第5章 软件体系结构风格

今日的大多数软件很像埃及金字塔，由千百万砖头堆砌起来，层层相切，没有着整体的结构，是由畜力和成千上万奴隶的力量建立起来的。

——Alan Kay

多年来，人们在开发某些类型软件过程中积累起来的组织规则和结构，形成了软件体系结构风格。软件体系结构风格是总结人们设计经验而形成结构较为巩固、组织较为统一的形式，是一种适合于多种场合的相似结构的抽象。

风格是艺术概念，是艺术作品在整体上呈现的有代表性的面貌。软件体系结构风格并不是某一种特定系统的结构，而是一个结构的类型。软件体系结构风格描述特定系统组织方式的惯用范例，强调了软件系统中通用的组织结构。一个软件体系结构风格定义了构件和连接件类型的符号集，以及规定了它们怎样组合起来的约束集合。

本章共分九个部分：5.1节为软件体系结构风格概述，5.2节为基本风格解析，5.3节为案例分析，5.4节为客户/服务器风格，5.5节为三层C/S体系结构风格，5.6节为浏览器/服务器风格，5.7节为C/S与B/S混合结构风格，5.8节为正交软件体系结构风格，5.9节为异构结构风格。

5.1 软件体系结构风格概述

软件体系结构设计的一个核心问题是，能否使用重复的体系结构模式，即能否达到体系结构级的软件重用。即，能否在不同的软件系统中使用同一种体系结构。基于该目的，相关研究者们开始研究和实践软件体系结构的风格和类型问题。

软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式。体系结构风格定义一个系统家族，即一个体系结构定义一个词汇表和一组约束。词汇表中包含了一些构件和连接件类型，这组约束指出系统是如何将这些构件和连接件组合起来的。体系结构风格反映了领域中众多系统所共有的结构和语义特征，并指导如何将各个模块和子系统有效地组织成一个完整的系统。按照这种方式理解，软件体系结构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

对软件体系结构风格的研究和实践，促进了对设计的重用，一些经过实践证实的解决方案也可以可靠地用于解决新的问题。体系结构风格的不变部分，使不同的系统可以共享同一个实现代码。只要系统是使用正常的、规范的方法来组织，就可以使别的设计者很容易地理解系统的体系结构。例如，如果某人把系统描述为“客户/服务器”模式，则不必给出设计

细节,我们立刻就会明白系统是如何组织和工作的。

软件体系结构风格为大粒度的软件重用提供了可能。然后,对于应用体系结构风格来说,由于视点的不同,系统设计师有很大的选择余地。要为系统选择或设计某一个体系结构风格,必须根据特定项目的具体特点,进行分析比较后再确定,体系结构风格的使用几乎完全是特定的。

讨论体系结构风格时,我们要回答这样一些问题。

(1) 设计词汇表是什么?

(2) 构件和连接件的类型是什么?

(3) 可允许的结构模式是什么?

(4) 基本的计算模型是什么?

(5) 风格的基本不变性是什么?

(6) 其使用的常见例子是什么?

(7) 使用此风格的优缺点是什么?

(8) 其常见的特例是什么?

这些问题的回答,包括了体系结构风格最关键的四要素内容,即提供一个词汇表、定义一套配置规则、定义一套语义解释原则和定义对基于这种风格的系统所进行的分析。

5.2 基本风格解析

5.2.1 管道-过滤器

管道-过滤器风格最早出现在UNIX中,至今已有20多年了,适用于对有序数据进行一系列已经定义的相互计算的应用程序。管道-过滤器模式下,每个功能模块都有一组输入和输出。

功能模块从输入集合读入数据流,并在输出集合产生输出数据流,即功能模块对输入数据流进行增量计算得到输出数据流。管道-过滤器模式下,功能模块称为过滤器(Filter),功能模块间的连接可以看作输入、输出数据流之间的通路,所以称其为管道(Pipe)。管道-过滤器模式的示意图如图5-1所示。

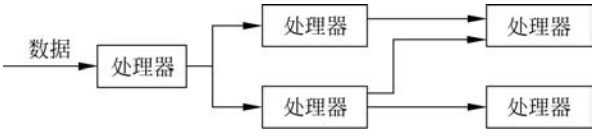

图5-1 管道-过滤器模式的示意图

1. 设计问题

如果要建立一个必须处理或转换输入数据流的系统,用单个组件实现会过于臃肿,需求不容易变动,所以可能要通过替换或重新排列处理步骤为灵活性做规划。处理步骤的内部连接必须考虑以下因素。

(1) 未来系统的升级通过替换处理步骤或重组步骤就可以做到。

(2) 不同的语境中,小的处理步骤要比组件更易于重用。

(3) 不相连的处理步骤不共享信息。

(4) 存在不同的输入数据源,例如,网络连接或通过硬件传感器提供读数。

(5) 可以用多种方式给出或存放最终结果。

(6) 明确存放中间结果。

(7) 可能有并行、异步处理的要求。

2. 解决方案

管道-过滤器体系结构模式把系统任务分成几个序贯的处理步骤。这些步骤通过系统的数据流连接,一个步骤的输出是下一个步骤的输入。每个处理步骤由一个过滤器组件实现。过滤器消耗和转发增长的数据,在产生任何输出之前消耗所有输入,以达到低延迟并能够真正地并行处理。系统的输入由诸如文本文件等数据源提供。输出流入数据汇点,如文件、终端。数据源、过滤器和数据汇点由管道顺序连接起来,实现相连处理步骤间的数据流动。通过管道联合的过滤器序列称为处理流水线。

从整个系统的输入和输出关系来看,各个过滤器可以对其输入进行局部独立处理变换,产生部分的计算结果。按过滤器的激活方式可以分为被动过滤器和主动过滤器,被动过滤器是通过函数或过程调用激发的,而主动过滤器是作为独立的线程任务激发工作的。

过滤器是独立运行的部件,不受其他过滤器运行的影响。非临近的过滤器之间不共享任何状态,自身也是无状态的。每次加工后,过滤器会回到初始等待状态。独立性还表现在对其上下游连接的过滤器的"无知",只需关心输入的到来和形式、加工处理的逻辑、产生的输出形式。整个结果的正确性不依赖于各个过滤器运行的先后次序。

这种风格有以下特征。

(1) 在管道-过滤器风格中,构件即过滤器,对输入流进行处理、转换,处理后的结果在输出端流出。这种计算常常是递进的,所以,可能在全部的输入接收完之前就开始输出。可以并行地使用过滤器。

(2) 连接件位于过滤器之间,起到信息流的导管的作用,即管道。

(3) 每个构件都有输入/输出集合,构件在输入处读取数据流,并在输出处生成数据流。

(4) 过滤器必须是独立的实体,不了解信息流从哪个过滤器流出,也不需要知道信息将流入哪个过滤器。可以指定输入的格式,可以确保输出的结果,但是可能不知道在管道之后将会是什么样子。过滤器之间也不共享状态。

管道-过滤器模式的特性之一是过滤器的相对独立性,即过滤器独立完成自身功能,相互之间无须进行状态交互。此外,各过滤器无须知道输入管道和输出管道所连接的过滤器的存在,仅仅需要对输入管道的输入数据流进行限制,并保证输出管道的输出数据流有合适的内容,但并不知道连接在其输入、输出管道上的其他过滤器的实现细节。同时,整个管道过滤网络的最终输出和网络中各个过滤器执行操作的顺序无关。

采用管道-过滤器模式建立的系统主要有以下几个优点。

(1) 由于每个构件的行为不受其他构件的影响,因此整个系统的行为比较易于理解。设计者可以将系统抽象成一个"黑匣子",其输入是系统中第一个过滤器的输入管道,输出是系统中最后一个过滤器的输出管道,其内部各功能模块的具体实现对用户完全透明。

(2) 支持功能模块的复用。任意两个过滤器只要在相互的输入、输出管道格式上达成一致,就可以连接在一起。如图 5-2 所示,过滤器 A 和过滤器 B 只要对管道 C 中传输的数

据格式达成一致,就可以实现互联。其中,过滤器A并不关心过滤器B如何处理管道C的内容,而过滤器B也不知道管道C的内容究竟是如何产生的,即在管道-过滤器模式中,过滤器之间仅需很少的信息交换,就可以完成互联。

(3) 具有较强的可维护性、可扩展性。可维护性体现在系统过滤器部件的更新或升级。由于技术改进等原因,过滤器A的实现发生了改变,采用新技术开发的过滤器D具有和过滤器A完全相同的输入、输出管道接口,这时可以直接将过滤器A替换为过滤器D,而无须对系统中的其他过滤器或管道进行任何修改。

可扩展性体现在系统功能的扩充上,如图5-3所示。由于业务流程的变化,系统必须增加新的功能模块,实现新功能的过滤器X通过和系统原有过滤器B互联成为系统的一部分。这时,系统的其余部分没有任何变化,整个系统的功能就得到了增强。

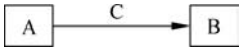

图5-2 管道-过滤器支持功能模块的复用

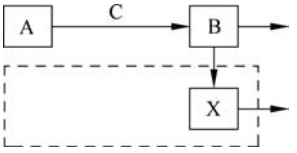

图5-3 管道-过滤器的可扩展性

(4) 支持特殊的分析,如吞吐量计算和死锁检测。利用管道-过滤器模式图可以很容易地得到系统的资源使用与请求状态图,然后,根据操作系统原理等相关理论中的死锁检测方法,就可以分析出系统目前所处的状态,是否存在死锁可能及如何消除死锁等问题。

(5) 支持并发执行。基于管道-过滤器模式的系统存在很多并行的过滤器,这样的系统在实际运行时,可以将存在并发可能的多个过滤器看作多个并发的任务并行执行,从而大大提高了系统的整体效率,加快了处理速度。当然,调度并行任务时,必须有相应的并行算法作为基础,否则可能导致系统功能的混乱。

任何事物都是一个矛盾的统一体,有利就有弊,管道-过滤器模式当然也不例外,其不足之处主要体现在以下几个方面。

(1) 往往会导致系统处理过程的成批操作。虽然系统中的过滤器对数据采取增量处理的方式,但过滤器的模块独立性很强,相互之间的耦合关系很弱,这样设计者必须要求每一个过滤器完成输入到输出的完整转换。此外,由于过滤器的传输特性,管道-过滤器模式通常不适于交互性很强的应用。尤其是在系统需要逐步显示数据流变化的过程时,问题就会变得更加难以解决,因为增量显示和过滤器的输出数据差距太大,几乎无法显示系统数据流细微的变化过程。

(2) 在处理两个独立但又相关的数据流时,可能会遇到困难。例如,多过滤器并发执行时数据流之间的同步问题等。

(3) 需要对数据传输进行特定的处理时,导致对于每个过滤器的解析输入和格式化输出要做更多的工作,带来系统复杂性的上升。根据实际设计要求,设计者也需要对数据传输进行特定的处理(如为了防止数据泄漏而采取加密等手段),导致过滤器必须对输入、输出管道中的数据流进行解析或反解析,增加了过滤器具体实现的复杂性。

(4) 并行处理获得的效率往往是一种假象,主要有以下几个原因。

① 过滤器之间传输数据的代价,比起单个过滤器负担的计算的代价来相对要高。对于

使用网络连接的小的过滤器组件或流水线，尤其如此。

② 一些过滤器在产生输出之前消耗所有输入，或者是任务所要求的（如存储），或者是过滤器代码写得很差。

③ 线程和进程之间的关联转换，在单处理器机器上通常是一个成本很高的操作。

④ 过滤器的同步化可能要经常终止或启动过滤器，尤其当管道仅有一个小缓冲区时。

5.2.2 数据抽象和面向对象风格

抽象数据类型概念对软件系统有着重要作用，目前，软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和相应操作封装在一个抽象数据类型或对象中。这种风格的构件是对象，或者说是抽象数据类型的实例。对象是一种被称为管理者的构件，负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

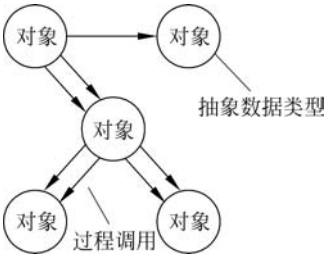

图 5-4 数据抽象和面向对象风格的体系结构

图 5-4 给出了数据抽象和面向对象风格的体系结构。

面向对象的系统有以下优点。

（1）因为对象对其他对象隐藏表示，所以可以改变一个对象的表示，而不影响其他的对象。

（2）设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

但是，面向对象的系统也存在着以下问题。

（1）为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识，只要一个对象的标识改变了，就必须修改所有其他明确调用它的对象。

（2）必须修改所有显式调用它的其他对象，并消除由此带来的一些副作用。例如，如果 A 使用了对象 B，C 也使用了对象 B，那么 C 对 B 的使用所造成的对 A 的影响，可能是预想不到的。

5.2.3 基于事件的隐式调用风格

基于事件的隐式调用风格的思想是构件不直接调用一个过程，而是触发或广播一个或多个事件。系统中的其他构件中的过程在一个或多个事件中注册，当一个事件被触发，系统自动调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。

从体系结构上来说，这种风格的构件是一些模块，这些模块既可以是一些过程，又可以是一些事件的集合。过程可以用通用的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被调用。

基于事件的隐式调用风格的主要特点是，事件的触发者并不知道哪些构件会被这些事件影响。这样，不能假定构件的处理顺序，甚至不知道哪些过程会被调用，因此许多隐式调用的系统也包含显式调用作为构件交互的补充形式。

支持基于事件的隐式调用的应用系统有很多。例如，在编程环境中用于集成各种工具，在数据库管理系统中确保数据的一致性约束，在用户界面系统中管理数据，以及在编辑器中

支持语法检查。例如在某系统中,编辑器和变量监视器可以登记相应 Debugger 的断点事件。当 Debugger 在断点处停下时,声明该事件,由系统自动调用处理程序,如编辑程序可以卷屏到断点,变量监视器刷新变量数值。而 Debugger 本身只声明事件,并不关心哪些过程会启动,也不关心这些过程做什么处理。

