第 1 章

编程语言的发展与程序结构

本章简述计算机编程语言的发展历史，分析在编程语言发展历史阶段中的非结构化、结构化与面向对象编程语言的特点以及所编写的应用程序的典型结构，从而使读者能够清晰地了解面向对象语言与其他编程语言的联系与区别。重要的是，让读者了解面向对象编程继承了结构化编程的某些优点，而又与结构化编程截然不同。与非结构化和结构化编程模式相比，面向对象编程模式产生了革命性的变化。

1.1 编程语言发展简史

本节从计算机及其编程语言发展历史的时间顺序角度出发，简述计算机编程语言的发展历史。其目的是让读者能够通过本节的讲述，粗略地了解各个阶段的编程语言特征，从而对后面章节要讲述的结构化编程、设计和分析与面向对象编程、设计和分析有一个连贯的理解。期望读者能够理解面向对象的概念、编程、设计和分析的出现并不是突然的，而是在结构化编程、设计和分析的基础上发展起来的。然而，其又是与结构化概念根本不同的革命性的概念。

1.1.1 机械计算机时代的“编程”

编程语言有很多定义，其中一个定义为：“编程语言是一种被设计为利用指令与机器（尤其是计算机）通信的人造语言。编程语言可以被用来创建程序，以控制机器的行为或者表示算法”。

按照以上定义，可以认为，最早的编程语言在计算机被发明之前就存在了。例如，在 19 世纪初，被用来控制、操纵提花织机的穿孔纸带所承载的内容即可理解为一种编程语言。提花织机（Jacquard Looms）是一种机械织机，由约瑟夫·玛丽·雅卡尔（Joseph Marie Jacquard）发明。1801 年，首次证实了该织机简化了在纺织品生产过程中复杂图案的生产过程。该织机是由一些排列成连续序列的穿孔卡片所控制的。每张卡片上带有多行穿孔。每个完整的卡片对应（织物）设计的一行。

另外一个例子是在电子计算机出现 100 年之前出现的机械计算机的设计。由英国数学家 Charles Babbage 在 1837 年设计的命名为分析引擎（Analytical Engine）的通用机械计算机与当代电子计算机原理非常类似。该分析引擎包括：① 算术逻辑单元；② 以条件分支和循环形式存在的控制流；③ 集成内存（memory）。该设计成为第一个符合当今意义下的图灵完

整性（Turing complete）通用目的的计算机。该分析引擎本身的设计逻辑非常先进，是电子通用计算机的先驱。

该分析引擎使用穿孔卡给计算机提供程序和数据的输入，该方法与曾被用来控制、操纵提花织机的穿孔纸带的原理是相同的。在输出方面，该分析引擎设计包括了打印机和曲线画图仪。该机器也可以在卡上打孔，代表数字，以便将来读出。该分析引擎采用十进制数计数。

在该分析引擎设计中，包括一个能存储 1000 个数字的“仓库”（store）。算术单元（mill）可以被用来进行四则运算、比较和开平方运算。和当今的 CPU 一样，mill 将依赖它内部的存储过程，以便执行用户程序中复杂的指令。

用户采用的编程语言有点像当代的汇编语言。该语言使用了三种不同类型的打孔卡：第一种用于进行算术运算；第二种是数值常数；第三种用于进行存储运算，将数字存储到算术单元中或者从算术单元中取回数字。该分析引擎使用三种不同的针对不同类型卡片的“读卡器”。

1.1.2 编程语言的发展历程

本节简述按照时间发展顺序、在各个阶段出现的编程语言特点，这些阶段包括：早期的编程语言阶段，计算机语言基本模式的建立阶段，面向对象编程大发展的年代，互联网年代——大规模软件开发阶段。

1. 早期的编程语言（20 世纪 50 年代初至 60 年代中期）

编程语言是用来编写程序的符号，被用来表示某个计算或算法的详细步骤。一些（不是所有的）作者将编程语言限制为那些能够表达所有可能的算法的语言。

第一批使用指令与计算机交互的编程语言出现于 20 世纪 50 年代。

于 1949 年发布（运行在 BINAC 计算机上），并且于 1950 年（运行在 UNIVAC I 计算机上）和 1952 年（运行在 UNIVAC II 计算机上）完善的 Short Code 编程语言，是有史以来开发的有影响力的早期电子计算机高级语言之一。与使用机器代码直接运行程序的做法不同，Short Code 语言以一般公众可以理解的形式表示数学表达式。因为没有现代概念上的编译器，所以，使用该语言编写的程序必须在每次运行之前被手工翻译成机器代码。这种手工翻译的做法很快被今天所熟知的编译器所替代。

另外一种具有开创意义的编程语言是 1952 年在美国曼彻斯特大学开发完成的 Autocode 语言。该语言的重要意义在于该语言带有编译器，程序员使用编译器将该语言写成的程序自动转换成机器代码。对于生活在今天的计算机和软件专业的师生和计算机工作者来讲，编译器已经习以为常。但是在当时，能想到利用一个程序（编译器）对另外一个程序进行自动（而非手动）编译，即自动将程序代码转换为机器代码，是具有划时代的意义的。

于 1955 年年初发布的 Flow-matic 编程语言的特点是，使用英语语句代替了数学符号。其目的是使得商业人士也会使用该语言编程。Flow-matic 对 COBOL 的设计产生了重大影响。

FORTRAN 编程语言 FORTRAN I 是 1956 年由 IBM 公司开发的，并成为第一个被广泛使用的高级通用编程语言。该语言中包含大量的表示数学公式的语句，主要应用于科学计算。其优点是，具有很强的数学计算能力，使得程序员能够更加容易地将数学公式写入程序，从

而使得程序员再也不用使用汇编语言或者机器语言编写烦琐的程序。其缺点是，缺乏用于商业输入与输出的字符串操作。该语言不断更新的后续版本一直被持续使用到今天。

随着商业的发展，急需可以用于商业、银行业与金融业方面的计算机编程语言，因此，作为世界上第一个商用语言的 COBOL 语言应运而生。COBOL 语言是 1959 年发布的专门用于商业的语言。该语言的特点是面向数据处理、面向文件与面向过程。该语言适用于数据描述与文件处理。其主要应用领域为文件处理与商业应用，例如银行业、金融业软件的编写。该语言直接使用英语语句，而不是数学公式。在该语言中，为了商业上的输入与输出，使用了字符串数据类型。COBOL 语言成为当今还在使用的早期语言之一（当然，增加了很多特点）。

