计算机根本无法完全顶替人来完成现实世界的工作，因此，讨论面向对象是否会直接将现实世界表示为程序本身就是毫无意义的。关于该主题，我们将在第 9 章中详细讨论。

2.11　与现实世界的相似扩大了可能性

本章介绍了面向对象与现实世界是似是而非的。这是妨碍大家理解面向对象的主要原因，但同时也是扩展该技术可能性的原动力。

面向对象之所以能够应用于业务分析、需求定义等上流工程，其中一个原因就在于类、实例、继承和消息传递等结构与现实世界的情形非常相似。另外，“面向对象”的概念激发了人的想象力，有些人开始想将其在哲学和认识论上扩展。另一方面，在实际的软件开发领域，出现了使用 OOP 结构的框架和类库等大规模的可重用构建群，还诞生了设计模式等优秀的思想。它们浑然一体，展示着“面向对象”概念的魅力，同时自身也成了 IT 领域的潮词。另外，关于使框架和设计模式等 OOP 成为可能的可重用技术，我们将在第 6 章中进行讨论。关于面向对象被作为归纳整理法应用的相关内容，我们将在第 7 章中加以介绍。

专栏

对象的另一面

成为潮词的面向对象

在IT领域，每年都会诞生一些表示技术趋势的新词，其中，在2000年之后流行的词语有“普适”“Web 2.0”“BPM”“网格计算”“SaaS”“SOA”和“云计算”等。

这种词语通常被称为“潮词”（buzzword）。英语单词“buzz”是形容蜜蜂或者机器发出的嗡嗡声的词汇，包含“令人生厌”“刺耳”的语气。

潮词具有开拓市场的作用。由于抓住了潜在需求和技术可能性的词语会吸引客户的注意，所以厂商会将其用于产品或者服务的营销上，活动策划公司会将其用作研讨会的主题，媒体也会通过出版物进行传播。如果这些活动开展得比较热烈，那么相关的产品和服务也会变得丰富起来，同时能吸引更多客户的注意，甚至还催热整个行业。

不过在很多情况下，潮词的寿命并不会太长，大多只有半年左右，能持续3年的就已经算是寿命很长的了。

潮词的寿命取决于其概念的魅力，同时还取决于技术的成熟度。就像我们在第2章的热身问答中为大家介绍的“NODM”一样，随着对象技术的广泛普及，有些潮词的寿命就结束了。反之，像“人工智能”等潮词，由于曾经的技术发展赶不上概念的超前，所以这也会导致其一度从大众视野中消失[1]。

* * *

“面向对象”也可以算作一个潮词，它是20世纪60年代出现的一门古老的技术，在2000年以后，随着Java和.NET等OOP开始在企业系统开发领域发挥重要作用，“面向对象”有时也被用于市场营销或者销售等用途。

面向对象是涵盖编程语言和设计

① 近年来由于机器学习和深度学习技术的进步，“人工智能”再度活跃，重新成为炙手可热的潮词。——译者注

手法等的开发技术，主要服务于“创建系统的人”。由于它并不会给“使用系统的人”直接带来好处，所以作为面向用户企业的市场营销术语来说吸引力并不是很强。

另一方面，对于很多“创建系统的人”即开发者来说，该词则具有极大的魅力。类、继承等独特的结构，以及将 Smalltalk、Java 中的 Object 类作为祖先类的类库等，这些在习惯了以往的编程环境的人看来都非常新奇。另外，设计模式、UML 建模和敏捷开发流程等也让很多开发者非常感兴趣。具有哲学意味的“面向对象”概念通过扩展其对象领域，使人越发难以把握其全貌。

在这种情况下，也有一些为开发者提供产品和服务的人们将“面向对象”用作营销术语，赋予其“魔法技术”的含义。

笔者认为，“面向对象是直接将现实世界表示为软件的技术”这种解释，也许源于开发者在最初接触该技术时产生的惊奇之情，之后便夹杂着各类人群的利益得失和困惑等而不断流传下来。

