Lewolf
保留版权

第一篇 理解面向状态

第一章 程序设计的发展

从世界上第一台计算机问世到现在，已经有 50 多年的历史了，计算机程序设计也经历了从简单到复杂，从功能单一的程序设计到大型软件开发的过程。在这期间，计算机的体积是越来越小，而软件的“体积”却是越来越大，软件实现的功能也越来越强大。计算机应用的核心逐渐由硬件向软件过渡已经成为一个不争的事实。

本章将从计算机语言和程序设计的发展讲起，简单介绍在这半个多世纪计算机语言演变的过程，帮助读者更好的理解本书将要讲述的面向状态的编程方法。

1 机器语言及汇编语言

在最初期，程序设计用的使机器语言，即将程序中机器执行的“0”“1”代码按照相应的功能排成一个序列，而且那时候没有现在的硬盘、显示器、键盘等设备，这些“0”和“1”组成的代码序列需要手工编制在纸带等存储设备中，需要运行时，将纸带放入纸带读写器中，计算机依次读入信息，运算完成后在将计算结果打在纸带上。

可以想象，这样的编程过程是非常庞大、非常繁琐，调试和修改也都非常的复杂，当然计算机的功能和用途也受到很大的限制。而且，机器语言最大的问题是其指令是针对机器而言的，对于不同的计算机，即使执行同一个操作，其指令代码也是不相同的，或者说，不同的计算机有不同的指令系统。例如有的计算机的指令长度为 8 位，有的机器则为 16 位或 32 位，显然，第一个机器上的程序指令代码拿到第二个计算机上是不能够执行，因此机器语言是依赖于具体计算机硬件，而不是各类计算机通用，它是面向机器的。

之后，随着技术的发展，人们发明了键盘和显示器等设备，同时也寻求更先进的编程语言。由于机器语言指令繁琐难记，人们想到可以用一些特定的符号来代替某些操作。例如用 ADD 代表“加”运算，SUB 代表“减”运算，LD 代表“传送”AX 代表微处理器的寄存器等等，这就是后来的汇编语言。

例如在 IBM 兼容 PC 机中：

```
MOV    AX,A
MOV    BX,B
ADD    AX,BX
```

表示，将 A 和读入 AX 寄存器，将 B 读入 BX 寄存器，然后相加，而把这一段代码翻

译成机器指令，则需要 8 个字节，也就是 64 位“ 0 ”“ 1 ”代码，显然，汇编语言比机器指令容易记也容易学的多，其中 ADD、SUB、LD、AX 等被称作助记符，因此汇编语言也称作“ 符号语言 ”。

但是计算机并不能直接识别和执行符号语言的指令，必须先将这些符号转换成为二进制代码机器才能执行，一般来说一条汇编指令对应的翻译成为一条机器指令，而且这种翻译工作通常是由计算机来完成的，我们把这个翻译程序称作：“ 汇编程序 ”。

一般来说，汇编语言随着不同计算机而有所不同，每一类型的计算机分别有自己的汇编语言，汇编语言也是面向机器的。用汇编语言编写程序，也需要了解计算机内部的结构，对使用者的要求较高，因此也不易推广普及。但是汇编语言程序和机器语言一样，执行速度快、占用存储器空间小。随着计算机技术的发展，除了单板机、操作系统的内核、底层的驱动程序和某些病毒程序之外，一般的应用程序几乎都不再使用汇编语言。

2 高级语言

由于机器语言和汇编语言是面向机器，不同的机器有不同的汇编语言，可移植性差、又难学难懂，可维护性差，而且每一条机器代码都对应一行汇编代码，通常原代码比汇编后的可执行代码体积还要大。因此人们希望能找到一种工具语言，更接近自然语言方式，易学易懂，又容易在其它计算机上使用，这就是后来的高级语言。