1958 年发布的 FORTRAN II，引入了可以独立编译的子程序的概念，其应用领域为科学与工程计算。

1959 年发布的 ALGOL 60 语言中引入了局部性、动态、递归、BNF 范式、块状结构与数据类型的概念，其应用领域为科学与工程计算。

1960 年发表的 Lisp 语言包含了列表处理、指针与垃圾处理，其特点是在编程中大量使用递归，其应用领域为人工智能。该语言中引入了算法抽象（子程序）的概念。

1964 年发布的 PL/1 语言综合了 FORTRAN、ALGOL 与 COBOL 语言的特点，支持数据抽象、面向过程编程、结构化编程与命令式编程。该语言既支持科学与工程计算，同时也支持商业应用。

从 20 世纪 50 年代末期开始到 60 年代末期出现的编程语言强调使用子程序或者过程进行算法抽象。

2. 计算机语言基本模式的建立（20 世纪 60 年代末期至 70 年代末期）

在 20 世纪 60 年代，尤其是随着半导体与集成电路技术的成熟，计算机硬件的价格大幅度降低，而处理能力却以几何形式大大增强，为使用计算机解决较大与较复杂的问题提供了可能性。然而，这需要对更多种的数据类型进行操作。所以，支持数据抽象的语言诞生了，例如，ALGOL 68 以及随后在 1970 年出现的 Pascal 语言。这些语言支持数据抽象、命令式与结构化编程。

在这个阶段，程序员可以描述相关种类的数据（它们的类型）并且让编程语言强制执行（带有数据结构的）设计决策。这一阶段出现的编程语言的共同特点是支持数据抽象（抽象数据类型）。

20 世纪 60 年代末到 70 年代期间产生了许多主要编程语言。现在使用的大部分语言模式，是在这一时期发明的。

1968 年出现的 ALGOL 68 支持数据抽象、科学计算与商业应用。

1967 年发布的 Simula 语言引入了对象、类、继承、子类与虚拟方法的概念。该语言支持数据抽象，为第一个面向对象编程语言，被用于模拟与仿真。

C 语言是通用的过程式的编程语言。在使用 C 语言编写的程序中，需要大量地使用指针，因此 C 程序运行速度快，并且可执行程序比较小。C 语言适用于所有领域，尤其适合仿真、电子游戏等需要高效率软件编写的领域。

FORTRAN 77 是在 1977 年发布的 ANSI 标准化的 FORTRAN 版本，主要用于进行科学与工程计算，为当今仍然在使用的语言。

1972 年发布的 Smalltalk 语言为支持自底向上设计的完整的面向对象语言。

值得注意的是，早在 1978 年就发布了作为关系型数据库管理系统的标准语言 SQL，其后在 1986 年 10 月，美国国家标准协会对其进行了规范，于 1987 年在国际标准化组织的支持下成为国际标准。

前面只介绍了少量的在 20 世纪 70 年代发布的编程语言，而实际上该时期是编程语言研究最活跃并且产生编程语言最多的年代。那时，越来越多的商业应用使得程序越来越大，因此需要新的语言机制来应对大应用程序开发的需要。令人难以置信的是，这 10 年左右的时间里共出现了 2 000 多种不同的编程语言。然而，只有少数语言存活了下来。虽然如此，许多语言机制为后来的编程语言提供了宝贵的借鉴之处，大多数的现代编程语言都可以在这些语言中找到其祖先。

20 世纪 70 年代也经历了关于“结构化编程”的优点方面相当多的争论，即是否使用 goto 语句编程的问题。在本章的稍后部分，我们将讨论一些 goto 语句的缺点。几乎所有的现代程序员都认为，即使允许使用提供了 goto 语句的编程语言，除了在极其特殊的情况下以外，使用 goto 语句也是不好的编程风格。

【例 1-1】 C 语言与 UNIX 操作系统的研发故事。C 语言是由在美国的 AT&T 和贝尔实验室（Bell Labs）工作的哈佛大学毕业的数学博士 Dennis Ritchie 在 1969 年到 1973 年之间开发的。UNIX 操作系统是由一个在贝尔实验室工作的包括 Ken Thompson、Dennis Ritchie、Brian Kernighan、Douglas McIlroy、Michael Lesk 和 Joe Ossanna 在内的团队在 1969 年开始研发的，其第一个版本在 1971 年发布。UNIX 操作系统原来是由汇编语言开发的，但是到了 1973 年，当 C 语言研发成功以后，研发组使用 C 语言重新改写了 UNIX 操作系统。由此可见，UNIX 操作系统和 C 语言就像一对孪生兄弟一样。C 语言与 UNIX 操作系统很快成为世界的主导。

1983 年，Ritchie 和 Thompson 一起获得了图灵奖（Turing Award），原因是他们研发了通用的操作系统理论，并且实现了 UNIX 操作系统。1990 年，Ritchie 和 Thompson 接受了 IEEE 研究所颁发的 IEEE Richard W. Hamming 奖牌。1999 年 4 月，Ritchie 和 Thompson 接受了 1998 年度美国国家技术奖牌，由克林顿总统颁奖。

UNIX 操作系统是发达国家已经使用了多年的操作系统，许多缺陷已经得到修改。与 Windows 操作系统相比较，UNIX 操作系统具有如下几方面的优点：

（1）稳定性：更稳定，不会经常死机，因此需要较少的维护和管理；

（2）安全性：具有强大的内置安全性；

（3）处理能力：具有更强大的处理能力。

3. 面向对象编程大发展的年代（20 世纪 80 年代）

在 20 世纪 80 年代，没有创造新的编程语言模式，主要的语言设计方面的工作都集中在巩固、细化过去 10 年发明的语言模式上。

毋庸置疑，整个 20 世纪 80 年代是面向对象理论、设计与编程大发展时期。基于 20 世纪 70 年代的面向对象的研究，在 20 世纪 80 年代初期，出现了在商业软件开发方面有实际应用的面向对象编程语言。

发布于 1980 年的 Smalltalk 80 为纯粹面向对象语言，最初为了教学目的，后来作为实际的商业软件开发。

1983 年发布的 C ++ 语言，结合了 C 和 Simula 的优点，为通用的面向对象语言。C ++ 语

言的特点是广泛地使用指针，因此用 C ++ 语言开发的程序运行速度快，可以用于所有的领域。C ++ 语言的出现受到了广大软件开发者的欢迎。经过了 30 多年，C ++ 语言仍然是当今人们青睐的面向对象编程语言，尤其用于模拟、网络电子游戏方面。

