CHAPTER 5

第5章 软件架构与架构建模技术

软件架构是一个系统的草图，它描述的对象是直接构成系统的抽象组件。各个组件之间的连接则明确和相对细致地描述组件之间的通信。在实现阶段，这些抽象组件被细化为实际的组件，例如具体某个类或者对象。在面向对象领域中，组件之间的连接通常用接口来实现。

软件体系结构是构建计算机软件实践的基础。与建筑师设定建筑项目的设计原则和目标作为绘图员画图的基础一样，一个软件架构师或者系统架构师陈述软件构架以作为满足不同客户需求的实际系统设计方案的基础。

软件架构已经在软件工程领域中有着广泛的应用，许多专家学者从不同角度和不同侧面对软件架构进行了刻画。软件架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、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。软件架构不仅指定了系统的组织结构和拓扑结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。

5.1 软件架构概况

5.1.1 软件架构的发展史

软件系统的规模在迅速增大的同时，软件开发方法也经历了一系列的变革。在此过程中，软件架构也由最初模糊的概念发展成为一个渐趋成熟的技术。

20世纪70年代以前，尤其是在以ALGOL 60为代表的高级语言出现以前，软件开发基本上都是汇编程序设计。此阶段系统规模较小，很少明确考虑系统结构，一般不存在系统建模工作。20世纪70年代中后期，由于结构化开发方法的出现与广泛应用，软件开发中出现了概要设计与详细设计，而且主要任务是数据流设计与控制流设计。因此，此时软件结构已作为一个明确的概念出现在系统的开发中。

20世纪80年代初到90年代中期,是面向对象开发方法的兴起与成熟阶段。由于对象是对数据与基于数据之上操作的封装,在面向对象开发方法下,数据流设计与控制流设计统一为对象建模。同时,面向对象方法还提出了一些其他的结构视图。如在OMT方法中提出了功能视图、对象视图与动态视图(包括状态图和事件追踪图);而BOOCH方法中则提出了类视图、对象视图、状态迁移图、交互作用图、模块图和进程图;在1997年出现的统一建模语言UML则从功能模型(用例视图)、静态模型(包括类图、对象图、构件图、包图)、动态模型(协作图、顺序图、状态图和活动图)和配置模型(配置图)描述应用系统的结构。

20世纪90年代以后则是基于构件的软件开发阶段,该阶段以过程为中心,强调软件开发采用构件化技术和体系结构技术,要求开发出的软件具备很强的自适应性、互操作性、可扩展性和可重用性。此阶段中,软件架构已经作为一个明确的文档和中间产品存在于软件开发过程中,同时,软件架构作为一门学科逐渐得到人们的重视,并成为软件工程领域的研究热点,因而Perry和Wolf认为,"未来的年代将是研究软件架构的时代!"。

纵观软件架构技术的发展过程,从最初的"无结构"设计到现行的基于体系结构软件开发,可以认为经历了4个阶段:①"无体系结构"设计阶段,以汇编语言进行小规模应用程序开发为特征;②萌芽阶段,出现了程序结构设计主题,以控制流图和数据流图构成软件结构为特征;③初级阶段,出现了从不同侧面描述系统的结构模型,以UML为典型代表;④高级阶段,以描述系统的高层抽象结构为中心,不关心具体的建模细节,划分了体系结构模型与传统的软件结构的界限,该阶段以Kruchten提出的"4+1"模型为标志。由于概念尚不统一,描述规范也不能达成一致认识,在软件开发实践中软件架构尚不能发挥重要作用,因此,软件架构技术达到成熟还需一段时日。

5.1.2 软件架构的定义

虽然软件架构已经在软件工程领域中有着广泛的应用,但迄今为止还没有一个被大家所公认的定义。许多专家学者从不同角度和不同侧面对软件架构进行了刻画,较为典型的定义有以下几个。

(1) Dewayne Perry和Alex Wolf曾这样定义:软件架构是具有一定形式的结构化元素,即构件的集合,包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工,数据构件是被加工的信息,连接构件把体系结构的不同部分组合连接起来。这一定义注重区分处理构件、数据构件和连接构件,这一方法在其他的定义和方法中基本上得到保持。

(2) Mary Shaw和David Garlan认为软件架构是软件设计过程中的一个层次,这一层次超越计算过程中的算法设计和数据结构设计。体系结构问题包括总体组织和全局控制、通信协议、同步、数据存取,给设计元素分配特定功能,设计元素的组织、规模和性能,在各设计方案间进行选择等。软件架构处理算法与数据结构层次之上关于整体系统结构设计和描述方面的一些问题,如全局组织和全局控制结构、关于通信、同步与数据存取的协议,设计构件功能定义,物理分布与合成,设计方案的选择、评估与实现等。

(3) Kruchten指出,软件架构有4个角度,它们从不同方面对系统进行描述:概念角度描述系统的主要构件及它们之间的关系;模块角度包含功能分解与层次结构;运行角度描述了一个系统的动态结构;代码角度描述了各种代码和库函数在开发环境中的组织。

(4) Hayes Roth则认为软件架构是一个抽象的系统规范,主要包括用其行为来描述的

功能构件和构件之间的相互连接、接口和关系。

(5) David Garlan 和 Dewne Perry 于 1995 年在 IEEE 软件工程学报上又采用如下的定义：软件架构是一个程序/系统各构件的结构、相互关系以及进行设计的原则和随时间进化的指导方针。

(6) Barry Boehm 和他的学生提出，一个软件架构包括一个软件和系统构件，互联及约束的集合；一个系统需求说明的集合；一个基本原理用以说明这一构件，互联和约束能够满足系统需求。

(7) 1997 年，Bass、Ctements 和 Kazman 在《使用软件架构》一书中给出如下的定义：一个程序或计算机系统的软件架构包括一个或一组软件构件、软件构件的外部的可见特性及其相互关系。其中，“软件外部的可见特性”是指软件构件提供的服务、性能、特性、错误处理以及共享资源使用等。

总之，软件架构的研究正在发展，软件架构的定义也必然随之完善。在以后的文章里，如果不特别指出，我们将使用软件架构的下列定义：

软件架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、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。软件架构不仅指定了系统的组织结构和拓扑结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。

5.2　客户机/服务器模式

当一台连入网络的计算机向其他计算机提供各种网络服务(如数据、文件的共享等)时，它就被叫做服务器。而那些用于访问服务器资料的计算机则被叫做客户机。客户机和服务器都是独立的计算机。采用客户机/服务器(Client/Server，C/S)结构的系统，有一台或多台服务器以及大量的客户机。服务器配备大容量存储器并安装数据库系统，用于数据的存放和数据检索；客户端安装专用的软件，负责数据的输入、运算和输出。严格说来，客户机/服务器模型并不是从物理分布的角度来定义的，它所体现的是一种网络数据访问的实现方式。采用这种结构的系统目前应用非常广泛。

