第7章　运行时刻环境

编译器必须准确地实现源程序语言中包含的各个抽象概念。这些抽象概念通常包括我们在1.6节中曾经讨论过的那些概念，如名字、作用域、绑定、数据类型、运算符、过程、参数以及控制流构造。编译器还必须和操作系统以及其他系统软件协作，在目标机上支持这些抽象概念。

为了做到这一点，编译器创建并管理一个运行时刻环境(run-time environment)，它编译得到的目标程序就运行在这个环境中。这个环境处理很多事务，包括为在源程序中命名的对象分配和安排存储位置，确定目标程序访问变量时使用的机制，过程间的连接，参数传递机制，以及与操作系统、输入输出设备及其他程序的接口。

本章的两个主题是存储位置的分配和对变量及数据的访问。我们将详细地讨论存储管理，包括栈分配、堆管理和垃圾回收。我们将在下一章中介绍为多种常见语言构造生成目标代码的技术。

7.1　存储组织

从编译器编写者的角度来看，正在执行的目标程序在它自己的逻辑地址空间内运行，其中每个程序值都在这个空间中有一个地址。对这个逻辑地址空间的管理和组织是由编译器、操作系统和目标机共同完成的。操作系统将逻辑地址映射为物理地址，而物理地址对整个内存空间编址。

一个目标程序在逻辑地址空间的运行时刻映像包含数据区和代码区，如图7-1所示。某个语言(比如C++)在某个操作系统(比如Linux)上的编译器可能按照这种方式划分存储空间。

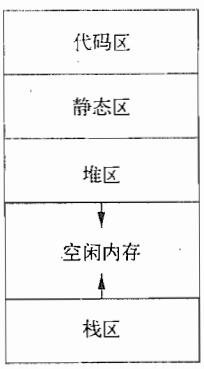


图7-1　运行时刻内存被划分成代码区和数据区的典型方式

在本书中，我们假定运行时刻存储是以多个连续字节块的方式出现的，其中字节是内存的最小编址单元。一个字节包含8个二进制位，4个字节构成一个机器字。多字节数据对象总是存储在一段连续的字节中，并把第一个字节作为它的地址。

第6章中讨论过，一个名字所需要的存储空间大小是由它的类型决定的。基本数据类型，比如字符、整数或浮点数，可以存储在整数个字节中。聚合类型(比如数组或结构)的存储空间大小必须足以存放这个类型的所有分量。

数据对象的存储布局受目标机的寻址约束的影响很大。在很多机器中，执行整数加法的指令可能要求整数是对齐的，也就是说这些数必须被放在一个能够被4整除的地址上。尽管在C语言或者类似的语言中一个有10个字符的数组只需要能够存放10个字符的空间，但是编译器可能为了对齐而给它分配12个字节，其中的两个字节未使用。因为对齐的原因而产生的闲置空间称为补白(padding)。如果空间比较紧张，编译器可能会压缩数据以消除补白。但是，在运行时刻可能需要额外的指令来定位被压缩数据，使得机器在操作这些数据时就好像它们是对齐的。

生成的目标代码的大小在编译时刻就已经固定下来了，因此编译器可以将可执行目标代码

放在一个静态确定的区域：*代码区*。这个区通常位于存储的低端。类似地，程序的某些数据对象的大小可以在编译时刻知道，它们可以被放置在另一个称为静态区的区域中，该区域可以被静态确定。放置在这个区域的数据对象包括全局常量和编译器产生的数据，比如用于支持垃圾回收的信息等。之所以要将尽可能多的数据对象进行静态分配，是因为这些对象的地址可以被编译到目标代码中。在 Fortran 的早期版本中，所有数据对象都可以进行静态分配。

为了将运行时刻的空间利用率最大化，另外两个区域——栈和堆被放在剩余地址空间的相对两端。这些区域是动态的，它们的大小会随着程序运行而改变。这两个区域根据需要向对方增长。栈区用来存放称为活动记录的数据结构，这些活动记录在函数调用过程中生成。

在实践中，栈向较低地址方向增长，而堆向较高地址方向增长。然而，在本章及下一章中，我们将假定栈向较高地址方向增长，以便我们能够在所有例子中方便地使用正的偏移量。

我们将在下一节看到，一个活动记录用于在一个过程调用发生时记录有关机器状态的信息，例如程序计数器和机器寄存器的值。当控制从该次调用返回时，相关寄存器的值被恢复，程序计数器被设置成指向紧跟在这次调用之后的点，然后调用过程的活动就可以重新开始。如果一个数据对象的生命周期包含在一次活动的生命期中，那么该对象可以和其他关于该活动的信息一起被分配到栈区上。

很多程序设计语言支持程序员通过程序控制人工分配和回收数据对象。例如，C 语言中的 `malloc` 和 `free` 函数可以用来获取及释放任意存储块。堆区被用来管理这种具有长生命周期的数据。7.4 节中将讨论多种可以用来维护堆区的存储管理算法。

