第八讲　运行时存储组织

2020-11

目标程序在目标机环境中运行时，都置身于自己的一个运行时存储空间。通常，在有操作系统的情况下，目标程序将在自己的逻辑地址空间内存储和运行。这样，编译程序在生成目标代码时应该明确程序的各类对象在逻辑地址空间内是如何存储的，以及目标代码运行时是如何使用和支配自己的逻辑存储空间的。在这一讲里，我们将讨论一些典型的与运行时存储组织相关的问题。首先，我们简要叙述运行时存储组织的作用与任务，程序运行时存储空间的典型布局，以及常见的运行时存储分配策略。然后，我们重点讨论实现栈式存储分配时栈帧（即活动记录）的组织。随后，我们讨论实现过程调用中参数传递的话题。最后，我们就面向对象程序的运行时存储组织的有关问题进行讨论。

1. 运行时存储组织的作用与任务

如上所述，编译程序在生成目标程序之前应该合理安排好目标程序在逻辑地址空间中存储资源的使用，这便是运行时存储组织所涉及的问题。

编译程序所产生的目标程序本身的大小通常是确定的，一般被存放在指定的专用存储区域，即**代码区**。相应地，目标程序运行过程中需要创建或访问的数据对象将被存放在**数据区**。数据对象包括用户定义的各种类型的命名对象（如变量和常量）、作为保留中间结果和传递参数的临时对象及调用过程时所需的连接信息等。

语言特征的差异对于存储组织方面的不同需求往往取决于数据对象的存储分配。因此，在这一讲里，我们讨论主要内容是面向数据对象的运行时存储组织与管理。以下列举了运行时存储组织通常所关注的几个重要问题：

- **数据对象的表示**。需要明确源语言中各类数据对象在目标机中的表示形式。
- **表达式计算**。需要明确如何正确有效地组织表达式的计算过程。
- **存储分配策略**。核心问题是如何正确有效地分配不同作用域或不同生命周期的数据对象的存储。
- **过程实现**。如何实现过程/函数调用以及参数传递。

数据对象在目标机中通常是以字节（byte）为单位分配存储空间。例如，对于基本数据类型，我们可以设定基本数据对象的大小为：char 数据对象，1 个字节；integer 数据对象，4 个字节；float 数据对象，8 个字节；boolean 数据对象，1 个字节。对于指针类型的数据对象，通常分配 1 个单位字长的空间，如在 32 位机器上 1 个单位字长为 4 个字节。

对于数据对象的存放，不同的目标机可能在某些方面有不同的要求。例如，一些机器中的数据是以**大端**（big endian）形式存放，而另一些机器中的数据则是以**小端**（little endian）形式存放。许多机器会要求数据对象的存储访问地址以一定方式**对齐**（alignment），如必须可以被 2，4，8 等整除，这种情况下某些字节数不足的数据对象在存放时需要考虑**留白**（padding）处理。

复合数据类型的数据对象通常根据它的组成部分依次分配存储空间。对于数组类型的数据对象，通常是分配一块连续的存储空间。对于多维数组，可以按行进行存放，也可以按列进行存放。对于结构体类型，通常以各个域为单位依次分配存储空间，对于复杂的域数据对象可以另辟空间进行存放。对于对象类型（类）的数据对象，实例变量像结构体的域一样存放在一块连续的存储区，而方法（成员函数）则是存放在其所属类的代码区。

表达式计算是程序状态变化的根本原因，频繁涉及到存储访问的操作。通常，表达式计算是利用栈区完成的，临时量和计算结果（或指向它们的指针）的存储空间一般被分配在当前过程活动记录（参见第 4 节）的顶部。

某些目标机设计了专门的运算数栈（或专用寄存器栈）用于表达式计算。对于普通表达式（不含函数调用）而言，一般可以估算出可否在运算数栈上实现完整的计算。在不能实现完整计算时，可以考虑在运算数栈上实现部分计算，而利用栈区辅助完成全部计算。当然，某些情况下表达式的计算只能利用栈区实现，比如对于使用了递归函数的表达式。

关于存储分配策略以及过程实现，我们在后续各节中讨论。

2. 程序运行时存储空间的布局

虽然一般来说程序运行时的存储空间从逻辑上可分为“代码区”和“数据区”两个主要部分，但为了方便存储组织与管理，往往需要将存储空间划分为更多的逻辑区域。具体的划分方法会依赖于目标机体系结构，但一般情况下至少含有保留地址区、代码区、静态数据区以及静态数据区等逻辑区域。图 1 给出一个程序运行时存储空间布局的典型例子。

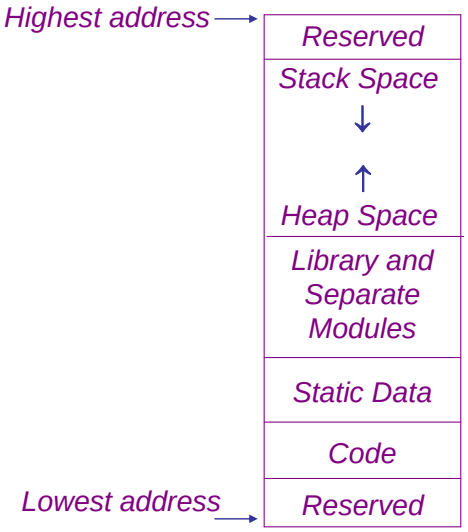


图 1　程序运行时存储空间布局的典型例子

对于图 1 中各逻辑存储区域，我们分别予以简单解释：

- **保留地址区**。专门为目标机体系结构和操作系统保留的内存地址区。通常，该区域不允许普通的用户程序存取，只允许操作系统的某些特权操作进行读写。
- **代码区**。静态存放编译程序产生的目标代码。

- **静态数据区**。静态存放全局数据，是普通程序可读可写的区域。该区域用于存放程序中用到的所有常量数据对象（如字符串常量，数值常量以及各种命名常量等），以及各类全局变量和静态变量所对应的数据对象。
- **共享库和分别编译模块区**。静态存放共享库模块和分别编译模块的代码和全局数据。运行库模块主要用来实现运行时支持，如 I/O、存储管理、执行期采样（profiling）以及调试等方面的例程。分别编译模块主要包含编译系统或用户预先订制的有用子程序和软件包（如数学子函数库）。这些模块是通过链接/装入（linker/loader）程序的装配而加入到当前程序的存储空间的。
- **动态数据区**。运行时动态变化的**堆区**和**栈区**。图 1 中假设栈区从高地址端向低地址变化，堆区从低地址端向高地址变化（而有些体系结构则刚好相反）。程序开始执行时会初始化堆区和栈区。一旦堆区和栈区在某个时刻相遇，则会发生存储访问冲突，因此每个会使堆区和栈区增长的操作都必须检查是否会产生这种冲突。如果冲突发生，则可能的解决方法是调用垃圾回收（garbage collection）或存储空间压缩（compation）程序将堆区和栈区分离。

值得注意的是，程序运行时的存储空间布局与目标机体系结构和操作系统密切相关。例如，IA-32 上某个 Linux 版本的用户程序虚拟存储空间如图 2 左图所示，MIPS-32 上 System V 的用户程序虚拟存储布局空间如图 2 右图所示。

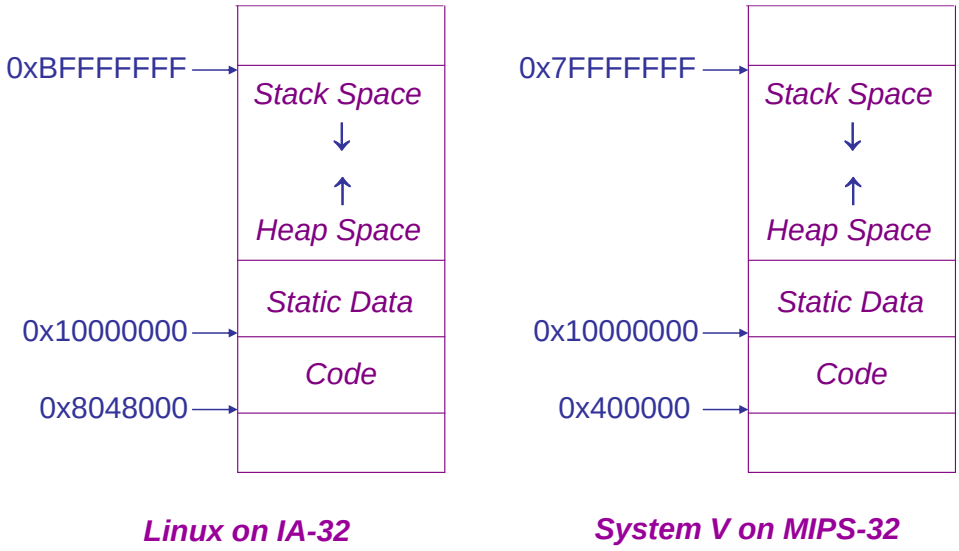


图 2　不同体系结构和操作系统的用户程序虚拟存储空间示例

3. 存储分配策略

从上面一节我们知道，数据区可以分为静态数据区（全局数据区）和动态数据区，后者又可分为堆区和栈区。之所以这样划分，是因为它们存放的数据和对应的管理方法的不同。静态数据区、栈区和堆区的存储空间分配分别遵循三种不同的规则：静态存储分配（Static Memory Allocation）、栈式存储分配（Stack-based Allocation）和堆式存储分配（Heap-based Allocation）。后两种分配方式皆称为“动态存储分配”，因为这两种方式中存储空间并不是在编译的时候静态分配好的，而是在运行时才进行的。

