武汉大学
WUHAN UNIVERSITY

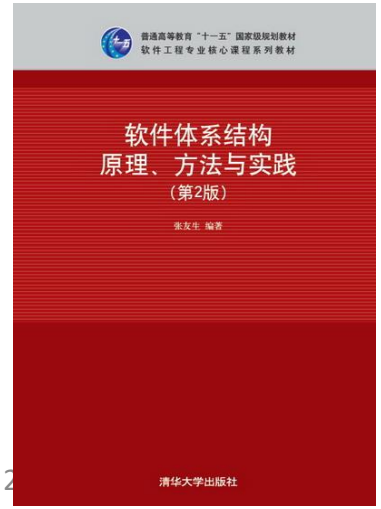
第3章 软件体系结构风格

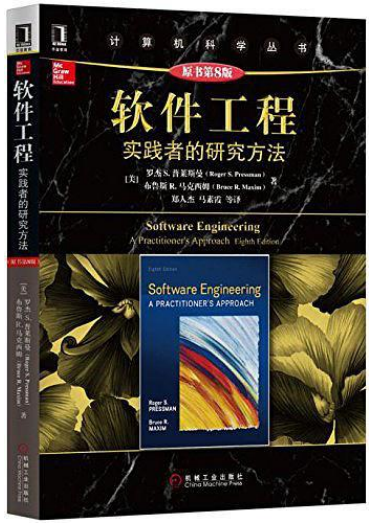
授课教师： 应时

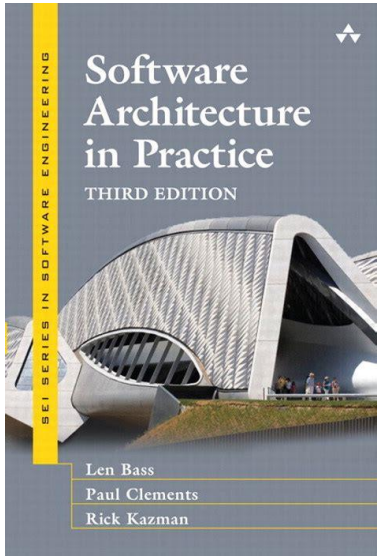
2019-10-23

参考书

- 张友生编著，软件体系结构原理、方法与实践（第2版），清华大学出版社，2014年1月。
- [美]罗杰 S. 普莱斯曼，译者: 郑人杰，软件工程：实践者的研究方法（原书第8版），机械工业出版社，2016年11月。
- 温昱，软件架构设计（第2版）—程序员向架构师转型必备，电子工业出版社，2012年07月。
- Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice (3rd Edition), Addison-Wesley Professional, Sep 25, 2012.







参考书

- 软件建模与设计：UML、用例、模式和软件体系结构，原书名：Software Modeling & Design: UML, Use Cases, Patterns, & Software Architectures，作者：(美)Hassan Gomaa，译者： 彭鑫、吴毅坚、赵文耘，机械工业出版社，2014 年8月。
- 软件架构建模和仿真：Palladio 方法，原书名：Modeling and Simulating Software Architectures: The Palladio Approach，作者： （德）拉尔夫•H.雷乌斯纳（Ralf H.Reussner） 等，译者： 李必信 等，机械工业出版社，2018 年9月。

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。**

软件体系结构风格

- **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。**
  - **术语表中包含一些构件和连接件类型。**
  - **这组规则指出系统是如何将这些构件和连接件组合起来的。**

- **软件体系结构风格描述了一个系统家族，或某一特定应用领域，关于组织系统方式的惯用模式，反映多个系统所共有的结构和语义特性。**

软件体系结构风格

- **使用常用的、规范的方法，来设计软件体系结构，就可以使别的设计师很容易地理解系统的设计方案。**
  - **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

- 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。
- 软件体系结构风格为大粒度的软件重用提供了可能。

- 对于应用体系结构风格来说，设计师在风格的基础上，还有很大的选择余地。
  - 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。

3.1 经典软件体系结构风格

- 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了体系结构风格的最关键的4要素内容，即
  - ① 提供一个词汇表
  - ② 定义一套配置规则
  - ③ 定义一套语义解释原则
  - ④ 定义对基于这种风格的系统所进行的分析。

3.1 经典软件体系结构风格

- **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：进程通讯、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。

3.1.1 管道与过滤器

- 在管道与过滤器风格中，
  - 连接件就象是管道，构件被称为过滤器。
  - 管道把数据流，从一个过滤器的输出，传输到另一过滤器的输入。

- 管道与过滤器风格的要点是：过滤器（即构件）必须是独立的实体，它不能与其它的过滤器共享数据，而且一个过滤器不知道它上游和下游过滤器的标识。
- 图3-1是管道与过滤器风格的示意图。

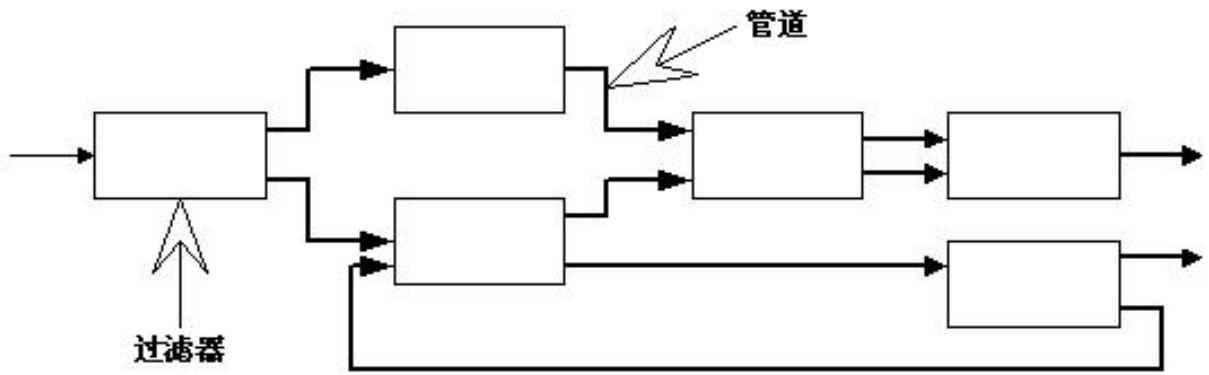
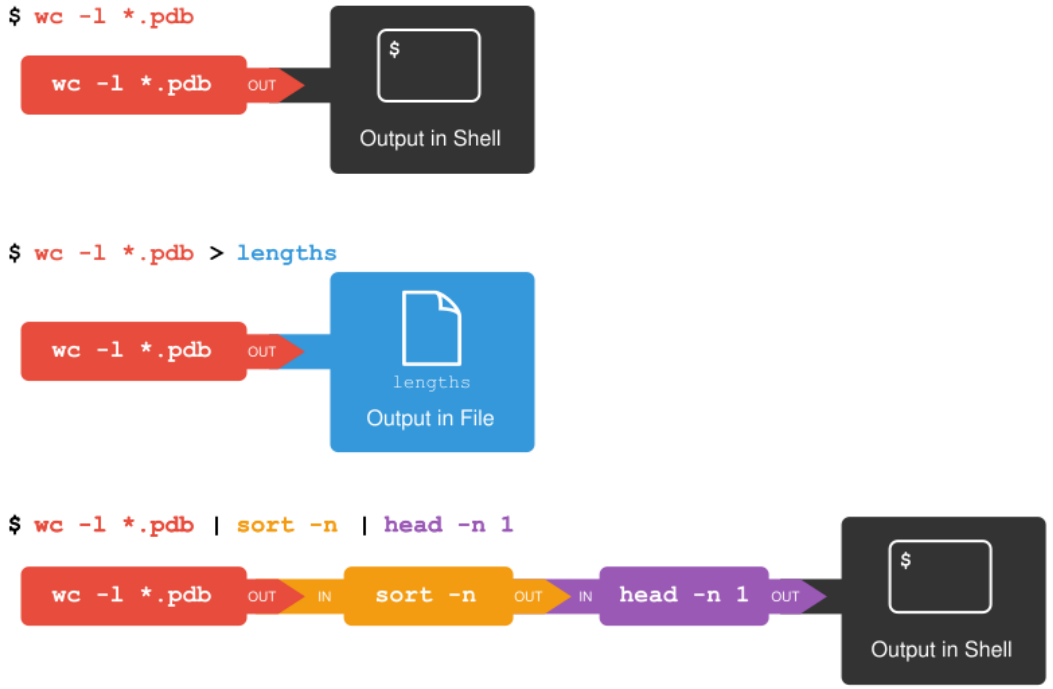


图3-1 管道与过滤器风格的体系结构

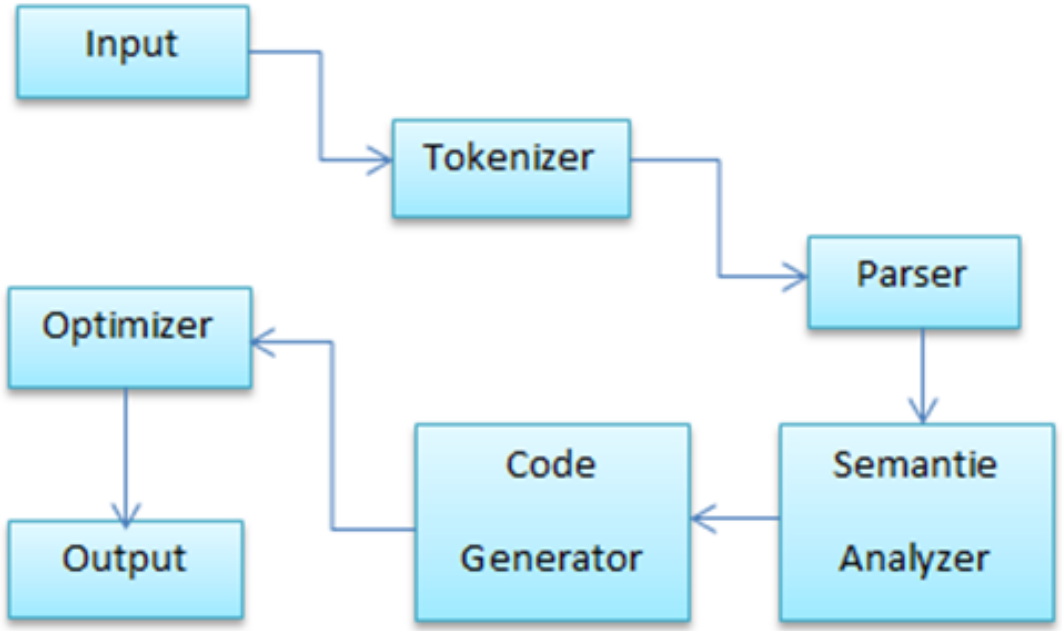
3.1.1 管道与过滤器

- 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。



3.1.1 管道与过滤器

- 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。



3.1.1 管道与过滤器

- **管道与过滤器风格的软件体系结构的优点：**
  - ① 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 允许对一些属性，如吞吐量、死锁等，进行分析。
  - ⑥ 支持并行执行。
    - ✓ 每个过滤器是作为一个单独的任务完成，因此可与其它任务并行执行。

3.1.1 管道与过滤器

- **管道与过滤器风格的软件体系结构的缺点：**
  - **通常导致进程成为批处理的结构。不适合处理交互的应用。**
    - ✓ **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。**
  - **如果在数据传输上没有通用的标准，那么每个过滤器都需要去解析或合成数据，这会导致系统性能下降，并增加了编写过滤器的复杂性。**

3.1.2 抽象数据类型和面向对象系统

- 抽象数据类型概念对软件系统有着重要作用，面向对象系统就是在抽象数据类型上的拓展。

- 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，被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。

3.1.2抽象数据类型和面向对象系统

- 面向对象的系统有许多优点，并早已为人所知：
  - 因为一个对象对其它对象隐藏了它的表示，所以可以改变一个对象的表示，而不影响其它的对象。
  - 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。

3.1.2抽象数据类型和面向对象系统

- 但是，面向对象的系统也存在着某些问题：
  - 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，就必须修改所有显式调用它的其它对象，还要去消除由此带来的一些副作用。

3.1.3 基于事件的系统

- 基于事件的体系结构风格的要点是：
- 系统有一个事件中心，
  - 接收来自构件对其所关注事件的注册请求，并构造事件关注注册表。
  - 接受构件发送来的事件，并根据事件关注注册表，把事件转发给那些注册了关注信息的构件。
- 系统中的构件，通过系统中的事件中心，彼此进行关联和协同。
  - 构件把其产生的事件，发送给事件中心
  - 构件接收到事件中心转发来的事件，触发自身拥有的相应处理方法的执行。

- 因此，该风格也称为（基于事件中心的）隐式调用。

3.1.3 基于事件的系统

- 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。

3.1.3 基于事件的系统

- 支持基于事件的隐式调用的应用系统很多。
  - 例如在某工具系统中，编辑器和变量监视器可以去注册，以关注相应Debugger的断点事件。
    - ✓ 当Debugger在断点处停下时，编辑器卷屏显示到断点，变量监视器刷新变量数值。
    - ✓ 此时，Debugger本身只触发事件，并不关心哪些过程会因本次事件而被启动，也不关心启动后的那些过程会如何处理这一事件。
  - 其它的例子还有：
    - ✓ 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
    - ✓ 在编辑器中支持语法检查

3.1.3 基于事件的系统

- **隐式调用系统的主要优点有：**
  - **为软件重用提供了强大的支持。**
    - ✓ **当需要将一个构件加入到现存的系统中时，只需将它注册到系统的事件中即可。**
  - **为改进系统带来了方便。**
    - ✓ **当用一个构件代替另一个构件时，不会影响到其它构件的接口。**

3.1.3 基于事件的系统

- 隐式调用系统的主要缺点有：
  - 构件相互间进行间接交互（有利有弊）
    - ✓ 一个构件触发一个事件时，不能确定其它构件是否会响应它。
  - 数据交换与性能问题
    - ✓ 数据交换需通过事件中心进行。
    - ✓ 事件中心可能会成为系统的性能瓶颈。
  - 验证过程正确性会更难，因为难以收集过程触发事件时的上下文约束信息。

3.1.4 分层系统

- 层次系统组织成一个层次结构。其中每一层是其上层的服务提供者，同时又是其下层的服务客户。
  - 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。
- 在分层系统的风格中
  - 构件是各层的组成主体。
  - 连接件通过层间的交互协议，来定义层间的交互过程。
  - 事实上，拓扑约束就包括了对相邻层间交互的约束。
- 图3-3是层次系统风格的示意图。

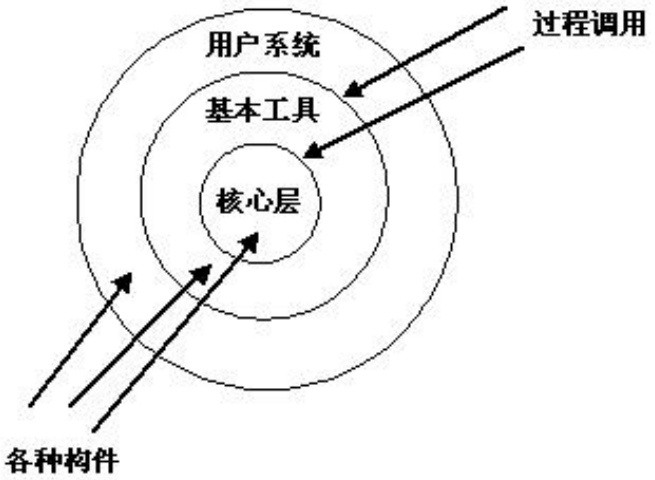
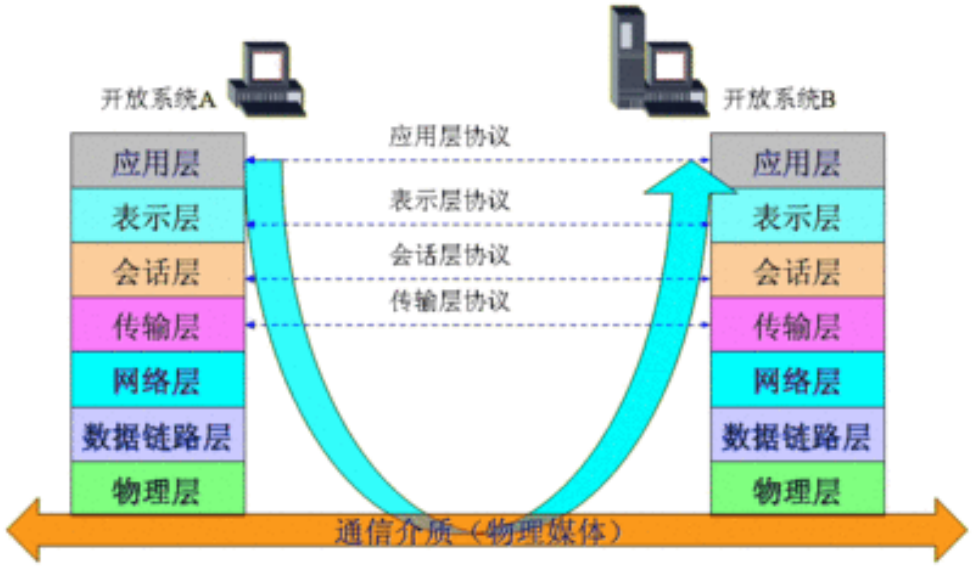
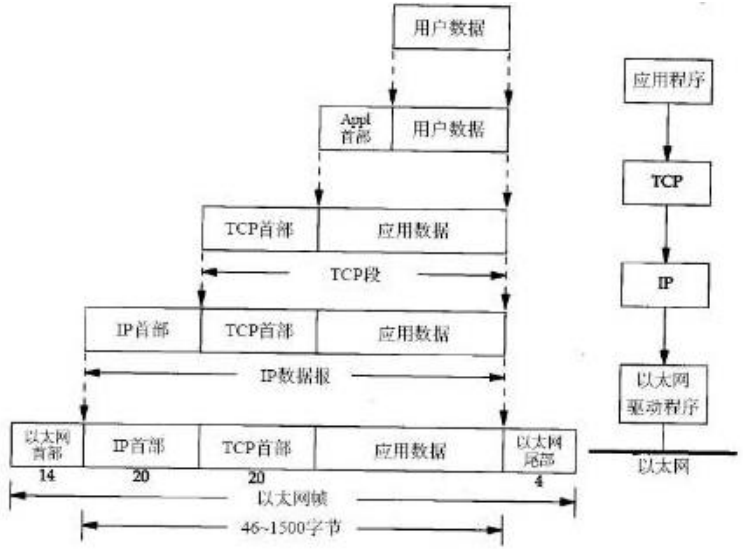


图3-3 层次系统风格的体系结构

3.1.4 分层系统

- 层次系统最广泛的应用是网络分层通信协议。
- 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。





3.1.4 分层系统

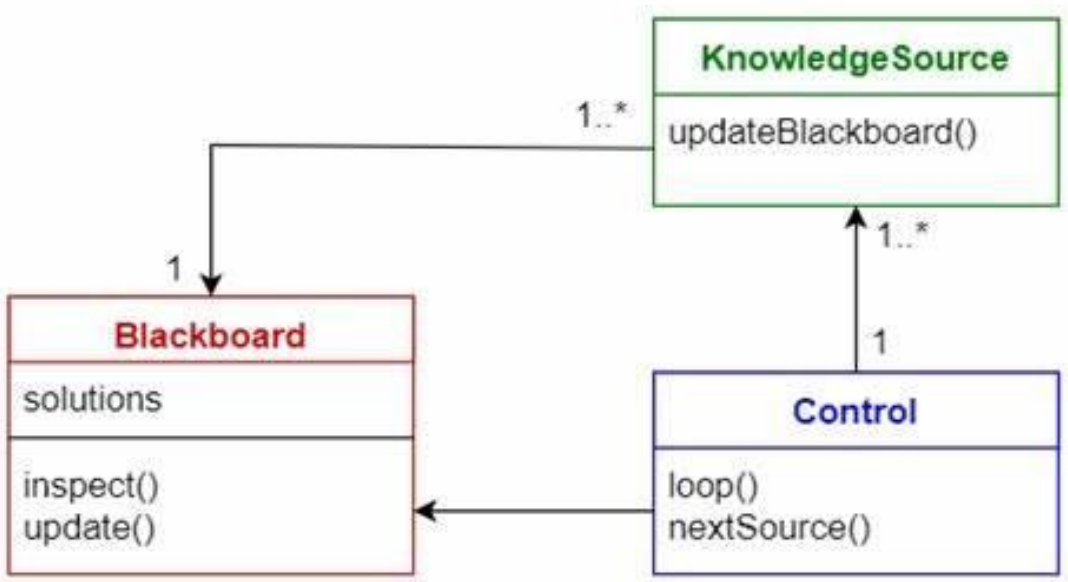
- **层次系统有许多可取的属性：**
  - **支持基于抽象层次的增量式设计，使设计师可以对一个复杂系统，进行层次化的抽象与分解，并进行设计。**
  - **支持重用**
    - ✓ **由于每一层最多只影响到其上下两层，而且层间接口是确定的。因此，只要能保证层间接口不变，每层可以拥有用不同的实现方案。**
    - ✓ **支持抽象层的设计概念，抽象层具有很好的可重用性。**

3.1.4 分层系统

- **层次系统的不足之处：**
  - **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。**
  - **很难找到一个合适的、正确的层次抽象方法。**

3.1.5 黑板风格与仓库风格

- **黑板系统有三部分组成：**
  - **黑板数据结构：一种中央数据结构，它用于说明当前数据的状态。**
  - **知识源：一种自主的构件，它用于对黑板（中央）数据进行操作，或利用黑板（中央）数据，完成自己的操作。**
  - **控制器：**



3.1.5 黑板风格与仓库风格

- The Blackboard pattern is a design pattern, used in software engineering, to coordinate separate, disparate systems that need to work together, or in sequence, continually prioritizing the actors (or knowledge sources).
- The blackboard consists of
  - a number of stores or "global variables", like a repository of messages, which can be accessed by separate autonomous processes, which could potentially be physically separate.
  - A "controller" monitors the properties on the blackboard and decides which actors (or knowledge sources) to prioritize.
- The blackboard pattern allows multiple processes to work closer together on separate threads, polling and reacting if needed.