隐式调用系统主要有以下两个优点。

(1) 为软件重用提供了强大的支持。当需要将一个构件加入现存系统时,只需注册到系统的事件中。

(2) 为改进系统带来了方便。当用一个构件代替另一个构件时,不会影响到其他构件的接口。

隐式调用系统主要有以下三个缺点。

(1) 构件放弃了对系统计算的控制。一个构件触发一个事件时,不能确定其他构件是否会响应它。而且即使知道事件注册了哪些构件的过程,也不能保证这些过程被调用的顺序。

(2) 数据交换的问题。有时数据可被一个事件传递,但在另一些情况下,基于事件的系统必须依靠一个共享的仓库进行交互。这些情况下,全局性能和资源管理便成了问题。

(3) 既然过程的语义必须依赖于被触发事件的上下文约束,那么关于正确性的推理就存在问题。

5.2.4 分层系统风格

诺贝尔经济学奖获得者西蒙(Herbert Alexander Simon)认为,"要构造一门关于复杂系统的比较正规的理论,有一条路,就是求助于层级理论。我们可以期望,一个复杂系统必然是从简单系统进化而来的,在这个世界中,复杂系统是层级结构的"。对于软件这样复杂的人造事务,发现层级和运用层级是分析、构建的基本原则。所谓分层式体系结构,是按照层次组织软件的一种软件体系结构。其中,每一层软件建立在低一层的软件层上。位于同一层的软件系统或子系统具有同等的通用性,在下一层的软件比在上一层的软件通用性更强。一个层次可视为同等通用档次的一组(子)系统。

因此,在分层的体系结构中,最高层是应用层,可包容许多应用系统;次高层是构件层,可包括多个可复用构件库系统,可用于建立应用系统。应用系统建立在构件层之上,而此构件层中的许多构件库系统又建立在更低层次的构件库系统之上。

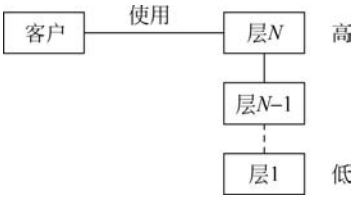

图 5-5 分层模型

分层风格适用于可以按照层次结构来组织不同类别的相关服务的应用程序。一个分层风格的系统按照层次结构组织,每一层向它的上层提供服务,同时又是它的下层客户,如图 5-5 所示。在某些系统中,除了邻接的层,一个内部层次对于其他外部层次是隐藏的,对体系结构的约束包括把系统内的交互限制在邻接层次之间。交互只在相邻的层间发生,同时,这些交互按照一定协议进行。连接件可以用层次间的交互协议来定义。每个独立层都要防止较高层直接访问较低层。每一层次是由不同的部件构成的实体集合。层内的部件可以交互;相邻层的部件,可直接从上向下调用,还可以设计统一的层调用接口对层进行保护。

下面给出使用层次设计的软件结构特性。并不是每个软件都会出现下列所有情况，在简单的层次结构中，仅能看到第 1 种和第 2 种情况。

情况 1：用户对层 N 产生需求，但层 N 不能独立完成这个请求，所以调用层 $N-1$ 的相应操作服务。在处理过程中又进一步向层 $N-2$ 发出请求，以此类推，最终达到层 1。如果需要，请求的回应将从层 1 逐层上传，最后到达层 N。这种自上而下调用的特点，是层 J 常把层 $J+1$ 发出的请求转换成对 $J-1$ 的多个请求。这是由于层 J 比层 $J-1$ 处于更高的抽象层上，把高层请求映射成低层的更基础的操作。

情况 2：这是从层 1 开始的自底向上的操作链过程。例如，输入设备驱动器检测到输入时的动作过程，驱动器把输入转制成内部的格式，并报告给层 2，由层 2 启动解释等，如此进行下去。通过这种方式，数据穿过各层一直到达最高层。

自顶向下的信息和控制通常被描述成“请求”，自底向上的方式被描述成“通知”。正如在情况 1 中提到的，自顶向下的请求常被分解成低层的几个请求。相对应，向上的多个通知常被合并成一条向上层的通知。当然，两种情况下都可以维持 1 对 1 关系。

情况 3：这是请求仅仅通过相邻的部分层的情况。例如，如果层 $N-1$ 能够满足要求，顶层的请求仅到达层 $N-1$ 就足够了，不需要再向下层分解和传递。该情况的一个例子是层 $N-1$ 为一个缓存器，而且来自层 N 的请求不必被发送到层 1 就可以得到完成的服务。这些缓存器维持着状态信息。

情况 4：这是与情况 3 类似的情况。层 1 检测到一个事件，但并没有一直传到顶层，而是仅仅向上传到部分层就停止了。例如，在通信协议中，过去的某个时刻对数据提出请求的用户，提出对同一数据重新发送的请求。与此同时，上层的服务器已经完成了对数据的请求应答，而应答与重新发送的请求交织在一起了。在这种情况下，服务器注意到这些情况，截获重新发送的请求，而不需要采取进一步的动作。

情况 5：这是具有两个 N 层结构的相互通信的堆栈。这种情形以通信协议中的“协议堆栈”而出名。一边堆栈的层 N 描述了一个请求，一直传递到层 1，接着又被传递到另一边堆栈的层 1，并且自下而上，穿过堆栈的各层到达顶层。处理过后的响应过程采取了相反的路径，最终回到发送请求堆栈的顶层。

分层风格的体系结构有以下优点。

(1) 由于对层次的邻接层数目进行了限制，所以系统易于改进和扩展。

(2) 每一层的软件都易于重用，并可为某一层次提供多种可互换的具体实现。如果一个独立层体现了一个良好定义的抽象，且有良好定义和文档化的接口，那么该层就可在多个语境中被重用。虽然已存在的层可以重用，但是开发人员往往宁愿重写这个功能，因为现有层不能准确符合他们的要求，而且分层会导致性能降低。这是一个误区，已存在层的黑盒重用会显著地减少开发工作量且会减少缺陷数。

(3) 分层系统所支持的设计体现了不断增加的抽象层次，这样，一个复杂问题的求解就被分解为一系列递增的步骤。

(4) 标准化支持。清晰定义和接受共同的抽象层，能促进标准化任务和接口的开发。同一接口的不同实现可以替换使用。这样，可以使用不同层的不同售主的产品。一个众所周知的标准化接口的例子是可移植操作系统接口(Portable Operating System Interface，POSIX)编程接口。

(5) 局部依赖性。层之间的标准化接口往往会限制被改动层的改动代码的影响。硬件、操作系统、窗口系统、特殊数据格式等,它们的变动往往只影响一层,不用改变其他层就可以适应被改变层,这支持了系统的可移植性。对可测试性的支持也很好,因为可以测试系统中独立于其他组件的特殊层。

(6) 可替换性。独立层实现不需要太费劲就可以被语义上等价的实现所替换。如果层之间的连接在代码中是硬连线的,则可以用新层的实现的名称来更新。

例如,一个新的硬件I/O设备通过安装正确的驱动程序就可投入使用,驱动程序可以插入或替换旧的,高层不受替换的影响。诸如以太网的一种传输媒介能用令牌环替换,这种情况中,高层不需要改动接口,并可以和以前一样继续向低层请求服务。但是如果想在两个接口和服务不完全匹配的层之间切换,则必须在这两层之上建立一个隔离层。可替换性增长了编程工作的代价,可能降低运行性能。

分层风格的体系结构有以下缺点。

(1) 应当如何界定层次间的划分是一个较为复杂的问题。

(2) 更改行为的重叠。层的行为改变时,会出现一个严重问题。高层往往不受低层改动的影响。这就允许系统通过去掉低层和/或代之以更快的解决方案(如硬件)等透明性调整。如果不得不在许多层上做相当数量的重复工作以合并外观上的局部变动,那么分层便成为一个缺点。

(3) 降低效率。说起来,一个分层体系结构的效率往往要低于整体结构或一个“对象的海洋”。如果在上层中的高层服务很大程度上依赖于最底层,则所有的相关数据必须通过一些中间层转换,且可以转换若干次。同样,由低层产生的所有结果或错误信息也传送到最高层。例如,通信协议通过添加消息头和尾,从高层传输消息。

(4) 不必要的工作。如果低层执行的某些服务执行了多余或重复的工作,而这些工作并非高层真正需要的,那么这对性能的影响是负面的。通信协议堆栈中的多路分解就是这种现象的一个例子,几个高层需求导致相同输入位序列被读多次,因为每一个高层需求对位的不同子集感兴趣。另一个例子是文件传输中的错误纠正。通用目的低层传输系统最先写,把可靠性建立在较高层中,如使用校验位。

(5) 难以认可层的正确粒度。层数太少的分层体系结构不能完全发挥这种模式在可重用性、可更改性和可移植性上的潜力。相反,层过过多会引入不必要的复杂性和层间分离的冗余以及变元和返回值传输的开销。层粒度的确定和层任务的分配是困难的,但对体系结构的质量是很关键的。如果潜在客户范围内的应用适应所定义的层,则可以仅使用一个标准体系结构。

在实现体系结构的技术能力方面,分层模式对抽象、信息隐藏、关注点分离、模块化、耦合和内聚、充分性、完整性和原始性的实现有益处。在非功能性属性方面,有益于易修改性、互操作性、可测试性、可重用性。

5.2.5 仓库风格和黑板风格

仓库风格的体系结构由两个构件组成:一个中央数据结构,表示当前状态;一个独立构件的集合,对中央数据结构进行操作。

对于系统中数据和状态的控制方法有两种。一种方法是由输入事务选择进行何种处

理，并把执行结果作为当前状态存储到中央数据结构中，这时，仓库是一个传统的数据库体系结构。另一种方法是由中央数据结构的当前状态决定进行何种处理。这时，仓库是一个黑板体系结构。即黑板体系结构是仓库体系结构的特殊化。

1. 设计问题

黑板体系结构主要应用于需要对数据做出复杂解释的信号处理领域中，包括语音和模式识别领域。在其他具有松散耦合关系数据的共享访问的系统中也有应用，这些系统尚不存在确定的解决方法，涉及从原始数据向高层结构转换的应用问题。例如，图、表、视觉、图像识别、语言识别、预警等应用领域都属于这类问题。这类问题的特点是，当把整个问题分解成子问题时，各个子问题涵盖了不同的领域知识和解决方法。每个子问题的解决需要不同的问题表达方式和求解模型，多数情况下找不到确定的求解策略。这与把问题分解成多个求解部分的功能分解形成对照。

在上面某些问题中，还需要考虑不确定性和近似推理知识。这种问题的每个求解步骤都可能产生多个可能的解，因而往往需要考虑寻求最佳或可接受的解。

2. 解决方案

黑板体系结构实现的基本出发点，是已经存在一个对公共数据结构进行协同操作的独立程序集合。每个这样的程序专门解决一个子问题，但需要协同工作共同完成整个问题的求解。这些专门程序是相互独立的，之间不存在互相调用，也不存在可事先确定的操作顺序。相反，操作次序是由问题求解的进行状态决定的。

因此，黑板体系结构存在一个中心控制部件，就是所谓的“黑板”。这是一个数据驱动或状态驱动的控制机制，保存着系统的输入、问题求解各个阶段的中间结果和反映整体问题求解进程的状态。这些是由系统的输入和各个求解程序“写”入的，因此被称为“黑板”。

系统在运行中，每当有新输入、新结果和新状态写入黑板时，中心控制部件就对黑板上的信息进行评价，并据此协调各专门程序进行工作，试探性地调用各个可能的求解算法，并根据试探导出的启发信息控制后续的处理。

在问题求解过程中，黑板上保存了所有的部分解，代表了问题求解的不同阶段，形成了问题可能的解空间，并以不同的抽象层次表达出来。其中，最底层的表达就是系统的原始输入。最终的问题解在抽象的最高层次。

“黑板”模式类似于这样一个情形，即让专家们坐在真实黑板前并一起工作，来解决一个问题。每个专家独立评估解法的当前状态，并可在任何时间到黑板上添加、更改、删除信息。人们往往要决定接下来谁去访问黑板。在黑板模式中，如果可用的组件超过一个，则仲裁者(Moderator)组件决定程序执行的顺序。

黑板风格的体系结构如图 5-6 所示。

不难看出，一个标准的黑板型仓库模式系统通常包括以下三个组成部分。

(1) 知识源。基于仓库模式的系统完全是依靠仓库状态的变化来驱动的，那么，仓库的建立，即知识的来源是系统设计时首先需要解决的问题。图 5-6 中 KS(Knowdge Source)表示知识源，是仓库中信息的来源。它们彼此之间在逻辑上和物理上都是独立的，只与产生它们的应用程序有关。多个数据源之间通过中央数据单元协调进行交互，对外部而言是透明的。

(2) 中央数据单元。它是整个系统的核心部件，对系统需要解决的问题预先进行了分析、定义，总结出了系统运行过程中将要出现的多种状态，并制定了这些状态下系统的相应

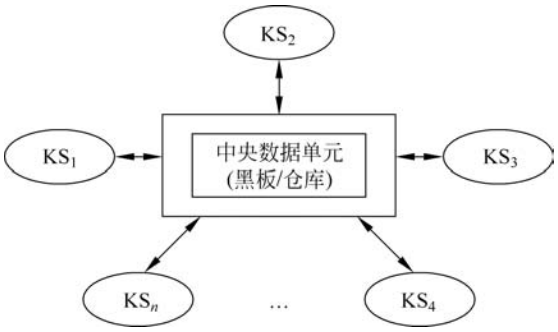

图 5-6　黑板风格的体系结构

对策。所以,中央数据单元中的数据不只是单纯的数据信息,还代表了某种系统的状态,属于状态数据。这些数据由数据源提供,在中央数据单元中依据一定的数据结构形式组织在一起,并随着数据源信息的改变而变化,从而实现系统的功能。

