第八章　目标代码运行时刻环境的组织

在生成目标代码之前需要讨论目标代码运行时所处的环境，目标代码和运行环境之间的关系就像“鱼”和“水”一样。本章首先概述目标代码运行时的软硬件环境，接着讨论源语言程序的构成特点、执行过程及对程序执行过程的描述；对目标代码运行时刻内存空间的管理与分配策略进行了详细介绍，包括运行时刻内存空间的典型划分和 3 种常见的内存空间分配策略：静态存储分配、栈式存储分配和堆式存储分配；讨论了几种代表性高级语言的特点，分别介绍了这几种高级语言非局部名字的访问方法。

8.1 目标代码运行时刻环境

编译器处理源程序的最终目的是要将其转换成目标代码。目标代码生成总是面向特定的计算机系统平台的，生成的目标代码必须能在具体的系统平台上运行，需要与这个系统平台进行交互，能直接使用这个系统平台提供的计算资源。

现代计算机系统采用冯·诺依曼体系结构，存储程序原理和程序控制原理是现代计算机系统的基本工作原理，编制好的二进制代码（或者编译器生成的目标程序）必须事先存放到计算机系统的内存中才能被执行。计算机系统分为硬件系统和软件系统，硬件系统的核心是 CPU，软件系统的核心是操作系统。每个 CPU 都有自己的指令系统，这个指令系统是设计 CPU 时的依据，也是编制或者生成目标程序的依据。目标程序只能由指令系统中的指令构成，CPU 也只能识别与执行其指令系统中的指令。指令由操作码和操作数构成，操作码的选择和操作数地址码的选择是目标代码生成时要解决的首要问题。现代计算机的 CPU 通常都集成了一个寄存器组，寄存器相对于内存来说，其访问速度一般要高出一个数量级。寄存器是计算机系统的宝贵资源，充分利用寄存器以提高目标程序的运行效率是代码生成算法的重要目标。目标代码运行时，操作系统会分配一块连续的内存空间作为它的运行空间。这块内存空间必须能容纳目标代码和目标代码运行时的数据空间（目标代码中指令能访问的空间），组织与利用这块内存空间也是由目标代码来实现的。生成目标代码时需考虑：

（1）目标代码运行时的硬件环境，即 CPU（及其指令系统）和寄存器；

（2）目标代码运行时的软件环境，即操作系统及目标代码运行的内存空间。

目标代码运行时刻环境如图 8-1 所示。面向特定 CPU 及其指令集，如何充分利用计算机系统提供的寄存器资源，生成高质量的目标代码将在下一章讨论。本章主要介绍目标代码如何组织和利用其运行时刻的空间环境。寄存器的分配和指派及内存空间组织最后都体现在生成的目标代码中。

待出版，仅供2020信息技术创新大课使用，如外传！

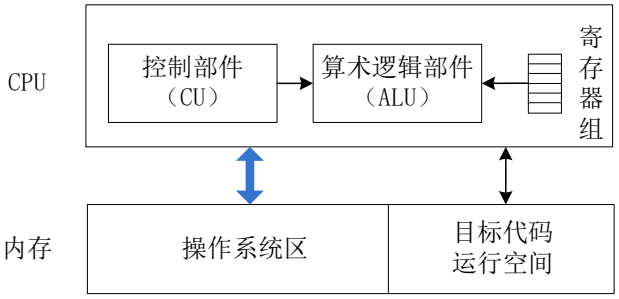


图 8-1 目标代码运行时刻环境

8.2 源语言相关问题讨论

源程序一般由一组过程（或函数）构成，不同的高级程序设计语言，由过程（或函数）构成源程序的方法会有所不同。构成源程序的两个过程（或函数）之间，要么是嵌套的，要么是不相交的。如 C 语言的程序由一个或多个函数构成，函数的定义是相互独立的，每个 C 语言程序中都有一个名为 main 的主函数，执行从 main 开始，main 可以调用其它函数。Pascal 语言是结构化程序设计语言的典型代表，它的程序主体是一个主过程，主过程可以定义其它过程或者函数，主过程中定义的过程又可以嵌套定义其它的过程或者函数。一个 Pascal 语言的例子如图 8-2 所示。在这个程序中，主过程是 sort，第（2）行是主过程的变量说明语句。第（3）-（7）行定义了一个嵌套过程 readarray，第（8）-（11）行定义了一个嵌套函数 partition，第（12）-（20）行定义了一个嵌套过程 quicksort。第（21）-（25）行是主过程 sort 的程序体，每个嵌套定义的过程和函数也都有各自的说明语句和程序体。执行从主过程 sort 的程序体开始，sort 首先调用 readarray，接着调用 quicksort。quicksort 执行时首先调用函数 partition，接着连续两次调用自身，即做递归调用。出现在过程（或函数）定义语句中过程（或函数）名后的标识符是形式参数，如第（12）行中的标识符 m 和 n 就是过程 quicksort 的形式参数。过程被调用时，形式参数会被替换为实在参数。函数和过程的区别在于函数有返回值，返回值也有类型，由函数说明语句确定，如第（8）行最后的 integer 说明函数 partition 的返回值是整型。以下如无特别说明，将过程和函数统称为过程。

过程的一次完整执行，称为过程的一次**活动**。所谓一次完整执行是指执行从第一条语句开始到执行完最后一条语句结束。过程如果还在执行中称它是**活着**的，如果执行完了称它已死亡。过程 P 的一次活动的**生存周期**是指它活着的这段时间。早期的一些程序设计语言不允许过程的递归调用，即不允许同一个过程直接或间接地调用自己。这样程序在执行过程中，不会出现某一时刻有一个过程的多个活动是活着的情况。反之，如果允许递归调用，程序的一次执行可能在某个时刻会有同一个过程的多个活动是活着的。见图 8-3 的示例，考察 P 的活动情况。首先第一个 P 开始执行，P 的一个活动开始，然后它递归调用自己，P 的第二个活动开始，接着 P 又调用了 P，P 的第三个活动开始，此时 P 的前面两个活动仍然没有结束，因此 P 共有 3 个活着的活动。当 P 的第三个活动结束返回到 P 的第二个活动时，有 P 的 2 个活动是活着的。当 P 的第二个活动结束返回到 P 的第一个活动时，只有 P 的第一个活动是活着的。

试用版，仅供2020清华大学编译课使用，勿外传！

```
（1）program sort（input, output）；
（2）        var a : array [0..10]   of   integer；
（3）        procedure readarray；
（4）                  var i : integer；
（5）             begin
（6）                                  for i := 1 to 9 do read(a[i])
（7）                       end；
（8）         function partition（y, z : integer ）: integer；
（9）                 var   i, j, x, v : integer；
（10）            begin

                          …

（11）                    end；
（12）        procedure quicksort（m, n : integer）；
（13）           var i : integer；
（14）           begin
（15）                        if    ( n > m)   then    begin
（16）                     i := partition(m, n)；
（17）                                  quicksort(m, i-1)；
（18）                                  quicksort(i+1, n)
（19）                              end
（20）                    end；
（21）    begin
（22）           a[0] := -9999；a[10] := 9999；
（23）                  readarray；
（24）                  quicksort(1, 9)；
（25）    end.
```

图 8-2 一个 Pascal 语言例子

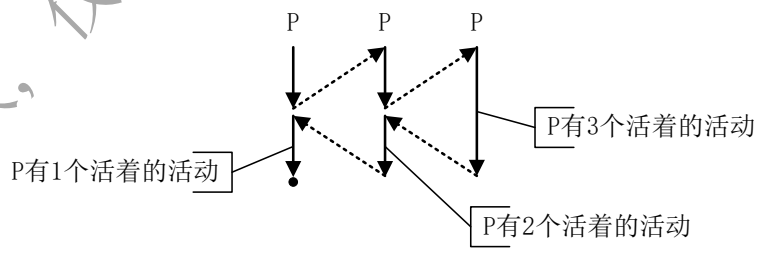


图 8-3 递归调用时过程 P 的活动情况

程序的运行表现为从主过程开始的一系列过程调用，可以用一个称为**活动树**的树状结构来描述程序的运行过程。活动树中的每个结点代表一个过程的一个活动，根结点代表主过程的活动；假设 a 和 b 是活动树中的两个结点，a 是 b 的父结点当且仅当活动 a 调用了活动 b，即程序的控制流程是从 a 进入 b；a 在 b 的左边当且仅当 a 的生存周期在 b 的生存周期之前。图 8-2 程序的一次执行可用图 8-4 中的活动树来描述。为方便起见，用过程的第一个字母表