3.1.5 黑板风格与仓库风格

- A blackboard system is an artificial intelligence approach based on the blackboard architectural model, where a common knowledge base, the "blackboard", is iteratively updated by a diverse group of specialist knowledge sources, starting with a problem specification and ending with a solution.
- Each knowledge source updates the blackboard with a partial solution when its internal constraints match the blackboard state.
- In this way, the specialists work together to solve the problem.

- The blackboard model was originally designed as a way to handle complex, ill-defined problems, where the solution is the sum of its parts.

3.1.5 黑板风格与仓库风格

➢ 黑板系统中的中央数据单元

- **中央数据单元是整个系统的核心部件。**
  - ✓ 它对系统需要解决的问题，预先进行了分析和定义。
  - ✓ 总结出了系统运行过程中将要出现的多种状态。
  - ✓ 制定了在这些状态下系统的响应策略。
- **中央数据单元中的数据不只是单纯的数据信息，它们代表了系统的某种状态，属于状态数据。**
  - ✓ 这些数据由数据源提供，
  - ✓ 这些数据在中央数据单元中，依据一定的数据结构形式，组织在一起。
  - ✓ 这些数据随着数据源信息的改变而变化。

3.1.5 黑板风格与仓库风格

➢ 黑板系统中的知识源

- 是特定应用程序知识的独立散片。
  - ✓ 知识源彼此之间在逻辑上和物理上都是独立的，只与产生它们的应用程序有关。
  - ✓ 多个数据源之间的交互只在黑板内部发生，对外部是透明的。
- 知识源把问题分成几个部分，每个部分独立计算，响应黑板上的变化。
  - ✓ 即每个知识源代理(agent) 都按照他们自己的方式，工作在它们感兴趣的方面或它们的知识能够处理的方面。
- 知识源在可能的时候，向黑板添加新的知识，以供其他知识源开展进一步的工作。

3.1.5 黑板风格与仓库风格

➢ 黑板系统中的控制器

- 控制器，根据仓库状态的变化，驱动知识源的执行。
- 知识源将系统需要处理的信息，源源不断地输入仓库中，从而导致仓库的状态信息发生变化。
- 当状态信息的变化，符合系统预先定义好的某些控制策略时，相应的操作就会被触发，这样就实现了系统的功能控制。

- 控制器还能限制知识源代理，对黑板访问。
- 控制器并不一定是独立的单元，它可以位于知识源的仓库中，或者作为一个独立部分单独存在，没有绝对的定式，需要由设计者，根据系统实际情况做出抉择。

3.1.5 黑板风格与仓库风格

- 黑板体系结构实现的基本出发点是：已经存在一个对公共数据结构进行协同操作的独立程序集合。
  - 每个这样的程序，专门解决一个子问题，但需要协同工作，才能共同完成整个问题的求解。
  - 这些专门程序是相互独立的，它们之间不存在互相调用，也不存在可事先确定的操作顺序。
  - 这些程序操作的次序是由问题求解的当前工作状态，而动态确定的。

3.1.5 黑板风格与仓库风格

- 黑板风格是某些对人类行为进行模拟的人工智能应用系统的重要设计方法之一。
  - 例如，语音识别、模式识别、三维分子结构建模。

- 最早应用黑板体系结构的也是一个人工智能领域的应用系统程序：Hearsay语音识别项目。
  - 该系统以自然语音的语音信号为输入，经过语音、词汇、句法和语义等多个方面的分析，得到用户对数据库的查询请求。

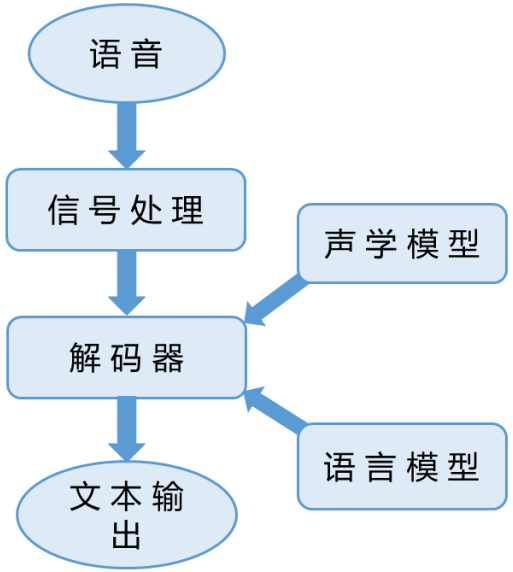
3.1.5 黑板风格与仓库风格

- 黑板系统被用于复杂的人工智能系统开发，这些系统
  - 找不到确定的求解策略。
  - 需要处理不确定性，需要解决各种冲突。
  - 当把整个问题分解成子问题时，各个子问题涵盖着不同的领域知识和解决方法。而解决每一个子问题，需要用到不同的问题表达方式和求解模型。

3.1.5 黑板风格与仓库风格

➢ 语音识别

- **信号处理模块**将根据人耳的听觉感知特点，抽取语音中最重要的特征，将语音信号，转换为**特征矢量序列**。
- **解码器**根据声学模型和语言模型，将输入的语音特征矢量序列，转化为**字符序列**。
- **声学模型**是对声学、语音学、环境的变量，以及说话人性别、口音的差异等的知识表示。
- **语言模型**则是对一组字序列构成的知识表示。



3.1.5 黑板风格与仓库风格

➢ 语音识别

- Separate threads can process different parts of the sound sample, updating the blackboard with words that have been recognized.
- Then another process can pick up these words and perform grammar and sentence formation.
- Meanwhile more words and meanings are coming in, and eventually even higher level processes can pick up the formed sentences and various alternative guesses and begin to formulate it's meaning,
- then further intelligence systems can start to choose the most appropriate answer.

- All these systems have access to the blackboard and work together through it's central platform.

3.1.5 黑板风格与仓库风格

- **自主机器人**
  - **自主机器人要执行多个任务。**
  - **自主机器人要处理多个输入信息，并按优先级的分级情况，调用动作。**

3.1.5 黑板风格与仓库风格

➢ 简单的应用案例—系统注册表

- 早期的计算机硬件和软件系统的配置信息均被各自保存在配置文件（.ini）中，这些文件散落在系统的各个角落，人们很难对其进行维护。
- 为此，引入系统注册表，将所有的.ini 文件集中起来，形成共享仓库，为系统运行起到了集中资源配置管理和控制调度的作用。
- 系统注册表信息，影响或控制系统的行为。
  - ✓ 在应用软件安装、运行、卸载时，通过对系统注册表进行添加、修改和删除信息，就可以达到改变系统功能和控制软件运行的目的。

3.1.5 黑板风格与仓库风格

- **简单的应用案例--剪贴板**
  - 剪贴板内置在Windows 中，是使用系统的内部资源RAM，或虚拟内存来临时保存剪切和复制的信息。
  - 剪贴板使得在各种应用程序之间，传递和共享信息成为可能。
  - 剪贴板是存储、传递和交换信息的公共区域，形成了共享仓库。
  - 注册表和剪贴板均像一个仓库一样，成为了计算机中存储和维护数据的重要场所。

3.1.5 黑板风格与仓库风格

- **仓库系统及知识库风格的优点**
  - **便于多客户共享大量数据，他们不关心数据是何时有的、由谁提供的、怎样提供的**
  - **既便于添加新的作为知识源代理的应用程序，也便于扩展共享的黑板数据结构。**
  - **较好的知识源可重用性**
  - **较好的系统鲁棒性**

3.1.5 黑板风格与仓库风格

- 仓库系统及知识库风格的缺点
  - 不同的知识源代理，对于共享数据结构要达成一致。
  - 对黑板数据结构的修改较为困难————要考虑到各个代理的调用。
  - 需要一定的同步/ 加锁机制，保证数据结构的完整性和一致性，增大了系统复杂度。
  - 不能保证有好的求解方案。
  - 低效。
  - 开发成本高。

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它构件和连接件连接。
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。

3.1.6 C2风格

- 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。

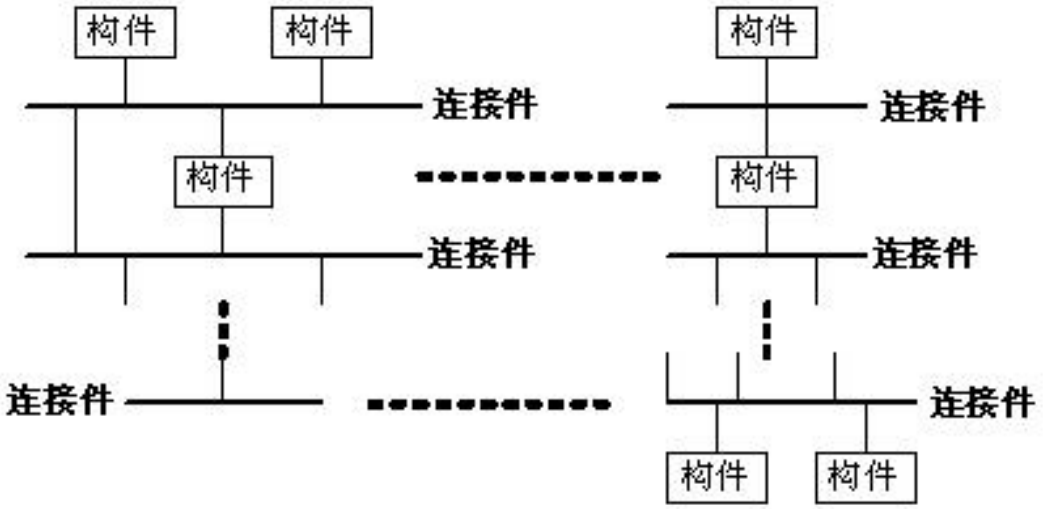


图3-5 C2风格的体系结构

3.1.6 C2风格

- 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 所有构件之间的通讯是，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。

④ 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

- 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（Client/Server，C/S）模型的演变。

- 在集中式计算技术时代，被广泛使用的是**大型机/小型机计算模型**。它是通过一台物理上与宿主机相连接的非智能终端，来运行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。
- 集中式的系统，使用户能共享特殊的硬件设备，如磁盘机、打印机等。
- 但随着用户的增多，对宿主机能力的要求很高。

3.2 客户/服务器风格

- 20世纪80年代以后，集中式结构逐渐被以PC机为主的微机网络所取代。
- 形成了个人计算模型
  - 个人计算机和工作站的采用，改变了协作计算模型，产生了分散进行计算的个人计算模型。
  - 解决了大型机系统固有的缺陷：如缺乏灵活性，无法适应信息量急剧增长的需求，并为整个企业提供全面的解决方案等等。
- 随后形成了网络计算模型
  - 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。

3.2 客户/服务器风格

- 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。

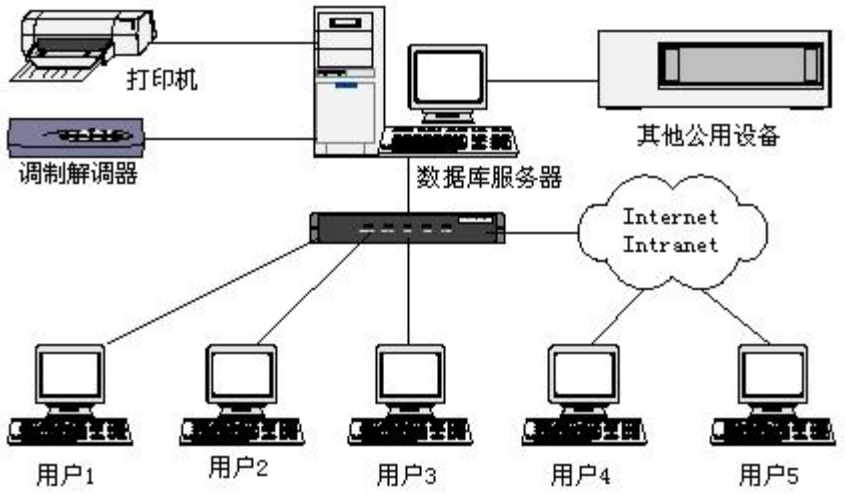


图3-6 C/S体系结构示意图

3.2 客户/服务器风格

- 服务器负责有效地管理系统的资源，其任务集中于：
  - 数据库安全性的要求；
  - 数据库访问并发性的控制；
  - 数据库前端的客户应用程序的全局数据完整性规则；
  - 数据库的备份与恢复；

3.2 客户/服务器风格

- **客户应用程序的主要任务是：**
  - **提供用户与数据库交互的界面；**
  - **向数据库服务器，提交用户请求，并接收来自数据库服务器的信息；**
  - **利用客户应用程序，对存在于客户端的数据，执行应用逻辑要求。**

3.2 客户/服务器风格

- 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。

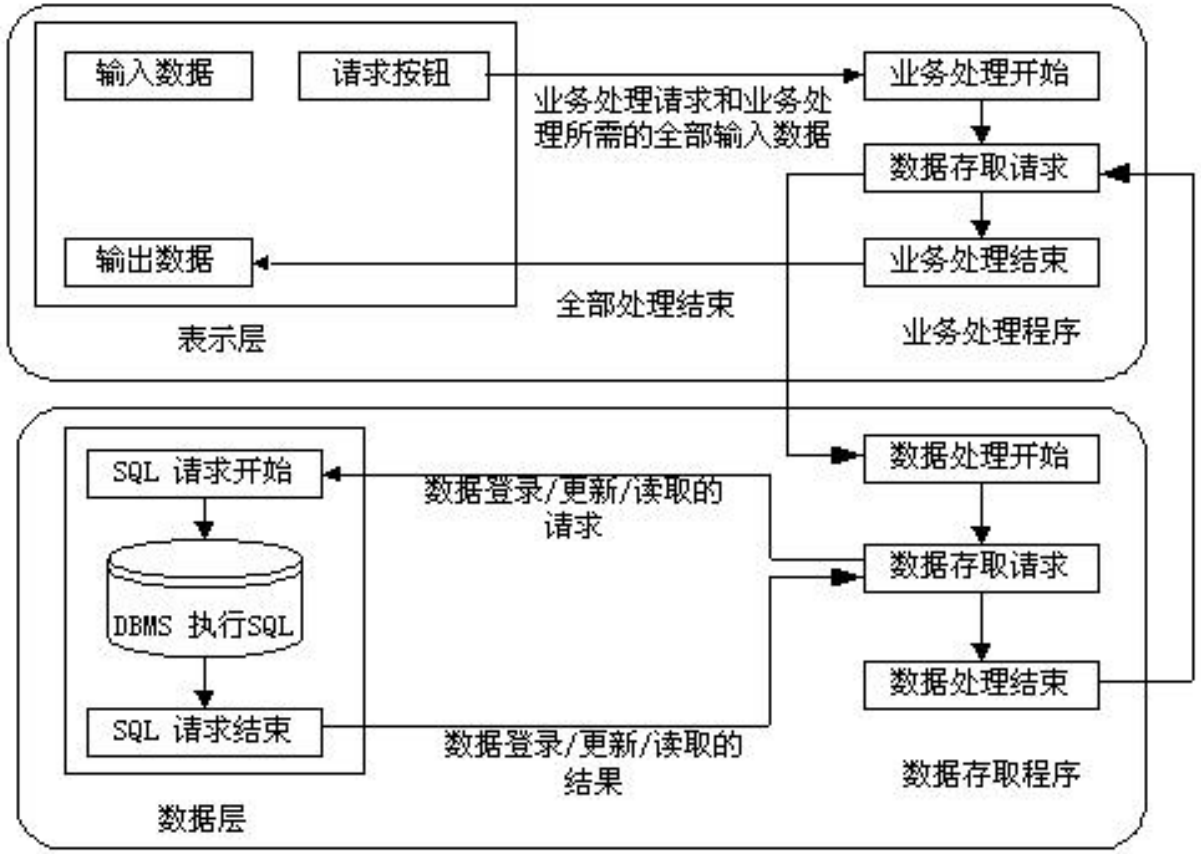


图3-7 C/S结构的一般处理流程

3.2 客户/服务器风格

- 在一个C/S体系结构的软件系统中，
  - 客户应用程序针对一个小的、特定的数据集，如一个表的某一行，进行操作，而不是像文件服务器那样对整个文件进行操作。
  - 因此，只需对某一条记录加锁，而不需要对整个文件加锁，这样就保证了系统的并发性，使网络上传输的数据量减到最少，从而改善了系统的性能。

3.2 客户/服务器风格

- 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。

3.2 客户/服务器风格

- 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。

3.2 客户/服务器风格

- 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：

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它平台上运行。
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

- 随着企业规模的日益扩大，软件的复杂程度不断提高，传统的二层C/S结构存在以下几个局限：
  1. 二层C/S结构是单一服务器，且以局域网为中心的，所以难以扩展至大型企业广域网，或Internet。
  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。

3.3 三层C/S结构风格

- 正是因为二层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。
- 与二层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）。

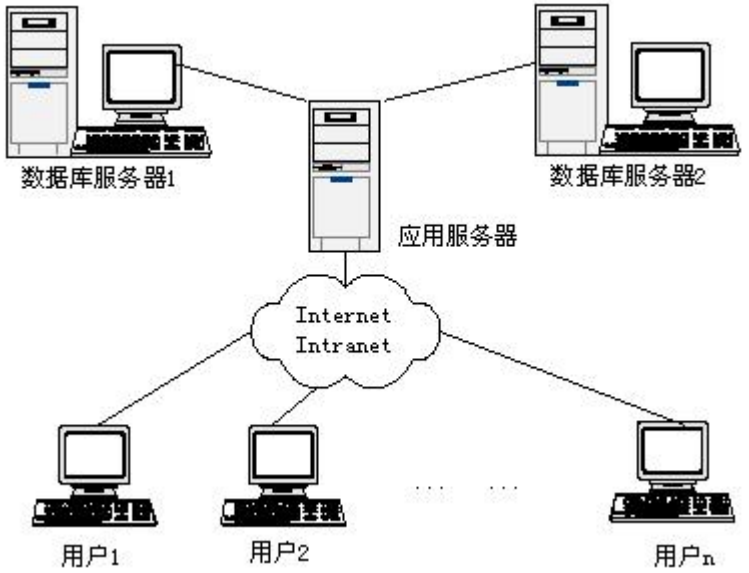


图3-8 三层C/S结构示意图

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。

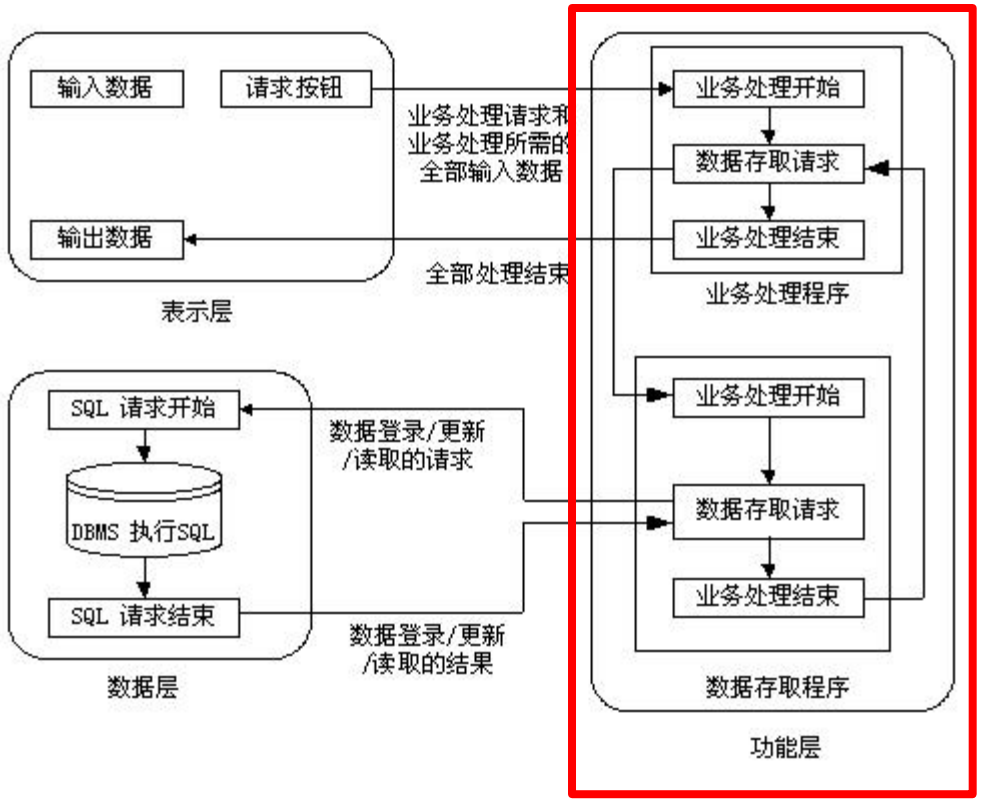


图3-9 三层C/S结构的一般处理流程

3.3.1 各层的功能

➢ 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。

3.3.1 各层的功能

➢ 2、功能层（应用服务器层）

- 功能层相当于应用的主体，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。按照定义好的格式，配置数据，打印订购合同，而这个处理所需要的数据，则要从数据层取得。
- 表示层和功能层之间的数据交往，要尽可能简洁。
  - ✓ 例如，用户检索数据时，要设法将有关检索要求的信息，一次性地传送给功能层；而由功能层处理过的检索结果数据，也一次性地传送给表示层。
- 通常，在功能层中包含有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。