5.2.1　传统两层客户机/服务器模式

1. 两层客户机/服务器模式的基本结构

客户机/服务器系统有三个主要部件：数据库服务器、客户应用程序和网络，如图 5.1 所示。

(1) 服务器负责有效地管理系统的资源，其任务集中于：

- 数据库安全性的要求；
- 数据库访问并发性的控制；
- 数据库前端的客户应用程序的全局数据完整性规则；
- 数据库的备份与恢复。

(2) 客户端应用程序的主要任务是：

- 提供用户与数据库交互的界面；

- 向数据库服务器提交用户请求并接收来自数据库服务器的信息;
- 利用客户应用程序对存在于客户端的数据执行应用逻辑要求。

(3) 网络通信软件的主要作用是:完成数据库服务器和客户应用程序之间的数据传输。

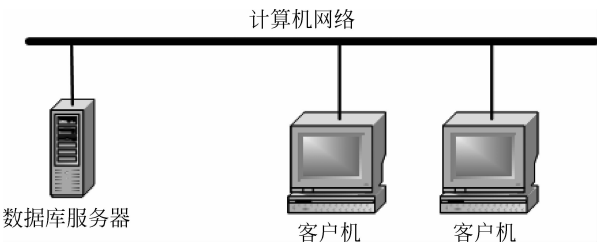

图 5.1 两层客户机/服务器模式的基本结构

上述结构是客户机/服务器模式的基本结构,随后出现的多层客户机/服务器模式以及浏览器/服务器模式都是以上述结构为基础,对客户机层、服务器层按功能进行了进一步细分后产生的模式。

客户机/服务器系统比文件服务器系统能提供更高的性能,因为客户端和服务器端将应用的处理要求分开,同时又共同实现其处理要求,对客户端程序的请求实现"分布式应用处理"。服务器为多个客户端应用程序管理数据,而客户端程序发送、请求和分析从服务器接收的数据,这是一种"胖客户机(Fat Client)","瘦服务器(Thin Server)"的网络计算模式。

2. 两层客户机/服务器模式的优缺点

在一个客户机/服务器应用中,客户端应用程序是针对一个小的、特定的数据集,如一个表的行来进行操作的,而不是像文件服务器那样针对整个文件进行的。它对某一条记录进行封锁,而不是对整个文件进行封锁,因此保证了系统的并发性,并使网络上传输的数据量减到最少,从而改善了系统的性能。

两层客户机/服务器模式软件架构的优点主要在于:

- 系统的客户端应用程序和服务器部件分别运行在不同的计算机上,系统中每台服务器都可以适应各部件的要求,这对于硬件和软件的变化显示出极大的适应性和灵活性,而且易于对系统进行扩充和维护;
- 在客户机/服务器模型中,系统中的功能部件充分隔离,客户端程序的开发集中于数据的显示和分析,而数据库服务器的开发则集中于数据的管理,不必在每一个新的应用开发中都要对一个数据库进行编码;
- 将大的应用处理任务分布到许多通过网络连接的低成本计算机上,使系统部署费用极大降低;
- 两层模式的客户机/服务器模式和多层模式相比,开发、部署和维护成本较低。

随着 C/S 结构应用范围的不断扩大和计算机网络技术的发展,这种结构带来的问题日益明显,主要表现在以下几方面:

- 系统的可靠性有所降低。一个客户机/服务器系统是由各自独立开发、制造和管理的各种硬件和软件组成的混合体,其内在的可靠性不如单一的、中央管理的大型机或小型机,出现问题时,很难立即获得技术支持和帮助;

- 维护费用较高。尽管这种应用模式在某种程度上提高了生产效率，但由于客户端需要安装庞大而复杂的应用程序，当网络用户的规模达到一定的数量之后，系统的维护量急剧增加，因而维护应用系统变得十分困难；
- 系统资源的浪费。随着客户端的规模越来越大，对客户机资源的要求也越来越高。客户机硬件要适应系统要求而不断更新，每个客户机都要重复购置、安装大量应用软件，这无疑是一种巨大的浪费；
- 系统缺乏灵活性。客户机/服务器需要对每一应用独立地开发应用程序，消耗了大量的资源；
- 二层 C/S 结构是单一服务器且以局域网为中心的，所以难以扩展至大型企业广域网或 Internet；
- 数据安全性不好。因为客户端程序可以直接访问数据库服务器，那么，安装在客户端计算机上的其他程序也可以访问数据库服务器，从而使数据库的安全性受到威胁。

3. 模式应用

以一个远程会诊系统为例。该系统设计的目的在于积聚各地专家的集体智慧就某一个医学难题进行协同会诊。系统硬件包括若干台客户机和一台服务器以及连接它们的网络环境，服务器上安装操作系统提供服务，数据库软件管理和存储数据，开发客户端管理软件并部署到每个管理人员使用的客户机上。服务器和客户机软件要采用一些安全控制策略和加密手段以保证数据的安全。

远程会诊系统的工作原理如图 5.2 所示。

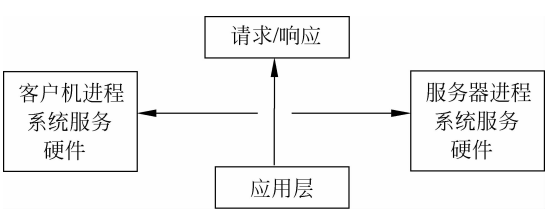

图 5.2　系统工作原理示意图

在这个系统中，客户机进程是主动的，先发出请求给服务器。客户机在应用层负责维持和处理与用户的全部会话，一般包含以下内容：屏幕处理、菜单或命令解释、数据输入和证实、帮助处理和错误恢复。在 GUI 应用中，还包括：窗口处理、鼠标输入、对话框控制、声音和影像管理。通过管理与用户的所有交互作用，使得服务器和网络对用户透明。服务器进程通常一直在运行，给许多客户机提供服务。

5.2.2　经典三层客户机/服务器模式

应用程序从结构上一般分为 4 层：形式逻辑、业务逻辑、数据逻辑和数据存储。传统的 C/S 计算多是基于两级模式，在这种模式中，所有的形式逻辑和业务逻辑均驻留在客户机端，而服务器则成为数据库服务器，负责各种数据的处理和维护。因此 Server 变得很“瘦”，被称为“瘦服务器(Thin Server)”。与之相反，这种模式需要在客户端运行庞大的应用程序，这就是所谓的“胖客户机(Fat Client)”。

