
软件体系结构

——原理、实践与思维

沈 军 著

junshen

junshen

东南大学出版社

第1章 概述

本章主要给出软件体系结构的定义，解析软件体系结构的重要性及涉及的主要内容。在此基础上，给出本书的组织结构及应有的学习策略。

1.1 什么是软件体系结构

所谓**体系结构**（或**体系**，Architecture），是指一种思想的抽象，它一般包含两个方面，一个是抽象级别较高的概念层次内容，主要是指这种思想核心理念、相关原则和方法等；另一个是抽象级别相对较低的技术层次内容，主要是指其作用的宿主系统的基本组成元素及其相互关系的定义及描述。显然，前者本质上是强调体系结构的“体系”涵义，后者本质上是强调体系结构的“结构”涵义。深入而言，从逻辑上看，前者是体系结构的核心要素，后者仅仅是前者的一种具体表现，用于诠释前者的内涵。

软件体系结构（Software Architecture）是指一系列关于软件系统组织的重大决策（即原则、思想），以及软件系统的基本组成要素及其相互关系，是软件系统结构的抽象，由软件元素、元素的外部可见属性及元素间的关系以及相关的设计准则组成。显然，软件体系结构是体系结构概念在软件上的投影或具体应用，软件系统成为体系结构作用的宿主系统。

对于体系结构，理解的核心在于抽象程度。首先，体系结构并不等同于系统结构的具体设计说明，而是该说明的一种抽象描述与表达。其次，根据上下文或特定应用背景，要区分究竟是侧重于“体系”还是侧重于“结构”。并且，对于“结构”，还应注意它是指“体系”的内部要素，还是指“体系”的外部认知框架（即体系自身的结构），由此决定其具体内涵。

1.2 为什么要研究软件体系结构

得益于计算机工具的特殊结构，计算机应用一般都是通过构造软件系统实现。随着人类文明信息化程度的深入，企业IT应用越来越复杂，导致软件系统越来越庞大。众所周知，如果要建造一个摩天大厦，首先必须给出其设计蓝图，然后才能在设计蓝图的指导下，一步步按照工程要求进行建造。绝不是仅仅凭想象随意施工。与之相似，如果要构造一个庞大、复杂的软件系统，显然也必须在具体实施之前进行其蓝图设计，即进行软件体系结构设计。这是复杂大系统构造方法与简单小系统构造方法之间的本质区别。

另外，相对于其他系统而言，软件系统有其特殊性。一方面，随着人们对软件认识程度的不断深入，软件构造的基本方法和技术不断发展，目前，已诞生了各种各样异质的方法和技术。并且，应用系统赖以执行的基础环境，包括硬件系统和系统软件平台，也存在异质性。另一方面，应用具有恒变性，业务规则多次被重新定义，新业务模型也屡屡出现。随着应用的不断发展，多年的反复改造使很多系统之间具有复杂的交叉依赖，异质和冗余度相当高。不同时期采用不同技术和平台构建的软件系统逐渐形成“信息孤岛”，企业应用程序环

境变成一个大杂烩，企业应用程序环境的维护和集成成本居高不下。特别是面对新世纪全球化、虚拟化的商业环境，许多企业由于不能及时提供所需功能而错失良机。因此，如何匹配技术的动态性和应用的动态性，显然必须从一个战略高度进行分析。也就是说，为了有效地“重用”现有系统并与之“协同”工作，及时开发新的业务功能，减轻成本压力，必须为企业计算建立十分“灵活”和“敏捷”的“完美”结构，使企业软件环境内部保持恒久的“有序度”。企业软件体系结构正是实现这一需求的有力武器。

更进一步地认识，软件体系结构也是软件自身发展的使然。按照事物发展的普遍规律，从软件发展的脉络来看，其目前正处于由初级阶段到高级阶段的过渡时期。软件体系结构的建立，以及以软件体系结构为核心的新一代软件开发方法学的研究与发展，标志着该学科的逐步成熟。

1.3 软件体系结构涉及的内容

软件体系结构研究源自于软件 engineering 研究，目前已基本上相对独立。事实上，软件工程主要关注软件开发的整个过程，涉及软件开发整个过程的各个方面，包括人员、资源和费用等等非技术因素，而软件体系结构主要关注软件系统本身的抽象结构定义和设计。从认识论层面，可以将软件工程和软件体系结构看做是针对软件开发的宏观视图和（某个）微观视图及其它们的辩证关系。

按照给出的软件体系结构定义，软件体系结构应该涉及基本软件构造范型、设计模式、基本风格、典型案例和描述与设计等内容。其中，基本软件构造范型是软件体系结构建立的基础和最小粒度的元素。不同的软件范型蕴涵了其直接支持的软件体系结构，同时，软件范型发展的轨迹也反映了软件体系结构发展的历程和进化理念，两者在思维本质上具有通约性。设计模式是面向对象软件设计中关于对象关系和结构的一种设计经验抽象，它能有效支持软件结构对于应用恒变特性的适应性，提高软件系统的维护能力。尽管设计模式来源于面向对象软件设计，但其思维本质对软件体系结构如何适应应用的恒变特性，也有同样的指导意义。软件体系结构中往往在某个局部大量采用设计模式，以重用久经考验的设计经验。软件体系结构基本风格抽象了一些经过实验验证、有效的软件体系结构基本设计方法，这些方法广泛应用于现实软件系统的设计之中。典型案例是指软件体系结构的各种具体实现，它集成了软件范型、设计模式和软件体系结构基本风格。软件体系结构描述是指通过一定的语言或符号，从形式上定义软件体系结构，实现软件体系结构蓝图的书面具现。从而，方便对软件体系结构进行交流、审核和分析验证。软件体系结构设计是指针对某个具体软件系统或应用，利用软件范型、设计模式和软件体系结构基本风格等知识以及相应描述手段，进行艺术创造，创建出能够满足该具体软件系统需求或应用需求的相应软件系统的一种完美蓝图。

软件体系结构所涉及的各部分内容，从逻辑上构成一个整体，如图 1-1 所示。

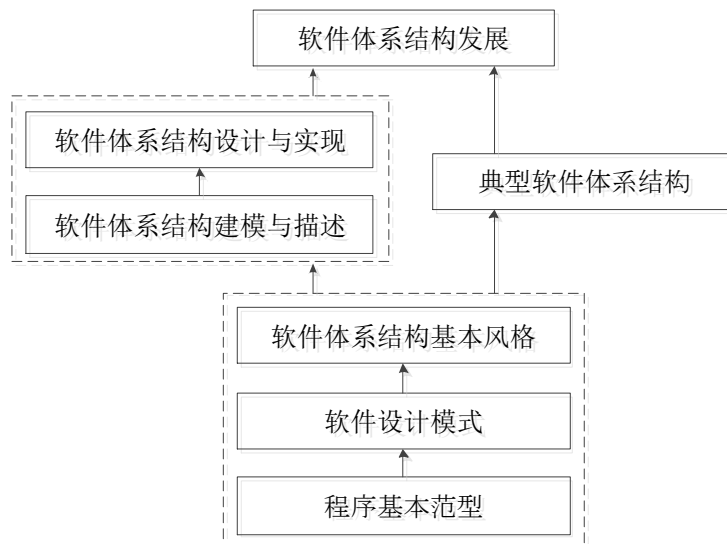


图 1-1 软件体系结构涉及的内容及其关系

1.4 本书的组织结构及学习策略

本书后面的章节，基于图 1-1 所示逻辑关系并叠加思维解析维度而展开。第 2 章到第 4 章，以及第 6 章构成一个认知单元，属于原理篇，从细粒度到粗粒度，从内核到外延，解析软件体系结构的多层次认知视图。第 5 章和第 7 章构成一个认知单元，属于实践篇，解析原理篇的具体应用。第 8 章构成一个认知单元，属于原理篇的进化，从内核程序范型和外延软件体系结构两个方面解析其相关的新发展。另外，针对每一章，最后都从思维层次给予了本章认知的深度进阶解析，这些内容采用横切方式分布于各章，从逻辑上构成隐式的思维篇。

针对本书的知识体系，宏观上可以采用面向应用和面向研究的两种基本学习策略。面向应用的学习策略，主要学习第 1-4 章、第 6 章的基本内容，以及第 5 章和第 7 章的内容，以理解目前软件开发领域中常用软件体系结构相关概念和知识及其对具体技术和产品的映射为核心，进行具体软件体系结构的设计与实现。面向研究的学习策略，主要学习第 1-4、6 章内容和第 8 章内容，并兼顾第 5、7 两章内容，以理解目前软件学科领域中软件体系结构相关概念、知识及其发展脉络及内在思维联系为核心，尤其是对隐式思维篇的学习和创新应用，进一步从认识论层次深刻理解软件体系结构发展轨迹内在的思维特征，以此把握软件体系发展的方向。通俗而言，面向应用的学习策略主要培养工程师，而面向研究的学习策略主要培养研究员。微观上，应该采用基于模式的学习策略，即重点认识具体范型、方法和技术背后所蕴涵的各种模式（隐性知识）及其在这些具体范型、方法和技术中的应用（模式建构），在模式层次进行学习。具体而言，映射到每一章内容的学习，首先学习各种具体模型、方法和技术，包括它（们）的概念、术语和基本原理等。然后，再深入一步，思考并发掘这些概念、术语和基本原理中共同采用的一些问题处理思想，例如分层处理思想、抽象粒度等。最后，还要在此基础上，对各种概念、术语和基本原理等进行比较和分析，认识它们之间的发展和演化的脉络及其原因，例如软件范型发展和演化的脉络及其原因等。针对本书整个内容的学习，应该在普通知

识（显性知识）和模式知识（隐性知识）两个层面展开学习。首先，在每个层面注意从各种具体知识点中形成知识链，例如普通知识层面中的知识关系、模式知识层面中的模式关系。然后，将两个知识面综合，形成知识球。从而，达到知识的触类旁通和融会贯通。

1.5 本章小结

软件体系结构对现代软件开发具有极其重要性。软件体系结构涉及内容和知识既抽象又具体，并且其形而上和形而下两极都开放。本章从定义、意义、内容和学习策略几个方面，基于系统化认知思维，为本书的展开及有效学习建立宏观的认知视图。

习 题

- 1、什么是软件体系结构？它一般包含哪两个方面的内容？两个方面的内容是什么关系？
- 2、如何理解软件体系结构和软件工程之间区别和联系？
- 3、如何理解软件范型、设计模式和软件体系结构基本设计风格三者之间的关系？
- 4、什么是应用的恒变特性？你认为对应用恒变特性的灵活适应是不是软件体系结构设计的核心，为什么？
- 5、什么是基于模式的学习策略？请举例说明。

拓展与思考：

- 6、对于 Architecture 的中文翻译，你认为到底应该是“体系”还是“体系结构”？为什么？

（提示：从中文构词规则角度，“体系结构”具有二义性，一是与“体系”等价（作为合成名词），另一个是“体系的结构”（不作为合成名词），是指“体系”本身的结构。对于后者，首先，“体系”的涵义既包括思想、原则和方法，也包括其作用的宿主系统的基本组成要素及其相互关系。其次，对这两者的统一认识或研究它们的关系就是“结构”的含义，即采用“结构化视图”去认识“体系”。因此，体系结构本质上是将“结构化思维”具体运用于“体系”。相对于“体系”所包含的“结构”涵义部分，“体系结构”是一种结构的结构。事实上，体系结构的核心在于“体系”，并侧重于“体系”中的“结构”含义，这是 Architecture 的本意，可以是指对“体系”（Architecture）的一种结构化的认知框架。）

- 7、建立一个精美的小狗窝与建立一个庞大的摩天大厦，在策略上究竟有何本质不同？
- 8、事物发展初级阶段和高级阶段分别基于什么样的思维特征？（提示：从人类认知的归纳思维和演绎思维两种基本思维方式角度考虑）

第2章 基础：基本程序范型

本章主要解析软件体系结构赖以建立的基础——程序构造基本范型。首先给出程序基本范型的定义，解析程序基本范型对软件体系结构的作用。然后，梳理程序基本范型发展脉络，解析各种程序基本范型的基本原理。最后，解析对程序范型的深入认识和思考。

2.1 什么是程序基本范型

所谓**范型**（Paradigm），一般是从事某一类科学活动所必须遵循的公认的模式。显然，**程序基本范型**是指程序构造所必须遵循的公认的基本模式，它定义了程序构造的基本原理及程序的基本结构形态。其中，基本原理是核心，属于方法论范畴，需要定义方法构建的基本要素及其关系。基本结构形态是基本原理的具体表现，诠释基本原理的思想和内涵。

程序基本范型是程序的一种抽象，它独立于各种具体程序，相对于具体程序的抽象模型，程序范型位于元模型抽象层次，其抽象级别高于具体程序的抽象模型。另外，程序基本范型也是程序设计语言赖以建立的基础，程序设计语言各种机制的建立本质上就是用于支持程序基本范型的实现。程序基本范型与程序设计语言的关系是多对多关系，一种范型可以由多种语言实现，一种语言也可以同时支持多种范型。

2.2 程序基本范型对软件体系结构的作用

程序基本范型既反映了软件体系结构构建的核心思想，也奠定了软件体系结构构建的基础。一方面，它定义了软件体系结构构建的基本单元元素的形态。另一方面，它定义了基本单元元素之间关系的基本形态。通过基本单元元素及其相互关系的定义，确定了软件系统构造的基本准则。因此，不同程序基本范型隐式地定义了软件体系结构构建的不同基本方法。

2.3 程序基本范型的发展脉络

审视程序基本范型从诞生到发展的历程，尽管各种程序基本范型的发展存在一定的时间交叉，但从其是否作为软件构造技术的主体支撑技术来看，程序基本范型的发展基本上符合如图 2-1 所示的轨迹。

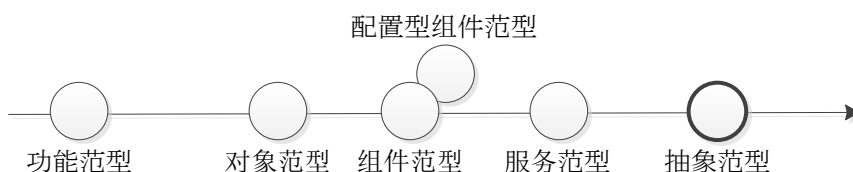


图 2-1 程序基本范型的发展轨迹

程序基本范型的发展脉络，也清晰地体现了计算机应用发展的历程以及计算机技术发展的历程。计算机应用发展和计算机技术发展两者相辅相成，一方面，计算机应用发展对计算机技术提出了新的要求，促进计算机技术发展；另一方面，计算机技术发展又为新型计算机应用发展提供了基础，促进计算机应用发展。作为计算机技术之一的软件技术，在计算机应用和其他计算机技术之间建立起桥梁。因此，程序基本范型的发展事实上也就是不断动态地粘合应用与技术。图 2-2 给出了计算机应用、计算机技术和程序基本范型的关系。

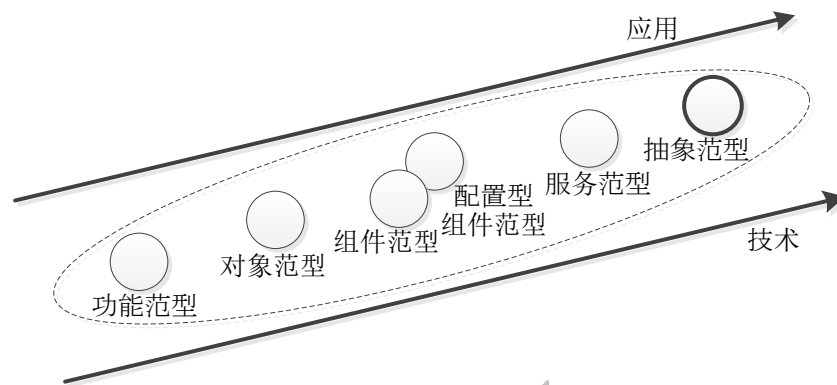


图 2-2 技术、应用和程序基本范型的关系

2.4 程序基本范型的原理解析

本节按照程序基本模型演化的轨迹，主要解析各种程序基本模型的基本原理及其思维本质，并阐述其对软件体系结构建立的影响。

2.4.1 功能范型

功能范型（Function Paradigm，也称为面向功能范型、功能模型）也可以称为面向过程范型（鉴于软件工程中已有过程模型的专有定义，本书不采纳该称谓）或函数式范型（鉴于计算机理论中已有函数式编程范型的专有定义，本书也不采纳该称谓），它是模型化软件构建方法的第一个基本范型。功能范型的基本原理是将一个系统分解为若干个基本功能模块，基本功能模块之间可以按需进行调用。基本功能模块集合及其调用关系集合构成一个软件系统的基本模型。