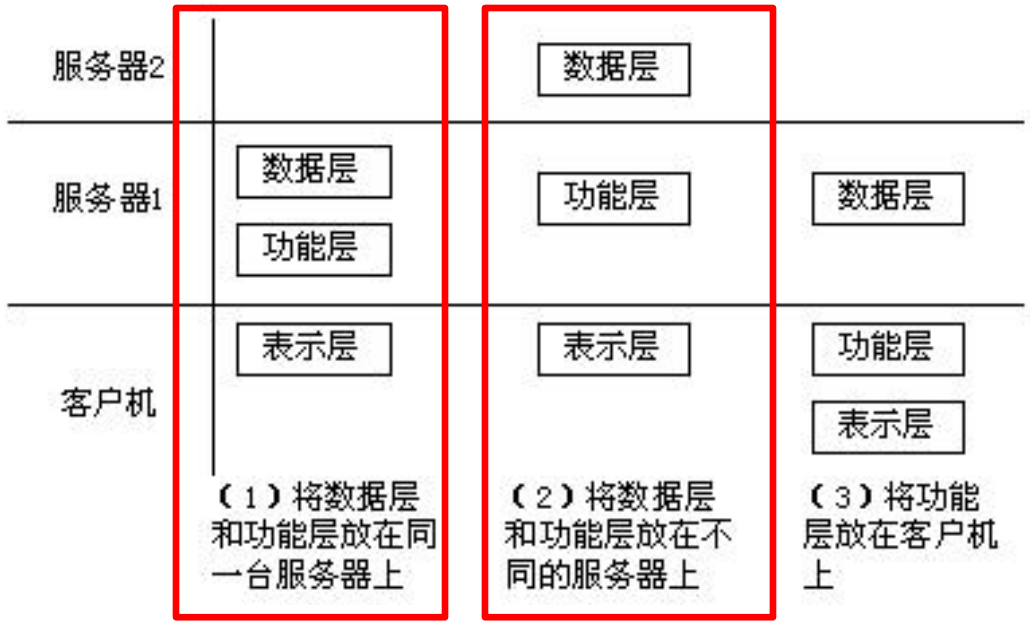
3.3.1 各层的功能

➢ 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进行读写。
- 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语言。

3.3.1 各层的功能

- 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（1）或3-10（2）所示。



3.3.1 各层的功能

- 如果象图3-10（3）所示的那样连功能层也放在客户机中，与二层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。

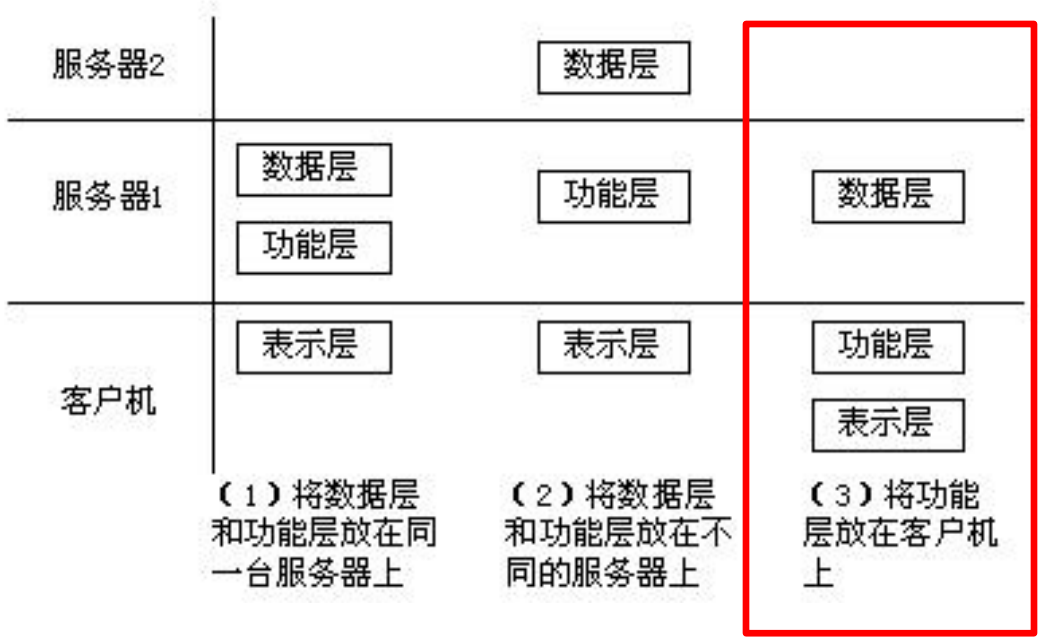


图3-10 三层C/S物理结构比较

3.3.1 各层的功能

- 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（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。

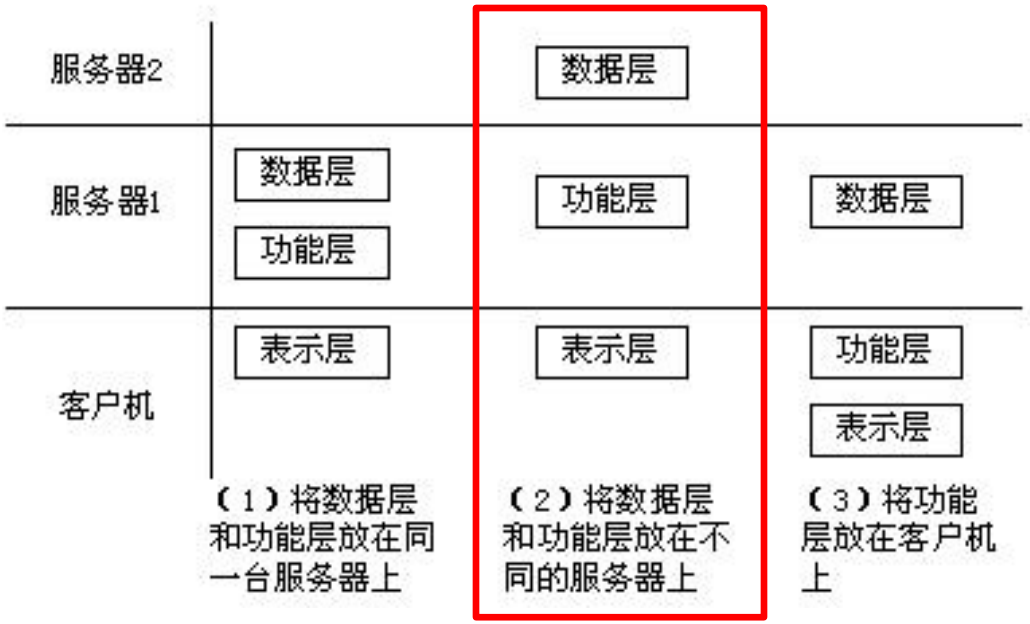


图3-10 三层C/S物理结构比较

3.3.1 各层的功能

- 为了更好地支撑三层体系结构的开发和运行，中间件是十分重要和必要的构件。
  - 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。

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它各种管理信息。
  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公里。
  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到2M。

3.3.2 三层C/S结构应用实例

- 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。

3.3.2 三层C/S结构应用实例

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。
- 每一层次都自己的特点
  - ✓ 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。
- 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与OOD技术，进行三层结构的分析与设计。

3.3.2 三层C/S结构应用实例

- 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。

3.3.2 三层C/S结构应用实例

- 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模型就是用来建模表示该现实世界的“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。

3.3.2 三层C/S结构应用实例

- **OOA模型划分为五个层次或视图，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100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了业务执行过程中的一部分业务处理逻辑，也是MIS中容易改变的部分。**
4. **结构层。结构层负责捕捉特定应用域的结构关系。分类结构表示类属成员的构成，反映通用性和特殊性。组装结构表示聚合，反映整体和组成部分。**
5. **主题层。主题层用于将对象归类到各个主题中，以简化OOA模型。为了简化劳动管理信息系统，将整个系统按业务职能划分为十三个主题，分别为：职工基本信息管理，工资管理，劳动组织计划管理，劳动定员定额管理，劳动合同管理，劳动统计管理，职工考核鉴定管理，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。**

3.3.2 三层C/S结构应用实例

- 在OOD方法中，OOD体系结构以OOA模型为设计模型的基础。
- OOD将OOA的模型作为OOD的问题论域部分（PDC），并增加其它三个部分：
  - 人机交互部分（HIC）
  - 任务管理部分（TMC）
  - 数据管理部分（DMC）
- 各部分与PDC一样划分为五个层次，但是针对系统的不同方面。

3.3.2 三层C/S结构应用实例

- **OOD的任务是为OOA所建立的应用模型，添加计算机技术特征。**

1. **问题论域部分（PDC）：以OOA模型为基础，包含那些执行基本应用功能的对象，可逐步细化，使其最终能解决实现限制、特性要求、性能缺陷等方面的问题。PDC封装了应用服务器功能层的业务逻辑。**
2. **人机交互部分（HIC）：指定了用于系统的某个特定实现的界面技术，在系统行为和用户界面的实现技术之间架起了一座桥梁。HIC封装了客户层的界面表达逻辑。**
3. **任务管理部分（TMC）：把有关特定平台的处理机制底层系统的其它部分隐藏了起来。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（DMC）：定义了那些与所有数据库技术接口的对象。DMC同样是三层结构中应用服务器的一部分。由于DMC封装了数据库访问逻辑，使应用独立于特定厂商的数据库产品，便于系统的移植和分发。**

3.3.2 三层C/S结构应用实例

- OOD的四个部分与三层结构的对应关系如图3-11所示。

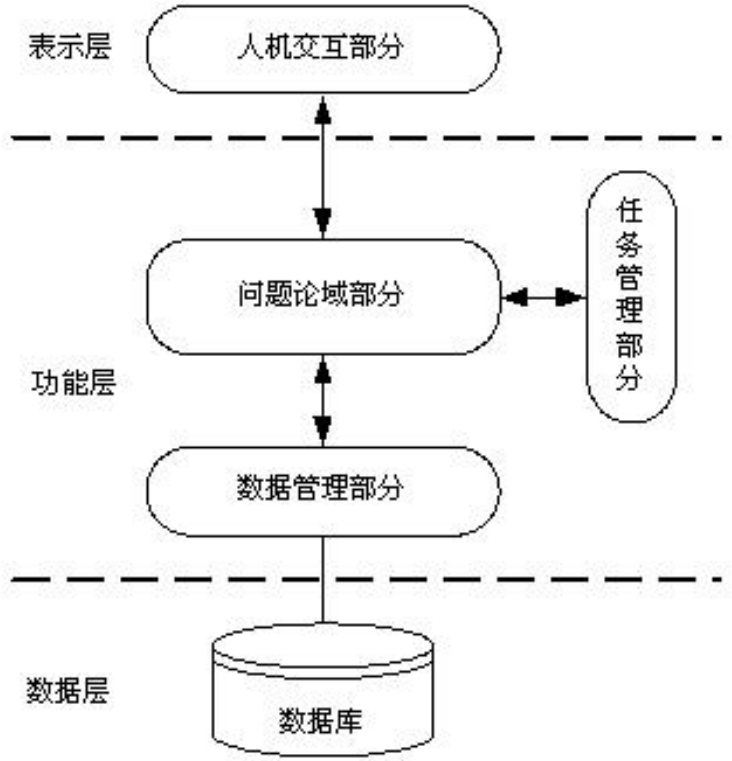


图3-11 OOD与三层C/S结构

3.3.2 三层C/S结构应用实例

➢ 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、本地或远程进程执行。

3.3.2 三层C/S结构应用实例

- 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。
- “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，提供给其它对象。

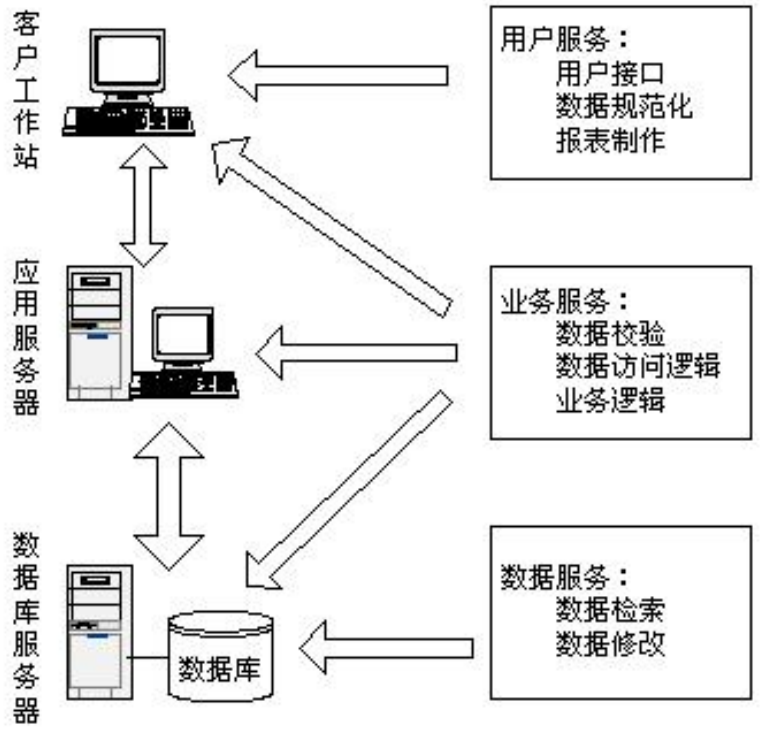


图3-12 服务模型结构图

3.3.2 三层C/S结构应用实例

- 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
  - 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。

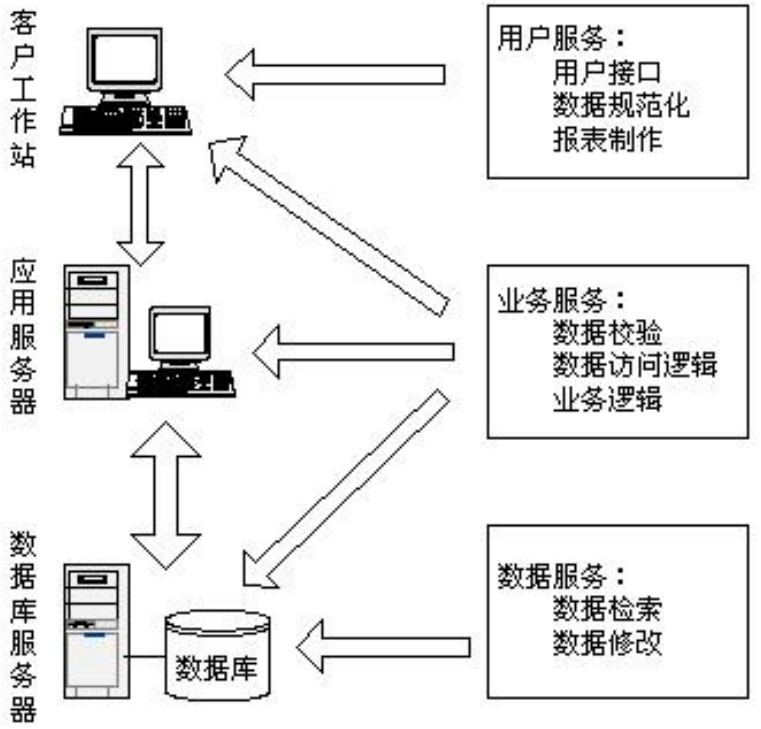


图3-12 服务模型结构图

3.3.2 三层C/S结构应用实例

- 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：
  - 一个构件能向当前层及构件层上下的任何一个层的其它构件，发出服务请求。
  - 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。

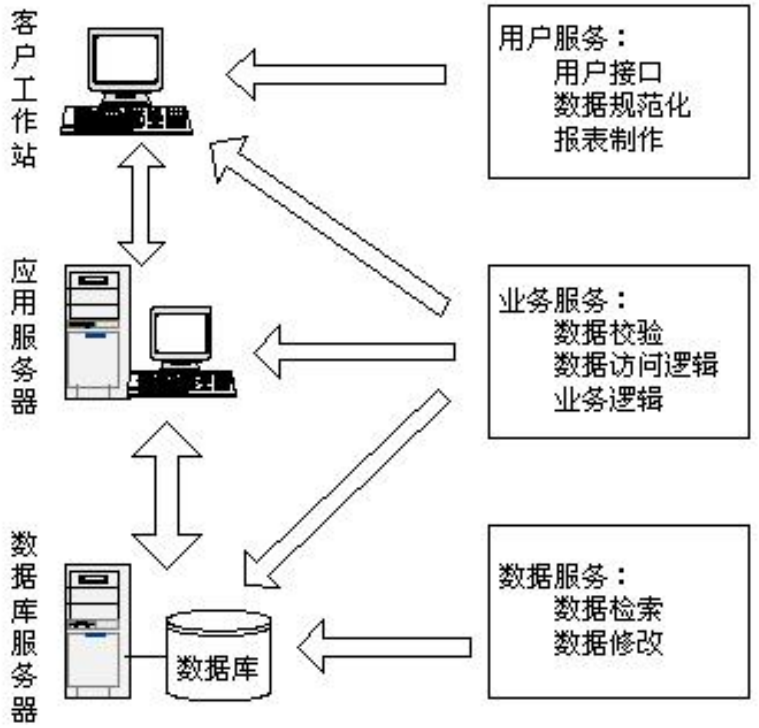


图3-12 服务模型结构图

3.3.2 三层C/S结构应用实例

- 在劳动管理信息系统的实现中，将PDC的十三个子系统以及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。

3.3.2 三层C/S结构应用实例

- 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
  - 事务服务器配置方案

3.3.2 三层C/S结构应用实例

（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。

3.3.2 三层C/S结构应用实例

➢ （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。

3.3.2 三层C/S结构应用实例

- （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，其它方面无须任何改变。

3.3.2 三层C/S结构应用实例

（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。

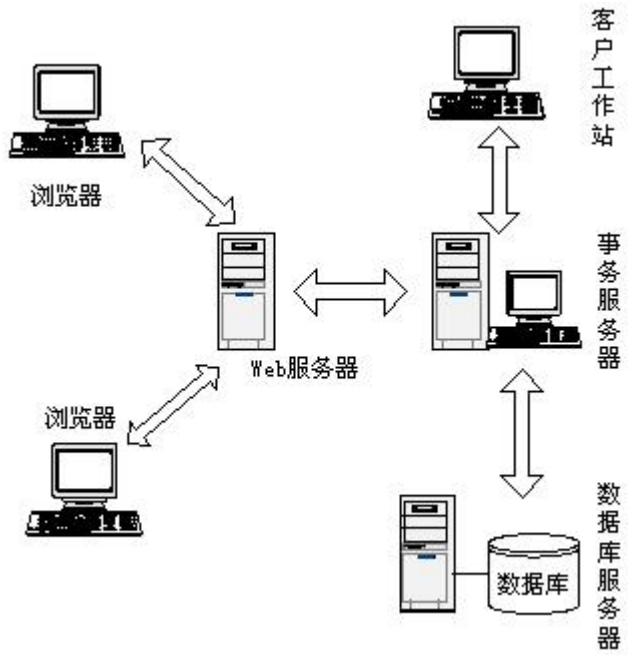


图3-13 向Intranet方式的转移

3.3.2 三层C/S结构应用实例

- 在上述各种方案中，除单机配置方案外，其它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。

3.3.2 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，与二层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。

