上篇　程序设计语言的设计

第1章　绪　　论

本章将讨论程序设计语言中的一些重要概念，为深入了解程序设计语言打下基础。最后一节简单介绍程序设计语言的发展历史。

1.1　引　　言

语言是人们交流思想的工具。人类在长期的历史发展过程中，为了交流思想、表达感情和交换信息，逐步形成了语言。这类语言，如汉语和英语，通常称为自然语言(Natual Language)。另一方面，人们为了某种用途，又创造出各种不同的语言，如旗语和哑语，这类语言通常称为人工语言(Artificial Language)。

1946 年出现了第一台电子数字计算机(Electronic Digital Computer)，它一问世就成为强有力的计算工具。只要针对预定的任务(问题)，告诉计算机“做什么”和“怎么做”，计算机就可以自动地进行计算，对给定的问题求解。为此，人们需要将有关的信息告诉计算机，同时也要求计算机将计算结果告诉人们。这样，人与计算机之间就要进行通信(Communication)，既然要通信，就需要信息的载体。人们设计出词汇量少、语法简单、意义明确的语言作为载体，这样的载体通常称为程序设计语言(Programmig Language)。这类语言有别于人类在长期交往中形成的自然语言，它是由人设计创造的，故属于人工语言。本书将讨论这类语言的设计(Design)和实现(Implementation)。

每当设计出一种类型的计算机，就随之产生一种该机器能理解并能直接执行的程序设计语言，这种语言称为机器语言(Machine Language)。用机器语言编写的程序由二进制代码组成，计算机可以直接执行。对人来说，机器语言程序既难编写，又难读懂。为了提高程序的可写性(Writability)和可读性(Readability)，人们将机器语言符号化，于是产生了汇编语言(Assemble Language)。机器语言和汇编语言都是与机器有关的语言(Machine-dependent Language)，通常称为低级语言(Low-level Language)。其他与机器无关的程序设计语言(Machine-independent Language)，通常称为高级语言(High-level Language)。由于计算机只能够理解机器语言，可直接执行用机器语言编写的程序，而用汇编语言和高级语言编写的程序，机器不能直接执行，必须将它们翻译成功能完全等价的机器语言程序才能执行。这个翻译工作是自动进行的，由一个特殊的程序来完成。将汇编语言的程序翻译为机器语言程序的程序称为汇编程序(Assembler)，又称为汇编器；将高级语言程序翻译为低级语言程序的程序称为编译程序(Compiler)，又称为编译器。编写一个高级语言的编译程序的工作，通常称为对这个语言的实现。

每种高级语言都有一个不大的词汇表(Vocabulary)及构造良好的语法(Syntax)规则和语义(Semantics)解释。规定这些基本属性，便于实现高级语言程序到低级语言程序的机器翻译。高级语言较接近于数学语言和自然语言，它具有直观、自然和易于理解的优点。用高级语言编写的程序易读、易写、易交流、易出版和易存档。由于易理解，使程序员容易编出正确的程

序，以便验证程序的正确性，发现错误后也容易修改。因此，用高级语言开发软件的成本比用低级语言低得多。今天，绝大多数的软件都是用高级语言开发的，因此，高级语言是软件开发最重要的工具。

由于高级语言独立于机器，用高级语言编写的程序很容易从一种机器应用到另一种机器上，因而具有较好的可移植性(Portability)。

高级语言至今还没有完全取代低级语言，在一些场合还必须使用机器语言或汇编语言，例如编译程序的目标程序和各种子程序，以及实时应用系统中要求快速执行的代码段等。但是，随着功能强大且具有高级语言和汇编语言特性的 C 语言的出现，使应用汇编语言的人越来越少。

人们在进行科学研究的过程中，总是对具体现象和事物进行观察、分析和综合，以发现它们的重要性质和特征，建立相应的模型。这种通过观察、分析和综合建立模型的过程称为抽象(Abstract)。利用抽象模型，人们可以把注意力集中在有关的性质和特征上，忽略那些不相干的因素。本书在后面的讨论中，大量使用抽象的方法阐述程序设计语言的概念和结构，然后以各种语言中的具体实例来说明这些概念和结构，从而教会读者如何去设计一个程序设计语言。事实上，程序设计语言中处处都使用了抽象概念，例如变量(Variable)是存储单元(Memory Cell，Memory Location)的抽象；子程序(Subroutine 或 Subprogram)是一段多处重复执行的程序段的抽象等。

在此，我们讨论的对象是高级语言，接下来利用抽象的方法讨论高级语言具有的共性概念和结构以及它们的属性。为了叙述简洁，在不引起混淆的情况下，以下将高级语言简称为语言。

一种语言涉及设计者、实现者和使用者，有了设计者和实现者，才可能有使用者。读者在中学或进入大学后，已经使用过这种或那种程序设计语言，也就是说，已经是使用者。本书的目标是引导读者成为语言的设计者和实现者。由于教学学时及篇幅的限制，本书仅给出入门知识和技术，读者如果要真正设计或实现一个语言，尚需查阅相关的文献资料，建议感兴趣的读者阅读参考文献[59]。通过本书的学习，读者可以提高鉴赏和评价语言(或语言设计方案)的能力；了解语言的重要概念、功能和限制，以便具有为某个目的选择一种恰当语言的能力；具有设计一种语言或扩充现有语言的能力；初步具有实现一个语言的能力。最终使读者能够鉴赏、分析、选择、设计和实现程序设计语言。

