第3章　软件体系结构风格

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。也就是说，能否在不同的软件系统中，使用同一体系结构。基于这个目的，学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(idiomatic paradigm)。体系结构风格定义了一个系统家族，即一个体系结构定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型，而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性，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。只要系统是使用常用的、规范的方法来组织，就可使别的设计师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，相关人员立刻就会明白系统是如何组织和工作的。

3.1　经典软件体系结构风格

软件体系结构风格为大粒度的软件重用提供了可能。然而，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要回答的问题是：

(1) 设计词汇表是什么？

(2) 构件和连接件的类型是什么？

(3) 可容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体系结构风格的最关键的4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。Garlan和Shaw根据此框架给出了通用体系结构风格的分类。

(1) 数据流风格：批处理序列、管道与过滤器。

(2) 调用/返回风格：主程序与子程序、面向对象风格、层次结构。

(3) 独立构件风格：进程通信、事件系统。

(4) 虚拟机风格：解释器、基于规则的系统。

(5) 仓库风格：数据库系统、超文本系统、黑板系统。

3.1.1　管道与过滤器

在管道与过滤器风格的软件体系结构中，每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全消费之前，输出便产生了。因此，这里的构件被称为过滤器，这种风格的连接件就像是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。此风格特别重要的过滤器必须是独立的实体，它不能与其他的过滤器共享数据，而且一个过滤器不知道它上游和下游的标识。一个管道与过滤器网络输出的正确性并不依赖于过滤器进行增量计算过程的顺序。

图 3-1 是管道与过滤器风格的示意图。一个典型的管道与过滤器体系结构的例子是以 UNIX Shell 编写的程序。UNIX 既提供一种符号，以连接各组成部分(UNIX 的进程)，又提供某种进程运行时机制以实现管道。另一个著名的例子是传统的编译器。传统的编译器一直被认为是一种管道系统，在该系统中，一个阶段(包括词法分析、语法分析、语义分析和代码生成)的输出是另一个阶段的输入。

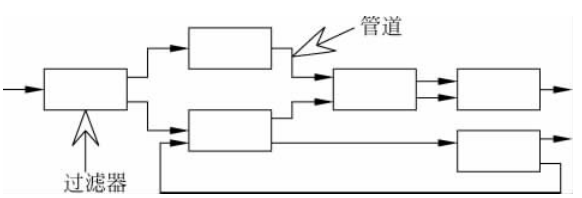

图 3-1　管道与过滤器风格的体系结构

管道与过滤器风格的软件体系结构具有许多很好的特点：

(1) 使得软构件具有良好的隐蔽性和高内聚、低耦合的特点。

(2) 允许设计师将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成。

(3) 支持软件重用。只要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来。

(4) 系统维护和增强系统性能简单。新的过滤器可以添加到现有系统中来；旧的可以被改进的过滤器替换掉。

(5) 允许对一些如吞吐量、死锁等属性的分析。

(6) 支持并行执行。每个过滤器是作为一个单独的任务完成，因此可与其他任务并行执行。

但是，这样的系统也存在着若干不利因素。

(1) 通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计师必须将每个过滤器看成一个完整的从输入到输出的转换。

(2) 不适合处理交互的应用。当需要增量地显示改变时,这个问题尤为严重。

(3) 因为在数据传输上没有通用的标准,每个过滤器都增加了解析和合成数据的工作,这样就导致了系统性能下降,并增加了编写过滤器的复杂性。

3.1.2 数据抽象和面向对象系统

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,数据的表示方法和它们的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。对象是一种被称作管理者的构件,因为它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 3-2 是数据抽象和面向对象风格的示意图。

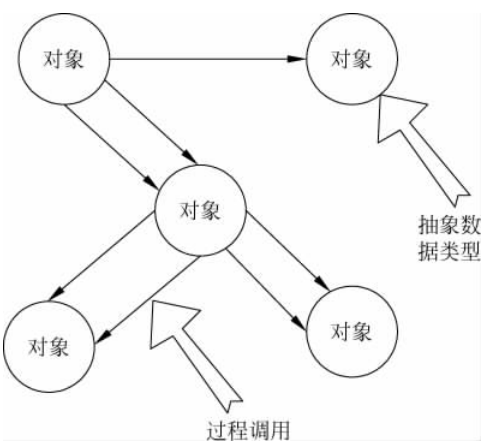

图 3-2 数据抽象和面向对象风格的体系结构

面向对象的系统有许多优点,并早已为人所知:

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

(2) 设计师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某些问题:

(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 的影响可能是料想不到的。

3.1.3 基于事件的系统

基于事件的系统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。因此，该风格也称为隐式调用。

从体系结构上说，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，它声明该事件，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。

隐式调用系统的主要优点有：

(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中时，只需将它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。

隐式调用系统的主要缺点有：

(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。而且即使它知道事件注册了哪些构件的过程，它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。在这些情况下，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，关于正确性的推理存在问题。

3.1.4　分层系统

层次系统组织成一个层次结构，每一层为上层服务，并作为下层客户。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中构件在一些层实现了虚拟机(在另一些层次系统中层是部分不透明的)。连接件通过决定层间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。

这种风格支持基于可增加抽象层的设计。这样，允许将一个复杂问题分解成一个增量步骤序列的实现。由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件重用提供了强大的支持。

图 3-3 是层次系统风格的示意图。层次系统最广泛的应用是分层通信协议。在这一应用领域中，每一层提供一个抽象的功能，作为上层通信的基础。较低的层次定义低层的交互，最低层通常只定义硬件物理连接。

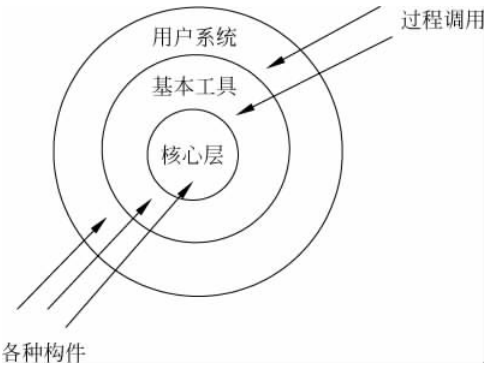

图 3-3 层次系统风格的体系结构

层次系统有许多可取的属性：

(1) 支持基于抽象程度递增的系统设计，使设计师可以把一个复杂系统按递增的步骤进行分解。

(2) 支持功能增强，因为每一层至多和相邻的上下层交互，因此功能的改变最多影响相邻的上下层。

(3) 支持重用。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。

但是，层次系统也有其不足之处：

(1) 并不是每个系统都可以很容易地划分为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计师不得不把一些低级或高级的功能综合起来。

(2) 很难找到一个合适的、正确的层次抽象方法。

3.1.5 仓库系统及知识库

在仓库(repository)风格中，有两种不同的构件：中央数据结构说明当前状态，独立构件在中央数据存储上执行，仓库与外构件间的相互作用在系统中会有大的变化。

控制原则的选取产生两个主要的子类。若输入流中某类时间触发进程执行的选择，则仓库是一传统型数据库；另外，若中央数据结构的当前状态触发进程执行的选择，则仓库是一黑板系统。

图 3-4 是黑板系统的组成。黑板系统的传统应用是信号处理领域，如语音和模式识别，另一应用是松耦合代理数据共享存取。

从图 3-4 中可以看出，黑板系统主要由三部分组成：

(1) 知识源。知识源中包含独立的、与应用程序相关的知识，知识源之间不直接进行通信，它们之间的交互只通过黑板来完成。

(2) 黑板数据结构。黑板数据是按照与应用程序相关的层次来组织的解决问题的数据，知识源通过不断地改变黑板数据来解决问题。

(3) 控制。控制完全由黑板的状态驱动，黑板状态的改变决定使用的特定知识。

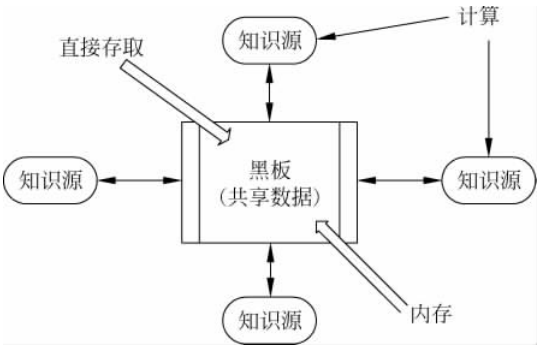

图 3-4　黑板系统的组成

3.1.6　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则如下：

(1) 系统中的构件和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，而构件与构件之间的直接连接是不允许的。

(3) 一个连接件可以和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 3-5 是 C2 风格的示意图。图中构件与连接件之间的连接体现了 C2 风格中构建系统的规则。

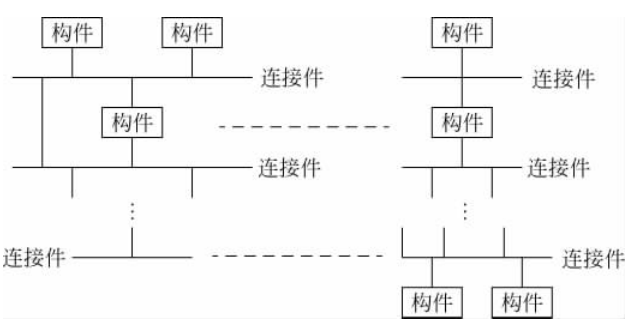

图 3-5　C2 风格的体系结构

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。

3.2 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计算机/小型计算机计算模型。它是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。在多用户环境中,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很高,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。

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,彻底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,由于大型计算机系统固有的缺陷,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(downsizing)。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软件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,C/S体系结构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,如图3-6所示。

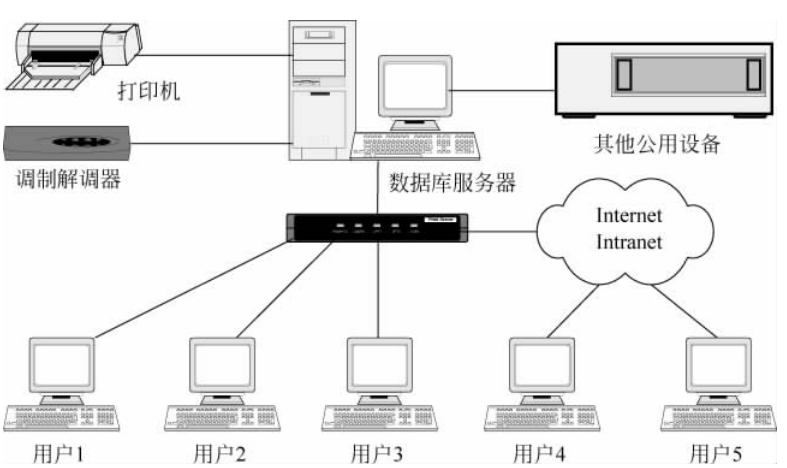

图3-6 C/S体系结构示意图

服务器负责有效地管理系统的资源,其任务集中于:

(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。

客户应用程序的主要任务是：

(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应用程序之间的数据传输。

C/S 体系结构将应用一分为二，服务器(后台)负责数据管理，客户机(前台)完成与用户的交互任务。服务器为多个客户应用程序管理数据，而客户程序发送、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)”的体系结构。其数据流图如图 3-7 所示。

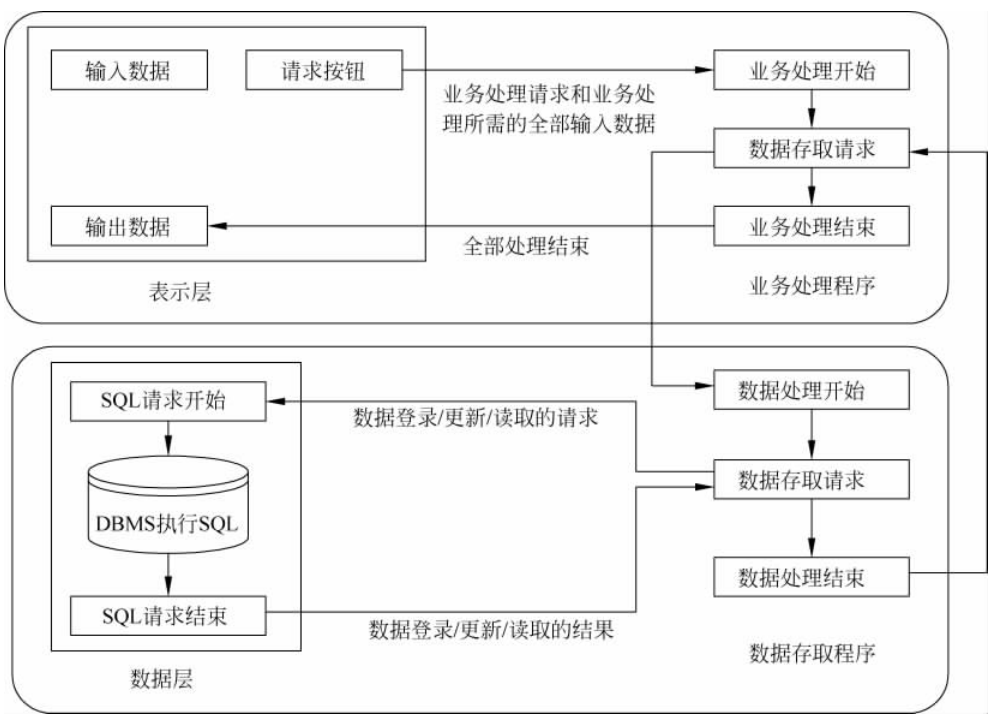

图 3-7　C/S 结构的一般处理流程

在一个 C/S 体系结构的软件系统中，客户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，而不是像文件服务器那样针对整个文件进行，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缩小。在 C/S 体系结构中，系统中的功能构件充分隔离，客户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,且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,开发之初就已确定,而且不能随时截取办公信息和档案等外部信息,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,使用繁杂,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) 软件维护和升级困难。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机升级,每个客户机上的软件都需维护。对软件的一个小小改动(例如只改动一个变量),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。

3.3 三层 C/S 结构风格

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,传统的两层 C/S 结构存在以下几个局限:

(1) 两层 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。

(2) 软、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,系统的性能容易变坏。

(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,那么,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。