在向广域网(如 Internet)扩充的过程中,由于信息量的迅速增大,专用的客户端已经无法满足多功能的需求。网络计算模式从两层模式扩展到 N 层模式,并且结合动态计算,解决了这一问题。

1. 三层客户机/服务器模式基本结构

三层 C/S 结构是将应用功能分成表示层、功能层和数据层三部分,如图 5.3 所示。

表示层是应用的用户接口部分,它担负着用户与应用间的对话功能。它用于检查用户从键盘等输入的数据,显示应用输出的数据。为使用户能直观地进行操作,一般要使用图形用户接口(GUI),使操作简单、易学易用。在变更用户接口时,只需改写显示控制和数据检查程序,而不影响其他两层。检查的内容也只限于数据的形式和值的范围,不包括有关业务本身的处理逻辑。

功能层相当于应用的本体,它将具体的业务处理逻辑地编入程序中。表示层和功能层之间的数据交互要尽可能简洁。

数据层就是 DBMS,负责管理对数据库数据的读写。DBMS 必须能迅速执行大量数据的更新和检索。目前的主流是采用关系数据库管理系统(RDBMS),因此一般从功能层传送到数据层的要求大都使用 SQL 语言。

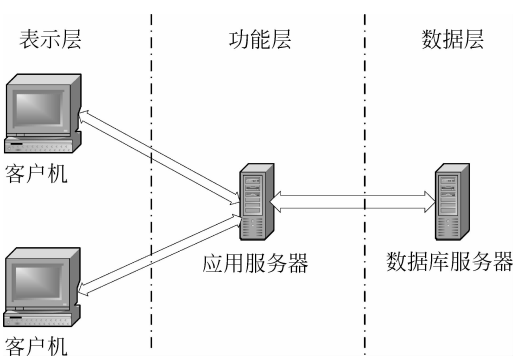

图 5.3　三层客户机/服务器模式的基本结构

这三个逻辑层在硬件的部署上也有两种方式,一是将表示层和功能层都部署在客户机上,与二层 C/S 结构相比,系统的逻辑层次更清晰,其程序的可维护性要好得多。但是其他问题并未得到解决:客户机的负荷太重,业务处理所需的数据要从服务器传给客户机,所以系统的性能容易变坏。

另一种方式是将功能层和数据层分别放在不同的服务器中,这样服务器和服务器之间也要进行数据传送。但是,由于在这种形态中三层是分别放在各自不同的硬件系统上的,所以灵活性很高,能够适应客户机数目的增加和处理负荷的变动。例如,在追加新业务处理时,可以相应增加装载功能层的服务器。因此,系统规模越大这种形态的优点就越显著。所以标准的三层 C/S 结构都采用这种部署方式。

值得注意的是:三层 C/S 结构各层间的通信效率若不高,即使分配给各层的硬件能力很强,其作为整体来说也达不到所要求的性能。此外,设计时必须慎重考虑三层间的通信方法、通信频度及数据量。这和提高各层的独立性一样是三层 C/S 结构的关键问题。

在三层或 N 层 C/S 结构中,中间件(Middleware)是最重要的部件。所谓中间件是一个

用 API 定义的软件层，是具有强大通信能力和良好可扩展性的分布式软件管理框架。它的功能是在客户机和服务器或者服务器和服务器之间传送数据，实现客户机群和服务器群之间的通信。其工作流程是：在客户机里的应用程序需要请求网络上某个服务器的数据或服务时，请求此数据的 C/S 应用程序需访问中间件系统；中间件系统将查找数据源或服务，并在发送应用程序请求后重新打包响应，将其传送回应用程序。随着网络计算模式的发展，中间件日益成为软件领域的新热点。中间件在整个分布式系统中起数据总线的作用，各种异构系统通过中间件有机地结合成一个整体。每个 C/S 环境，从最小的 LAN 环境到超级网络环境，都使用某种形式的中间件。无论客户机何时给服务器发送请求，也无论它何时存取数据库文件，都有某种形式的中间件传递 C/S 链路，用以消除通信协议、数据库查询语言、应用逻辑与操作系统之间潜在的不兼容问题。

2. 三层 C/S 模式的优缺点

和两层 C/S 结构相比，三层客户机/服务器模式具有以下优点：

- 三层 C/S 结构具有更灵活的硬件系统，对于各个层可以选择与其处理负荷和处理特性相适应的硬件；
- 合理地分割三层结构并使其独立，可以使系统的结构变得简单清晰，这样就提高了程序的可维护性；
- 三层 C/S 结构中，应用的各层可以并行开发，各层也可以选择各自最适合的开发语言，有利于变更和维护应用技术规范，按层分割功能使各个程序的处理逻辑变得十分简单；
- 允许充分利用功能层有效地隔离开表示层与数据层，未授权的用户难以绕过功能层而利用数据库工具或黑客手段非法地访问数据层，这就为严格的安全管理奠定了坚实的基础，整个系统的管理层次也更加合理和可控制；
- 系统可用性高，具有良好的开放性，可跨平台操作，支持异构数据库。

三层 C/S 模式当然也存在一些不足，例如相比两层 C/S 模式来讲，开发的技术难度加大，对于小企业和小型应用项目来说，部署成本过高等。

3. 模式应用

三层 C/S 模式应用较广泛，各行业的应用实例也不胜枚举，此处以学校学生信息管理系统为例。

为了克服单纯的 B /S 和 C /S 两种模式的缺点，并充分利用校园内部的局域网，我们对学生信息管理系统进行了升级改造，增加了两台应用服务器，其中一台处理学生信息查询业务，另一台处理学生信息修改业务。这两台应用服务器软件使用微软 COM＋组件开发，对客户机软件也进行了修改。这样改造的结果是：业务处理逻辑分配到了两个服务器节点上，系统性能得到提升，支持的终端数可以成倍增加；查询和修改两个业务逻辑分离，更有利于系统安全控制；客户机软件只处理表示逻辑，系统开发工作量大大减少；客户机软件和应用服务器软件可由两个小组分别开发和维护，提高了效率。

学生信息管理系统的结构如图 5.4 所示，在功能层加入两台服务器后，客户端提出的查询请求不再直接提交给后台数据库，而是通过功能层提供的高速数据通道传送到数据库，这种高速数据通道有效地降低了客户机与服务器以及客户机与数据库的连接数量。同时，查