1.2 强制式语言

通常的高级语言又称为强制式语言(Imperative Language)，本书主要讨论强制式语言的设计和实现。

1.2.1 程序设计语言的分类

语言的分类没有一个统一的标准，通常按不同的尺度有不同的分类方法和结果。例如，按语言设计的理论基础来分类，可分为 4 类语言，即强制式语言，其基础是冯·诺依曼(Von Neumann)模型；函数式语言(Functional Language)的基础是数学函数；逻辑式语言(Logic Language)的基础是数理逻辑谓词演算；对象式语言(Object-oriented Language)的基础是抽象数据类型(Abstract Data Type)。

人们习惯上按语言的发展历程来对语言进行分类。

1. 第一代语言

第一代语言(First-generation Language)通常称为机器语言,它与机器孪生。实际上,它完全依赖于机器的指令系统(Instruction System),以二进制代码表示。这类语言的程序既难编写,又难读懂。

2. 第二代语言

第二代语言(Second-generation Language)通常称为汇编语言,它将机器语言符号化,用符号来代表机器语言的某些属性。例如,用符号名来代表机器语言的地址码。这样可以帮助程序员记忆,摆脱使用二进制代码的烦恼,提高了程序的可写性和可读性。

不同的机器有不同的机器语言和汇编语言,通常人们又把它们称为与机器有关的语言,或面向机器的语言。

3. 第三代语言

第三代语言(Third-generation Language)通常是指高级语言,这类语言的设计基础与冯·诺依曼体系结构有关。高级语言程序按语句顺序执行,因此又称为面向语句的语言(Sentence-oriented Language)。通常,每条语句对应机器的一组命令,因此又称命令式语言(Order Language)。用这类语言编写的程序,实际上是描述对问题求解的计算过程,因此也有人称它为过程式语言(Procedure Language)。

这类语言书写自然,具有更好的可读性、可写性和可修改性(Modifiability),读者使用过的语言大多是这种高级语言。高级语言程序就是要告诉计算机"做什么"和"怎么做"。

4. 第四代语言

第四代语言(Fourth-generation Language)是说明性语言(Declaration Language),它只需要告诉(说明)计算机"做什么",不必告诉计算机"怎么做";也就是说不需要描述计算过程,系统就能自动完成所需要做的工作。所以,这类语言又称为超高级语言或甚高级语言(Very-high-level Language),典型的例子是 SQL 语言。

5. 新一代语言

另一类不同风格的语言,如函数式和逻辑式语言,它们的理论基础和程序设计风格均不同于高级程序设计语言。它们不适合称为第五代语言或第六代语言,因此,语言学家把它们称为新一代程序设计语言。

1.2.2 冯· 诺依曼体系结构

当今的计算机模型是由数学家冯·诺依曼提出来的,我们称为冯·诺依曼模型(Von Neumann Model)或冯·诺依曼机(Von Neumann Machine)。直到今天,几乎所有的计算机都是沿用这一模型设计的。1978 年,巴科斯(Backus)在获得图灵奖的颁奖大会上发表演说,批判了冯·诺依曼的体系结构和程序设计风格,称这种结构和风格影响了计算机系统的执行效率,提出了函数式程序设计风格,并发表了 FP 和 FFP 语言。今天,人们越来越多地强调使用并行体系结构和并行程序设计,以提高计算机的执行效率。

下面讨论冯·诺依曼体系结构和它对高级语言的影响。

冯·诺依曼机的概念基于以下思想:一个存储器(用来存放指令和数据),一个控制器和一个处理器(控制器负责从存储器中逐条取出指令,处理器通过算术或逻辑操作来处理数据),最后的处理结果还必须送回存储器中。我们可以把这些特点归结为以下 4 个方面。

(1) 数据和指令以二进制形式存储(数据和指令在外形上没有什么区别,但每位二进制数字有不同的含义)。

(2) "存储程序"方式工作(事先编好程序,执行之前先将程序存放到存储器某个可知的地方)。

(3) 程序顺序执行(可以强行改变执行顺序)。

(4) 存储器的内容可以被修改(存储器的某个单元一旦放入新的数据,则该单元原来的数据立即消失,且被新数据代替)。

冯· 诺依曼体系结构的作用体现在命令式语言的下述三大特性上。

(1) 变量　存储器由大量存储单元组成,数据就存放在这些单元中,汇编语言通过对存储单元的命名来访问数据。在命令式语言中,存储单元及其名称由变量的概念来代替。变量代表一个(或一组)已命名的存储单元,存储单元可存放变量的值(Value),变量的值可以被修改;也正是这种修改,产生了副作用(Side Effect)问题(参见 3.3.1 节)。

(2) 赋值　使用存储单元概念的另一个后果是每个计算结果都必须存储,即赋值于某个存储单元,从而改变该单元的值。

(3) 重复　指令按顺序执行,指令存储在有限的存储器中;要完成复杂的计算,有效的办法就是重复执行某些指令序列。

1.2.3　绑定和绑定时间

一个对象(或事物)与其各种属性建立起某种联系的过程称为绑定(Binding)。这种联系的建立,实际上就是建立了某种约束。绑定这个词是由英文 Binding 音译过来的,过去也曾翻译成"联编"、"汇集"、"拼接"或"约束"等。现在之所以选定"绑定"这个词,除了它能形象地表达上述过程外,它还与英文读音一致。