待出版，仅供2020杭电本科生上课使用，勿外传！

示过程的一个活动，括号内的内容是调用过程时的实在参数，如 q(1, 9)代表调用过程 quicksort，传递给它的参数是 1 和 9。图中的虚线是程序运行时过程的执行顺序，首先执行 s，s 调用 r，r 执行完返回到 s，s 又调用了 q(1, 9)，q(1, 9)调用了 p(1, 9)，p(1, 9)执行完返回 q(1, 9)，q(1, 9)又调用了 q(1, 3)，.....，最后返回到 s，结束整个程序的运行。

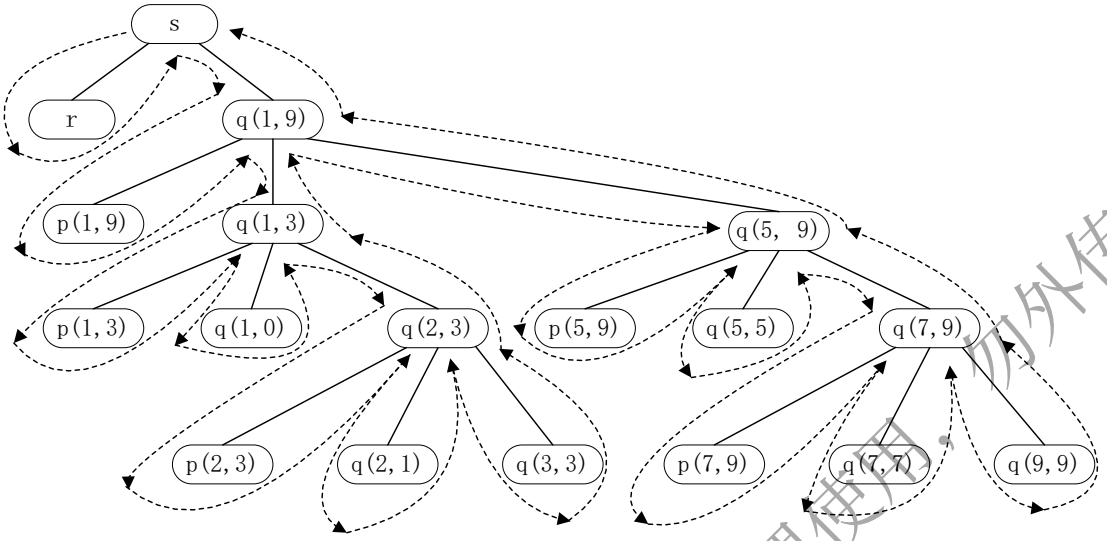
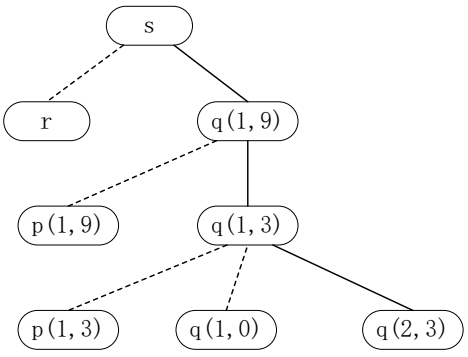


图 8-4 图 8-2 程序的一棵活动树

程序运行的任一时刻可能都会有多个活着的活动，可以用一个称为**控制栈**的数据结构保存过程活动的生存踪迹，即时监控当前活动的变化情况。控制栈使用的基本思想是：当一个活动开始时，把代表这个活动的结点压栈；当这个活动结束时，把代表这个活动的结点从栈中弹出。

栈
s
s r
s q(1, 9)
s q(1, 9) p(1, 9)
s q(1, 9) q(1, 3)
s q(1, 9) q(1, 3) p(1, 3)
s q(1, 9) q(1, 3) q(1, 0)
s q(1, 9) q(1, 3) q(2, 3)

(a)



(b)

图 8-5 图 8-2 程序的控制栈

图 8-5(a)展示的是图 8-2 中程序一次执行的前几步过程调用时控制栈的变化情况。栈中的每一行，代表的是每次过程调用后的情形，栈中的活动就是当前活着的活动，栈顶活动是当前正在执行的。从栈底到栈顶活动的排列，反映的是活动的调用关系。如图 8-5(a)的最后一行：

s q(1, 9) q(1, 3) q(2, 3)

待出版，仅供2020杭电本科生上课使用，勿外传！

反映的是当前有 4 个活动是活着的，分别是 s、q(1, 9)、q(1, 3)、q(2, 3)，其中 s 调用了 q(1, 9)，q(1, 9)调用了 q(1, 3)，q(1, 3)调用了 q(2, 3)，当前 q(2, 3)正在执行。图 8-5(b)是对应的活动树，虚线连接的活动结点代表生存周期已结束的活动，实线连接的结点代表当前还活着的活动。从当前结点 q(2, 3)依次向上搜索父结点及父结点的父结点直到根结点，遍历到的结点正好是当前控制栈中的活动结点。控制栈的使用与 8.4 节介绍的栈式存储分配的思想是一致的。

8.3 运行时刻内存空间的组织

操作系统在接到一个运行目标代码的命令时，通常会从当前空闲的内存空间中申请一块连续的存储区，并将目标代码加载到这块存储区中来运行。这块内存空间的一个典型划分如图 8-6 所示。

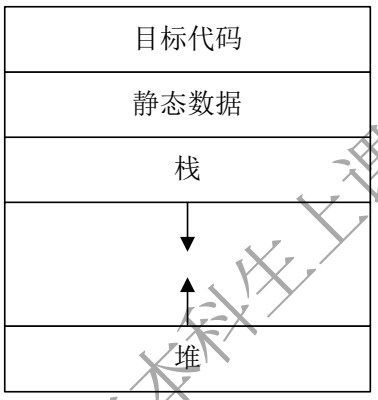


图 8-6 目标代码运行空间的典型划分

其中目标代码区存放待运行的目标代码。目标代码在编译阶段生成，编译结束后目标代码所需存储空间的大小就确定下来，因此目标代码区的大小是已知的。静态数据区通常用来存放全局变量，或者用来存放在编译阶段就可以确定存储地址的数据对象。栈是一个遵循“先进后出”访问规则的数据结构，只能对栈顶数据进行存取。在运行空间中划分一个栈区，可以支持 Pascal、C 这类允许过程递归调用的语言。这类语言的程序每次运行的过程调用轨迹取决于输入参数，数据对象存储空间的分配和管理是在运行过程中动态进行的，在编译阶段无法确定数据对象的存储地址和需要的总的数据空间大小。递归调用过程可以用活动树和控制栈来描述，类似地可以在运行空间中专门设置一个栈区来支持递归调用时数据空间的动态分配和管理。很多程序设计语言允许动态的内存申请，程序员可以根据自己的需要按任意顺序申请和释放内存空间，如 C 语言的 malloc 函数（内存申请）和 free 函数（内存释放）。由于内存申请和释放不一定按照栈的“先进后出”顺序，不能用栈结构来实现。这时候通常在运行空间中单独划分一个堆区来支持语言的这一特性。具体的存储分配方案将在 8.4 节中介绍。需要指出的是，图 8-6 所示的只是目标代码运行时空间的一个典型划分，不同的程序设计语言有各自不同的语言特性，它们运行空间的划分也不尽相同。运行空间的分配还取决于编译器的设计方法。

仅供2020级电本科生上课使用，勿外传！

如前所述，程序的运行过程是由一个过程调用序列组成的。每个过程在执行时都需要用到相关信息，如局部数据。通常用称为**活动记录**的一段连续的存储区来存放过程的一次执行所需要的信息。活动记录的一般结构如图 8-7 所示。