正是因为两层 C/S 体系结构有这么多缺点,因此,三层 C/S 体系结构应运而生。其结构如图 3-8 所示。

与两层 C/S 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(thin client)。

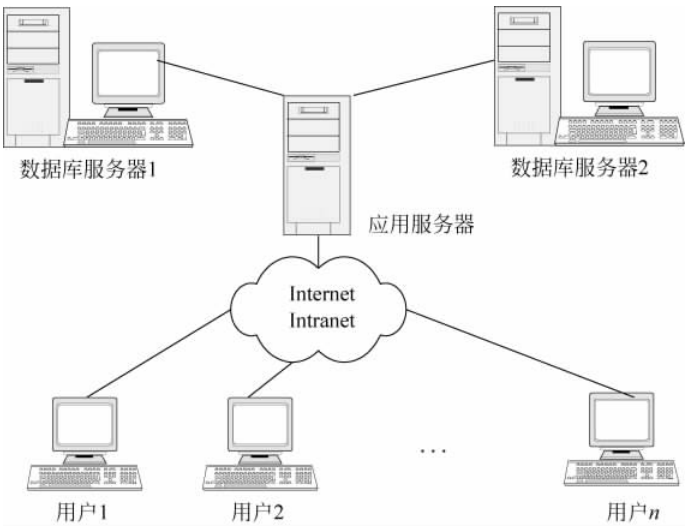

图 3-8　三层 C/S 结构示意图

3.3.1　各层的功能

三层 C/S 体系结构是将应用功能分成表示层、功能层和数据层三个部分，如图 3-9 所示。

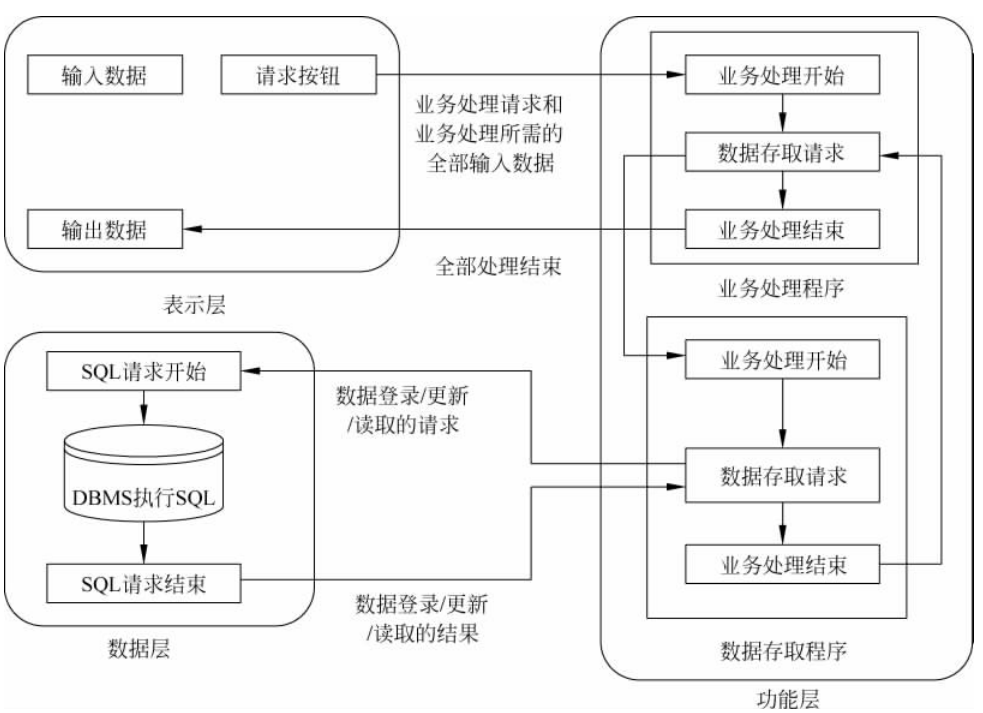

图 3-9　三层 C/S 结构的一般处理流程

1. 表示层

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界面(Graphic User Interface,GUI),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,它是将具体的业务处理逻辑编入程序中。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或数据层取得。表示层和功能层之间的数据交往要尽可能简捷。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和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(RDBMS),因此,一般从功能层传送到数据层的要求大都使用 SQL。

三层 C/S 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,关键是要将表示层和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 3-10 中(1)或(2)所示。如果像图 3-10 中(3)所示的那样连功能层也放在客户机中,与两层 C/S 体系结构相比,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,其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。

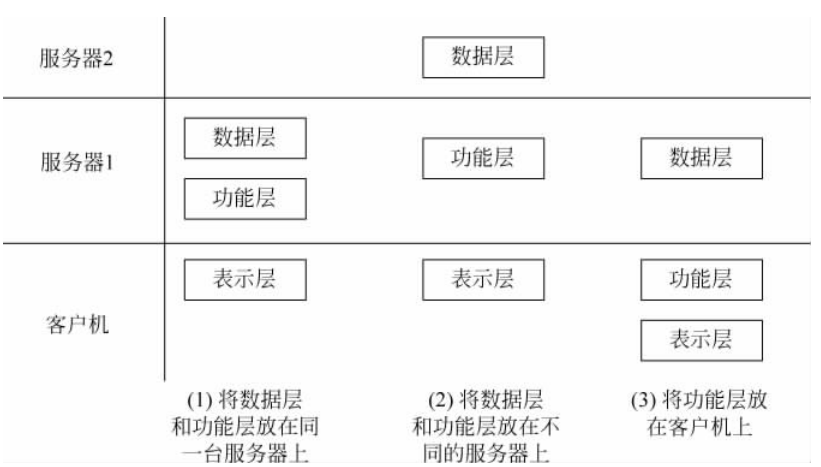

图 3-10 三层 C/S 物理结构比较

如果将功能层和数据层分别放在不同的服务器中,如图 3-10 中(2)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，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。

在三层 C/S 体系结构中，中间件是最重要的构件。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。

3.3.2　三层 C/S 结构应用实例

本节通过某石油管理局劳动管理系统的设计与开发，来介绍三层 C/S 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：

（1）信息量大，须存储并维护全油田近二十万名职工的基本信息以及其他各种管理信息。

（2）单位多，分布广，系统涵盖七十多个单位，分布范围八万余平方千米。

（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，系统用户总数达七百多个。

（4）网络环境不断发展，七十多个二级单位中有四十多个连入广域网，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，而已建成的广域网仅有骨干线路速度为 100M，大部分外围线路速率只有 64k～2M。

项目要求系统应具备较强的适应能力和演化能力，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和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（简称 Intranet 方式）演化。

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、数据层等三个层次，每一层次都有自己的特点，如表示层是图形化的、事件驱动的，功能层是过程化的，数据层则是结构化和非过程化的，难以用传统的结构化分析与设计技术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，采用 Coad 和 Yourdon 的 OOA（Object-Oriented Anaylzing，面向对象的分析）与 OOD（Object-Oriented Design，面向对象的设计）技术进行三层结构的分析与设计。

在 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。

Coad和Yourdon的OOA用于理解和掌握MIS应用域的业务运行框架,也就是应用域建模。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"视图"。它建立起各种对象,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和约束条件。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一组消息连接来表示的。

OOA模型划分为5个层次或视图,分别如下:

(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,它们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13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,职工考核鉴定管理,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(PDC),并增加其他三个部分:人机交互部分(HIC)、任务管理部分(TMC)和数据管理部分(DMC)。各部分与PDC一样划分为5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: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: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。HIC封装了客户层的界面表达逻辑。

(3) 任务管理部分: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结构中,TMC是应用服务器的一个组成部分。

(4) 数据管理部分: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，便于系统的移植和分发。

OOD 的 4 个部分与三层结构的对应关系如图 3-11 所示。

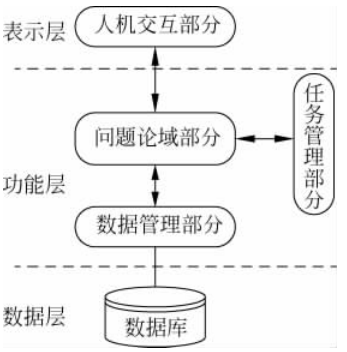

图 3-11　OOD 与三层 C/S 结构

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。三层结构在本质上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：Microsoft 的 DCOM、OMG 的 CORBA 和 Sun 的 JavaBeans。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 98/2000 和 Windows NT/2000，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，引入“服务模型”的概念。服务模型提供了一种逻辑性（而非物理性）的方式，如图 3-12 所示。

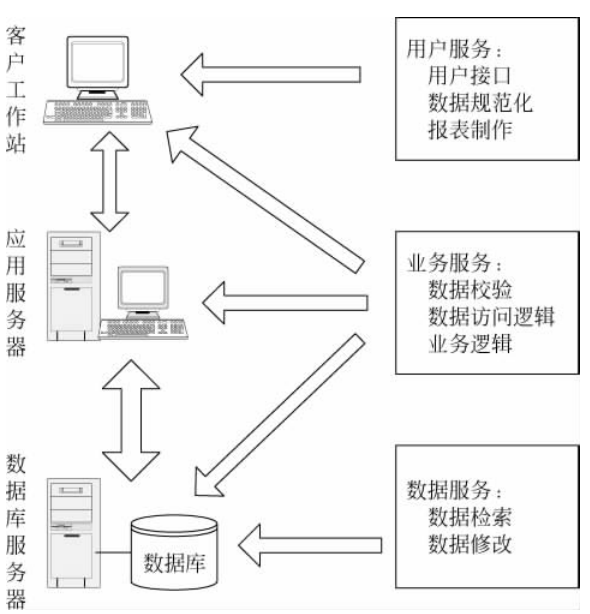

图 3-12　服务模型结构图

“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。

创建应用方案的时候,共有三种类型的服务可供选用:用户服务、业务服务以及数据服务。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13个子系统以及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MIS。

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演化可能,拟定了如下几种不同的应用配置方案: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

对于未能连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机器上,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。当它连入广域网时,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,此时只须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。

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,如图3-13所示。

在上述各种方案中,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

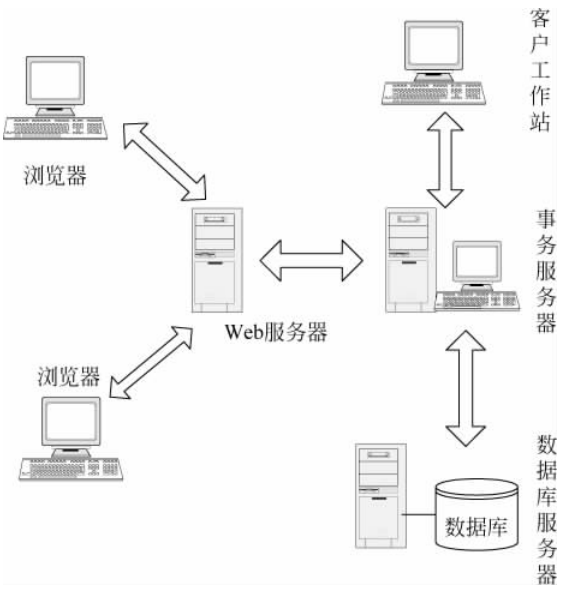

图 3-13　向 Intranet 方式的转移

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。

3.3.3　三层 C/S 结构的优点

根据三层 C/S 的概念及使用实例可以看出，与两层 C/S 结构相比，三层 C/S 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，能提高系统和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和硬件系统，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；并且这些平台和各个组成部分可以具有良好的可升级性和开放性。例如，最初用一台 UNIX 工作站作为服务器，将数据层和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 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 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。

3.4 浏览/服务器风格

在三层C/S体系结构中,表示层负责处理用户的输入和向客户的输出(出于效率的考虑,它可能在向上传输用户的输入前进行合法性验证)。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,并把结果返回给客户端。数据层负责实际的数据库存储和检索,响应功能层的数据处理请求,并将结果返回给功能层。

浏览/服务器(Browser-Server,B/S)风格就是上述三层应用结构的一种实现方式,其具体结构为:浏览器/Web服务器/数据库服务器。采用B/S结构的计算机应用系统的基本框架如图3-14所示。

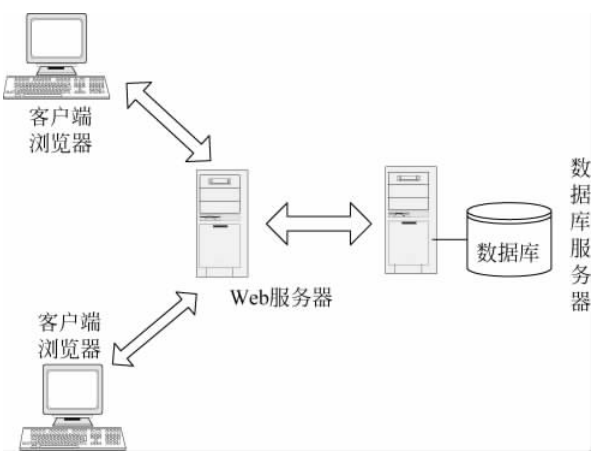

图3-14 B/S模式结构

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。从某种程度上来说,B/S结构是一种全新的软件体系结构。

在B/S结构中,除了数据库服务器外,应用程序以网页形式存放于Web服务器上,用户运行某个应用程序时只须在客户端上的浏览器中输入相应的网址,调用Web服务器上的应用程序并对数据库进行操作完成相应的数据处理工作,最后将结果通过浏览器显示给用户。可以说,在B/S模式的计算机应用系统中,应用(程序)在一定程度上具有集中特征。

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。

B/S结构出现之前,管理信息系统的功能覆盖范围主要是组织内部。B/S结构的"零客户端"方式,使组织的供应商和客户(这些供应商和客户有可能是潜在的,也就是说可能是事

先未知的)的计算机方便地成为管理信息系统的客户端,进而在限定的功能范围内查询组织相关信息,完成与组织的各种业务往来的数据交换和处理工作,扩大了组织计算机应用系统的功能覆盖范围,可以更加充分利用网络上的各种资源,同时应用程序维护的工作量也大大减少。另外,B/S 结构的计算机应用系统与 Internet 的结合也使新的企业计算机应用(如电子商务,客户关系管理)的实现成为可能。

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,要远远低于 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,不利于在线事务处理(OnLine Transaction Processing,OLTP)应用。

因此,虽然 B/S 结构的计算机应用系统有如此多的优越性,但由于 C/S 结构的成熟性且 C/S 结构的计算机应用系统网络负载较小,因此,未来一段时间内,将是 B/S 结构和 C/S 结构共存的情况。但是,很显然,计算机应用系统计算模式的发展趋势是向 B/S 结构转变。

3.5　公共对象请求代理体系结构

CORBA 是由 OMG 制定的一个工业标准,其主要目标是提供一种机制,使得对象可以透明地发出请求和获得应答,从而建立起一个异质的分布式应用环境。

由于分布式对象计算技术具有明显优势,OMG 提出了 CORBA 规范来适应该技术的进一步发展。1991 年,OMG 基于面向对象技术,给出了以对象请求代理(Object Request Broker,ORB)为中心的对象管理结构,如图 3-15 所示。

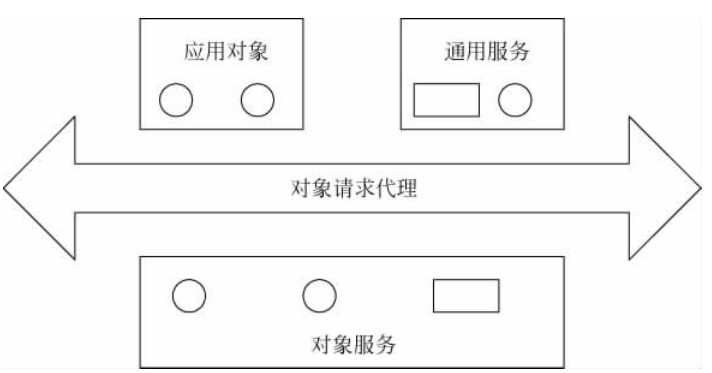

图 3-15　对象管理结构

在 OMG 的对象管理结构中,ORB 是一个关键的通信机制,它以实现互操作性为主要目标,处理对象之间的消息分布。对象服务实现基本的对象创建和管理功能,通用服务则使用对象管理结构所规定的类接口实现一些通用功能。

3.5.1 CORBA 技术规范

针对 ORB,OMG 又进一步提出了 CORBA 技术规范,主要内容包括接口定义语言(Interface Definition Language,IDL)、接口池(Interface Repository,IR)、动态调用接口(Dynamic Invocation Interface,DII)、对象适配器(Object Adapter,OA)等。

1. 接口定义语言

CORBA 利用 IDL 统一地描述服务器对象(向调用者提供服务的对象)的接口。IDL 本身也是面向对象的。它虽然不是编程语言,但它为客户对象(发出服务请求的对象)提供了语言的独立性,因为客户对象只需了解服务器对象的 IDL 接口,不必知道其编程语言。IDL 语言是 CORBA 规范中定义的一种中性语言,它用来描述对象的接口,而不涉及对象的具体实现。在 CORBA 中定义了 IDL 语言到 C、C++、SmallTalk 和 Java 语言的映射。

2. 接口池

CORBA 的接口池包括分布计算环境中所有可用的服务器对象的接口表示。它使动态搜索可用服务器的接口、动态构造请求及参数成为可能。

3. 动态调用接口

CORBA 的动态调用接口提供了一些标准函数以供客户对象动态创建请求、动态构造请求参数。客户对象将动态调用接口与接口池配合使用可实现服务器对象接口的动态搜索、请求及参数的动态构造与动态发送。当然,只要客户对象在编译之前能够确定服务器对象的 IDL 接口,CORBA 也允许客户对象使用静态调用机制。显然,静态机制的灵活性虽不及动态机制,但执行效率却胜过动态机制。

4. 对象适配器

在 CORBA 中,对象适配器用于屏蔽 ORB 内核的实现细节,为服务器对象的实现者提供抽象接口,以便他们使用 ORB 内部的某些功能。这些功能包括服务器对象的登录与激活、客户请求的认证等。

