并发程序的系统结构

〔美〕P.B.汉　森　著
杨芙清 等　编译

国防工业出版社

内 容 简 介

并发程序设计是操作系统的一个新的研究课题。本书介绍了开发并发程序设计的系统方法，和一种称为并发 Pascal 的结构化语言，并通过单用户操作系统、作业流系统、实时调度程序等三个重要的并发程序阐述了这种语言的应用。书中给出了这三个系统的程序，并说明了它们是怎样构造、编制、测试和描述的。

在附录部分还给出了并发 Pascal 语言的语法公式和并发Pascal 机器中的有关概念。

本书可作为计算机软件专业研究生和高年级学生的教材或教学参考书，也可用作计算机软件工作者使用的结构化并发程序设计手册。

THE ARCHITECTURE OF CONCURRENT PROGRAMS
PER BRINCH HANSEN
1977 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey

并发程序的系统结构

〔美〕P. B. 汉 森 著

杨芙清 等 编译

*

国防工业出版社 出版

新华书店北京发行所发行　　各地新华书店经售

廊坊日报印刷厂印刷

*

787×1092 1/16　印张18 1/2　424千字

1982年11月第一版　1982年11月第一次印刷　印数：0,001—6,200册

统一书号：15034·2364　　定价：1.90元

目　录

程序设计工具

并发程序

语言细节

程序设计工具

第一章 设 计 原 则

这本书讲述一种书写高质量并发程序的方法。由于各个程序员对于一个好的程序所必须具备的质量标准尚无共同的见解，所以我开始讲一下对一个好的程序所必须具备的一些质量要求。

1.1 程 序 质 量

好的程序必须是简明的、可靠的和可适应的。离开简明性，人们就很难了解一个大型程序的目的和细节；离开可靠性，人们就不能真正信赖程序；如果程序不能适应变动的要求，那么，它最终就会成为僵化的东西。

幸好，这些基本要求是相辅相成的。简明性使人们信任程序的工作，同时也使人们清楚地了解怎样才能改动它。简明性、可靠性和适应性三者使得程序易于管理。

此外，我们也希望构造一些能在几种不同的计算机上针对各种类似的应用而有效地工作的程序。但是，我们决不应该在牺牲简明性、可靠性和适应性的前提下去追求功效性、移植性和通用性。因为只有简明性、可靠性和适应性，才能使人们明白程序做什么事，才能使人们敢于信赖它们，并扩充它们的能力。

许多现存软件质量的低劣，在很大程度上就是由于把这个问题本末倒置而造成的。一些程序员对极端复杂而且难懂的程序的辩解理由就是“功效高”。另一些人则认为，他们的大型程序应用范围的广泛性足以弥补可靠性差和功效低的缺点。

我觉得，一件工具如果没有人充分理解，其功效也就无从谈起。同时也难于尝识那些慢得什么事也做不好的所谓通用工具。不过，这是兴趣和风格问题，以后或许还是这样。

每当各种程序质量标准彼此之间发生抵触时，我总是这样来解决的：即把易管理性放在第一位，功效性次之，通用性再次之。这就归结为这样一条简单的原则：把计算机的应用限制在程序员能充分理解，并且机器又能较好地实现的那些方面。虽然这个观点对于实验性的计算机用法来说是过于狭窄一些，但对于专业性的程序设计来说却是一个强有力的忠告。

现在让我们更仔细地来看看，怎样才能达到这些程序质量标准。

1.2 简 明 性

我们要书写一些大得使人不能马上看懂的并发程序。这样，我们必须在一些较小的片段上来讨论它们。这些片段应该具备什么性质呢？它们应该小到每个片段都易于了解。为了使人一目了然，理想的是每个片段的长度都不要超过一页。

对这样一个程序，就可以象读书一样一页一页地来研究。但最后，当搞清了每一个片段所做的工作时，还必须能看出它们作为一个整体所具备的功能是什么。如果它是一个包含许多页的程序，我们只能这样来研究它，即不去考虑这些片段的大部分细节，而只着眼于“它们做什么”和“它们如何在一起工作”这样一些简单得多的描述。

因此，程序片段必须能使我们分清片段的细节性态，以及当我们考虑这些片段的组合时所感兴趣的那一小部分信息。换句话说，必须分清一个程序片段的内部性态和外部性态。

我们建立程序片段以实现一些定义良好的简单功能，然后，把这些程序片段组合成更大的结构，以实现更复杂的功能。由于这种设计方法把一个复杂的任务分解成一些比较简单的任务，因而它是有效的。首先，我们相信这些片段能分别进行工作，然后，再考虑它们如何在一起工作。在后一步中，回避程序片段的工作细节这一点是很重要的——不然的话，问题就变得太复杂了。但在这样做时，作了一个基本的假定，即程序片段在实现其功能时总是做同样的事情。否则，在你考虑整个系统时就无法回避程序片段的详细性态了。