(3) 控制单元。控制单元的驱动完全是由仓库的状态变化承担的。知识源将系统需要处理的信息源源不断地输入仓库中,导致仓库的状态信息发生变化。当状态信息的变化符合系统预先定义好的某些控制策略时,相应的控制操作就得到了触发,也就实现了系统的功能控制。从图 5-6 中无法看到控制单元的明显表示,因为控制单元在基于仓库模式的系统中并不一定是独立的单元,可以位于知识源和仓库中,或者作为一个独立部分单独存在,没有绝对的定式,需要设计者根据系统的实际情况做出抉择。

黑板风格体系结构与传统体系结构有显著区别。它追求的是可能随时间变化的目标,各个代理需要不同资源、关心不同问题,但用一种相互协作的方式使用与维护共享数据结构。

黑板风格体系结构有以下优点。

(1) 便于多客户共享大量数据,不用关心数据是何时出现的、谁提供的、怎样提供的。

(2) 既便于添加新的作为知识源代理的应用程序,也便于扩展共享的黑板数据结构。

(3) 可重用的知识源。知识源是某类任务的独立专家,黑板体系结构有助于使它们可重用。重用的先决条件,是知识源和所基于的黑板系统理解相同的协议和数据,或者在这方面相当接近而不排斥协议或数据的自适应程序。

(4) 支持容错性和健壮性。黑板体系结构中,所有的结果都只是假设。只有那些被数据和其他假设强烈支持的才能生存,这提供了对噪声数据和不确定结论的容忍。

黑板风格体系结构有以下缺点。

(1) 不同的知识源代理,对于共享数据结构要达成一致,而且,这也造成对黑板数据结构的修改较为困难。

(2) 需要一定的同步锁机制,保证数据结构的完整性和一致性,增大了系统复杂度。

(3) 测试困难。由于黑板系统的计算没有依据一个确定的算法,所以,其结果常常不可再现。此外,错误假设也是求解过程的一部分。

(4) 不能保证有好的求解方案。一个黑板系统往往只能正确解决所给任务的某一百分比。难以建立一个好的控制策略,控制策略不能以一种直接方式设计,而需要一种试验的方法。

(5) 低效。黑板系统在拒绝错误假设中要承受多余的计算开销。但是,如果没有确定

的算法存在，那么与根本不存在的系统相比，低效总比没有强。

（6）开发成本高。绝大多数黑板系统要花几年时间来进化，我们把这归因于病态结构问题。

5.2.6 模型-视图-控制器风格

1. MVC模式

模型-视图-控制器（Model-View-Controller，MVC）由挪威奥斯陆大学教授 Trygve Reenskaug 提出，首先被应用在 SmallTalk-80 环境中，是许多交互和界面系统的构成基础。

MVC 结构是为那些需要为同样的数据提供多个视图的应用程序而设计的，很好地实现了数据层与表示层的分离。作为一种开发模型，MVC 通常用于分布式应用系统的设计和分析中，以及用于确定系统各部分间的组织关系。对于界面设计可变性的需求，MVC 把交互系统的组成分解成模型、视图、控制器三种部件。

MVC 是许多交互和界面系统的构成基础，微软的 MFC 基础类也遵循了 MVC 的思想。作为一种软件设计典范，MVC 用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑，其中：

（1）模型部件是软件所处理问题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、功能的计算关系，独立于具体的界面表达和 I/O 操作。

（2）视图部件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，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。

（3）控制部件是处理用户与软件的交互操作的，职责是控制提供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。它接收用户的输入，并将输入反馈给模型，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。通常一个视图具有一个控制器。

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，则所有其他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。

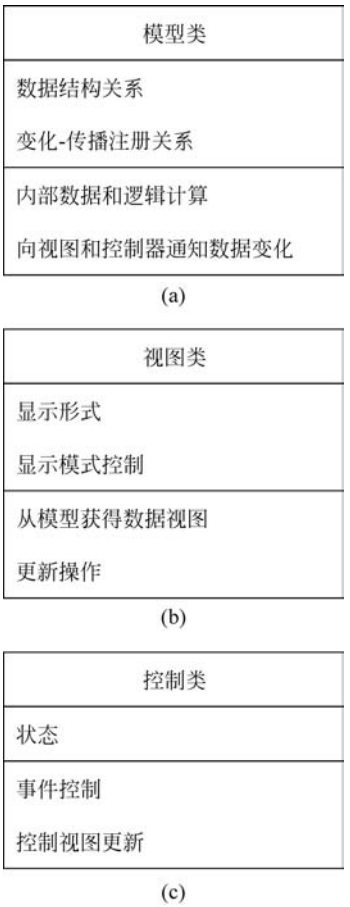

图 5-7 MVC 中的模型、视图和控制类

2. 模型、视图和控制类

MVC 中的模型、视图和控制类如图 5-7 所示。

1）模型类

模型包含了应用问题的核心数据、逻辑关系、计算功能，封装了所需的数据，提供了完成问题处理的操作过程。控制器依据 I/O 的需要调用这些操作过程。模型还为视图获取显示数据而提供了访问其数据的操作。这种变化-传播机制体现在各个相互依赖部件之间的注册关系上。模型数据和状态的变化会激发这种变化-传播机制，是模型、视图和控制器之间联系的纽带。

2）视图类

视图通过显示的形式把信息转达给用户。不同视图通过不同的显示来表达模型的数据和状态信息。每个视图有一个更新操作，可被变化-传播机制所激活。当调用更新操作时，视图获得来自模型的数据值，以更新显示。在初始化时，通过与变化-传播机制的注册关系建立起所有视图与模型间的关联。视图与控制器之间保持着一对一的关系，每个视图创建一个相应的控制器。视图提供给控制器处理显示的操作，因此，控制器可以获得主动激发界面更新的能力。

3）控制类

控制器通过时间触发的方式接收用户的输入。控制器如何获得事件，依赖于界面的运行平台。控制器通过事件处理过程对输入事件进行处理，并为每个输入事件提供相应的操作服务，把事件转化成对模型或相关视图的激发操作。

如果控制器的行为依赖于模型的状态，则控制器应该在变化-传播机制中进行注册，并提供一个更新操作，这样可以由模型的变化来改变控制器的行为，如禁止某些操作。

3. MVC 的实现

实现基于 MVC 的应用需要完成以下工作，如图 5-8 所示。

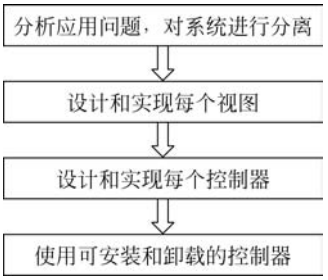

图 5-8　MVC 的实现

1）分析应用问题，对系统进行分离

分析应用问题，分离出系统的内核功能、对功能的控制输入、系统的输出行为三大部分。设计模型部件使其封装内核数据和计算功能，提供访问显示数据的操作，提供控制内部行为的操作以及其他必要的操作接口。这部分的构成与具体的应用问题紧密相关。

2）设计和实现每个视图

设计每个视图的显示形式，从模型中获取数据，显示在屏幕上。

3）设计和实现每个控制器

对于每个视图，指定对用户操作的响应时间、行为。在模型状态的影响下，控制器使用特定的方法接收和解释这些事件。控制器的初始化建立起与模型和视图的联系，并且启动事件处理机制。事件处理机制的具体实现方法依赖于界面的工作平台。

4）使用可安装和卸载的控制器

控制器的可安装性和可卸载性带来了更高的自由度，并且帮助形成高度灵活性的应用。控制器与视图的分离支持了视图与不同控制器结合的灵活性，以实现不同的操作模式，例如，对普通用户、专业用户或不使用控制器建立的只读视图。这种分离还为在应用中集成新的 I/O 设备提供了途径。

5.2.7 解释器风格

解释器风格通常被用于建立一种虚拟机，以弥合程序的语义与作为计算引擎的硬件的间隙。由于解释器实际上创建了一个软件虚拟出来的机器，所以，这种风格又常常被称为虚拟机风格。例如，程序设计语言的编译器，如 Java、Smalltalk 等；基于规则的系统，如专家系统领域的 Prolog 等；脚本语言，如 Awk、Perl 等。

解释器风格的系统通常包括一个作为执行引擎的状态机和三个存储器，即系统由四个构件组成，如图 5-9 所示。连接件包括过程调用和直接存储器访问。

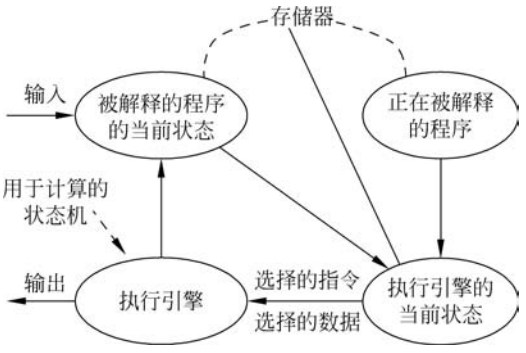

图 5-9 解释器风格的体系结构

解释器风格适用于这样的应用程序，即应用程序并不能直接运行在最合适的机器上，或者不能直接以最适合的语言执行。解释器风格的优点如下。

(1) 有助于应用程序的可移植性与程序设计语言的跨平台能力。

(2) 可以对未实现的硬件进行仿真。

解释器风格的缺点是额外的间接层次带来的系统性能的下降。

5.2.8 C2 风格

C2 体系结构风格可以概括为：通过连接件绑定在一起的、按照一组规则运作的并行构件网络。C2 风格中的系统组织规划如下。

(1) 系统中的构件与连接件都有一个顶部和一个底部。

(2) 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，构件与构件之间的直接连接是不允许的。

(3) 一个连接件可以与任意数目的其他构件和连接件连接。

(4) 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

图 5-10 所示为 C2 风格的体系结构，构件与连接件之间的连接体现了 C2 风格中构件系统的规划。

C2 风格是最常用的一种软件体系结构风格。从 C2 风格的组织规则和结构图中可以得出，C2 风格具有以下特点。

(1) 系统中的构件可实现应用需求，并能将任意复杂度的功能封装在一起。

(2) 所有构件之间的通信是通过以连接件为中介的异步消息交换机制来实现的。

(3) 构件相对独立，构件之间依赖性较少。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

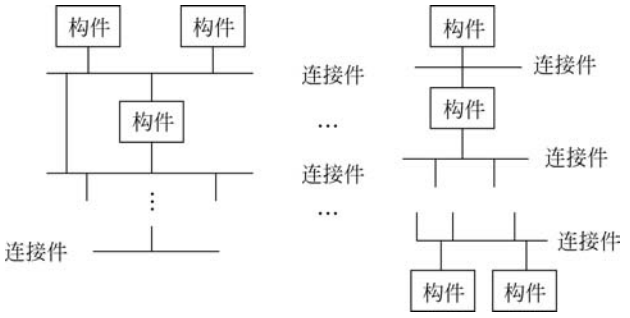

图 5-10　C2 风格的体系结构

5.3　案 例 分 析

下面通过两个案例来阐明怎样使用体系结构原则,以增强对软件系统的理解。第一个案例体现了对同一个问题使用不同的体系结构解决方案带来的不同好处。第二个案例总结了为工业产品族开发特定领域的体系结构风格的经验。

5.3.1　案例 1:上下文关键字

David Parnas 在 1972 年的论文中提出下述问题,上下文关键字(Key Word in Context,KWIC)检索系统接受有序的行集合;每一行是单词的有序集合;每一个单词又是字母的有序集合。通过重复地删除行中第一个单词并把它插入到行尾,每一行可以被"循环地移动"。KWIC 检索系统以字母表的顺序输出一个所有行循环移动的列表。

Pamas 通过这个问题来对系统模块化的不同策略进行对比,描述了两种解决方案:一种是基于功能分解,可以共享访问数据表示;另一种是基于隐藏设计决策的分解。这个问题一经提出就得到了人们的关注,并广泛地作为软件工程教学案例。

虽然 KWIC 可以作为一个相对小的系统被设计实现,但它不仅仅是一个简单的教学案例。在实践中,它的实例被计算机科学家广泛地使用。例如,UNIX 帮助页"改变序列"的索引,基本上就是这样的一个系统。

从软件体系结构的观点来看,这个问题非常吸引人,因为我们可以通过它来阐明软件设计变更的影响。Pamas 指出,不同的问题分解策略适应设计变更的能力有很大不同。他认为的变更有以下两种。

(1) 处理算法的变更:例如,输入设备可以每读入一行就执行一次行移动,也可以读完所有行再执行行移动,或者在需要以字母表的顺序排列行集合时才执行行移动。

(2) 数据表示的变更:例如,行、单词、字母可以用各种各样的方式储存。类似地,循环移动情况也可以被显式地或者隐式地储存(使用索引和偏移量)。

后来,Garlan、Kaiser、Notkin 也通过 KWIC 的问题来阐述基于隐式调用的模块化策略。为了做到这一点,他们扩展了 Pamas 的分析并考虑以下几个方面。

(1) 系统功能的扩充:例如,修改系统使其能够排除以某些干扰单词(如 a、an 等)开头的循环移动。把系统变成交互式的,允许用户从初始列表中(或者从循环移动的列表中)删

除某些行。

(2) 性能：空间和时间。

(3) 重用：作为可重用的实体，构件重用的程度。

下面简略论述针对 KWIC 系统的四种体系结构设计方案。这四种方案是所有方案(包括实现)中最基础的。前两个方案是在 Parnas 最初的论文中提出中。在 Garlan、Kaiser、Notkin 提出的解决方案中，使用了隐式调用风格，并且描述了解决方案的变化。第四种解决方案使用管道模式，灵感来自 UNIX 检索程序。

在介绍完每种解决方案及其优缺点后，将通过一张表从五个方面比较不同的体系结构分解策略。

1. 解决方案 1：使用共享数据的主程序/子程序

第一种解决方案根据四个基本功能将问题分解为输入、循环移动、按字母表排序、输出。所有计算构件作为子程序协同工作，并且由一个主程序顺序地调用这些子程序。构件通过共享存储区(核心存储区)交换数据。因为协同工作的子程序能够保证共享数据的顺序访问，因此，使得计算构件与共享数据之间基于一个不受约束的读写协议的通信成为可能，如图 5-11 所示。

