第 1 章 概述

这一章，我们重点概述数据结构中一些基本概念和基本方法，是以后各章的重要基础。

§1.1 数据结构的兴起与发展

数据结构问题起源于程序设计的发展。程序设计现在已经历了三个阶段：

- 无结构阶段
- 结构化程序设计阶段
- 面向对象阶段

第一阶段是无结构阶段，从四十年代起至六十年代，那时的程序设计是主要针对科学计算的，所涉及的数据对象比较单纯，程序以算法为中心，程序的设计技术以机器语言/汇编语言的机制为基础。**第二阶段**是结构化程序设计阶段，从六十年代末开始，约到八十年代。这时期，人们已认识到了程序设计的规范化的重要性，提出程序结构模块化，程序应适当限制类似于机器语言中的跳转指令（goto 语句）的使用，程序模块的内部以顺序、if-then 分枝和 while 循环为主。与此同时，计算机已开始广泛应用于非数值处理领域，操作系统、编译程序、数据库等系统软件的设计已进入方法化时期，对象/实体的数据表示法成为程序设计的重要问题，人们注意到了数据表示与操作的结构化，程序中常用的一些数据表示，如表(线性表、链表、广义表等)、栈、队、树、图等被单独抽出研究。数据结构及抽象数据类型就是在这种情况下形成的，其主要标志是 1968 年 D. E. 克努特（Knuth）的巨著“The art of computer programming”的第一卷的发表。该书首次系统地阐述了线性表（含链表）、树、图等常用的数据结构的存储与基本操作的设置与实现，可以认为是数据结构的鼻祖。同时，美国的一些大学开始开设数据结构课程。

数据结构概念的引入，对程序设计的规范化起了重大作用，使程序设计这个艺术性很强的工作，也有了一定的规范与方法可循，开始转入方法体系。这主要表现在这几个方面：程序设计与数据表示的系统化归纳、抽象数据类型观点的引入、以数据为重点的分析法等。

但是，真正使程序设计技术发生重大变化的是面向对象（Object Oriented）技术（**第三阶段**）。面向对象技术（首先是面向对象程序设计）约于八十年代初或更早些兴起，但真正流行是九十年代的事情。对象是描述实体的属性与操作的，是这二者的封装体。在

面向对象技术中，数据是程序的“主人”，对象是划分与构造程序的主要单位，这与以前的以功能为中心的观点与方法有很大不同。面向对象技术实质上是数据结构概念的自然扩展与继续，对象包含数据结构中的主要因素：数据成份与操作。

面向对象技术抓住了程序设计中的要害，可以说是程序设计问题的自然的解决方法。许多复杂问题，用对象的观点去处理就会“纲举目张”，获得自然的解决，所以，面向对象技术将是程序设计方法中的主流技术。作为程序设计与软件技术的基础的数据结构，它的方向与内容等自然应随着面向对象技术作相应的调整，这样才能保证它仍是程序设计的基础。

目前，“数据结构”是各类计算机专业的专业基础课，并且是主干课程，同时也是一些计算机软件相关专业的重要课程。

§1.2 数据结构的研究对象

要明白数据结构的概念，可以先说明数据结构研究/涉及的对象/问题。首先，让我们考查一下利用计算机系统的实质。任何一个可完成某些具体功能的计算机系统（软件是主体），都可以看作一个处理器（或输入输出器，或现实世界中的系统的一个模拟），对问题领域内的对象/实体进行处理或访问。因此，计算机应用系统中有两个关键问题：

- 表示：对象/实体及其关系在计算机中的表示。只有对象及其相互关系已存储（表示）在计算机中，才能被进一步处理；
- 操作：对对象/实体进行处理、访问。

下面举例说明。

[例] 解一元二次方程 $ax^2+bx+c=0$.

利用计算机解此方程，第一个问题就是如何在计算机中表示该方程。分析该方程，可知决定方程的是方程的三个系数值：a、b、c（在 $a\neq0$ 情况下，实际上只决定于一次项系数和常数项），而它们的次序表示它们分别属于那一项，其他符号是为增加可读性而引入的，因此，可用这三个系数的线性排列在计算机中表示该方程。例如，

$3x^2-x+1=0$ 表示为（3, -1, 1）

$x^2-3=0$ 表示为(1, 0, -3)

在数据结构中，将若干个线性排列的数（元素）称为线性表，因此，一元二次方程 $ax^2+bx+c=0$ 在计算机中表示为线性表（a, b, c）。解方程实质上是对线性表(a, b, c)进行操作。

[例] 计算机管理家谱。

家谱管理主要实现家庭成员的登记、查询及变更处理等。为突出主题，我们这里假

定只考虑家庭中的父子关系。在这个问题中，实体对象是人（家庭成员），关系是父子关系。每个实体用一个记录（元素）表示，包含姓名、出生日期、性别、死亡日期等。为了表示父子关系，在实体记录中可增加若干字段，每个字段用于指示一个儿子/女儿，这样，一个家族就构成了一个层次结构。在数据结构中，该层次结构称为树。图 1-0 给出了一个具体的例子，其中，位于某结点下方的与其相连的各个结点，表示该结点的子女。

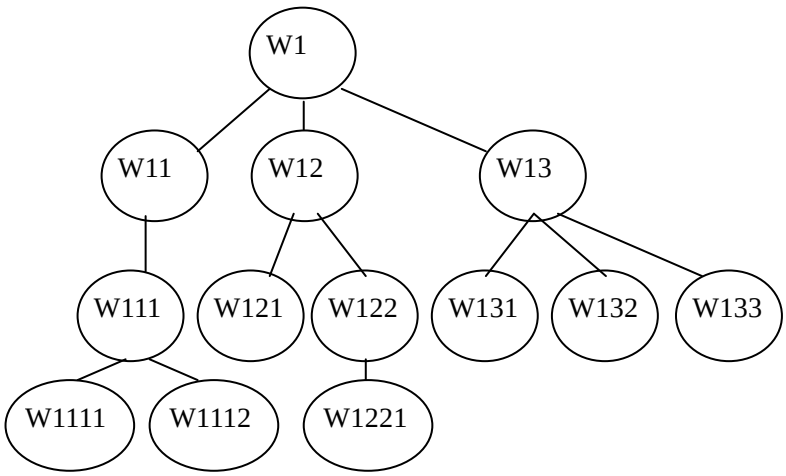


图 1-1 一个家族结构的树表示

这样表示后，对家族的所有操作，如查找成员（按姓名、生卒时间、辈分等），查找前辈/后代，增加、删除及修改成员、计算平均寿命，等等，都可统一在这棵树上进行。

除了确定表示方式外，数据结构的任务还有对这棵树的计算机物理存储、操作的抽象化。其中，操作的抽象化旨在建立存取/访问树结构的基本计算机程序，以支持其他各种操作的实现。