询过程中与数据库无关的逻辑处理任务也由功能层完成,从而进一步分担了很多原来需要数据库完成的工作,在很大程度上提高了数据库在处理大量并发服务请求时的性能,保证整个系统处于稳定的工作状态。

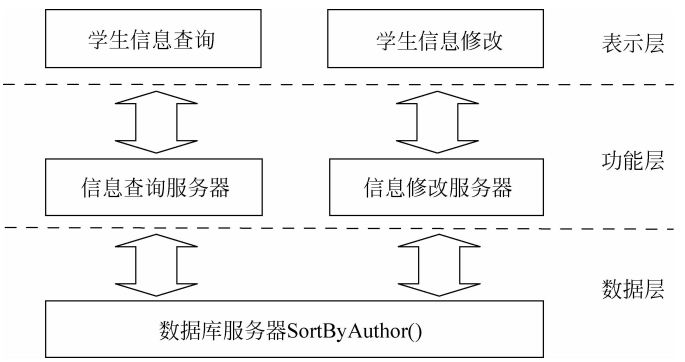

图 5.4　三层 C/S 结构的学生信息管理系统结构图

4. 多层客户机/服务器模式

在大型系统中,为了进一步提高系统性能,增加系统的灵活性,还可以对三层 C/S 结构中的功能层继续进行细分。例如将数据库存取的相关操作独立出来形成数据库访问层,该层专门负责和数据库交互,存取数据。这样就形成了四层乃至 N 层客户机/服务器模式。这样做的优点在于,对于大型系统来说,逻辑结构更加清晰,系统各层的开发和维护、部署更加灵活,但也带来了管理成本加大的问题。在实际应用中,系统架构师应根据项目的具体情况选择合适的架构模式。

5.3　浏览器/服务器模式

1. 浏览器/服务器模式的基本结构

5.2.2 节介绍过,三层 C/S 架构分为表示层、功能层、数据层。表示层负责处理用户的输入和向客户的输出。功能层负责建立数据库的连接,根据用户的请求生成访问数据库的 SQL 语句,并把结果以适当的形式返回给表示层。数据层负责数据的存储和检索,响应功能层的数据处理请求,并将结果返回给功能层。

浏览器/服务器(Browser/Server,B/S)架构模式实际上是上述三层 C/S 架构的一种实现方式,其具体结构为:浏览器/Web 服务器/数据库服务器。采用 B/S 结构的应用系统的基本框架如图 5.5 所示。

B/S 结构,主要是利用了不断成熟的 WWW 浏览器技术,结合浏览器的多种脚本语言(VBScript、JavaScript 等)和 ActiveX 技术,用通用浏览器就实现了原来需要复杂的专用软件才能实现的强大功能,并节约了开发成本,是一种全新的软件系统构造技术。随着 Windows 将浏览器技术植入操作系统内部,这种结构更成为当今应用软件的首选体系结构。显然 B/S 结构应用程序相对于传统的 C/S 结构应用程序是一个巨大的进步。

在 B/S 架构系统中,用户通过浏览器向分布在网络上的服务器发出请求,服务器对浏览器的请求进行处理,将用户所需信息返回到浏览器。而其余如数据请求、加工、结果返回

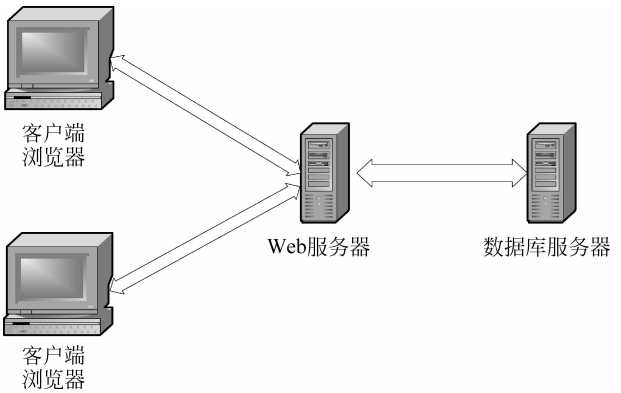

图 5.5　浏览器/服务器架构模式结构图

以及动态网页生成、对数据库的访问和应用程序的执行等工作全部由 Web 服务器完成。

2. 主要优缺点

B/S 结构的主要优点是分布性强、维护方便、开发简单且共享性强、总体拥有成本低。它提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的开放性基础。用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，真正达到了“零客户端”的功能，使系统很容易在运行时自动升级。

其主要缺点在于：

- 存在数据安全性问题，对服务器要求过高，数据传输速度慢，软件的个性化特点明显降低；
- 难以实现传统模式下的特殊功能要求，例如通过浏览器进行大量的数据输入或进行报表的应答、专用性打印输出都比较困难和不便；
- 实现复杂的应用构造有较大的困难，虽然可以用 ActiveX、Java 等技术开发较为复杂的应用，但是相对于已经非常成熟的 C/S 一系列应用工具来说，这些技术还不够成熟。

3. 模式应用

还以学校学生信息管理系统为实例说明这种架构模式。学校准备向全校学生开放部分信息，在校学生可以在校园网上用自己的计算机查询自己的诸如考试成绩、图书借阅记录等信息，学校难道要让每个学生都下载安装一个客户机信息软件吗？这种情况下，B/S 模式就显示出它的巨大优势，改用这种模式是一种必然的选择。

新系统采用 B/S 架构，结合了 ASP 技术，并将组件技术 COM＋和 ActiveX 技术分别应用在服务器端和客户端。该系统的实现主要分为三个部分：ASP 页面、COM＋组件和数据库，因此它是一个三层结构。表示层由 ASP 页面组成，用以实现 WEB 页面显示和调用 COM＋组件，业务逻辑和数据访问由一组用 VC 实现的 COM＋组件构成。为了便于维护、升级和实现分布式应用，在实现过程中，又将业务逻辑层和数据访问层分离开，ASP 页面不直接调用数据访问层，而是通过业务逻辑层调用数据库。一些需要用 Web 处理的、满足大多数访问者请求的功能界面采用 B/S 结构，例如任课教师可以通过浏览器查询所教班级学生的各种相关信息；学校管理人员通过浏览器对学校的学生、教师等信息进行管理与维护

以及查询统计；领导可通过浏览器进行数据的查询和决策等。

5.4 MVC 架构模式

MVC 全名是“模型-视图-控制器(Model-View-Controller)”,它是软件工程中非常经典的一种软件架构模式,在 UI 框架和 UI 设计思路中扮演着非常重要的角色。MVC 是一种软件设计典范,用一种业务逻辑和数据显式分离的方法组织代码,将业务逻辑聚集到一个部件里面,在界面和用户围绕数据的交互能被改进和个性化定制的同时而不需要重新编写业务逻辑。从设计模式的角度来看,MVC 模式是一种复合模式,其将多个设计模式在一种解决方案中结合起来,用来解决许多设计问题。MVC 模式把用户界面交互分拆到不同的三种角色中,使应用程序被分成三个核心部件：Model(模型)、View(视图)和 Controller(控制器)。