1983 年发表的 Ada 83 受 Pascal 语言的影响，属于强类型的面向对象编程语言。美国政府针对 Ada 语言进行了标准化，同时 Ada 语言成为用于国防承包商的一种编程语言。

这些面向对象语言的共同特点是：支持面向对象编程、设计与分析。

4. 互联网年代——大规模软件开发（20 世纪 90 年代至现在）

在 20 世纪 90 年代中期，互联网应用的快速增长对于编程语言产生了重大的影响。通过开放的计算机系统的全新的平台，互联网的发展给创造与之相适应的新语言一个全新的机会。此阶段有重要影响的语言是 Java 与 JavaScript 语言。

Java 是由 Sun Microsystems 公司在 1995 年推出的面向对象程序设计语言（以下简称 Java 语言）和 Java 平台的总称。Java 语言具有跨平台、动态 Web 开发与 Internet 计算的特点。Java 语言自面世后就非常流行，发展迅速。在全球云计算和移动互联网的产业环境下，Java 语言更具备了显著优势和广阔前景。Java 语言是一个纯粹的面向对象的程序设计语言，它继承了 C ++ 语言面向对象技术的核心。Java 语言舍弃了 C ++ 语言中容易引起错误的指针（以引用代替）、运算符重载与多重继承（以接口取代）等特性，增加了用于回收不再被引用的对象所占据的内存空间的垃圾回收器功能，使得程序员不用再为内存管理而担忧。

JavaScript 是一种直译式脚本语言，是一种动态类型、弱类型和基于原型的语言。JavaScript广泛用于客户端的脚本语言，最早在 HTML（标准通用标记语言下的一个应用）网页上使用，用来给 HTML 网页增加动态功能。

从 1990 年开始，强调利用 Framework 进行编程，从而导致了 J2EE（现已改为 JavaEE）与 . NET Framework 的产生。Framework 通过组件与服务为程序员提供了大量的支持。

可以说，一个 Framework 就是一个可复用的设计组件，它规定了应用的体系结构，阐明了整个设计与协作构件之间的依赖关系、职责分配和控制流程，表现为一组抽象类以及其实例之间协作的方法，为构件复用提供了上下文（Context）。Framework 的使用使得软件的生产率有了极大的提高。

1990 年以后发布的编程语言与 Framework 的特点是支持大规模软件开发。

1.2 非结构化编程简介

非结构化编程（程序设计）是历史上最早的能够实现图灵完整算法的编程模式。历史上，过程化编程、结构化编程与面向对象编程都是非结构化编程的后继者。

使用非结构化编程思想设计的编程语言既包括低级编程语言，也包括高级编程语言，包括早期的 BASIC、JOSS、FOCAL、MUMPS、TELCOMP 和 COBOL 语言。

通常，利用非结构化的编程语言所写的程序包括依次排列的命令或语句，每行包含一条语句，并且每行都标有编号，以方便程序执行流从一行跳转到任何其他的一行。

非结构化编程引入了基本的控制流的概念，如循环、分支和跳转。虽然在非结构化的范式中没有过程（procedures）的概念，但是子程序是允许的。和过程不同的是，子程序可以有多个入口和出口，直接跳入子程序或者跳出子程序在理论上是允许的。

非结构化的编程语言广泛使用 goto 语句，使得程序的执行可以从一行跳转到任何其他的一行。也就是说，goto 语句是改变程序执行流的重要语句。如果没有 goto 语句，则程序的执行只能按照源程序中的语句自第一行开始，逐行执行到最后一行，中间偶尔会调用某个子程序。可以认为，goto 语句是非结构化编程语言最重要的特征之一。

但问题是，在提供程序执行时跳转便利的同时，goto 语句也给分析用非结构化语言编写的程序，尤其是比较大的程序，带来了很大的混乱。原因是，如果 goto 语句使用过多，程序执行的逻辑流就变得很难跟踪。例如，考虑一个具有 2000 行代码的程序，在代码中使用了 85 次 goto 语句，一会儿从第 280 行跳转到第 1235 行，一会儿又从第 1235 行跳转回第 66 行，等等。正如 20 世纪六七十年代软件业人士批评的那样，这样的程序就像是意大利面条一样，卷曲在一起，要找出一根面条的头和尾在哪里很难，而要找出一碗面条中每根面条的头和尾的位置，就更加困难。关于 goto 语句的辩论进行了很多年才结束。最后，人们普遍认识到，由非结构化语言编写出的程序代码可读性很差，难以理解和维护，很难被用于实现重大的软件项目。

另外，非结构化语言只允许使用基本的数据类型，如数字、字符串和数组。尽管还缺乏结构化数据类型，在非结构化语言中引入数组却是一个显著的进步，使得流数据处理成为可能。

1.3 结构化系统分析与设计方法简史

在整个 20 世纪 50 年代到 60 年代中期，几乎没有较好的程序设计和编程技术方面的指导，也没有记录需求与设计的标准技术。在 20 世纪 60 年代后期，软件在多个行业中有了较为广泛的应用。随着软件系统变得越来越大，越来越复杂，软件设计与开发也变得越来越困难。随之而来的问题是产生了软件危机：软件超预算，不能按时交付，所生产的软件的功能与用户需求相去甚远，从而急需新的软件开发方法论。

在这种大背景下，在 20 世纪 60 年代中期，产生了结构化编程的概念。然后，在 20 世纪 60 年代后期由 Larry Constantine 提出了结构化设计的概念。20 世纪 70 年代《结构化设计》（Structured Design）一书出版之后，才出现了结构化分析的概念与技术。

虽然按照结构化技术发展的历史顺序，结构化编程在前，然后是结构化设计，最后才是结构化分析理论的出现，但是在下面的讲述中，还是遵循现代软件开发流程的正确顺序：即先讲结构化分析，然后是结构化设计，最后才讲述结构化编程。

1.3.1 结构化分析

结构化分析（Structured Analysis，SA）和结构化设计（Structured Design，SD）首先分析业务需求，然后将业务需求转换为软件需求规格说明书，最后获得计算机程序与硬件配置等。