功能范型诞生于 20 世纪 60 年代，它强调了对程序两个 DNA 之一数据处理（功能）的抽象，通过功能分解和综合的方法，降低系统构造的复杂性。从而，实现一体式程序体系结构向结构化程序体系结构的转变，并建立了结构化程序设计方法，该方法是第一代模型化方法，称为面向功能的程序开发范型。

功能范型的核心之一，是基本功能模块的抽象及耦合。事实上，基本功能模块是一种处理方法的抽象，这种方法独立于其处理的具体数据集，建立在抽象的数据集上。通过将抽象数据集具体化，就可以实现处理方法在某个具体数据集上的作用，从而实现处理方法的重用。因此，基本功能模块的抽象一般需

要定义其处理的抽象数据集。具体实现中，基本功能模块一般有**函数**（Function）和**过程**（Procedure）两种形式，前者返回处理结果，后者不返回处理结果。抽象数据集称为形式参数，具体数据集称为实际参数。图 2-3 分别给出了基本功能模块在 Python 语言和 C++语言中的实现。

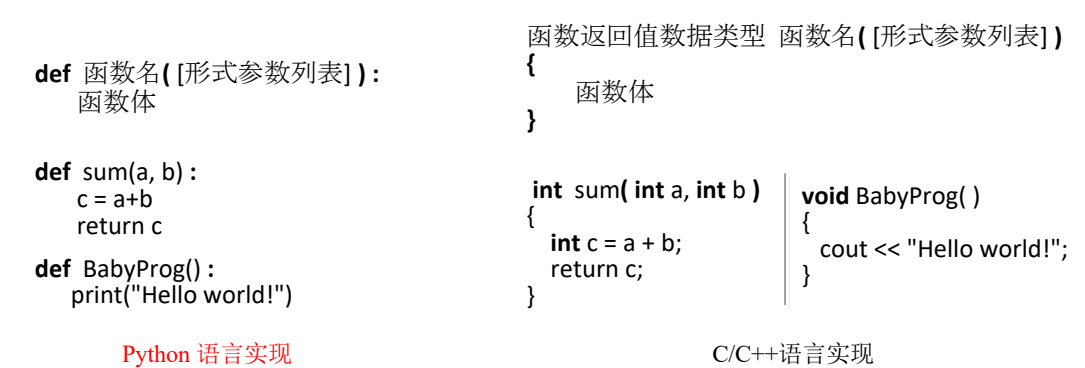


图 2-3 基本功能模块的实现

基本功能模块的耦合，是指一个模块调用另一个模块时如何实现被调模块抽象数据集的具体化以及被调模块如何返回其处理结果给主调模块。前者一般称为参数传递，后者称为函数调用返回。目前，考虑到模块嵌套调用的实际应用需求，参数传递和函数调用返回的实现方式基本上都是通过**堆栈**（Stack）机制进行，图 2-4 给出了参数传递和函数返回实现的基本思想。按照传递方式，传递基本上有**值传递**和**地址传递**两种形式。值传递将实际参数值（函数调用时）或函数处理结果值（函数调用返回时）复制到堆栈，地址传递则是将实际参数值（函数调用时）或函数处理结果值（函数调用返回时）存放的内存地址复制到堆栈。因此，当被调模块的抽象数据集具体化后，值传递方式不会因为被调模块的处理而改变原始的实际参数的值，而地址传递方式由于被调模块的处理会通过实际参数值存放的内存地址而间接地作用于原实际参数，从而改变原始的实际参数值。另外，对于函数调用返回，地址传递可能会带来错误隐患，因为程序设计语言中，为了提高内存使用效率，对模块中的数据往往采用按需即时存储分配策略。例如：C/C++语言中，地址传递会导致无效引用和内存泄漏错误。图 2-5 是两种参数传递方式的 C++语言实现。

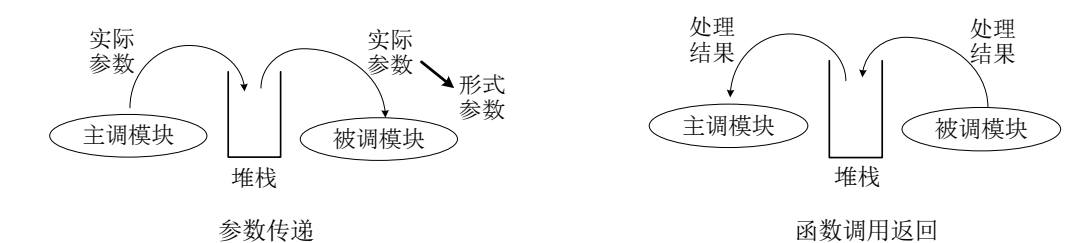


图 2-4 参数传递和函数调用返回实现的基本思想

<pre> int SubModule(int p, int q) { int r; r = p + q; p--; q++; return r; } int main() { int x = 10, y = 20, z = 0; z += SubModule(x, y); cout<<"The Result is:"<<z<<endl; cout<<"x="<<x<<" , "<<"y="<<y<<endl; } </pre>	<pre> int SubModule(int &p, int &q) { int r; r = p + q; p--; q++; return r; } int main() { int x = 10, y = 20, z = 0; z += SubModule(x, y); cout<<"The Result is:"<<z<<endl; cout<<"x="<<x<<" , "<<"y="<<y<<endl; } </pre>
--	--

The Result is: 30
x=10, y=20

The Result is: 30
x=9, y=21

图 2-5 两种参数传递方式的 C++语言实现

功能范型的核心之二，是递归思想的具体实现。所谓**递归**（Recursion），是指用同一种处理方法（该方法通过不断缩小待处理数据集规模）来处理缩小规模后的数据集，并通过不断综合小规模数据集的处理结果来得到大规模数据集处理结果的一种问题处理方法。图 2-6 给出了递归方法的基本思想。

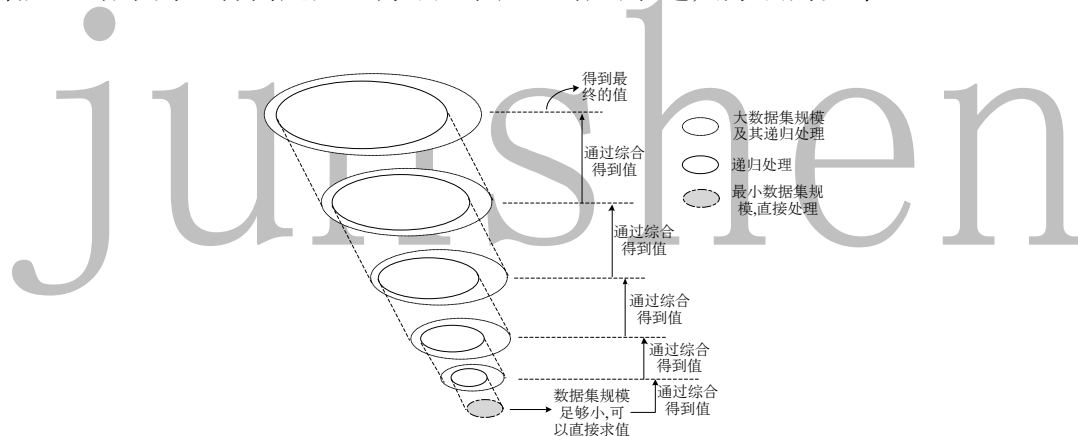
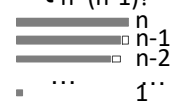


图 2-6 递归方法的基本思想

功能范型中，递归思想体现在两个方面：一个是基本功能模块的递归应用，另一个是处理逻辑或数据组织方式的递归应用。基本功能模块的递归应用，是将图 2-6 中的一种处理方法通过一个基本功能模块实现，将数据集规模作为基本功能模块的形式参数之一，这样在基本功能模块处理逻辑的定义中，显然需要在缩小后的数据规模上再调用其自身（使用同一种处理方法）。可见，递归是一种特殊的模块耦合关系，其主调模块和被调模块是同一个处理模块。图 2-7 给出了基本功能模块递归应用的一个具体案例。根据模块调用关系的不同，递归可以呈现多种具体应用形式，如图 2-8 所示。其中，相对于图 2-8（a），嵌套递归是一种“阶”拓展，其他都是“维”拓展。

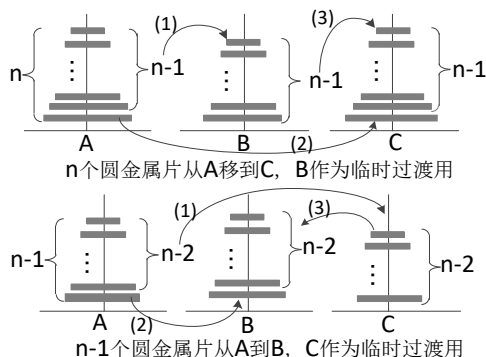
$$n! = \begin{cases} 1 & n = 0 \text{ 或 } 1 \\ n * (n-1)! & n > 1 \end{cases}$$


```

int Fac(int n)
{ // 求n的阶乘
  if (n == 0 || n == 1)
    return 1;
  return n*Fac(n-1);
}

```

(a) 阶乘问题求解（一维递归）



```

void Hanoi(int n, char from, char to, char temp)
{ // n个圆金属片从from柱移到 to柱,
  // temp柱作为临时用柱
  if (n > 0) {
    Hanoi(n-1, from, temp, to); → (1)
    cout<<n<<" : "<<from<<" -> "<<to<<endl;
    Hanoi(n-1, temp, to, from); → (3)
  }
}

```

(b) 汉诺塔问题求解（二维递归）

图 2-7 基本功能模块递归应用具体案例

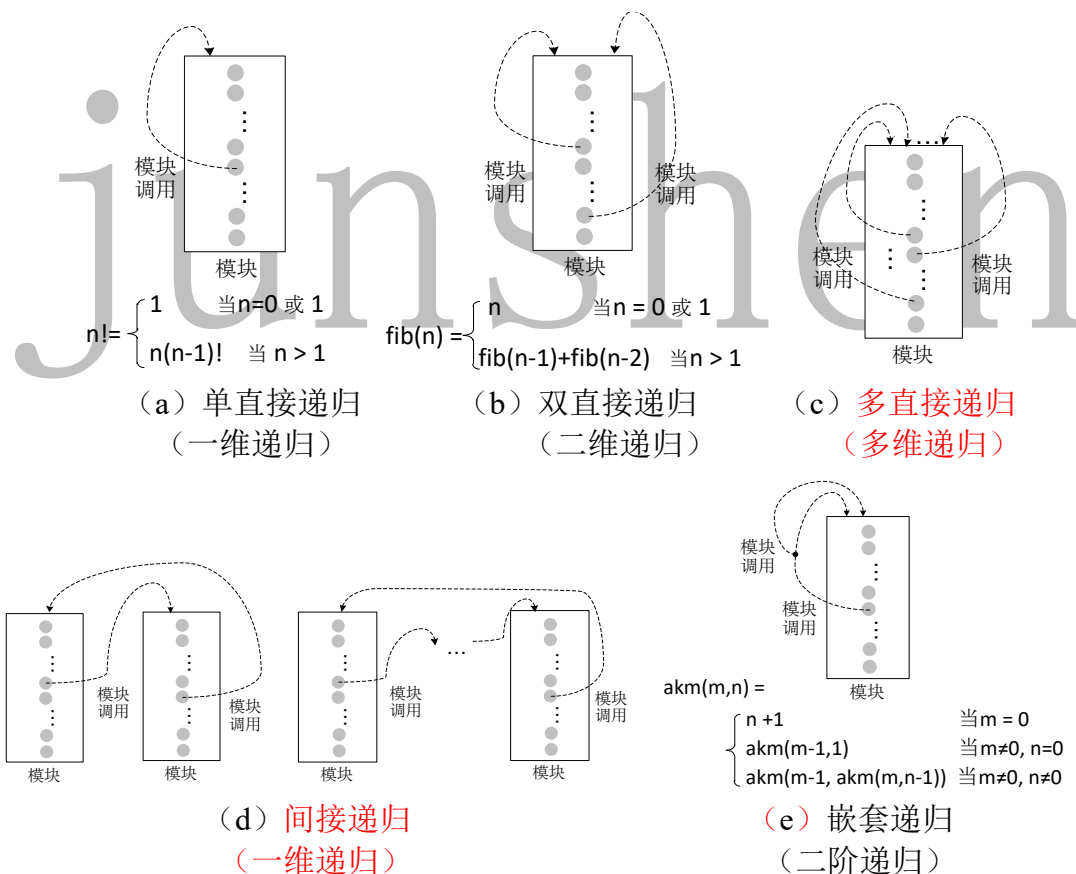


图 2-8 递归的多种具体应用形式

处理逻辑的递归应用，是指将问题的整个处理逻辑看作为数据集，将基本的处理逻辑看作为处理方法，从而实现用基本的处理逻辑及其递归组合来实现处理不同复杂度问题的整个处理逻辑。功能范型抽象了三种基本的处理逻辑：顺序、分支和循环。图 2-9 给出了它们的语义解析。图 2-10 解释了处理逻辑的

递归应用内涵。其中，程序 A（大程序）和子程序 B（小程序）在思维上具有显式通约性。因此，它演绎了大和小是辩证统一的哲学思想的具体应用。事实上，递归思想诠释了计算思维的本质内涵。

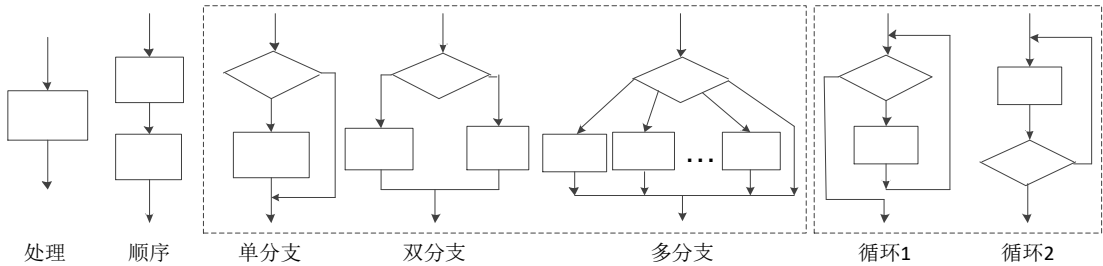


图 2-9 三种基本的处理逻辑及其语义

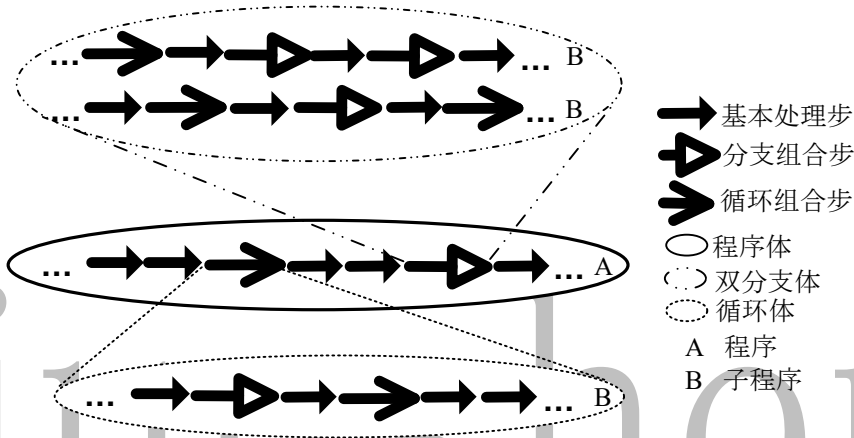


图 2-10 处理逻辑的递归应用内涵

数据组织方式的递归应用，是指将需要组织的全部数据看作为数据集，将基本的数据组织方法看作为处理方法，从而实现用基本数据组织方法及其递归应用来实现针对不同规模数据集的结构化组织。在计算机中，数据组织基本方法通过单个数据及其组合关系来构建，具体而言，首先给出单个数据的组织方法，然后给出数据之间的组合关系定义，这种方法本身具备自我演化能力，体现典型的递归应用特点。一般而言，单个数据具有不变与可变两者特性，与之对应，计算机中通过常量和变量来表达。对于数据关系，可以按需定义关系集。例如：C++语言中，数据关系有三种：绑定、堆叠和关联。另外，数据组织方式分为型和实例，型表示数据组织方式的形态和特性定义，是一个抽象的概念，计算机中一般通过**数据类型**（Data Type）实现。实例是指型的一个具体表现，是一个具体的实例或值。显然，一个型可以有多个不同的实例。例如：C++语言中，**复合数据类型**就是通过**基本数据类型**的递归应用而实现，具体解析参见图 2-11 所示。图 2-12 所示给出了 C++语言中数据组织递归应用的案例。