高级语言又称作面向过程的语言，面向过程的意思是：程序员不必考虑机器内部构造和特点，只需要针对所要解决问题而编写出相应的过程，不同的计算机会用不同的编译程序将其翻译成为可以运行的机器代码，这个翻译程序就叫做“ 编译器 ”。

由于编译器是一种中间工具，同一个源程序在不同的编译器下面，可能会被编译成不同的机器代码，但是其执行结果是相同的，这正是高级语言和汇编语言所不同的一个特点，例如同样一个标准 C++的程序源代码在 Visual C++和 Borland C++ 或 C++ Builder 下编译的结果可能有所不同，实现效率也有所不同，但是他们的运行状况和执行结果却是完全一样的。因此，我们有理由也有必要选择优秀的或者适合于自己使用的编译器。

目前使用比较多的高级语言有：BASIC、FORTRON、PASCAL、C、C++、ADA、COBOL、LISP 等等。

在这些语言中 CALBOL 是适合一种管理领域应用的高级语言；FORTRON 是出现最早的一种高级语言，适合于数值计算；PASCAL 是出现最早的一种结构化语言，常用于教学中；LISP 和 ADA 基本上是纯粹的面向对象的编程语言，属于人工智能型语言；C 则是上一世纪 80 年代到 90 年代初最流行的编程语言，在系统软件和应用软件领域都有着广泛的应用；C++则是在 C 语言的基础上进行扩充并增加了面向对象的机制而形成的一种面向对象的程序设计语言，同时 C++语言完全兼容 C 语言，很多用 C 语言编写的程序稍加改动或不作该动就可以利用 C++编译器进行编译，因此 C++语言一出现就赢得了广大程序设计者的喜爱，本书就是介绍如何运用 C++编译器的一个家族 C++ Builder 中提供的新的编程方法，来编写更实用、更优秀的程序。

3 结构化程序设计

高级语言的出现克服了汇编语言的许多缺点，当同时也出现了新的问题，由于技术的发展，计算机所能胜任的工作也越来越多，软件的规模也越来越大，流程越来越复杂，在这些复杂的流程中，很难有按顺序一条一条代码往下执行的情况，会出现很多的条件和非条件“跳转”。如下图是一个判断给定年份是否为闰年的一个流程：

