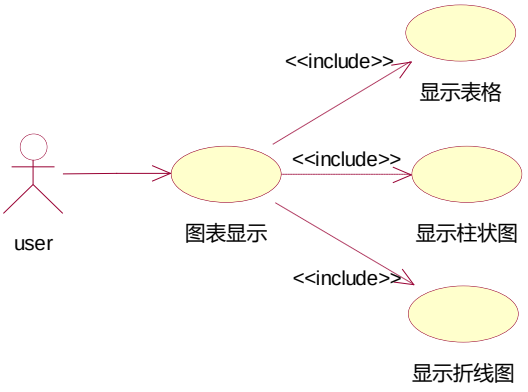


图 3.34 用例图

3.7.2 体系结构设计

1. 体系结构

案例描述的微型应用采用 B/S 模型是非常合理和明智的选择。这里，应用的 B/S 架构应

该是三层的：

- 客户端。客户端使用支持 HTML 5 和 CSS 3 的浏览器。浏览的页面需要编码；
- 服务器端。服务器端使用流行的 Web 服务器。为了能产生动态页面，需要编写一些脚本。这里用 PHP 脚本来响应用户行为和处理业务逻辑并产生显示用数据；页面采用模板形式；使用模板引擎将页面模板和数据绑定，并生成最终页面；
- 数据端。由于应用的数据模型很简单，因此使用 XML 文档保存数据，而不需要真正的数据库管理系统支持。

2. 控制模型

B/S 结构的客户端都采用事件驱动模型。一旦用户界面呈现在用户面前，那么浏览器处于挂起等待状态。当用户点击界面上的做出某个动作（例如点击<a>元素）时，与该动作绑定的事件处理器会将事件分发给对应的子系统去处理。

在这个层面上，我们不需要为这个话题考虑更多的设计问题。

3. 模块分解

虽然案例应用规模是微型的，但我们仍然采用 MVC 设计模式，以展示设计过程。

根据用户需求和 MVC 模式的要求，本系统将分解出三个主要的子系统，分别对应 MVC 模式中的 Model、View 和 Controller：

(1) 模型子系统：完成数据维护功能；

(2) 视图子系统：完成数据查询、页面显示功能；

(3) 控制器子系统：相应用户动作；控制页面更新。

由于本系统的业务逻辑非常简单，因此就不再对 Model 进行进一步的细分。

结构设计的结果如图 3.35。

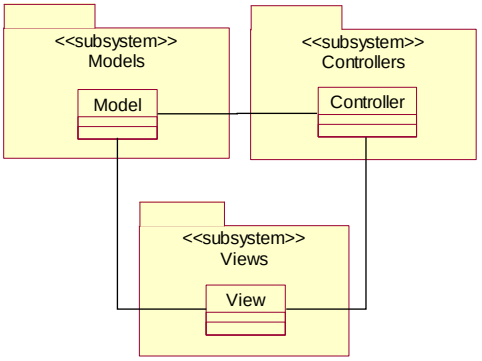


图 3.35 系统的模块/子系统

3.7.3 对象设计

本系统采用面向对象模型，系统的子系统都由对象/类来实现。

为了标识出系统当中的对象，我们可以想象一下系统的应用场景：用户输入网址，打开主页面，根据需求在页面中点击图表类型菜单，选中其中一项；浏览器将选择发送到服务器端，服务器根据选择执行相应的脚本，生成并返回页面。可以看到，所有主要的工作都发生

在服务器端，而服务器端的实际工作流程是：用户动作启动了控制器，控制器向相应视图发送视图更新消息；视图向模型查询数据，然后将数据转换为用于显示的格式，最后将数据与页面模板绑定并生成页面（该过程称为“页面渲染，Page Rendering”），该页面由服务器发送至浏览器。

从上述场景描述中，可以很容易地发现系统中的主要对象：模型、视图、控制器。一个不明显的对象存在于页面渲染的过程中，该对象负责页面渲染。我们不妨称之为渲染器。

下面我们就来对这几种类进行详细设计。

1. 顶层类

根据对象的设计原则，对象应该建立在抽象之上，这可以更好地实现软件复用。为此，我们首先根据对象间的关系，为系统设计 4 个顶层类：模型类、视图类、控制器类、渲染器类，它们都工作在抽象层面上。图 3.36 示意了这四个顶层类的关系和接口。

