第5章

详细设计

教学提示：第4章介绍了概要设计的有关知识，本章将介绍详细设计的内容，主要包括详细设计的任务与原则、设计方法以及规格说明及评审等。

教学目标：理解详细设计的设计任务和设计原则，掌握详细设计的方法和常用工具的使用，了解详细设计的规格说明书的内容和评审。

通过前面的学习了解了软件概要设计的主要任务是以比较抽象概括的方式提出解决问题的办法。从软件工程的观点看，在使用程序设计语言编制程序之前，还需要确定每个模块的具体算法，可以用程序流程图、N-S图、PAD图或伪码给予清晰的描述，以便在编码阶段直接翻译成在计算机上能够运行的程序代码，这就是详细设计的内容，因此详细设计(Program Design)也称过程设计或程序设计。详细设计阶段的任务就是把解法具体化，但这个阶段不是真正地编写程序，而是设计出程序的详细规格说明。这种规格说明的作用十分类似于其他工程领域中经常使用的工程蓝图，它们应该包含必要的细节，使程序员可以根据它们写出实际的程序代码。

5.1 详细设计的任务与原则

详细设计指在概要设计提供的文档及相关设计结果的基础上进一步确定如何实现目标系统。这一阶段所产生的设计文档将直接影响下一阶段的程序质量。为了保证软件质量，软件详细设计既要正确，又要清晰易读，以便于编码的实现和验证。

5.1.1 详细设计的任务

详细设计是在概要设计的指导下根据目标系统逻辑功能的要求，结合实际情况，详细地确定目标系统的结构和具体的实施方案，即对系统的各组成部分进行细致、具体的物理设计，使系统概要设计阶段所做的各种决定具体化，从而在编码阶段可以把这个描述直接翻译成用某种程序设计语言书写的程序。

详细设计不同于编码(Coding)，其目标是不仅为软件结构图(SC图或HC图)中的每个模块确定使用的算法和数据结构，而且用某种选定的表达工具给予清晰的描述，更重要的是使设计的处理过程尽可能简明易懂。

详细设计阶段的主要任务如下。

1. 模块的逻辑结构设计

逻辑结构设计是结合所开发项目的具体要求和对每个模块规定的功能开发出模块处理的详细算法，并选择某种适当的工具加以精确描述。

编码是根据详细设计的逻辑结构进行的程序设计，良好的详细设计是获得可维护性强、可理解性好的高质量软件的前提。

2. 模块的数据设计

模块的数据设计是为在需求分析阶段的数据对象定义逻辑数据结构，并且对不同的逻辑数据结构进行不同的算法设计，以便选择一个最有效的方案，同时确定实现逻辑数据结构所必需的操作模块，以便了解数据结构的影响范围。数据设计包括数据结构设计、数据库结构设计和文件设计等。

由于数据结构会直接影响程序结构和过程复杂性，因此会在很大程度上决定软件质量。

3. 模块的接口设计

接口设计是分析软件各部分之间的联系，确定该软件的内部接口和外部接口是否已经明确定义，模块是否满足高内聚和低耦合的要求，模块作用范围是否在其控制范围之内等。

4. 模块的测试用例设计

要为每个模块设计一组测试用例，以便在编码阶段对模块代码（即程序）进行预定的测试，模块的测试用例是软件测试计划的重要组成部分，通常应包括输入数据和期望的输出数据等内容，其要求和设计方法将在第 8 章详细介绍，这里需要说明的是，由于负责详细设计的软件人员对模块的功能、逻辑和接口最清楚，所以可以由他们在完成详细设计后提出对各个模块的测试要求。

5. 模块的其他设计

根据软件系统的具体要求，还可能进行以下设计：网络系统的设计、输入/输出格式的设计、系统配置的设计等。

6. 编写详细设计说明书

在详细设计结束时，应该把上述结果写入详细设计说明书，并对详细设计说明书进行评审。如果评审没有通过，则要再次进行详细设计，直到满足要求为止。通过复审的详细设计说明书将形成正式文档，交付给下一阶段（编码阶段）并成为其工作依据。

