第一章　面向对象计算概述

本章首先回顾了面向对象的历史，然后，以抽象为工具讨论了面向对象计算的原理和模型，这些原理和模型被以后用于讨论各种面向对象的技术。

1.1　面向对象的由来和发展

面向对象方法吸取了程序设计语言和数据建模技术等有益的成果，经过近二十年的演变和发展，逐渐形成了自己的范型，为人们提供了更强的问题求解能力。

最早的计算机语言是汇编语言。汇编语言使用符号来表示机器指令，这比程序员直接使用二进制的机器指令编程显然方便多了。在五十年代中期开发的 FORTRAN 语言是第一个具有划时代意义的程序设计语言，许多重要的程度设计语言概念，例如变量、数组和控制结构(循环和条件分枝)等被引入到程序设计语言中来。但在程序设计实践中人们发现，在一个 FORTRAN 程序的不同部分的变量名总容易发生混淆，COMMON 数据块的使用也不便于程序的查错和修改。于是，在五十年代后期，Algol 语言的设计者决定在程序段内部对变量实施隔离。因此，在 Algol 60 语言中出现了由"Begin……End"提供的块结构(类似于 C 语言中的块语句)，在一个块内出现的变量，只为这个块知道，这样，对它们的使用就不会与程序中其它块中的同名变量相混淆。这是程序设计语言中第一次尝试为数据提供保护和封装。

在六十年代开发的 Simula 67 语言现在被人们公认为是面向对象语言的鼻祖。虽然它是一个通用的程序设计语言，但直到八十年代中期之前，它主要的用途却是用来进行仿真建模。Simula 67 是在 Algol 语言的基础上开发的，它将 Algol 语言中的块结构概念向前发展了一步，引入了对象的概念，对象代表着仿真中的一个实体(例如一辆汽车，一个顾客或一个服务员等)，在仿真期间，一个对象可以以某种形式与其它对象通信。从概念上讲，一个对象是既包含有数据又包含有处理这些数据的操作的一个程序单元。Simula 也使用了类的概念，类用于描述特性相同或相似的一组对象的结构和行为。Simula 也支持类继承。继承将类组织成层次，允许共享结构和行为，因此，Simula 奠定了面向对象语言的基础和某些面向对象的术语。Simula 在 1986 年进行了标准化，有几个公司在其计算机平台上实现了 Simula。

为了管理大型程序的需要，在七十年代出现了数据抽象的概念。数据抽象是一个数据结构及作用于该数据结构上的操作组成的一个实体或单元，数据结构的表示被隐藏在操作接口的后面，这样，通过操作接口，外部只知道它做什么，而不知道如何做，数据如何表示也是外部不知道的。将类型的概念扩展到数据抽象，即将那些主要是针对某个类型(例如学生类型)的操作聚集起来作为一个整体来看待，并与该类型一起看作一个独立的单元，将操作的语义作为该类型的定义，数据类型的值集由操作集间接定义，就构成了一个抽象数据类型。

支持抽象数据类型的最重要的语言之一是由美国国防部主持开发的 Ada 语言，它是一个被用来开发嵌入式实时系统的语言，这个语言包含一些常见的程序控制流构造(例如，选择、循环等)和具有定义新的类型、函数和子例程的能力，Ada 语言中面向对象的构造是包。有关 Ada 是否是面向对象的语言的讨论持续了很长时间，Ada 虽然支持诸如抽象数据类型、函数和运算

符重载以及参数化多态性等面向对象的机制，但由于它不全面地支持继承，因而被认为是一种基于对象的语言。

在七十年代和八十年代这一时期，来自于 Simula 和其它早期的原型语言中的面向对象的概念在 Smalltalk 语言中做了完整的体现。Smalltalk 是当今最有影响的面向对象语言之一。Smalltalk 语言并入了 Simula 的许多面向对象的特征，包括类、继承，并支持对象标识。而且，在 Smalltalk 语言中，信息隐藏比 Simula 更严格。

Smalltalk 语言极大地丰富了面向对象的概念。在 Smalltalk 中，每个东西都是对象，包括类。这也就是说，在 Smalltalk 环境中，程序设计只是向对象发送消息，一个消息可以是两个数相加，或创建一个类的新的对象，或在一个类中加入一个新的方法（数据被处理的方式）。Smalltalk 语言是一种弱类型化的语言。一个程序中的同一个对象在不同时间可以表现为不同的类型。

自从 1986 年以来，面向对象的技术逐渐走出了实验室和研究部门，开始进入实际应用。现在，在工业和商业上更多地采用面向对象的问题求解方法。Object-C、C++和 Eiffel 等语言都是继 Smalltalk 语言之后有广泛影响的面向对象的程序设计语言。对这些语言可进行如下分类：

（1）开发全新的面向对象语言，其代表是 Object-C、Smalltalk 和 Eiffel。Object-C 在 1983 年前后开发，它是对 C 进行扩充之后形成的面向对象的语言，但它的语法更象 Smalltalk，可以说，它是一个 Smalltalk 语言的变种。Object-C 并不是象 C++那样通过扩充已有的 C 语言的构造来提供面向对象的支持的，而是完全地依赖于新引入的构造和运算符来完成类定义和消息传递等。Eiffel 是除 C++之外，另一个商业上有潜力的面向对象的语言。Eiffel 除了封装和继承之外，该语言还集成了几个强有力的面向对象的特征，例如，参数化多态性（类似于 C++中的模板或 Ada 中的类属），对方法实施前置条件和后置条件断言等。从理论上说，Eiffel 是最好的面向对象的语言。Smalltalk 在七十年代开始开发，经历了 Smalltalk-72、Smalltalk-76 和 Smalltalk-80 等几个版本，目前，人们在谈论 Smalltalk 语言或系统时，多指 Smalltlk-80。这类语言的缺点是，程序员需要从头学习一门全新的语言。

（2）对传统语言进行面向对象的扩充，这类语言又称为混合型语言。其主要代表是 C++。C++是在八十年代早期由贝尔实验室设计的一个面向对象的语言，它是在 C 语言的基础上增加了对面向对象程序设计的支持。C++为类定义提供了两种构造，其一是对结构（struct）进行扩充，其二是新的类（class）构造。混合语言的特点是，既支持传统的程序设计方法，又支持面向对象的程序设计方法，有丰富的应用基础和开发环境的支持，熟悉传统语言的程序员可以很快地学会使用这种语言，这是 C++能够很快普及的原因。