3.3.2 三层C/S结构的优点

- （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。
- 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。

3.3.2 三层C/S结构的优点

- （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。
  - ✓ 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并可控制。

3.3.2 三层C/S结构的优点

- 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 浏览器/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：存储和检索数据库，响应功能层的数据处理请求，并将结果返回给功能层。

3.4 浏览器/服务器风格

- 浏览器/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。

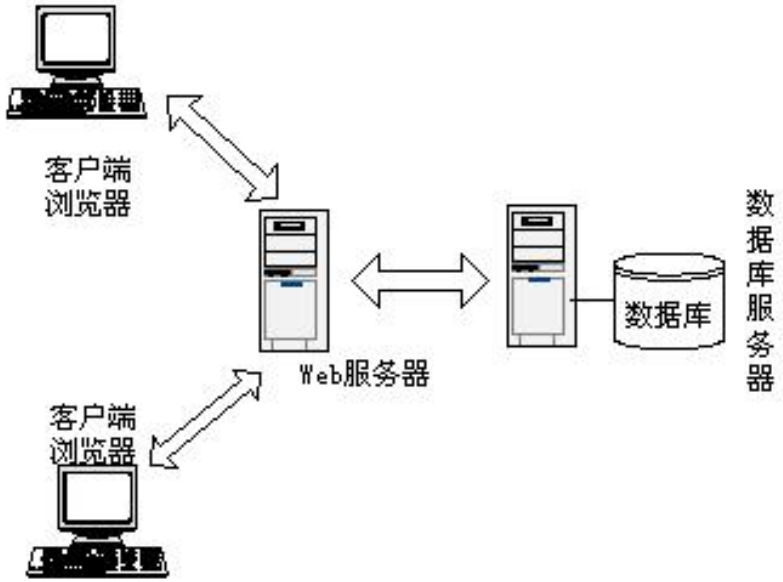


图3-14 B/S模式结构

3.4 浏览器/服务器风格

- 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，实现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，应用程序存放于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，键入相应的网址，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。

3.4 浏览器/服务器风格

- 基于B/S体系结构的软件和系统的安装、修改和维护，全在服务器端进行。
- 用户在使用系统时，仅仅需要一个浏览器，就可运行全部的模块，真正达到了“零客户端”的功能，很容易在运行时自动升级。
- B/S体系结构还提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的开放性基础。
- B/S结构的应用系统与Internet的结合，也使新的企业应用（例如，电子商务、客户关系管理等）的实现，成为可能。

3.4 浏览器/服务器风格

- 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. 采用B/S体系结构的应用系统，在数据查询等响应速度上，要远远地低于C/S体系结构。
3. 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。

3.4 浏览器/服务器风格

- 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，尽管B/S结构的计算机应用系统有很多的优点，但在未来一段时间内，仍将是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。

3.5 公共对象请求代理体系结构

- 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。
- 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：使用对象管理结构所规定的类接口，实现一些通用功能。

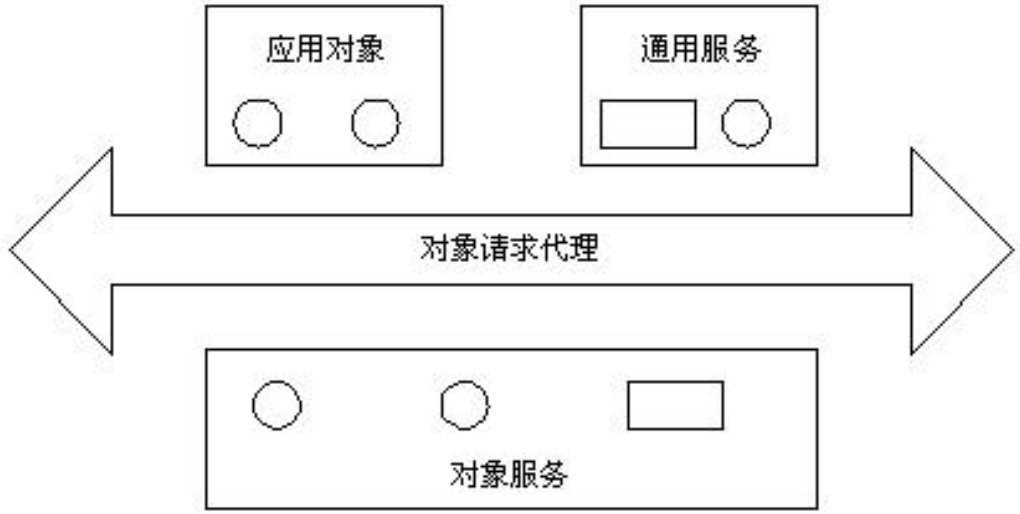


图3-15 对象管理结构

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）

3.5.1 CORBA技术规范

➢ 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，不必知道实现服务器对象的编程语言。
- 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。

3.5.1 CORBA技术规范

➢ 2、接口池

- CORBA的接口池包括了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。

3.5.1 CORBA技术规范

➢ 3、动态调用接口

- CORBA的动态调用接口，提供了一些标准函数，以供客户对象动态地创建请求、动态地构造请求参数。
- 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。

3.5.1 CORBA技术规范

➢ 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、对客户请求的认证等。

3.5.1 CORBA技术规范

- 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/其它构件
  - 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。

3.5.2 CORBA风格分析

- 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，设置需要的上下文状态，激发服务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，它包含了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。
  - 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。

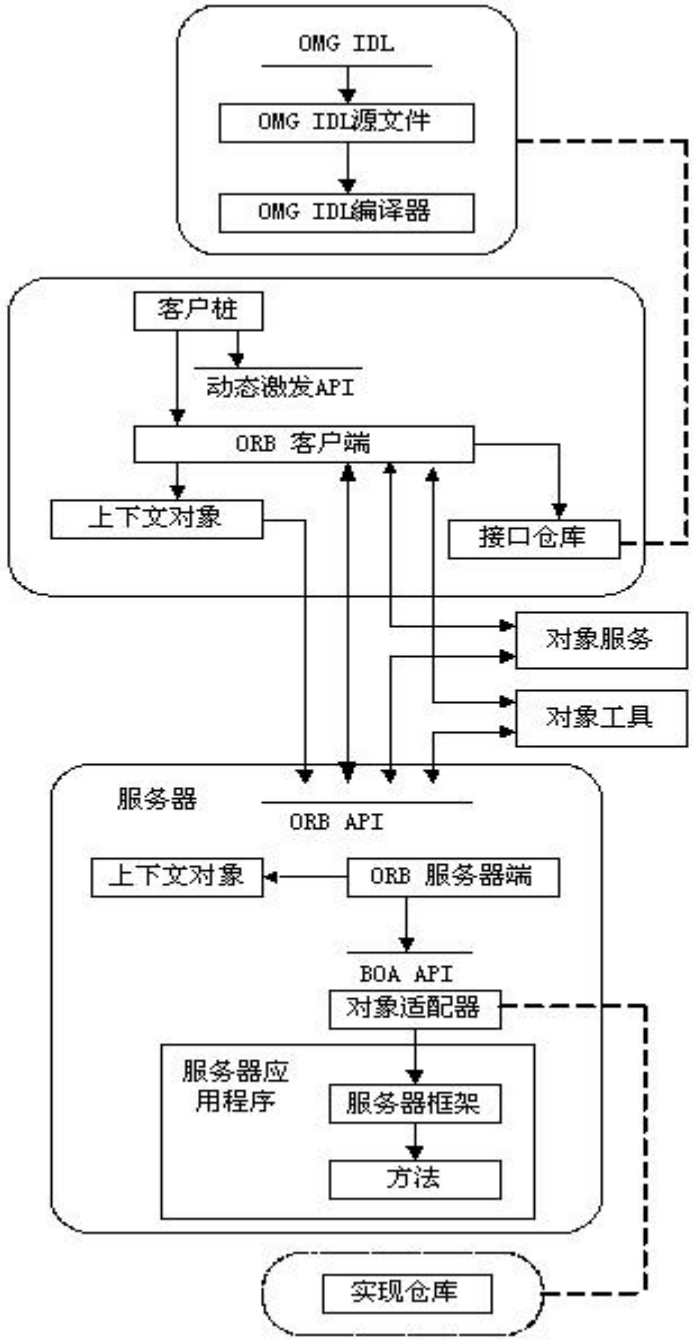


图3-16 CORBA的体系结构模式

3.5.2 CORBA风格分析

- CORBA体系结构模式充分利用了其时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。

3.5.2 CORBA风格分析

- **CORBA体系结构模式的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。**

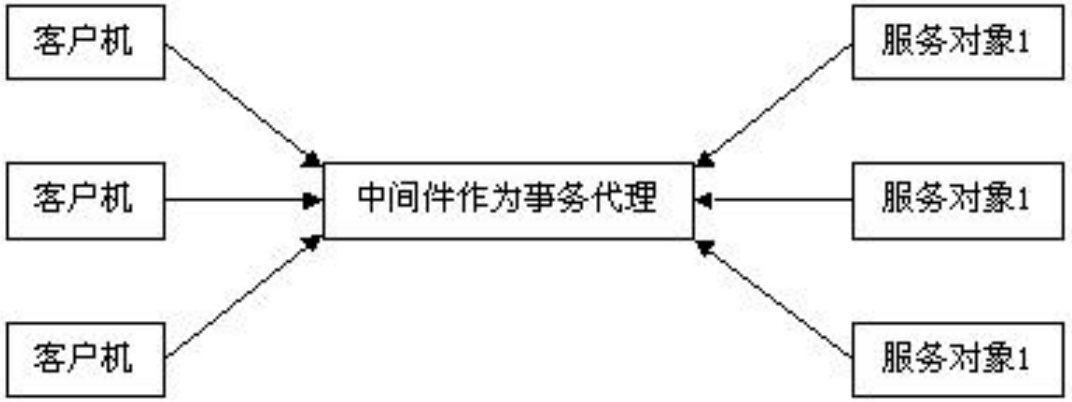


图3-17 引入中间件后客户机与服务器之间的关系

3.5.2 CORBA风格分析

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。

3.5.2 CORBA风格分析

- 由于CORBA使用了对象模型，将CORBA系统中所有的应用，看成是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）软件体系结构由层和线索中的构件，构成。

- 层是由一组具有相同抽象层次的构件构成。
- 线索是子系统的特例，它由不同抽象层次的构件组成（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，与其它线索的实现，无关或关联很少。

3.6 正交软件体系结构

- 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，利用功能的正交相关性，垂直分割为若干个线索（子系统）。
  - 每个线索由多个具有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。

3.6 正交软件体系结构

- **正交软件体系结构的主要特征如下：**

① 系统具有m（m > 1）个不同抽象级别的层。

② 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。

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）。

3.6 正交软件体系结构

- 大型复杂的软件系统，是一个多级的正交结构。
- 正交软件体系结构的框架如图3-18所示。
- 图3-18是一个五层结构的正交软件体系结构框架图，在该图中，ABDGK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，第五层是一个物理数据库连接构件或设备构件，供整个系统公用。

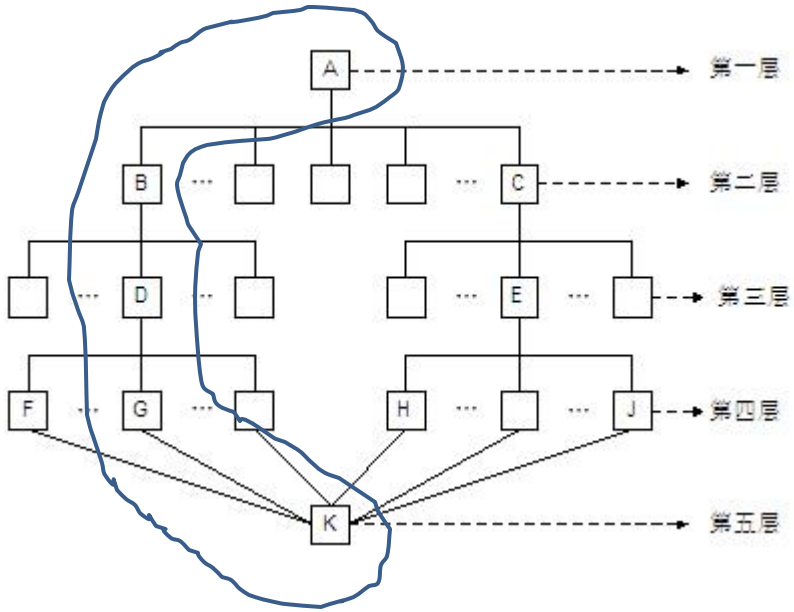


图3-18 正交软件体系结构框架

3.6 正交软件体系结构

- 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，而不会涉及到其它线索。
- 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。

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：
  - 某些部门的功能可能需要扩充，改变，或取消，或转入另外的部门，或其工作内容发生变更；
  - 某些的部门可能被撤消，其功能被整个并入其它部门，或分解并入数个部门；

3.6.3 正交软件体系结构的实例

- 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。

3.6.3 正交软件体系结构的实例

- 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。

3.6.3 正交软件体系结构的实例

- 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）。

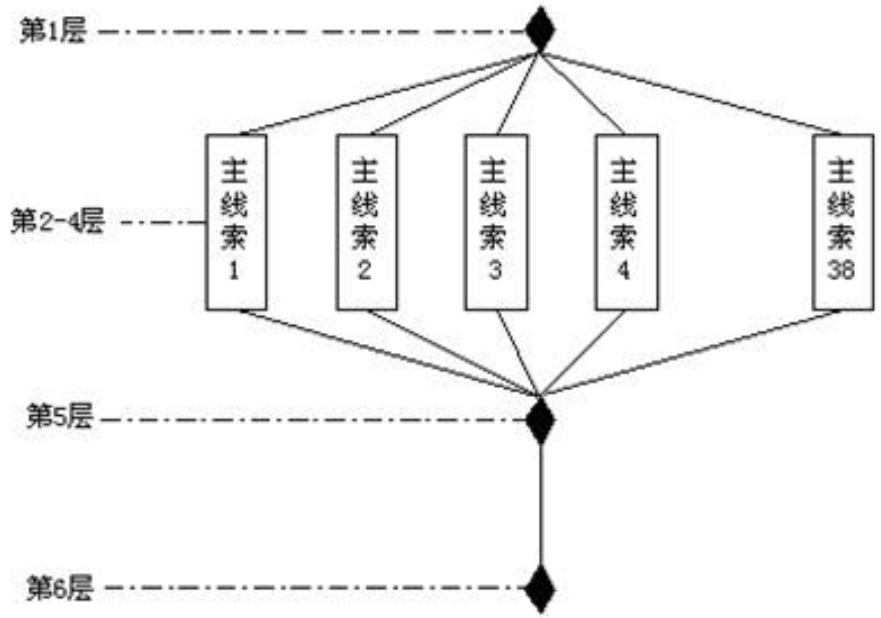


图3-19 系统总体正交体系结构设计

3.6.3 正交软件体系结构的实例

- 为了简单起见，下面仅就其中的一个主线索，进行说明；
  - 其它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。

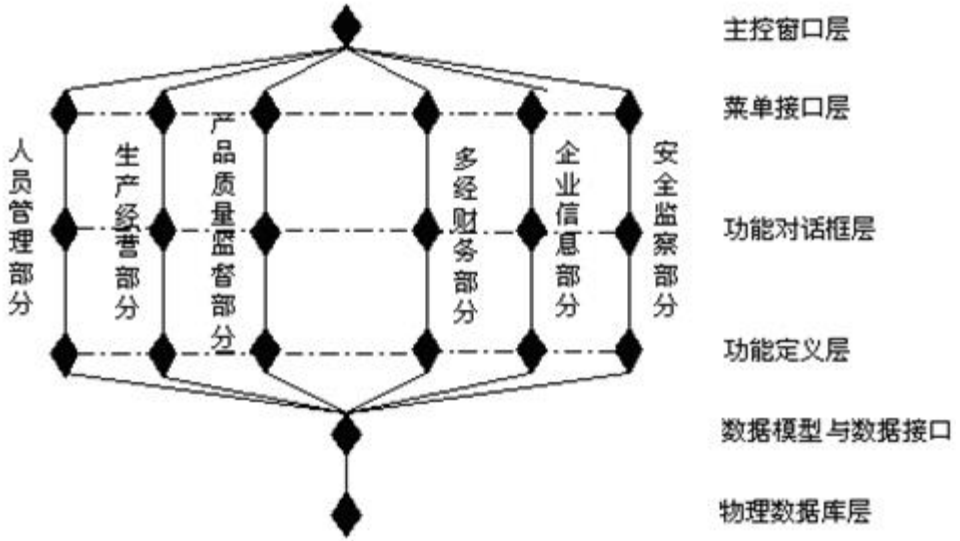


图3-20 多种经营主线索子正交结构图

3.6.3 正交软件体系结构的实例

- 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。

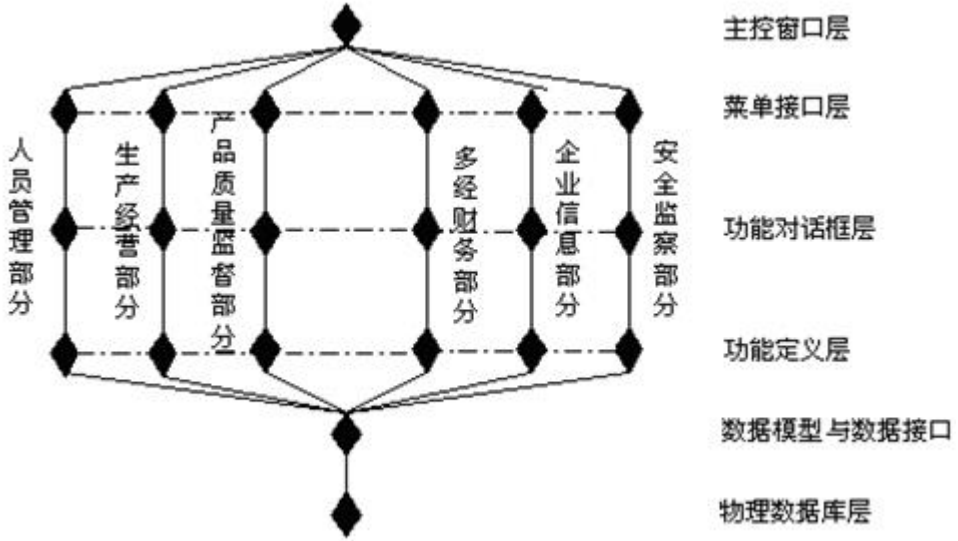


图3-20 多种经营主线索子正交结构图

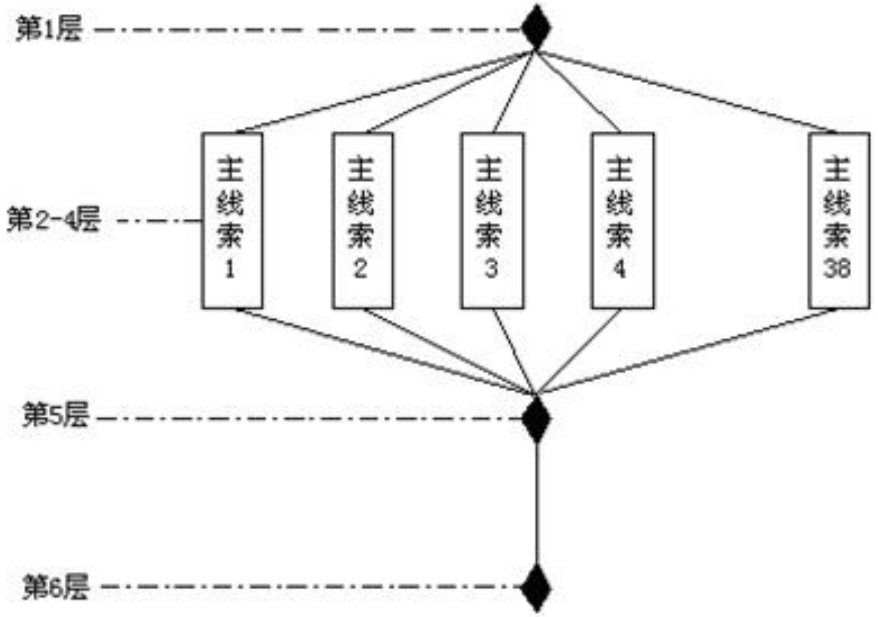


图3-19 系统总体正交体系结构设计

3.6.3 正交软件体系结构的实例

- 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 第一层实现主控窗口，由主控窗口对象，控制引发所有线索运行。
  - ② 第二层实现菜单接口，支持用户，选择不同的处理功能。
  - ③ 第三层涵盖了所有的功能对话框，这也就是与功能的真正接口。
  - ④ 第四层是真正的功能定义，在这一定义的构件有：数据录入构件（包括插入、删除、更新）、报表处理构件、快速查询构件、图形分析构件、报表打印构件等等。
  - ⑤ 第五层和第六层是数据服务的实现，第五层是包括了特定的数据模型和数据库接口。
  - ⑥ 第六层就是数据库本身。