静态和动态存储分配

数据在运行时刻环境中的内存位置的布局及分配是存储管理的关键问题。这些问题需要谨慎对待，因为程序文本中的同一个名字可能在运行时刻指向不同的存储位置。两个形容词*静态*(static)和*动态*(dynamic)分别表示编译时刻和运行时刻。如果编译器只需要通过观察程序文本即可做出某个存储分配决定，而不需要观察该程序在运行时做了什么，我们就认为这个存储分配决定是静态的。反过来，如果只有在程序运行时才能做出决定，那么这个决定就是动态的。很多编译器使用下列两种策略的某种组合进行动态存储分配：

1）*栈式存储*。一个过程的局部名字在栈中分配空间。我们将从 7.2 节开始讨论“运行时刻栈”。这种栈支持通常的过程调用/返回策略。

2）*堆存储*。有些数据的生命周期要比创造它的某次过程调用更长，这些数据通常被分配在一个可复用存储的“堆”中。我们将从 7.4 节开始讨论堆管理。堆是虚拟内存的一个区域，它允许对象或其他数据元素在被创建时获得存储空间，并在数据变得无效时释放该存储空间。

为了支持堆区管理，通过“垃圾回收”使得运行时刻系统能够检测出无用的数据元素，即使程序员没有显式地释放它们的空间，运行时刻系统也能够复用这些存储。尽管自动垃圾回收机制是一个难以高效完成的操作，但它仍是很多现代程序设计语言的一个重要特征。对于某些语言来说，垃圾回收甚至是不可能完成的。

7.2 空间的栈式分配

有些语言使用过程、函数或方法作为用户自定义动作的单元，几乎所有针对这些语言的编译器都把它们的(至少一部分的)运行时刻存储按照一个栈进行管理。每当一个过程[⊖]被调用时，

⊖ 请回忆一下，“过程”这个词是函数、过程、方法和子例程的统称。

用于存放该过程的局部变量的空间被压入栈；当这个过程结束时，该空间被弹出这个栈。我们将看到，这种安排不仅允许活跃时段不交叠的多个过程调用之间共享空间，而且允许我们以如下方式为一个过程编译代码：它的非局部变量的相对地址总是固定的，和过程调用的序列无关。