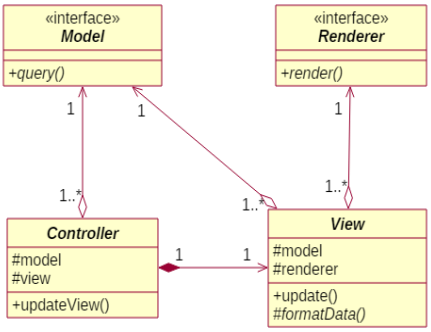


图 3.36 系统中的顶层对象/类

其中，Controller 和 View 两个类因为包含数据成员和实方法，所以不是接口类；但又因包含抽象方法，所以是抽象(Abstract)类。

以下是对上述对象/类的接口做出的一些说明。

(1) Model 类

Model 类的所有方法都是抽象的，其接口说明如下（类 Java 语法）。

```
interface Model
{
    public query(queryCriteria);
}
```

- 方法 query()用于供 View 查询数据并返回数据集；
- 由于本例很交单，因此模型的更新、编辑方法没有给出；
- 如果应用需要对业务逻辑做处理，那么可以在 Model 类中增加相应的成员。本例中没有必要这么做。

(2) View 类

```
abstract class View
{
    protected model;
```

```
    protected renderer;
    public update();
    abstract protected formatData(dataset);
}
```

- 属性 model 是对模型对象的引用；
- 属性 renderer 是对渲染器对象的引用；
- 方法 update()实现视图的更新，它的内部流程是：从模型查询数据→转换数据格式→交给渲染器生成页面。因为所有的视图都是这个节奏，所以该方法不是抽象的，并且是最终方法，其子类不需要覆盖它；
- 抽象方法 formatData()用于将从模型中查询的抽象数据集转换为用于输出的数据集。该方法需要在最终子类中实现。

(3) Controller 类

```
abstract class Controller
{
    protected model;
    protected view;
    public updateView();
}
```

- 属性 model 是对模型对象的引用；
- 属性 view 是对视图对象的引用；
- 方法 updateView()向引用的视图对象发出更新消息，其内部流程非常简单，就是调用视图对象的 update()方法，因此该方法不是抽象的，并且是最终版本；
- 将 Controller 类说明成是抽象的，目的是强制它成为基类，必须被继承。

(4) Renderer 类

```
interface Renderer
{
    public render(dataset);
}
```

- 抽象方法 render()用于渲染页面，使用的数据集是格式化后的。为了能实现这个功能，该类应该和实际使用的模板引擎发生关联。

2. 子类

上述顶层类工作在抽象层面，因此不能完成实际的工作。所以，我们必须从这些类中派生出与实际操作紧密结合的子类来。

基于同样的抽象原则，我们将子类设计为两层：上层描述所有图表类的共性；下层描述具体图表的特性。在下面的内容中，只对类的接口进行扼要说明，不再给出接口的示意代码。

(1) 模型类

Model_chart 类是 Model 类的子类，它专用于描述用于图表操作的数据。图 3.37 是模型类

族的设计图。

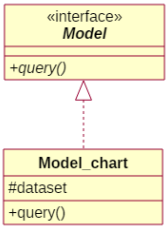


图 3.37 模型类族

子类 Model_chart 实现了父类的抽象方法 query()。该类在它的构造方法中从数据源获取原始数据并保存到 dataset 中，query()返回的就是这个数据集。

这里给出关于子类类图的一些说明：如果子类中出现与父类同名的方法，表明该子类方法实现或覆盖了父类的同名方法；如果子类暂时不需实现父类方法，或者直接继承而无需修改，那么该父类方法在子类中就没有出现。此后的类图采用这个相同的规则；

(2) 视图类和渲染器类

① 中间层次 View_chart 类，它是 View 类的子类，是所有图表类型视图类的公共父类。因为该类要涉及到实际的页面渲染，所以要包含如下属性或方法：

- 属性 colors 是渲染参数，其中主要包含页面的配色方案，以供不同的视图选择颜色。

此类未实现 formatData()方法，此方法推迟到最终子类中实现。

② 底层类 View_chart_table/line/bar，它们都是 View_chart 类的子类。这些类都是最终子类，实现了继承链中所有未实现的方法。

③ 渲染器类 Renderer_chart 实现了 Renderer 接口类，同时还继承自选定的模板引擎类 Smarty。Smarty 的介绍留在第 5 章软件实现案例中再做讨论。

以上设计符合 DIP 原则。图 3.38 是视图类族的设计方案。

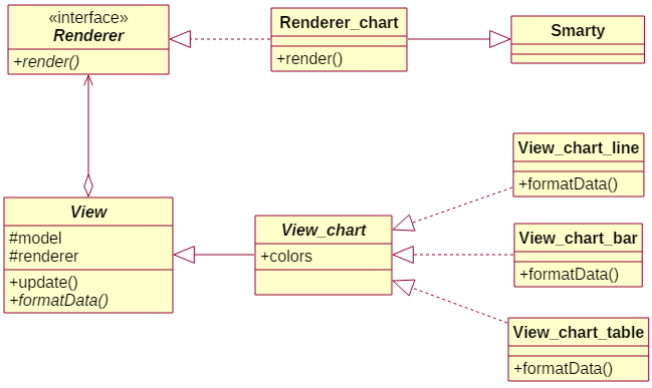


图 3.38 视图类族

(3) 控制器类

① 中间层次类 Controller_chart，它是 Controller 类的子类，是所有图表类型控制器的父类。这个类在它的构造方法中完成了 Model-View-Controller 三者关系的绑定；