5.4.1 MVC 结构

MVC 的结构如图 5.6 所示,视图中用户的输入被控制器解析后,控制器改变状态激活模型,模型根据业务逻辑维护数据,并通知视图数据发生变化,视图得到通知后从模型中获取数据刷新视图。

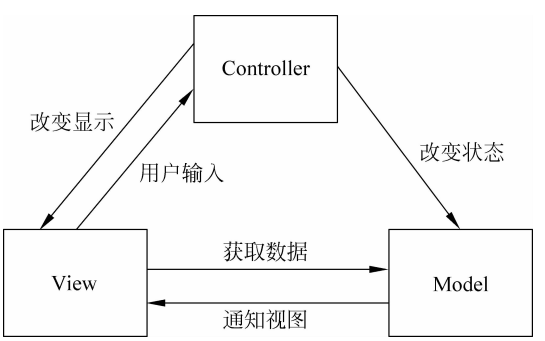

图 5.6 MVC 模式结构图

模型：模型持有所有的数据、状态和程序逻辑,独立于视图和控制器。模型表示企业数据和业务规则,在 MVC 的三个部件中,其拥有最多的处理任务。模型与数据格式无关,这样一个模型能为多个视图提供数据,由于应用于模型的代码只需写一次就可以被多个视图重用,所以减少了代码的重复性。

视图：用来呈现模型。视图是用户看到并与之交互的界面,通常直接从模型中取得其需要显示的状态与数据,对于相同的信息可以有多个不同的显示形式或视图。在客户端/服务器模式(C/S 模式)中,视图相当于客户端展示给用户的应用软件界面；在浏览器/服务器模式中(B/S 模式)中,视图就是由 HTML 元素组成的界面,但一些新的技术已层出不穷,包括 Adobe Flash 和像 XHTML、XML/XSL、WML 等一些标识语言和 Web Services。MVC 能为应用程序处理很多不同的视图,在视图中其实没有真正的处理发生,不管这些数据是联机存储的还是一个雇员列表,作为视图来讲,其只是作为一种输出数据并允许用户操纵的方式。

控制器：位于视图和模型中间，将输入进行解析并反馈给模型，通常一个视图具有一个控制器。控制器接收用户的输入并调用模型和视图去完成用户的需求，例如当单击 Web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何东西和做任何处理。其只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示返回的数据。

5.4.2　MVC 的特点

1. 优点

1）耦合性低

视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码，同样，一个应用的业务流程或者业务规则的改变只需要改动 MVC 的模型层即可。因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。

模型是自包含的，并且与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。如果把数据库从 MySQL 移植到 Oracle，或者改变基于 RDBMS 数据源到 LDAP，只需改变模型即可。一旦正确实现了模型，不管数据来自数据库或是 LDAP 服务器，视图将会正确地显示它们。由于运用 MVC 应用程序的三个部件是相互独立的，改变其中一个不会影响其他两个，所以依据这种设计思想能构造良好的松耦合的构件。

2）重用性高

随着技术的不断进步，需要用越来越多的方式来访问应用程序。MVC 模式允许使用各种不同样式的视图来访问同一个服务器端的代码，因为多个视图能共享一个模型，它包括任何 Web(HTTP)浏览器或者无线浏览器(WAP)，例如，用户可以通过电脑也可通过手机来订购某样产品，虽然订购的方式不一样，但处理订购产品的方式是一样的。由于模型返回的数据没有进行格式化，所以同样的构件能被不同的界面使用。例如，很多数据可能用 HTML 来表示，但是也有可能用 WAP 来表示，而这些表示所需要的命令是改变视图层的实现方式，而控制层和模型层无须做任何改变。由于已经将数据和业务规则从表示层分开，所以可以最大化地重用代码。模型也有状态管理和数据持久性处理的功能，例如，基于会话的购物车和电子商务过程也能被 Flash 网站或者无线联网的应用程序所重用。

3）生命周期成本低

MVC 使开发和维护用户接口的技术含量降低。

4）部署快

使用 MVC 模式可以使开发时间大大缩减，其使程序员(如 Java 开发人员)集中精力于业务逻辑，界面程序员(如 HTML 和 JSP 开发人员)集中精力于表现形式上。

5）可维护性高

分离视图层和业务逻辑层使得 Web 应用更易于维护和修改。

6）有利软件工程化管理

由于不同的层次各司其职，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、工具化管理程序代码。控制器也提供了一个好处，就是可以使用控制器来连接不同的模型和视图去完成用户的需求，这样控制器可以为构造应用程序提供强有力的手段。给定一些可重用的模型和视图，控制器可以根据用户的需求选择模型进行处理，然后选择视图将处理结果显示给用户。

2. 缺点

1) 没有明确的定义

完全理解 MVC 并不是很容易。使用 MVC 需要精心的计划,由于其内部原理比较复杂,所以需要花费一些时间去思考。同时由于模型和视图要严格分离,每个构件在使用之前都需要经过彻底的测试,这样也给调试应用程序带来了一定的困难。

2) 不适合小型、中等规模的应用程序

花费大量时间将 MVC 应用到规模并不是很大的应用程序通常会得不偿失。

3) 增加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。

4) 视图与控制器间过于紧密的连接

视图与控制器是相互分离,但却联系紧密的部件,视图没有控制器的存在,其应用是很有限的,反之亦然,这样就妨碍了其独立重用。

5) 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访问,也将损害操作性能。

6) 一般高级的界面工具或构造器不支持模式

改造这些工具以适应 MVC 需要和建立分离部件的代价是很高的,会造成 MVC 使用的困难。

3. 应用实例

基于 Web 的 MVC Framework 在 J2EE 的领域内已是空前繁荣,比较好的老牌 MVC 有 Struts、Webwork。新兴的 MVC 框架有 Spring MVC、Tapestry、JSF 等,这些大多是著名团队的作品。另外还有一些边缘团队的作品也相当出色,如 Dinamica、VRaptor 等。这些框架都提供了较好的层次分隔能力,在实现良好的 MVC 分隔的基础上,通过提供一些现成的辅助类库,同时也促进了生产效率的提高。有兴趣的读者,可以自行查找学习这些相关应用实例,对于理解和实践 MVC 有很大的帮助,本书限于篇幅,在此不一一列出。