3.6.3 正交软件体系结构的实例

➢ 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。

3.6.3 正交软件体系结构的实例

- 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。
  - 按照前面提出的设计思想和方法，首先将提出的新功能要求映射到原设计结构上，这里仍以“多种经营管理主线索”为例，总结出以下变动：

- （1）报表和报表处理功能的变动：
  - 例如财务管理子线索有如下变动，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）。
  - 其它一些子功能（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。

3.6.3 正交软件体系结构的实例

- （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。

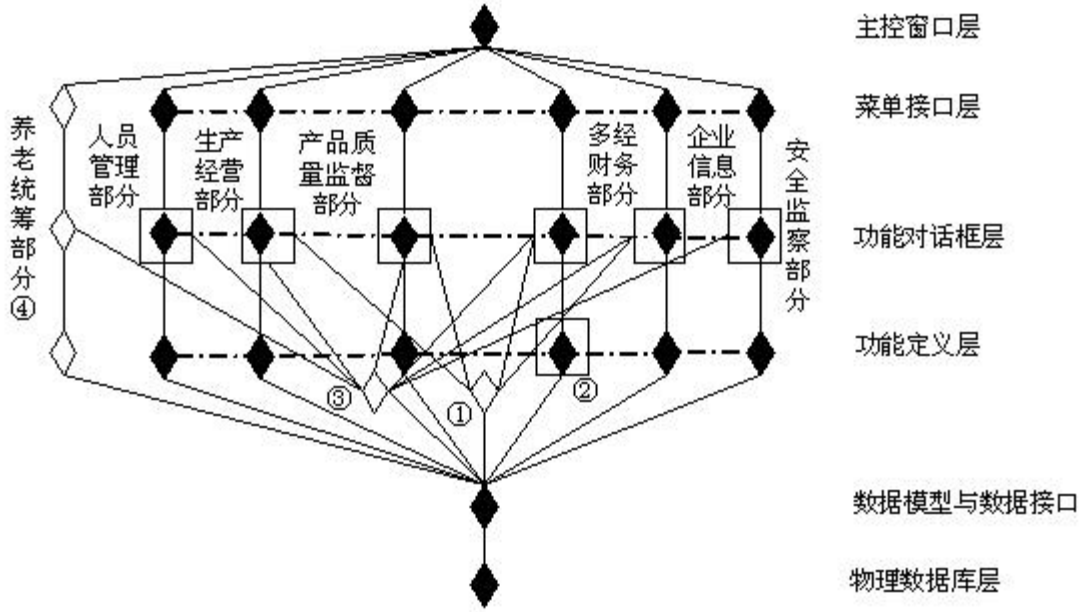


图3-21 多经主线索结构变动情况图

3.6.3 正交软件体系结构的实例

- （2）子线索的变动：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的进行。

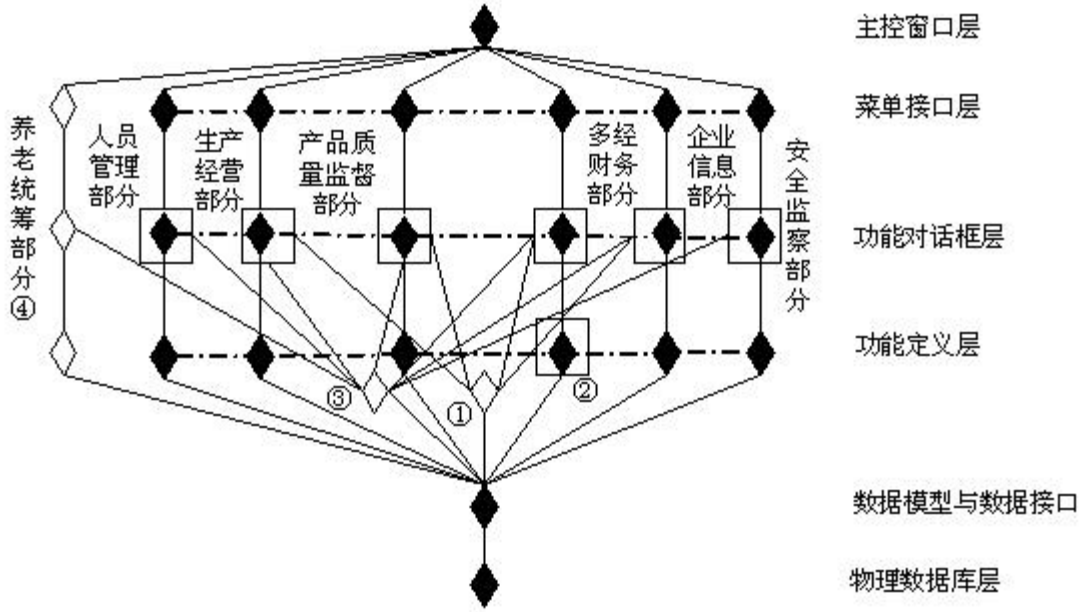


图3-21 多经主线索结构变动情况图

3.6.3 正交软件体系结构的实例

- 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。新增子线索一条。
- 由于涉及到添加公用共享构件，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。

3.6.3 正交软件体系结构的实例

- 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8小时，单位人/天）的统计分析。

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%

3.6.3 正交软件体系结构的实例

- 38个主线索的修改与开发工作量比例均未超过13%，比传统方法工作量减少20%左右。
- 可见多级正交结构，对于降低软件演化更新的开销，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。

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：**
  - **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。**
  - **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。**
  - **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HMB（Hierarchy Message Bus）体系结构风格提出的实际背景：**

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普遍的应用需求。**

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。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。**

3.7 基于层次消息总线的体系结构风格

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。

3.7 基于层次消息总线的体系结构风格

- 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讯，如图3-22所示。
- 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。

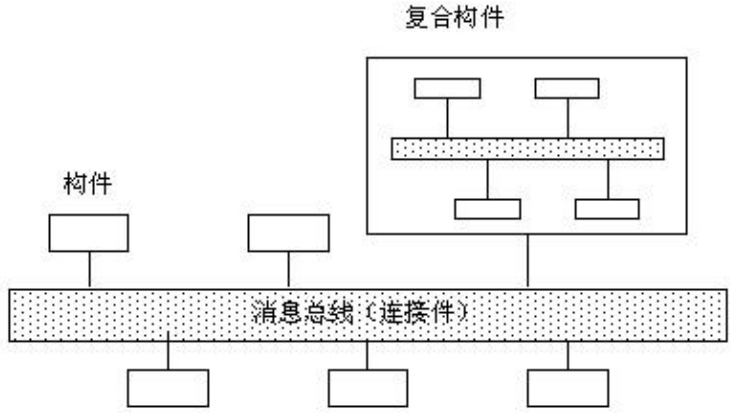


图3-22 HMB风格的系统示意图

- 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，消息是构件之间通讯的唯一方式。
- 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。

3.7 基于层次消息总线的体系结构风格

- 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。

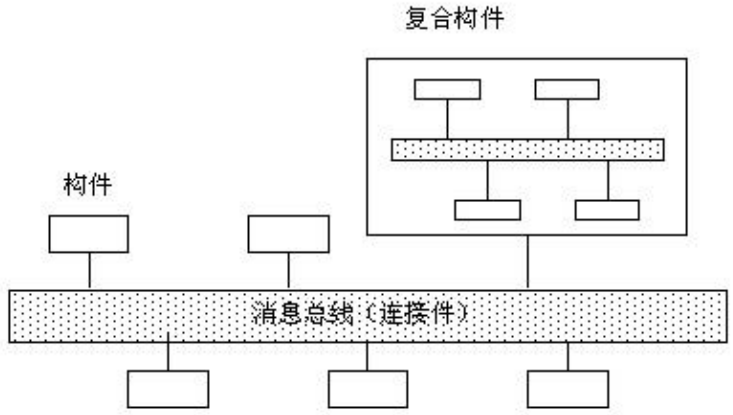


图3-22 HMB风格的系统示意图

3.7 基于层次消息总线的体系结构风格

- 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。

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
  - 功能等

3.7.1 构件模型

- 为满足体系结构设计的需要，HMB风格的构件模型，如图3-23所示，包括：
  - 接口
  - 静态结构
  - 动态行为三个部分
- 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。

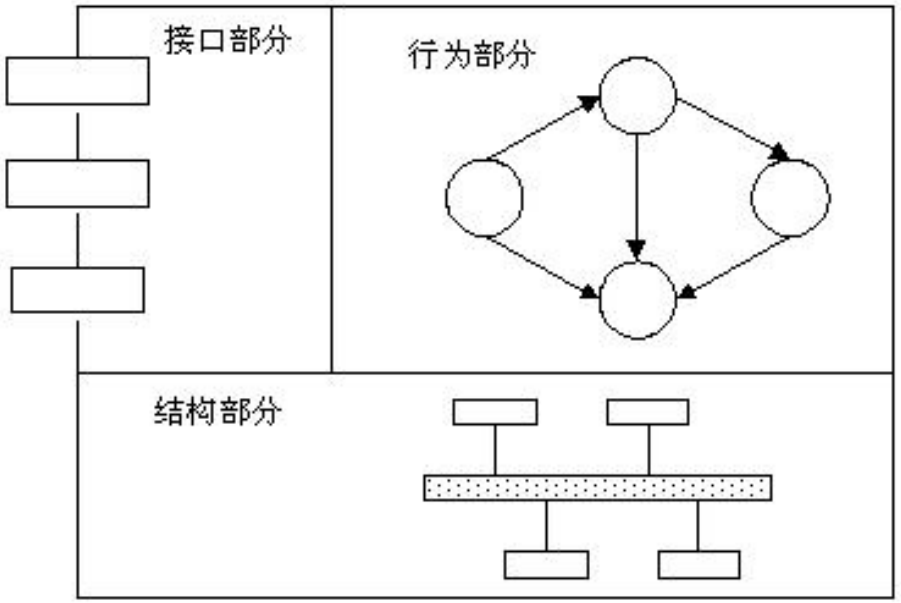


图3-23 HMB风格的构件模型

3.7.1 构件模型

- 在图3-23中，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。

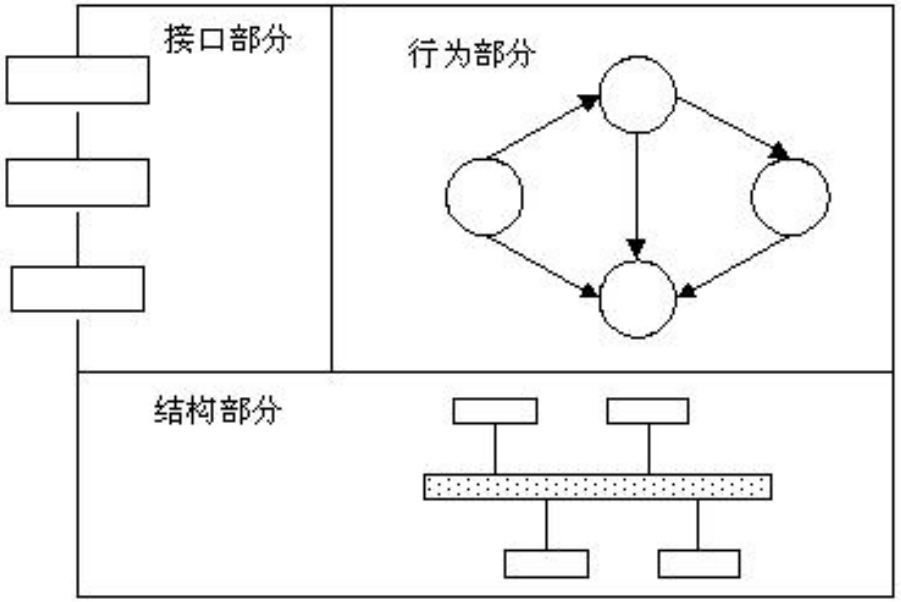


图3-23 HMB风格的构件模型

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。

3.7.2 构件接口

- 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讯，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。

3.7.2 构件接口

- 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对此消息感兴趣的构件。
- 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：
  - 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。
  - 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。

3.7.2 构件接口

- 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后，才可以继续运行的消息类型。常见的同步消息包括过程调用。
  - 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。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。

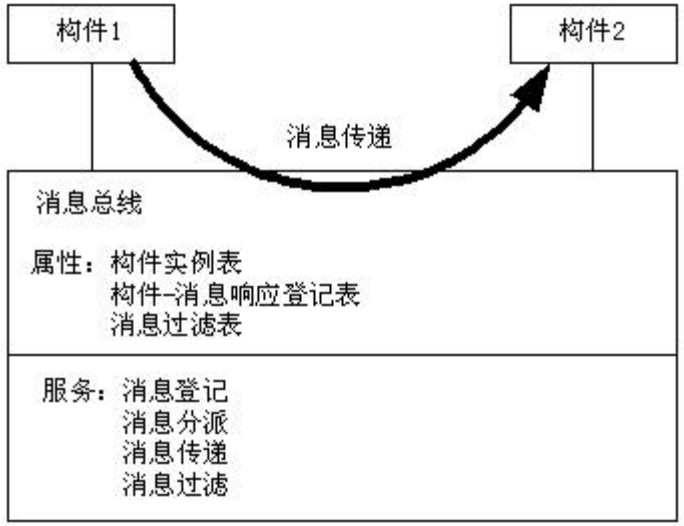


图3-24 消息总线的结构

3.7.3 消息总线

➢ 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。

3.7.3 消息总线

- 1、消息登记
- 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，缺省情况下，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。

3.7.3 消息总线

➢ 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。

3.7.3 消息总线

- 2、消息分派和传递
- **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讯，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。**

3.7.3 消息总线

➢ 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。
- 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。

3.7.3 消息总线

- 3、消息过滤
- 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符合客户/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，这样无需对构件进行修改，就解决了这两类构件集成问题。
- 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。

3.7.4 构件静态结构

- 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，把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。

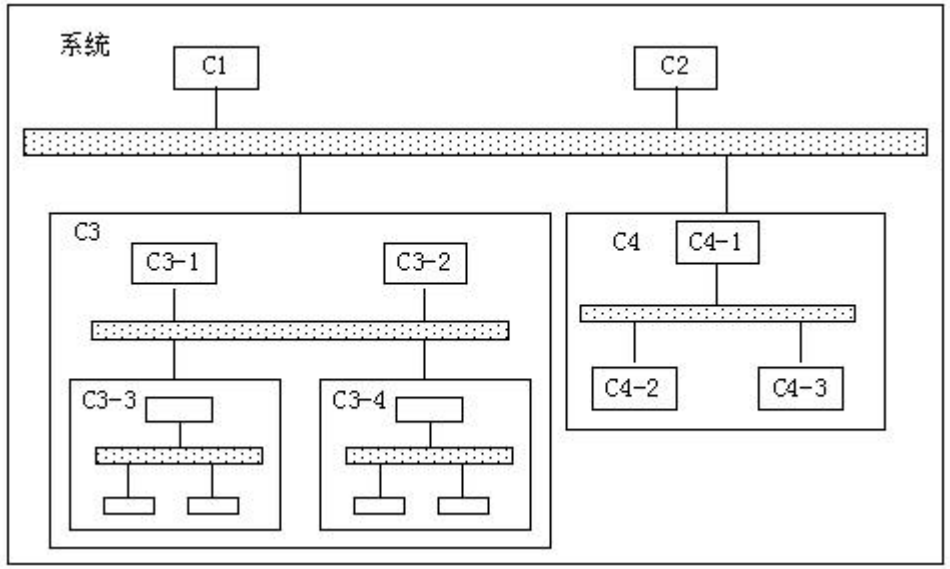


图3-25 系统/复合构件的静态结构示意图

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。

3.7.5 构件动态行为

- 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。

3.7.6 运行时刻的系统演化

- 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讯，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。
  - 把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。

3.7.6 运行时刻的系统演化

- 系统中可能增删改构件的情况包括：
  - 当系统功能需要扩充时，往系统中增加新的构件。
  - 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。
  - 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。

3.7.6 运行时刻的系统演化

➢ 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

- 一些所谓的“纯”体系结构，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。

3.8 异构结构风格

- 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、数据确认和跟踪等。
- 如果能用标准的构件，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，人们很有可能发现：把它们融合组织在一起也不是一件简单的事情。
- 通常问题出在：各构件已经作了关于数据表示、系统组织、通信协议等方面不同的假设。

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。
- 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。

3.8.1 异构结构的实例分析

- 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。

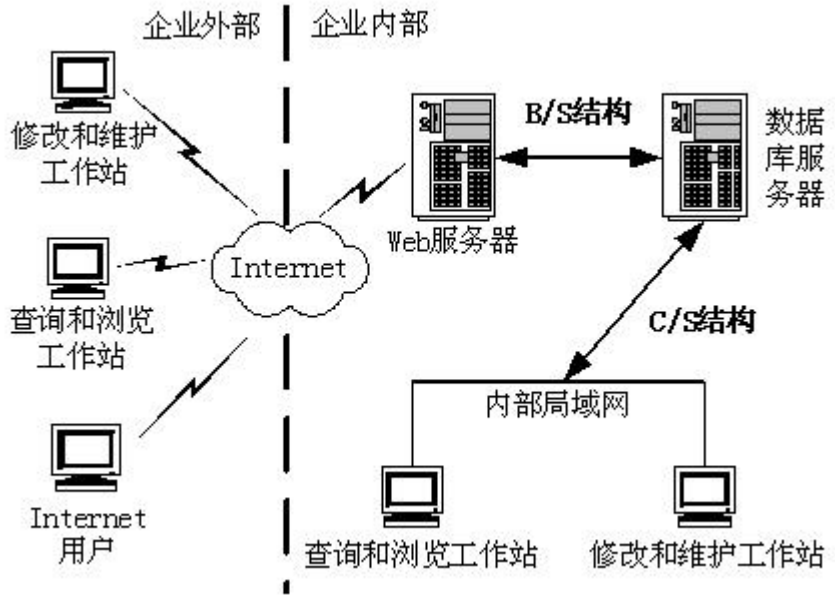


图3-26“内外有别”模型

3.8.1 异构结构的实例分析

- “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。

3.8.1 异构结构的实例分析

- 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。
- “查改有别”模型充分利用了B/S体系结构和C/S体系结构各自不同的优点。
- 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。

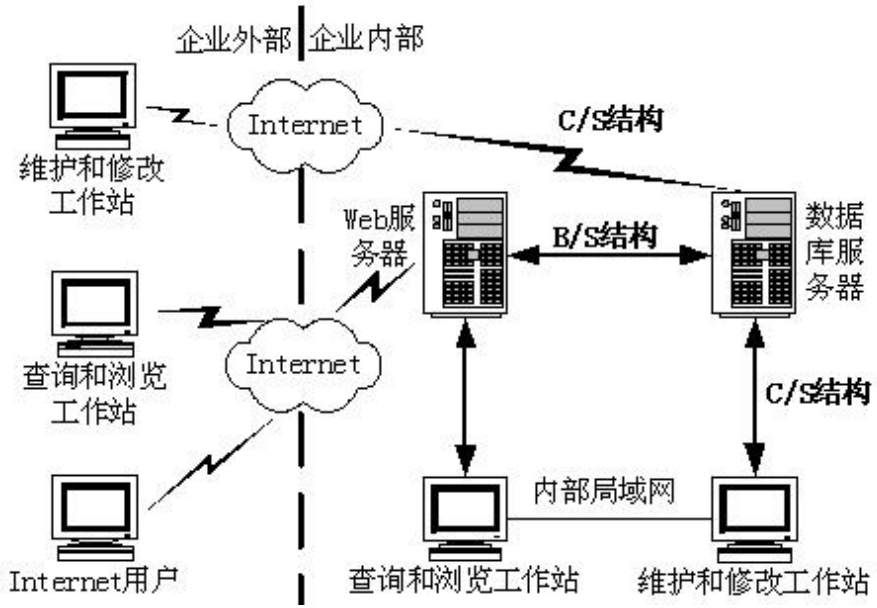


图3-27“查改有别”模型

3.8.1 异构结构的实例分析

➢ 3、几点说明

- 因为本节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。
- 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。
- 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。

3.8.1 异构结构的实例分析

➢ 4、应用实例

- 当前，我国电力系统正在进行精简机构的改革，变电站也在朝无人、少人的方向发展。
- “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，可以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，开发一套完整的变电综合信息管理系统。

3.8.1 异构结构的实例分析

- （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。
- 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。