某些编程语言，如早期的 FORTRAN 语言及 Cobol 语言等，其存储分配是完全静态的，程序的数据对象与其存储的绑定（biding）是在编译期间进行的，称为静态语言（Static Languages）。相比较，对于另一些语言，所有数据对象与其存储的绑定只能发生在运行期间，此类语言称为动态语言（Dynamic Languages），如 LISP，ML，Perl，等等。多数语言（如 C/C++，Java，Pascal 等）采取的存储分配策略是介于二者之间。

下面我们分别来讨论静态、栈式和堆式三种存储分配策略。

3.1 静态存储分配

所谓**静态存储分配**，即为在编译期间为数据对象分配存储空间。这要求在编译期间就可确定数据对象的大小，同时还可以确定数据对象的数目。

采用这种方式，存储分配及其简单，但也会带来存储空间的浪费。为解决存储空间浪费问题，人们设计了变量的重叠布局（overlaying）机制，如 FORTRAN 语言的 equivalence 语句。重叠布局带来的问题是程序难写难度。完全静态分配的语言还有另一个缺陷，就是无法支持递归过程或函数。

多数（现代）语言只实施部分静态存储分配。可静态分配的数据对象如大小固定且在程序执行期间可全程访问的全局变量、静态变量、程序中的常量（literals）以及 class 的虚函数表等。如 C 语言中的 static 和 extern 变量，以及 C++ 中的 static 变量。这些数据对象的存储将被分配在静态数据区。

道理上讲，或许可以将静态数据对象与某个绝对存储地址绑定。然而，通常的做法是将静态数据对象的存取地址对应到偶对（DataAreaStart，Offset）。Offset 是在编译时刻确定的固定偏移量，而 DataAreaStart 则可以推迟到链接或运行时刻才确定。有时，DataAreaStart 的地址也可以装入某个基地址寄存器 Register，此时数据对象的存取地址对应到偶对（Register，Offset），即所谓的寄存器偏址寻址方式。

然而，对于一些动态的数据结构，例如动态数组（C++中使用 new 关键字来分配内存）以及递归函数的局部变量等最终空间大小必须在运行时才能确定的场合，静态存储分配就无能为力了。

3.2 栈式存储分配

栈区是作为“栈”这样一种数据结构来使用的动态存储区，我们称之为**运行栈**（run-time stack）。运行栈数据空间的存储和管理方式称为**栈式存储分配**，它将数据对象的运行时存储按照栈的方式来管理，常用于有效实现可动态嵌套的程序结构，如过程、函数以及嵌套程序块（分程序）等。

与静态存储分配方式不同，栈式存储分配是动态的，也就是说必须是运行的时候才能确定数据对象的存储分配结果。例如，对如下 C 代码片断：

```
int factorial (int n) {
    int tmp;
    if (n <= 1)
        return 1;
    else {
```

```
            tmp = n – 1;
            tmp = n * factorial(tmp);
            return tmp;
        }
    }
```

随着 n 的不同，这段代码运行时所需要的总内存空间大小是不同的，而且每次递归的时候 tmp 对应的内存单元都不同。

在过程/函数的实现中，参与栈式存储分配的存储单元是**活动记录**（activation record）。运行时每当进入一个过程/函数，就在栈顶为该过程/函数分配存放活动记录的数据空间。当一个过程/函数工作完毕返回时，它在栈顶的活动记录数据空间也随即释放。

在过程/函数的某一次执行中，其活动记录中会存放生存期在该过程/函数本次执行中的数据对象，以及必要的控制信息单元。相关内容将在第 4 节进行专门讨论，同时，也会讨论关于嵌套程序块的栈式存储分配。一般来说，运行栈中的数据通常都是属于某个过程/函数的活动记录，因此若没有特别指明，我们提到的活动记录均是指过程/函数的活动记录。

在编译期间，过程、函数以及嵌套程序块的活动记录大小（最大值）应该是可以确定的（以便进入的时候动态地分配活动记录的空间），这是进行栈式存储分配的必要条件，如果不满足则应该使用堆式存储管理。

3.3 堆式存储分配

当数据对象的生存期与创建它的过程/函数的执行期无关时，例如某些数据对象可能在该过程/函数结束之后仍然长期存在，就不适合进行栈式存储分配。一种灵活但是较昂贵的存储分配方法是**堆式存储分配**。 在堆式存储分配中，可以在任意时刻以任意次序从数据段的堆区分配和释放数据对象的运行时存储空间。通常，分配和释放数据对象的操作是应用程序通过向操作系统提出申请来实现的，因此要占用相当的时间。

堆区存储空间的分配和释放，可以是显式的（explicit allocation / dealocation），也可以是隐式的（implicit allocation / dealocation）。前者是指由程序员来负责应用程序的（堆）存储空间管理，可借助于编译器和运行时系统所提供的默认存储管理机制。后者是指（堆）存储空间的分配或释放不需要程序员负责，而是由编译器和运行时系统自动完成。

某些语言有显式的存储空间分配和释放命令，如：Pascal 中的 new/deposit，C++ 中的 new/delete。在 C 语言中没有显式的存储空间分配和释放语句，但程序员可以使用标准库中的函数 malloc() 和 free() 来实现显式的分配和释放。

某些语言支持隐式的堆区存储空间释放，这需要借助垃圾回收（garbage collection）机制。比如，Java 程序员不需要考虑对象的析构，堆区存储空间的释放是由垃圾回收程序（garbage collector）自动完成的。

对于堆区存储空间的释放，我们简单讨论一下不释放、显式释放以及隐式释放等三种方案的利弊。

- 不释放堆区存储空间的方法。这种方法只分配空间，不释放空间，待空间耗尽时停止。如果多数堆数据对象为一旦分配后永久使用，或者在虚存很大而无用数据对象

不致带来很大零乱的情形下，那么这种方案有可能是适合的。这种方案的存储管理机制很简单，开销很小，但应用面很窄，不是一种通用的解决方案。

- 显式释放堆区存储空间的方法。这种方法是由用户通过执行释放命令负责清空无用的数据空间，存储管理机制比较简单，开销较小，堆管理程序只维护可供分配命令使用的空闲空间。然而，该方案的问题是对程序员要求过高，程序的逻辑错误有可能导致灾难性的后果。如图 3 中代码所示的指针悬挂（dangling pointer）问题。

Pascal 代码片断	**C++ 代码片断**

var p,q: ^real;
…
new(p);
q:=p;
dispose(p);
q^:=1.0;
	float * p,*q;
	…
	p=new float;
	q=p;
	delete p;
	*q:=1.0;

图 3　存在指针悬挂问题的代码片断

- 隐式释放堆区存储空间的方法。该方案的优点是程序员不必考虑存储空间的释放，不会发生上述指针悬挂之类的问题，但缺点是对存储管理机制要求较高，需要堆区存储空间管理程序具备垃圾回收的能力。

由于在堆式存储分配中可以在任意时刻以任意次序分配和释放数据对象的存储空间，因此程序运行一段时间之后堆区存储空间可能被划分成许多块，有些占用，有些空闲。对于堆区存储空间的管理，通常需要好的存储分配算法，使得在面对多个可用的空闲存储块时，根据某些优化原则选择最合适的一个分配给当前数据对象。以下是几类常见的存储分配算法：