结构化分析从 20 世纪 80 年代开始流行，今天仍然被许多人使用。在结构化分析中，使用数据流图（Data Flow Diagrams，DFD）将问题分解为各个处理步骤。数据流图采用图形方式来表达系统的逻辑功能、数据在系统内部的逻辑流向和逻辑变换过程，是结构化系统分析方法的主要表达工具。由于它只反映系统必须完成的逻辑功能，所以它是一种功能模型。数

据流图中的每个气泡代表一个包含多个子程序（或者函数）的处理模块；而对于层级较高的数据流图，一个气泡可能仅仅代表一个子程序或者函数。

明显地，数据和控制从一个气泡到数据存储再到另外一个气泡的跟踪是非常困难的，原因是气泡数目可能非常大（设想一个具有 200 个气泡的数据流图）。一种改进方法是首先定义需要系统反应的外部世界事件，然后用一个气泡代表一个事件，然后将需要交互的那些气泡连接起来，从而定义整个系统。大量的气泡导致系统很难理解，因此，通常将一些气泡组织成更大的（高层的）气泡。

结构化分析方法属于流程驱动。这种方法确定总体功能并且利用迭代的方法将功能（函数）划分成更小的功能（函数），同时保持输入、输出、控制和优化流程所需的机制不变。这种方法也被称为功能分解方法，它侧重于函数的内聚与函数间的耦合。

结构化分析从数据流经系统的视角考虑整个系统。系统的功能通过许许多多的变换数据流的过程（函数）描述。结构化分析利用分而治之（Divide and Conquer）技术，将一个问题分解为若干子问题，然后再次对每个子问题进行类似的分解，依次进行分解，一直到问题细化到比较容易理解与处理的程度才停止。这种技术被称为自顶向下的功能化逐层分解技术。自顶向下的功能化分解活动产生了很多层的 DFD。这使得设计者能够将注意力集中在相关的细节而避免纠缠无关的细节。随着细节层次的增加，问题变得越来越容易理解。

在结构化分析中，除了使用数据流图以外，还使用实体关系图（Entity Relationship Diagram，ERD），以表达系统中出现的实体及实体之间的关系。下面粗略地介绍数据流图与实体关系图并且给出两个例子。

（1）数据流图

数据流图是一个数据如何流过某个信息系统的图形表示。通常的做法是，首先画出系统环境图，表示系统和外部实体之间的相互作用。数据流图用来表示系统是如何被划分为更小的部分的，并且突出数据是怎样在这些部分之间流动的。然后，该环境图可以被扩展以便显示系统建模的更多细节。

数据流图对于软件开发者和最终客户都是很重要的。通过绘制数据流图，程序员可以分析、分解一个系统。通过观察数据流图，用户能够以可视化的方式了解系统将如何运作，这个系统将要完成什么以及该系统将怎样被实现。数据流图可以用来为最终用户提供一个比较好理解的物理概念。

【例 1-2】 考虑设计一个程序，使用二分搜索算法（Binary Search）对一个整数数组进行搜索。程序要求从一个文件输入一个整数数组，输入的时候要验证数组的每项是否都是整数。然后输入一个待查询的整数，针对此数组进行查询，看是否包含该整数。如果是，就返回该元素所在的数组中的位置；否则，返回值 -1。该程序的二层与三层 DFD 如图 1-1 和图 1-2 所示。

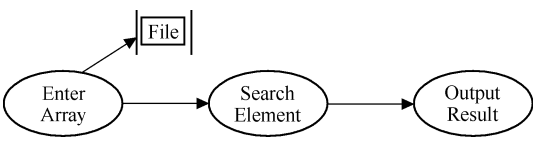

图 1-1　使用二分搜索算法实现的程序的二层 DFD

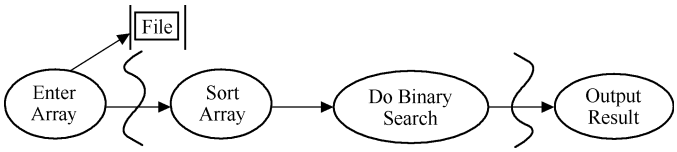

图 1-2　使用二分搜索算法实现的程序的三层 DFD

（2）实体关系图

实体关系图是指以实体、实体的属性和实体之间的关系三个基本概念概括数据的基本结构，从而描述静态数据结构的概念模式。实体关系图提供了表示实体、属性和关系的方法，用来描述现实世界的概念模型。实体关系图表示在信息系统中概念模型的数据存储。实体关系图的构成如下。

① 实体（Entity）：用矩形框表示，矩形框内写明实体名。例如轿车、驾驶员都是实体。实体可以是人、事物、地方，甚至是收集的数据。

② 属性（Attribute）：用椭圆形框表示，并用无向边将其与相应的实体连接起来。例如，轿车的生产商、品牌、型号，驾驶员的姓名、身份证号、驾照号、性别都是属性。

③ 联系（Relationship）：用菱形框表示，菱形框内写明联系名（如驾驶），并用无向边分别将有关实体连接起来，同时在无向边上标记联系的类型（1∶1，1∶n 或 m∶n），就是存在的三种关系（一对一，一对多，多对多）。例如，驾驶员与轿车之间存在关系，一个驾驶员可以驾驶多台轿车（1∶n）。

【例 1-3】 实体关系图举例。考虑学生实体和课程实体的对应关系。一个学生（student）实体拥有的属性包括姓名（name）、身份证号（idNum）、性别（sex）与学生号（studentNum）。课程（course）实体包含课程名（cName）、课程号（cNum）、学时（hours）与学分（credit）。实体关系图如图 1-3 所示。

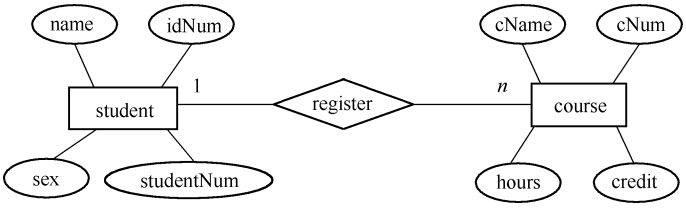

图 1-3　实体关系图的例子

以上的实体关系图表示学生与课程的关系是注册（register）关系，表示一个学生可以注册多门课程。

结构化分析还有很多内容，由于本书的目的是讲述面向对象技术，所以略去了与本书目的无关的其他内容。

1.3.2　结构化设计

结构化设计是指在结构化分析中产生的数据流图的基础上，设计出程序相应的模块和子程序（函数）。结构化设计产生程序组织结构图（Structure Chart）。

结构化设计关注的是程序模块的开发，这些模块被组织成层次状。结构化设计强调以下

两个应该遵守的众所周知的重要原则。

（1）高内聚原则：关注将功能相关的过程分配到某个特定的模块。

（2）低耦合原则：尽量减少模块之间的接口，因此可以减少由此设计而产生的软件复杂度。

组织结构图是由分析阶段所产生的 DFD 翻译而来的，用来显示模块的层次结构与模块调用的序列关系。组织结构图将程序模块组织成一棵根朝上的树状结构，其每个节点代表一个模块，表示怎样将系统分解为最低可管理的层次。每个模块由一个矩形盒子表示，包含模块的名称。该树状结构以可视化的方式表示模块之间的相互关系。