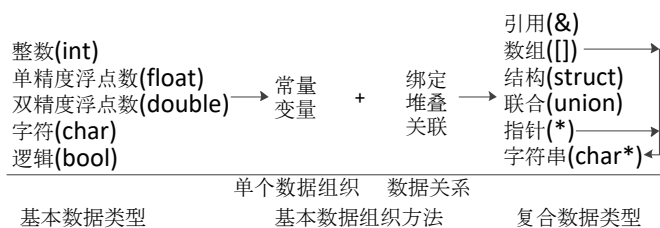


图 2-11 C++语言中基本数据组织方法

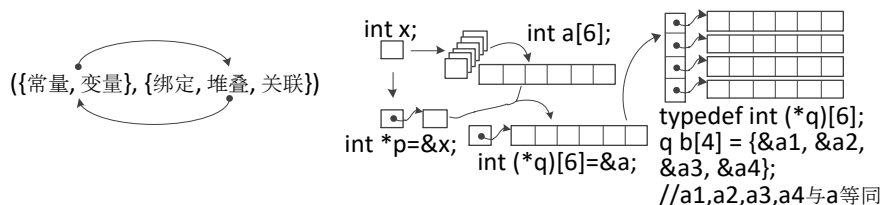


图 2-12 C++语言中递归数据组织的具体应用

由于功能范型则重于程序的数据处理（功能）部分，淡化了程序的数据组织部分以及数据组织与数据处理之间的关系，因此，对于大规模程序构造，功能范型具有其固有的（或天生的）缺陷——**数据波动效应**。所谓数据波动效应，是指如果一个模块修改了或调整了某个数据组织结构，但它没有及时通知其他与此数据组织结构相关的模块，则该数据组织结构的变动会引发意想不到的影响。这种现象会在整个程序中产生连锁反映（即波动效应），最终导致整个程序的不正确性。尽管功能范型存在固有缺陷，并由此失去其主流技术的地位，但其模型化构造方法的建立，以及模块化设计思想、递归构造思想的建立，对软件构造方法产生了深远的影响。

2. 4. 2 对象模型

对象模型（Object Paradigm，简称面向对象范型、对象模型）诞生于 20 世纪 80 年代，它以对象为核心，强调对程序中数据组织的抽象，并实现数据组织和数据处理的统一。在此基础上，建立面向对象的软件构造方法。对象范型的基本原理是将一个系统分解为若干个对象，对象之间可以通过发送消息按需进行协作。对象集合及其协作关系集合构成一个系统的基本模型。

所谓**对象**（object），是指客观世界中存在（已经存在或将要存在）的东西，可以是具体的（例如：人、桌子、书等），也可以是抽象的（例如：缓冲池、堆等）。在计算机中，为了描述一个对象（或以对象进行数据组织），显然必须给出对象的**型**，并按需建立对象的各个具体实例（称为对象的**值**）。例如：C++语言中，以**类**（class）描述一个对象的型，以变量描述对象的值。一个对象一般有静态属性和动态行为（即对象的职责），例如汽车有型号、颜色、价格、牌号、生产厂家等静态属性，有发动、刹车、转弯等动态行为等。因此，对象型的描述中必须将该类对象的静态属性和动态行为描述清楚。其中，静态属性对应于数据组织，动态行为对应于数据处理，从而以对象为核心，实现数据处理和数据组织进行统一。进一步，考虑到数据的隐私与保护，可以对属性和行为进行公开级别的控制，称为信息隐藏或封装。图 2-13 给出了对象描述的基本视图。图 2-14 给出了 C++语言中对象描述的一个具体案例。

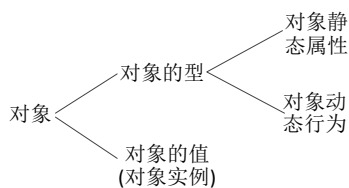
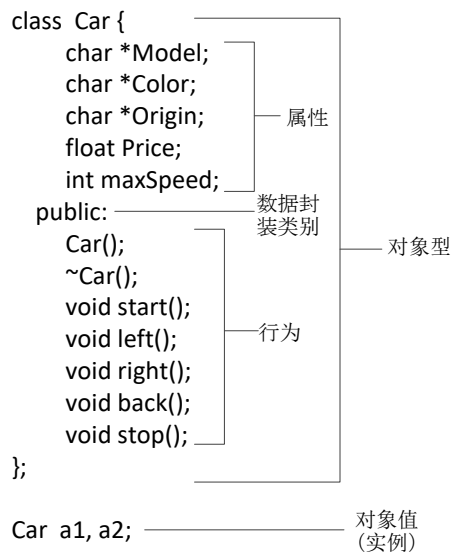


图 2-13 对象描述的基本视图



2-14 C++语言中对象描述的具体案例

对象范型的核心之一，是对数据类型的抽象与拓展。所谓数据类型的抽象，是指允许用户按需定义自己的数据类型并通过其进行数据组织。从而拓展某种程序设计语言固有的数据类型，为应用程序的构造带来灵活性。相对于传统固有数据类型，拓展的数据类型称为**抽象数据类型**（Abstract Data Type, ADT）。一般来说，一种数据类型既要规定其数据的取值范围，又要定义其数据的基本运算操作。因此，对象范型中的对象描述机制，可以满足数据类型定义的要求。其中，对象的属性描述对应于数据取值范围，对象的行为描述对应于数据基本运算操作。可见，对象本质上就是数据，对象描述就是定义一种数据类型。值得注意的是，抽象数据类型的定义体现了数据组织方式的递归应用特性。相对于功能范型中的复合数据类型，抽象数据类型的实现灵活性和扩展性要强得多。

对象范型的核心之二，是同构（或同族）对象关系的定义，这种关系体现在**继承**（Inheritance）和**多态**（Polymorphism）两个方面。所谓继承，是指同族对象中，后代可以共享前代的属性及行为特征。例如，麻雀作为子类可以是父类——鸟的一种类型，因此，麻雀可以继承鸟类的一些属性及行为特征。同时，麻雀也可以有自己的属性和行为特征；也可以改变父类的一些行为特征。所谓多态，是指对象的某种行为可以呈现不同的形态。根据不同形态表现的时机，多态可以分为静态多态和动态多态。静态多态是指在对象的具体实例建立前，对象的行为描述中就已经将各种形态的表现定义清楚。这种多态形式一般用于一个对象内部的行为描述中。动态多态是指各种形态的表现要在对象的具体实例建立后才能具体确定。这种多态形式一般用于具有继承关系的同族多个对象的行为描述中。此时，父类通常只给出抽象行为（即对象可以做什么），而不定义具体行为（即对象究竟如何做）。各个子类具体定义其对象应有的行为。这样当通过抽象引用概念性地要求对象做什么时，将会根据当前实例化的具体类（父类或子类）得到不同的行为，具体行为取决于对象的具体类型。例如，对于一个图形类，可以定义一个draw()行为。但是，对于直线、圆、矩形和椭圆等不同的图形（它们都是图形类的后代），显然各自的draw()会呈现出

不同的具体形态（绘制方法）。动态多态机制具有较强的灵活性，它提供了一种统一的控制机制。事实上，动态多态也可以看成是一种遗传变异，是作用在继承机制上的一种更细粒度的控制，是一种特殊的继承。C++语言中，静态多态通过函数重载（Function Overloading）机制实现，动态多态通过虚函数（Virtual Function）机制实现。

对象范型通过对象机制统一了数据组织和数据处理两个方面，并建立了抽象数据类型的构造方法。然而，其抽象级别仍然较低，认识视野仍然局限于实现层次，对概念层次和规约层次的重视不够，从而使得基于对象范型的面向对象设计思想和方法失去了应有的巨大效能。也就是说，抽象数据类型的具体实现仍然需要由具体程序设计语言来体现，对象不能独立于语言。另外，对象范型通过继承机制强化了同族对象关系，而对异族对象关系并没有显式说明。因此，尽管对象范型通过统一数据组织和数据处理两个方面可以解决功能范型的数据波动效应缺陷，但对于异构集成以及大规模软件开发，对象范型暴露出它的不足之处。

2.4.3 组件范型

组件范型（Component Paradigm，也称为面向组件范型、组件模型）诞生于 20 世纪 90 年代，它在对象范型基础上，强调了异族对象关系以及对象独立性问题。异族对象关系主要是指组件内部完成组件功能的各种对象不但是同族的，也可以是异族的。对象独立性是指组件建立在二进制基础上并独立封装，可以独立部署。组件范型的基本原理是以**接口**（Interface）为核心，通过接口抽象组件的行为。在此基础上，建立面向接口的软件构造方法。组件集合及其协作关系集合构成一个系统的基本模型。

所谓接口，是指对象动态行为的集合。接口也支持继承机制。因此，相对于对象范型，组件范型更加重视在概念层次和规约层次上认识面向对象的方法和思想，强调对象是一组责任，具有可以被其他对象或对象自身调用的方法（即行为）。也就是说，将数据组织部分封装在内部，不对外暴露。而将针对数据的处理部分以接口形式对外暴露，接口的具体实现也封装在内部，不对外暴露。

所谓组件，是指能完成特定功能并能独立部署的软件合成单元。一个组件一般具有一个或多个接口，每个接口的功能由一个或多个方法（或称为函数）来体现。接口的具体功能（即其每个方法的具体行为）由组件对象实现。组件对象之间可以通过聚合和委托方式进行功能重用。图 2-15 是组件的基本结构，图 2-16 是组件功能重用的两种基本方式。图 2-17a 是 Microsoft COM

（Component Object Model）组件的基本结构。COM 支持 DLL（Dynamic Link Library）和 EXE 两种封装结构。图 2-17b 是 COM 组件的运行时结构。

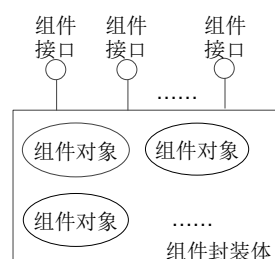


图 2-15 组件基本结构

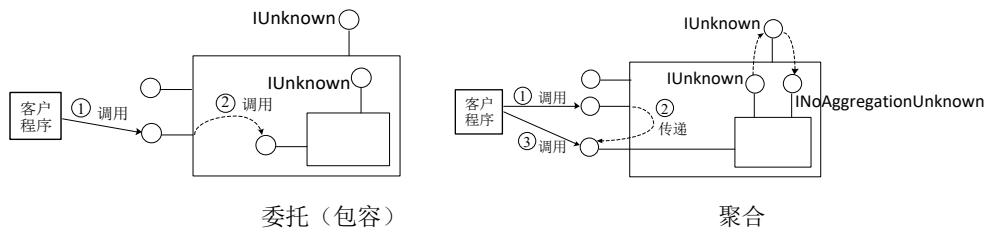
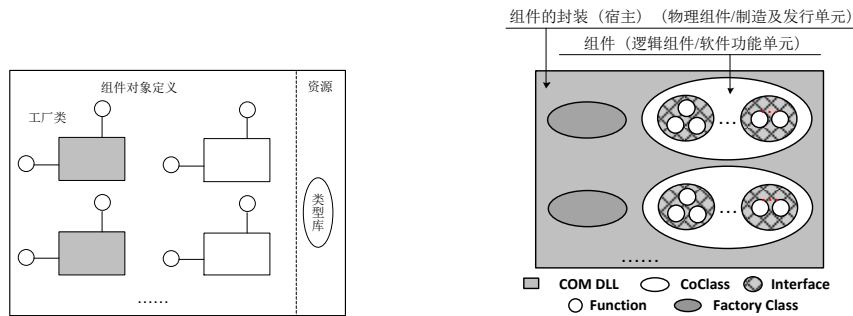
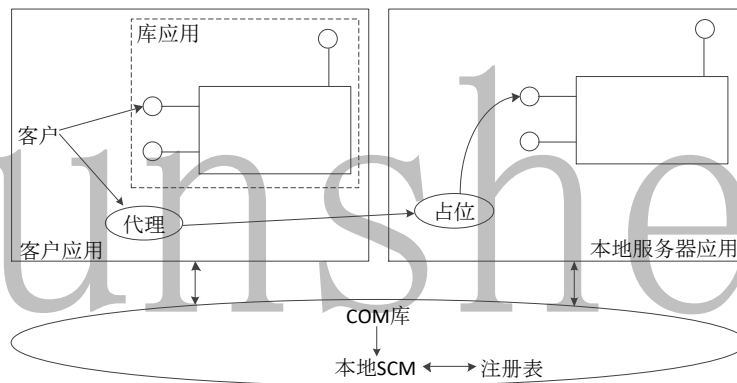


图 2-16 组件功能重用方式



a) 封装结构



b) 运行时结构

图 2-17 Microsoft COM 组件的基本结构

组件范型强调标准，以实现具有独立性的组件之间的集成。组件范型的标准一般称为**软件总线**（简称**软总线**，Software Bus），它定义组件的封装结构并提供基本的集成服务功能（例如：命名服务、查找服务等等）。满足同一标准的组件，可以通过软总线进行集成。目前，流行的组件范型标准有 Microsoft 的 COM、SUN 的 Java Beans 和 OMG 的 CORBA（Common Object Request Broker Architecture）。为了实现对组件的管理和集成，软总线除了提供各种基本服务功能外，还提供一些高级服务功能，例如：属性服务、持久化服务、安全服务、事务服务等等。图 2-18 是组件集成的基本原理。其中，组件对象首先必须在软总线中进行注册，然后才能被使用。某个客户应用或一个对象需要使用某个组件对象时，也是通过软总线进行查找，然后再使用。因此，软总线充当组件对象集成的中介。

