第八讲　运行时存储组织

2024-11

本讲讨论一些典型的与运行时存储组织相关的问题。首先，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，重点讨论实现栈式存储分配时栈帧（即活动记录）的组织，以及实现过程调用中参数传递的话题。随后，介绍几种典型的垃圾回收机制。最后，简要讨论面向对象和函数式语言的运行时存储组织的若干问题。

1. 运行时存储组织的作用与任务

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。简言之，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，是运行时存储组织的作用与任务。

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。

语言特性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，我们主要讨论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；等等。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。

图 1 所示为 C/C++数据类型的对象在目标处理器 RISC-V 32 和 RISC-V 64 中的数据宽度以及对齐方式所对应的字节数[1]，二者分别基于 ILP32 和 LP64 约定。

C/C++ 类型	类型描述	Bytes in RV32		Bytes in RV64	
		Size	Alignment	Size	Alignment
bool/_Bool	Boolean value	1	1	1	1
char	Character value/byte	1	1	1	1
short	Short integer	2	2	2	2
int	Integer	4	4	4	4
long	Long integer	4	4	8	8
long long	Long long integer	8	8	8	8
_int128	128-bit integer	-	-	16	16
void*	Pointer	4	4	8	8
_float16	16-bit float	2	2	2	2
float	Single-precision float	4	4	4	4
double	Double-precision float	8	8	8	8
long double	Extended-precision float	16	16	16	16

图 1 数据类型在处理器中的数据宽度及对齐方式（字节数）

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。对于记录/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，表达式计算至少可以利用栈区完成，即临时量和计算结果（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。当然，对于比如仅含算逻运算的普通表达式计算，往往可以充分利用寄存器高效完成。

某些处理器设计了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专用寄存器栈上实现完整的计算。在不能实现完整计算时，可以考虑利用栈区和通用寄存器来辅助完成全部计算。当然，某些情况下，比如对于包含递归函数的表达式的计算离不开使用栈区。

关于存储分配策略以及过程实现，本讲后续会讨论。

2. 程序运行时存储空间的布局

一般来说，应用程序运行时的虚拟存储空间从逻辑上可分为“代码区”和“数据区”两个主要部分。为方便存储组织与管理，往往需要将用户程序的虚拟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体系结构和操作系统，但一般情况下至少含有保留地址区、代码区、静态数据区以及动态数据区等逻辑区域。图 2 给出一个程序运行时存储空间

布局的典型例子。

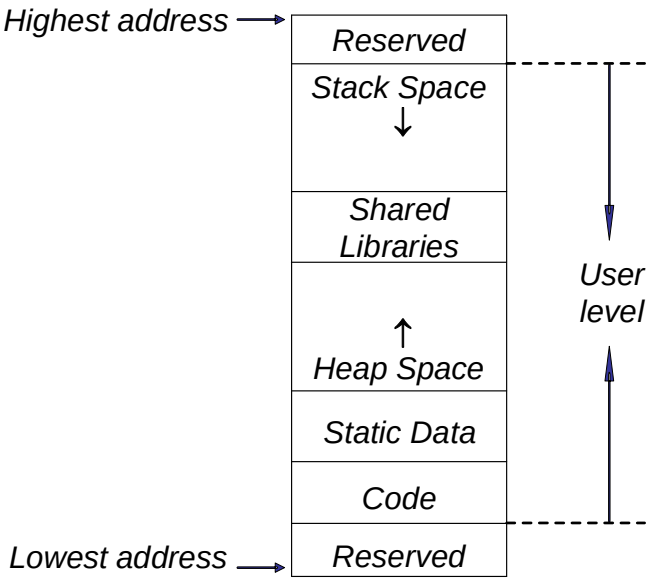


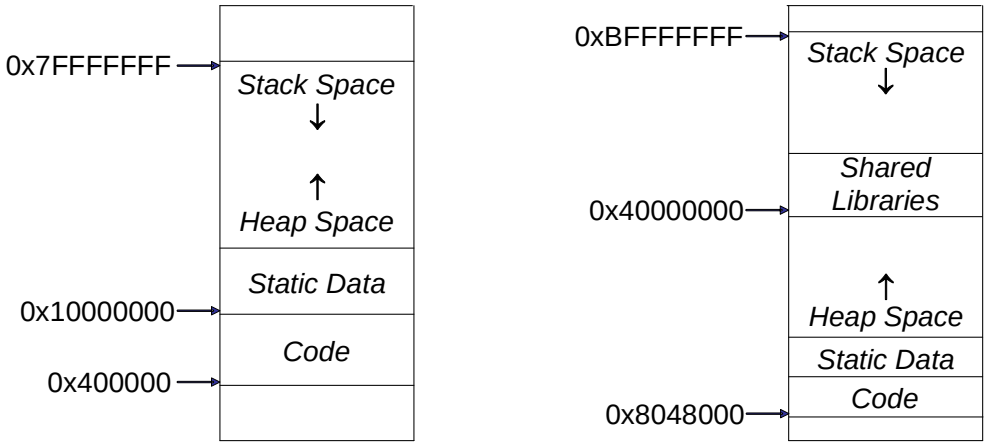
图 2　程序运行时存储空间布局的典型例子

对于图 2 中各逻辑存储区域，下面分别予以简单解释：

- 保留地址区。专门为目标体系结构/操作系统保留的内存地址区。通常，该区域不允许普通用户层程序存取，只允许操作系统的某些特权操作进行读写，处理器的操作模式需处于内核模式（*kernel mode*）。对于普通用户程序，处理器的操作模式一般处于用户模式（*user mode*），仅能访问保留地址区之外的虚拟存储区域。
- 代码区。用于存放编译期间产生的用户层代码。
- 静态数据区。用于存放编译期间产生的静态分配的全局数据。该区域通常会划分为两个部分：一个是静态初始化的数据存放区，用于存放程序中用到的所有常量数据对象（如字符串常量，数值常量以及各种名字常量等）和静态分配的有初值的全局变量数据；另一个是存放未初始化的静态分配的全局变量，一般是将其全部初始化为 0，传统上将这一静态数据区称为 *BSS* 区。
- 动态数据区。用于存放用户层代码运行期间所分配和使用的数据，可分为堆区和栈区。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。程序开始执行时会初始化堆区和栈区的内容，然后随着程序的运行其中存放的数据动态发生变化。
- 共享库和分别编译模块区。存放共享库模块和分别编译模块的代码和全局数据。这些模块是通过链接/装入（linker/loader）程序被静态或动态装配到当前程序的运行时空间。若是采用静态链接（static linking），由于所需空间可以在程序运行前静态确定，所以可选择将其分配在堆区或栈区的起始位置。若是采用动态链接（dynamic linking），则可以将其起始位置设在动态数据区中部的适当位置。

例如，MIPS-32 上 Linux/Unix 的用户程序虚拟存储布局空间如图 3(a)所示。虚拟存储空间的大小为 4 GB，可访问地址范围从 0x00000000 到 0xffffffff。用户程序使用的存储空间范围从 0x400000 到 0x7fffffff，代码区起始地址为 0x400000，静态数据区起始地址为

0x10000000。编译程序可以确定静态数据区的大小，因而动态数据区的范围也是确定的。



(*a*) MIPS-32 虚拟地址空间　　(*b*) IA-32 虚拟地址空间

图 3　用户程序虚拟存储空间示例

又如，IA-32 上 Linux/Unix 的用户程序虚拟存储布局空间如图 3(b)所示[2]。用户程序使用的存储空间范围从 0x08048000 到 0xbfffffff，代码区起始地址为 0x08048000，静态数据区从代码区之后的下一个 4-KB 对齐地址开始，堆区从静态数据区之后的第一个 4-KB 对齐地址开始向上生长，而用户程序使用的栈区则始于 0xbfffffff 向下生长。另外，动态链接时存放共享库模块（代码和数据）将被装入到从 0x40000000 起始的动态数据区。从 0xc0000000 往上是为操作系统内核代码和数据所保留的地址空间。从地址 0x08047fff 开始往下为保留空间，必要时可作他用。

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 语言和 Cobol 语言，以及安全实时控制领域的语言 Lustre，等等，其存储分配是完全静态的，程序的数据对象与其存储空间的绑定（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，数据对象与其存储的绑定基本上是发生在运行期间，此类语言称为动态语言（Dynamic Languages），如 Smalltalk，LISP，Perl，Python，等等。多数语言（如 C/C++，Java，Pascal 等）采取的存储分配策略介于二者之间。

下面分别讨论静态、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**，是指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读。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，后者无法在编译期间确定所访问的数据对象的大小。

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、程序中的字面量（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虚拟存储空间的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，而基地址 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，就不适合静态存储分配。

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 代码片段：

```
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，这段代码运行时所达到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）一般是明确的，这样进入的时候可以初始化所需要的栈空间。如果不确定，则更适合使用堆式存储管理。

有关活动记录的更多内容，参见第 4 节。

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，垃圾回收程序（*garbage collector*）会自动完成堆区存储空间的释放。

对于堆区存储空间的释放问题，我们简单讨论如下三种可选方案的利弊。

（1）不释放堆区存储空间的方法

这种方法只进行空间的分配，不考虑释放空间，待空间耗尽时停止。若是堆区多数数据对象一旦分配便会永久使用，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。

（2）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 4 中代码所示的指针悬挂（*dangling pointer*）问题。

C++ 代码片断	Pascal 代码片断

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

图 4　存在指针悬挂问题的代码片断举例

（3）隐式释放堆区存储空间的方法

该方法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存储块分配给当前数据对象。以下是常见的存储分配算法：

- 最佳适应算法。此类算法会选择空间浪费最少的存储块。
- 最先适应算法。此类算法会选择最先找到的足够大的存储块。
- 循环最先适应算法。此类算法相当于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，使存储空间的使用更加有效。

第 6 节将介绍几种常见的垃圾回收方法。其它有关堆存储空间管理的内容，如存储分配、碎片整理等算法超出了本课程范围，有兴趣的同学可参考龙书[4]和虎书[5]等教材中相关章节的讨论，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

这一节讨论栈式存储分配的若干问题，其内容均与活动记录密切相关。活动记录（*activation record*）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。

例如，图 5（a）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 5（b）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 5（a）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，如图 5（c）。为方便讨论，这里在绘制时假定栈空间的增长方向是自下而上，与现实的体系结构中运行栈不同，后者传统上是从高地址到低地址自上而下增长（参见第 2 和第 5 节）。如不特

别指明，本讲后续部分多数情况下也是这样来绘制。

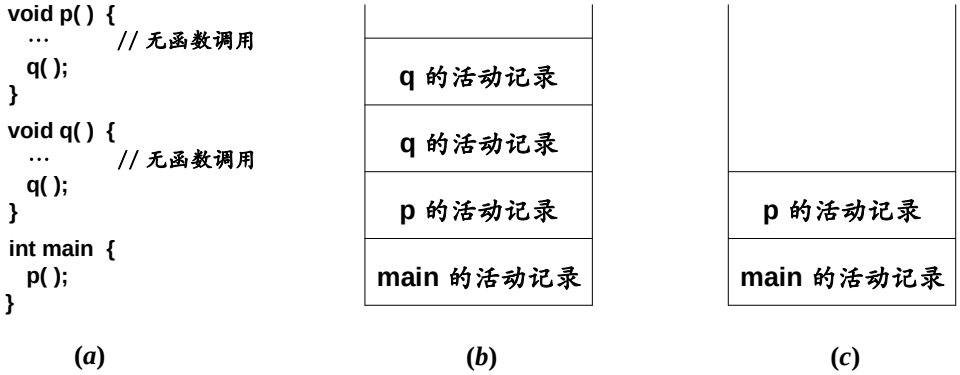


图 5　活动记录在运行栈上的分配

4.1 活动记录的创建与撤销

图 6 给出一种活动记录的典型结构。通常，活动记录中存放有函数/过程的参数单元，一些控制信息（如返回地址和栈帧基址等）的单元，其他需要保存的数据单元（如需要保存的寄存器），局部数据单元（如局部变量）以及保存中间计算结果的临时数据单元。运行栈空间需要动态分配的情形很少，因此一般在编译期间可以确定活动记录所需运行栈空间的大小 *framesize*（字节数）。

通常会使用两个专用寄存器用来指示或访问活动记录的栈帧空间。前者称为帧指针（*frame pointer*）寄存器或帧寄存器，作为基地址指向当前栈帧首个单元。后者称为栈指针（*stack pointer*）寄存器或栈寄存器，指向当前栈帧的最末单元，即栈顶单元。图 6 中，我们用 *FP* 表示帧寄存器，用 *SP* 表示栈寄存器。活动记录中的单元可以基于 *SP* 或 *FP* 加偏移量（*offset*）的方式寻址。

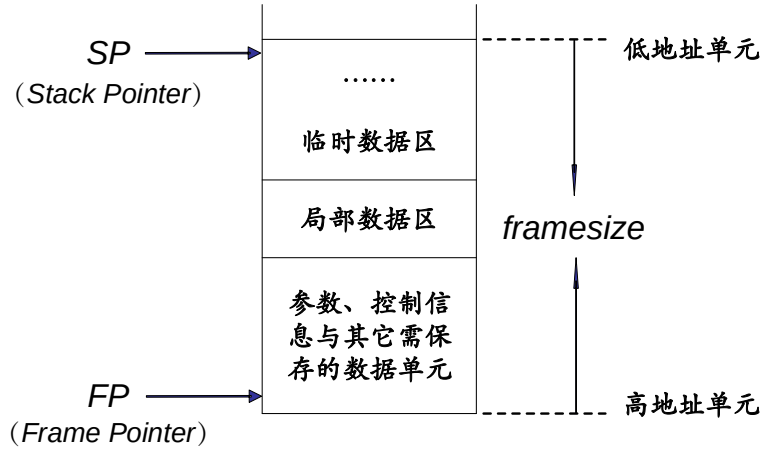


图 6　活动记录的典型结构

实现函数/过程的调用和返回时，编译器一般需要根据某种调用约定（*calling convention*）生成相关代码，参见第 5 节。在执行函数/过程体代码之前，首先需要执行一段调用发起阶段代码（*prologue*），创建并初始化新的活动记录。在执行函数/过程体代码之后从函数/过程

返回时，需要执行一段调用收尾阶段代码（*epilogue*），撤销当前栈帧并回到前一个活动记录。

例如，以下步骤描述了某个调用发起阶段代码的核心工作：

（1）*SP* ← *SP* – *framesize* ;　　// 在原来 *SP* 内容基础上减去 *framesize*，重置栈寄存器 *SP*，相当于设置新栈帧的空间大小为 *framesize* 个字节

（2）*framesize*-4 (*SP*) ← *RA* ;　　// 将返回地址 *RA* 的内容保存至新栈帧的首地址单元，栈帧首地址为当前栈寄存器 *SP* 的内容加上 *framesize*-4（这里每个单元占用 4 个字节，32 位字长）

（3）*framesize*-8 (*SP*) ← *FP* ;　　// 将前一个栈帧的帧寄存器（*FP*）值保存至新栈帧的第 2 个单元，其地址为当前栈寄存器 *SP* 的内容加上 *framesize*-8

（4）*FP* ← *SP* + *framesize*-4 ;　　// 置新栈帧的帧寄存器（*FP*）值为 *SP* 的内容加上 *framesize*-4，使帧指针指向新栈帧的首地址单元

从图 7（a）到图 7（b），可示意以上 4 个步骤（或指令）执行之后，活动记录及栈/帧寄存器的变化情况。

此例中，活动记录存放了返回地址 *RA* 的内容，大多数情况下是需要的，除非可以确定在当前被调用的函数（*callee*）是叶子函数（*leaf function*），即不会再调用其他函数。

此例的活动记录中还存放了前一个栈帧的帧寄存器 *FP* 的内容。保存这一 *FP* 信息许多情况下并非必须的。而当函数执行过程中需要动态分配栈空间时，*SP* 也会动态改变，此时活动记录中的单元需要借助 *FP* 来寻址。另外，若需支持动态作用域规则（参见 4.5 节），则栈中必须要保存 *FP* 的内容。这一信息称为动态链（参见 4.3 节）。

本例未考虑有关参数入栈的问题。对此，如果用于传参的寄存器（依据调用约定）数目足够，则参数无需占用栈空间。若需要，常常会把参数置于前一个栈帧的尾部或者当前栈帧的起始部分。

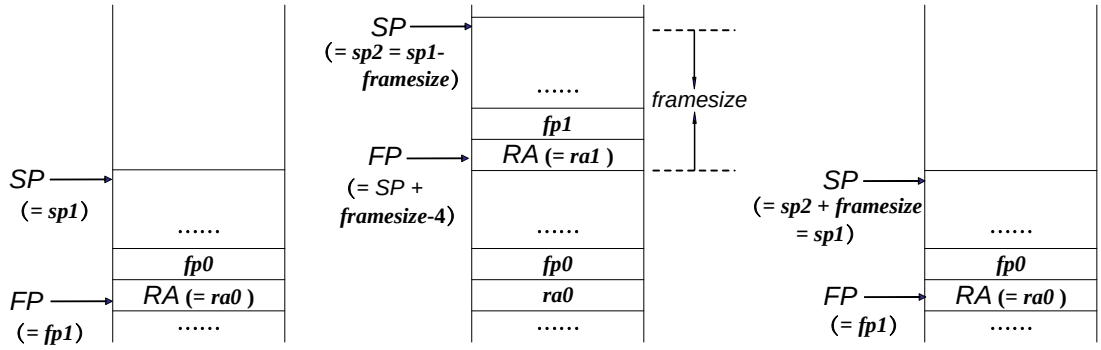


图 7　函数调用与返回所对应活动记录及栈/帧寄存器的变化

针对上面例子中的调用发起阶段代码（*prologue*）过程，以下是在函数体执行过后，可能的调用收尾阶段代码（*epilogue*）执行步骤：

（1）*RA* ← *framesize*-4 (*SP*) ;　　// 恢复返回地址

（2）*FP* ← *framesize*-8 (*SP*) ;　　// 恢复前一个栈帧的帧寄存器 *FP* 内容

（3）*SP* ← *SP* + *framesize* ;　　// 恢复上一栈帧的 *SP*，将当前活动记录弹出栈顶

从图 7（b）到图 7（c），可示意以上步骤（或指令）执行之后，活动记录及栈/帧寄存器的变化情况。

该例未考虑函数返回值的问题。返回值一般可以通过寄存器也可以通过栈帧中分配的存入单元传递，通常是前者。

当函数执行过程中 *SP* 会动态改变，而需要通过 *FP* 来寻址时，上例中至少收尾阶段代码（*epilogue*）的执行步骤需要进行调整，比如可修改为：

（1）*RA* ← 0 (*FP*) ;　　// 从当前栈帧首个单元读取返回地址

（2）*SP* ← (*FP*)+4 ;　　// 恢复上一栈帧的 *SP*，相当于将当前活动记录弹出栈顶

（3）*FP* ← -4 (*FP*) ;　　// 恢复前一个栈帧的帧寄存器 *FP* 内容

4.2 活动记录中的局部数据示例

为方便，本小节和下一小节的实例中，我们均假定过程/函数活动记录的结构如图 8 所示。每个栈帧所保存的控制信息包括静态连、动态连和返回地址，各占用一个数据单元。关于静态连与动态连的讨论，可参考后面的小节。假设输入参数全部存放于栈帧中，且占据最前面的单元。不考虑保存寄存器所需的栈帧空间。同时假设控制信息的 3 个单元之后先存放固定大小的局部数据，其后是编译时无法确定大小的动态数据（如动态数组或变长数组），然后是临时数据区。

局部数据对象局限于某个过程/函数，因此可以在栈中分配，当该过程/函数执行结束返回时，活动记录被撤销，这些数据对象随之不复存在，不再可以访问。动态数据因为在运行时才能确定其占用空间大小，因此多数情况下会在堆空间进行分配，而在栈中仅保存其指针/首地址和某些特征属性（如数组大小）。不过，经常也会选择将动态数组或记录/结构体对象分配在栈区。尽可能将局部数据对象在栈中分配有一个明显的好处，即无需进行垃圾回收，使得动态数据区的使用更加高效。

后续会讨论动态数组，栈指针 *SP* 会在运行时动态修改（即栈帧大小在运行时才能确定），因而活动记录内各单元的寻址会基于帧指针 *FP*，即 *FP*+*offset*（偏移量）作为寻址方式。此外，为方便，*offset* 以数据单元数为单位，而非字节数。实际应用时二者之间可以换算，比如，可将 1 个单元看作 1 个字长，对于 32 为字长来说，1 个单元相当于 4 个字节。

为在图中可以清晰表示以及方便讨论，不失一般性，在本节（第 4 节）剩余部分我们假定栈指针 *SP* 指向当前栈帧可以分配数据的下一个可用栈帧单元（如图 8 所示，仅用于示例）。在实际使用时，*SP* 通常指向当前栈帧最末单元（栈顶单元），参见 4.1 节和第 5 节。

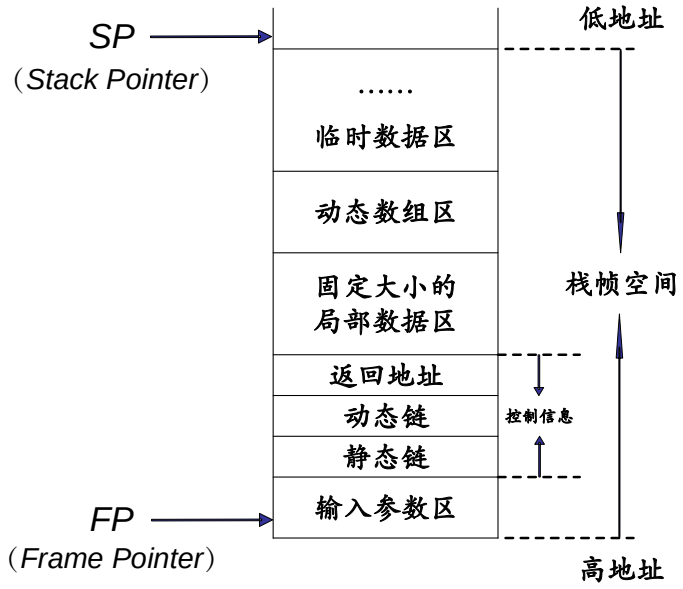


图 8　活动记录结构（示例用）

下面我们来看两个小例子，有关含有局部变量和数组的活动记录。设有 C 函数片段：

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

图 9 描述该函数的一个可能的初始活动记录，其中的信息依次包括：实际参数 a（int 类型的对象）占 1 个单元，3 个控制单元（静态连、动态连和返回地址）共占用 3 个数据单元，局部变量 b（float 类型的对象）占 2 个单元，以及 float 类型的数组变量 c 的各个分量（每个分量各占 2 个单元）。

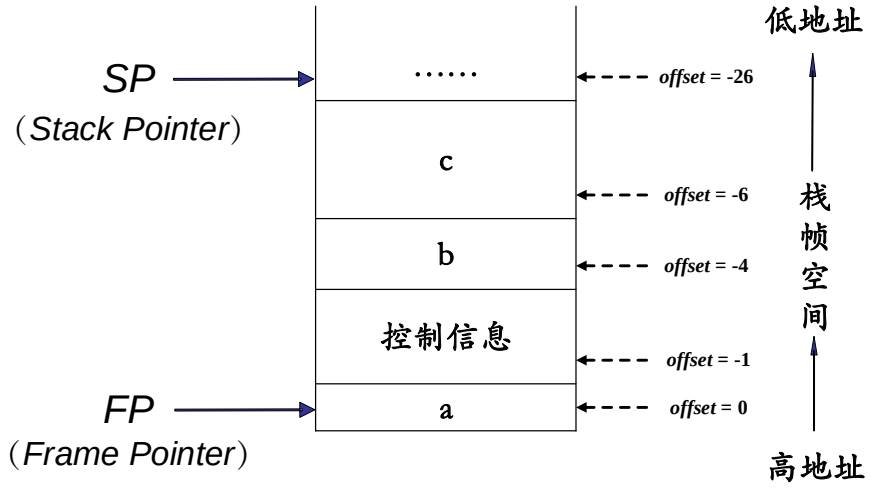


图 9　不含动态数据区的过程活动记录

如图 9 所示，数据对象 a 的 *offset* 为 0，3 个控制单元（静态连、动态连和返回地址）的 *offset* 分别为-1，-2 和-3，数据对象 b 的 *offset* 为-4，数组 c 的第 i（0≤ *i* ≤9）个元素的偏移量为-6-2*i*。设当前桢寄存器 *FP* 内容为 (*FP*)，则栈寄存器 *SP* 的内容 (*SP*)= (*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地址为 (*SP*)的单元开始分配栈帧空间。

下面我们看另外一个例子。设有如下 C 代码片段：