【例 1-4】 由例 1-2 在分析阶段所产生的三层 DFD，可以转换成如图 1-4 所示的程序组织结构图。从图 1-4 中可以看出，此程序由 6 个模块组成。由于此程序比较简单，所以组织结构图中的每个模块都可以由一个子程序（例如，C 语言中的函数）代表。

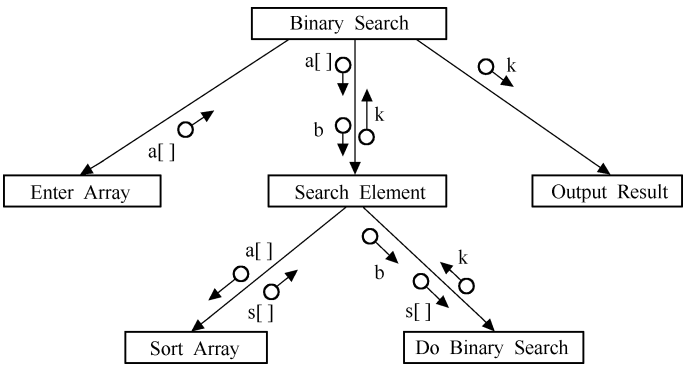

图 1-4　使用二分搜索算法实现的程序组织结构图

该程序可以看作使用自顶向下的功能化分解而得到的一棵根朝上的树。最上层代表高层逻辑，第二层模块的任务比最上层更具体一些，最低层模块的任务最具体。另外，由此程序组织结构图可以明显地看出，程序的执行顺序是自上而下、自左到右的，同时也可以明显地知道每个模块的参数与返回值。整个程序功能的正确与否，取决于最低层。这是结构化设计的重要特点之一。

在结构化设计中，组织结构图可以被用来明确指定一个程序的高层设计（或者架构）。作为一个设计工具，它可以帮助程序员将一个大的程序以递归的方式分解成小到能被人脑理解的部分。这个过程称为自顶向下的设计，或功能化分解。像建筑师使用蓝图建造房子一样，程序员使用结构图建立一个程序。在设计阶段，该图可以被用来为客户和软件设计师之间的沟通服务。在程序实现阶段，该图指导程序员开发出带有正确的输入与输出的模块。

1.3.3　结构化编程

结构化编程兴起于 20 世纪 60 年代。1968 年，荷兰计算机科学家 E. W. Dijkstra 在《ACM 通信》上发表了一篇批评 goto 语句的题目为“goto 语句看来是有害的”的论文，引起了一场长达 10 余年的程序设计界的长期论战。他认为 goto 语句太原始，是造成程序混乱不堪的祸根，应该从所有的高级程序设计语言中清除掉。第二年，他又首先提出结构化程序设计的概念。

于是，学者与软件工程师们再研究使用新的编程方法：使用其他的编程技术代替原来的为了控制程序执行流而使用的 goto 语句，即在没有 goto 语句的情况下，编写的程序也能完成预设的功能。事实上，这方面的理论研究与实践是基于之前发表的著名的结构化程序定理的。

结构化程序定理是在 1966 年 Corrado Böhm 和 Giuseppe Jacopini 发表的论文中提出的，是结构化编程的理论基础。它指出，只要一种编程语言可以依据以下三种调整控制流程的方式组合其子程序及调整控制流程，则每个可计算函数都可以用这种编程语言来表示。

① 顺序结构——按照顺序执行一个子程序，然后再执行另一个子程序；

② 选择结构——根据布尔表达式的值，执行两个子程序中的一个子程序；

③ 迭代结构——执行子程序，直到一个布尔表达式为 true。

换句话说，一种编程语言只要定义了以上三种结构，则可以实现任何可计算算法。

结构化编程理论基于以上的结构化程序定理，试图使用结构化程序定理中所提出的三种结构来代替之前非结构化编程中所使用的 goto 语句。

结构化编程定义：结构化编程是一种编程模式，其目的是改善计算机程序开发的透明度、质量和开发时间（节省开发时间）。通过广泛使用子程序、块状结构及 for 和 while 循环代替之前非结构化编程中所使用的“简单测试后再使用 goto 语句”的模式，而实现程序所需的业务逻辑。

具体地说，在较低的层次，结构化程序通常由简单的层次化的程序流结构控制，包括顺序结构、选择结构和迭代结构，如图 1-5 所示。

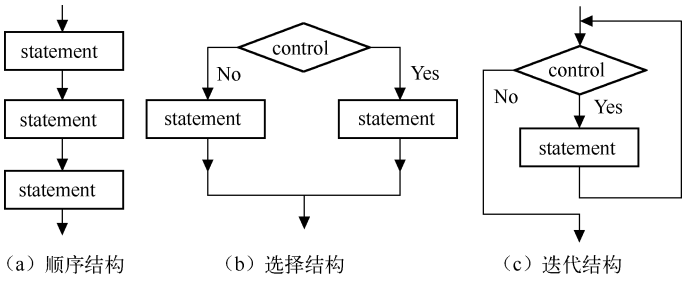

图 1-5　结构化编程的简单的层次化的程序流结构

图 1-5 中的三种结构具体解释如下。

①“顺序结构”：按照顺序执行程序中的语句。

②“选择结构”：根据程序的状态，选择执行一组语句中的某条语句。这通常可以用关键词 if…then…else…endif、switch 或 case 表示。

③“迭代结构”：反复执行某条语句，直到程序到达某个状态以后才停止。例如，将某种操作作用到一个集合中的所有元素。该循环结构通常由关键词 while、repeat、for 或者 do…until 所控制。建议每个循环应该只有一个入口点。

