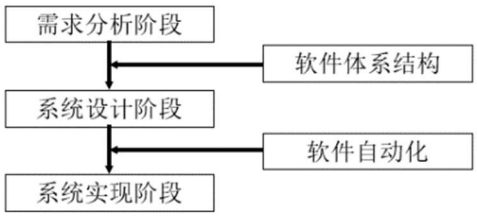
第一章 软件体系结构概论

传统软件工程的流程：



1.1 软件危机
1.2 发展历史
1.3 定义
1.4 研究内容
1.5 意义

###1.1 软件危机###

定义：落后的软件生产方式无法满足迅速增长的计算机软件需求，从而导致软件开发与维护过程中出现一系列严重问题的现象。

产生背景：20 世纪 60 年代以前，计算机刚刚投入实际使用，软件设计往往只是为了一个特定的应用而在指定的计算机上设计和编制，采用密切依赖于计算机的机器代码或汇编语言，软件的规模比较小，文档资料通常也不存在，很少使用系统化的开发方法，设计软件往往等同于编制程序，基本上是个人设计、个人使用、个人操作、自给自足的私人化的软件生产方式。

60 年代中期，大容量、高速度计算机的出现，使计算机的应用范围迅速扩大，软件开发急剧增长。高级语言开始出现；操作系统的发展引起了计算机应用方式的变化；大量数据处理导致第一代数据库管理系统的诞生。

软件系统的规模越来越大，复杂程度越来越高，软件可靠性问题也越来越突出。原来的个人设计、个人使用的方式不再能满足要求，迫切需要改变软件生产方式，提高软件生产率，软件危机开始爆发。

1968 年北大西洋公约组织的计算机科学家在联邦德国召开国际会议，第一次讨论软件危机问题，并正式提出“软件工程”一词，从此一门新兴的工程学科“软件工程学”为研究和克服软件危机应运而生。

主要表现：1）软件成本日益增长；2）开发进度难以控制；3）软件质量差；4）软件维护困难

20 世纪 60 年代的软件危机使得人们**开始重视软件工程的研究**。**起初**，人们把软件设计的重点放在数据结构和算法的选择上，随着软件系统规模越来越大、越来越复杂，整个系统的结构和规格说明显得越来越重要。软件危机的程度日益加剧，现有的软件工程方法对此显得力不从心。对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。在此种背景下，人们认识到软件体系结构的重要性，并认为对软件体系结构的系统、深入的研究将会成为提高软件生产率和解决软件维护问题的新的最有希望的途径。

###1.2 发展历史###

自从软件系统首次被分成许多模块，模块之间有相互作用，组合起来有整体的属性，就具有了体系结构。好的开发者常常会使用一些体系结构模式作为软件系统结构设计策略，但他们并没有规范地、明确地表达出来，这样就无法将他们的知识与别人交流。软件体系结构是设计抽象的进一步发展，满足了更好地理解软件系统，更方便地开发更大、更复杂的软件系统的需要。**软件体系结构这个术语第一次出现在 1969 年由 NATO 组织的一个关于软件工程技术的会议上**。从那时候直到 20 世纪 80 年代末期，出现了一些与软件体系结构相关的研究，其中典型的研究成果包括：1）Edsger Dijkstra 首先提出了软件结构的重要性，他认为“软件体系结构的研究应该注重软件系统的分割及其结构，而不应该仅仅局限于实现一个正确的软件系统”，提出了结构化设计的思想，并且指出良好的软件结构体现出来的概念完整性将使软件的开发和维护大大受益。2）Fred Brooks 提出了概念结构的思想，认为系统的体系结构是完整、详细的用户接口说明，提出了在系统结构设计中，概念完整性是最重要的考虑因素，并进一步指出体系结构设计必须和实现区分开来，他指出“体系结构陈述的是发生了什么，而实现描述的是如何实现”。3）David Parnas 发展了软件结构研究，先后提出了信息隐蔽模块、软件结构和程序家族等概念。总体来说，该阶段的研究还没有形成完整的技术体系，并且在概念上也比较混乱，如软件结构、软件系统结构、软件系统组织等。4）**软件体系结构作为一个单独的学科并对其开展系统的研究始于 20 世纪 90 年代初。**1991 年，在 Winston W. Royce 和 Walker Royce 父子的一篇论文中，软件体系结构第一次同时出现在论文标题和论文内容中。5）在 1992 年，Dewayne Perry 和 Alexander Wolf 发表了对以后发展有巨大影响的文章“Foundations for the Study of Software Architecture”，在这篇文章中提出了关于什么是软件体系结构的著名公式：**{elements, forms, rationale} = software architecture**. 这篇文章也被许多学者看作是**软件体系结构正式成为软件工程的一个研究方向的标志**。6）软件体系结构得到学术界和工业界广泛认同并进入全面发展始于 1995 年。1995 年第一届软件体系结构国际研讨会 IWSA-1 召开，IEEE Software 和 IEEE Transactions on Software Engineering 软件体系结构专刊出版。1996 年《Software Architecture : Perspectives on an Emerging Discipline》专著出版。标志着软件体系结构作为软件工程的一个研究分支正式提出。7）此后十年内，软件体系结构领域得到了蓬勃发展。越来越多的研究者关注并参与到软件体系结构的研究中来，与软件体系结构相关的会议、期刊、书籍等逐步增多，越来越多的知名国际会议将软件体系结构列入主要议题，并举行了大量直接以软件体系结构为主题的研讨会或国际会议（如软件体系结构国际研讨会 ISAW, WICSA 等）。8）许多知名国际期刊中，与软件体系结构相关的研究成果逐渐增多，并出版了大量

软件体系结构方面的书籍（如 SEI 软件工程系列丛书)。软件体系结构的研究还得到了工业界的广泛关注与认同，如 UML2 标准中引入了软件体系结构领域中连接子的概念，在实际软件开发过程(如统一软件开发过程)中也引入软件体系结构的概念和原则。2006 年出版的 IEEE Software 软件体系结构专刊总结了这十年间的软件体系结构研究与实践。9）随着软件系统的规模变得越来越大，复杂程度变得越来越高，软件体系结构也由最初模糊的概念发展到一个渐趋成熟的理论和技术。**软件体系结构的发展大致经历了 4 个阶段：①无体系结构阶段**(20 世纪 70 年代以前)。以汇编语言进行小规模应用程序开发为特征，这个阶段软件规模较小，很少明确考虑软件结构的问题。**②萌芽阶段**(20 世纪 70 年代中后期)。由于结构化开发方法的出现与广泛应用，软件开发中出现了概要设计与详细设计，而且主要任务是数据流设计与控制流设计。此时，软件结构已作为一个明确的概念出现在系统的开发中。**③初级阶段**(20 世纪 80 年代初至 90 年代中期)。面向对象开发方法逐渐兴起与成熟。因为对象是数据与基于数据之上操作的封装，因此在面向对象开发方法下，数据流设计与控制流设计则统一为对象建模。这个阶段出现了从不同侧面描述系统的结构模型，以 UML(Unified Modeling Language，统一建模语言)为典型代表。**④高级阶段**(20 世纪 90 年代中期以后)。以 philippe Kruehten 提出的“4+1”模型为标志，以描述系统的高层抽象结构为中心，不关心具体的建模细节，划分了体系结构模型与传统软件结构的界限。

在高级阶段，软件体系结构已经作为一个明确的文档和中间产品存在于软件开发过程中，同时，软件体系结构作为一门学科逐渐得到人们的重视，并成为软件工程领域的研究热点，因而 Dewayne Perry 和 Alexander Wolf 认为，“未来的年代将是研究软件体系结构的时代”。

###1.3 软件体系结构的定义###

虽然软件体系结构已经在软件工程领域中有着广泛的应用，但迄今为止还没有一个软件体系结构被大家所公认的定义。许多专家学者从不同角度和不同侧面对软件体系结构进行了刻画。长期以来，CMU-SEI（卡内基梅隆大学软件工程研究所）在其网站上公开征集 SA 的定义，至今已有百余种。

Perry & Wolf 1992: SA={elements，form，rational}. 软件体系结构是由一组元素(elements)、软件体系结构形式(form)和准则(rational)组成。元素(elements)包括处理元素(processing elements)、数据元素(data elements)和连接元素(connecting elements)共 3 类，处理元素负责对数据进行加工，数据元素是被加工的信息，连接元素把体系结构的不同部分组合连接起来。软件体系结构形式(form)是由专有特性(properties)和关系(relationship)组成，专有特性用于限制软件体系结构元素的选择，关系用于限制软件体系结构元素组合的拓扑结构。而在多个体系结构方案中选择合适的体系结构方案往往基于一组准则(rational)。

Garlan & Shaw 1993: 软件体系结构是设计过程的一个层次，它处理算法和数据结构之上关于整体系统结构设计和描述方面的一些问题，如大组织结构和全局控制结构；关于通讯、同步和数据存取的协议；设计元素的功能定义、物理分布和合成；设计方案的选择、评估和实现等。

Bass 等 1994: 一个系统的体系结构设计至少从以下三个方面进行描述：**该系统应用领域的功能分割、系统结构和结构的领域功能分配**。

Hayes-Roth 1994:软件体系结构是一个由功能构件组成的抽象系统的说明，按照功能构件的行为、界面和构件之间的相互作用进行描述。

Garlan & Perry 1995:软件体系结构包括一个程序/系统的构件的结构、构件的相互关系、以及控制构件设计演化的原则和指导三个方面。

Boehm 等 1995: 一个软件体系结构由以下三部分组成：一组软件系统的成分、连接和约束；一组系统仓库管理员提出的需求；一个理论，它证明用构件、连接和约束定义的一个系统在实现后能满足系统仓库管理员需求。

Soni, Nord, Hofmeister 1995: 软件体系结构至少由四个不同的实例化结构，这些结构从不同的方面描述了系统：1)概念体系结构按照主要设计元素和这些元素之间的关系描述系统；2)模块连接体系结构包含两个正交的结构：功能分解和分层；3)执行体系结构描述了一个系统的动态结构；4)代码结构描述了在开发环境中源代码、二进制和库是如何组织的。

Show 1995: 将软件体系结构定义分类为多个模型：1)结构模型认为软件体系结构是由构件、构件间的连接加上一些其他方面组成的，包括：配置，风格；条件，语义；分析，属性；理论，需求说明、仓库管理员需求。2)框架模型类似于结构观点，但主要强调整个系统的内在结构（常常是单一的），而不是它的组成。3)动态模型重在系统的行为质量，“动态”可以是整个系统的配置变化、通讯和交互通道的设置或取消，也可以是计算过程中的动态性，如数值改变等。4)过程模型重在体系结构的构造，构造的步骤和过程。按照这个观点，体系结构是实施过程描述的结果。

ANSI/IEEE 标准 1471-2000: SA={component，relationship，environment，prineiple}。软件体系结构是一个系统的基本组织结构，表现为构件、构件之间的相互关系、构件与环境之间的相互关系，以及指导其设计和演化的原理。

Bass, Clements, Kazman 2003: 一个程序或计算系统的软件体系结构是该系统的一个或多个结构，它们由软件元素、元素的外部可见属性以及这些元素之间的关系组成。其中，“外部可见”属性是指其他元素可以了解一个元素的前提，包括提供的服务、性能特性、错误处理、共享资源使用等。

Clements 等 2010: 软件体系结构是关于系统的一组结构，它们由软件元素、软件元素之间的关系和两者的属性组成。

张效祥，2005，软件体系结构 software architecture 是软件总体结构的抽象表示，或以此为研究对象的学科。**1)规定性含义。**软件体系结构=（结构元集，结构形，结构理）。结构元集为一组构成软件的结构元。结构元有三类，即处理元、信息元和连接元。处理元为对