5.1.2 详细设计的原则

由于详细设计是为程序员编码提供的依据，因此在进行详细设计时应遵循以下原则。

1. 模块的逻辑描述清晰易懂、正确可靠

详细设计的结果基本决定了最终的程序代码的质量。由于详细设计的蓝图是给后续阶段的工作人员看的，所以模块的逻辑描述正确可靠是软件设计正确的前提。详细设计结果的清晰易懂主要有两方面的作用：一是易于编码的实现，二是易于软件的测试和维护。

如果详细设计易于理解，又便于测试和排除所发现的错误，则能够有效地在开发期间消除在程序中隐藏的绝大多数故障，使得程序可以得到正确稳定的运行，极大地减小运行期间软件失效的可能性，大幅提高软件的可靠性。

2. 采用结构化设计方法

改善控制结构，降低程序复杂程度，提高程序的可读性、可测试性和可维护性。采用自顶向下逐步求精的方法进行程序设计，一般采用顺序、选择和循环 3 种结构，以确保程序的静态结构和动态结构的执行情况相一致，保证程序容易理解。

3. 选择恰当的工具进行各模块的算法描述

算法表达工具可以由开发单位或设计人员选择，但表达工具必须具有描述过程细节的能力，进而可以在编码阶段直接将它翻译为用程序设计语言书写的源程序。

5.2 详细设计的方法

使用不同的详细设计方法会影响详细设计的可读性、易理解性以及程序代码的质量和效率，从而进一步影响程序代码的可维护性。因此掌握详细设计的方法是详细设计的关键。

5.2.1 结构化程序设计技术

结构化程序设计技术是软件工程发展史中的重要成就之一。结构程序设计技术使得程序中的控制可以任意地、不受限制地转变成有限的固定结构，使程序的分支减少，易于阅读和理解，并易于测试，提高了软件开发的生产率和质量。

1. 结构化设计技术的形成

结构化设计技术是从对“取消 GOTO 语句”的争论开始而逐步形成的。GOTO 语句是程序设计语言的一个控制成分，它在给编程带来控制流程转移的方便与灵活、提高程序执行效率的同时，也使程序的可理解性降低。

20 世纪 70 年代以前，人们设计的绝大多数软件都是非结构化的。早在 1963 年，针对当时流行的 ALGOL 语言，Naur 指出在程序中大量、没有节制地使用 GOTO 语句会使程序结构变得非常混乱，但这在当时并没有引起人们足够的重视。随后，著名的荷兰科学家 E. W. Dijkstra 在 1965 年的 IFIP 会议上指出：可以从高级语言中消除 GOTO 语句，

程序的质量与程序中所包含的 GOTO 语句的数量呈反比。他提出了结构化程序的概念，认为 GOTO 语句太原始，引用太多会使程序一塌糊涂。由此引发了一场关于 GOTO 语句的争论。

1966 年，Boehm 和 Jacopini 在一篇文章中证明：只用“顺序”“选择”和“循环”3 种基本控制结构就能实现任何单入口、单出口的程序设计。

1972 年，IBM 公司的 Mills 进一步提出，程序应该只有一个入口和一个出口，采用自顶向下、逐步求精的设计和单入口、单出口的控制结构。改善程序设计控制结构可以降低程序的复杂程度，从而提高程序的可读性、可测试性、可维护性。

我们知道，程序的执行是动态的过程，如果不加限制地使用 GOTO 语句，则会使程序变得晦涩难懂，增加程序出错的概率，降低程序的可靠性；GOTO 语句会影响程序结构的清晰度，破坏程序基本结构的单入口、单出口原则；由于 GOTO 语句将程序串成一体，因此会给程序的测试和维护造成困难，在修改程序时也会引发副作用。