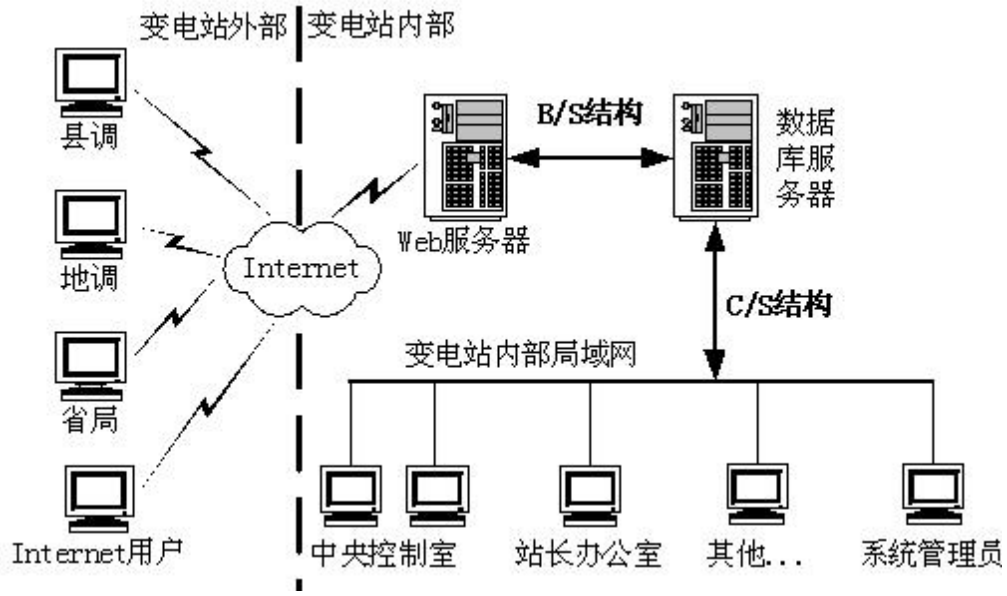


图3-28 TSMIS系统软件体系结构

3.8.1 异构结构的实例分析

- （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、技术台帐管理、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。

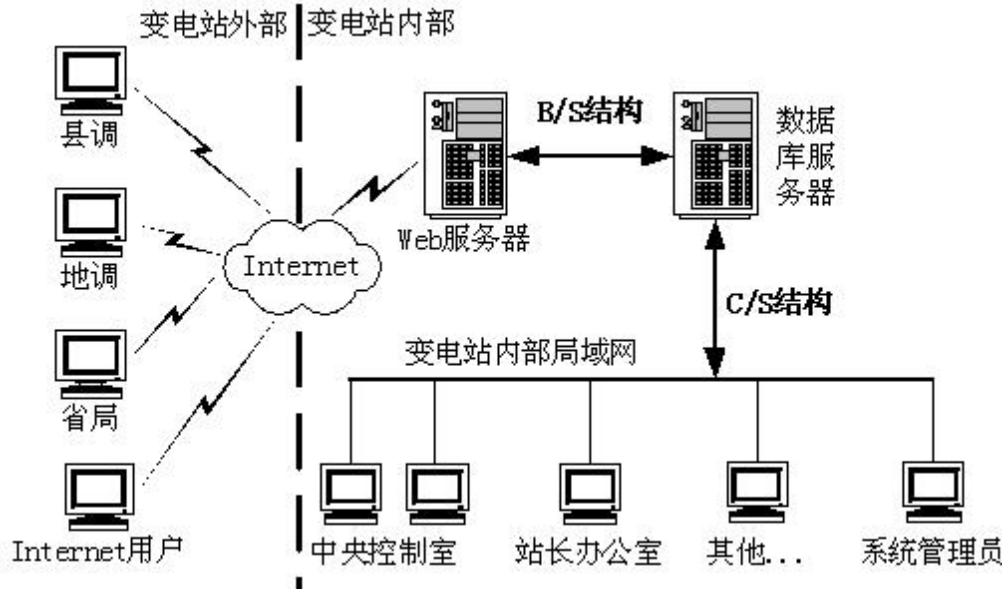


图3-28 TSMIS系统软件体系结构

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。
- 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、控制等方面不同的假设，把这些方面统称为形式（form）。

图3-29 构件协调问题

3.8.2 异构组合匹配问题

- 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 把A的形式，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。可以使用射影或视图，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。

3.8.2 异构组合匹配问题

- 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。

④ 通过协商，达成一个统一的形式。

⑤ 使A成为支持多种形式。

⑥ 为A提供进口/出口转换器。它们有两种方式：

- ✓ 其一，用独立的应用程序，提供表示转换服务。
- ✓ 其二，用某些系统，去协调扩充或外部插件，以完成内外部数据格式之间的相互转换。

3.8.2 异构组合匹配问题

- 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。

⑦ 引入中间形式

- ✓ 第一，引入外部相互交换的表示。有时通过接口描述语言IDL支持，能够提供一个中介层。这对于要把多种形式互不相同的构件，结合在一起的时候，特别有用。
- ✓ 第二，采用标准的发布形式。

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。

- 以上这些技术有它们各自不同的优点和缺点，。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

- 互连系统构成的系统SIS（System of Interconnected Systems）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，互连系统之间相互通信，共同履行系统的职责。
- 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）。
- 上级系统独立于其从属系统。
- 从属系统是子系统。每个从属系统是上级系统模型中所指定内容的一个实现。

3.9.1 互连系统构成的系统

- 互连系统构成的系统的软件体系结构SASIS（Software Architecture for SIS）如图3-30所示。
- 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。

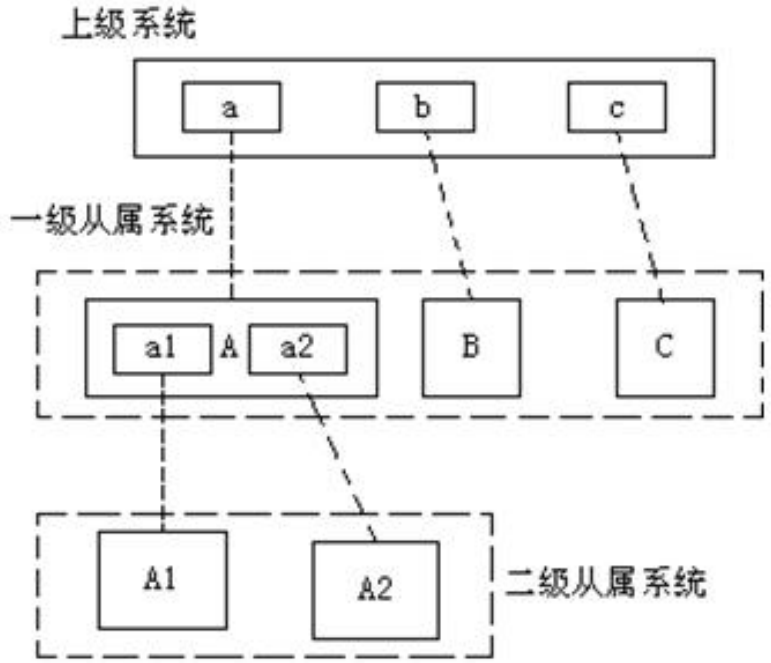


图3-30 SIS的体系结构

3.9.1 互连系统构成的系统

- 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。
- 从属系统自成一个软件系统，可以
  - 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它系统，就很容易使用从属系统，来实现其上级系统。

3.9.1 互连系统构成的系统

- 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。

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。
- 一旦上级系统经历过至少一次迭代，并且从属系统的接口相对稳定，那么从属系统的软件过程就可以开始。
- 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。

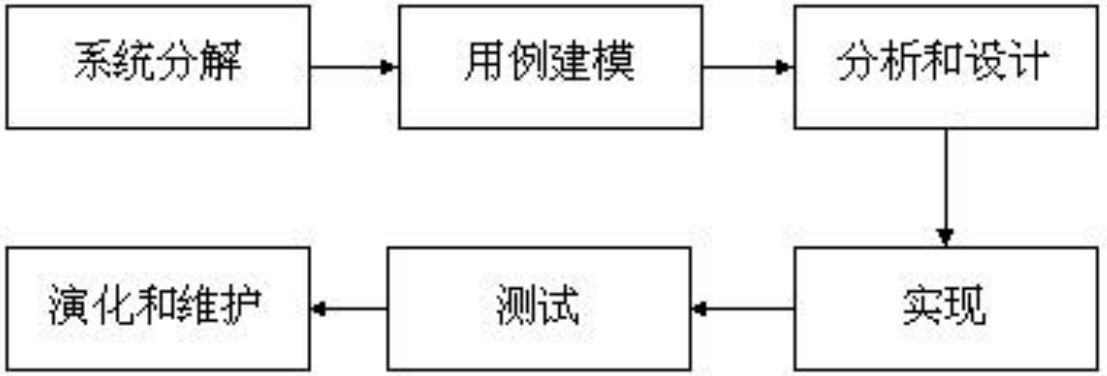


图3-31 基于SASIS的软件过程

3.9.2 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。

3.9.2 基于SASIS的软件过程

- 1、系统分解
- 一般根据系统的规模和复杂性，来作决定将系统分解为由互连系统构成的系统。
- 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。

3.9.2 基于SASIS的软件过程

- 1、系统分解
- 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。
  - 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。

3.9.2 基于SASIS的软件过程

- 1、系统分解
- 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：
  - 适当关注从属系统之间的重用和经验共享。
  - 对于开发什么制品(构件或文档)？从属系统制品与上级系统制品之间有什么关系？等问题有清晰的理解。
  - 定义一个有效的变更管理策略，并得到所有团队的遵守。

3.9.2 基于SASIS的软件过程

- 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模型。
- 上级系统的高级用例（SIS系统级的用例），通常可以分解到子系统上。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。
- 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）。

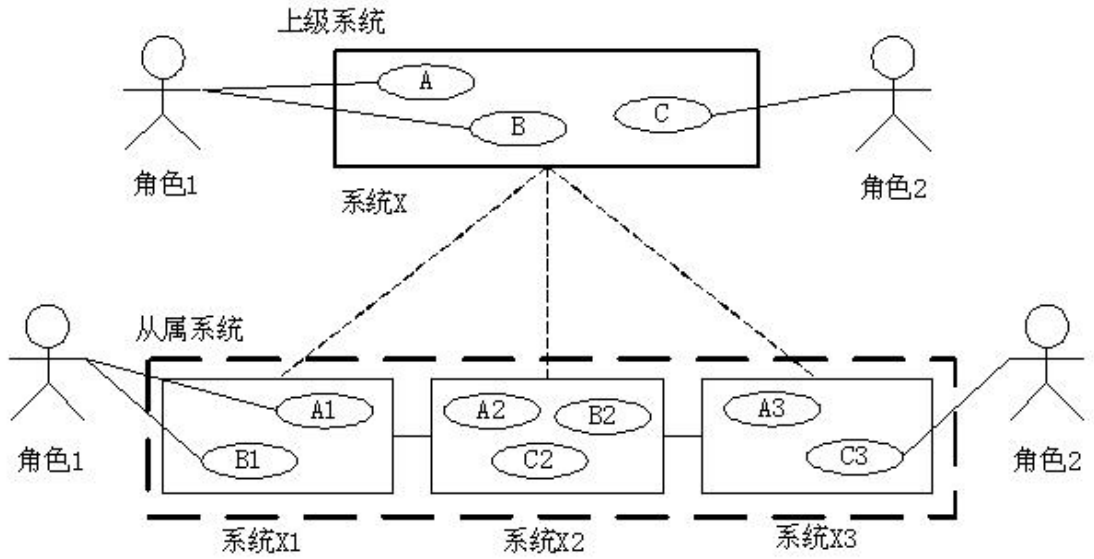


图3-32 上级系统的高级用例与从属系统的详细用例之间的关系

3.9.2 基于SASIS的软件过程

- 2、用例建模
- 从任何一个从属系统的角度来看，其它从属系统都是该从属系统用例模型的执行者，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 都被视为执行者。

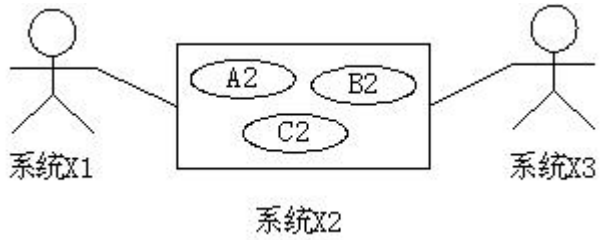


图3-33 一个从属系统是另一个从属系统的执行者

3.9.2 基于SASIS的软件过程

- 3、分析和设计
- 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。
- 分析设计的过程可分为5个子过程：
  - 标识构件
  - 选择体系结构风格
  - 映射构件
  - 分析构件相互作用
  - 产生体系结构

3.9.2 基于SASIS的软件过程

- 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
  - 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

3.9.2 基于SASIS的软件过程

- 在决定SIS系统的体系结构时，还要综合考虑两个问题:
  - 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。
  - 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。

3.9.2 基于SASIS的软件过程

- **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，只实现一些组合工作。（注：上级系统主要通过互连方式实现）
- **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。

3.9.2 基于SASIS的软件过程

- 6、演化和维护
  - 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

- **应用场景1：把互连系统构成的系统的体系结构及其建模技术，用于以下的多种系统开发，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**

3.9.3 应用范围

- 应用场景2：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，就可以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。

3.9.3 应用范围

➢ 1、实例1：电话网络

- 电话网络可能是世界上最大的由互连系统构成的系统。
- 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。
- 这里用移动电话网络 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。

3.9.3 应用范围

- 1、实例1：电话网络
- **大规模系统的功能通常包含几个业务领域。**
  - **例如，GSM 标准包括了从呼叫用户，到被呼叫用户的整个系统。它既包括移动电话的行为，也包括网络节点的行为。**
- **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，向话务公司销售网络节点。**
    - ✓ **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。**

3.9.3 应用范围

➢ 2、实例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。
- 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。

3.9.3 应用范围

➢ 3、实例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败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）。
  - ✓ 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，但需要去满足新的变化了的业务需求的系统。

3.9.3 应用范围

- 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。

② 系统在性能上已经落后，采用的技术已经过时。

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。

3.9.3 应用范围

- 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：
  - 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。
  - 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。
  - 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

- DSSA就是在一个特定应用领域中，为一组应用，提供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”。

3.10.1 DSSA的定义

- 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。

3.10.1 DSSA的定义

- 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：
  - 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），把它在某领域中作为系统的可行解决方案的一个通用软件体系结构。
  - 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。但无法为某个系统，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。

3.10.1 DSSA的基本活动

- 实施DSSA的过程中包含了一些基本的活动。
- 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。

3.10.2 DSSA的基本活动

- 1、领域分析
- 这个阶段的主要目标是获得领域模型。
- 领域模型描述领域中所有系统的共同的需求。领域模型所描述的需求，被称为领域需求。
- 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。

3.10.2 DSSA的基本活动

- 1、领域分析
- 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。

3.10.2 DSSA的基本活动

- 1、领域分析
- 领域分析的机制如图3-34所示。

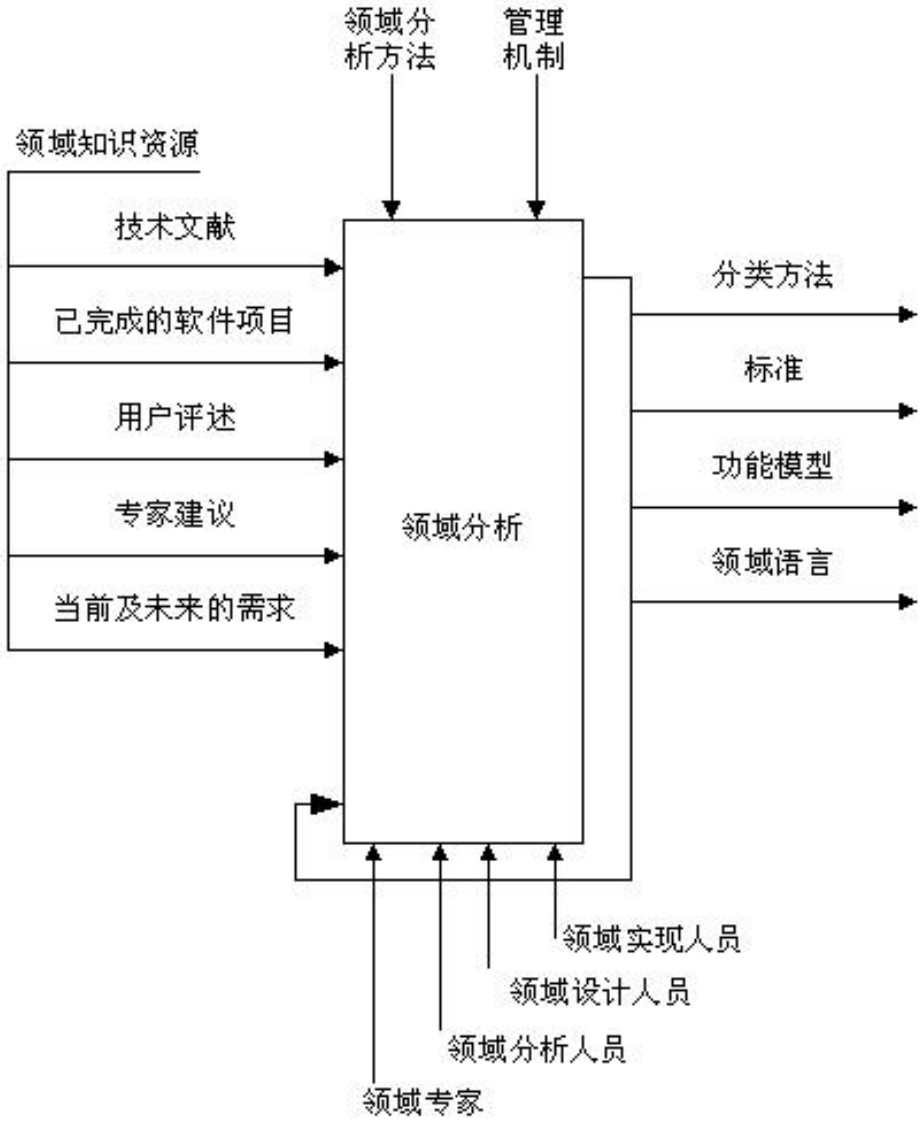


图3-34 领域分析机制

3.10.2 DSSA的基本活动

- 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。

3.10.2 DSSA的基本活动

➢ 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。

3.10.2 DSSA的基本活动

- 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
  - 领域分析师
  - 领域设计人员
  - 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?

