PLZ语言的

微处理机程序设计概论

中　册

Cornell大学　R. Conway

D. Gries

Zilog 公司　M. Fay

C. Bass

1979

北京自动化技术研究所译

一九八一年二月

中 册 目 录

第三章　程序的开发

第1节　各开发阶段

我们的任务是为解决某项问题而写程序。假设有了一个初步的问题描述。我们就要设计一种能用计算机解决该问题的方法。然后用某种程序语言非常准确地描述该设计。这个过程可分为四个阶段。

1、澄清问题的要求。

2、设计一个程序对策。

3、规定关键性的数据结构。

4、书写程序语句。

虽然这几个阶段大体应该是按照上面所列的顺序进行的。但是一般都存在着大量的重叠和交叉。而且特别对一些小问题。这几个阶段是很难划分开的。然而每个阶段毕竟表示一个必须执行的各不相同的功能。

至少在开始时。当你在努力学习编程语言的细节时。第四个阶段看来可能是最麻烦的。通过实践积累了经验之后。你将发现。其它三阶段如果做得恰当。则编写程序语句就是水到渠成了。它仍可能是费时的。但并不是关键的阶段。

1.1 问题的澄清

奇怪的是。问题的澄清是过程的一个主要阶段。给问题以清晰确切的说明。这看起来似乎是不成问题的。但实际上这种情况很少。而恰在这一点上比其它任何情况都更多地使编程归于失败。对确切的要求产生轻微的误解。因而写一个解决错误问题的程序。这是很容易发生的。甚至在一本程序设计教程中。说明问题的人（可以推测是）既懂得什么问题可以编程序。选定的问题的难度也仅以教学的需要为限。

即使这样。这个澄清阶段也是相当艰难的。实际的问题通常都是由既不能确切地肯定他想要做什么。更不懂计算机是怎样解决问题的人提出来的。而问题的定义通常仅仅在程序员那方面提出的前后一贯的和明确的问题得到回答后才变得确切。

澄清阶段对话的实质部分可归结为三点：

1、输入。它的格式和顺序是什么？输入量的限度是什么？以及怎样识别输入结束？对输入值的限制是什么？

2、输出。输出的内容、格式及顺序是什么？什么样的标题合适？预期的输出量的限度是什么？

3、错误。程序应当防止什么类型的错误（在输入及处理过程两方面）？当它们出现时应采取什么措施？哪些说明是有保证的。哪些只是力争做到的（由程序检测）？

例如：假定给你一个下述的问题：

（1，1a）写一个计算一串数字之和的程序。这个说明只给出了一个程序的目标的一般概念。需要更详细的资料才能开始设计该程序。例如：

1 a．数据可以用 GETINTEGER 语句所能接收的格式给出吗？

b、可以有多少数值？数值串的结束如何识别？

c、预期的数值大小与类型是什么？

2 a、总和以什么格式显示？

b、应提供什么识别的标题？

3 a、对于违反了 1 c 的规定的数值应采取什么措施？

b、如果数值的量违反 1b 中的规定，应采取什么措施？

获得了这些问题的答案之后。你就可以将（1.1a）明确表达如下：

（1.1b）写一个程序。计算一个表中以GETINTEGER能接受的

格式给出的正整数之和。数据表的结束以两个连续的—999 值表示。输入错误应被检查出来并通过给出部分值（partial）及终止字符作出报告。错误的输入应从总和中排除。终止时任何一个不合规则的情况都将导致一个错误信息。结果应在错误表（如果有的话）的后面用三行给出。

```
SUM OF POSITIVE INTEGERS
n VALUES INCLUDED
SUM IS s
```