通过下面的一个实例，我们可以体会到非结构化程序设计技术的缺点。

【例 5.1】 编程实现：从键盘输入 3 个数，判断最小值并输出。

方法一：用非结构化设计技术实现。

算法描述如图 5.1 所示。

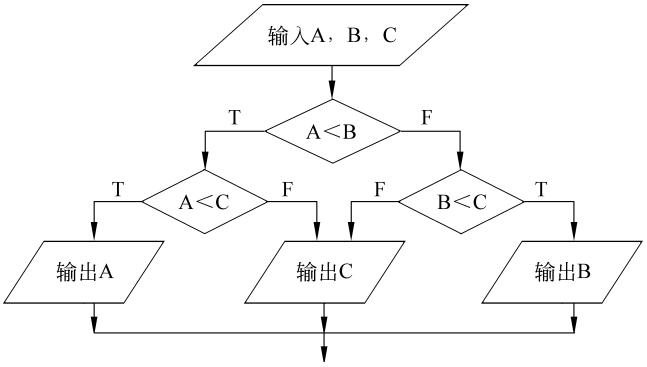


图 5.1　求 3 个数中最小值的非结构化设计的算法描述

具体代码如下。

```
#include<stdio.h>
main()
{
    int a, b, c;
    scanf("%d %d %d",&a,&b,&c);
    if (a<b)
        goto label2;
    if (b<c)
        goto label1;
    label0:
        printf("min=%d\n",c);
```

```
        goto label4;
    label1:
        printf("min=%d\n",b);
        goto label4;
    label2:
        if (a<c)
            goto label3;
        goto label0;
    label3:
        printf("min=%d\n",a);
    label4:
        getch();
}
```

方法二：用结构化设计技术实现。

算法描述如图 5.2 所示。

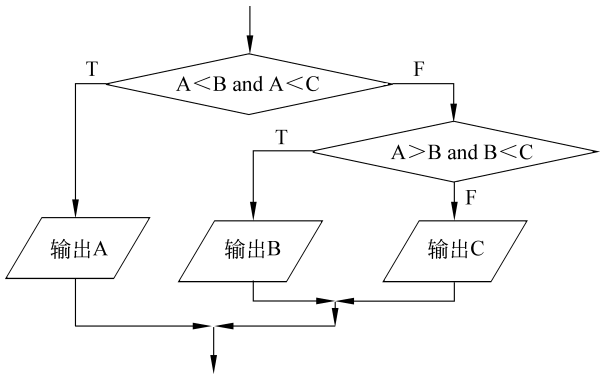


图 5.2　求 3 个数中最小值的结构化设计的算法描述

具体代码如下。

```
#include<stdio.h>
main()
{
    int a,b,c;
    scanf("%d %d %d",&a,&b,&c);
    if(a<b && a<c)
        printf("min=%d\n",a);
    else if(a>=b && b<c)
        printf("min=%d\n",b);
    else
        printf("min=%d\n",c);
    getch();
}
```

比较上述两个算法可以看出，前一算法在程序中使用了多个 GOTO 语句，在阅读程序时，顺着每个 GOTO 语句的流向必须一气呵成才能理解程序，程序的可读性很差；第二种算法程序逻辑结构清楚，条理分明。由此可以看出，GOTO 语句在程序中是可以消除的，至少它不是必不可少的。

难道一定要从高级语言中消除 GOTO 语句吗？实践证明，在有些算法设计中完全不用 GOTO 语句比用 GOTO 语句实现的可读性还要差，例如在查找结果时、文件访问结束时、出现错误情况时。要尽可能快地从当前程序跳转到一个出错处理程序，使用布尔变量和选择结构实现不如用 GOTO 语句简洁易懂。

【例 5.2】 假设要在表 A[1]…A[m]中找出给定值 X，若 X 不出现在表中，就作为附加的表元素将给定值 X 插入表中。另假设数组 B，其中 B[i]中存放已经检索 A[i]的次数。