- 最佳适应算法，即选择空间浪费最少的存储块。
- 最先适应算法，即选择最先找到的足够大的存储块。
- 循环最先适应算法，即起始点不同的最先适应算法。

另外，由于每次分配后一般不会用尽空闲存储块的全部空间，而这些剩余的空间又不适合于分配给其它数据对象，因而在程序运行一段时间之后堆区存储空间可能出现许多“碎片”。这样，堆区存储空间的管理中通常需要用到碎片整理算法，用于压缩合并小的存储块，使其更可用。

有关垃圾回收、存储分配、碎片整理等算法的内容超出了本课范围，有兴趣的同学可参考龙书虎书等相关章节的讨论，部分内容也可参考数据结构和操作系统课程的相关话题。

4. 活动记录

在这一节里，我们将进一步讨论栈式存储分配的若干重要内容，这些内容都与活动记录密切相关。首先，我们介绍过程/函数运行时活动记录（简称过程活动记录）的典型结构。然后，我们讨论针对含嵌套过程说明语言的栈式分配中需要解决一个重要问题，即非局部量的访问。最后，我们简要讨论关于嵌套程序块中非局部量的访问。

4.1 过程活动记录

过程活动记录是指运行栈上的**栈帧**（frame），它在函数/过程调用时被创建，在函数/过程运行过程中被访问和修改，在函数/过程返回时被撤销。该栈帧包含局部变量，函数实参，临时值（用于表达式计算的中间单元）等数据信息以及必要的控制信息。

先看一个简单的例子，示意关于过程活动记录在运行栈上被创建的过程。首先，图 4 中的程序从函数 main 开始执行，在运行栈上创建 main 的活动记录；然后，从函数 main 中调用函数 p，在运行栈上创建 p 的活动记录；之后，p 中调用 q，又从 q 中再次调用 q。结果，函数 q 被第二次激活时运行栈上活动记录分配情况如图 4 所示。若某函数从它的一次执行返回时，相应的活动记录将从运行栈上撤销。例如，图三中的递归函数 q 执行完正常返回后的时刻，运行栈上将只包含 main 和 p 的活动记录。我们这里假定栈空间的增长方向是自下而上（不同于图 1），如不特别指明，本讲后续部分也这样假设。

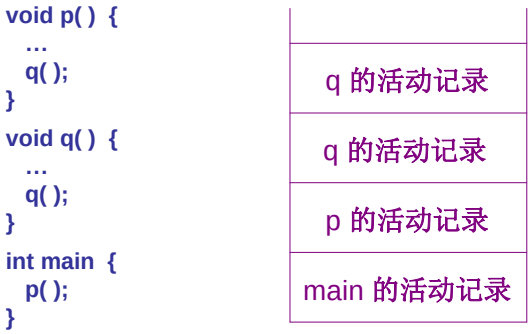


图 4　活动记录在运行栈上的分配

如图 5 所示，活动记录中的数据通常是使用寄存器偏址寻址方式进行访问的，即在一个基地址寄存器中存放着活动记录的首地址，在访问活动记录某一项内容的时候只需要使用该首地址以及该项内容相对这个首地址的偏移量即可计算出要访问的内容在虚拟内存中的逻辑地址。

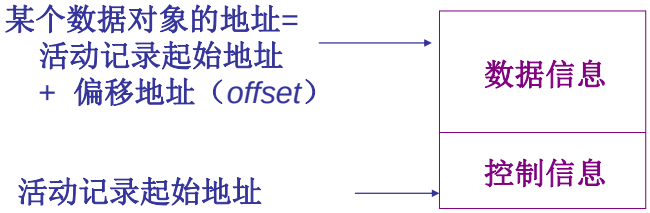


图 5　活动记录中数据对象的寻址

图 6 描述了一个典型过程活动记录的结构，其中的数据信息包括参数区，局部数据区，动态数据（如动态数组）区，临时数据区，以及过程/函数调用所需要的其它数据信息等。FP 为栈桢的基地值寄存器；TOP 为栈顶指针寄存器，通常指向运行栈中下一个可分配的单元。FP 和 TOP 的组合所确定的区域即为当前活动记录的存储区。控制信息通常包含一些联系单元，如返回地址、静态链及动态链等。有关静态链和动态链的内容参见 4.2 节。有关参数区，过程/函数调用以及参数传递方式的讨论参见第 5 节。

图 6　典型的过程活动记录结构

下面我们来看有关过程活动记录的两个小例子。设有如下 C 函数：

```
void p( int a)   {
    float b;
    float c[10];
    b=c[a];
}
```

图 7 描述该函数的一个可能的初始活动记录，其中的数据信息依次包括：实际参数 a（int 类型对象占 1 个单元），局部变量 b（float 类型对象占 2 个单元），以及数组变量 c 的各个分量（每个分量各占 2 个单元）。假设控制信息占 3 个单元，那么数据对象 a 和 b 的偏移量分别为 3，4，6。数组 c 的第 i（0≤ i ≤9）个元素的偏移量为 6+2i。设当前栈桢指针寄存器内容为$FP，则栈顶指针寄存器 TOP 的内容为$TOP = $FP +26。当语句 b=c[a] 开始执行时，所使用的临时数据对象将从$TOP 开始分配存储空间。

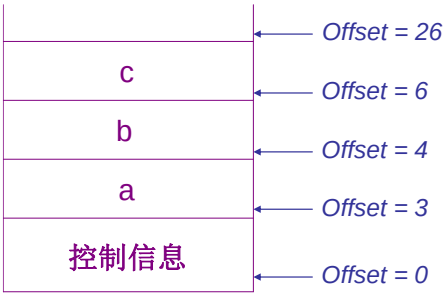


图 7　不含动态数据区的过程活动记录

下面我们看另外一个例子。设有如下 C 代码片断：

```
static int N;

void p( int a)   {
    float b;
    float c[10];
    float d[N];
    float e;
```

…

}

其中，d 被申明为一个动态数组。图 8 描述该函数的一个可能的初始活动记录，其中的数据信息依次包括：实际参数 a，局部变量 b，静态数组变量 c 的各个分量，动态数组 d 的内情向量和起始位置指针，然后是局部变量 e。对于动态数组 d，编译器并不能确定将需要多少存储空间，因此初始活动记录中占用了 2 个单元，其中内情向量单元用于存放 d 的上界 N，它的值将在运行时获得；另一个单元存放 d 的起始位置指针。如果采用相对$FP 的偏移量表示 d 的起始位置，那么可以在编译时确定 d 的起始偏移量为 offaet=30（思考：当有 2 个或 2 个以上动态数组时，则第 2 以后的数组将不能静态地确定起始位置）。如图 8 所示，数组 d 的第 i（0≤ i ≤N-1）个元素的偏移量为 30+2i。设当前栈桢指针寄存器内容为$FP，则在为数组 d 的所有元素动态分配空间后，栈顶指针寄存器 TOP 的内容为$TOP = $FP + 30 + 2N。

图 8　含动态数据区的过程活动记录

4.2　嵌套过程定义中非局部量的访问

Pascal，ML 等程序设计语言允许嵌套的过程/函数定义，这种情况下需要解决的一个重要问题就是非局部量的访问。图 9 所示是一个类 Pascal 程序的过程定义示例，其中过程 P 的定义内部含有过程 Q 的定义，而过程 Q 的定义中又含有过程 R 的定义。在嵌套的过程定义中，内层定义的过程体内可以访问包含它的外层过程中的数据对象。比如，在 R 的过程体内可以访问过程 Q、过程 P 以及主程序 Main 所定义的数据对象。更确切地说，在过程 R 被激活时，R 过程体内部可以访问过程 Q 最新一次被调用的活动记录中所保存的局部数据对象，同样也可以访问过程 P 最新一次被调用的活动记录中所保存的局部数据对象，以及可以访问主程序 Main 的活动记录中所保存的全局数据对象（假设全局数据对象也存放在栈区，此时 Main 的活动记录总存在且是唯一的）。这种对于不在当前活动记录中的数据对象的访问，我们称之为**非局部量的访问**。

在 C 语言等不支持嵌套过程/函数定义的程序设计语言中，非局部量只有全局变量，通常情况下可以分配在静态数据区。所以，我们不考虑这些语言中非局部量访问的问题。

```
program Main( I,O)
procedure P;
    procedure Q;
        procedure R;
            begin
                … R; …
            end;   /*R*/
        begin
            … R; …
        end;   /*Q*/
    begin
        … Q; …
    end;   /*P*/
procedure S;
    begin
        … P; …
    end;   /*S*/
begin
    …  S; …
end.   /*main*/
```