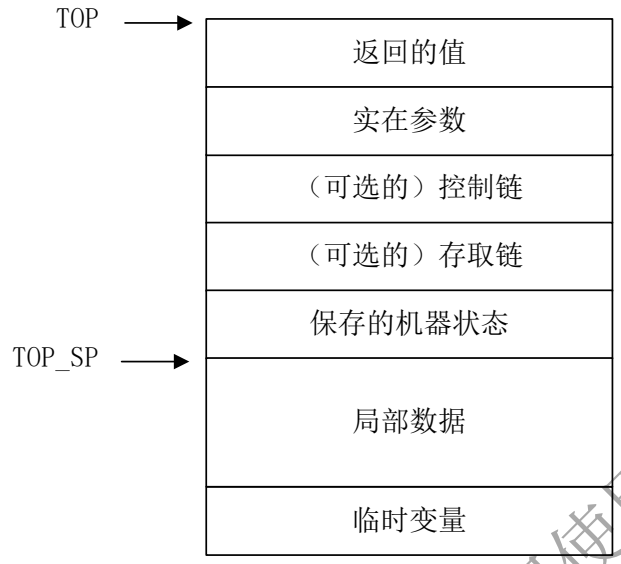


图 8-7 活动记录的结构

活动记录一般分为 3 个域，又可以进一步分为 7 个区：

（1）参数域：含“返回的值”区和“实在参数”区。令过程 P 调用了过程 Q，Q 的活动记录的“实在参数”区用来存放 P 传递给 Q 的实在参数。Q 执行完之后计算出的中间结果，存放在 Q 的活动记录的“返回的值”区，P 可以将这个中间结果复制回自己的活动记录。

（2）状态域：含“（可选的）控制链”，“（可选的）存取链”和“保存的机器状态”。控制链是一个指针，用来反映过程之间的调用关系，如 P 调用了 Q，则 Q 的活动记录的存取链指向 P 的活动记录的存取链。存取链也是一个指针，用来访问非局部名字，具体用法在 8.5 节中介绍。“可选的”是指不是必须的，在某些存储分配方案中是不需要的。机器状态保存区存放的是调用过程在调用被调用过程之前的有关的机器状态信息。这些信息包括当从被调用过程中返回时必须恢复的程序计数器和一些寄存器的值。

（3）数据域：含“局部数据”区和“临时变量”区。“局部数据”区用来存放过程执行需要的局部数据，这些局部数据是在当前过程中被说明的。“临时变量”区用来存放过程执行时计算出来的临时变量。

TOP_SP 是一个指向“临时变量”区起始地址的指针， TOP_SP 的值加上变量的偏移地址（offset 值）就可以对局部变量进行访问。在栈式存储分配中，通常有一个指针 TOP 指向栈顶活动记录的起始地址。图 8-7 所示的活动记录的结构只是一个一般的结构，具体语言不同，它的活动记录的结构和内容也会有差异。

非出版（仅供2020杭电本科生上课使用，勿外传！

8.4 运行时刻内存空间分配策略

确定程序中每个名字的存储地址是目标代码运行时内存空间分配的重要内容。在目标程序运行过程中，要访问名字的值需要经过两个步骤，第一步是确定名字的存储地址，第二步是取出该存储地址中的值。下面引入两个函数：environment 和 state。environment 是一个将名字映射为存储位置的函数，state 函数用于将存储位置映射为在那里存放的值。名字的存储地址又称为**左值**（l-value），名字的当前值又称为**右值**（r-value），因此也可以说 environment 函数把名字映射为一个左值，state 函数把一个左值映射为一个右值，见图 8-8。

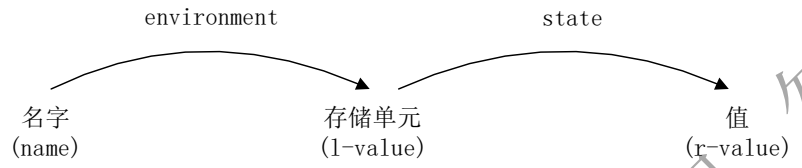


图 8-8 名字的左值和右值

将一个内存地址与一个名字联系起来，称为**名字的绑定**。在程序的一次运行中，一个名字的右值可能会经常改变。如果语言允许递归调用，一个名字也有可能被绑定到多个地址。

目标代码运行时刻的存储分配策略主要有静态存储分配和动态存储分配，其中动态存储分配又分为栈式存储分配和堆式存储分配。采用哪种分配策略主要由源语言的特点决定。

8.4.1 静态存储分配

如果一个语言不允许过程的递归调用，也没有动态内存申请机制，并且不支持可变数组等可变数据结构，那么在编译时刻就可以将每个名字绑定到运行空间中，安排好每个数据对象在运行时的存储位置，而且可以确定运行时刻所需的全部数据空间的大小。这就是所谓的**静态存储分配**。由于不允许递归调用，因此不需要栈区，没有动态的内存申请和可变数据结构，也不需要堆区，静态存储分配的运行空间中只有目标代码区和静态数据区。支持过程执行的活动记录，被分配到静态数据区中。程序有几个过程，就有几个活动记录，一个过程对应一个活动记录。活动记录中每个区的大小在编译时刻可以计算得到，每个名字绑定到固定的内存地址，在运行过程中不会发生变化。

采用静态存储分配的典型语言是 Fortran 77。一个 Fortran 77 程序由一个主程序段和若干个子程序段构成（每个程序段就是一个过程或函数）。图 8-9 是一个 Fortran 77 的程序例子，该程序包括两个程序段，CNSUME 是主程序段，PRDUCE 是子程序段。编译时分别将 CNSUME 和 PRDUCE 翻译为目标代码，同时计算这两个过程的活动记录的大小，安排好活动记录中每个字节存储的数据对象。运行时刻，将 CNSUME 和 PRDUCE 的目标代码装载到目标代码区，在静态数据区为 CNSUME 和 PRDUCE 的活动记录分配空间。图 8-10 是例子程序运行时刻的空间分配示意图。CNSUME 中定义了局部名字 BUF（长度为 50 的字符串变

仅供2020杭电本科生上课使用，勿外传！

量）、NEXT（整型变量）、C（字符型变量），需要将它们绑定到 CNSUME 活动记录的局部数据区。同样，PRDUCE 中也定义了几个局部名字，分别是 BUFFER（长度为 80 的字符串变量）和 NEXT（整型变量），需要将它们绑定到 PRDUCE 活动记录的局部数据区。

```
（1）PROGRAM    CNSUME
（2）       CHARACTER   *50    BUF
（3）       INTEGER   NEXT
（4）       CHARACTER   C，PRDUCE
（5）       DATA    NEXT / 1 / ，BUF/ '    '/
（6）  6         C＝PRDUCE（）
（7）            BUF（NEXT：NEXT）＝C
（8）            NEXT＝NEXT＋1
（9）            IF（C   .NE.   '   '）  GOTO   6
（10）          WRITE（*，'(A)'）BUF
（11）           END

（12）CHARACTER   FUNCTION   PRDUCE（    ）
（13）        CHARACTER   * 80   BUFFER
（14）       INTEGER NEXT
（15）       SAVE   BUFFER，NEXT
 (16)          DATA   NEXT   / 81 /
（17）           IF ( NEXT .GT. 80 ) THEN
（18）                READ（*，'（A）'）  BUFFER
（19）                NEXT＝1
（20）           END IF
（21）            PRDUCE＝BUFFER（NEXT：NEXT）
（22 ）          NEXT＝NEXT＋1
（23 ）           END
```

图 8-9 一个 Fortran 77 程序

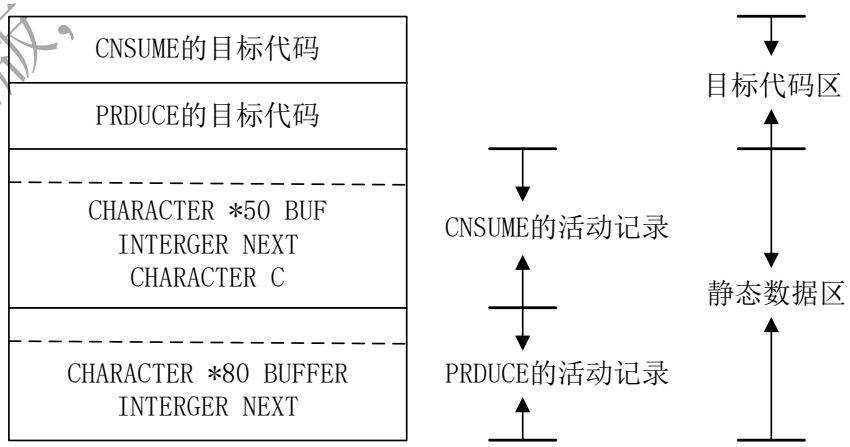


图 8-10 Fortran 程序的静态存储分配

待出版，仅供2020计算机科学上课使用，勿外传！

8.4.2 栈式存储分配

大部分结构化程序设计语言都支持过程的递归调用，如 C、Pascal、Algol 等。过程递归调用使得程序运行的某些时刻会有同一个过程的多个活动在执行，而且活动的数量是动态变化的，见图 8-3。每个活动的执行都需要独立的数据空间，这个数据空间需要容纳两部分内容：