② 底层类 Controller_chart_table/bar/line，它们都是 Controller_chart 的子类。这些类都是

最终子类。

图 3.39 示意了控制器类的设计。

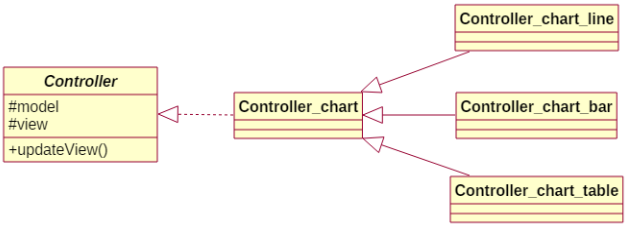


图 3.39 控制器类族

3. 顺序图

根据操作流程，上述各类的工作流程如图 3.40 所示。

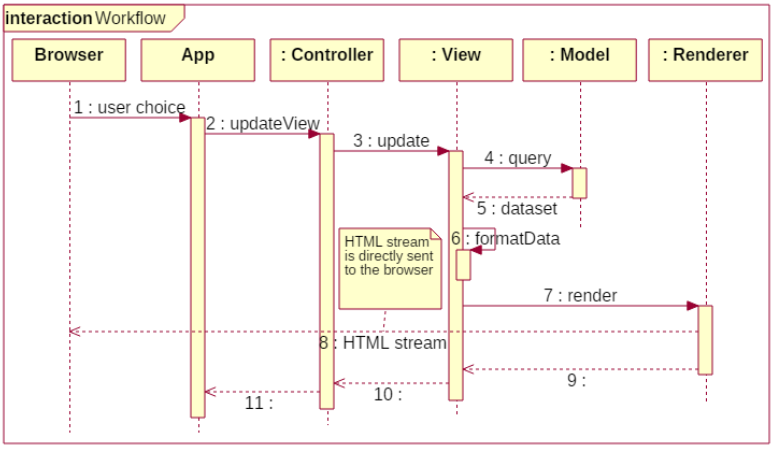


图 3.40 顺序图

3.7.4 用户界面设计

1. 交互风格

用户与页面的交互风格选择菜单选择模式：在页面上呈现三个用于选择不同视图的菜单选项。用户选择（点击）其中一个选项后重新载入口脚本以刷新页面，显示不同的视图。

2. 页面布局

三个视图的页面布局采用相同的格局，如图 3.41 所示。

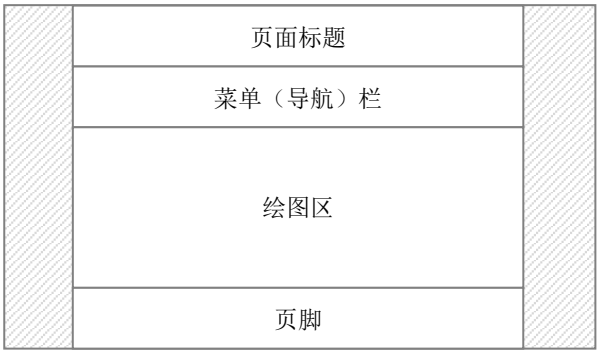


图 3.41 页面布局

- 版芯宽 54em，高度自适应；
- 标题区高 2.5em，宽度 100%；
- 导航栏高 3em，宽度 54em；
- 绘图区高度根据图标内容自适应，宽度 100%；
- 页脚区高 2.5em，宽度 100%。

注：em 是一种自适应的、相对的尺寸单位。这种单位的最大的两个特点是：一是基本单位 1em 依据浏览器的设定来确定其与多少像素对应；二是页面子元素的 em 尺寸会根据父元素的进行自动调整。

3. 风格、字体和配色方案

① 风格

页面风格采用流行的扁平化设计风格，采用大按钮、大图形的方式以便用户动作和观察效果。

② 字体

- 字型：微软雅黑
- 字号：标题 2em；菜单 1.2em；正文 1em；页脚 0.9em。

③ 配色方案

- 主色：选用青色系的三色配色方案，用于菜单底色、表格等主色；
  - 颜色 1：rgb(0, 106, 115)
  - 颜色 2：rgb(0, 139, 156)
  - 颜色 3：rgb(49, 165, 185)
  - 图表颜色填充透明度（alpha 值）：0.7
- 白色：用于背景、菜单文字色、标题和页脚文字色；
- 灰色：有几个等级，用于图表文字、标题和页脚背景色。