CORBA 定义了一种面向对象的软件构件构造方法,使不同的应用可以共享由此构造出来的软件构件。每个对象都将其内部操作细节封装起来,同时又向外界提供了精确定义的接口,从而降低了应用系统的复杂性,也降低了软件开发费用。CORBA 的平台无关性实现了对象的跨平台引用,开发人员可以在更大的范围内选择最实用的对象加入到自己的应用系统之中。CORBA 的语言无关性使开发人员可以在更大的范围内相互利用别人的编程技能和成果。

3.5.2 CORBA 风格分析

CORBA 的设计词汇表 = [构件 ::= 客户机系统/服务器系统/其他构件; 连接件 ::= 请求/服务]。其中,客户机系统包括客户机应用程序、客户桩(stump)、上下文对象和接口仓库等构件,以及桩类型激发 API 和动态激发 API 等连接件。服务器系统包括服务器应用程序方法库,服务器框架和对象请求代理等构件,以及对象适配器等连接件。CORBA 的体系结构模式如图 3-16 所示。

在此体系结构中,客户机应用程序用桩类型激发 API 或者动态激发 API 向服务器发送请求。在服务器端接受方法调用请求,不进行参数引导,设置需要的上下文状态,激发服务

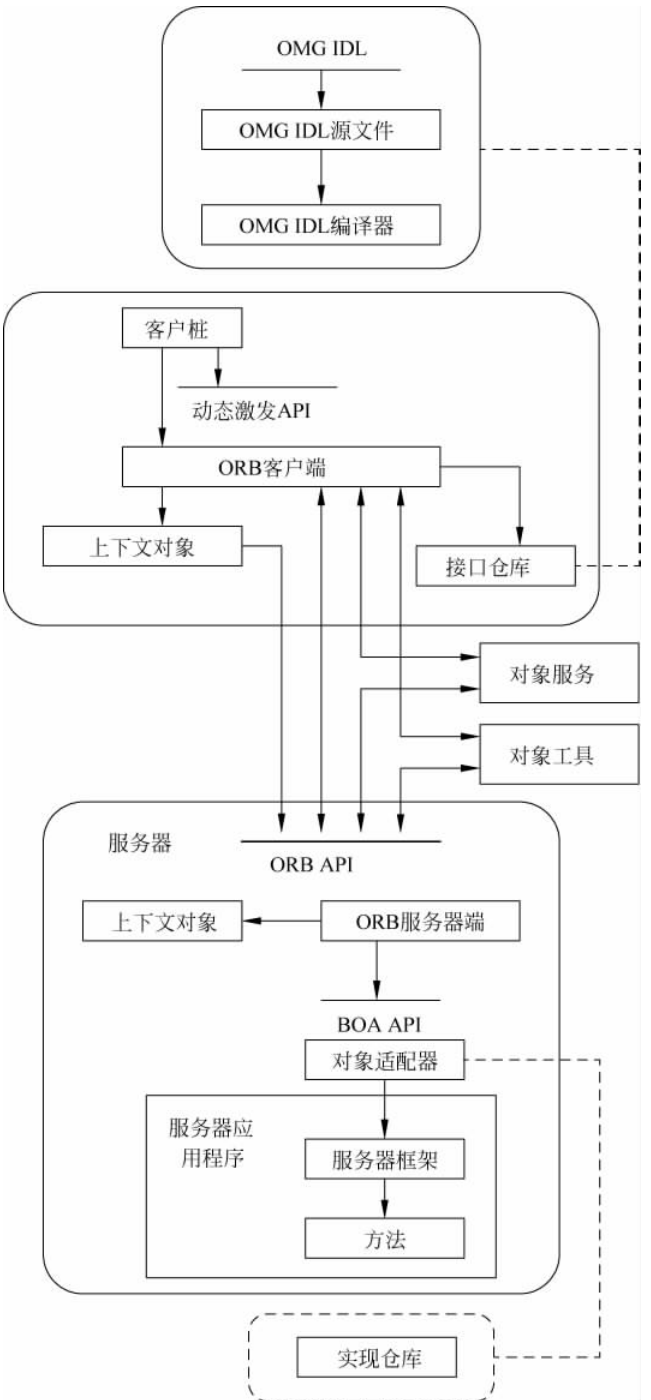

图 3-16　CORBA 的体系结构模式

器框架中的方法调度器,引导输出参数,并完成激发。服务器应用程序使用服务器端的服务部分,它包含某个对象的一个或者多个实现,用于满足客户机对指定对象上的某个操作的请求。

很明显,客户机系统是独立于服务器系统的,同样,服务器系统也独立于客户机系统。

CORBA体系结构模式充分利用了现今软件技术发展的最新成果,在基于网络的分布式应用环境下实现应用软件的集成,使得面向对象的软件在分布、异构环境下实现可重用、可移植和互操作。其特点可以总结为如下几个方面:

(1) 引入中间件作为事务代理,完成客户机向服务对象方(Server)提出的业务请求,引入中间件概念后分布计算模式如图3-17所示。

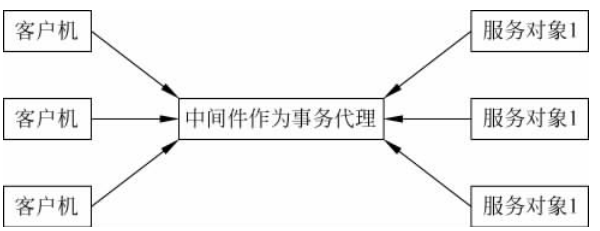

图3-17 引入中间件后客户机与服务器之间的关系

(2) 实现客户与服务对象的完全分离,客户不需要了解服务对象的实现过程以及具体位置。

(3) 提供软总线机制,使得在任何环境下、采用任何语言开发的软件只要符合接口规范的定义,均能够集成到分布式系统中。

(4) CORBA规范软件系统采用面向对象的软件实现方法开发应用系统,实现对象内部细节的完整封装,保留对象方法的对外接口定义。

在以上特点中,最突出的是中间件的引入。对象模型是应用开发人员对客观事物属性和功能的具体抽象。由于CORBA使用了对象模型,将CORBA系统中所有的应用看成对象及相关操作的集合,因此通过对象请求代理,使CORBA系统中分布在网络中应用对象的获取只取决于网络的畅通性和服务对象特征获取的准确程度,而与对象的位置以及对象所处的设备环境无关。

3.6 正交软件体系结构

正交(orthogonal)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。线索是子系统的特例,它是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和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 3-18 所示。

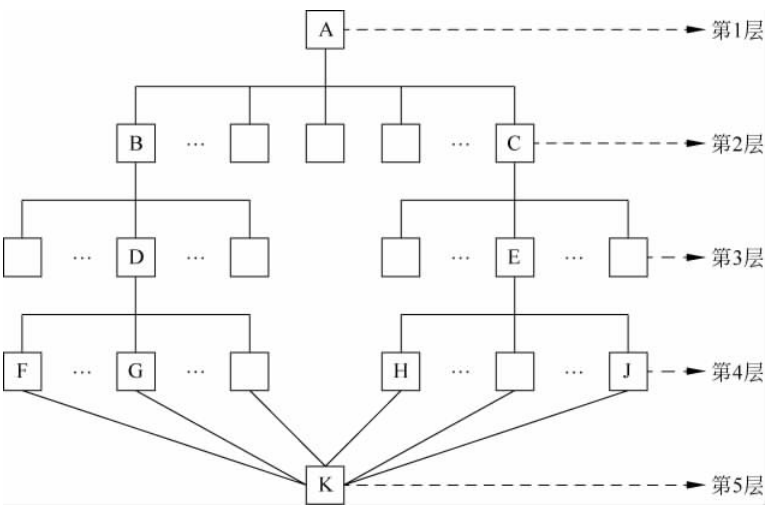

图 3-18　正交软件体系结构框架

图 3-18 是一个三级线索、五层结构的正交软件体系结构框架图,在该图中,ABDFK 组成了一条线索,ACEJK 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 5 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。

在软件演化过程中,系统需求会不断发生变化。在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。

3.6.1　正交软件体系结构的抽象模型

在本节中,将继续使用第 2 章对构件的定义,在此不再重复。

定义 1　连接件 $L=<R,G>$。这里 R 是连接件中角色进程所组成的集合,通常集合中有等于或多于两个的元素; G 是胶水(Glue)的集合,该集合中只有一个元素。

定义 2　称两个构件 C_i 和 $C_j(j\neq i)$是相关的,如果存在连接件 L,使得 C_i 与 C_j 之间有连接,记作 $C_i \overset{L}{\longleftrightarrow} C_j$。否则,就称 C_i 和 C_j 是无关的。

定义 3　线索由构件和与这些构件相关的连接件组成,记作 Th。

$\mathrm{Th}=\{<C_i,C_{i+1}>, C_i$ 是构件,且 $\exists L_j$,使得 $C_i \overset{L_j}{\longleftrightarrow} C_{i+1}$ 成立,$0<i\leqslant n\}$。其中,n 称为软件体系结构的层数。

定义 4　称线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$是相交的,如果存在构件 C,使得 $C\in \mathrm{Th}_i$ 且 $C\in \mathrm{Th}_j$

成立。把所有这样的 C 的集合记作 S_{ij}，即 $S_{ij}=\mathrm{Th}_i \cap \mathrm{Th}_j$。

定义 5　软件体系结构 SA 的所有线索组成一个集合，称该集合为线索空间。表示为 $\mathrm{TS}=(\mathrm{Th}_1,\mathrm{Th}_2,\cdots,\mathrm{Th}_m)$，其中 m 称为线索空间的维数，也就是一个软件体系结构中包含的垂直线索的条数。

定义 6　称构件 C 与线索 Th 是正交的，如果对 $\forall C_i \in \mathrm{Th}$，$C$ 和 C_i 都是无关的。记作 $C \perp \mathrm{Th}$。

定义 7　称线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$是正交的，如果 Th_i 和 Th_j 不相交，即 $\mathrm{Th}_i \cap \mathrm{Th}_j=\phi$，且线索 Th_i 的所有构件都与 Th_j 正交，线索 Th_j 的所有构件都与 Th_i 正交。即对 $\forall C_i \in \mathrm{Th}_i$，有 $C_i \perp \mathrm{Th}_j$ 且 $\forall C_j \in \mathrm{Th}_j$，有 $C_j \perp \mathrm{Th}_i$。如果线索 Th_i 和 Th_j 正交，记作 $\mathrm{Th}_i \perp \mathrm{Th}_j$。

定义 8　称线索空间 TS 为正交线索空间，如果对于 TS 的任意两条线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$，都有 $\mathrm{Th}_i \perp \mathrm{Th}_j$ 成立。

定义 9　称软件体系结构 SA 是正交软件体系结构，如果组成 SA 的所有线索的集合组成一个正交线索空间。

定义 10　一个软件体系结构 SA 的所有构件按下列方式排列：属于同一条线索的构件排在同一列，属于不同线索的构件排在不同的列。如此排列之后，以构件最多的那一列为标准，设为 m，不足 m 个构件的列补充 NULL（空构件），则这种排列组成一个 $m\times n$ 矩阵，其中 n 是 SA 的线索的条数。把这个矩阵称为 SA 的构件矩阵，记作 $\boldsymbol{M}_{\mathrm{SA}}$，表示为：

$$\boldsymbol{M}_{\mathrm{SA}}=\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ C_{i1} & C_{i2} & \cdots & C_{in} \\ N & C_{(i+1)2} & \cdots & C_{(i+1)n} \\ N & N & \cdots & C_{(i+2)n} \\ \vdots & \vdots & \vdots & \vdots \\ N & N & \cdots & C_{mn} \end{pmatrix}$$

由 $\boldsymbol{M}_{\mathrm{SA}}$的表示可以看出，一个软件体系结构 SA 的构件矩阵在经过矩阵的若干次列变换后，最终会变换成一个非严格意义的上三角矩阵。

约定　NULL 构件与任意构件正交，即对任意的构件 C_i，都有 $\mathrm{NULL} \perp C_i$ 成立。

定义 11　一个构件矩阵 $\boldsymbol{M}_{\mathrm{SA}}$被称为正交矩阵，如果对于任意两列元素组成的线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$，都有 $\mathrm{Th}_i \perp \mathrm{Th}_j$ 成立。

定理 1　在正交矩阵 $\boldsymbol{M}_{\mathrm{SA}}$中，任意两个元素 C_{ij} 和 $C_{kl}(j\neq l)$都是无关的。

证明：用反证法。假设在 $\boldsymbol{M}_{\mathrm{SA}}$中，存在两个元素 C_{ij} 和 $C_{kl}(j\neq l)$，且 C_{ij} 和 C_{kl} 相关。因为 $j\neq l$，根据定义 11，在体系结构 SA 中，C_{ij} 和 C_{kl} 分别属于两条不同的线索 Th_j 和 Th_l。再根据定义 8，可知 Th_j 和 Th_l 不是正交的，这与正交矩阵的定义矛盾。

3.6.2　软件体系结构的正交化

在实际的应用系统中，并不是所有的软件体系结构都是正交结构，为了使用正交结构的优点，必须对这些非正交软件体系结构进行正交化。

定义 12　对任意两条线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$，如果 $\exists C_i \in \mathrm{Th}_i - S_{ij}$，$C_j \in \mathrm{Th}_j - S_{ij}$，且

$\exists L$，使得 $C_i \overset{L}{\longleftrightarrow} C_j$ 成立，则可把这两个构件进行改造为 $C_{i'}$ 与 $C_{j'}$，使 $C_{i'}$ 与 $C_{j'}$ 无关。这个过程称为正交化过程。

定义 13　关系 R 称为相关关系，如果 R 由且仅由以下规则生成：

(1) 如果 C_i 和 C_j 是软件体系结构 SA 的任意两个构件，且 C_i 和 C_j 是相关的，则$<C_i, C_j>\in R$；

(2) 对 SA 的任意构件 C_i，都有$<C_i, C_i>\in R$；

(3) 对 SA 的任意两个构件 C_i, C_j，如果$<C_i, C_j>\in R$，则$<C_j, C_i>\in R$；

(4) 对 SA 的任意三个构件 C_i, C_j 和 $C_k (i\neq j\neq k)$，如果$<C_i, C_j>\in R$ 且$<C_j, C_k>\in R$，则$<C_i, C_k>\in R$。

定理 2　相关关系 R 是一个等价关系。

根据定义 13，定理 2 的结论是显然的。

由于 R 是一个等价关系，对于软件体系结构 SA 的构件集合 $C=\{C_1, C_2, \cdots, C_n\}$，可以构造 C 关于 R 的商集 $C/R=\{\mathrm{Cr}_1, \mathrm{Cr}_2, \cdots, \mathrm{Cr}_n\}$，$C/R$ 构成 C 的一个划分，$\mathrm{Cr}_i \subseteq C (i=1, 2, \cdots, n)$。商集 C/R 是构件集合 C 上关于 R 的不同等价类的集合。因此，不同划分块中的构件之间无相关关系，同一划分块中的构件之间存在相关关系。对每一划分块 Cr_i，定义一个线索 $\mathrm{Th}_i = C_{i1} C_{i2}, \cdots, C_{ik}$（对不同的 Th_i，k 值各不相同），且 $C_{ij} \in \mathrm{Cr}_i (j=1, 2, \cdots, k)$，$i1 < i2 < \cdots < ik$，即 Th_i 中的构件序列不改变原构件连接的顺序。这样获得的线索 Th_1，$\mathrm{Th}_2, \cdots, \mathrm{Th}_n$ 之间无相关关系，它们是正交的。于是，体系结构 SA 就可正交化为线索集 $\{\mathrm{Th}_1, \mathrm{Th}_2, \cdots, \mathrm{Th}_n\}$。

算法（软件体系结构正交化）

输入：非正交软件体系结构 SA

输出：正交软件体系结构 SA′

BEGIN

构造 SA 中构件的相关关系 R；

构造 SA 中构件集合 C 关于 R 的商集 C/R，得划分块的集合$\{\mathrm{Cr}_1, \mathrm{Cr}_2, \cdots, \mathrm{Cr}_n\}$；

将每一个划分块定义为一个线索，线索中不改变原构件连接的顺序，得正交

线索集合 $\mathrm{TS}=(\mathrm{Th}_1, \mathrm{Th}_2, \cdots, \mathrm{Th}_m)$；

END

软件体系结构和线索的正交化过程是需要工作量的，且正交化之后的整个线索空间的冗余度会增加。所以，在一个实际系统的设计和开发过程中，要根据正交化所需的工作量和系统冗余度与性能改进之间的关系，来确定是否进一步进行正交化。