信息元施行处理的构件，信息元为处理元的处理对象，连结元负责构件间的连接。结构形包括特性、联系以及权重。特性用以约束结构元的选取，联系则约束结构元间的交互与组织，权重表示特性及联系的重要程度。结构理刻画体系结构人员选取体系结构风格、结构元、结构形的动因与根据。体系结构风格是各种特定体系结构中结构元与结构形的抽象，它不如特定体系结构约束严格，也不如特定体系结构完备。**2)描述性含义。**软件体系结构=（构件集，连件集，模式，约束集）构件集表示构成软件的一组组成元素，连件集为一组连件，用以刻画各构件间的交互，模式为软件设计风格的描述，反映由构件及连件构成软件的构成原则，约束集中的约束表示对模式所加的限制条件。**3)多视面含义。**软件体系结构为软件的一个或多个结构，每一个结构反应一种视面。软件体系结构=结构集；结构=（构件集，外部可见特性集，联系集）；构件集表示构成软件的一组组成元素，外部可见特性反映为其他构件可利用该构件所作的假定，联系用以沟通相关构件。

###1.4 研究内容###

1、软件体系结构建模：研究软件体系结构的首要问题是如何表示软件体系结构，即如何对软件体系结构建模。根据建模的侧重点不同，可以将软件体系结构的模型分为 5 种，分别是**结构模型、框架模型、动态模型、过程模型和功能模型**。在这 5 个模型中，最常用的是**结构模型和动态模型**。上述 5 种模型各有所长，将 5 种模型有机地统一在一起，形成一个完整的模型来刻画软件体系结构更合适。Phillppe Kruchien 在 1995 年提出了一个“4+1”的视图模型。“4+l”视图模型从 5 个不同的视角包括逻辑视图、进程视图、物理视图、开发视图和场景视图来描述软件体系结构。每一个视图只关心系统的一个侧面，5 个视图结合在一起才能反映软件体系结构的全部内容。

2、软件体系结构描述方法：1)图形表达工具。用矩形框代表抽象构件，框内标注的文字为抽象构件的名称，用有向线段代表辅助各构件进行通讯、控制或关联的连接件。**2)模块内连接语言。**模块内连接语言 (Module Interconnection Language，MIL)将一种或几种传统程序设计语言的模块连接起来。由于程序设计语言和模块内连接语言具有严格的语义基础，因此能支持对较大的软件单元进行描述。**3)基于软构件的系统描述语言。**基于软构件的系统描述语言将软件系统描述成一种是由许多以特定形式相互作用的特殊软件实体组成的系统。**4)软件体系结构描述语言。**软件体系结构描述语言(Architecture Description Language，ADL)是参照传统程序设计语言的设计和开发经验，针对软件体系结构特点，重新设计、开发的描述方式。**5)软件体系结构描述框架标准。**鉴于体系结构描述的概念与实践的不统一，IEEE 于 1995 年 8 月成立了体系结构工作组，综合体系结构描述研究成果，并参考业界的体系结构描述的实践，负责起草了体系结构描述框架标准即 IEEE P1471，并于 2000 年 9 月 21 日通过 IEEE-SA 标准委员会评审。IEEE P1471 仅仅提供了体系结构描述的概念框架、体系结构描述应该遵循的规范，但如何描述以及具体的描述技术等方面缺乏更进一步的指导。

3、软件体系结构分析、设计与验证：软件体系结构分析的内容可分为**结构分析、功能分析和非功能分析**。

体系结构设计是**生成一个满足软件需求的体系结构的过程**，其本质在于将系统分解成相应的组成成分，并将这些成分重新组装成一个系统。目前，体系结构设计主要有两大类方法，分别是**过程驱动方法**和**问题列表驱动方法**。体系结构设计研究的重点内容之一就是体系结构风格/模式，体系结构风格在本质上反映了一些特定的元素、按照特定的方式组成一个特定的结构，该结构应有利于上下文环境下的特定问题的解决。**体系结构风格**可分为两个大类，分别为**固定术语**和**参考模型**。

体系结构验证是对系统结构进行测试和验证，体系结构验证有两大类方法：**体系结构测试**和**体系结构形式化验证**。

4、软件体系结构发现、演化与重用：体系结构发现解决如何从已经存在的系统中提取软件体系结构的问题，属于逆向工程范畴。

体系结构演化是指由于系统需求、技术、环境、分布等因素的变化而最终导致软件体系结构变动的过程。软件在运行时刻的体系结构变化称为体系结构的动态性，而将体系结构的静态修改称为体系结构扩展。**体系结构的动态性**分为**有约束的和无约束的以及结构动态性和语义动态性**。

体系结构重用属于设计重用，比代码重用更抽象。由于软件体系结构是系统的高层抽象，反映了系统的主要组成元素及其交互关系，因而较算法更稳定，更适合于重用。

5、基于体系结构的软件开发方法：引入了体系结构之后，软件开发的过程变为“问题定义->软件需求->软件体系结构设计->软件设计->软件实现”，可以认为软件体系结构架起了软件需求与软件设计之间的一座桥梁。而在由软件体系结构到实现的过程中，借助中间件技术与软件总线技术，软件体系结构将易于映射成相应的实现。

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

7、软件体系结构评估方法：软件体系结构的设计是整个软件开发过程中关键的一步，但是，怎样才能知道为软件系统所选用的体系结构是恰当的呢?如何确保按照所选用的体系结构能顺利地开发出成功的软件产品呢?要回答这些问题并不容易，因为它受到很多因素的影响，需要专门的方法来对其进行评估。**体系结构评估可以只针对一个体系结构，也可以针对一组体系结构**。从目前已有的软件体系结构评估技术来看，可以归纳为三类主要的评估方式，分别是**基于调查问卷或检查表的方式**、**基于场景的方式**和**基于度量的方式**。

###1.5 意义###

1)风险承担者进行交流的手段：构建软件体系结构的过程是一种对软件系统的抽象过程，它有选择性地突出和隐蔽某些系统特征，完成对系统的高层建模活动，从而更有利于人们对复杂软件的系统理解：其一，有效增强系统相关人员的信息交流；其二，有效改进软件

系统和软件过程的理解。一个良好的软件体系结构是一张融合了诸多需求角度的多维视图，它提供了一种通用语言，使不同的需求得以表达、交流、协商和达成共识。良好的体系结构不仅有效改善了人员间的交流，而且使得基于体系结构描述的系统理解取得了巨大成功。

2)早期设计决策的体现: 软件体系结构体现了系统的最早的一组设计决策，这些早期的约束比起后期的工作重要的多，对系统生命周期的影响也大得多。（1）明确了对系统实现的约束条件.（2）决定了开发和维护组织的结构.（3）制约着系统的质量属性.（4）使推理和控制变更更简单.（5）有助于循序尽进的原型设计.

3)可传递和可重用的模型: 软件体系结构级的重用意味着体系结构的决策能在具有相似需求的多个系统中发生影响，这比代码级的重用有更大的好处。

第二章 软件体系结构建模

2.1 模型

2.2 元模型

2.3 建模方法

2.4 软件体系结构的生命周期模型

在软件开发的各个阶段，都需要运用不同的方法对系统建立各种各样的模型，比如需求模型、功能模型、数据模型和物理模型等等，可以说**整个软件开发的过程就是一个模型不断建立和不断转化的过程**。软件体系结构的**建模方法**是由**建模语言**和**建模过程**两部分组成。其中，建模语言是用来表述设计方法的表示法，建模过程是对设计中所应采取的步骤的描述。

###2.1 模型###

1)、结构化模型。结构化模型是最常见的体系结构模型，综合软件体系结构的概念，软件体系结构的结构化模型由 5 种元素组成：**构件(component)、连接件（connector）、配置（configuration）、端口（port）和角色（role）**，其中**构件、连接件和配置**是最基本的元素。

构件是具有某种功能可重用的软件模板单元，表示了系统中主要的计算元素和数据存储。构件有两种：**复合构件和原子构件**，复合构件由其它复合构件和原子构件通过连接构成；原子构件是不可再分的构件，底层由实现该构件的类组成，这种关于构件的划分方法提供了体系结构的分层表示能力，有助于简化体系结构的设计。

连接件表示了构件之间的交互，简单的连接件如：管道(pipes)、过程调用(procedure call)、事件广播(event broadcast)等，更为复杂的交互如：客户-服务器(client-server)通讯协议、数据库和应用之间的 SQL 连接等。

配置表示了构件和连接件的拓扑逻辑和约束。

另外，构件作为一个封装的实体，只能通过其接口与外部环境交互，构件的接口由一组**端口**组成，每个端口表示了构件和外部环境的交互点。通过不同的端口类型，一个构件可以提供多重接口。一个端口可以非常简单，如过程调用，也可以表示更为复杂的界面(包含一些约束)，如必须以某种顺序调用的一组过程调用。

连接件作为建模软件体系结构的主要实体，同样也有接口，连接件的接口由一组**角色**组成，连接件的每一个角色定义了该连接件表示的交互的参与者，二元连接件有两个角色，如 RPC 的角色为 Caller 和 Called，pipe 的角色是 reading 和 writing，消息传递连接件的角色是 sender 和 receiver。有的连接件有多于两个的角色，如事件广播有一个事件发布者角色和任意多个事件接受者角色。

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

3)、动态模型。是对结构或框架模型的补充，研究系统的“大颗粒”的行为性质。例如，描述系统的重新配置或演化。

4)、过程模型。研究构造系统的步骤和过程，体系结构是遵循某些过程脚本的结果。

5)、功能模型。认为体系结构是由一组功能构件按层次组成，下层向上层提供服务。功能模型可以看作是一种特殊的框架模型。

6)、4+1 模型。从多个视图描述软件体系结构，每一个视图描述软件体系结构的不同特征，这样有助于减少体系结构建模的复杂度，有助于设计人员对体系结构的理解。

Kruehten 在 1995 提出了软件体系结构的 4+1 视图描述，show 在 1996 中提出了多维设计空间的概念用来描述软件体系结构的不同特征，分为功能设计空间(表示功能和性能需求)和结构设计空间(表示系统的初始分解结果)，相比较而言，4+1 视图描述较为全面地描述了软件体系结构，它是以 Booch 方法的图形描述为基础，建立在对象模型之上，但缺少形式化描述基础。下面我们详细介绍”4+1”视图模型，通过它更好地理解软件体系结构概念。

“4+1”视图模型从 5 个不同的视角来描述软件体系结构。

1)逻辑视图。逻辑视图（logic view），也称概念视图，主要支持对系统功能方面需求的抽象描述，即系统最终将提供给用户什么样的服务。

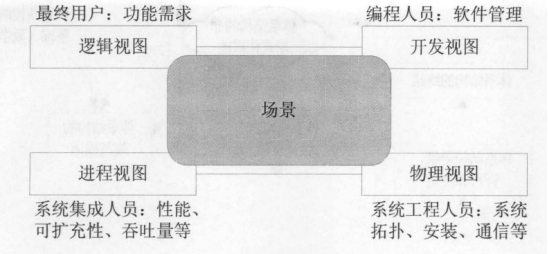


图 1-1 “4+1”视图模型

在逻辑视图中，系统分解成一些列的功能抽象，这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析，而且可用作标识在整个系统的各个不同部分的通用机制和设计元素，是系统工程师与领域专家交流的有效媒介。

逻辑视图强调问题空间中各实体间的相互作用，用实体一关系图来描述，若采用面向对象技术，则可以用类图来描述。其中的构件主要是类，风格通常是面向对象风格。

2)开发视图。开发视图（development view），也称模块视图，主要侧重于系统模块之间的组织和管理。

根据系统模块的组织方式，这个视图可以有不同的形式。通常是根据分配给项目组的开发和维护工作来组织，例如：可根据信息隐蔽原则来组织模块，也可把系统运行时关系紧密或执行相关任务的模块组织在一起，也可把模块组织成层次结构。

该视图同逻辑视图不同，它与实现紧密相连，通常用模块和子系统图来表示接口输人输出。该视图的风格主要是层次结构风格，通常将层次限制在 4 一 6 层左右。在同一个系统中，子图中的模块之间可以相互作用，同一层中不同子图间的模块之间也可以相互作用，也可与相邻层的模块之间相互作用。这样可使每个层次的接口既完备又精练，减少各模块之间的复杂关系。此外，对于各个层次，越是底层通用性越强，当应用系统的需求发生变化时，所需的改变越少。

3)进程视图。进程视图（process view）主要侧重于描述系统的动态属性，即系统运行时的特性。

该视图着重解决系统的并发和分布，及系统的完整性和容错性，同时也定义在概念视图中的各个类中的操作是在哪一个控制线索中被执行的。该角度的部件通常是进程，进程是一个有其自己控制线索的命令语句序列。当系统运行时，一个进程可被执行、挂起、唤醒等，同时可与其它进程通讯、同步等，并且通过进程间的通讯模式可对系统性能进行评估。进程视图有许多风格，如数据流风格、客户/服务器风格等。