7.2.1 活动树

假如过程调用(或者说过程的*活动*)在时间上不是嵌套的，那么栈式分配就不可行了。下面的例子说明了过程调用的嵌套情形。

例7.1 图7-2给出了一个程序的概要。该程序将9个整数读入到一个数组 a，并使用递归的快速排序算法对这些整数排序。

```
int a[11];
void readArray() { /* 将9个整数读入到a[1],.... a[9]中。*/
    int i;
    ...
}
int partition(int m, int n) {
    /* 选择一个分割值 v，划分 a[m..n]，
       使得 a[m..p-1] 小于 v, a[p] = v，
       并且 a[p+1..n] 大于等于 v。返回 p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

图7-2 一个快速排序程序的概要

程序的主函数有三个任务。它调用 *readArray*，设定上下限值，然后在整个数组之上调用 *quicksort*。图7-3给出了可能在程序的某次执行中得到的调用序列。在这次执行中，对 *partition*(1, 9)的调用返回4，因此 $a[1]$ 到 $a[3]$ 存放了小于被选定的分割值 v 的元素，而较大的元素被存放在 $a[5]$ 到 $a[9]$。 □

在这个例子中，过程活动在时间上是嵌套的，在一般情况下也是这样。如果过程 p 的一个活动调用了过程 q，那么 q 的该次活动必定在 p 的活动结束之前结束。有三种常见的情况：

1) q 的该次活动正常结束，那么基本上在任何语言中，控制流从 p 中调用 q 的点之后继续。

2) q 的该次活动(或 q 调用的某个过程)直接或间接地中止了，也就是说不能再继续执行了。在这种情况下，q 和 p 同时结束。

3) q 的该次活动因为 q 不能处理的某个异常而结束。过程 p 可能会处理这个异常。此时 q 的活动已经结束而 p 的活动继续执行，尽管 p 的活动不一定从调用 q 的点开始。如果 p 不能处理这个异常，那么 p 的活动和 q 的活动一起结束。一般来说某个过程的尚未结束的活动将处理这个异常。

因此，我们可以用一棵树来表示在整个程序运行期间的所有过程的活动，这棵树称为*活动树*(activation tree)。树中的每个结点对应于一个活动，根结点是启动程序执行的 main 过程的活动。

在表示过程 p 的某个活动的结点上，其子结点对应于被 p 的这次活动调用的各个过程的活动。我们按照这些活动被调用的顺序，自左向右地显示它们。值得注意的是，一个子结点必须在其右兄弟结点的活动开始之前结束。

一种快速排序

图 7-2 中的快速排序程序概要使用了两个辅助函数 *readArray* 和 *partition*。函数 *readArray* 仅用于将数据加载到数组 a 中。数组 a 的第一个和最后一个元素没有用于存放输入数据，而是用于存放主函数中设定的“限值”。我们假定 $a[0]$ 被设为小于所有可能输入数据值的值，而 $a[10]$ 被设为大于所有数据值的值。

函数 *partition* 对数组中第 m 个元素到第 n 个元素的部分进行分割，使得 $a[m]$ 到 $a[n]$ 之间的小元素存放在前面，而大的元素存放在尾部，但是这两组内部不一定是排好序的。我们将不会探究 *partition* 的工作方式，只需要知道这个过程要求前面提到的上下限值必须存在。图 9-1 中的更加详细的代码给出了实现 *partition* 的一种可能的算法。

递归过程 *quicksort* 首先确定它是否需要对多个数组元素进行排序。请注意，单个元素总是“有序的”，因此在这种情况下 *quicksort* 不需要做任何事。如果有多个元素需要排序，*quicksort* 首先调用 *partition*。这次调用会返回一个数组下标 i，它是小元素和大元素之间的分界线。然后通过递归调用 *quicksort* 对这两组元素排序。

例 7.2 图 7-3 给出了一个调用和返回序列，而图 7-4 中显示了一棵完成这个调用/返回序列的可能的活动树。各个函数用它的函数名的第一个字母表示。请记住，这个树只代表了一种可能性，因为后续调用的参数会有不同，并且各个分支上的调用次数会受到 *partition* 的返回值的影响。 □

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

图 7-3 图 7-2 中程序的可能的活动序列

在活动树和程序行为之间存在下列多种有用的对应关系，正是因为这些关系使我们可以使用运行时刻栈：

