结构程序设计

O.-J.达尔　E.W.戴克斯特拉　C.A.R.霍尔 著

陈火旺　吴明霞　丁宪理 译

科学出版社

1980

内 容 简 介

这本论文集包括结构程序设计、数据结构设计和层次程序结构三部分．结构程序设计(包括数据设计)是六十年代末进入软件工程化时期所形成的程序设计方法学的主要内容．本书介绍了结构程序设计的基本思想、原则和方法，探讨了程序设计的一般思维方法和可用的数学工具，研究了防止程序出错和正确性证明的方法，讨论了大型程序和数据结构的组成方法和质量标准．

本书可供计算机科学和计算机工程的科研人员，高等院校有关专业的师生和软件工作人员阅读。

O. -J. Dahl, E. W. Dijkstra and C. A. R. Hoare

STRUCTURED PROGRAMMING

Academic Press Inc., 1972

结 构 程 序 设 计

O. -J. 达尔　E. W. 戴克斯特拉　C. A. R. 霍尔 著

陈火旺　吴明霞　丁宪理 译

科学出版社出版

北京朝阳门内大街 137 号

中国科学院印刷厂印刷

新华书店北京发行所发行　各地新华书店经售

*

1980 年 11 月 第 一 版　开本：787×1092 1/32

1980 年 11 月第一次印刷　印张：8 3/4

印数：0001—7,250　字数：199,000

统一书号：15031·306

本社书号：1897·15—8

定 价：1.35 元

目　录

I. 结构程序设计札记

E. W. 戴克斯特拉

1. 致 读 者

这些札记有着"写给自己看的信"之特色. 我觉得，如果不把它们写下来，我自己将一次又一次地重复着一些相同的争议. 但当我读自己所写的这些札记时，我又总感到不甚满意.

一则，我嫌这些札记写得很累赘. 但(现在)我还不想精炼它们. 首先，因为这样做可能会出现别的迟误，而我还想"细细加以捉摸". 其次，先前的经验使我耽心，问题可能会被误解：许多程序员往往把他们的困难(有时是相当特殊的困难)看成是问题的症结，结果甚至连"程序设计到底是什么"这样的问题也存在着广泛的分歧.

尽管有这些不足，我仍希望读者至少将欣赏其中的某些部分. 如果这些札记对你有所鼓舞或者使你对程序员的职业有某种新的评鉴，那末我的部分目的也就达到了.

在出版以前，《结构程序设计札记》就已经非公开地传播了. 人们对它的兴趣——对此我表示感谢——更加鼓励了我再增添某些材料，并把它改写成可公开出版的形式. 在此，我感谢 B. Floyd, R. London 和 M. Woodger 等人的鼓励性的评论，感谢 P. Naur 提出的批评. 最后，我还感谢 E. L. Dijkstra-Tucker 夫人对我英语方面的帮助.

2. 人的能力的局限性

我面临着一个表达方式的基本问题. 我所关心的是大程序的组成方法和技术，这种程序的文本，譬如说，可能和这一章的全部文句一样长. 为了说明各种技术方法，我不得不需要引用一些例子. 但由于实践上的原因，这些示范性的例子只能很小很小，比我心目中的"实际大程序"要小很多. 我的基本看法是，正是这种量级上的差别，是我们在程序设计中所遇到的种种困难的主要原因之一.

如果我能用简短的范例来阐明各种程序方法，并能概括地指出："对于成千倍大的程序可如法组成"，那就简直太好了. 但是，这种在教学上可行的举一反三的办法在程序设计上未必行得通. 我的中心论点之一是，任何两个在某些方面有着千差万别(即量级上的差别)的东西是完全不可比较的.

历史表明这种真实性是难以置信的. 显然，我们常会学究式地忽略一些量级上的差别，以致把它们当作"非本质的渐近差别"来对待. 我们相信，凡我们能做一次的事情，我们也能做两次，归纳法使我们愚弄自己，似乎我们能做它任意多次. 这是不真实的! 因为数以千倍之后，就大大地超出了我们的想象力!

让我用两个例子来加以说明. 譬如说，一个周岁的孩子用四肢爬行，每小时可走一英里. 而一架超音速飞机每小时可飞行一千英里. 就移动能力来说，小孩和飞机是不可比较的. 因为这二者所具有的能力是互不相及的. 又例如，你可以想象自己站在一个开阔的地方，一个大草原或海边，而在很远的地方有一匹脱缰的马飞奔而来，你可以详细"看清"它如何奔驰而来，呼啸而过. 但是，假定不是一匹马，而是数以千

计的大群野兽. 你要再"看清"它们的每一只如何奔腾而来，呼啸而过，那就不可能了；如果你能的话，你的心也受不了惊慌的打击.