归纳起来，数据结构的研究内容为：

为了在计算机上实现具体问题，所需的表示数据/信息及其关系应如何组织（组织起来的数据就具有了结构关系），以及如何对它们进行基本操作。简言之，研究数据的组织方式（结构）及相应的抽象操作。

§1.3 数据结构的概念

上面我们从“外部”回答了什么是数据结构的问题，说明了数据结构在计算机程序中的作用。下面，我们从另一角度阐述这一问题，并给出几个基本概念。

（1）数据：数据是描述客观事物的**信息**的符号化，是计算机系统可加工处理的对

象。数据是个广义的概念，可以指普通的数据（可参加算术运算），也可以指符号（源程序、产品名称等）或符号化了的声音、图形、图象等。

（2）数据类型：现实世界中的数据是多种多样的，不同的用途，需要不同种类的数据，而不同种类的数据，对应不同种类的使用方法（操作）。因此，有必要对数据进行分类。在计算机中，以操作种类划分数据类型，即数据类型定义为：

一个值的集合和定义在这个值集上的一组操作的总称。

数据类型主要表明了数据的取值范围和操作特性，是具有相同取值范围和可施于同样种类操作的数据集合的总称。例如，C 中的无符号字符型（unsigned char）代表闭区间[0,255]中的整数，可作用于这种数的操作有加、减、乘、除、 取模等算术运算。如果在特定的环境下，某种数据类型不能由其它数据类型复合而成，则称这种数据类型为原子类型，否则，称为复合类型（或导出类型或结构类型）。

（3） 数据元素、数据项：能独立、完整地描述问题世界中的实体的最小数据单位称为**数据元素**（也称**记录**）。构成数据元素的不可分割的数据单位，称为**数据项**。例如，对于学生花名册，其中每个学生记录就是一个数据元素，而学生的姓名、年龄等项目为数据项。数据元素是我们以后讨论数据结构时涉及的最小数据单位，它中的数据项一般不予考虑。

（4）数据对象：同类数据元素的集合称为数据对象。例如，所有学生记录的集合，就是该问题世界的一个数据对象。

有了上面几个概念，我们就可以给出数据结构的概念了。

（5）数据结构：问题世界是由实体构成的，数据元素是用来表征实体的。问题世界中的实体一般不是孤立的，而是存在着一定的关系，亦即数据元素之间存在着一定的关系。我们把数据元素之间的这类关系称为**结构**。进一步地，我们称相互之间存在着一定关系的数据元素的集合及定义在其上的基本操作（运算）为**数据结构**。为了与后面要介绍的存储结构区别，有时也强调地称数据结构为数据的**逻辑结构**。

我们在下面将看到，之所以将彼此间存在一定关系的数据元素与定义在其上的基本操作总称为数据的逻辑结构，是因为基本操作刻画了数据元素之间的关系，反映了它们的性质、功能。

如果不考虑定义在数据结构上的操作，则数据结构也可借助集合论述语定义为：

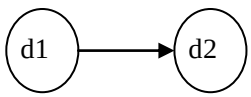
数据结构是一个二元组（D,S），其中 D 是数据元素的有限集，S 是 D 上的关系的有限集。

在这个定义中，数据元素之间的关系采用集合论中关系的形式化描述方法来定义。型为<d1, d2>的二元关系中，我们称 d1 为关系的**前件**，d2 为**后件**。称 d2 为 d1 的**后继**，而 d1 为 d2 的**前驱**。

采用形式化方法表示关系的优点是描述严密，易于使用数学方法研究。但在某些情况下，这种方法显得很繁。

§1.4 数据结构的图示

有时，为了讨论方便，用示意图表示数据结构（有时为了与存贮结构图区别，称这里的示意图为数据逻辑结构图）。具体表示法为，用**小圆圈**代表数据元素，用小圆圈之间的**连线**代表小圆圈对应的数据元素之间的关系，如果强调关系的方向性，可用**带箭头的线段**表示关系。具体地讲，若 d1 和 d2 表示两个数据元素，它们具有关系＜d1,d2＞，则表示为



这只是一个抽象关系（关系模式），不代表具体意义。现实中的父子关系就属于这种关系模式，例如，<d1, d2>可代表 d1 是 d2 的父亲。

§1.5 数据结构的分类

根据数据元素之间的关系的不同，将数据的逻辑结构分为集合、线性结构、树、图等。

§1.5.1 集合

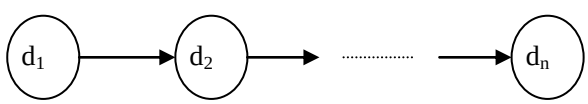
如果数据结构中，数据元素之间不考虑关系问题（无前驱/后继之分），则称这种结构为集合。

在集合中，各元素是“平等”的，它们的共同关系是：都属于同一个集合。

§1.5.2 线性结构

如果数据结构中，数据元素之间只存在前后顺序关系（每个元素都有唯一前趋和后继，第一个元素可以没有前驱，最后一个可以没有后继），则称这种结构为线性结构。

线性结构是一种最常见的数据结构。线性表、栈、队列、串等均为线性结构。



上图表示的数据结构可表示为：

DS=(D, S)

$D=\{d_1, d_2, \ldots, d_n\}$

$S=\{ r \}$

$r=\{<d_1, d_2>, <d_2, d_3> , \ldots, <d_{n-1}, d_n>\}$

§1.5.3 树形结构

如果除一个特殊元素没有前驱外，其他每个元素都有唯一的前驱（后继个数不限），则称该结构为树型结构（简称树）。其中，将无前驱的元素称为树根。

用图表示树时，通常习惯将树根画在最上面。某元素的各后继画在该元素的下面，且连线不带箭头，隐含着从上到下。这样，树型结构就象用一棵倒立的树。

图 1-0 就是一个树的例子，它代表的结构的形式描述为：

DS= (D, S)

D = {W1, W11, W12, W13, W111, W121, W122,
W131, W132, W133, W1111, W1112, W1221}

S = {r}

r = {<W1, W11>, <W1, W12>, <W1,W13>,
<W11, W111>, <W12, W121>, <W12, W122>,
<W13, W131>, <W13, W132,>, <W13, W133>,
<W111, W1111>, <W111, W1112>, <W122, W1221> }

树形结构中的数据元素（结点）可分为三种，一种称为根，每个树结构只有一个根（空树无根），根无前趋，但可有若干个后继；一种为叶子，叶子无后继，但有且仅有一个前趋，其余为普通结点，有且必有一个前趋，后继有若干个。