1）过程调用的序列和活动树的前序遍历相对应。

2）过程返回的序列和活动树的后序遍历相对应。

3）假定控制流位于某个过程的特定活动中，且该过程活动对应于活动树上的某个结点 N。那么当前尚未结束的（即活跃的）活动就是结点 N 及其祖先结点对应的活动。这些活动被调用的顺序就是它们在从根结点到 N 的路径上的出现顺序。这些活动将按照这个顺序的反序返回。

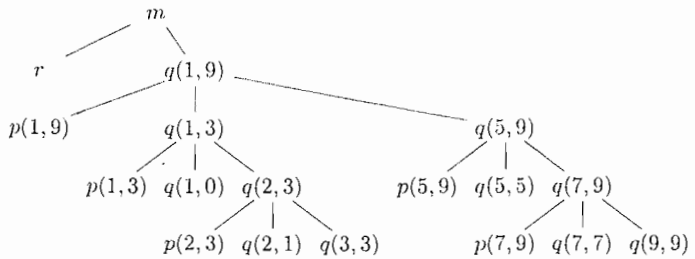


图 7-4 表示 *quicksort* 的某次运行中的调用的活动树

7.2.2 活动记录

过程调用和返回通常由一个称为控制栈(control stack)的运行时刻栈进行管理。每个活跃的活动都有一个位于这个控制栈中的活动记录(activation record，有时也称为帧(frame))。活动树的根位于栈底，栈中全部活动记录的序列对应于在活动树中到达当前控制所在的活动结点的路径。程序控制所在的活动的记录位于栈顶。

例7.3 如果当前的控制位于图7-4的树中的活动 $q(2,3)$ 上，那么 $q(2,3)$ 对应的活动记录在控制栈的顶端。紧跟在下面的是 $q(1,3)$ 的活动记录，即树中 $q(2,3)$ 的父结点。再下面是 $q(1,9)$ 的活动记录。栈的底端是主函数 m 的活动记录，也就是活动树的根。 □

按照惯例，我们在画控制栈的时候将把栈底画在栈顶之上。因此在一个活动记录中出现在页面最下方的元素实际上最靠近栈顶。

根据所实现语言的不同，其活动记录的内容也有所不同。这里列举出可能出现在一个活动记录中的各种类型的数据(图7-5列出了这些元素以及它们之间的可能顺序)：

1) 临时值。比如当表达式求值过程中产生的中间结果无法存放在寄存器中时，就会生成这些临时值。

2) 对应于这个活动记录的过程的局部数据。

3) 保存的机器状态，其中包括对此过程的此次调用之前的机器状态信息。这些信息通常包括返回地址(程序计数器的值，被调用过程必须返回到该值所指位置)和一些寄存器中的内容(调用过程会使用这些内容，被调用过程必须在返回时恢复这些内容)。

4) 一个“访问链”。当被调用过程需要其他地方(比如另一个活动记录)的某个数据时需要使用访问链进行定位。访问链将在7.3.5节中讨论。

5) 一个控制链(control link)，指向调用者的活动记录。

6) 当被调用函数有返回值时，要有一个用于存放这个返回值的空间。不是所有的被调用过程都有返回值，即使有，我们也可能倾向于将该值放到一个寄存器中以提高效率。

7) 调用过程使用的实在参数(actual parameter)。这些值通常将尽可能地放在寄存器中，而不是放在活动记录中，因为放在寄存器中会得到更好的效率。然而，我们仍然为它们预留了相应的空间，使得我们的活动记录具有完全的通用性。

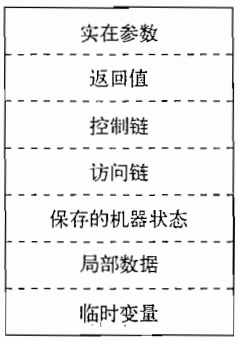
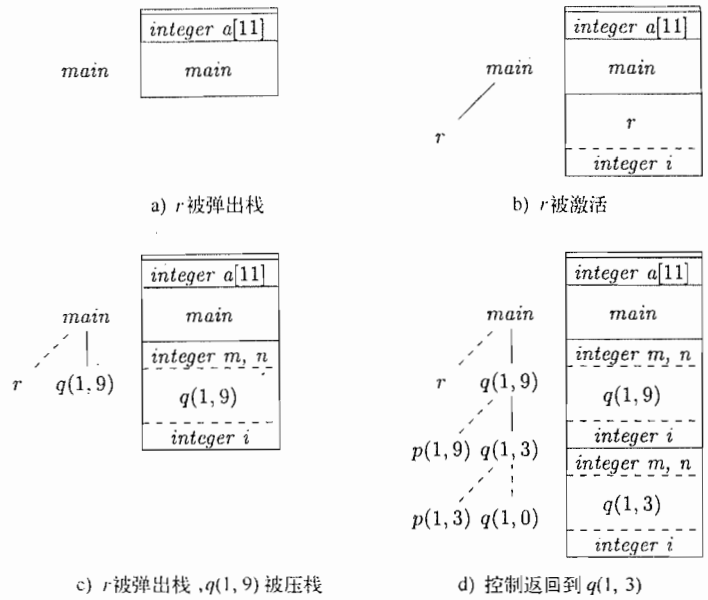


图7-5 一个概括性的活动记录

例7.4 图7-6给出了当控制流在图7-4所示的活动树中运行时运行时刻栈的多个快照。这些不完整的树中的虚线指向已经结束的活动。程序的执行随着过程 *main* 的一次活动而开始。因为数组 a 是全局的，在此之前已经为 a 分配了存储空间，如图7-6a所示。

当控制到达 *main* 的函数体中的第一个函数调用时，过程 r 被激活，它的活动记录被压入栈中(参见图7-6b)。r 的活动记录包含了局部变量 i 的空间。请记住栈顶是在图的下方。当控制从这次活动中返回时，它的记录被弹出栈，栈中只留下 *main* 的记录。

然后控制到达实在参数为1和9的对 q(即快速排序)的调用，这次调用的活动记录被放置在栈顶，如图7-6c所示。q 的活动记录中包括了参数 m 和 n 以及局部变量 i 的空间。它们按照图7-5所示的通用布局放置。请注意，曾经被 r 的调用使用的空间被复用了。函数调用 $q(1,9)$ 没有任何方法找到 r 的局部数据。当 $q(1,9)$ 返回时，栈中再次只剩下了 *main* 的活动记录。



a) r被弹出栈　b) r被激活

c) r被弹出栈，$q(1,9)$被压栈　d) 控制返回到 $q(1,3)$

图 7-6　向下增长的活动记录栈

在图 7-6 的最后两个快照之间发生了多个活动。$q(1,9)$递归地调用了 $q(1,3)$。在 $q(1,3)$的生命期内，活动 $p(1,3)$和 $q(1,0)$开始执行并结束，栈顶只留下了活动记录 $q(1,3)$（见图 7-6d）。注意，当一个过程是递归的时，常常会有该过程的多个活动记录同时出现在栈中。 □