美国 Byte 杂志在 1983 年和 1986 年以专刊的形式发表了一系列介绍面向对象程序设计方法的文章，1986 年在美国举办的第一届面向对象的程序设计语言、系统和应用（OOPSLA'86）国际会议，使面向对象受到世人的重视。到目前为止，从所发表的论文中可以看出，许多研究者都在期望使用面向对象的技术来解决各种问题。面向对象的方法已被广泛地应用于程序设计语言、设计方法学、形式定义、操作系统、分布式系统、人工智能、实时系统、数据库、人机接口，甚至硬件设计。这种情况的出现有许多原因，动力之一是工作站技术的快速发展。现代的工作站能够支持复杂的个人程序设计环境，而且提供了基于图形的人机界面。二进制大对象（BLOB——Binary Large OBject，例如存储声音和图象数据等的对象）、可视编程、分布式处理和用户对控制应用程序应做什么等的需求和发展，都是面向对象计算发展的动因。

1.2 程序设计范型

范型(paradigm)指的是一种通用的一般化的关于现实世界的模型，这个模型为我们提供了观察和研究现实世界的一个视图。换句话说，范型就是被大家共享的概念集。程序设计范型是关于计算机系统的思考方法，是分类一组语言的抽象特征的集合，它体现了一类语言的主要特点，这些特点能用来支持应用域所希望的设计风格。

面向过程的范型是广为大家了解的一种程序设计范型，象 C 和 Pascal 等大多数程序设计语言都属于这一类范型，我们称它们为面向过程的语言。在这种范型中，语言基本上是命令式的，命令是某个程序段，执行这个程序段的目的是为了更新对象，命令执行的时序已在程序中预先做了规定，即语句顺序。这类语言还提供了顺序、循环和选择控制结构。在这种范型中，程序设计的首要问题是设计过程。基于面向过程的程序设计范型和功能分解方法的软件设计技术形成了结构化软件开发方法的基础。

随着程序规模的增大，数据的组织成为一个重要的问题。1972 年，Parnas 提出了著名的信息隐藏原理。其基本思想是，把需求和求解的方法相分离，把相关信息——数据结构和算法——集中在一个模块中，和其它模块隔离，其它模块不能随便访问这个模块的内部信息，只能通过严格定义的接口进行。在信息隐藏原理的指导下，产生了模块化程序设计范型，例如，Modula-2 语言是属于这一范型的程序设计语言。在这种程序设计范型中，程序设计的首要问题是划分模块，数据结构被隐藏在模块中。每个模块都有一个接口，模块的表示只能通过接口进行访问。

在程序分解成模块的概念形成以后，人们很快认识到，把模块中有关实现的详细信息应予以局部化，把模块类型化，对一种类型的模块设置足够的操作集，由此形成了抽象数据类型的概念。程序设计的准则将注意力集中到决定所需要的类型。抽象数据类型对程序设计的影响很大，面向对象的程序设计即是以抽象数据类型为重要基础。

面向对象的程序设计范型就是在以上程序设计范型之上发展起来的，它的关键在于加入了类及其继承性，对象以类为样板被创建。这种范型将计算视为一个系统开发过程，系统由对象组成，对象经历一系列的状态变化以完成计算任务。在面向对象的程序设计范型中，首要的任务是决定所需要的类，每个类应设置足够的操作，并利用继承机制显式地共享共同的特性。

1.3 抽象在问题求解中的作用

从 1.1 节的介绍中可以发现，程序设计语言的发展历史是语言提供的抽象支持程度不断提高的历史。面向对象的程序设计范型比传统的程序设计范型更加强调抽象在软件开发中的重要性。研究面向对象的计算不可避免地要对抽象进行认真地研究。

从一般观点而言，抽象是通过从特定的实例或例子中抽取共同性质以形成一般化的概念的过程。抽象是对系统的简化描述或规范说明，它强调系统中某一部分细节或特性，而忽略了其它部分。抽象的描述被称为它的规范说明，而将对抽象的解释称为它的实现。抽象是有层次的，即高层抽象将其低一层的抽象作为它的实现。

抽象是人类在两种不同的领域中处理复杂性的主要工具：在理解复杂现象和求解复杂问题中。这两个领域有根本的不同，前者关心组织关于世界的观点，而后者则集中于应用这些观

点作为问题求解的方法。

作为辅助理解的工具，抽象被用于建立论域的概念模型。一般化的概念通过观察过程被识别出来，并通过与一个标签或名字相关联而被定义下来。这个名字然后被用于对这个概念的各种属性和行为进行抽象。这种形式的抽象的一个主要例子是生物学。经过长期的工作，一系列的生物现象被进行了详细的分类，现在已代表着一个成熟的学科。例如，术语食肉类动物现在被通用来表示共享某些行为或拥有某些特性的动物。

抽象对于辅助求解复杂的问题也是很有价值的。其普遍的方法是将复杂问题分解为几个子问题，然后集中于单个的子问题上。这些子问题可以被进一步分解，直到它能被容易求解为止。通过定义与问题域的一个特定的分解相对应的子问题，抽象在这个过程中是显而易见的。子问题本身也是问题，并可被单独处理。一旦求解完成，解的详细内容可以通过这些结果抽象出来。

在计算中，抽象也是一种非常重要的工具。计算的大部分历史都与提供合适的工具和技术以支持某种抽象过程有关。在计算中，摆在一个程序员面前的问题通常是开发一个系统的某部分以完成一个特定的任务，例如，管理一个公司的财务系统。求解这个问题包括与开发过程的各个阶段相对应的各级抽象。在最高级，存在着系统必须满足的某些抽象需求。紧接着，系统开发者产生出在求解这个问题的过程中所需组件的经验模型。这个抽象模型必须被映射到被选择来解决这个问题的工具和技术所提供的抽象上。最后，这些工具和技术本身必须使用计算机系统的硬件所提供的抽象来实现。

目前，计算机科学家所关心的最重要的抽象层是提供合适的工具和技术。事实上，计算预先已被这个层所能提供的工具和技术所左右。主要的困难是解决两个通常是相互矛盾的准则：

- 工具和技术必须很好地适应所要处理的问题
- 工具和技术必须能在基础硬件上有效实现。

这对应于从用户模型到工具和技术的映射，和从工具和技术到机器代码指令的映射之间的矛盾。在工具和技术这一层，这个矛盾必须解决。

人们一直试图定义合适的抽象来支持问题求解。有些技术被开发用来针对特定的问题域，而其它一些技术则试图支持通用问题求解。然而，合理的说法是，在开发工具和技术来处理复杂性方面，计算机科学只取得了有限的成功。