3.6.3　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

在设计初期，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,设计师使用了多级正交软件体系结构的设计思想。在本系统中,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,使得既能对功能进行分类,又能在每一类中对功能进行细分。使功能划分既有序,又合理,能控制在一定粒度以内,合理的粒度又为线索和层次中构件的实现打下良好的基础。

在 3.6.1 节提到的完全正交结构不能很好地适用于本应用,因此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。进一步,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 3-19 所示,每个主线索又可划分为数个子线索(≥2)。

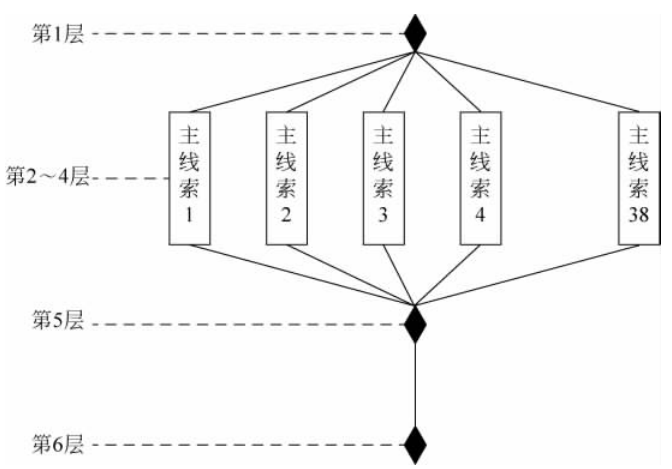

图 3-19　系统总体正交体系结构设计

为了简单起见,下面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 3-20 所示。

在图 3-20 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 3-19 中的第 1、第 5 和第 6 层。组合图 3-19 和图 3-20 可看出整个 MIS 的结构包括 6 个层次:

(1) 第 1 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 2 层实现菜单接口,支持用户选择不同的处理功能。

(3) 第 3 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 4 层是真正的功能定义,在这定义的构件有:数据录入构件(包括插入、删除、更

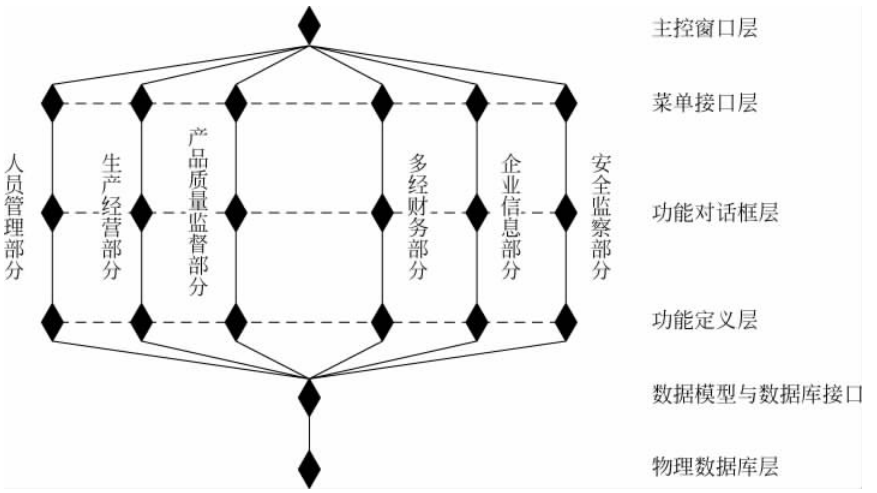

图 3-20　多种经营主线索子正交结构图

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。

(5) 第 5 层和第 6 层是数据服务的实现,第 5 层是包括特定的数据模型和数据库接口,第 6 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,由于采用正交结构的思想,可以分数个小组并行开发。每个小组分配一条或数条线索,由专门一个小组来设计通用共享构件。由于构件是通用的,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,从而大大提高了编程效率,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. 演化控制

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,按照在第 1 部分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,财务管理子线索有如下变动,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(对它的功能定义层作上修改标记),其他一些子功能(子线索)也需增加自动上报功能,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 3-21 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作标记,其他未作标记的则表示可直接重用的构件,确定演化的结构图之后,按照自顶向下,由左至右的原则,更新演化工作得以有条不紊地进行。

现对多经主线索子正交结构演化情况进行统计: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

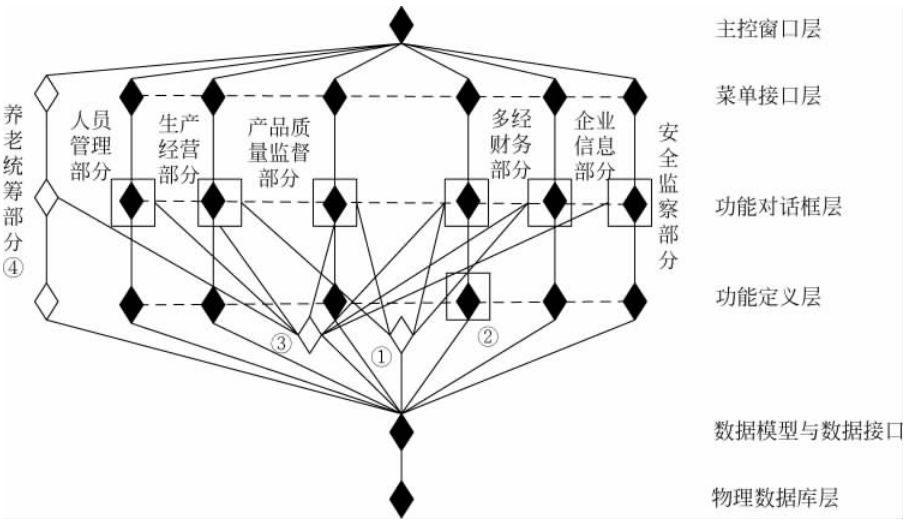

图 3-21 多经主线索结构变动情况图

线索一条。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,但是大部分只用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3-1是对工作量(每天8h,单位人/天)的统计分析。

38个主线索的修改与开发工作量比例均未超过30%,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演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 3-1 劳动量比较表

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	2	3.33%
6	60	3	5.00%
⋮	⋮	⋮	⋮
37	60	2	3.33%
38	30	3	10%

3.6.4 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,构件在结构图中的位置已经说明它所实现的是哪一

级抽象，担负的是什么功能。

（2）易修改，可维护性强。由于线索之间是相互独立的，所以对一个线索的修改不会影响到其他线索。因此，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，只需相应的增删线索构件族，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。因为正交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。

3.7　基于层次消息总线的体系结构风格

层次消息总线（Hierarchy Message Bus，HMB）体系结构风格是由北京大学杨芙清院士等人提出的一种风格，该体系结构风格的提出基于以下的实际背景：

（1）随着计算机网络技术的发展，特别是分布式构件技术的日渐成熟和构件互操作标准的出现，如 CORBA、DCOM 和 EJB 等，加速了基于分布式构件的软件开发趋势，具有分布和并发特点的软件系统已成为一种广泛的应用需求。

（2）基于事件驱动的编程模式已在图形用户界面程序设计中获得广泛应用。在此之前的程序设计中，通常使用一个大的分支语句控制程序的转移，对不同的输入情况分别进行处理，程序结构不甚清晰。基于事件驱动的编程模式在对多个不同事件响应的情况下，系统自动调用相应的处理函数，程序具有清晰的结构。

（3）计算机硬件体系结构和总线的概念为软件体系结构的研究提供了很好的借鉴和启发，在统一的体系结构框架下（即总线和接口规范），系统具有良好的扩展性和适应性。任何计算机厂商生产的配件，甚至是在设计体系结构时根本没有预料到的配件，只要遵循标准的接口规范，都可以方便地集成到系统中，对系统功能进行扩充，甚至是即插即用（即运行时刻的系统演化）。正是标准的总线和接口规范的指定，以及标准化配件的生产，促进了计算机硬件的产业分工和蓬勃发展。

HMB 风格基于层次消息总线、支持构件的分布和并发，构件之间通过消息总线进行通信，如图 3-22 所示。

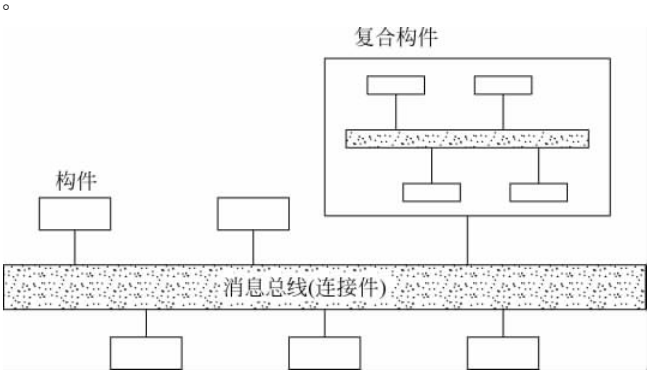

图 3-22　HMB 风格的系统示意图

消息总线是系统的连接件,负责消息的分派、传递和过滤以及处理结果的返回。各个构件挂接在消息总线上,向总线登记感兴趣的消息类型。构件根据需要发出消息,由消息总线负责把该消息分派到系统中所有对此消息感兴趣的构件,消息是构件之间通信的唯一方式。构件接收到消息后,根据自身状态对消息进行响应,并通过总线返回处理结果。由于构件通过总线进行连接,并不要求各个构件具有相同的地址空间或局限在一台机器上。该风格可以较好地刻画分布式并发系统,以及基于 CORBA、DCOM 和 EJB 规范的系统。

如图 3-22 所示,系统中的复杂构件可以分解为比较低层的子构件,这些子构件通过局部消息总线进行连接,这种复杂的构件称为复合构件。如果子构件仍然比较复杂,可以进一步分解,如此分解下去,整个系统形成了树状的拓扑结构,树结构的末端结点称为叶结点,它们是系统中的原子构件,不再包含子构件,原子构件的内部可以采用不同于 HMB 的风格,例如前面提到的数据流风格、面向对象风格及管道-过滤器风格等,这些属于构件的内部实现细节。但要集成到 HMB 风格的系统中,必须满足 HMB 风格的构件模型的要求,主要是在接口规约方面的要求。另外,整个系统也可以作为一个构件,通过更高层的消息总线,集成到更大的系统中。于是,可以采用统一的方式刻画整个系统和组成系统的单个构件。

下面讨论 HMB 风格中的各个组成要素。

3.7.1 构件模型

系统和组成系统的成分通常是比较复杂的,难以从一个视角获得对它们的完整理解,因此一个好的软件工程方法往往从多个视角对系统进行建模,一般包括系统的静态结构、动态行为和功能等方面。例如,OMT 方法采用了对象模型、动态模型和功能模型刻画系统的以上三个方面。

借鉴以上思想,为满足体系结构设计的需要,HMB 风格的构件模型包括接口、静态结构和动态行为三个部分,如图 3-23 所示。

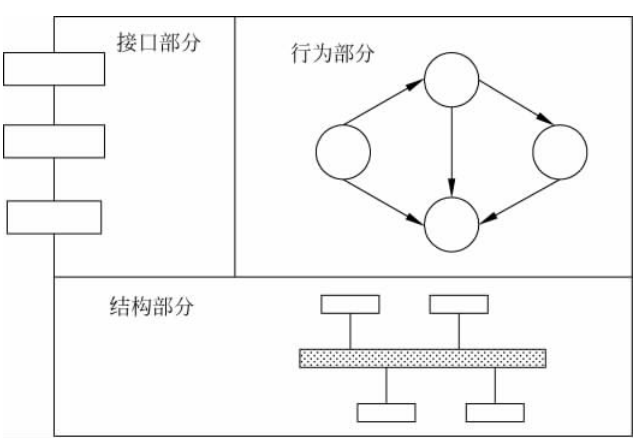

图 3-23 HMB 风格的构件模型

在图 3-23 中,左上方是构件的接口部分,一个构件可以支持多个不同的接口,每个接口定义了一组输入和输出的消息,刻画了构件对外提供的服务以及要求的环境服务,体现了该构件同环境的交互。右上方是用带输出的有限状态自动机刻画的构件行为,构件接收到外来消息后,根据当前所处的状态对消息进行响应,并可能导致状态的变迁。下方是复合构件

的内部结构定义，复合构件是由更简单的子构件通过局部消息总线连接而成的。消息总线为整个系统和各个层次的构件提供了统一的集成机制。

3.7.2　构件接口

在体系结构设计层次上，构件通过接口定义了同外界的信息传递和承担的系统责任，构件接口代表了构件同环境的全部交互内容，也是唯一的交互途径。除此之外，环境不应对构件做任何其他与接口无关的假设，例如实现细节等。

HMB 风格的构件接口是一种基于消息的互联接口，可以较好地支持体系结构设计。构件之间通过消息进行通信，接口定义了构件发出和接收的消息集合。同一般的互联接口相比，HMB 的构件接口具有两个显著的特点。首先，构件只对消息本身感兴趣，并不关心消息是如何产生的，消息的发出者和接收者不必知道彼此的情况，这样就切断了构件之间的直接联系，降低了构件之间的耦合程度，进一步增强了构件的重用潜力，并使得构件的替换变得更为容易。另外，在一般的互联接口定义的系统中，构件之间的连接是在要求的服务和提供的服务之间进行固定的匹配，而在 HMB 的构件接口定义的系统中，构件对外来消息进行响应后，可能会引起状态的变迁。因此，一个构件在接收到同样的消息后，在不同时刻所处的不同状态下，可能会有不同的响应。

消息是关于某个事件发生的信息，上述接口定义中的消息分为两类：

(1) 构件发出的消息，通知系统中其他构件某个事件的发生或请求其他构件的服务。

(2) 构件接收的消息，对系统中某个事件的响应或提供其他构件所需的服务。接口中的每个消息定义了构件的一个端口，具有互补端口的构件可以通过消息总线进行通信，互补端口指的是除了消息进出构件的方向不同之外，消息名称、消息带有的参数和返回结果的类型完全相同的两个端口。

当某个事件发生后，系统或构件发出相应的消息，消息总线负责把该消息传递到此消息感兴趣的构件。按照响应方式的不同，消息可分为同步消息和异步消息。同步消息是指消息的发送者必须等待消息处理结果返回才可以继续运行的消息类型。异步消息是指消息的发送者不必等待消息处理结果的返回即可继续执行的消息类型。常见的同步消息包括过程调用，异步消息包括信号、时钟和异步过程调用等。

3.7.3　消息总线

HMB 风格的消息总线是系统的连接件，构件向消息总线登记感兴趣的消息，形成构件消息响应登记表。消息总线根据接收到的消息类型和构件-消息响应登记表的信息，定位并传递该消息给相应的响应者，并负责返回处理结果。必要时，消息总线还对特定的消息进行过滤和阻塞。图 3-24 给出了采用对象类符号表示的消息总线的结构。

1. 消息登记

在基于消息的系统中，构件需要向消息总线登记当前响应的消息集合，消息响应者只对消息类型感兴趣，通常并不关心是谁发出的消息。在 HMB 风格的系统中，对挂接在同一消息总线上的构件而言，消息是一种共享的资源，构件-消息响应登记表记录了该总线上所有构件和消息的响应关系。类似于程序设计中的“间接地址调用”，避免了将构件之间的连接“硬编码”到构件的实例中，使得构件之间保持了灵活的连接关系，便于系统的演化。

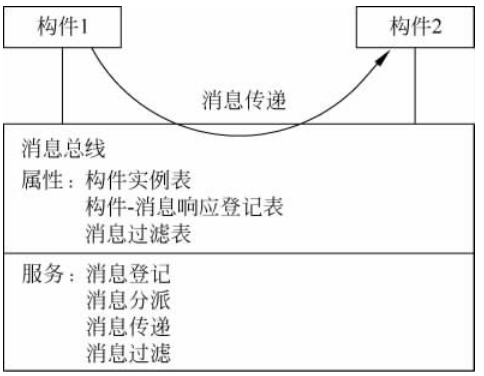

图 3-24 消息总线的结构

构件接口中的接收消息集合意味着构件具有响应这些消息类型的潜力，默认情况下，构件对其接口中定义的所有接收消息都可以进行响应。但在某些特殊的情况下，例如，当一个构件在部分功能上存在缺陷时，就难以对其接口中定义的某些消息进行正确的响应，这时应阻塞那些不希望接收到的消息。这就是需要显式进行消息登记的原因，以便消息响应者更灵活地发挥自身的潜力。

