第3章 软件体系结构风格

3.1 软件体系结构风格概述

软件体系结构是随着软件工程的发展逐步发展起来的。20 世纪 90 年代，人们开始对软件体系结构进行研究，当时，软件系统程度的提高和规模的扩大使得软件开发人员开始感到很困难，而且，开发出软件的质量也难以保证。这好比早期盖房，谈不上房屋的结构设计，现在要盖高楼大厦了，才兴起体系结构的设计研究。此时，MarShaw 和 DavidGarlan 在论文中提出了软件体系结构的概念，定义为“能够用来具体描述软件系统控制结构和整体组织的一种体系结构，能够表示系统的框架结构，用于从较高的层次上来描述各部分之间的关系和接口”。这给人们进行软件开发带来了曙光，至今，它已成为现代软件开发过程中一个至关重要的部分。

通常认为，D. E. Perry 等的论文是软件体系结构研究真正开始的标志。由于软件体系结构作为软件工程的一个独立的研究领域出现的时间不长，虽然对其重要性和意义已取得了比较广泛的共识，但对于软件体系结构概念并没有统一的定义。但研究者们对软件体系结构也达成了一些共识：①软件体系结构是对系统的一种高层次的抽象描述，主要反映拓扑属性，有意忽略细节；②软件体系结构由构件和构件之间的联系组成，构件又有它自身的体系结构；③构件的描述有计算功能、结构特性及其他特性 3 个方面。计算功能是指构件实现的整体功能。结构特性描述与其他构件的组织和联系方法是软件体系结构中最重要的内容。其他特性描述了构件的执行效率、环境要求和整体特性等方面的要求，这些大多是定量的描述，例如时间、空间、精确度、安全性、保密性、带宽、吞吐量和最低软/硬件要求等。

从软件体系结构的定义可以看出，软件体系结构主要涉及构件、构件之间的联系与约束、由构件通过相互交互形成的系统架构 3 个方面的内容，可以用图 3-1 简单表示软件体系结构。

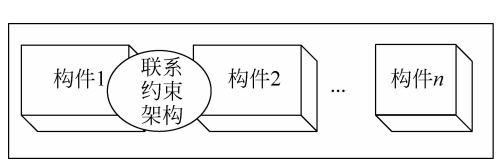

图 3-1　软件体系结构架构图

通过使用软件体系结构，可以有效地分析用户需求、方便系统的修改以及减小程序构造风险。随着软件规模的不断扩大和复杂程度的日益提高，系统框架结构的设计变得越来越关键。软件框架设计的核心问题是能否重用已经成型的体系结构方案。由此，产生了软件体系结构风格的概念。

当人们谈到体系结构时，经常会使用“风格”一词。对于建筑行业而言，有罗马式风格、哥特式风格和维多利亚式风格等。虽然这些建筑风格各有差异，但其框架结构都是相似的。同理，软件开发也一样，不同系统的设计方案存在着许多共性，把这些共性部分抽取出来，就形成了具有代表性的、可被人广泛接受的体系结构风格。

通常，软件体系结构风格也称为软件体系结构惯用模式，它是不同系统所拥有的共同组织结构和语义特征，是构件和连接件之间相互作用的形式化说明，用于指导将多个模块组织成一个完整的应用程序。软件体系结构风格定义了用于系统描述的术语表和一组用于指导系统构建的规则。软件体系结构风格包括构件、连接件和一组将它们结合在一起的约束限制，例如拓扑限制和语义限制等。

对于高质量的软件产品而言，首先要为其选择合适的体系结构风格，这样才能更好地重用已有的设计方案和实现方案。利用软件体系结构风格中的不变部分，可以使系统大粒度地重用已有的实现代码。由于采用了常用的手段和规范的方法来组织应用系统，因此，可以使其他设计者很容易地理解软件的框架结构。

3.2　常用的软件体系结构风格

软件体系结构的风格是人们在开发软件的过程中不断积累起来的，是多年探索研究和工程实践的结果。它由组织规则及结构构成，是描述领域中系统组织方式的惯用模式，是对某一特定领域中系统所共有的结构和语义特性的反映。

大粒度软件重用的可能性，正是基于了软件体系结构的风格。Garlan 和 Shaw 将体系结构风格进行了以下分类。

(1) 数据流风格：批处理序列、管道/过滤器；

(2) 仓库风格：数据库系统、超文本系统、黑板系统；

(3) 独立构件风格：进程通信、事件系统；

(4) 调用/返回风格：主程序/子程序、面向对象、层次结构；

(5) 虚拟机构风格：解释器、基于规则的系统。

软件体系结构风格的 5 种分类不能完全代表体系结构风格的组成，随着软件研发技术的不断进步，近些年接连总结出的几种新型的软件体系结构风格将在本章后半部分介绍。

3.2.1　管道/过滤器体系结构风格

管道/过滤器结构是典型的数据流软件体系结构风格，管道/过滤器结构主要包括过滤器和管道两种元素。在这种体系结构中，每个模块都有一组输入和一组输出。每个模块从它的输入端接收输入数据流，在其内部经过处理后，按照标准的顺序将结果数据流送到输出端，以达到传递一组完整的计算结果的目的。管道/过滤器模型的基本部件都有一套输入/

输出接口。每个部件从输入接口中读取数据，经过处理后将结果数据置于输出接口中，这样的部件称为过滤器。这种模型的连接者将一个过滤器的输出传送到另一个过滤器的输入，这种连接者称为管道。过滤器的基本结构如图 3-2 所示。

图 3-2　管道/过滤器中的基本单元——过滤器

管道/过滤器结构将数据流处理分为几个步骤进行，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来实现。数据类型的约束使得通常在输入和输出端有一个本地的数据类型转换器，数据在过滤器中经过计算处理然后通过管道传输给另一个过滤器，依次生成增量式的处理结果。在管道/过滤器结构中，过滤器必须是相互独立的实体，它们相互之间的状态不可共享。由于每一个过滤器并不能识别它的数据流上游和下游的过滤器的身份，需要在过滤器的输入和输出端的管道保证输入数据和输出数据类型衔接的正确性。此外，该结构还要求整个管道/过滤器网的最后处理结果的正确性与过滤器组进行的增量处理次序不能相关。管道/过滤器风格的体系结构图如图 3-3 所示。

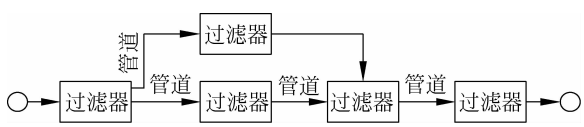

图 3-3　管道/过滤器风格的体系结构

管道/过滤器体系结构风格具有以下优点：

(1) 设计人员将整个系统的输入/输出行为理解为单个过滤器行为的叠加与组合，这样可以将问题分解，化繁为简。

(2) 对于任何两个过滤器，只要它们之间传送的数据遵守共同的规约就可以相互连接。每个过滤器都有自己独立的输入/输出接口，如果过滤器间传输的数据遵守其规约，只要用管道将它们连接就可以正常工作。

(3) 整个系统易于维护和升级，旧的过滤器可以被替代，新的过滤器可以添加到已有的系统上。软件的易于维护和升级是衡量软件系统质量的重要指标之一，在管道/过滤器结构中，只要遵守输入/输出数据规约，任何一个过滤器都可以被另一个新的过滤器代替。同时，为了增强程序功能，可以添加新的过滤器，这样系统的可维护性和可升级性就得到了保证。

(4) 每个过滤器作为一个单独的执行任务，可以与其他过滤器并发执行。即过滤器的执行是独立的，不依赖于其他过滤器。

但是，管道/过滤器结构也存在着若干不利因素，具体表现在以下方面：

(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，所以设计者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的下降，并增加了编写过滤器的复杂性。

(4) 管道/过滤器结构的固有特性,决定了很难制定错误处理的一般性策略。

管道/过滤器体系结构风格典型的应用例子是类 UNIX 系统下的基于管道的进程间通信机制。它包括无名管道和有名管道两种,前者用于父进程和子进程间的通信,后者用于运行于同一台计算机上的任意两个进程间的通信。管道是通过文件读/写接口存取的字节流,对于管道两端的进程而言,管道就是一个文件,但与普通文件不同的是,管道是一个固定大小的内存缓冲区。一个进程向管道中写入的内容被管道另一端的进程读出,写入的内容每次都添加在管道缓冲区的末尾,每次都从缓冲区的头部读出数据,数据一旦被读,它就被从管道中抛弃,释放空间,以便写入更多的数据。

传统的编译器是管道/过滤器体系结构风格的另一个典型例子。编译器由词法分析、语法分析、语义分析、中间代码生成、中间代码优化和目标代码生成等几个模块组成,一个模块的输出是另一个模块的输入。源程序经过各个模块的独立处理之后,最终产生目标程序。编译器的框架结构如图 3-4 所示。

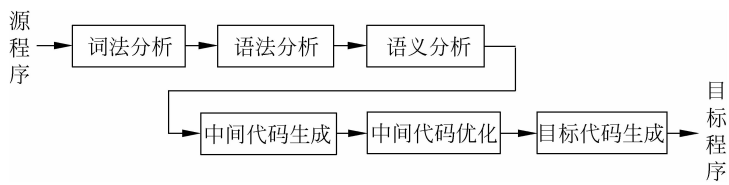

图 3-4　编译器的框架结构

此外,如果对管道/过滤器体系结构做一些限制和约束,就可以得到不同类型的体系结构。将所有过滤器都限制为单输入和单输出,系统拓扑结构就只能是线性序列,这就是所谓的管线。过滤器之间是通过有名称的管道来传送数据的,例如文件就是有名称的管道/过滤器。限定过滤器的数据存储容量,就可以得到有界管道/过滤器。过滤器将所有输入数据作为单个实体进行处理,这就是批处理系统。

3.2.2　面向对象体系结构风格

抽象数据类型概念对软件系统有着重要的作用,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,数据表示和相关的基本操作封装在抽象数据类型或对象中。这种模式的构件是对象,或者称为抽象数据类型的实例。这种模式有两个重要的方面,一是对象维护自身表示的完整性;二是这种表示对其他对象是隐藏的。

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。这种风格的组件是对象,或者说是抽象数据类型的实例。对象是一种被称为“管理者”的组件,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,面向对象体系结构如图 3-5 所示。

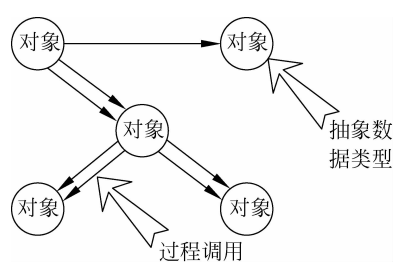

图 3-5　数据抽象和面向对象体系结构的示意图

对象抽象可以使构件与构件之间以黑盒方式来进行操作。这种结构支持信息隐藏,封装

技术可以使对象结构和实现方法对外透明。利用封装技术，可以将属性和方法包装在一起，由对象对它们进行统一管理。

例如，面向对象的业务处理环节装配体系结构（Object Oriented Processing Steps Assembling Architecture，OPSAA）是建立在对象池、对象池服务和业务处理环节概念基础上的一种适用于企业管理信息系统开发的软件体系结构。

对象池实现系统内所有的业务对象，通过业务对象封装企业的业务信息、业务活动和业务规则，并根据业务流程的需要组建一组功能服务，系统内的其他部分只能通过使用这些功能服务来进行业务运作，业务对象之间的交互逻辑、对象之间的制约关系也在这些功能服务中体现。

对象服务池，也就是对象池功能服务，除了为业务功能调用提供一致的接口，为业务信息的正确性、完整性和一致性提供保证外，还能够简化具体应用程序的开发复杂性、降低对象池和系统其他部分的耦合程度。总的来说，对象池服务的具体功能是根据业务流程的需要由对象池内相关业务对象的行为组装而成。

业务流程是由一组业务处理环节协作完成的，通过顺序、选择、循环调用相关业务处理环节的功能，实现业务流程的任务。

企业的业务流程是围绕着对企业业务对象的处理展开的，业务流程对业务对象的处理过程可以划分为一系列功能相对独立的处理环节。在 OPSAA 中，对象池实现所有业务对象，并向外界提供功能服务以完成业务运作；业务处理环节调用对象池服务完成自己的功能；具体应用程序由业务处理环节组装而成，面向对象的业务处理环节装配体系结构以对象池和业务处理环节为主要的构件，通过对象池服务和功能调用实现构件之间的连接。图 3-6 是 OPSAA 的结构示意图。

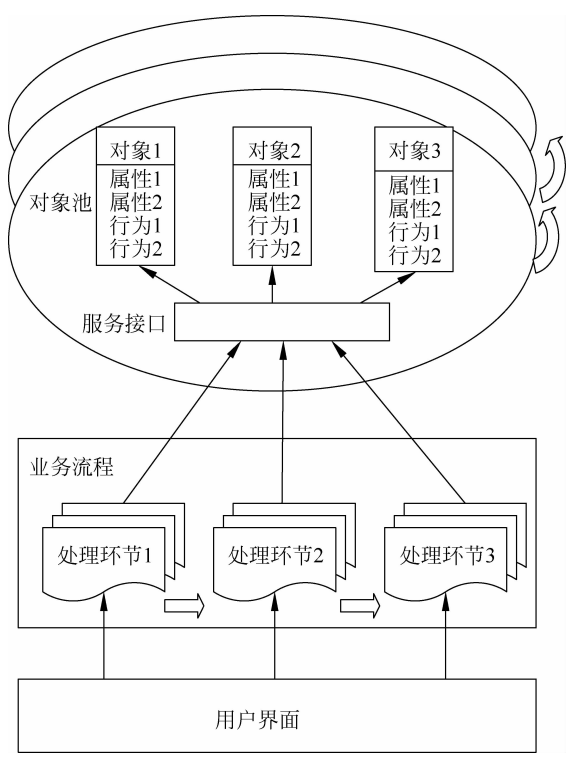

图 3-6　面向对象的业务处理环节装配体系结构

面向对象体系结构风格有许多优点，例如：

（1）一个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，而不影响其他对象；

（2）对象将数据和操作封装在一起，提高了系统内聚性，减小了模块之间的耦合程度，使系统更容易分解为既相互作用又相互独立的对象集合；

（3）继承和封装方法为对象重用提供了技术支持。

但是，面向对象体系结构风格也存在着一些问题，具体表现在以下方面：

（1）如果一个对象要调用另一个对象，必须知道它的标识和名称，因此，只要一个对象的标识发生改变，就必须修改所有显式调用这个对象的程序语句。而在管道/过滤器体系结构风格中，过滤器不需要知道与之交互的构件。

（2）如果一个对象的标识发生改变，那么必须修改所有显式调用它的其他对象，并消除由此引发的副作用。例如，对象 A 调用了对象 B，对象 C 也调用了对象 B，那么 A 对 B 的调用可能会影响到 C。

3.2.3　分层体系结构风格

分层体系结构风格是调用/返回风格的一个代表。分层体系结构风格组织成一个层次结构，通过分解，能够将复杂系统划分为多个独立的层次，每一层都具有高度的内聚性，并要求每一层为上层服务，并作为下层的客户，较高层面向特定应用问题，较低层则更具有一般性。在分层体系结构中，层间的连接器通过层间交互的协议来定义，且上、下层之间是单向调用关系，即上层通过下层提供的接口来使用下层的功能，而下层却不能使用上层的功能。

在一些分层体系结构中，除了精心挑选的输出函数外，内部层只对相邻层可见，这样的结构中，构件在一些层上实现了虚拟机（在另一些分层体系结构中层是部分不透明的）。拓扑约束包括对相邻层间交互的约束，这样的结构能够允许将一个复杂问题分解成一个增量步骤的序列来实现，从而简化程序的设计和实现。另外，由于每一层最多只影响两层，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，同样能为软件重用提供强大的支持。

分层体系结构如图 3-7 所示。在分层体系结构中，利用接口可以将下层实现细节隐藏起来，从而有助于抽象设计，形成松散耦合的结构模型，并有助于对逻辑功能实施灵活的增加、删除和修改，以及不同平台之间的快速移植。

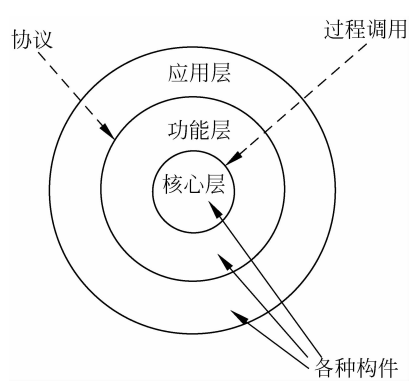

图 3-7　分层体系结构

分层体系结构风格具有以下优点。

（1）支持基于抽象程度递增的系统设计：设计者可以将系统分解为一个增量的步骤序列，从而完成复杂的业务逻辑。

（2）支持功能增强：每一层最多和相邻的上、下两层进行交互，每一层的功能变化最多只影响相邻两层，便于实现系统功能的扩展。

（3）支持重用：只要给相邻层提供相同的接口，就可以使用不同的方法来实现每一层，支持软件资源的重用。

但是，分层体系结构风格也存在着一些问题，具体表现在以下方面：

（1）并非所有系统都能够容易地按照层次来进行划分，即使一个系统的逻辑结构是层次化的，但出于对系统性能的考虑，设计者不得不把不同抽象程度的功能合并到一层，破坏了逻辑独立性。

（2）很难找到一种合适、正确的层次抽象方法，其应用范围受到限制。

（3）在传输数据时，需要经过多个层次，导致了系统性能下降。

（4）多层结构难以调试，往往需要通过一系列的跨层次调用来实现。

在实际开发过程中，分层体系结构具有很高的应用价值，提高了系统的可变性、可维护性、可靠性和可重用性。分层体系结构应用的实例很多，例如，开放系统互联国际标准组织（Open Systems Interconnection-International Standards Organization，OSI-ISO）指定的分层通信协议、计算机网络协议 TCP/IP、操作系统和数据库系统，都采用了这种框架结构。

ZigBee 协议栈标准采用的是 OSI 的分层结构，ZigBee 协议栈由高层应用规范、应用汇聚层、网络层、数据链路层和物理层组成。IEEE 802.15.4—2003 标准定义了物理层和链路层标准，网络层以上的协议由 ZigBee 联盟制定。应用汇聚层的框架包括了应用支持子层、ZigBee 设备对象及由制造商指定的对象，该层把不同的应用映射到 ZigBee 网络上，主要包括多个业务数据流的汇聚、安全属性设置等功能。网络层不仅包含了通用的网络层功能，还和底层的 IEEE 802.15.4 标准一样可以省电，网络层将采用基于 Adhoc 技术的路由协议来实现网络的自组织和自维护，以最大程度地降低网络的维护成本。其协议栈体系结构如图 3-8所示。