```
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 10 描述该函数的一个可能的初始活动记录，其中实际参数 a，3 个控制单元（静态连、动态连和返回地址），局部变量 b，以及数组变量 c 等的栈空间使用情况与图 9 所示相同。

对于动态数组 d，编译器并不能确定将需要多少存储空间，故先在初始活动记录中分配 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。这 2 个单元的 *offset* 分别为-26 和-27。之后，先将 *offset* 为-28 和-29 的 2 个单元分配给固定大小的局部数据对象 e。这样，便可以在编译期间确定动态数组 d 的首元素 *offset* 为-30。设当前桢寄存器 *FP* 内容为 (*FP*)，则 *offset* 为-27 的单元中存放的内容为 (*FP*)-30。当运行时 *N* 的值确定后，将存放于地址为 (*FP*)-26 的单元中，同时将栈寄存器 *SP* 的内容动态设置为 (*SP*)= (*FP*)-30-2*N*。

程序运行时，若需要进行动态检查（比如数组访问是否越界），则可通过访问地址为 (*FP*)-26 的单元获取数组上界 *N* 的取值。

除数组外，记录/结构体类型的局部数据对象的栈式分配，也可以采取相类似的方案，能在编译期间静态确定且随着函数返回不再使用的内容可以考虑在栈上分配。

对于更加复杂的数据对象，如类实例（对象）以及函数本身，因动态因素较多，一般会与堆空间的分配密切相关，参见第 7 和第 8 节。

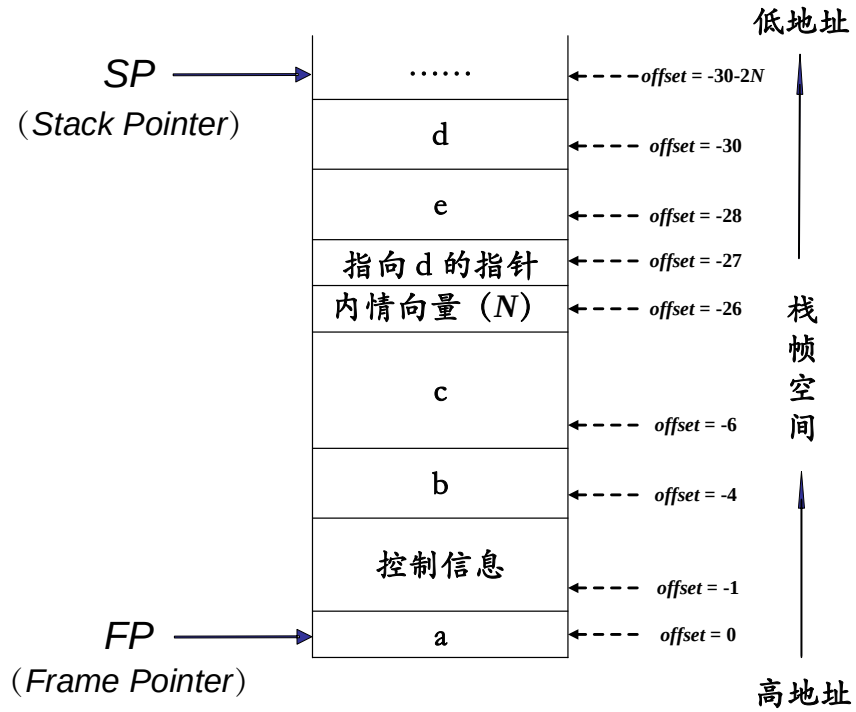


图 10　含动态数据区的过程活动记录

4.3 嵌套过程/函数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 11 所示是两个相似的类 Pascal 程序的过程定义示例（二者仅在第 6 行有差异），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。为方便讨论，不失一般性，图 11 中仅用到无参过程，并省略了所有的变量声明以及全部非过程调用的语句。

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，对于图 11 的两个程序，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。这种对于不在当前活动记录中的数据对象的访问，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，这里我们不考虑 C 之类的语言，而是以类 Pascal 语言为例，来讨论过程/函数层面的非局部量的访问。

```
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*/
```

(*a*)

```
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*/
```

(*b*)

图 11　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：。

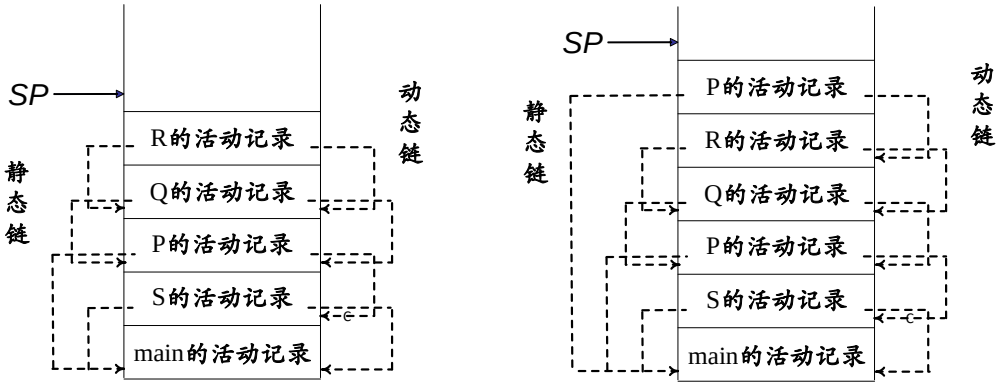
- 活动记录中包含静态链单元。参见 4.3.1 节。
- 采用 Display 表。参见 4.3.2 节。

4.3.1 静态链

采用静态链（*static link*），也称访问链（*access link*），是实现非局部量访问最常见的方法，即在所有活动记录都增加一个域，指向当前过程的直接外过程运行时的最新活动记录（的基址）。

与静态链对应的另一个概念是动态链（*dynamic link*），有时也称控制链（*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（恢复帧寄存器 *FP*），需要在被调用过程的活动记录中有这样一个动态链的域，指向该调用过程的活动记录（的基址）。

例如，对于图 11（a）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 12（a）所示（假设无其它调用语句）。又如，对于图 11（b）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 12（b）所示。



(*a*) (*b*)

图 12　静态链与动态链

例 1. 图 13 左边是某简单（类 Pascal）语言的一段程序。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 ':='。每一个过程的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。图 13 左边程序执行到过程 p 被第二次激活时，运行栈的当前状态如图 13 右半部分所示（栈寄存器指向单元 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

图 13　静态链与动态链（例）

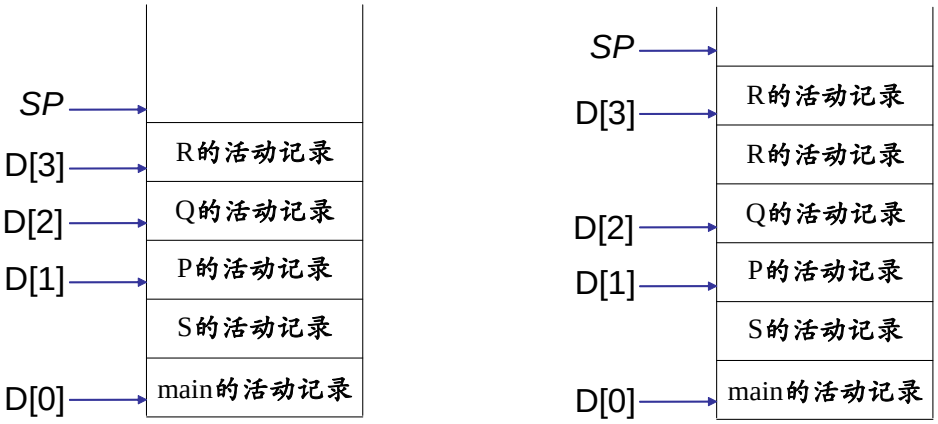
解：单元 18 和 19 中的内容分别为：0 和 13；单元 21 中的内容为：q 中 x 的内容；以及，

单元 22 和 23 中的内容分别为：0 和 18。

4.3.2 Display 表

采用静态链实现非局部量访问，需要沿静态链进行多次访存操作。另一种方法是采用全局 Display 表，可以提高非局部量访问的效率。后者的实现比起前者略复杂一些，且需要用到多个存储单元或多个寄存器。

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）。若当前激活的过程的层次为 *k*（最外的层次设为 0），则对应的 Display 表含有 *k*+1 个单元，依次存放着当前层，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 表的大小取决于具体实现（即嵌套层数有限定的上界）。Display 表对于寄存器的使用也取决于具体实现。



(*a*) (*b*)

图 14 Display 表

例如，对于图 11（a）中的程序，过程 R 被第一次激活后运行栈和 Display 表各表项 D[*i*]（0≤*i*≤3）的情况如图 14（a）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 14（b）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 表项 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 *k* 层，则需要保存 D[0]~D[*k*]。一个过程（处于第 *k* 层）被调用时，从调用过程的 Display 表中自下向上抄录 *k* 个 *FP* 值（*FP* 为帧寄存器），再加上本层的 *FP* 值。

例如，若采用这种方法，对于图 11 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 15 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 15 右边所示。

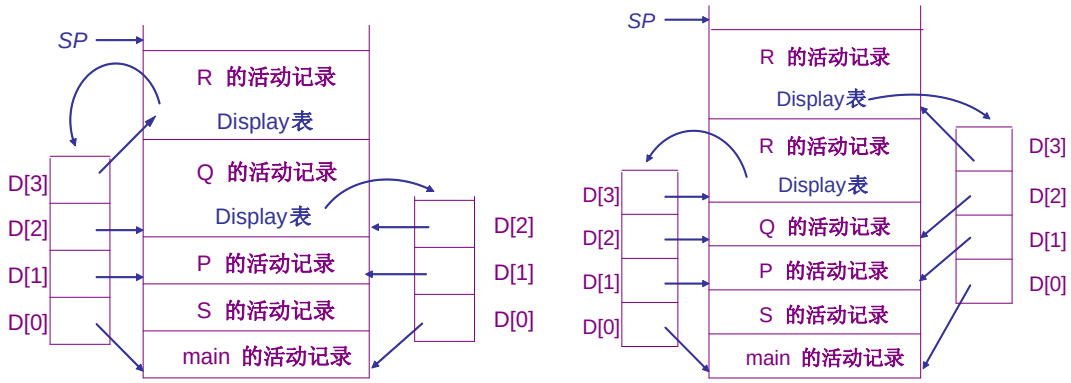


图 15 Display 表的维护（一）

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 *k* 层的过程被调用，则只需要在该过程的活动记录中保存 D[*k*]先前的值；如果 D[*k*]先前没有定义，那么我们用“_”代替。

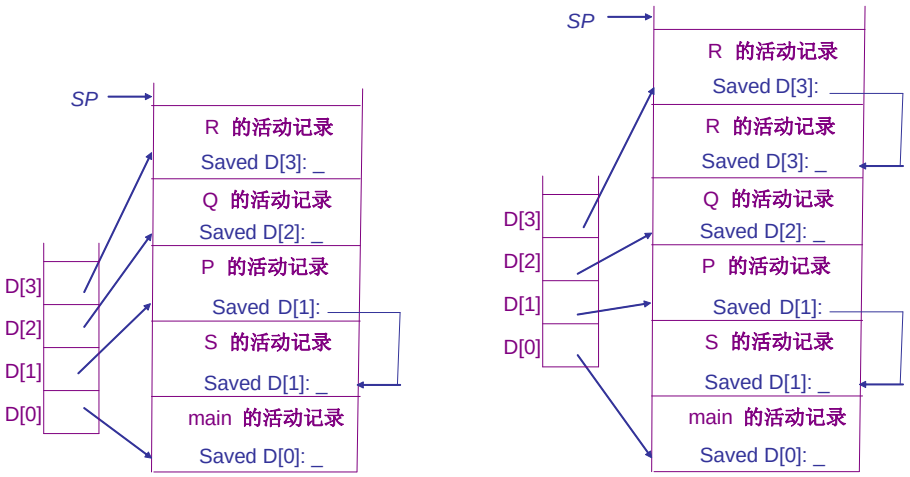


图 16 Display 表的维护（二）

例如，若采用第二种方法，对于图 11（a）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 16 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 16 右边所示。

为进一步解释后一种方法，我们来看图 11（b）的程序。这一程序将图 11（a）中的程序略加修改，在 R 的过程体中原来调用 R 之处改为了调用 P。若采用第二种 Display 表维护方法，对于图 11（b）的程序，在调用一次 S 以及执行头两轮 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 17 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。

calls	S	P	Q	R	P′	Q′	R′
D[3]	–	–	–	R	R	R	R′
D[2]	–	–	Q	Q	Q	Q′	Q′
D[1]	S	P	P	P	P′	P′	P′
D[0]	Main	Main	Main	Main	Main	Main	Main
saved	–	S	–	–	P	Q	R

图 17　Display 表的维护示例

例 2. 若在例 1 中采用 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）指出各活动记录中保存的 Display 表项的内容（即图 13 中所有 SL 位置的新内容）。

解：（1）Display 表的内容：D[0] = 0，D[1] = 22，以及 D[2] = 13。

（2）单元 0、单元 5 以及单元 13 中的内容为“_”（无定义）。单元 9、18 和 22 中的内容分别为：5、9 和 18。

4.4 嵌套语句块的非局部量访问

一些语言（如 C 语言）支持嵌套的语句块，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。可用的常规实现方法有两种：

- 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，一般很少采用。
- 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 代码片段：

```
int p（）
{
    int A;
    …                    // 无其它声明语句
    {
        int B, C;
        …                // 无其它声明语句
```

```
    }
    {
        int D, E, F;
         …                    // 无其它声明语句
        {
            int G;
            …   /*here*/
        }
    }
}
```

针对上述代码片断中的嵌套语句块的非局部量访问，可以采用过程级活动记录，如图18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。

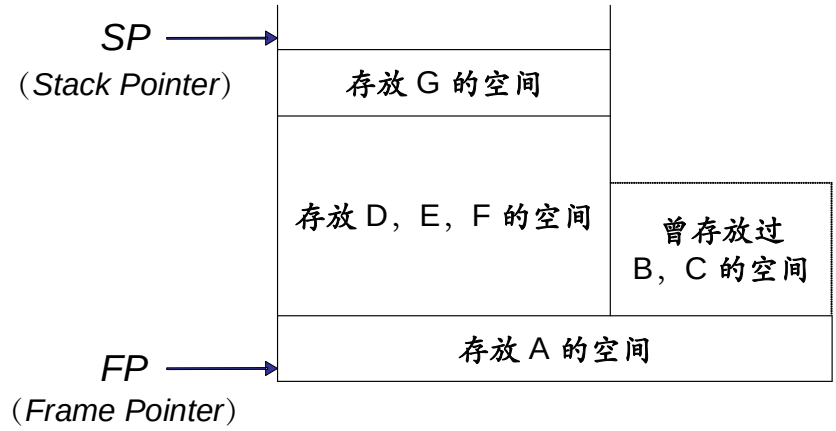


图 18　过程级活动记录中嵌套语句块的存储分配

4.5 动态作用域规则下的非局部量访问

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：。

多数情况下，常见的语言（比如 C，Pascal，和 Java 等）均采用所谓的静态作用域（*static scope*）规则，通过观察程序本身就可以确定一个声明的作用域，即程序某处所使用名字的声明之处是可以静态确定的。即使一些面向对象语言（如 C++）使用了像 public，private 和 protected 之类的关键字，但对于超类中成员名字的访问均提供了显式的控制机制。静态作用域有时也称为词法作用域（*lexical scope*）。

另一种情况是动态作用域（dynamiic scope）规则，即只有在程序执行时才能确定程序某处所使用名字的声明位置。对于常用语言，仅在特殊情况下采用动态作用域，比如 C 语言预处理程序中的宏展开（*macro expansion*）和面向对象程序中确定所要调用的方法。宏定义中使用的同一变量，在不同上下文中进行宏展开后可能对应着不同的变量声明。一个 superclass 声明的变量，在不同场合会代表不同的 subclass 对象，在运行时才能确定，同名方法的调用可能对应不同 class 中声明的方法。

为理解动态作用域规则和静态作用域规则的差异，我们来看一个简单的例子。设有Pascal 程序片断：

```
var    r: real
procedure    show;
      begin
            write(r:5:3)       // 以长度为 5 小数位数为 3 的格式显示实型量 r 的值
      end;
procedure    small;
      var      r: real;
      begin
            r:=0.125;    show
      end;
begin
      r:=0.25;
      show; small; writeln;
      show; small; writeln;
end.
```

若采用静态作用域规则，无论在哪个上下文中执行，过程 show 中的变量 r 总是指全局声明的 r，因此执行结果是：

0.250　　0.250

0.250　　0.250

若采用动态作用域规则，则在不同的上下文中执行，过程 show 中的变量 r 会被认为是最近的调用过程所声明的 r，因此执行结果是：

0.250　　0.125

0.250　　0.125

由此例可以看出，针对嵌套过程中非局部量的使用，若遵循静态作用域规则，则要沿着过程活动记录的静态链（或 display 表项）查找最近一个过程中所声明的同名变量；若遵循动态作用域规则，则要沿着过程活动记录的动态链查找最近一个过程中所声明的同名变量。

对于同一层次的非局部量，沿动态链查找比起延静态链查找需要的访存次数更多。可见在动态作用域规则下，程序的执行效率通常会比静态作用域规则下更低。

除性能问题外，动态作用域规则对静态类型检查工作有负面影响，同时也违背信息隐蔽原则，不利于程序的可靠性。仅有个别语言遵循动态作用域，若非特别指明，本讲所讨论的相关内容均假定是基于静态作用域规则。