4)物理视图。物理视图主要描述如何把软件映射到硬件上，通常要考虑系统的性能、规模、容错等。

当软件运行于不同的节点上时，各视图中的部件都直接或间接地对应在系统的不同节点上。因此，从软件到节点的映射要有较高的灵活性，当环境改变时，对系统其它视图的影响才比较小。此外，这种映射关系直接影响到系统的性能。

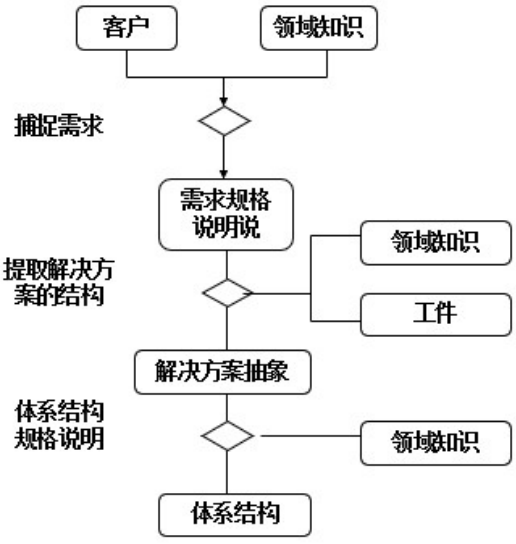
5)场景视图。不同的视图描述了同一系统的不同侧面，所以它们之间并不是相互独立的。通过“场景”可将不同的视图联系起来。

所谓“场景”，可看作是重要的系统活动的抽象，在开发系统体系结构时，它可帮助设计人员找到体系结构的构件及构件间的相互作用关系。通过场景，可以分析系统的体系结构或某个特定视图，场景还可以描述不同视图的构件如何相互作用。场景可用文本表示，也可用图形表示，如 OID 图。

从以上介绍可知，逻辑和开发视图描述的是系统的静态结构，进程和物理视图描述的是系统的动态结构。逻辑视图与开发视图虽密切相关，但它们的侧重点不同，系统规模越大，它们的差别也越明显。此外，值得注意的是，系统的拓扑结构在不同的视图下保持不变。

###2.2 元模型###

元模型是对各种体系结构设计模型的抽象。使用这个模型对当前的各种体系结构设计方法进行分析和比较。各种体系结构设计方法都可以描述成元模型的实例，每种方法在过程的顺序上、在概念的特定内容上有所不同。



体系结构设计的元模型

###2.3 建模方法###

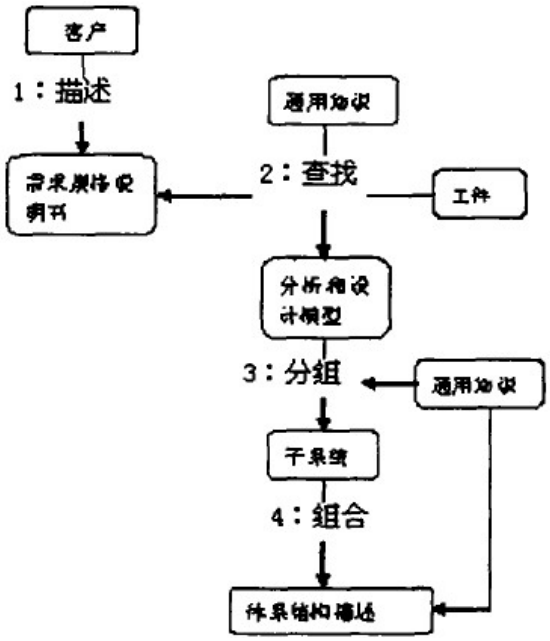
为了获取对体系结构设计的抽象，人们已经提出了许多方法。我们把这些体系结构设计方法分类为：

- 工件驱动(Artifact-Driver)的方法

- 用例驱动(Use-Case-Driven)的方法
- 模式驱动(Pattern-Driven)的方法
- 领域驱动(Domain-Driven)的方法

下面所介绍的每一种方法都可视为元模型的一种实现。

工件驱动(Artifact-Driver)的方法：工件驱动的体系结构设计方法从工件描述中提取体系结构描述。工件驱动的体系结构设计方法的例子包括广为流行的面向对象分析和设计方法 OMT(Object Modeling Technology)和 OAD(Object orient Analysis & Design)。



加以标号的箭头表示体系结构设计步骤的过程顺序：“分析和设计模型”和“子系统”的概念共同表示了元模型中的“解决方案抽象”概念；“通用知识”概念表示了元模型中“领域知识”概念的特殊化。

用 OMT 解释这一模型，可以把 OMT 认为是这一策略的适当代表。在 OMT 中，体系结构设计并不是软件开发过程中的一个明确阶段，而是设计阶段的一个隐含部分。

OMT 方法主要由以下阶段组成：分析、系统设计、对象设计。箭头线“l：描述”表示需求规格说明书的描述；箭头线“2：查找”表示对工件的查找，如系统分析阶段中需求规格说明的类。

查找过程得到了软件工程师的通用知识的支持，也得到了构成工件的启发式规则的支持，这些工件是该方法的重要构成部分。“2：查找”的结果是一组工件实例，在元模型中用“分析和设计模型”的概念来表示。

在 OMT 方法中，接下来是系统设计阶段。该阶段将工件分组为子系统，为单个软件系统的全局结构的开发定义整体体系结构。这一功能被表示为“3: 分组”。软件体系结构由子系统组合而成，被表示成“4: 组合”。这一功能也用到了“通用知识”概念提供的支持。

在体系结构开发方面，该方法**存在着以下问题：**

(1)文本形式的系统需求含混不清，不够精确、不够完整，因此，将它作为体系结构抽象的来源作用是不够的。

(2)子系统的语义过于简单，难以作为体系结构构件。

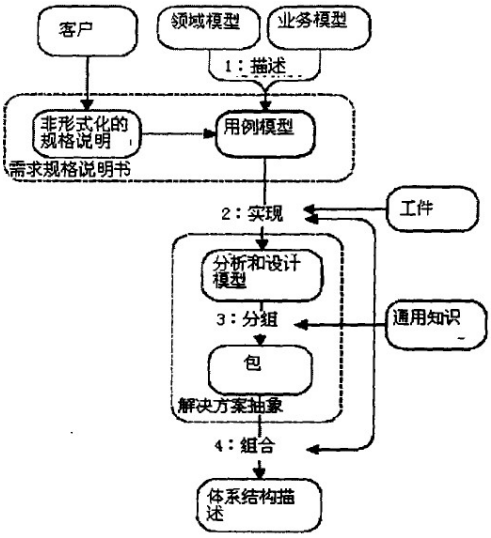
(3)对子系统的组合支持不足 。

用例驱动(Use-Case-Driven)的方法：用例驱动的体系结构设计方法主要从用例导出体系结构抽象。

一个用例，是指系统进行的一个活动系列，它为参与者提供一些结果值。参与者通过用例使用系统。参与者和用例共同构成了用例模型。用例模型的目的是作为系统预期功能及其环境的模型，并在客户和开发者之间起到合约的作用。

统一过程使用的是一种用例驱动的体系结构设计方法。它由核心工作流(core workflows)组成。

核心工作流定义了过程的静态内容，用活动、工人和工件描述了过程。随时间变化的过程的组织被定义为阶段。下图给出了用统一过程描述的用例驱动的体系结构设计方法的概念模型。



统一过程由六个核心工作流组成：商业模型、需求、分析、设计、实现和测试。这些核心工作流的结果分别是下列模型：商业和领域模型、用例模型、分析模型、设计模型、实现模型和测试模型。

在需求工作流中，以用例的形式捕捉客户的需求，构成用例模型。这一过程在上图中被定义为“1：描述”。用例模型和非形式化的需求规格说明共同构成了系统的需求规格说明。

用例模型的开发得到了“非形式化的规格说明”、“域模型”、“商业模型”等概念的支持，在设置系统的上下文时这些概念是必需的。如前所述，“非形式化的规格说明”表示文本形式的需求规格说明。“商业模型”描述一个组织的商业过程。“域模型”描述上下文中最重要的类。

从用例模型可以选择出对于体系结构有重要意义的用例，并创建“用例实现”，如图中“2：实现”所述。用例实现决定了任务在系统内部是怎样进行的。

用例实现受到相关工件的知识和通用知识的支持。这在图中被表示为分别从“工件”和“通用知识”引出的指向“2：实现”的箭头线。这一功能的输出是“分析和设计模型”概念，它表示在用例实现之后标识出的工件。

然后，分析和设计模型被分组为包(packages)，这在图中被表示为“3：分组”。图中的“4：组合”代表定义这些包之间的接口，其结果是“体系结构描述”的概念。“3：分组”和“4：组合”这两个功能受到“通用知识”概念的支持。

在统一过程中，为了理解上下文，首先要开发商业模型和域模型。然后，主要从非形式化的规格说明、商业模型和域模型中导出用例模型。从用例模型中选择用例实现，导出体系结构抽象。

在使用这一方法标识体系结构抽象时，必须处理**下面几个问题**:

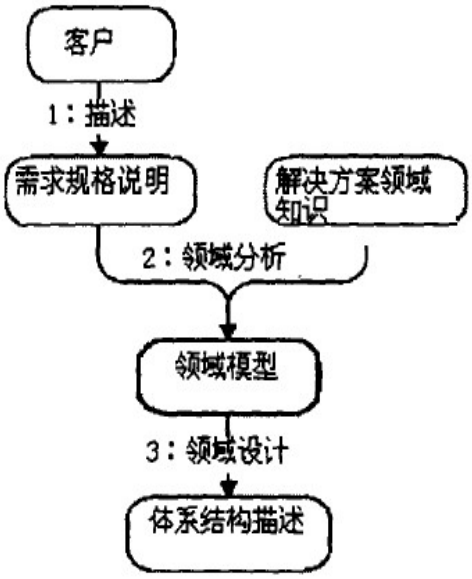
(1)难以适度把握域模型和商业模型的细节

(2)对于如何选择与体系结构相关的用例没有提供系统的支持

(3)用例没有为体系结构抽象提供坚实的基础

(4)包的语义过于简单，难以作为体系结构构件

领域驱动(Domain-Driven)的方法: 在领域驱动的体系结构设计方法中，体系结构抽象是从领域模型导出的。这一方法的概念模型如下图所示。



领域模型是在领域分析阶段开发的，这在图中被表示为“2：领域分析”。领域分析可以被定义成一个重用为目标的，确认、捕捉和组织问题领域的领域知识的过程。

图中“2：领域分析”以“需求规格说明”和“解决方案领域知识”的概念为前提，产生“领域模型”的概念作为结果。需要注意的是，图中的“解决方案领域知识”和“领域模型”都属于元模型的“领域知识”的概念。