图 9　嵌套的过程定义

对于非局部量的访问，常见的实现方法有两种：。

- 采用 Display 表。参见 4.2.1 节。
- 为活动记录增加静态链域。参见 4.2.2 节。

4.2.1 Display 表

Display 表记录各嵌套层当前过程的活动记录在运行栈上的起始位置（基地址）。若当前激活过程的层次为 K（主程序的层次设为 0），则对应的 Display 表含有 K+1 个单元，依次存放着现行层，直接外层，… ，直至最外层的每一过程的最新活动记录的基地址。嵌套作用域规则可以确保每一时刻 Display 表内容的唯一性。Display 表的大小（即最多嵌套的层数）取决于具体实现。

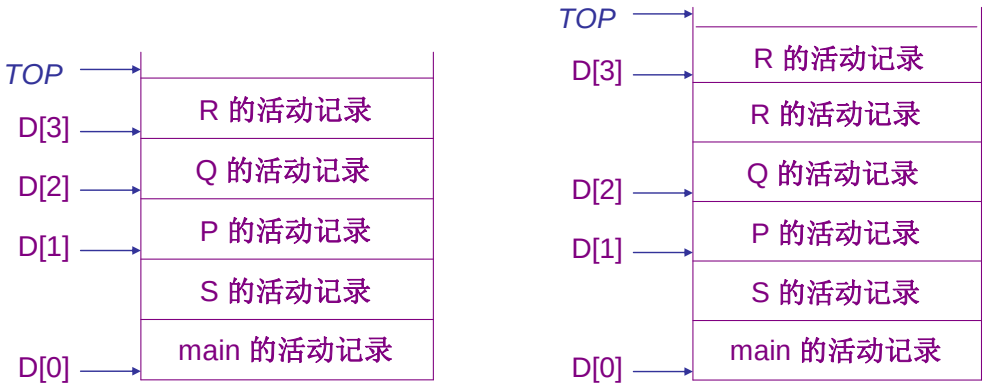


图 10　Display 表

例如，对于图 9 中的程序，过程 R 被第一次激活后运行栈和 Display 寄存器 D[i] 的情况如图 10 左边所示（假设无其它调用语句）。可以看出，当前 D[1]指向过程 P 活动记录的基地址，而非另一个第 1 层过程 S 的活动记录。当过程 R 被第二次激活后，D[3] 则指向过程

R 最新的活动记录，如图 10 右边所示。

在过程被调用和返回时，需要对 Display 表进行维护，涉及到 Display 寄存器 D[i]的保存和恢复。一种极端的方法是把整个 Display 表存入活动记录。若过程为第 n 层，则需要保存 D[0]~D[n]。一个过程（处于第 n 层）被调用时，从调用过程的 Display 表中自下向上抄录 n 个 FP 值，再加上本层的 FP 值。

例如，若采用这种方法，对于图 9 中的程序，过程 R 被第一次激活后 R 活动记录和 Q 活动记录中 Display 表的情况如图 11 左边所示。当过程 R 被第二次激活后，过程 R 的两个活动记录中 Display 表的情况如图 11 右边所示。

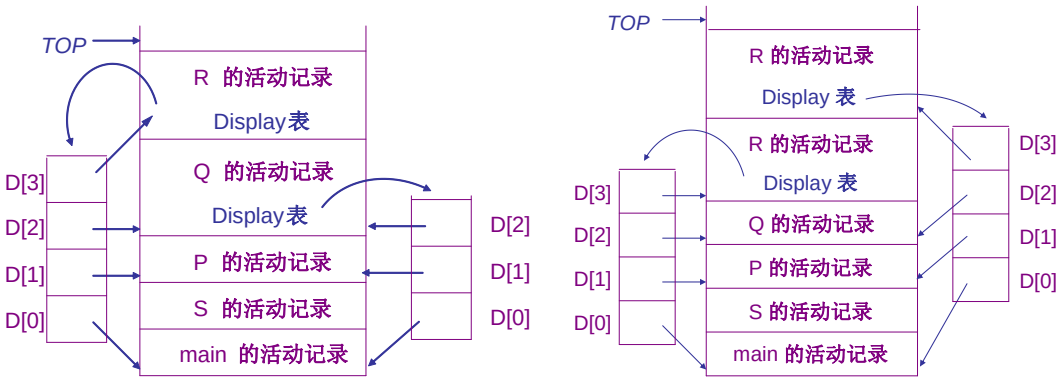


图 11　Display 表的维护（一）

显然，上述方案所记录的信息冗余度较大。可以采用的另一种方法是只在活动记录保存的一个 Display 表项，而在静态存储区或专用寄存器中维护一个全局 Display 表。如果一个处于第 n 层的过程被调用，则只需要在该过程的活动记录中保存 D[n]先前的值；如果 D[n]先前没有定义，那么我们用“_”代替。

例如，若采用第二种方法，对于图 9 中的程序，当过程 R 被第一次激活后，全局 Display 表以及各过程的活动记录中所保存的 Display 表项内容如图 12 左边所示。当过程 R 被第二次激活后，全局 Display 表以及各过程的活动记录中所保存的 Display 表项内容如图 12 右边所示。

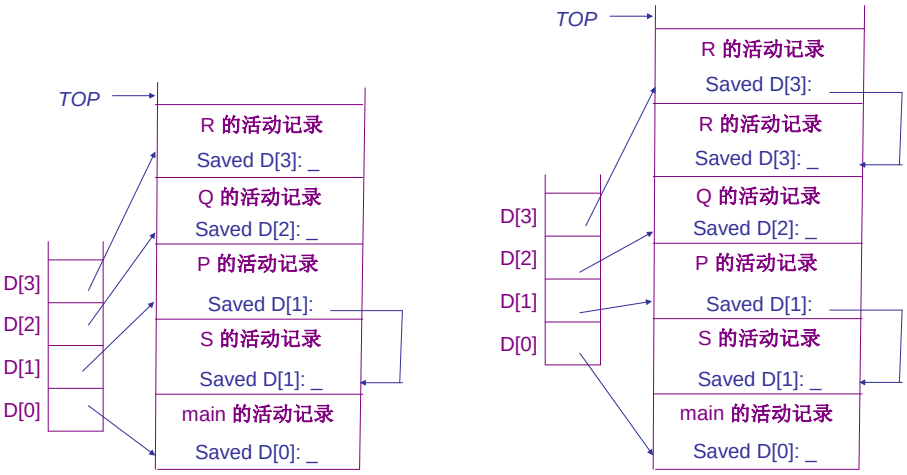


图 12　Display 表的维护（二）

为了进一步解释后一种方法，我们将图 9 中的程序略加修改，在 R 的过程体中原来调用 R 之处现改为调用 P，如图 13 所示。同样，我们假设无其它调用语句，则该程序的过程调用序列为 Main, S, P, Q, R, P, Q, R, …。若采用第二种 Display 表维护方法，对于图 13 中的程序，在执行头两轮 P, Q, R 调用序列时，全局 Display 表的内容以及各过程的活动记录中所保存的 Display 表项内容如图 13 右边所示。为了不至混淆，我们把其中第二轮调用序列表示为 P', Q', R'。从该图中可以看出，D[0]总是对应主程序的活动记录；在第一次调用 P 时，D[1] 由原来指向 S 的活动记录改为指向 P 的活动记录，而在 P 的活动记录中记录 S 活动记录的基地址以便从 P 返回时恢复原先的 D[1]值；在第一次调用 Q 和 R 时，由于 D[2]和 D[3]无定义，所以它们的活动记录中所保存的 Display 表项也无定义；在第二次调用时过程 P 时，D[1]的活动记录改为指向该过程的一个新活动记录（P'），而将原来的 P 活动记录基地址保存于 P' 活动记录的 Display 表项中；第二次调用过程 Q 和 R 时，情况也类似。

```
program Main( I,O)
procedure P;
    procedure Q;
        procedure R;
            begin
                … P; …
            end;   /*R*/
        begin
            … R; …
        end;   /*Q*/
    begin
        … Q; …
    end;   /*P*/
procedure S;
    begin
        … P; …
    end;   /*S*/
begin
    …  S; …
end.   /*main*/
```

calls	P	Q	R	P'	Q'	R'
D[3]	_	_	R	R	R	R'
D[2]	_	Q	Q	Q	Q'	Q'
D[1]	P	P	P	P'	P'	P'
D[0]	Main	Main	Main	Main	Main	Main
saved	S	_	_	P	Q	R