更进一步地说，"量"的问题不仅使我觉得存在表达方式的问题，而且我的中心论题正是：普遍的低估量的特殊困难，似乎是现今软件失败的主要原因之一. 对所有这些问题，我看只有一个回答，即尽量明确地处理量的问题. 这就是本节标题的含意.

首先，我们来看一看经常碰到的计算"量"的问题. 所谓计算量，指所涉及的信息量和运算次数的全体. 计算量的大小是一个本质的问题. 因为，如果计算量很小的话，根本可以不用计算机，用手算即可，计算机之所以能够存在，正是在于它能完成人所不能进行的大量计算. 我们希望计算机能做那些我们自己可能从未做过的事. 但是，现代计算机的能力如此之大，乃至对机器来说即使是计算量很小的事情，也远远超出了我们的想象力.

迄今我们仍需按一定方法组织在计算机上计算，使得我们有限的能力足于保证：这个计算将达到我们所期望的结果. 这种组织包括程序的组成. 从而，我们面临着第二个问题，即程序长度问题. 对这个问题，我们也必须有明确的认识. 我们必须知道这样的事实，即程序的长度和所描述的问题在量方面的大小是密切相关的. 例如，在我的国家里，电话簿是按城镇和乡村的名字分类的，在每一类中，户主姓名按字母顺序排列. 我自己住在一个小村庄里，随便给我一个电话号码，只需找几行，我就能知道这个号码的户主是谁. 但在一个大城市里，要做同样的事情，可能是一件巨大的数据处理任务!

同样的心情，我希望读者注意程序的"清晰性"问题. 这

个问题确有量方面的特征. 奇怪得很，许多数学家似乎并不理解这一点. 例如，假定一条定理的前提条件竟要写上十来页，那末这一定理很难想象是一个方便的工具. 因为，每当要用这个定理时就必须论证所有条件均被满足. 在欧氏几何中，毕达哥斯定理的条件是满足下述性质的任意三个点 A, B 和 C: 通过 A 和 C 可作一直线垂直于通过 B 和 C 的直线. 但是，有多少数学家欣赏这个定理在三个点部分或全部重合时仍得成立这一事实呢？ 但是，这确实是毕氏定理的方便之所在.

总之，作为一个迟钝的人，我只有尊重这种局限性，蛮干只会导致错误.

3. 机器的可靠性

作为一个职业程序员，我所谈的是程序，而本节的主题实际上是程序的可靠性问题. 本节的标题之所以冠以"机器"字眼，是因为我把程序看成为机器的具体体现. 我强烈地感觉到，关于软件的可靠性的许多考虑，稍加变换，也适用于硬件的设计.

现今的计算机是一组非凡的设备. 最使我们惊奇的是，计算机具有人们赖以得到种种有意义的结果的无尽能力. 当然，首先我们相信硬件的工作是正常的.

让我们暂时注意一下硬件，并假定我们不清楚人们能在多大程度上弄清它的真实结构. 几年前，在我的大学里安装了一台机器；该机的说明书指出，它具有一个 27 位整数定点乘法器. 一个合理的问题是："这个乘法器正确吗？ 它能象所说明的那样正确地执行吗？"

对于这个问题的天真回答是，"这个乘法器所能正确执行

的乘法的全体只有有限个，即 2^{54} 个，让我们逐一测试吧！”但是，这种回答并不是有道理的．因为，虽然一个乘法只需几十微秒，但要试遍所有的 2^{54} 个数却要 10,000 多年．我们可以断言，穷尽测试法，对即使只有一个象乘法器那样的部件，也是不现实的．（同样地，要测试一整台机器意味着检查所有可能的程序的正确运行！）

一万年意味着这个乘法器在它的生存期间里只能完成它所可能完成的乘法的很小很小的一部分，即可忽略的一部分．事实上，没有一台机器经得起穷尽测试．有趣得很，我们仍然要求，当需要它时，机器能正确地执行任何乘法．形成这种离奇的质量要求的理由是，我们并不预先知道，将付诸测试的那个可忽略的小部分将包括哪些乘法．当我们分析和论证程序时，我们所涉及的是“积”，而不是形成这个积的具体因子．对于后者，我们常常不了解它们，也不希望去了解它们，我们没有义务去了解它们，我们的义务正是不要去了解它们！我们可以期望用“积”的概念进行思考，而不问其计算过程的具体情形．但是，为此而花的代价正是对可靠性的要求，即任何一个它可能做的乘法应被正确执行．这些就是我们期望一个正确乘法器的理由．