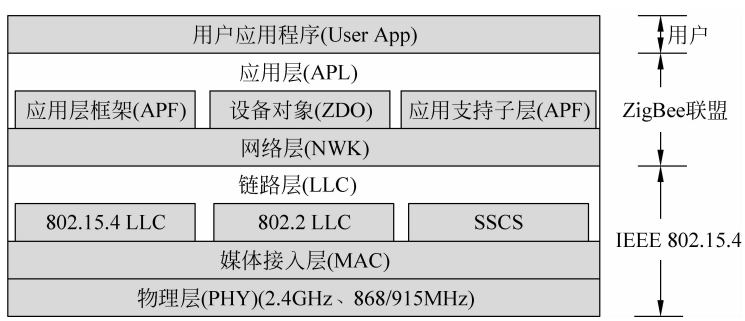

图 3-8　ZigBee 协议栈体系结构

分层体系结构风格的另外一个例子是 Android 系统，如图 3-9 所示。

从结构图来看，Android 系统分为 4 个层次，从高层到低层分别是应用程序层、应用程序框架层、系统运行库层和 Linux 核心层。在应用程序层中，Android 会和一系列核心应用程序包一起发布，该应用程序包包括 Email 客户端、SMS 短消息程序、日历、地图、浏览器、联系人管理程序等。在应用程序框架层中，开发人员也可以完全访问核心应用程序所使用的 API 框架。该应用程序的结构设计简化了组件的重用，任何一个应用程序都可以发布它的功能块，并且任何其他的应用程序都可以使用其发布的功能块（不过要遵循框架的安全性限制）。在系统运行库层中包括两个部分，一是程序库，其包含一些 C/C++ 库，这些库能被 Android 系统中的不同组件使用，它们通过 Android 应用程序框架为开发者提供服务。二是 Android 运行库，它包括一个核心库，该核心库提供了 Java 编程语言核心库的大多数功

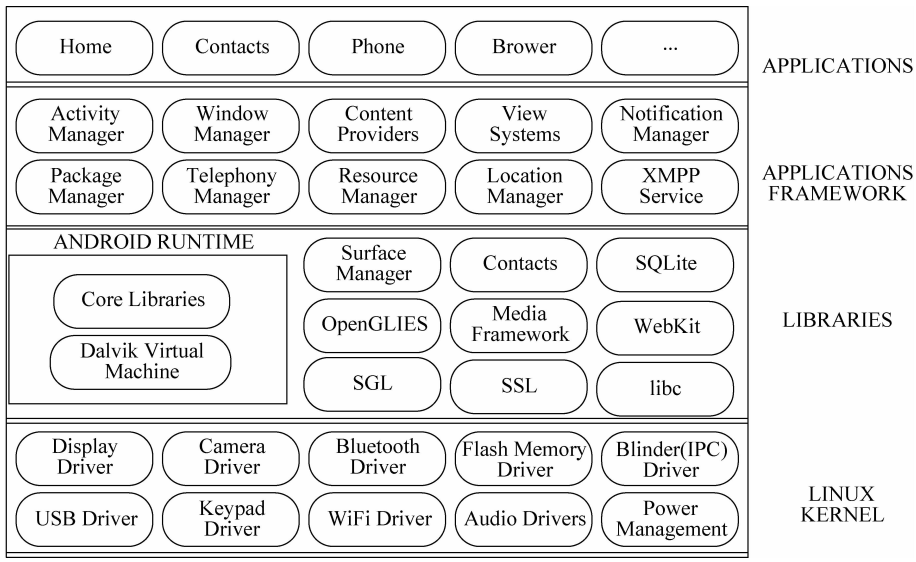

图 3-9　Android 系统结构

能。最低层是 Linux 内核层，Android 的核心系统服务依赖于 Linux 2.6 内核，例如安全性、内存管理、进程管理、网络协议栈和驱动模型等。Linux 内核也同时作为硬件和软件栈之间的抽象层。

3.2.4　客户机/服务器体系结构风格

客户机/服务器(C/S)是 20 世纪 90 年代成熟的一项技术，主要针对资源不对等问题提出的一种共享策略。所谓的 C/S 就是 Client/Server 模式，Client 是请求服务的部分，Server 是提供服务的部分，C/S 结构通过将任务合理分配到 Client 端和 Server 端，降低了系统的通信开销，充分利用了两端硬件环境的优势。Client 和 Server 一般是相距很远的两台计算机，Client 程序将用户的请求提交给 Server 程序，再将 Server 程序处理结果返回给用户；Server 程序接收客户程序提出的服务请求，处理后将结果返回给客户程序。

在 C/S 体系结构中，主要包括数据库服务器、客户机和网络 3 个部分。服务器处理与应用和数据库相关的请求，客户端负责显示数据、处理部分数据以及将用户输入的数据传送给服务器。为了更好地理解 C/S 结构，下面将业务逻辑划分为表示层、功能层和数据层 3 个部分。

(1) 表示层是系统的用户接口部分，也就是人机界面，它是用户与系统交互信息的窗口。它的主要功能是指导操作人员使用自己定义好的服务或函数检查用户输入的数据，显示系统输出的数据。表示层可以不拥有任何企业逻辑，也可根据需要将一部分不经常变化、不涉及企业秘密的应用逻辑放在表示层。

(2) 功能层是应用的主体，它包括了系统中所有重要的和易变的企业逻辑(企业的规划、运作方法、计算条件等)。它要完成的功能通常是接受输入、进行处理并返回结果。表示层和功能层之间的数据交换要尽可能简洁，应尽量避免一次业务处理在表示层和功能层间进行多次数据交换的现象发生。

(3) 数据层由数据库管理系统(DataBase Management System,DBMS)负责管理,包括对数据库数据的读/写和维护存储、数据的访问、数据的完整性约束等工作。它必须能迅速执行大量数据的更新和检索。

早期的 C/S 结构是两层结构,例如早期的 Web 应用体系结构如图 3-10 所示。

在此结构中,表示层和功能层被组合在一起,运行在客户端,通过网络连接访问远端的运行于数据库服务器中的数据层。浏览器向服务器发送一个请求,这个请求包含两个部分,即程序名及其参数或输入。服务器将指定的参数发送给指定的程序,借助于 API 接口,例如业界标准的 SQL 语言,客户端的应用组件从数据库中读取数据,执行程序的运算逻辑,然后把数据送回数据库。在实现两层结构应用时,一个应用的三大组成部件(描述、处理和数据)被分离于两个实体(客户应用代码和数据库服务器)或层次中。一般来说,对于数据库应用程序,DBMS 可以为应用程序提供针对底层结构的管理。应用的服务器部分是运行在远程主机上的数据库引擎,而该应用的客户部分则是运行在本地计算机上的数据库查询程序,它们之间的通信是借助于 DBMS 提供的网络协议实现的,如图 3-11 所示。

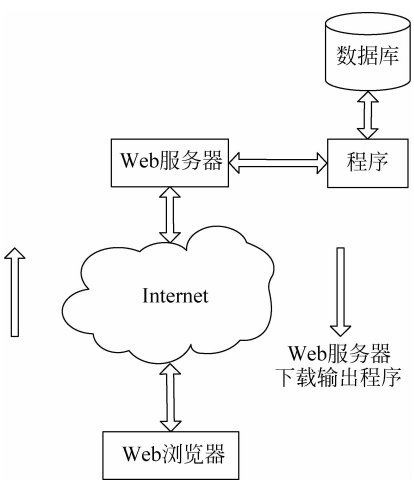

图 3-10 两层的 Web 应用体系结构

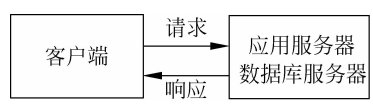

图 3-11 两层 C/S 结构

因此,服务器程序负责管理系统资源,包括管理数据库的安全性、控制数据库访问的并发性、定义全局数据完整性规则以及备份恢复数据库。服务器永远处于激活状态,监听用户请求,为客户提供服务操作。客户机程序负责提供用户与数据库交互的界面、向服务器提交用户请求、接收来自服务器的信息以及对客户机数据执行业务逻辑操作。网络通信软件的主要功能是完成服务器程序和客户机程序之间的数据传输。

在这种结构下,一个功能强大的客户应用开发语言和一个多用途的用于传送客户请求到服务器的机构是整个两层结构的核心。描述只受客户机的操纵,处理由客户机和服务器共同分担,数据由服务器实施存储和访问。在一个数据存取事件中,数据库引擎负责处理从客户端发来的请求。在服务器中,请求还将得到存储逻辑和处理上的优化,例如使用权限、数据完整性和保密性等,数据返回后会在客户机上得到处理,以适应进一步的查询、商业应用、预测分析和报表等各种要求。这种基于 Internet 的客户/服务器结构的 Web 系统简单、实用,而且与用户的接入地点和具体设备无关。两层 C/S 体系结构的处理流程如图 3-12

所示。

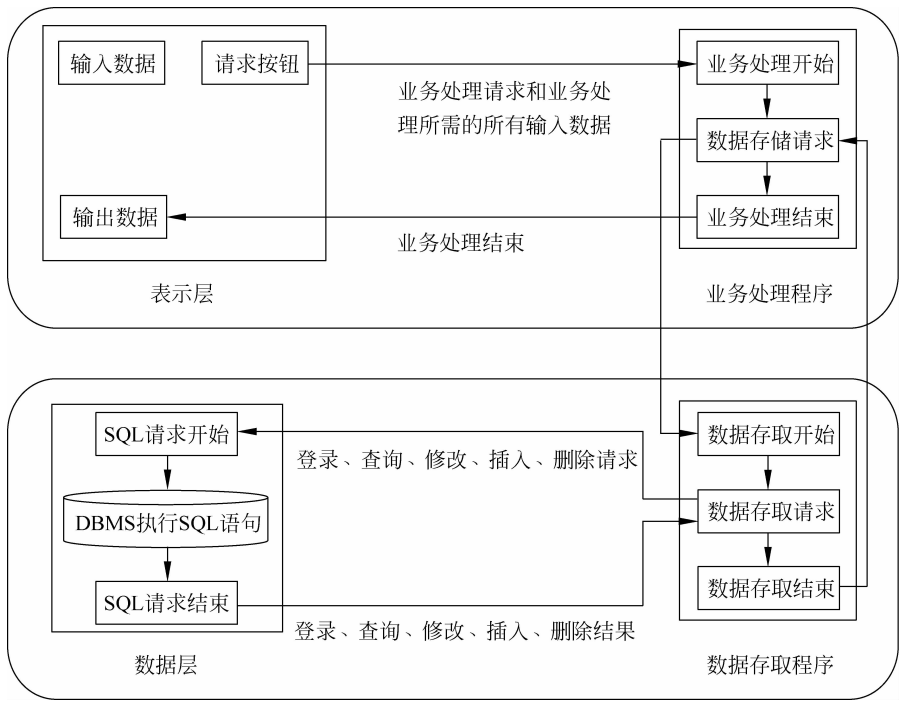

图 3-12　两层 C/S 体系结构的处理流程

C/S 体系结构具有以下优点：

（1）客户机构件和服务器构件分别运行在不同的计算机上，有利于分布式数据的组织和处理；

（2）构件之间的位置是相互透明的，客户机程序和服务器程序都不必考虑对方的实际存储位置；

（3）客户机侧重数据的显示和分析，服务器侧重数据的管理，因此，客户机程序和服务器程序可以运行在不同的操作系统上，便于实现异构环境和多种不同开发技术的融合；

（4）构件之间是彼此独立和充分隔离的，这使得软件环境和硬件环境的配置具有极大的灵活性，易于系统功能的扩展；

（5）将大规模的业务逻辑分布到多个通过网络连接的低成本计算机上，降低了系统的整体开销。

尽管 C/S 体系结构具有强大的数据操作和事务处理能力，其模型构造简单，并且易于理解，但是，随着企业规模的日益扩大和软件复杂程度的不断提高，C/S 体系结构也逐渐暴露出一些问题。

（1）开发成本较高：在 C/S 体系结构中，客户机的软件配置和硬件配置的要求比较高，随着软件版本的升级，对硬件性能的要求也越来越高，从而增加了系统成本，使客户机变得臃肿。

（2）在开发 C/S 结构系统时，大部分工作集中在客户机程序的设计上，增加了设计的复杂度，客户机负荷太重，难以应对客户端的大量业务处理，降低了系统性能。

(3) 信息内容和形式单一：传统应用一般是事务处理型，界面基本遵循数据库的字段解释，在开发之初就已经确定，用户无法及时获取办公信息和文档信息，只能获取单纯的字符和数字，非常枯燥和死板。

(4) 如果对 C/S 体系结构的系统进行升级，开发人员需要到现场更新客户机程序，同时需要对运行环境进行重新配置，增加了维护费用。

(5) 两层 C/S 结构采用了单一的服务器，同时以局域网为中心，因此难以扩展到 Intranet 和 Internet。

(6) 数据安全性不高：客户机程序可以直接访问数据库服务器，因此，客户机上的其他恶意性程序也有可能访问到数据库，无法保证中心数据库的安全。

为了克服两层 C/S 结构的缺点，可以将客户机和服务器中的部分业务逻辑抽取出来，形成功能层，放在应用服务器上，这就是所谓的三层 C/S 体系结构。

三层结构(也称多层结构)是在分布式技术不断发展、成熟的基础上建立起来的，它的基本思想是在分布式技术的基础上，将用户界面和应用的企业逻辑分离，把信息系统按功能划分为表示、应用及数据三大块，分别放置在相同或不同的硬件平台上。三层 C/S 结构将应用的三部分(表示部分、应用逻辑部分、数据访问部分)明确地进行分割，使它们在逻辑上各自独立，并且单独实现，分别称之为客户端、应用服务器、数据库服务器。与两层 C/S 结构相比，其应用逻辑部分被明确地划分出来。三层 C/S 结构同样包括客户端、应用服务器和数据库服务器 3 个部分。三层 C/S 体系结构如图 3-13 所示。

图 3-13　三层 C/S 体系结构

三层 C/S 结构中的功能层是应用的主体，它包括系统中所有重要的和易变的企业逻辑，应用服务器是应用逻辑处理的核心，它是具体业务的实现。客户端将请求信息发送给应用服务器，应用服务器返回数据和结果。应用服务器一般和数据库服务器有密集的数据交往，应用服务器向数据库服务器发送 SQL 请求，数据库服务器将数据访问结果返回给应用服务器。此外，应用服务器也可能和数据库服务器之间没有数据交换，而作为客户的独立服务器使用，负责处理所有的业务逻辑。当应用逻辑变得复杂或增加新的应用时，可增加新的应用服务器，它可与原应用服务器驻留于同一主机或不同主机上。三层 C/S 体系结构的处理流程如图 3-14 所示。

在硬件实现上，三层 C/S 体系结构有两种方式：

(1) 客户位于客户机上，应用服务器和数据库服务器位于同一主机上。这种方式在主机具有良好性能的前提下，能保证应用服务器和数据库服务器之间的通信效率，减少客户和应用服务器之间网络上的数据传输，使系统具有好的性能。

(2) 客户位于客户机上，应用服务器和数据库服务器位于不同主机上。这种方式比前一种方式更加灵活，能够适应客户机数目的增加和应用处理负荷的变动。在增加新的应用逻辑时，可以追加新的应用服务器。当系统规模较大时，这种方式的优点较显著。

上述两种方式在复杂应用下，整个系统达到高性能的关键是应用服务器和数据库服务

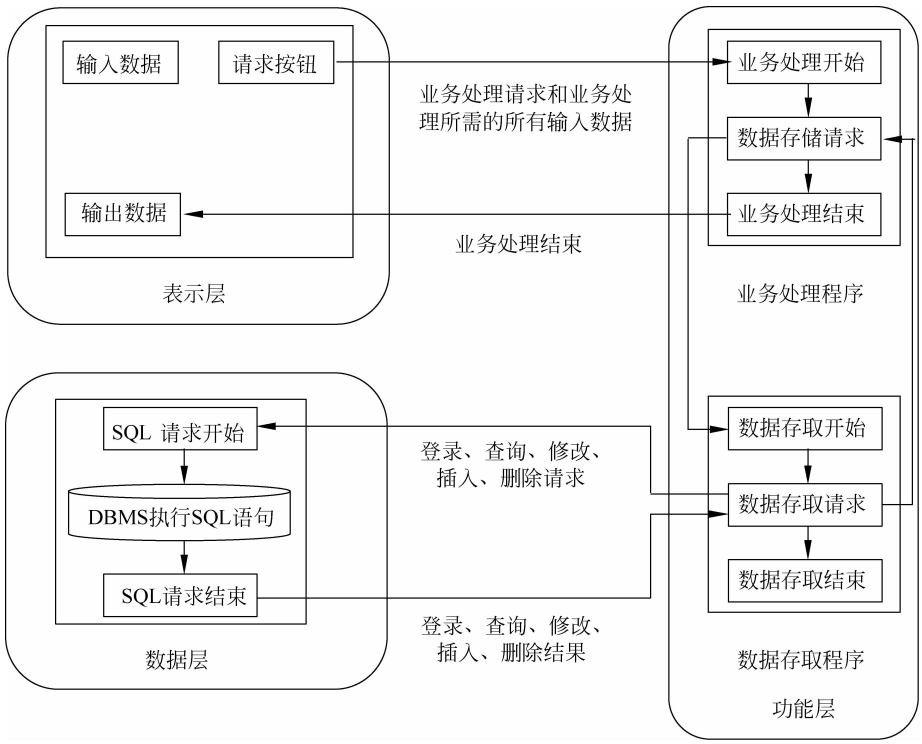

图 3-14　三层 C/S 体系结构的处理流程

器间的数据通信效率，对于应用服务器和数据库服务器位于不同主机上的第二种方式来说显得尤为重要。

两层 C/S 结构在效率、性能方面存在着很大的不足，这种体系的发布系统在其灵活性、扩展性上受到了极大的限制，在此基础上进行系统扩充几乎是不可能的，所以，应采用灵活的三层 C/S 结构。由于业务逻辑层是三层结构的“灵魂”，它把商业逻辑和数据规则从客户端分离出来，形成独立的一层，从而很好地解决了两层结构中出现的弊病。