树型结构常用来表达层次关系，这是它的一种自然应用，另一个重要应用是快速检索：为了提高数据检索速度，可将数据按树结构组织，如二叉排序树、平衡二叉排序树、B 树等。

§1.5.4 图状结构

在图状结构中，任一数据元素，均可有多个前趋和多个后继。该种结构也称网状结构。图状结构表达能力最强，它可表达任意复杂的数据结构。

例如，交通图就是一种图状结构，结点代表城市，连线（关系）代表城市间的道路。

树形结构与图状结构均称为非线性结构。

还有一些数据结构，如多维数组和广义表，尽管本质上属于网状结构，但由于具有很多特点，与网状结构的处理方法有很大不同，所以常单独讨论。

§1.6 数据结构的存储----存储结构

§1.6.1 存贮器表示问题

归根结底，数据要存贮在计算机的存贮器中的。当代计算机的存贮器，从使用方式来讲，分内部存储器（简称内存，也称主存）和外部存储器（简称外存，也称辅存）。内存由有限多个存贮单元构成,每个存贮单元为一个基本的存取单位。存贮单元用地址(码）标识，每个存贮单元都有一个唯一的地址，访问存贮单元时，必须指明要访问的存贮单元的地址。存贮单元是最小的访问单位。各存贮单元是连续编址的。若干地址连续的存贮单元称为一个存贮区。内存数据是通过 CPU 指令按存储单元的地址直接被操作的。

外存是内存的辅助，CPU 指令一般不直接控制外存，而将其作为外部设备访问，外存的数据一般是通过外设访问指令装入内存后接受 CPU 的操作。

本课程中，数据结构的存储问题，除最后一章外，都针对内存。最后一章将集中讨论外存的数据结构存储。

对内存的使用，最直接的方式是通过具体机器的机器指令（语言），但我们不采用这种方式，原因主要有两种：一是实际的计算机应用，大多不直接使用机器语言，而主要使用高级程序设计语言（如 C, C++, Java, Pascal, Basic 等）；二是机器语言相当繁琐，且可移植性差，会使我们陷入到一些没必要的细节中。

大多数高级语言支持的访问存贮器的方式是相当高级的，隐蔽了存贮器的许多特性，不利于表示数据的存贮结构。不过，许多高级语言都提供按数组访问存贮器的方式。数组很接近存贮器，所以我们决定**用高级语言中的数组模拟计算机存贮器**。另外，C/C++中的指针的概念相当接近内存的地址，所以，使用 C/C++，可用简单的方式，近乎得到机器指令的效果。

§1.6.2 存贮映象(存储结构)问题

数据结构的存贮映象，是指数据结构在计算机中的**存储方式**/方法，是数据结构的另一种表示方式，称为数据结构的**存储结构/方式**。将一个逻辑上的数据结构存储在计算机中，必需满足下列两点：

内容存储：数据结构中的各数据元素的内容（数据），都分别存贮在一个独立的可访问的存贮区中;

关系存储：数据元素的存放方式方法，必须能显示地或隐式地体现数据元素间的逻辑关系。

这两点是存贮映象所应有的必要条件。

在满足上述必要条件的基础上，存贮映象还应考虑存贮使用效率（空间复杂度）及数据结构的操作的实现的方便性等。

基本的存贮结构有顺序方法、链接方法、散列方法、索引方法。实际的存贮映象方法，也可能是这些基本方法的复杂组合。

§1.6.3 基本存储方法

(一) 顺序方法

这是一种主要面向线性关系的存贮方法。对线性数据结构，可将其数据元素，按相应的线性关系下的前后次序，存贮在物理存贮器中，使得数据元素在此线性关系下的逻辑次序与它们在存贮器中的存放次序一致。这种存贮方式称为**顺序方法（也称连续方法）**。

由于这种存贮方式下数据元素的逻辑次序与物理存贮次序一致，所以数据元素间的线性关系由它们的存贮次序体现，而不需要专门存储。

这种方法主要面向线性结构，但不局限于此。若某结构中存在一种线性关系，而通过此线性关系就可以确定元素关系，就可使用这种方法。如多维数组、顺序二叉树等结构的存贮。

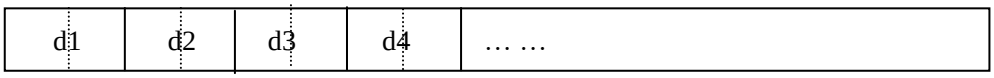
为了能使存贮次序表达逻辑次序，显然，在存贮器中，任意相邻两数据元素之间的存贮单元数目应相等。

[例] 设有数据结构 DS=（D,S），其中，

D={d1, d2, d3, d4}，S={r}，

r={<d1,d2>，<d2,d3>，<d3,d4>}

D 中每个元素需占用两个存贮单元，则该结构的顺序存贮结构如下图所示（称存贮结构图）.



(二) 链式存储

每个数据元素的存储区分两大部分，第一部分为数据区，存贮元素的内容，第二部分为指针区，存放该数据元素与其它数据元素之间的关系信息，这种关系信息一般为地址（与此数据元素相关的其他数据元素的存贮地址）。对于线性结构，指针区中可以只设一个地址；对非线性关系，可能需多个地址。另外需指出的是，数据元素的存贮区之间，可以是连续的，也可不连续。

链式方法具有很大的灵活性，可适合于大多数的数据结构。与顺序方法相比，它的存储空间开销较大（增加了指针区），但由于各数据元素的存贮区不要求是连续排列的，故对内存空闲区分布的要求不高，很适合动态存贮管理。

[例] 对上例中的数据结构，它的一种链式存储结构如表 1-1 所示。

在该表中，假定每个元素占 2 个存储单元，第一个存储元素内容，第二个存储关系。这里，关系用后继地址描述，元素 x 的“关系指示”中存放 x 的后继的地址。

这里，为数据元素所分配的存贮区的位置与数据结构本身无关，而与具体的存贮管理方法有关。

表 1-1

地址	数据	关系指示
a + 0		
2	d3	8
4	d1	10
6		
8	d4	0
10	d2	2
…	…	…

[例] 设数据结构为 DS=（D,S），其中

D={d1, d2, d3, d4, d5, d6}，

S={r1, r2}，

r1={<d1,d3>，<d3,d4>, <d2, d6>}

r2={<d1,d2>，<d3,d5>}

它的一种链式存贮映象如表 1-2 所示。

表 1-2

地址	数据	关系 1	关系 2
a+ 0			
4	d2	28	
8			
12	d1	24	4
16	d5		
20			
24	d3	32	16
28	d6		
32	d4		
…	…	…	…