一个程序往往要涉及若干实体,如变量、子程序和语句等。实体具有某些特性,这些特性称为实体的属性(Attribute)。变量的属性有名字(Name)、类型(Type)和保留其值的存储区等。子程序的属性有名字、某些类型的形参(Formal Parameter)和某种参数传递方式的约定等。语句的属性是与之相关的一系列动作。在处理实体之前,必须将实体与相关的属性联系起来(即绑定)。每个实体的绑定信息来源于所谓的描述符(Descriptor)。描述符实际上是各种形式的表格的统称(抽象),用来存放实体的属性。例如,程序员用类型说明语句来描述变量的类型属性,编译时将它存放在符号表(Symbol Table)中;程序员用数组说明语句来描述一个数组的属性,编译时将这些属性存放在一个专门设计的表格中,这个表格称为数组描述符,又称内情向量(Dope Vector)。

对于计算机科学来说,绑定是一个随处遇到且重复使用的重要概念,借助于它可以阐明许多其他概念。把对象(实体)与它的某个属性联系起来的时刻称为绑定时间(Binding Time)。一旦把某种属性与一个实体绑定,这种约束关系就一直存在下去,直到对这一实体的另一次绑定实现,该属性的约束才会改变。

某些属性可能在语言定义时绑定,例如,FORTRAN 语言中的**INTEGER** 类型,在语言定义的说明中就绑定了,它由语言编译器来确定这个类型所包含的值的集合。Pascal 语言中允

许重新定义**integer**类型，因此 integer 类型在编译时才能绑定一个具体表示。若一个绑定在运行之前(即编译时)完成，且在运行时不会改变，则称为静态绑定(Static Binding)。若一个绑定在运行时完成(此后可能在运行过程中被改变)，则称为动态绑定(Dynamic Binding)。

今后讲到的许多特性，有的是在编译时所具有的，有的是在运行时所具有的，凡是在编译时确定的特性均称为静态的(Static)；凡是在运行时确定的特性均称为动态的(Dynamic)。

1.2.4 变量

强制式语言最重要的概念之一是变量，它是一个抽象概念，是对存储单元的抽象。如前所述，冯·诺依曼机基于存储单元组成的主存储器(Main Memory)概念，它的每个存储单元用地址来标识，可以对它进行读或写操作。写操作就是指修改存储单元的值，即以一个新值代替原来的值。语言中引入变量的概念，实质上是对一个(或若干个)存储单元的抽象，赋值(Assignment)语句则是对修改存储单元内容的抽象。

变量用名字来标识，此外它还有 4 个属性：作用域(Scope)、生存期(Lifetime)、值和类型。变量可以不具有名字，这类变量称为匿名变量(Anonymous Variable)。下面将讨论上述 4 个属性，以及它们在不同语言中所采用的绑定策略。

1. 变量的作用域

变量的作用域是指可访问该变量的程序范围。在作用域内，变量是可控制的(Manipulable)。变量可以被静态地或动态地绑定于某个程序范围。在作用域内变量是可见的(Visible)，在作用域外变量是不可见的(Invisible)。按照程序的语法结构定义变量的作用域的方法，称为静态作用域绑定(Static Scope Binding)。这时，对变量的每次引用都静态地绑定于一个实际(隐式或显式)的变量说明。大多数传统语言采用静态作用域绑定规则。有的语言在程序执行中动态地定义变量的作用域，这种情况称为动态作用域绑定(Dynamic Scope Binding)。每个变量说明延伸其作用域到它后面的所有指令(语句)，直到遇到一个同名变量的新说明为止。APL，LISP 和 SNOBL4 语言是采用动态作用域规则的语言。

动态作用域规则很容易实现，但掌握这类语言的程序设计比较困难，实现的有效性也偏低。对于动态作用域语言，给定变量绑定于特定说明之后程序执行到的某个特定点，因为其不能静态确定，所以程序很难读懂。

2. 变量的生存期

一个存储区绑定于一个变量的时间区间称为变量的生存期。这个存储区用来保存变量的值。我们将使用术语“数据对象”(Data Object)，或简称“对象”(Object)来同时表示存储区和它保存的值。

变量获得存储区的活动称为分配(Allocation)。某些语言在运行前进行分配，这类分配称为静态分配(Static Allocation)，如 FORTRAN 语言；某些语言在运行时进行分配，这类分配称为动态分配(Dynamic Allocation)，如 C、C++语言。动态分配可以通过两种途径来实现：或者程序员用相关的语句显式提出请求，如 C++语言通过 new 进行；或者在进入变量的作用域时隐式自动进行分配，如 C 语言的活动记录分配。采用什么样的分配，这要看语言是如何规定的。

变量所分配的存储单元的个数，称为变量的长度(Length)。

3. 变量的值

变量在生存期内绑定于一个存储区，该存储区中每个存储单元的内容是以二进制编码方

式表示的变量值,并绑定于变量。编码表示按变量所绑定的类型来进行解释。

在某些语言中,变量的值可能是指向某个对象的指针(Pointer),若这个对象的值也是指针,那么,可能形成一个引用链(Reference Chain),这个引用链通常称为访问路径(Access Path)。

若两个变量都有一条访问路径指向同一对象,那么,这两个变量共享(Share)一个对象。经由某个访问路径修改一个共享对象的值时,这种修改能被所有共享这个对象的访问路径获知。多变量共享一个对象,可以节省存储空间。但是,由于一个变量的值被修改,就造成所有共享这个对象的变量的值都被修改,使程序很难读懂。

特别地,访问匿名变量的基本方法是通过访问路径来实现的。

变量的值在程序运行时可以通过赋值操作来修改,因此,变量与它的值的绑定是动态的。一个赋值操作,例如

```
b: = a
```

将变量 a 绑定的值复制到变量 b 绑定的存储区内,从而修改变量 b 绑定的值,以一个新值(a 的值)来代替 b 原来的值。

然而,有的语言允许变量与它的值一旦绑定完成就被冻结(Frozen),不能再修改。例如,在 Pascal 语言中,符号常数语句定义为

```
const pi = 3.1416
```