（1）过程在本生存周期内的数据对象，如局部变量、实在参数、临时变量等；

（2）管理过程活动的记录信息，如反映过程调用关系的控制链信息、过程调用前调用过程的机器状态信息等。

以上内容组织在过程的活动记录中，每当一个过程被调用，开始该过程的一次活动，需要为该活动分配数据空间以存放它的活动记录。当活动结束后，它的数据空间没有必要保留，可以收回供其它活动使用。可以基于控制栈的思想，采用栈式存储分配策略来实现上面讨论的动态分配和释放程序运行过程中活动的数据空间。

栈式存储分配的思想：在运行空间中划分一块存储空间作为栈区，程序运行时每当调用一个过程，就将该过程的活动记录压入栈中，过程执行完毕将它的活动记录从栈中弹出。

以图 8-2 中的 Pascal 语言程序为例，它的一次运行活动树和栈区内容的变化情况如图 8-11 所示。

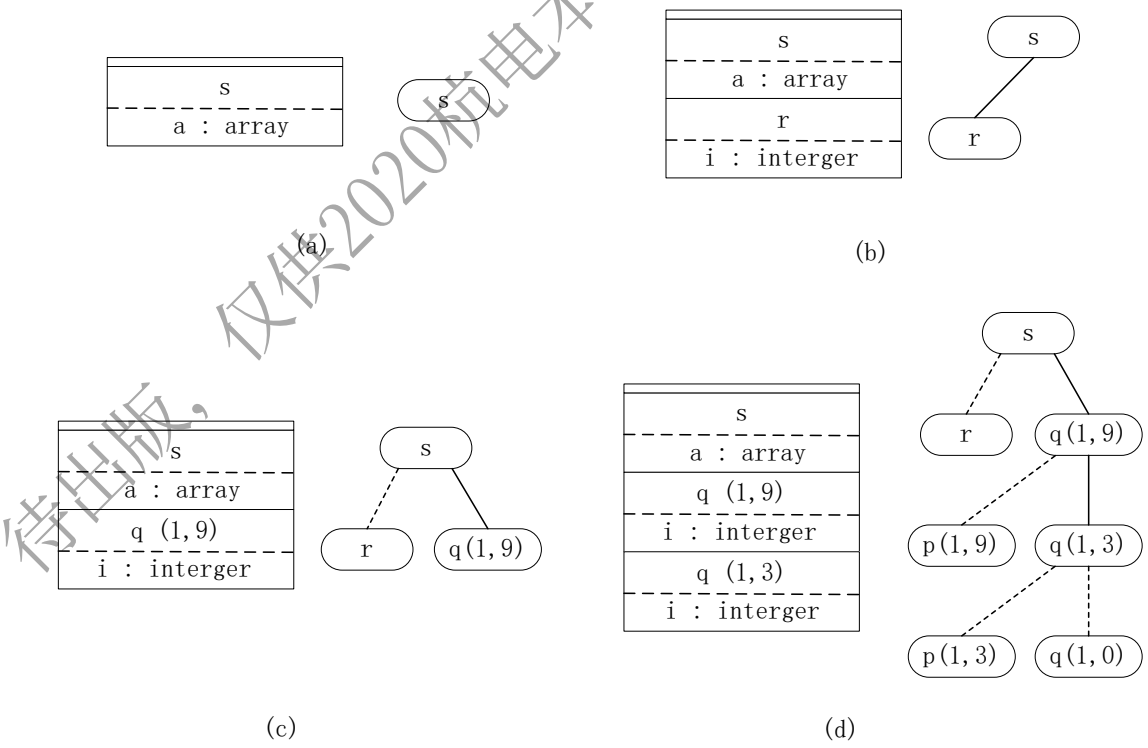


图 8-11 Pascal 程序的栈式存储分配

程序执行前，栈区为空，栈底在图的上方。以过程名的首字母表示过程，首先主过程 s 执行，将它的活动记录压到栈顶（即在栈顶分配一块存储空间存放活动记录），s 的局部名字

待出版，仅供2020级电本科生上课使用，前火传！

a 绑定到活动记录的局部数据区中，见图 8-11(a)；接着 s 调用了 r，将 r 的活动记录压栈，r 的局部名字 i 绑定到它的活动记录中，此时栈中有两个活动记录（分别对应 s 和 r 的活动），见图 8-11(b)；r 执行完之后，从栈顶收回 r 的活动记录，控制回到 s，接下来 s 调用 q(1, 9)，将 q(1, 9)的活动记录压到栈顶，q 的局部名字 i 绑定到它的活动记录中，此时栈中的两个活动记录分别对应 s 和 q(1, 9)，见图 8-11(c)；图 8-11(d)是经过 6 步过程调用之后的情形，此时 q(1, 3)正在执行，它的活动记录在栈顶，栈中有 3 个活动记录，分别对应 s、q(1, 9)和 q(1, 3)。注意这时栈中有两个 q 的活动，q 的局部变量 i 被绑定到两个存储地址。活动树中虚线连接的活动结点是生存周期已结束的活动，它们的活动记录曾经被压入栈中但已被收回，实线连接的结点代表当前还在执行的活动，它们的活动记录还在栈中。

采用栈式存储分配，栈区的内容在目标代码运行过程中是动态变化的，表现为一系列的活动记录在栈顶的分配与释放，运行开始前栈区为空，当主过程运行结束，从栈顶释放主过程的活动记录，栈区重新变为空。

以上介绍的是栈式存储分配的策略，下面讨论栈式存储分配的具体实现。栈式存储分配的实现反映在目标代码生成器的构造策略中，最终体现为生成的目标代码，由目标代码中的调用序列和返回序列来完成。

调用序列

调用序列是目标代码中的一个指令序列，完成一个过程调用另一个过程的一系列操作，包括为被调用过程分配一个活动记录，并在相应的区中填入信息。假设过程 P 调用过程 Q，调用序列需完成如下工作：

（1）参数传递：P 计算实在参数的值，并将它写入 Q 的活动记录的实在参数区。

（2）控制信息设置：P 将返回地址写入 Q 的活动记录的机器状态域中；P 将当前的控制链地址写入 Q 的活动记录的控制链区；P 为 Q 建立存取链；P 设置新的 TOP 和 TOP_SP 的值（分别指向 TOP’和 TOP_SP’的位置，见图 8-12）。

（3）通过 goto 语句跳转到 Q 的代码。

（4）P 保存寄存器的值、以及其它机器状态信息。

（5）Q 初始化局部变量，开始执行 Q 的代码。

返回序列

返回序列也是目标代码中的一个指令序列，完成从一个被调用过程返回到它的调用过程的一系列操作，包括释放被调用过程的活动记录，并复制出返回值。具体完成如下工作：

（1）Q 把返回值写入自己活动记录的返回值区。

（2）Q 恢复断点状态，包括寄存器的值、TOP 的值、机器状态等。

待出版，仅供2020秋电本学生上课使用，勿外传！

（3）根据返回地址返回到 P 的代码中（通过 goto 语句）。

（4）P 把返回值复制到自己的活动记录中。

（5）P 恢复 TOP_SP 的值。

（6）继续执行 P 的代码。

图 8-12 是调用序列和返回序列责任划分示意图。在 P 调用 Q 之前，栈顶是 P 的活动记录，P 调用 Q 后在栈顶压入 Q 的活动记录并通过调用序列作初始化。Q 执行完毕，收回栈顶 Q 的活动记录并通过返回序列返回到 P 中继续执行。图 8-12 中“调用者 P 的责任”覆盖的区域由调用序列负责赋值，“被调用者 Q 的责任”覆盖的区域由返回序列负责赋值。