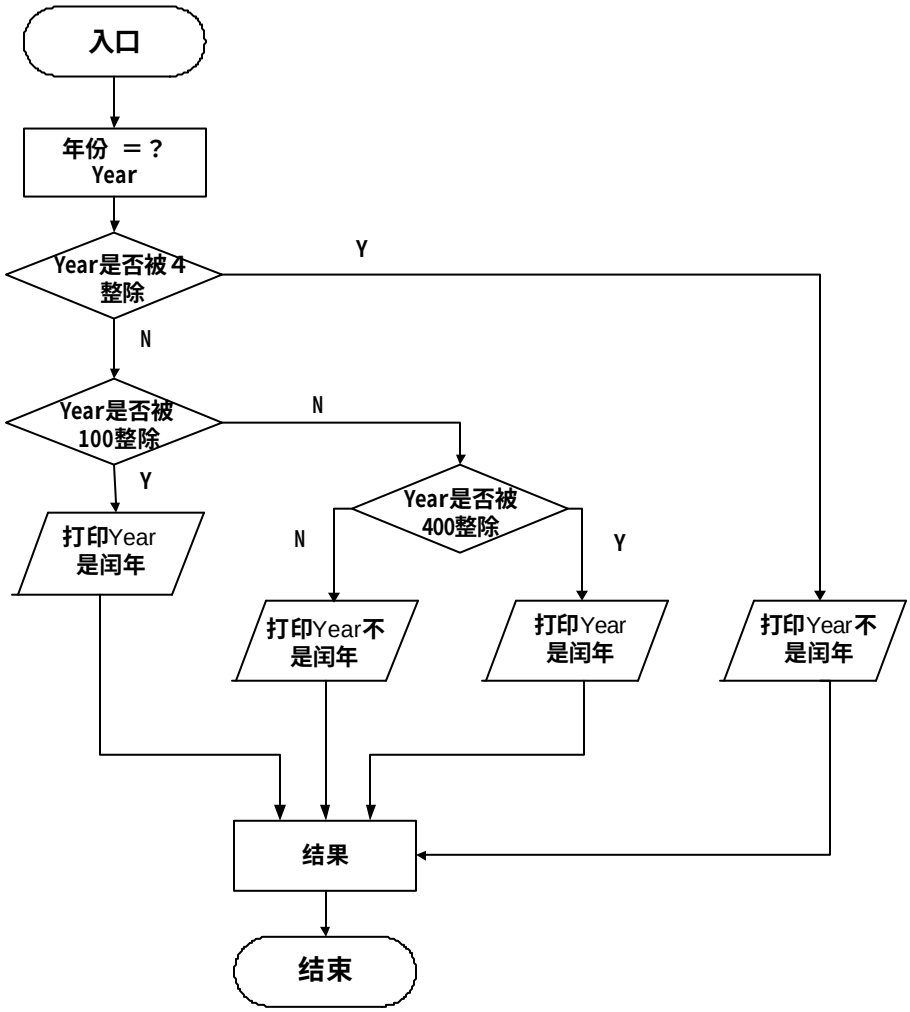


图 1.1 判断 Year 是否为闰年的流程

在这一个短短的流程中就出现了 3、4 次跳转，在一个大型程序中，这种跳转是非常多的，若程序也按照这样的流程去设计，将会混乱不堪难以阅读和理解，最初的 FORTRON 语言中就包含着 GOTO 语句，是从汇编语言的 Jump 语句演化而来的，这个语句允许程序员无条件地、丝毫不受限制地从程序代码中跳转来跳转去，这样编出来的程序就难以阅读、难以修改，从而使软件的开发受到了很大的制约。

这是一个方法论的问题，于是众多的软件工作者便试图从实现方式的改变来解决这种

混乱的局面。

到上一世纪 60 年代，便出现了结构化程序设计。结构化程序设计的设计思想是：自顶向下、逐步细化、模块化设计、结构化编码。其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构；各模块之间的关系尽可能简单，在功能上相对独立；每一模块内部均由几种基本结构组成；其模块化实现的具体方法是使用子程序，因此，在各种高级语言里基本都支持子程序。

1966 年，Bobra 和 Jacopini 提出了三种基本结构，并且现在已经证明任何复杂的算法都可以由这三种基本结构组合而成，这三种基本结构是：顺序结构、选择结构、循环结构，这三种基本结构的共同点是：只有一个入口、只有一个出口、结构内每一部分都有机会被执行到即没有落空代码、机构内不存在“死循环”。

由于模块化的设计和结构化的编码，可以将一个较为复杂的程序系统分解为许多个容易控制和容易处理的子任务，而每一个子任务都是可以独立编程的子程序模块，而每个模块内又都是由若干个基本结构组成，这样程序的代码就非常清晰，界面明确，使用起来也相对简单容易。

这在上一世纪 70 年代到 80 年代各个领域开发软件所都采用的设计方法，但从本质上来讲还是面向过程的程序设计，其数据和过程是相互独立，并没有内在联系，程序员在编程时必须考虑其代码所要处理的数据，以及数据格式，有时候可能还会出现数据和处理程序不匹配的情况，而且设计出来的程序可重用性也差，这是结构化程序设计在所难免的，正是这些问题促使了软件开发技术的进一步发展。

4 面向对象的程序设计

和计算机发展初期相比较，今天的计算机软件已经有了翻天覆地的变化，尤其是图形界面出现以后，计算机软件更是由最初的抽象化变得越来越形象化，程序也逐渐由纯粹的“计算机思维方式”向“人类的思维方式”转变，在软件系统工程中，著名的 Jackson 方法就是要求程序中实现的算法应该尽量和人的思维方式一致，这样，所规定的程序才具有更好的适用性和可维护性。如：