在表 1-2 中，假定每个元素占 4 个存储单元，第一和第二个单元存储元素内容，第三和第四个单元分别存储两个关系 r1 和 r2。这里，关系用后继地址描述，元素 x 的“关系 1”中存放 x 的第一个后继的地址，“关系 2”中存放第二个后继的地址。

在数据结构中，往往不考虑具体的地址值，而只关心它们所代表的关系，所以在画

示意图（存贮结构图）时，用带箭头的线表示地址，这也是指针的由来。

链式存储适用范围很大，理论上讲，可适用于任何数据结构的存储。与顺序结构相比，空间利用率低（因为显式地存储了关系）。

(三) 索引存储

索引存储主要针对集合和线性表，面向检索（查找）操作。它主要是在数据结构的存储区（称数据区）外，增加一个（或若干个）索引区。

索引区本身也是个线性表或其他数据结构，结构中每个元素用于记录数据区中一个（对稠密索引）或一组（对非稠密索引）元素的存储位置（起始位置）。

索引存储并不强调对关系的存储，而主要针对数据内容，所以，一般只适合集合结构或线性结构。

(四) 散列存储

散列存储（也称杂凑法）是一种按元素内容存储元素的方法。其基本思想是：

- 设置一个函数，称为散列函数：元素内容→地址；规定元素内容到存储地址的映射；
- 存储时，通过散列函数求出元素的应存储的地址，按此地址存储；
- 读取时，通过散列函数求出元素的存储地址，按此地址读取；

与索引存储类似，散列存储也是面向内容存储，不适合存储复杂数据结构。

§1.7 数据结构访问接口

§1.7.1 访问接口与逻辑结构

数据的访问（也称操作）是指对数据的读、修改、加工、处理等操作。数据存在的目的是进行操作。操作的种类很多，随不同的应用而不同。数据结构中的一个重要的问题是：

对每种数据结构，如何设置一些操作，使得各种应用都能通过这些操作就能实现对数据结构的各种操作，我们把这类操作称为数据结构的**基本操作**或**运算**。

操作的调用形式与规范，称为该操作的**接口**；将针对某一数据结构的基本操作的接口的全体，称为该数据结构的**访问接口**；

基本操作有下列关键点：

- 抽象性：不涉及具体应用，是关于“结构”（即语法）的操作，不带应用语义；
- 基本性：“粒度”很小，作为构造更复杂操作的基础；
- 完备性：只通过基本操作就能进一步实现各种操作；
- 支撑性：要有灵活方便的功能，足以供其他应用进一步构造其他高级操作。

数据结构的基本操作是数据结构的逻辑特性的体现，是对数据结构的抽象。基本操作定义/体现了数据元素间的关系及它们对外呈现的性质和功能，所以，一般称数据元素间的关系和定义在数据元素集合上的基本操作为数据的**逻辑结构**。由基本操作定义应用程序通过它访问数据结构，它完全屏蔽了存储结构的细节，使得应用程序只需了解接口，就能使用数据结构。

§1.7.2 基本操作的种类

抽象数据类型实质上是指数据结构及定义在其上的一组操作，操作与数据结构紧密相连，每种数据（逻辑）结构都有一个操作集合。一个数据结构应设立哪些操作的问题，与数据结构本身及数据结构的应用目标有关。操作的实现与数据结构的存贮映射方法相关。同一种数据结构的同一操作，对不同的存贮映射，实现方法也不同。

尽管不同的数据结构对应不同的基本操作集合，但可按功能归纳为下列几种基本类型：

- 属性读取(Get)：读取数据结构的各基本属性（数据项）的值；
- 属性设置(Set)：给数据结构的基本属性（数据项）赋值；
- 查找：在数据结构中寻找满足一定条件的数据元素（位置或值）；
- 插入：在数据结构的指定位置上添加新的数据元素；
- 删除：删去数据结构中某个指定元素；
- 关系访问：访问数据结构中有特定关系的元素。例如，在树中求出某结点的第 n 代后代；
- 遍历：按某种方式访问数据结构中各元素，使得每个元素恰好被访问一次。

通过 Get 和 Set 类的操作，实现了数据结构的名称级的抽象，使具体的数据项名称与其他应用程序独立。通过关系访问，对应用系统隐蔽了数据结构中的关系。

§1.7.3 基本操作的实现

操作的实体是计算机程序，因此，基本操作的实现，是个针对相应的数据结构编程的问题。由于程序是算法的实现，所以可以讲操作实现也是算法实现问题。

§1.8 面向对象方法

由于我们要将面向对象的概念与方法引入数据结构，所以本节先简单介绍面向对象的概念。

§1.8.1 对象与类

在面向对象方法中，将问题世界中所涉及的实体抽象为**对象（Objects）**，每个对象，是对应的实体的一个抽象模型，它刻画实体的状态和行为——状态称为对象的**属性**或**数据成员**，行为称为对象的**方法**或**操作**或**服务**，亦即对象由描述实体的状态的属性与用于改变自身状态的操作两大部分构成：

对象 = 属性 + 操作

类是对象的型，它与数据类型的概念是类似的——定义了对象的属性的取值范围与一组操作。每个类都是一批属性与结构类似且操作相同的对象的抽象。由于类中含数据与操作两部分，所以它既可看作类型，又可看作模块。继承（下面即将介绍）的引入，会使这点更加明确。

§1.8.2 面向对象方法要素

面向对象方法以对象为中心分析解决问题，以数据封装、继承、多态性、消息传递为要素。面象对象程序设计语言既支持归纳方法，又支持演绎方法，是一种很好的认知方法。

(一) 封装（Encapsulation）

封装是指将数据与相应的操作作为一个整体看待。操作主要针对相应的数据，用于改变对象的状态。操作一般只改变相应的数据，不改变封装体（对象）外部的数据，使用者使用封装体时，只需知道访问操作，而无需顾及内部实现方法。

乍看起来，我们这里讲的封装与具有数据隐蔽性的功能模块一样，其实不然。首先，功能模块（子程序）可能具有人为的副作用（即人为地设计成可以改变入口参数），但对象无此问题。其次，功能模块的入口参数密切依赖于功能模块的实现方法。例如，一个检索模块，入口参数至少应指出在什么地方检索什么东西，调用检索模块时，调用者要按模块实现的要求定义这两种入口参数。如果是在线性链表中检索，则要按线性链表的要求定义入口参数；如果是在树结构中检索，则要按树结构要求去定义。面向对象的封装体则与此不同，调用者只需定义相应的对象即可调用检索功能，并不需要知道对象内部的细节。当然，这种特性常常需多态性的支持。更直接更高级的支持是模板（template）概念。