例3. 若采取动态作用域规则，图13左边程序的执行效果与之前静态作用域规则下有何不同？

解：在第二次执行到语句L时，若是静态作用域规则，则a=1, b=2，因此会再次调用 P；若是动态作用域规则，则a=3, b=2，因此不会调用P。

5. 过程/函数调用与参数传递

5.1 典型的函数/过程调用序列

如前所述，每一次函数/过程调用会在运行栈上为被调用的函数/过程（*callee*）创建活动记录（*activation record*）或栈帧（*stack frame*），除非 *callee* 是叶子函数（*leaf function*）且函数内部计算无需用到栈空间。图 19 给出一种典型的栈帧结构（*stack frame layout*），其中栈空间是从高地址到低地址自上而下增长。

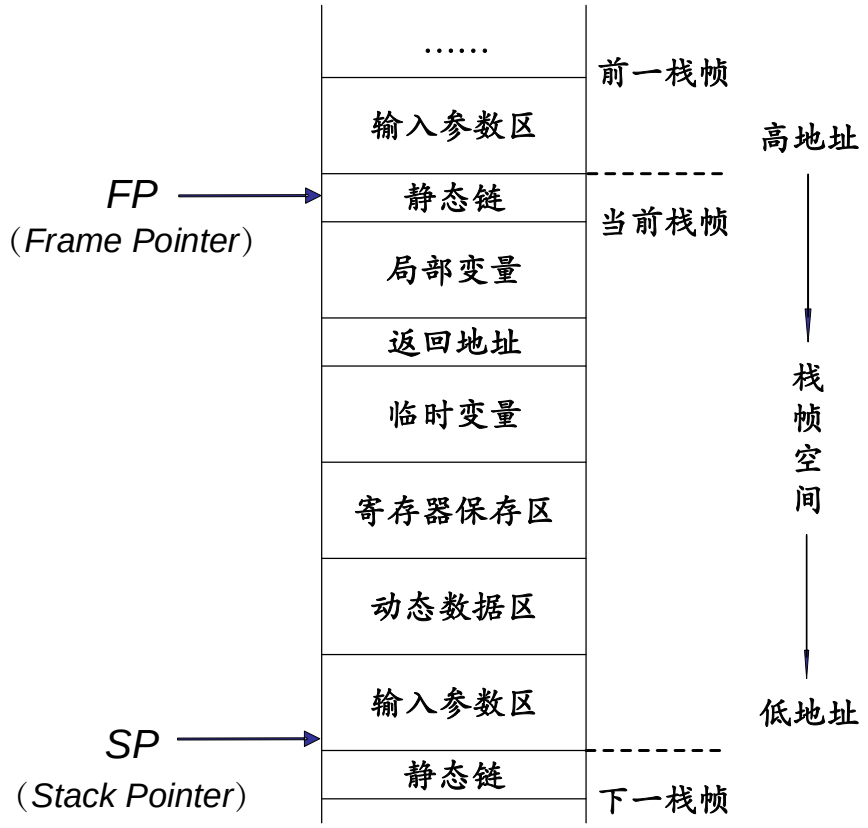


图 19　典型栈帧结构

如图 19 所示，实现过程/函数调用的控制信息中一般需包含“调用程序返回地址”信息（*callee* 是叶子函数时可以不包含），以保证当前过程/函数（*callee*）运行结束时返回到调用过程/函数（*caller*）继续执行。其它的控制信息可以包含某些必要的联系单元，如上一节介绍的动态链，以及静态链/display 表等内容。动态链可以通过将前一栈帧的帧寄存器（*FP*）保存至当前栈帧的寄存器保存区来实现。对于 Pascal、ML 等允许嵌套过程/函数定义的语言，需设置静态链/display 表相关的域，用于支持静态作用域规则下的非局部量访问；而对于类似 C 之类的语言，则栈帧中可以忽略静态链/display 表。

过程/函数调用时 *caller* 传递给 *callee* 的输入参数存放在栈帧的输入参数区，如图 19 所示，多处于 *caller* 栈帧且紧靠 *callee* 栈帧的单元。早期（如上世纪 70 年代），多数机器所约定的输入参数区位于 *callee* 栈帧中的单元，如第 4 节中的示例（参见图 6，图 8，图 9 和图 10 等）。由于第 4 节所讨论的核心内容与传参无关，且实例多使用无参过程，因而参数在栈

帧中的布局对所述内容影响不大。相比而言，现代计算机系统所约定的输入参数区通常位于 *caller* 栈帧中的单元，比起存放在 *callee* 栈帧中，将会更加节省对内存空间访问（即执行 *load* 和 *store* 指令）的次数。

根据现代计算机系统常用的调用约定（*calling convention*），一般会将固定数目的输入参数通过特殊寄存器进行传递，寄存器不够用时才会将多出来的输入参数置于内存栈帧的输入参数区。研究表明[5]，实际程序中少有函数使用超过 4 个参数，而使用超过 6 个参数的函数很难见到。因此，在现代计算机系统 ABI（*application binary interface*）的调用约定中，用于传参的寄存器数目 *k* 较为典型的取值为 *k*=4，*k*=6，等等。

此外，“返回值”也通常会将置于调用约定所指明的特殊寄存器中。若是需要将返回值直接存入到 *caller* 栈帧中的变量，一种常见方案是将该变量的存储单元置于栈帧的顶部，比如，在图 19 的 *caller* 栈帧输入参数区与 *callee* 栈帧的基址（*FP*）单元之间设置返回变量的存储单元。

局部变量和临时变量以及动态数据变量在栈帧中的存储分配可参见第 4 节所作的讨论。这里需要指出的是，这些变量也可能会被分配在寄存器中，这种情形下可以不占用也可以占用栈帧空间（后一种情况通常是为了腾出寄存器给其它变量使用）。

“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。假设函数 *f* 中的某个局部量占用寄存器 *r*，在调用函数 *g* 后在其内部计算中也用到寄存器 *r*，那么在 *g* 使用 *r* 之前就必须将其内容保存在栈帧单元中，当 *g* 执行完返回时在将其恢复，重置为栈帧单元中所保存的内容，从而 *f* 可以继续执行。寄存器 *r* 的保存和恢复需要 *f*（*caller*）或者 *g*（*callee*）来完成。若是前者，则 *r* 为调用函数/过程保存的寄存器（*caller-saved registers*）；若为后者，则 *r* 为被调用函数/过程保存的寄存器（*callee-saved registers*）。通常情况下，后者多用来保存生存期长的值（可能跨越函数/过程调用），而前者则更适合用于保存生存期短的值（一般不会跨越函数/过程调用）。

图 20 列举了两种典型体系结构 MIPS-32 和 RISC-V32 的整数寄存器使用约定（二者均拥有 32 个整数寄存器），参见相应的 ABI（*application binary interface*）规范[1, 6, 7]。MIPS 和 RISC-V 很大程度上共享在 Unix/Linux 中的 ABI 约定，二者有相似的栈帧结构。

图 20（a）是关于 MIPS-32 整数寄存器使用的约定，其中$fp / $s8（编号 30）为帧指针寄存器，$sp（编号 29）为栈指针寄存器，$gp（编号 28）为静态全局量访问指针寄存器，$ra（编号 31）为返回地址寄存器。寄存器$a0-$a3（编号 4-7）为 4 个输入参数寄存器，存放函数/过程调用的前 4 个参数，其余的参数需要存放在栈帧上进行参数传递。寄存器$v0 和$v1（编号 2 和 3）用作函数返回值（或返回值的指针）；如果只有一个返回值（如 C/C++ 默认最多一个返回值），则仅使用$v0；若是没有返回值，则$v0 和$v1 可用于普通的表达式计算。此外，$t0-$t9（编号 8-15，24-25）为 caller-saved 寄存器，无需跨调用保存；$s0-$s7（编号 16-23）为 callee-saved 寄存器，一般用于存放函数/过程调用的长生存期的值。

图 20（b）是关于 RISC-V32 整数寄存器使用的约定，其中 sp（编号 2），gp（编号 3）和 ra（编号 1）分别为栈指针寄存器，全局量访问指针寄存器和返回地址寄存器。寄存器 s0/fp（编号 8）可用作帧指针寄存器。寄存器 a0−a7（编号 10-17）为 8 个参数寄存器，其中头两个（a0 和 a1）可用作返回值寄存器。此外，t2-t6（编号 5-7，28-31）为 caller-saved 寄存器，s0-s11（编号 8-9，18-27）为 callee-saved 寄存器。

以上未涉及浮点寄存器的情况，有兴趣的读者可参阅相应的 ABI 规范[1, 6, 7]。

寄存器编号	ABI约定名称	用途
$0	$zero	常量（取值总为 0）
$1	$at	为汇编器保留
$2-$3	$v0-$v1	表达式求值/函数返回值
$4-$7	$a0-$a3	函数参数
$8-$15	$t0-$t7	临时寄存器
$16-$23	$s0-$s7	callee 保存的寄存器
$24-$25	$t8-$t9	临时寄存器
$26-$27	$k0-$k1	操作系统内核保留
$28	$gp	静态全局量访问指针
$29	$sp	栈（stack）指针
$30	$fp / $s8	帧（frame）指针 / $s8
$31	$ra	函数调用的返回地址

(*a*) MIPS-32**整数寄存器使用约定**

寄存器编号	ABI约定名称	用途
x0	zero	常量（取值总为 0）
x1	ra	函数调用的返回地址
x2	sp	栈（stack）指针
x3	gp	静态全局量访问指针
x4	tp	线程（thread）指针
x5-x7	t0-t2	临时寄存器
x8	s0 / fp	s0 / 帧（frame）指针
x9	s1	callee 保存的寄存器
x10-x11	a0-a1	函数参数/返回值
x12-x17	a2-a7	函数参数
x18-x27	s2-s11	callee 保存的寄存器
x28-x31	t3-t6	临时寄存器

(*b*) RV-32**整数寄存器使用约定**

图 20　典型体系结构的寄存器使用约定（示例）

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。

虽然不同机器的调用约定细节有些差异，但最基本的工作步骤可以统一要求。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，分别由调用函数/过程（*caller*）和被调用函数/过程（*callee*）分担执行，称为调用发起阶段（*prologue*）。第三段是返回代码序列，由被调用函数/过程（*callee*）执行，称为调用收尾阶段（*epilogue*）。

先来看调用发起阶段（*prologue*）需要完成的典型工作。

首先，在调用前，调用函数/过程（*caller*）完成：

（1）保存必要的（caller-saved）寄存器。

（2）参数传递。一些参数会传给寄存器；剩余的参数将压入栈中。

（3）如需要，保存其它控制信息（如静态链、display 表等）。

（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）跳转到目标地址（*callee* 代码入口）。

然后，在调用期间，被调用函数/过程（*callee*）完成：

（1）为局部量开辟栈帧空间（设置新 *SP* 寄存器的值）。

（2）保存旧栈帧基址（保存旧 *FP* 寄存器的值），即建立动态链信息。

（3）保存其它必要的（callee-saved）寄存器。

（4）建立新栈帧基址（设置新 FP）。

（5）转移控制，启动被调用函数/过程体的执行。

下面再看被调用过程（*callee*）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（*callee* 可方便访问的 *caller* 活动记录/栈帧位置，通常靠近 *callee* 栈帧）。

（2）恢复所有被调用过程保存的（*callee-saved*）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 *SP* 指针）。

（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，控制返回到调用函数/过程（*caller*），根据需要恢复保存的（*caller-saved*）寄存器，继续当前函数/过程（*caller*）的执行。

对于不同的调用约定（*calling convention*），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。例如，根据 Unix/Linux 中 MIPS-32 的 ABI 约定[7]，典型的调用发起阶段（*prologue*）代码的工作包括：

（1）调用函数/过程（*caller*）负责完成：

a. 头 4 个参数的内容存入寄存器（$a0-$a3），其余参数传递至 caller 栈帧；

b. 根据需要保存后续可能用到的（*caller-saved*）寄存器（$t0~$t9，$a0~$a3）；

c. 执行跳转指令 jal。

（2）被调用函数/过程（*callee*）负责完成：

d. 依据栈帧大小 *framesize* 分配 *callee* 的栈帧空间；参考指令；

```
subu    $sp, $sp, framesize     //  $sp ← $sp – framesize
```

e. 在栈帧中保存（*callee-saved*）寄存器（$s0-$s7, $fp, $ra）；参考指令：

```
sw      $ra, framesize-4 ($sp)
sw      $fp, framesize-8 ($sp)
….
```

f. 建立新栈帧基址；参考指令；

```
addiu   $fp, $sp, framesize-4   //  $fp ← $sp P + framesize–4
```

返回时，被调用过程（*callee*）在调用收尾阶段（*epilogue*）代码负责完成：

a. 若是函数，则将返回值（或其指针）存入寄存器$v0；

b. 恢复保存在栈帧的（*callee-saved*）寄存器；参考指令：

```
lw      $ra, framesize-4 ($sp)
lw      $fp, framesize-8 ($sp)
....
```

c. 将当前活动记录/栈帧弹出栈顶；参考指令：

```
addiu       $sp, $sp, framesize ;     // $sp ← $sp + framesize
```

d. 跳转至 $ra 所指地址继续执行；参考指令：

```
jr      $ra
```

最后，控制返回到调用函数/过程（*caller*），根据需要恢复保存的（*caller-saved*）寄存器（$t0~$t9, $a0~$a3），继续当前函数/过程（*caller*）的执行。

5.2 参数传递方式

在实现函数/过程调用时，参数的传递方式也是很重要的环节。最常见的参数传递方式如传值（*pass by value*，*call-by-value*），传引用（*pass by reference*，*call-by-reference*），以及传闭包（*pass by closure*，*call-by-closure*）等。另外，不同语言特性及其实现的需求还催生出如传名字（*pass by name*，*call-by-name*）以及其它多种参数传递方式。

5.2.1 传值

这一参数传递方式，是在调用时计算实参表达式的值，然后拷贝给形参变量，实际传递的是实际参数的右值。一个表达式的右值（*r-value*）代表该表达式的取值。

大多数语言均支持传值的参数传递方式，如 Pascal，C，C++，Ada，等等。值得注意的是，传值调用是 C 语言中唯一的参数传递方式。

考虑下列 Pascal 程序片段：

```
procedure swap1 (x, y : integer);
    var temp : integer;
    begin
        temp := x;
        x := y;
        y := temp
    end;
```

传值是 Pascal 语言默认的参数传递方式，在传参后形参变量 x 和 y 各自均有独立于实参表达式右值的副本。在调用过程 swap1 (a, b)时，swap1 过程内部执行后，的确将 a 和 b 副本的取值内容进行了交换，然而所交换的并非调用前 a 和 b 的内容。因此，调用过程 swap1 (a, b)之后将不会影响 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
```

以下是与上述 Pascal 程序片段作用相同的 C 程序片段：

```
void swap2 (int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

因传值调用是 C 语言中唯一的参数传递方式，所以函数调用 swap2(a, b)同样不会影响 a 和 b 的值。

再考虑如下 C 程序片段：

```
void swap3 (int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

若是采用函数调用 swap3(&a, &b)，则可以使两个整型变量 a 和 b 的取值互换。然而，这一调用的参数传递方式仍然是传值调用，只不过传递的是 a 和 b 的地址值，即指向 a 和 b 的指针的值。

值传递的一个问题是对执行效率的影响。若是很大的数据值，值传递时生成副本的代价很高，因此在 C 语言中一般都是建议通过传指针值（数据值的首地址）的方式来实现。但这又带来另一个问题，在函数体内对这一数据值的访问和修改可能用到大量的指针操作，非常不方便且容易出错。一种解决方案是采用另一种参数传递方式—传引用，参见下一小节。

5.2.2 传引用

传引用（*pass by reference*，*call-by-reference*）又称传地址（*pass by address*，*call-by-address*）。这一参数传递方式，无需拷贝实参表达式的值传给形参变量，所传递的是引用。实际上，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地址。传引用相当于将实参表达式的左值拷贝给相应的形参变量，这样，通过该形参就可以间接引用到（或指向）实参表达式的右值，相当于是实参表达式的一个别名。

相比 C 语言，C++中增加了引用特性。例如，C++中函数的形参冠以&开头则表示是通过传引用方式进行参数传递。考虑如下 C++程序片段：

```
void swap4 (int &x, int &y) {
    int temp;
    temp = x;
```

```
    x = y;
    y = temp;
}
```

这里将函数的形参指定为引用的形式，这样在调用函数时就会将实参和形参绑定在一起，让它们都指代同一份数据。如此一来，如果在函数体中修改了形参的数据，那么实参的数据也会被修改。对于此例，如果执行函数调用 swap4 (a, b)，则可以使两个整型变量 a 和 b 的取值互换。

Pascal 也支持传引用（或传地址）的参数传递方式。例如，下列 Pascal 程序段：

```
procedure swap5 (var x, y: integer);
    var temp: integer;
      begin
        temp := x;
        x := y;
        y := temp
      end;
```

其中，函数声明中的保留字 var 告诉我们将采用传引用（或传地址）的参数传递方式。此时，调用过程 swap5 (a, b)将会交换 a 和 b 的值。

在实现传引用（或传地址）调用时，将把实际参数的地址传递给相应的形参。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，并存入一个相应类型的存储结构，然后将传此存储结构的基地址放于对应形参的存储单元。被调用过程执行时，就像使用局部变量一样使用这些参数，但使用的是左值。虽然最终会将过程体内对这些形参变量的访问/修改转换成类似于指针操作的效果，但这是由编译器自动完成的，无需程序员手工编写。例如，执行函数调用 swap4 (a, b)会自动转换成类似函数调用 swap3 (&a, &b)的执行效果。可见，使用传引用调用，可以避免程序员在函数体内使用繁琐且易出错的指针操作，而是直接使用与实参表达式类型相同的形参变量即可。

另一方面，虽然利用传引用（或传地址）调用，可以获得“在函数内部影响函数外部数据”的效果，然而，这也是这种参数传递方式的一个弱点，因为调用者（*caller*）可能并不希望实际参数的值以这种方式加以篡改。C++中，若在形参声明时借用 const 修饰符，则可消除这种副作用。

5.2.3 传闭包

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。例如，下列 C 程序片段：

```
#include <stdio.h>
int a = 2;
int b = 1;
int add (int a) {
    return a + b;
}
int sub (int b) {
```