程序设计如下。

```
    for i:=1 to M do
        If A[i]=x then goto 10;
    I:=m+1;
    M:=I;
    A[i]:=x;
    B[i]:=0;
10: B[i]:=B[i]+1;
```

如果此例不用 GOTO 语句，则也可用若干方法表达上例，但都要求更多的计算，而实际上也没有达到更清楚的目的。人们常用这样的例子捍卫 GOTO 语句，读者也可以写出一些等价的例子加以比较。

可以看出，使用 GOTO 语句后，上例的概念表达得非常清楚，使用方便，而且易于掌握。因此在有些情况下也不必消除 GOTO 语句，特别是在提高程序执行效率且又不大影响结构的同时，可以适当地使用 GOTO 语句。另一方面，用一大堆控制语句代替 GOTO 语句，用户也难以掌握。

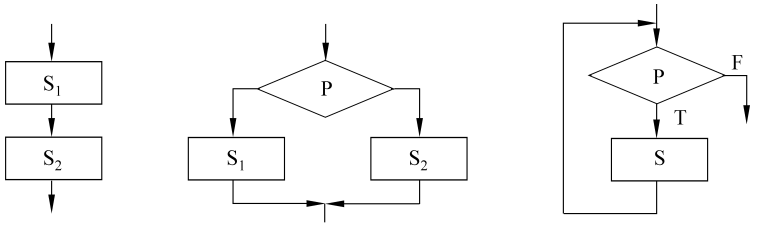
综合以上情况可知，GOTO 语句具有二重性。一直到 1974 年 Kunth 发表了“带 GOTO 语句的结构程序设计”一文后，才平息了这场旷日持久的争论。

2. 结构化设计技术的概念

结构程序设计的经典定义为：如果一个程序的代码仅通过顺序、选择和循环这 3 种基本控制结构进行连接，并且每个代码块只有一个入口和一个出口，则称这个程序为结构化的。

3 种基本结构的流程如图 5.3 所示。

然而结构化设计的经典定义过于狭义，结构化设计的重点不是关注有无 GOTO 语句，而是应该把注意力集中在程序结构方面，使设计的程序容易阅读、容易理解，以推动软件设计方法的发展。在某些情况下，为了达到上述目的，反而需要使用 GOTO 语句，因此出现了结构化设计的定义。

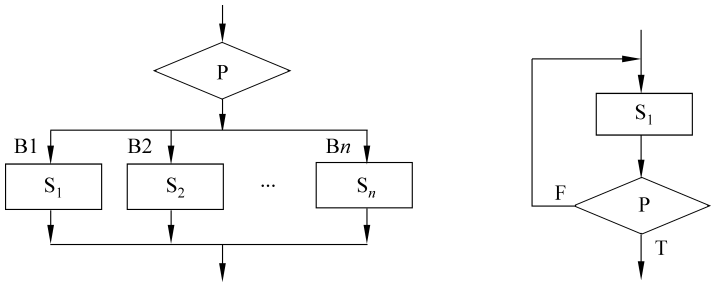


(a) 顺序结构　(b) 选择结构　(c) 循环结构

图 5.3　3 种基本控制结构

结构程序设计是尽可能少地用 GOTO 语句的程序设计方法。最好仅在检测出错误时才使用 GOTO 语句，而且应该总是使用向前的 GOTO 语句。

虽然从理论上说只用上述 3 种基本的控制结构就可以实现任何单入口、单出口的程序，但是为了使用方便，通常还允许使用扩展的控制结构，包括直到型循环(Do-Until)和多分支选择结构(Do-Case)，它们的框图表示如图 5.4 所示。