在 ALGOL 68 中,语句

```
real pi = 3.1416
```

定义了 pi 绑定于值 3.1416。在表达式中使用值 3.1416 可写成

```
circumference : = 2 * pi * radius
```

式中的 pi 具有它所绑定的值 3.1416 。这个绑定在整个程序执行过程中不能改变,即不能向 pi 赋新值。显然,语句

```
pi: = 2 * pi
```

是错误的。Pascal 语言中的符号常数(Symbolic Constant)的值可以是一个数,也可以是一个字符串,这类变量在编译时即可完成对值的绑定。同时,编译程序可以在编译过程中合法地以这个值去替代程序中出现的相应符号名。ALGOL 68 的符号常数还可以定义为

```
real pi: = 3.1416 + x
```

它允许将值定义成变量(或表达式中含变量),由于有变量,它只能在程序运行中待这些变量建立时才能完成绑定。

对一般变量而言,当它建立时,才会获得所分配的存储区,同时完成变量与存储区的绑定。此时,该变量绑定的值是什么呢? 这是变量值的初始化问题,这个问题十分微妙。例如,程序段

```
procedure
    integer x,y;
    x: = y + 3;
```

中的语句

```
integer x,y;
```

建立了两个整型变量，允许执行到该过程时，变量 x 和 y 绑定于不同的两个存储单元，但它们绑定的是什么整数值并不确定，原因是未对变量 x 和 y 赋初值。当执行到语句

```
x: = y + 3;
```

时 y 绑定什么值是不明确的，即未对 y 赋初值，因而计算的结果 x 绑定什么值也不明确。在上述程序段中未对 y 赋初值就引用了它，程序可能出错。因此，在程序设计中，读者应当遵从对变量“先赋初值后引用”的原则。

虽然有许多方法可以解决“初值问题”，但遗憾的是，通过语言定义来解决这个问题的大多数尝试都不太成功，因为同一语言的不同实现可能采用不同的方法。例如，FORTRAN 语言定义了一个初值语句(Initial Value Statement)，不同的编译程序采用不同的方法来实现，程序员用起来很不方便。要在语言定义中设定一种方式，实现对所有变量初始化是十分困难的。

最简单也是最常用的解决办法是，在语言设计时，忽略初始化问题，在这种情况下，一旦存储区绑定于某个变量，该存储区当前的内容就是该变量绑定的初值。实际上，这个值是随机的位串。类似的方法还有，规定一个非初始化值(Uninitialized Value)，由编译程序在把某存储区分配给变量时，将这个特殊的值赋给每个已分配的存储单元。当程序运行时，每引用一个变量，先检查引用单元的值是否是这个特殊的非初始化值，若是，则出现错误，由系统报告这一错误，这种方法可以彻底解决非初始化问题，但执行效率较低。

有些实现方法提供了定义初值的强制手段，例如，若定义整型变量，初值强制置 0，若定义字符串变量，初值强制置空串。总之，程序员在使用一个语言编程时，一定要注意初值问题。

4. 变量的类型

变量的类型可以看成与变量相关联的值的类，以及对这些值进行的操作(例如，整数加、浮点数加、建立、存取和修改等操作)的说明。类型也可用来解释变量绑定的存储区的内容(二进制位串)的意义。

根据上述对类型的定义，在定义语言时，类型名通常绑定于某一个值类和某一组操作。例如，布尔类型**boolean** 绑定于值**true** 和**false**，以及操作**and**，**or** 和**not**。

语言实现时，值和操作绑定于某种机器的二进制代码表示。例如，**false** 可以绑定于位串 00000000，**true** 绑定于位串 11111111。**and**，**or** 和**not** 操作可以通过表示布尔量位串操作的机器指令来实现。

在某些语言中，程序员可以用类型说明方式来定义新类型。例如，Pascal 语言的语句

```
type t = array[1..10] of boolean
```

在编译时就能建立一个名为 t 的类型，并使它绑定于一个实现(即由 10 个布尔值组成的数组，借助于下标 1～10 可访问每个数组元素)。类型 t 继承了它所代表的数据结构(布尔数组)的所有操作，用数组内的下标能够读取和修改类型 t 的对象的每个分量(元素)。

变量可以静态或动态地绑定于类型，大多数传统语言都采用静态绑定。例如，FORTRAN，ALGOL 60，COBOL，Pascal，ALGOL 68，SIMULA 67，CLU，C 和 Ada 语言等。在这些语言中，变量和它的类型定义之间的绑定通常都是由显式的变量说明来规定的。例如，语句

```
var x,y : integer;
    z : boolean;
```

将变量 x 和 y 绑定为整型，将变量 z 绑定为布尔型。然而，在某些语言中，例如 FORTRAN 语言允许隐式说明并绑定变量的类型。一个未被说明的新变量，它的第一次出现即可

根据变量名的第一个字母来绑定该变量是整型还是实型。语言设计时提出 I-N 隐式说明规则,以 I 到 N 开头的变量为整型,其他字母开头的变量为实型。这种隐式说明方式的优点是,可以省去显式类型说明语句,写程序要方便些。然而,它的缺点是,不利于编译时检查拼写错误。例如,有 FORTRAN 程序段:

```
INTEGER I,J,ALPHA
ALPHA = 5
ALPHA = ALPH + I
```