在历史上，计算机系统所提供的许多抽象是为了适应基础硬件结构，而不是为了适应问题域。许多开发工作使用 FORTRAN 和 Pascal 等命令式语言来进行。这些语言所提供的抽象很显然被设计成能在冯·诺依曼(Von Neumann)结构的计算机上有效实现，例如，变量映射为存储单元，循环映射为转移指令等。这样做的原因完全是为了效率。

在人工智能领域还出现其它抽象方法。在人工智能领域中遇到的大量问题是需要工具和技术能更好支持问题域。在人工智能中最为成功的方法之一是基于逻辑来提供工具和技术。原因是认为，逻辑更符合人类处理复杂问题的方式，因此，能支持更优雅和更具表达力的问题求解方法。例如，Prolog 语言就是基于一阶逻辑提供的抽象的语言。

在基于面向问题的抽象的工具和技术中存在有风险，即很难在已有的硬件上进行有效的实现，这是 Prolog 等语言从一开始就存在的问题。另一个问题是，它们过于关注特定的问题域，因此，很难被应用于各类应用中。

1.4 面向对象抽象的原理

面向对象的计算通过特定的方法提供抽象。面向对象计算的观点是提供工具和技术以支持问题求解，虽然实现效率不是考虑的重点，但现在已有了高效实现的软件系统。面向对象的抽象是非常通用的，因此，适用于大范围的应用域。为达到这一目的，重要的是抓住面向对象的抽象问题求解原理，这些原理适用于各类可能的应用。然后，考察支持这些原理的特定的工具和技术。

下面将要讨论的四个原理定义了面向对象抽象的方法，这四个原理是：数据抽象；行为共享；进化；正确性。这四个原理概括了面向对象计算的本质。

1.4.1 数据抽象

数据抽象为程序员提供了一种较高级的对数据和为操作数据所需要的算法的抽象。用户的视图是一系列的操作，这些操作集中在一起定义了这个抽象的行为；用户不必知道内部实现细节。数据抽象实际上包含了两个独立的但又密切相关的概念：模块化和信息隐藏。

模块化指的是将一个复杂系统分解为几个自包含的实体(即模块)，与系统中一个特定的实体有关的信息保持在该模块内。这样，一个模块是对整个系统结构的某一部分的一个自包含的和完整的描述。从计算的术语来说，这表示一个模块将包含为实现系统的这个部分所需要的所有数据结构和算法。模块化的优点是，当需要进行修改或出现问题时，我们可以立即确定需要在哪些模块上进行。模块化强调了这样一种设计方法：程序员将问题域分解为几个可识别的概念实体。这种设计方法构成面向对象计算的本质。

信息隐藏通过将一个模块的细节对用户隐藏起来将抽象的级别向前推进了一步。使用信息隐藏，用户必须通过一个受保护的接口访问一个实体，而不能直接访问诸如数据结构或局部过程等这些内部细节。这个接口一般由一些操作组成，正如前面所言，这些操作定义了一个实体的行为。在处理复杂性中，这是一个主要工具，因为它允许用户在系统的某层上进行抽象。通过严格控制一个模块的入口点，信息隐藏也支持开发更可靠的程序。由于不能直接访问数据结构，因此，也就不可能对数据执行不期望的操作。访问数据结构的唯一途径是通过操作接口。对操作接口的完全测试为一个模块的正确性提供了高的置信度。类似地，在支持信息隐藏的系统中，错误的影响通常被限制在一个模块内。因此，就没有可能使一个模块破坏另一个模块，这增强了一个系统的可靠性。

在实践中，一个实体的行为可以两种方式来表示：使用过程接口抽象地表示，或使用接口和相应的实现具体地表示。面向对象的计算在两种情形中都使用术语行为。在后面的章节中将看到，将抽象的规范说明与实现相分离，以更抽象的形式解释行为是有益的。

数据抽象一般被认为是迈向更加结构化的程序设计的一个重大步骤。从本书的角度而言，数据抽象的重要性在于，它提供了面向对象计算的始点，即系统应该被分解为概念上的实体，实体的内部细节应该被隐藏起来。由模块化和信息隐藏这个过程所提供的抽象是面向对象方法的核心，实现面向对象方法的首要一点是提供技术支持数据抽象。

1.4.2 行为共享

面向对象计算的第二个原理是系统应该支持行为共享。数据抽象引入了一个重要概念

——行为，即实体可以以它们的外部接口进行定义。行为共享通过允许许多实体具有相同的接口集将这个概念又向前进行发展。这可以在很大程度上增加系统的灵活性。例如，考虑打印一个实体的动作，利用数据抽象，可以定义一个 print 操作显示数据结构的一种特定的表示。将 print 操作定义在几个实体上也是很自然的。这精确地说明了行为共享的含义，即用于打印的接口被系统中的几个实体共享，虽然各自的实现可能不同。另一种方案是为系统中的每个实体定义不同的打印操作，例如，printX 和 printY，从这个例子中可以看出，行为共享极大地增强了在一个系统中的抽象；print 是一个比 printX 或 printY 更抽象的概念。

行为共享可以以几种不同的方式提供。在面向对象的计算中普遍使用下面两种方式：

• **分类(classification)**：支持行为共享最明显的方式是在系统中引入分类。一个分类是基于一组实体共同的行为而形成的。因此，由定义可知，在一个特定的分类中的所有实体将共享共同的行为。例如，一个队列可以被归类为一个支持插入、删除和打印的实体，这样，所有的队列将共享插入、删除和打印行为。

• **层次分类(taxonomies)**：层次分类是面向对象计算中一种非常普遍的行为共享形式。实际上，层次分类代表着面向对象方法的最主要的特征。层次分类是分类的求精，它允许一个分类包含另一个分类。包含隐含着行为共享的一种更受控的形式。假设分类 A 被包含在分类 B 中，A 将表现出特定的行为，B 将共享这些行为，但是 B 还具有它自己的其它行为。例如，具有行为 push 和 pop 的一个栈 stack 被包含在具有行为 push，pop 和 print 的栈 new_stack 中。这种共享形式的结果是，基于包含关系建立了一个层次的结构。