(二) 继承（Inheritance）

在客观世界中，就内部联系方面，实体之间存在着两种关系：整体/部分关系、一般/特殊关系。继承主要是为支持后者而引入的。

这里的继承是指对象/类之间的继承。对两个对象/类 A 和 B ，A 的属性是 B 的属性的子集，A 的操作在名称与调用界面（与实现方法无关）方面是 B 的操作的子集，则称 B 通过**继承** A 而来，或曰 B 由 A **派生**而来（B 是 A 的派生物）。这说明，B 可以定义新的属性和操作，B 共享 A 的所有属性，B 可以重定义 A 中的操作（不改变名称与接口），如果没有重定义，B 可以共享 A 中未被重定义的操作。

继承是个全新的概念，一般的高级语言中没有它的痕迹。继承支持一般到特殊（简单到复杂）的构造方法。有了继承机制，就可以在已有对象类的基础上，定义新的对象类，而不必从头做起。这也是一种软件复用机制，不过，更重要的是，它支持描述现实世界中大量存在的一般/特殊关系。

[例] 考查出版物管理系统中的出版物。出版物可分为两大类：书和期刊。它们有共同属性：编号、书刊名、出版日期、出版社（编辑部）、页数等，不同的属性是，书具有作者名，期刊具有卷号与期号。据此，我们可以定义一个公共对象类“出版物类”，用以描述书与期刊的公共属性，实现这些公共操作，而书和期刊分别从它派生。

(三) 多态性（Polymorphism）

多态性译于 Polymorphism 一词，这里 Poly 是许多的意思，morphus 是采用某种格式的意思，这二者合起来的意思是：可采用多种形式的能力。在面向对象方法中，它的意思是：一个名字，多种语义；或相同界面，多种实现。

多态性广泛存在于现实世界中。让我们先看看汽车的刹车器。按内部原理，刹车器有多种多样，有机械传动的、汽压传动的、电信号传动的，有鼓式的、盘式的，但它们有着相同作用、相同的使用方法，这就是一种多态性。

再看看高级程序设计语言中的算术运算，比如说加法，一般都用符号“＋”代表加法操作，它可以是两个整数相加，也可以是两个浮点数相加，也可以是混合相加（有的语言中，甚至用“＋”表示字符串连接），这不同类型加法，内部实现有着很大的不同，但我们以相同的方式使用，具体的不同的实现是由编译器自动完成的。这表明，编译器提供了这种多态性。试设想，若没有这种多态性，即使是小小的加法操作的使用，也会给我们带来许多麻烦。

上面指出的加法操作的多态性，是针对高级语言预定的数据类型的，能否使程序员自定义的（抽象）数据类型对应的操作也具有多态性呢？如能这样，用户自定义类型也会象标准类型一样容易使用。这就是面向对象方法中的多态性问题。

多态性是更高级别的信息隐蔽，它不仅隐蔽内部数据与内部实现，而且隐蔽入口数据的细节，这实质可以看作是一种操作调用的模糊化（我们借用了模糊数学中的“模糊”的概念，而没有使用“透明”这个词）。

在 C++中，多态性通过“覆盖”（重定义）基类中的虚函数实现的。

(四) 消息传递

对任意一个问题世界（系统），都是以系统中的实体间的相互作用为存在与运行标志，即系统中的实体及实体间的相互作用体现着系统的功能。在面向对象方法中，对象确切地代表了实体，实体间的相互作用，就是对象间的相互作用。又对象是描述实体状态（属性）与操作的封装体，所以系统实质上是一个以对象为状态的状态自动机，自动机中状态的变化，就是系统的运行。驱动状态的转化就称为消息（或称事件，有时区分事件与消息，将事件看作是消息序列（也称其为脚本）的总称），即对象状态的改变是消息传递的结果。这种情况也称事件驱动。注意这里使用的动词“驱动”与“传递”的含意，要强调的是，它们并不含“实现”的含意，仅仅含转达、告知等意义。要改变某对象的状态时，先要向该对象发送消息，告知对象要“做什么”样的改变，而具体状态改变的实施，是由对象自身的操作根据收到的消息进行的，即对象的操作负责“怎么做”的问题。

这种消息传递机制是与现实世界的运行相吻合的。这里用一个企业职工工资管理的例子说明。给某职工增加工资时，由总经理做出决定，秘书将决定通知给劳资部，劳资部接通知后，负责有关登录事宜，然后向财务部发出通知，财务部接通知后具体实施。从这里看出，“增加工资”这个事务，是通过向若干有关部门（对象）传递消息实现的。面向对象中的这种机制，正好顺应了现实世界的自然形态。

乍看起来，消息传递与功能模块调用很相似，其实不然，它们有着本质的区别。这可以从以下几个方面说明。

消息传递至少要指出消息的接收者，但模块调用可带也可不带参数。

消息操作的名称类似于模块名称，但模块名代表一段可执行代码，入口参数确定了，则模块运行的功能也就确定了。消息传递则不同，消息产生的作用，不仅取决于消息的内容是什么，还要取决于接收消息的对象是谁，当前状态为何，这实质上是一种动态连接，它比模块的静态连接有更大的灵活性。

模块调用是过程式的，与功能的“如何做”密切相关，调用者需对模块有较多的了解。消息传递是说明式的，消息只需表明要“做什么”，具体如何做的问题，由接收消息的对象自行确定。

在 C++中，消息传递实质上是函数调用，但由于对象与普通数据结构不同， 所以这种函数调用也与常规函数调用不同，例如，这种函数调用可具有动态性。

§1.8.3 面向对象方法的若干述评*

面向对象方法与计算机联系在一起的历史并不长，有关它的许多问题尚处于众说纷云之中。在此，我们也就这些问题作些阐述与评论。

面向对象方法是客观世界的自然体现。应该说，对象的概念决不是为了解决问题而生造出来的，而是构成客观世界的直接可见的东西。消息传递机制也与客观世界的运转

机制相吻合。所以，用面向对象方法去开发系统是对系统的自然表示，开发者与用户会有更多的共同语言。

面向对象方法支持从一般到特殊的演绎方法。现实世界中存在两大类推理方法：演绎和归纳。归纳是一种从特殊到一般的方法，先认识具体的事物，然后抽取出它们的共同特性，反过来再用所抽取出的共同特性来处理具体的事物。这种方法是程序设计方法的重要内容，如语言中的类型与循环机制就支持归纳方法。

演绎是一种从一般到特殊的方法、从粗到细的方法。从较一般较普遍的东西出发，逐步导出各个具体的东西。这种处理问题的方式，是从共性导出特性。从另一角度看，是一种共享机制或一种扩充机制。缺少演绎机制就不能从归纳结果导出具体问题。

面向对象的方法中的继承机制支持演绎方法。利用继承机制，可以在现有的对象类的基础上，扩充或修改部分内容（属性或操作），导出新的对象类。新导出的类继承了原对象类的内容并有所扩充与修改，与原对象类相比，它更具体化、特殊化。