本章的关键词

机器语言、汇编语言、高级语言、结构化编程、GOTO 语句、全局变量、局部变量

第3章 理解 OOP：编程语言的历史

热身问答

在阅读正文之前，请挑战一下下面的问题来热热身吧。

问题

下列哪一项是在 20 世纪 60 年代后半期 NATO（北大西洋公约组织）召开的国际会议中声明的“软件危机”的内容？

A. 20 世纪末，从事计算机相关工作的人口数量会增加，而从事农业和水产业的人口数量则会减少，因此粮食危机会变得很严重

B. 20 世纪末，即使全人类都成为程序员，也无法满足日益增大的软件需求

C. 20 世纪末，全世界的计算机都会联网，而计算机病毒带来的危害将变得很大

D. 20 世纪末，软件的非法复制将横行，这将导致版权销售业务进展不下去

B. 20 世纪末，即使全人类都成为程序员，也无法满足日益增大的软件需求

解 析

“软件危机”（software crisis）是指人类的供给能力满足不了日益增大的软件开发需求的状况，这一概念是 1968 年 NATO 在联邦德国召开的国际会议中提出的。

在同一时期，“软件工程”（software engineering）一词也被创造出来，对高效开发高质量软件的各种手法和编程技巧的研究成了一门学问。

另外，由于“软件工程”一词具有很强的学术气息，所以本书多使用“软件开发技术”一词。

本章重点

虽然笔者也想立刻就开始对OOP（Object Oriented Programming，面向对象编程）进行讲解，但是在这之前，还是先介绍一下OOP出现之前的编程语言吧。

OOP 的结构非常简练，但另一方面也非常复杂，因此理解其结构及用途并不简单。不过，理解 OOP 也有捷径可循，那就是先掌握在 OOP 之前产生的编程技术能够实现什么、存在哪些限制。

因此，本章将介绍从机器语言到汇编语言、高级语言和结构化语言的编程语言进化史。格言说得好，“欲速则不达”“温故而知新”。相信大家一定会有新的发现。

3.1 OOP 的出现具有必然性

人们有时会认为面向对象是一种能够按照现实世界中的观点来创建软件的全新思想，将取代传统的开发技术。不过，笔者却对此持反对意见，认为 OOP 以在它之前出现的编程技术为基础，用于弥补这些技术的缺陷。也可以说在不断改进编程技术的历史中，OOP 的出现具有必然性。此外，面向对象框架内涵盖的其他技术也是对 OOP 的发展和应用，是之前优秀的开发技术的延伸。

因此，在开始介绍 OOP 结构之前，本章将简单回顾在 OOP 之前出现的编程语言以及从机器语言到结构化编程的进化史。如果大家能够充分理解该历史，就一定能明白面向对象是一门有助于高效创建高质量程序的实践性技术。

3.2 最初使用机器语言编写程序

计算机只可以解释用二进制数编写的**机器语言**。并且，计算机对机器语言不进行任何检查，只是飞快地执行。因此，为了让计算机执行预期的

工作，最终必须有使用机器语言编写的命令群。幸好现在有 Java、C 语言、COBOL 和 FORTRAN 等编程语言，程序员才基本无须在意机器语言。不过，在计算机刚刚出现的 20 世纪 40 年代是没有这么方便的编程语言的，程序员必须亲自用机器语言一行一行地编写程序。

代码清单 3.1 是使用机器语言编写的、用十六进制数表示的程序示例。

代码清单3.1　使用机器语言编写的程序示例

```
A10010
8B160210
01D0
A10410
```

这里只编写了能执行极其简单的算术计算的命令，但是我们却看不出来这写的是什么。在计算机诞生初期，只有极少数掌握这种机器语言的超级程序员能操作计算机。顺便提一下，当时的计算机使用真空管制造，体积非常庞大，据说在制造计算机之前要先建造存放计算机的建筑。即便如此，当时的计算机的性能却比如今低得多。在现在这个时代，想必再怎么用机器语言编写程序也不够用吧。

3.3　编程语言的第一步是汇编语言

为了改善这种低效的编程，**汇编语言**就应运而生了。汇编语言将无含义的机器语言用人类容易理解的符号表示出来。如果我们使用汇编语言改写代码清单 3.1 的程序，就会得到如下的代码清单 3.2。