2. 消息分派和传递

消息总线负责消息在构件之间的传递，根据构件-消息响应登记表把消息分派到对此消息感兴趣的构件，并负责处理结果的返回。在消息广播的情况下，可以有多个构件同时响应一个消息，也可以没有构件对该消息进行响应。在后一种情况下，该消息就丢失了，消息总线可以对系统的这种异常情况发出警告，或通知消息的发送构件进行相应的处理。

实际上，构件-消息响应登记表定义了消息的发送构件和接收构件之间的一个二元关系，以此作为消息分派的依据。

消息总线是一个逻辑上的整体，在物理上可以跨越多个机器，因此挂接在总线上的构件也就可以分布在不同的机器上，并发运行。由于系统中的构件不是直接交互，而是通过消息总线进行通信，因此实现了构件位置的透明性。根据当前各个机器的负载情况和效率方面的考虑，构件可以在不同的物理位置上透明地迁移，而不影响系统中的其他构件。

3. 消息过滤

消息总线对消息过滤提供了转换和阻塞两种方式。消息过滤的原因主要在于不同来源的构件事先并不知道各自的接口，因此可能同一消息在不同的构件中使用了不同的名字，或不同的消息使用了相同的名字。前面提到，对挂接在同一消息总线上的构件而言，消息是一种共享的资源，这样就会造成构件集成时消息的冲突和不匹配。

消息转换是针对构件实例而言的，即所有构件实例发出和接收的消息类型都经过消息总线的过滤，这里采取简单换名的方法，其目标是保证每种类型的消息名字在其所处的局部总线范围内是唯一的。例如，假设复合构件 A 复合客户/服务器风格，由构件 C 的两个实例 c1 和 c2 以及构件 S 的一个实例 s1 构成，构件 C 发出的消息 msgC 和构件 S 接收的消息 msgS 是相同的消息。但由于某种原因，它们的命名并不一致(除此之外，消息的参数和返回值完全一样)。可以采取简单换名的方法，把构件 C 发出的消息 msgC 换名为 msgS，这样无须对构件进行修改，就解决了这两类构件的集成问题。

由简单的换名机制解决不了的构件集成的不匹配问题，例如参数类型和个数不一致等，可以采取更为复杂的包装器(wrapper)技术对构件进行封装。

3.7.4　构件静态结构

HMB 风格支持系统自顶向下的层次化分解，复合构件是由比较简单的子构件组装而成的，子构件通过复合构件内部的消息总线连接，各个层次的消息总线在逻辑功能上是一致的，负责相应构件或系统范围内消息的登记、分派、传递和过滤。如果子构件仍然比较复杂，可以进一步分解。图 3-25 是某个系统经过逐层分解所呈现出的结构示意图，不同的消息总线分别属于系统和各层次的复合构件，消息总线之间没有直接的连接，HMB 风格中的这种总线称为层次消息总线。另外，整个系统也可以作为一个构件，集成到更大的系统中。因为各个层次的构件以及整个系统采取了统一的方式进行刻画，所以定义一个系统的同时也就定义了一组“系统”，每个构件都可看作一个独立的子系统。

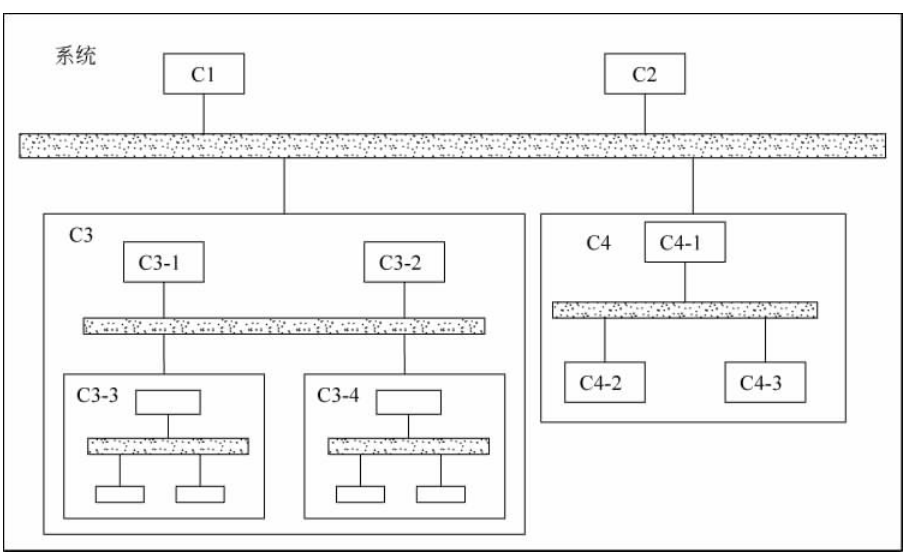

图 3-25　系统/复合构件的静态结构示意图

3.7.5　构件动态行为

在一般的基于事件风格的系统中，如图形用户界面系统 X-Window，对于同一类事件，构件(这里指的是回调函数)总是采取同样的动作进行响应。这样，构件的行为就由外来消息的类型唯一确定，即一个消息和构件的某个操作之间存在着固定的对应关系。对于这类构件，可以认为构件只有一个状态，或者在每次对消息响应之前，构件处于初始状态。虽然在操作的执行过程中，会发生状态的变迁，但在操作结束之前，构件又恢复到初始状态。无论以上哪种情况，都不需要构件在对两个消息响应之间，保持其状态信息。

更通常的情况是，构件的行为同时受外来消息类型和自身当前所处状态的影响。类似一些面向对象方法中用状态机刻画对象的行为，在 HMB 风格的系统中，采用带输出的有限状态机描述构件的行为。

带输出的有限状态机分为 Moore 机和 Mealy 机两种类型，它们具有相同的表达能力。

在一般的面向对象方法中,通常混合采用 Moore 机和 Mealy 机表达对象的行为。为了实现简单起见,选择采用 Mealy 机来描述构件的行为。一个 Mealy 机包括一组有穷的状态集合、状态之间的变迁和在变迁发生时的动作。其中,状态表达了在构件的生命周期内,构件所满足的特定条件、实施的活动或等待某个事件的发生。

3.7.6 运行时刻的系统演化

在许多重要的应用领域中,例如金融、电力、电信及空中交通管制等,系统的持续可用性是一个关键性的要求,运行时刻的系统演化可减少因关机和重新启动而带来的损失和风险。此外,越来越多的其他类型的应用软件也提出了运行时刻演化的要求,在不必对应用软件进行重新编译和加载的前提下,为最终用户提供系统定制和扩展的能力。

HMB 风格方便地支持运行时刻的系统演化,主要体现在动态增加或删除构件、动态改变构件响应的消息类型、消息过滤等三个方面。

1. *动态增加或删除构件*

在 HMB 风格的系统中,构件接口中定义的输入和输出消息刻画了一个构件承担的系统责任和对外部环境的要求,构件之间通过消息总线进行通信,彼此并不知道对方的存在。因此只要保持接口不变,构件就可以方便地替换。一个构件加入到系统中的方法很简单,只需向系统登记其所感兴趣的消息即可。但删除一个构件可能会引起系统中对于某些消息没有构件响应的异常情况,这时可以采取两种措施:一是阻塞那些没有构件响应的消息,二是首先使系统中的其他构件或增加新的构件对该消息进行响应,然后再删除相应的构件。

系统中可能增删改构件的情况包括:

(1) 当系统功能需要扩充时,往系统中增加新的构件。

(2) 当对系统功能进行裁减,或当系统中的某个构件出现问题时,需要删除系统中的某个构件。

(3) 用带有增强功能或修正了错误的构件新版本代替原有的旧版本。

2. *动态改变构件响应的消息类型*

类似地,构件可以动态地改变对外提供的服务(即接收的消息类型),这时应通过消息总线对发生的改变进行重新登记。

3. *消息过滤*

利用消息过滤机制,可以解决某些构件集成的不匹配问题。消息过滤通过阻塞构件对某些消息的响应,提供了另一种动态改变构件对消息进行响应的方式。

3.8 异构结构风格

前面几节的内容介绍和讨论了一些所谓的"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:

(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、通信以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,但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)的基本定义等。

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,很有可能发现它们不会很容易组织在一起。通常问题出在:各构件作了数据表示、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。可以举一个简单的例子,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。

3.8.1　异构结构的实例分析

本节将通过实例讨论 C/S 与 B/S 混合软件体系结构。从 3.2 节、3.3 节和 3.4 节的讨论中可以看出,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,B/S 体系结构要想在软件开发中起主导作用,要走的路还很长。作者认为,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 混合软件体系结构的两个模型。

1. "内外有别"模型

在 C/S 与 B/S 混合软件体系结构的"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 Internet 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"模型的结构如图 3-26 所示。

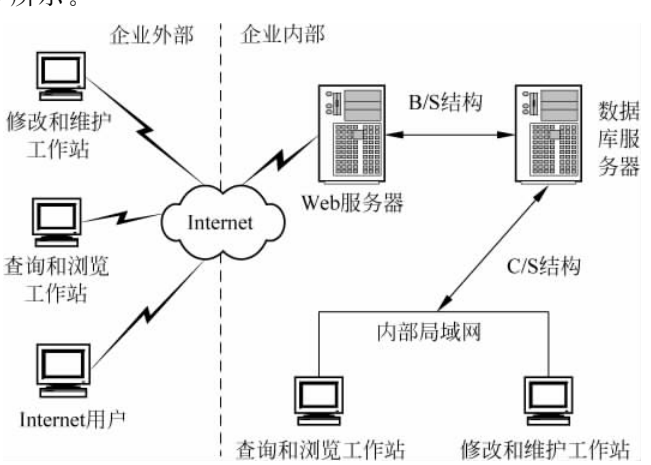

图 3-26　"内外有别"模型

“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。

“内外有别”模型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。

2.“查改有别”模型

在C/S与B/S混合软件体系结构的“查改有别”模型中,不管用户是通过什么方式(局域网或Internet)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”模型的结构如图3-27所示。

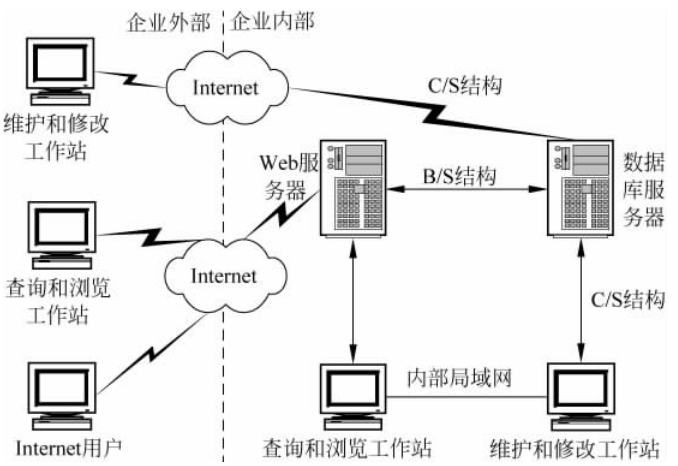

图3-27 “查改有别”模型

“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。但因为外部用户能直接通过Internet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。

3.几点说明

(1)因为在本节中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Internet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、ADSL,也可以使用DDN方式等。

(3)本节对内部与外部的区分,是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。

4.应用实例

当前,我国电力系统正在进行精简机构的改革,变电站也在朝着无人、少人和一点带面的方向发展(如,一个有人值班220kV变电站带若干个无人值班220kV和110kV变电站),

“减人增效”是必然的趋势，而要很好地达到这个目的，使用一套完善的变电综合信息管理系统（以下简称为 TSMIS）显得很有必要。为此，作者曾组织有关力量，针对电力系统变电运行管理工作的需要，结合变电站运行工作经验，开发了一套完整的变电综合信息管理系统。

1）体系结构设计

在设计 TSMIS 系统时，充分考虑到变电站分布管理的需要，采用 C/S 与 B/S 混合软件体系结构的“内外有别”模型，如图 3-28 所示。

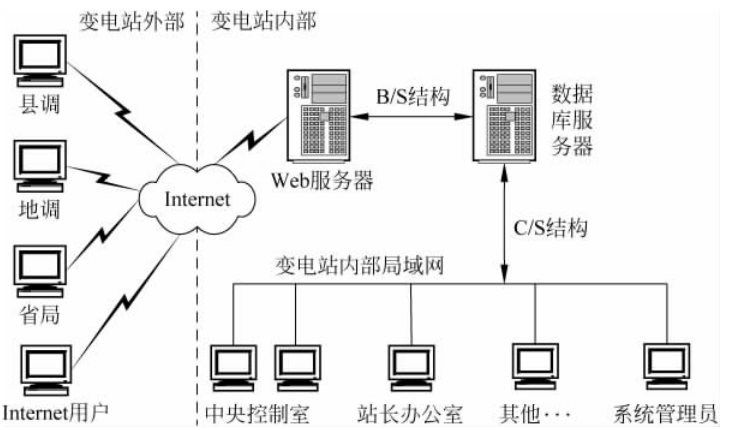

图 3-28　TSMIS 系统软件体系结构

在 TSMIS 系统中，变电站内部用户通过局域网直接访问数据库服务器，外部用户（包括县调、地调和省局的用户及普通 Internet 用户）通过 Internet 访问 Web 服务器，再通过 Web 服务器访问数据库服务器。外部用户只需一台接入 Internet 的计算机，就可以通过 Internet 查询运行生产管理情况，无须做太大的投入和复杂的设置。这样也方便所属电业局及时了解各变电站所的运行生产情况，对各变电站的运行生产进行宏观调控。此设计能很好地满足用户的需求，符合可持续发展的原则，使系统有较好的开放性和易扩展性。

2）系统实现

TSMIS 系统包括变电运行所需的运行记录、图形开票、安全生产管理、生产技术管理、行政管理、总体信息管理、技术台账管理、班组建设、学习培训、系统维护等各个业务层次模块。实际使用时，用户可以根据实际情况的需要选择模块进行自由组合，以达到充分利用变电站资源和充分发挥系统作用的目的。

系统的实现采用 Visual C++、Visual Basic 和 Java 等语言和开发平台进行混合编程。服务器操作系统使用 Windows 2003 Advanced Server，后台数据库采用 SQL Server 2005。系统的实现充分考虑到我国变电站所电压等级的分布，可以适用于大、中、小电压等级的变电站所。

3.8.2　异构组合匹配问题

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。基本形式如图3-29所示。

图3-29 构件协调问题

A和B不能协调工作的原因可能是它们事先作了对数据表示、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(form)。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,Macintosh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。它们有两种重要形式。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add-ons,它们可以完成内外部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的时候特别有用。

第二,标准的发布形式,如RTF、MIF、Postscript;或Adobe Acrobat提供了另一种可供选择的方式,一种广泛流通的安全的表示法。

第三,活跃的调解者(active media)可以被插入系统,作为中介。

(8) 在A上添加一个适配器(adapter)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。

以上这些技术有它们的优点和缺点,它们在初始化、时间和空间效率、灵活性和绝对的处理能力上差别很大。

3.9　互连系统构成的系统及其体系结构

开发大规模系统时，其复杂程度将大大增加。它不但要求开发人员能够理解一套更为复杂的流程，并且由于需要管理更多的资源，为此要负担额外的开销。互连系统构成的系统(System of Interconnected Systems，SIS)是由 Herbert H. Simon 在 1981 年提出的一个概念。1995 年，Jacobson 等人对这种系统进行了专门的讨论。

3.9.1　互连系统构成的系统

SIS 是指系统可以分成若干个不同的部分，每个部分作为单独的系统独立开发。整个系统通过一组互连系统实现，而互连系统之间相互通信，履行系统的职责。其中一个系统体现整体性能，称为上级系统(superordinate system)；其余系统代表整体的一个部分，称为从属系统(subordinate system)。从上级系统的角度来看，从属系统是子系统，上级系统独立于其从属系统，每个从属系统仅仅是上级系统模型中所指定内容的一个实现，并不属于上级系统功能约束的一部分。互连系统构成的系统的软件体系结构(Software Architecture for SIS，SASIS)如图 3-30 所示。