(a) 多分支选择结构　(b) 直到型循环

图 5.4　扩展的控制结构

如果在允许使用 3 种基本控制结构的基础上，还允许使用多分支结构和 Do-Until 循环结构，则称为扩展的结构化设计；如果在扩展的结构化设计的基础上，还允许使用转到循环结构下面的受限的 GOTO 语句(如 leave 或 break)，则称为修正的结构化设计。

通过对 GOTO 语句的讨论，人们认识到单纯强调程序效率和只关注是否使用了 GOTO 语句的片面性，认识到设计简明易懂的算法的重要性，这实际上也就形成了一种新的设计思想、方法和风格。

综上所述，结构化程序设计的基本内容可归纳如下。

① 程序的控制结构一般采用顺序、选择、循环 3 种结构构成，以确保结构简单。

② 使用单入口、单出口的控制结构。

③ 程序设计中应尽量少用 GOTO 语句，以确保程序结构的独立性。

④ 采用自顶向下、逐步求精方法完成算法设计。结构化程序设计的缺点是存储容量和运行时间会增加 10%～20%，但可读性、可维护性好。

5.2.2 详细设计基础

在详细设计中用于描述过程算法的工具有图形、表格和语言三类。这些工具虽然能明确指出算法的控制流程、处理过程和数据组织等细节，但都有各自的优缺点，在设计时可针对不同的情况选用，甚至可以同时采用多种工具描述设计结果，为编码阶段提供依据。

1. 程序流程图

程序流程图(Program Flow Chart，PFC)又称程序框图，它是传统的、使用最广泛的描述程序逻辑结构的方法，也是软件开发者最熟悉的一种算法表达工具。PFC 独立于任何一种程序设计语言，能比较直观和清晰地描述过程的控制流程，易于学习和掌握。因此，程序流程图是软件开发者最普遍采用的一种工具。图 5.5 所示为程序流程图中使用的基本符号。

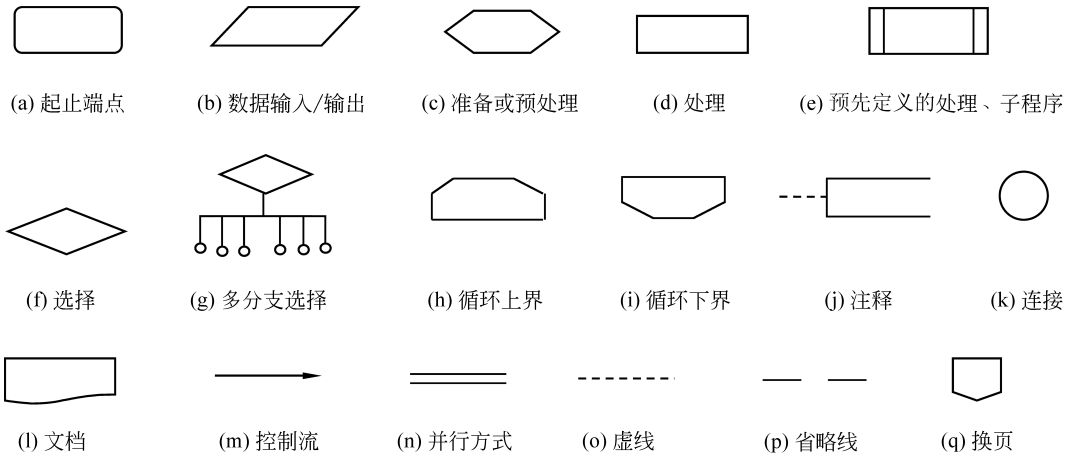


图 5.5 程序流程图的基本符号

【例 5.3】 分析图 5.6 所示的基于嵌套的结构程序设计流程图算法。

经过分析可知：图 5.6 描述的算法由 3 个顺序的语句单元组成，即 a、Do-Until 循环和 h；其中，Do-Until 循环的循环体部分又由多分支结构 P1 和 g 组成；而 P1 的每个分支又分别由顺序结构、循环结构和选择结构等组成。由此可见，任何复杂的程序流程图都可由图 5.3 和图 5.4 所示的 5 种基本结构组合或嵌套而成。