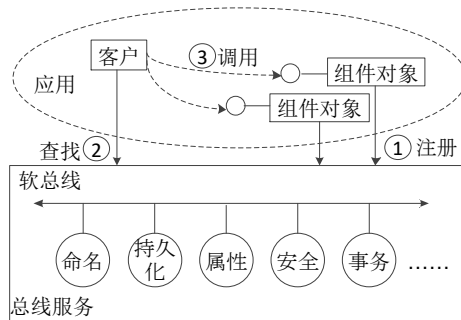
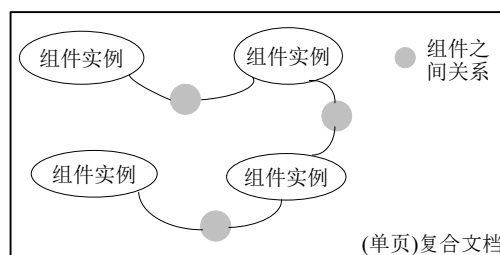
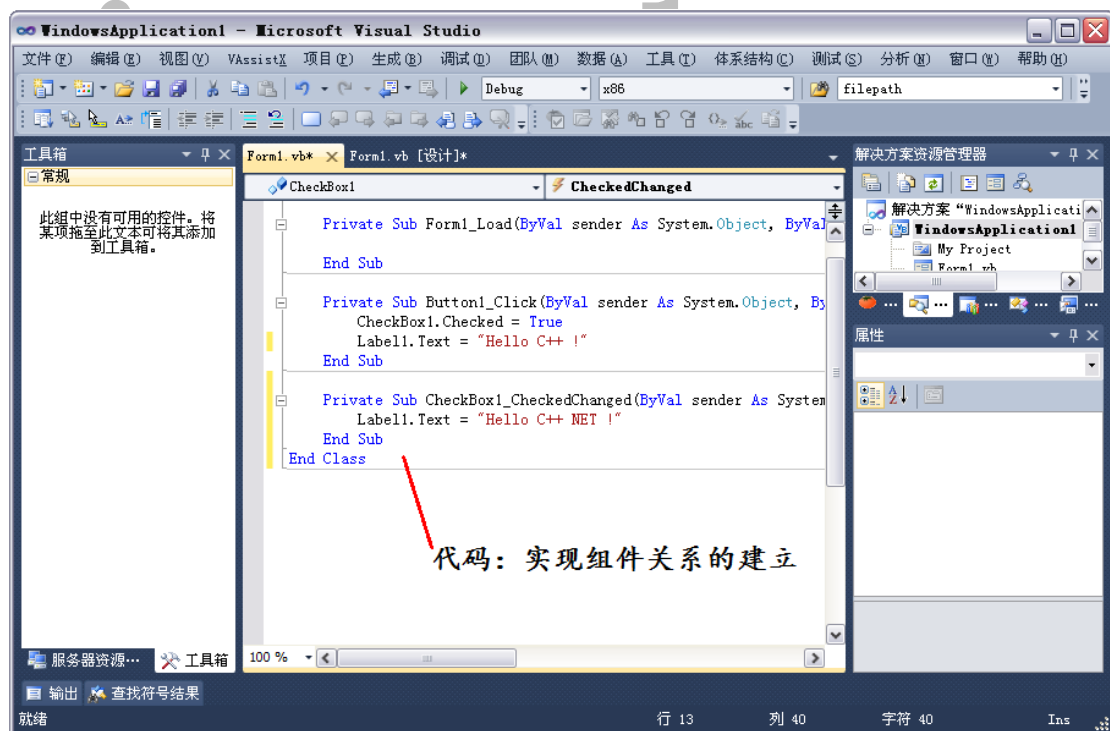
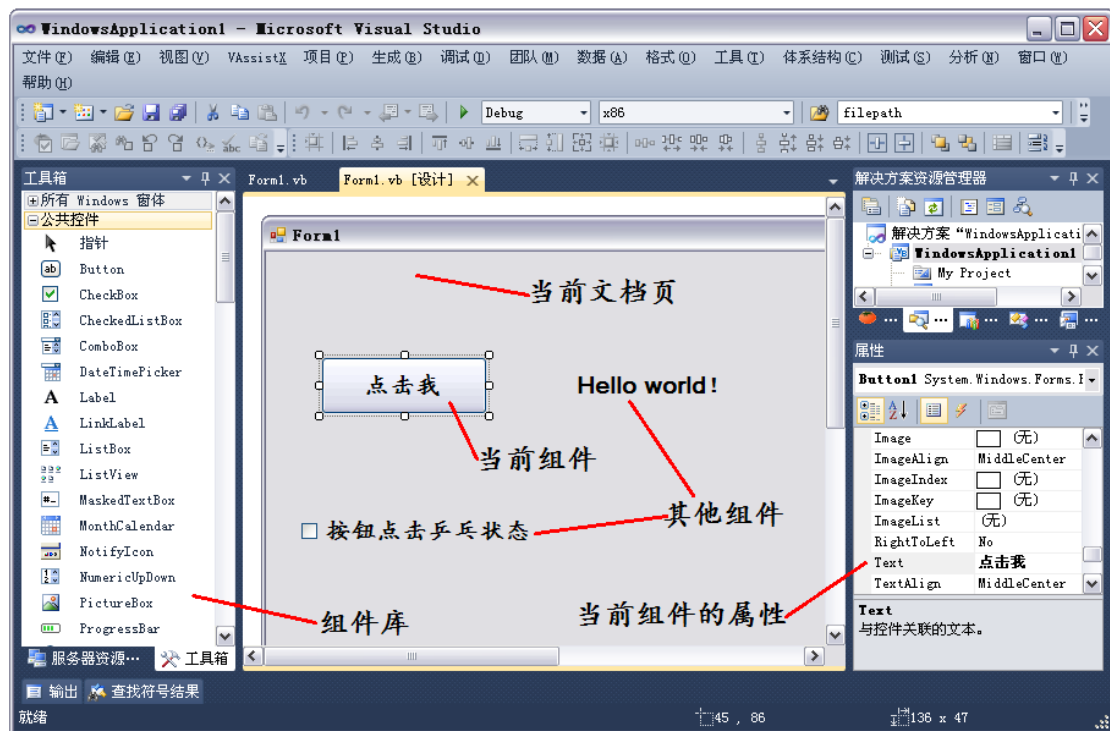


图 2-18 组件集成的基本原理

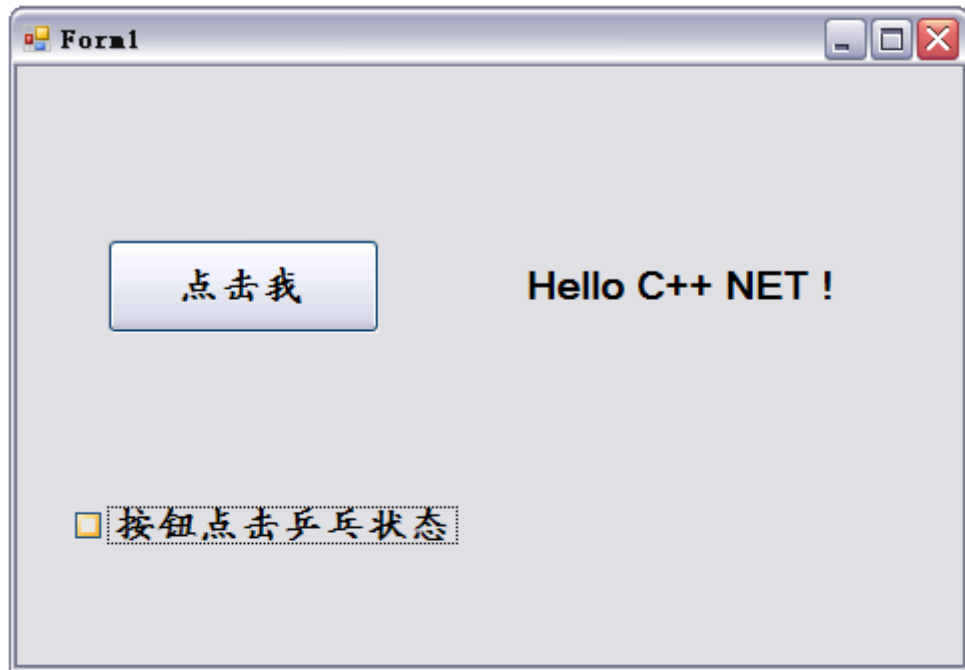
组件范型一般采用基于框架的程序构造方法。所谓**框架（Framework）**，是指已实现部分共性功能的某类程序结构的基本实现。框架抽象了某类程序的结构，定义其中各个功能组件及其相互关系并实现部分共性功能。框架类似于生产线，通过框架构造程序，相当于按照生产线进行各种组件的装配。框架可以分为**水平型**、**垂直型**和**复合文档型**三种。水平型框架一般面向通用类程序的构造，与特定应用领域不相关。例如 Microsoft Visual C++ 支持的各种工程类型就是各种水平型框架。垂直型框架一般面向特定应用领域，它抽象和封装了该应用领域应用程序的基本结构和共性基本组件。例如 San Francisco 就是一种垂直型框架的雏形。复合文档型框架是一种比较通用的框架，它将一个程序抽象为一个文档（可以是单页或多页），将构成程序的各个组件实例看作是文档中的一个一个独立元素，各个独立元素通过事件消息相互联系。通过复合文档框架构造程序，相当于用各种各样的独立元素（带有界面或不带界面）在文档上创作一幅动态的美丽图画。因此，随着图形用户界面的流行以及计算机应用的广泛普及，复合文档型框架已经成为程序构造的主流框架。目前，除了 Microsoft Visual C++ 外，基本上所有开发工具都是支持复合文档型框架的。图 2-19 是利用复合文档型框架进行程序构造的基本原理及样例。图 2-19（a）是单页式复合文档框架构造程序的基本原理，组件之间的关系通过相应的代码实现。图 2-19（b）和图 2-19（c）分别给出了利用 Visual Studio 开发环境和利用 C++ Builder 开发环境开发程序的一个样例，其中，第一张图片建立一个窗体，表示一页文档，然后在其上分别放置并建立了命令按钮、复选框和标签三个组件实例；第二张图片给出了组件之间的关系，分别通过 Visual BASIC 语言的过程和 C++ 语言的函数来实现，具体关系是：当用鼠标点击命令按钮时，复选框被选中，并且标签显示“Hello C++！”（分别如图 2-19（b）和图 2-19（c）中的第四张图片，第三张图片为程序原始运行状态），再次用鼠标点击命令按钮时，复选框释放选中，并且标签分别显示“Hello C++ NET！”和“Hello C++ Builder！”（分别如图 2-19（b）和图 2-19（c）中的第五张图片）。



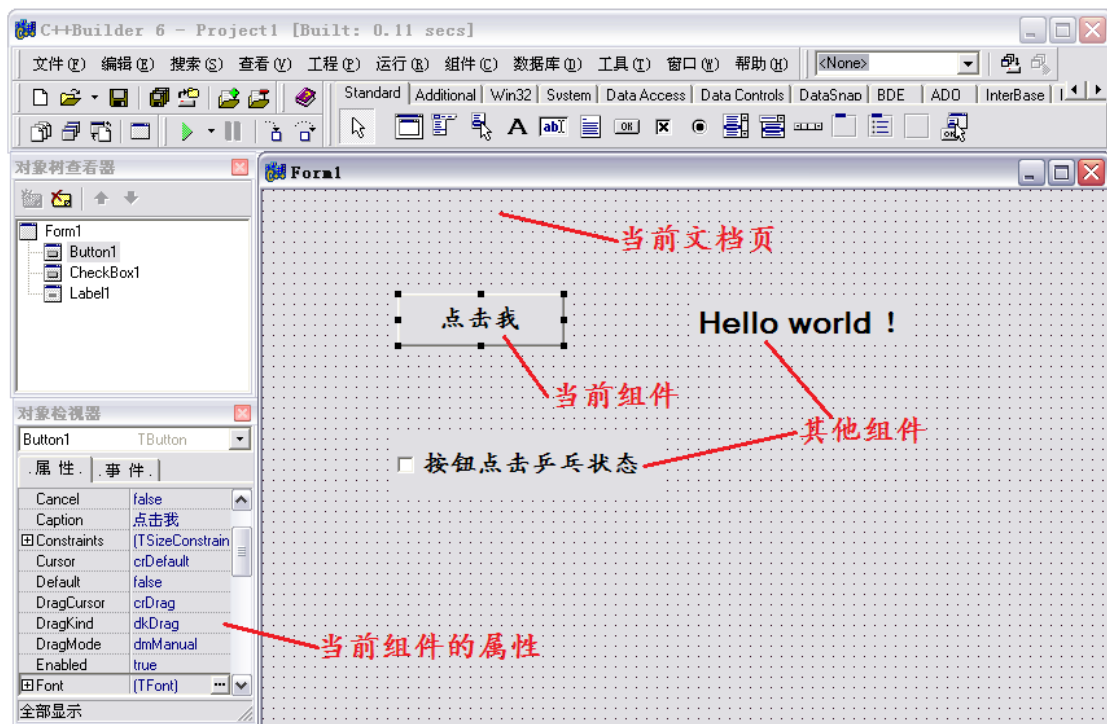
a) 基本原理

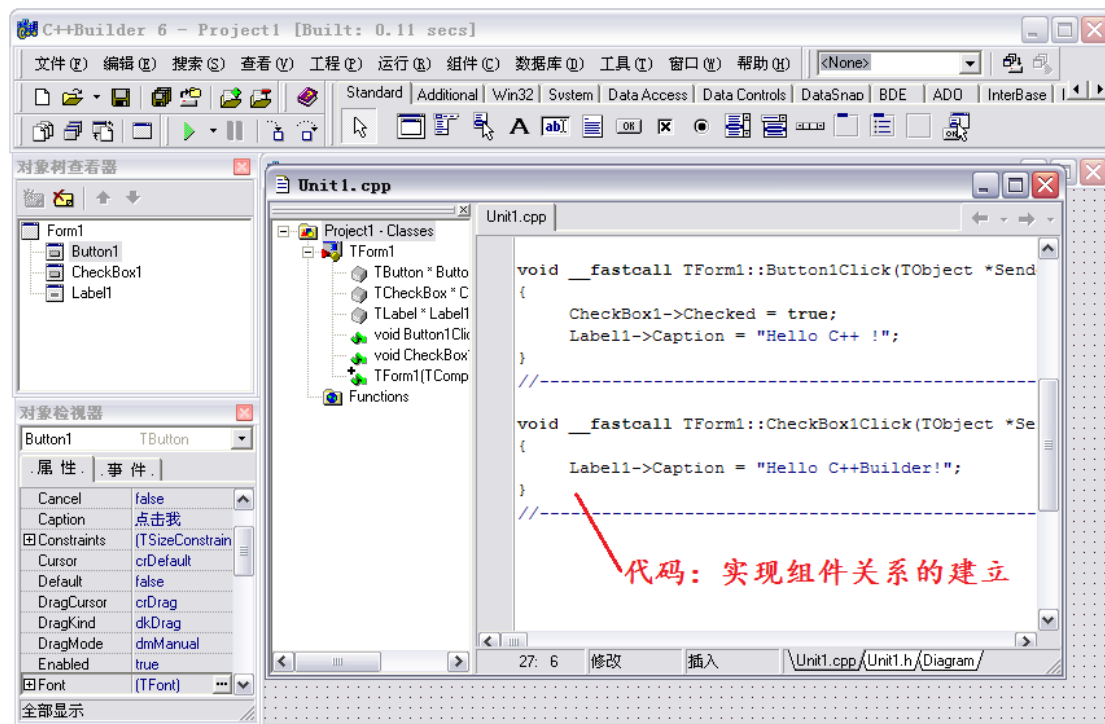


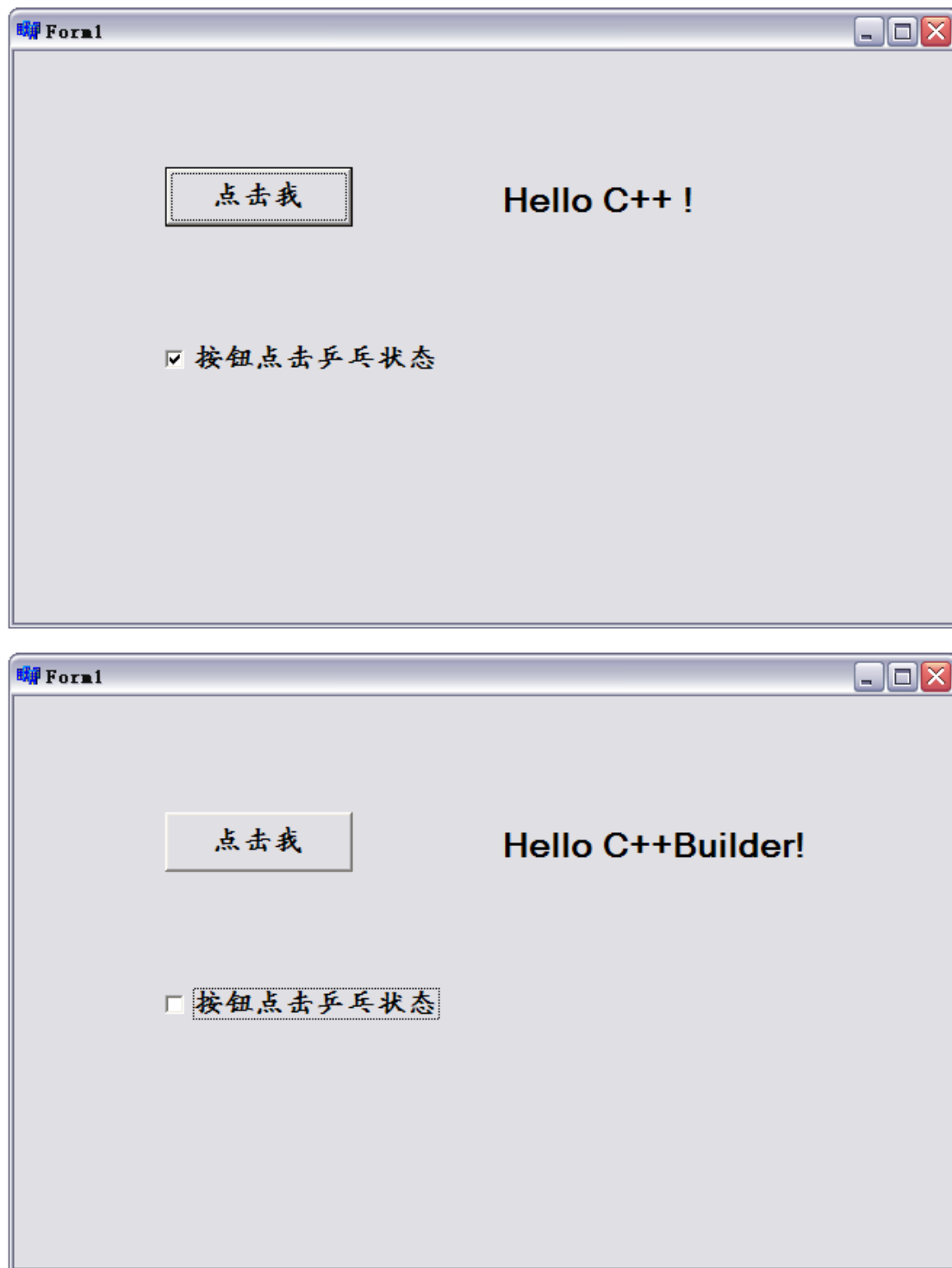




b) 一个样例（Visual Studio 开发环境）







c) 另一个样例 (C++ Builder 开发环境)