代码清单3.2　使用汇编语言编写的程序示例

```
MOV   AX, X
MOV   DX, Y
ADD   AX, DX
MOV   Z, AX
```

除非是专业人士，否则也很难理解程序内容吧。不过，即使是不了解汇编语言的程序员，只要稍加想象，大概也能理解代码清单 3.2 中的 MOV 是信息传送、ADD 是加法运算的意思。

汇编语言是编程语言的第一步。使用汇编语言编写的程序被读入对其进行编译的其他程序（称为“汇编程序”）中，从而生成机器语言。计算机本是为了轻松执行人类的工作而设计的机器，所以人们希望编写程序驱动计算机工作的任务也可以使用计算机轻松进行。得益于汇编语言，程序简单易懂很多，错误也有所减少，之后进行修改也变得非常轻松。

不过，在使用汇编语言编写的程序中，即使命令存在一点点错误也会导致程序运行异常。另外，虽然汇编语言容易理解，但是在编程时逐个指定计算机的执行命令是非常麻烦的。

3.4 高级语言的发明使程序更加接近人类

随后，用更贴近人类的表达形式来编写程序的**高级语言**被发明出来。高级语言并不是逐个编写能让计算机理解的命令，而是采用人类更容易理解的“高级”形式。

采用 FORTRAN 改写代码清单 3.2 的程序，可得到如下的代码清单 3.3。

代码清单3.3　使用FORTRAN编写的程序示例

```
Z=X+Y
```

对比代码清单 3.2 和代码清单 3.3 的程序，我们会发现高级语言的便捷性是非常明显的。代码清单 3.3 中程序的形式与数学计算公式非常相似，因此即使是完全没有编程经验的初高中生也能理解[1]。这种高级语言在计算机刚出现不久的 20 世纪 50 年代前半期被设计出来。现在仍在使用的

① 实际上，这与算式并不相同。不懂程序的初高中生看到代码清单 3.3 时，会将该程序理解为是两边的值相等的方程式，而其实这里的等号表示将右边赋值给左边。

FORTRAN 出现于 1957 年，COBOL 大概出现于 1960 年，它们已经存续了 50 多年。在技术革新非常快的计算机领域，这些高级语言仍然能被长时间地使用，真是了不起的发明。

随着高级语言的出现，编程的效率和质量都得到了很大提升。不过，由于计算机的普及和发展速度更加惊人，所以人类对提高编程效率的需求并未止步。于是在 20 世纪 60 年代后半期 NATO 召开的一次国际会议上提出了所谓的**软件危机**——20 世纪末，即使全人类都成为程序员，也无法满足日益增大的软件需求。

3.5 重视易懂性的结构化编程

为了应对软件危机，人们提出了各种新的思想和编程语言。

其中，最受关注的就是**结构化编程**。

结构化编程由荷兰计算机科学家戴克斯特拉（Dijkstra）提出，其基本思想是：为了编写出能够正确运行的程序，采用简单易懂的结构是非常重要的。

具体方法就是废除程序中难以理解的 GOTO 语句[1]，提倡只使用循序、选择和重复这三种结构来表达逻辑（图 3-1）。

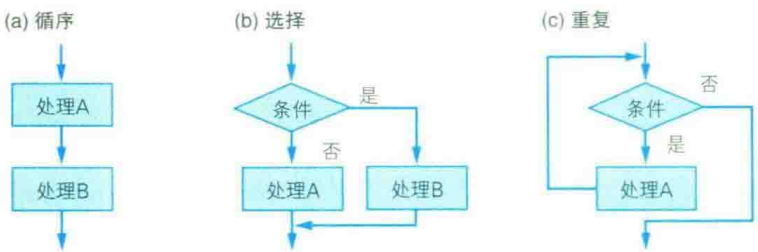

图 3–1 三种基本结构的流程

① 无条件跳转到程序中的任意标签的命令。虽然 Java 中没有该语句，但是 C 语言和 C++ 中都有。

“循序”是指从头开始按顺序执行程序中编写的命令。“选择”是指执行某些判断，根据判断结果决定接下来要执行的命令。诸如 if 语句和 case 语句，有编程经验的人应该马上就能理解。“重复”是指在规定次数内或者某个条件成立期间，重复执行指定的命令群，相当于程序命令中的 for 语句和 while 语句。这三种结构被称为**三种基本结构**。由于该理论非常强大而且易于被接受，所以得到了广泛的支持。