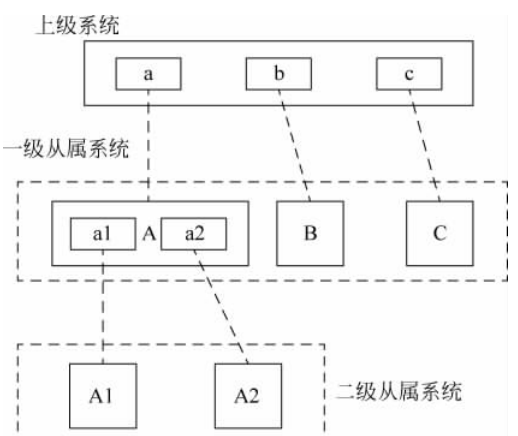

图 3-30　SIS 的体系结构

图 3-30 是一个三级结构的 SIS 体系结构示意图，上级系统的功能约束由一个互连系统构成的系统来实现，其中一级从属系统 A、B 和 C 分别是上级系统的子系统 a、b 和 c 的具体实现。二级从属系统中的 A1 和 A2 分别是一级从属系统 A 的子系统 a1 和 a2 的具体实现。

在 SASIS 中，上级系统与从属系统是相对而言的。例如，在图 3-30 中，一级从属系统既是上级系统的从属系统，同时又是二级从属系统的上级系统。

需要说明的是，并不是每一个 SIS 系统都只能分解为三级结构。需要根据系统的规模来进行分解，可以只分解为两级结构。如果要开发的系统规模很大，可能需要进一步划分从属系统，形成多级结构的 SIS 系统。

从属系统可以自成一个软件系统,脱离上级系统而运行,有其自己的软件生命周期,在生命周期内的所有活动中都可以单独管理,可以使用不同的开发流程来开发各个从属系统。

常用的系统开发过程也可应用于SIS系统,可以将上级系统与从属系统的实现分离。通过把从属系统插入到由互连系统构成的其他系统,就很容易使用从属系统来实现其上级系统。

在SIS系统中,要注意各从属系统之间的独立性,在上级系统的设计模型中,每个从属系统实现一个子系统。子系统依赖于彼此的接口,但相互之间是相对独立的。因此,当某从属系统的需求发生变化时,不必开发新版本的上级系统,只需对该从属系统内部构件进行变更,不会影响到其他从属系统。只有在主要功能变更时才要求开发上级系统的新版本。

3.9.2 基于SASIS的软件过程

在SIS系统中,首先开始的是上级系统的软件过程(software process,有关软件过程的详细介绍见第12章)。一旦上级系统经历过至少一次迭代而且从属系统的接口相对稳定,从属系统的软件过程即可开始。上级系统和从属系统可以使用相同的软件过程来进行开发。其软件过程如图3-31所示。

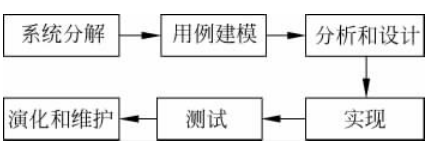

图3-31 基于SASIS的软件过程

1. 系统分解

在开发基于SASIS的系统时,需要做的工作是确定如何能够将一个上级系统的功能分布在几个从属系统之间,每个从属系统负责一个明确定义的功能子集。也就是说,主要目标是定义这些从属系统间的接口。实现上述主要目标之后,其余工作就可以根据“分而治之”的原则,由每个从属系统独立完成。

那么,在什么情况下,可以将系统分解为由互连系统构成的系统呢?一般根据系统的规模和复杂性来作决定,如果系统具有相当(并没有一个固定的标准)的规模和复杂性,则可以把问题分成许多小问题,这样更容易理解。另外,如果所要处理的系统可由若干个物理上独立的系统组成(例如,处理遗留系统),则也可分解为由互连系统构成的系统。

当对一个系统进行分解时,既可分成两级结构,也可分成多级结构。那么,分解到何种程度才能停止呢?要做到适度分解,体系结构设计师需要有丰富的实践经验。过度的分解会导致资源管理困难,项目难以协调。而且,不适度地使用物理上分离的系统或者物理上分离的团队,会在很大程度上扼杀任何形式的软件重用。

为了降低风险,做到适度分解,需要成立一个体系结构设计师小组,负责监督整个开发工作。体系结构设计师小组应该重点关注以下问题:

(1) 适当关注从属系统之间的重用和经验共享。

(2) 对开发什么工件(构件或文档)、从属系统工件与上级系统工件之间有什么关系有清晰的理解。

(3) 定义一个有效的变更管理策略,并得到所有团队的遵守。

2. 用例建模

开发人员应该为每个系统(包括上级系统和从属系统)在 SIS 系统中建立一个用例(use case,关于用例的细节问题将在第 5 章介绍)模型,上级系统的高级用例通常可以分解到(所有或部分)子系统上,每个"分块"将成为从属系统模型中的一个用例,如图 3-32 所示。

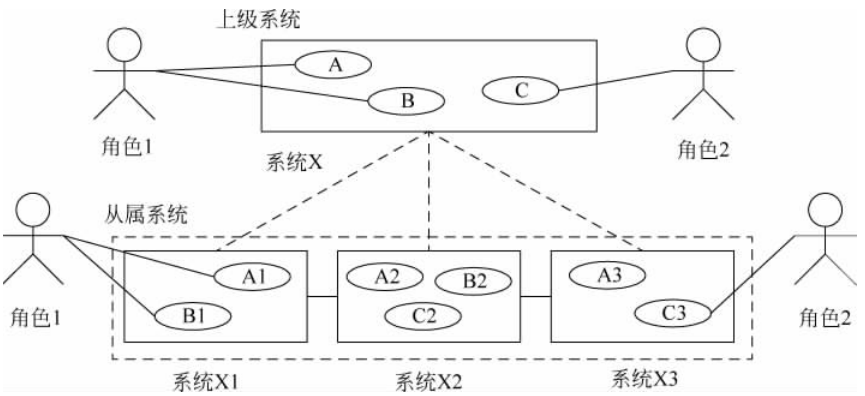

图 3-32　上级系统的高级用例与从属系统的详细用例之间的关系

在图 3-32 中,上级系统 X 有三个用例,分别为 A、B 和 C。其中,用例 A 分解到从属系统 X1 的分块为 A1,分解到从属系统 X2 的分块为 A2,分解到从属系统 X3 的分块为 A3;用例 B 只分解到从属系统 X1 和 X2(分别为 B1 和 B2);用例 C 只分解到从属系统 X2 和 X3(分别为 C2 和 C3)。

从任何一个从属系统的角度来看,其他从属系统都是该从属系统用例模型的角色(actor,有的文献翻译为"执行者"),如图 3-33 所示。在从属系统 X2 的用例模型中,从属系统 X1 和 X3 都被视为角色。一个从属系统把另一个从属系统的接口看作是由相关角色提供的,这意味着可以更换某个从属系统,只要新的从属系统对其他从属系统扮演的角色相同就可以了。例如,新的从属系统可以用相同的接口集来表示。

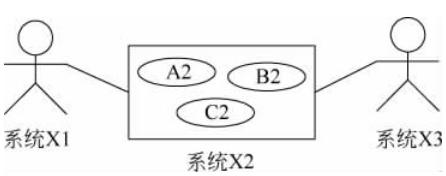

图 3-33　一个从属系统是另一个从属系统的角色

3. 分析和设计

分析和设计的目的是为了获得一个健壮的系统体系结构,这对 SIS 系统来说是极其重要的。SIS 中的每个系统,包括上级系统和从属系统,都应该定义自己的体系结构,选择相应的体系结构风格。分析设计的过程又可分为标识构件、选择体系结构风格、映射构件、分析构件相互作用和产生体系结构等 5 个子过程,将在第 12 章详细讨论这些过程。

对于上级系统,选择体系结构风格时应考虑上级系统的关键用例或场景(scenario)、SIS 的分级结构以及如何处理从属系统之间的重用和重用内容等问题。对于从属系统,选择体

系结构风格时应考虑从属系统在SIS系统内的角色、从属系统的关键用例或场景、从属系统如何履行在SIS系统的分级结构中为它定义的职责、如何重用等问题。

在决定SIS系统的体系结构时,还要综合考虑两个问题:

(1) 软件重用。要明确哪些构件是两个以上从属系统公有的,需要建立哪些机制来允许从属系统互相进行通信。

(2) 所有从属系统都能使用的构件及其实现。所有从属系统都应该使用通用的通信、错误报告、容错管理机制,提供统一的人机界面。

4. 实现

在从属系统的软件过程中,实现过程主要完成构件的开发和测试工作。在上级系统的软件过程中,不必经过实现过程,而只执行一些原型设计工作,研究系统特定方面的技术。

5. 测试

这里的测试指的是组装不同从属系统时的集成测试,并测试每个上级用例是否根据其规约通过互连系统协作获得执行。

6. 演化和维护

在SIS系统中,各从属系统之间是相对独立的。在上级系统的设计模型中,每个从属系统实现一个子系统。子系统依赖于彼此的接口,因此,当某从属系统的需求发生变化时,不必开发新版本的上级系统,只需对该从属系统内部构件进行变更,不会影响到其他从属系统。只有在主要功能变更时才要求开发上级系统的新版本。

3.9.3 应用范围

互连系统构成的系统的体系结构及其建模技术可以用于多种系统,如:

(1) 分布式系统。

(2) 很大或者很复杂的系统。

(3) 综合几个业务领域的系统。

(4) 重用其他系统的系统。

(5) 系统的分布式开发。

情况也可以反过来:从一组现有的系统,通过组装系统来定义由互连系统构成的系统。实际上,在某些情况下,大型系统在演化的早期阶段就是以这种方式发展的。如果开发人员意识到有几个可以互连的系统,那么让系统互连可以创建一个“大型”系统,它比两个独立的系统能创造更多的价值。

事实上,如果一个系统的不同部分本身可以看作系统,作者建议把它定义为由互连系统构成的系统。即使现在它还是单个的系统,由于分布式开发、重用或者客户只需购买它的几个部分等原因,把系统分割成几个独立的产品是有必要的。

1. 大规模系统

电话网络可能是世界上最大的由互连系统构成的系统。这是一个很好的示例,在电话网络中需要管理两个以上系统级别的复杂性。它还作为这类案例的示例:顶层上级系统由一个标准化实体掌握,不同的竞争公司开发一个或几个必须符合该标准的从属系统。这里将讨论移动电话网络GSM(全球移动电话系统),说明将大规模系统作为互连系统构成的系统来实施的优势。

大规模系统的功能通常包含几个业务领域。例如，GSM 标准包括从呼叫用户到被呼叫用户的整个系统。换句话说，它既包括移动电话的行为，也包括网络结点的行为。由于系统的不同部分是单独购买的产品本身，甚至是由不同客户购买的，因而它们本身可当作系统。例如，开发完整 GSM 系统的公司向用户销售移动电话，向话务员销售网络结点。这是把 GSM 系统的不同部分当作不同从属系统的一个原因。另一个原因是，把 GSM 这样大型复杂的系统作为单个系统进行开发的时间太长；不同的部分必须由几个开发团队并行开发。

另外，由于 GSM 标准包括整个系统，因此有理由将系统作为一个整体即上级系统来考虑。这有助于开发人员理解问题领域，以及不同部分是怎样彼此相关的。

2. 分布式系统

在分布在几个计算机系统的系统中，使用由互连系统构成的系统体系结构是非常合适的。顾名思义，分布式系统(distributed system)至少由两个部分组成。由于分布式系统中必须有明确定义的接口，因此这些系统也非常适合进行分布式开发，也就是说，几个独立的开发团队并行开发。分布式系统的从属系统本身甚至可以当作产品来销售。因而，将分布式系统视为独立系统的集合是很自然的。

分布式系统的需求通常包括整个系统的功能，有时不预先定义不同部分之间的接口。而且，如果对开发者而言问题领域是陌生的，他们首先需要考虑整个系统的功能，而不管它将如何分布。这是把它当作单个系统看待的两个重要原因。

3. 遗留系统的重用

许多企业因为业务发展的需要和市场竞争的压力，需要建设新的企业信息系统。在这种升级改造的过程中，怎么处理和利用那些历史遗留下来的老系统，成为影响新系统建设成功和开发效率的关键因素之一。一般称这些老系统为遗留系统(legacy system)。

目前，学术和工业界对遗留系统的定义没有统一的意见。Bennett 在 1995 年对遗留系统作了如下的定义：遗留系统是用户不知道如何处理但对用户的组织又是至关重要的系统。Brodie 和 Stonebraker 对遗留系统的定义如下：遗留系统是指任何基本上不能进行修改和演化以满足新的变化了的业务需求的信息系统。

作者认为，遗留系统应该具有以下特点：

(1) 系统虽然完成企业中许多重要的业务管理工作，但已经不能完全满足要求。一般实现业务处理电子化及部分企业管理功能，很少涉及经营决策。

(2) 系统在性能上已经落后，采用的技术已经过时。如多采用主机/终端形式或小型计算机系统，软件使用汇编语言或第三代程序设计语言的早期版本开发，使用文件系统而不是数据库。

(3) 通常是大型的软件系统，已经融入企业的业务运行和决策管理机制之中，维护工作十分困难。

(4) 系统没有使用现代软件工程方法进行管理和开发，现在基本上已经没有文档，很难理解。

在企业信息系统升级改造过程中，如何处理和利用遗留系统，成为新系统建设的重要组成部分。处理恰当与否，直接关系到新系统的成败和开发效率高低。

大型系统常常重用遗留系统，遗留系统可作为从属系统来描述。为了理解遗留系统如何在上级系统的大环境中工作，需要为它“重建”一个用例模型，有时还有一个分析模型。这

些重建的模型不必完整,但至少要包含对互连系统构成的系统的其余部分有直接影响的遗留功能,否则需要进行修改。

总之,在大规模系统的软件开发中,采用SASIS可以处理相当复杂的问题,且具有以下优点:

(1) 使用"分而治之"的原则来理解系统,能有效地降低系统的复杂性。

(2) 因为各从属系统相对独立,自成系统,提高了系统开发的并行性。

(3) 当某从属系统的需求发生变化时,不必开发新版本的上级系统,只需对该从属系统内部构件进行变更,提高了系统的可维护性。

然而,SASIS也有固有的缺点,如资源管理开销增大,各从属系统的开发进度无法同步等。

3.10 特定领域软件体系结构

早在20世纪70年代就有人提出程序族、应用族的概念,并开拓了对特定领域软件体系结构的早期研究,这与软件体系结构研究的主要目的"在一组相关的应用中共享软件体系结构"也是一致的。为了解脱因为缺乏可用的软件构件以及现有软件构件难以集成而导致软件开发过程中难以进行重用的困境,1990年Mettala提出了特定领域软件体系结构(Domain Specific Software Architecture,DSSA),尝试解决这类问题。

3.10.1 DSSA的定义

简单地说,DSSA就是在一个特定应用领域中为一组应用提供组织结构参考的标准软件体系结构。对DSSA研究的角度、关心的问题不同导致了对DSSA的不同定义。

Hayes-Roth对DSSA的定义如下:"DSSA就是专用于一类特定类型的任务(领域)的、在整个领域中能有效地使用的、为成功构造应用系统限定了标准的组合结构的软件构件的集合"。

Tracz的定义为:"DSSA就是一个特定的问题领域中支持一组应用的领域模型、参考需求、参考体系结构等组成的开发基础,其目标就是支持在一个特定领域中多个应用的生成"。

通过对众多的DSSA的定义和描述的分析,可知DSSA的必备特征为:

(1) 一个严格定义的问题域和/或解决域。

(2) 具有普遍性,使其可以用于领域中某个特定应用的开发。

(3) 对整个领域的合适程度的抽象。

(4) 具备该领域固定的、典型的在开发过程中可重用元素。

一般的DSSA的定义并没有对领域的确定和划分给出明确说明。从功能覆盖的范围角度有两种理解DSSA中领域的含义的方式。

(1) 垂直域:定义了一个特定的系统族,包含整个系统族内的多个系统,结果是在该领域中可作为系统的可行解决方案的一个通用软件体系结构。

(2) 水平域:定义了在多个系统和多个系统族中功能区域的共有部分,在子系统级上

涵盖多个系统族的特定部分功能，无法为系统提供完整的通用体系结构。

在垂直域上定义的 DSSA 只能应用于一个成熟的、稳定的领域，但这个条件比较难以满足；若将领域分割成较小的范围，则更相对容易，也容易得到一个一致的解决方案。

3.10.2　DSSA 的基本活动

实施 DSSA 的过程中包含一些基本的活动。虽然具体的 DSSA 方法可能定义不同的概念、步骤、产品等，但这些基本活动是大体上一致的。以下将分三个阶段介绍这些活动。

1. 领域分析

这个阶段的主要目标是获得领域模型(domain model)。领域模型描述领域中系统之间的共同的需求。称领域模型所描述的需求为领域需求(domain requirement)。在这个阶段中首先要进行一些准备性的活动，包括定义领域的边界，从而明确分析的对象；识别信息源，即领域分析和整个领域工程过程中信息的来源，可能的信息源包括现存系统、技术文献、问题域和系统开发的专家、用户调查和市场分析、领域演化的历史记录等。在此基础上，就可以分析领域中系统的需求，确定哪些需求是被领域中的系统广泛共享的，从而建立领域模型。当领域中存在大量系统时，需要选择它们的一个子集作为样本系统。对样本系统需求的考察将显示领域需求的一个变化范围。一些需求对所有被考察的系统是共同的，一些需求是单个系统所独有的。很多需求位于这两个极端之间，即被部分系统共享。

领域分析的机制如图 3-34 所示。

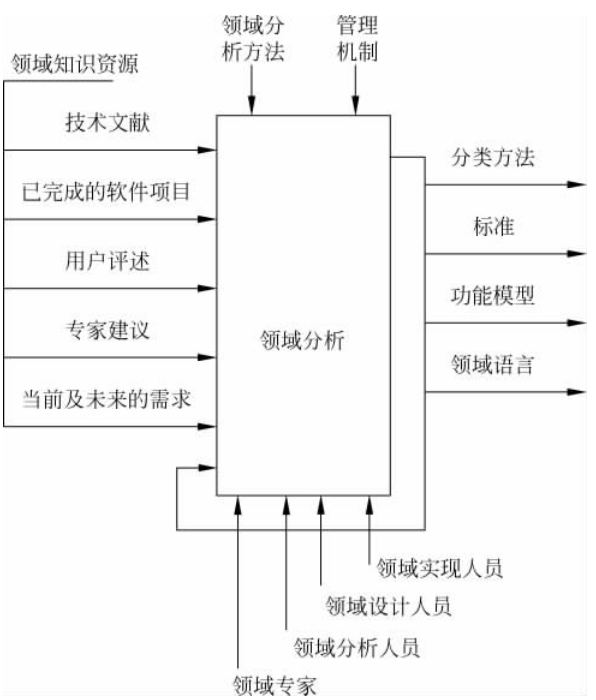

图 3-34　领域分析机制

2. 领域设计

这个阶段的目标是获得 DSSA。DSSA 描述在领域模型中表示的需求的解决方案，它

不是单个系统的表示,而是能够适应领域中多个系统的需求的一个高层次的设计。建立了领域模型之后,就可以派生出满足这些被建模的领域需求的DSSA。由于领域模型中的领域需求具有一定的变化性,DSSA也要相应地具有变化性。它可以通过表示多选一的(alternative)、可选的(optional)解决方案等来做到这一点。由于重用基础设施是依据领域模型和DSSA来组织的,因此在这个阶段通过获得DSSA,也就同时形成了重用基础设施的规约。

3. 领域实现

这个阶段的主要目标是依据领域模型和DSSA开发和组织可重用信息。这些可重用信息可能是从现有系统中提取得到,也可能需要通过新的开发得到。它们依据领域模型和DSSA进行组织,也就是领域模型和DSSA定义了这些可重用信息的重用时机,从而支持了系统化的软件重用。这个阶段也可以看作重用基础设施的实现阶段。

值得注意的是,以上过程是一个反复的、逐渐求精的过程。在实施领域工程的每个阶段中,都可能返回到以前的步骤,对以前的步骤得到的结果进行修改和完善,再回到当前步骤,在新的基础上进行本阶段的活动。

3.10.3 参与DSSA的人员

如图3-34所示,参与DSSA的人员可以划分为4种角色:领域专家、领域分析人员、领域设计人员和领域实现人员。下面将对这4种角色分别通过回答三个问题进行介绍:这种角色由什么人员来担任?这种角色在DSSA中承担什么任务?这种角色需要哪些技能?

1. 领域专家

领域专家可能包括该领域中系统的有经验的用户、从事该领域中系统的需求分析、设计、实现以及项目管理的有经验的软件工程师等。

领域专家的主要任务包括提供关于领域中系统的需求规约和实现的知识,帮助组织规范的、一致的领域字典,帮助选择样本系统作为领域工程的依据,复审领域模型、DSSA等领域工程产品,等等。

领域专家应该熟悉该领域中系统的软件设计和实现、硬件限制、未来的用户需求及技术走向等。

2. 领域分析人员

领域分析人员应由具有知识工程背景的有经验的系统分析员来担任。

领域分析人员的主要任务包括控制整个领域分析过程,进行知识获取,将获取的知识组织到领域模型中,根据现有系统、标准规范等验证领域模型的准确性和一致性,维护领域模型。

领域分析人员应熟悉软件重用和领域分析方法;熟悉进行知识获取和知识表示所需的技术、语言和工具;应具有一定的该领域的经验,以便于分析领域中的问题及与领域专家进行交互;应具有较高的进行抽象、关联和类比的能力;应具有较高的与他人交互和合作的能力。

3. 领域设计人员

领域设计人员应由有经验的软件设计人员来担任。

领域设计人员的主要任务包括控制整个软件设计过程,根据领域模型和现有的系统开发出DSSA,对DSSA的准确性和一致性进行验证,建立领域模型和DSSA之间的联系。

领域设计人员应熟悉软件重用和领域设计方法；熟悉软件设计方法；应有一定的该领域的经验，以便于分析领域中的问题及与领域专家进行交互。

4. 领域实现人员

领域实现人员应由有经验的程序设计人员来担任。

领域实现人员的主要任务包括根据领域模型和 DSSA，或者从头开发可重用构件，或者利用再工程的技术从现有系统中提取可重用构件，对可重用构件进行验证，建立 DSSA 与可重用构件间的联系。

领域实现人员应熟悉软件重用、领域实现及软件再工程技术；熟悉程序设计；具有一定的该领域的经验。

3.10.4　DSSA 的建立过程

因所在的领域不同，DSSA 的创建和使用过程也各有差异，Tracz 曾提出了一个通用的 DSSA 应用过程，这些过程也需要根据所应用到的领域来进行调整。一般情况下，需要用所应用领域的应用开发者习惯使用的工具和方法来建立 DSSA 模型。同时，Tracz 强调了 DSSA 参考体系结构文档工作的重要性，因为新应用的开发和对现有应用的维护都要以此为基础。

DSSA 的建立过程分为 5 个阶段，每个阶段可以进一步划分为一些步骤或子阶段。每个阶段包括一组需要回答的问题，一组需要的输入，一组将产生的输出和验证标准。本过程是并发的(concurrent)、递归的(recursive)、反复的(iterative)。或者可以说，它是螺旋型(spiral)。完成本过程可能需要对每个阶段经历几遍，每次增加更多的细节。

(1) 定义领域范围：本阶段的重点是确定什么在感兴趣的领域中以及本过程到何时结束。这个阶段的一个主要输出是领域中的应用需要满足一系列用户的需求。

(2) 定义领域特定的元素：本阶段的目标是编译领域字典和领域术语的同义词词典。在领域工程过程的前一个阶段产生的高层块图将被增加更多的细节，特别是识别领域中应用间的共同性和差异性。

(3) 定义领域特定的设计和实现需求约束：本阶段的目标是描述解空间中有差别的特性。不仅要识别出约束，并且要记录约束对设计和实现决定造成的后果，还要记录对处理这些问题时产生的所有问题的讨论。

(4) 定义领域模型和体系结构：本阶段的目标是产生一般的体系结构，并说明构成它们的模块或构件的语法和语义。

(5) 产生、搜集可重用的产品单元：本阶段的目标是为 DSSA 增加构件使得它可以被用来产生问题域中的新应用。

DSSA 的建立过程是并发的、递归的和反复进行的。该过程的目的是将用户的需要映射为基于实现约束集合的软件需求，这些需求定义了 DSSA。在此之前的领域工程和领域分析过程并没有对系统的功能性需求和实现约束进行区分，而是统称为“需求”。图 3-35 是 DSSA 的一个三层次系统模型。

DSSA 的建立需要设计人员对所在特定应用领域(包括问题域和解决域)必须精通，他们要找到合适的抽象方式来实现 DSSA 的通用性和可重用性。通常 DSSA 以一种逐渐演化的方式发展。

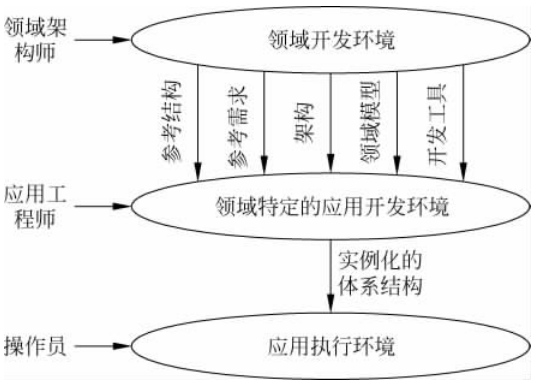

图 3-35 DSSA 的三层次的系统模型

3.10.5 DSSA 实例

本节将介绍一个保险行业特定领域软件体系结构。本节所指的保险行业应用系统,特指财产险的险种业务管理系统,它同样可以适用于人寿险的业务管理系统。图 3-36 是一个简化了的保险行业 DSSA 整体结构图。

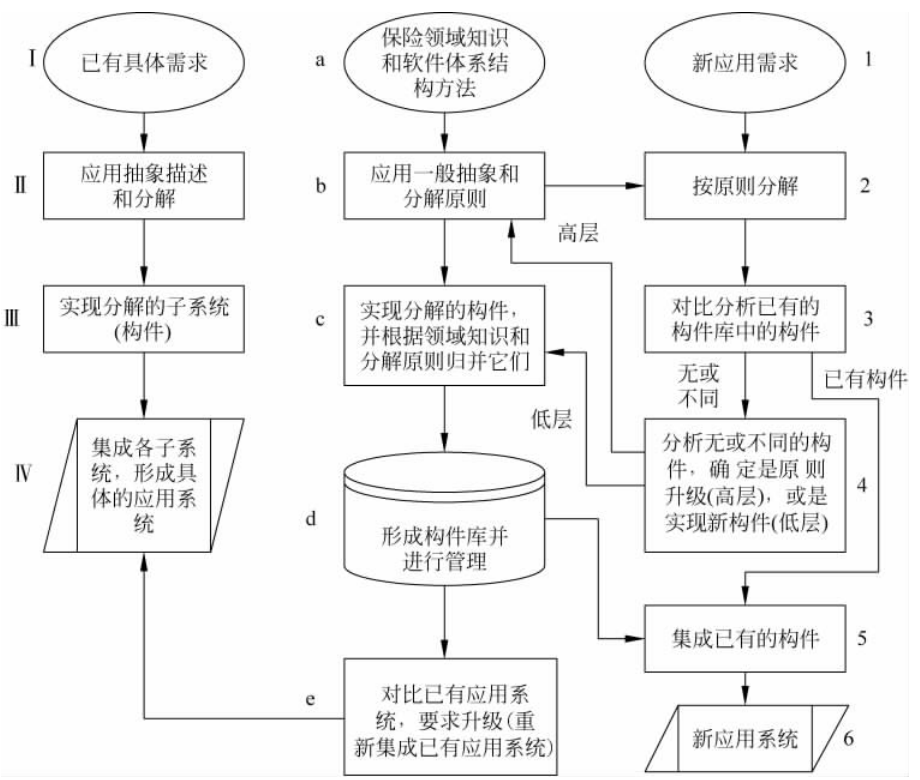

图 3-36 保险行业 DSSA 整体结构图

图 3-36 从左到右反映了研究和开发大型保险业务应用系统的历程,中间的环节引入了 DSSA 的概念。从实践的角度看,通用和共享的概念是自发的。但是当研究了国外有关

DSSA 的最新发展之后，建立了整体的方法论，并以此可以与国外同类技术发展进行沟通，取长补短，形成适合中国国情的 DSSA 体系，避免应用总是在图 3-36 中最左一列反复地重复开发，失去了与国外同行交流和吸取精华的机会。

图 3-36 中分为左、中、右三个纵向，分别采用不同的标号方式，在实例说明时，将对照标号进行详细讨论，三个纵向也分别反映应用开发从低层次体系结构向高层次体系结构转化的过程，而这种转化的依据恰好是保险领域的特殊知识。

1. 传统构造保险应用方法

对应于保险行业 DSSA 结构图 3-36 的左列，回顾传统的应用构造方法。标号Ⅰ反映传统应用的具体要求，所谓险种业务管理系统是保险产品在整个销售和服务过程中的计算机管理系统，险种相当于产品的品种。国内的保险从大类上可以分为机动车保险、企业财产保险、家庭财产保险、货物运输保险、船舶保险、建筑工程和安装工程保险、责任保险、信用保险、飞机/卫星保险等；从保险条款上区分又可分成为国内保险和涉外保险，总之具体的险种产品多达上百种。从前台销售和服务过程上看，可分成以下主要环节，展业、承保、交/退费、批改、理赔、汇总统计。这样一个涉及如此多专业内容的应用系统，开发量是巨大的，更由于险种条款的多变性，产品服务设计中更多地注意条款的专业性，而给信息系统建设带来困难，所以多年来系统开发技术水平发展缓慢，存在大量的低水平反复，从整体需求上看，只能保证在某一时刻上的稳定性。

传统上，人们在标号Ⅱ的工作主要是划分险种的分类，及分类后的过程管理，分类的依据也是险种产品的条款特征。把一种分类的管理，称为险种纵向管理，主要仍是包括展业、承保、交/退费、批改、理赔、汇兑统计等，它们对应一套独立的子系统，子系统的组成可以粗粒度分成以上 6 个部分，这样对一个财产险业务管理构成近 20 个子系统，且按管理过程都分成非常类似的 6 个部分，如果全面实现应用将面对上百个粗粒度独立部分的设计和开发。

在标号Ⅲ中，实际上是组织开发这上百个粗粒度的部分，从条款上看，只是相似但绝不相同，由于开发所用的平台是关系数据库，所以库表都十分类似，但是业务含义不同，必须分别开发。从工程组织上看当然可以使用一些低层次的共享手段，突出表现在函数/模块化的管理、编程约定和机构移植等方面。

传统的标号Ⅳ的内容主要是集成各子系统，由于各部分组合较紧密，所以对子系统内部集成工作等同于更大范围的开发，但结果是构成了近二十个子系统，它们分别管理着险种纵向的业务操作，这就是最初级的保险应用系统。

从以上的过程不难看出其中的不足，其关键是大量基本单元相似和共享问题，其隐含的需求还包括基本单元的需求稳定性，这种需求实际上是一种更高层面的软件体系结构上的需求。

2. 采用 DSSA 后的变化

为了解决保险行业 DSSA 整体结构图左列方法引起的不足，引入了 DSSA 方法，它形成了图 3-36 中的中间一列。标号 a 实际上反映了对领域工程一般原理的学习，以及保险领域知识的深入理解，它反映领域专家的技术背景，结合标号 b 的内容，对保险行业内共同的特征进行提取，所用方法是一种高层抽象的方法，把相似抽象成相同，就像把算术问题抽象成代数问题一样，形成概念定义上的提升，这是实现保险行业 DSSA 的关键步骤之一，由此形成了相对抽象的构件分解原则和方法。其前提条件是充分理解行业的发展，对细节概念