但是，如何用一种可信赖的方式建立可靠性呢？只要我们把乘法器当作一只黑箱来考虑，我们所能做的只是“抽样检查”，即，给这个乘法器提供一组可能的因子配对，逐一检查相乘的结果．但是，用一万年的眼光来看，显然我们只能检查所有可能的乘法的很小一部分，即可忽略的一部分．而所有那些在某种意义上是“危险”的乘法，可能仍未被检查到．从我们刚才所期望的可靠性的观点来看，这种质量控制方法，仍然是非常令人不满意的．因此，不能用这种方法．

直接的结论是，只要把机器看作是一只黑箱，就不可

能用一种可信赖的方式建立可靠性. 我们唯一的希望是不把机器当作一只黑箱. 我将称此为"考虑机器的结构".

从现在开始,我们所处理的机器类型是指程序(作为程序的机器,在许多方面比电路的机器更容易处理,后者是一种模拟设备,可装可拆). 但即使是程序,也没有希望在不考虑结构的情形下,能用检测法建立令人信服的正确性. 换言之,我们觉得,程序正确性不单纯依赖于程序的外部说明和工作状态,而密切地依赖于它的内部结构.

由于我们真正关心的是实际的大程序,作为旁白,我们知道,量(指程序的"大小")本身要求程序的各组成部分是高度可信赖的. 假定一个组成部分的正确性概率为 p, 一个含有 N 个组成部分的完整程序的正确性概率就有点象是

$$P=p^N.$$

当 N 足够大时, 如果我们希望 P 是个异于 0 的有意义值的话, p 就应该非常接近于 1.

如果我们认为, 程序员的任务不仅在于编制一个正确的程序,而且还应该以一种可信赖的方式证明程序的正确性,那末,上述的旁白对程序员的活动就有深刻的影响:他所编制的程序应是有效结构的.

本文的后一部分主要是探索怎么样的程序结构比较可取. 随之,显然的是,我不仅关心程序的正确性,而且关心程序的适应性(或可修改性). 强调程序的可修改性,是经仔细推敲的,因此有必要加以说明.

在过去, 由于一般设备能力的增长而缓和了对效率方面的紧迫要求. 但是,正是这种增长,产生了新的困难. 因为,当人们有一台高效能的机器任其摆布时, 人们设法充分利用它, 并将所处理的问题的大小自动地对准于这台机器的能力所能及的范围: 没有人会想去编一个竟要执行二十年的程

序. 在过去的十至十五年中，随着计算机的处理能力成千上万倍的增长，人们选择那些“技术上可能做到的”的课题的胃口愈来愈大. 程序的长度、复杂性和高度技巧性打破了历史纪录. 但在过去的十多年中，十分明显的是，从总的情况来说，我们的程序设计能力并没有满足需要.

机器的能力还得不断提高. 我们能够期望工厂发展更快速的计算机. 即便不是为此，我们也能目击现今的快速计算机愈来愈普遍. 从而我们想利用这些机器做的事情也将成比例地增长. 正是根据这种估计，形成了我自己关于程序设计任务的见解.

我认为，现在应是停止把程序设计主要看成为价格/效能最小比的时候了. 我们必须认识，现在的程序设计已经肯定是一种智力的挑战：程序设计技术是一种组织复杂性的技术，是一种控制巨量数据和尽量有效地回避混乱的技术.

我不把效能看成为程序员的首要任务，并不意味着我不重视效能的问题. 相反的，提出正确程序的可修改性的主要动机之一，正是出自对于效能的考虑. 我的看法是，只有当程序是可修改的，我们才能优化它(不论哪一方面).

在结束这一节之前，我想再强调一下计算机的意义. 计算机是一种十分灵活而有效的工具，从而使许多人认为，计算机的使用改变着地球的面貌. 我敢大胆地说，只要我们把计算机看成为一个工具，我们就可能低估它的意义. 作为工具来说，在文化上的影响大致不过是一斑涟漪，而我期望它在智力上的挑战方面有着更深刻的影响.

本节第一部分的推论：

程序调试只能指出弊端的存在，但不能说明没有弊端.

4. 思 维 方 法

在上节我们说过，程序员的任务是把他的程序搞成为"有效结构的"，而且这种结构是和建立对程序正确性的某种可信赖的证明密切相关的.