因此，再现性是我们希望通过一些小步骤来建立和研究的各个程序片段的一个重要特性。在选择构成大型并发程序的程序片段种类时，必须清楚地记住这一点。在书写顺序程序时，我们认为，程序性态的再现能力是理所当然的。因为，在这种情况下，事件序列完全是由程序和它的输入数据所决定的。但在并发程序中，程序员并不能完全控制同时发生的事件发生的速率，它们取决于机器中存在的其他作业，以及执行它们所采用的调度策略。这就意味着，必须有意识地致力于设计具有再现性的并发程序。

如果能用一些比较简单的片段（它们本身是由一些更简单的片段构成的）来解释每一片段，并借助这种解释重复地进行下列过程，即先考虑一个程序片段做什么，然后从细节上研究它怎样做，则这种方法便是最有效的。因此，我们将仅限于讨论由一些程序片段的层次所组成的层次结构。

如果每一部分只依赖于少数别的部分，那就一定会简化对层次结构的理解。因此，我们将试图建立一些在各个组成部分之间具有最小界面的结构。

用机器语言来做到这一点是极为困难的。这是因为最微小的程序设计错误就可能使得一条指令破坏任何指令或变量。这里，整个存贮器都可能是任意两条指令间的界面。过去，仅仅为了找出一个程序设计错误，就得打印整个存贮器内容这一事实，就清楚地说明了这一点。

用抽象语言（例如 Fortran、Algol 和 Pascal）书写的程序，自身不能修改。但它们仍然以全程变量的形式拥有广泛的界面，每一个语句都可能（有意地或错误地）改变这些全程变量。

我们将使用一个称做并发 Pascal 的程序设计语言，它使我们可以把所有的全程变量分组，其中每一组都只能被少数几个语句所存取。

一个好的程序设计语言对于简明性的主要贡献，就是向读者提供一个抽象的易读的记法，使程序的各部分以及整个结构都显而易见。一个抽象的程序设计语言略去了机器的细节（如地址、寄存器、位组模式、中断，有时甚至是可用的处理机数目等）。相反，语言却以一些抽象的概念（如变量、数据类型、同步操作以及并发进程等）为基础。因此，用抽

象语言书写的程序正文，常常要比用机器语言书写的程序正文短一个数量级。正文的缩减大大简化了程序工程。

要想知道你是否创造了一个简单的程序结构，最快的方法是试着以完全易读的术语来描述它——采用在杂志上发表的综述性文章那样的清晰标准。如果你对自己的描述感到满意，那么，可能你已经创造了一个好的程序结构。要是发现没有简单的方法来描述你想做的事，那么，你也许应该去寻找做这件事的其它方法了。

一旦你认识到描述作为多余复杂性早期警告信号的意义时，那么，不言而喻，在建立程序结构之前就应该对这些结构（除细节外）加以描述，而且应该由设计者本人（而不是任何别的人）来描述。程序设计是一种用透明、清晰的散文体来写文章，并使它们可以执行的技艺。

1.3 可 靠 性

即使最易读的语言记法也不能避免程序员出错。为了在一些大型程序中寻找错误，我们需要所有可能的帮助。下面一整套技术是可用的

正确性证明

校对

编译检查

运行检查

系统测试

除了正确性证明之外，所有这些技术在研制本书讲述的并发程序中均发挥了重要作用。

形式证明还处于实验阶段，特别对并发程序来说更是如此。因为我的目的是讲述对专业性软件的开发直接有用的一些技术，所以这里就不去讨论证明了。

在有用的验证技术中，我感到应该强调那些在程序开发过程中能尽早地揭示错误的技术，以便能尽快获得可靠性。

并发Pascal的主要目标之一是最大限度地发挥编译检查的作用，并尽可能减少使用运行检查。由于减少了运行检查的开销，使编出的程序具有更高的功效。在程序工程中，编译检查和运行检查分别起到航空中的预防性维修和飞行自动记录仪那样的作用。后者只是告诉你系统为什么坠毁，而前者则是防患于未然。当设计一些将在社会上控制要害职能的实时系统时，这两种检查的区别对我们来说似乎是本质的。这样的系统在投入运行之前必须是高度可靠的。

只有在语言记法裕余的情况下才可能进行广泛的编译检查。为了使编译程序能找出可能存在的前后矛盾，程序员必须至少能用两种不同的方法指明一些重要的特性。例如，变量被语句使用之前，必须利用说明来引进这些变量及其类型。编译程序能够很容易地从语句中推出这些信息。这里，假定这些语句总是正确的。