领域模型可以有多种不同的表示方法，比如类、实体关系图、框架、语义网络、规则等。与此相应，领域分析的方法也有多种。

在这里，主要关注用领域模型导出体系结构抽象的方法。在图中，这被表示为“3：领域设计”。下面，考虑两种领域驱动的方法，它们从领域模型到体系结构设计抽象。

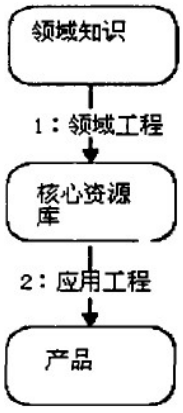
(1)产品线体系结构设计

在产品线体系结构设计方法中，软件体系结构是为一个软件产品线而开发的。

可以这样定义软件产品线：它是一组以软件为主的产品，具有多种共同的、受控的特点，能够满足特定市场或任务领域的需求。

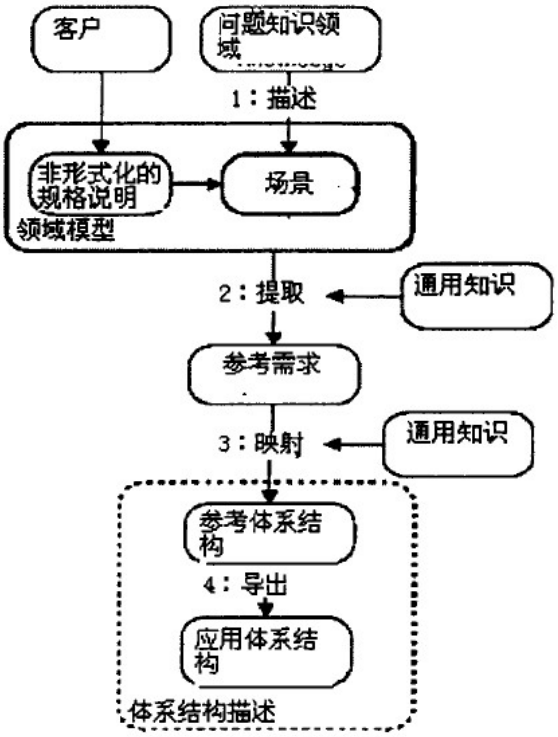
软件产品线体系结构是对一组相关产品的体系结构的抽象。产品线体系结构设计方法主要关注组织内部的重用，基本是由核心资源开发和产品开发这两个部分组成的。核心资源库通常包括体系结构、可重用软件构件、需求、文档和规格说明，性能模型，方案，预算，以及测试计划和用例。核心资源库用于从产品线生成或集成产品。

产品线体系结构设计的概念模型如下图所示。其中，“1：领域工程”表示核心资源库的开发，“2：应用工程”表示从核心资源库开发产品。



(2)特定领域的软件体系结构设计

可以把特定领域的软件体系结构看成是多系统范围内的体系结构，即它是从一组系统中导出的，而不是某一单独的系统。下图表示了 DSSA 方法的概念模型。



DSSA 方法的基本工件是领域模型、参考需求和参考体系结构。DSSA 方法从领域分析阶段开始，面向一组有共同问题或功能的应用程序。这种分析以场景为基础，从中导出功能需求、数据流和控制流等信息。

领域模型包括场景、领域字典、上下文图、实体关系图、数据流模型、状态转换图以及对象图。

从图中可以看出，除了领域模型之外，还定义了参考需求，它包括功能需求、非功能需求、设计需求、实现需求，而且它主要关注解决方案空间。领域模型和参考需求被用于导出参考体系结构。

DSSA 过程明确区别参考体系结构和应用体系结构。

参考体系结构被定义为用于一个应用系统族的体系结构，应用体系结构被定义为用于一个单一应用系统的体系结构。

应用体系结构是从参考体系结构实例化或求精而来的。实例化/求精的过程和对参考体系结构进行扩展的过程被称为应用工程。

由于对“领域”这一术语有着不同的解释，领域驱动的体系结构设计方法也有多种，其中**存在的一些问题如下**:

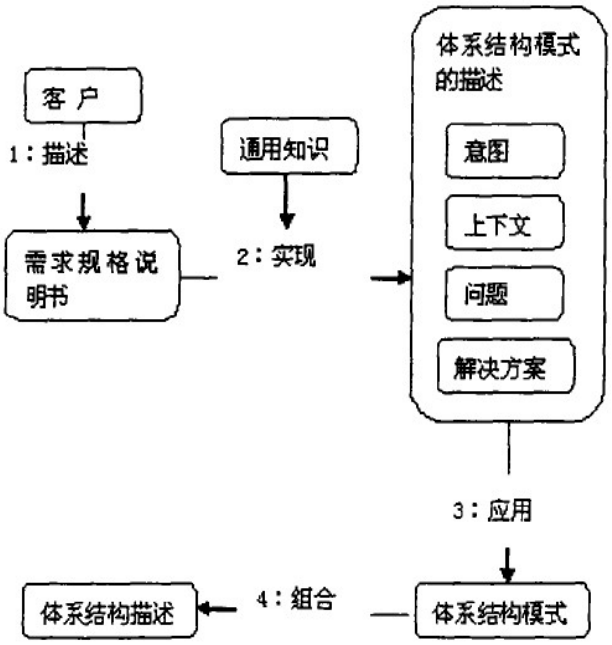
(1)问题领域分析在导出体系结构抽象方面效果较差；

(2)解决方案领域分析不够充分.

模式驱动(Pattern-Driven)的方法: 软件工业界已经广泛接受了软件设计模式的概念。软件设计模式的目的在于编制一套可重用的基本原则，用于开发高质量的软件系统。软件设计模式常常用在设计阶段，但是，人们已经开始在软件开发过程中的其他阶段定义并使用设计模式。比如，在实现阶段，可以定义从面向对象的设计到面向对象的语言构造的设计模式；在分析阶段，可以使用设计模式导出分析模型。

近年来，也有研究者在软件开发过程中的体系结构分析阶段应用设计模式。体系结构模式类似于设计模式，但它关心的是更粗粒度的系统结构及其交互。实际上，它也就是体系结构风格的另一种名称。体系结构设计模式是体系结构层次的一种抽象表示。

模式驱动的体系结构设计方法从模式导出体系结构抽象。下图描述了这一方法的概念模型。



“需求规格说明”的概念表示对问题的规格说明，该问题可能通过模式得以解决。图中的“实现”表示了为给出的问题描述查找适当的模式的过程，它受到“通用知识”概念的支持。

“体系结构模式描述”的概念指的是对体系结构模式的描述。它主要由 4 个概念组成：意图、上下文、问题和解决方案。

意图(intent)表示使用模式的基本原则；上下文(context)表示问题的产生环境；问题表示上下文环境中经常出现的问题；解决方案是以元素及其关系的抽象描述的形式来表示对问题的解决方案。

为了确认模式，要对各个可用模式的意图进行扫描。如果发现一个模式的意图和给出的问题相关，那么就分析它的上下文描述。这时，如果上下文描述仍然能够和给出的问题相匹配，则处理过程进入图中的“3：应用”。

进而，用解决方案这一子概念来提供所给出问题的解决方案。概念“体系结构模式”表示了“3：应用”的结果。最后，“4：组合”表示在导出体系结构描述时，体系结构模式之间的相互协作。

在许多体系结构设计方法中，都包括作为一个子过程的模式驱动的体系结构设计方法。尽管体系结构模式在构建软件体系结构时能够起到一定的作用，但是在选择模式、应用，以及把模式组成体系结构等问题上，当前的方法并不能提供足够的支持。下面更详细地**介绍了这些问题。**

(1)在处理范围广泛的体系结构问题时，模式库可能不够充足

(2)对模式的选择仅依靠通用知识和软件工程师的经验

(3)模式的应用并不是一个简单直接的过程，它需要对问题进行全面的分析

(4)对于模式的组合没有提供很好的支持

###2.4 软件体系结构的生命周期模型###

软件体系结构生命周期模型是对软件体系结构在整个生存期间所需经历的所有阶段和步骤的描述，这种描述独立于具体的体系结构，使得体系结构的设计遵循一定的理论基础和工程原则。

一个软件体系结构生命周期模型主要由以下几个阶段组成：

(1)软件体系结构的非形式化描述；

(2)软件体系结构的规范描述和分析；

(3)软件体系结构的求精及其验证；

(4)软件体系结构的实施；

(5)软件体系结构的演化和扩展；

(6)软件体系结构的提供、评价和度量；

(7)软件体系结构的终结.

(1)软件体系结构的非形式化描述(Software Architecture Informal Description)

一种软件体系结构在其产生时，其思想通常是简单的，并常常由软件设计师用非形式化的自然语言表示概念、原则。例如：客户机-服务器体系结构就是为适应分布式系统的要求，从主从式演变而来的一种软件体系结构。

尽管该阶段的描述常是用自然语言描述的，但是该阶段的工作却是创造性和开拓性的。

(2)软件体系结构的规范描述和分析(Software Architecture Specification and Analysis)

这一阶段通过运用合适的形式化数学理论模型对第 1 阶段的体系结构的非形式化描述进行规范定义，从而得到软件体系结构的形式化规范描述，以使软件体系结构的描述精确、无歧义，并进而分析软件体系结构的性质，如无死锁性、安全性、活性等。分析软件体系结构的性质有利于在系统设计时选择合适的软件体系结构，从而对软件体系结构的选择起指导作用，避免盲目选择。

(3)软件体系结构的求精及其验证(Software Architecture Refinement and Its Verification)

大型系统的软件体系结构总是通过从抽象到具体，逐步求精而达到的，因为一般说来，由于系统的复杂性，抽象是人们在处理复杂问题和对象时必不可少的思维方式，软件体系结构也不例外。但是过高的抽象却使软件体系结构难以真正在系统设计中实施。因而，如果软件体系结构的抽象粒度过大，就需要对体系结构进行求精、细化，直至能够在系统设计中实施为止。在软件体系结构的每一步求精过程中，需要对不同抽象层次的软件体系结构进行验证，以判断较具体的软件体系结构是否与较抽象的软件体系结构的语义一致，并能实现抽象的软件体系结构。我们这里并不排斥在不求精的情况下对软件体系结构的验证，而只是侧重于研究和讨论软件体系结构的求精及其验证。

(4)软件体系结构的实施(Software Architecture Enactment)

这一阶段将求精后的软件体系结构实施于系统的设计中，并将软件体系结构的构件和连接件等有机地组织在一起，形成系统设计的框架，以便据此实施于软件设计和构造中。

(5)软件体系结构的演化和扩展(Software Architecture Evolution and Extension)

在实施软件体系结构时，根据系统的需求，常常是非功能需求，如性能、容错、安全性、互操作性、自适应性等非功能性质影响软件体系结构的扩展和改动，这称为软件体系结构的演化。由于对软件体系结构的演化常常由非功能性质的非形式化需求描述引起，因而需要重复第 1 步，如果由于功能和非功能性质对以前的软件体系结构进行演化，就要涉及软件体系结构的理解，需要进行软件体系结构的逆向工程和再造工程。

(6)软件体系结构的提供、评价和度量(Software Architecture Provision, Evaluation and Metric)

这一阶段通过将软件体系结构实施于系统设计后系统实际的运行情况，对软件体系结构进行定性的评价和定量的度量，以利于对软件体系结构的重用，并取得经验教训。

(7)软件体系结构的终结(Software Architecture Termination)

如果一个软件系统的软件体系结构进行多次演化和修改，软件体系结构已变得难以理解，更重要的是不能达到系统设计的要求，不能适应系统的发展。这时，对该软件体系结构的再造工程即不必要、也不可行，说明该软件体系结构已经过时，应该摈弃，以全新的满足系统设计要求的软件体系结构取而代之。

第三章 软件体系结构风格

3.1 概述

3.2 经典软件结构风格 1)管道和过滤器 2)数据抽象和面向对象组织 3)基于事件的隐式调用 4)分层系统 5)、仓库系统及知识库 6)C2 风格

3.3 客户/服务器风格

3.4 三层 C/S 结构风格

3.5 浏览器/服务器风格

3.6 公共对象请求代理体系结构

3.7 正交软件体系结构

3.8 基于层次消息总线的体系结构风格

3.9 异构结构风格

3.10 互联系统构成的系统及其体系结构

3.11 特定领域软件体系结构

###3.1 概述###

软件体系结构设计的一个核心问题是能否使用重复的体系结构模式，即能否达到体系结构级的软件重用。也就是说，能否在不同的软件系统中，使用同一体系结构。基于这个目的，学者们开始研究和实践软件体系结构的风格和类型问题。

体系结构风格概念率先由 Perry 等人在 1992 年引入：软件体系结构风格是由从各种相似的、具体详细的体系结构中抽象出的组成元素及其组成关系表示的，比软件体系结构受到的限制更少、更不完全。

他们在软件体系风格上关注的主要是软件系统的组成元素以及元素间的连接关系。

Buschmann 等人进一步完善了此定义：

软件体系结构风格根据软件系统的结构组织定义了软件系统族；

通过构件应用的限制及其与系统结构、构建有关的组成和设计规则，来表示组成元素和组成元素之间的关系；

为一个软件系统及其怎样构造该系统表示一种特殊的基本结构；

也包括何时使用它所描述的体系结构、它的不变量和特例，以及其应用的效果等信息。

并且，他们将体系结构风格与体系结构模式两个概念区别开来，不仅关注构成软件系统的组成元素及其之间的连接关系，而且关注组成元素使用的约束条件、表示组成元素间关系的角度，以及体系结构风格的适用场合和应用效果。