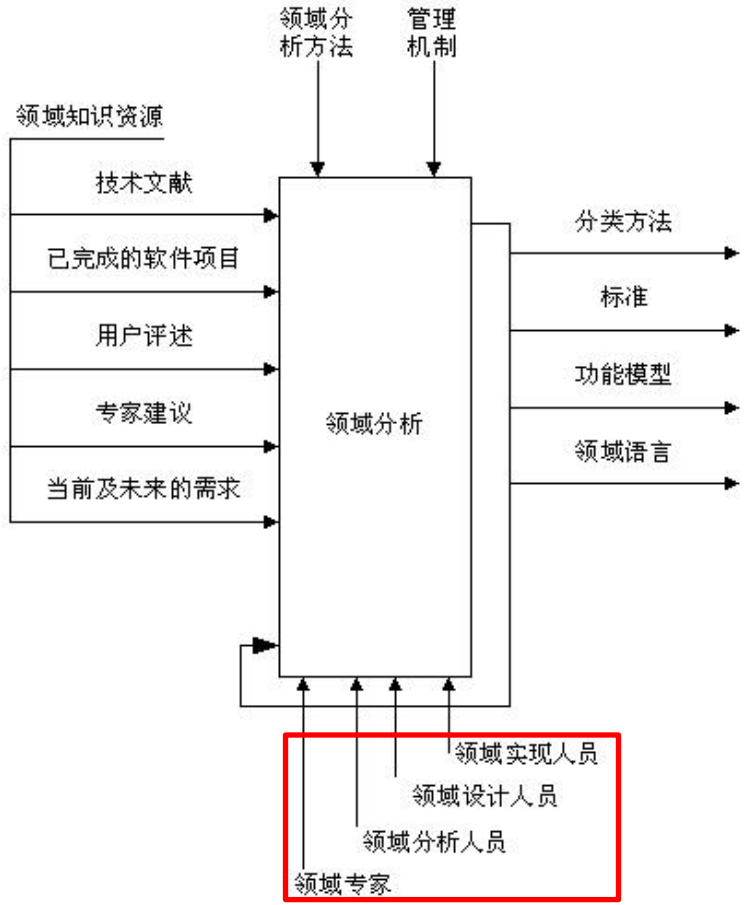


图3-34 领域分析机制

3.10.3 参与DSSA的人员

➢ 1、领域专家

- 领域专家可能包括
  - ✓ 在该领域中具有使用系统的工作经验的用户
  - ✓ 在该领域中具有从事系统的需求分析、设计、实现以及项目管理等工作经验的软件工程师等。

3.10.3 参与DSSA的人员

- 1、领域专家
- 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。

3.10.3 参与DSSA的人员

- 2、领域分析人员
- 领域分析人员应由具有业务知识和技术知识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。

3.10.3 参与DSSA的人员

- 2、领域分析人员

- **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。**

3.10.3 参与DSSA的人员

➢ 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。

3.10.3 参与DSSA的人员

- 3、领域设计人员

- **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。**

3.10.3 参与DSSA的人员

➢ 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。

3.10.3 参与DSSA的人员

- 4、领域实现人员
- **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。

3.10.4 DSSA的建立过程

- DSSA的建立过程分为五个阶段，每个阶段可以，进一步划分为一些步骤或子阶段。
- 每个阶段都需要有：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。
- 这个过程是并发的、递归的、迭代的。或者可以说，它是螺旋型。
- 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。

3.10.4 DSSA的建立过程

- DSSA建立过程的5个阶段
- （1）定义领域范围
  - 本阶段的重点是确定哪些东西在感兴趣的领域中，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
  - 本阶段的目标是编辑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模块图，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。

3.10.4 DSSA的建立过程

- DSSA建立过程的5个阶段
- （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。
    - ✓ 识别出约束。
    - ✓ 记录约束对设计和实现决定，造成的后果。
    - ✓ 记录对处理这些问题时，产生的所有问题的讨论。

3.10.4 DSSA的建立过程

- DSSA建立过程的5个阶段
- （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，使得它们可以被用来产生问题域中的新应用。

3.10.4 DSSA的建立过程

- DSSA的建立过程是并发的、递归的和迭代进行的。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。
- 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。

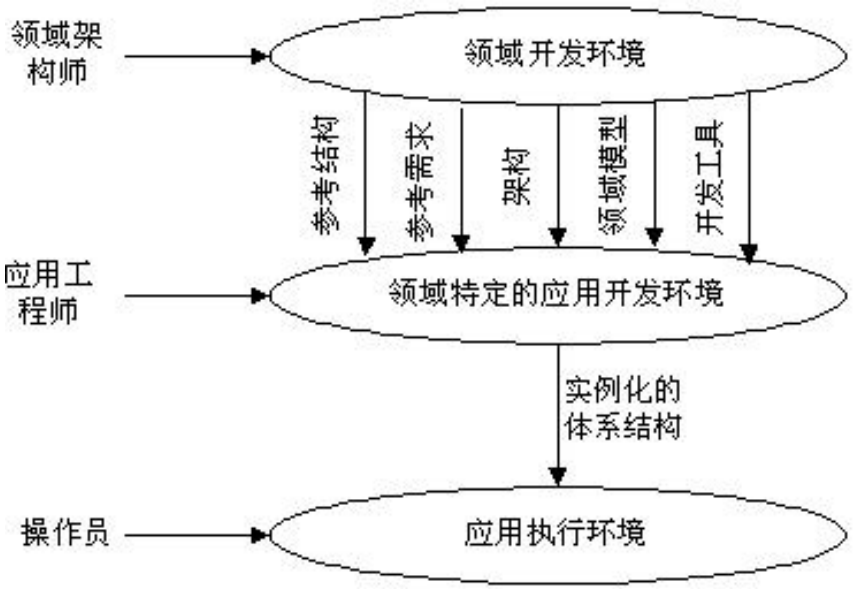


图3-35 DSSA的三层次的系统模型

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。在中间的环节，引入了DSSA的概念。

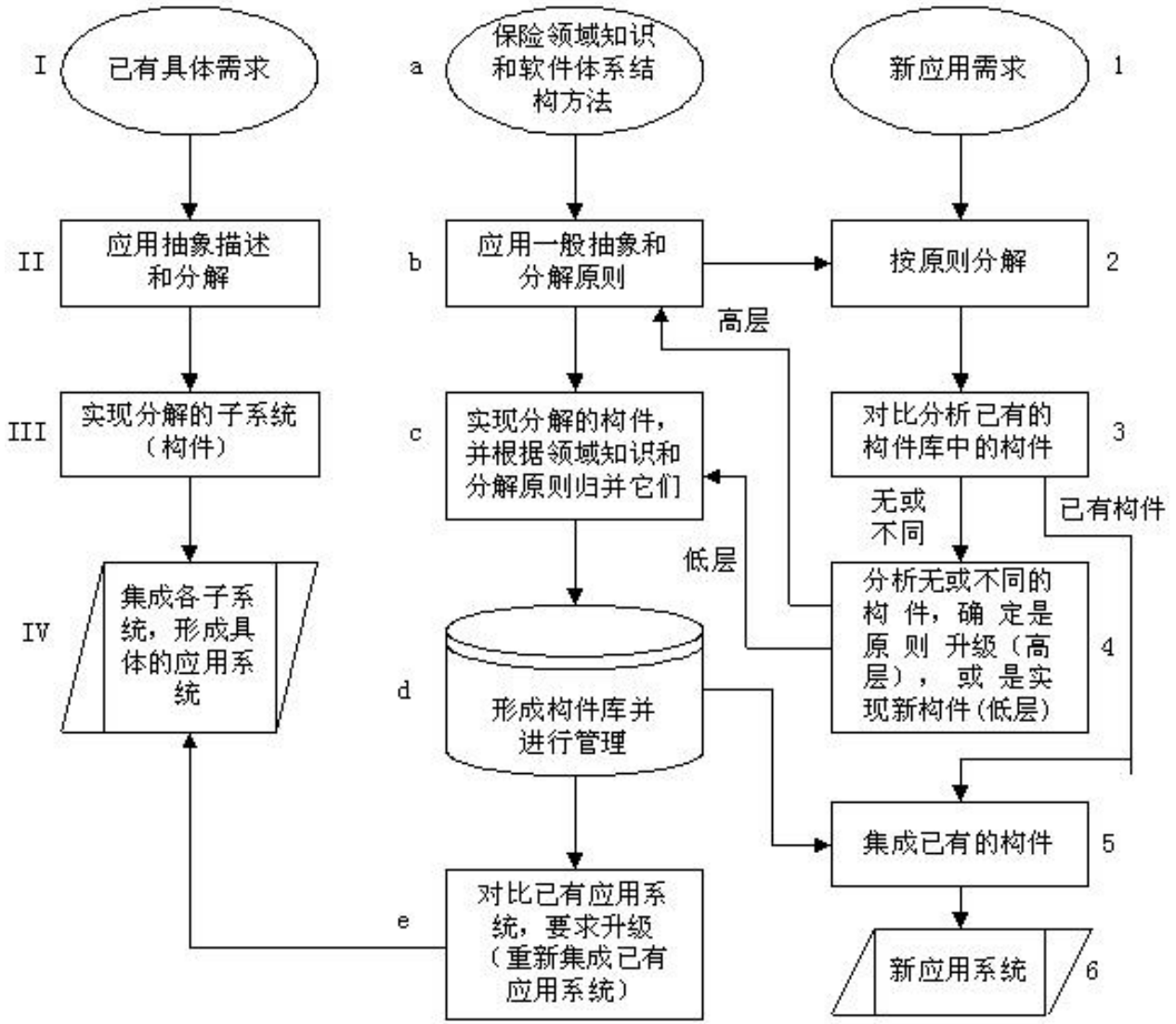


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 图3-36中分左、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，3个纵向也分别反映应用开发从高层体系结构，向低层次体系结构转化的过程。
- 而这种转化过程，正是保险领域特殊专业知识的应用和实现。

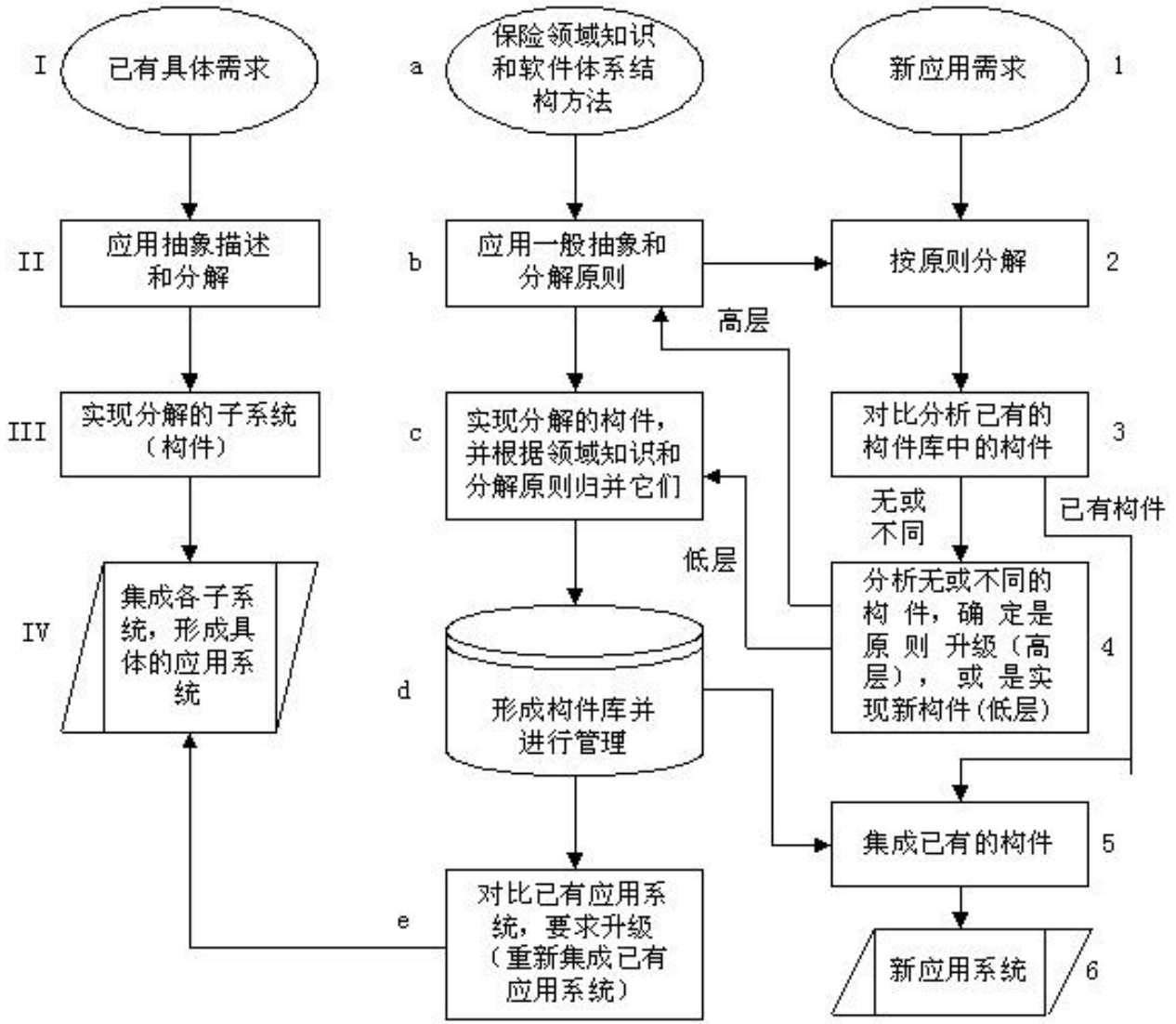


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 1、传统的构造保险应用方法
- 传统的应用构造方法，对应于保险行业DSSA结构图3-36的左列。
  - 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。

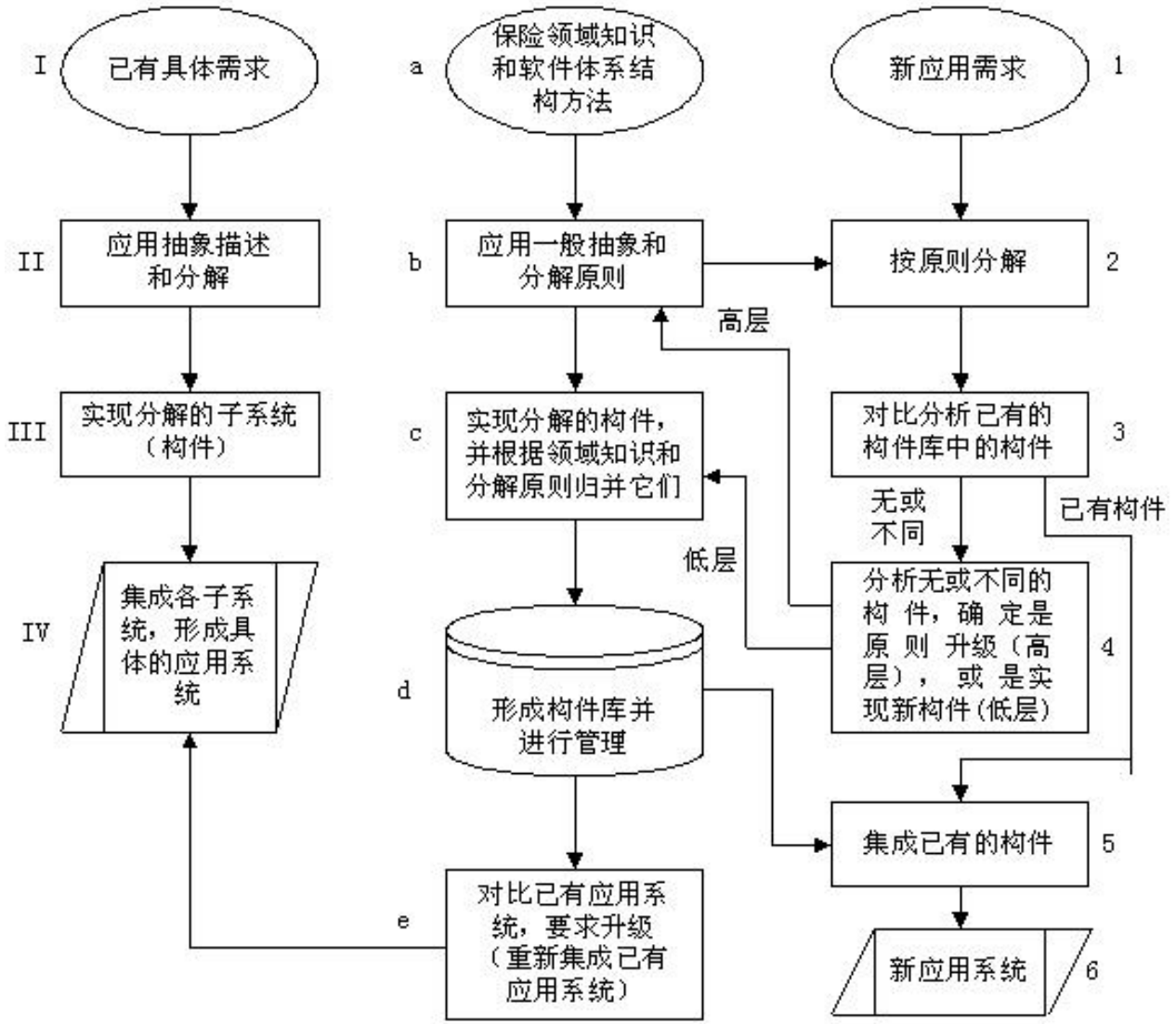


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 1、传统的构造保险应用方法
- 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。

3.10.5 DSSA实例

- 1、传统的构造保险应用方法
- 标号II的工作主要是确定险种的分类，以及分类后的过程管理。
  - 分类的依据，仍然是险种产品的特征。
  - 一个财产险业务管理系统，被分成20多个子系统。
- 标号III是子系统的实现
- 标号IV的内容主要是集成各子系统

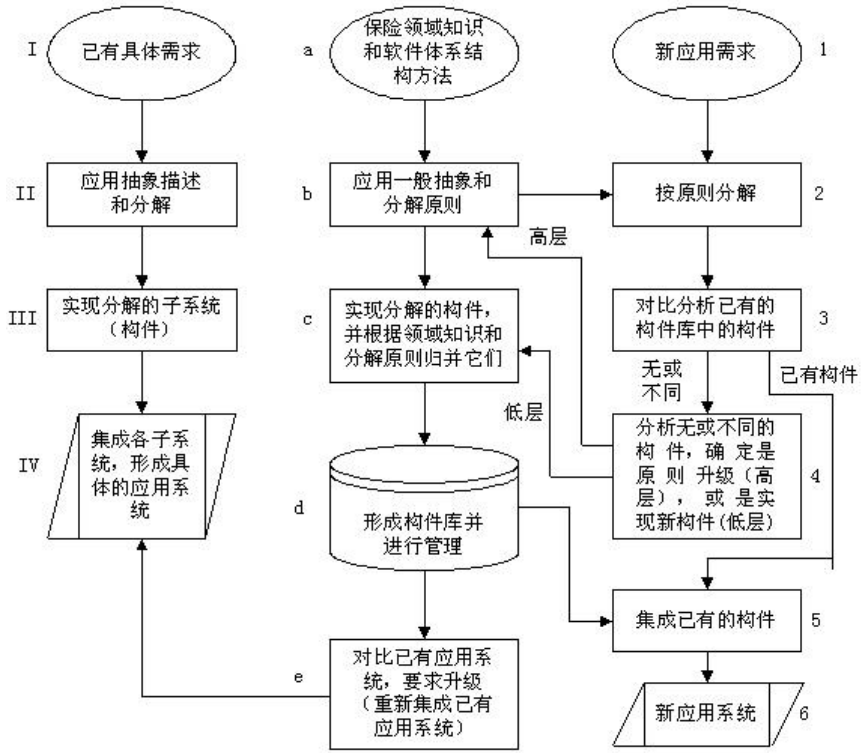


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 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反映的是：对领域工程一般原理的学习，以及对保险领域知识的深入理解。
- 结合标号b的内容，对保险行业内共同的特征进行提取。这是实现保险行业DSSA的关键步骤之一，是**形成构件分解原则和方法**的基础。

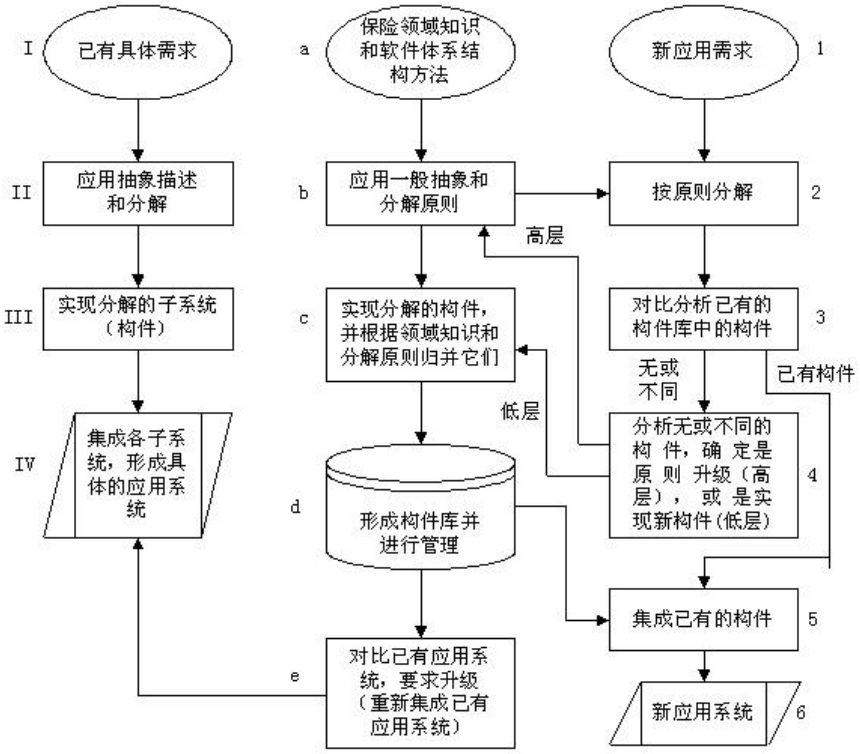


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 2、采用DSSA后的变化
- 在标号c中，主要的目标是实现标号b中所划分的构件。
- 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，在考虑领域知识的背景下，重新实现它们。

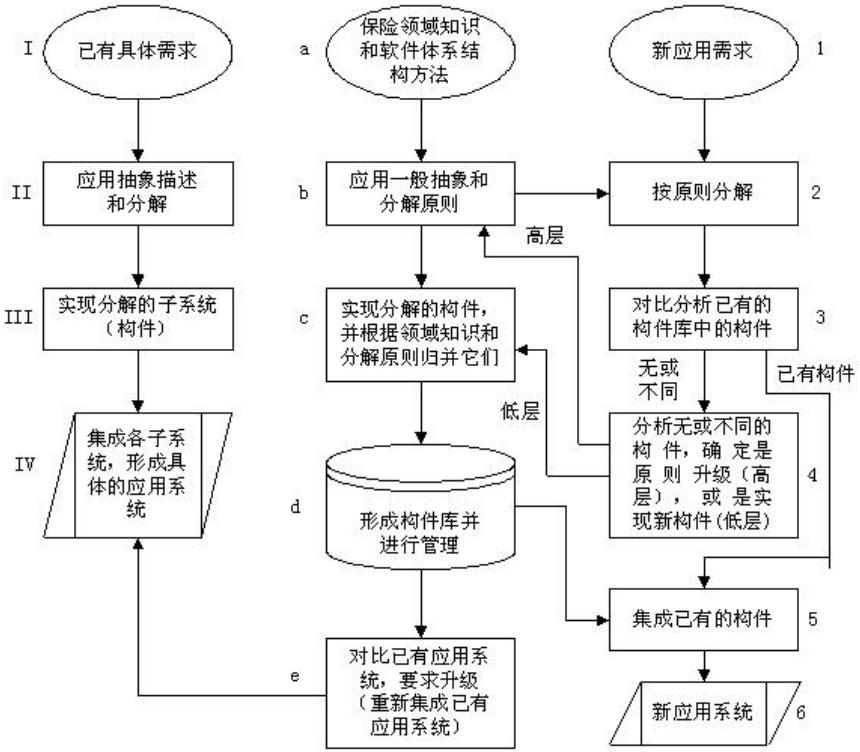


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

2、采用DSSA后的变化

- **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。**
  - **其次，构成实体构件的操作部分。**
  - **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。**