图 13　Display 表的维护示例

4.2.2 静态链

Display 表的方法要用到多个存储单元或多个寄存器，但有时并不情愿这样做。一种可选的方法是采用**静态链**（static link），也称**访问链**（access link），即在所有活动记录都增加一个域，指向定义该过程的直接外过程（或主程序）运行时最新的活动记录（的基址）。

与静态链对应的另一个概念是**动态链**（dynamic link），也称**控制链**（control link）。在过程返回时，当前活动记录要被撤销，为回卷（unwind）到调用过程的活动记录（恢复 FP），我们需要在被调用过程的活动记录中有这样一个域，即动态链，指向该调用过程的活动记录（的基址）。

例如，对于图 9 中的程序，当过程 R 被第一次激活后，运行栈以及各个活动记录的静态链和动态链域的情况如图 14 左边所示（假设无其它调用语句）。又如，对于图 13 中的程序，当过程 P 被第二次激活后，运行栈以及各个活动记录的静态链和动态链域的情况如图 14 右边所示。

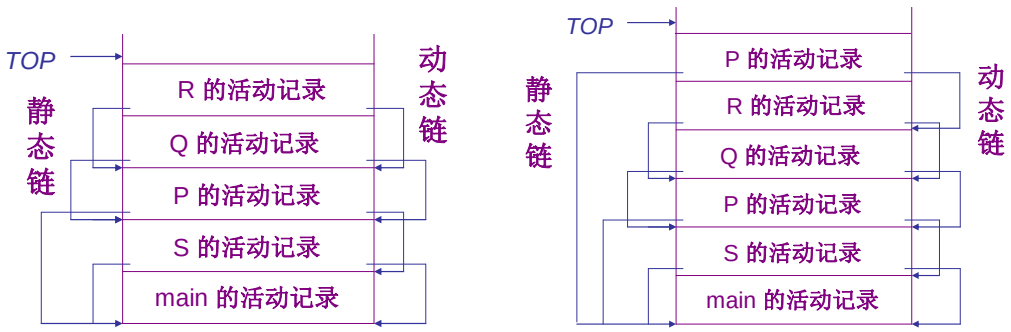


图 14　动态链与静态链

采用静态链比采用全局 Display 表的方法容易实现，但在进行非局部量访问时效率要比后者差。

4.3 嵌套程序块的非局部量访问

一些语言（如 C 语言）支持嵌套的块，在这些块的内部也允许声明局部变量，同样要解决依嵌套层次规则进行非局部量使用（访问）的问题。常见的实现方法有两种：。

- 将每个块看作为内嵌的无参过程，为它创建一个新的活动记录，称为**块级活动记录**。该方法的代价很高。
- 由于每个块中变量的相对位置在编译时就能确定下来，因此可以不创建块级活动记录，仅需要借用其所属的过程级活动记录就可解决问题（参见下面的例子）。

例如，对如下 C 代码片断：

```
int p（）
{
        int A;
        …
        {
                int B,C;
                 …
        }
        {
                int D,E,F;
                 …
                {
                        int G;
                        …   /*here*/
                }
        }
 }
```

针对上述代码片断中的嵌套程序块的非局部量访问，可以采用过程级活动记录，如图 15 所示。从图中可以看出，当程序运行至 /*here*/ 处时，存放 D 和 E 的空间重用了曾经存

放 B 和 C 的空间。

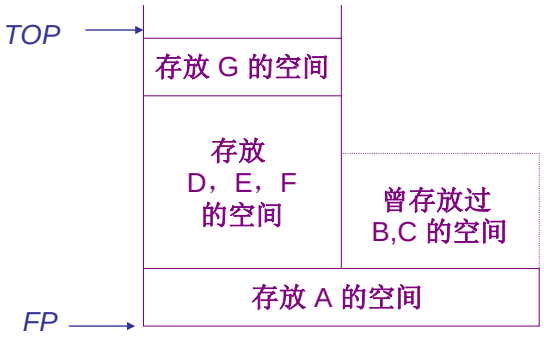


图 15　过程级活动记录中嵌套程序块的存储分配

4.4 动态作用域规则 vs. 静态（词法）作用域规则

一些语言（如 C 语言）支持嵌套的块，在这些块的内部也允许声明局部变量，同样要解决依嵌套层次规则进行非局部量使用（访问）的问题。常见的实现方法有两种：。

- 将每个块看作为内嵌的无参过程，为它创建一个新的活动记录，称为**块级活动记录**。该方法的代价很高。
- 由于每个块中变量的相对位置在编译时就能确定下来，因此可以不创建块级活动记录，仅需要借用其所属的过程级活动记录就可解决问题（参见下面的例子）。

例如，对如下 C 代码片断：

本课程中，若非特别指明，所讨论的内容均假定是基于静态作用域规则。

5. 过程调用与参数传递

图 16 给出了活动记录中与过程/函数调用相关的常见信息。实现过程/函数调用的控制信息中必需包含“调用程序返回地址”信息，以保证当前过程/函数运行结束时返回到**调用过程**（caller）继续执行。其它的控制信息包含某些必要的联系单元，如上一节介绍的动态链，静态链，display 表等。实现过程/函数调用的数据信息包含“实际参数”，“寄存器保存区”等。在初始化**被调用过程**（callee）的活动记录时将包含这些数据单元，同时也包含可以静态确定大小的局部数据单元。

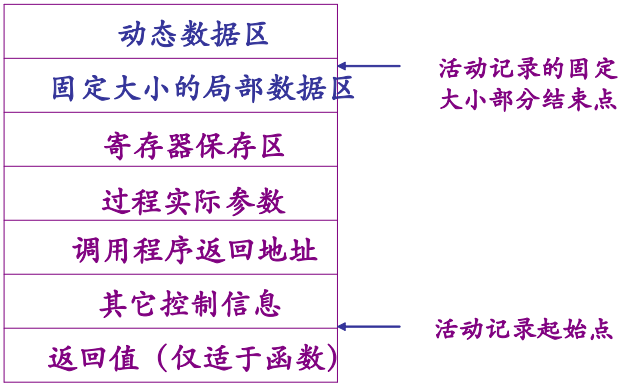


图 16　活动记录中与过程/函数调用相关的信息

实现过程调用时需要**调用代码序列**（calling sequence）为活动记录在栈中分配空间，并在相应单元中填写相应的信息。**返回代码序列**（return sequence）则与之呼应，它恢复机器状态（寄存器取值），使调用过程在调用结束后从返回地址开始继续执行。很自然，返回代码序列通常分配给被调用过程来完成。对于调用代码序列，通常是分配给被调用过程和调用过程分工来完成。这种分工没有严格的界限。然而，一般的原则是期望把调用代码序列中尽可能多的部分由被调用过程来完成，原因是调用点有多个而被调用过程只有一个，若分给后者则相应代码只需生成一次。调用代码序列和返回代码序列如何分工取决于不同系统平台（体系结构和操作系统）的**调用约定**（calling convention），是相应平台 ABI（application binary interface）的重要组成部分。

“返回值”信息仅适用于函数调用。该信息可以放在调用过程的活动记录，也可以放在被调用过程的活动记录。但若是后者，通常会把它的存放位置尽可能靠近调用过程的活动记录，以便调用过程可以在自身活动记录的顶部对返回值进行操作。

类似地，“过程实际参数区” 可以放在被调用过程的活动记录，也可以放在调用过程的活动记录。若是后者，通常会把它的存放位置靠近被调用过程的活动记录，以便被调用过程可以在自身活动记录的底部对参数值进行操作。

在许多平台的调用约定中，参数和返回值中有一部分是存放在特定的寄存器中，寄存器不够用时才会存放在活动记录中。

“寄存器保存区”用于保存过程/函数执行中可能被修改的寄存器值。依据不同的调用约定，有些寄存器值保存在被调用过程的活动记录，而另一些寄存器值则保存在调用过程的活动记录。

一般情况下，一个典型的过程调用和返回周期中需要执行三段代码。前两段是调用代码序列，分别由调用过程和被调用过程分担执行，称为**调用起始阶段**（prologue）。第三段是返回代码序列，由被调用过程执行，称为**调用收尾阶段**（epilogue）。