(1) Client 从与数据库服务器直接连接转变为与中间层的应用服务器连接，通过中间层的服务得到数据。服务与数据库连接是有本质区别的，一个服务可以在应用服务器管理软件(如 MTS)的管理下生成多个实例，同时为申请该服务的多个用户提供服务，这是数据库连接所不能胜任的，这样就有效解决了对用户数量的限制。

(2) 客户端摆脱了业务逻辑的束缚，对业务逻辑的变更不再敏感，给应用程序的维护带来了便利，并且业务逻辑以组件方式存在于中间层服务器上，提高了代码重用的机会。

另外，服务往往可以独立于任何特定的客户应用程序来设计和实现。因此，对于很多应用程序来说，提供了很大的灵活性和重复使用的潜力。三层应用程序可以被看成是服务消费者与服务提供者的逻辑网络。由于三层结构的灵活性好、安全性高、可移植性好，所以，三层 C/S 结构是开发信息发布系统的理想技术。

使用 C/S 体系结构的例子很多，在局域网游戏中大部分使用了该结构。例如紫金桥公司开发的中国象棋游戏，考虑到在局域网内没有人—机对战的内容，紫金桥公司开发了此款游戏。该游戏的大体思路是以一台计算机作为服务器创建游戏，如图 3-15 所示。

图 3-15 创建游戏服务器

然后,以另外一台计算机作为客户端,连接服务器进入游戏,从而达到数据双向传输的目的。注意,作为服务器端,必须开启网络服务器,向外提供数据;作为客户端,必须建立一个数据源,以连接到服务器。假设服务器的 IP 为 192.168.1.6,建立的数据源名为“Server”,建立数据源的图形界面如图 3-16 所示。

游戏的运行界面如图 3-17 所示。

图 3-16 建立一个数据源

另外,在一个实际应用中,可能会有多种应用和平台加入到这个 Client/Server 模型中,这就要求在客户和服务器之间有一组正式的接口以支持这些应用。从结构上讲,这一层位于客户和服务器之间,因而被称为中间件(Middleware)。从概念上讲,中间件是客户从服务器获得服务的“粘合剂”,它的引入使原来较为简单的两层分布模型(客户—服务器)被更加精确的三层模型(客户—中间件—服务器)所替代。从理论上讲,中间件具有以下工作机制:客户端上的应用程序需要从网络中的某个地方获取一定的数据或服务,这些数据或服务可能处于一个运行着不同操作系统和特定查询语言数据库的服务器中,客户/服务器应用程序中负责寻找数据的部分只需访问一个中间件系统即可,由中间件完成到网络中找到数据源或服务,进而传输客户请求、重组答复信息,最后将结果送回应用程序的任务。中间件的工作机制如图 3-18 所示。

近年来,以中间件为框架基础的三层结构 C/S 模式已被广泛证实为建立开放式关键业务应用系统的最佳环境。因为,作为构造三层结构业务应用系统的基础平台,中间件提供了以下两个功能:

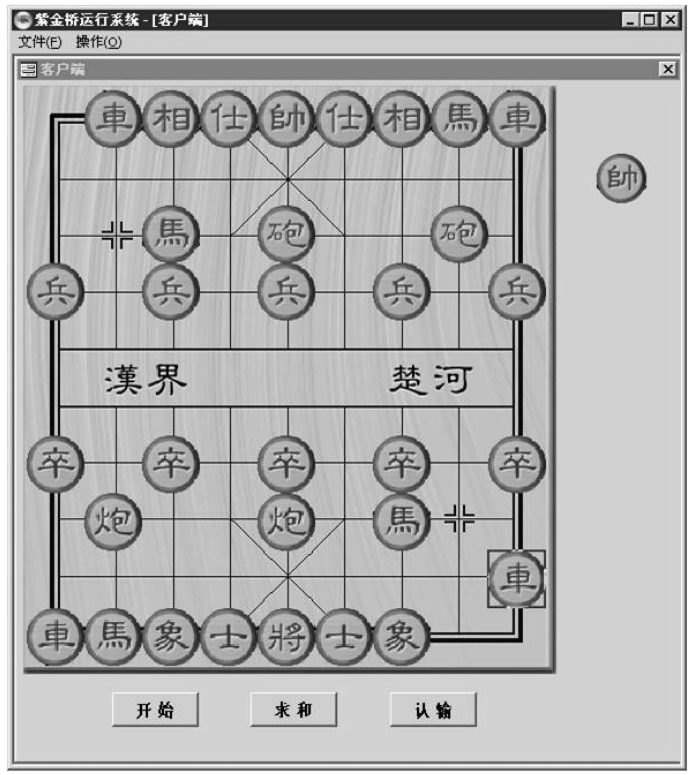

图 3-17　C/S 体系结构中国象棋游戏的运行界面

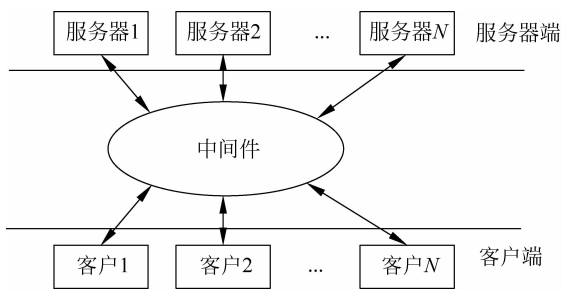

图 3-18　中间件的工作机制

（1）负责客户机与服务器之间的联系和通信，提供了表示层与功能层之间、功能层与功能层之间、功能层与数据层之间、数据层与数据层之间的连接和完善的通信机制。

（2）提供了一个三层结构应用开发和运行的平台，包括支持模块化应用开发的框架；硬件、操作系统、数据库和网络差异的屏蔽；保证事务完整性和数据一致性的事务管理机制；应用的负载均衡和管理功能；应用的高可用性及安全控制机制等。

由此可见，中间件为建立、运行、管理和维护三层 C/S 体系结构的应用提供了一个基础框架，将大大降低应用开发、管理和维护的人力、物力开销，提高其成功率，真正使大型企业应用的高效实现成为可能。

3.2.5 浏览器/服务器体系结构风格

在过去,很多企业的管理软件和办公系统采用 C/S 结构。随着对应用软件要求的进一步提高和应用软件的普及,C/S 结构体系的缺点使其越来越不适应现代管理软件和办公应用软件的要求,而浏览器/服务器结构(Browser/Server,B/S)的出现在很大程度上弥补了 C/S 结构的缺陷,它是对 C/S 结构的一种变化或者改进。在这种结构下,用户界面完全通过 WWW 浏览器实现,一部分事务逻辑在前端实现(主要事务逻辑在服务器端实现),并结合了浏览器的多种 Script 语言(VBScript、JavaScript)和 ActiveX 技术,形成所谓的三层结构。该结构用通用浏览器就能实现原来需要复杂专用软件才能实现的强大功能,大大简化了客户端计算机的载荷,减轻了系统维护与升级的成本和工作量,降低了用户的总成本。B/S 体系结构如图 3-19 所示。

图 3-19 三层 B/S 结构

B/S 结构的工作原理是,将 Web 服务器作为体系结构的核心,将应用程序以网页的形式放在 Web 服务器上。当用户运行某个应用程序时,只需在客户端的浏览器中输入相应的 URL,浏览器会以超文本 HTTP 的形式向 Web 服务器提出访问数据库的要求,当 Web 服务器接收到 HTTP 请求之后,会调用相关的应用程序,同时向数据库服务器发送数据操作请求,数据库服务器得到请求后,验证其合法性,并进行数据处理将结果返回给 Web 服务器。Web 服务器再一次将得到的所有结果进行转化,变成 HTML 文档形式,转发给客户端浏览器并以友好的 Web 页面形式显示出来。

在整个 B/S 结构模式中,用户使用客户浏览器通过互联网向 Web 服务器发送 HTTP 请求,Web 服务器将与用户建立连接,然后根据发来的 HTTP 请求的不同,建立不同的配置。如果请求对象是 HTML 脚本、静态图像等静态资源,Web 服务器将所需的资源从本地的文件系统中读出,然后返回给用户。如果请求的是 CGI、ASP、PHP 等动态资源,Web 服务器将请求发送给相应的 CGI 程序或脚本解释器。应用程序服务器是整个系统的核心所在,系统所提供的功能基本上都是由应用程序服务器完成的。它的作用是寻找能够提供服务的应用对象,并为客户端和服务对象之间提供通道。在 B/S 结构中,数据库是存储数据的主要场所,客户端提交的数据都保存在数据库中,应用对象与数据库建立连接之后,才能对数据库进行相关的操作。

在 B/S 结构中,数据的请求、网页的生成、数据库的访问和应用程序的执行全部由 Web 服务器来完成,当企业对网络应用进行升级时,只需要更新服务器端的软件就可以了,从而大大简化了客户端。在使用系统时,用户仅使用一个浏览器就可以运行全部的应用程序,真正实现了“零客户端”的运作模式,同时,在系统运行期间可以对浏览器进行自动升级。B/S 结构为异构机、异构网和异构应用服务的集成提供了有效的框架基础。此外,B/S 结构与 Internet 技术相结合也成为了电子商务和客户关系管理的基础。B/S 结构模式如图 3-20 所示。

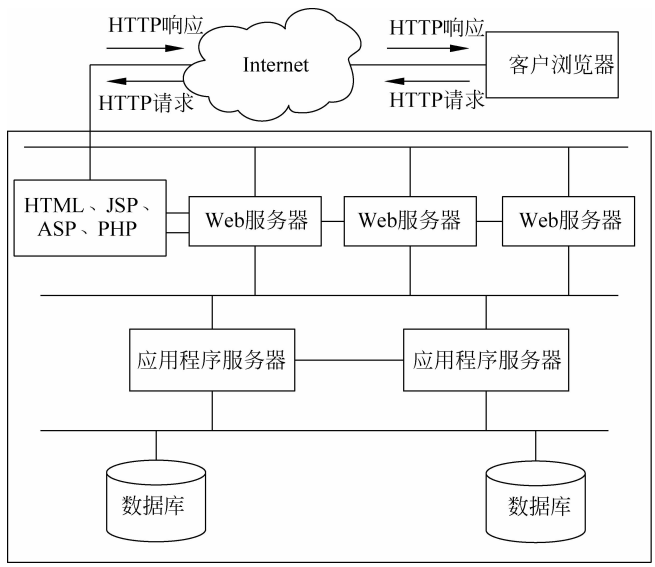

图 3-20　B/S 结构模式

B/S 结构的一个典型例子是哈尔滨理工大学的教务在线系统，如图 3-21 所示。

图 3-21　哈尔滨理工大学的教务在线系统

利用B/S结构开发哈尔滨理工大学的教务在线系统，组织机构、工作职责等各级菜单可使用客户端的浏览器通过Internet访问网站Web服务器，提交相关的HTTP请求，Web服务器响应处理后，根据需要对数据库进行访问操作，将调取的数据处理后生成结果页面返回给用户浏览器，同时根据安全需要，对数据库进行系统备份。该教务在线系统的结构如图3-22所示。

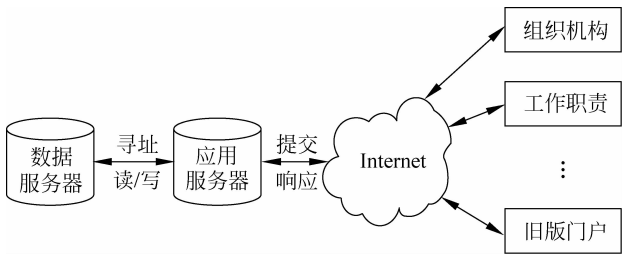

图3-22 哈尔滨理工大学教务在线系统的结构

B/S结构具有很多优点，具体体现在以下方面。

(1) 系统开发、维护和升级方便且经济；升级服务器应用程序时，只需在服务器上升级应用程序即可，系统开发和升级维护比较方便；软件开发、维护与升级的费用相对C/S结构系统大大降低；同时，B/S结构对前台客户机的要求并不高，可以避免盲目进行硬件升级造成的巨大浪费。

(2) 具有很强的开放性：在B/S结构下，用户可以通过浏览器进行访问，系统开放性比较好。

(3) 具有易扩展性：B/S结构由于Web的平台无关性，可以任意扩展，可以从一台服务器、几个用户的工作组扩展成为拥有多用户的大型系统。

(4) 用户界面具有一致性：B/S结构的应用软件都是基于Web浏览器的，而Web浏览器的界面是类似的，用户使用方便，从而可以降低软件的培训费用等。

(5) 具有更强的信息系统集成性：在B/S结构下，集成了解决各种问题的服务，而非零散的单一功能的多系统模式，因此，它能提供更高效的工作效率。

(6) 提供了灵活的信息交流和信息发布服务：B/S结构借助Internet强大的信息传送与信息发布能力，可以有效地解决大量信息交流的问题，可以实现信息的远程访问、共享、查询等。

B/S结构的缺点如下：

(1) B/S结构没有集成有效的数据库处理功能，缺乏对动态页面的支持能力，响应速度相对比较低。

(2) 系统个性化特点相对降低，无法实现个性化的功能要求。

(3) B/S结构软件只安装在服务器端上，用户界面的主要事务逻辑在服务器端完全通过浏览器实现，只有少数事务逻辑在前端实现，所有的客户端只有浏览器，因此，应用服务器运行数据负荷较重，一旦发生服务器“崩溃”等问题，后果不堪设想。

3.2.6　事件驱动体系结构风格

事件驱动结构(Event Driven Architecture,EDA)是由 Gartner 公司于 2003 年提出的。事件驱动体系结构的基本思想是,系统对外部的行为表现可以通过它对事件的处理来实现。事件驱动就是根据事件的声明和发展状况来驱动整个应用程序的运行。如果用户要了解一个系统,只要输入一个事件,然后观察它的输出结果即可。一个基于事件驱动构架的应用程序系统,各个功能设计为封装的、模块化的、可用于共享的事件服务组件,并在这些独立非耦合的组件之间将事件所触发的信息进行传递。

事件驱动结构提供了一个动态响应事件的机制。事件驱动结构能够快速过滤、聚合和关联企业业务事件,从复杂的业务中快速提取事件并进行类型判断,从而帮助企业迅速而准确地处理事件所反映的业务问题。在一个事件驱动结构系统里,事件有生产者和消费者,定义了事件的来源和去向。事件的生产者将业务活动中的重要事件发布出去,快速及时地传递给感兴趣的订阅者。然后订阅方根据事件的重要性快速地获得业务事件的信息并做出反应,从而实时地响应企业业务活动中的事件。事件驱动的动作机制帮助系统激活相应的后续事件,完成业务流程。

事件驱动结构是由一系列系统组件构成的,组件之间共同作用完成系统的功能。如图 3-23所示,这些组件之间的连接是管道化的和多模块化的,通过形成并发的事件流对企业业务事件进行处理。

事件驱动结构的组成如图 3-24 所示。

用户可以将这些不同的组件细分为以下五类,在某个具体实现中,可能会包含五类中的多类。

(1) 事件元数据:如同数据库的元数据存储一样,事件驱动结构也必须有事件元数据,用来实现事件定义和事件处理规则预定义。事件定义包括事件格式定义、事件格式转换、事件生产者、事件消费者和事件处理引擎。事件处理规则是事件驱动结构的核心元数据。

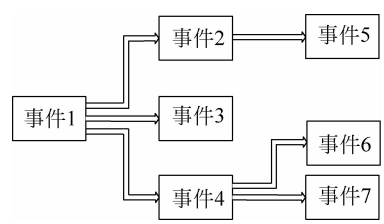

图 3-23　事件驱动结构的模式

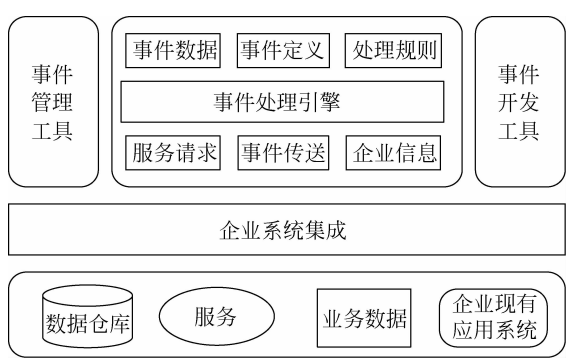

图 3-24　事件驱动结构的组成

(2) 事件处理:事件处理包括事件处理引擎和事件处理对象实例两部分。事件处理引擎按照所处理的事件类型分为简单事件处理和复杂事件处理两种。事件对象实例用于判

断、分析事件趋势。

(3) 事件工具：事件工具由事件开发工具和事件管理工具组成。前者用来定义事件的规格、事件处理规则、事件消费者等。而事件生产者、事件流、事件处理元素和事件处理行为的统计分析的管理和监控属于事件管理工具的范畴。

(4) 企业系统集成：事件驱动结构中为了实现结构与现有系统的融合及事件的提取和传递，必须有一个连接枢纽，这就是企业系统集成。企业系统集成提供了事件过滤、事件变换、事件发布、事件传送、服务请求、服务订阅、服务访问等的连接通道。

(5) 事件资源：事件驱动结构是在现有的企业资源的基础上建立的，包括企业已有的各种资源，如业务数据、数据仓库、服务、现有系统等。事件资源是企业事件的来源和基础，为事件驱动的后续动作提供基础事件数据。事件驱动结构在各企业中具体应用时，根据企业实际的事件流、对象、事件源等的不同，其组件构成的系统结构会呈现出各种特色。

事件驱动体系结构在很多领域中均有应用，例如，当其成为一项会计术语时，便可作为会计信息系统对象的经济实体中的一项业务(事件)，一经发生，会计业务(事件)处理程序就被触发。也就是说，会计信息的采集、存储、处理、输出嵌入在业务执行处理过程中，实现财务业务一体化数据处理模式。基于对事件驱动机制的分析，可以建立如图 3-25 所示的会计信息系统的数据处理模式。