和定义加以屏蔽和提升,同时充分考虑数据库实现方法上的技巧性,保证实现上的可操作性和新技术方法的应用。

在标号c中,主要的目标是实现标号b中抽象和划分的构件,实际上是针对标号Ⅲ中的展业、承保、交/退费、批改、理赔、汇兑统计6大部分,把领域知识加入,重新实现它们,很显然这是从思想到实现的关键一步,结合在标号Ⅳ中使用的模块化设计,归纳为如下实现方法:

首先,构件要实体分类,这是因为实现机制是关系数据库。在抽象时,要保证数据结构分类的一致性,但是在用户操作方面又要考虑外层的语义,所以提供相对专用的界面。因此,主要的应用操作逻辑,应适应数据结构的通用性和操作界面的专用性,并按对象模式进行分类实现。例如,在承保构件中,可能由于主、从表的不同而有两种方式,一种是主表方式,它没有更细的表达;而另一种是既有主表,又有从表,且更新数据时先从后主,查看数据时,先主后从。无论主、从表设计,在数据结构中,充分利用"权利、义务"对等原则设计属性项,并提供若干概念相对通用的编码属性,以解决分类标识问题和需求扩充问题,以此保证结构的稳定性。同样,对交/退费、批改构件、理赔构件、汇兑统计构件等使用类似的方法分别实现。

其次,构成实体构件的操作部分,还要进一步加强模块化的思想,提供通用的工具,大至菜单、报表、查询打印工具,小至通用的计算函数、约定编程技巧等。这部分的内容,从构件实现上保证了大粒度,因为即使使用第四代语言开发应用系统,在模式一级也需要提供更大粒度的实现体,它对构件内部的更新和升级提供了基础条件。

总之,在整个实现过程中,充分使用应用构件内、外部重用的技术,充分利用领域知识抽象通用特性和提高应用操作的有效性,最终完成各构件的编程实现。

标号d只反映构件管理的辅助内容,因为当系统开发越来越庞大之后,管理的工作越来越多,它也包括传统软件开发版本管理的全部内容。

标号e实际上集成应用子系统,因为构件只有在装配和配置之后,才能成为最后的应用系统。当作为产品实施时,它充分反映客户化的内容,目前国外大型的应用系统有时要客户化一年或更多的时间。

3. 采用DSSA后的应用构成和系统升级

当建立了一套构件库之后,新的业务应用系统生成就采用新的方式,参考图3-36的右列,作如下讨论。

标号1反映新的应用需求,它可能是新险种应用,也可能是对老险种的管理新要求,总之要适应保险行业竞争发展的管理需要。标号2表示使用标号b的抽象和分解原则明确化需求,使其具有统一的划分构件的方法。标号3的内容是与构件库的构件进行对比分析,简单的情况表示已有相应的构件,这样可以通过标号5集成和客户化新的应用系统;复杂的情况,没有对应的构件或已有构件只是其中的一部分或构件的粒度太大,都将进入标号4。标号4是决定构件库做什么层次升级的决定机构,高层反映要对过去抽象和分解原则进行变化,低层可能直接完善构件库,以下可以对返b(高层)和返c(低层)进行实例说明。

返b的实例,例如,在承保的构件中,近期要加入独立的客户管理内容,保险公司内部的人员管理、系统的操作权限管理、核保管理等,这实际上要对原来的构件划分原则作新的调整,并由此产生新的构件,同时引起标号c中的新的实现,同样的内容也会发生在批改、理赔

的构件中，由此引起了构件库的全面升级。

返 c 的实例，例如，在承保的构件中，在从表中加入新的一对多子表，从而使原来的主、从结构模式变成了新的三级结构，并从实现机制上增加了更详细信息的表达方式。从发展的眼光看，应用不是一次开发到位的，系统实现之初，构件的粒度可能相对较大，但随着应用的深入和对业务的本质把握，将逐步细化实现粒度，以适应整个系统的灵活性和效率。

总之，标号 b 和标号 c 的变动都将引起构件库标号 d 的变动，标号 e 反映这种变化后，应用系统是否升级。在实际应用中，标号Ⅳ的系统也是定期升级的，当所有的标号Ⅳ的应用系统都不使用标号 d 中的某些构件时，就可以做构件的清理，使构件库更加高效。

由标号 5 集成的新应用系统，构成了标号 6。从应用角度看，它们也是标号Ⅳ的同类，所不同的是当构件库发生变化时，老应用的整体升级是有选择的，即使它们不升级，作为遗留系统，它们仍然运行良好，而且这种升级受整体客户化费用的考虑，不一定马上进行。从前面给出的实例看，很多升级意味着整体系统功能的增加，对新的应用实施单位往往要进行一定选择。

当采用了 DSSA 结构之后，获得了如下的好处：

(1) 相对于过去的开发方法，系统开发、维护的工作量大幅度减少，整个应用系统的构件重用程序相当大。

(2) 便于系统开发的组织管理，在大型系统开发过程中，最突出的问题是人员的组织问题。采用了 DSSA 之后，开发中涉及核心技术的人员从 15 人左右下降到 5 人左右，其他的人力进入外围产品化的工作，如产品包装、市场销售、工具的开发、客户化服务等。而过去这方面内容在技术部门是被忽视的，而在应用软件工程中，它们也占重要的位置。

(3) 系统有较好的环境适应性，构件的升级引发应用系统的升级，并在构件库中合理地控制粒度，使系统的总体结构设计与算法和模块化设计同等重要，并灵活地保证新、老应用系统的共存。

3.10.6　DSSA 与体系结构风格的比较

在软件体系结构的发展过程中，因为研究者的出发点不同，出现了两个互相直交的方法和学科分支：以问题域为出发点的 DSSA 和以解决域为出发点的软件体系结构风格。因为两者侧重点不同，它们在软件开发中具有不同的应用特点。

DSSA 只对某一个领域进行设计专家知识的提取、存储和组织，但可以同时使用多种体系结构风格；而在某个体系结构风格中进行体系结构设计专家知识的组织时，可以将提取的公共结构和设计方法扩展到多个应用领域。

DSSA 的特定领域参考体系结构通常选用一个或多个适合所研究领域的体系结构风格，并设计一个该领域专用的体系结构分析设计工具。但该方法提取的专家知识只能用于一个较小的范围——所在领域中。不同参考体系结构之间的基础和概念有较少的共同点，所以为一个领域开发 DSSA 及其工具在另一个领域中是不适应的或不可重用的，而工具的开发成本是相当高的。

体系结构风格的定义和该风格应用的领域是直交的，提取的设计知识比用 DSSA 提取的设计专家知识的应用范围要广。一般的、可调整的系统基础可以避免涉及特定的领域背景，所以建立一个特定风格的体系结构设计环境的成本比建立一个 DSSA 参考体系结构和

工具库的成本要低得很多。因为对特定领域内的专家知识和经验的忽略,使其在一个具体的应用开发中所起的作用并不比DSSA要大。

DSSA和体系结构风格是互为补充的两种技术。在大型软件开发项目中基于领域的设计专家知识和以风格为中心的体系结构设计专家知识都扮演着重要的角色。

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. 选择两种风格,设计简单的体系结构,并实现简单的原型系统。

3. 查阅相关资料,尝试从面向服务的体系结构(SOA)的实现技术的角度出发,深入分析CORBA、DCOM以及基于UDDI的Web Service等分布式计算技术,在此基础上,对这些技术做比较和归纳,涉及CORBA、DCOM和基于UDDI的Web Service之间的互操作的方案。

4. 本章独立介绍了每种体系结构,但事实上,所有的体系结构不仅有紧密的联系,而且在大多数情况下是被一起使用的。对于一个实际的系统甚至不能判断它是哪种风格,因此就存在异构风格的共存。图3-37展示了一个虚拟系统,它整合了哪种体系结构风格。试说明该系统中包含哪些体系结构风格?

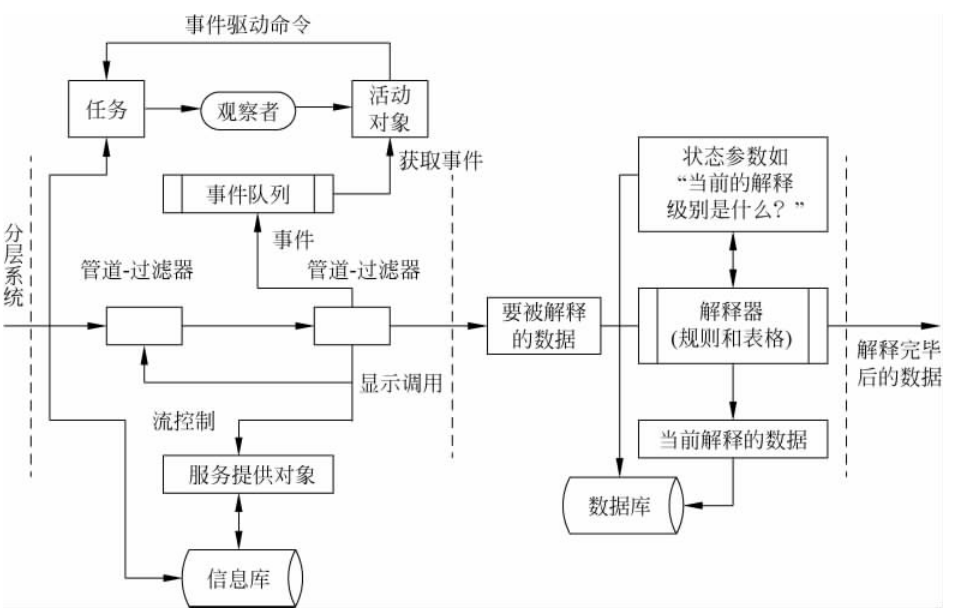

图3-37 一个虚拟系统

5. 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、事件驱动风格、分层系统、C2风格和基于消息总线的风格进行分析比较。

6. 试说明特定领域软件体系结构包括哪些基本活动，应该如何选择参与人员及其创建过程。

7. 希赛公司欲开发一个车辆定速巡航控制系统，以确保车辆在不断变化的地形中以固定的速度行驶。图 3-38 给出了该系统的简化示意图，表 3-2 描述了各种系统输入的含义。

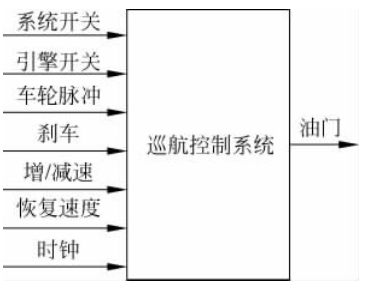

图 3-38　定速巡航控制系统简化示意图

表 3-2　定速巡航控制系统输入说明

输入名称	作　　用
系统开关	开启/关闭巡航控制系统
引擎开关	开启/关闭汽车引擎(引擎开启时，巡航控制系统处于就绪状态)
车轮脉冲	车轮每转一次，相应地发出一次脉冲
刹车	当刹车被踩下时，定速巡航控制系统会临时恢复到人工控制
增/减速	增加或减慢当前车速(仅在定速巡航控制系统处于开启的状态下可用)
恢复速度	恢复原来保持的车速(仅在定速巡航控制系统处于开启的状态下可用)
时钟	每毫秒定时脉冲

公司的领域专家对需求进行深入分析后，将系统需求认定为：任何时刻，只要定速巡航控制系统处于工作状态，就要有确定的期望速度，并通过调整引擎油门的设定值来维持期望速度。

在对车辆定速巡航控制系统的体系结构进行设计时，公司的设计师王工提出采用面向对象的体系结构风格，而李工则主张采用控制环路的体系结构风格。在体系结构评估会议上，专家对这两种方案进行综合评价，最终采用了面向对象和控制环路相结合的混合体系结构风格。

【问题 1】

在实际的软件项目开发中，采用成熟的体系结构风格是项目成功的保证。请用 200 字以内的文字说明：什么是软件体系结构风格；面向对象和控制环路两种体系结构风格各自的特点。

【问颐 2】

用户需求没有明确给出该系统如何根据输入集合计算输出。请用 300 字以内的文字针对该系统的增减速功能，分别给出两种体系结构风格中的主要构件，并详细描述计算过程。

【问题3】

实际的软件系统体系结构通常是多种体系结构风格的混合,不同的体系结构风格都有其适合的应用场景。以该系统为例,针对面向对象体系结构风格和控制环路体系结构风格,各给出两个适合的应用场景,并简要说明理由。

主要参考文献

[1] 张友生.软件体系结构的风格. 程序员,2002,(8):45-38.

[2] 张友生.几种新型软件体系结构. 程序员,2002,(9):49-51.

[3] 张友生,陈松乔.正交软件体系结构的设计与演化.小型微型计算机系统,2004,(2): 30-35.

[4] 张友生,陈松乔.层次式软件体系结构的设计与实现.计算机工程与应用,2002,(22):154-156.

[5] 张友生,陈松乔.C/S与B/S混合软件体系结构模型.计算机工程与应用,2002,(23):138-140.

[6] 张友生,钱盛友.异构软件体系结构的设计.计算机工程与应用,2003,(22):126-128.

[7] 张友生. 遗留系统的评价方法和演化策略.计算机工程与应用,2003,(13):29-31.

[8] 叶俊民,赵恒,曹瀚等.软件体系结构风格的实例研究.小型微型计算机系统,2003,(10):1158-1160.

[9] 王琰,徐重阳,蔷薇等.基于C/S结构的网络计算模型.计算机应用研究,2000,(9):50-53.

[10] 浦江.网络计算模式的演变与发展.电子技术,2001,(1):15-19.

[11] 周之英.现代软件工程[中].北京: 科学出版社,2000.

[12] 齐治昌,谭庆平,宁洪.软件工程.北京: 高等教育出版社,1997.

[13] 张世琨,王立福,杨芙清.基于层次消息总线的软件体系结构风格.中国科学(E辑),2002,(6): 393-400.

[14] 王广昌.软件产品线关键方法与技术研究.浙江大学博士学位论文,2001,10.

[15] 李克勤,陈兆良,梅宏等.领域工程概述.计算机科学,1999,(5):21-25.

[16] 邢立,左春,孙玉芳.关于保险行业特定领域软件体系结构的研究.计算机工程,2000,(4): 47-49.

[17] 谭凯,林子禹,彭德纯等.多级正交软件体系结构及其应用.小型微型计算机系统,2000,(2): 138-141.

[18] 陈豪,孙正义,张德富.三层客户/服务器体系结构德一个应用实例.计算机工程与应用,2000,(3): 173-176.

[19] Hayes-Roth. Architecture-based acquisition and development of software: guidelines and recommendations from the ARPA domain-specific software architecture (DSSA) program. Teknowledge Federal Systems. Version 1.01 February 4,1994.

[20] Will Tracz and Lou Coglianese. Domain-specific software architecture engineering process guidelines. ADAGE-IBM-92-02B Version 2.1,1992.

[21] L Bass, P Clcmcnts and R Kazman. Software Architecture in Practice. Addison Wcslcy Longman, 1998.

[22] M Show and D Garlan. Software Architecture: Perspectives on An Emerging Discipline. Englewood Cliffs, New York: Prentice Hall, 1996.

[23] Maria Wricsson. Developing Large-scale Systems with the Rational Unified Process. Rational Software White Paper, 2000.

[24] Herbert A Simon. The Sciences of the Artificial, MIT Press, 1981.

[25] I Jacobson, K Palmkvist, S Dyrhage. Systems of interconnected systems, ROAD, 1995(1): 81-93.

[26] I Jacobson, M Griss, P Jonsson. Software Reuse-Architecture, Process and Organization for Business Success, Addison Wesley Longman, 1997.

[27] J Rumbaugh, G Booch, I Jacobson. UML Reference Manual, Addison Wesley Longman, 1999.
[28] K H Bennett. Legacy systems: coping with success. IEEE Software, 1995, (1): 19-23.
[29] M L Brodie, M Stonebraker. Migrating Legacy Systems. Morgan Kaufmann Publishers, 1995.
[30] H M Sneed. Planning the reengineering of legacy systems. IEEE Software, 1995, (1): 24-34.
[31] H M Sneed. Encapsulating legacy software for use in client/server systems, Proc. IEEE Conference on Software Maintenance, 1996: 104-120.
[32] Nelson H Weiderman, John K Bergey, Dennis B. Smith and etc.. Approaches to legacy system evolution. Technical Report CMU/SEI-97-TR-014, 1997, 11.