相对于 1.4.1，行为共享既可以基于规范说明的共享，也可以基于规范说明和实现的共享。

行为共享是面向对象的计算中的另一个较为重要的概念。实现面向对象方法的一个重要任务是为基于一个或多个分类的行为共享研究特定的技术。

1.4.3 进化

面向对象计算的第三个原理是支持进化，这是考虑到在计算环境中需求会很快地发生变化。过了一段时间，系统发生变化，需要增加其它功能。面向对象的方法将支持进化过程作为计算模型的一个基本方面。要考虑进化的两个方面：

• **需求的进化**：一旦开始开发一个系统，很有可能发生需要修改或增加需求的情况，尤其是在一个动态变化的环境中，很难获取稳定的需求。

• **进化式问题求解**：进化的一个更一般的观点是从初始实验到最终结果以一种增量的方式进行问题求解。当一个系统的最终目标不能进行很好的定义时，这种方法就很有吸引力。

面向对象的观点就是提供一种单一的方法来包括这两个方面。因此，进化涉及一个系统从初始试验一直到后续的维护整个生命期。一般的方法是通过从当前系统到最终系统经过一系列渐进步骤逐步得到最终解。程序设计中这种进化方法是面向对象的计算所特有的。

1.4.4 正确性

在计算中，正确性有许多解释，例如程序正确性，相容测试或容错等。在本书中，正确性具有较特殊的含义。这个术语用于描述一个系统确定的行为。为说明这个概念，考虑由几个互相作用的对象组成的一个系统。每个对象向系统中的其它对象给出一个行为规范说明。在某一点，一个对象可能需要另一个对象执行某个操作。换句话说，一个特定的行为项被请求。一个

确定的系统是这样的一个系统，在这个系统中，确保对这个行为项有一个解释，即系统决不会因为不能响应一个操作而失效。确定性在许多大型和复杂系统中是一个重要的特性，例如，在实时控制应用(例如，核电站计算机控制系统)中，由于不能对一个外部事件进行响应而出现错误是不能接受的。

上面所描述的系统的确定性与类型特别是类型的正确性有关。事实上，这种保证要求类型错误不会在一个系统中出现。然而，面向对象系统的某些特征使正确性比传统的类型化的系统更难于处理。特别是行为共享和进化增加了确保正确性的复杂性。由于这种动态和灵活的环境，断定一个系统的全部行为是非常困难的。虽然进行近似的考察，有可能确定一个行为项是否有一个解释，但不可能确定这个解释是什么，因为解释可能随时间变化。因此，面向对象系统中的确定性与抽象的规范说明层有关，而与实现无关。

1.5 面向对象计算的模型

上节所述的原理揭示了面向对象计算的本质，描述了在计算机系统中处理复杂性问题的一种固有的方法。然而，为从中受益，必须将这些抽象的原理转化为一系列特定的技术，这里，技术一词指为实现面向对象的原理所需要的关于实现的机制。

面向对象的系统提供特定的技术来支持这些原理，有些技术只支持特定的某个原理，而有些技术却用于支持多个原理，这节所提出的模型将这些技术进行了分类，使我们可以从一般的观点来考察什么是面向对象，同时，也可以更清楚地明确原理和技术的关系。这个模型将技术分成四类，在面向对象的系统中，每一类起着特定的作用。这四类是：

- **封装**：封装是一个通用术语，用于指那些实现数据抽象的技术。因此，封装隐含着提供机制以支持模块化和信息隐藏。这样，在支持封装的技术和数据抽象原理之间存在着一一对应的关系。因此，封装提供了数据抽象所能提供的所有益处，例如，提供概念构造块和保护。
- **分类**：面向对象的系统一般都提供有一项或多项技术来支持具有相似行为的对象的分类。这组技术对应于1.4.2节描述的行为共享方法的第一类。为实现分类，必须提供相应的机制以支持以共同的行为构成分组。这里，行为或是由规范说明来定义，或是使用规范说明和实现来定义。这样就允许用户基于在一个组中的包含关系推理某个对象的行为。同样，也能够在一个分组中的所有对象之间共享行为。
- **灵活的共享**：这类技术强化了系统的行为共享，同样的技术也用于支持进化，因此，它包括了面向对象计算的原理中的两个原理。在类型化的系统中，灵活的共享被称为多态性。
- **解释**：面向对象系统中有了灵活和进化的行为共享形式，还必须提供技术来解决一个行为项的精确解释。有各种不同的技术用于支持解释。这些技术的区别是进行这种解释的时间，即是在编译时进行，还是在运行时进行。编译时进行的技术可以保证面向对象系统的正确性，相比较而言，运行时进行的技术有更大的灵活性。

上述的四个方面组成面向对象系统的一个模型，这个模型从各个方面详细陈述了面向对象计算的本质。封装是面向对象计算的最基本的支持解释，并且是面向对象的所有益处(例如模块性和维护性)的根源，而分类通过允许在封装的对象集上形成分组增加了这些益处，而且可以通过取消它们的差异，强调它们共享的行为，来对这些组进行推理。另外，可以形成不同的分类以满足不同的需要，例如，抽象数据类型和类都是合法的分类。而多态性通过允许存在对象的不同解释为系统注入了一定程度的灵活性。可以提供各种形式的多态性，例如，重载、参数化多态

性和包含多态性。继承也可以被认为是一种形式的多态性。最后，解释用于确定一个操作的精确语义，这在一个多态的面向对象的环境中是必要的。关键的地方是，这个解释是静态执行的(在编译时)，还是动态执行的(在运行时)。

应强调一点的是，模型不是想为面向对象作一定义，重要的是避开这种观点，即在面向对象领域中对面向对象这一术语存在有绝对的定义。现在，不可能，可能也不值得为面向对象给出一个精确的定义。相反，模型应该是讨论各种实现策略的一个框架。满足上述各方面的所有语言和系统可以合法地在面向对象这个论域中进行讨论。讨论应集中于不同策略的相对价值，而不是一个特定的语言是否是面向对象的。下面详细讨论每个方面，包括每个方面的精确性质和各种实现策略。