面向对象方法是一种以数据结构为中心的自底向上增值式的开发方法。传统的结构化方法是以功能为中心的，它按功能划分系统、按功能对系统分层，所以，数据的组织是杂乱的，然而，系统功能是由系统中实体性质决定的，也就是由数据决定的，将数据放在次要地位就等于没有抓住问题的关键。方法不得力，问题解决起来自然就会变得困难。

面向对象的方法抓住了“数据”这个纲。对象是数据与操作的封装体，数据是对象的决定因素，操作服从于数据，操作的设置，是根据数据的要求而确定的。有什么样的数据结构，就有什么样的操作。

由继承可以导出新的对象类，这些导出的对象类是按数据分层的，每个对象可以视作一个功能模块，所以，这种功能模块是以数据为基准分层构造的。由于对象类的继承是个从一般到具体，从简单到复杂的传递、扩展、修改的过程，所以，相应的功能模块的构造也具有这种特点。这是对传统的自底向上方法的重要发展。

面向对象方法支持软件 IC 概念。软件 IC 概念来源于集成电路的应用。一个集成电路是由基本元件（如三极管、电阻等）构成的具有独立功能的硬件模块，它有标准的输入输出特性，可作为构造数字系统的功用构件，具有良好的通用性。这种情况大大减轻了硬件工程师的负担，提高了劳动生产率。软件生产则与这不同。程序员一次又一次地重复地构造一些基本模块（如分类、搜索、更新、存贮……），他们时时不断地在同一主题上，精心雕琢着一个个新的变种，因此，迫切期望提高软件可重用性。那么，软件的生产，也是否能采用类似的方法呢？这就是软件 IC 概念的由来。

然而，要想将软件模块也做得象集成电路模块那样通用，是件相当困难的事情。主要原因是，软件问题本身非常复杂，常常与复杂的数据结构关联，很难象数字电路系统那样用逻辑表达式描述。对于一个数字集成电路模块，不论如何复杂，其任一输出都是其输入的逻辑函数。而软件模块则不同，它的输出与输入的关系相当复杂。

尽管实现软件 IC 是件困难的事，但由于软件工程师极其羡慕硬件工程师在开发数字系统的轻松自如，所以人们还是致力于它的研究。特别是面向对象方法的出现，给这项工作带来了新的希望。

面向对象中的对象类的概念是对软件 IC 的很好的支持，这可从下面几个方面看出。

类是数据与操作的封装体，操作只改变与其封装在一起的数据，只有通过消息传递，才能与外部联系。这表明类具有很强的模块性，高度的自治性、独立性。

对象/类的调用是更高级别的，是说明性的。 调用者不需知道（或较少地需要知道）所要调用的类对调用环境的要求。原则上，调用者只需说明“做什么”。

面向对象的方法具有潜在的并发性、分布协同性。从动态观点看，面向对象的方法的最大收获是摆脱了传统计算机体系结构中顺序机制的制约。各对象由消息触发可各自独立地工作，总体效应相当于协同完成任务。各对象的工作方式，有的是顺序的，有的是并行的。对象之间是按异步方式通讯的。如果将每个对象看作是一个子任务，那么，系统的运行，就是一种任务的分布执行，多任务并行/ 并发的机制。

面向对象的这种任务分布并发/并行协同执行，可导致一种新的计算机体系结构——面向对象的体系结构。在这里，每个对象对应一个处理器。现在开始流行的分布对象（Distributed Object）技术，就是对象这种特性的体现。

从另一方面讲，对象的这种特性，可用来表达神经网络，或智能代理(Agent)。事实上，对象是一种智能体，是可胜任这些角色的。

§1.8.4 面向对象程序设计语言*

面向对象是一种方法、思想，可用在许多方面，如面向对象程序设计 OOP(Object Oriented Programming)、面向对象分析 OOA(Object Oriented Analysis)、面向对象设计 OOD（Object Oriented Design）、面向对象数据库 OODB(Object Oriented Database)等。但在计算机科学技术中，最早使用面向对象思想的是程序设计，并出现支持按面向对象设计程序的高级语言----面向对象程序设计语言。

面向对象的程序设计语言是支持面向对象方法的最基本的一级，甚至许多面向对象的概念最早产生于面向对象的程序设计语言。面向对象程序设计语言是目前面向对象领域内最重要、最成熟的成份。它萌芽于 60 年代末，发展于 80 年代，广泛应用于 90 年代，下面将最有影响的几种 OOP 语言作一简单介绍。

(一) Simula.

最早正式地将面向对象概念做为语言成份的是 Simula67，该语言由 Ole Dahl 和 Krisrten Nygaaard 等人于 1967 年设计的。正象它的名字表示的一样，它本意是用于编写模拟系统的。它基于 ALGOL60，首次引入了类、对象、 继承和共行子程序等新概念。当今流行的 OOP 语言，绝大多数源于 Simula67。

(二) Smalltalk

Smalltalk-80 是一个集成化的程序设计语言和程序设计环境。它是 Xerox 公司 PARC 研究中心所属软件概念组（Software Concepts Group ）经十多年努力取得的成果。它除

吸取 Simula-67 中的面向对象的概念外，还受了 Lisp 语言的影响：无类型设施、名字的动态结合以及运行时刻类型动态检查。

Smalltalk 同时也是一个很好的集成化开发环境。它是一个窗口系统，程序设计语言、支撑工具（编辑器、连编器、调试器）甚至操作系统本身被集成在同一虚拟地址空间中。它的用户接口设计采用一种颇具特色的方式：MVC（模型/视图/控制），它的整个环境也是面向对象的。它的环境会牵着人们按面向对象方法行事。它是迄今为止被认为是最纯的面向对象语言。

Smalltalk 还有许多贡献，如目前的高分辨率图形工作站、 图符式用户界面及个人用计算机等都直接来源于 Smalltalk-80 系统。。

不过，Smalltalk-80 只在大学圈子里著名，在工业界它还是默默无闻的。

(三) Eiffel.