我们亦将遵循霍尔（Hoare）所提出的程序设计的重要原则：用一种抽象语言书写的程序，其性态应该总是能用这种语言的概念加以说明的，而不必洞悉编译程序和计算机的细节。否则，一种抽象语言记法在减少复杂性方面也就没有很重要的价值了。

这一原则直接排除了在程序设计语言中使用面向机器的特色。因此我们假定，

所有的程序都将用抽象程序设计语言书写。

戴克斯特拉谈到，测试只能用来指出错误的存在，而不能说明它不存在。虽然这句话可能是正确的，但我看，能指出错误的存在，从而逐一地加以排除，这就很有价值。根据我的经验，仔细的校对、广泛的编译检查和系统测试相结合是一种很有效的方法，它可以把程序搞得相当可靠，以致工作好几个月都不出问题。它与我们所依赖的大多数其它技术一样可靠。此刻，我还不知道有什么更好的检查大型程序的方法。

我把程序设计看做一门建造程序宝塔的技艺，每次向结构体增加一块砖，并且确保它在此过程中不致倒塌。这座宝塔在整个建造过程中必须保持稳定。只要一个程序（可能是不完全的）按照预定的方式运转，我就认为它是稳定的。

为什么程序测试往往很困难呢？我想，主要是由于增加新的程序片段可能引起一串错误而遍及程序的其余部分，从而造成以前测试过的程序片段运行异常。这显然和下述正确原则相抵触：即应能设想，当你建立并测试了一个大型程序的一部分时，它将在各种情况下继续正确地运转。

因此，我们将提出一个强硬的要求：添加到一些老程序片段顶上的各个新程序片段，一定不能使这些老程序片段发生错误。由于这个性质必须在程序测试之前得到证实，所以这件事就必须由编译程序来做。因此，必须用一个语言记法，使得程序片段之间的关系是清楚的。把程序错误明确地限制到发生错误的部分，这就使我们更易于由一个大型程序的性态来判定错误发生的位置。

1.4 适 应 性

开发一个大型程序的代价很昂贵，因此，为了使付出的努力合算，这个程序就必须能使用多年。随着时间的推移，用户的要求有所变动，这就需要对程序做出某些修改，以满足用户的要求。往往做修改的人并不是当初的程序开发者，他们的主要困难是查明程序如何工作，以及修改之后是否还能正常工作。

一小组人往往能成功地用低级语言设计出一个程序的最初版本，而不用或者很少用支援性的文档。他们通过日常的彼此交谈，并且在脑子里共享一幅简单的结构图，就可以做到这一点。

可是以后，当同一个程序必须由跟原先的设计者联系很少的其他程序员进行扩充时，令人头痛的是，这个“简单的”结构在任何地方都没有描述过，并且肯定也没有通过所用的原始语言记法展示出来。因此，认识到如下一点是很重要的：一个简单的和归档良好的结构，在程序维护阶段比在程序开发阶段更为重要。这里，对于那种既不简单又不是归档良好的程序需要改动的情况，我们不去讨论它。

程序设计错误和变动着的用户要求之间，有个令人感兴趣的关系。在程序的构造过程中，二者都是不稳定的根源，这就使你对程序做的事很难完全信任。这是由于我们无法立即完全了解一个大型程序具体地做些什么而引起的。

程序错误和变动的要求之间的相对频率是很重要的。如果程序设计过程中出现一些难以定位的错误，则当用户要求改变程序功能时，其中好多错误可能仍然留在程序中。如果一个工程师经常发现他在改变一个以往未能使之正确工作的系统，则他最终生产出的产品

将是很不稳定的。

另一方面，如果程序查错和纠错的速度比系统的开发速度快得多时，那么，往程序中添加一个新程序片段(或进行改动)，不久便会达到一个稳定状态，在这种状态下，现行的程序版本能够可靠地按预定要求工作。于是，工程师就有很大的把握使他的产品适应于缓慢变化的要求。这对于加速程序证明和测试来说是个很好的推动。

一个层次结构，由可以逐个研究的一些程序片段组成。这种结构使得程序比较易读，比较容易初步了解它做什么和怎样做。一旦你有了这种洞察能力，改动一个分层程序所造成的后果也就很清楚了。当你改动程序宝塔的一部分时，你必须准备去检查，或许还要修改处在它顶上的那些程序部分（因为只有它们才可能与你所改动的那一部分有关)。

1.5 移 植 性

由于经济上的原因，同一个程序适应不同机器的能力是有吸引力的：许多用户拥有不同的计算机；有时他们需要用新机器取代旧机器；他们共同感兴趣的往往是共享在不同机器上研制的程序。