5.5 基于构件的模式

构件来源于英文“Component”,在有些文献中也称为组件。目前对构件的定义,软件产业界还未形成统一的认识,北京大学的杨芙清教授将构件定义为应用系统中可以明确辨识的构成成分,可复用构件是具有相对独立的功能和可复用价值的构件。

构件是组成软件的基本单位,它包含以下 3 个内容:

- 构件是可复用的、自包含的、独立于具体应用的软件对象模块;
- 对构件的访问只能通过其接口进行;
- 构件不直接与别的构件通信。

1. 基于构件模式的基本结构

一般认为,构件是指语义完整、语法正确和有可重用价值的单位软件,是软件重用过程

中可以明确辨识的系统；结构上，它是语义描述、通信接口和实现代码的复合体。简单地说，构件是具有一定的功能，能够独立工作或能同其他构件装配起来协调工作的程序体，构件的使用同它的开发、生产无关。从抽象程度来看，面向对象技术已达到了类级重用（代码重用），它以类为封装的单位。这样的重用粒度还太小，不足以实现异构互操作和效率更高的重用。构件将抽象的程度提到一个更高的层次，它是对一组类的组合进行封装，并代表完成一个或多个功能的特定服务，也为用户提供了多个接口。整个构件隐藏了具体的实现，只用接口提供服务。

基于构件模式的软件架构的基本结构如图 5.7 所示。

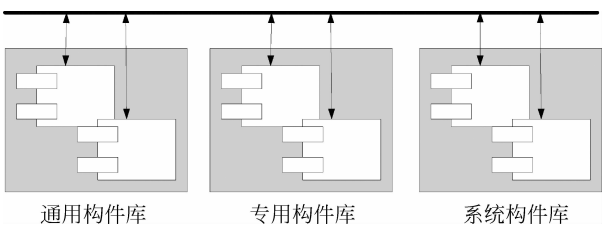

图 5.7　基于构件的软件架构模式的基本结构图

在基于构件的软件架构模式中，各种类型的构件是系统的主体，系统采用一定的构件组合方式将各构件有机结合在一起，完成系统功能。

近年来，构件技术发展迅速，已形成三个主要流派，分别是 IBM 的 CORBA、Sun 的 Java 平台和 Microsoft 的 COM+。

如果把软件系统看成是构件的集合，那么从构件的外部形态来看，构成一个系统的构件可分为 5 类。

- 独立而成熟的构件。独立而成熟的构件得到了实际运行环境的多次检验，该类构件隐藏了所有接口，用户只需用规定好的命令进行使用。例如，数据库管理系统和操作系统等。
- 有限制的构件。有限制的构件提供了接口，指出了使用的条件和前提，这种构件在装配时，会产生资源冲突、覆盖等影响，在使用时需要加以测试。例如，各种面向对象程序设计语言中的基础类库等。
- 适应性构件。适应性构件进行了包装或使用了接口技术，把不兼容性、资源冲突等进行了处理，可以不加修改地使用在各种环境中，例如 ActiveX 等。
- 装配的构件。装配的构件在安装时，已经装配在操作系统、数据库管理系统或信息系统的不同层次上，使用胶水代码（Glue Code）就可以进行连接使用。目前一些软件商提供的大多数软件产品都属于这一类。
- 可修改的构件。可修改的构件可以进行版本替换。如果对原构件修改错误或增加新功能，可以利用重新“包装”或写接口来实现构件的替换。这种构件在应用系统开发中使用得比较多。

基于构件的软件开发通常包括构件获取、构件分类和检索、构件评估、适应性修改以及将现有构件在新的语境下组装成新的系统。构件获取可以有多种不同的途径：

- 从现有构件中获得符合要求的构件，直接使用或做适应性修改，得到可重用的构件；

- 通过遗产工程,将具有潜在重用价值的构件提取出来,得到可重用的构件;
- 从市场上购买现成的商业构件;
- 开发新的符合要求的构件。

一个企业或组织在进行以上决策时,必须考虑到不同方式获取构件的一次性成本和以后的维护成本,从而做出最优的选择。

2. 基于构件模式的优缺点

基于构件的软件架构模式是软件开发发展到一定阶段的产物,是目前主流的应用软件架构模式,其优点体现在以下几个方面:

- 完全黑盒的软件复用,整个构件隐藏了具体的实现,只用接口提供服务,不仅实现了代码复用,还实现了核心功能复用;
- 总体架构的松耦合,构件与构件之间都是松耦合的,相同接口而不同实现的构件完全可以互换;
- 软件生产周期缩短,目前,人们更倾向于购买使用已开发好的构件来构建自己的系统,从而使开发时间大大缩短;
- 能适应远程访问的分布式、多层次异构系统;
- 具有灵活方便的升级能力和系统模块的更新维护能力。

基于架构的软件架构模式存在的缺点主要有以下几点:一是大量使用第三方构件给软件的安全带来隐患;二是购买构件可能使软件成本增加;三是非定制的构件中存在的无用部分可能带来性能负担。

5.6 软件架构建模技术

研究软件架构的首要问题是如何表示软件架构,即如何对软件架构建模。根据建模的侧重点的不同,可以将软件架构的模型分为 5 种:结构模型、框架模型、动态模型、过程模型和功能模型。在这 5 个模型中,最常用的是结构模型和动态模型。

1. 结构模型

结构模型是一个最直观、最普遍的建模方法。这种方法以架构的构件、连接件和其他概念来刻画结构,并力图通过结构来反映系统的重要语义内容,包括系统的配置、约束、隐含的假设条件、风格和性质。研究结构模型的核心是架构描述语言。

2. 框架模型

框架模型与结构模型类似,但它不太侧重描述结构的细节而更侧重于整体的结构。框架模型主要以一些特殊的问题为目标建立只针对和适应该问题的结构。

3. 动态模型

动态模型是对结构或框架模型的补充,研究系统“大颗粒”的行为性质,例如,描述系统的重新配置或演化。动态可能指系统总体结构的配置、建立或拆除通信通道或计算的过程。这类系统常是激励型的。

4. 过程模型

过程模型研究构造系统的步骤和过程,因而其结构是遵循某些过程脚本的结果。

5. 功能模型

功能模型认为架构是由一组功能构件按层次组成的，下层向上层提供服务。它可以看作是一种特殊的框架模型。

这 5 种模型各有所长，如果将 5 种模型有机地统一在一起，形成一个完整的模型来刻画软件架构，将能更加准确、全面地反映软件架构。

5.6.1　软件架构“4＋1”视图模型

Philippe Kruchten 在 1995 年提出了一个“4＋1”的视角模型。“4＋1”模型从 5 个不同的视角包括逻辑视角、过程视角、物理视角、开发视角和场景视角来描述软件架构。每一个视角只关心系统的一个侧面，5 个视角结合在一起才能够反映系统的软件架构的全部内容。“4＋1”视图模型如图 5.8 所示。