作为第二个例子。考虑一个简单的正文处理问题：

（1.1c）写一个程序。从一个字表中删除重复。（1.1c）澄清后可以表述如下：

（1.1d）写一个打印字表的程序。按给定的顺序。每行一个字。但除去任何一个前面表中已出现过的字。数据的格式为一个整数 n（$0 \leq n \leq 100$），后面跟n个字。每个字是一个加引号的字符串。每个字必须包含从1到20个字符。每个字只由字母A～Z组成。不允许有数字、空白。或特殊字符。输出表示如下：

```
WORD LIST WITHOUT DUPLICATES.
m ENTRIES
first word
second word
...
```

在标题中所给的m值。表示在已排除重复和不恰当的字之后该表的最后长度。在表的长度上有任何困难问题时即提出报告。但只要有可能就要产生出一个表来。

（1.1b)和(1,1d）预先考虑了程序员必须面对的许多困难。

如果有了这种形式及这样详细的描述。程序就被说明得很清楚。写起来就没有困难了。但一般说来。（1.1a)及(1.1c）是你可以期望得到的初始问题说明的例子。而（1.1b)及(1.1d）是你必须通过提出正确问题而得到的结果的例子。在一个编程教程中。你的作业更象(1.1b)和(1.1d)，但有时你也要面对象(1.1a)和(1.1c）这样的问题。

只要有可能就要设法得到输入实样 及相应的输出。英语是令人失望的含糊不清。一个具体的例子常能澄清（或排除）冗长的描述。

当一个问题刚开始的时候。你不要太多地想你所要编的程序。直到所有的细节要求都绝对清楚之后再把精力集中在程序上。构成几个输入数据集。画出相应的输出结果。设计这个数据实例是为了测试及增加你对问题各细节的理解。而不是它的一般性质。构造输入数据及手工计算看起来是耗费时间的。而这样做。你实际上是为解决问题而执行了一个算法。这样。当你集中精力于理解问题时。也是在为设计解题而工作。

当你研究问题的时候。你要提问题。像“如果输入数字不正确怎么办？”。“当该特定数字为零时怎么办？它是正确还是不正确及我该怎么做？”。以及“在这里问题的说明似乎含混不清。我怎么办？”在编写程序以前，提出这些问题并得到回答是很重要的。在编程之前，正确地理解问题是极其重要的。没有这样的理解，程序决不可能正确。

对一个重大课题的澄清，作为开发的第一步就能充分完成的情况是少有的。课题要求中的一些必须的细节，常常只是在后面阶段的详细执行着手之后，你才会发现。例如，只是当你企图确定数据结构及需要一个特殊值用以作为数组长度时，才发现你还缺乏有关数据量的某些细节。或者当实际书写检查某个错误的语句时，你发现你不知道

怎样处理这个错误。这两种情况都说明，第一阶段并没有完全完成，还需要做更多的工作。

1．2解题策略的设计

这确实是编程过程最难以提出一般忠告的部分。在编程中创造性的发挥大部分集中于这个阶段。在这里，我们可以给一些提示，告诉你在什么地方有可能产生新思路，并介绍一些在你一旦有了这样的新思路之后发展它们的有用过程。但我们承认，我们是在你最需要帮助的地方给你最少的帮助。我们向你推荐 POLYA 的典范著作《How To Solve It》。

一个有用的策略是，在一开始不考虑计算机。设计出如何用手工解决问题。假设你得到的输入数据是在卡片上，一次只能看到一项，而且只能按规定的顺序看。进一步假定，提供给你用的“草稿纸”是大量的小卡片（一张上一个数字或一个字符串）。你可以在上面写、涂、重写数字。如果你能够在这些限制下，用手工解决问题，你就可以用程序描写这个方法。

重要的事情是要把解题方法的计划过程与用编程语言描述该解题方法的任务区分开。一旦区分开之后，编程过程中的复杂而困难的部分将是计划工作。例如，如果你不能写一个程序按递增顺序对一个数表进行“分类”，这大概是由于你不能设计出如何用手工的方法系统地做这件事，而并不是因为你不知道 PL/Z 。相反地，如果你的分类程序是有效的，那是因为你有一个聪明的计划，而不是因为在 PL/Z 中聪明地描述了一个普通的计划。