7.2.3　调用代码序列

实现过程调用的代码段称为*调用代码序列*（calling sequence）。这个代码序列为一个活动记录在栈中分配空间，并在此记录的字段中填写信息。*返回代码序列*（return sequence）是一段类似的代码，它恢复机器状态，使得调用过程能够在调用结束之后继续执行。

即使对于同一种语言，不同实现中的调用代码序列和活动记录的布局也可能千差万别。一个调用代码序列中的代码通常被分割到调用过程（调用者）和被调用过程（被调用者）中。在分割运行时刻任务时，调用者和被调用者之间不存在明确界限。源语言、目标机器、操作系统会提出某些要求，使得能够选择出一种较好的分割方案。总的来说，如果一个过程在 n 个不同点上被调用，分配给调用者的那部分调用代码序列会被生成 n 次。然而，分配给被调用者的部分只被生成一次。因此，我们期望把调用代码序列中尽可能多的部分放在被调用者中——能够根据被调用者的信息确定的部分都应该放到被调用者中。不过，我们将看到，被调用者不可能知道所有的事情。

在设计调用代码序列和活动记录的布局时，可以使用下列的设计原则：

1）在调用者和被调用者之间传递的值一般被放在被调用者的活动记录的开始位置，因此它们尽可能地靠近调用者的活动记录。这样做的动机是，调用者能够计算该次调用的实在参数的值并将它放在自身活动记录的顶部，而不用创建整个被调用者的活动记录，甚至不用知道该记录的布局。不仅如此，它还使得语言可以使用参数个数或类型可变的过程，比如 C 语言中的 `printf` 函数。被调用者知道应该把返回值放置在相对于它自己的活动记录的哪个位置。同时，不管有多少个参数，它们都将在栈中顺序地出现在该位置之下。

2）固定长度的项被放置在中间位置。根据图 7-5，这样的项通常包括控制链、访问链和机器状态字段。如果每次调用中保存的机器状态的成分相同，那么可以使用同一段代码来保存和恢

复每次调用的数据。不仅如此，如果我们将机器状态信息标准化，那么当错误发生时，诸如调试器这样的程序将可以更容易地将栈中的内容解码。

3）那些在早期不知道大小的项将被放置在活动记录的尾部。大部分局部变量具有固定的长度，编译器通过检查该变量的类型就可以确定其长度。然而，有些局部变量的大小只有在程序运行时才能确定。最常见的例子是动态数组，数组大小根据被调用者的某个参数决定。另外，临时量所需空间的大小通常依赖于代码生成阶段能够将多少临时变量放在寄存器中。因此，虽然编译器最终可以知道临时变量所需要的空间，但在刚开始生成中间代码时可能并不知道该空间的大小。

4）我们必须小心地确定栈顶指针所指的位置。一个常用的方法是让这个指针指向活动记录中固定长度字段的末端。这样，固定长度的数据就可以通过固定的相对于栈顶指针的偏移量来访问，而中间代码生成器知道这些偏移量。使用这种方法的后果是活动记录中的变长域实际上位于栈顶“之上”。它们的偏移量需要在运行时刻进行计算，但是它们仍然可以基于栈顶指针进行访问，但是偏移量为正。

图 7-7 给出了调用者和被调用者如何合作管理调用栈的一个例子。寄存器 *top_sp* 指向当前的顶层活动记录中机器状态字段的末端。调用者知道这个位于被调用者的活动记录中的位置。因此，调用者可以负责在控制转向被调用者之前设定 *top_sp* 的值。这个调用代码序列以及它在调用者和被调用者之间的划分描述如下：

1）调用者计算实在参数的值。

2）调用者将返回地址和原来的 *top_sp* 值存放到被调用者的活动记录中。然后，调用者增加 *top_sp* 的值，使之指向图 7-7 所示的位置。也就是说，*top_sp* 越过了调用者的局部数据和临时变量以及被调用者的参数和机器状态字段。

3）被调用者保存寄存器值和其他状态信息。

4）被调用者初始化其局部数据并开始执行。

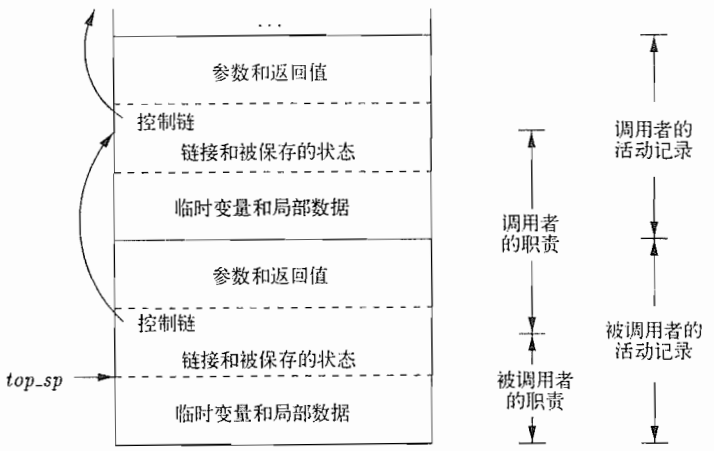


图 7-7 调用者和被调用者之间的任务划分

一个与此匹配的返回代码序列如下：

1）如图 7-5 所示，被调用者将返回值放到与参数相邻的位置。

2）使用机器状态字段中的信息，被调用者恢复 *top_sp* 和其他寄存器，然后跳转到由调用者放在机器状态字段中的返回地址。

3）尽管 *top_sp* 已经被减小，但调用者仍然知道返回值相对于当前 *top_sp* 值的位置。因此，调

用者可以使用那个返回值。