```
    return b - a;
}
int bop (int (*f) (int), int c)
    return f(c);
}
void main ( ) {
    printf("%d ", bop(sub, bop(add, 5)));
}
```

函数 bop 声明中有函数形参 f。对于如 C 这类语言，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/栈帧没有什么差异，局部数据在栈帧中访问，而非局部数据只有全局量，均在静态区访问。例如，bop(sub, bop(add, 5))中的函数实参 add 和 sub，在 bop 过程体被激活时，会分别访问非局部量 b 和 a，二者均保存在全局静态区。

然而，对于包含嵌套过程/函数声明的语言，情况要复杂一些。以下是源自[4]的一个类 ML 程序片段：

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

在该程序片段中，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，对于上面的例子程序，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 21 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。这里，假设调用实参存放于被调用函数（*callee*）的活动记录中。

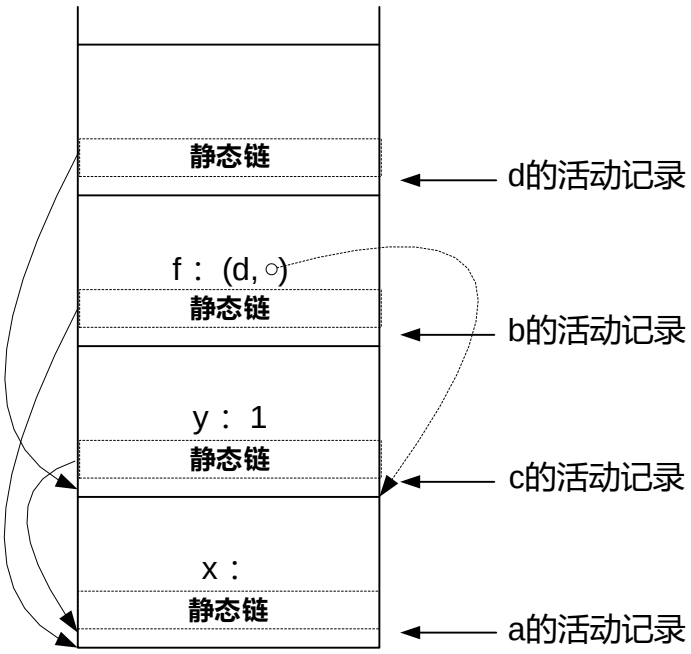


图 21　带静态链的函数实参

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。

这里，在 c 中调用 b 时给形参 f 所传参数包含两个部分：d 的目标代码入口指针，以及用于访问非局部量的静态链。这种同时包含代码指针以及针对非局部量访问方式的记录结构，称之为闭包（*closure*）。将闭包作为实参传递给函数形参的参数传递方式，称之为传闭包（*pass by closure*，*call-by-closure*）

闭包的结构不一定要基于栈帧上的静态链，而且在一些情况下也不适用，参见 8.3 节的讨论。

5.2.4 其它方式（选讲）

在各种不同的程序设计语言中，还存在有多种风格的参数传递方式，下面列举常见的几种：

- 传结果（*pass by result*）。对采用这种方式的参数，在调用结束时，会将该参数的最后计算结果拷贝出来。
- 传值-结果（*pass by value-result*）。传值方式是将实参的值拷贝给形参，传结果方式是将形参的最终结果拷贝出来传给是实参。传值-结果方式则是双向传递，在调用开始时将实参的值拷贝给形参，而在调用结束时会将形参的计算结果拷贝给实参。

  Ada 中参数有 3 类，分别为 in，out，和 in out。in 参数从调用方传递给被调用方，只能读不能写。out 参数，在能被访问或者调用返回之前需要先赋好值。in out 参数在被赋值前是可以访问的，在返回前还可以被多次赋值。in，out，和 in out 参数分别对应传值，传结果，和传值-结果，默认是通过不同方向或者双向的传值（*pass*

by value）方式来实现，但对于数组与记录/结构体等非基础类型的参数，则可以选择采用单向或双向传引用（*pass by reference*）的方式来实现。。

- 传名字（*pass by name，call-by- name*）。这种方式最早是在 Algol 语言中引入的，是对参数名字的每一处出现用实参文本进行替换，替换时不会执行，而是会在替换后的上下文中一起执行。其实现原理如同 C/C++的宏替换机制，预处理程序在扫描 #define 时只进行宏替换/宏展开，而不会直接执行替换后的内容，后者将在每一处替换所在上下文中被进一步编译执行。这种机制常用于实现惰性求值（*lazy evaluation*），也称为按需调用（*call-by-need*），参见 8.4 的讨论。

- 只读属性参数（*read-only parameters*）。若参数拥有只读（*read-only*）属性，如 C++ 的 const 参数，Ada 的 in 参数，当使用传引用（*pass by reference*）方式时，仅仅是为了提高效率，而实参的内容则是不允许修改的。

- 含默认值参数（*default parameters*）。这种方式可以为参数提供一个默认的值，当调用时未提供对应的实参值时，可使用该默认值。

- 位置无关参数（*position-independent parameters*）。这种方式下，可以仅通过参数名字进行实形替换，而无需考虑参数的位置（次序），或者说不用担心参数的次序，只要名字对的上就行。Ada 和 Lisp 等语言提供这种机制。

6. 垃圾回收（选讲）

6.1 概述

如果能知道堆中有代表对象/记录的存储单元在动态语义下不是活跃的，即在程序的后续计算中不会用到与其关联的变量，则把这些存储单元作为垃圾（废弃单元）清除掉（或者回收）是没有任何问题的，不会影响到程序的正确执行。然而，一个变量是否动态活跃的（*dynamically live*）是不可判定的问题（容易将图灵机停机问题归约至该问题[5]），即没有通用算法可以静态地判定任何对象/记录在动态语义下的活跃与否，因此无法实现执行这种理想意义下的垃圾回收算法。

一般情况下，不能从任何当前活跃的程序变量的引用链可达的堆中对象/记录称为垃圾（*garbage*）。当前活跃的（*active*）程序变量包含所有静态或全局的变量、以及当前处于运行栈中的变量。从当前活跃的程序变量的引用链可到达的堆中对象/记录称为可达的（*reachable*）对象/记录。

被垃圾所占用的存储块（*chunks of storage*）可以被重新收回（*reclamation*）并分配给新的对象/记录，这一过程称为垃圾回收（*garbage collection*）。比起上述理想意义下的垃圾回收，只能说这只是一种近似的保守的垃圾回收。编译器应当保证任何活跃的对象/记录都是可达的，但并不能保证可达的对象/记录一定是活跃的。然而，可以要求编译器尽可能使可达的不活跃对象/记录的数目最小化。

实际的垃圾回收过程最终会交由运行时系统自动完成，编译器生成的代码可直接或间接调用运行时系统的堆存储分配与管理功能，同时根据不同的垃圾回收策略通过实现特殊约定与垃圾回收程序进行交互。为方便性和灵活性，这些工作的完成，通常会借助一种专门用于维护引用链可达性即时变化的运行时用户程序，称为更变程序（*mutator*）。更变程序的设计

与具体的垃圾回收策略密切相关。编译器生成的代码直接通过更变程序来实现堆存储分配以及自动垃圾回收的功能。更变程序负责维护和监视当前堆中对象/记录的引用信息，并根据需要触发垃圾回收过程。

垃圾回收算法的核心是找到不可达的对象/记录，从而可以重新收回相应的存储块。垃圾回收最早是在早期函数式语言（如 Lisp）中实现的，至今已有超过 50 年的历史。垃圾回收机制曾被认为是函数式和逻辑式语言的标配，但后来在面向对象语言（如 Smalltalk、Java）也得到广泛应用。如今，垃圾回收机制已经被人们认为是一种语言特性。

为找到不可达的对象/记录，可以采取两种不同的策略：一种是观察当前可达的对象/记录是否转变为不可达的；另一种是周期性查询所有可达的对象/记录，从而断定其他对象/记录是不可达的。采取第一种策略的方法，比如引用计数（*reference counting*）法，维护一个对象/记录的引用计数值，当计数值变为 0 时，则该对象/记录就变为了不可达的。采用第二种策略的垃圾回收算法，需要借助引用关系的传递来跟踪（*tracing*）所有可达的对象/记录，统称其为基于跟踪的垃圾回收（*trace-based garbage collection*）方法。

基于跟踪的垃圾回收方法，最基本的有标记-清除（*mark-and-sweep*）法和拷贝回收（*copying collection*）法。为避免长时间停顿，从而影响到用户程序的实时响应能力，在此基础上人们提出了短停顿（*short-pause*）的垃圾回收方法，如每次仅回收部分废弃存储空间的分代回收（*generational collection*），以及每次仅执行有限时间的增量回收（*incremental collection*）。

一般来说，类型安全的（*type-safe*）语言或强类型的（*strongly typed*）语言才适合配备垃圾回收功能。类型安全的语言能够保证任何给定数据单元的类型是确定的，从而能够精确获知该数据单元的大小及其所内部包含的对其他数据单元的引用，并保障该数据单元可以被正确引用，即用于访问该数据的引用恰好指向所分配给它的存储块的起始地址，而不会指向其某个内部的位置。有些类型安全的语言在编译期间就可以确定数据单元的类型，如 ML、Ada 等。有些类型安全的语言在运行期间才能确定任意给定数据单元的类型，如 Java、Scheme、Smalltalk 等。本讲介绍经典的垃圾回收算法，均默认是针对类型安全的语言。

像 C 和 C++之类的弱类型的语言，可以对存储地址进行随意修改以及允许类型的强制转换，导致可能产生目标不确定的指针，所指向的内容无法对应到具体数据单元的存储块，因此难以准确检测出数据单元的可到达性与不可到达性。然而，由于 C 和 C++的使用非常广泛，人们也提出一些针对性的保守的垃圾回收方略，对此本课不予以讨论。

6.2 引用计数

引用计数（*reference counting*）方法的技术原理是，对堆中所维护的每一个数据单元（对象），都设置一个对该数据单元（对象）的引用次数（*number of references*）计数器，当计数器数值为 0，则该数据单元（对象）可以被回收。

初始时，假定堆是一条由连续的结点（*nodes*）构成的链表 *free_list*，每个节点有一个附加的整数域用作引用次数计数器，置该计数器的初值为 0。随着程序的运行，不断从 *free_list* 获取结点，这些结点相互之间通过指针链接形成引用链，同时，引用次数计数器的值（简称引用计数值）也会在程序运行过程中不断变化。

在程序运行中，若遇到会导致堆中结点的引用计数值发生变化的操作，更变程序会负责维护引用计数值。当新创建一个数据单元（对象）时，则调用分配程序（*allocator*）从 *free_list*

获得一个可用结点，并将其引用计数值初始化为 0。每当遇到使当前指针或引用不再关联到某个结点的操作时，则相应减少该结点的引用计数值。每当遇到有新的指针或引用关联到某个结点的操作时，则相应增加该结点的引用计数值。一旦发现当前结点的引用计数值变为 0，则调用回收程序（*collector*）将该结点返还给 *free_list*，然后修改该结点内部所关联到的所有其它结点的引用计数值，对引用计数值有变化的结点从判断该结点的引用计数值是否变为 0 开始再递归执行最后这一步骤。

图 22 是使用引用计数方法的一个示例，图中给出某个时刻堆中结点及相关链表的结构，其中每个结点的引用次数的当前计数由虚线框内的数值给出。此刻，该程序在堆中分配有 6 个结点，其余结点均包含在 *free_list* 中，作为空闲结点后续可以被分配使用。假设 *p* 和 *q* 是当前活跃的程序变量（静态/全局或正处于运行栈中的变量），则这 6 个结点都是可达的，它们可以通过 *p* 和 *q* 的引用链直接或间接到达。

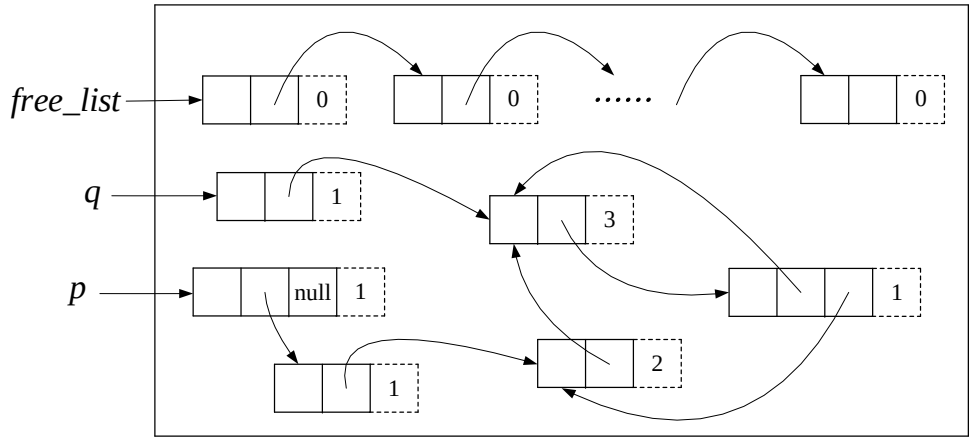


图 22　引用计数方法的一个示例（1）

引发堆中结点的引用计数值发生变化的操作有多种，主要包括：

（1）*x* 原来关联到某个数据单元（对象）*X*，在执行赋值 *x = y* 后，就切断了的 *x* 到 *X* 的联系，会导致 *X* 以及 *y* 所关联到的数据单元（对象）结点的引用计数值发生变化。

例如，在图 22 所描述的当前堆状态下，执行赋值操作 *q = p* 后，将导致 *q* 原来所关联到的结点的引用计数值由 1 变为 0，而 *p* 所关联到的结点的引用计数值将由 1 变为 2。由于 *q* 原来所关联到的结点的引用计数值变为 0，因而将该结点内部所关联到的引用计数值由 3 变为 2，与此同时，启动回收程序将该结点返还给 *free_list*。经过这些变化，堆中结点及引用链的当前状态如图 23 所示。

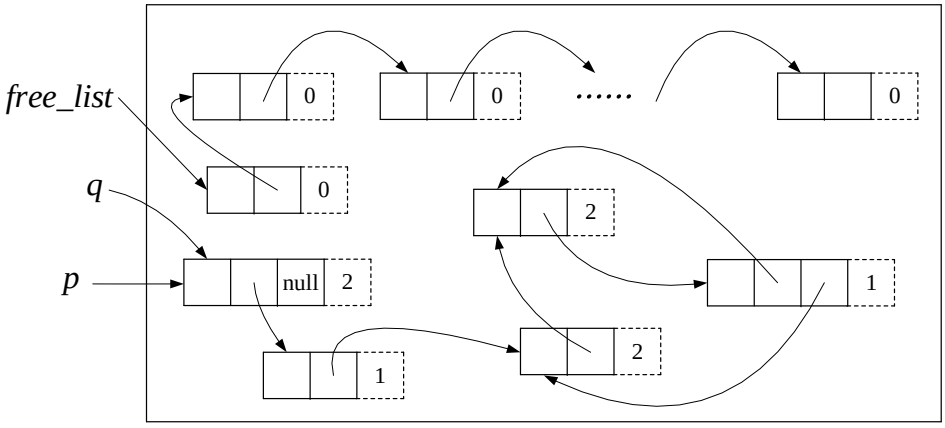


图 23　引用计数方法的一个示例（2）

（2）*x* 原来关联到某个数据单元（对象）*X*，在创建一个新的数据单元（对象）并使 *x* 关联到该数据单元（对象）之后，就切断了的 *x* 到 *X* 的联系，在新增数据单元（对象）结点的同时，导致 *X* 结点的引用计数值发生变化。

（3）*x* 原来关联到某个数据单元（对象）*X*，在执行赋值 *x = null* 后，就切断了的 *x* 到 *X* 的联系，会导致 *X* 结点的引用计数值发生变化。

例如，在图 23 所描述的堆状态下，执行赋值操作 *p.left.next = null* 后（这里假定所涉及的两个结点的指针域分别为 *left* 和 *next*），将导致所关联到的结点的引用计数值由 2 变为 1，参见图 24 中最下方的结点。

（4）函数调用时的参数传递相当于赋值，如果实在参数中有在堆中分配的数据单元（对象），则函数调用时实在参数赋值给形式参数，就使形式参数联系到堆中的这些实在参数单元（对象）。这样，当退出函数调用时就需要切断相关数据单元（对象）的联系，导致相关结点的引用计数值发生变化。

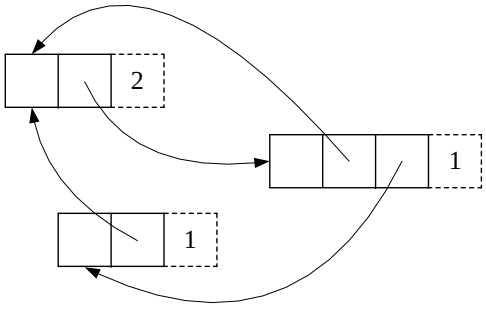


图 24　引用计数方法可能导致无法到达的循环引用链

引用计数方法的优点之一是看起来较为直观，易于理解。另外还有一个更主要的优点，因引发堆中结点的引用计数值发生变化的操作是在程序执行过程动态发生的，触发垃圾回收的相关过程随机分布在程序执行期间，这样程序的运行并不会因此出现很长时间的停顿。

该方法的主要缺点包括：（1）无法收回相互之间有循环引用的不可达的结点链，如图 24 所描述的前面例子中的 3 个循环引用的结点；（2）堆中每个结点需要附加一个用于记录

引用次数的计数器；以及（3）需编译生成维护引用次数计数器的辅助代码，实现代价较高。

6.3 标记-清除

不像引用计数方法是在程序执行过程动态更变引用计数值而随时可能触发垃圾回收过程，标记-清除（*mark-and-sweep*）回收算法仅当堆空间满了的时候（*free_list* 为空）才被调用。相比前者，后者的优点是对程序执行过程的干扰很小，也不存在不能回收的循环引用链。

基本的标记-清除回收算法被触发后，会执行两个阶段任务：（1）标记（*mark*）；（2）清除（*sweep*）。图 25~27 描述了使用标记-清除方法的一个示例。

图 25 示意某个时刻堆中结点及相关链表的结构，其中每个结点附加有一个“标记位”（虚线框内）并初始化为 0。此刻，该程序在堆中至少分配有 6 个结点。假设 *p* 和 *q* 是当前活跃的程序变量（静态/全局或正处于运行栈中的变量），则这 6 个结点都是可达的，它们可以通过 *p* 和 *q* 的引用链直接或间接到达。包含在 *free_list* 中的结点为空闲结点，后续可以被分配使用。

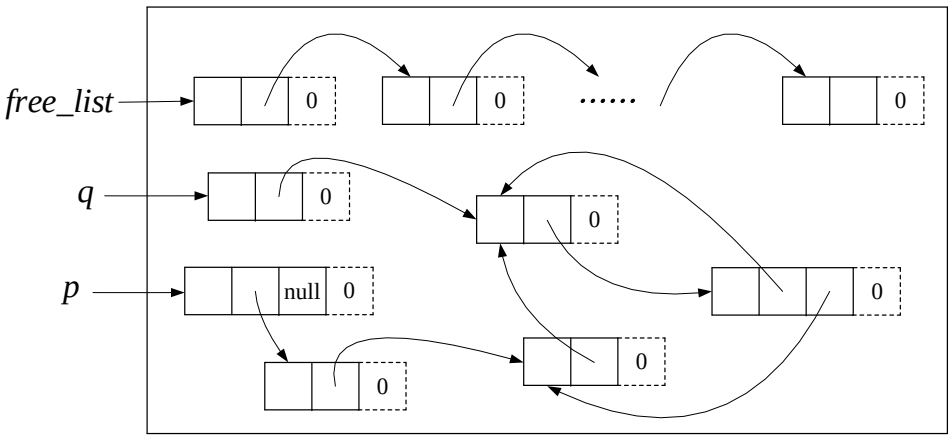


图 25　标记-清除方法的示例（1）：触发前

在程序运行中，每当申请分配新结点时，比如执行 r = new node()为变量 r 申请一个新的堆结点，则更变程序可能执行以下步骤：

（1）若堆空间未满（*free_list* != null），则调用堆存储分配算法，称之为 alloc()，从 *free_list* 获取可用结点分别配给 r，正常返回；若堆空间已满（*free_list* == null）或无合适的空闲结点给 r 分配，则转下一步；

（2）激活标记-清除回收算法 mark_sweep()；

（3）若执行 mark_sweep()后堆空间未满（*free_list* != null），则再次调用 alloc()为 r 分配可用的堆结点；否则，转（5）；

（4）若分配成功，则正常返回；否则，转（5）；

（5）返回“执行 r = new node()堆空间分配异常”相关信息。

若上述过程未触发堆空间分配异常，则从执行 r = new node()正常返回，程序将继续后面语句的执行。

图 26 刻画了因堆空间已满（*free_list* 为 null）而触发标记-清除回收算法、并经过“标记”阶段任务之后的情形。由图可知，在触发回收算法之前，可能因执行赋值操作 *q* = *p* 以及 *p.left.next* = *null* 后（这里同 6.2 节，假定所涉及的两个结点的指针域分别为 *left* 和 *next*）导致原来可通过 *q* 能联系到的结点“失联”。在“标记”过程结束后，由当前活跃的程序变量（静态/全局或正处于运行栈中的变量，如 *p* 和 *q*）的引用链可以直接或间接到达的全部结点，其“标记位”都会变为“1”。“标记”的过程可借助于各种有向图遍历算法（如深度优先遍历），这里不再进一步具体到细节。

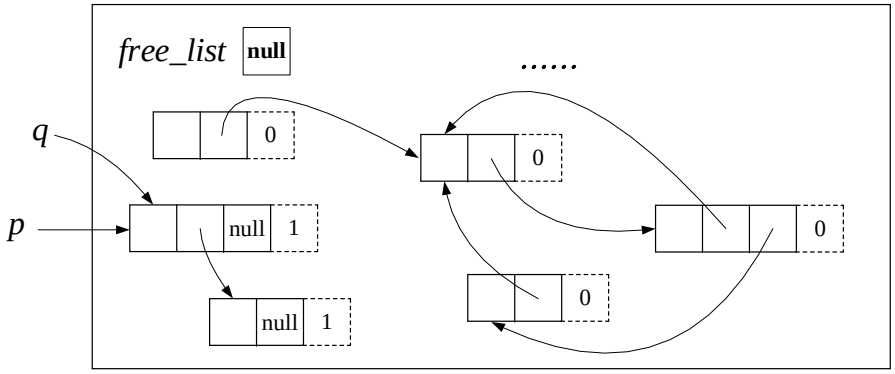


图 26　标记-清除方法的示例（2）：标记后

“标记”阶段之后进入第二个阶段“清除”，这一个过程会遍历整个堆中的结点，无论“标记位”是“0”还是“1”。当遇到“标记位”为“0”的结点，则将其放回到 *free_list* 中。当到“标记位”为“1”的结点，则将其“标记位”恢复为“0”。如图 27，原来在图 26 中所示的一些“失联”结点均被退还回 *free_list*。原来那些通过当前活跃程序变量（如 *p* 和 *q*）的引用链可以直接或间接到达的结点，其“标记位”均被重置为“0”。

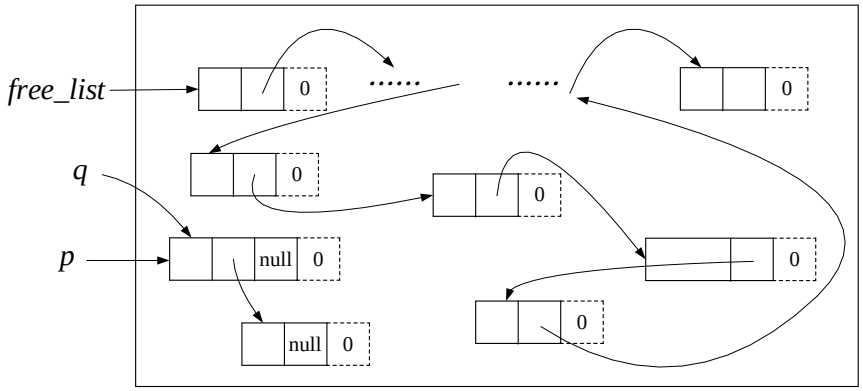


图 27　标记-清除方法的示例（3）：清除后

对标记-清除回收算法的一个改进，是在“标记”阶段顺带对可达结点进行重定位，将其集中压缩至堆中某一端地址连续的空间，实现碎片整理的效果。此类算法称为标记-压缩（*mark-and-compact*）回收算法，一般包含下列主要步骤：

（1）“标记”。如同标记-清楚方法的“标记”阶段，对当前活跃程序变量的引用链可以直接或间接到达的全部结点进行标记；

（2）“重定位”。为全部可达结点计算新地址，比如将它们均重定位到连续的栈空间低

端地址。

（3）“转储”。将每个可达结点拷贝（转存）至其新的地址空间，同时更新节点内部所有引用至新的地址。由于已经分析过各个可达结点内部的引用情况，因而这个过程并不需要额外负担太多的工作。

标记-清除（包括标记-压缩）回收算法的主要问题，是在回收算法触发后到执行结束，再回到程序执行，其间的停顿时间较长，可能会严重干扰到程序的正常执行。下面介绍的拷贝回收方法可减少停顿时间。

6.4 拷贝回收

相比标记-清除（包括标记-压缩）回收，拷贝回收（*copying collection*）方法是时间和空间上进行折中的一种方案。在实现拷贝回收方法时，会将对空间划分为大小相等的两块，分别称之为 *from_space* 和 *to_space*。这里，无需任何“标记”位，也不需要维护一个 *free_list*。不同的是，需要维护一个 *from_space* 新一轮可分配空间起始位置的指针 *free*。*free* 指针在初始化时置初值，以及每一次拷贝回收算法执行之后重新设置。任何时刻，可用于堆存储分配的范围是 *from_space* 中从 *free* 开始的空间（下面假设其上界为 *from_space_top*）。

随后的讨论中，假定存储分配算法为 alloc()。本课程不讨论具体的存储分配算法，因此这里不限定 alloc()如何设计。实际上，考虑到拷贝回收方法的特性，采用连续分配的方案即可，即每次分配都是从前到后查找从 *free* 开始到 *from_space_top* 结束的空白区域。还可以在每次分配后更新 *free* 指针，这样，下一次分配可以直接从 *free* 开始。

下面先简单介绍使用拷贝回收方法的前后过程以及更变程序的基本功能。

程序执行时，所有活跃结点均在 *from_space* 这一半空间中分配，而另一半空间 *to_space* 没有使用。堆空间会通过下列步骤进行初始化：

（1）令 *from_space* = *l*，为可用堆空间的下界；

（2）置 *from_space_top* = *l* + (*u* - *l*) / 2，这里 *u* 为可用堆空间的上界；

（3）令 *to_space* = *from_space_top* + 1；

（4）置 *free* = *from_space*。

在程序运行中，每当申请分配新结点时，比如执行 r = new node()为变量 r 申请一个新的堆结点，则更变程序将会执行下列步骤：

（1）调用存储分配算法 alloc()，从可分配的堆空间（*free* 开始到 *from_space_top* 结束）获取适合的可用堆空间存储块分配给 r；若分配成功，则正常返回；否则，转下一步；

（2）激活拷贝回收算法，如 Cheney 算法（随后介绍），将 *from_space* 空间中分配的所有可达结点拷贝至 *to_space* 空间相邻的存储块依次排放，且令指针 *free* 指向紧其后的空闲空间起始位置；

（3）对调 *from_space* 与 *to_space* 的角色，即令

from_space, *to_space* = *to_space*, *from_space*

以及 *from_space_top* = *from_space* + (*u* - *l*) / 2（*from_space* 为调换前的 *to_space*）；

（4）以新的参数 *free* 和 *from_space_top*，再次调用 alloc()为 r 获取可用的存储块；若分配成功，则正常返回；否则，转下一步；

（5）返回“执行 r = new node()堆空间分配异常”相关信息。

若上述过程未触发堆空间分配异常，则从执行 r = new node() 正常返回，程序将继续后面语句的执行。

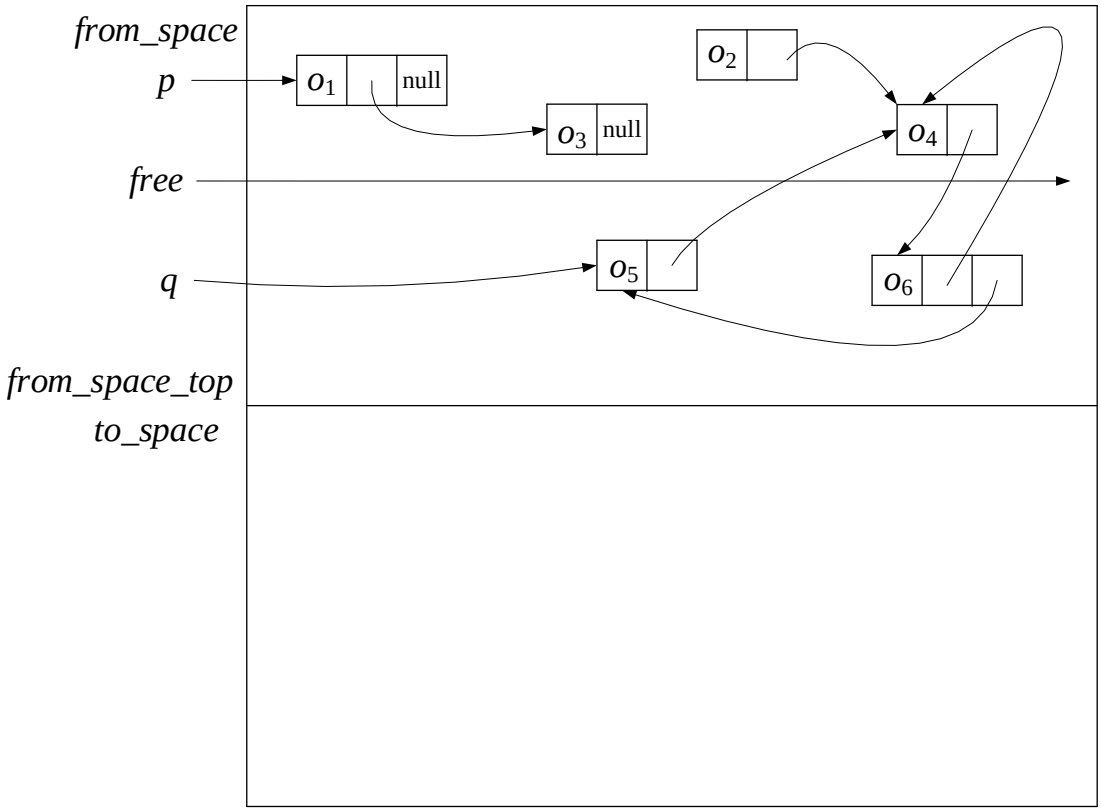


图 28　拷贝回收方法的示例（1）：触发前

图 28 和图 29 所示是关于拷贝回收算法执行前后的一个示例。在某程序开始运行后，经过若干次执行拷贝回收算法，*from_space* 和 *to_space* 角色互换，以及重置 *free* 和 *from_space_top*，到达有一个准备激活拷贝回收算法的情形，如图 28 所示。此时，分配在 *from_space* 和 *free* 之间的结点是前一次执行拷贝回收算法时从另一半存储区拷贝至当前 *from_space* 的数据对象（均为当时可达的数据对象）；分配在 *free* 和 *from_space_top* 之间的结点是执行拷贝回收算法之后新分配的数据对象。假设在如图 28 所示的状态下，无论是分配在 *from_space* 和 *free* 之间的对象，还是分配在 *free* 和 *from_space_top* 之间的对象，有些是可达的，如 O_1、O_3、O_4、O_5 和 O_6。而另一些是不可达的，如 O_2。

在图 28 所示状态下激活拷贝回收算法，执行后，所有可达对象的结点被拷贝至 *to_space* 空间相邻的存储块依次排放。在 *from_space* 和 *to_space* 角色互换，并重置 *free* 和 *from_space_top* 后，到达如图 29 所示的新状态。

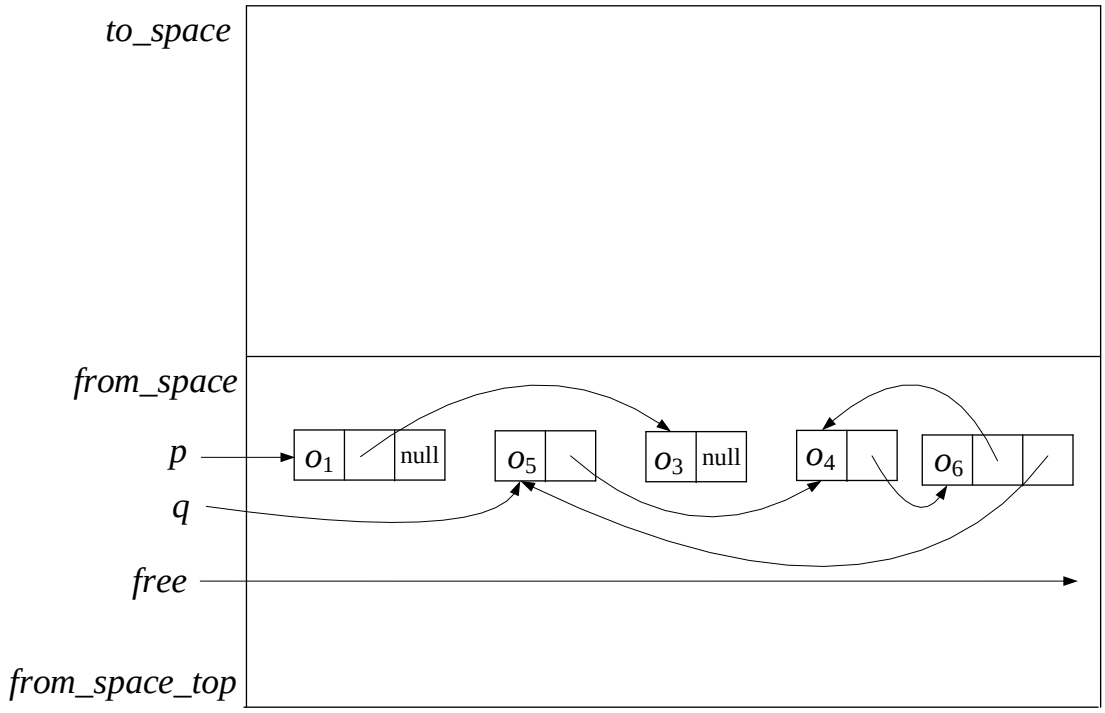


图 29　拷贝回收方法的示例（2）：回收后

上述步骤（3）中提到的 Cheney 算法是一种最基本的基于广度优先搜索遍历可达数据对象的拷贝回收算法，其核心思想可描述为：

（1）令 *scanned* = *free* = *to_space*，即开始时 *scanned* 和 *free* 均指向 *to_space* 的起点；

（2）设根对象集合 *roots* = { *r* | *r* 为当前活跃的程序变量（静态/全局或正处于运行栈中的变量）所分配的堆空间数据对象}；为 *roots* 中对象设定一个次序，对于每个根对象 *r*∈*roots*，依次完成：1）将 *r* 拷贝至 *to_space* 中 *free* 指向的位置；2）置 *free* = *free* + *sizeof* (*r*)；这里，*sizeof* (*r*)表示 *r* 所占用空间的大小；

（3）如果 *scanned* = *free*，则表明所有可达对象均已完成拷贝，所有指针也都指向 *to_space* 中的恰当位置，转（7）；

（4）设 *o* 为当前 *scanned* 所指向的 *to_space* 中的数据对象；

（5）对于 *o* 的每一个指向 *from_space* 空间的非空指针域所指的数据对象 $o.f_i$，依次完成：1）将 $o.f_i$ 拷贝至 *to_space* 中 *free* 指向的位置；2）置 *free* = *free* + *sizeof* ($o.f_i$)；

（6）置 *scanned* = *scanned* + *sizeof* (*o*)；转（3）；

（7）返回。

图 30 是使用 Cheney 算法的一个例子。假设 *p* 和 *q* 是触发拷贝回收算法时当前仅有的活跃程序变量，因此根对象集合 *roots* = { O_1, O_5}。初始时，*scanned* = *free* 指向 *to_space* 的起始位置。

经上述步骤（2），首先将 *from_space* 的 O_1 和 O_5 拷贝至 *to_space* 最开始的连续空间依序存储。此时，*scanned* 指向 O_1 起始位置，*free* 指向 O_5 存储块的下一位置。