只有用尽量隐蔽各种机器间差别的抽象语言来编写程序，移植性才是实用的。否则，为了把程序从一台机器移植到另一台机器，便需要进行大量的改写和测试工作。采用如下几种方法，可以把同一种语言写的程序变成可移植的：

（1）不同的机器用不同的编译程序。这只对大多数广泛流行的语言才是一种实用的方法。

（2）用一个可以经过修改而为不同机器生成代码的编译程序。这就需要在这个编译程序中能清楚地区分出程序检查部分和代码生成部分。

（3）具有一台能在不同的机器上对其进行有效模拟的计算机。

并发 Pascal 编译程序为一台简单的计算机生成代码，而该机器是为这种语言而特制的。这台机器是用 4K 字的汇编语言程序在 PDP11/45 计算机上模拟产生的。为了把这种语言移植到另一台机器上，我们只要改写该解释程序。这种方法为实现可移植性而牺牲了一些功效。但在具有微程序控制的机器上却可以使功效不受损失。

1.6 功 效 性

高功效的程序能节省人们等待结果的时间和降低计算的费用。这里所讲述的程序功效较高的原因是

专用算法

静态存贮分配

最少的运行时检查

最初，在 PDP11/45 计算机上，从磁盘装入一个大型程序（比如一个编译程序)，大约需要 16 秒钟。后来用一种磁盘分配算法使它减少到 5 秒钟。这种算法依赖于程序文件(相对于数据文件而言）的特殊性质。通常用来减少磁头移动的调度算法在这里是没有用的。其原因在以后会清楚。

动态存贮算法在执行过程中来回移动程序和数据段，这可能是程序员无法控制的低功

效的重要根源。并发 Pascal 的实现不需要收集无用单元或申请调页，它在固定数目的进程之间采用静态存贮分配。存贮容量要求是由编译程序决定的。

如果程序是用汇编语言书写的，则无法预知它们将要做些什么。大多数计算机依靠硬件机构来防止这些程序间的互相破坏或破坏操作系统。在并发 Pascal 中，这种保护大部分是由编译程序来保证的，而不是在执行期间由硬件机构支援的。因为所有的程序都是用一种抽象语言写的，所以才有可能大幅度地减少运行时检查。

1.7 通 用 性

为了达到简明性和可靠性，我们只使用一种与机器无关的语言，它使得程序易读并有可能进行广泛的编译检查。为了提高功效，我们将采用尽可能简单的存贮分配。

这些抉择无疑会降低并发 Pascal 对某些应用领域的实用性，但我看这是不可避免的。把框框加给你自己，就是限制程序设计自由度。你再也不能随心所欲地使用机器(因为这种语言不能直接涉及机器的某些特性)。也不能把有关程序的某些抉择推迟到执行时刻（因为编译程序要更早地检查和决定这些事)。但你所丢失的自由往往是一种错觉，这是因为，这种自由会使程序复杂到使你无法应付的程度。

本书讲述一些小型操作系统。其中，每一个都以最有效和最简单的方式提供了一种专门的职能。这些系统表明，对于小型计算机操作系统和实时应用来说，并发 Pascal 是一种很有用的程序设计语言。我希望这种语言对于编写大型通用操作系统也是有用的（但不是足够的)。然而这还有待于观察。我试图创造这样一种程序设计工具：它对于许多应用来说都非常方便，而不是对一切应用都勉强过得去。

1.8 结 论

我已经讨论了下述的程序设计目标

简明性

可靠性

适应性

功效性

移植性

并且还指出，这些目标可以通过精心设计程序结构、语言记法、编译程序以及代码解释程序来达到。我们必须寻求以下的性能：

结构：　层次结构

　　小的组成部分

　　最小界面

　　再现性

　　易读的归档说明

记法：　抽象和易读

　　结构化的和裕余的

编译程序：　可靠而快速

广泛的检查
可移植的代码

解释程序：可靠而快速
最少的检查
静态存贮分配

这就是我们在并发程序设计中所要遵循的原则。

参 考 文 献

ALEXANDER, C., *Notes on the synthesis of form*. Harvard University Press, Cambridge, MA, 1964.

BRONOWSKI, J., *The ascent of man*. Little, Brown and Company, Boston, MA, 1973.

BROOKS, F. P., *The mythical man-month. Essays on software engineering*. Addison-Wesley, Reading, MA, 1975.

ELSASSER, W. M., *The chief abstractions of biology*. American Elsevier, New York, NY, 1975.

HARDY, G. H., *A mathematician's apology*. Cambridge University Press, New York, NY, 1967.

LANGER, S. K., *An introduction to symbolic logic*. Dover Publications, New York, NY, 1967.

MCNEILL, W. H., *The shape of European history*. Oxford University Press, New York, NY, 1974.