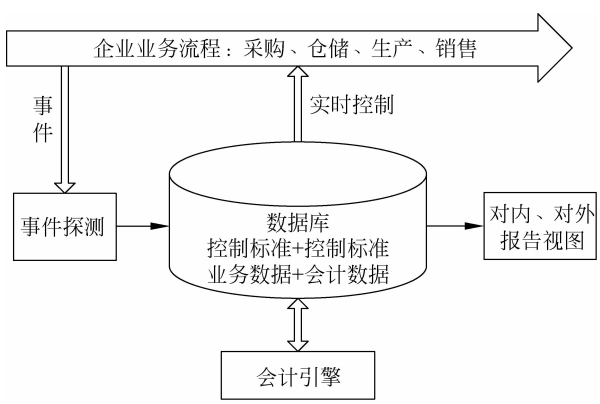

图 3-25　会计信息系统的数据处理模式

事件驱动体系结构的另外一个实例是基于 Reich 模型的飞行控制系统。现今的空域容量日益成为制约航空业快速发展的瓶颈之一，对飞机在航线飞行中偏离航线情况进行研究，建立碰撞危险模型是间隔标准研究的基础理论。Reich 模型是由英国的 P. G. Reich 针对平行航路系统中相邻航线飞机之间单个碰撞危险建立的数学模型。在 Reich 模型中，重要的假设条件是管制员仅负责将飞机引导到正确的飞行路径上，然后飞机自主导航飞行。当试图在 Reich 模型中加入危险分析要素时，其模型需要使用基于快照概念的概率密度函数等非常复杂的数学运算。开发基于 Reich 模型的飞行控制 GUI 系统可选择基于事件的隐式调用体系风格，因为基于事件的隐式调用体系结构风格把系统对象作为封装体隐藏了其复杂性，对象的设计接口与实现相分离，使得实现的修改具有局部化的特性。基于事件的体系结构风格能够降低对象之间的耦合度，对象之间的交互不是采用直接的过程调用而是采用广播等事件通信方式。采用基于事件的隐式调用体系结构风格普遍适用于 GUI 系统的开

发，能够隐藏系统内部数学运算的复杂性，简化使用者的操作过程。

事件驱动体系结构风格具有以下优点：

(1) 事件声明者不需要知道哪些构件会响应事件，因此，不能确定构件处理的先后顺序，甚至不能确定事件会引发哪些过程调用。

(2) 提高了软件重用能力，只要在系统事件中注册构件，就可以将该构件集成到系统中。

(3) 便于系统升级，只要构件名和事件中所注册的过程名保持不变，原有构件就可以被新构件所替代。

但是，事件驱动体系结构风格也存在着一些问题，具体表现在以下方面：

(1) 构件放弃了对计算的控制权，完全由系统来决定。当构件触发一个事件时，它不知道其余构件是如何对其进行处理的。

(2) 存在数据传输问题，数据可以通过事件来进行传输，但是，在大多数情况下，系统本身需要维护一定的存储空间，这将对系统的逻辑功能和资源管理有一定影响。

3.2.7　数据共享体系结构风格

数据共享体系结构风格也称为仓库体系结构风格。这种风格的典型代表有数据库系统、超文本系统、黑板系统。在该风格中主要有两类部件，一类是中心数据结构部件，又可称为数据仓库(Repository)，表示系统的当前状态；另一类是一组相对独立的部件集，它们可以用不同方式与数据仓库进行交互，这也是数据共享体系结构的技术实现基础。

由于系统功能各不相同，信息交换模式也不会完全一样，不同的交换模式导致了不同的数据和状态控制策略。根据所使用的控制策略不同，数据共享体系结构主要有两大分支：如果系统输入业务流的类型是激发进程执行的主要原因，则数据仓库是传统的数据库；如果中心数据结构的当前状态是激发进程执行的主要原因，则数据仓库是黑板(Blackboard)，其中，黑板体系结构风格主要应用于需要进行复杂解释的信号处理领域中，例如语音与模式识别等。

之所以称为黑板，其原因是它反映了信息共享，如同教室里的黑板一样，其模拟一组(围坐在桌子边讨论一个问题的)人类专家，对于同一个问题或者一个问题的各个方面，每一位专家都根据自己的专业经验提出自己的看法，写在黑板上，其他专家都能看到，都能随意使用，从而共同解决这个问题。在黑板体系结构中，可以有多个人读“黑板”上面的字，也可以有多个人在上面写字。黑板体系结构如图 3-26 所示。

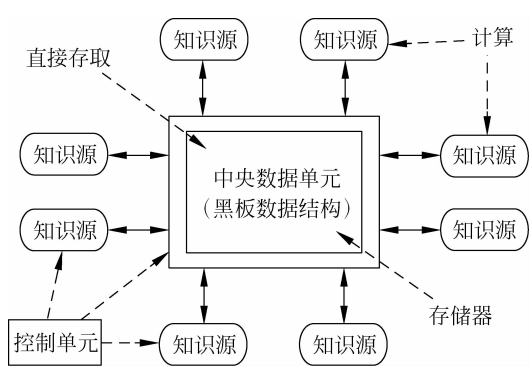

图 3-26　黑板体系结构

黑板系统是一种问题求解模型，它将问题的解空间组织成一个或多个与应用相关的分级结构。分级结构的每一层信息由一个唯一的词汇来描述，它代表了问题的部分解。与领域相关的知识被分成独立的知识模块，它将某一层次中的信息转换成同层或相邻层的信息。各种应用通过不同的知识表达方法、推理框架和控制机制的组合来实现。影响黑板系统设计的最大因素是应用问题本身的特性，但是支撑应用程序的黑板体系结构有许多相似的特征和构件。对于特定的应用问题，黑板系统可以通过选取各种黑板、知识源和控制模块的构件来设计，也可以利用预先定制的黑板体系结构的编程环境建造。

下面介绍黑板系统的组成。

(1) 知识源：知识源是用于存储解决问题所需相关知识的独立模块，是问题求解的与领域相关的知识。每个知识源的目标是为问题求解提供信息，它由条件部分与动作部分组成，可以表达为过程、规则集或逻辑命题。知识源伺机对黑板中发生的变化做出反应。如果当前状态满足知识源的条件部分，则该知识源即被触发，执行动作部分，产生一个新的状态。

黑板系统中讨论的知识局限于知识源，即活动知识，它由算法、启发式规则组成，能将黑板的一种状态变换成另一种状态。其他难以用算法或规则表示的领域知识，例如定义、分类等“静态”知识，最好用对象、框架或表格来表达。

(2) 黑板：黑板是一个全局数据库，用于保存输入数据、中间结果以及在解决许多不同问题时所需的其他数据，这些数据以松散结构来进行存储。黑板是一个存放问题求解状态数据的全局存储结构，由输入数据、部分解、备选方案、最终解和控制数据等对象组成。它可以划分成多个子黑板，即解空间可划分成多个分级结构。每个分级结构中的节点模板是预先确定的，但节点实例是动态创建的。黑板结构的设计实质上是对问题求解方案的一种设计。

黑板结构设计首先是黑板的概念设计，即确定哪些状态变化需要记录在黑板中，如何划分数据结构；其次要决定是动态地还是静态地划分黑板系统，黑板能否重构；同时，还要决定知识源及其知识的表示方法。

(3) 控制组件：控制组件主要作用于问题解决过程中运行时间及其他相关资源的分配。它对黑板上发生的变化进行监控，决定下一步采取的行动。各种类型的信息对于控制机制是全程可存取的，这些信息可以存放在黑板上或另外单独存放。控制信息被用来决定关注的焦点以指出下一个被处理的对象。

控制组件设计是黑板系统设计中最复杂的任务，可变性最多，目标是在恰当的上下文中选择和运用恰当的知识源。

黑板体系结构风格具有以下优点：

(1) 便于多客户共享大量数据，而不必关心数据是何时产生的、由谁提供的以及通过何种途径来提供。

(2) 便于将构件作为知识源添加到系统中。

但是，黑板体系结构风格也存在着一些问题，具体表现为以下方面：

(1) 对于共享数据结构，不同知识源要达成一致，因为要考虑各个知识源的调用问题，这会使共享数据结构的修改变得非常困难。

(2) 需要同步机制和加锁机制来保证数据的完整性和一致性，增大了系统设计的复杂度。

编译器可被认为是数据共享体系结构的一个实例。编译器结构如图 3-27 所示。从图 3-27 中可以看出，编译器通过不同模块访问、更新解析树和字符表完成源代码到目标代码的转换工作，同时，源代码调试器、句法编辑器也需要访问解析树和字符表。

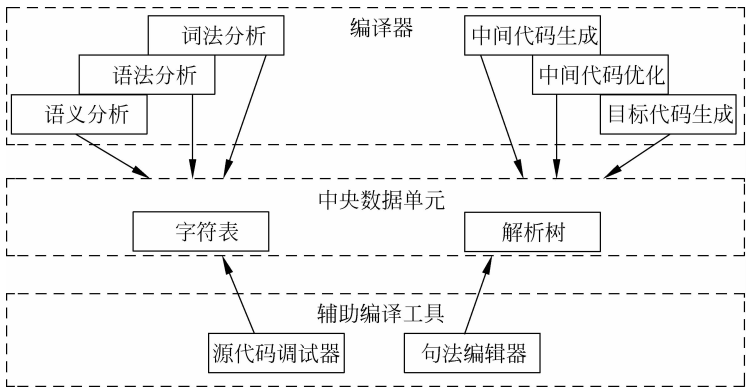

图 3-27　数据共享结构的编译器

3.2.8　解释器体系结构风格

解释器作为一种体系结构，主要用于构建虚拟机，以弥合程序语义和计算机硬件之间的间隙。实际上，解释器是利用软件创建的一种虚拟机，因此，解释器风格又被称为虚拟机风格。

如果程序的逻辑功能很复杂，需要采用复杂的方式来进行操作，一个较好的解决方案是提供面向领域的虚拟机语言。用户使用虚拟机语言来描述复杂操作，解释器执行这种语言序列，产生相应的动作行为。在解释器结构中，主要包括一个执行引擎和 3 个存储器。

解释器系统由 4 个部分组成：被解释的程序、执行引擎、被解释程序的当前状态和执行引擎的当前状态。系统的连接件包括过程调用和直接存储器访问。解释器只接受符合语法规则的程序作为它的输入，经过每个状态的执行，最终得出它的结果是否为真。如果为真，则找到它的一个模型，如果为假，则会提示错误。解释器的框架如图 3-28 所示。

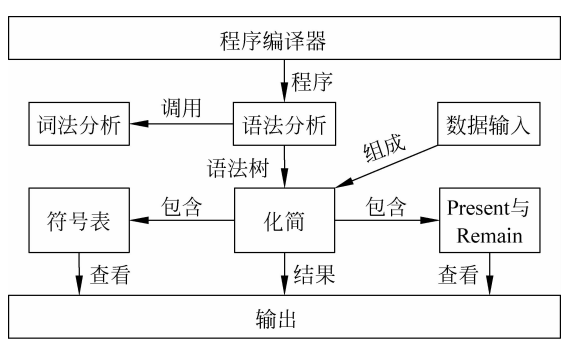

图 3-28　解释器框架

解释器一共包含 8 个模块。第一个模块程序编译器用来编辑所要输入的程序。因为解释器只接受符合一定规则的程序，所以它需要语法分析模块来检测程序是否有语法错误，在这个过程中，需要首先调用词法分析器来检测程序中是否存在词法错误。经过词法分析和

语法分析以后，如果程序中不存在错误，则会得到一个程序的语法树，它将会在化简模块中得到执行。在化简模块中，解释器将遍历这个语法树并把它化简为范式的形式，化简结束后，程序就要终止并输出结果，在化简过程中，如果需要用户输入数据，则调用数据输入模块。符号表模块用来处理程序中各个状态所出现的全局变量，在程序执行结束的时候，符号表会被删除。Present 与 Remain 模块则用来存储每个状态中程序范式的两个部分，该模块的内容也可以在输出模块中进行查看，以观测程序化简的整个过程。

目前，解释器体系结构有许多现实应用，可以将其作为整个软件系统的一个组成部分。以下是一些具体的应用实例：

(1) Java 和 Smalltalk 的编译器。

(2) 基于规则的系统，例如专家系统领域中的 Prolog 语言。

(3) 脚本语言，例如 Awk 和 Perl。

解释器的另外一个例子是手机浏览器，其使用了 JavaScript 解释器，例如 WebKit 浏览器内核、Gecko 浏览器内核以及基于 Java ME 的手机 JavaScript 解释器等，基于手机中间件平台设计的 JavaScript 解释器系统的结构如图 3-29 所示。

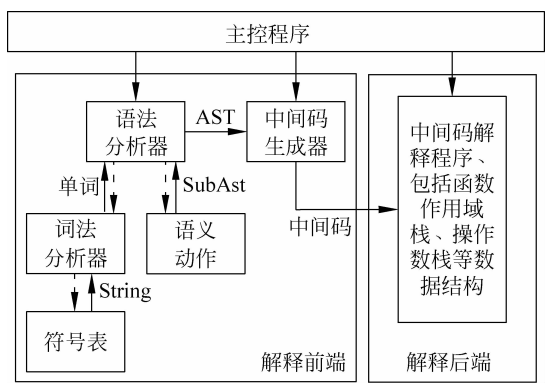

图 3-29 JavaScript 解释器系统结构图

从图 3-29 可以看出，整个 JavaScript 解释器靠一个主控程序来驱动，整个解释器按照功能分成解释前端和解释后端，而维系两端的是中间码。

在系统前端中，语法分析模块驱动词法分析和语义动作模块，前端系统通过语法分析模块分析动作的驱动，调用词法分析模块中产生式归约所需要的单词符号，当一个产生式归约成功后，调用语义动作模块生成抽象语法子树。当一个可执行语句集归约完毕后，与之对应的抽象语法树(AST)构造完毕。由 JavaScript 解释器主控模块对抽象语法树进行遍历，生成自定义格式的中间字节码(简称中间码)。对于后端系统而言，前端系统产生的中间码即为所需的目标码，对其进行解释执行并输出结果即可。在前端系统中，根据 JavaScript 词法规则构造确定有限自动机(DFA)，由确定有限自动机可以很容易地构造词法分析器。语法分析和语义部分则采用 LALR 分析方法和语法制导翻译方法。前端系统产生的中间代码采用自定义的字节码，采用字节码的好处是字节码与平台无关，在不同的平台上使用不同的解释器对它进行解释执行，即可实现在字节码级与各平台兼容，而不仅仅局限于某个手机平台，不必对字节码做任何修改。解释器体系结构风格具有以下优点：

(1) 能够提高应用程序的移植能力和编程语言的跨平台移植能力。

(2) 实际测试工作可能非常复杂,测试代价极其昂贵,具有一定的风险性,但可以利用解释器对未实现的硬件进行仿真。

但是,解释器体系结构风格也存在着一些问题,具体表现为以下方面:

(1) 由于使用了特定语言和自定义操作规则,因此增加了系统运行的开销。

(2) 解释器系统难以设计和测试。

3.2.9　C2 体系结构风格

C2 体系结构风格最开始用来设计具有用户界面的应用程序,用户界面软件常常可以看作是由大量应用软件片段所组成的,对用户界面领域的软件重用也只是局限于一些窗口小部件代码(Widget)的重用。这种体系结构风格使得对话框、各种抽象程度的结构图形模型以及约束管理器等用户界面构件很自然地得到了重用。随着软件体系结构风格研究的不断深入,C2 体系结构风格逐渐融合了许多其他体系结构风格的特点,也可以用来支持其他类型的应用程序的开发。

C2 体系结构风格是一种基于构件和消息的结构风格,可用来创建灵活的、可伸缩的软件系统。用户可以将 C2 结构看作是按照一定规则由连接件(如消息路由设备)连接的许多构件组成的层次网络,在这个结构中,构件和连接件都有一个“顶部”和“底部”;一个构件的“顶部”或“底部”可以连接到一个连接件的“底部”或“顶部”;对于一个连接件,与其相连的构件和连接件的数量没有限制,但是构件和构件之间不能直接相连。C2 体系结构风格如图 3-30 所示。

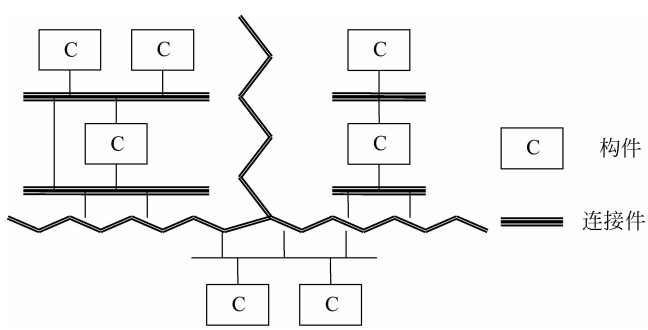

图 3-30　C2 体系结构风格图

C2 体系结构风格定义了两种类型的消息:向上发送的请求消息(Requests)和向下发送的通知消息(Notifications)。请求是通过向上层构件发送消息来获得某种服务,而通知则是告知构件的内部状态发生了改变。连接件负责消息的路由、过滤和广播,构件之间不存在直接的通信手段,而只能通过交换连接件发送的异步消息进行通信。

请求消息是由下层构件的对话框产生的,用来请求它上层的构件执行某些操作。正如构件可以产生什么样的通知消息一样,构件可接收什么样的请求消息也是由构件内部对象的接口来决定的。两者的不同之处在于,通知消息是声明对象的什么接口被调用、参数是什么和返回值是什么;而请求消息则是声明期望调用对象的某一可访问的功能函数。

1. C2 构件

C2 构件有自己的状态、控制线程，必须包括顶层域（Top Domain）和底层域（Bottom Domain）。顶层域规定了该构件所能响应的通知消息集，以及它能向上产生的请求消息集；底层域规定了该构件所能向下产生的通知消息集，以及它能响应的来自于下层的请求消息集。构件可以根据需要来定义对话框约束器部分的功能，它通常包括下面 3 种功能：

（1）对构件上方的连接件发送过来的通知消息提供响应；

（2）对构件下方连接件产生的请求消息执行相应的操作；

（3）维护一些在对话框中定义好的约束条件。

2. 连接件

在 C2 体系结构风格中，连接件负责把 C2 构件绑定在一起，其上可以连接任何数量的 C2 构件和连接件。连接件的主要职责是消息的路由和广播，次要职责是消息的过滤。连接件可以提供多个消息过滤和广播策略，主要有以下几种情况：

（1）不过滤消息；

（2）通知消息过滤；

（3）优先过滤策略；

（4）消息屏蔽。

3. 域翻译器

在 C2 体系结构风格中，构件不知道它下面构件的接口，可以说，构件是独立于下层构件的。这种构件相对于下层构件的独立性有利于在 C2 体系结构风格中对构件进行替换和重用。另一方面，构件要向上发出请求消息，那么它必须知道上层构件的底层域。基于以上原因，在 C2 体系结构风格中引入了域翻译器的概念。域翻译器就是把构件所收到的请求和通知消息转换成它能识别的特定形式。