图 2-19 利用复合文档型框架进行程序构造的基本原理及案例

相对于对象范型，组件范型的基于二进制黑盒重用机制，为软件维护提供了技术上的保障。基于框架的程序构造模式，为软件工业的大规模生产奠定了基础。然而，尽管组件范型的独立性特性拓展了对象范型的双重用机制，但是，各种标准之间的异构集成和重用仍然是一个问题。

由于组件范型解决了异构集成问题，因此分布对象计算模型得到迅速发展。分布对象计算的基本模型如图 2-20（a）所示。它在软总线基础上，通过在客户端和服务端分别增加代理机制来实现分布环境下组件对象之间的集成。客户端代理、服务器端代理以及软总线为应用开发者屏蔽低层网络通信的细节和异构环境的特性，建立一个面向分布式环境应用开发的通用基础结构。

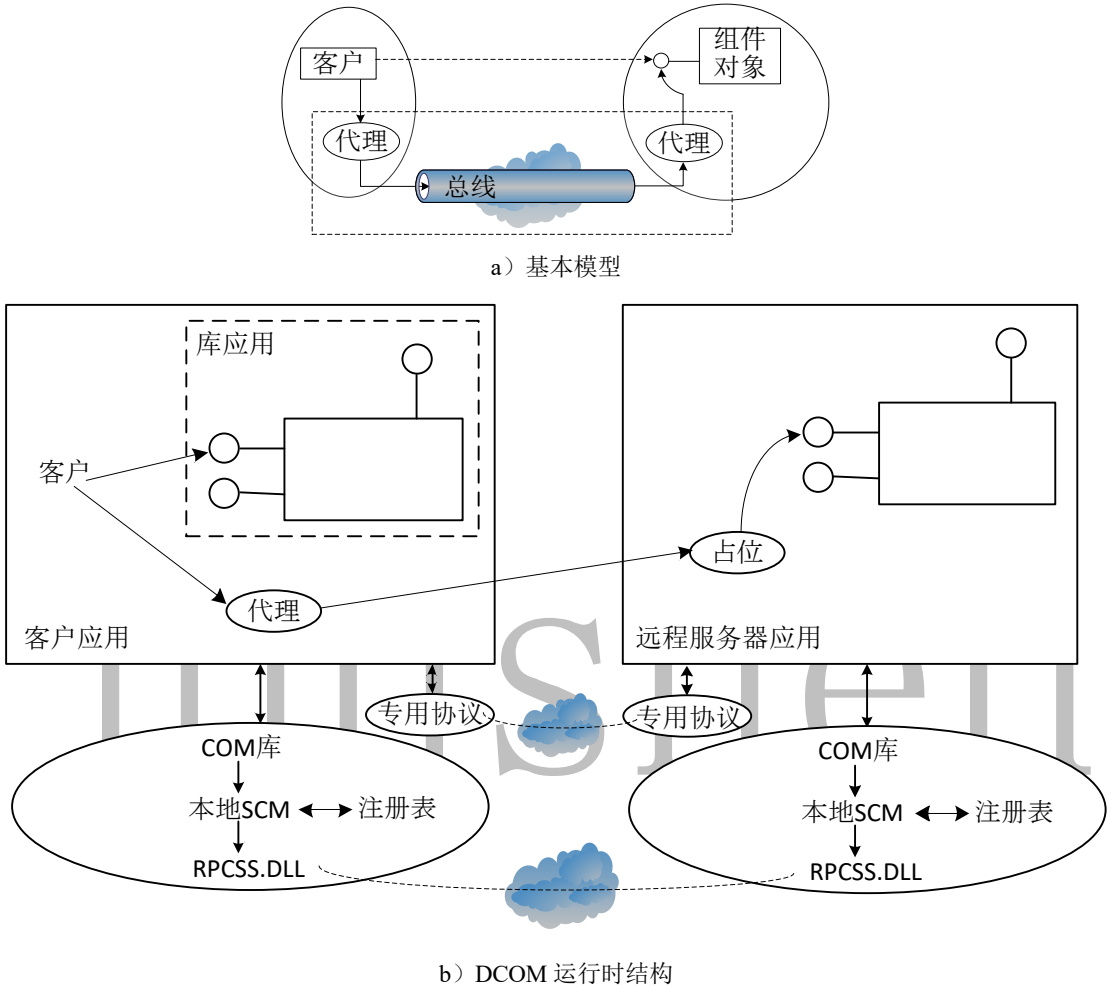


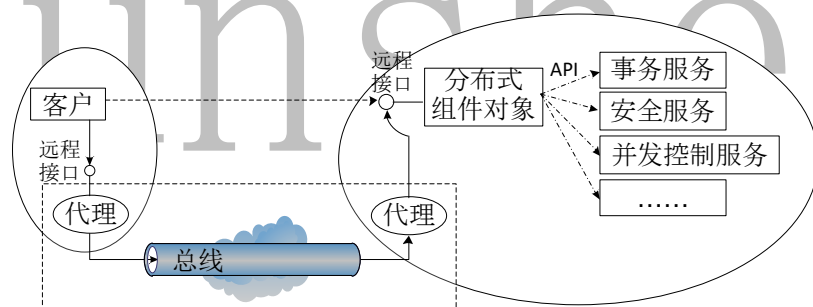
图 2-20 分布对象计算基本模型

CORBA 直接支持分布式计算模型，Java Beans 通过 Java RMI（Remote Method Invoke）将其组件范型拓展为分布式计算模型，COM 通过 RPC（Remote Procedure Call）将其拓展为 DCOM（Distributed Component Object Model）。图 2-20（b）所示是 DCOM 运行时基本结构。对于 DLL 封装的组件，DCOM 通过自动加载一个 Dllhost.exe 作为其宿主，以支持其在远程机器中独立运行。

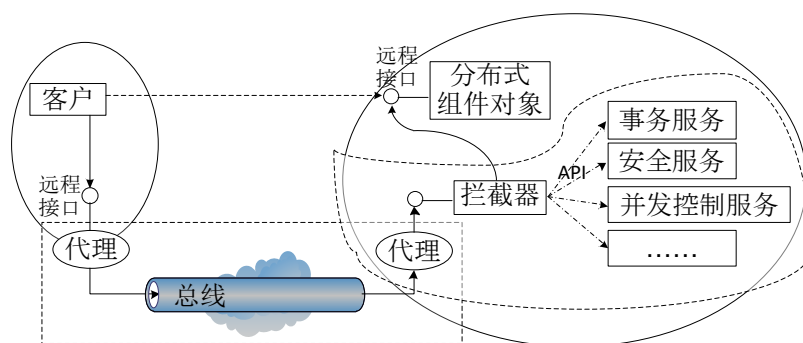
2. 4. 4 配置型组件范型

配置型组件范型（Configurable Component Model，也称为配置型组件模型），又称为服务器端组件范型。它专门针对应用服务器，定义其基于组件的程序构造基础结构模型。在传统分布对象计算范型中，软总线提供的附加基础

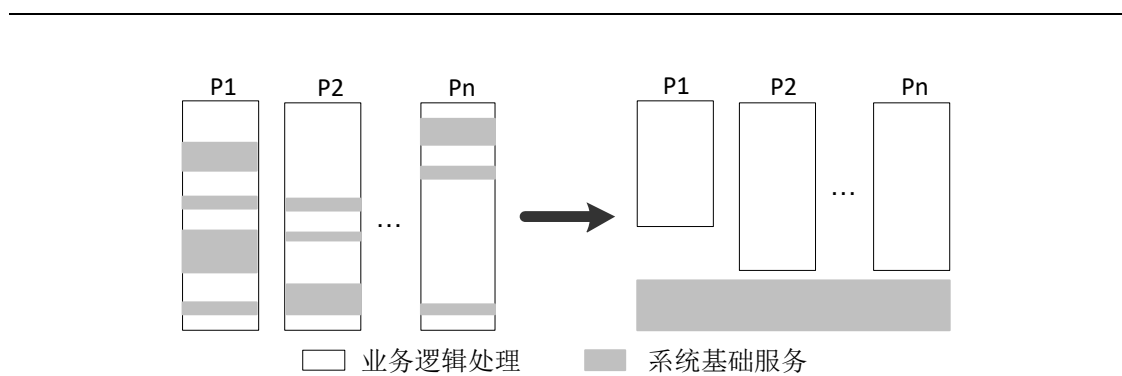
服务需要被业务逻辑代码显式地使用。如图 2-21 (a) 所示。然而，对于响应大量客户端的服务器而言，基础服务的提供涉及系统资源的有效利用，基础服务需要与资源管理技术一起使用。因此，如果这两者都由业务逻辑代码来显式使用，那么应用开发的复杂度就会急剧增加！特别是随着组件交互行为的更加复杂，协调这些基础服务就成为一项十分困难的任务，这就需要与编写业务逻辑代码无关的系统级专门技术来处理。同时，这样处理也可使业务逻辑代码的开发者避免陷入各种基础服务和资源管理机制的系统级事务处理泥潭之中，从而将其思维重心服务于其工作的本质——业务模型建立。因此，配置型组件范型的基本原理，是将应用业务逻辑与系统基础服务两者解耦，由系统基础服务构成一个服务容器，自动隐式地统一为各种应用业务逻辑按需提供相应的基础服务。也就是说，应用业务逻辑可以按需提出不同的基础服务要求，即可配置。这样，基于问题分离（separation of concerns）原理和功能可变性（functional variability）原理，将业务逻辑的功能部分和系统资源有效管理的技术部分两者分开，可以允许两者独立演化并促进基础服务和资源的重用。图 2-21 (b) 是隐式使用软总线基础服务的原理，图 2-21 (c) 是配置型组件范型的基本思想。图 2-22 是配置型组件范型的基本体系。其中，容器提供的基础服务主要包括基本服务和高级服务两类，基本服务主要用于组件实例的运行管理，例如实例池管理、JITA（Just-in-Time Activation）、并发管理等；高级服务主要用于为组件实例提供附加的高级处理功能，例如事务处理、安全管理、事件服务、消息服务等。容器内置并扩展了软总线的基础服务集。



a) 基础服务显式使用

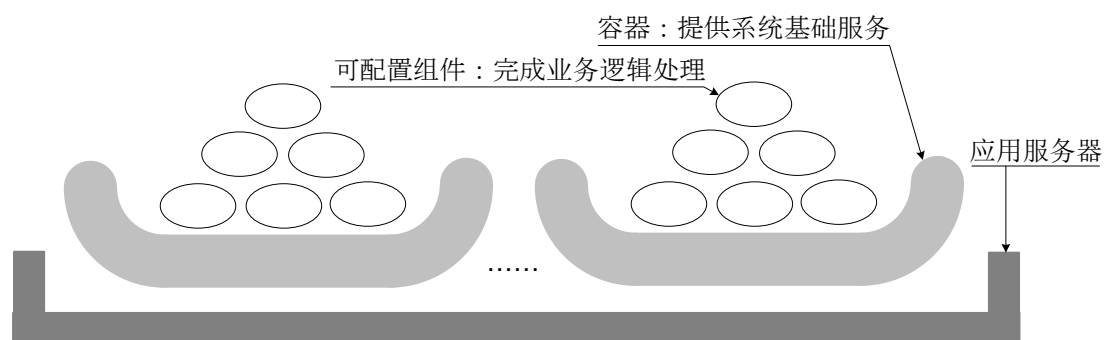


b) 基础服务隐式使用

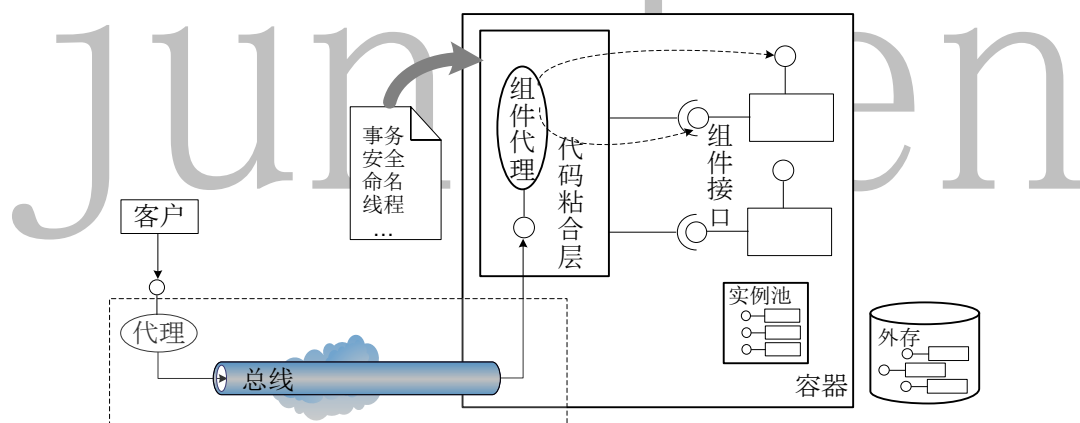


c) 基本实现思想

图 2-21 配置型组件范型的基本原理

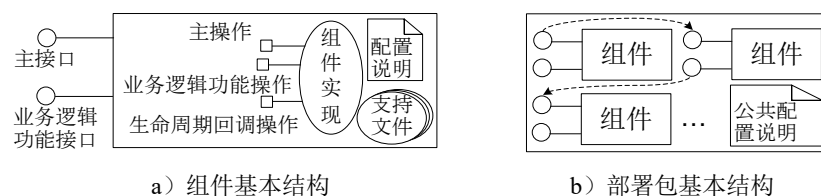


a) 宏观体系（服务器结构）



b) 微观体系（容器结构）

图 2-22 配置型组件范型的基本体系



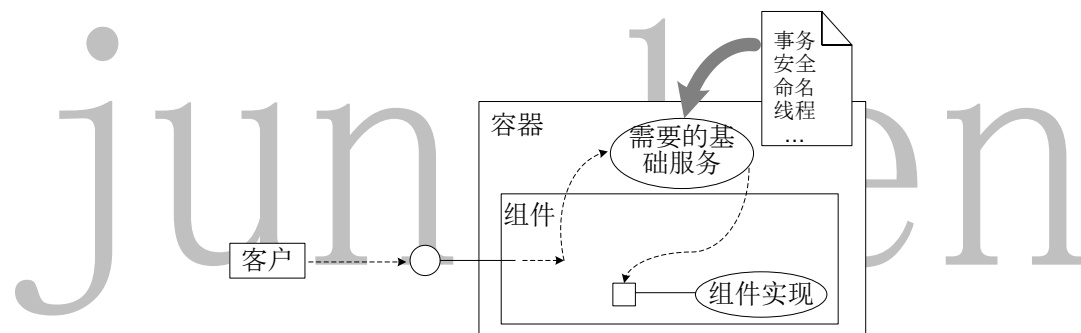
a) 组件基本结构

b) 部署包基本结构

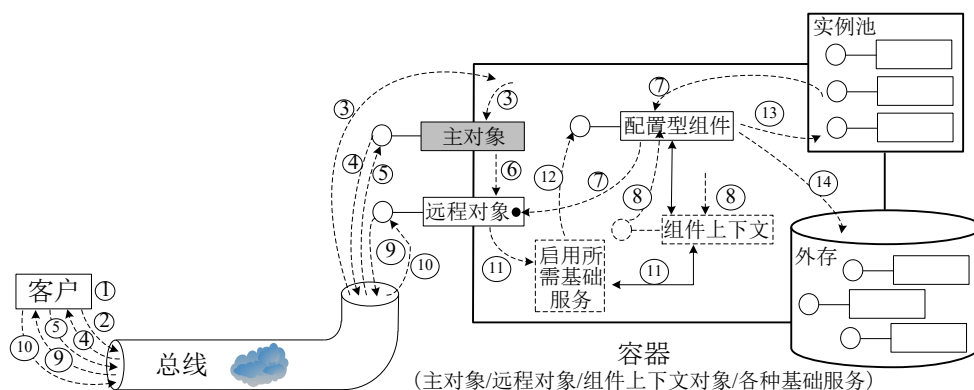
图 2-23 配置型组件范型

配置型组件范型中，配置型组件是一种分布式、可部署的服务器端组件。图 2-23（a）是配置型组件的基本模型。其中，“组件实现”具体实现组件接口的功能，“配置说明”用来配置组件运行时所需要的基础服务功能。组件实现体一般实现三个方面的功能：业务逻辑功能操作实现具体的业务逻辑规则；生命周