随后，执行步骤（3）~（6）的循环，依次针对当前 *scanned* 所指的数据对象。首先，

针对 O_1 执行步骤（5）描述的内层循环，将其引用的唯一数据对象 O_3 拷贝至当前 *free* 所指位置（即 *to_space* 中紧挨着 O_5 的位置）存放。执行步骤（6）之后，*scanned* 指向 O_5 起始位置，*free* 指向 O_3 存储块的下一位置。

接着，针对 O_5 执行步骤（5），将 O_4 拷贝至 *to_space* 中紧靠 O_3 的位置。步骤（6）结束后，*scanned* 指向 O_3 起始位置，*free* 指向 O_4 的下一位置。

然后下一轮针对 O_3，引起未引用其他对象，因此执行步骤（5）后 *free* 无变化，而在步骤（6）之后，*scanned* 指向 O_4 起始位置。

接下来针对 O_4，将 O_6 拷贝至 *to_space* 中紧靠 O_4 的位置。步骤（6）结束后，*scanned* 指向 O_6 起始位置，*free* 指向 O_6 的下一位置。

最后，针对 O_6。执行后，*scanned* 和 *free* 指向同一位置，循环（3）~（6）终止。

执行到步骤（7）结束时，*from_space* 中的数据对象 O_1、O_3、O_4、O_5 和 O_6，被“搬迁”（*forwarded*）至 *to_space* 连续存放，如图 30 所示。而 O_2 以及 *from_space* 中所分配的其它数据对象，则随着 *from_space* 空间整体释放将被回收。

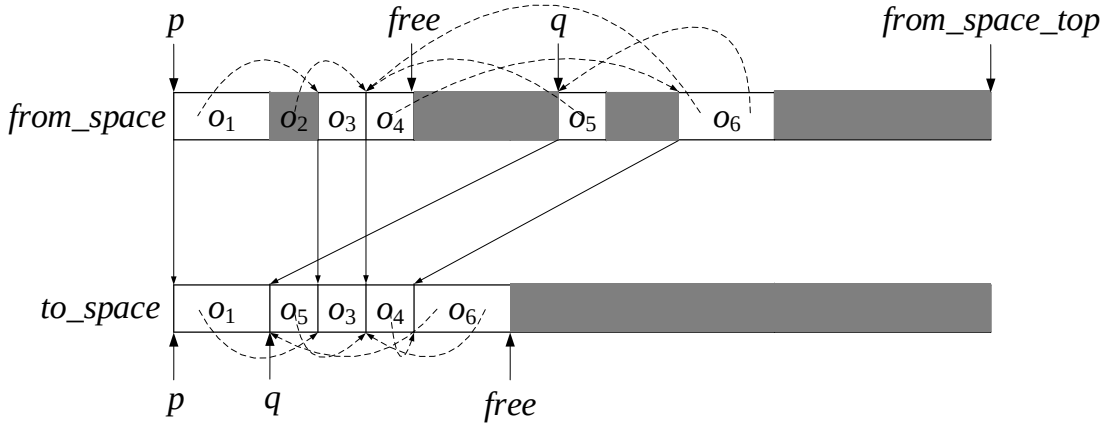


图 30　Cheney 算法示例

上述的 Cheney 算法采用了广度优先搜索技术，仅用到从 *scanned* 至 *free* 的队列，不需要使用栈，也不需要遍历指针，因而比起采用深度优先技术更方便实现。然而，使用广度优先技术带来的问题是引用的局部性（*locality*）较差，影响到虚拟存储系统（含高速缓存）性能的发挥。可采用一些折中的办法，比如穿插深度优先技术，对 Cheney 算法进行优化[5]。

此外，相比前面的标记-清除（包括标记-压缩）回收方法，拷贝回收算法仅对堆空间扫描一遍，因而时间上显著加快，可以大幅节省时间。然而，拷贝回收需要更多空间，因为任何时候，整个堆空间只有一半可以用来为程序分配新的存储块。

研究表明，当活跃数据对象所需空间相比堆空间大小的一半少比较多的时候，拷贝回收方法比标记-清除方法更加有效；而当前者接近后者时，会出现反转，标记-清除法会更有效。具体结论，可参见[8]。

6.5　分代回收

拷贝回收（*copying collection*）法的一大优势就是处理“短命”对象，即两次回收之间

被创建后又被撤销的数据对象。实际上，无需进行任何特殊处理，这些对象会自生自灭。虽然 “短命”对象的占比一般很大[4,5]，然而，实际应用中“长寿”对象也不少见，比如，许多程序开始时就会创建一些贯穿整个执行周期的动态数据对象。这些对象会被反复跟踪，且在可用堆空间的两个半区之间不断搬来搬去，效率很低。

对于标记-清除（包括标记-压缩）回收法，也存在类似的问题。为提高效率，同样也应该聚焦于优先回收较“年轻”的数据对象，相比成熟的（*mature*）“长寿”对象来说，年轻的“短命”对象成为垃圾的比率要高很多。

采用分代回收（*generational collection*）技术，能够达到优先回收年轻“短命”数据对象的效果。下面以在拷贝回收算法中的使用为例，简述分代回收法的核心思想。

这种方法中，堆空间被划分成多个分区，称为“代”（*generations*），假设分别编号为 G_0，G_1，G_2，…，等等。最年轻的对象首先分配在 G_0；G_1 中的所有对象比 G_0 中的任何对象都“年长”；类似地，G_2 中的要比 G_1 中的更“年长”；等等。我们假设是与拷贝回收技术相结合，因此每个 G_i 均有自己的 *from_space* 和 *to_space*。

每次新产生的数据对象均是在 G_0 分区的 *from_space* 分配，当空间不足时触发拷贝回收算法进行垃圾回收。如同 6.4 节中的介绍，拷贝回收过程需从当前根对象集合 *roots* 出发跟踪扫描 G_0 分区 *from_space* 中的可达对象。现在有所不同的是，根对象集合 *roots* 不仅仅包含当前活跃程序变量所分配的数据对象，还要包含比 G_0 更年长分区中的对象，即 *roots* 中还要额外包含 G_1，G_2，…，G_n 中那些有可能指向 G_0 中对象的年长数据对象。如果不这样做，就有可能将活跃的数据对象误判为是不可达的。

如果某个对象在经历 G_0 的若干轮（至少 1 轮，比如 2~3 轮）拷贝回收后仍能存活下来，则将其“提升”（*promoted*）至年长一些的 G_1。若后面再有 G_0 中的对象被提升至 G_1 的，将同 G_1 中原有的对象合并。可预先设定好一个达到提升标准的阈值，在各个对象中添加特殊的位作为计数器，其位数能满足该阈值即可。

若是某个时刻，G_1 中的对象也已拥挤到不能再接纳从 G_0 提升的对象时，则启动对 G_0 和 G_1 相联合的拷贝回收算法。这一联合的拷贝回收算法，所依赖的根对象集合 *roots*，除包含当前活跃程序变量所分配的数据对象，还包含 G_2，G_3，…，G_n 中的那些可能指向 G_0 和 G_1 中对象的数据对象。G_0 和 G_1 中对象的拷贝回收，可以分别使用各自的 *from_space* 和 *to_space*。

然后，在 G_1 中能够经历若干轮次拷贝回收后存活下来的对象将被“提升”至 G_2。某个时刻 G_2 也满了，就要启动对 G_0，G_1 和 G_2 相联合的拷贝回收算法。

依此类推，直到一些十分长寿的对象可以到达最年长的代 G_n。这样，针对这个 G_n 分区，执行（联合）拷贝回收的几率会非常低，甚至可能不再进行。

通过这种方法，“长寿”对象被逐级筛选出来，被重复处理的代价将大大降低。

一般情况下，每一轮次的拷贝回收算法都是针对编号小于等于某个 i 的全部分区，即 G_0，G_1，…，G_i。此时，G_i 是当前空间已满的最年长分区。当然，此时 G_0，G_1，…，G_{i-1} 分区也都已满。这里，在针对 G_i 进行回收时，将全部编号小于 i 的分区也考虑进来，主要是出于如下考虑：1）当前上下文中，比 G_i 更年轻的“代”所包含的垃圾一般也会更多，我们可以将这些年轻分区中的垃圾与 G_i 分区一并回收，同时会使整个回收过程更加高效；2）因为更年轻“代”的分区中可能包含不少指向 G_i 的指针，若根据这一策略，将这些年轻分区中的垃圾与 G_i 分区一并回收，则仅需记住从较 G_i 更年长“代”指向 G_i 及更年轻“代”的引用即

可。

实际中很少有年长的对象引用比它年轻很多的对象。在常见的程序设计风格中，当一个对象创建时，一般会同时初始化其各个域，而后者所指向的对象一定是更年长的。年长的对象引用比它年轻的对象，仅可能是年长对象的某个域有新的更新，这种情况确实不常见。这对于上述分代回收方法来说，是十分利好的一面。尽管如此，如何记住从较年长“代”指向较年轻“代”的引用是这一方法的重要一环，需要妥善处理好这一问题。

如上所述，在针对 G_i 进行回收时，其根对象集合 *roots* 除程序活跃变量所引用的对象外，还包含较 G_i 更年长分区中可能指向比其年轻的分区（G_0，G_1，…，G_i）数据对象的那些对象。为避免搜索 G_{i+1}，G_{i+2}，…，G_n 中所有的对象，需要让编译生成的代码能够记住有哪些年长的对象对年轻对象进行了引用。为此，每当某个对象被“提升”至年长的一代，可以检查它是否包含指向年轻一代对象的指针。如果包含，则将其地址记录下来，以便将来能够跟踪和修改相应的指针。另外，必须要随时监测当前某个年长对象域的指针是否被修改，然后用一个“被修改对象”的向量或集合记录下来，使该对象包含其中；还可以为对象内部设置标志位表明该对象是否被修改，编译器可产生专门的代码负责检查该标志位，以免重复检测。有时，还可以通过检查年长对象所在的堆页面“污染位”是否被修改过，如果是，可将该对象纳入“被修改对象”集合。

经验表明，如果走心设计，分代回收方法可以很有效率，因而在实践中得到广泛使用。根据分代垃圾回收的特点。

然而，上述分代回收方法的本质是重点聚焦于处理会快速变为的“短命”对象上，年轻的“代”进行垃圾回收的机会要更频繁，也更高效，而年长的“代”进行回收的机会较少，而且代价更高。年长的“代”进行回收时，虽然垃圾占比较少，但花费时间较长，需要与所有更年轻的“代”一并进行回收。最年长的“代”拥有最“长寿”的对象，其垃圾回收的代价相当昂贵，基本上相当于是完整的回收，而不是分代回收方法赖以发挥优势的“部分”（*partial*）回收。既然存在这一缺陷，分代回收方法就难免有时引起程序执行时的长时间“停顿”。一个较好的解决方案是与其它可有效处理“长寿”对象回收的算法联用，取长补短。

列车算法（*train algorithm*）是一种专注于“长寿”对象的垃圾回收方法，有兴趣的读者可参阅相关书籍[4]。

6.6 增量回收

每一轮次的垃圾回收如果能将当前堆空间的所有垃圾全部回收固然很好，但有可能造成程序运行的长时间停顿。尽管从整个执行周期来看，停顿时间的占比可能微不足道，但对于响应时间要求高的交互式或实时系统而言，即使一次长暂停也可能是难以容忍的。

增量回收（*incremental collection*）的核心思路是，每一轮次不必要回收堆中全部垃圾，仅回收其中一部分甚至是很少一部分，但要确保遗留下来的所谓漂流垃圾（*floating garbage*）可以在后面的轮次中回收。当然，在确保正确性（回收的一定是垃圾，不确定是垃圾的先留着不能回收，同时能保持内存一致性，不会导致内存泄漏）和满足实时性要求的前提下，每一轮次所遗留的漂流垃圾越少越好。

实现增量回收方法的关键技术是允许增量回收程序（*incremental collector*）和更变程序（*mutator*）交互或并发执行，一般可基于交叉执行（*interleaving*）机制来实现。基本场景是，在垃圾回收程序工作期间，更变程序同时会记录下来可达图信息变更的情况，二者通过

原子操作对共享数据和信息的读取和修改，垃圾回收程序根据更变程序的记录来决定是否继续本轮回收工作。垃圾回收程序所做的决定通常是保守的（*conservative*），要求更变程序对可达性信息的更新满足一定的不变量要求，以确保将来所回收的一定是垃圾，不能确定是垃圾的应在可进一步跟踪的范围内。

先介绍一种采用三色标记（*tricolor marking*）的方案[5]，在基于跟踪的垃圾回收（标记-清除或拷贝回收）方法基础上，可以设计基本的增量回收算法。

三色标记方案中，每时每刻，堆中每个对象都被标记为下列三种颜色之一：

- White（白色）。该对象目前尚未在基于深度优先或宽度优先搜索的跟踪过程中被访问过。
- Grey（灰色）。该对象被访问过，但尚未跟踪其子对象（孩子）。在标记-清除回收算法中，此类对象被记录在栈中；在 Cheney 拷贝回收算法中，此类对象被记录在从 *scanned* 至 *free* 的队列里。
- Black（黑色）。该对象被访问过，其所有子对象（孩子）也都被访问过。在标记-清除回收算法中，此类对象已被弹出堆栈；在 Cheney 算法中，此类对象已完成扫描（*scanned* 已指向其它对象）。

在启动新的一轮增量回收过程之前，假定堆中所有对象均标记为 White。回收程序从根对象（静态/全局或正处于运行栈中的变量所引用的对象）集合 *roots* 开始跟踪可达对象，初始时 *roots* 中对象被标记为 Grey，并记录在当前栈或队列。

回收程序通过下列迭代过程扫描并标记可达对象：

（1）从栈或队列获取当前对象 *o*，并将其弹出栈顶，或 Cheney 算法中执行 *scanned* = *scanned* + *sizeof* (*o*)；若已无 Grey 对象（栈空，或 Cheney 算法中有 *scanned* = *free*），则转（4），结束迭代过程；

（2）对于 *o* 的每一个非空指针域所指的数据对象 $o.f_i$，若该对象是一个 White 对象，则依次完成：1）将 $o.f_i$ 压入栈中（标记-清除法），或 Cheney 算法中将 $o.f_i$ 拷贝至 *free* 指向的位置并置 *free* = *free* + *sizeof* ($o.f_i$)；2）将 $o.f_i$ 标记为 Grey；

（3）对象 *o* 标记为 Black；转（2）；

（4）返回。

回收程序执行期间，可能有更变程序在交替/并发执行，然而二者针对共享数据的原子操作前后，应满足以下不变量要求：

（1）无 Black 对象指向 White 对象；

（2）各个 Grey 对象均在回收程序当前维护的工作栈或者队列中，反之亦然（当前工作栈或者队列中的对象均为 Grey 对象）。

如果回收程序可正常结束，所维护的灰色对象集合（*gray-set*）将变成空集，现存的所有 Black 对象即为本轮增量回收过程跟踪到的可达对象集合（可能是保守的可达对象集合），而当前剩余的全部 White 对象即被当作垃圾而自动回收。

一种简单的算法是将回收程序执行期间新创建的对象直接纳入可达对象范畴，而忽略因

更新引用值可能引起的可达性信息的改变。具体来说，在遇到新分配存储的对象，更变程序直接将其标记为 Grey 对象，并加入栈或队列；而遇到引用值更新操作时，更变程序选择忽略，不影响引用值更新前所指向的对象的颜色。很显然，这一算法满足上述不变量需求。虽然能够确保正确性，但这是一种最保守的方案。根据该算法，当某一轮垃圾回收过程正常结束时，可达对象集合可以表示为：

$$R \cup New$$

其中，*R* 为该轮回收过程启动那一时刻的可达对象集合，*New* 表示在回收程序执行期间新创建对象的集合。容易理解，对于启动时不可达的那些对象，之后也决不会变成可达的对象。

通常情况下，应该考虑到引用值更新时所导致的可达性信息变化，即可能使得某些引用链接断链，丢失一些可达对象。若使用 *Lost* 来表示可能丢失的全部可达对象的集合，则理想情况下最终的可达对象集合可以表示为：

$$(R \cup New) - Lost$$

为确保正确性，增量回收算法一般都是保守的，因此最终的可达对象集合 *S* 满足[4]：

$$R \cup New \supseteq S \supseteq (R \cup New) - Lost$$

多数算法，在回收程序执行期间，若遇到新创建的对象，更变程序先是将其标记为 White 对象；若遇到引用值更新，通常的做法是允许更变程序对可达图和可达性信息进行更新，但需要采取相应的变通措施，以保障不会违反上述不变量。以下是实施保障措施的几类算法。

（1）写保障（*write-barrier*）算法，即保障更变程序在执行“写入”的时候不打破上述不变量要求，可以采取的措施比如：

- 每当更变程序将指向 White 对象 *a* 的指针写入 Black 对象 *b* 的某个域中时，将 *a* 重新标记为 Grey 对象，并使其回到当前工作栈/队列中。
- 每当更变程序将指向 White 对象 *a* 的指针写入 Black 对象 *b* 的某个域中时，将 *b* 标记为 Grey 对象，并将其加入到当前工作栈/队列中。
- 借助虚拟存储系统，将所含全部对象均为 Black 的页面设置为“只读”；每当更变程序将任何值存入这样的全 Black 页面时，缺页错误（*page fault*）的中断处理程序会将该页面中全部对象标记为 Grey，使其回到当前工作栈/队列中，同时将页面恢复为“可写”。

（2）读保障（*read-barrier*）算法，即保障更变程序在执行“读取”的时候不打破上述不变量约束，可以采取的措施比如：

- 每当更变程序读取一个指向 White 对象 *a* 的指针时，如果 *a* 原来不是 Grey 对象，则将其标记为 Grey，并将其加入到当前工作栈/队列中。
- 借助虚拟存储系统，每当更变程序从某个含有非 Black 对象的页面进行读取时，缺页处理程序将该页面中所含的全部对象均标记为 Black，同时将这些对象的每个非空域所指向的对象（子对象）标记为 Grey（对于原来不是 Grey 对象的，需将其加入到当前工作栈/队列中）。

增量回收算法的具体实现细节，与采用的基础算法（如标记-清除或拷贝回收）相关。比如，在基于 Cheney 算法的分代回收方法基础上实现的 Baker 增量回收算法（有兴趣的读

者可参阅[5]）。

7. 面向对象程序运行时组织（选讲）

面向对象是当今程序设计语言的主流特性，面向对象语言的运行时存储组织，是理解面向对象特性实现机制的重要环节。面向对象语言的特性丰富多样，限于篇幅，本节主要围绕最基本的对象存储组织方式展开讨论，有兴趣的读者可进一步以此为基础，探究更加广泛而深入的话题。

7.1 “类”和“对象”的角色

面向对象语言中，与运行时存储组织关系密切的概念是“类”（*class*）和“对象”（*object*）。首先，需要对“类”和“对象”在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成员（*feature member*）：属性（*attribute*），以及例程（*routine*）。在 C++/Java 中，分别对应于成员变量（*member variable*），以及成员函数（*member function*）或者方法（*method*）。

- 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。

在许多语言中，属性和例程可以被定义为静态的（*static*），作为在整个类的范围内共享的属性和例程，类的每个实例均可访问。例如，在 C++/Java 中，成员变量分为类变量（静态变量）和实例变量（局部变量），成员函数（或方法）也分为静态成员函数（方法）和普通成员函数（方法）。这里，假定读者已熟悉 C++/Java 中有关静态成员和普通成员的差异。

除此之外，还需要熟知面向对象机制的许多重要特性，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，默认读者已经在面向对象编程或面向对象软件开发方法等相关领域的学习或工作中已有基本的了解。

7.2 面向对象程序的运行时特征

进一步，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，每个域对应于类中的一个非静态（*non-static*）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，并调用该实例的构造过程创建根对象（*root object*）。例如，纯面向对象语言 Smalltalk 的根类

Object，实际上也是根对象（Smalltalk 程序中的类也是对象），程序的执行就是从创建一个 Object 对象开始。在非纯面向对象情形下，创建根对象相当于通常程序启动 main 过程/函数，如 C++中，通常是采用启动 main 过程/函数的方式创建根对象。又如，每一个 Java 应用需要指定一个主类，该类中必定要包含一个 main 方法（函数），通过启动 main 方法来创建根对象。

- 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 31 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。

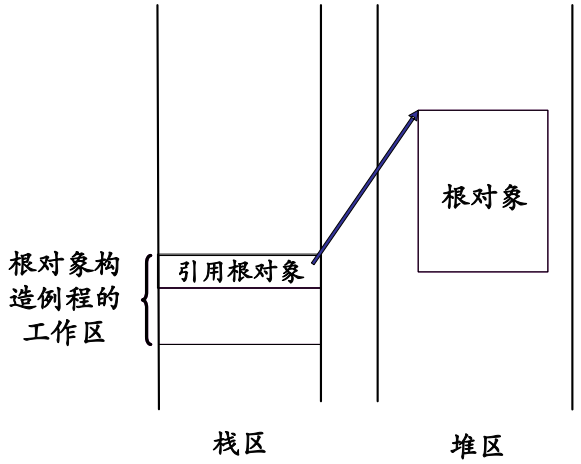


图 31　创建根对象时的存储结构

- 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。另外，实例对象可以接受的消息（*message*）或方法对应于类中的非静态成员函数（方法）。

- 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）。例如，C++和 Java 中的 this 指针。当然，this 只能在非静态成员函数（方法）中使用。

- 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数据结构的一个引用，并在堆区创建一个该数据结构的对象。这一点同普通函数是一致的。

7.3 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，这是一种极端的组织方式。

另一种可能的方法是：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（形成所谓的继承关系图）。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。这种方法在类结构中只记录一次例程指针，且对于每个对象并不将其复制到自身的存储空间。然而，其缺点在于：虽然属性变量具有可预测的偏移量（如在常规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，所以这对于诸如 Smalltalk 等强动态性语言来说是合理的结构。然而，这实际上是另一种极端的方法，虽然节省了对象的存储空间，但却增加了类层次结构的维护，访存次数会比较多，故而使得运行效率会受到很大影响。

下面介绍一种折中的方案，这一方法的基础是例程索引表，指每个类的可用例程的代码指针列表。在许多语言（如 C++/Java）中，例程索引表也称虚函数表（*virtual function table*）或虚方法表（*virtual method table*），简称虚表（*virtual table*）。这一方法的优点在于：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）。对于像 C++/Java 等静态类型的语言，虚表完全可以在编译期间静态确定，可做出安排以使每个例程的指针都有一个可预测的偏移量，并可以采用数组方式实现。

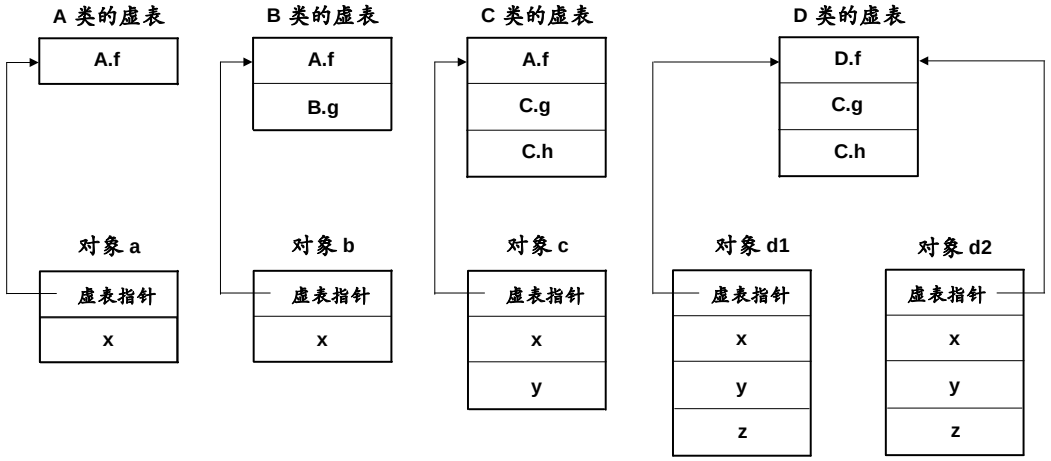


图 32　对象存储示例（1）

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 extends B { int y; void g() {…}; void h() {…} }
class D extends C{ bool z; 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 的声明中含一个属性变量 y，一个例程 g（重载了 class B 中声明的例程 g），以及另一个例程 h，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 z，重载了例程 f，继承了（class A 声明的）属性变量 x，（class C 声明的）属性变量 y、例程 g 和例程 h。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 class 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 32 给出了采用这种折衷方法的对象存储示例。

从图 32 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虚表指针（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 z 是这些对象的所属类 class D 中声明的，而属性变量 x 和 y 是 D 继承父辈类的，所以在 d1 和 d2 的存储区中，属性变量 x 和 y 的存储位置排在属性变量 z 的存储位置之前（x 和 y 之间的次序关系也是二者“辈分”关系的体现）。

根据这一方法，每个类都对应一个虚表（例程索引表）。如图 32 所示，class A 的虚表包含指向 class A 中声明的例程 f 的指针 A.f；class B 的虚表包含指向 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；class C 的虚表包含指向 class A 所声明的例程 f 的指针 A.f，指向 class C 中声明的例程 g 的指针 C.g，以及指向 class C 中声明的例程 h 的指针 C.h；最后，class D 的虚表包含指向 class D 所声明的例程 f 的指针 D.f，指向 class C 中声明的例程 g 的指针 C.g，以及指向 class C 中声明的例程 h 的指针 C.h。在虚表中，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 class B 和 class C 的虚表中，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 class D 的虚表中，D_f 排在 C_g 之前）。

值得注意的是（前面也提到过），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，所以虚表（例程索引表）中不包含静态例程的指针。类似地，若是属性变量被声明为静态的，也不会在对象内存空间中出现。

7.4 例程/方法的动态绑定

我们先了解一下针对面向对象语言中 this 变量/关键字的一种常见处理方式，这有助于理解例程/方法的动态绑定。

在通常的面向对象语言（如 C++/Java）中，在例程内部可以使用 this 变量/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，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。实现这一特性的一种方法是把 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。

下面看一个例子。设有某个简单的单继承面向对象语言（类似 Java）的如下代码片段：