图 3-31 所示的是一个简单系统的 C2 结构。该系统由一个连接件和两个 C2 构件组成。上面的 ADT 构件负责存储二叉树的抽象数据类型（ADT），下面的 Artist 构件负责把这棵树的结构表示出来。在这个系统中，ADT 构件可能要产生一个通知消息以表明有一个新的元素插入到树中，这个消息是由监视其内部对象的包装器自动产生的。当此通知消息到达连接件下面的 Artist 构件时，Artist 构件就会对它的内部对象进行操作以更新这棵树的表示。

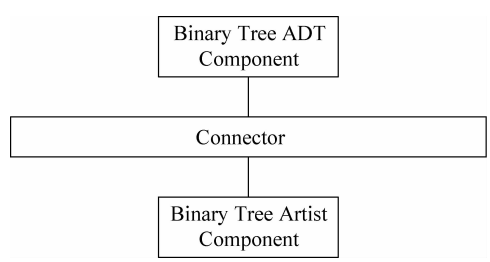

图 3-31 C2 体系结构风格的一个简单实例

3.2.10 MVC 体系结构风格

设计模式就是通过一系列的对象以及对象间的关系，对某一特定的软件设计问题提供一个久经检验的、可扩展的方案，MVC（Model-View-Controller）模式无疑是其中最广为人

知、最著名的设计模式。

MVC 模式属于结构型设计模式，即将应用类和对象组合获得比较复杂的结构，它是第一个将表示逻辑和业务逻辑分开的设计模式。MVC 设计模式的出现使得模型层、视图层和控制层层次分明，各个模块之间相互独立，提高了灵活性。MVC 的核心是实现三层甚至多层的松散耦合，它将应用程序抽象为 3 个部分，这 3 个部分既分工又合作地完成用户提交的每一项任务。

(1) 视图(View)：视图代表用户交互界面，对于 Web 应用来说，可以概括为 HTML 界面，但也有可能是 XHTML、XML 和 Applet。

(2) 模型(Model)：模型就是业务流程状态的处理以及业务规则的制定。

(3) 控制器(Controller)：可以将控制器理解为从用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。

当系统采用 MVC 模式设计系统结构并利用 JavaEE 技术实现时，设计整体框架如图 3-32 所示。

(1) 总控模块：用户请求由总控模块进行统一管理，总控模块作为用户请求总的入口，接收用户请求，调用业务逻辑处理。

(2) 业务逻辑处理：业务逻辑处理模块是用户请求的具体功能的实现部分。

(3) 表示逻辑处理：业务逻辑处理结束后，表示逻辑(JSP、XML、HTML 等)从业务逻辑处理中提取处理结果进行显示。

(4) 数据访问：数据库服务模块必须对数据库资源的访问进行有效的管理，考虑到系统的可扩展性，数据库服务定义统一数据库访问接口，可自行扩展数据源和数据库的访问方式。

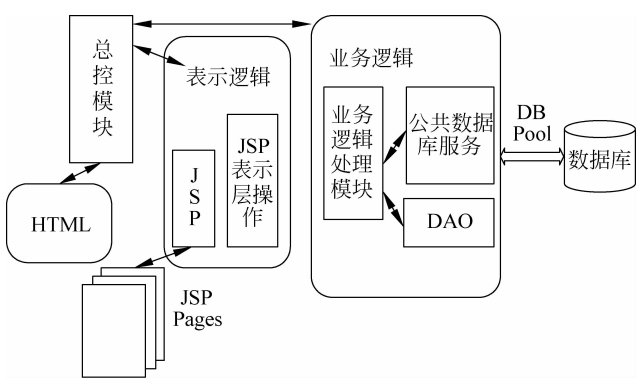

图 3-32　系统整体框架

3.2.11　反馈控制环体系结构风格

在开发过程中，通常把软件看成一种算法，包括输入、计算和输出。这种软件模型只具有“开环控制”特性，不允许有任何的外部扰动。当系统的执行受到外部因素干扰时，这种模型就不再适用了。为了解决这一问题，需要采用控制系统。

控制的目的是使被控对象的功能和属性达到理想的目标，即满足最终的要求或在一定

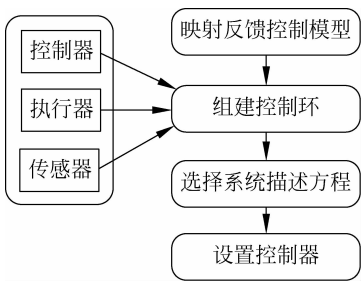

图 3-33 反馈控制环设计过程

的约束条件下达到最优值。为了成功地设计控制系统，用户必须知道被控对象的特征和属性，还必须知道在条件发生改变时这些属性的变化范围。在运行过程中，控制系统需要对被控对象的属性进行测量，并由此来制定相应的控制策略，使系统最终达到理想状态。

反馈控制环结构的思想源自过程控制理论，将过程控制理论融入软件体系结构中，从过程控制的角度分析和解释构件之间的交互，同时，应用这种交互改善系统性能。

若要应用反馈控制策略提供性能保证，首先要设计出满足用户需求的反馈控制环，该过程如图 3-33 所示。

(1) 映射反馈控制模型：根据应用程序的性能要求，抽象并设计出一个反馈控制环，将参数等映射成控制环的相应控制变量、状态变量、设定值。由于不同应用系统有不同的性能要求，因此，需要提供多种反馈控制结构以满足不同的应用。目前，学术界已提出的反馈控制模型有单项性能收敛确保、资源预留控制、任务优先级确保、静态互用确保、任务相对性能确保、利用率最大化控制等。

(2) 组建控制环：用于从构件库中选择适当的控制器、执行器和传感器来组建指定控制环。

(3) 选择系统描述方程：用于为被控系统确定一个适当的微分方程来描述实际系统的动态过程。

(4) 设置控制器：根据设计好的反馈控制环、系统描述方程为控制器设置适当参数，以便让实际系统获得期望的瞬态性能与稳态性能。

需要说明的是，基本控制构件(控制器、执行器、传感器)是以构件库的形式实现的，开发人员通过宏定义来构造所需的控制环，不同的控制环代表不同的性能确保需求。此外，构件库是一个可扩展库，控制工程师可以根据性能要求设计出新的控制模型和控制结构，然后软件工程师根据控制工程师的要求开发出相应的基本控制构件，扩展原有构件库。

完成上述工作之后，一个配置好的面向具体应用的反馈控制环便可用在动态与不确定环境下提供指定的性能确保，该过程与控制工程师配置一个分布式过程控制系统类似，不同之处在于，这里的控制环所控制的是软件性能而非物理过程。通过这样的设计，可以很好地将系统功能实现与系统性能确保分开，使软件工程师能更好地专注于系统功能的实现，而将有关性能确保方面的工作交给控制工程师解决。这样，软件工程师不必关心反馈控制理论却能利用这种理论为系统提供性能确保，而控制工程师不必关心控制环是如何构造的，以及用何种形式向系统提供服务的。

反馈控制环结构能够处理复杂的自适应问题，其中，机器学习就是一个典型的实例。机器学习模型如图 3-34 所示。首先将训练样本输入到学习构件中作为被查询的基本数据和知识源；然后输入真实数据，经过学习构件的分析和计算，输出学习结果。与此同时，检测构件要检查学习结果与预期结果之

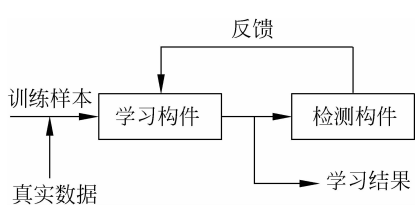

图 3-34 机器学习模型

间的差异，并反馈给学习构件。通过引入反馈机制，使学习构件的学习能力得到增强，丰富了知识源。

3.2.12　公共对象请求代理体系结构风格

公共对象请求代理(Common Object Request Broker Architecture，CORBA)是由对象管理组织(Object Management Group，OMG)提出的，是一套完整的对象技术规范，其核心包括标准语言、接口和协议。在异构分布式环境下，可以利用 CORBA 来实现应用程序之间的交互操作，同时，CORBA 也提供了独立于开发平台和编程语言的对象重用方法。

在 1991 年，对象管理组织提出了 CORBA 1.1，经过不断的努力和完善，随后推出了 CORBA 2.0 以及 CORBA 3.0。目前，对象管理组织制定的 CORBA 标准已经成为分布对象计算技术的一个重要标准，也已经被标准机构和众多软件公司广泛采纳。对象管理组织定义了对象管理体系结构(OMA)作为分布在异构环境中的对象之间交互的参考模型。对象管理组织由 5 个部分组成，即对象请求代理(ORB)、对象服务、通用设施、域接口和应用接口。对象请求代理实现客户和服务对象之间的通信交互，是最核心的部分，其他 4 个部分则是构架于对象请求代理之上适用于不同场合的部件。CORBA 标准就是针对对象请求代理系统制定的规范，图 3-35 给出了 CORBA 系统的体系结构图。

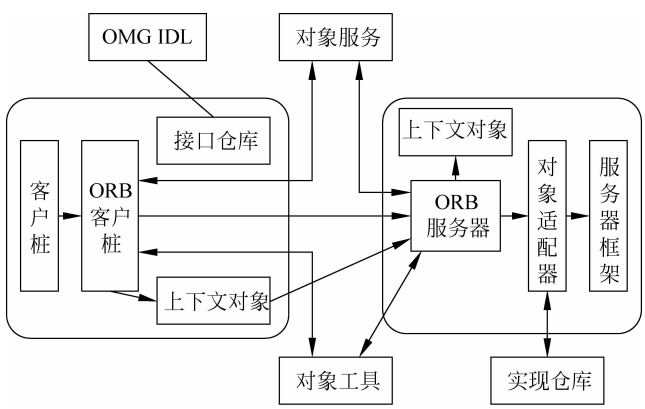

图 3-35　CORBA 系统的体系结构图

(1) 对象请求代理(ORB)：在传统的基于客户机/服务器模式的应用程序的开发过程中，项目开发人员遵循公开的标准或自由设计模块间的协议，这样的协议依赖于网络类型、实现语言、应用方式等。引入对象请求代理后，客户只要遵循服务对象的对外接口标准向服务对象提出业务请求，由对象请求代理在分布式对象间建立客户服务对象关系。对象请求代理的作用包括接受客户发出的服务请求，完成请求在服务对象端的映射；自动设定路由寻找服务对象；提交客户参数；携带服务对象计算结果返回客户端等。

对象请求代理通过一系列接口和接口定义中说明的要实现操作的类型，确定提供的服务和实现客户与服务对象通信的方式。通过 IDL 接口定义、接口库或适配器的协调，对象请求代理可以向客户机和具备服务功能的对象提供服务。基于对象请求代理实现的不同类型接口，一个客户端请求可以同时访问多个由不同对象请求代理实现通信管理的对象引用。

对于对象请求的实现方式，CORBA 规范中定义客户程序可以用动态调用接口

(Dynamic Invocation Interface,DII)的方式或通过 OMG IDL 桩文件经编译后在客户端生成的桩方式提出服务请求。这两种实现方式的区别在于,通过 OMG IDL 桩文件方式实现的调用请求中,客户能够访问的服务对象方法取决于服务对象所支持的接口;而动态调用接口调用方式则与服务对象的接口无关。尽管实现调用请求的方式有所区别,但客户发出的请求服务调用的语义是相同的,服务对象不分析服务请求提出的方式。

(2) 接口仓库:CORBA 引入接口仓库的目的在于使服务对象能够提供持久的对象服务。接口仓库由一组接口仓库对象组成,代表接口仓库中的接口信息。接口仓库提供各种操作来完成接口的寻址、管理等功能。

(3) 对象适配器:对象适配器是为服务对象端管理对象的引用和实现引入的。CORBA 规范中要求系统实现时必须有一种对象适配器。对象适配器完成以下功能:

① 生成并解释对象的引用,把客户端的对象引用映射到服务对象的功能中;

② 激活或撤销对象的实现;

③ 注册服务功能的实现;

④ 确保对象引用的安全性;

⑤ 完成对服务对象方法的调用。

对象请求代理的互操作性(Inter Operability)体现在分布于网络中的多个对象借助 Internet 对象请求代理间协议和通用对象请求代理间协议,达到不同厂商对象请求代理之间操作的一致性。在 CORBA 规范中定义了两种在不同厂商对象请求代理间进行通信的协议,即通用对象请求代理间互操作协议(General Inter-ORB Protocol,GIOP)和环境相关的对象请求代理间互操作协议(Environment Specific Inter-ORB Protocol,ESIOP)。这两种协议屏蔽了操作系统类型、实现语言以及具体厂商等因素。

CORBA 体系结构风格具有以下优点:

(1) 实现了客户端程序与服务器程序的分离,客户不再直接与服务器发生联系,而仅需要和对象请求代理进行通信,客户端和服务器之间的关系显得更加灵活。

(2) 将分布式计算模式与面向对象技术结合起来,提高了软件重用效率。

(3) 提供了软件总线机制。软件总线是指一组定义完整的接口规范。应用程序、软件构件和相关工具只要具有与接口规范相符的接口定义,就能集成到应用系统中。这个接口规范是独立于编程语言和开发环境的。

(4) CORBA 支持不同的编程语言和操作系统,开发人员能够在更大的范围内相互利用已有的开发成果。

CORBA 充分利用了现有的各种开发技术,将面向对象思想融入分布式计算模式中,定义了一组与实现无关的接口,引入了代理机制来分离客户端和服务器。目前,CORBA 规范已经成为面向对象分布式计算中的工业化标准。

3.2.13 层次消息总线体系结构风格

随着构件技术的成熟和构件互操作标准的出现,出现了层次消息总线(Hierarchy Message Bus,HMB)体系结构风格。在图形界面应用程序中,消息驱动的编程方法得到了广泛应用。在消息驱动的编程方法中,系统调用相关处理函数来响应不同消息,程序结构比较清晰。同时,计算机硬件总线的概念为软件体系结构的设计提供了很好的借鉴和启示。

在统一的接口和总线规范下，所开发的应用系统将具有良好的扩展性和适应性。

层次消息总线体系结构风格基于层次消息总线，支持构件的分布和并发，所有构件之间通过消息总线进行通信，如图 3-36 所示。其中，消息总线是整个系统的连接件，负责系统内消息的分配、传递、过滤以及处理结果的返回。所有构件均挂接在消息总线上，向消息总线登记自己感兴趣的消息类型。构件根据需要发出的消息，由消息总线负责把它们分配到系统中所有对此消息感兴趣的构件，消息是系统中所有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处理，在必要情况下可以通过消息总线向目标构件返回处理结果。

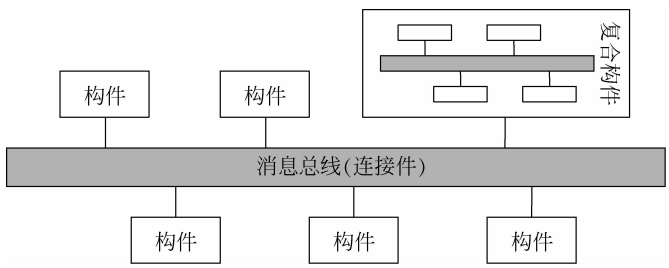

图 3-36　HMB 体系结构风格

复杂构件可以分解为粒度更小的子构件，通过局部消息总线进行连接，从而形成复合构件。如果子构件仍然比较复杂，则可以进一步分解。如此分解下去，系统将形成树状的拓扑结构。叶节点是系统的原子构件，不再包含子构件。原子构件的设计可以采用不同的软件体系结构风格，例如管道/过滤器风格、面向对象风格和数据共享风格等。此外，整个系统可以作为一个构件，通过更高层次的消息总线集成到更大的应用系统中。

层次消息总线系统和组成系统的成分通常是比较复杂的，很难从一个视角获得对它们的完整理解。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，在 Rumbaugh 等人提出的 OMT(Object Modeling Technology)方法中采用了对象模型、动态模型和功能模型来刻画系统的以上 3 个方面。

1. 构件模型

层次消息总线风格的构件模型包括了构件接口、静态结构和动态行为 3 个部分，如图 3-37 所示。

在图 3-37 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，如对内的标准消息接口，对外的消息接口以及对外的 API 接口，每个接口都定义了一组输入和输出消息，刻画了构件对外提供的服务以及要求的环境服务，体现了该构件与环境的交互。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化。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

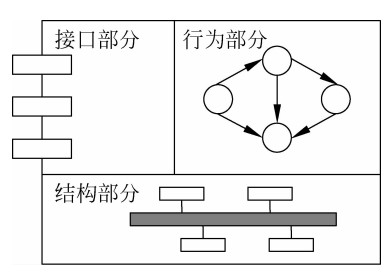

图 3-37　HMB 构件

统和各个层次的构件提供了统一的集成机制。

2. 构件接口

在体系结构设计层次上，构件通过接口定义了与外界的信息传递和承担的系统责任，构件接口代表了构件与环境的全部交互内容，也是唯一的交互途径。

层次消息总线风格的构件接口（此处指对内接口）是一种基于消息的互连接口，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。

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对此消息感兴趣的构件。按照响应方式的不同，消息可以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、时钟等。

3. 消息总线

层次消息总线风格的消息总线是系统的连接件，系统中的所有构件向消息总线登记自己感兴趣的消息类型，形成构件—消息响应登记表。在系统运行过程中，消息总线根据自己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，必要时，负责返回处理结果。此外，消息总线还可以对特定的消息类型进行过滤或阻塞。

这里以计算机考试系统为例对层次消息总线风格的体系结构进行说明。针对考试系统对可扩展、可维护性有较高要求以及本身业务的特点，可将系统分为有机结合的多个构件库及考试系统的通用服务，构件之间、构件与考试系统通用服务之间通过消息总线进行通信。

如图 3-38 所示，一个层次消息总线结构的考试系统至少由 6 个部分组成：考试系统的通用服务、试卷成分组织构件、消息总线、独立题型资源构件库、独立题型作答界面构件库、独立题型评分构件库。考试系统的通用服务包括考生身份验证、试卷分析统计等业务逻辑，这些业务逻辑和通常的 MIS 系统相类似，这里仅介绍消息总线。