“打印 10 级杨辉三角”这一个简单问题，你可以有两种办法实现：

一．直接打印出来，实现方法如下

```
begin
   print     1
   print     1   1
   print     1   2   1
   print     1   3   3   1

      ……  ……   …
end
```

二．按照杨辉三角的构成规律，让计算机运算：

```
begin
   arrey  Data[10][10]     //定义 10×10 数组
   Data[1] = 1
   do  I = 2  until  I > 10
      do  j = 1  until  j > I
         If  j = 1
           Data[I][j] = 1
         Else If  j = I
            Data[I][j] = 1
         Else
            Data[I][j] = Data[I-1][j-1] + Data[I-1][ j ]
         End If
         print  Data[I][j]
      end do
      print  ↙                 //换行
end do
end
```

在这个例子中，第二种方法比第一种方法更加容易维护，比如：将要求改为打印 20 级杨辉三角，第二种方法只需将数组下标和循环次数改为 20 就可以，而第一种方法最需要重新输入一些代码。而实际的程序中，所用到算法比这个例子复杂得多，即使在设计过程中，也可能多次不断的对代码进行修改，因此，就要求计算机语言尽可能和人的意识及思维方式一致。

面向对象的程序设计思想就是从这里产生的，其主要思想是：将计算机程序设计问题抽象化，建立符合人类思维习惯和认识过程的模型。这种模型在程序中称为对象，程序设计是以对象为核心的。

人类总是将相关性较强的几个物体联系在一起，而抽象的表达为一个新的物体或一个新的类别，这种抽象的认识过程，就可以认为是一种对象的生成。如：由车轮、车厢组成并可以移动的物体，我们称之为车；装有四个车轮、方向盘和汽油发动机的车叫做汽车等等，都是对象的生成过程。可见面向对象是一个不断抽象和延续的过程。

在计算机程序设计过程中，程序员使用面对的可能有很多的数据以及对这些数据进行处理的程序过程，这些过程和数据之间的关系错综复杂，程序员要处理这些复杂的关系往往会消耗很大的精力，面向对象的设计思想就是将关系比较密切的数据和处理过程有效地组织在一起，形成一个相对独立的个体——对象，在对象内部采用数据抽象和信息隐藏的技术，程序员所做的是生成对象和处理对象与对象之间的联系。

面向对象的最大优点是提高了软件的生产率和解决了软件复杂性控制的问题。目前，比较常用的开发软件都是面向对象型，比如：C++的各种编译器，Turbo Pascal、Object

Pascal、Visual C++、Visual Basic、Java 的各种开发工具等等，都属于面向对象的开发工具。

面向对象的程序设计方法本身和结构化程序设计并不矛盾，前者也是在后者的基础上发展而形成了，在一个对象的内部也应该是由许多基本结构组成的。面向对象的程序虽然降低了代码的执行效率，但是却大大提高软件的开发速度，提高了软件的可维护性，使代码共享更加容易。许多编译器和开发工具包都习惯了许多可以给我们利用的基本类库，如：Borland C++的 OWL（Object Windows Library）、Microsoft 的 MFC（Microsoft Foundation Class）以及 AutoCAD 2000 的 Object ARX 类库等。应用这些类库我们就可以很方便地开发出功能非常强大的应用软件。目前，面向对象的程序设计已经成为软件开发的主流。

5 面向状态的程序设计

首先声明的是：面向状态的程序设计并不是一个公认的叫法，很多书籍里都将其归结为面向对象的程序设计，而且试图以面向对象的思想解释整个过程，似乎不尽完善。作者根据多年的 C++使用经验，将其称为面向状态以区别于面向对象，本书将试图从理论的角度对这种新的编程思想加以分析、描述，并结合一些实例分析，希望能对 C++ Builder 的初学者更好的使用该软件有所帮助；对喜欢钻研 C++ Builder 的读者起到抛砖引玉的作用。