注意，在结构化程序设计中，一个程序被设计为许多层，每层由若干模块组成，每个模块都有一个单一的入口点和一个单一的出口点。程序控制是自顶向下的，下层不允许无条件转移到更高的层。这也是结构化程序设计与非结构化程序设计的本质区别之一。

注：一种语言如果存在在封闭的关键词内包含结构的语法，则被称为具有块状结构的语言。例如，ALGOL 68 中的“if 某段代码 fi”，或者是 PL/1 语言中的“BEGIN 某段代码 END”，或者 C 语言和后来的许多语言中的花括号“{}”。

一些最初用于结构化编程的语言包括 ALGOL、Pascal、PL/1 与 Ada。但是大多数从那时开始出现的过程化编程语言中都包含了鼓励结构化编程的语言特点。有趣的是，在有的编程语言中，故意取消了 goto 语句，其目的是防止利用结构化语言进行非结构化编程。

虽然结构化程序定理保证了一种语言如果具备了顺序、选择和迭代三种结构，则可以实现任何可计算算法，并且 goto 语句已经在很大程度上被结构化编程所需要的结构体（if/then/else）和循环（while 和 for）所代替，但是，大多数结构化语言不是纯粹的结构化编程语言。例如，结构化编程所要求的每个模块（子程序，函数）都有一个单一的入口点和一个单一的出口点的规则，在实际编程中却很难遵守。在实际的编程中，往往需要在特定的情况下，方便地从一个子程序（例如 C 函数）中提前退出。出于这方面的需要，在许多支持结构化编程的语言中，都提供了 return 语句，以便程序的执行可以从一个子程序中提前退出；而为了从某个循环中退出，则需要提供 break 或 continue 语句，以便程序的执行能够终止当前的循环，而进行下一次循环。这就导致出现了多个退出点，而不是结构化程序设计所需的单一出口点。

关于这方面的讨论细节，已经超出了本书的范围，读者可以查询相应的文献。

尽管结构化分析与设计有其优点，但是与后来的面向对象分析与设计相比较，仍然有其缺点。结构化分析与设计的缺点如下。

（1）在需求非常清楚的情况下，经过分析步骤的 DFD，可以比较准确地获得程序组织结构图。而且组织结构图中的每个模块（函数）的输入和输出都是很明确的。但是，问题是当需求发生变化的时候，就要求对程序结构图的某些部位进行“剪枝”与“嫁接”，因此，程序的改变与功能扩展都比较困难。

（2）DFD 具有强烈的面向功能的特点，因而经常被改变。DFD 尽管强调了“数据流”，但是并没有强调“数据模型”，所以对系统主题到底是什么仍然理解甚少。

（3）由于 DFD 与程序组织结构图使用不一样的符号，因此对于比较复杂的问题，从 DFD 到程序组织结构图的翻译也可能不太准确，有的时候还比较困难。

1.4 非结构化程序设计与结构化程序设计的区别

本节讨论非结构化与结构化程序的拓扑结构。拓扑结构反映了编程语言的最根本的特征。

1.4.1 非结构化程序的特点

非结构化语言是历史上最早的能够创建图灵完备算法的程序语言。在它之后，又出现了适用于结构化程序设计、面向对象程序设计的编程语言。非结构化编程语言编写的程序具有如图 1-6 所示的样式。

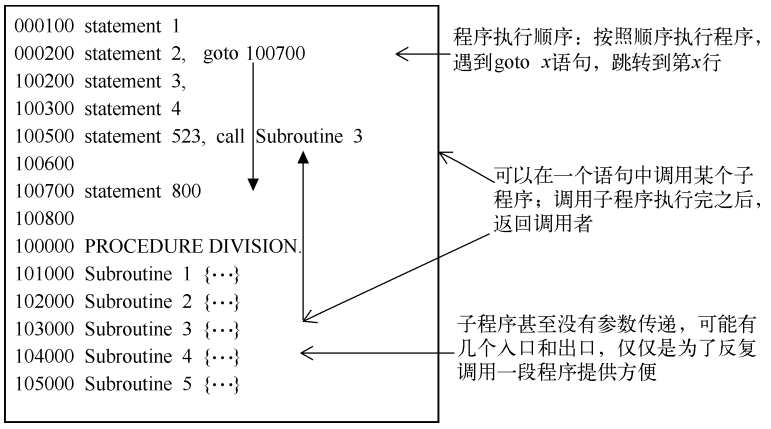

图 1-6　非结构化程序具有的样式

在如图 1-6 所示的程序中，程序的每行都有一个行标号，程序执行流是按照从上至下逐行执行的原则进行的。可以在一条语句中调用某个子程序；调用子程序执行完之后，返回调用者。这些子程序还不是今天理解的函数。子程序甚至没有参数传递，可能有几个入口和出口。那时的子程序往往就是为了方便调用的程序片段。

在程序执行的时候，如果遇到 goto *x* 语句，跳转到第 *x* 行。goto 语句给程序执行中的跳转提供了极大的方便。正如一位作者所赞美的那样："利用 goto 语句写的程序就像写 14 行诗那样潇洒。"然而非结构化程序中的 goto 语句，导致非结构化程序在逻辑上很难理解。程序中，goto 语句不仅可以从前向后跳转，还可以从后向前跳转，程序执行流可以从任何一行跳转到任意另外一行。如果一个程序中多处出现这种跳转情况，将会导致程序流程无序可寻，程序结构杂乱无章，这样的程序是令人难以理解和接受的，并且容易出错。在实际软件产品的开发项目中，更多地追求软件的可读性、可扩展性与可维护性，像这种结构和风格的程序是不允许出现的。尽管如此，在计算机编程语言发展的特定历史时期，即探索时期，非结构化编程还是有很多成功的应用的。另外，非结构化程序设计也为后来的结构化程序设计与面向对象分析与设计提供了宝贵的经验。

非结构化程序设计的缺点如下。

① 由于大量使用 goto 语句，程序的逻辑难以掌握；

② 所有的代码，包括子程序，都可以随意修改全局数据；

③ 程序的调试困难。一行代码的错误可能引起毁灭性的传导效应，将错误传导到整个系统。

1.4.2　结构化程序的特点

进入 20 世纪 60 年代中期，出现过程抽象的概念。虽然子程序的概念在 20 世纪 50 年代就出现了，但是当时的子程序只是为了使得编程能够更省力，只是一个可供反复调用的程序块，没有参数传递。一直到 20 世纪 60 年代中期，子程序才被真正地理解为抽象。