另外，由于结构化编程主张废除 GOTO 语句，所以又被称为**无 GOTO 编程**。对现如今的程序员来说，不使用 GOTO 语句基本上已经是常识了，而在 20 世纪 70 年代，由于计算机的内存容量和 CPU 速度等硬件性能都很差，所以当时推崇将程序编写得尽可能简练，哪怕只是减少一字节或者一步也好，因此那时的程序常常难以处理。特别是在滥用 GOTO 语句的情况下，程序的整体结构变得很杂乱，让人非常难以理解。现在我们将这些难以理解的程序统称为“面条式代码”，用来表示因滥用 GOTO 语句等而导致控制流程像面条一样扭曲纠结在一起的状态。

在当时那个年代，据说最初有人对结构化编程持批判态度，理由是它会导致程序的代码量增加、执行速度变慢。然而随着时代的进步和计算机硬件性能的提升，我们渐渐发现，从系统整体来看，执行效率的细微改善并没有什么效果，与之相比，编写易懂的程序才是更加重要的课题。

3.6 提高子程序的独立性，强化可维护性

在当时，为了强化程序的可维护性，还有另外一种方法，就是提高**子程序**[1]的独立性。

早在计算机诞生初期的 20 世纪 40 年代，子程序就已经被发明出来了。该结构被用于将在程序中的多个位置出现的相同命令汇总到一处，以减小程序的大小，提高编程的效率。不过现在大家开始意识到，只是简单地将

① 子程序（subroutine）还有其他叫法，如过程（procedure）、函数等。

相同的命令语句汇总到一处还不够，为了强化程序的可维护性，提高子程序的独立性也是很重要的。

提高子程序独立性的方法是减少在调用端（主程序）和子程序之间的共享信息。所谓共享的信息是指变量中存储的数据。这种能在多个子程序之间共享的变量被称为**全局变量**。

程序逻辑可以按顺序进行解读。不过，由于我们很难一眼看出变量在程序的哪个位置被引用，所以如果在程序中定义了很多变量，那么维护就会变得很困难。特别是全局变量，由于从整个程序的所有位置都可以对其进行访问，所以如果在调试时发现变量的内容有误，就必须检查所有源代码。由此可见，减少全局变量对提高程序整体的可维护性而言非常重要。

下面我们就来具体介绍一下。如图3-2所示，这里有A、B、C三个子程序，它们之间使用全局变量交换信息。在这种结构中，我们很难知道哪一个子程序在什么时间点修改或者引用了变量。由于从程序的任意位置都可以访问全局变量，所以在因某种情况而修改了全局变量的情况下，就必须确认程序的所有逻辑。该示例中只有三个子程序，逻辑也很短，所以不会有很大的麻烦。

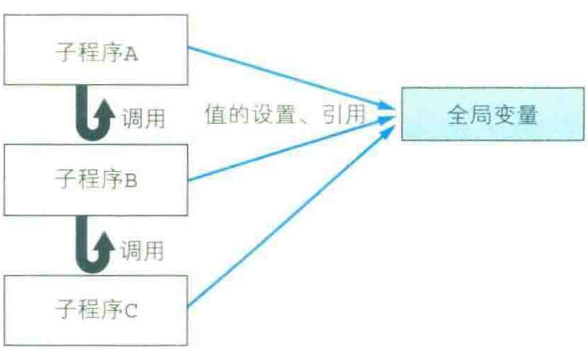

图3-2 使用全局变量来交换信息

但对于包含了成百上千个子程序的应用程序而言，修改全局变量就是

一个很严峻的问题。手动确认影响范围的话自不必说，以当时的机器性能，使用计算机进行确认也是极其困难的。毕竟在当时，应用程序即使稍微修改一下也要花费几小时的时间进行编译，而编译确认修改全局变量造成的影响则需要等待一个晚上。