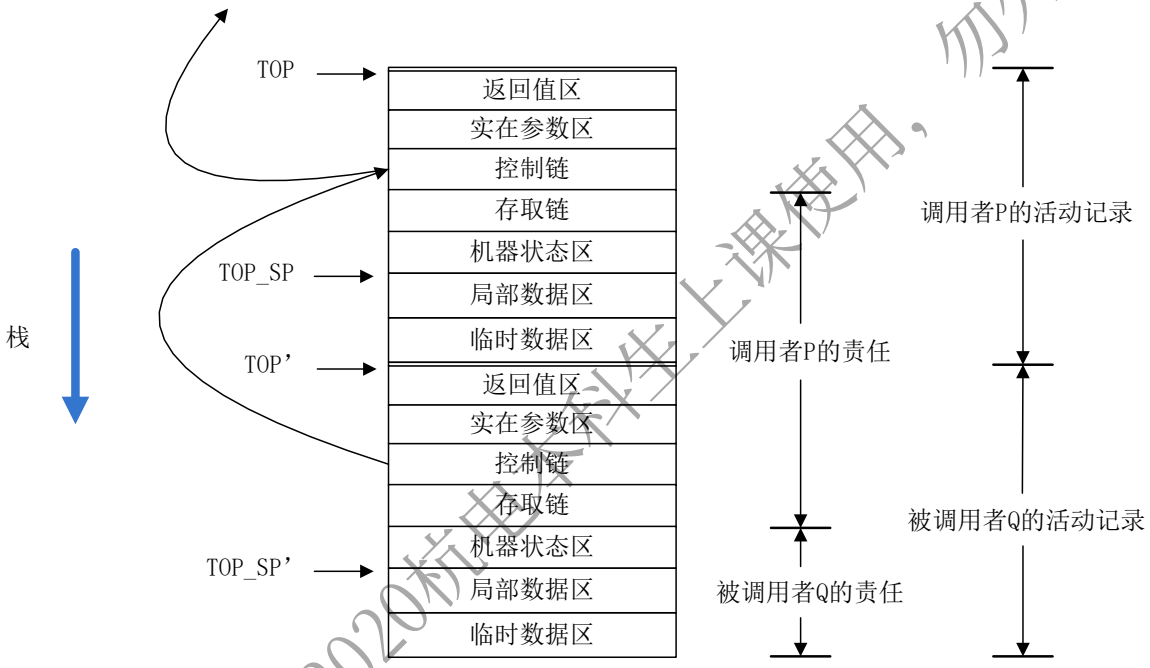


图 8-12 调用序列和返回序列的责任划分

8.4.3 堆式存储分配

有些程序设计语言允许程序员自由申请和释放内存空间，如 C++中的 new 和 delete，C 的 malloc 和 free，Pascal 的 new 和 dispose，有些程序设计语言不仅有过程还有进程。这类语言的目标代码的运行空间使用未必服从“先申请后释放，后申请先释放”的原则，栈式存储分配策略就不适用了。这时通常采用一种称为**堆式存储分配**的完全动态的存储分配方案。堆式存储分配的基本思想是：在运行空间中划分出一块连续的存储空间作为堆区，每当程序申请空间时，就从堆区中寻找一块符合要求的存储块返回给程序，当程序释放存储块时则将它回收。由于申请和释放存储块的顺序完全取决于程序，每次申请或释放的存储块的大小也不同，经过一段时间运行之后，堆区空间将被划分成很多块，有些被占用，有些空闲。这时候如果程序申请一块大小为 N 个字节的空间时，需要决定应该从哪个空闲块得到这个空间，这取决于空间管理程序。空间管理程序可以采用三种策略：

仅供2020杭电本科生上课使用，勿外传！

（1）首次匹配法：顺次扫描堆区中的存储块，一旦发现一个空闲的并且大于等于 N 个字节的存储块就从中切分出一个 N 字节的存储块返回。这种方法时间效率最高。

（2）最优匹配法：扫描堆区中的每个存储块，寻找一个大小最接近 N 的略大于 N 的空闲块作为目标存储块。这种方法看起来最合理，但是时间效率不高，同时会造成许多无法再利用的“碎片”。

（3）最差匹配法：扫描堆区中的每个存储块，寻找一个最大的空闲块作为目标存储块，当然这个最大的存储块应大于 N 字节。这种方法将使堆区中的块的大小趋于一致。由于要扫描所有的存储块，这种方法时间效率也不高。

无论哪种分配方法，堆区的“碎片化”都是不可避免的，最后将出现“虽然总的空闲空间够大，但是内存申请却失败”的情况。在堆式存储分配和管理中，应该配合存储块合并、垃圾空间回收等技术避免上述情况的发生。

8.5 对非局部名字的访问

8.5.1 程序设计语言的作用域规则

作用域是指一个说明起作用的范围。作用域的概念最早是在 ALGOL 60 中提出来的，它和分程序概念相关。ALGOL 60 中的分程序是用语句 begin、end 括起来的说明序列和语句序列，它的一般形式是：

```
begin
        < 说明序列 >
        < 语句序列 >
end
```

ALGOL 60 是单模块结构语言，它的程序由单个分程序组成，分程序中还可以嵌套一个或多个别的分程序。在一个分程序中不仅可以使用其中定义的局部名字，还可以使用其外围分程序（或外围分程序的外围分程序）中说明的名字，这就是所谓的非局部名字。非局部名字是相对于引用点所在的过程或分程序来说的。程序执行时引用的在当前过程（或分程序）之外定义的变量称为**非局部名字**。

一个语言的作用域规则决定了如何处理对非局部名字的访问。主要有两种作用域规则：静态作用域规则和动态作用域规则。在静态作用域的情况下，当程序内某个语句中使用了一个变量名字，就根据程序的静态文本由里向外查找该变量名字的定义性出现。采用静态作用域规则的语言包括 Pascal、C、Ada 等。在动态作用域的情况下，当程序的某个语句中使用了一个变量名字，根据过程调用的顺序反过来查找该变量名字的定义性出现，即先在最新调用的过程中查找，如果没有，就在直接调用它的过程中查找。如此反复下去，直到找到该变量名字的定义性出现。采用动态作用域规则的语言有 LISP、APL 等。

待出版，仅供2020机电本科生上课使用，勿外传！

本节重点讨论采用静态作用域规则的非局部名字的访问。静态作用域由最近嵌套规则定义：

（1）过程（或分程序）B 中的一个说明的作用域包括 B；

（2）如果名字 x 在过程（或分程序）B 中没有说明，那么，x 在 B 中的出现是在一个外围过程（或分程序）B'中的 x 的说明的作用域之内，并且使得：

a）B'中有 x 的说明；

b）B'是包围 B 的，相对于其它任何具有名字 x 的说明且包围 B 的过程（或分程序）而言，B'是离 B 最近的。

以上作用域规则又称为**最近嵌套的作用域规则**，大部分结构化程序设计语言都采用这种作用域规则。

```
（1）  program sort（input，output）；
（2）    var a：  array[0..10］  of integer；
（3）           x : integer；
（4）    procedure   readarray；
（5）      var i  ：  integer；
（6）      begin  … a…    end { readarray};
（7）    procedure   exchange（i，j：  integer）；
（8）      begin
（9）        x：= a[i]  ；a[i]:=a[j]; a[j]:＝x
（10）     end { exchange};
（11）   procedure quicksort（m，n：  integer）；
（12）     var k，v：  integer；
（13）     function partition（y，z：  integer）  ：  integer；
（14）       var i，j  ：  integer；
（15）       begin …a…
（16）             …v…
（17）             …exchange（i，j）；…
（18）       end { partition};
（19）     begin  …  end { quicksort};
（20）   begin  …  end. { sort }
```

图 8-13　Pascal 语言的作用域

图 8-13 是第 6 章中讨论过的 Pascal 语言程序。（2）-（3）行是主过程的变量说明语句，定义了 a 和 x 两个变量，它们的作用域覆盖整个程序，包括嵌套定义在主过程中的嵌套过程。第（5）行是嵌套过程 readarray 的说明语句，定义了局部变量 i，它的作用域限于过程 readarray 内。第（12）行是过程 quicksort 的变量说明语句，定义了 k 和 v 两个局部变量，它们的作用域覆盖过程 quicksort 的程序体，还包括嵌套函数 partition。函数 partition 在第（14）行定义了两个局部变量 i 和 j，其作用域限于 partition 内。第（6）行是 readarray 的程序体，引用了变量 a，而 a 在 readarray 中没有定义，因此这里引用的是非局部名字。根据最

待出版，仅供2020秋电子科技大学学生上课使用，勿外传！

近嵌套的作用域规则，这个非局部名字 a 处在第（2）行说明的变量 a 的作用域内，因此是对这个名字说明的引用。同理，第（9）、（15）、（16）行也引用了非局部名字，可以根据最近嵌套的作用域规则确定它们是在哪里被说明的。

8.5.2 分程序结构的处理

分程序是含有自身局部数据说明的复合语句。分程序之间可以嵌套，但是不能交叉。除了 ALGOL 语言，C 语言也有分程序结构。C 语言的分程序由一对大括号“{、}”括起，每个分程序都可以定义局部于自身的局部变量。