1.2.1 算法

一个程序的计划通常称为“算法”。一个算法是一系列的步骤，

通过这些步骤的执行，能解决一个问题。

无论用什么语言写算法，标记法（notation）使这作算法能为读者所理解。通常，我们假设读者方面有一定的基础和知识，并采用可以使算法更准确和简炼的技术词汇。算法是写给人看的，而不是写给计算机读的，因此，一般不用编程语言写。对于一个特定的问题，无论是哪种语言，只要是最好的，我们就采用或者发明出来。通常它是英语、算术式及专对一定的问题领域的技术词汇的组合。

程序就是已被翻译成编程语言的算法。我们把一个程序看做为一个算法的具体实现。问题在于，算法是用一种非正式（但仍精确）的类似于英语的语言首先写出来的，然后才产生出程序。借助的是一个翻译过程而非创造过程。

一个算法可以翻译成不同的编程语言。然而，通常我们事先了解我们所要使用的语言，我们倾向于使算法靠近该语言的便利的操作。在本书情况下，我们了解我们的目的是PLZ 程序，我们以英语和PLZ的混合开始着手，在方便之处使用PLZ的术语，但是只要更为方便，也发明术语。一般说，发明的术语是比PLZ 语句“更高级”的。例如如，我们可以说“Swap A and B”（即：交换A和B）。因为PLZ没有单独的语句来“交换”。最终它必将被翻译成三个赋值语句：

```
T:=A ;    A:=B ;    B:=T
```

在算法中，我们可以用“find” “Solve” 和“Search”这样一些术语，它们比程序语句高级，但它们最终仍必须表示为一段程序实体。在我们的例子中，当我们说明如何将算法逐步地有系统地转换为程序时，我们用大写字母写PLZ 短语，而用小写字母表示还未被被翻译成PLZ 的短语。

有时我们用PLZ 中的等价语句简单地代替英语短语，但一般我

们在程序中用英语短语作为注释语句，后面跟着的是执行该注释所描述的任务的缩格PLZ 语句。

1.3 数据结构的选定

必须作出的关键性决定是：问题的什么要素将需要存贮。要对变量加以规定，包括名字、维数、类型属性以及对各变量之间逻辑关系的描述。这些信息显而易见大多数来自对问题的澄清，但数据结构的选定同样重要地取决于程序策略。

如程序策略选定之后，就要规定数据结构。有时最后的决定要延迟到大部分实际程序已写完之后。一般情况下，尽可能地拖延数据结构的最后决定。这样可减少必要的返工。细节常常要在程序写完之后才变得清楚。

变量间的相互关系是十分重要的，必须通过说明的顺序强调出这些关系。请重读第二章第一节，它能帮助你在写最后程序语句之前，准确地写出这些关系。

1.4 程序语句的编写

关于这个编程阶段的细节，需要说的比较少。如果第二阶段完成得很恰当，那么你就有了一个描述该程序每个段的作用的详细注释的轮廓。第三阶段，已经定义了那些段作用的对象。所剩下来的一切便是将这些详细说明转换成PLZ 语句（或所使用的任何编程语言）。

然而，正如早已预料的那样，在写详细程序语句的同时，你常常发现数据结构不完全正确，这样你必须返回到第三阶段。你也可能发现需要返回到第二阶段，改变程序的组织结构。你能完成详细的翻译而没有一次返回到第一阶段澄清问题定义的某一方面，是十分罕见的。

而返回可能反过来又要求改变程序的组织结构或数据结构。返工及进行修改是一件巧妙的事情。困难在于如何保证你已识别了这种改变的全部复杂牵连并完成了必须的调整。

很少有人能使用穿孔键及终端熟练地编写。通常的作法是用手写程序语句，然后把手写稿用键盘输入。键盘打入的准确性，在很大程度上取决于手写稿是否字迹清楚。我们常看见学生们从一个勉强可以辨认的手稿进行键盘打入。这样不可避免地将错误打入程序。如I错为1，2错为Z，O错为零。变量名拼错，句子遗漏，插入句插在错误的地方。如果能想一想这些错误一旦写进程序，将它们再查出来并排除掉有多困难，那么花费时间和精力来首先防止它们就是值得的了