```
class Fruit
{
    static string day;
    int price;
    string name;
    void init(int p, string s) {
        price=p;
        name=s;
        …   /*here*/
    }
    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");
    }
}

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

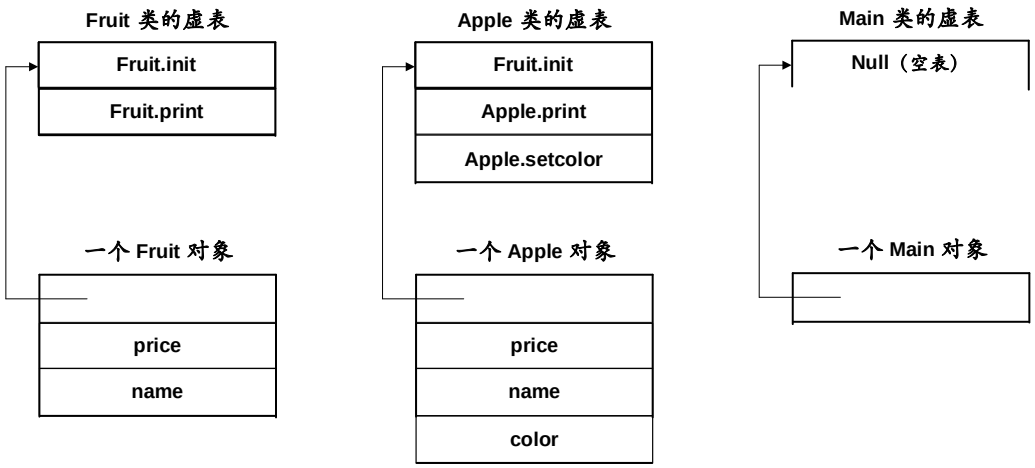


图 33　对象存储示例（2）

参考 7.3，如果创建该例中各个类的实例对象，则其存储组织如图 33 所示。这里应注意，静态属性不在对象空间中分配单元，如每个 Fruit 对象的存储空间不含 static string day；虚表中不包含静态函数/方法，如 static void main()不在 Main 类的虚表中。

初始化一个类的实例对象，常常是通过显式地向该类发送一条 new 消息或者是调用以该类为参数的 new 函数/方法来完成。比如，通过调用 New (Apple)创建一个 Apple 对象。

在上面的例子中，假定 class Main 为所指定的主类。类似于 Java，我们通过运行该类中的 main 函数/方法来创建根对象。下面简要分析这一 main 函数/方法的执行。

首先，语句 day="Tuesday"为分配在静态存储区的变量 day 赋值，使其指向静态数据（串）"Tuesday"。静态变量与常量一般位于静态存储区的不同区域。比如，前者所分配的区域习称为 BSS 区。

接着，通过调用 New (Apple)创建一个 Apple 对象。可以将 New 看作 Apple 的一个特殊的类方法（静态方法），它首先是向存储分配程序（通常是调用库函数）申请适当大小的堆存储空间，将第一个单元置为指向 Apple 类虚表的指针，后续单元依次存放该对象的成员变量（先放继承的变量，并且辈分越大越靠前），如图 34 或图 33 中间部分所示。紧接着，这一 Apple 对象被赋值给一个 Apple 实例变量 a，可通过 a 访问该对象的存储空间。

接下来，执行 a.setcolor("red")所动态绑定的是 Apple 类中的方法 setcolor，将 a 所指对象（后面简称其为对象 a）的 color 属性域置为指向静态数据（串）"red"的指针。

当执行语句 a.init(100,"apple")时，实际上是调用 Fruit 类中声明的 init 方法的代码，为对象 a 的属性域 price 和 name 设置初值。当执行到/*here*/处时，当前的存储空间分配情况如图 34 所示。