Eiffel 是由 Betrand Meyer 开发的强类型 OOP 语言，Eiffel 程序由包括方法在内的类说明集合组成，它支持参数化类、多重继承、内存管理，还提供小容量的类库。从技术角度来看，它是目前最好的商用 OOP 语言。

(四) C++

C++是 AT&T 贝尔实验室的 Bjarne stroustrup 于 80 年代初设计的一种强类型语言，它是通过在 C 语言中加进一些面向对象成分构成的。所以它是一种混合型语言。 C++的面向对象成份源于 Simula 与 ALGOL68。当时开发 C++ 的目的是为了编写一些大型的系统模拟程序。

C++最初称为“带类的 C”， 1983 年正式称为 C++，此后经历了三次修订后，于 1994 年制定了 ANSI C++标准草案。

C++几乎具备了目前公认的面向对象的全部特征，除了支持基本的类、继承等概念外，还支持友元、函数与运算符重载、虚基类、虚函数、抽象类以及多重继承、模板（类属）、异常处理等较深入的概念。

目前，支持 C++的环境/工具/编译器很多，如 MS Windows 环境下的 Visual C++、Borland C++、Borland C++ Builder 等，UNIX/LINX 环境下的 GNU C++等。

由于 C++具有良好的面向对象特性，又有接近机器语言的特性（如指针），所以我们选定它为数据结构的描述语言。

(五) Java

Java 可以说是一种最年轻的高级程序设计语言，1993 诞生于 Sun 公司的一个名为 Green 的项目的开发，当时的主要目的是作为家电产品的软件开发语言。起初，他们试图使用 C++，但很快觉得 C++过于庞大，对家电这类小软件的开发而言，有许多不必要的成分，所以，他们决定建立一种新的语言，其支持面向对象编程，但必须简明，这种

语言就是 Java.

单从语言方面看，Java 几乎是 C++的精简品，它首先去掉了 C++的非面向对象成分和指针的概念，然后对 C++的面向对象成分也做了精心挑选，去掉了诸如多继承、虚基类、模板、友元、运算符重载等功能。因此，可以称 Java 为傻瓜 C++

但是，Java 定义了大量的标准类库，除包括传统的 I/O 类库外，还包括用户图形接口 AWT(Abstract Window Toolkit)、浏览器类库、网络类库、多线程类库等。这些类库大大丰富了 Java 的功能，因此，有人称 Java 是网络编程语言、多线程语言，等等。姑且不论这些说法是否确切，但也反映了 Java 这些方面的能力。值得注意的是，其他语言，如 C++，也有强大的网络类库、多线程类库等，但它们不属于 C++标准，是 C++的非标准扩充。

Java 的另一个特性是：编译生成中间代码（非目标代码，不能直接在目标计算机上执行），然后在目标计算机上设置中间代码解释器（称为 Java 虚拟机 JVM）来解释执行已编译为中间代码的 Java. 这种机制虽然不利提高 Java 的运行效率，但带来了迎合 Internet 的好处：跨平台、可移动。之所以这样，是因为 Java 中间代码是文本，所以，只要在不同的平台上设置不同的 JVM，则 Java 就好象可四处游动了。

Java 的这种可移动特性其实不是什么新的东西，早在 80 年代初，就有人为 Pascal 定义过中间代码。但是，由于当时计算机网络不象今天这么广泛而深入，所以 Pascal 的这种特性，也未引起人们的关注。

(六) C#

C#是一种比 Java 还要年轻的高级语言，是 Microsoft 公司最近提出的.NET 架构中主要编程语言。C#的语法与 Java 十分类似，也源自 C/C++家族，但它不只是对 C/C++的简单扩展，而是在其基础之上推出的一种崭新的语言，它吸取了 C/C++、Java、Modula 2、SmallTalk 等多种语言的特征。在 C#的设计过程中，C++的强大功能使得设计难于上手，不容易扩展，如果只是在 C++的基础上进行有限的扩展，并不能得到人们想要的新一代编程语言。C#的首席设计师 Anders Hejlsberg 表示，为了解决设计过程中的种种难题，C#没有只是从 C++着手并扩展它，而是以退为进，以 C++的灵魂设计出了新的方案。最终它不仅给 C++程序员带来了快速开发能力，同时并没有牺牲 C 和 C++所特有的功能和控制。

从 C 到 C++，增加了面向对象编程的概念；而从 C++到 C#，则增加了面向组件(Component)编程的概念。Anders Hejlsberg 在介绍 C#时说:“C#是 C/C++家族里第一个面向组件的语言”。这是 C#不同于 Java 与 C++的最重要的地方。

组件的最基本特征是属性、方法和事件。这些功能，在 C++和 Java 中都是间接实现的，而 C#通过属性、方法、索引器、委派、事件、操作符重载、特征、版本等，支持对组件的直接实现。

在运行方面，Java 和 C#都是基于中间语言的。C#的中间语言是 MSIL(Microsoft Intermediary Language)，其运行环境是.NET 基础框架的 CLI(Common Language

Infrastructue)，主要部分是 BCL(Basic Class Libray)和 CLR(Common Language Runtime)，内置了 XML(eXtensible Markup Language)、SOAP(Sample Object Access Protocol)、UDDI(Universal Description，Discovery and Integration)、WSDL(Web Service Description Language)等底层协议，从而可全面支持 Web 服务（Web Services）。 但在 Java 中，JVM 和 Java 字节代码并没有直接实现这些功能，而主要是通过 API 集来支持的。

C#和 Java 都对传统 C++中的艰深、晦涩的语法语义进行了简化和改良。在语法方面，两者都摈弃了 C++中函数及其参数的 const 修饰、宏代换、全局变量、全局函数等许多华而不实的地方；在继承方面，两者都采用了更易于理解和建构的单根继承和多接口实现的方式;在源代码组织方面，都支持声明与实现于一体的逻辑封装；

但在大刀阔斧地对 C++进行改革的同时，C#显得更为保守，它对很多原来 C++中很好的特性予以了保留，如基于栈分配的轻量级结构类型、枚举类型、引用、输出、数组修饰的参数传递方式等，这些在 Java 中都被很可惜地丢掉了。在基本类型和单根继承的对象之间的类型统一方面，C#提出的 box/unbox 要比 Java 的包装类显得高明，效率也更高。对 C++不安全的指针及内存分配方式，C#和 Java 都采用了托管执行环境。

“一次编程，多处执行”一直是程序设计的一个诉求，尤其是在现代 Internet 时代。在跨平台方面，Java 的支持和实现都是为人称道的，而 C#虽然在底层构造方面对移植性进行了充分的考虑，但至少目前还没有出现成熟的、经过检验的产品。C#在跨平台方面似乎更热衷于 XML Web 服务的互操作，而不是代码的跨平台移动执行。由于 C#通过其基础语言设施（CLI）对二十多种主流语言对象级的互操作支持，所以，这种通过互操作实现的跨平台，也不失为一种好的方式。

当然，“语言选择乃艺术而非技术问题”，所以，我们不论选择何种语言，只要它们是同一类的（如过程型、函数型、逻辑型等，C/C++、 C#、 Java、,Pascal、Basic 等都属于过程型），则一般的编程技术都是类似的，只是方便性不同。