但是，如何信赖呢？理由何在？哪些思维的经典形式可作依托？本节将粗略地考察一下这些问题. 遗憾得很，因为我对这些问题的认识仍十分肤浅. 但我希望(而且相信)，我所写的这些东西将为我们提供一种尺度，用以衡量任何一种拟议的结构方法的有效性.

可用来理解一个程序的种种思维方法之中，我提及以下三种:

(1) 枚举法

(2) 数学归纳法

(3) 抽象

4.1. 枚举

我把论证下述计算过程具有某种性质的努力视之为诉诸于枚举，这种计算过程可由一组顺序执行的语句——包括含有两个或多个分支的条件语句——的可枚举集来表示. 下面是一个使用"枚举法"进行论证的简单例子.

证明下面两个语句

$$\text{“}dd:=dd/2;$$

$$\textbf{if } dd\leqslant r \textbf{ do } r:=r-dd\text{”}$$

连续执行的结果(它们运算于变量"r"和"dd")保持关系

$$0\leqslant r<dd \tag{1}$$

的不变性. 假定关系(1)在这个小程序执行前是成立的. 你可

"跟随"着这个小程序开始步步验证. 第一个语句执行之后, dd 的值减半, 但 r 不变, 因而

$$0 \leqslant r < 2*dd \tag{2}$$

成立. 现在就两种互斥的情形论证第二个语句执行后的情形:

(a) $dd \leqslant r$. 由(2), 所以

$$dd \leqslant r < 2*dd; \tag{3}$$

因为在这种情形下, **do**后的语句将被执行, 即 r 将被减去 dd, 于是从(3)最终可得

$$0 \leqslant r < dd,$$

即(1)保持成立.

(b) **non** $dd \leqslant r$(即 $dd > r$). 在这种情形下, **do**后的语句不执行, 所以第一个语句执行后的 r 也就是它的最终值. 由"$dd > r$"和(2)——它是第一句执行后所得的关系, 立即可知

$$0 \leqslant r < dd$$

所以在第二种情形下, 关系(1)也保持成立.

至此我们证明了关系(1)的不变性, 同时也做完了包括含有条件语句的枚举法的例子.

4.2. 数学归纳法

我之所以明显地提及数学归纳法, 是因为, 它是我们所知的唯一能对付循环(如那些可由"重复语句"表示的循环)和递归过程的一种论证方法. 例如,

让我们考虑值的序列

$$d_0, d_1, d_2, d_3, \cdots\cdots \tag{1}$$

它的定义式为

$$d_i = D \qquad i = 0 \tag{2a}$$

$$d_i = f(d_{i-1}) \qquad i > 0 \tag{2b}$$

其中 D 为已知值，f 为已知(可计算)函数. 假定"prop"是个已知(可计算)条件，并假定序列(1)中存在着满足这个条件的值. 要求我们求出变量"d"的值，此值等于序列(1)中首先满足条件 prop 的那个 d_k. 更加形式的说法是，要求我们建立关系

$$d=d_k \tag{3}$$

其中 k 满足

$$\text{prop}(d_k) \tag{4}$$

和

$$\textbf{non prop}(d_i),\ 0\leqslant i<k \tag{5}$$

让我们考虑下面的程序段

"$d:=D$;

while non prop(d) **do** $d:=f(d)$" (6)

其中第一行置初值，第二行是个由(意义自明的)重复语句 **while**…**do**… 所控制的循环.

(按前例中所用的条件句 **if**…**do**…，重复语句的语义的一种较为形式的定义可为:

"**while** B **do** S"

语义上等价于

"**if** B **do**

begin S; **while** B **do** S **end**"

后者指明，"**non** B"是终止循环的必要充分条件.)

我们称结构"**while** B **do** S"中的语句 S 为"被重复语句". 我们将证明：在程序段(6)中，被重复语句在第 $n(n\geqslant 0)$ 次执行后将有

$$d=d_n \tag{7a}$$

和

$$\textbf{non prop}(d_i)\quad 0\leqslant i<n \tag{7b}$$

上述命题对于 $n=0$ 显然是成立的(由枚举法). 假定这

个命题对于$n=N(\geqslant 0)$已证为成立，我们要证明对$n=N+1$，命题同样成立(用枚举法).

由于在$n=N$的情形下，被重复语句的第N次执行后，关系(7a)和(7b)成立. 因为，被重复语句能再次执行(即第$N+1$次)的必要充分条件是

$$\textbf{non prop}(d)$$

为真. 根据归纳假设，当$n=N$时，(7a)成立(即$d=d_N$)，于是

$$\textbf{non prop}(d_N).$$

这意味着，对$n=N+1$，条件(7b)仍然成立. 另者，由$d=d_N$和(2b)，导得

$$f(d)=d_{N+1}$$

所以，被重复语句

$$"d:=f(d)"$$

第$N+1$次执行后所得的全部结果是建立关系

$$d=d_{N+1}$$

即，关系(7a)对于$n=N+1$时保持成立. 从而归纳步骤(7)获得证明.