程序流程图虽然容易掌握，使用广泛，但总的发展趋势是越来越多的人不再使用程序流程图，主要原因是程序流程图存在许多缺陷如下。

① 程序流程图使用的符号不够规范，使用的灵活性极大，程序员可以完全不顾结构程序设计的精神，不受任何约束随意转移控制。

② 在实际使用中，程序流程图本质上并不具备逐步求精的特点，对于提高大型系统的可理解性作用甚微。

③ 由于程序流程图中的控制流可以随意转向，因此会诱惑程序员过早地考虑程序的

控制流程，而忽略整体结构。

④ 程序流程图不易表示模块的数据结构。

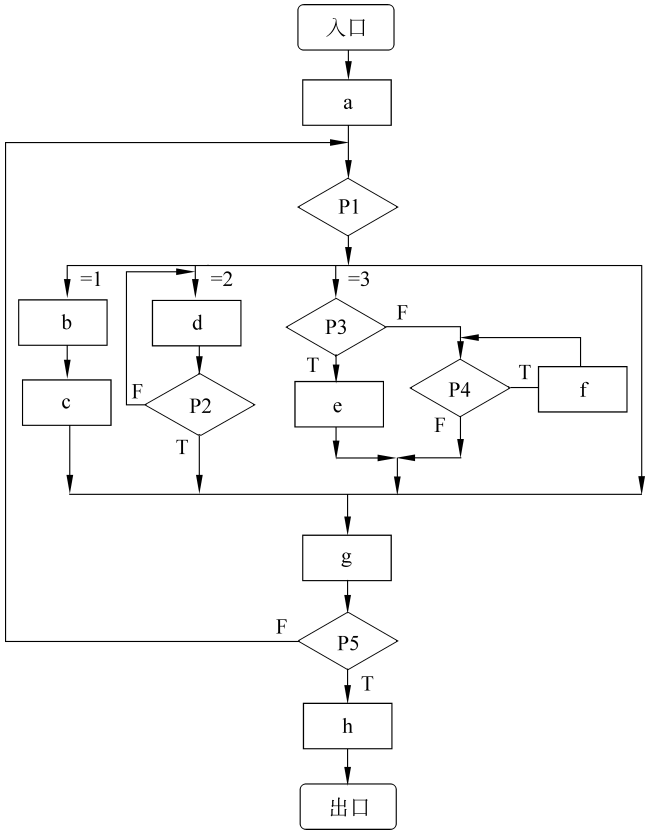


图 5.6 结构程序设计流程图

2. 盒式图

盒式图(Nassi-Shneiderman，N-S)最早由 Nassi 和 Shneiderman 在 1973 年发表的题为“结构化程序的流程图技术”的一文中提出。盒式图强调使用 3 种基本控制结构构造程序逻辑，符合结构化程序设计原则。在 N-S 图中规定的基本图形符号如图 5.7 所示。

【例 5.4】 将图 5.6 所示的程序流程图转换为 N-S 图，结果如图 5.8 所示。

任何复杂的 N-S 图都应由图 5.7 所示的 5 种基本结构组合或嵌套而成。当一个问题较复杂时，对应的 N-S 图会相对较大，这时可以使用带有子程序调用的 N-S 图表示。