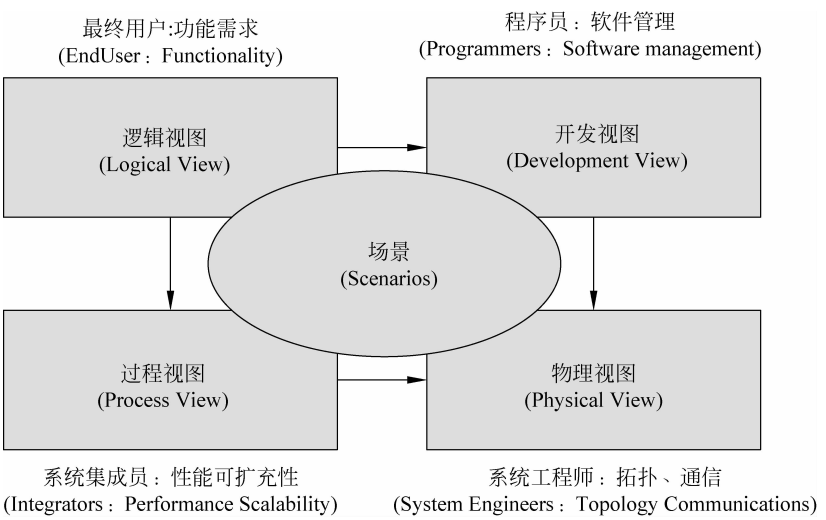

图 5.8　软件架构“4＋1”视图模型

- 逻辑视图(Logical View)，设计的对象模型(使用面向对象的设计方法时)；
- 过程视图(Process View)，捕捉设计的并发和同步特征；
- 物理视图(Physical View)，描述了软件到硬件的映射，反映了分布式特性；
- 开发视图(Development View)，描述了在开发环境中软件的静态组织结构；
- 场景(Scenarios)，是用例表述的需求的抽象。

5.6.2　“4＋1”视图模型建模方法

下面简述每个不同视角的建模方法。

1. 逻辑视角－逻辑结构

逻辑视角主要对系统的逻辑结构进行建模，它采用的是面向对象分解的方法。

逻辑架构主要支持功能性需求，即为用户提供服务方面系统所应该提供的功能。系统分解为一系列的关键抽象，大多数来自于问题域，表现为对象或对象类的形式。它们采用抽象、封装和继承的原理。分解并不仅仅是为了功能分析，而且用来识别遍布系统各个部分的

通用机制和设计元素。一般使用 Rational/Booch 方法来表示逻辑架构,主要借助于类图和类模板的手段。类图用来显示一个类的集合和它们的逻辑关系:关联、使用、组合和继承等等。相似的类可以划分成类集合。类模板关注于单个类,它强调主要的类操作,并且识别关键的对象特征。如果需要定义对象的内部行为,则使用状态转换图或状态图来完成。公共机制或服务可以在类功能(Class Utilities)中定义。对于数据驱动程度高的应用程序,可以使用其他形式的逻辑视图,例如 E-R 图,来代替面向对象的方法。

2. 进程视角 - 进程架构

进程视角主要对系统的进程架构进行建模,采用过程分解的方法。

进程架构考虑一些非功能性的需求,如性能和可用性。它解决并发性、分布性、系统完整性和容错性的问题,以及逻辑视图的主要抽象如何与进程结构相配合在一起,即在哪个控制线程上,对象的操作被实际执行。

进程架构可以在几种层次的抽象上进行描述,每个层次针对不同的问题。在最高的层次上,进程架构可以视为一组独立执行的通信程序的逻辑网络,它们分布在整个一组硬件资源上,这些资源通过 LAN 或者 WAN 连接起来。多个逻辑网络可能同时并存,共享相同的物理资源。例如,独立的逻辑网络可能用于支持离线系统与在线系统的分离,或者支持软件的模拟版本和测试版本的共存。

进程是构成可执行单元任务的分组。进程代表了可以进行策略控制过程架构的层次(即开始、恢复、重新配置及关闭)。另外,进程可以就处理负载的分布式增强或可用性的提高而不断地被重复。

软件被划分为一系列单独的任务。任务是独立的控制线程,可以在处理节点上单独地被调度。接着,区分一下主要任务、次要任务。主要任务是可以唯一处理的架构元素;次要任务是由于实施原因而引入的局部附加任务(周期性活动、缓冲和暂停等等),它们可以作为轻量线程来实施。主要任务的通信途径是定义好的交互任务通信机制,这些通信机制包括:基于消息的同步或异步通信服务、远程过程调用及事件广播等。次要任务则以共享内存的形式来通信。

进程视图的架构模式:许多模式可以适用于进程视图。例如管道和过滤器、客户端/服务器以及各种多个客户端/单个服务器和多个客户端/多个服务器的变体等。

3. 开发视角 - 开发架构

开发视角主要对系统的开发架构进行建模,采用子系统分解的方法。

开发架构关注软件开发环境下实际模块的组织。软件打包成小的程序块(程序库或子系统),它们可以由一位或几位开发人员来开发。子系统可以组织成分层结构,每个层为上一层提供良好定义的接口。

系统的开发架构用模块和子系统图来表达,显示了“输出”和“输入”关系。完整的开发架构只有当所有软件元素被识别后才能加以描述。但是,可以列出控制开发架构的规则:分块、分组和可见性。

大部分情况下,开发架构考虑的内部需求与以下几项因素有关:开发难度、软件管理、重用性和通用性及由工具集、编程语言所带来的限制。开发架构视图是各种活动的基础,如:需求分配、团队工作的分配(或团队机构)、成本评估和计划、项目进度的监控、软件重用

性、移植性和安全性。它是建立产品线的基础。

开发视图的架构模式推荐使用分层的架构模式，定义 4～6 个子系统层。每层均具有良好定义的职责。设计规则是某层子系统依赖同一层或低一层的子系统，从而最大程度地减少具有复杂模块依赖关系的网络的开发量，得到层次式的简单策略。