1.5.1 封装

封装是将系统中的一个可识认的实体的各种特性聚集在一个词法的或逻辑的单元即对象中，而且，对对象的访问被限制在一个严格定义的接口上。面向对象计算的所有方面始于这个基本概念，即现实世界可以被描绘为一系列完全自治的、封装的对象，这些对象通过一个受保护的接口访问其它对象。封装是一种通过定义严格的外部接口在单独编写的模块之间减少相互依赖的技术。一个模块的外部接口在模块和它的用户之间(因此也就是在模块的设计者和其它设计者之间)起着契约的作用。一个模块可以被重新实现，只要新的实现支持相同的(或向上兼容的)外部接口，这种改变就不会影响它的用户，因此，就可以控制这种改变所产生的后果。

上述定义中所使用的术语特性从一般意义上讲，是对象所固有的信息。不同系统在对各种特性的支持上有变化。下面是一个对象的可能特性的一个完整的列表：

- **表示**：一个对象的表示是用于模似和记录这个对象状态的内部数据结构。例如，一个栈表示可能是一个链表数据结构。
- **操作**：一个对象的操作是与处理这个对象的内部状态的过程和函数的接口。例如，栈对象可能支持 push，pop 和 is empty 操作。
- **算法**：算法则是定义在一个对象上的过程和函数的实现。
- **属性**：属性是值的名字，用于增强一个对象的描述。例如，一个栈可能具有一个定义栈的最大尺寸的属性。
- **约束**：有几个对象模型也封装了对象所具有的约束。约束对一个对象的行为强加一些限制，例如，禁止在一个空栈上进行 pop 操作。实现约束的一种方法是在操作的开始和结束处设置断言，以检查操作的一致性，例如，Eiffel 语言分别使用关键字 require 和 ensure 引出前置条件和后置条件，或使用关键字 check 设置检查条件等实现断言。在有的 C++系统中，使用宏 PRECONDITIONX()和宏 CHECKX()也可以达到类似的效果。
- **触发**：有些实现将触发与一个对象关联在一起。一个触发是一个异步动作，当状态发生变化时被唤醒。例如，一个触发可以与地址的变化相关联，以便更新被其它对象管理的邮件表。在模拟与某个操作不直接相关的对象的行为时，触发是很有用的。

属性和约束常出现在面向对象的数据库中，而在程序设计语言中，封装的则是表示、操作和算法。在办公室信息系统中，则强调触发这一特征，以提供对各种办公过程的支持。

一旦定义了一个对象的特性，则有必要决定这些特性的可见性，即哪些特性对外部世界是可见的，哪些特性用于表示内部状态。在这个阶段定义对象的接口。通常，应禁止直接访问一个对象的实际表示，而应该通过操作接口访问对象，这通常被称为信息隐藏。在本书中，一个对

象的可见接口被称为这个对象的(外部)行为。

1.5.2 分类

在支持封装的系统中,环境由一个有限的对象集合来表示,每个对象表现出特定的行为,即外部特性。将这种模型扩充为包含分类则是非常自然的。分类是依据共同的行为将有关的对象进行分组。在一个特定分组内的所有对象将共享这个分组中所有公共的特性,但可以有其它方面的差异。分类是一个非常有力的工具,并且在许多科学和工程训练中是很重要的环节。

一般而言,一个分类通过对环境中的对象所作的一个断言来定义。例如,一个有效的分类可以是颜色属性为红的所有对象。在这个分组中的所有对象确保满足这个断言,但可以有其它的差异。一个特定的分类也具有一个内涵和一个外延。一个分类的内涵是这个分类的行为的描述,内涵代表呈现某个特定行为的所有可能的对象,因此,潜在地讲是无限的。相对比而言,一个分类的外延是当前环境中呈现这种行为特征的对象的有限集合。因此,一个分类的内涵由一个特定的断言来定义,而外延则通过将这个断言应用到一个特定的环境来获得。

例如,考虑由下列值组成的一个环境:

46, "Gordon", −10, True, 12, "Blair"。

一个断言被定义为:

$\{x : x \in N\}$ (这里 N 是所有自然数的集合)

这个断言的内涵是所有自然数的集合,即{1, 2, 3, 4, 5, …},然而外延是包含 46 和 12 的集合。

在面向对象计算中,内涵和外延这两个方面都是重要的。特别是在面向对象的数据库中,必须能够表示内涵和外延这些概念。

在系统中可以存在各种分类来表达不同的分组形式。重要的问题是,在一个特定的系统中应表示哪些分类。上述的例子是以对象的属性描述的一种分类形式。另一种可能的分类形式是以对象提供的操作来定义分类。例如,一种有用的分类是支持 open, read, write 和 close 操作的对象集。在一个特定的面向对象的系统中选择如何模拟分类是设计过程的根本焦点。最有意义的选择如下:

- **集合(Set)**:在一个系统中表示分类最一般的方式是使用集合。集合足能够来模似--个系统中所需的任何可能的分类。任何断言的结果是满足这个断言的对象的一个集合。在面向对象的语言中很少出现集合,语义数据模型则是基于简单的集合理论。使用集合的优势是,存在来自于数学的有关知识和很好理解的语义。集合的内涵由一个特定的断言或查询给出,外延则由能回答这个断言或查询的对象给出。

- **抽象数据类型**:类型本质上是一种分类形式。然而,它是比集合更特殊的分类方法,特殊之处在于,它仅关心对象的外部接口。一个类型是对象的一个集合,这些对象共享共同的行为。因此,所有的类型是集合,但集合不是类型。抽象数据类型则是定义一个特定分类的样板。在效果上,断言由满足抽象数据类型语法和可能语义的对象的集合给出。因此,这种分类的内涵是抽象数据类型的描述,而外延是所有这些对象的集合,这些对象被认为具有这种抽象数据类型。

- **类(具体数据类型)**:在面向对象计算中出现的最受限制的分类形式是类。具有讽刺意味的是,许多人认为这是与面向对象的计算相容的分类。类是--个样板,以操作、表示和算法的形式(更一般地讲,是全部的内部和外部行为)完整地定义了一组对象的行为。因此,类可以被

认为是一种具体数据类型，完整地描述了规范说明和实现。类的内涵由类样板给出，而外延是一个特定类的有关对象。

因此，分类是一个非常一般的概念，它包含了集合、抽象数据类型和类。重要的是明白，选择一种分类方法并不排斥选择另一种分类方法。在有些语言中，抽象数据类型和类共同存在。同样，在一些数据库模型中，集合和类共同存在。