期回调操作面向服务容器，服务容器通过这些操作管理组件实例的运行（例如钝化、激活等）；主操作主要用来创建、查找和删除组件实例。一个应用往往涉及多个组件，因此，在配置型组件基本模型基础上建立应用部署包模型，如图 2-23（b）所示。执行应用业务逻辑的配置型组件的可配置性体现在如下两个层面。首先，在应用部署包进行部署时，相应的部署工具（例如 Windows 中的服务管理器）根据部署包中的公共配置以及各个业务逻辑组件的配置说明进行相应配置要求的登记。同时，自动生成客户端和服务容器端的相应代码粘合层（Glue code layer）（含面向分布计算的两端代理）。例如，COM+中，通过 Windows 服务管理器，可以在组件安装时为每一个应用生成一个特殊的安装文件（*.msi），该文件记录所有组件及其配置参数以及相应的粘合层代码。然后，每当客户端调用组件的相应业务操作功能时，客户端代理附加调用上下文，经过总线将请求发送给服务容器，服务容器中的代码粘合层负责创建组件实例，并且通过拦截机制在将客户端请求发送到目标业务逻辑操作之前，根据其预先登记的配置要求启用相应的基础服务，将附加的基础服务功能加入到应用业务逻辑中。同时，为了实现服务器资源的有效利用和提高各个组件实例的执行性能，服务容器通过组件的生命周期回调接口对运行中的组件实例进行管理。图 2-24 是配置型组件范型的基本运行原理。



a) 通过拦截机制加载需要的基础服务



- ① 客户启动总线；② 客户请求总线查找一个配置型组件并创建其实例；③ 容器拦截请求，并创建作为对象工厂的主对象；④ 容器将主对象引用返回客户；⑤ 客户请求主对象创建配置型组件实例；⑥ 主对象创建一个远程对象（该对象具有与配置型组件一样的业务逻辑功能接口），用来实现基础服务的启用；⑦ 容器按池管理策略建立一个配置型组件实例，并将其链接到远程对象（即将其引用传给远程对象）；⑧ 容器创建一个组件上下文对象并通过配置型组件的生命周期回调操作将其引用传给配置型组件实例；⑨ 容器返回远程对象的引用；⑩ 客户调用远程对象的业务逻辑功能接口（即要调用配置型组件的业务逻辑功能接口）；⑪ 远程对象根据预先登记的配置要求，进行所需基础服务操作，并将操作状态设置到组件上下文中；⑫ 远程对象再将请求转发给配置型组件实例（即调用其业务逻辑功能操作）；配置型组件实例可以通过组件上下文对象获取容器的当前工作状态并使用系统资源；⑬ 容器对配置型组件实例进行池化管理；⑭ 容器对配置型组件实例进行持久化管理；（注：客户端代理和服务端代理省略）

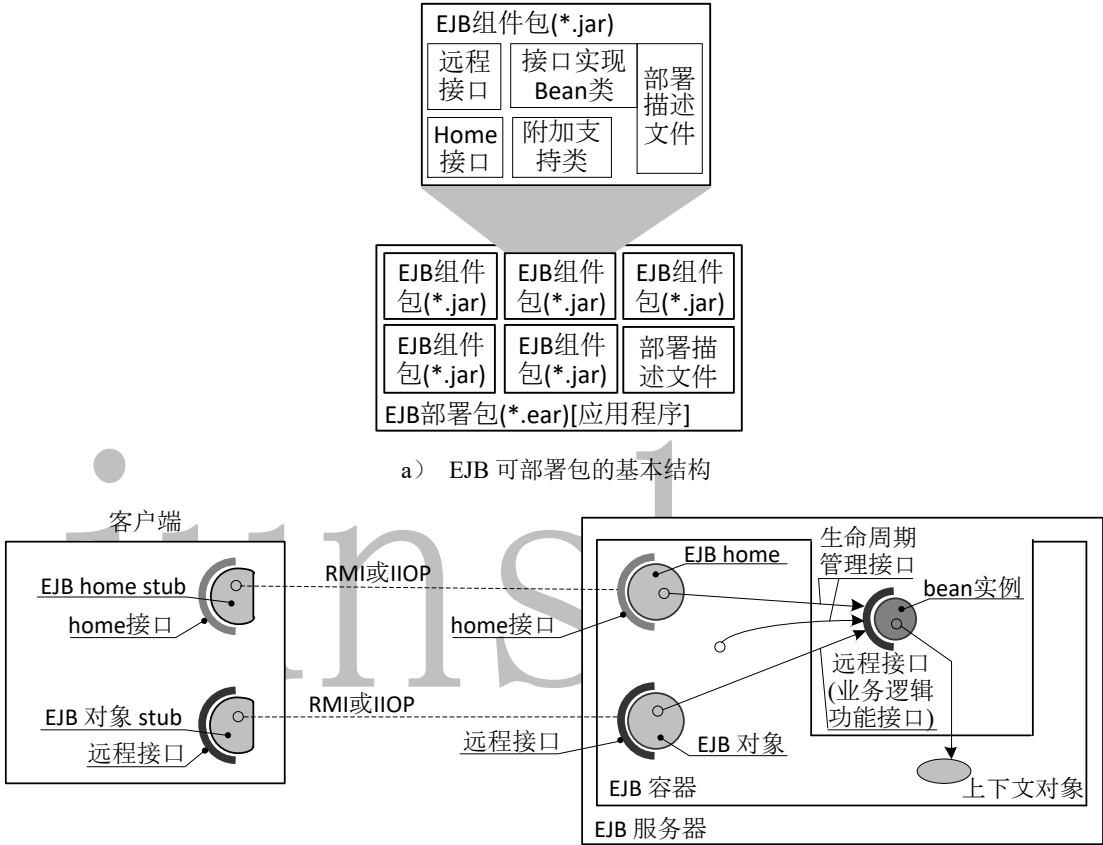
b) 基本运行过程

图 2-24 配置型组件范型的基本运行原理

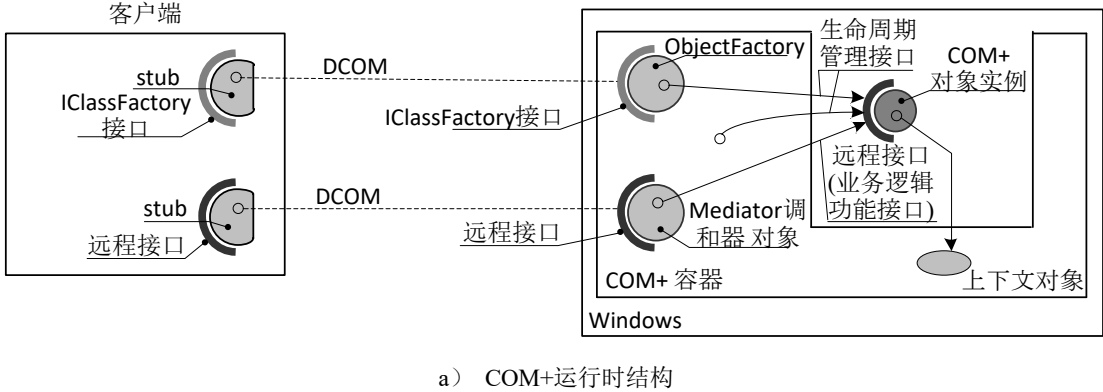
目前，流行的配置型组件范型标准有 EJB、COM+ 和 CCM（CORBA Component Model）。CCM 定义了一种较为完善的可配置组件范型，但目前还没有具体的产品。EJB 中，主对象称为 **Home 对象**，远程对象称为 **EJB 对象**，配置型组件称为 **Enterprise Bean**，配置型组件的配置说明用 **配置描述器** 给出。另外，还可以通过一个特定 **属性描述器** 描述 Enterprise Bean 的一些附加特性，以便 Enterprise Bean 在运行时使用。目前，一个 EJB 对象只能支持一个接口。EJB 基础服务包括生命周期管理、持久性服务、事务处理服务、安全管理服务等。对于这些基础服务，Enterprise Bean 都可以按需进行配置。图 2-25 (a) 是 EJB 配置型组件包以及可部署包（应用程序）的基本结构。图 2-25 (b) 是 EJB 运行时的基本结构。COM+ 中，主对象称为 **工厂对象**（ClassFactory 实例），配置型组件称为 **组件对象**（CoClass 实例），远程对象是组件对象的一种包装，COM+ 中通过 **调和器对象**（Mediator）给出。配置型组件的封装一般只能以一个 DLL 文件给出，配置型组件的配置说明可以通过在程序中加入属性描述给出（这种方式称为说明性程序设计，即 **declarative programming**，或称为基于属性的编程）或在组件安装时进行交互式配置。COM+ 的一个配置型组件可以实现一个或多个接口的功能。COM+ 组件的基本结构参见图 2-17 所示。图 2-26

(a) 是 COM+ 运行时的基本结构。事实上，COM+ 并没有特别地定义一种可配置组件范型，而是在 COM/DCOM 组件基础上融入 MTS（Microsoft transaction Server，微软事务服务器）及其他基础服务功能并借助于 Windows 操作系统构成一个运行时环境，由 Windows 操作系统及各种基础服务套件充当应用服务器和容器的角色。任何 DLL 封装格式的 COM 组件，只要在 Windows 服务管理器中导入到某个应用程序（即安装组件）并进行相应配置即可成为 COM+ 组件。因此，COM+ 应用程序可以看成是一种逻辑可部署包，它可以含有多个 COM+

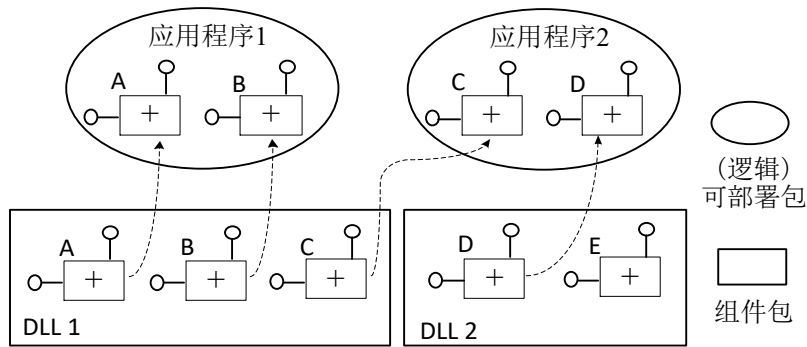
组件，这些组件具有相同的已配置特性。图 2-26（b）是 COM+应用程序与 COM+组件之间的关系。COM+基础服务增强了有关线程同步方面的控制服务（基于套间、活动的同步控制机制），提供对组件实例并发运行的细粒度控制。图 2-26（c）是 COM+并发控制的基本模型。COM+服务容器除了提供实例池管理、分布式事务处理外，还提供基于角色的安全管理、松散耦合的事件服务等。并且，还集成了 MSMQ（Microsoft Message Queue）、负载平衡、内存数据库等服务功能。