为了避免出现这样的问题，人们设计出了两种结构：一种是**局部变量**[1]，另一种是**按值传递**（call by value）（图 3-3）。

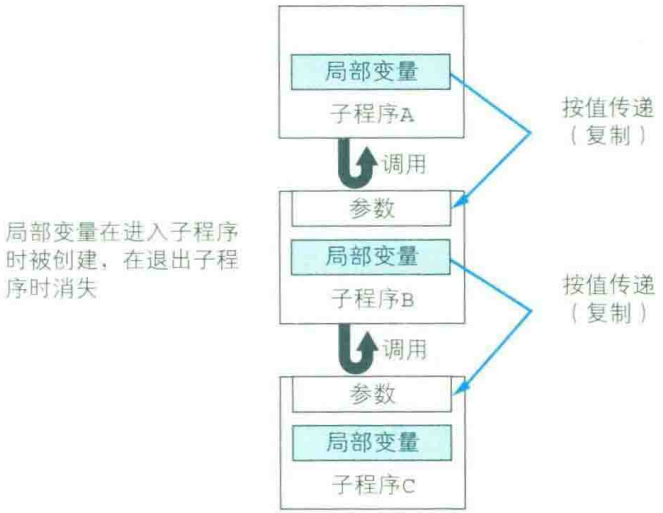

图 3-3　使用局部变量和按值传递来交换信息

局部变量是只可以在子程序中使用的变量，在进入子程序时被创建，退出子程序时消失[2]。

通过参数向子程序传递信息时，不直接使用调用端引用的变量，而是复制值以进行传递，这就是按值传递的结构。在使用这种结构的情况下，即使修改了被调用的子程序接收的参数值，也不会影响调用端的变量。

① 有时也被称为“本地变量”“自动变量”。

② COBOL 的 WORKING-STORAGE SECTION 中定义的变量基本上在退出子程序时仍会保持原状态，因此该局部变量的结构对 COBOL 程序员来说可能有些难以理解，然而在 C 语言等结构化语言中则是最基本的结构。

这两种结构可以使全局变量的使用控制在最小限度，减少子程序之间共同访问的变量。通过巧妙地使用这种结构，可以提高子程序的独立性。

3.7 实现无 GOTO 编程的结构化语言

随着结构化编程理论的渗透，出现了以此为基础的编程语言，即**结构化语言**，具有代表性的结构化语言有 ALGOL、Pascal 和 C 语言等。

结构化语言可以使用 `if` 语句、`while` 语句和 `for` 语句等命令编写明确的控制结构。现在大家可能认为这是理所当然的，然而之前的主流语言，例如 COBOL 和 FORTRAN 等，其语法并不一定可以直接编写三种基本结构[①]，因此结构化语言的出现是极大的进步。

另外，虽然结构化编程也被称为无 GOTO 编程，但是有趣的是，Pascal 和 C 语言中却提供了 GOTO 语句。也许看起来有些自相矛盾，但其实这是为了跳出嵌套循环结构等特殊情况而准备的。为了平衡执行效率，使用 GOTO 语句有时也是一种备选方案。虽然当时计算机性能已经有所提升，但是在那个年代，在执行效率上多下一点功夫就会有很好的效果。

结构化语言也提供了前文中介绍的局部变量和按值传递功能，现在很多编程语言也都采用了这些结构。

结构化语言中最有名的就是 **C 语言**。C 语言完全支持结构化编程的功能，此外还具备之前只有汇编语言才可以执行的位运算以及高效使用内存区域的指针等细致功能。因此，它可以被广泛应用于从应用程序开发到系统编程等诸多领域，例如可以被应用于 UNIX OS 的实现语言等。

C 语言的另一个特征就是，编程所需的全部功能并不是通过语言规范提供，而是由函数库构成的。例如，在 C 语言中，格式化输出字符串的处理是使用 `printf` 函数实现的，而在 COBOL 和 FORTRAN 中则对应为语