1.5.3 多态性

多态性隐含着表明对象可以属于多于一个的分类，因此分类可以重叠和相交。这样，就有可能让两个不同的分类共享共同的行为。多态性可以应用于各种分类方法。例如，对象能够属于多个分类可以解释为对象可以具有多种的类型，或对象可以属于多个类，因此，多态性是一个与分类正交的概念。

多态性的强有力之处在于它支持更灵活的计算模型。在非多态系统(即单态系统)中，对象只能属于一个分类，因此使不同的分类共享共同的行为是不可能的。分类变成了保护墙，将对象隔离成各种不相交的类别。然而，在多态环境中，分类可以重叠和相交，因此，使两个分类共享共同的行为是可能的。同样，使一个分类具有另一个分类的行为的一个子集也是可能的。

在一个面向对象的语言中存在多种风格的多态性是可能的，然而，只有包含多态性真正使这种风格的程序设计成为一种典范。包含多态性支持一种强调分类之间关系的程序设计风格。在这种风格中，一个分类可以用于代替另一个分类，因为这个分类呈现出被代替分类的所有行为。分类因而不再是孤立存在的，而是通过一系列的 IS-A 关系内部相关的。如果 C1 呈现了 C2 的所有行为，则称 C1 是一个 C2。在后面，我们将从类型的角度详细讨论这种形式的多态性，但结论同样适用于其它分类方法。在类型化系统中，IS-A 关系经常被称为子类型化关系。在类中，IS-A 关系被称为子类化关系。在所有的情况下，IS-A 关系为环境提供了一个偏序关系，因此建立了一个分类的格。

通常期望面向对象的语言和系统应该呈现包含多态性的特征，然而，主要的问题是，如何建立分类之间的关系。一种方法是从分类的结构中由系统隐式地建立分类之间的关系，这样，由于分类共享共同的行为而在它们之间存在关系。抽象数据类型的适应规则给出了使用隐式技术决定抽象数据类型之间的关系。另一种方法是由程序员显式地定义分类之间的关系，这样，由于它们建立的方式而使分类之间共享共同的行为，继承是通过显式陈述来建立类之间关系的例子，其优点是，由于一个分类是通过其它分类构造的，因此，可以再用其个组成要素。通过由已有的分类来建立新的分类直到最终的环境被建立为止，继承是一个理想的进行由底向上的设计的工具。

在面向对象的语言和系统中引入其它风格的多态性也是可能的。这可以用于增强包含多态性或可以被作为实现包含多态性的一种选择。主要的选择是重载和参数化多态性。

- **重载**，使用这种技术，名字可以在不同的语境中被重载来为不同的特性(例如属性或操作)署名。语境信息被用于解决模糊性。
- **参数化多态性**，这种多态性形式在面向对象的模型中常以类属出现。简单地说，类属是使用一个或多个类型参数参数化一段软件的能力。参数化的类型可以被以后用来构造软件要素。类属和包含多态性提供了不同但互补的多态性风格。类属提供了表示共享一个通用描述的几个不同的分类的速写形式，结果，类属节省了程序员书写一个通用算法在各种使用情况下需要对相同部分进行描述的时间；而包含多态性则提供了一种更一般的机制来表达分类之

间的关系。

1.5.4 解释

解释定义为解决多态性。在多态环境中，根据语境的不同，一个特定的行为项有几种不同的含义是可能的。一个行为项可能跨越许多分类，例如，print 操作在一个系统中可能有许多解释。

解释的任务就是要解决多态环境中存在的这种多义性，并决定在某个特定的语境中，一个操作的精确解释。这可以通过以下两个具体的步骤来完成：

- 决定在特定的语境中一个操作是否合法；
- 决定要被调用的精确实现。

这两步分别对应于类型检查和束定。类型检查决定一个特定的对象是否支持操作，而束定则确定操作的正确实现在哪里。

在这两种情况下，问题的焦点仍是解释过程的执行时机。存在两种可能的选择：多态行为或者在编译时解决，或者在运行时解决。这种选择等同地适用于类型检查和束定，引出下面的决策：

- 静态类型检查与动态类型检查；
- 静态束定和动态束定。

表 1-1 是上述各种组合情况的说明。

表 1-1 类型检查和束定策略的组合

	静态类型检查	动态类型检查
静态束定	确保正确性　解释方式不灵活	非法组合
动态束定	确保正确性　解释方式灵活	正确性无保证　解释方式灵活

在这一层上所做的决策对灵活性和正确性这些系统特性有很大的影响。

1.6 小结

抽象是问题求解的主要工具，本章介绍了面向对象抽象的四个原理，即数据抽象、行为共享、进化和正确性。这些原理概括了面向对象计算的本质。本章所介绍的面向对象的计算模型则给出了讨论面向对象计算的一个框架，在这个框架内，可以很容易地理解某个特定的技术所起的作用，这个模型比第二章所提到的传统模型更完善，特别适合于考察静态类型化的面向对象的语言。第二章和第三章将在这个框架内讨论各种技术。

练习

1.1 为什么说 Simula 是面向对象语言的鼻祖。

1.2 面向对象的范型与面向过程的范型各有什么特点。

1.3 简述抽象在计算中的意义。

1.4 解释下列原理：

数据抽象　行为共享　进化　正确性

1.5 什么是模块化，什么是信息隐藏。

1.6 面向对象的计算是怎样支持行为共享的。

1.7 名词解释：

封装 分类 灵活的共享 解释

1.8 在程序设计语言中强调封装对象的哪些特性。

1.9 面向对象的系统有哪些主要分类形式。

1.10 什么是多态性。解释包含多态、重载和参数化多态。

1.11 类型检查和束定的任务是什么。

1.12 各种类型检查方式与束定方式对灵活性和正确性有什么意义。

第二章　面向对象计算的基本特征

本章介绍针对面向对象计算来讲是核心的几个基础技术，并集中于对象、类和继承这些基本概念，对诸如消息传递、对象实例化、特化、方法束定和多态性也作了一定的解释。在本章中对技术性的描述相对简单，并常常简化了设计的一些细节特征，这样做的目的是使读者对面向对象系统有一个概略的了解，第三章将讨论其它细节。最后，通过对这些机制进行评价以及它们与上一章所介绍的面向对象计算的原理的关系对本章的内容进行了总结。