上面的调用和返回代码序列支持使用不同数量的参数来调用同一个被调用程序(就像 C 语言中的 printf 函数那样)。请注意,在编译时刻,调用者的目标代码知道它向被调用者提供的参数的数量和类型。因此,调用者知道参数区域的大小。然而,被调用者的目标代码必须还能处理其他调用,因此,它要等到被调用时再检查相应的参数字段。使用图 7-7 中的组织方法,描述参数的信息必定放置在状态字段的相邻位置,因此被调用者可以找到这个信息。例如,在 C 语言的 printf 函数中,第一个参数描述了其余的参数,因此一旦找到了第一个参数,调用者就可以找到所有的其他参数。

7.2.4 栈中的变长数据

运行时刻存储管理系统必须频繁地处理某些数据对象的空间分配。这些数据对象的大小在编译时刻未知,但是它们是这个过程的局部对象,因而可以被分配在运行时刻栈中。在现代程序设计语言中,在编译时刻不能决定大小的对象将被分配在堆区。堆区的存储结构将在 7.4 节中讨论。不过,也可以将未知大小的对象、数组以及其他结构分配在栈中。我们在这里将讨论如何进行这种分配。尽可能将对象放置在栈区的原因是我们可以避免对它们的空间进行垃圾回收,也就减少了相应的开销。注意,只有一个数据对象局限于某个过程,且当此过程结束时它变得不可访问,才可以使用栈为这个对象分配空间。

为变长数组(即其大小依赖于被调用过程的一个或多个参数值的数组)分配空间的一个常用策略如图 7-8 所示。同样的方案可以用于任何类型的对象的分配,只要它们对被调用的过程而言是局部的,并且其大小依赖于该次调用的参数即可。

在图 7-8 中,过程 p 有三个局部数组,我们假设它们的大小无法在编译时刻确定。尽管这些数组的存储出现在栈中,它们并不是 p 的活动记录的一部分。只有指向各个数组的开始位置的指针存放在活动记录中。因此当 p 执行时,这些指针的位置相对于栈顶指针的偏移量是已知的,因而目标代码可以通过这些指针访问数组元素。

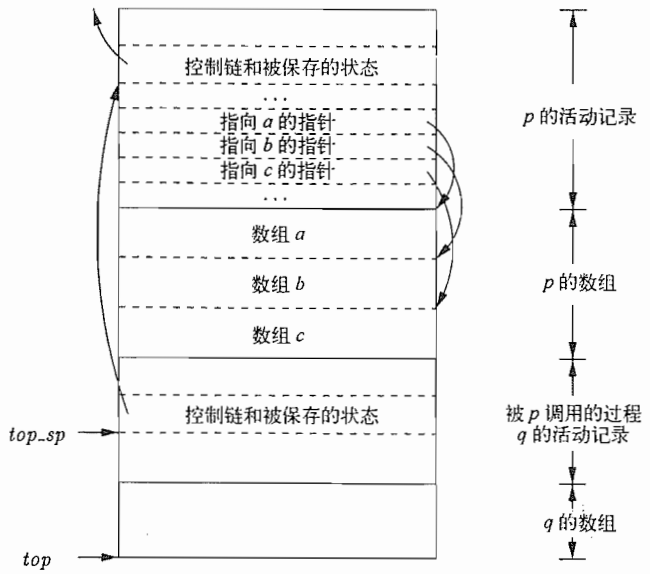


图 7-8 访问动态分配的数组

图 7-8 中还给出了一个被 p 调用的过程 q 的活动记录。q 的这个活动记录从 p 的数组之后开始，q 的所有变长数组被分配在 q 的活动记录之外。

对栈中数据的访问通过指针 *top* 和 *top_sp* 完成。这里，*top* 标记了实际的栈顶位置，它指向下一个活动记录将开始的位置，第二个指针 *top_sp* 用来找到顶层活动记录的局部的定长字段。为了和图 7-7 保持一致，我们将假定 *top_sp* 指向机器状态字段的末端。在图 7-8 中，*top_sp* 指向 q 的活动记录的机器状态字段的末端。从那里，我们可以找到 q 的控制链字段，根据这个字段我们可以知道当 p 位于栈顶时，*top_sp* 所指的 p 的活动记录中的位置。

重新设置 *top* 和 *top_sp* 所指位置的代码可以在编译时刻生成。这些代码根据将根据在运行时刻获知的记录大小来计算 *top* 和 *top_sp* 的新值。当 q 返回时，可以根据 q 的活动记录中的被保存的控制链来恢复 *top_sp* 的值。*top* 的新值等于(未经恢复的原来的) *top_sp* 值减去 q 的活动记录中机器状态、控制链、访问链、返回值、参数字段(如图 7-5 所示)的总长度。调用者可以在编译时刻知道这个长度，尽管当调用参数的个数可变时，它仍取决于调用者(如果调用 q 的参数个数可变)。

7.2.5 7.2 节的练习

练习 7.2.1：假设图 7-2 中的程序使用如下的 *partition* 函数：该函数总是将 $a[m]$ 作为分割值 v。同时假设在对数组 $a[m],\cdots,a[n]$ 重新排序时总是尽量保存原来的顺序。也就是说，首先是以原顺序保持所有小于 v 的元素，然后保存所有等于 v 的元素，最后按原来顺序保存所有大于 v 的元素。

1) 画出对数字 9、8、7、6、5、4、3、2、1 进行排序时的活动树。

2) 同时在栈中出现的活动记录最多有多少个?

练习 7.2.2：当初始顺序为 1、3、5、7、9、2、4、6、8 时，重复练习 7.2.1。

练习 7.2.3：图 7-9 中是递归计算 Fiabonacci 数列的 C 语言代码。假设 f 的活动记录按顺序包含下列元素：(返回值，参数 n，局部变量 s，局部变量 t)。通常在活动记录中还会有其他元素。下面的问题假设初始调用是 $f(5)$。

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

图 7-9 练习 7.2.3 的 Fibonacci 程序

1) 给出完整的活动树。

2) 当第 1 个 $f(1)$ 调用即将返回时，运行时刻栈和其中的活动记录是什么样子的?

! 3) 当第 5 个 $f(1)$ 调用即将返回时，运行时刻栈和其中的活动记录是什么样子的?

练习 7.2.4：下面是两个 C 语言函数 f 和 g 的概述：