以下是调用起始阶段（prologue）需要完成的典型工作：

（1）参数传递。一些参数会传给寄存器；剩余的将被压入栈中（位于被调用程序活动记录或调用程序活动记录）。

（2）为被调用过程的活动记录分配栈上的存储空间。

（3）保存旧栈帧基址（保存旧 FP），即动态链信息。

（4）保存调用过程的返回地址（保存调用指令之后下一条指令的地址）。

（5）保存其它控制信息（如静态链、display 表等）。

（6）保存寄存器信息。通常可以分为由调用过程保存的寄存器（caller-saved registers）和被调用过程保存的寄存器（callee-saved registers）。后者最好用来保存生存期长的值，而前者则适合用于保存生存期短的值（不会跨越过程调用）。

（7）建立新栈帧基址（设置新 FP）。

（8）建立新栈顶（设置新 TOP）。

（9）转移控制，启动被调用过程的执行。

以下是被调用过程在调用收尾阶段（epilogue）需要完成的典型步骤：

（1）如果被调用过程是函数，则需要返回一个值。函数返回值可以存入专门的寄存器，也可以存入栈中（通常位于调用过程的活动记录）。

（2）恢复所有被调用过程保存的寄存器。

（3）弹出被调用过程的栈帧，恢复旧栈帧（即恢复调用时的 FP 和 TOP）。

（4）将控制返回给调用过程（恢复调用时保存的返回地址至指令计数器）。

最后，又调用过程保存的寄存器自然是由调用过程负责恢复。

对于不同的调用约定，上述各阶段的工作及其每一阶段工作中各步骤的完成者（调用过程或被调用过程）和先后次序会有一些差异。

在实现过程调用时，参数的传递方式也是很重要的环节。常见的参数传递方式如：

- **传值**（call-by-value）。传递的是实际参数的**右值**（r-value）。一个表达式的右值代表该表达式的取值。
- **传地址**（call-by-reference）。传递的是实际参数的**左值**（l-value）。一个表达式的左值代表存放该表达式值的存储单元地址。

考虑下列 Pascal 程序段：

```
procedure    swap(x,y:integer);
             var    temp:integer;
             begin
                        temp:=x;
                        x:=y;
                        y:=temp
             end;
```

若采用传值调用，调用过程 swap(a,b) 将不会影响 a 和 b 的值，其效果等价于执行下列

语句序列：

```
x :=a；
y :=b；
temp :=x；
x :=y；
y :=temp
```

在实现传值调用时，形式参数当作过程的局部数据对象处理，即在被调过程的活动记录中开辟了形参的存储空间，这些存储位置初始时用以存放实际参数值。调用过程计算实参的值，将其放于对应的存储空间。被调用过程执行时，就像使用局部变量一样使用这些参数单元。

若是换作下列 Pascal 程序段：

```
procedure    swap(var    x,y: integer);
        var    temp: integer;
        begin
                temp:=x;
                x:=y；
                y:=temp
        end；
```

保留字 var 告诉我们将采用传地址调用的参数传递方式。此时，调用过程 swap(a,b) 将交换 a 和 b 的值。

在实现传地址调用时，将把实际参数的地址传递给相应的形参，即调用过程把一个指向实参的存储地址的指针传递给被调用过程相应的形参：若实参是一个名字，或具有左值的表达式，则传递左值；若实参是无左值的表达式，则计算该表达式的值，放入一存储单元，然后将传此存储单元地址放于对应形参的存储空间。同样，被调用过程执行时，就像使用局部变量一样使用这些参数单元，但使用的是左值。

一些语言支持高阶函数（或过程），允许过程/函数作为输入参数（或者返回参数）。对于不含嵌套过程/函数声明的语言，比如 C 语言，处理起来比较简单。对于此类语言，任何过程/函数内部访问的非局部量只有全局量，我们可以将所有全局量分配在静态区。这样，无论采取什么方式激活一个过程/函数，不管是直接调用还是通过参数或返回值给出的过程/函数地址进行的间接调用，被激活过程/函数的活动记录没有什么差异，局部数据在活动记录中访问，而非局部数据只有全局量，均在静态区访问。对于包含嵌套过程/函数声明的语言，情况要复杂一些。为便于大家对照学习，我们照搬龙书中所给的下列 ML 程序来说明这种情形的一种解决方案。

```
fun a(x) =
    let
        fun b(f) =
            … f … ;
        fun c(y) =
            let
                fun d(z) = …
            in
                … b(d) …
            end
    in
        … c(1) …
    end;
```

在该程序中，函数 a 中嵌套声明了函数 b 和 c，并在函数体内调用了函数 c。函数 b 有一个函数形参 f，b 的函数体内调用了这个参数，激活了传进来的这个函数的一个实例。函数 c 内部定义了一个函数 d，且在函数体内用 d 作为实参调用了函数 b。同时，这里假设该程序中省略的部分不影响到我们的分析。

需要解决的一个核心问题是，在被调用的实参函数内部，如何访问其非局部量。具体来说，对于上面的例子，c 中调用 b 时通过调用实参函数激活了函数 d 的一个实例，那么在该实例的活动记录中，如何能够正确访问相对于函数 d 的非局部量？一种解决方案是，在执行 b(d) 时，所传递的函数实参不仅包含函数 d 本身，而且也包含其静态链（指向最新的 c 函数活动记录）。图 17 刻画了执行函数 a 时，当执行到 b 中通过调用 f 激活实参函数 d 的一个实例后栈上活动记录的情况。

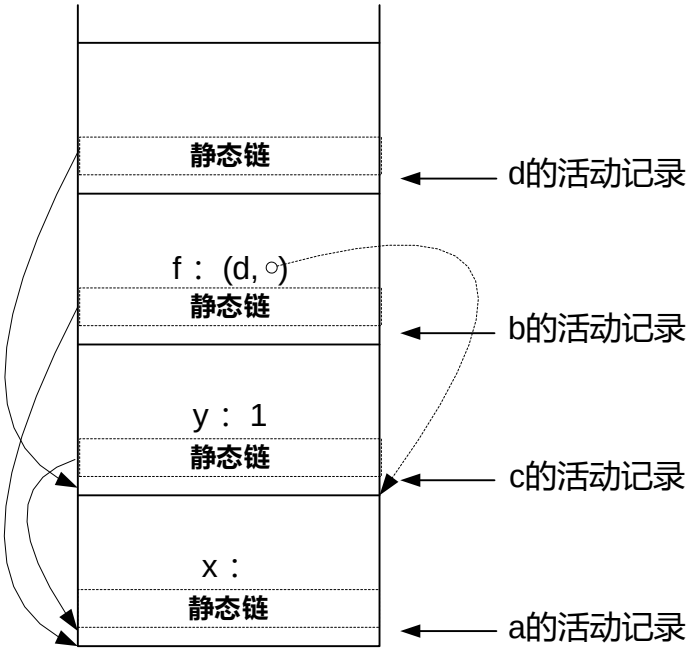


图 17　带静态链的函数实参

在该程序中，函数 a 中嵌套声明了函数 b 和 c，并在函数体内调用了函数 c。函数 b 有一个函数形参 f，b 的函数体内调用了这个参数，激活了传进来的这个函数的一个实例。函数 c 内部定义了一个函数 d，且在函数体内用 d 作为实参调用了函数 b。同时，这里假设该程序中省略的部分不影响到我们的分析。

这一解决方案中，值得注意的是，在 c 中能够获得知 d 的静态链信息（指向当前的 c 活动记录），然后连同 d 一起作为参数传递至所调用的 b 活动记录。这样，即使 b 不在 c 和 d 的定义中，b 在创立 d 的活动记录时也能够正确设置静态链。

语言设计中，也存在其它风格的参数传递方式（如传名字，call-by-name），本课不作进一步讨论，有兴趣的同学可参阅龙书等教材。

6. 面向对象程序运行时组织（选讲）

面向对象编程语言已经成为当今主要的程序设计语言，然而限于课时，本课程不打算全方位介绍面向对象语言的实现机制。其实，课程实验已经打造了一个较好的基础，通过总结和升华，大家不难应对实验中为涉及到的其它一些语言特征。第十讲的课堂教案中有不少内容是关于面向对象语言实现的，可以作为重要的参考材料。

在理解面向对象语言的实现机制时，对象的运行时存储组织是比较关键的环节。本节将讨论与此相关的几个问题。

6.1 “类”和“对象”的角色

面向对象语言中，与存储组织关系密切的概念是“**类**”和“**对象**”。首先，我们需要对“类”和“对象”在面向对象程序中所扮演的角色应该有很好的理解：

- 类扮演的角色是程序的静态定义。类是一组运行时对象的共同性质的静态描述。类声明中包含两类**特征**（feature）成员：**属性**（attribute）和**例程**（routine）。
- 对象扮演的角色是程序运行时的动态结构。每个对象都必定是某个类的一个**实例**（instance），而针对一个类可以创建有许多个对象。