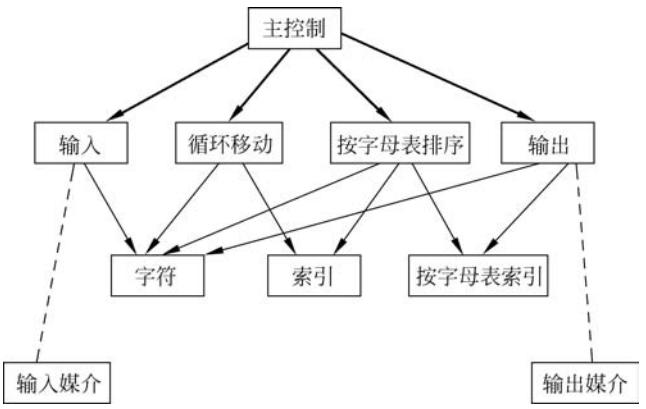

图 5-11　KWIC：共享数据解决方案

这种方案具有很高的数据访问效率，因为计算共享同一个存储区。其另一个显而易见的好处是，不同的计算功能被划分到不同的模块中。

然而，正如 Pamas 所说，这种方案在处理变更的能力上有许多严重的缺陷。特别是，数据存储格式的变化会影响到所有模块，同样，整体处理算法的变更与系统功能扩充的问题也很难调和。最后，这种分解方案并不支持重用。

2. 解决方案 2：抽象数据类型

第二种解决方案将系统分解成五个模块。然而在这种情况下，数据不再直接地被计算构件共享。取而代之的是，每个模块提供一个接口，该接口允许其他构件通过调用接口中的过程来访问数据。如图 5-12 所示，每个构件提供了一个过程集合，这些过程决定了系统中其他构件访问该构件的形式。

这种解决方案与第一种方案一样，将系统在逻辑上分成几个处理模块。然而，当设计变更时，这种方案比第一种具有一些优势。特别是在一个独立的模块中，算法和数据表示的改

变不会影响其他模块。另外,它还为重用提供了更好的支持,因为模块几乎不需要考虑与其交互的其他模块的情况。

另一方面,正如 Garlan、Kaiser、Notkin 所述,这种解决方案并不能很好地适用于功能扩展的情况。其主要的问题是,向系统中加入一个新的功能时,实现者要么平衡其简明性和完整性而修改现存模块,要么添加新的模块而导致性能下降。

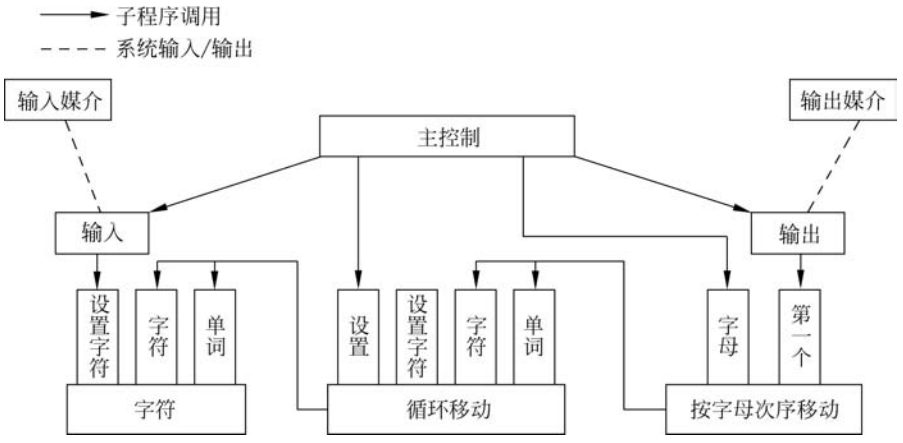

图 5-12　KWIC：抽象数据类型解决方案

3. 解决方案 3：隐式调用

第三种方案采用基于共享数据的构件集成的方式,这和第一种方案有些相似,如图 5-13 所示。然而,这里有两个主要的不同之处。第一,数据访问接口更加抽象,这种方案可以抽象地访问数据(比如作为表或集合),而不需要知道数据的存储格式;第二,当数据被修改时,计算被隐式地调用,因而交互是基于“动态数据”模型的。例如,向行存储区添加一个新行的动作,会激发一个事件,这个事件被发送到移动模块;然后,移动模块进行循环移动(在一个独立的、抽象的共享数据存储区),又会引起字母表排序程序被隐式地调用,字母表排序程序再对行进行排序。

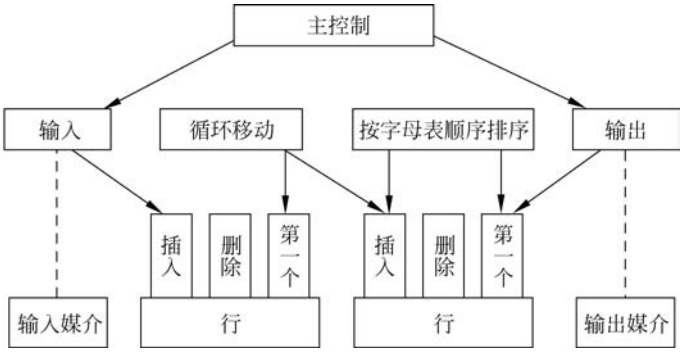

图 5-13　KWIC：隐式调用解决方案

这种解决方案很容易支持系统功能的扩展,通过注册添加的模块很容易和系统整合,当发生数据交换事件时,这些添加的模块就会被调用。数据被抽象地访问,所以这种解决方案也将计算和数据表示分开。由于被隐式调用的模块仅仅依赖于某些外部的触发事件,所以

这种方案也支持重用。

然而，这种解决方案的缺点是难以控制隐式调用模块的处理顺序。另外，由于隐式调用是数据驱动的，这种分解策略实现起来会比之前讨论的方案占用更大的空间。

4. 解决方案 4：管道-过滤器

第四种解决方案采用管道的方式。这种情况下有四个过滤器：输入、循环移动、按字母表顺序排序、输出。每一个过滤器处理完数据并发送到下一个过滤器。控制是分布式的，只要有数据通过，过滤器就会进行处理。过滤器之间的数据共享严格地局限于管道中传输的数据，如图 5-14 所示。

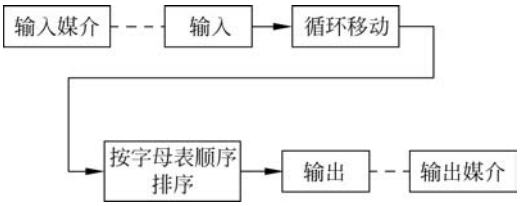

图 5-14　KWIC：管道-过滤器方案

这种方案有几个好的特性：第一，能够维持处理的自然流动；第二，支持重用，因为每个过滤器可以独立处理（只要上游过滤器产生的数据格式是过滤器所期望的格式），通过在处理序列中的合适位置插入过滤器，新的功能很容易加入系统中；第三，既然每个过滤器与其他过滤器在逻辑上是独立的，那么每个过滤器也很容易替换或修改。

另一方面，这种解决方案也有一些缺点：第一，不可能通过修改使其支持交互，例如，删除一行，可能需要一些持久性共享存储区，但是这样就违反了这种方案的基本原则；第二，这种方案空间使用的效率非常低，因为每个过滤器必须复制所有的数据到输出端口。

5. 各种方案的比较

通过表 5-1 对这些方案进行比较，表中列出了各种解决方案处理设计考虑因素的能力。对于更详细的比较，还必须考虑一些与这个系统相关的因素，如用于批处理还是交互式、更新频繁还是查询频繁等。

表 5-1 提供了对于设计考虑因素的分析近似值，这些近似值基于之前对于这些体系结构风格的讨论。正如 Pamas 指出，共享数据方案对整体处理算法、数据表示的变更及重用的支持非常弱。另一方面，由于对数据的直接共享，获得了相对好的性能。另外，也相对比较容易加入新的处理构件（同样通过访问共享数据）。抽象数据类型方案在保证性能的情况下，允许数据表示变更并且支持重用。但是，由于构件的交互依赖于模块本身，所以，改变整体处理算法或者加入新的功能可能要对现有系统做很大修改。

表 5-1　KWIC：各种方案的比较

项　　目	共享数据	抽象数据类型	隐式调用	管道-过滤器
算法变更	−	−	+	+
数据表示变更	−	+	−	−
功能变更	+	−	+	+
性能	+	+	−	−
重用	+	+	−	+

隐式调用方案对于添加新功能的支持非常好。然而,由于共享数据自身的一些问题,该方案对于数据表示和重用的支持非常弱。另外,还可能引起额外的执行开销。管道-过滤器方案允许在文本处理流中放置新的过滤器,因此支持处理算法的改变、功能的变化和重用。另一方面,数据表示的选择过分地依赖于管道中传输的数据类型的假定。而且,由于需要数据转换,对管道中数据进行编码和解码也会造成额外的开销。

5.3.2 案例 2:仪器软件

第二个案例描述了泰克公司的一个软件体系结构的工业发展。这项工程历时三年,是由泰克公司的几个产品部门和计算机研究实验室合作展开的。

这项工程的目的是为示波器开发一种可重用的系统体系结构。示波器是一个仪器系统,能对电信号取样,并且在屏幕上显示电信号的图像(踪迹)。示波器通常对信号进行测量,并显示在屏幕上。尽管示波器曾经是一个简单的模拟设备,几乎不需要软件,但是现代示波器主要依靠数字技术,并且使用非常复杂的软件。

现代示波器能够完成大量的测量,提供兆字节的内存,支持工作站网络和其他仪器的接口,并能提供完善的用户界面,包括带有菜单的触摸屏、内置的帮助工具和彩色显示,能提供这么复杂的功能并不令人奇怪。

像很多公司越来越依赖软件对他们产品的支持一样,泰克公司也面临了一系列问题。首先,几乎没有在不同示波器上可重用的软件组织结构。确实,不同的示波器由不同的产品部门生产,每个部门都有自己的开发约定、软件组织结构、编程语言、开发工具。另外,甚至在一个产品部门内,每一个新的示波器通常也不得不重新设计来适应硬件性能的变化和用户界面的新需求。而硬件和界面需求变化速度越来越快也加剧了这种情况。此外,开发“特殊的市场”的需要,意味着必须为特殊的用户量身定做多种用途的仪器,如病人监护或汽车诊断。

其次,性能问题越来越严重,因为软件在仪器中不能被快速配置。这些问题的出现,是由于根据用户的任务需要,示波器需要在不同的模式下配置。以前的示波器只需要简单地载入处理新模式的软件就能重新配置。软件的规模越来越大,这导致了在用户的请求和仪器重新配置之间出现延迟。

这项工程的目标是为示波器开发一种解决上述问题的体系结构框架。工程的结果产生了一个特定领域的软件体系结构,这种体系结构将是下一代泰克示波器的基础。之后,这个框架被扩展、修改,以适应更广泛的系统种类,同时也是为了适应仪器软件的特殊需要。下面将简述这种软件体系结构发展的各个阶段。

1. 面向对象模型

开发一个可重用体系结构的最初尝试,是开发一种面向对象的软件模型,这个模型阐明了在示波器中使用的对象类型:波形、信号、测量值、触发模型等,如图 5-15 所示。虽然这是一个有益的实践,但是由于没有产生出期望的结果而失败了。尽管很多对象类型被确定,但是没有一个整体模型解释怎样结合这些对象类型,这会导致功能划分的混乱。例如,量度是否应该与被测

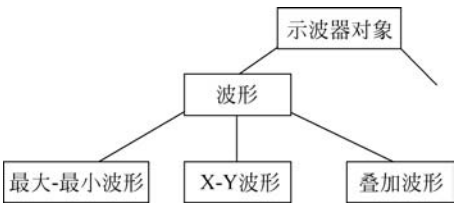

图 5-15 示波器:面向对象模型

量的或者被外部表示的数据类型相关联？用户界面应该和哪些对象交互？

2. 分层模型

第二阶段尝试解决这些问题，这一阶段提出了示波器的一个分层模型，如图 5-16 所示。在模型中，核心层提供信号处理功能，当信号进入示波器时，使用这些功能过滤信号。这些功能通常通过硬件实现。第二层提供波形采集功能，在这层中信号被数字化，并且被内部保存用于以后的处理。第三层提供波形处理功能，包括测量、波形叠加、傅里叶转换。第四层提供显示功能，即负责将数字化的波形与测量值直观表示出来。最外层是用户界面，这一层负责与用户进行交互，并决定在屏幕上显示哪些数据。

图 5-16 示波器：分层模型

这种分层模型将示波器的功能分成一些明确定义的组，所以具有显而易见的吸引力。遗憾的是，对于应用领域，这种模型是错误的。其主要问题是层次间强加的抽象边界和各功能间交互的需要是相互冲突的。例如，这种模型提出所有用户与示波器交互必须通过显示层。但是在实践中，真正的示波器用户需要直接与各层打交道，例如在信号处理层中设置衰减，在采集层中选择采集模式和参数，或者在波形处理层中制作导出波形。

3. 管道-过滤器模型

第三种尝试产生了一个模型，在这个模型中，示波器功能被看成是数据的增量转换器。信号转换器用来检测外部信号，采集转换器用来从这些信号中导出数字化波形，显示转换器再将这些波形转换成可显示的数据，如图 5-17 所示。

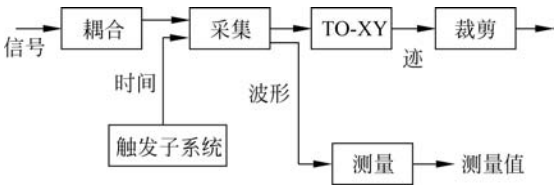

图 5-17 示波器：管道-过滤器模型

这种体系结构模型和分层模型相比有很大改进，因为没有在功能划分中将各个功能孤立起来。例如，信号数据可以直接流入显示过滤器，而不会被其他过滤器干扰。另外，从工程师的角度来看，模型将信号处理作为数据流问题是比较合适的，模型也允许在系统设计中硬件与软件构件的灵活混合和替换。