Shaw 等人没有区分体系结构风格和体系结构模式，将两者视为相同的概念来使用。他们认为，软件体系结构风格是软件系统结构层次上的组织风格，它根据结构组织模式定义一个系统族 (具体地，就是定义了关于构件和连接器类型的术语以及它们如何组织在一起)；某些风格还存在一个或多个语义模型，指明如何根据系统各组成成分的属性来确定系统的整体属性。

Shaw 等人在体系结构风格上强调的是构成软件系统的组成元素 (构件)和连接关系 (连接器)的类型、构件和连接器间的组织方式以及系统的属性。他们采用基于不同连接方式的分类方式首次将熟知的软件体系结构风格分为数据流系统、调用与返回系统、独立构件系统、虚拟机和以数据为中心的系统五大类。

Lee 等人则将软件体系结构风格定义为是对构件类型、构件运行时的控制方式与 /或构件间数据传递的描述。一种体系结构风格可看做是在结构上有关构件类型约束及构件间交互约束的一个约束集合，这些约束可以定义一个系统结构族集来满足。

可见, Lee 等人在软件体系结构风格上不仅关注构件在体系结构中受到的静态约束，更关注运行时在体系结构中的动态约束。这在之前关于软件体系结构风格定义中从未提到过。在区别软件体系结构风格与软件体系结构时，他们与 Perry 等人持类似的观点，均认为软件体系结构风格与软件体系结构没有明确的界限，只是相对于描述比较具体的软件体系结构而言，软件体系结构风格更抽象、更概括。

软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式。它反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。按这种方式理解，软件体系结构风格定义了用于描述系统的术语表和一组指导构件系统的规则。

对软件体系结构风格的研究和实践促进了对设计的复用，一些经过实践证实的解决方案也可以可靠地用于解决新的问题。体系结构风格的不变部分使不同的系统可以共享同一个实现代码。只要系统是使用常用的、规范的方法来组织，就可使别的设计者很容易地理解系统的体系结构。例如，如果某人把系统描述为“客户/服务器”模式，则不必给出设计细节，我们立刻就会明白系统是如何组织和工作的。

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四要素**内容，即**提供一个词汇表、定义一套配置规则、定义一套语义解释原则**和**定义对基于这种风格的系统所进行的分析**。

体系结构风格分类：

Shaw 等人采用基于不同连接方式的分类方式首次将熟知的软件体系结构风格分为：

- 数据流风格：批处理、管道/过滤器
- 调用/返回风格：主/子程序、OO、层次结构
- 独立构件风格：进程系统、事件系统
- 虚拟机风格：解释器、基于规则的系统
- 仓库风格：数据库、超文本、黑板系统

###3.2 经典软件体系结构###

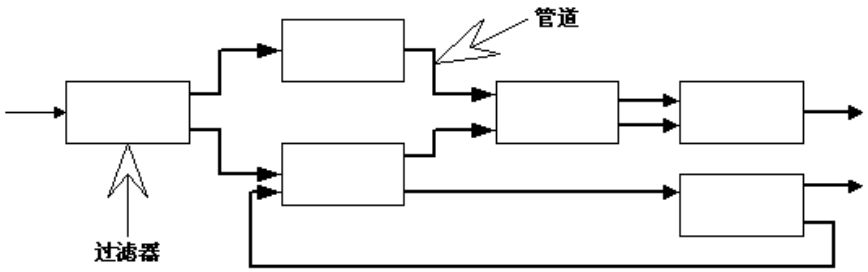
1)、管道和过滤器

在管道/过滤器风格的软件体系结构中，每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全消费之前，输出便产生了。因此，这里的构件被称为过

滤器，这种风格的连接件就象是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。

此风格特别重要的是过滤器必须是独立的实体，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的标识。一个管道/过滤器网络输出的正确性并不依赖于过滤器进行增量计算过程的顺序。

下图是管道/过滤器风格的示意图。



一个典型的管道/过滤器体系结构的例子是以 Unix shell 编写的程序。Unix 既提供一种符号，以连接各组成部分(Unix 的进程)，又提供某种进程运行时机制以实现管道。

另一个著名的例子是传统的编译器。传统的编译器一直被认为是一种管道系统，在该系统中，一个阶段(包括词法分析、语法分析、语义分析和代码生成)的输出是另一个阶段的输入。

管道/过滤器风格的软件体系结构具有许多很好的特点：

(1)使得软构件具有良好的隐蔽性和高内聚、低耦合的特点；

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

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

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

(5)允许对一些如吞吐量、死锁等属性的分析；

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

但是，这样的系统也存在着若干不利因素。

(1)通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计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。

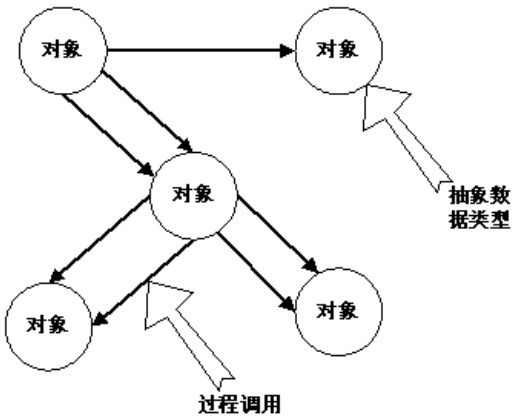
(3)因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这样就导致了系统性能下降，并增加了编写过滤器的复杂性。

2)、数据抽象和面向对象组织

抽象数据类型概念对软件系统有着重要作用，目前软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。

这种风格的构件是对象，或者说是抽象数据类型的实例。对象是一种被称作管理者的构件，因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

下图是数据抽象和面向对象风格的示意图。



面向对象的系统有许多的优点，并早已为人所知：

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

(2)设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

但是，面向对象的系统也存在着某些问题：

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

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

3)、基于事件的隐式调用

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

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

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

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

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

隐式调用系统的主要优点有：

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

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

隐式调用系统的主要缺点有：

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

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

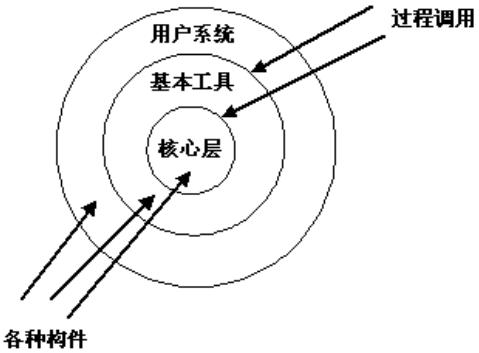
(3)既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。

4)、分层系统

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

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

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



层次系统有许多可取的属性：

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

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

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

但是，层次系统也有其不足之处：

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

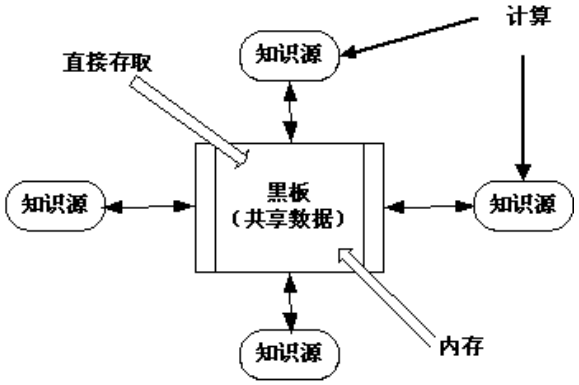
(2)很难找到一个合适的、正确的层次抽象方法。

5)、仓库系统及知识库

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

控制原则的选取产生两个主要的子类。若输入流是某类时间触发进程执行的选择，则仓库是传统型数据库；另一方面，若中央数据结构是当前状态触发进程执行的选择，则仓库是黑板系统。

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



我们从图中可以看出，**黑板系统主要由三部分组成：**

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

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

(3)控制。控制完全由黑板的状态驱动，黑板状态的改变决定使用的特定知识。

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：**

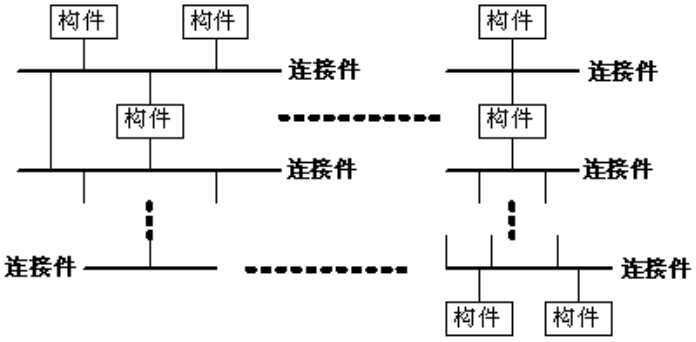
(1)系统中的构件和连接件都有一个顶部和一个底部;

(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连到另一个的顶部。

下图是 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。



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，我们可以得出，**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。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。

###3.3 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/服务器计算模型的演变。

在软件体系结构的风格设计中，客户/服务器（Client/Server，C/S）风格无疑曾是最重要的风格。客户/服务器计算技术在信息产业中占有重要的地位。

C/S 软件体系结构，即 Client/Server（客户机/服务器）结构，是基于资源不对等，且为实现共享而提出来的。

C/S 结构以局域网为中心，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。

服务器端的主要任务：

（1）数据库安全性的要求。

（2）数据库访问并发性的控制。

（3）数据库前端的客户应用程序的全局完整性规则。

（4）数据库的备份与恢复。

客户端的主要任务：

（1）提供用户与数据库交互的界面。

（2）向数据库服务器提交用户请求，并接收来自数据库服务器的信息。

（3）利用客户应用程序对存在于客户端的数据进行应用逻辑要求。

网络的主要任务：数据传输。

客户/服务器应用模式的特点是大都基于“胖客户机”结构下的两层结构应用软件。

优点：

（1）模型思想简单，易于理解和实现。

（2）C/S 结构的优点是能充分发挥客户端 PC 的处理能力，很多工作可以在客户端处理后再提交给服务器，客户端响应速度快。

缺点：

（1）客户端很庞大，以致于应用程序升级和维护时十分困难且耗资很大。

（2）事务层不能与跨平台的客户端共享。

（3）孤立了不同的逻辑组件。

（4）没有统一的数据逻辑层来提供不同种类的数据存储层。

###3.4 三层 C/S 结构风格###

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。

三层 C/S 体系结构是**将应用功能分成表示层、功能层（业务逻辑层、业务层、逻辑层、商务层等）、数据层三个部分**。

（1）表示层

表示层是应用的用户接口部分，它承担系统与用户间的对话功能，用于简单检查（不涉及有关业务本身的处理逻辑）用户输入的数据，显示应用的输出数据。在变更用户接口时，只需改写显示控制和数据检查程序，而不影响其他两层。

（2）功能层

功能层又称业务逻辑层，它将具体的业务处理逻辑编入程序中。表示层和功能层之间的数据传输要尽可能简洁。

（3）数据层

数据层就是数据库管理系统，负责管理对数据库数据的读写。

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。

与传统的二层结构相比，**三层 C/S 结构具有以下优点：**

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

（2）允许更灵活有效地选用相应的平台和硬件系统，使之在处理负荷能力上与处理特性上分别适应于结构清晰的三层。并且这些平台和各个组成部分可以具有良好的可升级性和开放性。

三层 C/S 结构具有以下缺点：

（1）开发难度加大。

（2）访问效率降低。

###3.5 浏览器/服务器风格（B/S）###

B/S 风格是上述三层 C/S 结构的一种实现方式，其体系结构为：浏览器/Web 服务器/数据库服务器。

B/S 体系结构主要是利用不断成熟的 WWW 浏览器技术，结合浏览器的多种脚本语言，用通用浏览器就实现了原来需要复杂的专用软件才能实现的强大功能，并节约了开发成本。

基于 B/S 体系结构的软件，系统安装、修改和维护全在服务器端解决。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 体系结构也有许多不足之处，例如：