2.1　概述

在第一章，我们介绍了抽象的普遍概念对人类推理，特别是理解和解决复杂问题的价值；我们还给出了面向对象的抽象的四个原理，同时还强调了基于这四个原理出现了众多的设计。本章通过介绍面向对象计算的几个最基础的概念开始向读者介绍这些设计。本章讨论的技术是一些非常传统的面向对象的技术，它们来自早期在 Simula 以及以后在 Smalltalk 上所做的工作。

我们所采用的方法是先考虑面向对象计算的一个简单模型，然后逐渐介绍实现这个模型所需的技术。这个模型认为面向对象的系统包含三个要素，即对象、类和继承，如图 2-1 所示。该模型反映了看待面向对象的传统观点。

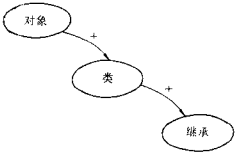

图 2-1　面向对象系统的一个简单模型

面向对象的许多工作来自于面向对象的语言、结果。毫不奇怪，对面向对象术语的解释也来自于这个领域的研究。一般的观点是，面向对象要经过下面一系列的步骤才达到真正的面向对象。

第一步，语言必须包括对象的概念，这里，一个对象是状态和操作的一个封装体，状态记忆操作的结果。满足这个标准的语言被认为是基于对象的语言。

第二步，语言应该支持类的概念和特征，类是以接口和实现定义对象行为的一个样板，对象由类来创建。支持对象和类的语言被称为基于类的语言。

最后，语言应该支持继承，即具有由已存在的类建立子类的能力，由此建立一个类格，能支持所有这三个方面的语言被称为面向对象的语言。这个模型提供了讨论各种语言是否支持面向对象计算的框架，例如，Ada 是基于对象的语言，Clu 是基于类的，而 Simula 和 Smalltalk 是真正的面向对象的语言。

这种分类方法较为简洁和清楚，并提供了用于讨论在面向对象语言中所进行的大多数工作的框架，但是，当将这种分类方法用于近年来在面向对象计算中的开发工作时，就显得有一定的局限性，特别是用于研究静态类型化的语言时，例如，无法解释在面向对象的计算中，类属和类型适应的作用。

这种分类方法的主要限制是，它是基于面向对象的语言应该支持的一个特定的机制集合上，那些采用其它机制的语言就无法被认为是面向对象的。

2.2 对象

在基于对象的系统中，计算由一系列的实体（即对象）来体现，这些实体相互作用以达到预期的结果。本节首先讨论对象的概念，然后讨论对象交互的方式。

2.2.1 什么是对象

在第一章中已提到，对术语面向对象存在着许多不同的解释，但是，在这一领域中的每个人都认识到对象是计算的基础构造块。但给出对象含义的一个精确的解释却很困难，回答可能是每一个事物是一个对象。在面向对象的计算中，这在某种程度上是正确的。然而，这对理解对象的本质很少有帮助。问题最好从两个不同的角度来回答：什么是概念级（用户的观点）上的对象和在实际系统中怎样实现一个对象（实现者的观点）。

在概念级上，对象是正开发的系统中任何被观察到的实体。在构造一个系统中，典型地，程序员将分析问题域，因此将对解决这个问题所需要的组件有一个觉察。在一个基于对象的系统中，这些组件被直接由对象来表示。我们经常说，对象表示问题域中的真实世界实体，这基本上是正确的。对象代表正被建模的系统中的实际实体。但是，经常有必要引入一些支持正被开发的环境的对象。为说明这点，考虑实现一个邮件系统，使用面向对象的方法，系统将被分解为几个真实世界对象，象顾客、信件、信箱和收信人等，但可能还必须有文本数据库和链接表等支持对象，这些对象对实现系统的完整功能是必须的。因此，对象的关键特性是它们匹配问题求解者的域。一个对象的抽象概念是支持应用程序的开发者所需要的功能。相比较而言，以前的方法（由 Pascal 和 FORTRAN 这样的命令式语言典型支持的方法）提供了与基础硬件结构非常密切的一组抽象，因此需要更多的工作以生成应用程序。

从物理实现形式上说，对象直接映射到封装的概念。更具体说，一个对象是一个状态和一系列可被外部调用的操作或方法的一个封装体。对象具有状态，状态记忆方法的后果。封装的概念如图 2-2 所示。

方法定义了一系列的（计算）步骤，当请求一个对象执行一个方法时，我们称对这个对象施加了一个操作，所以，操作是将一个方法作为整体对待，将其作为一个单独的方法请求的过程。

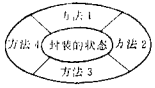

图 2-2 封装的对象

方法只允许在对象的语境内执行。它们是一些访问过程（对象只能通过这些过程进行访问）。状态给出在任一特定的时刻对象的态势，它由对象的数据结构的内容和值定义。外部接口由对象上所需要进行的操作这样的信息组成，除此之外，没有其它东西。

我们以一个饮料机为例来说明封装的对象的概念。一台饮料机可以被认为是封装的，因为它的功能被密封在一个金属盒子内。它有两个方法：

- 倒一杯茶水；

· 制一杯咖啡。

这个对象的状态由剩余的茶叶量、咖啡量、牛奶量、糖量和水量给出。与这个对象的接口由盒子前分别标以"咖啡"和"茶"的两个按钮提供,这两个按钮使用户能够执行这个对象的方法。

与一个对象的操作接口被限制在只是用户需要的上面,方法的实现外部是不可见的,也就是说,我们具有了信息隐藏。正如前面所提到的,这是封装的主要目的,它阻止非法的访问,因为金属盒阻止这台机器的用户(当然这个用户从未研究过饮料机复杂难懂的机制)改动这台机器。操作接口的另一个很重要的方面是,它提供了一个对象的行为的视图,即所知道的是一个对象提供了某种功能,但除此之外,对其它细节一无所知。在处理某个问题中的复杂性时,这是很重要的,因为一旦实现了一个对象,了解这个对象的算法和数据结构的内部细节不再是重要的,所需要的是知道这个对象提供的操作接口。

2.2.2 对象交互

有了对象,就有必要提供一种机制以允许一个对象与另一个对象交互。例如,在上面的饮料机的例子中,饮料机可能必须与一个供水系统交互,以便得到新鲜的水。在面向对象的计算中,这种交互应该被严格控制,即交互应该被限制在严格定义的方法接口上。这样,对象交互等价于调用与其它对象相关联的方法,而这种调用在面向对象中经常被刻画为通过消息传递来完成。每当一个操作被调用,一条消息被发送到这个对象上,消息带有将被执行的这个操作的详细内容。然而应该强调的是,消息传递经常纯粹是一个概念上的东西,并非一个实际的消息被发送到一个对象上,最有可能的是,方法调用被构造成过程调用的一种受控形式。然而,将一个面向对象的系统想象成为一系列自包含的对象只通过消息传递交互是很有吸引力的。