将子程序看作一种抽象机制对编程语言与程序设计具有重要的意义。在这样的思维模式下，出现了结构化设计方法，为建造将子程序作为基本的物理编程块的大程序的设计者提供指导。结构化设计方法采取自顶向下的功能化设计方法。与非结构化的语言相比，使用结构

化设计方法产生的程序可以适应较大规模的编程。

与非结构化程序设计相比，在结构化程序设计中，子程序得到了改进。子程序被视为“功能抽象”，支持参数传递。子程序成为基本的“编程块”，并且子程序可以嵌套。子程序中的局部变量不允许其他代码访问。另外，结构化程序设计的一大标志性特点是，抛弃了goto 语句。使用结构化语言编写的程序具有如图 1–7 所示的样式。

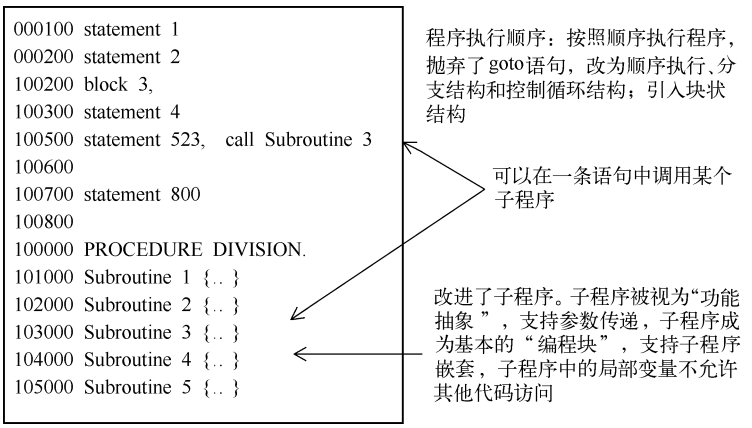

图 1–7　结构化程序具有的样式

利用结构化语言编写的程序的优点如下。

（1）与非结构化程序相比较，利用结构化语言编写的程序，由于抛弃了 goto 语句，程序逻辑变得比较容易追踪。

（2）与非结构化程序相比较，改进了子程序，支持模块与子程序的参数传递。

（3）模块的引入，为大规模编程奠定了基础。

利用结构化语言编写的程序的缺点如下。

（1）与非结构化设计类似，用结构化语言设计的程序中，仍然存在全局数据。每个过程（子程序、函数）都可以直接访问与修改全局数据。

（2）大多数 20 世纪 60 年代出现的编程语言对于数据抽象与强类型的支持较少，所以，很多数据类型的错误只有到了程序执行的时候才能被发现。

1.5　面向对象编程中对象与类的初步概念

本节的目的是结合软件设计发展的历史，介绍在面向对象编程中，对象与类引进的原因，以及对象与类的本质。然后给出对象与类的简单定义与例子，使得在正式给出对象与类的定义之前，读者可以对本书中出现的对象与类之类的词汇不再陌生。事实上，只要读者已经学习了一门面向对象编程语言，就会对对象与类的概念有初步的了解。

在计算机编程语言从非结构化到结构化程序设计的发展历程中，出现了两种具有重大影响的趋势：

（1）首先是强调过程抽象，体现在子程序和函数之中；

（2）然后是进行数据抽象，体现在抽象数据类型之中。

过程抽象适合描述抽象操作。通过过程抽象的概念，产生了编程语言中的带有参数的子程序与后来的函数（例如 C 语言中的函数）。这对于结构化设计中采用的自顶向下的功能化设计非常重要。在这样的设计中，按照分而治之的策略，将一项大的任务分离为若干个子任务，按照逐层细化的策略进行设计。

但是过程抽象特别不适合描述抽象对象。这是一个严重的缺点，因为在许多应用程序中，被操纵的数据对象的复杂性决定了问题的整体复杂性。数据抽象对于管理复杂度很重要。

20 世纪 70 年代后期，人们对数据抽象与数据类型方面进行了广泛的研究。这些研究产生了两方面的重要结果：

（1）数据抽象的研究，导致数据驱动方法产生。该方法为利用面向算法的语言进行数据抽象的问题提供了系统的解决方案。

（2）数据类型的研究，导致类型的概念和理论出现，最终在如 Pascal 语言中实现。

结合以上强调过程抽象与数据抽象的研究结果，产生了面向对象设计中对象的概念。

对象的通俗定义：一个对象可以被理解为一个封装了数据结构与方法的实体。也就是说，在一个对象中，既包含数据抽象（对象封装的数据部分），也包含过程抽象（对象的函数或者方法部分）。

过程抽象与数据抽象的研究结果导致了面向对象语言 Simula、Smalltalk、Object Pascal、C ++ 、Ada、Eiffel 与 Java 等语言的出现。在面向对象语言中，使用类作为编程的最小模块。

类的通俗定义：类是创建对象实例的模板，是同种对象的集合与抽象。它包含所创建对象的属性描述和行为特征的定义。类是一个集合，而对象是这个集合中的一个实例。例如，各种不同的具体的鸟都属于鸟类。各种不同的鸟对象都可以由该鸟类生成。类的属性和方法定义了类的接口。

例 1–5 利用一个简单的类图（将在第 2 章与第 3 章中正式介绍类图，而学习过C ++ 或者 Java 语言的读者可以很容易地理解此简单类图）来说明面向对象分析、设计与编程中的对象是结合了数据抽象与过程抽象的实体。

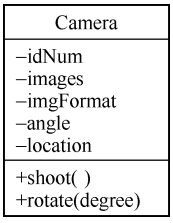

图 1–8　摄像头类的设计

【例 1–5】　一个简单的类。如图 1–8 所示的摄像头类的设计说明类中包含数据抽象与过程抽象。在此类图的三个区域中，最上边的区域代表类名，中间的区域代表属性部分，最下面的区域代表操作（C ++ 函数、Java 方法）部分。

在图 1–8 中，数据抽象部分体现在类的属性部分中所封装的数据 idNum、images、imgFormat、angle 与 location 分别代表该摄像头的号码、所摄制的图像、图像格式、摄像头旋转角度与摄像头所安装的物理地点；而该类所封装的方法 shoot( )代表摄制图像功能的过程抽象，rotate（degree）代表摄像头可以旋转一定角度功能的过程抽象。该类图的属性部分在 Java 语言实现中表现为私有变量，是客户类不能直接访问的变量；而 shoot( ) 和 rotate (degree)为 Java 公有方法，是客户类可以调用的方法。