除此之外，我们还必须熟知面向对象机制的主要特点，如封装（encapsulation）、继承（inheritance）、多态（polymorphism）、重载（overloading）及动态绑定（dynamic biding）等。关于这些内容，大家在面向对象编程或面向对象软件开发方法等相关的课程中已经有相当的了解。

6.2 面向对象程序运行时的特征

进一步，我们还需要充分理解面向对象程序运行时的基本特征：

- 对象是类的一个实例，是系统动态运行时一个物理结构的模块，是按需要创建、而不是预先分配的。对象是在类实例化过程中，由类的属性定义所确定的一组域动态地组成，每个域对应类中的一个属性。

- 执行一个面向对象程序就是创建系统**根类**（root class）的一个实例，并调用该实例的创建过程。创建**根对象**相当于通常程序启动 main 过程/函数，在非纯面向对象方式下，通常也采用启动 main 过程/函数的方式创建根对象。
- 创建对象的过程实现该对象初始化；对于根类而言，创建其对象即执行该系统。图 18 描绘了创建根对象时的存储结构。运行根对象构造例程时，在堆区为根对象申请空间并创建根对象，同时在栈区保存引用根对象的存储单元。

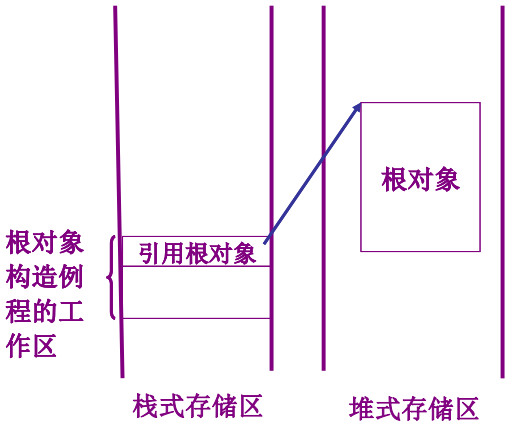


图 18　创建根对象时的存储结构

对象例程的运行一般具有如下特征：

- 每个例程都必定是某个类的成员，且每个例程都只能把它的计算施加在它所属类所创建的对象上。因而在一个例程执行前，首先要求它所施加计算的对象已经存在，否则要求先创建该对象。
- 一个例程执行时，其参数除实参外，还用到它所施加计算的对象，它们与该例程的局部量及返回值一起组成一个该例程的工作区（在栈区）。参见 6.4 节。
- 例程工作区中的局部量若是较为复杂数据结构，则在工作区中存放对该复杂数据结构的一个引用，并在堆区创建一个该复杂数据结构的对象。

6.3　对象的存储组织

关于对象的存储组织，一种最容易想到的设计方法可以是：初始化代码将所有当前的继承特征（属性和例程）直接地复制到对象存储区中（将例程当作代码指针）。但这样做的后果是空间浪费相当大。实际上是一种极端的方法。

另一种方法是：在对象存储区不保存任何继承而来的例程，而是在执行时将类结构的一个完整的描述保存在每个类的存储中，由超类指针维护继承性（形成所谓的继承图）。每个对象保存一个指向其所属类的指针，作为一个附加的域和它的属性变量放在一起，通过这个类就可找到所有（局部和继承的）的例程。这种方法只记录一次例程指针（在类结构中），且对于每个对象并不将其复制到存储器中。然而，其缺点在于：虽然属性变量具有可预测的偏移量（如在标准环境中的局部变量一样），但例程却没有，它们必须通过带有查询功能的符号表结构中的名字来维护。因为类结构是可以在执行中改变的，所以这是对于诸如 Smalltalk 等强动态性语言的合理的结构。实际上是另一种极端的方法，虽然节省了对象的

存储空间，但却增加了类层次（符号表）结构的维护，访存次数增多，故运行效率会受到很大影响。

下面我们介绍一种折衷的方案：计算出每个类的可用例程的代码指针列表（称为**例程索引表**，如 C++ 的 Vtable，简称**虚表**）。这一方法的优点在于：可做出安排以使每个例程都有一个可预测的偏移量，而且也不再需要用一系列表查询遍历类的层次结构。这样，每个对象不仅包括属性变量，还包括了一个相应的例程索引表的指针（不是类结构的指针）。

设有如下类和对象声明的片段：

```
class A { int x; void f (){…} }
class B extends A {void g(){…} }
class C entends B {void g(){…} }
class D extends C{bool y; void f (){…}}
class A a;
class B b;
class C c;
class D d1,d2;
```

这里，class A 的声明中含一个属性变量 x 和一个例程 f；class B 的声明中含一个例程 g，同时继承 class A 所声明的属性变量 x 和例程 f；class C 的声明中含一个例程 g（重载了 class B 中声明的例程 g），同时继承其祖先类中所声明的属性变量 x 和例程 f；类似地，class D 声明了属性变量 y，重载了例程 f，继承了（class A 声明的）属性变量 x 和（class C 声明的）例程 g。该代码片段声明了 5 个由类声明的变量：a，类型为 class A；b，类型为 class B；c，类型为 class C；d1 和 d2，类型为 class C。变量 a 初始化后（如在我们随后的例子中采用表达式 new（A）来初始化一个类 A 的对象）创建对象 a，它将占据独立的内存空间。类似地，我们有对象 b，对象 c，对象 d1 和对象 d2。

针对以上所声明的类和对象，图 19 给出了采用这种折衷方法的对象存储示例。

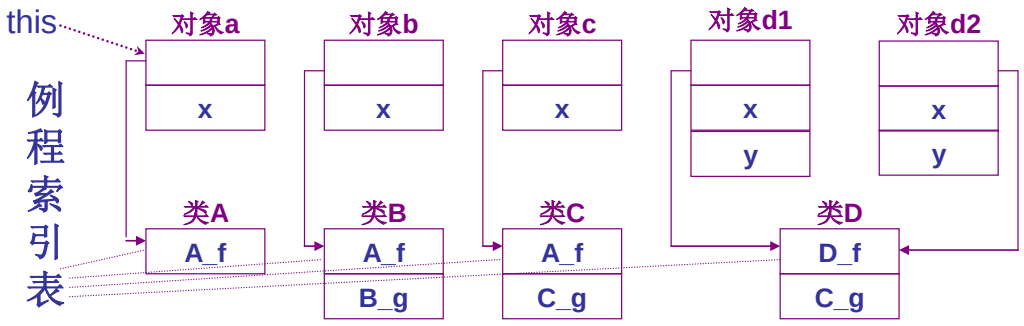


图 19　对象存储示例

从图 19 中可以看出，每一个对象都对应着一个记录这个对象状态的内存块（存放于堆区），其中包括了这个对象所属类的例程索引表指针（位于内存块开始的位置）和所有用于说明这个对象状态的属性变量。属性变量的排列顺序是：“辈分”越高的属性变量越靠前。具体到对象 d1 和 d2，属性变量 y 是这些对象的所属类 C 中声明的，而属性变量 x 是 C 继承父辈类的，所以在 d1 和 d2 的存储区中，属性变量 x 的存储位置排在属性变量 y 的存储位置之前。

根据这一方法，每个类都对应一个例程索引表。例程索引表的结构类似于 C++ 中的虚表。如图 19 所示，类 A 的例程索引表包含指向 class A 中声明的例程 f 的指针 A_f；类 B 的例程索引表包含指向 class A 所声明的例程 f 的指针 A_f，以及指向 class B 中声明的例程 g 的指针 B_g；类 C 的例程索引表包含指向 class A 所声明的例程 f 的指针 A_f，以及指向 class C 中声明的例程 g 的指针 C_g；最后，类 D 的例程索引表包含指向 class D 所声明的例程 f 的指针 D_f，以及指向 class C 中声明的例程 g 的指针 C_g。值得注意的是，在例程索引表中，我们安排继承而来的例程靠前排列，“辈分”越高的例程越靠前（如在类 B 和类 C 的例程索引表中，A_f 排列靠前），但重载例程的位置仍然保持被重载例程的位置（如类 D 的例程索引表中，D_f 排在 C_g 之前）。