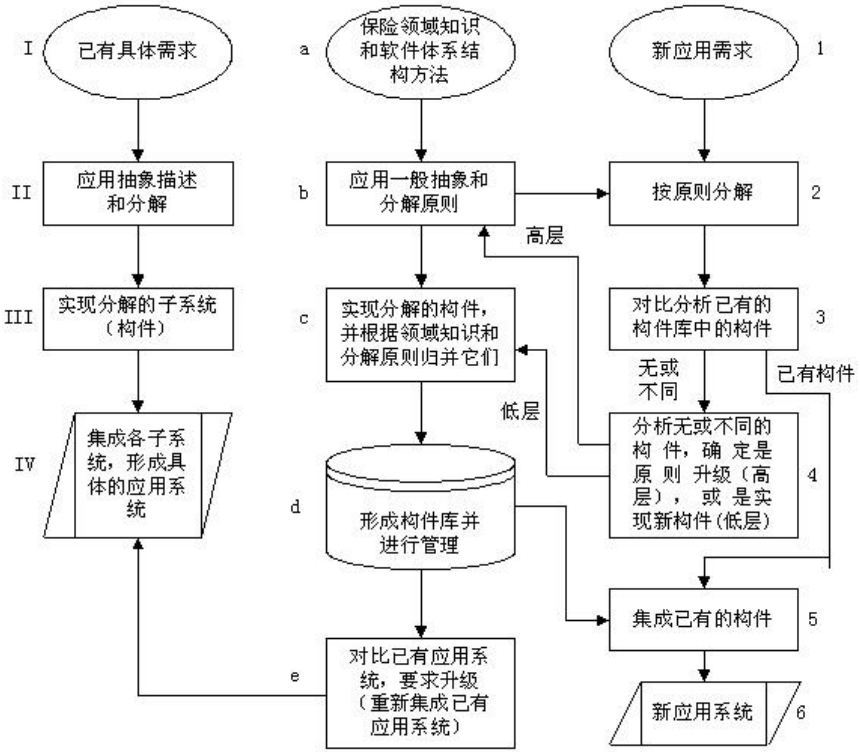


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 2、采用DSSA后的变化
- 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。
- 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。

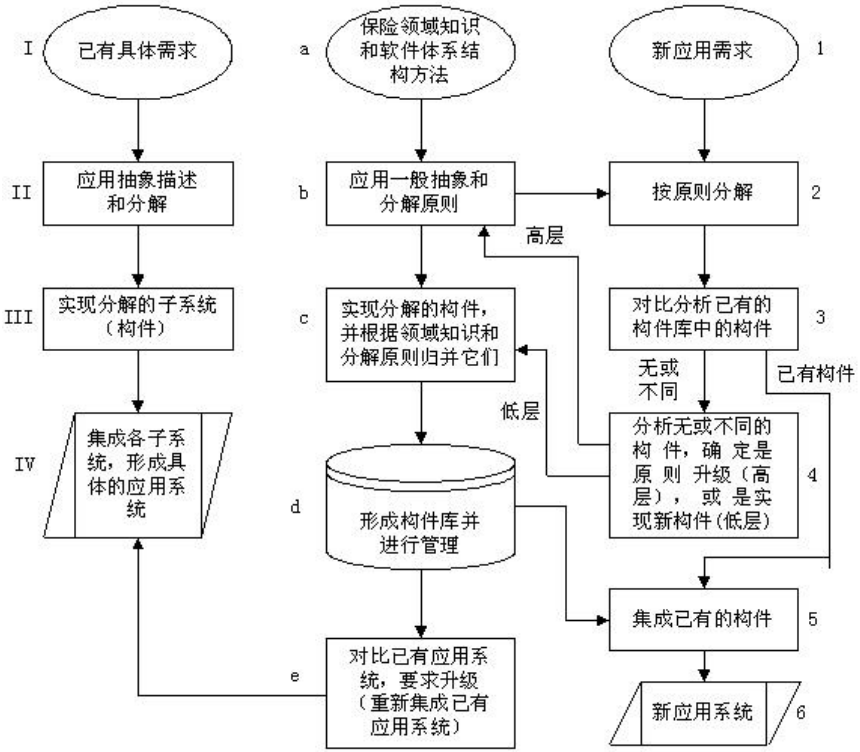


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

➢ 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。

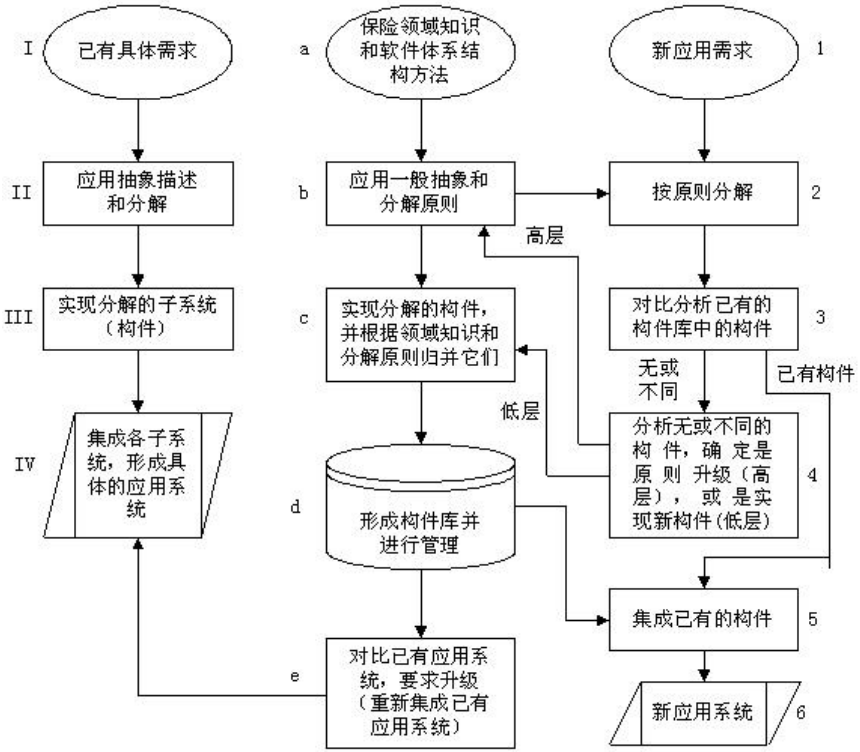


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

➢ 3、采用DSSA后的应用构成和系统升级

- 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。

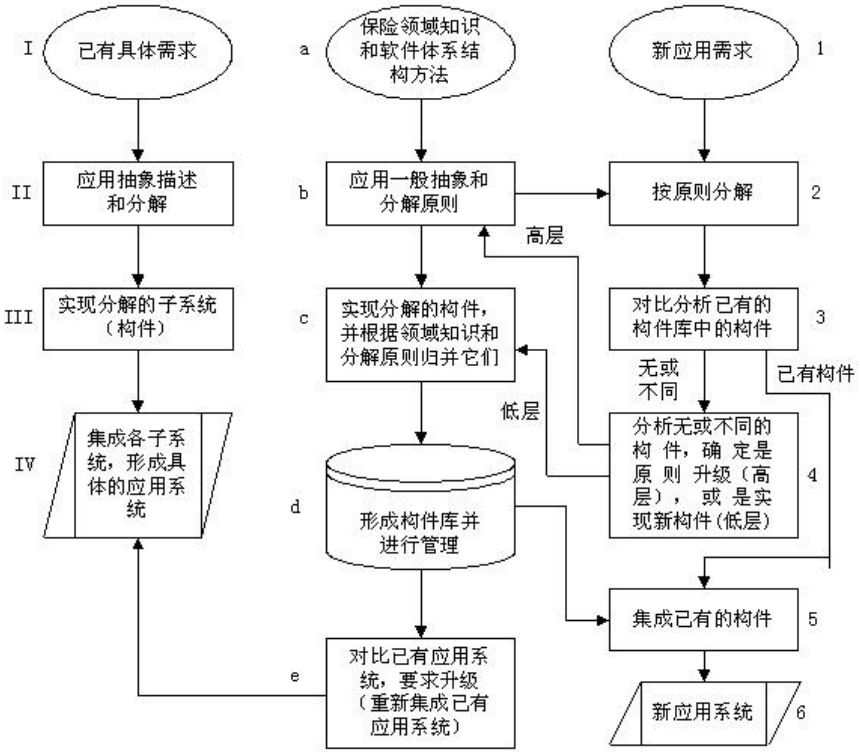


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

3、采用DSSA后的应用构成和系统升级

- 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。

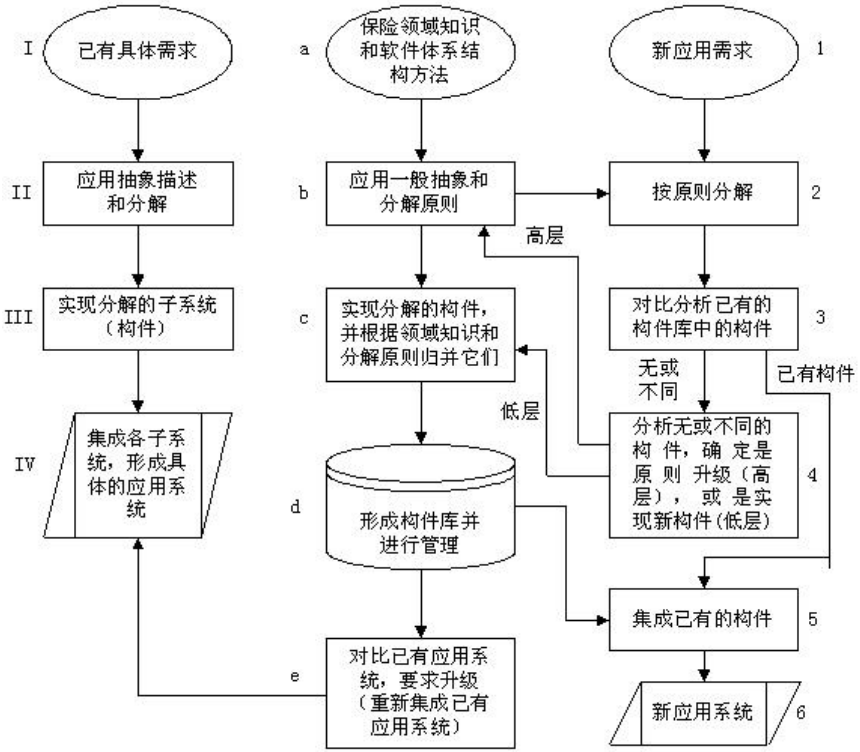


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 3、采用DSSA后的应用构成和系统升级
- 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、核保管理等内容。
  - 这实际上需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。
  - 同时引发标号c中的新的实现。
  - 同样的情况，也会发生在批改、理赔的构件中，并由此引起构件库的升级。

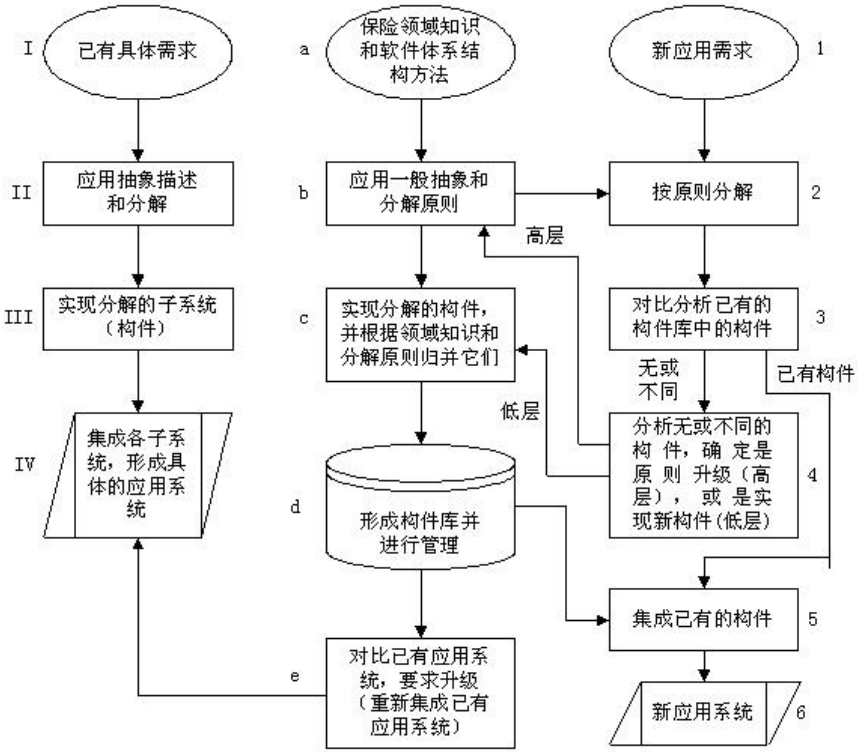


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 3、采用DSSA后的应用构成和系统升级
- 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。

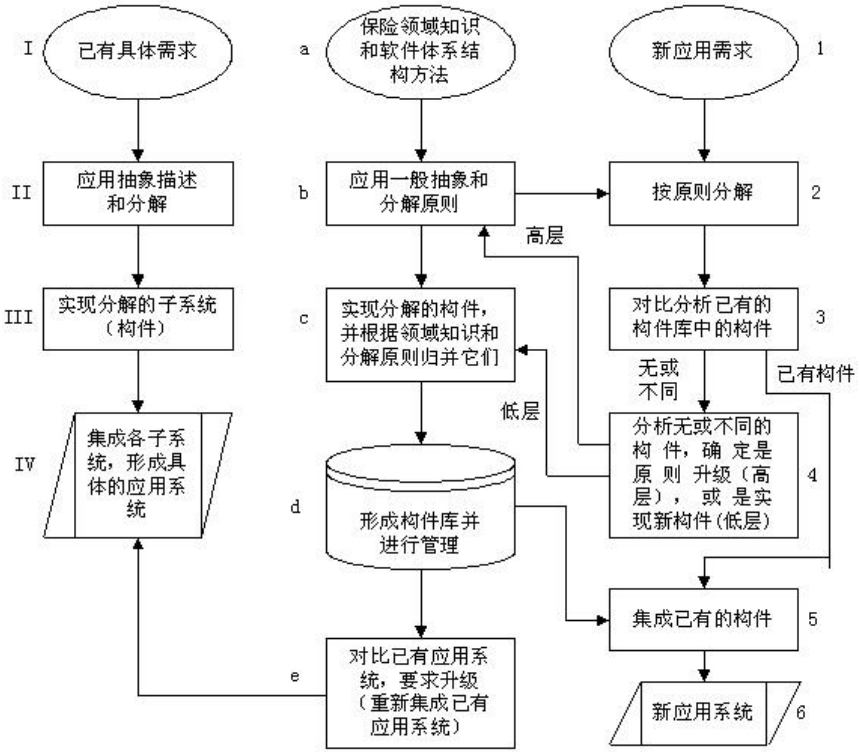


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 3、采用DSSA后的应用构成和系统升级
- 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所反映的是：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。
  - 构件库完善后，对标号IV的应用系统，进行升级是必要的。

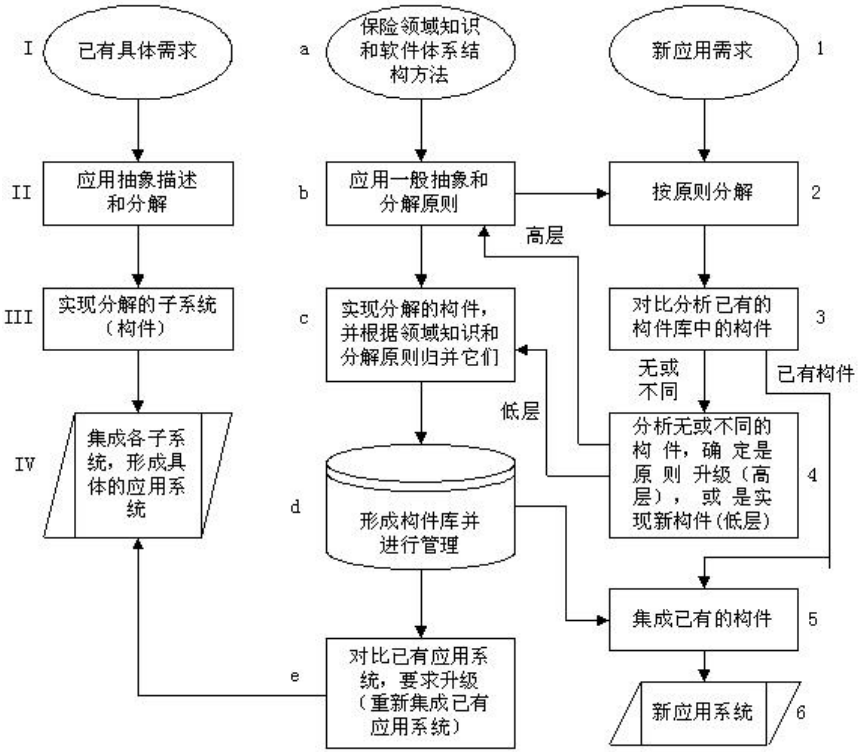


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 3、采用DSSA后的应用构成和系统升级
- 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，而且这种升级因整体费用的考虑，不一定马上进行。

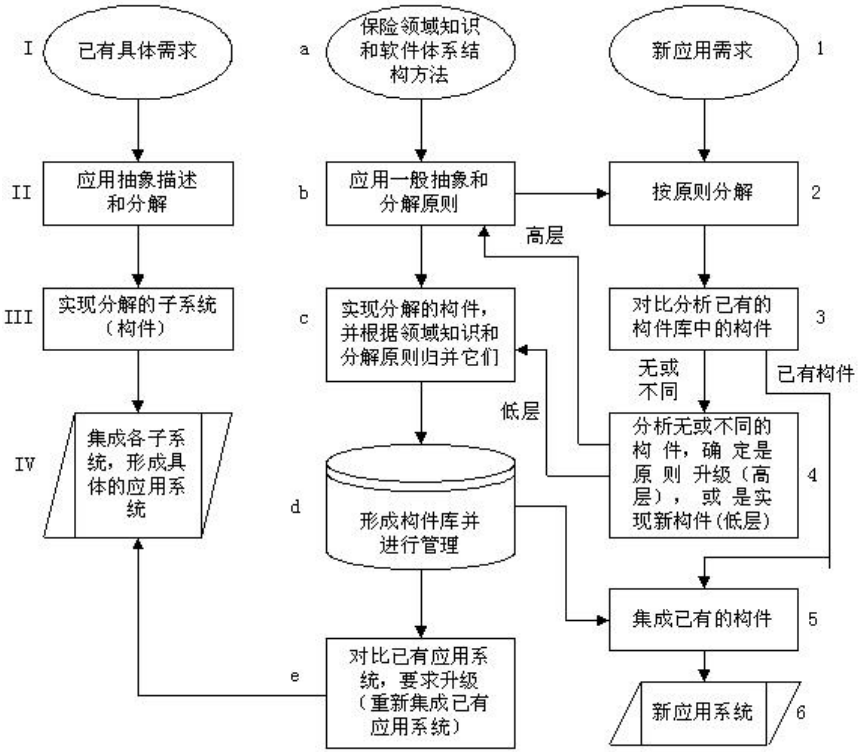


图3-36 保险行业DSSA整体结构图

3.10.5 DSSA实例

- 3、采用DSSA后的应用构成和系统升级
- **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度相当地大。**

② **便于系统开发的组织管理。**

- ✓ **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。**

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。**
- ✓ **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，出现了两种方法：
  - 以问题域为出发点的DSSA
  - 以解决方案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。

3.10.6 DSSA与体系结构风格的比较

- 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。

3.10.6 DSSA与体系结构风格的比较

- 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。

3.10.6 DSSA与体系结构风格的比较

- 体系结构风格所提取的设计知识，比用DSSA提取的设计专家知识的应用范围要广。
- 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。

- 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。

The End