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

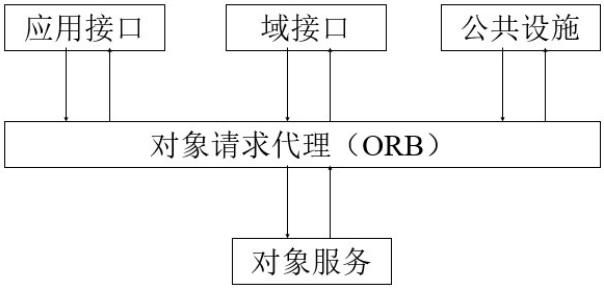
（2）B/S 体系结构的系统扩展能力差，安全性难以控制。

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

（4）B/S 体系结构的数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理（OLTP）应用。

CORBA（Common Object Request Broker Architecture 公共对象请求代理体系结构）是由 OMG 组织制订的一种标准的面向对象应用程序体系规范。或者说 CORBA 体系结构是对象管理组织（OMG）为解决分布式处理环境(DCE)中，硬件和软件系统的互连而提出的一种解决方案；OMG 组织是一个国际性的非盈利组织，其职责是为应用开发提供一个公共框架，制订工业指南和对象管理规范，加快对象技术的发展。 OMG 组织成立后不久就制订了 OMA(Object Management Architecture，对象管理体系结构)参考模型，该模型描述了 OMG 规范所遵循的概念化的基础结构。OMA 由**对象请求代理 ORB、对象服务、公共设施、域接口和应用接口**这几个部分组成，其核心部分是对象请求代理 ORB（Object Request Broker）。

###3.6 公共对象请求代理体系结构###



对象请求代理（Object Request Broker，ORB）：提供了一种机制，通过这种机制，对象可以透明的发出请求和接收响应。分布的、可以互操作的对象可以利用 ORB 构造可以互操作的应用。

对象服务（Object Services）：为使用和实现对象而提供的基本对象集合，这些服务应独立于应用领域。主要的 CORBA 服务有：名录服务（Naming Service）、事件服务（Event Service）、生命周期服务（Life Cycle Service）、关系服务（Relationship Service）以及事务服务（Transaction Service）等。这些服务几乎包括分布系统和面向对象系统的各个方面，每个组成部分都非常复杂；

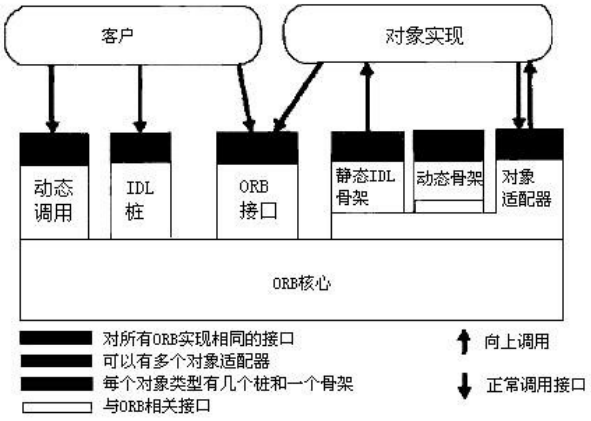
公共设施（Common Facilitites）：向终端用户提供一组共享服务接口，例如系统管理、组合文档和电子邮件等；

域接口（Domain Interfaces）：是为应用领域服务而提供的接口，如 OMG 组织为 PDM 系统制定的规范；

应用接口（Application Interfaces）：是由开发商提供的产品，用于它们的接口，不属于 OMG 标准的内容，相应于传统的应用层表示，处于参考模型的最高层。

CORBA 主要包括以下几个部分：

- ORB 核心(core)—软总线
- OMG 接口定义语言— IDL
- 语言映射—C++或 Java
- 接口仓库或实现仓库—动态调用
- 静态桩和骨架—静态调用
- 动态桩和骨架—动态调用
- 对象适配器(object adapter)
- ORB 之间的互操作—GIOP/IIOP



ORB 能识别的协议是 OMG 的接口定义语言 IDL。它可完整地确定部件的接口——即客户所需使用接口的全部信息。

IDL 定义的接口经 IDL 编译器编译后能产生客户的桩(stub)及执行对象的骨架(skeleton)这类能与 ORB 通信的接口：

(1)接口桩 stub。在编译时确定的静态接口。它位于客户对象的本地，接受客户的请求，对客户来说相当于远程的执行对象。接口桩向 ORB 提交请求。

(2)动态调用接口(DII)。用于编译时不能确定的请求。与接口桩作用相同。

(3)骨架 Skeleton。分动态骨架与静态骨架。它针对执行对象来说代表了远程客户的作用，可在本地调用执行对象服务，并与 ORB 通信。

(4)对象适配器。提供执行对象可以访问 ORB 服务的基本接口，其作用是产生及解释对象引用、安全交互、登记和执行等等。

1、对象请求代理(ORB，Object Request Broker)的作用

在传统的客户/服务器程序中，开发者使用他们自己设计的或者公认的标准定义设备之间的协议。协议的定义依赖于实现的语言、网络的传输和其他许许多多因素。

ORB 是一个中间件，他在对象间建立客户-服务器的关系。通过 ORB，一个客户可以很简单地使用服务器对象的方法，这个服务对象可以在本地，也可以在通过网络连接的其他机器上。

ORB 截获这一调用，同时负责查找实现服务的对象并向其传递参数、调用方法并返回最终结果。客户不用知道对象在哪里、它的编程语言和操作系统是什么，也不知道不属于对象接口的其他系统部分。

总结起来，**ORB 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。

2、ORB 的结构及类型

ORB 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(Adapter)的协调，ORB 可以向客户机和具备服务功能的对象实现(Object Implementation)提供服务。

作为 CORBA 体系结构的核心，**ORB 可以实现如下三种类型的接口：**

●对于所有 ORB 实现均相同的接口；

●指定于特定对象类型的操作；

●指定于对象实现的特定形式的操作。

不同的 ORB 可以采用不同的实现策略，加上 IDL 编译器，库和不同的对象适配器，这一切提供了一系列对客户的服务和对具有不同属性对象的实现。

可以存在多个 ORB 实现，它们有不同的名称和不同的实现方法与调用方法。

基于 ORB 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 ORB 实现通信管理的对象引用。

在实际应用中，只要遵循公共的 ORB 体系结构，程序设计可以选择 ORB 的多种实现方式，其中包括：

(1)客户和实现驻留(Client-Implementation Resident)ORB：采用驻留在客户和服务对象实现程序的方式实现 ORB。在这种实现方式下，客户端可以通过桩(Stub)程序，以位置透明的方式向具体的实现对象提出服务请求，实现客户与服务对象的通信。

(2)基于服务(Server-based)ORB：客户对象和实现对象均可以与一个或多个服务对象进行通信，服务对象的功能是将请求从客户端发送到对象实现。在这种方式中，ORB 的作用是完成客户对象与实现对象的通信，为对象之间的交互提供服务。

(3)基于系统(System-based)ORB：在这种实现方式中，ORB 被操作系统认为是系统所提供的一项基本服务。由于操作系统了解调用方与服务对象的位置，因而可以充分地实现 ORB 功能的优化。

(4)基于库(Library-based)ORB：如果认为对象实现可以共享，则可以将实现功能放入实现库(Implementation Repository)中，从而创建基于库的 ORB。

3、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的接口。IDL 本身也是面向对象的。

你可以用 IDL 定义 types（类型），constans（常量）和 interfaces（接口）。这与 C++中定义类型、常量和类相似。IDL 是定义界面和类型的语言，它没有供你编写实现部分的元素。

IDL 提供的数据类型有：基本数据类型、构造类型、模板类型、和复合类型、操作说明。这些类型可以用来定义变元的类型和返回类型，操作说明则可以用来定义对象提供的服务。

IDL 还提供模块构造，其中可以包含接口，而接口是 IDL 各类型中最重要的，它除了描述 CORBA 对象以外，还可以用作对象引用类型。

（2）接口池（接口仓库）

CORBA 引入接口仓库(Interface Repository)的目的在于使服务对象能够提供持久的对象服务。将接口信息存入接口仓库后，如果客

户端应用提交动态调用请求（Dynamic Invocation），ORB 可以根据接口仓库中的接口信息及分布环境下数据对象的描述，获取请求调用所需的信息。接口仓库作为 CORBA 系统的组成部分，管理和提供到 OMG IDL 映射接口定义的访问。

接口仓库中信息的重要作用是连接各个 ORB，当请求将对象从一个 ORB 传递给另一个 ORB 时，接收端 ORB 需要创建一个新对象来代表所传递的对象，这就需要在接收端 ORB 的接口仓库中匹配接口信息。通过从服务请求端 ORB 的接口仓库中获得接口标识，就可以在接收端的接口仓库中匹配到该接口。

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。在实现过程中，可以选择对象永久存在还是引用时再创建等方式。

在接口仓库的实现形式中，接口仓库中对象定义的形式是公开的，库中对象定义的信息可以供客户端和服务端使用。应用程序开发人员可以在如下方面使用接口功能：

- 管理接口定义的安装和部署；
- 提供到高级语言编译环境的接口信息；
- 提供终端用户环境的组件。

（3）动态调用接口

动态调用接口（DII，Dynamic Invacation Interface）允许在客户端动态创建和调用对服务对象的请求。一个请求包括对象引用、操作和参数列表。

（4）对象适配器

对象适配器(Object Adapter)是为服务对象端管理对象引用和实现而引入的。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。

对象适配器完成如下功能：

- 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；
- 激活或撤消对象的实现；
- 注册服务功能的实现；
- 确保对象引用的安全性；
- 完成对服务对象方法的调用。

（5）上下文对象

上下文对象(Context)包含客户机、运行环境或者在请求中没有作为参数进行传递的信息，上下文对象是一组由标识符和相应字符串对构成的列表，程序设计人员可以用定义在上下文接口上的操作来创建和操作上下文对象。

上下文对象可以以永久或临时方式存储，客户机应用程序用上下文对象来获取运行环境；而 ORB 用上下文对象中的信息来决定服务器的定位及被请求方法激活。

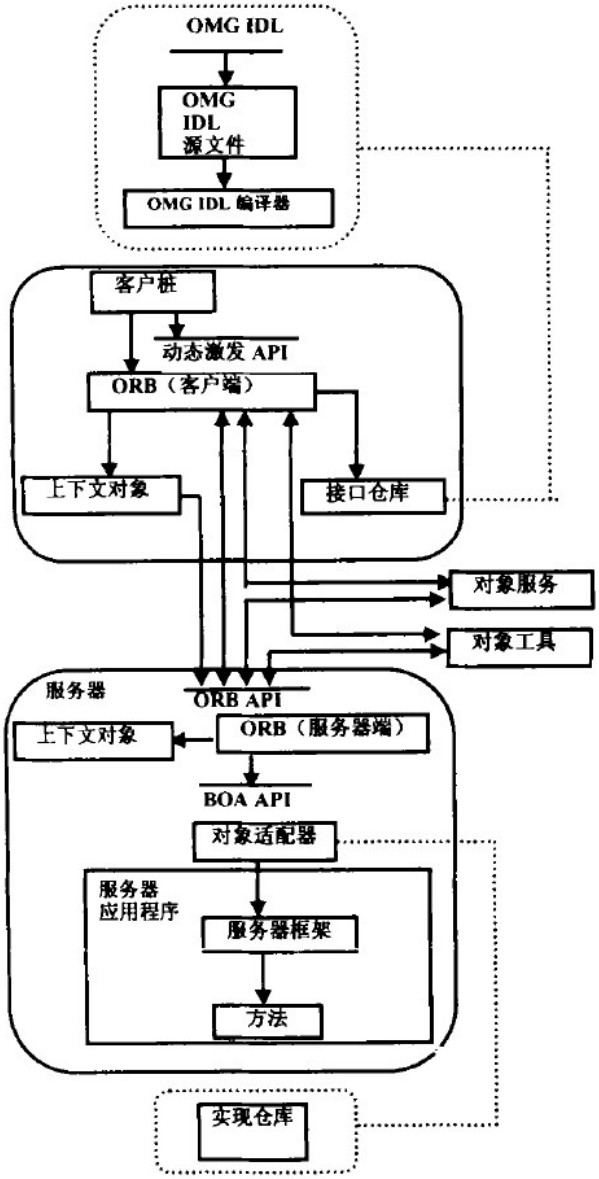
4、服务请求的实现方式

下面是对 CORBA 结构风格的分析。

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、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 ORB 等部件，以及对象适配器等连接件。