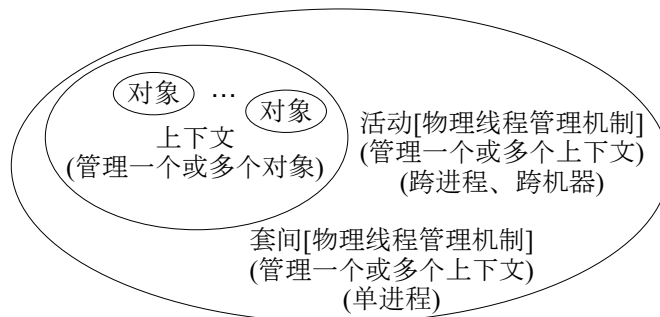
(b) EJB 运行时结构
图 2-25 EJB 的基本结构



a) COM+运行时结构

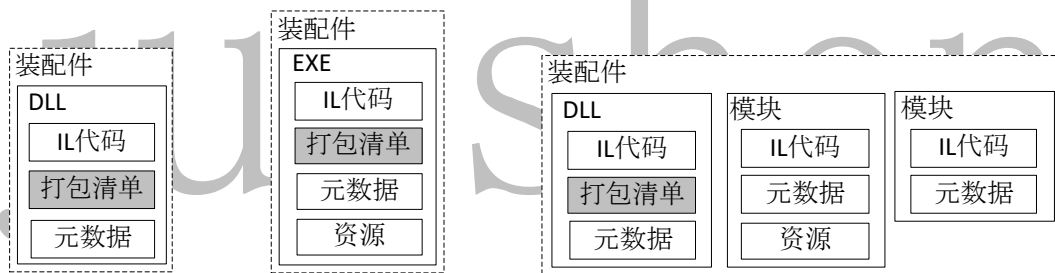


b) COM+应用程序可部署包



c) COM+并发管理模型

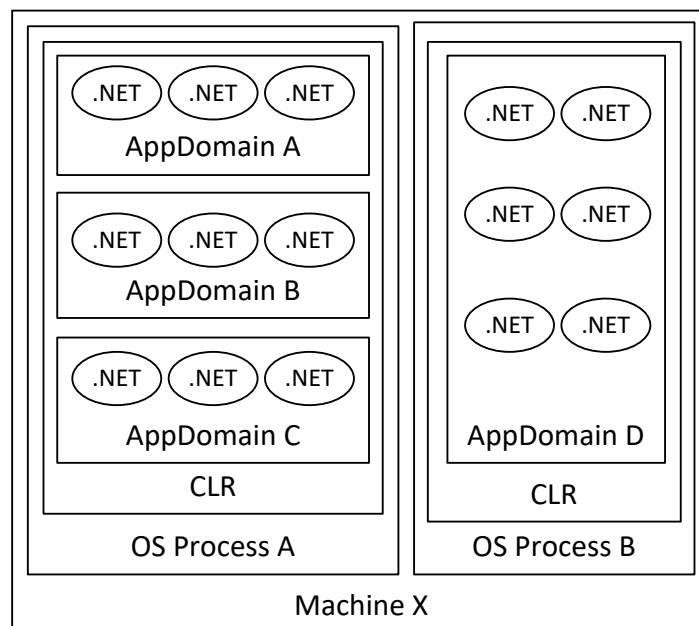
图 2-26 COM+的基本结构



a) 单 DLL (模块) 文件 b) 单 EXE (模块) 文件

c) 多 DLL (模块) 文件

图 2-27 .NET 装配件基本结构



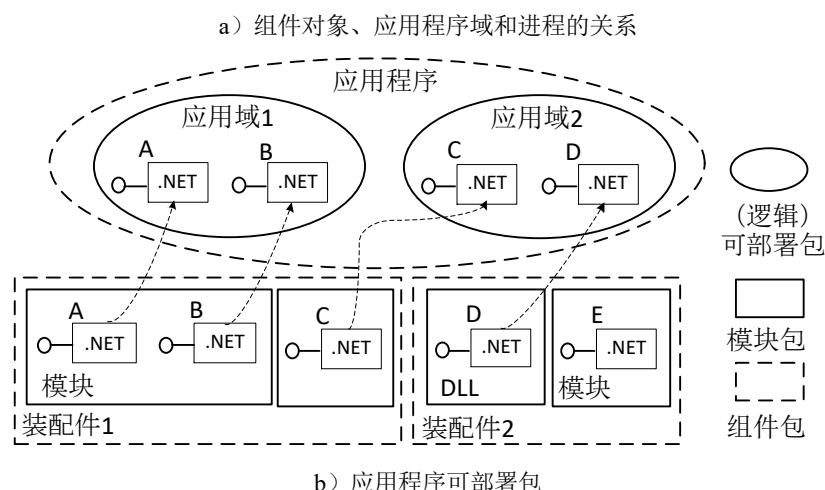


图 2-28 .NET 应用程序结构

随着 Microsoft .NET 平台的推出，COM+组件标准得到了修正和改善。.NET 采用**装配件**（assembly）作为基本的组件打包及部署单元，它是一种**逻辑组件**（Logical Component），可以包含**打包清单**（Manifest，描述该配件以及所需的其他配件，包括版本号、编译链接号以及编译器在编译时捕获的修正号、文化特征等版本信息）以及一个（或多个）物理 DLL 或 EXE 模块，每个模块内部包含 **IL 代码**（Intermediate Language Codes，中间语言代码）、**元数据**（metadata，描述配件中声明的所有类型及其关系）及**资源**（包括图标、图像等），模块以文件形式存储。装配件可以是静态的（由开发工具生成，存储在磁盘中），也可以是动态的（在内存中动态生成并立即运行，可以保存在磁盘中再次使用）。模块及装配件的结构如图 2-27 所示。在装配件基础上，.NET 环境进一步通过**应用程序域**（AppDomain）概念定义装配件的运行模型（即逻辑组件的运行结构及作用域），建立.NET 环境中的可配置组件范型。图 2-28（a）所示给出了应用程序域与装配件的关系。并且，.NET 平台通过 CLR（Common Language Runtime）环境封装并提供类似于 COM+容器所提供的基础服务（参见.NET 命名空间 System.EnterpriseServices），拓展了 COM+的容器机制。从而，使得.NET 中的每个应用程序域成为一个在 CLR 中可以独立配置并部署的逻辑可部署包（参见图 2-28（b）所示）。因此，.NET 平台通过元数据机制及 CLR 改善了 COM+的各种缺陷，简化了组件的开发和部署。例如：.NET 没有派生出所有组件的正式基本接口（例如：IUnknown），而是将所有组件都从 System.Object 类派生，即所有.NET 组件对象都是 System.Object 类的多态表现；.NET 没有类工厂，而是由运行时环境将类型声明解析成包含该类型的装配件以及该装配件内确切的类或结构；通过完善的无用单元回收机制改善由引用计数的缺陷而导致的内存泄漏和资源泄漏；采用元数据方法代替类型库和 IDL（Interface Description Language，接口描述语言）文件；采用命名空间和装配件名称来确定类型范围的方法以提供类型（类或接口）的唯一性来改善基于 GUID（Globally Unique Identifier）的注册的脆弱性；.NET 没有套间，默认情况下，所有.NET 组件都在自由线程环境中运行，由开发者负责同步组件的访问。开发者可以依赖.NET 的同步锁或使用 COM+的活动来实现同步；.NET 中通过在类定义中使用关键字 internal 告知运行时环境拒绝一个装配件之外的任何调用者访问该装配件内的组件。从而，可以防止 COM+中通过搜

通过注册表找到私有组件的 CLSID（CoClass IDentifier，组件类标识符）以使用它的漏洞；.NET 通过为指定代码段配置许可并提供证据，将 COM+ 基于角色的安全性拓展为角色安全和调用身份验证双向的安全控制，为高度分散、面向组件的环境提供新的安全模型；.NET 中有关组件的一切操作都不依赖于注册表，并严格维护版本控制，从而，简化组件的部署；与 COM+ 只支持运行时不同语言所实现的组件的集成不同，.NET 同时支持运行时和开发时的组件的无缝集成（例如：允许使用一种语言开发的组件从使用另一种语言开发的组件派生）；采用基于属性的编程方法增加组件配置的灵活性；等等。另外，.NET 支持两种组件：**使用基础服务的组件（Serviced component）**和**标准的被管理组件（Managed component，也称为托管代码或受控代码）**。其中，前者就是 COM+ 标准组件的演化和发展，而后者（总是含有元数据）则是新的组件封装模型。.NET 实现了两者的统一。为了支持说明性程序设计，.NET 提供的新型程序设计语言 C# 直接支持将配置要求以属性方式写入程序中，建立基于属性的程序设计方法（Attribute-based programming）。

2. 4. 5 服务范型

进入 21 世纪，随着互联网应用的不断普及，配置型组件范型及其衍生的分布式对象计算范型，由于其紧耦合、多标准和低层传输协议的依赖性等弊端，导致其不能适应面向互联网的应用动态集成需求。因此，面向服务范型应运而生，并在此基础上建立服务计算范型。

所谓**服务（service）**，是指一个封装着高级业务概念、实现公共需求功能、可远程访问的独立应用程序模块。服务一般由**数据、业务逻辑、接口和服务描述**组成，如图 2-29 所示。服务独立于具体的技术细节，一般提供业务功能，而不是技术功能。

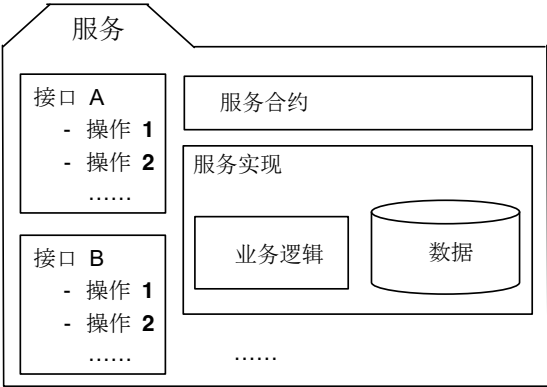


图 2-29 服务的一般结构

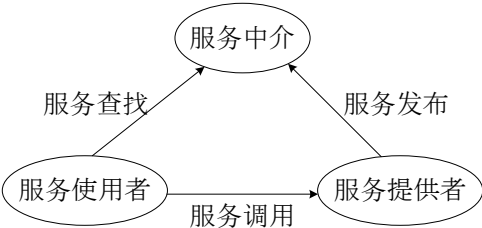


图 2-30 服务模型的基本原理

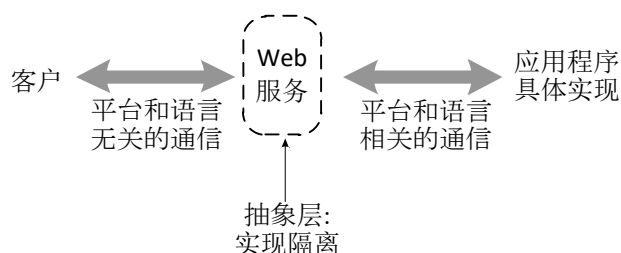


图 2-31 服务模型的抽象作用

服务范型的基本原理是明确服务提供者和服务使用者，并通过服务中介实现两者的耦合。如图 2-30 所示。服务范型通过定义独立于具体技术、可以扩展的通用描述手段来描述服务和实现服务交互，而将服务实现的具体技术细节隐藏在内部。从而，实现服务的无缝集成。图 2-31 所示给出了服务范型的抽象作用。

目前，服务范型的标准主要是 Web Services。Web Services 以 XML (eXtensible Markup Language, 可扩展标记语言, 也称为元标记语言, 它给出一套定义语义标记的规则, 即定义元句法) 作为最基本的通用描述规范, 并定义其各种规范, 例如服务定义与描述、服务访问以及服务发布、发现与集成等。

信息是人类社会必要和重要资源, 信息处理技术是现代社会的的基本要求。因此, 如何组织和描述信息并对信息进行访问和各种处理成为一个核心问题。特别是互联网的蓬勃发展, 信息的可交换性变得越来越重要。XML 技术体系就是针对这一问题的一套标准。一个信息实体一般涉及结构、种类、属性和内容几个方面, 为了描述各种不同的信息, XML 首先通过 Infoset 定义一个信息实体 (称为信息文档) 的抽象概念模型, 然后基于该模型, 通过 XPath (识别 Infoset 信息模型的子集)、XPointer (基于 XPath, 引用外部文档的子集并扩展 XPath 的数据模型, 可以寻址文档中的点和范围)、XLink (两个文档之间的各种链接关系)、XBase (为相对 URI 引用指定基本参照点 URI。URI 是 Uniform Resource Identifier 的简称, 一般包括两个子集: Uniform Resource Locator, URL 和 Uniform Resource Name, URN) 和 XInclude (引用外部已析通用实体) 定义如何创建信息文档内和信息文档间的关系, 通过 DOM (Document Object Model, 在内存中创建整个信息文档的树型结构, 对文档内容随机访问, 适用于 XML 文档的结构化处理)、SAX 和 StAX (Simple API for XML, Streaming API for XML, 都是基于 XML 事件的解析, 不需将整个文档读入内存, 也不创建信息文档的树型结构, 不能对文档内容随机访问, 适用于对大型文件的简单快速、高效率的处理。SAX 采用被动的 push 方式, StAX 采用主动的 pull 方式) 定义如何访问信息文档的编程接口。同时, 通过 Schema (一种基于 XML 的语言) 为信息文档定义结构和类型, 通过 Namespace (命名空间) 将一个信息元素与其关联的 Schema 关联起来, 避免 XML 文档中的元素和属性等名称在使用中发生冲突, 通过 XSLT (XSL Transformation, 一种基于 XML 的语言) 定义一个词汇表 (比如 Schema 类型) 到另一个词汇表的转换规则。从而, 建立 XML 技术体系中的核心规范集。最后, 基于这些规范, 面向各种应用, 定义各种具体的应用规范。例如 Web Services 中的 SOAP (Simple Object

Access Protocol)、WSDL (Web Services Description Language) 和 UDDI (Universal Description, Discovery and Integration) 等。图 2-32 所示给出了 XML 技术的体系。图 2-33 到图 2-35 所示分别给出 SOAP 规范、WSDL 规范和 UDDI (Universal Description, Discovery and Integration) 规范的直观视图。

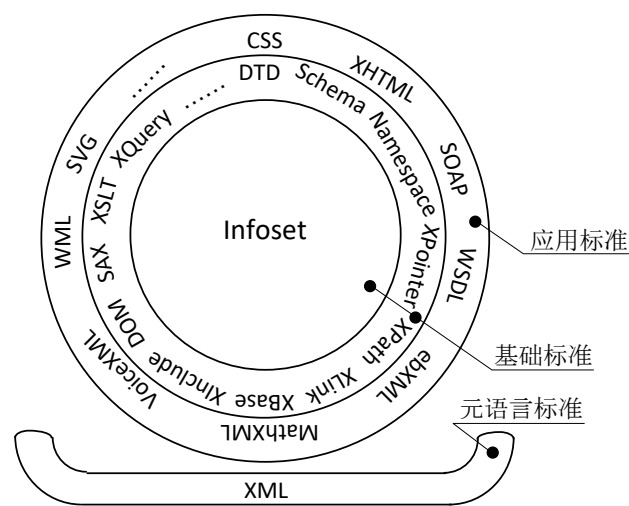


图 2-32 XML 技术规范

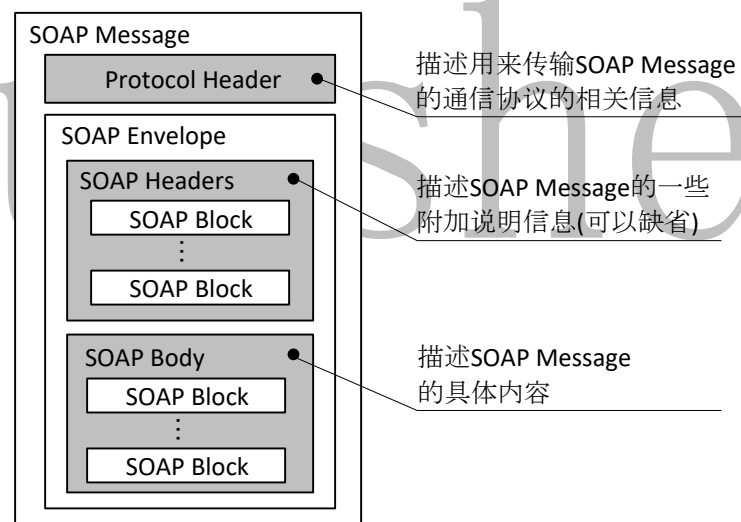


图 2-33 SOAP 规范

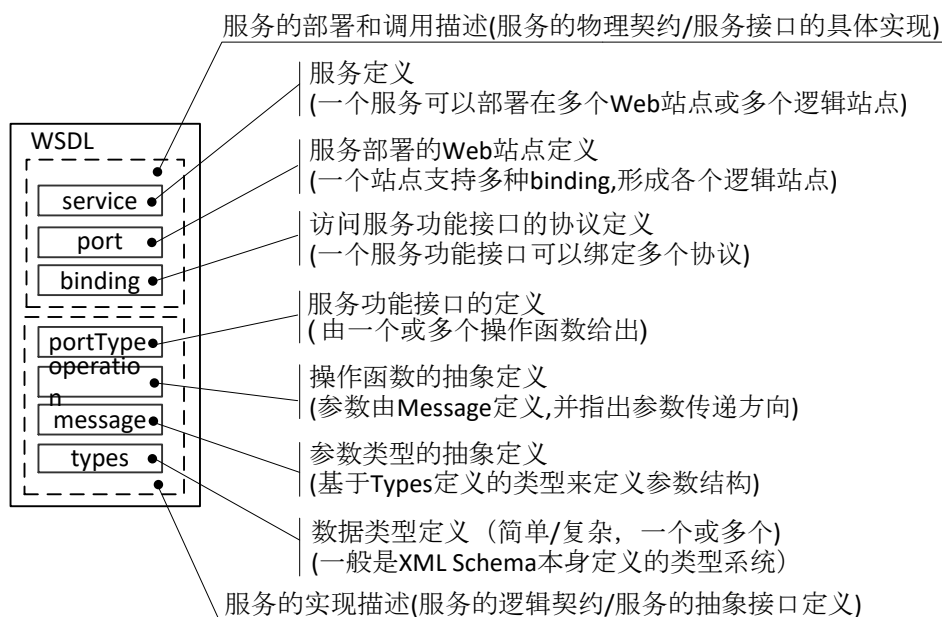


图 2-34 WSDL 规范

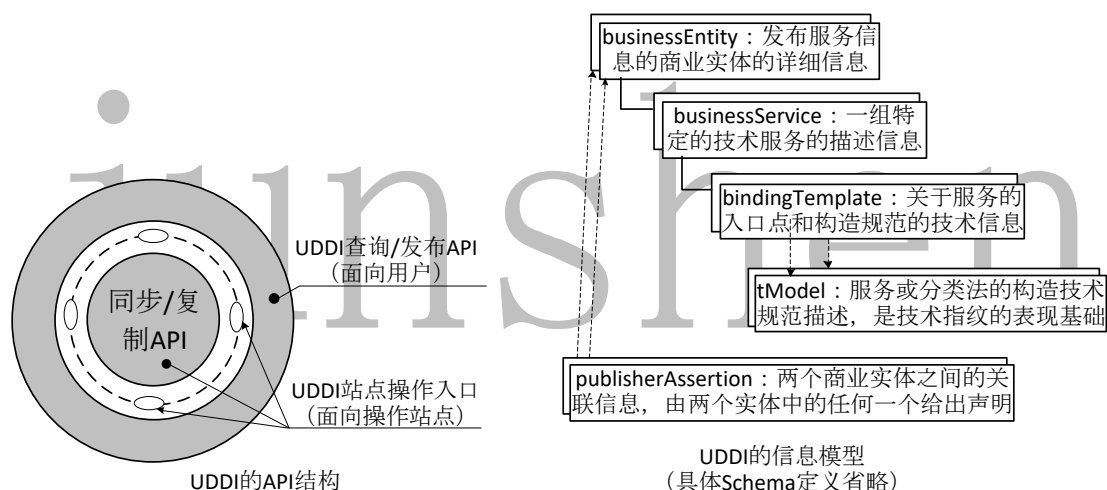
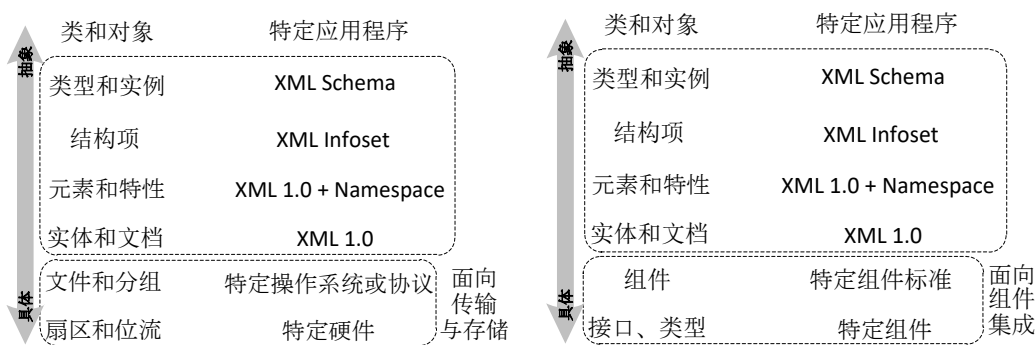