SIMON, H. A., *The sciences of the artificial*. M.I.T. Press, Cambridge, MA, 1969.

STRUNK, W., and WHITE, E. B., *The elements of style*. Macmillan, New York, NY, 1959.

第二章　程序设计概念

我们用较小的成分按层次来构造大型并发程序。每个成分都应该具有定义良好的功能，并且可以作为几乎是独立的程序来实现和测试。这些成分及其组合应具有再现性。当这些程序由于要适应新的需求而发生变动时，对它们的验证及测试必须比这种变动要快得多。

这一章介绍我们就要使用的那一类成分，并且讲述如何连接它们。我们用来进行程序设计的工具，是一种称作并发 *Pascal* 的语言。它用进程、管程和类程这些新概念扩充了顺序程序设计语言 Pascal。

这是关于并发 Pascal 的一个非形式的描述。它使用一些例子、插图和文字来说明新的程序设计概念中创造性的方面，而并不涉及进一步的细节。另几章将介绍这些概念的语言记法，并简明扼要地定义这些概念。从形式的观点来说，这种陈述方式也许是不严格的，但是，我希望，从人的观点来说，它是更为有效的。

2.1 并发进程

下面，通过求解一个简单而有用的问题来介绍这种语言：如何尽快地将一段正文从读卡机复制到行式打印机上去？

图 2.1 表示出一台读卡机、一台行式打印机和一个把数据从前者复制到后者去的程序。读卡机和行式打印机以 1000 行/分和 600 行/分的速度传输信息（即每传输一行信息分别用 60 毫秒和 100 毫秒）。

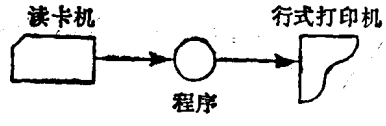

图2.1　数据复制

此问题最简单的解法是如下的循环顺序程序

```
cycle input; output end
```

该程序每次从读卡机上输入一行并把它送往行式打印机。

令人遗憾的是，它的功效非常低。这是因为，它让读卡机和行式打印机交替地

输入，输出，输入，输出，……

以致当一个在操作时，另一个总是处于等待状态。因此，复制速度仅为 375 行/分（或160 毫秒/行）。

只有让读卡机和行式打印机同时操作才能提高速度（图 2.2）。这时，复制程序由两个同时执行的顺序进程组成

```
卡片机进程：cycle input; send end
打印机进程：cycle receive; output end
```

卡片机进程每次从读卡机上输入一行，然后，通过一个缓冲区把它发送给打印机进程。打印机进程接收信息，而后将信息输出给行式打印机。此程序以速度最慢的设备的速度（即 600 行/分）复制正文。

因为我们感兴趣的是抽象的程序设计，所以，并发进程在计算机上究竟如何具体实现，那是无关紧要的。所需要知道的是：这两个进程是同时执行的，它们能使有关的外围设备同时运行。

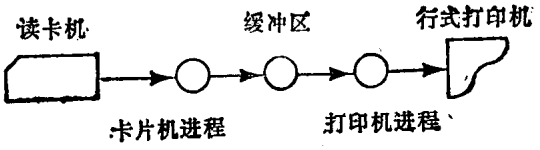

图2.2 并发进程间的数据流程

在有些计算机上，利用时钟中断的方法，可以使并发进程多路复用单处理机。在另一些计算机上，每个进程都由其各自的处理机来执行。我们将有意不管这些细节，而假定执行抽象并发程序经编译生成的代码的机器是会处理这些细节的（第九章将讲述并发 Pascal 在 PDP11/45 计算机上的实现）。

由于不考虑机器的细节，所以，就无法预测进程的绝对速度或相对速度。不过，我们假定所有的进程都具有确定的速度。（说到底，要是不知道机器将执行这段程序的话，何苦要来写它呢？）通常，主机总是比它的外围设备快得多，因此，我们可以期望进程大致与它们所管理的设备的速度相当。

2.2 专用数据

让多个顺序进程同时执行，便构造出并发程序。由于大多数程序员对顺序程序设计都已经有深刻的、直观的了解，因此这是颇有吸引力的。

一个顺序进程是由一个数据结构和在此数据结构上进行操作的顺序程序组成的(图2.3)。各个程序语句严格地一条接一条地顺序执行下去。

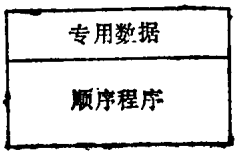

图2.3 进程

对顺序程序而言，重要的是：只要它对相同的数据进行操作，所得到的结果总是相同的，而与它的执行速度无关，有关系的只是操作执行的顺序。

顺序程序中的程序设计错误总可以采用下面的方法查出，即，利用暴露错误的数据多次重复执行程序。在每次这样的试验中，把所搜集到的变量值都记录下来，以便确定究竟是哪一段程序造成了错误。这个排错过程一直继续到查出错误为止。