值得注意的是，有些面向对象语言允许将例程声明为静态的。由于静态例程可以像普通函数那样直接调用，不需要动态绑定，所以例程索引表中不包含静态例程的指针。

6.4 例程的动态绑定

我们先了解一下针对面向对象语言中 this 关键字的通常处理方法，这有助于理解例程的动态绑定。

在通常的面向对象语言中，在例程内部可以使用 this 关键字来获得对当前对象的引用，同时在例程内部对属性变量或者例程的访问实际上都隐含着对 this 的访问。例如，若在名为 writeName 例程内使用了 this 关键字，则调用 who.writeName() 的时候 this 所引用的对象即为变量 who 所引用的对象。同样，如果是调用 you.writeName()，则 writeName 里面的 this 将引用 you 所指的对象。实现这一特征的一种方法是把 who 或者 you 作为 writeName 的一个实际参数在调用 writeName 的时候传进去，这样就可以把对 this 的引用全部转化为对这个参数的引用。这样，调用 who.writeName() 实际上相当于调用 writeName(who)。

这种技术可以推广至任何情形下的例程动态绑定的实现，即例程在实际运行时所邦定的对象是作为参数动态告诉它的。

下面是一个例子。设有某个简单的单继承面向对象语言的如下代码片段：

```
string day;

class Fruit
{
    int price;
    string name;
    void init(int p,string s) {price=p; name=s;}
    void print(){
        Print("On ",day,", the price of ",name, " is ",price,"\n");
    }
}

class Apple extends Fruit
{
    string color;
    void setcolor(string c) {color=c;}
    void print(){
```

```
            Print("On ",day,", the price of ",color, " ",name," is ", price,"\n");
        }
    }

    void foo()
    {
        class Apple a;
        a=New (Apple);
        a.setcolor("red");
        a.init(100,"apple");
        day="Tuesday";
        a.print();
    }
```

当上述程序执行语句 a.init(100,"apple") 时，实际上是调用 Fruit 类中声明的 init 例程的代码。换句话说，例程调用 init(100,"apple") 动态绑定到变量 a 所指示的对象，即一个 Apple 对象，如图 20 所示。此时，a 作为实际参数传给 this，后者是调用 init(100,"apple") 时的隐含的参数。

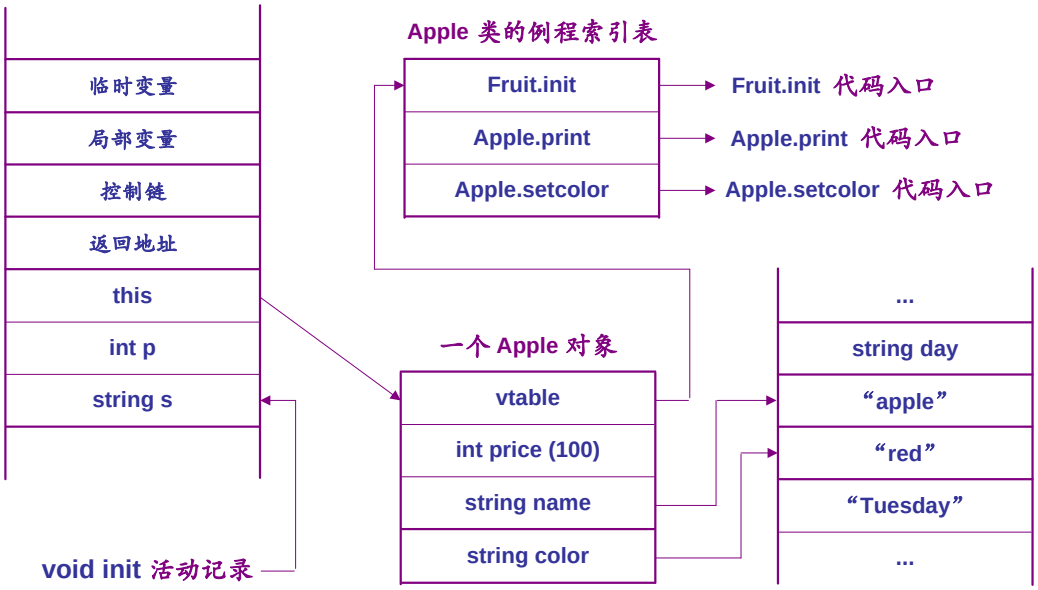


图 20　例程的动态绑定

这个面向对象程序的例子类似于我们主体实验中的 Decaf 程序，主要差别是后者不允许独立定义函数（即函数只能作为某个类中的方法来定义）。

6.5 其他话题

关于面向对象语言的实现机制还有许多有趣的话题，限于篇幅我们不能一一讨论，例如：

- 类成员测试（Testing Class Membership）。在课程实验中我们尝试过实现这一语言特征（回顾 instanceof 的实现），我们的方案是在每个类的 Vtable 中增加两个单元的信息（指向父类 Vtable 的指针和指向本类名字串的指针）。实现这一特征时，还可

以采用其他效率更高的方法，如采用 Display 表的方法，有兴趣的同学可以参考虎书介绍的相关内容。

- 对象的创建和撤消。如对象的构造和析构、垃圾回收等内容。
- 对象的操作。如对象的赋值、克隆、比较、持久存储等内容。
- 多继承性。
- 例外处理机制。
- ……

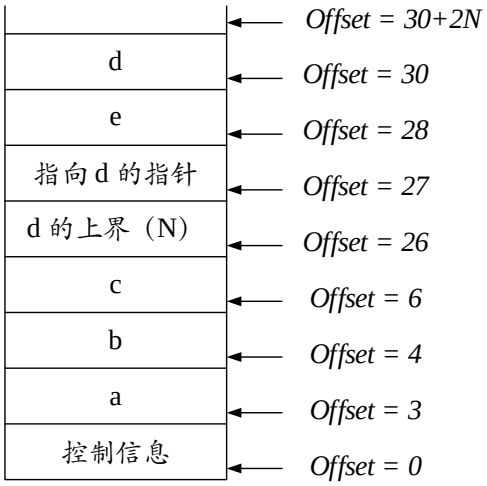
练习

1.

若按照某种运行时组织方式，如下函数 p 被激活时的过程活动记录如图 2 所示。其中 d 是动态数组。

```
static int N;

void p( int a)   {
    float b;
    float c[10];
    float d[N];
    float e;
    …
}
```



试指出函数 p 中访问 d[i]（0 ≤ i < N）时相对于活动记录基址的 *Offset* 值如何计算？若将数组 c 和 d 的声明次序颠倒，则d[i]（0 ≤ i < N）又如何计算？（对于后一问题可选多种不同的运行时组织方式，回答可多样，但需作相应的解释）

2. 下图左边是某简单语言的一段代码。语言中不包含数据类型的声明，所有变量的类型默认为整型（假设占用一个存储单元）。语句块的括号为‘begin’和‘end’组合；赋值号为 ‘:=’。每一个过程声明对应一个静态作用域。该语言支持嵌套的过程声明，但只能定义无参过程，且没有返回值。过程活动记录中的控制信息包括静态链 SL，动态链 DL，以及返回地址 RA。程序的执行遵循静态作用域规则。下图左边的 PL/0 程序执行到过程 p 被第二次激活时，运行栈的当前状态如下图右半部分所示（栈顶指向单元 26），其中变量的名字用于代表相应的内容。试补齐该运行状态下，单元18、19、21、22、及 23 中的内容。

```
（1）  var a,b;
（2）  procedure  p ;
（3）        var x;
（4）         procedure  r ;
（5）              var x, a;
（6）              begin
（7）                   a := 3;
（8）                   if   a > b  then call q;
 .                     …… /*仅含符号 x*/
 .                end;
 .           begin
 .                call r ;
 .                …… /*仅含符号 x*/
 .           end ;
 .      procedure  q ;
 .           var x;
 .           begin
（L）             if   a < b  then call p ;
 .                …… /*仅含符号 x*/
 .           end ;
 .      begin
 .           a := 1;
 .           b := 2;
 .           call q;
 .           ……
 .      end .
```

25	x	
24	?	RA
23		DL
22		SL
21		
20	?	RA
19		DL
18		SL
17	a	
16	x	
15	?	RA
14	9	DL
13	9	SL
12	x	
11	?	RA
10	5	DL
9	0	SL
8	x	
7	?	RA
6	0	DL
5	0	SL
4	b	
3	a	
2	?	RA
1	0	DL
0	0	SL

3. 若在第2题中，我们采用 Display 表来代替静态链。假设采用只在活动记录保存一个Display 表项的方法，且该表项占居图中SL的位置。（1）指出当前运行状态下Display 表的内容；（2）指出各活动记录中所保存的Display 表项的内容（即图中所有SL位置的新内容）

4. 在第2题中，若采取动态作用域规则，该程序的执行效果与之前有何不同？