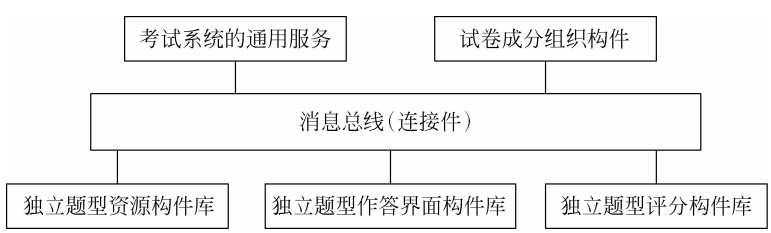

图 3-38　基于消息总线的考试系统结构示意图

图 3-39 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：对挂接在消息总线上的构件而言，消息是一种共享资源，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。类似于程序中的“间接地址调用”，使得构件之间保持了灵活的连接关系，便于系统的演化。当独立题型相关构件库新增了构件时，应该向消息总线发出登记消息，对构件实例表及构件—消息响应登记表做消息登记。

同样，当独立题型的资源构件和作答界面构件加入相应的构件库，但没有给出该题型的评分构件时，也应该向消息总线的消息过滤表中显示登记该题型的评分阻塞消息。通过显示的登记消息，使消息的响应者能更灵活地发挥自身的潜力。

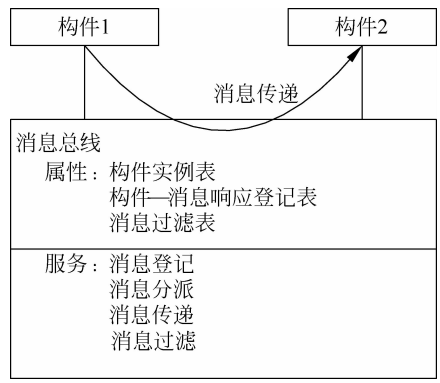

图 3-39　对象类符号表示的消息总线的结构

（2）消息分派和消息传递：构件—消息响应登记表记录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。消息总线根据构件—消息响应登记表把消息分配到相应的构件，并负责处理结果的返回。消息总线是一个逻辑上的整体，系统中的构件通过消息总线进行通信，因而实现了构件位置的透明性。系统在部署时可以根据当前各计算机的负载情况和效率方面的考虑，把构件在不同的物理位置上进行透明的迁移，而不影响系统的其他构件及相互间的通信。

（3）消息过滤：考试系统的各构件库是对各类独立题型考试方式进行抽象而得到的，是在对各类考试题型进行分析、抽象的基础上得到的所有独立考试题型的共同抽象接口。它是较高层次上的抽象，可以适用于所有类型的考试。但由于某些考试题型不同，现有的计算机软件技术难以实现全部的构件，如简答、论述题型的评分过程，目前的软件技术还无法提供一个令人满意的自动评分结果。因此，消息总线应该对请求不存在构件响应的消息进行过滤，过滤的原则主要是比对请求的消息是否存在于消息过滤表中。

3.3　新型软件体系结构风格

软件体系结构风格的 5 种分类也不能完全代表体系结构风格的组成，随着软件研发技术的不断进步，近些年接连总结出了几种新型的软件体系结构风格，如正交体系结构风格、REST 体系结构风格、面向服务（Service Oriented Architecture，SOA）体系结构风格、插件体系结构风格以及富互联网应用（Rich Internet Application，RIA）体系结构风格等。

3.3.1　正交体系结构风格

正交体系结构由组织层和线索的构件构成。层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）的，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。正交体系结构的主要特征如下：

（1）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。在软件进化过程中,系统需求会不断发生变化。在正交体系结构中,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,因此容易实现。

根据正交体系结构的概念,正交体系结构的核心模型由 5 种元素组成,包括构件、连接件、端口、角色、线索。在这里,只简要地介绍构件和线索的定义。

(1) 构件:构件是一个计算单元或数据存储。也就是说,构件是计算与状态存在的场所。在体系结构中,一个构件可能小到只有一个过程或大到整个应用程序,它可以有自己的数据域或执行空间,也可以和其他构件共享这些空间。

(2) 线索: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(通过连接件来关联)的,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,可将正交体系结构的核心模型表示为图 3-40。

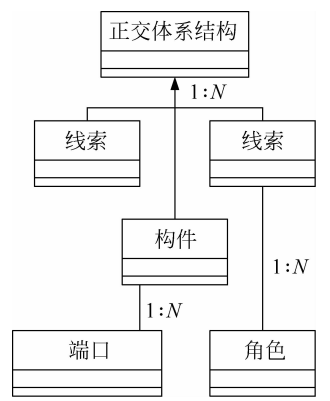

图 3-40 正交体系结构的核心模型

以下给出汽修服务管理系统的设计方案。考虑到用户需求可能会经常发生变化,在设计时采用了正交体系结构,大部分线索是独立的,不同线索之间不存在相互调用关系。维修收银功能需要涉及维修时的派工、外出服务和维修用料,因此,适当放宽了要求,采用了非完全正交体系结构,允许线索之间有适当的调用,不同线索之间可以共享构件。由于非完全正交结构的范围不大,因此,对整个系统框架的影响可以忽略。汽修服务管理系统的体系结构如图 3-41 所示。其中,系统、维修登记、派工、增加和数据接口形成了一条完整的线索。

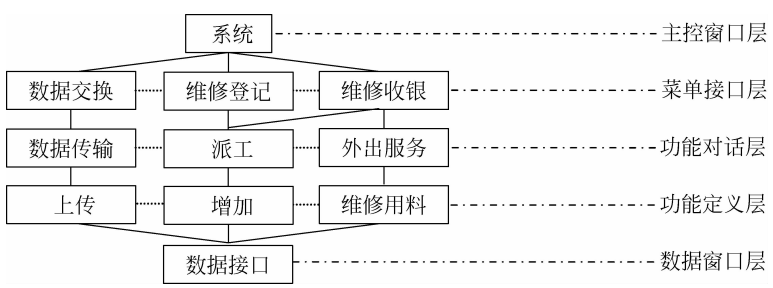

图 3-41 汽修服务管理系统的框架结构

3.3.2 富互联网应用体系结构风格

随着应用软件业务逻辑越来越复杂,原先的主流显示技术愈发不能满足需求。为提升用户体验,出现了一种新类型的 Internet 应用程序,就是 RIA(Rich Internet Application),也称为富互联网应用体系结构风格或富客户体系结构风格。

RIA 将桌面型计算机软件应用的最佳用户界面功能性与 Web 应用程序的普遍采纳和

低成本部署以及互动多媒体通信的长处集于一体，可以提供更直观、响应更快和更有效的用户体验，简化并改进了 Web 应用程序的用户交互。它不仅具备桌面型系统的长处，包括在确认和格式编排方面提供互动用户界面、在无刷新页面之下提供快捷的界面响应时间、提供通用的用户界面特性，如拖放式以及在线和离线操作能力，而且保留了 Web 的优点，如立即部署、跨越平台可用性、采用逐步下载来检索内容和数据、拥有杂志式布局的网页以及充分利用被广泛采纳的互联网标准等，并且支持双向互动声音和图像。图 3-42 描述了 RIA 应用程序的层次模型。

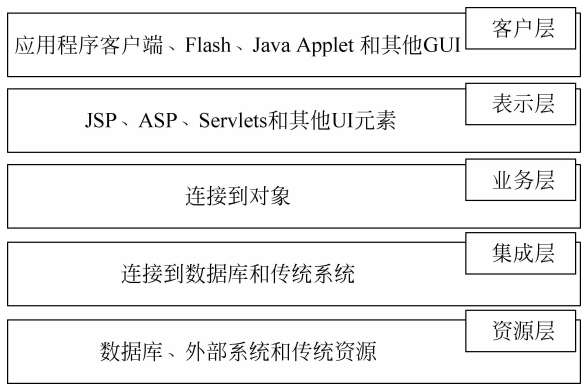

图 3-42　RIA 应用程序模型图

RIA 风格的优点如下。

(1) 强交互性：RIA 支持丰富的 UI 组件。

(2) 直接管理：局部的数据更新，通过客户端计算可直接实现对用户请求的响应。

(3) 多步骤处理：所有内容在一个界面中添加转换效果，使应用程序的状态在各步骤中轻松移动。

(4) 文本独立性：RIA 集成 XML 特性，简化异质系统的通信，方便数据的存取。

(5) 平台无关性：应用层次对所有的 RIA 客户端都是一致的。

RIA 风格的缺点如下。

(1) Sandbox(沙箱)：因为 RIA 必须运行在 Sandbox 中，所以它们对系统资源的访问必须要受到严格控制，否则可能会出现一些问题。

(2) 需要脚本的支持：RIA 总是需要诸如 JavaScript 一类的脚本，用户不能关闭浏览器的动态脚本支持。

(3) 客户端处理的速度：为了实现跨平台的效果，一些 RIA 使用 JavaScript 一类的客户端未编译脚本，可能会对性能造成比较大的影响。但是如果使用经过编译的 Java Applet、Flash、Flex 或者 Sliverlight 等语言，则性能不会出现太大问题。

(4) 脚本下载时间：虽然 RIA 无须安装，但客户端引擎的脚本总需下载。

(5) 可搜索度降低：目前的搜索引擎还不能很好地支持这样的内容。

(6) 不可部署性：目前，除了 Adobe AIR 技术外，其他 RIA 应用都不具备像传统桌面应用那样的可部署性。

Ext 框架是一个典型的 RIA 应用于客户端方面的富客户端应用。ExtJS 由一系列的类库组成，一旦页面成功加载了 ExtJS 库，就可以在页面中通过 JavaScript 调用 ExtJS 的类及控件来实现需要的功能。ExtJS 的类库由以下几部分组成。

(1) 底层 API(Core)：底层 API 中提供了对 DOM 操作、查询的封装、事件处理、DOM 查询器等基础的功能。其他控件都建立在这些底层 API 的基础上，底层 API 位于源代码目录的 Core 子目录中，包括 DomHelper.js、Element.js 等文件。

(2) 控件(Widgets)：控件是指可以直接在页面中创建的可视化组件，例如面板、选项板、表格、树、窗口、菜单、工具栏、按钮等，在我们的例子应用程序中可以直接应用这些控件来实现友好、交互性强的应用程序的 UI。控件位于源代码目录的 Widgets 子目录中。

(3) 实用工具 Utils：Ext 提供了很多实用工具，可以方便地实现数据内容格式化、JSON 数据解码或反解码、对 Date 和 Array 发送 Ajax 请求、Cookie 管理、CSS 管理扩展功能。

Ext4 对框架进行了非常大的重构，其中最重要的就是形成了一个结构及层次分明的组件体系，由这些组件形成了 Ext 的控件，Ext 组件是由 Component 类定义，每一种组件都有一个指定的 xtype 属性值，通过该值可以得到一个组件的类型或者是定义一个指定类型的组件。ExtJS 的运行效果如图 3-43 所示。

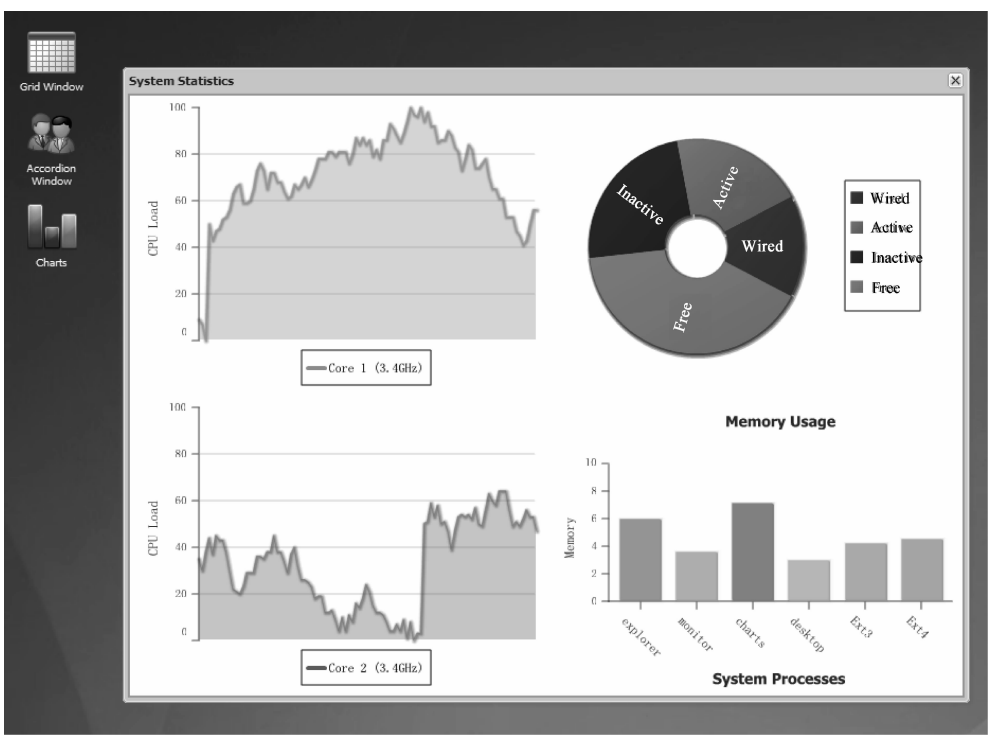

图 3-43 ExtJS 运行效果图

3.3.3 表述性状态转移体系结构风格

表述性状态转移(Representational State Transfer，REST)是 Roy Fielding 在他的博士论文中发明的一个新名词，是对 Web 体系结构设计原则的一种描述。REST 的目的是决定

图 3-44　REST 体系结构风格交互模式图

如何使一个良好定义的 Web 程序向前推进：一个程序可以通过选择一个带有超链接的 Web 页面上的链接，使得另一个 Web 页面（代表程序的下一个状态）返回给用户，使程序进一步运行。如图 3-44 所示，客户请求用户信息的服务（使用逻辑 URI），返回结果页面中包含该用户的表述。得到返回结果后，客户选择一个链接来决定下一步动作，这样可以做到客户维护自己的程序状态。

因为 REST 是对当今 Web 体系结构设计原则的一种抽象和描述，所以 REST 的设计准则是对当今 Web 中已成功应用的要素的总结。从这个意义上讲，Web 是 REST 风格的一个实例。REST 描述了如何设计和开发分布式系统，对 REST 的应用只能是理解它，把它的原则应用到 Web 应用系统的设计中。从正在应用的 Web 上讲，Web 上的每个资源通过一个 URI 来标识，可以通过简洁通用的接口（如 HTTP 的 GET、POST、PUT 和 DELETE）来操作 Web 上的资源。资源使用者与资源之间有代理服务器、缓存服务器来解决安全及性能等问题。REST 系统中的组件必须是自描述的，这样，客户可根据这些自描述信息来维护自己的程序状态。

REST 提出了以下设计准则：

（1）网络上的所有事物都被抽象为资源；

（2）每个资源对应一个唯一的资源标识符；

（3）通过通用的连接器接口（Generic Connector Interface）对资源进行操作；

（4）对资源的各种操作不会改变资源标识符；

（5）所有的操作都是无状态的；

（6）REST 强调中间媒介的作用（包括代理服务器、缓存服务器和网关）。

REST 体系结构风格的优点如下：

（1）统一接口，简化了对资源的操作；

（2）REST 的无状态性提高了系统的伸缩性（无状态性使得服务器端可以很容易地释放资源，因为服务器端不必在多个 Request 中保存状态）和可靠性（无状态性减少了服务器从局部错误中恢复的任务量）；

（3）基于缓存机制，提高了系统的处理性能和负载量；

REST 体系结构风格的缺点如下：

（1）缺少有效的服务发现能力；

（2）后台复杂逻辑封装和中间代理的引入会影响用户可察觉的性能；

（3）由于 REST 无状态性，增加的每次请求传送状态数据的开销，影响了交互效率。

REST fulWeb 服务是符合 REST 风格的轻量级 Web 服务结构，它以完成业务为目标，将一切与业务相关的事物抽象为资源，并为每个资源赋予一个 URI 标识。用户在提交请求时，将作用域信息置于 URI 中，并且使用不同的 HTTP 方法提交请求，即可对该 URI 代表的资源执行相关操作，其中常见的 HTTP 方法为 POST、GET、PUT 和 DELETE，对应资源的创建、读取、更新和删除操作，简称 CRUD 操作。而作用域信息则通常表现为 URI 中包含的参数，如 http：/ /xxx? title＝book，其中，title＝ book 即作用域信息，代表了指定资源中更明确的作用对象。由此可见，URI 即资源的统一访问接口，REST fulWeb 服务只要对外界暴露 URI 即对外发布服务，REST fulWeb 服务的请求和响应如图 3-45 所示。

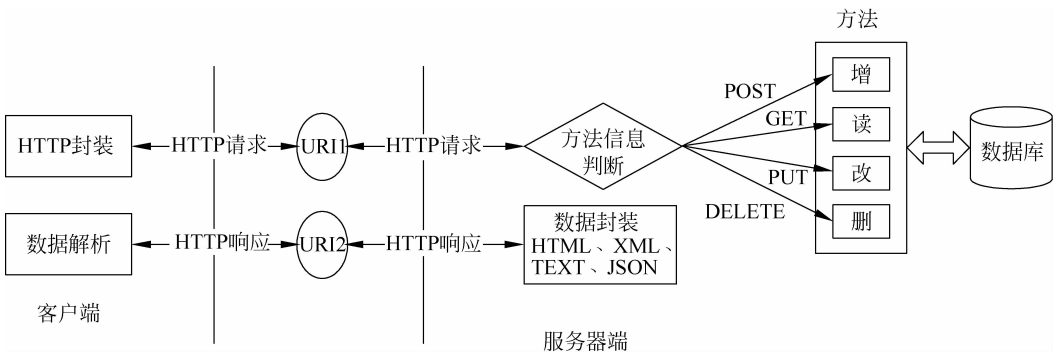

图 3-45 REST fulWeb 服务的请求和响应过程

客户端发送请求时，将请求数据置于 HTTP 文档主体中，并使用 HTTP 请求方法向 URI1 提交请求。服务器端接收到请求后，根据请求的方法类型，调用对应的方法执行请求。当服务器端处理完请求后，返回 HTTP 响应代码和报头，并将得到的相关数据置于 HTTP 文档主体中，转向 URI2，最后返回给客户端。

REST fulWeb 服务对响应数据的格式并没有特殊要求，可以使用 HTML、XML、TEXT 或 JSON 等多种格式。客户端还可通过 WADL 对 REST fulWeb 服务进行有目的的访问，WADL 定义了客户端可以发送的各种 HTTP 请求、可访问的 URI、可执行的方法及相关参数以及返回的数据格式等。