这种模型的主要问题是没有清晰地说明用户怎样与其交互。如果用户仅仅是站在屏幕前等待结果，那么，这种模型的分解策略甚至比分层系统还糟糕。

4. 改进后的管道-过滤器模型

第四种解决方案解决了用户输入问题，即为每个过滤器添加一个控制界面，这个界面允

许外部实体为过滤器设置操作参数。例如,采集过滤器具有某些参数,用来确定采样频率和波幅。这些输入可以作为示波器的配置参数。可以将这些过滤器想象成一个具有“控制面板”的界面,通过它可以控制在输入/输出界面上将要执行哪些功能。在形式上,过滤器可以被模拟成函数,配置参数决定了过滤器将执行什么数据转换。图 5-18 显示了这个体系结构。

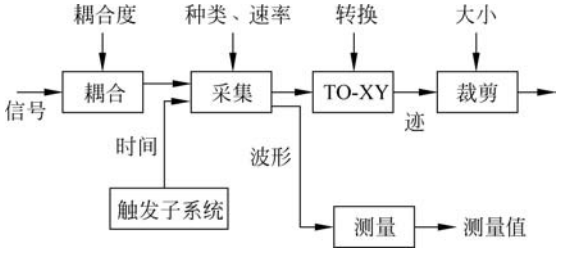

图 5-18　示波器：改进后的管道-过滤器模型

控制界面的引入解决了很大一部分用户界面的问题。第一,它提供了一系列设置,这些设置决定了示波器的哪些方面可以被用户动态地修改,同时也解释了用户如何通过不断地调整软件来改变示波器的功能。第二,控制界面将示波器的信号处理功能和实际的用户界面分离,信号处理软件不需要考虑用户实际设置控制参数的方式。相反地,实际的用户界面仅仅通过控制参数来控制信号处理功能。这样,设计者可以在不影响用户界面实现的情况下更改信号处理软件和硬件的实现(假设控制界面不变)。

5. 专用化模型

改进后的管道-过滤器模型虽然有了很大进步,但是同样还存在一些问题。最显著的问题是管道-过滤器计算模式的性能非常差,特别是波形数据占用了很大的内存容量,过滤器每次处理波形时都复制波形数据是不切实际的。另外,不同的过滤器以完全不同的速度运行,由于其他过滤器仍然在处理数据,所以不会降低单个过滤器的处理速度。

为了解决这些问题,需要将模型进一步地专用化。引进多种“颜色”的管道,而不只是使用一种管道。一些管道允许某些过滤器在处理数据时不必复制数据;另一些管道允许慢速过滤器在数据没有处理完时忽略新来的数据。这些附加的管道增加了风格词汇表,并且可以根据产品的性能定制管道-过滤器的计算模式。

5.4　客户/服务器风格

客户/服务器(Client/Server,C/S)计算技术在信息产业中占有重要的地位。网络计算经历了从基于宿主机的计算模型到客户/服务器计算模型的演变。

在集中式计算技术时代广泛使用的是大型机/小型机计算模型,通过一台物理上与宿主机相连接的非智能终端来实现宿主机上的应用程序,如图 5-19 所示。多用户环境中,宿主机应用程序既负责与用户的交互,又负责对数据的管理。宿主机上的应用程序一般也分为与用户交互的前端和管理数据的后端,即数据库管理系统(Database Management System,DBMS)。集中式的系统使用户能共享贵重的硬件设备,如磁盘机、打印机和调制解调器等。但随着用户的增多,对宿主机能力的要求提高,而且开发者必须为每个新的应用重新设计同

样的数据管理构件。

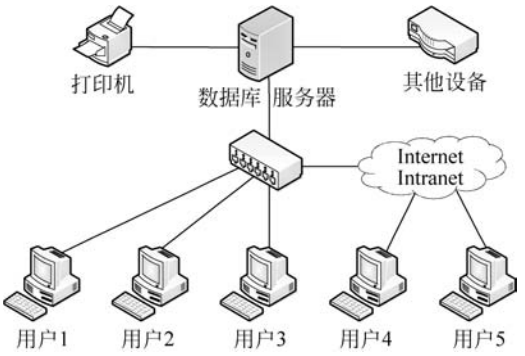

图 5-19　C/S 体系结构示意图

20 世纪 80 年代以后，集中式结构逐渐被以 PC 为主的微型计算机网络所取代。个人计算机和工作站的采用永远改变了协作计算模型，从而导致了分散的个人计算模型的产生。一方面，由于大型机系统固有的缺陷（如缺乏灵活性），无法适应信息量急剧增长的需求，并为整个企业提供全面的解决方案等；另一方面，由于微处理器的日新月异，其强大的处理能力和低廉的价格使微型计算机网络迅速发展，已不仅仅是简单的个人系统，这便形成了计算机界的向下规模化。其主要优点是，用户可以选择适合自己需要的工作站、操作系统和应用程序。

C/S 体系结构是基于资源不对等，且为实现共享而提出来的，是 20 世纪 90 年代成熟起来的技术，它定义了工作站如何与服务器相连，以实现数据和应用分布到多个处理机上。C/S 体系结构有三个主要组成部分：数据库服务器、客户应用程序和网络。

服务器负责有效地管理系统的资源，其任务集中于以下几个方面。

（1）数据库安全性的要求。

（2）数据库访问并发性的控制。

（3）数据库前端的客户应用程序的全局数据完整性规则。

（4）数据库的备份与恢复。

客户端应用程序的主要任务如下。

（1）提供用户与数据库交互的界面。

（2）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。

（3）利用客户端应用程序，对存在于客户端的数据执行应用逻辑要求。

网络通信软件的主要作用是完成数据库服务器和客户端应用程序之间的数据传输。

在一个 C/S 体系结构的软件系统中，客户端应用程序是针对一个小的、特定的数据集，如对一个表的行来进行操作，而不是像文件服务器那样针对整个文件进行，是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。

C/S 体系结构的优点主要在于，系统的客户端应用程序和服务器构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合各构件的要求，这对于硬件与软件的变化显示出极大的适应性、灵活性，而且易于对系统进行扩充和缩小。C/S 体系结构中，系统中的功能

件充分隔离,客户端应用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程序中都要对一个 DBMS 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,以节约大量费用。

C/S 体系结构具有强大的数据操作和事务处理能力,模型思想简单,易于人们理解和接受,但随着企业规模的日益扩大,软件的复杂程度不断提高,C/S 体系结构逐渐暴露了以下缺点。

(1) 开发成本较高。C/S 体系结构对客户端软件与硬件配置要求较高,尤其是软件的不断升级,对硬件要求不断提高,增加了整个系统的成本,客户端变得越来越臃肿。

(2) 客户端程序设计复杂。采用 C/S 体系结构进行软件开发,大部分工作量放在客户端的程序设计上,客户端显得十分庞大。

(3) 信息内容和形式单一,因为传统应用一般为事务处理,界面基本遵循数据库的字段解释,在开发之初就已确定,而且不能随时截取办公信息与档案等外部信息,用户获得的只是单纯的字符和数字,既枯燥又死板。

(4) 用户界面风格不一,使用复杂,不利于推广使用。

(5) 软件移植困难。采用不同开发工具或平台开发的软件一般互不兼容,不能或很难移植到其他平台上运行。

(6) 软件维护与升级困难。采用 C/S 体系结构的软件要升级,开发人员必须到现场为客户端升级,每个客户端上的软件都需要维护。对软件的一个小小改动(如只改动一个变量),每一个客户端都必须更新。

(7) 新技术不能轻易应用。因为一个软件平台及开发工具一旦选定,不可能轻易更改。

5.5 三层 C/S 体系结构风格

传统的二层 C/S 体系结构存在以下几个局限。

(1) 二层 C/S 体系结构是单一服务器且以局域网为中心,所以难以扩展至大型企业广域网或互联网。

(2) 软件与硬件的组合及集成能力有限。

(3) 客户机的负荷太重,难以管理大量的客户机,系统的性能容易变差。

(4) 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,所以,在客户端计算机上的其他程序也可想办法访问数据库服务器,从而使数据库的安全性受到威胁。

正是因为二层 C/S 体系结构有以上缺点,所以三层 C/S 体系结构应运而生,其结构如图 5-20 所示。

与二层 C/S 体系结构相比,在三层 C/S 体系结构中增加了一个应用服务器,可以将整个应用逻辑驻留在应用服务器上,而只有表示层存在于客户机上。这种结构被称为“瘦客户机”。三层 C/S 体系结构将应用功能分成表示层、功能层和数据层三个部分,如图 5-21 所示。

1. 表示层

表示层是应用的用户接口部分,担负着用户与应用之间的对话功能,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

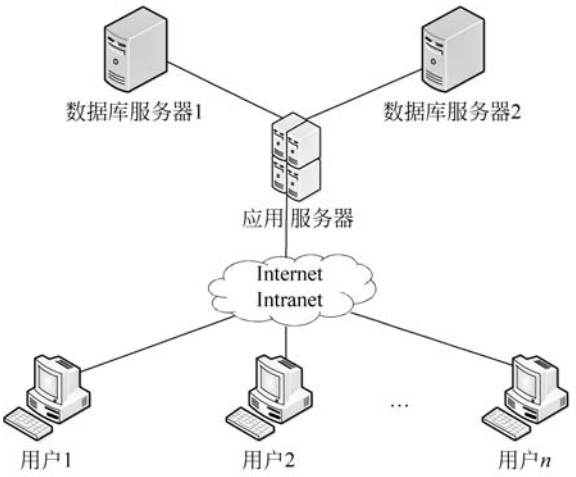

图 5-20　三层 C/S 体系结构示意图

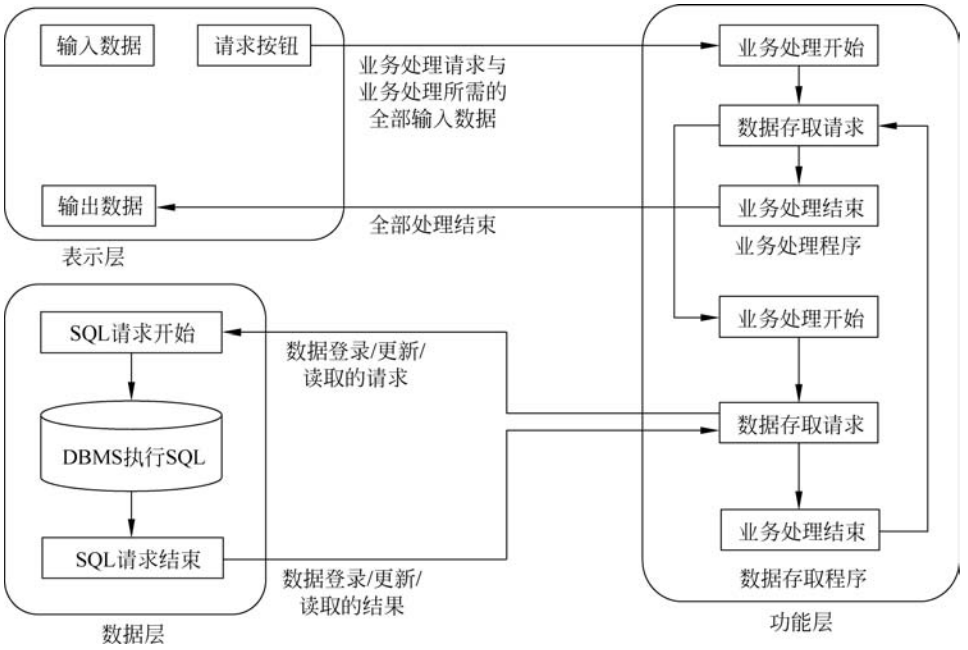

图 5-21　三层 C/S 体系结构的一般处理流程

界面(Graphic User Interface,GUI),其操作简单、易学易用。在变更用户界面时,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取值的范围,不包括有关业务本身的处理逻辑。

2. 功能层

功能层相当于应用的本体,用于将具体的业务处理逻辑编入程序。例如,在制作订购合同时要计算合同金额,按照定好的格式配置数据、打印订购合同,而处理所需的数据则要从表示层、数据层取得。表示层与功能层之间的数据交往要尽可能简洁。例如,用户检索数据时,要设法将有关检索要求的信息一次性地传送给功能层,而由功能层处理过的检索结果数

据也应一次性地传送给表示层。

通常,在功能层中包含确认用户对应用与数据库存取权限的功能以及记录系统处理日志的功能。功能层的程序多半是用可视化编程工具开发的,也有使用 COBOL 和 C 语言的。

3. 数据层

数据层就是数据库管理系统,负责管理对数据库数据的读写。数据库管理系统必须能迅速执行大量数据的更新和检索。现在的主流是关系型数据库管理系统(Relational Database Management System,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。

三层 C/S 体系结构的解决方案是对这三层进行明确分割,并在逻辑上使其独立。原来的数据层作为数据库管理系统已经独立出来,所以,关键是要将表示层与功能层分离成各自独立的程序,并且还要使这两层间的接口简洁明了。

一般情况是只将表示层配置在客户机中,如图 5-22 所示。如果像图 5-22(c)所示的那样,连功能层也放在客户机中,则与二层 C/S 体系结构相比,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差。

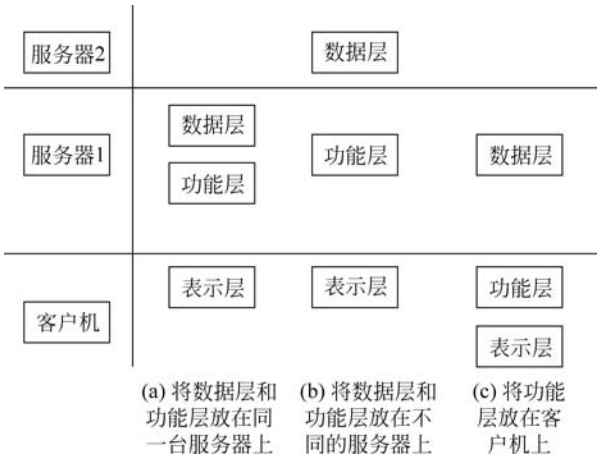

图 5-22　三层 C/S 物理结构比较

如果将功能层和数据层分别放在不同的服务器中,如图 5-22(b)所示,则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,因此灵活性很高,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,这种形态的优势就越显著。