只要在一次测试中发现某程序段运行正确，我们就可以在后继的测试中排除该程序段(及其变量)。这是因为：只要程序针对相同的数据执行，每次所表现出的效果就完全相同。基于程序的这种再现性，我们就能用分步法测试大型顺序程序。

然而，顺序进程的这种与时间的无关性，仅当它的变量不为其它进程所存取时才能确保。反之，如果一个进程所使用的变量可以被另一进程改变其值，那么，结果就和这些进程的相对速度有关了。

一个并发程序用相同的数据多次执行时，进程的相对速度总会有所改变。在一个多路计算机中，一个进程的执行将因其它进程的存在而受影响（虽然它们可能是毫不相干的）。而在一个多处理机系统中，执行速度将依赖于操作员对程序请求的反应速度。

如果并发程序中包含这样一个错误，它在不能预测的时刻改变了另一个进程的变量，那么，尽管这个程序每次都是针对相同的数据执行，但是每次得到的结果却不一样。

这种不能预测的程序性态，使得我们不可能用系统测试的方法查出错误。或许，这要通过对程序正文详细检查好多天才能发现。但是，对那些由数千行组成的、毫无线索可寻

的程序来说，即使这种检查是有可能实现的，那也太令人望而生畏了。

如果我们希望成功地构造出可靠的大型并发程序，就必须使用那样一些程序设计语言，这些语言结构良好，使得编译程序能够查出大多数与时间有关的错误（因为，此外再无任何人能做到这一点）。所以，我们将选择一种语言记法，它能清楚地指出，一个进程拥有哪些变量，而编译程序则保证这些专用变量不为其它进程所存取。

2.3 外围设备

外围设备是程序性态不稳定的潜在的根源，值得高度重视。为使数据输入和数据处理并行，传统的程序设计技术是：设置一对缓冲区，为一个顺序程序及其输入装置所存取。

程序将第一批数据项输入给缓冲区变量 x。当程序对 x 进行操作时，设备又将第二批数据项输入给另一个缓冲区变量 y。然后程序对 y 进行处理，与此同时，第三批数据项又输入给 x，如此等等。

不止一个程序员犯过这样的错误：在数据项完全输入进来以前，就去对数据进行查询。这就使程序结果依赖于程序的执行和数据传输二者的相对速度。

问题在于这种程序设计技术把一个程序及其外围设备转化成并发进程，而这些并发进程可能会由于差错而相互访问对方的专用变量。

在并发 Pascal 中，一台外围设备只能被一种输入/输出操作所存取，该操作把调用进程延迟到输入/输出完成为止。同样，在任何时刻，一个变量可以由单个进程或单台设备所存取（但是不允许它们二者同时存取）。数据传输正是另一种完全能再现其结果的顺序操作。

当一个进程正在等待数据传输完成时，计算机可以执行另外一些进程。因此，这种处理方式可减少主机的空闲。由一个缓冲区联系起来的两个进程可同时实现数据项的输入与处理（图 2.2）。

使输入/输出作为不可分的操作的另一个好处是，外部中断变得与程序员毫不相干。它们完全是在机器一级处理的。

当计算机的问题初次提出时，它们往往是用非常复杂的方法来解决的。为了找到一个明显的解法，要花很长时间。而要熟悉这些方法，还要再花一段时间。输入/输出程序设计就很好地说明了这一点。

2.4 共享数据

虽然确保一些变量为进程所专用是极其重要的，但它们也必须能够共享一些数据结构（如缓冲区），否则，并发进程便不能交换数据，当然也就不能合作以完成共同的任务。但是，由于共享数据是并发程序设计中较大的难点，所以我们做起来必须格外小心，并且应严格规定什么进程才能共享这样的数据结构。

在前面提到的复制程序中，缓冲区是一个被两个并发进程共享的数据结构（图 2.2）。至于这样一个缓冲区是如何构造的，其细节与用户无关。这两个进程所需要知道的只是，它们能通过缓冲区发送和接收数据。如果进程打算用任何其它方式对缓冲区进行操作，那么，不是一个程序设计错误，就是一个程序设计花招。在这两种情况下，我们都希望编译

程序能查出这种滥用共享数据结构的情况。

为使这一点成为可能，我们必须引进一种语言构造，它能使程序员告诉编译程序，进程如何使用共享数据结构。这类系统成分称为管程。管程能使并发进程同步，并在它们之间传送数据。管程也能控制竞争使用共享物理资源的各个进程所应遵循的次序。