面向状态的技术最早是出现在可视化编程工具中，如：Visual Basic、Visual FoxPro、Delphi、C++ Builder、Java Beans 等，基本上都使用了面向状态的技术，ActiveX、Com 等也都是使用了面向状态的技术，应该说是流派比较多。本书主要介绍的是在 Delphi 和 C++ Builder 中广泛使用的 VCL（Visual Component Library）可视化组件技术。

如同当初面向对象的产生过程一样，软件日趋庞大也使程序员的开发过程越来越繁杂，软件维护难度越来越大。尽管面向对象技术对事物的描述已经非常接近人的思维和认识过程，但面向对象只是反映了一个静态的过程。计算机应用软件日益人性化的今天，程序员所面临的最复杂的、最令人头疼的就是人和计算机之间的相互交流，或者叫人机界面的设计。计算机软件是一个运行着不断变化的过程，每时每刻都要和操作者进行信息的交流，而软件内部又要对操作者的输入信息进行正确的处理，同样的一个处理过程，可能会在不同的运行状态被触发或者被调用。要保证这种处理的正确性，采用面向对象的设计思想，显然是不够客观、不够科学的。对于一个软件，人们并不是运行后就置之不理，而是不断的判断程序在做什么？程序已经做了什么？我让程序做什么？我将要让程序做什么……

仍然以车为例：

对于一个汽车，人们显然不是关心它本身是什么样的，而是关心汽车是停着的，还是开动着的，车门关着还是开着，车速是多少……

用面向对象的技术对汽车的描述可以表示如下：

```
Class Car
Begin
  Member：
    Wheel                  //车轮
    Steering wheel         //方向盘
    Engine                 //引擎
    Door                   //车门
      ……     ……
  Method：
    StartCar               //点火
    Run                    //开车
    Accelerate             //加速
    Decelerate             //减速
    CloseDoor              //关门
    OpenDoor               //开门
    ……      ……
End
```

我们假设现在要求将车速变为 60Km/h**，可能会出现的初始状态是：**

一．车速现在为 90Km/h**；**
二．车速现在为 30Km/h**；**
三．车停止，发动机熄火；
四．车停止，发动机熄火，车门未关；

可能还有更多的情况，我们不多举例。那么，针对这四种不同的情况，我们所要执行的操作也是不一样的。

第一种情况是减速；实现过程为：

```
Procedure    Fn1
Begin
   Decelerate          //减速
End
```

第二种情况是加速；

```
Procedure    Fn2
Begin
   Accelerate          //加速
```

```
End
```

第三种情况首先是点火，然后是加速至 60Km/h；

```
Procedure    Fn3
Begin
   StartCar            //点火
   Run                 //启动
   Accelerate          //加速
End
```

而第四种情况是先关车门，然后点火，最后加速至 60Km/h；

```
Procedure   Fn4
Begin
   CloseDoor           //关门
   StartCar            //点火
   Run                 //启动
   Accelerate          //加速
End
```

在这四种情况里，第一、第二种都是改变车速，一个为正方向的，一个为反方向，但都属于同一种操作；第三种情况，多了一个执行过程——点火，也同样包含一个改变车速的操作；第四种情况，多了关车门、点火两个过程，也包含了改变车速的操作。它们的共同点是都包含车速改变的过程，因为这个操作是车速状态改变的核心操作过程。进一步分析归纳可以发现这四种情况也包含了一个共同之处，它们必须有两个共同的约束条件：

一．车门必须是关着的
二．发动机必须处于点火状态

因为第一、第二种情况隐含的满足这两个条件，而第三种情况隐含的满足第二种条件，第四种中情况两个条件都不满足。根据这一特点，我们可以设想建立以下几个机制：

- 将汽车的运行抽象出三个独立的运行状态：车速状态、车门状态、发动机状态。
- 每个状态都和一个操作过程和其相对应。
- 每个状态对应的操作过程都要负责其核心操作。
- 每个状态对应的操作过程必须负责其约束条件的检测和设置。