三层 C/S 体系结构中,中间件是最重要的构件。中间件(Middleware)是一个用 API 定义的软件层,是具有强大通信能力与良好可扩展性的分布式软件管理框架。中间件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驻留网络上某个服务器的数据或服务时,搜索此数据的 C/S 应用程序需访问中间件系统,该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。

5.5.1 三层 C/S 体系结构的优点

可以看出，与传统的二层 C/S 体系结构相比，三层 C/S 体系结构具有以下优点。

(1) 允许合理地划分三层结构的功能，使之在逻辑上保持相对独立性，从而使整个系统的逻辑结构更为清晰，能提高系统与软件的可维护性和可扩展性。

(2) 允许更灵活有效地选用相应的平台与硬件系统，使之在处理负荷能力上与处理特性上分别适应于结构清晰的三层，并且这些平台与各个组成部分可以具有良好的可升级性与开放性。例如，最初用一台 UNIX 工作站作为服务器，将数据层、功能层都配置在这台服务器上。随着业务的发展，用户数和数据量逐渐增加，这时就可以将 UNIX 工作站作为功能层的专用服务器，另外追加一台专用于数据层的服务器。若业务进一步扩大，用户数进一步增加，则可以继续增加功能层的服务器数目，用以分割数据库。清晰、合理地分割三层结构并使其独立，可以使系统构成的变更非常简单。因此，被分成三层的应用基本上不需要修正。

(3) 三层 C/S 体系结构中，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，使之能并行地而且是高效率地进行开发，达到较高的性能价格比，对每一层的处理逻辑的开发和维护也会更容易些。

(4) 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去非法地访问数据层，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。

值得注意的是，三层 C/S 体系结构各层间的通信效率若不高，则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 体系结构的关键问题。

5.5.2 实例：某石油管理局劳动管理信息系统

本节以某石油管理局劳动管理信息系统的设计与开发为例，介绍三层 C/S 体系结构的应用。

1. 系统背景介绍

该石油管理局是国有特大型企业，其劳动管理信息系统(Management information System，MIS)具有较强的特点，具体如下。

(1) 信息量大：须存储并维护全油田近 20 万名职工的基本信息和其他各种管理信息。

(2) 单位多、分布广：系统涵盖 70 多个单位，分布范围 8 万多 km^2。

(3) 用户类型多、数量大：劳动管理工作涉及管理局(一级)、厂矿(二级)、基层大队(三级)等三级层次，各层次的业务职责不同，各层次领导对系统的查询功能的要求和权限也不同，系统用户总数达 700 多个。

(4) 网络环境不断发展：70 多个二级单位中有 40 多个联入广域网，其他二级单位只有局域网，而绝大部分三级单位只有单机，需要陆续接入广域网。

项目要求系统应具备较强的适应能力和进化能力，不论单机还是网络环境均能运行，并

保证数据的一致性,且能随着网络环境的改善与管理水平的提高平稳地从单机方式向网络方式、从集中式数据库方式向分布式数据库方式,以及从独立的应用程序方式向适应Intranet环境的方式进化。

2. 系统分析与设计

三层C/S体系结构运用事务分离的原则将MIS应用分为表示层、功能层、数据层三个层次,每一层次都有自己的特点,例如,表示层是图形化的、事件驱动的,功能层是过程化的,数据层则是结构化与非过程化的,传统的结构化分析与设计技术难以统一表达这三个层次。面向对象的分析与设计技术则可以将这三个层次统一利用对象的概念进行表达。当前有很多面向对象的分析和设计方法,这里采用OOA与OOD技术进行三层结构的分析与设计。

MIS的三层结构中,中间的功能层是关键。运行MIS应用程序的最基本的任务,就是执行数千条定义业务如何运转的业务逻辑。一个业务处理过程就是一组业务处理规则的集合。中间层反映的是应用域模型,是MIS系统的核心内容。

OOA用于理解与掌握MIS应用域的业务运行框架,即应用域建模,OOA模型描述应用域中的对象,以及对象间各种各样的结构关系和通信关系。OOA模型有两个用途。首先,每个软件系统都建立在特定的现实世界中,OOA模型就是用来形式化该现实世界的"视网",建立起各种对象,分别表示软件系统主要的组织结构以及现实世界强加给软件系统的各种规则与约束条件。其次,给定一组对象,OOA模型规定了它们如何协同才能完成软件系统所指定的工作。这种协同在模型中,是以表明对象之间通信方式的组消息连接来表示的。

OOA模型划分为以下五个层次可视图。

(1) 对象-类层:表示待开发系统的基本构造块。对象都是现实世界中应用域概念的抽象。这一层是整个OOA模型的基础,在劳动管理信息系统中,存在一百多个类。

(2) 属性层:对象所存储(或容纳)的数据称为对象的属性。类的实例之间互相约束,必须遵从应用域的某些限制条件或业务规则,这些约束称为实例连接。对象的属性和实例连接共同组成了OOA模型的属性层。属性层中的业务规则是MIS中最易变化的部分。

(3) 服务层:对象的服务加上对象实例之间的消息通信共同组成了OOA模型的服务层。服务层中的服务包含了业务执行过程中的一部分业务处理逻辑,也是MIS中容易改变的部分。

(4) 结构层:结构层负责捕捉特定应用域的结构关系。分类结构表示类属成员的构成,反映通用性和特殊性。组装结构表示聚合,反映整体和组成部分。

(5) 主题层:主题层用于将对象归类到各个主题中,以简化OOA模型。为了简化劳动管理信息系统,将整个系统按业务职能划分为十三个主题,分别为职工基本信息管理、工资管理、劳动组织计划管理、劳动定员定额管理、劳动合同管理、劳动统计管理、职工等级鉴定管理、劳动保险管理、劳动力市场管理、劳动政策查询管理、领导查询系统、系统维护管理和系统安全控制。

在OOD方法中,OOD体系结构以OOA模型为设计模型的雏形。OOD将OOA的模型作为OOD的问题论域部分,并增加其他三个部分,即人机交互部分、任务管理部分和数据管理部分。各部分与PDC一样划分为五个层次,但是针对系统的不同方面。OOD的任务是将OOA所建立的应用模型计算机化,OOD所增加的三个部分是为应用模型添加计算

机的特征。

(1) 问题论域部分(Problem Domain Component,PDC):以 OOA 模型为基础,包含那些执行基本应用功能的对象,可逐步细化,使其最终能解决实现限制、特性要求、性能缺陷等方面的问题。PDC 封装了应用服务器功能层的业务逻辑。

(2) 人机交互部分(Human Interaction Component,HIC):指定了用于系统的某个特定实现的界面技术,在系统行为和用户界面的实现技术之间架起了一座桥梁,封装了客户层的界面表达逻辑。

(3) 任务管理部分(Task Management Component,TMC):把有关特定平台的处理机制底层系统的其他部分隐藏了起来。在该项目中,利用 TMC 实现分布式数据库的一致性管理。在三层 C/S 体系结构中,TMC 是应用服务器的一个组成部分。

(4) 数据管理部分(Data Management Component,DMC):定义了那些与所有数据库技术接口的对象,DMC 同样是三层结构中应用服务器的一部分。由于 DMC 封装了数据库访问逻辑使应用独立于特定厂商的数据库产品,因此便于系统的移植和分发。

OOD 的四个部分与三层结构的对应关系如图 5-23 所示。

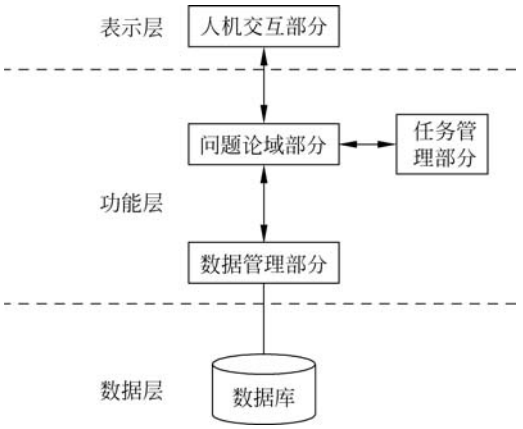

图 5-23　OOD 的四个部分与三层结构的对应关系

3. 系统实现与配置

三层 C/S 体系结构提供了良好的结构扩展能力。本质上,三层结构是一种开发分布式应用程序的框架,在系统实现时,可采用支持分布式应用的构件技术实现。

当前,已有三种分布式构件标准:微软的 DCOM、OMG 的 CORBA 和 SUN 的 EJB。这三种构件标准各有特点。考虑到在该项目应用环境的客户端和应用服务器均采用 Windows 和 Windows Server,这里采用在这些平台上具有较高效率的支持 DCOM 的 ActiveX 方式实现客户端和应用服务器的程序。

ActiveX 可将程序逻辑封装起来,并划分到进程内、本地或远程进程外执行。为将应用程序划分到不同的构件里面,这里引入"服务模型"的概念。服务模型提供了一种逻辑性(非物理性)的方式,如图 5-24 所示。

"服务模型"是对所创建的构件进行分组的一种逻辑方式,这种模型与语言无关。服务模型基于这样一个概念,每个构件都是一系列服务的集合,这些服务由构件提供给其他对象。

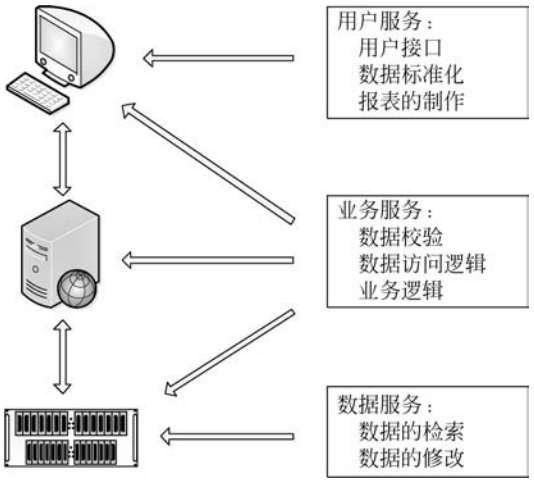

图 5-24　服务模型结构图

创建应用方案时，共有三种类型的服务可供选用，即用户服务、业务服务和数据服务。每种服务类型都对应于三层 C/S 体系结构中的某一层。在服务模型里，为实现构件间的相互通信，必须遵守以下两条基本的规则。

(1) 一个构件能向当前层及构件层上下的任何一个层的其他构件发出服务请示。

(2) 不能跳层发出服务请求，用户服务层内的构件不能直接与数据服务层内的构件通信，反之亦然。

在劳动管理信息系统的实现中，将 PDC 的十三个子系统及 TMC 和 DMC 分别用单独的构件实现，这样，系统可根据各单位的实际情况进行组合，实现系统的灵活配置。而且这些构件还可以作为一个部件用于构造新的更大的系统。

根据各种用户不同阶段对系统的不同需求以及系统未来的进化可能，这里拟定了以下几种不同的应用配置方案，包括单机配置方案、单服务器配置方案、业务服务器配置方案和事务服务器配置方案。

1) 单机配置方案

对于未能联入广域网的二级单位和三级单位单机用户，将三层结构的所有构件连同数据库系统均安装在同一台计算机上，与中心数据库的数据交换采用拨号上网或交换磁介质的方式完成，当联入广域网时，可根据业务量情况采用单服务器配置方案或业务服务器配置方案。

2) 单服务器配置方案

对于已建有局域网的二级单位，当建立了本地数据库且其系统负载不大时，可将业务服务构件与数据服务构件配置在同一台物理服务器中，而应用客户(表示层)构件在各用户的计算机内安装。

3) 业务服务器配置方案

这是三层结构的理想配置方案。工作负荷大的单位，采用将业务服务构件和数据服务构件分别配置于独立的物理服务器内以改善性能。该方案也适用于暂时不建立自己的数据库，而使用局劳资处的中心数据库的单位，此时只需建立一台业务服务器。该单位需要建立

自己的数据库时，只需把业务服务器的数据库访问接口改动一下，其他方面无须任何改变。图 5-25 给出了向 Intranet 方式的转移。

4）事务服务器配置方案

当系统采用 Intranet 方式提供服务时，将应用客户由构件方式改为 Web 页面方式，应用客户与业务服务构件之间的联系，由 Web 服务器与事务服务器之间的连接提供，事务服务器对业务服务构件进行统一管理和调度，业务服务构件和数据服务构件不必做任何修改，这样，既可以保证以前的投资不受损失，又可以保证业务运行的稳定性，向 Intranet 方式的转移是渐进的，两种运行方式将长期共存。

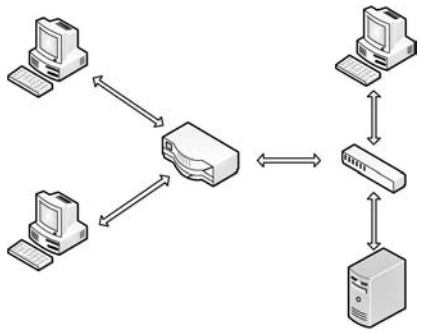

图 5-25　向 Intranet 方式的转移

上述各种方案中，除单机配置方案外，其他方案均能对系统的维护和安全管理提供极大的方便。任何应用程序的更新，只需在对应的服务器上更新有关的构件即可，安全性则由在服务器上对操作应用构件的用户进行相应的授权来保障，由于任何用户不直接拥有对数据库的访问权限，其操作必须通过系统提供的构件进行，这样就保证了系统的数据不被滥用，具有很高的安全性。同时，三层 C/S 体系结构具有很强的可扩展性，可以根据需要选择不同的配置方案，并且在应用扩展时方便地转移为另一种配置。