其中,ALPH 是一个新变量还是变量 ALPHA 的拼写错误,编译程序无法辨别,它总是把 ALPH当作一个新变量来处理,这个新变量隐式说明为实型。

隐式说明的缺点不是语义上的问题,单就变量的类型而言,Pascal 语言和 FORTRAN 语言中变量的类型在语义上是等效的,两者均是在定义或编译时绑定于变量,它们都是静态绑定。不同的是,FORTRAN 语言具有确定的默认(隐式)类型说明规则,两种语言的绑定时间在某种意义下是相同的。

APL 和 SNOBOL4 语言的类型绑定是动态的,程序编译时无法确定变量的类型,只有在程序运行到某个语句时,才能确定变量的类型。例如,在 APL 语言中,一个变量名可以在执行期间的不同执行点分别表示一个简单变量、一个一维数组、一个多维数组或一个标号。实际上,APL 语言变量的类型不是显式说明的,而是在程序执行时隐式说明的,且动态变化。例如,APL 程序段

```
⋮
A←5
⋮
→A
A←1 2 51 0
A←0
A[2:3]←5
⋮
```

当执行到语句

```
A←5
```

时,A 成为具有值 5 的整型变量。若其后执行到语句

```
→A
```

时,把 A 处理成标号变量,执行这条语句的意义是把控制转移到 A 绑定的值所标记的那个语句,若 A 的值是 5,则转移到标号为 5 的语句执行。当执行到语句

```
A←1 2 51 0
```

时,使 A 成为具有 4 个元素,且其值分别为 1,2,51 和 0 的一维数组,隐含的下标下界为 1。

动态绑定类型对建立和操作数据提供了很大的灵活性,但也影响程序的编写、检查和实现的效率。因为在语句中出现的变量类型,在编译时不能确定,它依赖于给定程序的执行路径,所以程序的可读性差。在上述程序片段中,语句

```
A[2;3]←5
```

的意义是对一个二维数组中位于第 2 行第 3 列的元素赋值为 5，当程序执行到这个语句时，A 应该已经绑定一个合适的二维数组，但在上述程序中，A 此时绑定的类型是整型，其值为 0，因此这个语句是错误的。而这个错误要执行到这个语句时才能发现，这就需要动态类型检查来查证上述错误。采用动态类型检查方式，可在程序运行时查证所使用的每个变量与它的类型是否一致。我们再考查语句

```
A←B+C
```

若运行到这个语句时，B 和 C 都是简单变量，或其中一个是简单变量，语句是正确的；若 B 和 C 是维数相同且大小相同的数组，语句也是正确的。这个语句的执行依赖于 B 和 C，若 B 和 C 都是简单变量，则语句隐含的操作是简单加和赋值；若 B 和 C 是一维数组，则隐含的操作是由加和赋值构成的一个循环，且 A 被绑定成一个一维数组。

上例表明，有关 APL 语言变量的类型信息必须在运行时使用，它不仅用于执行动态类型检查，而且还用于选择执行这个语句的合适操作。为了能在运行时使用运行信息，有关描述符（说明）必须保留到运行时，并且在每次新的绑定完成时，必须对描述符中所保留的类型信息进行相应的修改。对于静态绑定的语言，例如 Pascal 语言的类型信息，在编译时就能确定并加以处理，所以在运行时不必保留类型描述符。

动态绑定的语言实现采用解释（Interpretation）方式处理更合适，因为对于一个不能确定变量类型的表达式，在运行之前没有足够的信息来生成合适的代码。语言实现采用编译还是解释方式，受到变量与类型绑定规则的严重影响。动态绑定的语言是面向解释的语言（Interpretation-oriented Language），静态绑定的语言是面向编译的语言（Compilation-oriented Language）。

动态类型绑定的语言，往往其作用域也是动态绑定的，因此，这类语言又称为动态语言（Dynamic Language）。

1.2.5 虚拟机

我们知道，程序设计语言早期经历了从机器语言发展到汇编语言的时期。机器语言实际上是计算机的指令系统，它是直接由电子线路（硬件）实现的，是实际的机器，我们把它记为 M_1。

为了使电子线路尽量简单，机器采用二进制代码，使用起来非常不便，于是出现了汇编语言。机器不能直接执行用汇编语言编写的程序，必须先将汇编语言程序（L_2）经汇编程序翻译成等效的机器语言程序（L_1）。也就是说，汇编语言程序要在实际机器（M_1）和汇编程序上才能执行，若把汇编语言看成某个虚拟机器（Virtual Machine，Virtual Computer）的机器语言，那么这个虚拟机 M_2 = M_1 + 汇编程序。

后来，程序设计语言又从汇编语言发展到高级语言，用高级语言编写的程序 L_3 必须经过编译程序编译成等效的汇编语言程序 L_2（或机器语言程序 L_1），机器才能执行。也就是说，高级语言要在虚拟机 M_2 和编译程序上才能实现。若把高级语言看成某个虚拟机的机器语言，那么 M_3 = M_2 + 编译程序。

由此可见，虚拟机是由实际机器加软件实现的机器，它可看成一个有别于实际机器的新机

器。若一台实际机器配置上 Pascal 和 FORTRAN 编译程序，对 Pascal 用户来说，这台机器就是以 Pascal 语言为机器语言的虚拟机(Pascal 机)；对 FORTRAN 用户来说，这台机器就是以 FORTRAN 语言为机器语言的虚拟机(FORTRAN 机)，它们按各自的需要去使用，互不干扰。在计算机系统结构课程中，将专门讨论虚拟机的层次分级概念。图 1-1 给出一个网络应用程序的虚拟机层次。