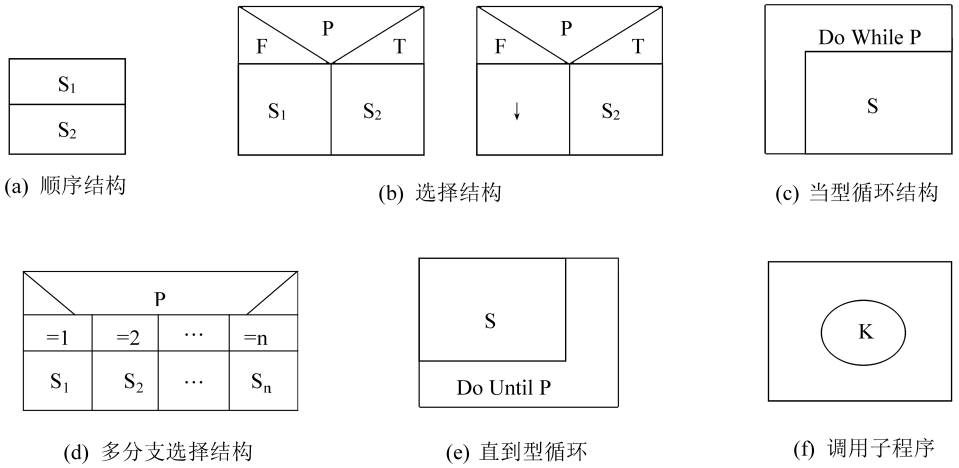
从以上分析可以看出，N-S 图具有如下特点。

① 逻辑结构表示清晰、准确，每个矩形框都是一个功能域，但在多分支选择结构中的条件取值例外。

② 由于取消了控制流符号，不允许随意转移控制，必须遵守结构化程序设计原则。

③ 很容易确定局部数据和全局数据的作用域。

④ 能清晰地表现嵌套关系和模块的层次结构。



(a) 顺序结构　(b) 选择结构　(c) 当型循环结构

(d) 多分支选择结构　(e) 直到型循环　(f) 调用子程序

图 5.7　N-S 图实例

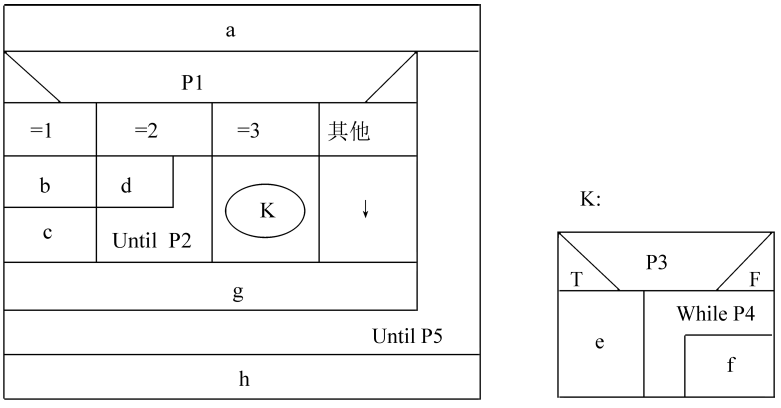


图 5.8　带有子程序调用的 N-S 图

因此，坚持使用 N-S 图可以使程序员逐步养成用结构化的方式思考和解决问题。

3. 问题分析图

问题分析图（Problem Analysis Diagram，PAD）是由日本日立公司二村良彦等于 1979 年提出，由 PFC 演变过来的一种支持结构化程序设计的图形工具。该图用二维树状结构表示程序的逻辑结构。PAD 的基本控制结构图形（基本符号）如图 5.9 所示。

【例 5.5】 将图 5.6 所示的程序流程图转换为 PAD 图，结果如图 5.10 所示。

PAD 所描述程序的层次关系表现在纵线上，每条纵线表示一个层次。把 PAD 图从左到右展开，随着程序层次的增加，PAD 逐渐向右延伸，有可能会超过一页纸，这时对 PAD 增加一种如图 5.9(g)所示的扩充形式。当一个模块 A 在一页纸上画不下时，可在图中该模块的相应位置的矩形框中简记一个 K，再在另一页纸上详细画出 K 的内容，用 def 及双下画线定义作出 K 的 PAD。这种方式可使在一张纸上画不下的图分在几张纸上画出，也可以用它定义子程序。