我们现在回头证明，在被重复语句的第k次执行后，循环即应终止. 显然，对于$n>k$，不能发生第n次执行；否则，由(7b)，将有

$$\textbf{non prop}(d_k)$$

这和(4)相违背. 如果在被重复语句的第n次执行后循环就终止，其充分必要条件

$$\textbf{non}(\textbf{non prop}(d))$$

就变成为(由(7a))

$$\text{prop}(d_n). \tag{8}$$

这说明，在$n<k$时，循环不应终止；否则，违背(5). 于是，循环必将在$n=k$时终止. 从而由(7a)可得(3)，由(8)得(4)，由

(7b)得(5). 证明完毕.

这里，我想就这个例子对于作为一种论证方式的数学归纳法的应用再加些解释. 因为我有一种不平静的感觉，我的某些读者(特别是那些经验丰富和能力很强的程序员)现在将会勃然大怒. 即对他们来说，程序段(6)的正确性是如此之明显，以致于他们不明白这种小题大作到底是为了什么："对于象(3)，(4)和(5)所定义的简单命题，为何需要大书特论？ 任何人都知道序列中满足某条件的第一个值是什么！ 每当我们使用如此简单的一个循环时，人们肯定不会要求我们无所事事地提供那样一个穿上华丽的数学外衣的冗长证明！". 凡此等等.

说实在话，上述的笨重证明，也使我自己感到烦脑！ 但是，在现在，如果真的希望证明这个程序的正确性，我确实没有更好的办法. 以前，平面几何里的第一批定理的荒诞证明，也常常使我感到同样的愤怒，因为这些定理所论证的事情几乎和欧几里得公理自身一样地"明显".

当然，我不敢建议说(至少在现在)，每当程序员在他的程序中写下一个简单循环时就得提供那样的一个证明. 如果那样的话，他可能无从下手. 因此这种要求在今天是不实际的. 但是，企图把欧氏几何中的每一个定理的证明全都明显地归到欧氏公理，同样也是不实际的！ (参见第 2 节，人的能力的局限性.)

我的寓意有三重. 首先，当一个程序员认为象(6)那样的结构是显然正确的时候，他可以那样认为，因为他是熟悉这种结构的. 我情愿把他的认识看成是他对自己所知道的一个定理的不自觉应用，尽管他可能从未为证明这个定理而烦脑过. 但是，他深信那个定理的正确性，虽然他可能忘记了他曾是怎样证明过它的，或者，他的证明方法(可能)不便公诸于众.

我们可以把我们对程序段(6)的论断称为，譬如说，“线性查找定理”．有了这样的名字，就易于自觉地使用它．

其次，就我所知，现在还没有象我们在上面所解释的那种类型的公认定理．但我们不必为此惊奇，之所以还没有这种定理，乃是因为我们研究的对象——程序——尚未定型．程序员所处理的对象(程序)，远没有象平面几何所处理的对象那样的定型．一方面，直觉能力很强的程序员，可能是一个善于尽可能把自己局限于使用他所熟悉的那些程序结构的人，但当要他构造某种不一般的程序时，他就变得谨小慎微了．所以，为定型的程序设计格式，寻找一组相应的定理或许是一项有意义的工作．

第三，上例中的证明所需的长度本身是一个不可忽视的警告．当然，一个较高明的数学家可能作出更短更妙的证明．但就我个人的意见来说，我倾向于从这种长度作出相应的结论，即程序设计比一般想象的要难得多．也就是说，我们应该恭恭敬敬地把这种证明的长度解释为一种劝告，即我们应尽可能地使用一些简单型的结构，并且尽可能回避那种令人讨厌的“万能结构”．

4.3. 抽象

在现阶段，我发现难于把抽象的作用说得非常清楚，部分原因是它遍及我们的整个论题．考虑一个算法和它所能引起的各种计算：从这些计算开始，抽去每次计算所处理的特殊值，余下不变的东西就是这个“算法”*自身．“变量”的概念是对它的现行值的一种抽象．曾有人提醒我注意（遗憾得很我忘了他的名字），人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华．对于循环使用数学归纳法是对

* 一个“算法”可认为是它所能引起的各种计算的共同特征．——译注