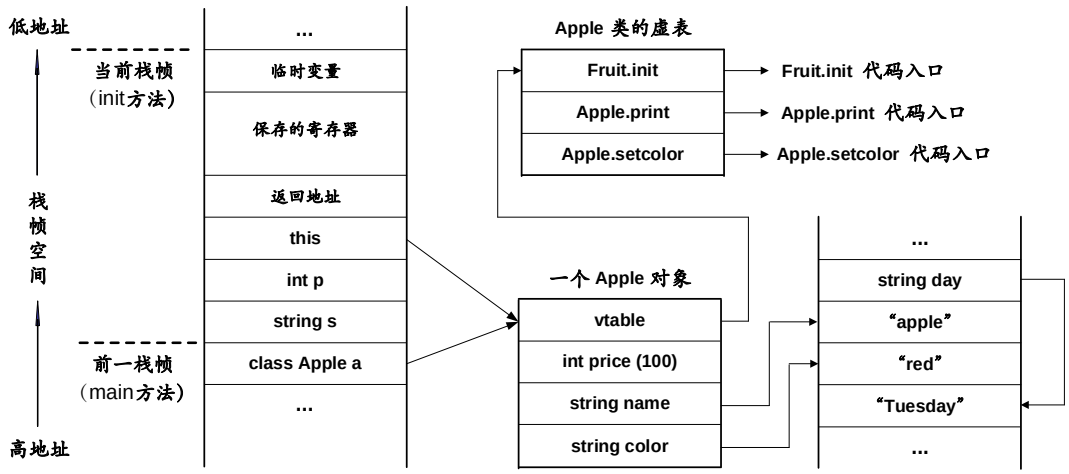


图 34　例程/方法的动态绑定

图 34 左边展示了栈区的当前状态，包含调用 class Main 中 static void main()以及 class Fruit 中 void init(int p, string s)所对应的两个活动记录。这里，假定所有输入参数依次存放在当前活动记录/栈帧的起始位置。值得注意，方法 init 有默认参数 this，指向对象 a。图中所示的栈帧中其余单元的内容仅为示意，不必细究。

图 34 中间部分刻画了当前堆区分配的 Apple 对象。本例中，虚表及其分量的访问方式在编译期间即可确定，因而可分配在静态数据区。

前面已提到，Fruit 类的静态变量 day 以及一些串常量态数分配在静态数据区，如图 34 靠右边的部分所示。

当执行完 a.init(100,"apple")，图 34 所示的当前栈帧被撤销，然后开始执行 a.print()。此时，通过访问图中所示虚表的第二个代码入口，绑定的是 Apple 类中所定义的 print()方法。

7.5 其他话题

面向对象语言特性的实现中还有许多与运行时存储组织密切相关的有趣话题，限于篇幅本课不作专门讨论，例如：

- 类成员测试（Testing Class Membership）。比如 Java 中，执行 o instanceof C 可用来测试 o 是否为 class C 的实例对象，这里 C 可以是 o 所属类的父辈。例如，若 o instanceof D 返回 true，而 D 是继承 C 的子类，则 o instanceof C 也返回 true。一种直接的实现方案是维护完整的类继承层次，类成员的测试相当于对类层次关系进行搜索，这种方案对于强动态类型的语言（如 Smalltalk）可能是必要的，因为类层次关系也可能是动态改变的。对于静态类型的语言（如 Java），除了通过静态的类层次关系来实现，另一种实现方式，可以在每个类的虚表中增加两个单元的信息（指向父类虚表的指针和指向本类名字串的指针）。还可以采用其他效率更高的方法，如采用 Display 表的方法[5]。

- 对象的创建和撤销。涉及到对象的构造（*construction*）和析构（*destruction*）、垃圾回收（*garbage collection*）等内容。关于垃圾回收，可参考本讲第 6 节。

- 更多对象的操作。如对象的赋值、克隆（*clone*）、比较、持久化（*persistence*）存储等内容。这些方面可以学习面向对象编程或面向对象软件开发方法等相关领域的相关内容，这里我们不展开讨论。

- 多继承性。有些语言（如 C++）支持多继承性，允许继承多个父类。Java 是单继承语言，仅允许继承一个父类，但增加了接口（interface）这一折衷的特性。比起前面着重讨论的单继承性，实现多继承以及接口特性显然更加复杂，但若是不考虑性能，读者应该能够设计出可行的实现方案。

- 例外处理（*exception handling*）机制。多数现代语言均支持例外处理，这是一个相对复杂的特性，实现起来需要考虑的细节也较多。需要解决的一个基本问题，是当程序运行有例外发生时，如何定位到并执行相应的例外处理程序（*exception handler*）。最直接的实现方法是维护一个特定的运行时数据结构，用以刻画当前处于活跃状态的例外处理程序，当需要的时候通过它可以找到相应的例外处理程序。这一方法的缺点是运行时开销比较大，每当进入和退出包含例外处理程序的作用域时，都需要对这一数据结构进行修改，即使不发生例外，也要对其进行动态维护。有关改进的实现方法，有兴趣的读者可以参考其它相关文献资料。

8 函数式程序运行时组织（选讲）

函数式编程一直是程序设计语言领域所研究的主流编程范型之一，除典型的函数式语言（如 Lisp/Scheme，Haskell，ML）外，如今越来越多的语言支持函数式编程特性，不仅是一些新兴语言（如 Scala，Go，Rust），甚至包括传统的命令式语言（如现代 C++语言）。

有关函数式语言编译（含优化）的内容相当广泛，本节主要讨论与函数式语言运行时存储组织密切相关的一些话题。

8.1 “等式推理”与“高阶函数”

提到函数式语言的特性，最核心的两个概念是“等式推理”和“高阶函数”。

在命令式语言（*imperative programming languages*）中，可以认为，变量代表的是某一物理对象，该物理对象对应有存储单元（*memory cell*），存储单元的地址（首地址）称为左值（*l-value*），而存储单元的内容（数据）称为右值（*r-value*），其具体大小和存放格式取决于变量的类型。这样，在命令式语言中可以有赋值语句。例如下列 C 语言语句

```
x = x +1
```

的执行效果，可以理解为对变量 x 所代表的物理对象的右值进行修改（加 1），结果仍然保存于该物理对象的左值所指向的存储单元。换句话说，一个物理对象（唯一的左值）可以拥有不同的右值，而赋值就是一种可以修改某个变量（代表某个物理对象）右值的操作。

然而，在纯函数式语言（*pure functional programming languages*）中，变量代表的是数学对象。不同于物理对象，针对数学对象，没有“存储单元”或“修改存储单元内容”的概念，也没有左值和右值的概念。因此，纯函数式语言中不支持赋值操作。这种情况下，用某个函数（操作）对一个变量进行定义，如

y = f (x)

形式上看似命令式语言中的赋值，而实际上与赋值的含义完全不同，在纯函数式语言（如标准 ML）中，每个变量只能被定义一次，指向某个数学对象，因此这里的 y 与其说是“变量”，其实是可以将其看作是一个“名字”，其含义（数学对象的内容）一经定义将会永久保留，不会发生改变（*immutable*），直至它所指向的对象消失，比如，栈中的对象随栈帧撤销而失效或分配在堆中的对象被垃圾回收器回收。

另外，纯函数式语言满足引用透明性（*referential transparency*），即一个函数的计算结果仅取决于其参数的值，无论其之前是否执行或执行过多少次计算，以及其参数是以何种次序求值，均不会影响函数的计算结果。具备这种引用透明性，就可以进行类似于数学（代数）中的等式推理（*equational reasoning*）。

例如上述定义 y = f (x)中，无论何时，对于同样的 x，则 f (x)的返回值都是一样的。又因为 y 仅被定义一次，这样，y = f (x)实际上可以看作是一个等式，在任何时候以及 y 的任何上下文中，均可以将其替换为 f (x)。因此，g(f (x), y)，g(y, f (x))，g(y, y)以及 g(f (x), f (x)) 所返回的结果都是相等的。

命令式语言中函数的计算不具有这种引用透明性，可能会有副作用（*side effects*）。例如，若使用 a = f (x)和 b = f (x) 定义两个数学对象 a 和 b，由于存在副作用，则函数 f (x)的两次计算结果可能不同，因此不能保证等式 a = b 成立。

现实中，许多现代的函数式语言会保有某些形式的赋值操作，因此是非纯（*impure*）函数式语言。

函数式语言的一个核心特性是支持高阶函数（*higher-order functions*）。无论是纯的还是非纯的函数式语言，函数均被看作是一类对象（*first-class object*）或头等公民（*first-class citizen*），即如同程序中的普通对象（如变量、常量、普通表达式等），可以用于程序中其它对象的定义中，也可以用作其他函数的参数或返回值。

函数式程序中的函数类似于数学函数，是从其定义域或论域（*domain*）到值域（*range*）或协域（*codomain*）的映射。例如，由下列映射

sqr : Integer → Integer，其中 sqr (n) = n^2

所定义的函数 sqr 是整数 Integer 集合上的一个全函数（*total function*）。

那么 sqr 具有什么形式的“值”，在函数式程序中如何表示它呢？一种办法是借助λ-表达式（*lambda expression*），源自于λ-演算（*lambda calculus*）[11]。λ-演算是函数式编程的基础，在函数式编程语言出现之前（始于 1930 年代，Alonzo Church）已被广泛研究。基于λ-演算的计算模型具有与图灵机等价的计算能力。

λ-表达式可以方便地描述匿名函数，即不出现名字，仅给出参数和函数的定义。例如，下列λ-表达式

(λn . n*n)

其中的标识符 n 为函数参数，n*n 为函数体。在实际的函数式语言中，λ-表达式有各自不同的写法，例如在 ML 中，(λn . n*n)可表示为：

fn n => n*n

上述匿名函数是多态的（*polymorphic*），在严格静态类型化的语言（如标准 ML）中，编译器通过类型推导（*type inference*）可以推断出它在程序上下文所对应的类型。比如，在（fn n => n*n）(2）中，函数（fn n => n*n）的类型是 int->int；在（fn n => n*n）(2.0）中，函数（fn n => n*n）的类型是 real->real；而在（fn n => n*n）(fn n => n+n）(2）中，函数（fn n => n*n）的类型则是 (int->int)->(int->int)。

在λ-表达式中可以指定参数的类型，如在 ML 中，下列λ-表达式

fn n : int => n*n

是一个类型为int->int的函数，其输入参数的类型为 int。

这两个λ-表达式（fn n => n*n）和（fn n : int => n*n）均可用在前面的有名数学函数 sqr 的定义中，此时可以将前者看作名字（或变量）sqr 的取值。例如在 ML 中，sqr 可声明为

```
val sqr : int -> int =
      fn n : int => n*n
```

或者换作如下看起来更自然的方式来描述

fun sqr (n: int) : int = n*n

函数式语言中，函数为一类对象，可用作其他函数的参数或返回值。例如在 ML 中，

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)
```

定义了高阶函数 map，具有类型 ('a->'b) -> 'a list -> 'b list，其中'a 和'b 代表任意类型。map 以类型'a -> 'b 的函数 f 为参数，而返回值为另一个类型为'a list -> 'b list 的函数。比如，若取以函数 fn x => x+5 为参数，则 map(fn x => x+5)返回一个以 int list -> int list 的函数。对于后者，若以表[1,2,3,4]为输入，则可得对 map(fn x => x+5)([1,2,3,4])的求值结果为[6,7,8,9]。

8.2 函数式程序的运行时存储特征

在函数式语言的设计与实现中，需要充分考虑运行时的存储组织与管理。首先，需要对函数式语言的基本特性提供支持，比如无论是对纯函数式语言（无副作用，等式推理），还是针对有副作用的非纯函数式语言，均可以方便地实现高阶函数这一核心特性。其次，还需要充分面对函数式程序运行时的性能问题，能够在存储组织和管理方面尽可能多地提供对相关编译优化的支持。相关内容涉及面较广，下面仅列举通常较为关注的几方面基本内容：

- 函数式语言特别是纯函数式语言，所涉及的对象主体是数学对象。不像物理对象那样允许修改/共享/复用，数学对象具有不可更变（*immutable*）的性质，即只能被定义（初始化）一次。在函数式程序运行过程中，数学对象的增长速率一般会很快，其中运算的中间结果占比可能居多数。这样一来，不可避免会伴随着不断产生新的垃圾对象。因此，高效的垃圾回收程序对于函数式语言的实现来说意义重大。实际上，垃圾回收机制也正是伴随函数式语言的出现而开始受到人们重视的，后来被推广成为现代程序设计语言的基本特性之一。本讲第 6 节讨论了实现垃圾回收机制的基本策略和算法。

- 函数式语言中，函数是一类对象，函数本身可以用作其他函数的参数或返回值。函数的代码自然是函数对象的重要部分，然而还不够，函数对象的求值还需要一个求值环境，后者依赖其定义时的上下文。函数对象的代码及其求值环境的组合称之为闭包（*closure*），后者是函数式语言的设计与实现需要考虑的基本环节。有关闭包的运行时存储组织，参见 8.3 节的讨论。

- 因当今计算机体系结构的不适配性，相比命令式语言，函数式语言程序在性能方面处于劣势。因此，函数式语言的实现必须在编译优化方面下足功夫，同时也需要程序员培养某些良好的函数式编程风格和技巧。许多在命令式语言编译器中使用的传统优化均可用在函数式语言实现中，比如函数调用的内联展开[5]（*inline extension*）。对于这些常规优化方法，有时也可以利用（纯函数式语言中）等式推理特性达到某些在命令式语言中无法达到的效果。还有一些是函数式语言特有的编译优化工作，比如闭包变换（*closure conversion*）[5]。无论是手工编写，还是编译器自动转换，其中的一些函数式程序相关变换与目标代码的运行时存储组织关系较为密切，在随后的小节里我们主要介绍其中两种：

  （1）8.4 节主要介绍尾调用和尾递归对函数活动记录的影响。程序员在解决应用问题时尽可能使用尾调用和尾递归，可以简化活动记录的设计，优化调用代码序列。此外，有一些关于将函数调用自动转化为尾递归形式的方法可以在编译器设计时使用，如变换为 CPS（*continuation-passing style*）形式的尾调用[5]。

  （2）8.5 节介绍延迟求值（*delayed evaluation*）和惰性求值（*lazy evaluation*）及相关的变换，后者需要用到特殊的存储组织，即函数记忆簿（*memoization*）。对于纯函数式语言，可以通过延迟求值（或传名调用）来支持函数的某些非严格定义，从而可拓展等式推理的内涵。为避免延迟求值可能带来的重复计算，可借助创建和维护函数记忆簿，从而实现以存储代价换取优化性能的惰性求值机制。

8.3 闭包的存储组织

如前所述，闭包（*closure*）是函数定义（代码）及其求值环境之间的绑定。函数式语言支持高阶函数，允许函数用作其他函数的参数或返回值，具体实现时是将闭包用作参数和返回值。因此，闭包的存储组织在函数式语言的设计与实现中至关重要。

在 5.2.3 节通过一个简单 ML 例子讨论过，对于包含嵌套声明的语言，可以通过传闭包（*call-by-closure*）的方式实现函数参数的传递。类似地，闭包也可作为其他函数的返回值。

按照 5.2.3 节例子中闭包的存储组织方式，函数的求值环境是基于栈上活动记录中的静态链，即通过栈帧中的静态链查找相对于当前函数的非局部量。然而，这种方式不具有一般性。例如，对于下面的 ML 程序片段：

```
let
   type intfun = int -> int

   fun add (n: int) : intfun =
      let fun h (m: int) : int =
         n+m
      in h
      end

   val addFive : intfun = add (5)

   fun map f nil = nil
      | map f (h::t) = (f h) :: (map f t)

   val and5Map = map (addFive)
in
   val and5Map([1,2,3,4])
end
```

其中，函数 add 基于输入参数 n 返回类型为 int -> int 的函数 h。针对不同的 n，将对应有函数 h 的不同实例，因而需要有不同的闭包来记录 h 函数内所要访问的非局部量 n。如果如 5.2.3 节那样，使用栈帧中的静态链作为闭包的非局部量求值环境，那么就必须要求当 add 返回时其栈帧不能被撤销，否则 h 函数将会失去求值环境。像 n 这种需要在内层嵌套函数中使用的变量，称为逃逸变量（*escaping variables*）。

这样，如果存在逃逸变量，则如 5.2.3 节的方法（栈上活动记录静态链作为闭包的非局部量求值环境）不再有效。一种可行的解决方案，是在堆中创建活动记录，其何时被撤销或收回交由垃圾回收程序负责。我们将闭包的非局部量求值环境，修正为这种基于堆式活动记录中的静态链信息。

在[5]的实现示例中，函数的堆式活动记录也称为逃逸变量记录（*escaping-variable record*），其中仅包含：1）内层嵌套函数中可能用到的该函数的任何局部变量；2）该函数求值环境（即外层函数的逃逸变量记录）的静态链信息。

在图 35~37 中，我们以上面的 ML 程序片段为例，采用[5]中介绍的实现技术以及其相同的记号，示意了这种闭包存储组织方式。记号 EP 表示当前函数的堆式活动记录（逃逸变量记录）指针（亦即环境指针），SL 表示堆式活动记录中的静态链信息（指向外层函数的堆式活动记录），RV 代表返回值（或函数闭包）。

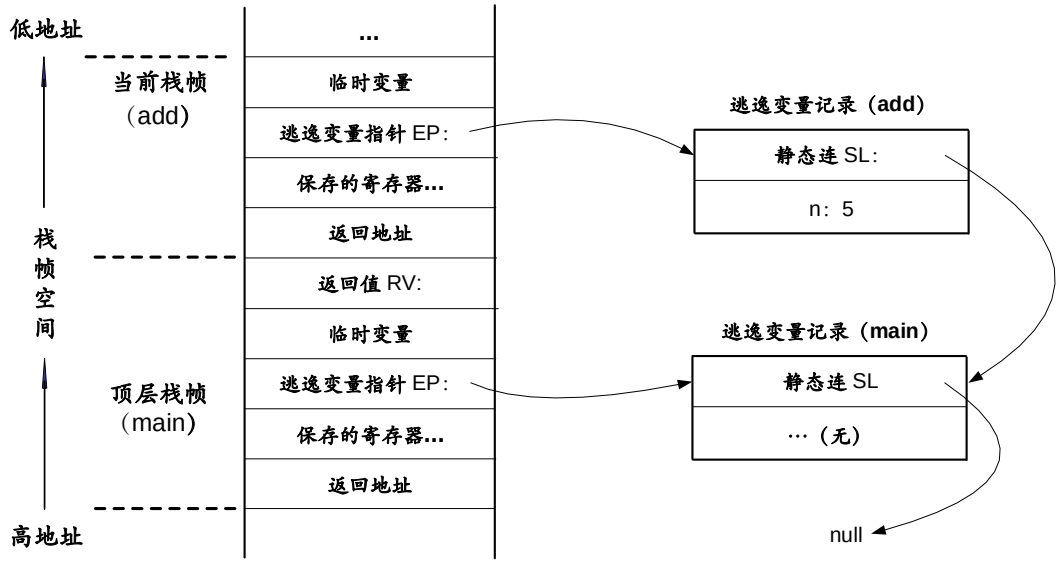


图 35　闭包的存储组织示例（1）

图 35 展示的是在主函数开始执行后，先执行 add 的调用，当进入 add 函数的内部时，两个函数（add 和顶层函数 main）的当前栈帧以及堆式活动记录（逃逸变量记录）。注：add 是叶子函数（内部未调用任何函数），因此栈帧中未包含返回值 RV。

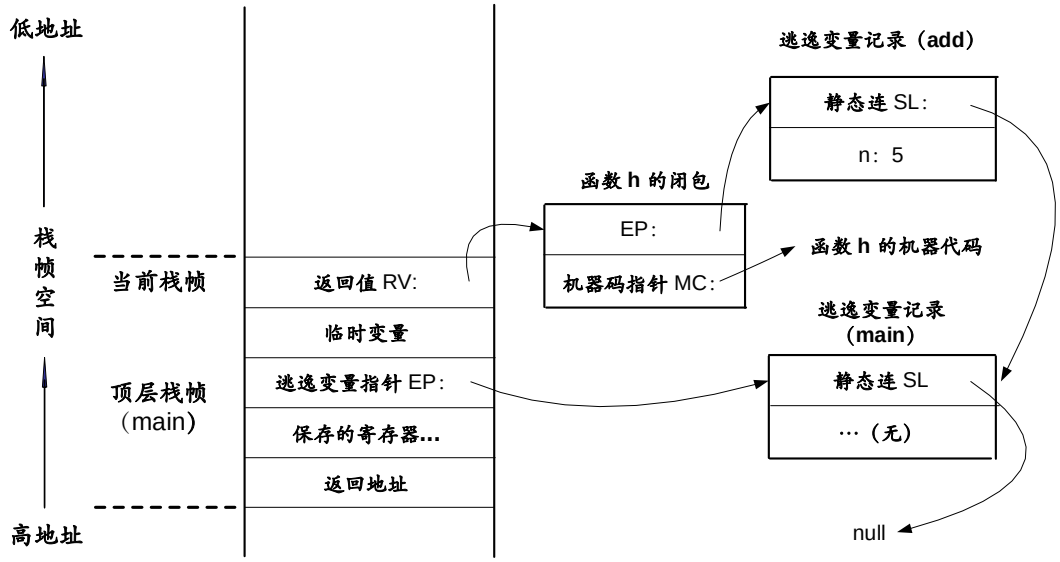


图 36　闭包的存储组织示例（2）

图 36 描述了在撤销 add 函数栈帧并返回 main 时，当前（顶层）栈帧以及堆中的闭包和堆式记录（逃逸变量记录）相关内容。注意，调用 add 函数后返回一个函数 h 的实例，返回值是该函数实例的一个闭包，RV 指向这一闭包，该闭包由 h 的机器代码指针及其求值环境构成，后者为指向 add 的堆式活动记录（内含逃逸变量 n）。

最后，图 37 刻画了在调用 map (add (5))后，并执行至 map 函数内部时的当前栈帧以及堆中的闭包和两个函数（map 和 main）的堆式活动记录（逃逸变量记录）。这里，我们假设 map 函数是尾递归，调用时共享一个栈帧。

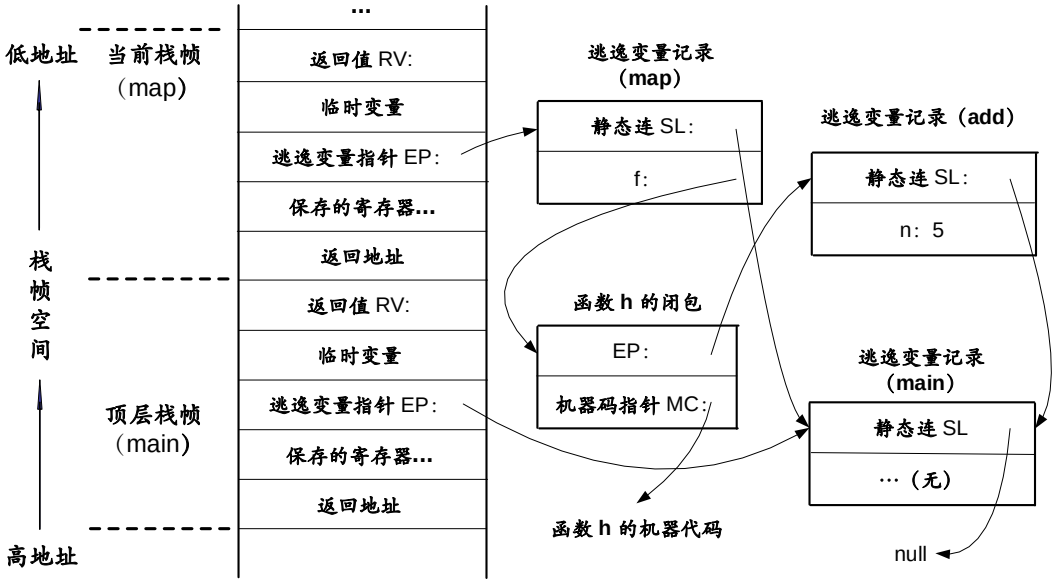


图 37　闭包的存储组织示例（3）

8.4 尾调用/尾递归的活动记录

数学中，阶乘函数可由下列公式来定义：

$$factorial(n) = \begin{cases} 1 & , \quad n = 0 \\ n \times factorial(n-1) & , \quad n > 0 \end{cases}$$

这种递归定义的函数，很方便在函数式语言中编写。例如，以下标准 ML 语言的代码片段[12]，定义了一个 factorial 函数，看起来几乎与上述数学定义一致：