5.6　浏览器/服务器风格

浏览器/服务器（Browser/Server，B/S）结构是随着互联网技术的兴起，对 C/S 体系结构进行改进后形成的一种结构。在 B/S 体系结构下，用户界面完全通过 WWW 浏览器实现，部分事务逻辑在前端实现，但是主要事务逻辑在服务器端实现。它利用浏览器技术，结合浏览器的多种脚本语言并通过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，是一种全新的软件体系结构。B/S 体系结构图如图 5-26 所示。

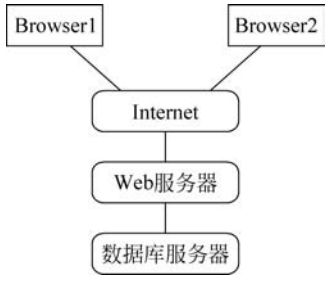

图 5-26　B/S 体系结构图

要实现一个完整的 Browser/Server 应用系统，需要由 Browser、Web Server 和 DB Server 三个部分组成。

B/S 模式第一层客户机的主体是浏览器，如 Netscape Navigator 或微软公司的 IE 等，它是 B/S 结构中用户与整个系统交互的界面，用于向服务器发送特定的数据或请求，以及接收从服务器发送来的数据。浏览器将 HTML 代码转化成图文并茂的网页，网页还具备一定的交互功能，允许用户在网页提供的申请表上输入信息提交给后台，并提出处理请求。这个后台就是第二层的 Web 服务器。

第二层 Web 服务器是实现 B/S 结构的关键。Web 服务器的引入，使得通过浏览器来访问数据库服务器成为可能，从而免去开发与维护客户端界面的大量工作。分散在各地的用户，只要安装了浏览器软件，都可以访问数据库服务器。Web 服务器作为一种应用服务器，可以将原来分布于客户端或服务器端的应用集中在一起，使系统的结构更加清晰和精

细,有利于系统的扩展。Web服务器作为客户端和服务器端的中介,起着沟通与协调的作用。

Web服务器的作用是接收浏览器的页面请求,找到正确的页面并将其回传给浏览器。随着互联网的发展以及用户对Web要求的提高,Web服务器含义覆盖的范围也有很大扩展。它需要接收浏览器的页面请求,对其中非Web页面制作语言的部分解释执行,承担着浏览器与数据库的接口作用。不同的系统平台提供的Web服务器是不同的,但是基本功能应该没有差别,只是其实现功能的方法不一样。

浏览器通过将URL发送给Web服务器请求信息,服务器通过返回超文本标记语言(HyperText Markup Language,HTML)进行页面响应,页面可以是已经格式化并存储在Web结点中的静态页面,也可能是服务器动态创建以响应用户所提供信息的页面,或者是列出在Web结点上可用的文件和文件夹的页面。

第三层数据库服务器的任务类似于C/S模式,负责协调不同的Web服务器发出的请求来管理数据库。在该层中存储了系统中所有需要发布的数据信息,因此为了保证Web站点的高速和高效,一般需要把数据库服务器放在硬件配置较好的机器上,可以和Web服务器在同一台计算机上,也可以位于两台甚至是多台计算机上。

B/S体系结构的优点表现为以下几个方面。

(1) 它简化了客户端,无须像C/S模式那样,在不同的客户机上安装不同的客户端应用程序,而只需安装通用的浏览器软件,就可享受到无限丰富的永远在不断变化和发展着的信息服务。这样,不但可以节省客户机的硬盘空间与内存,而且使客户端的安装和升级过程更加简便。原则上,取消了所有在客户机一侧的维护工作。

(2) B/S体系结构模式特别适用于网上信息的发布。

(3) B/S体系结构通过互联网技术统一访问不同种类的数据库,提供了异种机器、异种网络、异种应用服务之间统一服务的最现实的开放性基础。

同时,B/S体系结构也存在着以下问题。一方面,企业是一个有结构、有管理、有确定任务的有序实体,而互联网面向的却是一个无序的集合,B/S模式必须适应并迎合长期使用C/S模式的有序需求方式。另一方面,企业中已经积累了或多或少的各种基于非互联网技术的应用,如何恰当地与这些应用连接,是B/S模式中的一项极其重要的任务。此外,缺乏对动态页面的支持能力,没有集成有效的数据库处理功能,系统的扩展能力差,安全性难以控制以及集成工具不足等,都让我们在使用B/S体系结构模式时应慎重。

5.7 C/S与B/S混合结构风格

C/S与B/S混合体系结构是一种典型的异构体系结构。B/S体系结构即Browser/Server(浏览器/服务器)结构,是随着互联网技术的兴起,对C/S体系结构的一种变化或者改进的结构。在B/S体系结构下,用户界面完全通过WWW浏览器实现,部分事务逻辑在前端实现,但是主要事务逻辑在服务器端实现。

B/S体系结构主要是利用不断成熟的WWW浏览器技术,结合浏览器的多种脚本语言,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,是一种全新的软件体系结构。基于B/S体系结构的软件、系统安装、修改和维护全在服

务器端解决，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，很容易在运行时自动升级。B/S 体系结构还提供了异种机、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。

但是，与 C/S 体系结构相比，B/S 体系结构也有许多不足之处，例如：

(1) B/S 体系结构缺乏对动态页面的支持能力，没有集成有效的数据库处理功能。

(2) B/S 体系结构的系统扩展能力差，安全性难以控制。

(3) 采用 B/S 体系结构的应用系统，在数据查询等响应速度上远远低于 C/S 体系结构。

(4) B/S 体系结构的数据提交一般以页面为单位，数据的动态交互性不强，不利于联机事务处理过程(On-Line Transaction Processing，OLTP)的应用。

从上面的对比分析中可以看出，C/S 体系结构并非一无是处，而 B/S 体系结构也并非十全十美。由于 C/S 体系结构根深蒂固，技术成熟，原来的很多软件系统都是建立在该体系结构基础上的，因此，C/S 体系结构与 B/S 体系结构还将长期共存。

C/S 与 B/S 混合体系结构的优点是：外部用户不直接访问数据库服务器，能保证企业数据库的相对安全；企业内部用户的交互性较强，数据查询和修改的响应速度较快。

C/S 与 B/S 混合体系结构的缺点是：企业外部用户修改和维护数据时，速度较慢，较烦琐，数据的动态交互性不强。

5.8 正交软件体系结构风格

5.8.1 正交软件体系结构的概念

正交软件体系结构由组织层和线索的构件构成。

层是由一组具有相同抽象级别的构件构成的。线索是子系统的特例，由完成不同层次功能的构件组成(通过相互调用来关联)，每一条线索完成整个系统中相对独立的一部分功能。每一条线索的实现与其他线索的实现无关或关联很少，在同一层中的构件之间是不存在相互调用的。

如果线索是相互独立的，即不同线索中的构件之间没有相互调用，那么这个结构就是完全正交的。从以上定义可以看出，正交软件体系结构是一种以垂直线索构件族为基础的层次化结构，其基本思想是把应用系统的结构按功能的正交相关性垂直分割为若干个线索(子系统)，线索又分为几个层次，每个线索由多个具有不同层次功能与不同抽象级别的构件构成。各线索的相同层次的构件具有相同的抽象级别。因此，可以归纳正交软件体系结构的主要特征如下。

(1) 正交软件体系结构由完成不同功能的 $n(n>1)$ 个线索(子系统)组成。

(2) 系统具有 $m(m>1)$ 个不同抽象级别的层。

(3) 线索之间是相互独立的(正交的)。

(4) 系统有一个公共驱动层(一般为最高层)和公共数据结构(一般为最低层)。

对于大型的和复杂的软件系统，其子线索(一级子线索)还可以划分为更低一级的子线索(二级子线索)，形成多级正交结构。正交软件体系结构的框架如图 5-27 所示。

图 5-27 所示是一个三级线索、五层结构的正交软件体系结构框架图，该图中，A、B、D、

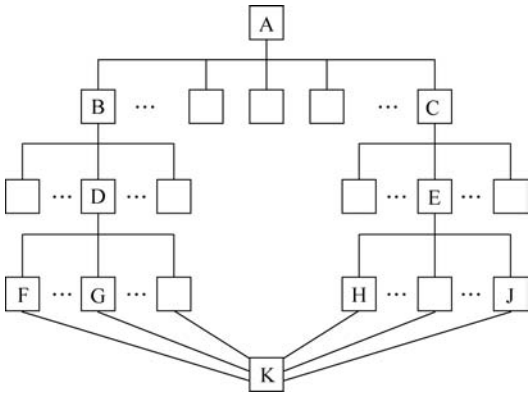

图 5-27　正交软件体系结构的框架

F、K 组成了一条线索，A、C、E、J、K 也是一条线索。因为 B、C 处于同一层次中，所以不允许进行互相调用；H、J 处于同一层次中，也不允许进行互相调用。一般来讲，第五层是一个物理数据库连接构件或设备构件，供整个系统公用。

在软件进化过程中，系统需求会不断发生变化。正交软件体系结构中，因线索的正交性，每一个需求变动仅影响某一条线索，而不会涉及其他线索。这样，就把软件需求的变动局部化了，产生的影响也被限制在一定范围内，因此实现容易。

5.8.2　正交软件体系结构的优点

正交软件体系结构具有以下优点。

(1) 结构清晰，易于理解。正交软件体系结构的形式有利于理解。由于线索功能相互独立，不进行互相调用，结构简单、清晰，因此构件在结构图中的位置已经说明所实现的是哪一级抽象，担负的是什么功能。

(2) 易修改，可维护性强。由于线索之间相互独立，因此对一个线索的修改不会影响到其他线索。当软件需求发生变化时，可以将新需求分解为独立的子需求，然后以线索和其中的构件为主要对象，分别对各个子需求进行处理，这样，软件修改就很容易实现。系统功能的增加或减少，只需相应地增删线索构件族，而不影响整个正交体系结构，因此能方便地实现结构调整。

(3) 可移植性强，重用粒度大。正交软件体系结构可以为一个领域内的所有应用程序所共享，这些软件有着相同或类似的层次和线索，可以实现体系结构级的重用。

5.8.3　正交软件体系结构的实例

本节以某省电力局的一个管理信息系统为例，讨论正交软件体系结构的应用。

1. 设计思想

设计初期，考虑到未来可能进行的机构改革，在系统投入运行后，单位各部门的功能有可能发生以下变化：

(1) 某些部门的功能可能改变或取消，或转入另外的部门，或其工作内容发生变更。

(2) 有的部门可能被撤销，其功能被整个并入其他部门或分解并入数个部门。

(3) 某些部门的功能可能需要扩充。

为适应将来用户需求可能发生的变化，尽量降低维护成本，提高可用性和重用性，这里使用了多级正交软件体系结构的设计思想。本系统中，考虑到其系统较大和实际应用的需要，将线索分为两级：主线索和子线索。总体结构包含数个主线索(第一级)，每个主线索又包含数个子线索(第二级)，因此，一个主线索也可以看成一个小的正交结构。

这样为大型软件结构功能的划分提供了便利，使得其既能对功能进行分类，又能在每一类中对功能进行细分，而且使功能划分既有序又合理，能控制在一定粒度以内，合理的粒度又能为线索和层次中构件的实现打下良好的基础。

然而完全正交结构不能很好地适用于本应用，故放宽了对结构正交的严格性，允许在线索间有适当的相互调用，因为各功能或多或少会有相互重叠的地方，因此会发生共享某些构件的情况。同时，还放宽了对结构分层的限制，允许某些线索(少数)的层次与其他线索(多数)不同。这些均是反复权衡理想情况时的优点和实现代价后总结出的原则，这样既易于实现，又能充分利用正交结构的优点。

2. 结构设计

按照上述思想，首先将整个系统设计为两级正交结构，第一级划分为 38 个主线索(子系统)，系统总体结构如图 5-28 所示，每个主线索又可划分为数个子线索(≥2)。

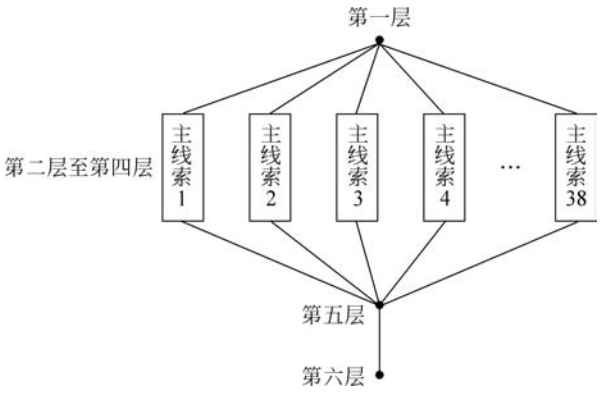

图 5-28　系统总体正交体系结构设计

为了简单起见，仅就其中的一个主线索进行说明，其他主线索的子线索划分也采用大致相同的策略。该主线索所实现的功能属于多种经营管理处的范围：该处有生产经营科、安全监察科、财务科、劳务科，包括 11 个管理功能(如人员管理、产品质量监督、安全监察、生产经营、劳资统计等)，即 11 个子线索，该主线索子正交体系结构如图 5-29 所示。

在图 5-29 中，主控窗口层、数据模型与数据库接口、物理数据库层分别对应图 5-28 中的第一层、第五层、第六层。综合图 5-28 和图 5-29 可以看出，整个 MIS 的结构包括以下六个层次。