管程定义了一个共享的数据结构和各个进程在该数据结构上所能执行的全部操作（图2.4)。这些同步操作称为管程过程。管程还定义了一个在建立它的数据结构时所执行的初启操作。

我们可以把一个缓冲区定义为一个管程，它由表示缓冲区内容的一些共享变量所组成。它还包括两个管程过程：发送和接收。开始，初启操作将缓冲区置空。

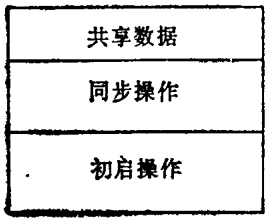

图2.4 管程

进程不能直接对共享数据进行操作。它们只能调用对数据具有存取功能的管程过程(如发送和接收)。管程过程（和任何其它过程一样）作为“调用进程”的一部分来执行。

如果多个并发进程同时调用在同一个共享数据上进行操作的那些管程过程，那么，必须严格地按照一次执行一个过程的原则进行。否则进程便会发现数据结构处于某种（未知的）中间状态，这就会使调用管程后所产生的结果成为不可预测的。

这就是说，主机必须能使进程延迟很短的一段时间，直到轮到进程去执行管程过程为止。关于如何做到这一点，就不去讨论了。但是，我们恰恰要注意，当进程执行一个管程过程时，它具有对共享数据的独占存取权（第九章解释有关的实现细节)。

因此，有并发进程运行的主机，要有管理同时调用管程的短程调度。但是，程序员还必须能使进程延迟稍长一段时间，直到进程对数据和其它资源的需求得到满足为止。例如，若一个进程打算从一个空缓冲区接收数据，它就必须等待其它进程送来数据。

并发 Pascal 包含一个称作队列的简单数据类型，管程过程利用这种数据类型来控制进程的中程调度。管程或者让调用进程到一个队列中去排队等待，或者让在一个队列中等待的进程继续运行。

至于这些队列是如何工作的并不重要，重要的是应该了解下述规则：一个进程只有在继续执行管程过程的语句时，才具有对共享数据的独占存取权。一旦一个进程被延迟到某个队列中等待，该进程就失去了它的独占存取权，直到别的进程调用该管程并且使它继续执行时为止。

编译程序将检查进程是否仅通过管程过程进入管程。这一点对程序可靠性影响很大。它表明，一旦一个管程已经正确实现，则程序的其它部分就不能使它失效。不管程序的其余部分干些什么，这个管程总是一个稳定的正确成分。编译时对专用变量的保护性检查，对进程也有同样作用。

Fortran、Cobol、PL/1 和 Pascal这些程序设计语言，都使用公用数据（全程变量）作为各别的程序段之间的界面。这就使得一段程序很容易以意想不到的方式改变另一段程序的数据结构从而使其失败。

并发 Pascal 基于如下的假设，即过程是远比公用数据结构安全得多的界面机构。程序

员使用与一数据结构有关的各个过程，就可以定义在该数据结构上的所有可能的操作，并且依靠编译程序使程序的其余部分不能以任何其它方式使用该数据。

2.5 存 取 权

到目前为止，我们只介绍了能够构造并发程序的两种成分——进程和管程。我们还需要一个精确的方法来描述怎样把这些成分连接成层次结构。

图 2.2 一目了然地表示出从一个卡片机进程经由一个缓冲区到一个打印机进程的数据流程，我们把它称作数据流程图。

图 2.5 给出了用另一种观点来看的同一个系统。其中，圆圈表示系统成分，箭头表示这些成分的存取权。由图可知，两个进程都能使用缓冲区，但是，读卡机只能为卡片机进程所使用，行式打印机只能为打印机进程所使用。我们把这类图称为存取图。

进程的存取权只能使这些进程调用由缓冲区管程所定义的发送过程和接收过程，它并没有授予进程对代表缓冲区的数据结构直接进行操作的权力（顺便提及，外围设备可看作是由硬件实现的、只能被单个的输入/输出过程所存取的管程）。

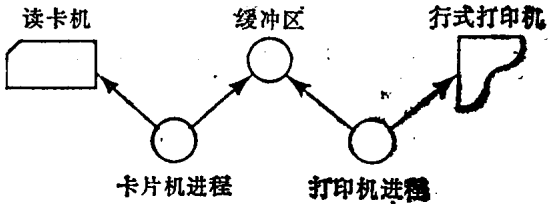

图2.5 系统成分及其存取权

这就是我们的结构化机构：用存取权将程序的成分连结成层次系统，在该系统中，并发进程通过调用管程来进行通讯。

在一个大型并发程序中，应该把这些存取权写出来，从而使程序结构对读者来说是清晰的，同时，对于编译程序来说，则是便于验证的。这样，我们就把进程扩充为带有存取权（图 2.6），存取权指出进程所能调用的一些管程。