第2节　程序开发的方法

编程过程中最艰苦的部分无疑是程序的计划及设计，而不是将那些设计转换成编程语言的语句。学生在编程中发生困难时常常认为他们的课题与语言距离远，而实际上是他们不善于发现一个与该程序相宜的设计。编程中的这种情况是不少的。例如，学习国际象棋中各种棋子的走法比学习如何用这些走法去赢一盘棋要容易些。学习几何学的公理和定律比学习如何用这些东西去构成记法要容易。这些都是一个叫做“综合”的过程的实例，它是需要最多的智力活动的一种。在任何形式下它都需要一定的见识和创造力，而且据我们所知，在确切地描述如何去搞发明创造方面，还没有一个人获得巨大成功的。

另外，有几个通用的方法看来是有用的。它不过是考虑编程的系统化的方法，它们的确不能把整个编程过程简化为一套能保证得到正确而有效的程序的例行步骤。一开始，你会觉得这些方法没多大用途，但当你写过一些中型程序之后再来重谈这一节，就会更有收获，并能

帮助你熟练地掌握解决更困难问题的方法。

我们先简要地介绍有关程序开发的三种一般的概念，然后在第三节中用几个例子来加以说明。这些概念如下：

* 从问题说明到程序完成的系统的“自顶向下的开发”。
* 用比单个PLZ 语句更大的单位来观察程序。
* 在不同程序之间探求相类似的地方。

这些概念都已在前面各节中介绍过。这里只是一个复习和小结。

8.1 “自顶向下”的开发

我们第一次介绍“自顶向下”的方法是在第一章 6.5 节及 7.4 节中。在那里我们讨论了条件语句和嵌套循环。这个概念简单地说，是一种从顶级到最低级一级一级地开发程序的方法。每一级可以看成是一个单独的任务。实质上是把相同的思考过程在每一级上重复。

在第二章第二节，我们按不同的级分析了程序的结构，并说明了把每一级作为一个顺序的或“几乎是顺序的”程序进行设计为什么是有用的。现在我们要用程序开发中必须遵循的一定的步骤顺序，再次说明这些概念。这实际上是一个开发程序的“程序”：

1、确定一个简单的步骤顺序，使这些步骤按顺序的执行能解决提出的问题，据此找出程序的顶级。每一步都是下述之一：

用英语写语句

循环

抉择二者之一

赋值语句

输入或输出语句

2、把顶级上每一个“复合”步看作是一个单独的问题，并重复该过程。就是说，对每一个复合步的实体制订一个计划，有如它

最初就已被指定是一个问题一样。

3、一级一级地继续这个过程，直到再也没有“复合步”的一级为止。

4、在整个这个过程中，对每一个使用的变量保持跟踪（在一个单独的表格中），注明它们的值的类型及在程序中的作用。在将最低级展开成为非复合步后，返回来填写所需的变量说明。

5、最后，写 PROCEDURE，MODULE，SINCLUDE，ENTRY 及 END 各封闭行。

由于上述2和3步中程序的每个复合步都扩展成下一级中更为放大的细节，这个过程有时称为“按级细化的程序开发”。

我们试图通过第三节的例子，使你相信这是把一个难以解决的问题分解成易于管理和识别的子问题的一种有效方法。

作为一个实际情况，一个循环或交换语句的子成分可以是 PLZ 语句，因而将不需要进一步细化。例如

```
IF VALUE < 0 THEN
        PUTSTRTNG(CONOUT,#'%r lmproper
                         VQLUE(negative)%r/')
FI
```