(1) 第一层实现主控窗口，由主控窗口对象控制引发所有线索运行。

(2) 第二层实现菜单接口，支持用户选择不同的处理功能。

(3) 第三层涵盖了所有的功能对话框，这也就是与功能的真正接口。

(4) 第四层是真正的功能定义，在这一层定义的构件有数据录入构件(包括插入、删除、更新)、报表处理构件、快速查询构件、图形分析构件、报表打印构件等。

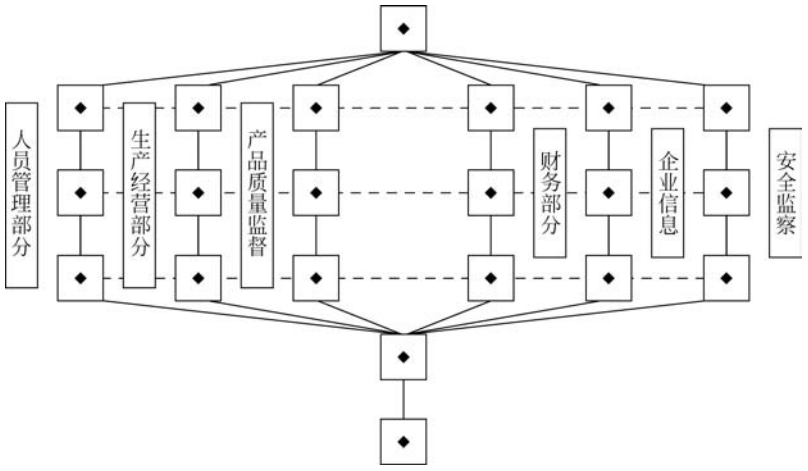

图 5-29　多种经营管理主线索子正交体系结构图

(5) 第五层和第六层是数据服务的实现,第五层包括了特定的数据模型和数据库接口,第六层就是数据库本身。

3. 程序编制

在软件结构设计方案确定之后,就可以开始正式的开发工作了,由于采用正交体系结构的思想,可以分数个小组并行开发。每个小组分配一条或数条线索,再专门一个小组来设计通用共享构件。构件是通用的,因此不必与其他小组频繁联系,加上各条线索之间相互调用少,所以各小组不会互相牵制,再加上构件的重用,从而大大提高编程效率,给设计带来极大的灵活性,缩短了开发周期,降低了工作量。

4. 进化控制

软件开发完成并运行一年后,用户单位提出了新的要求,要对原设计方案进行修改,按照前面提出的设计思想和方法,首先将提出的新功能要求映射到原设计结构上,这里仍以“多种经营管理主线索”为例,总结出以下变动。

1) 报表和报表处理功能的变动

例如,财务管理子线索有以下变动:财务报表有增删(直接在数据库中添加和删除表),某些报表需要增加一些汇兑处理和计算功能(对它的功能定义层做上修改标记),其他一些子功能(子线索)也需要增加自动上报功能,另外,所有的子线索都需要增加浏览器功能,以便对数据进行网上浏览。

2) 子线索的变动:增加养老统筹子线索

对初始结构的变动如图 5-30 所示。其中①、②所指不变,③表示在功能定义层新添加的一个构件,其中包含网上浏览功能和自动上报(E-mail 发送)功能,④表示新增加养老统筹子线索。新添加的构件用空心菱形表示,要修改的构件在其上套一个矩形做标记,其他未做标记的则表示可直接重用的构件,确定进化的结构图之后,按照自顶向下、由左至右的原则,更新进化工作得以有条不紊地进行。

现对多种经营管理主线索子正交结构进化情况进行统计:原结构包含子线索 11 条,构件 36 个;新结构包含子线索 12 条,构件 41 个,其中,重用构件 24 个,修改 11 个,新增 6 个;新增子线索 1 条。由于涉及添加公用共享构件,所以没有完全重用某条子线索的情况,但是

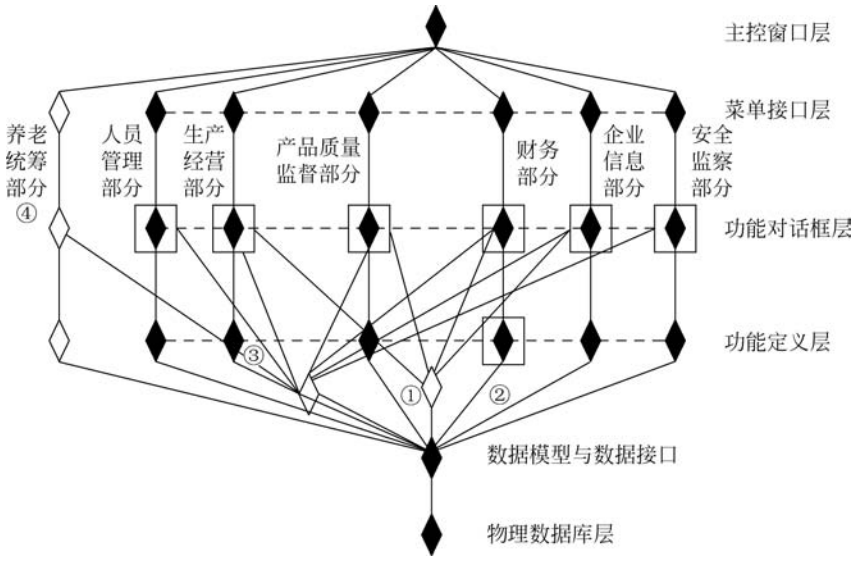

图 5-30　多种经营管理主线索结构变动情况图

大部分只需在功能对话框层做少量修改即可。

经分析，构件重用率为 58.6%，修改率为 26.8%，增加率为 14.6%。表 5-2 是对工作量的统计分析，其中，按每天 8 小时，单位为人/天。

38 个主线索的修改与开发工作量比例均未超过 13%，比传统方法工作量减少 20%左右。可见多级正交结构对于降低软件进化更新的开销是行之有效的，而且非常适合大型软件开发，特别是在 MIS 领域，由于其结构在一定应用领域内均有许多共同点，因此有一定的通用性。表 5-2 给出了劳动量比较。

表 5-2　劳动量比较

主线索	修改前开发工作量	修改工作量	修改与开发工作量之比
1	60	4	6.67%
2	120	4	3.33%
3	90	6	6.67%
4	60	4	6.67%
5	60	3	3.33%
6	60	2	5.00%
⋮			
37	60	2	3.33%
38	30	3	10%

5.9　异构结构风格

前面介绍并讨论了一些所谓的“纯”体系结构，但随着软件系统规模的扩大，系统也越来越复杂，所有的系统不可能都在单一的标准的结构上进行设计，这是因为以下几个方面的

原因。

(1) 从最根本上来说,不同的结构有不同的处理能力的强项和弱点,一个系统的体系结构应该根据实际需要进行选择,以解决实际问题。

(2) 关于软件包、框架、通信及其他一些体系结构上的问题,目前存在多种标准。即使在某段时间内某一种标准占统治地位,但变动最终是绝对的。

(3) 实际工作中,总会遇到一些遗留下来的代码,它们仍有效用,但是却与新系统有某种程度上的不协调。然而在许多场合,将技术与经济综合进行考虑时,总是决定不再重写它们。

(4) 即使在某一单位中,规定了共享共同的软件包或相互关系的一些标准,仍会存在解释或表示习惯上的不同,在 UNIX 中,就可以发现这类问题:即使规定用单一的标准(ASCII)来保证过滤器之间的通信,因为不同人对关于在 ASCII 流中信息如何表示的不同的假设,不同的过滤器之间仍可能不协调。

大多数应用程序只使用 10%的代码实现系统的公开的功能,剩下 90%的代码完成系统管理功能:输入和输出、用户界面、文本编辑、基本图表、标准对话框、通信、数据确认和旁听跟踪、特定领域(如数学或统计库)的基本定义。

如果能从标准的构件构造系统 90%的代码是很理想的,但不幸的是,即使能找到一组符合要求的构件,也很有可能发现,它们不会很融合地组织在一起。通常问题出在,各构件做了数据表示、系统组织、通信协议细节或某些明确的决定(如谁将拥有主控制线程)等不同的假设。举个简单的例子,UNIX 中的排序操作,过滤器和系统调用都是标准配置中的部分,虽然都是排序,但它们之间不可互换。

本部分将讨论 C/S 与 B/S 混合体系结构。从之前的讨论中可以看出,传统的 C/S 体系结构并非一无是处,而新兴的 B/S 体系结构也并非十全十美。由于 C/S 体系结构根深蒂固,技术成熟,原来的很多软件系统都是建立在 C/S 体系结构基础上的,因此,C/S 体系结构与 B/S 体系结构还将长期共存,其结合方式主要有两种。下面分别讨论 C/S 与 B/S 混合体系结构的两个模型。

1. “内外有别”模型

在 C/S 与 B/S 混合体系结构的“内外有别”模型中,企业内部用户通过局域网直接访问数据库服务器,软件系统采用 C/S 体系结构。企业外部用户通过互联网访问 Web 服务器,通过 Web 服务器再访问数据库服务器,软件系统采用 B/S 体系结构。

“内外有别”模型的优点是外部用户不直接访问数据库服务器,能保证企业数据库的相对安全。企业内部用户的交互性较强,数据查询和修改的响应速度较快;其缺点是企业外部用户修改和维护数据时,速度较慢,较烦琐,数据的动态交互性不强。

2. “查改有别”模型

在 C/S 与 B/S 混合体系结构的“查改有别”模型中,不管用户是通过什么方式(局域网或互联网)连接到系统,凡是需执行维护和修改数据操作的,就使用 C/S 体系结构;如果只是执行一般的查询和浏览操作,则使用 B/S 体系结构。

“查改有别”模型体现了 B/S 体系结构和 C/S 体系结构的共同优点。但因为外部用户能够直接通过互联网连接到数据库服务器,企业数据容易暴露给外部用户,给数据安全造成了一定的威胁。

(1) 因为本部分只讨论软件体系结构问题，所以在模型图中省略了有关网络安全的设备，如防火墙等，这些安全设备和措施是保证数据安全的重要手段。

(2) 在这两个模型中，只注明(外部用户)通过互联网连接到服务器，但并没有解释具体的连接方式，这种连接方式取决于系统建设的成本和企业规模等因素。例如，某集团公司的子公司要访问总公司的数据库服务器，既可以使用拨号方式，也可以使用 DDN 方式等。

(3) 本部分对内部与外部的区分指是否直接通过内部局域网连接到数据库服务器进行软件规定的操作，而不是指软件用户所在的物理位置。例如，某个用户在企业内部办公室里，其计算机也通过局域网连接到了数据库服务器，但当他使用软件时，是通过拨号的方式连接到 Web 服务器或数据库服务器，则该用户属于外部用户。

软件工程师有许多技术来处理结构上的不匹配。最简单的描述这些技术的例子是只有两个构件的情况。这两个构件可以是对等构件、一对相互独立的应用程序、一个库和一个调用者、客户机和服务器等，其基本形式如图 5-31 所示。

图 5-31　构件协调问题

A 和 B 不能协调工作的原因，可能是事先做了对数据表示、通信、包装、同步、语法、控制等方面的假设，这些方面统称为形式。下面给出若干种解决 A 与 B 之间不匹配问题的方法。这里假设 A 和 B 是对称的，它们可以互换。

(1) 形式 A 改变成 B 的形式。为了与另一构件协调，彻底重写其中之一的代码是可能的，但有时又是很昂贵的。

(2) 公布 A 的形式的抽象化信息。例如，应用程序编程接口公布了控制一个构件的过程调用，开放接口时通常提供某种附加的抽象信息，可以使用凸出部(projections)或视图(views)来提供数据库，特别是联合数据库的抽象。

(3) 在数据传输过程中，从 A 的形式转变到 B 的形式。例如，某些分布式系统在数据传输中进行从 big-endian 到 little-endian 的转变。

(4) 通过协商，达成一个统一的形式。例如，调制解调器通常协商以发现最快的通信协议。

(5) 使 B 成为支持多种形式。例如，Machintosh 的 fat binaries 可以在 680X0 或 PowerPC 处理器上执行，可移动的 UNIX 代码可以在多种处理器上运转。

(6) 为 B 提供进口/出口转换器，有两种重要形式。其一，独立的应用程序提供表示转换服务，如通常使用的图像格式转换程序(可以在至少 50 种格式之间、10 种平台上进行转换)；其二，某些系统协调扩充或外部 additions，它们可以完成内外数据格式之间的相互转换。

(7) 引入中间形式。第一，外部相互交换表示，有时通过界面描述语言(Interface Description Languages，IDLs)支持，能够提供一个中介层，这在存在多个形式互不相同的构件结合在一起时特别有用；第二，标准的发布形式，如 RTF、MIF、Postscript，由 Adobe Acrobat 提供，作为一种广泛流通的安全的表示法；第三，活跃的调解者可以被插入系统，作为中介。

(8) 在 A 上添加一个适配器或包装器。最终的包装器可能是一种处理器模拟另一种处理器的代码。软件包装器可以在形式上掩盖不同，如 Mosaic 和其他 Web 浏览器隐藏了所显示的文件的表示一样。

(9) 保持对 A 和 B 的版本并行一致。虽然较微妙，但保持 A 和 B 自己的形式，并完成

两种形式的所有改变是有可能的。

以上这些技术有各自的优缺点,在初始化、时间和空间效率、灵活性和绝对的处理能力上有很大差别。

5.10 小　　结

软件体系结构是软件工程领域中的一个重要研究方向,是大型软件开发中的关键技术。一种体系结构风格以结构组织模式定义了一个系统家族。软件体系结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。经过多年的发展,软件体系结构也超出了传统的对于软件设计阶段的支持,逐渐扩展到整个软件生命周期。

本章综述了体系结构的基本风格、三层C/S体系结构风格、浏览器/服务器风格、C/S与B/S混合结构风格、正交软件体系结构风格、基于层次消息总线的体系风格与异构结构风格的概念与优劣,并对现有若干种体系结构风格进行归纳、比较与总结,描述了其主要发展方向以及应用的主要意义。

5.11 思　考　题

(1) 选择一个熟悉的大型软件系统,分析其体系结构中用到的风格,以及表现出的特点(为什么要采用这种风格?采用这种风格带来哪些优势?具有哪些不足?)

(2) 选择四种风格,设计简单的体系结构,并实现简单的原型系统。

(3) 不同的体系结构风格具有各自的特点、优劣和用途。试对管道-过滤器风格、分层系统风格、C2风格和基于消息总线的风格进行分析比较。