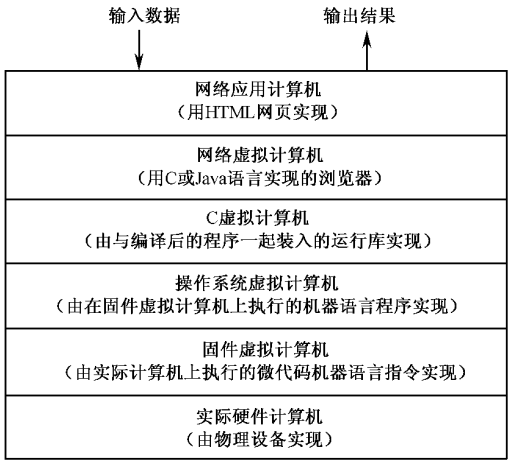

图 1-1　一个网络应用程序的虚拟机层次

这一节我们介绍了强制式语言的一些概念，这些概念在非强制式语言中，有些也适用。几十年来，已经有许多很成熟的强制式语言，图 1-2 给出了主要的强制式语言，以及它们之间的关系。

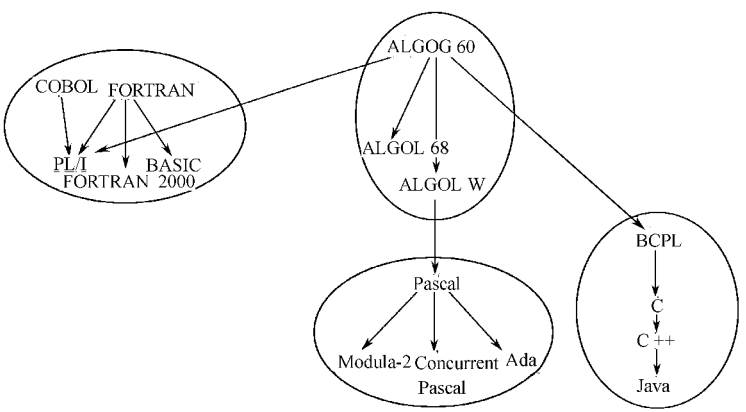

图 1-2　主要的强制式语言及其关系

1.3　程 序 单 元

为了后面讨论程序结构和语言实现问题，在此引入程序单元(Program Unit)和单元实例(Unit Instance)的概念。

在程序设计语言中存在一些实体，例如 FORTRAN 语言的子程序和 ALGOL 60 语言的分程序。它们作为程序执行过程中的独立调用单位，称为程序单元。在不引起混淆的情况下，

简称单元(Unit)。通常,单元可以独立开发,有的语言(如 FORTRAN)允许程序单元独立编译,然后把若干个编译好的单元组合起来,成为一个完整的可执行程序。一个程序中的若干个程序单元在程序运行时依照控制流程逐一被激活(Activation)。

在编译时,一个单元的源程序称为单元表示(Unit Representation);在运行时,一个单元表示由一个代码段(Code Segment)和一个活动记录(Activation Record)组成,此时称单元表示为单元实例。程序单元可以看成一个抽象的概念,程序运行时对它赋予具体的代码段和活动记录,也就构成了一个单元实例。

代码段的内容是单元所具有的指令,这些可执行的指令对程序单元的每一个单元实例都是不变的。所谓活动记录就是包含执行这个单元所必需的信息,以及该单元的局部变量(Local Variable)所绑定的数据对象的存储区。活动记录的内容是可变的。数据对象在活动记录中的相对位置称为位移(Offset),活动记录所占存储单元个数称为活动记录的长度。局部变量在程序单元中是可见的。

程序单元可以命名,也可不命名。Pascal 的过程与函数和 C 语言的函数是命名的程序单元,ALGOL 60 语言的子程序是不命名的程序单元。程序单元不是孤立的,即不一定是一个完全独立的程序。若一个程序单元是子程序,那么它可由别的程序单元通过子程序调用来激活并开始执行,执行后返回调用点,因此返回位置是必须保留的信息。子程序被调用时将建立并激活该子程序的一个实例,其返回地址保存在这个实例的活动记录中。另外,在语言作用域规则允许的前提下,一个程序单元可以引用未被自己说明而由其他单元说明的变量,这种变量称为非局部变量(Nonlocal Variable)。在一个程序中,各个程序单元都可以引用的变量称为全局变量(Global Variable)。

一个程序单元可以引用哪些变量呢?按照上述定义,可以引用局部变量和非局部变量。我们把一个程序单元 U 可以引用的局部变量和非局部变量定义为程序单元 U 的引用环境(Referencing Environment)。局部变量绑定于存储在 U 的当前活动记录中的数据对象,它被称为局部环境(Local Environment)。非局部变量绑定于别的(说明该非局部变量)程序单元的活动记录中的数据对象,它被称为非局部环境(Nonlocal Environment)。显然,对程序单元 U 来说,引用环境中的变量是可见和可以访问的,其他变量均是不可见和不可以访问的。若一个程序单元的引用环境中有两个变量绑定于同一个数据对象,则称这些变量具有别名(Alias)。当对绑定的一个非局部变量进行修改时,将产生副作用(参见 3.3.1 节)。