这样以来问题将会大大简化，程序员不必去考虑在不同的条件下是如何经过不同的过程而达到目的的，只需要关心对象中各个状态之间的约束关系，以及每个状态改变时的核心操作的实现过程就行。

简单讲，这就是面向状态的编程思想，它可以化解数据与数据之间的相关性，将一些复杂的逻辑关系转化成为较为简单的逻辑关系，使程序的设计、修改和维护都相对简单得多。用面向状态的思想描述上面的例子可以表述如下：

```
Class   Car
Begin
  Member：
    Wheel                  //车轮
    Steeringwheel          //方向盘
    Engine                 //引擎
    Door                   //车门
      ……     ……
  Method：
    StartCar               //点火
    ShutDown               //熄火
    Run                    //开车
    Accelerate             //加速
    Decelerate             //减速
    CloseDoor              //关门
    OpenDoor               //开门
    ……      ……
    SetSpeed               //对应车速状态
    SetDoor                //对应车门状态
    SetEngine              //对应发动机状态
    ……      ……
  State：
    Speed                  //车速状态
    IsDoorOpen             //车门状态
    IsTurnOn               //发动机状态
    ……      ……
End

Procedure  SetSpeed(X)
Begin
   IsDoorOpen = False     //约束条件，实际/执行 SetDoor（False）
   IsTurnOn  = True       //约束条件，实际执行 SetEngine（True）
   If  Speed  > X
     Decelerate
   Else If  Speed = 0
```

```
      Run
      Accelerate
    Else
      Accelerate
    End If
End

Procedure  SetDoor (Bool)
Begin
   If IsDoorOpen  != Bool
     If  ……                    //关车门的约束条件，这里没有
        ……
     End If
     If IsDoorOpen = True
       CloseDoor
     Else
       OpenDoor
     End If
   End If
End

Procedure  SetEngine (Bool)
Begin
   If  IsTurnOn != Bool
     If  IsTurnOn = True
       ShutDown
     Else
       StartCar
     End If
   End If
End
```

要求将车速变为 60Km/h**，我们仅仅需要做的是**

```
   Speed = 60          //执行 SetSpeed (60)
```

读者可以尝试使用结构化程序设计方法和面向对象的设计方法将上面这一条执行语句的整个流程画出来，跟上面这一段伪代码的描述比较一下，似乎这个很简单的逻辑用其他设计方法来描述都会变得复杂起来。其根本原因是，编写上面的这段伪代码，你根本不需要思考，因为这就是“现实”，这就是你的思维。是你的思维，当然不用费力的思考。

尽管使用面向状态的方法在建立对象的时候似乎比面向对象的方法要冗长得多，但在

使用的时候却非常方便，简洁得多、直观得多了。而且，这个对象已经包含了目前的所有可能出现的不同使用环境和条件制约关系。从逻辑上也更加符合人类的思维，更加简单明了，修改和维护都相对简单得多。

6 小结

本章简单的介绍了计算机程序设计的发展过程，并且引入了面向状态的概念，通过一个形象的实例读为者建立一个面向相状态的感性认识。希望能对部分读者阅读后面的内容有所帮助，

跟任何事物一样，软件设计的发展也是循序渐进的，新的思想、新的方法总是在以前的基础上建立的，都是对以前的方法改进、提高和扩充。面向对象的语言基本上都需要结构化程序设计的支持，同样的面向状态的开发工具本身就应该是面向对象的，迄今为止，Bobra 和 Jacopini 提出的三种基本结构仍然是程序设计中亘古不变的真理，而且任何语言编写的程序最终都是要机器代码来实现的。你完全可以使用 C++ Builder 而不使用面向状态的技术，当然没有人会去怪你，你的上司也不会扣你的薪水。但既然有这样的技术，我们就应该充分发挥它的长处，给我们的工作带来便利。