图 8-14 是一个 C 语言程序，该程序共有 4 个分程序，分别是 B_0、B_1、B_2、B_3。其中 B_0 嵌套 B_1，B_1 嵌套 B_2 和 B_3，B_2 和 B_3 是并列的。4 个分程序都定义了自己的局部变量，并被分别初始化为分程序的序号。

```
（1）        main()
（2）        {
（3）            int a=0;
（4）            int b=0;
（5）            {
（6）                int b=1;
（7）                {
（8）                    int a=2;
（9）                    printf( “%d %d\n” ,a,b);
（10）               }
（11）               {
（12）                   int b=3;
（13）                   printf( “%d %d\n” ,a,b);
（14）               }
（15）               printf( “%d %d\n” ,a,b);
（16）           }
（17）           printf( “%d %d\n” ,a,b);
（18）       }
```

图 8-14　只包含 1 个函数的 C 语言例子

各变量说明的作用域见表 8-1。其中 int a=0 在 B_0 中被定义，作用域应该覆盖整个函数，但是在 B_2 中也定义了 a，B_2 中对 a 的引用是在 B_2 中说明的，因此 int a=0 的作用域为 B_0-B_2，B_2 称为作用域中的一个“洞”。类似地，int b=0 和 int b=1 的作用域中也存在“洞”。

表 8-1 图 8-14 例子中各说明的作用域

说明	作用域
int a=0	B_0-B_2
int b=0	B_0-B_1
int b=1	B_1-B_3
int a=2	B_2
int b=3	B_3

待出版，仅供2020杭电本科生上课使用，勿外传！

图 8-14 中的 C 语言程序，只有一个函数，运行时用一个活动记录来组织它的局部数据。程序中说明的每个变量都绑定到活动记录的局部数据区。对于这种分程序结构可以采用栈式存储分配来实现，即将活动记录的局部数据区看作是一个栈来组织局部变量。由于说明的作用域不会超出它所在的分程序，在进入分程序时可以为被说明的名字分配空间，当控制离开分程序时释放空间。由于 B_2 和 B_3 是并列的，运行时刻永远不会同时执行，在 B_2 和 B_3 中定义的局部变量不会同时被引用，因此可以共用同一个存储空间。运行时刻活动记录局部数据区的空间分配如图 8-15 所示。局部变量 a、b 的角标表示它在哪个分程序中被说明。

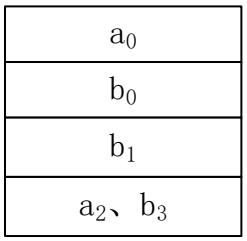


图 8-15　局部数据区的栈式存储分配

图 8-14 中程序的各分程序都有打印语句，运行时打印当前 a、b 的值。第（9）行打印出 2 和 1；第（13）行打印出 0 和 3；第（15）行打印出 0 和 1；第（17）行打印出 0 和 0。

8.5.3 无嵌套过程语言的处理

有些语言不允许过程的嵌套定义，即在过程的说明部分不能定义其它的过程或者函数。典型的不允许过程（函数）嵌套的语言是 C 语言。C 语言程序由多个函数构成，函数之间相互独立。每个函数可引用自己的说明语句定义的局部变量，如果函数中引用了一个非局部名字 a，那么 a 必须在所有函数之外被说明。在函数外面的一个说明的作用域包括此说明之后的所有函数，如果名字在某个函数中被再次说明，那么它的作用域带有“洞”。

图 8-16 是一个包含 2 个函数的 C 语言例子。第（1）行定义了一个全局变量 s，第（4）行引用了变量 s，这个变量对于函数 round 来说是非局部名字。

```
（1）    double s;
（2）    void round ( double r )
（3）    {
（4）        s=3.1415926*r*r;
（5）    }
（6）    int main( )
（7）    {
（8）        double x=3;
（9）        round ( x );
（10）       printf( “%.21f” , s );
（11）       return 0;
（12）   }
```

图 8-16　包含 2 个函数的 C 语言例子

待出版，仅供2020届本科生上课使用，勿外传！

程序运行时刻，全局变量被绑定到静态数据区，每个过程（或函数）定义的局部变量绑定到各自活动记录的局部数据区中。引用非局部名字时，根据静态数据区的首地址加上名字的偏移地址（offset 值），直接到静态数据区中去访问。图 8-16 中程序的运行时刻空间分配如图 8-17 所示。

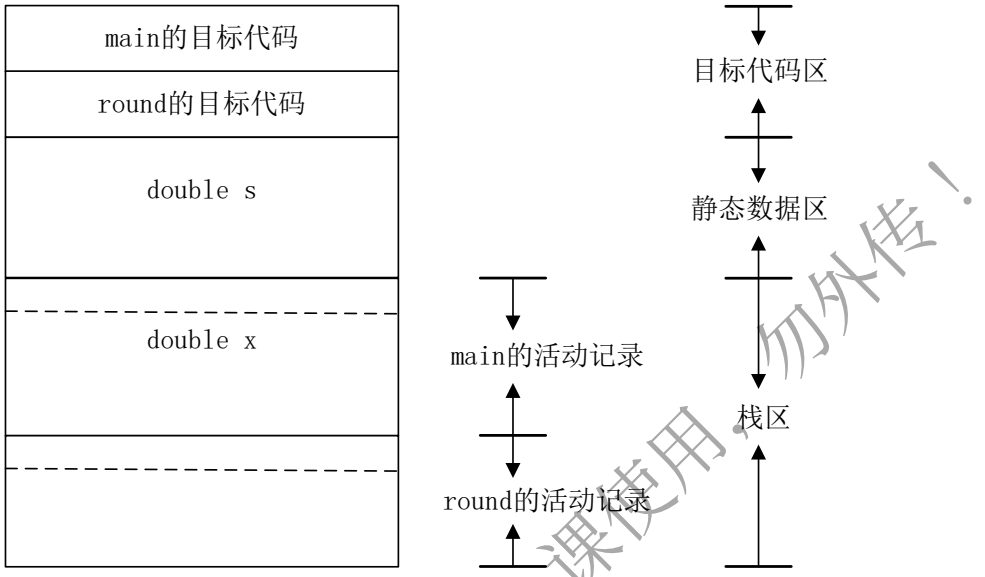


图 8-17　图 8-16 中程序的运行时刻空间分配

8.5.4 有嵌套过程语言的处理

对于 Pascal 语言这类允许过程嵌套定义的语言，访问非局部名字的机制相对比较复杂。当程序中引用非局部名字时，该名字的值被绑定在其它过程的活动记录中，该活动记录有可能位于栈的深处。再次考察图 8-13 给出的 Pascal 语言程序，程序的第（6）、（9）、（15）、（16）行都对非局部名字进行了引用，根据最近嵌套的作用域规则可以确定它们引用的是哪个说明。在运行时刻，同一个过程可能有多个活动是活着的，若程序的某点引用了非局部名字 c，而 c 处在过程 m 中 c 的说明的作用域内，则应该到 m 的最近开始的活着的活动的活动记录中去访问 c 的值。对于图 8-13 中的程序，它运行过程中的几种典型情况如图 8-18 所示。非局部名字访问描述如下：

（1）图 8-13(a)：s 调用 q(1, 9)，由于 quicksort 直接嵌套在 sort 中，若 quicksort 第（19）行的程序体中引用了非局部名字，应该到位于栈中的 q(1, 9)的活动记录上方的 s 的活动记录中去访问。

（2）图 8-13(b)：q(1, 9)调用 q(1, 3)，q(1, 3)是 quicksort 的一个新的活动，同样 quicksort 程序体中如果引用了非局部名字，应该到栈中 s 的活动记录中去访问。

（3）图 8-13(c)：q(1, 3)调用 p(1, 3)，由于 partition 直接嵌套在 quicksort 中，partition 中引用的非局部名字，应该到 quicksort 的活动记录中去访问。当前 quicksort 有两个活着的活动，分别为 q(1, 9)和 q(1, 3)，但最近的活动是 q(1, 3)，p(1, 3)引用的非局部名字首先要到 q(1, 3)活动记录中去找，如果没有再到 s 的活动记录中去访问。

待出版，仅供2020计本科生上课使用，勿外传！

（4）图 8-13(d)：p(1, 3)调用 e(1, 3)，由于 exchange 直接嵌套在 sort 中，exchange 程序体中引用的非局部名字，应该到 s 的活动记录中去访问。