程序单元可以被递归地激活。当一个程序单元自己调用自己时,产生直接递归(Direct Recursion)。当一个程序单元调用别的程序单元,再由别的程序单元调用这个程序单元时,产生间接递归(Indirect Recursion)。当一个程序单元被递归激活,即它的上次活动尚未终止,又再次被激活,此时它的前一个活动记录尚未释放而又产生一个新的活动记录。按照程序单元实例的定义可知,一个程序单元可能具有多个实例。同一程序单元的不同实例的代码段是相同的,所不同的仅仅是活动记录。因此,在递归激活的情况下,活动记录与它的代码段之间的绑定必须是动态的。每次激活一个程序单元,必须完成活动记录与其代码段之间的绑定,形成一个单元实例。

某些语言,例如 FORTRAN,不支持单元的递归激活,因此这类语言的单元实例最多只能有一个,即只有一个代码段和一个活动记录。这类语言程序的单元实例代码段与活动记录之间的绑定是静态的。单元局部变量的数据对象的初始化(建立)可以在程序执行之前完成,即

可静态实现分配。这类语言又称静态语言(Static Language)(参见 11.2 节)。

1.4 程序设计语言发展简介

程序设计语言在短短的 50 余年里发生了很大的变化,得到了很大的发展。风格迥异的上千种语言成为开发计算机软件的强有力工具。

从今天软件开发过程的观点来看,最早的软件开发仅有编码阶段。早期的计算机仅用于科学计算,其特点是:计算复杂,数据量少。那时一个任务完全可以由一个人完成程序的编写并由计算机加以实现。程序员所求解的问题(例如微分方程求解)都是他能理解的问题,他只需从数学家那里找到相应的数学求解算法就可以编程实现,没有必要进行需求分析或设计规范说明,其程序的维护也非常简单,只需要改正一些编程错误。当时的语言着眼于支持程序员个人,按照今天的观点来说,程序员所进行的工作仅仅是简单的应用。

随着计算机技术的发展,人们期望计算机的应用更加广泛,求解任务变得越来越复杂,计算机处于一个难于理解和更加复杂的环境之中。这时只靠程序员个人在头脑中进行需求分析和设计已不适应复杂任务的要求,它需要一个程序员"队伍"来共同进行需求分析和设计,并协同工作来完成预定的任务。另外,开发一个大系统需要花费大量的人力和物力,所以在建立新系统时,出于经济上的考虑,也不能完全把旧系统抛开,而是将它修改、补充和完善,以满足新的要求。因而程序的可维护性(Maintainability)就成为一个重要的现实问题。

随着系统变得越来越庞大,越来越复杂,系统的可靠性(Reliability)也成为另一个重要的现实问题。例如,若把一个不可靠的系统用来控制核电站,可能出现灾难性的后果。

除此之外,程序设计语言一般是针对某类计算机应用而设计的,在使用过程中可能发现它的不足(局限性),或者需要对其提出更高的要求并扩充其应用能力,这需要设计新的语言来适应这种变化,程序设计语言也由此得到发展。

在此我们关注的是高级程序设计语言,下面将沿着上述发展过程展开讨论。由于篇幅的限制,讨论中将只涉及那些对程序设计产生过一定影响的高级程序设计语言。

1.4.1 早期的高级语言

第一个高级语言要追溯到 20 世纪 50 年代,那时计算机非常昂贵而又稀少,有效地使用计算机是人们重视的问题。另一方面,机器的执行效率也是人们追求的目标。为此人们设计了高级语言,用以方便地表达用户的需求。然而高级语言程序只有经过编译,机器才能执行,而编译要占去一部分机器时间。更为严重的是,在当时编译生成的目标程序的执行效率要低于人工直接编制的程序的执行效率。因此,那时设计的高级语言以冯·诺依曼模型为基础,并且特别强调执行效率。实践证明,高级语言是有效地使用机器与机器执行效率之间的一个很好的折中。

这一历史时期最重要而又最具代表性的语言是 FORTRAN(FORmula TRANslation),ALGOL 60(ALGOrithmic Language 60)和 COBOL(COmmon Business Oriented Language)。使用这些语言可以用接近人们习惯的语言编制程序,大大提高了机器的使用效率。

1. FORTRAN 语言

FORTRAN 是 1954 年设计并于 1957 年在 IBM 704 机上实现的第一个高级语言。1958 年又实现了 FORTRAN Ⅱ,其后不久实现了 FORTRAN Ⅲ,几年之后出现 FORTRAN

Ⅳ。1966 年,美国标准学会(American Standards Association)公布了 ASA FORTRAN,后来又产生了 FORTRAN 77 和 FORTRAN 90,今天,已经有了 FORTRAN 2003 和 FORTRAN 2007。

FORTRAN 语言完成了第一个编译器,它的文本编辑器用于程序的创建。FORTRAN 语言将主程序、子程序和函数看成独立的模块进行编译,它没有递归调用。各个模块先要编译成可执行形式,再通过连接器(Linker)将主程序、子程序、函数和运行库连接成一个可执行程序,然后按照控制流程(执行步骤)执行这个可执行程序。

FORTRAN 是典型的强调执行效率的语言,它结构简单,不强调程序设计技巧,达到了提高执行效率的目标。

FORTRAN 语言主要用于科学计算。作为数值计算工具,它特别适合解决数据量少而计算复杂的问题。它提供 4 种数值型数据类型(整数类型、实数类型、复数类型和双精度实数类型)及布尔数据(逻辑类型)、数组、字符串和文件等数据类型,这些类型使得它可以在编译时静态分配存储空间。FORTRAN 语言中引入了最原始的语言概念,如将变量作为存储单元的抽象,语句顺序执行,goto 语句强制改变语句的执行顺序,通过全局(COMMON)环境共享数据对象(实现模块之间的通信)等。语法采用自然语言(英语)描述。