还有一点，前面所述的复制例子没有体现出来，即：管程过程也应该能调用在其它管程中所定义的过程。否则，这种语言对于层次式的设计来说就显不出特别有用了。因此，一个管程对其它的管程也可以有存取权（图 2.7）。

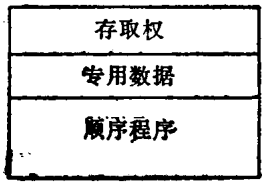

图2.6 带有存取权的进程

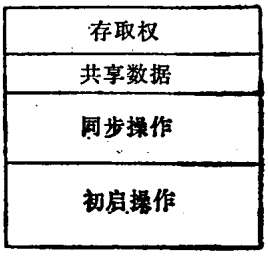

图2.7 带有存取权的管程

进程之间只能通过管程进行通讯。编译程序将检查一个进程是不是只使用了它拥有存取权的那些管程。

2.6 抽象数据类型

进程执行顺序程序——进程是主动成分。管程则是过程的集合，这些过程在被进程调用之前，什么事情也不做——管程是被动成分。然而进程与管程却有很多类似之处：二者

都定义了一个数据结构（专用的或共享的）以及在各自的数据结构上的意义明确的操作。进程和管程之间的主要区别在于：为执行它们而采用的调度方式不同。

因此，我们很自然地把进程和管程都看成是抽象数据类型，它们正是用我们能对之施行的一些操作来定义的。之所以说它们是抽象的，是由于程序的其余部分只需知道能用它们做什么而不管这些数据是怎样构造和处理的。只要所定义的操作保持不变，即使改变了数据表示，也不致影响程序的其它部分。

在复制系统中，缓冲区可以表示成一行、若干行、一个连接表或者一个树形结构。并且，它既可以存放在磁心存贮器中，也可以存放在磁盘存贮器中。进程对此是不关心的。从进程的角度来看，只要能通过缓冲区发送或接收信息就行了。这样，就使程序员能自由地采用不同的数据表示进行实验，以改进程序的性能。

在抽象的数据类型中隐蔽了实现的细节，这使我们易于对一个程序进行局部调整，也使我们更易于了解，程序作为一个整体应该做什么。这是因为，所有这些数据虽然表示方式不同，但贯穿其中的发送与接收这一抽象的思想却是相同的。

因为编译程序能检查这些操作是否就是可以在有关的抽象数据结构上施行的全部操作，所以我们有希望构造出一些非常可靠的并发程序。对这些程序而言，在它们执行之前（甚至在测试之前）就可以保证对数据和物理资源的控制存取权。这样，就以很低的代价基本上解决了资源保护问题（既不用硬件机构，又无运行时的开销）。

一个有用的概念能反复（即不止一次）使用。因此，我们把进程和管程都定义成数据类型，这样，便可能在一个系统中使用进程和管程的若干实例。比如，我们可以用两个缓冲区建立一个流水线系统——在该系统中，数据流经一个卡片机进程、一个复制进程和一个打印机进程（图 2.8）。

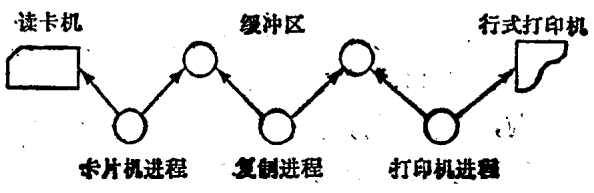

图2.8 流水线系统

复制进程按如下格式构造正文：每个文件都用一个空页开头、一个空页结尾；每页都用一个空行开头、一个空行结尾；而每一行的两端都留有空格。

由于在外围设备进程中，输入、输出和执行是严格地交替出现的，因此，我们希望，为使设备运行尽可能快一些，它们所做的数据处理工作量要尽可能小。把构造正文格式的工作交给一个独立的进程来完成便能实现上述愿望，这个独立的进程能在其它进程等待输入/输出时运行。复制系统的这一扩充具有这样的优点，即所有以前的成分都不必改变(图 2.5)。因此，这个例子也可以说明，一个程序能在完全不用改变的情况下去适应新的需要。

在并发程序中，程序员只定义缓冲区类型一次，但要给出缓冲区的两个实例说明。我们把缓冲区的定义称为系统类型，而把缓冲区的实例称为系统成分，以示二者之间的区别。存取图（如图 2.8）总是表示系统成分而不表示系统类型。

在程序执行期间，机器为每一个系统成分分别创建一个数据结构。但是，同一类型的成分却共享一套与数据有关的过程。因此，尽管流水线系统使用两个缓冲区变量，却只使用一个发送过程和一个接收过程。