1. REST fulWeb 服务图书管理系统

本部分以基于 REST fulWeb 服务图书管理系统的设计为例，对使用 REST fulWeb 构建应用系统的基本思路进行说明。REST fulWeb 服务是面向资源的服务，因此，使用 REST fulWeb 服务构建图书管理系统，重点是分析图书管理业务，将业务涉及的事物抽象成资源，根据业务为每个资源设计 URI 和资源表示。资源确定后，才可以设计图书管理系统的总体结构，进而根据总体结构，使用开发工具实现整个系统。

图书管理是图书馆工作中的一个重要环节，业务复杂多样。为了清晰地描述开发过程，以图书管理中图书编目的入库、修改、报废、借阅、归还和查询 6 个典型业务作为系统的核心业务。这 6 个业务均围绕图书展开，因此，图书可被抽象成资源，其中入库对应资源的创建、查询对应资源的读取、修改/借阅/归还对应资源的修改、报废对应资源的删除。查询还可细分为三类：查询所有图书、查询单本图书、按关键词查询。进一步分析，图书入库、查询所有图书和按关键词查询均是针对整个图书资源而言，因此，图书资源还需要划分为所有图书资源和单本图书资源，其 URI 设计如下：

(1) 所有图书资源。

```
URI:http://hostnam e/resource/books
```

(2) 单本图书资源。

```
URI:http://hostnam e/resource/books/{bookid}
```

一般来说，通过以上两种形式的 URI 发送 HTTP 请求可以实现所有的图书管理业务，

但在实际业务处理时，往往存在同一个 URI 和相同的 HTTP 方法，需要实现不同的功能，或者返回资源的不同表示的情况，这时候通常使用作用域信息来区分。相应的 URI 资源下 HTTP 方法与作用域信息结合，触发不同的系统功能，其对应关系如表 3-1 所示。

表 3-1　系统功能对应关系

资源 URI	HTTP 方法	作用域信息	系 统 功 能
http://hostname/resource/books	POST	无	入库
	GET	title =	查询所有图书
		title ='{ 关键词}'	按关键词查询
http://hostname. resource. books/{bookid}	PUT	Status = '更新'	修改图书信息
		Status ='可借'	归还图书
		Status ='借出'	借阅图书
	DELETE	无	报废
	GET	无	查阅图书的表示
		type = 0	借阅图书的表示
		type = 1	归还图书的表示
		type = 2	修改图书的表示

对于资源表示的格式，图书管理系统的本地用户使用较友好的 HTML 格式，而对于第三方系统则采用计算机可读的 JSON 格式。

系统的总体架构如图 3-46 所示。

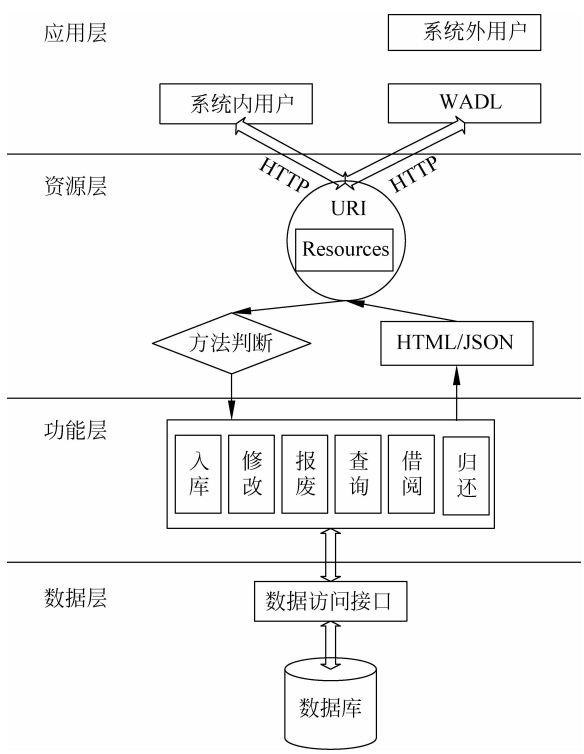

图 3-46　基于 REST fulWeb 服务的图书管理系统总体架构

系统分为数据层、功能层、资源层和应用层。

(1) 数据层：该层由数据库和数据访问接口组成。数据访问接口定义了对数据库记录的查询、增加、修改和删除，任何涉及这 4 种操作的方法均要实现该接口才能对数据库记录进行操作，起到规范访问数据库的作用。

(2) 功能层：该层接受资源层的方法调度，执行对应的功能方法，这些方法与数据层进行通信，将得到的数据返回给资源层。

(3) 资源层：该层由资源和 URI 组成，通过对外显示图书资源的 URI 来发布服务。资源层接受用户请求，根据 HTTP 请求的方法类型调用功能层的方法，并视用户类型的不同，对返回数据以 JSON 或 HTML 格式封装后返回给用户。资源层为整个系统的中心，既是用户提交请求和接收数据的接口，也是系统接受和响应请求的接口，体现了以资源为中心的 REST fulWeb 服务架构的特点。

(4) 应用层：该层管理用户提交的请求。系统用户分为系统内用户和系统外用户。两种类型，其中，系统内用户是在图书管理系统数据库中注册的用户，这类用户对系统的操作均可按照系统页面的提示来完成；系统外用户是为获取本系统图书信息的第三方图书管理系统，由于这类用户并不是自然人，无法根据页面提示来访问本系统，因此系统对外提供了功能描述的 WADL，系统外用户根据 WADL 提供的访问规则即可正常获取信息。对于系统外用户，一般只开放查询的功能。

2. Restlet 项目

此外，Restlet 项目也是 REST 体系结构的一个实例。当复杂核心化模式日趋强大，面向对象设计范例已经不总是 Web 开发中的最佳选择时，Restlet 这个开源项目为那些要采用 REST 结构体系来构建应用程序的 Java 开发者提供了一个具体的解决方案。它的非常简单易用的功能和 RESTfully 的 Web 框架使其成为了 Web 2.0 开发中的又一利器。

Restlet 项目为“建立 REST 概念与 Java 类之间的映射”提供了一个轻量级而全面的框架。它可用于实现任何类型的 REST 式系统，而不仅仅是 REST 式 Web 服务。Restlet 项目的主要目标是在提供同等功能的同时，尽量遵守 Roy Fielding 博士论文中所阐述的 REST 的目标。此外，它还提出一个既适于客户端应用又适于服务端应用的、统一的 Web 视图。

Restlet 在术语上参照了 Roy Fielding 博士论文在讲解 REST 时采用的术语，并且 Restlet 增加了一些专门的类(如 Application、Filter、Finder、Router 和 Route)，用于简化 Restlets 的彼此结合，以及把收到的请求映射为处理它们的资源。

下面以基于 JAX-RS 的 REST 服务为例，说明 Restlet 项目的创建过程。JAX-RS (JSR-311)是一种 Java API，可使 Java Restful 服务的开发变得迅速而轻松。这个 API 提供了一种基于注解的模型来描述分布式资源。注解被用来提供资源的位置、资源的表示和可移植的(Pluggable)数据绑定架构。

(1) 新建 Java Web Project RestService 工程如图 3-47 所示。

(2) 将%RESTLET_HOME%\lib 复制到\RestService\WebContent\WEB-INF\lib 下，并加入工程引用。为了测试方便，可以将全部的 lib 包加进去，且 org. restlet. jar 必须加入其中。如果是用 JAX-RS 发布 rest，还需要 javax. ws. rs. jar、javax. xml. bind. jar、org.

图 3-47　新建的 Java Web Project RestService 工程

json.jar、org.restlet.ext.jaxrs.jar、org.restlet.ext.json.jar、org.restlet.ext.servlet.jar 几个包。

(3) 接下来创建 Student 实体类，用于返回数据。Student 使用 JAXB 绑定技术，自动解析为 XML 返回给客户端或浏览器。JAXB 是一套自动映射 XML 和 Java 实例的开发接口和工具，可以使 XML 更加方便地编译一个 XML SCHEMA 到一个或若干个 Java CLASS。

```
@XmlRootElement(name = "Student")
public class Student {
private int id;
private String name;
private int sex;
private int clsId;
private int age;
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
```

```
        this.name = name;
    }
    public int getSex() {
        return sex;
    }
    public void setSex(int sex) {
        this.sex = sex;
    }
    public int getClsId() {
        return clsId;
    }
    public void setClsId(int clsId) {
        this.clsId = clsId;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    }
```

(4) Restlet 架构主要是 Application 和 Resource 的概念,在程序中可以定义多个 Resource,一个 Application 可以管理多个 Resource。

① 创建应用类:StudentApplication 继承了抽象类 javax. ws. rs. core. Application,并重载了 getClasses()方法。代码如下:

```
Set<Class<?>> rrcs = new HashSet<Class<?>>();
rrcs.add(StudentResource.class);
```

② 绑定 StudentResource:有多个资源可以在这里绑定,例如 Course 等,可以相应地定义为 CourseResource 及 Course,然后加入 rrcs. add(CourseResource. class)。

③ 创建资源类:StudentResource 管理 Student 实体类,代码如下。

```
@Path("student")
public class StudentResource {
@GET
@Path("{id}/xml")
@Produces("application/xml")
public Student getStudentXml(@PathParam("id") int id) {
return ResourceHelper.getDefaultStudent();
}
}
```

其中,@Path("student")执行了 URI 路径,student 路径进来的都会调用 StudentResource 来处理;@GET 说明了 HTTP 的方法是 GET 方法;对于@Path("{id}/xml"),每个方法前都有对应的 Path,用来声明对应的 URI 路径;@Produces("application/xml")用于指定返回的数据格式为 xml;@PathParam("id") int id 表示接受传递进来的 id 值,并且 id 与{id}定义的占位符要一致。

(5) 定义了相应的 Resource 和 Application 后，还要创建运行环境。Restlet 架构为了更好地支持 JAX-RS 规范，定义了 JaxRsApplication 类来初始化基于 JAX-RS 的 Web Service 运行环境。

① 创建运行类：RestJaxRsApplication 继承了类 org. restlet. ext. jaxrs. JaxRsApplication，构造方法的代码如下。

```
public RestJaxRsApplication(Context context) {
super(context);
this.add(new StudentApplication());
}
```

将 StudentApplication 加入运行环境后，如果有多个 Application 可以在此绑定。

② 发布和部署 restlet 服务：

首先将 Restlet 服务部署到 Tomcat 容器中，在 web. xml 中加入以下代码：

```
<context - param>
<param - name>org.restlet.application</param - name>
<param - value>ws.app.RestJaxRsApplication</param - value>
</context - param>
<servlet>
<servlet - name>RestletServlet</servlet - name>
<servlet - class>org.restlet.ext.servlet.ServerServlet</servlet - class>
</servlet>
<servlet - mapping>
<servlet - name>RestletServlet</servlet - name>
<url - pattern>/*</url - pattern>
</servlet - mapping>
```

如果启动 Tomcat 没报错，说明配置正确。

然后将 Restlet 服务当做单独的 Java 程序部署创建类 RestJaxRsServer，代码如下：

```
public static void main(String[] args) throws Exception {
Component component = new Component();
component.getServers().add(Protocol.HTTP, 8085);
component.getDefaultHost().attach(new RestJaxRsApplication(null));
component.start();
}
```

该类中创建了一个新的 HTTP Server，添加监听端口 8085。将 RestJaxRsApplication 加入到 HTTP Server 中，完成系统启动。

3.3.4　插件体系结构风格

插件(Plug-in)技术是现代软件设计思想的体现，它可以将需要开发的目标软件分为若干功能构件，各构件只要遵循标准接口即可。整个软件集成时，只需要将构件进行组装，而不是集成源代码或链接库进行编译与链接；需要新的功能构件时，也只需要按规定独立开发，完成后组装到原软件平台中即可使用，实现了一种二进制的软件集成方法。

插件体系结构由主体部分和扩展部分组成，主体部分完成系统的基本功能，扩展部分（插件库）完成系统的扩展功能。插件风格中存在两类接口：主体扩展接口和插件接口。主体扩展接口由主体实现，插件只是调用和使用；插件接口由插件实现，主体只是调用和使用，插件体系结构风格图如图 3-48 所示。

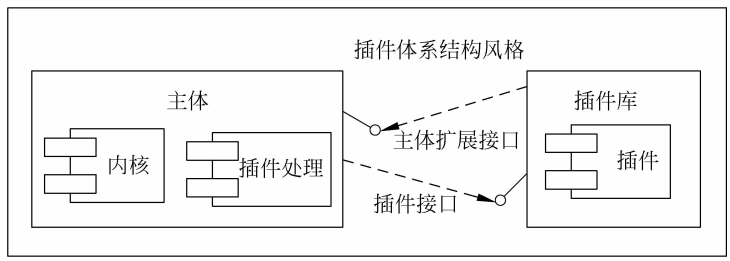

图 3-48 插件体系结构风格图

主体插件处理功能包括插件注册、管理、调用以及主体扩展接口的功能实现。插件注册为按照某种机制首先在系统中搜索已安装插件，之后将搜索到的插件注册到主体上，并在主体上生成相应的调用机制，包括菜单选项、工具栏、内部调用等；插件管理完成插件与主体的协调，为各插件在主体上生成管理信息以及进行插件的状态跟踪；插件调用用来调用各插件所实现的功能；主体插件处理模块的另一部分是主体扩展接口的具体实现。插件体系结构风格的优点如下：

（1）实现真正意义上的软件部件的“即插即用”；

（2）在二进制级上集成软件，避免重新编译内核功能，方便功能扩展和升级；

（3）能够很好地实现软件模块的分工和分期开发；

插件体系结构风格的缺点如下：

（1）过多的插件给维护带来了困难，特别是没有一个优秀的接口规范会使系统混乱；

（2）基于插件的软件开发，完全改变了已有的开发模式和软件销售模式，给软件团队增加了管理方面的一些成本。

目前，有很多这种插件风格的软件系统实例，如 Adobe 公司的图形处理软件 Photoshop。为了提高图形的处理功能，Photoshop 提供了标准插件开发接口，这样，第三方软件开发商就可以按标准插件接口开发独具特色的图形功能扩展，开发的插件安装后，系统即可使用，而不影响主程序和其他插件；除此之外，使用插件技术的软件还有 IE、Netscape 和 Macromedia 公司的系列软件，以及 Microsoft 的 Visual Studio 开发工具和 Office 办公软件等。目前也有很多支持插件风格的框架技术和标准，例如 OSGI 标准等。

开发平台 Eclipse 是插件风格的经典。Eclipse 平台是 IBM 向开发源码社区捐赠的开发框架，它是巨大投入所带来的成果：一个成熟的、精心设计的以及可扩展的体系结构。Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源码平台，这个平台允许任何人构建与环境和其他工具无缝集成的工具。

工具与 Eclipse 无缝集成的关键是插件。除了小型的运行时内核之外，Eclipse 中的所有东西都是插件。从这个角度来讲，所有功能部件都是以同等的方式创建的。

但是，某些插件比其他插件更重要。Workbench 和 Workspace 是 Eclipse 平台的两个

必备的插件，它们提供了大多数插件使用的扩展点，插件需要扩展点才可以插入，这样它才能运行，如图 3-49 所示。

Workbench 组件包含了一些扩展点，例如，允许插件扩展 Eclipse 用户界面，使这些用户界面带有菜单选择和工具栏按钮；请求不同类型事件的通知；创建新视图。Workspace 组件包含了可以让用户与资源（包括项目和文件）交互的扩展点。

当然，其他插件可以扩展的 Eclipse 组件并非只有 Workbench 和 Workspace。此外，还有一个 Debug 组件可以让用户的插件启动程序、与正在运行的程序交互，以及处理错误，这是构建调试器所必需的。虽然 Debug 组件对于某些类型的应用程序是必需的，但大多数应用程序并不需要它。

还有一个 Team 组件允许 Eclipse 资源与版本控制系统（VCS）交互，但除非用户正在构建 VCS 的 Eclipse 客户机，否则 Team 组件就像 Debug 组件一样，不会扩展或增强它的功能。

最后，还有一个 Help 组件可以让用户提供应用程序的联机文档和与上下文敏感的帮助。没有人会否认帮助文档是专业应用程序必备的部分，但它并不是插件功能的必要部分。

下面以“Hello，World”插件的创建过程为例，来说明、理解插件体系结构风格。

1. 创建插件

最简单的方法是使用 Plug-in Development Environment（PDE），PDE 和 Java Development Tooling（JDT）IDE 是 Eclipse 的标准扩展。PDE 提供了一些向导来帮助用户创建插件，包括将在这里研究的“Hello，World”示例。选择“Hello，World”代码生成向导如图 3-50 所示。

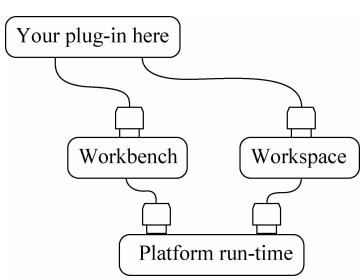

图 3-49　Eclipse Workbench 和 Workspace 必备的插件支持

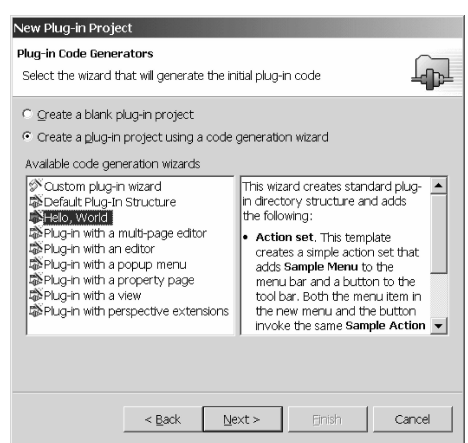

图 3-50　选择“Hello，World”代码生成向导

其中要求一些附加信息，包括插件名称、版本号、提供者名称和类名。这些是关于插件的重要信息，可以接受向导提供的默认值。此外，还接受包名、类名和消息文本的默认值。向导完成后，在工作区中会有一个新的项目，名称为 com. example. hello，如图 3-51 所示。