消息传递的语法随系统不同而不同,一般具有下面这些组成部分:

- 接受者的名字,这个接受者又被称为目标对象;
- 所请求的方法;
- 一个或多个参数。

这些信息使请求者对象向接受者对象发送一条消息,要求调用特定的方法。所调用的方法反过来又可能生成更多的消息,因此导致调用在其它对象中的方法。如果一个对象不知道如何响应一个请求,即该对象没有定义相应的方法,则系统给出异常错误信息,或编译器给出有关的类型错误信息。

2.3 类

类通过在系统中引入分类使系统更前进了一步。完成分类所必须的机制是类和对象实例化。

2.3.1 什么是类

支持类的目的是提供一种基本的分类形式。类通过允许建立对象的组支持分类,这组对象共享完全相同的行为。这是通过为系统中要建立的对象提供相应的样板(即类)来做到的。类的精确定义是:类是创建对象的样板,它包含所创建的对象的状态描述和方法的定义。

因此,类样板以外部接口和内部算法以及数据结构的形式提供了一个类的完整描述。这种方法的主要优点是,实现一个类的努力只需进行一次。

可以使用一个点心切割器为例来对类作一个模拟说明。一种点心切割器以尺寸和形状定义了一种点心的精确特征。例如，一种特定的点心切割器可能定义一个直径为3英寸的圆形形状。由这个特定的切割器制造的所有点心将被确保精确地具有这种尺寸和形状。

回到饮料机的例子，我们可以定义这样的一个类：

类名：	DrinksMachine
状态描述：	牛奶量，糖量，水量，茶叶量，咖啡量
方法：	制一杯茶的方法的定义
	制一杯咖啡的方法的定义

2.3.2 对象实例化

一旦引入了类，就有必要提供一种机制来建立一个特定类的对象。对象的建立可以通过一个实例化机制动态地进行，实例化基于样板去建立一个新的对象，这个对象在建立时被初始化，它存在于系统中，并将被其它对象访问。实例化过程使一个对象知道它所属的类。由一个特定的类建立的对象被称为这个类的实例，并且注意到，虽然一个特定的类的所有实例呈现共同的行为，但它们并不是完全同一的，它们在接口和实现上是同一的，然而，每个对象拥有它自己的状态，并且这个状态依赖于在这个对象上所进行的操作调用而随时间变化着。

下面的例子说明了一个饮料机的一个特定的实例的实例化是怎样进行的：

```
Machine_A = new DrinksMachine ;
```

使用实例化机制，也可以创建其它实例，例如：

```
Machine_B = new DrinksMachine ;
Machine_C = new DrinksMachine ;
```

对象实例化的语义将确保饮料机A、B和C都将呈现共同的行为。

2.4 继承

类提供了声明一组对象结构的机制，但是，当借助于继承这一机制扩充类定义时，才真正地实现了面向对象计算的所有益处。这节详细讨论继承的概念，同时也讨论与此有关的类层次、方法束定和多态性的概念。

2.4.1 什么是继承

只有类，所有的新类必须从零建立起，即以数据结构、算法和接口定义的所有功能必须以系统中的基本组件的形式进行描述。在许多情况下，这种方法包含不必要的劳动。因为，在系统中存在其它相似的类。因此，在定义新类时能够使用已有的类就变得很有吸引力。继承提供了精确地完成这个工作的系统化的方法。一个新类可以通过修改或扩充已存在的类以满足这个新类的要求来进行描述。新类因而共享已有类的行为，但具有修改的或额外添加的行为。这种行为共享是继承的本质特征。

从一个类继承定义的新类类将继承这个类的所有的方法和属性，它也可以根据需要添加新的方法和属性。新类被称为已有类的子类，已存在的那个类被称为新类的父类。继承可以以饮料机的例子为例进行说明。

假定要求实现一个名为 Mark2 的饮料机类，这类机器能够制作热咖啡。代替描述一个新的饮料机，明智的方法是象下面这样从已有类继承：

类名：　Mark2
父类：　DrinksMachine
状态描述：　热咖啡量
方法：　制一杯热咖啡的方法的定义

新的 Mark2 饮料机类因而具有所有来自于 DrinksMachine 的状态描述和方法，另外还具有额外的用于描述热咖啡的状态描述，因此，上面的类描述等价于：

类名：　Mark2
状态描述：　牛奶量，糖量，水量，茶叶量，咖啡量，热咖啡量
方法：　制一杯茶的方法的定义
　　制一杯咖啡的方法的定义
　　制一杯热咖啡的方法的定义

2.4.2　特化的形式

当从一个已存在的类建立一个新类时，用户实际上是在进行一个具体化过程，它向更接近应用域的需求迈进了一步，因而将这种方式称为特化。一个特化可以有多种形式：

- **添加新行为**　最常用的方法是向已存在的类中(以状态和方法的方式)添加新的行为，即新类从父类继承状态和方法，并添加有额外的功能。
- **改变行为**　第二种方法是重新定义一个已有方法的实现，例如，重新编码一个 print 方法以给出一个对象表示的一个更复杂的外观显示。

2.4.3　类层次

随着一个面向对象的系统的发展，子类从已有的类中构造，直到开发出合适的功能为止。结果形成一个类层次，节点代表类，弧代表子类关系。这种层次经常以一个特定的类为根，它只包含共用于所有其它类的一个最小的行为集。

为说明类层次，考虑饮料机的例子。经过一段时间，建立了各种的饮料机类，例如 Mark3 由原始版本生产而来，以后的 Mark4 来自于对 Mark3 的开发，结果类层次如图 2-3 所示。

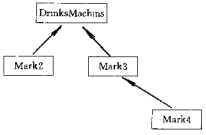

图 2-3　类层次示例

从面向对象的系统的角度来说，子类关系经常被称为 isA 关系。因此，如果类 A 是类 B 的子类，则 A 是一个(IS-A)B。这种解释的根源来自于这个事实，一个子类总可以被用于替代它的父类，因为子类至少具有父类的行为(当然它可能有额外的行为)。因此，子类在效果上是父