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

很明显，客户机系统是独立于服务器系统，同理服务器系统也独立于客户机系统。

优点：

（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。

（2）实现客户与服务对象的完全分开。

（3）提供软总线。

（4）采用面向对象的软件实现方法。

不足：

（1）尽管有多家供应商提供 CORBA 产品，但是仍找不到能够单独为异种网络中的所有 CORBA 系统环境提供实现的供应商。不同的 CORBA 实现之间会出现缺乏互操作性的现象，从而造成一些问题；

（2）由于供应商常常会自行定义扩展，而 CORBA 又缺乏针对多线程环境的规范，对于像 C 或 C++这样的语言，源码兼容性并未完全实现。

（3）CORBA 过于复杂，要熟悉 CORBA，并进行相应的设计和编程，需要许多个月来掌握，而要达到专家水平，则需要好几年。

###3.7 正交软件体系结构###

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，它由组织层和线索的构件构成，不同线索中的构件之间没有相

互调用(即线索相互独立)。

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，它由完成不同层次功能的构件组成(可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。

正交软件体系结构基本思想是把应用系统的结构按功能的正交相关性，垂直分割为若干个线索(子系统)，线索又分为几个层次，每个线索由多个具有不同层次功能和不同抽象级别的构件构成。各线索的相同层次的构件具有相同的抽象级别。

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，而不会涉及到其他线索。这样，就把软件需求的变动局部化了，产生的影响也被限制在一定范围内，因此实现容易。

主要特征：

（1）正交软件体系结构由完成不同功能的 n（n>1）个线索（子系统）组成；

（2）系统具有 m（m>1）个不同抽象级别的层；

（3）线索之间是相互独立的（正交的）；

（4）系统有一个公共驱动层（一般为最高层）和公共数据结构（一般为最底层）。

###3.8 基于层次消息总线的体系结构风格###

正交软件体系结构的优点：

（1）结构清晰、易于理解。

（2）易修改、可维护性强。

（3）可移植性强、重用粒度大。

青鸟工程在“九五”期间，对基于构件一构架模式的软件工业化生产技术进行了研究，并实现了青鸟软件生产线系统。以青鸟软件生产线的实践为背景，提出了基于层次消息总线的软件体系结构风格（Jade bird hierarchy message bus-based style），简称 JB/HMB，设计了相应的体系结构描述语言，开发了支持软件体系结构设计的辅助工具集，并研究了采用 JB/HMB 风格进行应用系统开发的过程框架。

JB/HMB 风格的提出基于以下的实际背景：

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

（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。在此之前的程序设计中，通常使用一个大的分支语句控制程序的转移，对不同的输人情况分别进行处理，程序结构不甚清晰。基于事件驱动的编程模式在对多个不同事件响应的情况下，系统自动调用相应的处理函数，程序具有清晰的结构。

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

JB/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，如下图所示：

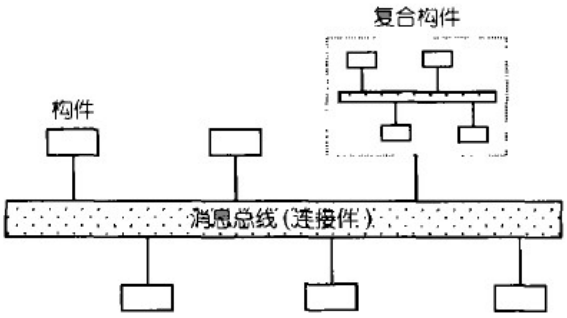


图 1　JB/HMB 风格的系统示意图

消息总线是系统的连接件，负责消息的分派、传递和过滤以及处理结果的返回。

各个构件挂接在消息总线上，向总线登记感兴趣的消息类型。构件根据需要发出消息，由消息总线负责把该消息分派到系统中所有对此消息感兴趣的构件，消息是构件之间通讯的唯一方式，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。

如图所示，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 JB/HMB 的风格，例如前面提到的数据流风格、面向对象风格及管道一过滤器风格等，这些属于构件的内部实现细节。

但要集成到 JB/HMB 风格的系统中，必须满足 JB/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。

系统和组成系统的成分通常是比较复杂的，难以从一个视角获得对它们的完整理解，因此一个好的软件工程方法往往从多个视角对系统进行建模，一般包括系统的静态结构、动态行为和功能等方面。例如，在 OMT 方法中，采用了对象模型、动态模型和功能模型刻画系统的以上 3 个方面。

借鉴上述思想，为满足体系结构设计的需要，JB/HMB 风格的构件模型包括了接口、静态结构和动态行为 3 个部分 。

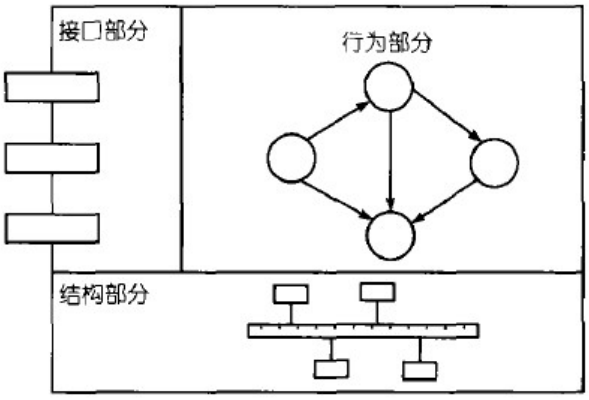


图 2　JB/HMB 风格的构件模型

在图中的构件模型中，左上方是构件的接口部分，一个构件可以支持多个不同的接口，每个接口定义了一组输入和输出的消息，刻画了构件对外提供的服务以及要求的环境服务，体现了该构件同环境的交互。

右上方是用带输出的有限状态自动机刻画的构件行为，构件接收到外来消息后，根据当前所处的状态对消息进行响应，并可能导致状态的变迁。

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

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。

JB/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，JB/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，降低了构件之间的藕合强度，进一步增强了构件的复用潜力，并使得构件的替换变得更为容易。

另外，在一般的互联接口定义的系统中，构件之间的连接是在要求的服务和提供的服务之间进行固定的匹配，而在 JB/HMB 的构件接口定义的系统中，构件对外来消息的响应，不但同接收到的消息类型相关，而且同构件当前所处的状态相关。构件对外来消息进行响应后，可能会引起状态的变迁。因此，一个构件在接收到同样的消息后，在不同时刻所处的不同状态下，可能会有不同的响应。

消息是关于某个事件发生的信息，上述接口定义中的消息分为两类。

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

（2）构件接收的消息，对系统中某个事件的响应或提供其他构件所需的服务。

接口中的每个消息定义了构件的一个端口，具有互补端口的构件可以通过消息总线进行通讯，互补端口指的是除了消息进出构件的方向不同之外，消息名称、消息带有的参数和返回结果的类型完全相同的两个消息。

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（一般的）过程调用，异步消息包括信号、时钟和异步过程调用等。

3、消息总线

JB/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。

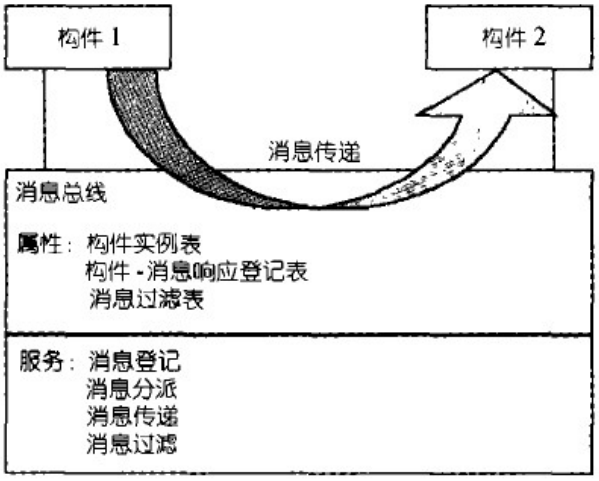


图 3　消息总线的结构

1）消息登记

在基于消息的系统中，构件需要向消息总线登记当前响应的消息集合，消息响应者只对消息类型感兴趣，通常并不关心是谁发出的消息。

在 JB/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”到构件的实现中，使得构件之间保持了灵活的连接关系，便于系统的演化。

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，这时应阻塞掉那些不希望接收到的消息。这就是需要显式进行消息登记的原因，以便消息响应者更灵活地发挥自身的潜力。

2）消息分派和传递

消息总线负责消息在构件之间的传递，根据构件-消息响应登记表把消息分派到对此消息感兴趣的构件，并负责处理结果的返回。

在消息广播的情况下，可以有多个构件同时响应一个消息，也可以没有构件对该消息进行响应。在后一种情况下，该消息就丢失了，消息总线可以对系统的这种异常情况发出警告，或通知消息的发送构件进行相应地处理。

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

消息总线是一个逻辑上的整体，在物理上可以跨越多个机器，因此挂接在总线上的构件也就可以分布在不同的机器上，并发运行。由于系统中的构件不是直接交互，而是通过消息总线进行通讯，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）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构件中使用了不同的名字，或不同的消息使用了相同的名字。前面我们提到，对挂接在同一消息总线上的构件而言，消息是一种共享的资源，这样就会造成构件集成时消息的冲突和不匹配。

消息转换是针对构件实例而言的，即所有构件实例发出和接收的消息类型都经过消息总线的过滤，这里采取简单换名的方法，其目标是保证每种类型的消息名字在其所处的局部总线范围内是唯一的。例如，假设复合构件 A 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）。我们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的集成问题。

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

4、构件静态结构

JB/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。

下图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，我们把 JB/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。

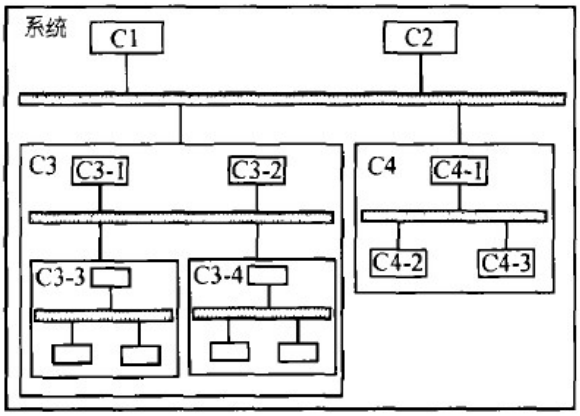


图 4　系统/复合构件的静态结构示意图

5、构件动态行为

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

更通常的情况是，构件的行为同时受外来消息类型和自身当前所处状态的影响。类似一些面向对象方法中用状态机刻画对象的行为，在 JB/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， JB/HMB 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。

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。

JB/HMB 风格方便地支持运行时刻的系统演化，主要体现在以下 3 方面：

（1）动态增加或删除构件。

在 JB/HMB 风格的系统中，构件接口中定义的输人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。一个构件加人到系统中的方法很简单，只需向系统登记其所感兴趣的消息即可。

但删除一个构件可能会引起系统中对于某些消息没有构件响应的异常情况，这时可以采取两种措施：一是阻塞那些没有构件响应的消息，二是首先使系统中的其他构件或增加新的构件对该消息进行响应，然后再删除相应的构件。

系统中可能增删改构件的情况包括：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剪，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。

（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。

###3.9 异构结构风格###

1、异构结构

上面介绍的是理想的“纯”软件体系结构风格，理解每一种风格的特点及优缺点对软件开发非常有帮助，但是应该注意的是，现实中的应用系统通常包含了不止一种风格。

体系结构风格的组合方式多种多样，主要有

（1）通过分层方法，即系统的部件以某种体系结构风格组织，而部件内部又以另一种风格实现；特别地，“连接”也可以逐层分解，如管道型连接的下层(内部)实现可以是先进先出方式的队列；

（2）一个部件有多个不同类型的接口，分别与不同风格的“连接”相连。例如，部件一方面可以访问仓库，另一方面可以通过“管道”与系统中的其它部件交互，同时还可以接收控制信息。unix 管道-过滤器系统就是这种风格，其中，文件充当了仓库的角色，初始化开关充当了控制的角色