另一方面，当 THEN 后的结构是复合的时：

```
IF condition THEN
    Body
FI
```

则把本体作为需要进一步细化的单独一级来处理是有用的。

2.2 基本程序单位

随着编程经验的积累，你应掌握更高级的思考能力，用比PLZ语句大得多的功能单位来思考问题。例如，一个许多程序都要执行的公共功能是通过读一个数据表的值，装入一数组的各个元素。做这件事，我们有三种不同的方法，都包含循环（第一章7.5节）。而在你为这个具体情况寻找恰当的方法而为难之前，你要把它作为单个任务来考虑：

```
Load A(1:N) from data
```

用一个注解语句把它作为单个任务来描述常常是有用的：

```
!Load A(1:N) from data !
```

然后你可以进一步确定怎样具体地执行“装入”，并在注解语句之下，缩格地写出详细的PLZ语句。例如：

```
!Load A(1:N)from data !
N := GETINTEGER(DISKIN)
I := 1
DO
    IF I > N THEN EXIT FI
    A(I) := GETINTEGER(DISKIN)
    I += 1
    OD
```

另一个公共任务是在一个数组中找最大值。你可以考虑通过这样的“步”：

```
!Set AMAX to maximum value in A(1:N)!
```

然后再考虑如何实际地执行该任务。实际上，这是一个极常用的公共任务，有的编程语言可能已有一个单个语句来执行它。PLZ不是这样，你必须照下面那样写：

```
!Set AMAX to maximum value in A(1:N)!
   AMAX := A(1)
   I := 2
   DO
      IF I > N THEN EXIT FI
      IF AMAX < A(I) THEN AMAX := A(I)FI
      I += 1
   OD
```

（在第四章里描述的“过程”，你实际上可以用来给该语言增加操作，因此你能够有一个单个语句来实现这个任务。）而在这点上，最初你作为单个高级步来考虑时，是不关心更准确的细节的。但在这里关键在于，不管如何做到这一点的细节如何，你都要在开始时把它看作是一个单个的高级步骤。

学会用比PLZ 语句更大的单位进行思考，你也将按比例地增大你能编程的问题的规模。这是一个自然过程，而且是迟早要发生的。在学习任何技巧之时都会发生。例如，当你第一次学习用手动换挡开小汽车时，你所关心的详细步骤是：

1、关油门的同时解脱离合器。

2、移动齿轮变换杠杆从位置 i 到位置 j 。

3、开油门使引擎和轴的速度同步。

4、挂上离合器。

这些步骤不久就变成一个整体，不加思考地操作，你把它当作一个动作：

从 i 挡变换到 j 挡

在这个更高的级上，你可以集中学习各种离合器的功能以及怎样

使用它们来达到不同的目的。最后你站在更高一级上考虑，强调的是目标而不是方法：

从静止加速到每小时几哩。

在这一级上，你忽略掉很多细节，象车辆配备的是手动还是自动换挡；你集中精力于你要实现的事，而不在于操作的技术细节。

学习编程序也很相似。各个PLZ 语句相当于踩离合器踏板及移动变换杠杆这样的细节。直到那些细节成为第二天性，你对“装载数组”及“寻找最大值”这样的步骤已运用自如时，你才能为一个非同寻常的问题编程序。

2.3 程序间的相似性

编程的第三个特性，也是使创造的任务可以办得到的特性，是这样一个事实：完全独特的程序是很少的。实际上只有你的第一个程序是完全独特的。而那以后的每个新程序都要借用它以前的程序。有时是完全的借用——你可以引用（或照抄）它某个程序中使用的一组语句（在第四章中讲述的“过程”大大增加了程序段的这种互换能力。）有时这种借用只是在概念上的，你再次使用以前的程序的概念。

在程序之间有两种不同的相似情况。可以在结构上类似，但完成的任务不同。例如，考虑下面两段程序：

```
INTERNAL
        SUM  INTEGER  := 0
        ITEM  INTEGER
        NBROFITEMS,  ITEMNUMBER  BYTE
        ...
        NBROFITEMS  := BYTE  GETINTEGER( DISKIN)
        DO
```