```
int f(int x) { int i; ··· return i+1; ··· }
int g(int y) { int j; ··· f(j+1) ··· }
```

也就是说，函数 g 调用 f。画出在 g 调用 f 而 f 即将返回时，运行时刻栈中从 g 的活动记录开始的顶端部分。你可以只考虑返回值、参数、控制链以及存放局部数据的空间。你不用考虑存放的机器状态，也不用考虑没有在代码中显示的局部值和临时值。但是你应该指出：

1) 哪个函数在栈中为各个元素创建了所使用的空间?

2) 哪个函数写入了各个元素的值?

3) 这些元素属于哪个活动记录?

练习 7.2.5：在一个通过引用传递参数的语言中，有一个函数 $f(x, y)$ 完成下面的计算：

```
x = x + 1; y = y + 2; return x+y;
```

如果将 a 赋值为 3，然后调用 $f(a, a)$，那么返回值是什么?

练习 7.2.6：C 语言函数 f 的定义如下：

```
int f(int x, *py, **ppz) {
    **ppz += 1; *py += 2; x += 3; return x+**py+***ppz;
}
```

变量 a 是一个指向 b 的指针；变量 b 是一个指向 c 的指针，而 c 是一个当前值为 4 的整数变量。如果我们调用 $f(c, b, a)$，返回值是什么？

7.3 栈中非局部数据的访问

在本节中，我们将探讨过程如何访问它们的数据。尤其重要的是找到在过程 p 中被使用但又不属于 p 的数据的机制。对于那些可以在过程中声明其他过程的语言，这种访问将变得更加复杂。因此，我们首先从 C 函数这种简单情况开始，然后介绍另一种语言 ML，该语言支持嵌套的函数声明，并支持将函数看成是“一阶对象”。也就是说，函数可以将函数作为参数，并把函数当做值返回。通过修改运行时刻栈的实现方法就可以支持这种能力。我们将考虑几种可选的修改 7.2 节所述的活动记录的方法。

7.3.1 没有嵌套过程时的数据访问

在 C 系列语言中，各个变量要么在某个函数内定义，要么在所有函数之外（全局地）定义。最重要的是，不可能声明一个过程使其作用域完全位于另一个过程之内。反过来，一个全局变量 v 的作用域包含了在该变量声明之后出现的所有函数，但那些存在标识符 v 的局部定义的地方除外。在一个函数内部声明的变量的作用域就是这个函数，或者像在 1.6.3 节中讨论过的那样，如果该函数具有嵌套的语句块，这个变量的作用域可能是该函数的部分区域。

对于不允许声明嵌套过程的语言而言，变量的存储分配和访问这些变量是比较简单的：

1）全局变量被分配在静态区。这些变量的位置保持不变，并且在编译时刻可知。因此要访问当前正在运行的过程的非局部变量时，我们可以直接使用这些静态确定的地址。

2）其他变量一定是栈顶活动的局部变量。我们可以通过运行时刻栈的 *top_sp* 指针来访问这些变量。

对于全局变量进行静态分配的一个好处是，被声明的过程可以作为参数传递，也可以作为结果返回（在 C 语言中可以传递指向该函数的指针），实现这样的传递不需要对数据访问策略做出本质的改变。使用 C 语言的静态作用域规则且不允许使用嵌套过程声明时，一个过程的任何非局部变量也是所有过程的非局部变量，不管这些过程是如何被激活的。类似地，如果一个过程作为结果返回，那么任何非局部的变量都指向为该变量静态分配的存储位置。

7.3.2 和嵌套过程相关的问题

当一种语言允许嵌套地声明过程并且仍然遵循通常的静态作用域规则时，数据访问变得比较复杂。也就是说，根据 1.6.3 节中描述的针对语句块的嵌套作用域规则，一个过程能够访问另一个过程的变量，只要后一个过程的声明包含了前一过程的声明即可。其原因在于，即使在编译时刻知道 p 的声明直接嵌套在 q 之内，我们并不能由此确定它们的活动记录在运行时刻的相对位置。实际上，因为 p 或 q 或者两者都可能是递归的，在栈中可能有多个 p 和/或 q 的活动记录。

为一个内嵌过程 p 中的一个非局部名字 x 找出对应的声明是一个静态的决定过程，将块结构的静态作用域规则进行扩展就可以解决这个问题。假定 x 在一个外围过程 q 中声明。根据 p 的一个活动找到相关的 q 的活动则是一个动态的决定过程，它需要额外的有关活动的运行时刻信息。这个问题的可能解决方法之一是使用“访问链”，我们将在 7.3.5 节中介绍这个概念。

7.3.3 一个支持嵌套过程声明的语言

在C系列语言中，还有很多常见的语言不支持嵌套的过程，因此我们介绍一种支持嵌套过程的语言。在语言中支持嵌套过程的历史比较长。Alogl 60(C语言的前身之一)就具备这种能力。Algol 60语言的后继Pascal(一个一度很流行的教学语言)也支持嵌套过程。在较晚的支持嵌套过程的语言中，最有影响力的语言之一是ML。我们将通过这个语言的语法和语义进行相关介绍(要了解ML的一些有趣特征，请见"ML的更多特性"部分)。