至此，本系统设计阶段的任务基本告一段落，而其他的具体细节这里就不做更多的讨论了。

在第五章中，我们将讨论这个系统的具体实现。

小　结

软件设计阶段决定了软件的风格、外观和功能。所以，软件设计是软件开发阶段的一个非常重要的环节，设计的好坏直接决定了开发的周期和产品的质量。

本章介绍了软件设计的基本任务和设计原则，希望读者能主动运用这些原则，指导自己的设计。当然，原则并不是一成不变的死教条，需要具体问题具体分析，在实际中灵活运用。同时还讨论了软件设计中常用的软件体系结构及其特点，包括仓库模型、分布式结构、两层及多层 C/S 模型、云计算模型等。还讨论了面向对象设计（OOD）的基本任务和设计准则。对直接影响软件生命期和提高对用户的吸引力的用户界面设计也作了较详细的讨论。最后，用一个案例来示范如何做出系统设计。

习题三

一、选择题

1. 模块的基本特征是()。
   A) 外部特征(输入/输出、功能)　　B) 内部特征(输入/输出、功能)
   C) 内部特征(局部数据、代码)　　D) 外部特征(局部数据、代码)
2. SD 方法的设计总则是()。
   A) 程序简洁、操作方便　　B) 结构清晰、合理
   C) 模块内聚性强　　D) 模块之间耦合度低
3. 软件设计的主要任务是()。
   A) 完成模块的编码和测试
   B) 完成系统的数据结构和程序结构设计
   C) 完成系统的体系结构设计和用户界面设计。
   D) 对模块内部的过程进行设计
4. 设计阶段应达到的目标有()。
   A) 提高可靠性和可维护性　　B) 提高应用范围
   C) 结构清晰　　D) 提高可理解性和效率
5. 从工程管理的角度来看，软件设计分两步完成()。
   A) ①系统分析②模块设计　　B) ①详细设计②总体设计
   C) ①模块设计②详细设计　　D) ①总体设计②详细设计
6. 模块独立性准则由以下定性指标来衡量()。
   A) 分解度　　B) 耦合度　C) 屏蔽性　　D) 内聚性
7. 用户界面设计的任务包括()。
   A) 确定用户界面类型　　B) 建立任务模型
   C) 建立用户模型　　D) 建立功能模型

二、判断题

1. 划分模块可以降低软件的复杂度和工作量，所以应该将模块分得越小越好。()
2. 在网状结构中任何两个模块都是平等的，没有从属关系，所以在软件开发过程中常常被使用。()

3. 中心变换型的 DFD 图可看成是对输入数据进行转换而得到输出数据的处理，因此可以使用事务分析技术得到初始的模块结构图。()
4. 信息隐蔽原则有利于提高模块的内聚性。()
5. SD 法是一种面向数据结构的设计方法，强调程序结构与问题结构相对应。()
6. 当模块的控制范围是其作用范围的子集时，模块之间的耦合度较低。()
7. 面向对象的设计的主要目标是提高生产效率，提高质量和可维护性。()

三、简答题

1. 请解释为什么需要体系结构设计。
2. 集中式模型和分布式模型相比各有什么优缺点？
3. 请举出一种集中式模型的实例，并图示它的结构。
4. 胖客户模型和瘦客户模型的区别是什么？它们分别被应用在什么样的场合？
5. 请举出一种分布式模型的实例，并图示它的结构。
6. 请为一个公司的电子商务网站建设提出体系结构设计方案。
7. B/S 模型与 C/S 模型有什么异同？
8. 模块分解的最终目的是什么？
9. 模块分解应该遵循什么样的标准？
10. 面向对象设计的准则有哪些？
11. 请用你熟悉的程序设计语言编写一个数据结构，来模拟队列 Queue。队列是一种先进先出的数据存储结构，它一般拥有下列操作：数据入列、数据出列、查找、清空等。
12. 编写非面向对象和面向对象的两种不同版本的队列，然后对比它们的特点。
13. 请根据对象设计的 7 条原则，分析例 3-5 给出的类违背了哪些原则？如何改进？
14. 观察你所使用的操作系统和应用程序的界面，从中找出信息表示的方法来。
15. 在你找出的信息表示中，哪些信息表示方法是可以改进的？
16. 请查阅相关的资料，写一篇关于颜色对人的生理作用的文章。
17. 前面提到的电子商务公司的标准色为深绿色，请为该公司的电子商务网站设计提供几套备选的配色方案。
18. 请采用 MVC 模式，为某个银行的 ATM 系统设计用户界面。