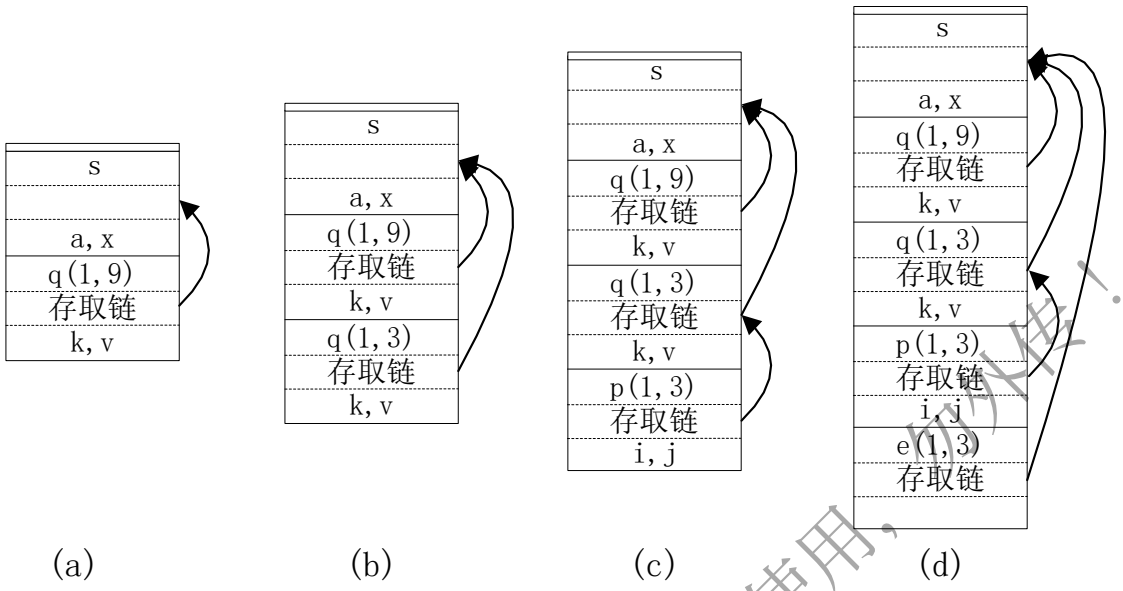


图 8-18　利用存取链访问非局部名字

非局部名字访问的具体实现有不同的方法，一种方法是通过存取链，另一种方式是通过 DISPLAY 表。

存取链是一个存放在活动记录中的指针。若过程 P 直接嵌入在过程 Q 中，则 P 的活动记录中的存取链指向 Q 的最近活着的活动记录的存取链。图 8-18 中各活动记录的存取链已经建立好了，主过程没有外围过程，它的存取链为空。接下来为每个过程引入一个**嵌套深度**，令主过程的嵌套深度为 1，当从一个包围过程进入一个被包围过程时嵌套深度加 1。以图 8-13 程序为例，sort、readarray、exchange、quicksort、partition 的嵌套深度分别为：1、2、2、2、3。类似地，为过程中说明的名字也引入嵌套深度。过程中说明的名字的嵌套深度和过程的嵌套深度的值相同。

有了存取链和各过程（及其说明的名字）的嵌套深度，可以按如下方法访问非局部名字：令嵌套深度为 N_p 的过程 P，引用了一个嵌套深度为 N_a 的非局部名字 a，则顺着 P 的这次活动的活动记录的存取链前进 N_p-N_a 步到达的活动记录就是非局部名字所在的活动记录，到这个活动记录的局部数据区中就可以对非局部名字 a 进行访问。

例如，在图 8-13(c)中，q(1, 3)调用 p(1, 3)，partition 引用了非局部名字 a，partition 的嵌套深度为 3，a 的嵌套深度为 1，顺着 partition 的活动记录前进 2（3-1=2）步可以到达 s 的活动记录，在这个活动记录的局部数据区中可以对 a 进行访问。

下面讨论如何建立存取链。假设嵌套深度为 N_p 的过程 P 调用嵌套深度为 N_x 的过程 X，为被调用过程 X 的活动记录建立存取链存在 3 种情况：

待出版，仅供2020级电子科生上课使用，勿外传！

（1）N_p < N_x。由于被调用过程 X 的嵌套深度比调用过程 P 的嵌套深度大，因此 X 必须在 P 中被定义，否则 P 不能调用 X。此时被调用过程 X 的活动记录的存取链指向栈中刚好在其上方的调用过程活动记录的存取链。图 8-19(a)是 P 和 X 之间的关系，图 8-19(b)是存取链指向。图 8-18(a)和图 8-18(c)就是这种情况。

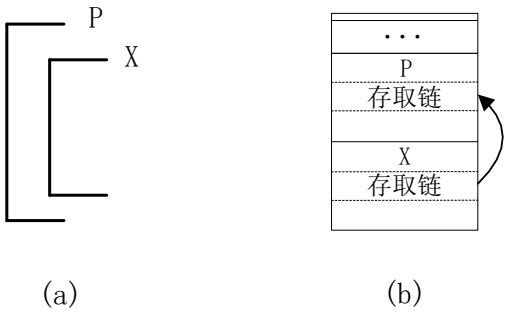


图 8-19　N_p < N_x 时存取链的建立

（2）N_p = N_x 。由于调用过程 P 的嵌套深度和被调用过程 X 的嵌套深度相同，因此它们肯定有共同的直接外围过程，令这个外围过程为 Q，见图 8-20(a)。在 P 调用 X 之前，应该有 Q 对 P 的调用，顺着 P 的活动记录的存取链前进 1 步可以到达 Q 的活动记录。而 X 也直接嵌入在 Q 中，它的活动记录的存取链也应该指向 Q 的活动记录的存取链。因此在这种情况下，顺着调用过程 P 的活动记录的存取链前进 1 步到达的活动记录的存取链就是被调用过程 X 的活动记录的存取链要指向的位置，见图 8-20(b)。图 8-18(b)就是第 2 种情况。

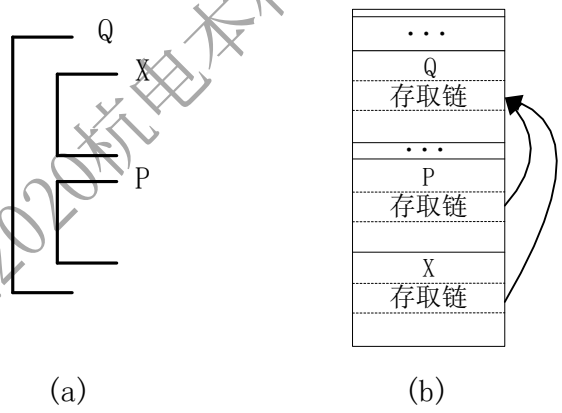


图 8-20　N_p = N_x 时存取链的建立

（3）N_p > N_x 。由于调用过程 P 的嵌套深度大于被调用过程 X 的嵌套深度，X 的直接外围过程肯定也是 P 的外围过程，令这个外围过程为 R，见图 8-21(a)。令 P 和 R 之间的过程为 X_1、X_2、...、X_{Np-Nx}。在 P 调用 X 之前，应该有 R 对 P 的调用，顺着 P 的活动记录的存取链前进 N_p-N_x+1 步可以到达 R 的活动记录。而 X 直接嵌入在 R 中，它的活动记录的存取链也应该指向 R 的活动记录的存取链。因此顺着调用过程 P 的活动记录的存取链前进 N_p-N_x+1 步到达的活动记录的存取链就是被调用过程 X 的活动记录的存取链要指向的位置，见图 8-21(b)。图 8-18(d)就是第 3 种情况。