（3）同一层中以完全不同的体系结构风格描述系统。

2、实例

C/S 与 B/S 两种结构混合使用是一种非常常见也是有效的异构体系结构风格。一种典型的 C/S 与 B/S 混合软件体系结构就是企业内部(局域网内)使用 C/S 结构，外部(Internet 用户)使用 B/S 结构，称之为“内外有别”模型。

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

3、异构组合匹配问题

两个构件不能协调工作的原因可能是它们事先做了对数据表示、通信、包装、同步、语法和控制等方面的假设，这些方面统称为形式。

（1）把一个构件的形式改变为另一个构件的形式；

（2）公布一个构件的形式的抽象化信息；

（3）在数据传递过程中从一个构件的形式转变到另一个构件的形式；

（4）通过协商，达成一个统一的形式；

（5）使一个构件支持多种形式；

（6）提供进口/出口转换器；

（7）引入中间形式；

（8）添加一个适配器或包装器；

（9）保持两构件的版本并行一致。

###3.10 互联系统构成的系统###

互联系统构成的系统是由 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。

1、定义

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联系统实现，而互联系统之间相互通信，履行系统的职责。其中一个系统体现整体性能，称为上级系统，其余系统代表整体的一个部分，称为从属系统。

2、软件过程

1）系统分解

2）用例建模

3）分析和设计

4）实现

5）测试

6）演化和维护

3、应用范围

1）分布式系统

2）很大或者很复杂的系统（大规模系统）

3）综合几个业务领域的系统

4）重用其他系统的系统（遗产系统的重用）

5）系统的分布式开发

###3.11 特定领域软件体系结构###

早在 20 世纪 70 年代就有人提出程序族、应用族的概念，1990 年 Mettala 提出了特定领域软件体系结构（domain specific software architecture, DSSA）的概念。

1、DSSA 定义

对 DSSA 研究的角度、关心的问题不同导致了对特定领域的软件体系结构不同的定义。

Rick Hayes-Roth：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集合。

Wlli Tracz：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。

Rick Hayes-Roth 侧重于 DSSA 的特征，强调系统有构件组成，适用于特定领域，有利于开发成功应用程序的标准结构；Wlli Tracz 更侧重于 DSSA 组成要素，指出 DSSA 应该包括领域模型、参考需求、参考体系结构、相应的支持环境或设施、实例化、细化或评估的方法与过程。两种 DSSA 定义都强调了参考体系结构的重要性。

DSSA 必备特征：

1）一个严格的问题域和/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。

领域：

1）垂直域

2）水平域

一般的软件过程针对某个特定软件系统，获取其需求，设计其构架。特定领域软件架构方法与此不同，它不以开发某个特定的应用为目标，而是关注于某个特定的领域，通过对某个特定领域的分析，提出了该领域的典型需求，得到相应的领域模型，设计相应的参考架构，实现其中的组成模块。在随后的特定应用开发过程中，对照应用需求和参考需求，配置参考架构，选取合适构件，完成该应用的开发。因而 DSSA 方法重点不是应用，而是重用，最终目的是开发一个领域中的一族应用。使用这种方法，有助于对问题的更加广泛而深刻的理解，有助于开发出面向重用的构架和构件，有助于提高软件生产率。

2、开发模型

DSSA 包含两个过程。

一是域工程，在域开发环境中由域专家确定域的范围，建立域模型，由软件工程师输出领域内可重用的软件资产，可重用组件等，为领域内的产品谱系提供一个灵活的、结构化的、系统性重用的基础；

二是应用工程，由应用工程师重用软件资产，以领域的通用体系结构为开发框架，开发一个系列应用软件的过程，主要包括完成需求分析、实例化参考体系结构、实例化类属组件、自动或半自动地创建系统等。

1）域工程

域是指一个问题或任务的范围，在其中可以开发多个高度相似的应用系统来满足不同用户的特殊需求。

域工程是识别和创建一组面向域的可重用组件的过程。域工程是特定领域软件体系结构的基础，其主要目标是系统化地将多个已存在的系统软件制品转化为软件资产，通过标识、构造、分类和传播面向域的可重用的软件资产，从而建立对现存和未来的软件系统都具有很强适用性的、高效的重用机制，使一个或一组产品谱系具有很好的可靠性、可理解性、可维护性和可使用性。

域工程主要包括以下四个部分：域分析、域设计、域实现和域传播。

域分析就是在一个特定的领域内，对域中一系列共性、个性、动态元素进行识别、收集和组织，并最终形成可指导软件重用模型的过程。

域分析搜集所有的关于软件资产的信息，帮助域工程师评估软件资产的可重用性，域分析的目的是定义正确的、可重用的域模型。

域分析的最终结果以域模型的形式表示出来。

域设计是在域分析的基础上，将域模型转化并映射到一个可实现的信息框架内，建立领域体系结构模型，为应用提供解决方案。

域体系结构模型需要从不同的方面来描述组件、组件间的关系、系统构造原理和指导原则。

域实现要根据领域体系结构模型创建可重用的软件组件。为了组件间的无缝连接，在实现上各个组件应该遵从一定的组件模型标准。

域传播就是要以无二义的、可分类的术语描述组件库中的组件，以便在开发软件产品时可以准确利用这些组件。

2）应用工程

应用工程是在域工程成果的基础上进行某一具体应用开发的过程，是对域模型的实例化过程，是为单个应用设计最佳解决方案，同其他的软件工程方法的步骤类似，只是每一步骤都是以域工程各阶段的成果为基础。这样，用户在开发应用系统时不必一切再从头开始。

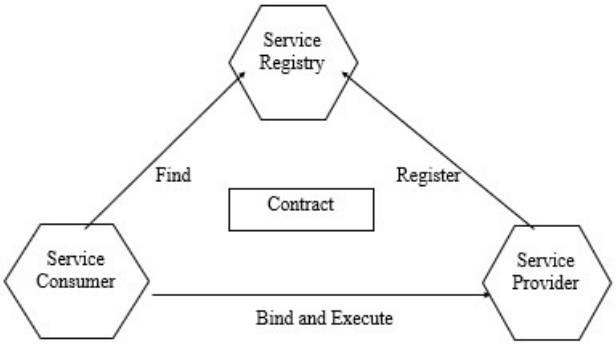
###3.12 SOA###

面向服务的体系结构（service-oriented architecture，SOA）来源于早期的基于构件的分布式计算方式，Allen 在《Component-Based Development for Enterprise Systems》一书中首次提到了服务的概念，他认为服务是将构件描述成提供相关服务的物理黑盒封装的可执行代码单元，它的服务只能通过一致的已发布的接口进行访问，构件必须能够连接到其他构件以构成一个更大的构件[1]。

JINI 的发展又进一步推动了 SOA 的早期概念体系的建立。1996 年国际咨询机构 Gartner 公司第一次阐述了 SOA 的概念，并认为未来 SOA 会成为占绝对优势的软件工程方法，从而结束已长达 40 多年的传统软件体系架构的主导地位。但是由于当时技术发展的限制，始终没有广泛应用。

近几年以来，随着 Web Service、XML 技术的发展，实现 SOA 时机逐渐成熟，并且由于 Microsoft、IBM 等厂商不遗余力的推崇，SOA 已经成为企业应用中的核心概念之一，并成为软件工程技术发展的重要趋势。

SOA 在刚开始的时候，它的关注点在企业应用集成和厂商相关的解决方案上，这个时期对于 SOA 的研究主要集中在软件系统集成和整合方面，紧接着 SOA 的发展逐步过渡到以 Web Service 以及 Web Service 组合为主要研究关注点，现在 SOA 的研究重点在业务流程管理以及业务流程的开放标准方面。



（1）服务提供者(Service Provider)。发布自己的服务,并且对使用自身服务的请求进行响应。

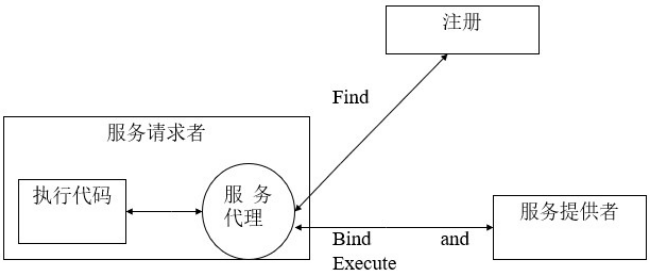
（2）服务注册(Service Registry)。注册已经发布的服务提供者,对其进行分类并提供搜索服务。

（3）服务请求者(Service Consumer)。利用服务代理查找所需的服务,然后使用该服务。

（4）协议（Contract）。是服务请求者和与它相互作用的服务提供者之间的方法说明。它对服务的请求和相应的格式格式化。一个服务协议要求一系列的前置条件和后置条件。前置条件和后置条件规定了为完成特定的功能，服务必须处在特定的状态。协议还声明

了服务质量水平。例如，服务质量的一个属性就是完成一个特定的服务方法所花费的总时间。

（5）服务代理（Service Proxy）。服务提供者为服务请求者提供一个服务代理。服务请求者通过调用代理上的应用接口函数来完成请求。服务代理在服务注册处找到服务提供者的协议和参数。接着服务代理格式化请求消息并完成代表服务请求者的请求。服务代理对服务请求者来说是一个很方便的实体。它不是必需的，服务请求的开发者通过开发需要的软件可以直接访问服务。



服务代理通过缓冲远程参数和数据可以提高性能。当代理缓冲了远程的参数，后面来的服务调用就不需要额外的注册调用。通过在本地存储协议，服务请求者减少了为完成服务所需的网络跳跃的次数。

此外，通过在本地完成一些功能，代理可以完全避免网络调用，从而提高性能。对那些不需要服务数据的服务程序来说，整个的程序就可以在代理本地完成。如果程序需要很少的服务数据，代理可以一次下载一小部分数据并可在接下来的程序调用中使用。程序在代理处完成而不是被发送到服务处完成的事实就是对服务请求者的透明度。然而，在使用这项技术时，代理只支持服务自己提供的程序，这是很重要的。代理设计模式表明代理只是对远程对象来说的本地参数。如果代理以任何方式改变了远程服务的接口，那么严格地说它就不再是代理了。

服务提供者会为不同的环境提供代理。服务代理用服务请求者的当地语言书写。例如，服务提供者可以为 Java, Visual Basic 和 Delphi 提供代理，如果这些是服务请求者最有可能使用的平台。虽然服务代理不是必需的，但它能大大地提高服务请求者的方便性和性能。

（6）服务租借。是服务注册给予服务请求者的，规定了协议有效的总时间：从服务请求者向注册发出请求到租借所规定的时间。当租借时间被用完，服务请求者就必须向服务注册请求新的租借。

租借对需要维持请求者和提供者间绑定信息状态的服务是必需的。租借定义了状态可能保持的总时间。通过限制请求者和提供者间绑定的总时间，租借大大降低了服务请求者和服务提供者间的耦合。如果没有租借的概念，一个请求者可能会永久地绑定在一个服务上，并永不再重绑定在其它的协议上。这就会导致服务请求者和服务提供者间更加紧的耦合。

有了服务租借，如果开发者需要实现一些改变，他就可以在服务请求者拥有的租借用完时来达成。实现可以改变，而不影响服务请求者的完成，因为这些请求者可以请求一个新的协议和租借。当获得新的协议和租借时，他们并没有保证与以前的完全一致，他们可能已经改变了，并且理解和处理这些改变是服务请求者的责任。

SOA 体系结构中的组件必须具有上述一种或多种角色，在这些角色之间使用了三种操作：

(1) 发布(Publish)。使服务提供者可以向服务代理注册自己的功能及访问接口

(2) 查找(Find)。使服务请求者可以通过服务代理查找特定种类的服务。

(3) 绑定(Bind and Execute)。使服务请求者能够真正使用服务提供者提供的服务。

理论上，面向服务的体系结构这种思想，在其简易性上十分吸引人。如果能够用定义很好的机制封装应用，就有可能将一个单一的应用加入到一个服务的集合中。封装的过程创建了一个抽象层，屏蔽了应用中复杂的细节(你将不必关心用的是哪一种编程语言，什么操作系统，应用程序用的是什么数据库产品)。唯一相关的就是服务所描述的接口。