随着计算机的发展,FORTRAN 语言已经吸取了许多后来出现的语言的优点,得到了很大的改进和发展。

2. ALGOL 60 语言

由于 FORTRAN 语言的成功,使欧洲人担心 IBM 会统治业界,因此,德国应用数学协会 GAMM 组织了一个工作组来设计一种通用语言。在美国,计算机协会也组织了类似的工作组,后来两个工作组合并,在彼得·诺尔(Peter Naur)的领导下开发出国际算法语言 IAL ,并在 1958 年以 ALGOL 58 命名,1960 年公布了著名的算法语言 ALGOL 60, 1963 年公布了 ALGOL 60 的修订版。

ALGOL 60 语言在美国商业界未能取得成功应用,因为美国商业界已经习惯了应用 FORTRAN 语言,但它在欧洲还是取得了一些成就。

巴科斯(Backus)是定义 ALGOL 报告的编辑,他用乔姆斯基(Chomsky)开发的语法(Grammar ,参见 4.2 节)分类中的上下文无关文法来描述 ALGOL 语言的语法,成功地将形式语言理论引入程序设计语言中,后来大多数高级语言都采用这种方法来定义语法。ALGOL 60报告是以诺尔名义发表的。由于巴科斯和诺尔在发展 ALGOL 中所起的巨大作用,这一方法被称为巴科斯-诺尔范式(BNF)。这种形式语言描述方法为语言定义开辟了新纪元,提供了精确的语法定义方法,从而减少了用自然语言说明的二义性,使得编程的语法错误大为减少。有趣的是,科学家们在严密定义 ALGOL 60 的语法和成分时,竟忽略了“程序”的定义,这个疏忽在 ALGOL 60 正式公布之后才被发现,并在 1963 年的修订版中补充定义。ALGOL 60 引入了子程序(Block)结构、递归过程和动态数组等新概念。

3. COBOL 语言

1959 年,美国国防部为了开发一种通用的商业语言(CBL)召开了一个专门会议,希望该语言尽量使用自然语言(英语)。会上意见分歧较大,会后专门组织了一个短期协会(Short Range Committee)来快速开发这一语言。1960 年,公布了命名为 COBOL 的商用语言,完成

了编译实现，并于 1961 年和 1962 年进行了修订。1968 年公布了 ANSI 标准，1974 年又公布了 ANSI 新标准。

由于 COBOL 语言使用近似于自然语言的方式编程，其可读性比较强，但它还是保留了形式语法，程序员不经过专门培训要完成编程还是比较困难的。COBOL 语言使用了大量不同的数据表示，以及语句中包含大量的不同变体选择，编译工作是相当复杂的。早期的编译器执行起来都很慢，但随着编译技术的发展，新近开发的编译器速度较快，产生的目标程序执行起来也更加有效。

COBOL 语言成功地将若干新概念引入程序设计语言中，除了上面提到的类自然语言编程外，还引入了文件和数据描述、变体记录等概念。

COBOL 语言的出现，打破了计算机应用领域仅限于科学数值计算的局面，开始应用于各种事务处理领域，特别是数据处理（计算简单、数据量特别大）领域，为计算机的应用和发展开辟了新纪元。COBOL 语言一直用得很好，我国早些年引进的应用软件，大多数是用 COBOL 语言开发的，只是后来才被 C 语言和其他一些语言所代替。

1.4.2 早期语言的发展阶段

随着计算机的发展和应用的日益广泛，实现的效率已经不再是人们唯一的追求目标。早在 20 世纪 60 年代就出现了一些基于数学原则的机器计算表示法语言，它们基于数学函数和函数作用，而不是基于冯·诺依曼模型的。这些语言的代表是 LISP，APL（A Programming Language）和 SNOBOL4。

1. LISP 语言

1960 年，John McCarthy 在麻省理工学院（MIT）设计和实现了 LISP 语言，它的设计基于函数和函数作用的数学概念，它奠定了函数式（或作用式）语言风格的基础。由于没有适用于函数式语言的计算机体系结构问世，因此它不得不在冯·诺依曼体系结构的计算机上执行，其执行效率低，执行速度慢。

通常，LISP 程序不经过编译而是通过解释来执行。人们为了提高执行速度，各种实现对 LISP 进行了不同的改造，出现了许多不同的版本。1981 年 4 月，各个不同的 LISP 学派召开了一次会议，试图将各种版本统一起来，于是出现了通用 LISP（Common LISP）语言。

由于 LISP 语言特有的数学特性，使它一出现就在计算机科学的研究中得到大量应用，特别是在人工智能领域。例如，在机器人、自然语言理解、定理证明和智能系统等研究领域应用非常广泛。

纯 LISP 语言从变量值可以被修改、赋值语句、goto 语句等冯·诺依曼体系结构概念中解放出来。它主要用来处理符号表达式，并引入了许多新概念。例如，语言有统一的数据结构（表）；数据和程序有统一的表示方法（S 表达式），其中包括递归表达式、前缀表达式，并将递归作为基本控制结构等。LISP 语言的语义很容易用 LISP 程序描述，用 LISP 语言编写的函数 EVAL，可用来计算任何给定的 LISP 表达式，它是 LISP 语言的语义定义。

2. APL 语言

20 世纪 60 年代，由柯沃尔森研制并实现了 APL 语言。APL 语言表达式简洁，操作符丰富，采用非标准字符集，程序具有单行（One-Liner）结构特点（参见 1.2.4 节）。