例 1–6 介绍类的继承关系，该关系是面向对象设计中十分重要的关系。

【例 1–6】　类的继承关系。假设要使用 Java 类来设计一个供鸟类专家研究用的鸟类。由于鸟类是一个庞大的家族，所以要将鸟类归类。图 1–9 代表鸟类的设计。

在该类的设计中，其超类 Bird 代表了鸟类的类型，被设计为一个抽象类（用斜体表示）；两个子类 Swallow 和 Swan 代表两个具体的鸟类。子类继承了超类中的某些特征，在此类的设计中，超类的属性被标记为受保护（符号#），因此两个子类可以继承与修改其超类的属性部分 name、species、sex 和 age。在两个子类的代码中，可以为其超类的属性赋值，改变属性值与直接获取属性值。另外，两个子类也继承了其超类 Bird 的 describe( ) 方法，该方法用于描述鸟类的共同特征，例如，鸟类有羽毛，可以飞等。在本例子中，超类有两个抽象方法fly( ) 和 feed( )（用斜体表示）。这两个抽象方法可以被其两个子类 Swallow 和 Swan 具体地实现。原因是小燕子与天鹅的飞行方式与进食方式都不同，因此需要有不同的实现。另外，值得注意的是，当客户类创建了某个子类的对象以后，要想获取超类的属性，必须调用其超类中所声明的 gets 方法。

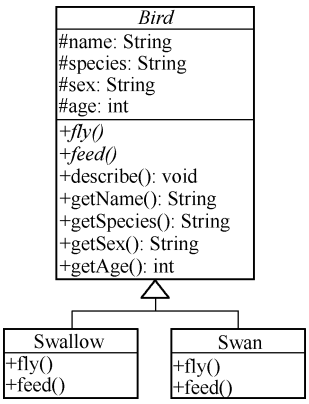

图 1-9　鸟类的设计

1.6　面向对象程序与结构化程序的区别

为了克服结构化程序存在全局数据（变量），并且全局数据可以被随意修改的缺点，面向对象设计将数据进行“局部化”，即把程序中的数据分配到各个类之中，以数据隐藏的方式，使得数据不能被非法访问。具体地说，一个类将数据与作用在数据上的操作以及其他操作捆绑在一起，封装在类里面。如果其他的类要访问或者修改某个类的数据，必须调用该类的相关的数据访问或数据修改方法，才能做到。类的简单介绍已经在 1.5 节中给出，第 4 章还要对类进行更加详细的介绍。

面向对象设计与结构化设计具有很多不同之处，下面仅仅列出最本质性的几点区别。

（1）在结构化程序设计中，数据与子程序（函数）是分离的，如图 1-10 所示。每个子程序都可以修改数据，使得在一处出现的错误可以被“传导”到很远地方。而在面向对象设计中，将数据以及作用在数据上的操作封装在类中，使得数据不能被该类之外的类非法访问，即针对数据进行封装，实现信息隐藏，如图 1-11 所示。

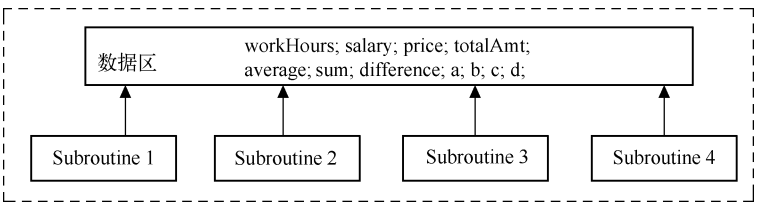

图 1-10　结构化设计的特点——全局数据或变量可以被随意修改

（2）在结构化设计中，设计的基本单元是过程（子程序，函数），过程是由动词表达的。而在面向对象设计中，设计的基本单元是类，而类名是由名词表达的。因此，可以认为，结构化程序由动词组成，而面向对象程序由名词组成。

（3）采取结构化设计的程序结构像一棵根朝上的树，而面向对象程序的结构像一个图

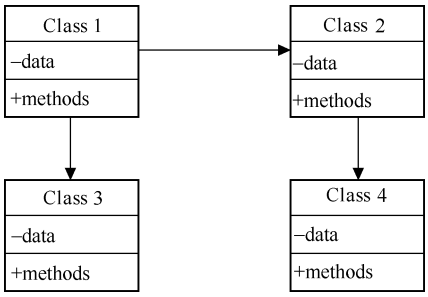

图 1–11　面向对象设计将数据封装在类里面——确保数据不可以被非法访问

(这一点将在后续的章节中体现)。

1.7　本 章 总 结

本章从软件开发与设计的历史视角讲述面向对象软件的逻辑架构。作为一个计算机科学与技术或者软件工程学科的毕业生，必须对历史上的软件开发与设计的情况有所了解，尤其应该清楚地了解结构化分析与设计、面向对象分析与设计之间的联系和区别。这样才能对面向对象分析、设计、编程有更深入的理解。

与非结构化设计相比较，结构化设计的优点是，使用自顶向下的功能化分解方法，使得设计成果，即程序组织结构图表现为从主函数出发，调用下一层函数，然后下一层的函数还可以调用更下一层函数。设计的结构可以被理解为一棵根朝上的树。结构化设计适用于比较大的程序（实践表明，不超过 100 000 行代码)。而面向对象设计所产生的文件为一系列相互关联的类。面向对象设计与编程适合任意大小的程序。

1.8　练　习　题

1. 在非结构化设计中，起关键作用的 goto 语句在当时的历史条件下，有何重要意义?

2. 为什么结构化设计主张抛弃 goto 语句? 没有 goto 语句的结构化语言是否可以实现任何可计算函数（有何理论根据)?

3. 指出利用结构化语言实现的程序结构与利用面向对象语言实现的程序结构有何不同?

4. 面向对象程序中的数据为什么不能被非法访问?

5. 为什么说采用结构化设计的程序由动词组成，而采用面向对象设计的程序由名词组成?

6. 结构化设计拓扑结构像一棵根朝上的树，这一点是否说明整个程序的高层逻辑位于这棵树的高层节点部分，而低层逻辑位于较低的节点部分?