仅供2020杭电本科生上课使用 勿外传！ 非出版

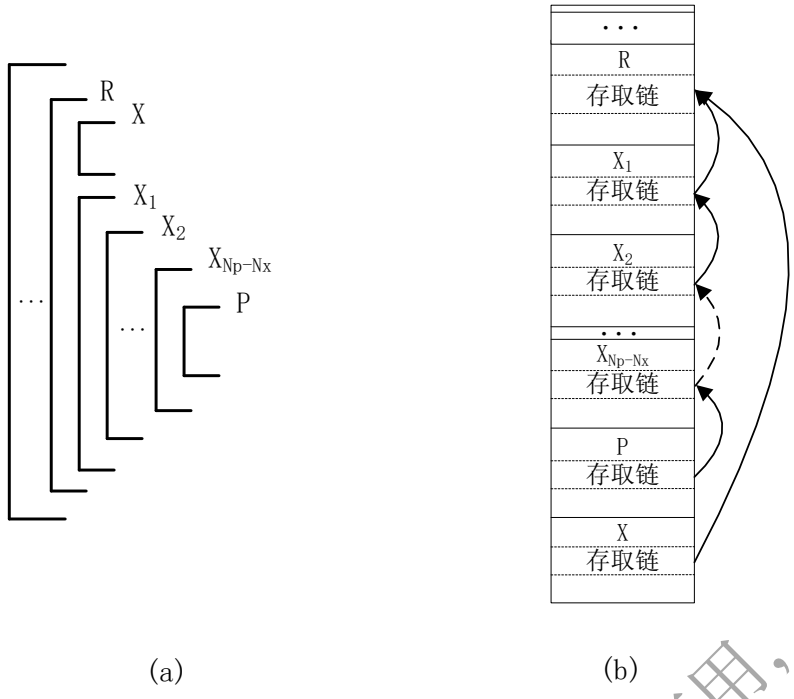


图 8-21　$N_p > N_x$ 时存取链的建立

下面讨论如何利用 DISPLAY 表来访问非局部名字。DISPLAY 表是一个指向活动记录的指针数组，记为 d。DISPLAY 表的大小由程序中嵌套深度最大的过程决定。运行时刻要访问的嵌套深度为 i 的非局部名字就在 d [i] 所指的活动记录中。这种方法只要 1 步就可到达要访问的非局部名字的活动记录，比存取链方法高效。对于图 8-13 中的程序，如果采用 DISPLAY 表来访问非局部名字，它运行时的情况如图 8-22 所示。

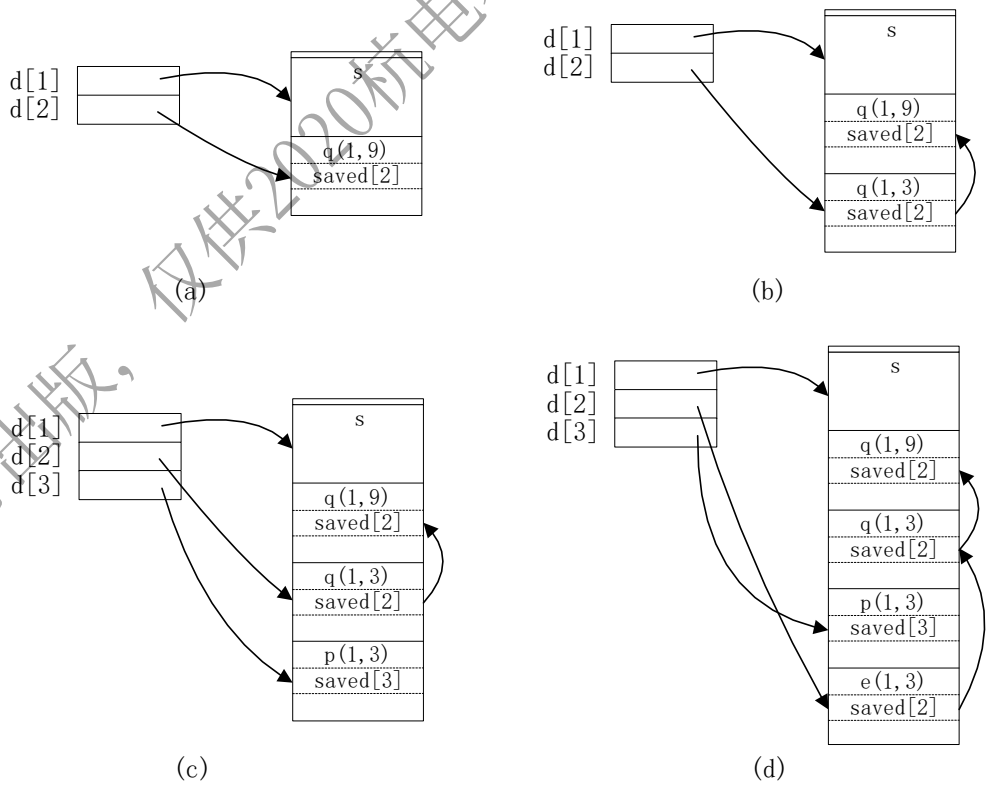


图 8-22　利用 DISPLAY 表访问非局部名字

待出版，仅供2020杭电东北大学课使用，勿外传！

关键是如何建立与维护 DISPLAY 表。假设当前正在执行嵌套深度为 j 的过程 P，那么 P 的活动记录在栈顶，DISPLAY 表中前 j-1 个元素指向包围过程 P 的那些过程的最新的活动记录，而 d [j] 指向过程 P 的活动记录。假设这时 P 去调用嵌套深度为 i 的过程 Q，DISPLAY 表的变化分两种情况：

（1）j < i。这时候 i=j+1，并且 Q 直接嵌套在 P 中。DISPLAY 表中的前 j 个元素不用变化，只需把 d [i] 置为指向新的 Q 的活动记录。图 8-22(a)和图 8-22(c)就是这种情况。

（2）j ≥ i。这种情况下 P 和 Q 的嵌套深度为 1，2，...，i-1 的外围过程必然是相同的。这时需要把旧的 d [i] 的值保存在新的 Q 的活动记录中，并置 d [i] 指向新的 Q 的活动记录。图 8-22(b)和图 8-22(d)都是这种情况。

在活动记录中之所以设置一个 saved 区来保存旧的 d [i] 的值，是为了在从被调用过程中返回时将 DISPLAY 表恢复到过程调用之前的状态。如图 8-22(d)中，当 exchange 执行完毕，从 e(1, 3)的活动记录中取出旧的 saved [2] 的值赋给 d [2]，并且将 e(1, 3)的活动记录从栈中弹出，这时候回到如图 8-22(c)所示的状态。

8.6 小结

编译器生成的目标代码总是要在一定的物理环境下运行的，目标代码运行时刻环境包括硬件环境和软件环境。高级语言源程序通常都是由一组过程（或函数）构成的，程序的运行体现为过程之间的一系列调用，执行从主过程开始，也以主过程结束。过程如果当前正在执行，称它是活着的。程序的一次运行可以用一棵活动树来描述，也可以用一个控制栈来跟踪。过程的每次执行，都需要一个活动记录来提供支持。活动记录是一段连续的内存空间，典型地分为参数域、状态域和数据域等 3 个组成部分。运行时刻内存空间可划分为代码区和数据区，数据区又分为静态数据区、栈区和堆区。静态数据区通常用来存放全局变量或编译阶段可以确定存储地址的数据对象。如果一个语言支持过程的递归调用，则它的运行空间需要有一个栈区，如果一个语言允许动态的内存申请或者支持动态的数据结构，则它的运行空间需要设置一个堆区。高级语言都有自己的特点，语言特点决定了目标代码运行时刻的内存空间的分配策略，典型的分配策略包括静态存储分配、栈式存储分配和堆式存储分配。高级语言可采用静态作用域规则或动态作用域规则，静态作用域大多遵循最近嵌套的作用域规则。非局部名字是相对于引用点来说的，在某个引用点引用的在当前过程（或分程序）之外定义的变量称为非局部名字。非局部名字有两类，一类是分程序之间的，另一类是过程之间的。分程序之间非局部名字的访问相对比较简单，因为局部名字和非局部名字都组织在同一个活动记录里。对于无嵌套过程语言的过程之间的非局部名字，通常可以到静态数据区中去访问。对于有嵌套过程语言的过程之间的非局部名字，可以结合栈式存储分配，采用存取链方式或者 DISPALY 表方式来访问。本章涉及几个重要的概念，包括过程的活动、活动记录、名字的绑定、调用序列和返回序列等。学习本章需理解运行时存储空间的典型划分及各区域作用；活动记录的结构和功能；理解 3 种存储分配策略的特点，特别是栈式存储分配的基本思想及实现方法；掌握代表性语言非局部名字的访问方法，尤其是基于存取链和 DISPALY 表的有嵌套过程语言的非局部名字访问方法。

习题

8.1 常见的存储分配策略有哪些？它们分别适用于什么情况？

8.2 设有一个 Pascal 程序段：

```
Program pp( in, out):
    Var k : integer;
    Function f(n: integer): integer;
        Begin
            If n<=0 then f:=1
            Else f:= n*f(n-1)
        End;
    Begin
        K:=f(10);
    End.
```

如果我们采用 DISPLAY 表来解决非局部名字的访问，则当第二次递归进入 f 函数后，DISPLAY 表的内容是什么？此刻整个程序运行的栈区内容是怎么样的？

待出版，仅供2020杭电本科生上课使用，勿外传！