4. 物理视图－物理架构

物理视角主要对系统的物理架构进行建模，是软件至硬件的映射。

物理架构主要关注系统非功能性的需求，如可用性、可靠性（容错性）、性能（吞吐量）和可伸缩性。软件在计算机网络或处理节点上运行，被识别的各种元素（网络、过程、任务和对象）需要被映射至不同的节点；一般希望使用不同的物理配置：一些用于开发和测试，另外一些则用于不同地点和不同客户的部署。因此软件至节点的映射需要高度的灵活性及对源代码产生最小的影响。

5. 场景

场景综合所有的视图，对系统总体架构进行建模。

4 种视图的元素通过数量比较少的一组重要场景（更常见的是用例）进行无缝协同工作，需要为场景描述相应的脚本（对象之间和过程之间的交互序列）。在某种意义上场景是最重要的需求抽象。

场景是其他视图的冗余（因此才称为“＋1”），但它起到了两个作用：一是作为一项驱动因素来发现架构设计过程中的架构元素；二是作为架构设计结束后的一项验证和说明功能，既以视图的角度来说明，又作为架构原型测试的出发点。

6. 模型的剪裁

并不是所有的软件架构都需要“4＋1”视图。无用的视图可以从架构描述中省略，例如：只有一个处理器，则可以省略物理视图；而如果仅有一个进程或程序，则可以省略过程视图。对于非常小型的系统，可能逻辑视图与开发视图非常相似，因此不需要分开描述。场景对于所有的情况均适用，一般不能省略。

5.6.3　软件架构建模的迭代过程

软件架构的建模过程不是一个简单的线性过程，而是一个循环迭代的过程。软件架构的建模使用一种更具有迭代性质的方法，即架构先被原型化、测试、估量、分析，然后在一系列的迭代过程中被细化。该方法除了减少了与架构相关的风险之外，对于项目而言还有其他优点：团队合作、培训，加深对架构的理解，深入程序和工具等等（此处提及的是演进的原型，逐渐发展成为系统，而不是一次性的试验性的原型）。这种迭代方法还能够使需求被细化、成熟化并能够被更好地理解。

这种方法称为场景驱动（Scenario-Driven）的方法。在这种方法中系统大多数关键的功能以场景或用例的形式被捕获。“关键”意味着：最重要的功能，系统存在的理由，使用频率最高的功能，或体现了必须减轻的一些重要的技术风险。

软件架构建模的迭代过程分为两个阶段。

1. 开始阶段

这个阶段的工作任务归纳为以下几个方面。

- 基于风险和重要性为某次迭代选择一些场景。场景可能被归纳为对若干用户需求的抽象。
- 形成“稻草人式的架构”,然后对场景进行“描述”,以识别主要的抽象(类、机制、过程和子系统),分解成为序列对(对象和操作)。
- 所发现的架构元素被分布到 4 个视图中:逻辑视图、进程视图、开发视图和物理视图。
- 然后实施、测试、度量该架构,这项分析可能检测到一些缺点或潜在的增强要求。
- 总结经验教训。

2. 循环阶段

这个阶段是个不断往复循环的过程,其工作任务可以归纳为以下几个方面。

- 重新评估风险。
- 选择能减轻风险或提高结构覆盖的额外的少量场景。
- 然后试着在原先的架构中描述这些场景。
- 发现额外的架构元素,或有时还需要找出适应这些场景所需的重要架构变更。
- 更新 4 个主要视图:逻辑视图、进程视图、开发视图和物理视图。
- 根据变更修改现有的场景。
- 升级实现工具(架构原型)来支持新的、扩展了的场景集合。
- 测试。如果可能的话,在实际的目标环境和负载下进行测试。
- 然后评审这 5 个视图来检测简洁性、可重用性和通用性的潜在问题。
- 更新设计准则和基本原理。
- 总结经验教训。

经过多次循环迭代后,系统架构模型趋于稳定,循环终止,迭代过程结束。

为了实际的系统,初始的架构原型需要进行演进。较好的情况是在经过 2 次或 3 次迭代之后,结构变得稳定:主要的抽象都已被找到;子系统和过程都已经完成;所有的接口都已经实现。接下来则是软件设计的范畴,这个阶段可能也会用到相似的方法和过程。

这些迭代过程的持续时间可能参差不齐,主要影响因素有:所实施项目的规模,参与项目人员的数量,他们对本领域和方法的熟悉程度,以及对该系统和开发组织的熟悉程度等等。

本章小结

软件架构作为软件工程中的一个新兴研究领域,是随着描述大型、复杂系统结构的需要和开发人员及计算机科学家在大型软件系统的研制过程中对软件系统理解的逐步深入而发展起来的。尽管目前人们对于软件架构的名称、定义还存在许多争议,但学术界和软件工业界已经普遍认同软件架构研究的意义。

管道和过滤器模式、面向对象模式、分层模式和知识库模式等是从经典的软件架构模式中抽取出的最常见的架构模式。而客户机/服务器模式和浏览器/服务器模式则是网络环境下流行的架构模式。本章对这些模式分门别类进行介绍,总结了各个模式的基本结构,简述了其优缺点,并给出了模式应用。

本章最后一节是软件架构的建模技术，介绍了软件架构“4＋1”视图模型，分析了该视图模型的基本结构，讨论了采用该视图模型进行软件架构建模的方法以及软件架构建模的迭代过程。

习题

1. 促使人们要研究软件架构的动力是什么？
2. 简述软件架构技术发展的 4 个阶段。
3. 管道和过滤器模式能应用于交互式应用系统中吗？为什么？
4. 试举出一个应用面向对象架构的应用实例。
5. 在 ISO/OSI 七层网络模型中体现出了分层模式的哪些优点？
6. 简述多层客户机/服务器模式相对于单层客户机/服务器模式的优缺点，以及这两种模式分别适用于什么应用系统。
7. 为什么浏览器/服务器模式会成为当前网络应用系统的主流架构模式？
8. 简述软件架构建模的迭代过程。

参考文献

[1] 覃征，何坚. 软件体系结构[M]. 西安：西安交通大学出版社，2002.
[2] 张友生. 软件体系结构[M]. 北京：清华大学出版社，2004.
[3] 李代平. 软件体系结构教程[M]. 北京：清华大学出版社，2008.
[4] 余雪丽. 软件体系结构及实例分析[M]. 北京：科学出版社，2004.
[5] Stephen T Albin. The art of software architecture：design methods and techniques. Indianapolis，Ind.：Wiley Pub.，c2003.
[6] 李千目. 软件体系结构设计[M]. 北京：清华大学出版社，2008.
[7] 温昱. 软件架构设计[M]. 北京：电子工业出版社，2007.
[8] David M Dikel，David Kane，James R Wilson. Software architecture：organizational principles and patterns[M]. 北京：高等教育出版社；Pearson Educatio 出版集团，2002.