① 随着之后对语言规范的修订，现在 COBOL 和 FORTRAN 中也可以显式地编写三种基本结构了。

言规范。在采用这种结构的情况下，即使不改进语言编译器，也可以添加语言规范层的功能。现在的 Java 等面向对象语言中也继承了这种思想。

现在被广泛使用的 Java、C++ 和 C# 等编程语言都是 C 语言的直系子孙，继承了它的许多性质。

3.8 进化方向演变为重视可维护性和可重用性

在此让我们试着总结一下编程语言的进化历史吧。

在从机器语言到汇编语言乃至高级语言的进化过程中，人们希望提高编程语言的表现能力，即用更贴近人类的方法简单地表示出希望让计算机执行的作业。为此人们开发出了一系列具有代表性的高级语言，其中 FORTRAN 使用算式、COBOL 使用英文报告的形式来编写程序［COBOL 中将整体结构分为 4 个 DIVISION（部），其下再设置 SECTION（节）］。迄今为止，可以说使用贴近人类的形式编写程序的目的已经基本实现了，不过遗憾的是，仅凭这一点还无法拯救软件危机。

因此，在向接下来的结构化语言进化时就需要改变方向（图 3-4），即提高可维护性。无 GOTO 编程以及提高子程序独立性的结构都是为了便于既有程序的理解和修改。

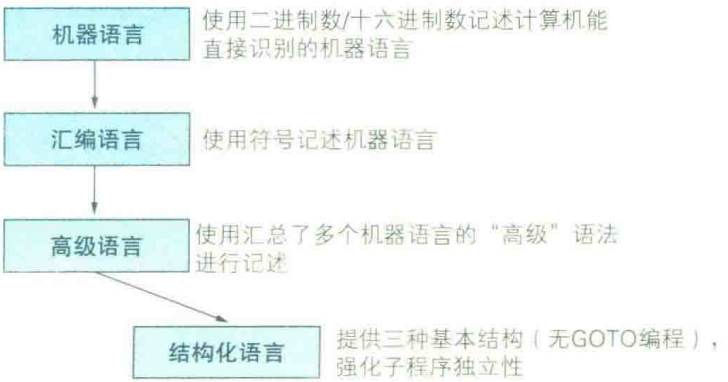

图 3-4 编程语言的进化方向

在这种背景下，程序的寿命比最初设想的长了很多。在计算机刚出现时，每次都要从零开始重写程序。然而，随着对计算机的要求越来越高，程序规模也越来越大，每次都要重写程序的话就太费事了。因此，通过修改已完成的程序进行应对的情况就开始增多了。

20 世纪末轰动世界的“计算机 2000 年问题”就是一个程序的实际寿命比设想寿命长很多的例子。造成该问题的原因在于许多程序中都用两位数来表示年份。不只是应用程序，计算机厂商提供的操作系统和基础软件等都受到了影响。但是回过头来想一想，为什么在当时谁都没有考虑到这么简单的事呢？真是难以想象。或许是因为当时内存的价格的确非常高，所以连 2 个字节也不能浪费吧。但笔者认为，更重要的原因在于大家对程序寿命的认识不正确。

出现问题的代码很多都是通用计算机或者 UNIX 运行的操作系统等基础软件，这些软件的最初版本都是在 20 世纪六七十年代编写的。当时的程序员可能都认为自己编写的程序只有 5 年左右的寿命，最多也不会超过 10 年。毕竟当时人们都认为进入 21 世纪后，机器人会代替人做家务，铁臂阿童木将在空中穿梭，大家都可以轻松地去宇宙旅行，所以应该没有哪个程序员能想象到在那样遥远的未来，自己编写的程序在被修改之后还能一直使用吧。

随着程序寿命的延长，人们对编程语言的功能要求也发生了改变。编程语言最开始只是被用来简单地表示机器语言的命令，之后新的要求不断出现，便于理解既有程序的功能（提高可维护性）、降低复杂度、不引起错误的功能（提高质量）等开始受到重视。另外，充分利用既有程序来提高整体生产率（提高可重用性）的功能也变得非常重要。

对上述内容加以总结，如图 3-5 所示。