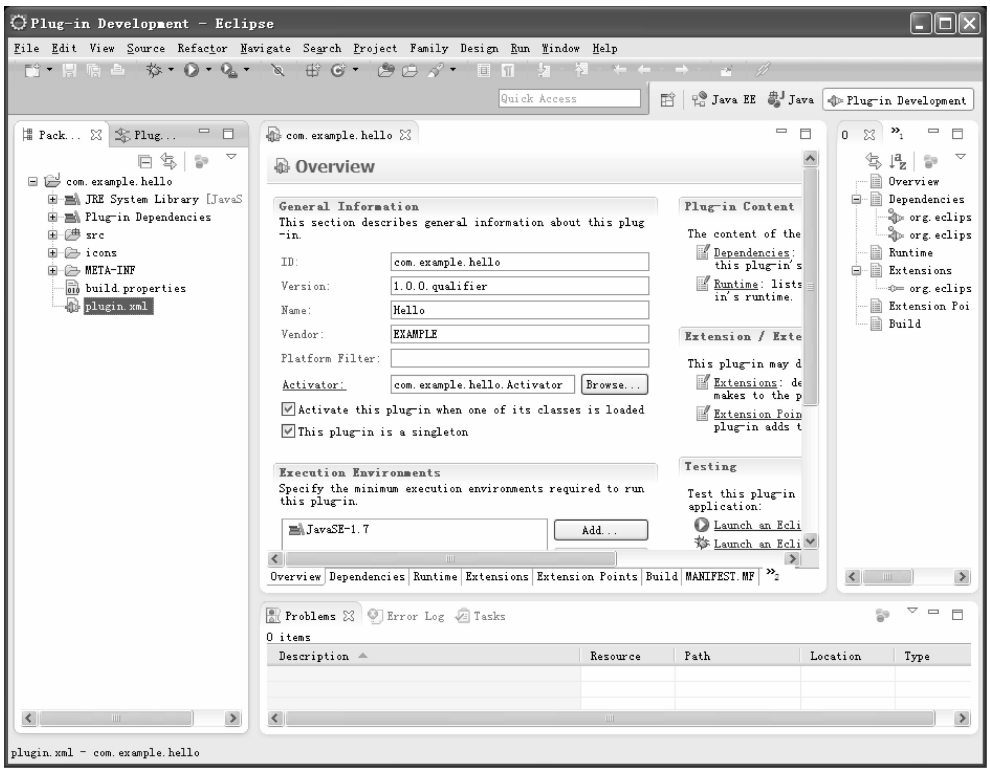

图 3-51 PDE 透视图 Welcome to Hello Plug-in

在 Package Explorer 中，工作台的左边是向导创建的一些东西的概述，包括项目类路径中的许多.jar 文件(包括插件和 Java 运行时所需的 Eclipse 类)、一个图标文件夹(包含了工具栏按钮的图形)，以及 build.properties 文件(包含自动构建脚本所使用的变量)。

2. 插件清单文件

在 src 文件夹中包含了插件和 plugin.xml 文件的源代码，plug-in.xml 是插件的清单文件。插件清单文件 plugin.xml 包含了 Eclipse 将插件集成到框架所使用的描述信息。通过编辑器底部的选项卡可以选择关于插件的不同信息集合，Welcome 选项卡显示了消息“Welcome to Hello Plug-in”，并且简要讨论了所使用的模板和关于使用 Eclipse 实现插件的提示。

首先是关于插件的常规信息，包括它的名称、版本号，以及实现它的类文件的名称和.jar文件名。

清单 1：插件清单文件——常规信息。

```
<?xmlversion = "1.0" encoding = "UTF - 8"?>
< plugin
    id = "com.example.hello"
    name = "Hello Plug - in"
    version = "1.0.0"
    provider - name = "EXAMPLE"
```

```
class = "com.example.hello.HelloPlugin">
<runtime>
    <library name = "hello.jar"/>
</runtime>
```

接着，列出了所创建插件所需的插件。

清单 2：插件清单文件，即必需的插件。

```
<requires>
    <import plugin = "org.eclipse.core.resources"/>
    <import plugin = "org.eclipse.ui"/>
</requires>
```

此处列出的第一个插件 org.eclipse.core.resources 是工作区插件，但实际上在创建插件时并不需要它。第二个插件 org.eclipse.ui 是工作台。这里需要工作台插件，因为将扩展它的两个扩展点，正如后面的 extension 标记所指出的。

第一个 extension 标记拥有点属性 org.eclipse.ui.actionSets。操作集合是插件添加到工作台用户界面的一组基值，即菜单、菜单项和工具栏，用户可以更方便地管理它们。例如，“Hello，World”插件的菜单和工具栏项将出现在 Resource 透视图（图 3-52）中。

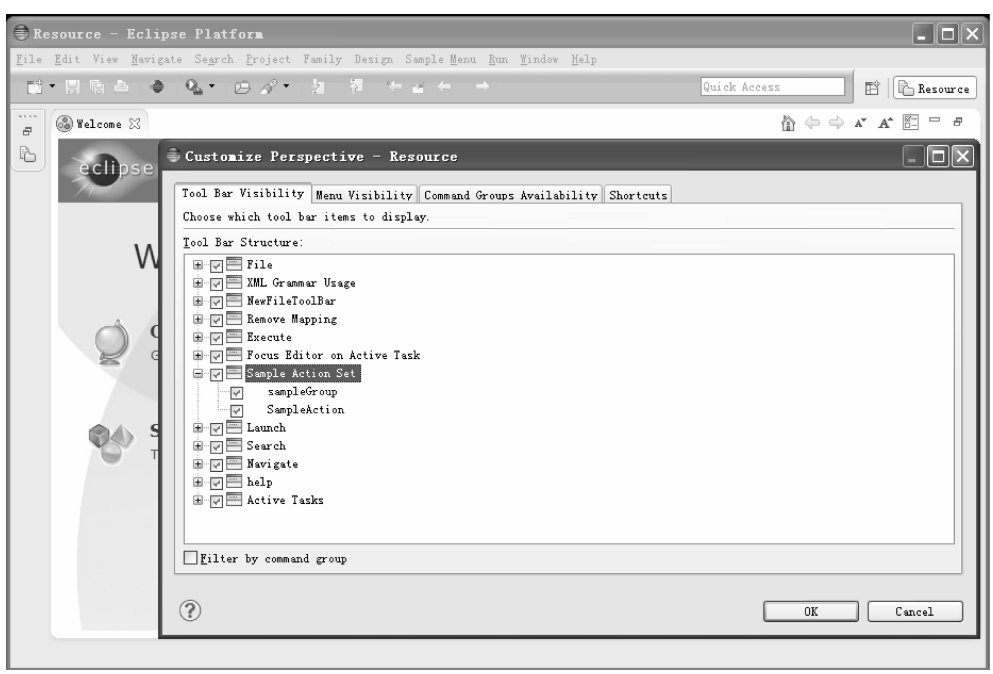

图 3-52　定制 Resource 透视图

操作集合包含了两个标记：menu 标记（描述菜单项应该出现在工作台菜单的什么位置，以及如何出现）和 action 标记（描述它应该做什么），尤其是 action 标记标识了执行操作的类。

清单 3：操作集合。

```
<extension
```

```
        point = "org.eclipse.ui.actionSets">
      <actionSet
          label = "Sample Action Set"
          visible = "true"
          id = "com.example.hello.actionSet">
      <menu
              label = "Sample &Menu"
              id = "sampleMenu">
          <separator
                 name = "sampleGroup">
          </separator>
      </menu>
      <action
              label = "&Sample Action"
              icon = "icons/sample.gif"
              class = "com.example.hello.actions.SampleAction"
              tooltip = "Hello, Eclipse world"
              menubarPath = "sampleMenu/sampleGroup"
              toolbarPath = "sampleGroup"
              id = "com.example.hello.actions.SampleAction">
          </action>
          </actionSet>
  </extension>
```

许多菜单和操作属性的目的相当明显，例如，提供工具、提示文本和标识工具栏项的图形。但用户还要注意 action 标记中的 menubarPath，这个属性标识了 menu 标记中定义的哪个菜单项调用 action 标记中定义的操作。

由于选择了将插件添加到 Resource 透视图，于是生成了第二个 extension 标记。这个标记会导致当 Eclipse 第一次启动并装入“Hello，World”插件时，将插件添加到 Resource 透视图。

清单 4：extension 标记。

```
  <extension
      point = "org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
        targetID = "org.eclipse.ui.resourcePerspective">
      <actionSet
        id = "com.example.hello.actionSet">
      </actionSet>
    </perspectiveExtension>
  </extension>
</plugin>
```

代码生成向导生成了两个 Java 源文件，其位于 PDE Package Explorer 中的 src 文件夹中。第一个文件 HelloPlugin.java 是插件类，它继承了 AbstractUIPlugin 抽象类。HelloPlugin 负责管理插件的生命周期，在更为扩展的应用程序中，它负责维护对话框设置和用户首选项等内容。

清单 5：HelloPlugin。

```
packagecom. example. hello. actions;
import org. eclipse. ui. plugin. * ;
import org. eclipse. core. runtime. * ;
import org. eclipse. core. resources. * ;
import java. util. * ;
public class HelloPlugin extends AbstractUIPlugin {
    //The shared instance.
    private static HelloPlugin plugin;
    //Resource bundle.
    private ResourceBundle resourceBundle;
    public HelloPlugin(IPluginDescriptor descriptor) {
        super(descriptor);
        plugin = this;
        try {
            resourceBundle = ResourceBundle. getBundle(
                    "com. example. hello. HelloPluginResources");
        } catch (MissingResourceException x) {
            resourceBundle = null;
        }
    }
    public static HelloPlugin getDefault() {
        return plugin;
    }
    public static IWorkspace getWorkspace() {
        return ResourcesPlugin. getWorkspace();
    }
    public static String getResourceString(String key) {
        ResourceBundle bundle = HelloPlugin. getDefault(). getResourceBundle();
        try {
            return bundle. getString(key);
        } catch (MissingResourceException e) {
            return key;
        }
    }
    public ResourceBundle getResourceBundle() {
        return resourceBundle;
    }
}
```

第二个源文件 SampleAction. java 包含的类将执行在清单文件的操作集合中指定的操作。SampleAction 实现了 IWorkbenchWindowActionDelegate 接口，它允许 Eclipse 使用插件的代理，这样在一般情况下，Eclipse 就无须装入插件。IWorkbenchWindowActionDelegate 接口方法使插件可以与代理进行交互。

清单 6：IWorkbenchWindowActionDelegate 接口方法。

```
package com. example. hello. actions;
import org. eclipse. jface. action. IAction;
import org. eclipse. jface. viewers. ISelection;
import org. eclipse. ui. IWorkbenchWindow;
import org. eclipse. ui. IWorkbenchWindowActionDelegate;
```

```
import org.eclipse.jface.dialogs.MessageDialog;
public class SampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;
    public SampleAction() {
    }
    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "Hello Plug-in",
            "Hello, Eclipse world");
    }
    public void selectionChanged(IAction action, ISelection selection) {
    }
    public void dispose() {
    }

    public void init(IWorkbenchWindow window) {
        this.window = window;
    }
}
```

3.3.5 面向服务体系结构风格

面向服务的计算(Service Oriented Computing,SOC)是基于 Internet 的新一代分布式计算平台,它把在 Internet 上的大量资源转化为服务,用来作为软件系统应用开发的基本元素。服务是一种粗粒度、可发现、松耦合、自治的分布式组件。服务的这些独特特征,使面向服务的体系结构(Service Oriented Architecture,SOA)明显区别于传统软件架构。SOA 是通过一定的原则来组合一系列可以相互交互的服务进行软件应用开发的一种架构解决方案。SOA 本身不是一种具体的技术,而是一个组件模型,一种架构风格。随着越来越多基于服务的技术标准的制定,SOA 逐渐走向成熟,转变成了一种生产力,但是仍然面临着服务管理、服务事务处理、服务之间的协同机制和安全等一系列问题的挑战。自 Gartner 公司在 1996 年首次提出 SOA 的概念以来,SOA 依然没有一个统一的、被业内同行普遍认可的定义,不同企业和个人对 SOA 有着不同的理解,大致上可分为以下狭义和广义两大定义:

(1) 狭义上认为 SOA 主要是一种架构风格,是以基于服务、IT 与业务对齐作为原则的 IT 架构方式;

(2) 广义上把 SOA 看作是包括编程模型、运行环境和方法论等在内的一系列企业应用解决方案和企业环境,而不仅仅是一种架构风格。SOA 覆盖了软件开发的整个生命周期,包括软件建模、开发、整合、部署、运行等。

3.3.6 异构体系结构风格

传统的软件开发过程可以划分为从概念到实现的若干个阶段,包括问题定义、需求分析、软件设计、软件实现及软件测试等,软件体系结构的建立位于需求分析与软件设计之间。

软件体系结构设计中的一个核心问题是能否使用重复的体系结构模式，但对如何选择体系结构风格却没有固定的模式。

在设计软件系统时，从不同角度来观察和思考问题，会对体系结构风格的选择产生影响。每一种体系结构风格都有不同的特点，适用于不同的应用问题，因此，体系结构风格的选择是多样化的和复杂的。在实际应用中，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和比较，而且各种软件体系结构并不是独立存在的，在一个系统中，往往会有多种体系结构共存和相互融合，形成更复杂的框架结构，即异构体系结构。随着信息时代的发展，用户对软件的需求越来越多，因此，系统设计采用的软件体系结构要求多变和复杂，异构体系结构的使用也越来越广泛。

异构体系结构是多种纯软件体系结构的融合，为大粒度重用软件元素提供了便利条件。采用异构体系结构来设计软件系统，其原因如下：

(1) 从根本上来说，不同体系结构风格有各自的优点和缺陷，用户应该根据具体情况来选择系统的框架结构，以解决实际问题。

(2) 关于框架、通信和体系结构问题，目前存在着多种不同的标准。在某段时间内，一种标准占据了统治地位，但其变动最终是绝对的。

(3) 在实际工作中，总会遇到一些遗留下来的代码，它们仍然有用，但是与新系统的框架结构不一致。出于技术与经济因素的考虑，决定不再重写它们。选择异构体系结构风格，可以实现遗留代码的重用。

(4) 在某一单位中，规定了共享软件包和某些标准，但仍会存在解释和表示习惯上的不同。选择异构体系结构风格，可以解决这一问题。

不同体系结构的组合主要包括以下两种方式。

(1) 空间异构：允许构件使用不同的连接件，不同子系统采用不同的体系结构。构件可以通过连接件来访问仓库，也可以用管道和其他构件进行交互。用户应该根据功能和性能，为每个子系统选择合适的体系结构风格。

(2) 分层异构：软件元素按层次结构进行组织，每一层使用不同的体系结构。

异构体系结构的组合方式很多种，例如，可以采用平行的方式，即根据软件各个子系统的结构、功能和性能，为每个或每类子系统选择相应的体系结构；也可以利用分层组织，即某种体系结构的一个组成部分在其内部可以是另一种与之完全不同的结构，以完全不同的结构类型完整描述体系结构中的每一层描述等。

例如，在设计开发的社会保险管理信息系统中，系统需要完成劳动和社会保险的所有业务管理，即“五保合一”管理的功能。整个业务流程十分复杂，牵涉面广泛。为了尽量降低维护成本，提高可重用性和软件开发效率，可引入异构软件体系结构的设计思想。

首先，整个系统可采用层次式软件体系结构，在层次式结构的业务管理层中又采用了正交体系结构。

层次式软件体系结构的使用不仅能够满足不同规模的用户的需求，还可以方便地在具有基本功能的基本系统和具有复杂功能的扩充系统之间进行选择，而且，各个抽象的层次同时可以作为一种知识积累，对于同类软件的快速开发有着很大的作用。

这里，社会保险管理信息系统的设计可分成具有 4 个层次的层次式软件体系结构，如

图 3-53所示。通用核心层完成的是软件的一些通用的公共操作，这些操作能够尽量做到不与具体的数据库和表结构相关。通用核心层操作不仅可以应用于社会保险管理信息系统中，还可以方便地移植到其他的应用软件上。

基层单位管理平台是社会保险管理信息系统数据采集的重要来源，其包括劳资人事管理、工资管理、岗位管理和社会保险管理系统。其中，社会保险管理系统是社会保险管理信息系统的一个子集，是社会保险管理信息数据采集的主要来源。

扩展应用层是在典型应用系统的基础上扩充了一些更为复杂的功能，如对政策决策提供依据和支持，对政策执行状况进行监测，以及社会保险信息发布及个人账户电话语音查询等。

业务管理系统能够实现数据的初步汇总，它与其内包含的两层一起构成了社会保险管理信息的典型应用系统，完成社会保险的主要业务管理。在社会保险管理信息的业务管理系统的设计中引入了正交体系结构，将整个系统设计为三级正交结构，第一级划分为 8 个线索，如图 3-54 所示。

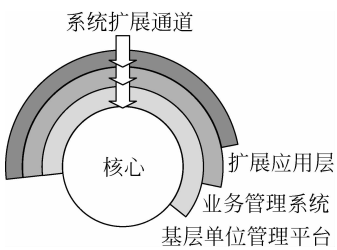

图 3-53　社会保险管理信息系统软件的层次结构

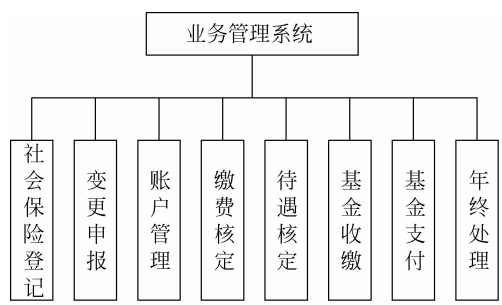

图 3-54　系统一级线索结构

每个一级线索又可划分为若干个二级线索。例如，一级线索“年终处理”可划分为“下年工作准备”、“发放账户通知单”、“保险金收支余额”3 个二级线索。每个二级线索又可划分为若干个三级线索。例如，二级线索“下年工作准备”可划分为“政策参数”、“企业缴费参数”、“缴费基数核算”及“工资和待遇预录入”4 个三级线索。最终，整个业务管理系统为三级正交线索，5 个层次。其中的一条完整的线索结构如图 3-55 所示。

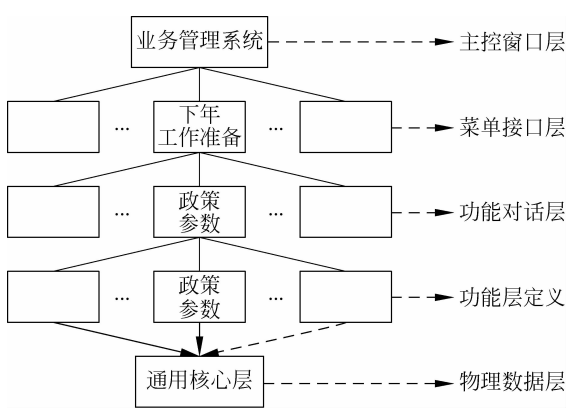

图 3-55　一条完整的线索结构