§1.9 面向对象与数据结构

§1.9.1 面向对象与数据结构的关系

数据结构起初是研究表、树、图等结构的，发展到现在，人们已将它的研究对象上升到了抽象的高度。综合而言，数据结构主要强调两个方面的内容，

a) 同类数据元素间的依赖关系；

b) 针对这些依赖关系的基本操作：这些操作是充分的（完备的），依赖它们可以实现对这些具有特定关系的元素的任意访问。

这两个方面实质上是对象的雏形，萌芽着面向对象的思想。

在面向对象方法中，问题世界中相关实体被视为一个对象。用对象描述实体，对象由属性、方法、事件构成。属性用以描述实体的物质特征。是对实体状态的数字描述；

方法是定义在属性上的操作，用以改变实体状态或访问实体；事件用于定义实体与其它实体间的通信关系，它说明了实体能感知什么样的外来信息。对象与数据结构具有下列对应关系：

对象——数据结构

属性——数据元素间的关系的描述

方法——基本操作

事件——无

对象重点描述实体的状态与行为，而数据结构重点描述同类数据元素间的关系及其操作。数据结构中的元素间关系描述比起对象的属性，更为基本。属性的表达是基于数据结构中的元素间的关系描述的，数据元素及其相互关系就构成了对实体的状态的描述，由此可见，一个数据结构就是一个独立的对象（类）。

近年来，人们越来越多地用抽象数据类型观点讨论数据结构。抽象数据类型（Abstract Data Type —ADT）是指一个数学模型与定义在该模型上的一组操作。这里的数学模型实质上指用符号化操作定义的数据对象（数据集），这种数据对象具有这样的性质，不论它们内部结构如何变化，只要它们对应的符号化操作不变，都不影响对它们的使用。抽象数据类型中的数学模型的概念，对应于面向对象中的数据封装，而操作的概念对应于方法的封装。抽象数据类型的这种思想，在数据与操作封装方面，比数据结构概念更强调抽象化，同时也更贴近对象的概念。

然而，对象有更广泛的概念，它更注重不同对象间的关系（继承），同时也更强调对实体的状态的改变，这些都是数据结构与抽象数据类型的概念中所不具有的。

§1.9.2 面向对象数据结构

面向对象数据结构从下列几个方面改造传统的数据结构：

a) 将基本元素视为对象；
b) 将元素间的关系视为对象；
c) 将数据元素的集合视为对象；
d) 对“相似”数据结构应用继承
e) 用继承机制扩展基本数据结构
f) 用面向对象的形式描述数据结构。

数据结构可视为二元组（D，S），其中 D 为同类数据元素集合，S 为定义在该集合上的关系的集合。这是狭义的数据结构定义，广义的数据结构还涉及的另外两点——基本操作定义与存贮实现。

在(D,S)中，D 为元素的集合。一个元素，是问题世界中的一个实体。其实，实体具有对象特征，其内容相当于对象的属性，而属性的改变通过设立相应的方法实现。但在数据结构中，并不单独关注实体（元素）的操作，实体的操作是混杂在结构的操作中。所以，将元素视为对象更加自然、方便。

对(D, S)中的关系集合 S，数据结构中，将其视为一个整体，基本操作也是直接针对 S，因此，其本身就具有对象特征。

从面向对象观点看，一些基本数据结构的结构很“相似”，可以使用继承机制处理。例如栈、队、字符串等都是线性表的特例，因此，可以视为从线性表继承而来，这样面向对象的继承的优点就体现到了数据结构。

基本数据结构的设立，是为了应用程序的进一步使用。但不同的应用，对基本数据结构的使用都有不同的新要求。按传统的数据结构思想，就必须修改基本数据结构。对面向对象，基本数据结构的使用是通过继承机制实现的。继承机制容许在不改动原对象的条件下，修改原对象（通过重载/多态性），使它符合新要求。因此，将基本数据结构设置为对象（类），极大地方便了基本数据结构的应用，使其成为真正意义上的通用构件。

§1.9.3 数据结构的对象模型

数据结构的对象模型，是指将逻辑数据结构及其成分看做一个带有属性、方法和事件的容器(Container)，定义出它的属性、方法和事件。由于我们这里涉及的是基本数据结构，所以这里不考虑事件。

从实体角度看，数据结构包括元素、元素间的关系以及数据结构整体三大部分，所以，对它的对象定义，也分这三部分。

(一) 元素对象模型

将构成数据结构的数据元素看作单独的实体，定义它的属性和方法。元素的属性是对元素的数据的抽象，元素的方法是对元素数据的操作的代理，它的执行一般改变元素数据和属性。

(二) 关系对象模型

关系也是一种实体，所以，对它的访问，抽象为对象模型有时会更方便。

(三) 数据结构对象模型

数据结构对象表示的是数据结构的全局性的特性，它与多个数据元素及其关系相关。例如，查找操作，由于它涉及多个数据元素，所以属于全局操作。

本章小结

本章主要概述了数据结构中的一些通用的、一般化的概念和方法，主要包括下列几点：

■ 程序设计的三个阶段：算法为中心阶段、结构化程序设计（数据结构）阶段、面向对象阶段；

■ 数据结构的兴起：结构化程序设计带动数据结构的兴起，面向对象将数据结构推向新阶段；

■ 数据结构的研究内容：研究用于描述对象/实体的基本方法（包括计算机表示）与操作。

■ 数据类型、数据元素、数据对象数据结构的概念；

■ 数据结构的分类：集合、线性结构、树形结构、图状结构；

■ 数据结构的基本存储方式：顺序、链式、索引、杂凑；

■ 数据结构访问接口（基本操作）：包括读、写、查找、插入、删除、遍历等，其应满足抽象性、基本性、完备性和支撑性；

■ 对象的概念

■ 面向对象方法要素：封装、继承、多态性、信息传递；

■ 面向对象方法与数据结构的关系

■ 数据结构的面向对象模型；

习 题

1．数据结构的主要研究对象是什么？

2．有那几种基本逻辑结构？它们各自的特点是什么？

3．有那几种基本的存储结构？它们各自的特点是什么？

4．数据结构有哪些要素？

5．什么是数据结构的访问接口？它们分别与逻辑结构和存储结构有什么关系？

6．什么是对象？面向对象技术有哪些要素？

7．数据结构与对象有什么联系？

8．设有一数据结构的形式定义如下：

DS=(D, S)

D={a, b, c, d, e, f}

S={ r1, r2 }

r1={<a,b>, <c,d> }

r2={<a,c>,<c,e>,<e,f>}

请画出它的示意图（数据结构的图示），并指出它属于什么数据结构，然后，画出它的链式存储结构图，这里假定存储区为一块连续空间，元素的存储位置随意。