图 2-35 UDDI 规范

事实上, XML 是一种通用的结构化信息编码标准。也就是说, 作为一种可以创建其他专用标记语言的通用元标记语言, XML 可以对任意结构化信息进行定义 (即编码)。XML Infoset 所定义的信息模型是层次型模型, 层次型结构的递归特性决定了其广泛的适用性和描述能力。尽管目前的 XML 标准 (1.0 或 1.1) 主要面向传输和存储 (即串行化) 定义了语法细节, 但是, XML 中的许多技术标准都是建立在 Infoset 抽象模型基础上, 因此, 对于不使用 XML 1.0 串行化格式的应用 (即不是用于传输或存储, 而是用于数据交换、互操作、类型化值、Web 发布和分布式计算、组件集成等等), 这些技术也具有通用性。也就是说, XML 可以是软件集成问题的统一解决方案。如图 2-36 所示。

针对信息的可交换性, JSON (JavaScript Object Notation) 提供了一种轻量的处理方法, 相对于 XML, JSON 具有数据格式简单、易于读写、易于解析、占用带宽小等特点。

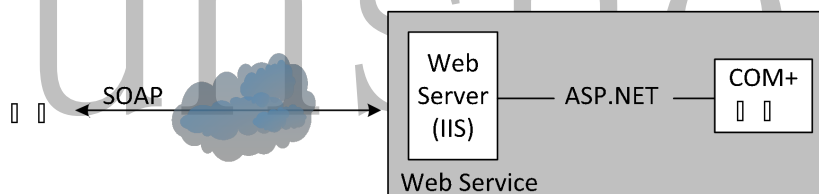


串行化格式视图

抽象数据模型视图

2-36 XML 作为一种集成技术

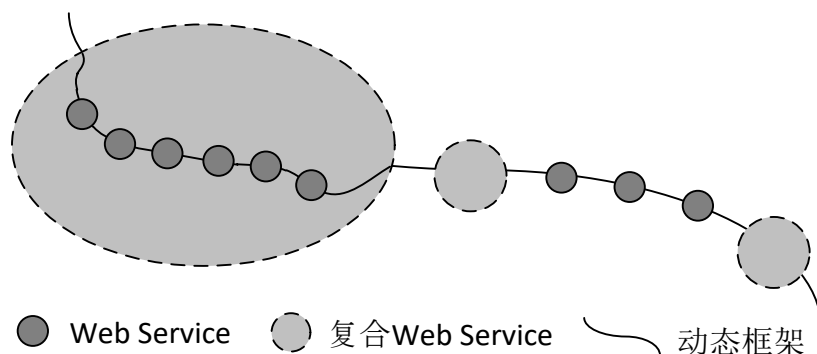
Microsoft .NET 平台面向新一代 Web 应用开发，通过在开发工具 Visual Studio .NET 中提供的 Visual C# Projects 中的 **ASP.NET Web Service 模板** 类型以及建立支持属性编程的新型程序设计语言 C#，直接支持面向 Web Services 的应用开发。通过 Web Services，将 COM+ 对象包装为面向互联网的一种服务对象。如图 2-37 所示。另外，.NET 平台还提供了一系列公共 Web 服务，称为 .NET My Services。这些服务类似于传统程序设计中的系统函数、类库中的类或者组件库的组件，供应用开发时按需调用。



2-37 面向互联网的 Web 服务对象

相对于组件范型而言，服务范型也采用基于框架的程序构造方法。但是，与组件范型的框架不同，服务范型的框架是一种动态框架，它根据业务流程的需要，动态集成各个 Web 服务。因此，这种方法也称为基于流程的程序设计方法，即流程是一种动态框架。组件范型采用与机器相关的二进制，并且标准不统一，从而导致对服务接口与执行环境的分离不够彻底。而服务范型采用与机器无关、基于 XML 的文本描述，并且标准统一。从而使得服务接口与执行环境彻底分离。因此，服务范型的思维重心在于业务服务和业务流程的抽象，专注于业务描述，将业务逻辑与代码实现（依赖特定执行环境）相互分离（参见图 2-31）。因此，服务范型比组件范型更加抽象。另外，从支持软件大工业生产角度来看，组件范型侧重于细粒度的技术基础，而服务范型则是面向粗粒度的应用基础。

服务范型也演绎了递归思想，主要体现在动态框架本身又可以作为一个服务被其他动态框架集成。图 2-38 所示是服务范型递归思想的解析。



2-38 服务范型的递归思想

相对于 Web Services, RESTful (Representational State Transfer) 从资源及其维护角度建立了一种轻量的 Web 服务范型, 其具体解析参见第 8.2.2 小节。

2. 4. 6 抽象范型

尽管服务范型已经强调业务逻辑的抽象并使之独立于具体的平台和环境, 但其系列标准中对于动态框架的建立仍然局限于传统控制流的设计思维, 具有封闭性, 不能适应应用不断变化所带来的复杂交互场景的应用业务流的描述。因此, 服务模型的抽象层次还不够高, 抽象程度还不够深。于是, 抽象范型得到自然演化并诞生。

目前, 抽象范型主要包括基于归纳思维的可恢复语句组件模型 (Resumable Program Statements Component) 和基于演绎思维的元模型 (Meta Model)。两者异曲同工, 都是面向抽象层的应用业务逻辑的描述, 而不关注描述的具体实现平台和环境, 实现完整的技术独立性和应用发展适应性 (有关抽象范型的具体解析, 请读者参见第 6 章)。

2. 5 深入认识程序基本模型

从程序基本范型演化和发展的轨迹来看, 人们对软件的认识不断深入, 针对软件构造方法和技术的思维核心也不不断变迁。从面向机器的功能范型, 到面向问题的对象范型以及组件范型, 再到面向应用或业务的服务范型。这种发展, 显式地区分了软件需要解决的应用问题和软件本身构造技术问题, 并通过程序基本范型进行粘合 (参见图 2-2)。从而, 回归了软件的本质, 即软件用来描述客观世界。

从软件工程角度来看, 功能范型的思维核心强调实现阶段, 对象范型的思维核心强调分析和设计阶段, 组件范型的思维核心强调技术相关的维护阶段, 服务范型的思维核心强调技术无关的维护阶段。而且, 软件重用的粒度越来越大。

从软件体系结构角度来看, 随着程序基本范型的演化和发展, 软件体系结构也不断演化和发展。从无模型的一体式钢板结构到基于功能范型的结构化体系结构, 比如 Client/Server 结构、老三层结构; 从结构化体系结构到基于对象范型、组件范型的框架结构和新三层结构; 再到基于服务范型的 SOA

(Services-Oriented Architecture, 面向服务的体系结构。关于 SOA 的相关解析

请参见第 5.2 小节和第 8.2 小节) 结构, 体系结构的发展由无到有、由技术和平台的相关性到技术和平台的无关性、由基于归纳思维的策略到基于演绎思维的策略、由相对固定和封闭的框架到灵活和开放的框架。从而, 使体系结构的思维核心由其外延转向其内涵。

从认识论角度来看, 程序基本范型演化和发展的抽象性越来越高, 由以机器为中心到以应用为中心, 再到以企业为中心。从而, 真正诠释了软件设计的内涵。另外, 尽管程序基本范型经历了多代发展, 但从本质上看, 对象范型以后的各种范型, 基本上都是采用面向对象的思维策略, 只是其抽象级别越来越高。特别是服务范型的诞生, 使得我们可以将整个互联网看做是一台计算机, 将 Web Service 看做是部署在该计算机中的各个软件功能组件, 在此基础上建立新型的 Web 应用程序类型及其构造方法。从而, 从软件视角诠释网络就是计算机的深刻内涵。

更为深入地看, 功能范型的函数及其调用呈现线性特性, 而对象范型 (及以后的范型) 的对象及其交互关系呈现非线性平面 (及超平面) 特性, 范型蕴含的思维维度不断拓展。另外, 功能范型带来的程序流程基本控制结构及其递归应用, 实现了维拓展 (顺序结构) 和阶拓展 (分支结构和循环结构、语句的堆叠与嵌套结构), 两者的综合, 奠定了实现无限创造的基础。

近年来, 人工智能 (Artificial Intelligence, AI) 发展迅猛, 伴随着其发展, 各个领域都嵌入了 AI。事实上, 任何领域应用的发展, 其高级阶段必然都会需要 AI, 因此, 各领域嵌入 AI 也是其自身发展使然。作为现代计算基础, 程序范型必然也会嵌入 AI, 程序范型的发展趋势必然是一种智能件范型, 它从根本上构建统一的智能应用的基础结构, 将 AI 属性从应用层面提升到核心的范型层面, 实现抽象的深入。基于智能件范型, 也必将诞生一种崭新的智能型软件体系结构, 这是软件范型及软件体系结构自身发展的使然和必然。

2.6 本章小结

作为软件的主体, 程序及其构造基本范型是软件体系结构的基础, 对程序基本模型的理解可以提高对软件体系结构的认识, 特别是对程序基本模型演化动因和规律的认识, 有利于把握软件体系结构发展的脉络。本章为后面章节的展开建立思维基础。

习 题

- 1、程序基本模型对软件体系结构的作用主要体现在哪些方面?
- 2、请给出程序基本模型的基本发展脉络及其演化的规律。
- 3、如何理解程序基本模型是应用与技术的粘合剂?
- 4、什么是内存泄漏? 什么是无效引用? 请以 C/C++ 为例, 举例说明之。Java 和 C# 中是通过什么技术解决内存泄漏问题的?
- 5、功能模型的两个核心问题是什么? 功能模型的主要缺点是什么?
- 6、请举出两个递归应用的例子。

-
- 7、所谓递归，即是分解（递）和综合（归）。以求 P_5^3 和 C_5^3 为例，解析其递归求解的过程，给出分解阶段和综合阶段。
- 6、对象模型的三个基本要素是什么？
- 8、分治法也是用同样的处理方法处理不断缩小的数据集，以最终求解。它与递归方法的区别是什么？
- 9、请举例说明对象、类和实例三个概念之间的区别和联系。
- 10、什么是抽象数据类型？为什么说抽象数据类型可以实现数据类型的扩展？如何理解递归思想在抽象数据类型中的应用？
- 11、什么是多态？什么是静态多态？什么是动态多态？请用 C++ 举例说明。
- 12、相对于复合数据类型，ADT（抽象数据类型）为什么灵活性和扩展性都要强？
- 13、从封装的角度看，“对象”只是封装数据，而“组件”则封装数据和处理。关于这一点，你是如何理解的？
- 14、对象可以理解为“具有责任的东西”，对象应该清楚地定义责任，并且自己负责自己。请问：对象的责任是通过什么描述的？它与功能模型有什么联系？
- 15、什么是逻辑组件（组件对象及其关系）？什么是物理组件（封装结构）？对于内存芯片和内存条，哪个属于逻辑组件，哪个属于物理组件？
- 16、什么是垂直框架、水平框架和复合文档框架？请给出利用复合文档框架构造程序的基本步骤。
- 17、普通组件和可配置组件，在基本逻辑结构上有何区别？
- 18、配置型组件模型中，组件实例通过上下文使用系统资源，而上下文由服务容器进行控制，这种运行结构对系统资源的有效利用有什么好处？
- 19、请指出下列 COM+ 相关机制在 .NET 中的对应物：
- 1) 接口 IUnknown;
 - 2) 类工厂;
 - 3) 引用计数;
 - 4) IDL 文件或类型库;
 - 5) GUID;
- 20、.NET 模块文件中必须的要素是什么？.NET 装配件中必须的要素是什么？
- 21、相对 COM、COM+ 的组件封装机制，.NET 装配件采用两层封装机制（逻辑组件和物理文件）的优点是什么？
- 22、.NET 中，装配件是逻辑封装机制，DLL、EXE 或模块是物理封装机制，请问真正的组件是什么？它位于何处？请举例说明。
- 23、.NET 中，AppDomain 和装配件的关系是什么？它们分别与 COM+ 中的什么概念相对应？（提示：可部署包、应用程序、可配置组件模型与组件包、组件的封装件）
- 24、XML Infoset 所定义的抽象信息模型是怎样的？
- 25、XML 1.0 是如何给 Infoset 的抽象模型定义其面向串型化的语法结构的？它规定的一个文档的结构是怎样的？

-
- 26、Namespace 规范的作用是什么？为什么要建立这种规范？
- 27、XML Schema 是干什么的？与 DTD 相比它有哪些主要优点？
- 28、给出 XPath 的基本结构，并说明每个部分的作用。举例说明 XPath 的应用。
- 29、给出 XPointer 的基本结构，并说明它与 XPath 的关系。举例说明 XPointer 的应用。
- 30、请给出 XML1.0+NameSpace、SOAP 的 Schema、SOAP、一个具体的 SOAP 报文四者之间的关系，并以编译原理中的文法和语言为例，分别找出两者之间四层抽象关系的对应视图。
- 31、Web Services 中是如何描述一个服务的？一个 Web Service 和一个组件，在基本逻辑结构上有何区别？
- 32、如果将服务看成是一种组件，则 XML 也可以看作是一种组件技术。如何理解这一点？（提示：从 XML 的描述和集成能力着手）
- 33、传统的 Web 应用以网站上部署的整个应用为基础，而 Web Services 使新型的 Web 应用以网站上部署的 Web Service（服务组件）为基础，这两者有什么本质区别？

拓展与思考：

- 34、配置型组件模型主要解决如何在客户端之间共享（服务器）资源的问题。如何理解这一点？它是如何实现的？
- 35、什么是元模型？为什么说程序基本模型是一种元模型？
- 36、针对功能模型，C 语言中函数调用有 PASCAL、CEDEL、STDCALL 三种调用形式，请详细说明这三种调用机制的含义。
- 37、分层处理方法的本质是什么？如何理解分层处理方法在配置型组件模型中的作用？（提示：分布式计算模型中，两端代理之间的关系是紧耦合的，一般不能通过改变代理来实现服务端基础服务的应用。配置型组件模型中，通过在代理外面增加容器机制实现扩展的灵活性）。
- 38、如果将整个互联网看做是一台计算机，请你畅想一下软件视图、硬件视图和系统视图，并将目前你所知道的各种软硬件技术映射到相应的视图中。
- 39、对于数学模型和非数学模型，请各举出一个例子。
- 40、对于 5 个客户端、每个客户端每分钟 1000 个请求和 1000 个客户端、每个客户端每分钟 5 个请求这两种情况，服务器在处理上有何区别？哪种会发生拥塞和崩溃？为什么？
- 41、对于如下两种分布式对象计算应用场景，哪一种使用池机制比较好？为什么？
- 1) 为了向用户提供丰富的体验，胖客户端应用程序长时间占用对象，而只在其中一小段时间内使用对象；
 - 2) 面向众多的客户访问，瘦客户端需要不断地为每个客户端建立对象、分配资源、执行计算机之间或进程之间的相关调用、然后清除对象。
- 42、“对象”是递归的化生。关于这一点，你是如何理解的？
- 43、处理逻辑递归应用的哲学含义

junshen