```
fun factorial 0 = 1
  | factorial (n: int) = n * factorial (n-1)
```

然而，这种采用递归实现的函数/过程，由于需要频繁进行函数/过程调用，人们常常会诟病其效率。若是不采用递归，而是通过迭代，可以显著提高执行效率。例如，以下是实现 factorial 函数的一个类 C 语言代码片段：

```
int factorial (int n) {     /* 假设 n 不小于 0 */
    int tmp =1;
    while (n > 1)
        tmp = n * tmp;
        n = n-1;
    }
    return tmp;
}
```

该函数通过内部循环（迭代）实现阶乘的计算，不需要额外的函数调用，因而效率很高。但对于函数式语言来说，这种循环（迭代）计算还是需要通过递归方式来实现。对于这种固有的效率问题，实现函数式语言的编译器往往可以通过一些好的“优化”技术为递归函数生成高效的代码。例如，一种好的函数式编程风格是尽可能使用“尾递归”方式来实现各种递

归或迭代计算，或者可能的话，由编译器自动转换为这种形式的递归。以下是实现 factorial 函数的另一个标准 ML 代码片段[12]：

```
fun helper (0, r: int) = r
  | helper (n: int, r: int) = helper (n-1, n*r)
fun factorial (n: int) = helper (n, 1)
```

这一代码片段中，函数 factorial 的定义被分成了两个部分。值得注意的是，这里的 helper 是一种“尾递归”形式的函数。通常，函数式语言编译器会将这种形式的递归函数优化为迭代计算的形式，生成一种循环结构的代码，类似于上述类 C 语言实现的 factorial 函数。

设有函数（过程）f 内部有函数（过程）g 的调用，如果当 g”返回“时，f 需要做的仅剩”返回“，则称这一调用为尾调用（*tail call*）。若其中的 f 和 g 是同一个函数（过程），则称该调用为尾递归（*tail recursive*）调用。

尾递归调用实际上仅完成迭代计算，类似于命令式语言中循环的作用，因此在优化翻译时，可以将返回语句对应生成跳转语句，使得返回时可以跳转到当前函数/过程的顶部，外加某些赋值语句达到参数传递的效果。这样，得以重用当前活动记录（如上述尾递归函数 helper 可以仅创建一个共享的活动记录），代码也就避免了因函数/过程调用带来的许多负担。比如，可以省略创建活动记录，对参数求值，保存和恢复寄存器，以及调整 display 表和静态链等。多数调用发起阶段（*prologue*）的工作不再需要，而调用收尾阶段（*epilogue*）则完全消失。

对于尾调用，可进行类似的优化，比如可以将调用改为跳转，不用完整创建新的活动记录，被调用者执行结束后撤销调用者的活动记录，等等。不过有两点值得特别关注，首先，需要确保调用者和被调用者的两个函数（过程）体对编译器同时可见，或者至少在生成调用者代码时能有被调用者的足够信息可供转移控制。其次，调用者和被调用者所需的活动记录大小会有差异，当被调用者的活动记录需要更大空间时需要予以补足。具体来说，为替换掉尾调用，需要完成的核心工作包括：1）对尾调用的参数进行求值，并且存放至被调用者能够找到的地方；2）当被调用者栈帧空间大于调用者栈帧空间时，需要有指令能根据二者的差值进行扩展；3）跳转至被调用者的被调用者函数（过程）体的起始位置。

8.5 函数记忆簿与惰性求值

函数式语言可以是纯的和非纯的（参见 8.1）。进一步，纯函数式又有严格（*strict*）和非严格（*non-strict*）之分。如果一个（无副作用的）函数的求值要求其所有参数均有定义，则称该函数为严格的，这种情况下函数的计算结果与其参数的求值顺序无关。没有这一要求的函数称为非严格的。如果一个函数式语言中，要求其所有函数都是严格的，则称其为严格的语言。否则，若一个函数式语言允许定义非严格的函数，则称其为非严格的语言。例如，一般认为，标准 ML 和 Scheme 为严格的函数式语言（非标准实现或某些宏支持除外），Haskell 为非严格的函数式语言。

在严格的函数式语言里，函数调用（*application*）时所有的参数均要正常求值，类似于传值（*call-by-value*）的参数传递方式，通常也称为严格求值/即时求值（*eager evaluation*）。根据这一求值策略，变量在绑定时均需要进行求值计算，无论在整个运行期间变量是否会真正用到。

下列 ML 函数定义

```
fun reciprocal (x : real) : real =
  let
    fun cond (x, y : real) : real =
      if x = 0 then 0 else y
  in
    cond (x, 1/x)
  end
```

在严格求值策略下，cond (x, 1/x)的参数求值可能会发生运行时错误（x=0 时）。一种解决方案可以是编写带有例外处理的代码，在遇到 x=0 时通过例外处理程序对函数定义进行完善。然而，其实在 x=0 时，参数 1/x 的取值在函数 cond 内部是不会用到的。注意，这里的 if-else 表达式求值在 ML 中是非严格的[12]，而 cond 像其他函数要求一定要求是严格的。

又如，在计算针对某一参数执行函数 fn x => 1 调用的结果时，实际上是不需要对参数进行求值的，但因 ML 遵循严格求值策略，所以必须对参数进行求值并能返回正常结果。

为解决这一问题，使得可以仅在实际用到的时候再对表达式进行求值，需要一种对表达式/变量的延迟求值（*delayed evaluation*）技术。在函数式语言中，延迟求值可以通过传递算壳（*thunks*）来实现，后者也有人翻译为形实替换函数/程序。

例如，在 ML 中，可以将算壳被定义为一种类型为 unit -> 'a 的函数，这里 unit 代表 0-元的积类型（即 0 元组，0-*tuple* 或 *null tuple*）。对于类型为'a 的表达式 *exp*，其算壳可写作 fn () => exp。函数被调用前，算壳 fn () => *exp* 作为一个函数值传递给形参变量，可以成功将表达式 *exp*“封冻”（*frozen*），规避了对其求值。在函数体内，若实际执行时遇到算壳 fn () => *exp*，会对其进行强迫求值，相当于针对 0 元组（即无任何参数的参数列表）来调用该函数，实际上是“解冻”（*thaw*）了表达式 *exp*，并完成对 *exp* 的求值。

可见，这种“封冻”和“解冻”的策略，实际效果相当于前面（5.2.4 节）提到的传名字（*pass by name*，*call-by- name*）的参数传递方法式（简称传名调用），类似于宏替换/宏展开，仅对参数名字的每一处出现用实参文本进行替换，替换时不对形参表达式进行求值，而是在替换后的上下文中才去执行。

技术环节上，上述“封冻”和“解冻”功能的代码常常会被分别包装成 delay 和 force 两类阶段性函数。比如，对于前面 reciprocal 代码中调用的 cond 函数，通过 delay 函数，会将形参改写为算壳类型（unit -> real），两个实参分别为 fn () => x 和 fn () => (1/x)；而在 cond 函数内部，通过 force 函数，会将 x 的求值替换为对 fn () => x 的求值，将 y 的求值替换为对 fn () => (1/x)的求值。delay 和 force 函数可以由程序员手工编写，也可以交给编译器统一实现。

下列简单的 ML 代码片段[12]，可帮助理解“替换为算壳”和“对算壳求值”的含义：

```
val thunk =
    fn () => print "hello world"          (* 不打印任何内容 *)
val _ = thunk ()                          (* 打印"hello world" *)
```

借助这种传递算壳（*pass by thunk*）的方式，可以很好解决延迟求值的技术问题。然而，伴随而来的是另一个需要解决的重要问题：同一个表达式的求值可能会重复多次。例如，

```
fun f x => x+x
```

假设函数调用 f(*exp*)按照上面的方法被替换为 f(fn () => *exp*)，那么在函数体（x+x）内，*exp* 将会被“解冻”两次，即需要对 *exp* 重复计算两次。

针对这种对算壳函数重复求值的问题，可以借鉴一种面向普通函数重复计算问题的解决方案：基于函数记忆簿（*memoization*）。借助函数记忆簿存储空间，可以将函数的参数和结果值保存在某种数据结构当中，用一部分的内存开销来提高计算的整体性能，常用在递归和重复运算较多的场景。这一技术的核心，是在函数记忆簿中缓存（*caching*）函数先前的的计算结果，如果函数的当前输入值可以匹配函数记忆簿中已有的记录，则这一“有记忆的”函数能够立即返回预先已经计算好的值。

以下是一个用到函数记忆簿的简单例子（类似于[12]中计算 Catalan 数的函数定义）：

```
Local
   val limit = 20
   val memopad = Array.array (20, NONE)
in
   let
      fun factorial 0 = 1
        | factorial n = n * fact (n-1)
      and fact n =
            if n < limit then
               case Array.sub(memopad, n) of
                  SOME r => r
                  | NONE =>
                     Let
                        val r = factorial n
                     in
                        Array.update (memopad, n, SOME r);
                        r
                     end
            else
               factorial n
   in
      factorial 20 ;
      factorial 15
   end
end
```

上面这一段 ML 代码中，采用数组结构的函数记忆簿，可缓存固定数目（20）的函数计算结果。在计算出 20！后，紧接着需要计算 15！时，可直接从函数记忆簿 memopad 得到 15！的结果。注意，这里的 factorial 和是互递归（*mutually recursive*）定义的函数。

惰性求值（*lazy evaluation*）技术是对延迟求值（*delayed evaluation*）的改进，要求进一步解决好表达式重复求值的问题。在函数式语言中，可以通过传递算壳（*pass by thunk*）与函数记忆簿（*memoization*）相结合的方式实现惰性求值，这种函数调用机制也称为按需调用（*call-by-need*）。

通常，惰性求值或按需调用中，一个表达式最多进行一次求值，即要么根本不会求值（它不被需要），要么它会进行一次求值（在被需要时，可能不止需要一次）。

惰性求值的具体实现，可以由程序员手工编写，也可以由编译器来完成。有兴趣的读者可以在[12]找到精心设计的用来实现惰性求值的代码片段。

从编译器角度，函数记忆簿的设计是实现惰性求值的重要一环。除了正确的实现，还需要在多个方面进行权衡，比如高效的数据结构和查询算法，记忆簿的大小（上限），缓存的内容、替换算法与命中率，总体性能收益，以及存储开销等许多因素。

非严格的纯函数式语言，如 Hascal，其编译器支持惰性求值。标准 ML 是严格的纯函数式语言，其语言规范中不包含惰性求值，然而，它也存在可支持惰性求值的编译器（如 SML/NJ）。

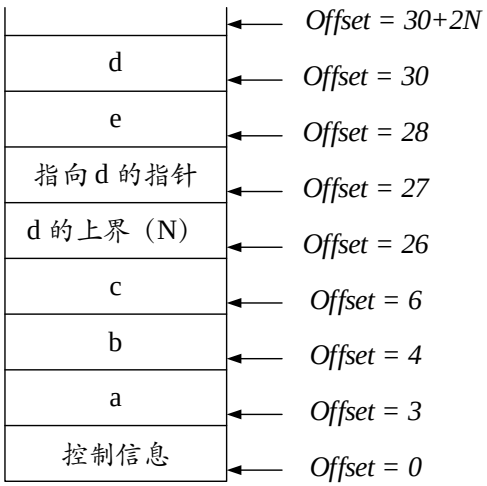
练习

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右图所示。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;
    …
}
```



函数p的活动记录中，依次包括：实际参数a（int类型的对象）占1个单元，3个控制单元（静态链、动态链和返回地址）共占用3个数据单元，局部变量 b（float 类型的对象）占2个单元，以及float类型的数组变量c的各个分量（每个分量各占2个单元），等等。

试指出函数p中访问d[i]（$0 \le i < N$）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]（$0 \le i < N$）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，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？

5. 思考与讨论：假设当前被调用的函数（callee）是叶子函数（leaf function），即它不会再调用其他函数，则与普通函数相比，调用序列可以进行哪些化简？

6. 思考与讨论：如果一个语言不允许递归调用，是否可以不设置栈式动态存储区？若可以，试对比设置于不设置两种方式的利弊。

7. 思考与讨论：试对比各类参数方式的机理，以及它们各自的适用场景。

8. 简述什么是垃圾（*garbage*）以及垃圾回收（*garbage collection*）的作用。

9. 简述垃圾回收机制中更变程序（*mutator*）的作用。

10. 对于垃圾回收算法，说出引用计数（*reference counting*）法和基于跟踪的垃圾回收（*trace-based garbage collection*）方法之间的异同，并简要对比二者的利弊。

11. 充分理解标记-清除法（*mark-and-sweep*）和拷贝回收（*copying collection*）两类最基本的基于跟踪的垃圾回收方法，比较二者的主要差异。

12. 充分理解分代回收（*generational collection*）和增量回收（*incremental collection*）

两类追求短停顿（*short-pause*）效果的垃圾回收方法，比较它们的主要差异和利弊。结合两类方法中的典型算法来讨论。

13. 思考与讨论：针对本章介绍的各种垃圾回收算法，相应的更变程序（*mutator*）分别如何设计？

14. 下面是某简单面向对象语言（类Java）的一个程序片段（同第5章的Prog2）：

```
class Computer {
    int cpu;
    void Crash(int numTimes) {
        int i;
        for (i = 0; i < numTimes; i = i + 1)
            Print("sad\n");
    }
}

class Mac extends Computer {
    int mouse;
    void Crash(int numTimes) {
        Print("ack!");
    }
}

class Main {
    static void main() {
        class Mac powerbook;
        powerbook = new Mac();
        powerbook.Crash(2);
    }
}
```

假设采用7.3节介绍的折衷的对象存储组织方式。

（1）若是创建各个类（Computer，Mac，及Main）的实例对象，试描述这些对象的存储组织（参考图33）。

（2）在进行过各类初始化后开始执行主函数，执行完语句 powerbook = new Mac() 时，与执行前相比栈区和堆区的数据信息有什么变化？概要叙述这些信息的具体内容（不必给出每项内容的相对位置和大小等信息）。

（3）在执行 powerbook.Crash(2) 时，如何找到方法Crash的代码位置？

15. 一些面向对象语言支持类成员测试，如Java中的表达式x isinstanceof C可用来判别对象x是否为类C的实例。若x直接由类C创建（如通过new函数），则x是C的实例。同时，若x是D的实例，且D继承C，则x也是C的实例。试基于7.3节介绍的折衷的对象存储组织方式，尝试一种对虚表结构进行扩展的方案，使其可用于实现对类成员测试的支持。

16. 设有如下类 ML 程序片段：

```
let
    type intfun = int -> int

    fun sqr (n: int) : int = n*n

    fun coSqr (k: int) : intfun =
        let fun h (y: int) : int = k*sqr(y)
    in h
    end

    fun coQuar (k: int) : intfun = =
        let fun g (x: int) : int = sqr ( coSqr (k) (x))
        in g
        end
in
   val coQuar (3) (5)
end
```

试参考 8.3 中关于[5]介绍的技术以及相关记号，设计一种可行的闭包存储组织方式，并给出：

（1）主函数开始执行后，先调用 coQuar (3)，当进入 coQuar 函数的内部时，两个函数（coQuar 和顶层函数 main）的当前栈帧以及堆中的闭包和堆式活动记录（逃逸变量记录）相关内容；

（2）函数 coSqr 首次激活时，当前的栈帧以及堆中的闭包和堆式活动记录相关内容；

（3）函数 sqr 首次激活时，当前的栈帧以及堆中的闭包和堆式活动记录相关内容；

（4）上述这些函数（sqr、coSqr 和 coQuar）依次返回后，函数 main 的栈帧以及堆中所分配的相关内容（分别给出）；

（5）在 coQuar (3) (5)执行完之后，main 的栈帧以及当前通过引用可达的堆中相关内容。

17. 思考与讨论：在参考8.4节内容基础上，进一步深入讨论尾调用（*tail call*）和尾递归（*tail recursive*）调用，它们对于函数/过程活动记录以及调用序列可能带来哪些优化？

18. 简述延迟求值、惰性求值、按名调用、按需调用以及严格和非严格的函数式语言等术语的含义，区别它们之间的异同。

19. 简述算壳（*thunks*）以及函数记忆簿（*memoization*）技术的本质内涵。

20. 思考与讨论：基于函数记忆簿实现惰性计算，有时可以带来显著的性能提升，而有时反而会使总体性能下降。试在调研工作基础上，进一步深入讨论该问题。

参考文献

[1] RISC-V ABIs Specification, Version 1.0. https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0/riscv-abi.pdf .

[2] R. E. Bryant, D. R. O'Hallaron. Computer.Systems: A.Programmer's Perspective, Second Edition, Prentice Hall，2011.

[3] R. E. Bryant, D. R. O'Hallaron. Computer Systems: A Programmer's Perspective, Third Edition, Pearson Education, Inc. , 2015.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers：Principles, Techniques, and Tools, Addison Wesley, 2007.

[5] Andrew W. Appel, Maia Ginsburg, Modern Compiler Implementation in C, Andrew W.Appel, Maia Ginsburg，Cambridge University Press, 1998.

[6] RISC-V ELF psABI Specification. https://github.com/riscv/riscv-elf-psabi-doc/.

[7] System V Application Binary Interface，MIPS® RISC Processor Supplement, 3rd Edition, The Santa Cruz Operation, Inc.

[8] Richard Jones and Rafael Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, 1996.

[9] Charles N. Fischer, Ronald K.Cytron, Richard J. LeBlanc, Crafting a Compiler, Jr., 2010.

[10] Charles N. Fischer, Ronald K.Cytron, Richard J. LeBlanc, Crafting a Compiler with C, Jr., 1991.

[11] A. Church, The Calculi of Lambda Conversion, Princeton University Press., 1941.

[12] Robert Harper , Programming in Standard ML , Carnegie Mellon University , Spring Semester, 2011,available at https://www.cs.cmu.edu/~rwh/isml/book.pdf.