- ML是一种函数式语言(functional language)，这意味着变量一旦被声明并初始化就不会再改变。其中只有少数几个例外，比如数组的元素可以通过特殊的函数调用改变。
- 定义变量并设定它们的不可更改的初始值的语句具有如下形式：

  `val` ⟨name⟩ = ⟨expression⟩

- 函数使用如下语法进行定义：

  `fun` ⟨name⟩ (⟨arguments⟩) = ⟨body⟩

- 我们使用下列形式的let语句来定义函数体：

  `let` ⟨list of definitions⟩ `in` ⟨statements⟩ `end`

  其中，定义(definition)通常是`val`或`fun`语句。每个这样的定义的作用域包括从该定义之后直到`in`为止的所有定义，以及直到`end`为止的所有语句。最重要的是，函数可以嵌套地定义。例如，函数p的函数体可能包括一个let语句，而该语句又包含了另一个(嵌套的)函数q的定义。类似地，q自身的函数体中也可能有函数定义，这就形成了任意深度的函数嵌套。

7.3.4 嵌套深度

对于不内嵌在任何其他过程中的过程，我们设定其*嵌套深度*(nesting depth)为1。例如，所有C函数的嵌套深度为1。然而，如果一个过程p在一个嵌套深度为i的过程中定义，那么我们设定p的嵌套深度为$i+1$。

例7.5 图7-10给出了我们连续使用的快速排序例子的一个ML程序的概要。唯一的嵌套深度为1的函数是最外层的函数*sort*。它读入一个有9个整数的数组a，并使用快速排序算法对它们进行排序。在*sort*内部的第二行上定义了数组a本身。请注意ML声明的形式。*array*的第一个参数说明我们要求该数组具有11个元素。所有的ML数组的下标都是从0开始的整数，因此这个数组与图7-2中的C语言数组a很相似。`array`的第二个参数说明数组a中的所有元素的初始值都是0。因为0是整数，选择这样的初始值使得ML编译器推断出a是一个整型数组，因此我们就不需要为a声明一个类型。

ML的更多特性

ML几乎是纯函数式的语言。除此之外，ML还具有多个令那些熟悉C及C系列语言的程序员感到惊奇的特性：

- ML支持高阶函数(higher-order function)。也就是说，一个函数可以将函数作为参数，并且能够构造并返回其他函数。而这些函数又可以将函数作为参数。从而构造出任何层次的函数。
- ML本质上没有像C中的for和while语句那样的迭代语句，而是通过递归来达到循环的效果。这种方法在一个函数式语言中是很重要的，因为我们不能改变迭代变量，比如C语言中的"`for (i=0; i<10; i++)`"的i的值。ML将会把i作为一个函数的参数，该函数将用不断增加的i值作为参数递归地调用自身，直到到达循环界限为止。

- ML 将列表和带标号的树结构作为其基本数据类型。
- ML 不需要声明变量的类型。准确地说，它在编译时刻推导出类型，并且当它不能推导出结果时就将其作为错误处理。例如，`val x = 1` 显然使得 x 具有整数类型，并且如果我们还看到 `val y = 2 * x`，那么我们就知道 y 也是一个整数。

在 *sort* 中声明的函数还有：*readArray*、*exchange* 和 *quicksort*。在第(4)行和第(6)行中，我们说明 *readArray* 和 *exchange* 都访问了数组 a。请注意，ML 中的数组访问可能违反这个语言的函数式特性。就像 C 版本的 *quicksort* 中那样，这两个函数实际上都改变了 a 中元素的值。因为这三个函数都是直接在嵌套深度为 1 的函数中定义的，所以它们的嵌套深度都是 2。

第(7)行到第(11)行给出了 *quicksort* 的一些细节。局部值 v(即分划算法的分割值)在第 8 行声明。第(9)行则给出了函数 *partition* 的定义。在第(10)行中我们指出 *partition* 访问了数组 a 和分割值 v，并且还调用了函数 *exchange*。因为 *partition* 直接在嵌套深度为 2 的函数中定义，所以其嵌套深度为 3。第(11)行表明 *quicksort* 访问变量 a 和 v 以及函数 *partition*，并递归调用其自身。

第(12)行表明最外层函数 *sort* 访问 a，并调用两个过程 *readArray* 和 *quicksort*。 □

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

图 7-10 一个使用嵌套函数声明的 ML 风格的 quicksort 版本

7.3.5 访问链

针对嵌套函数的通常的静态作用域规则的一个直接实现方法是在每个活动记录中增加一个被称为*访问链*(access link)的指针。如果过程 p 在源代码中直接嵌套在过程 q 中，那么 p 的任何活动中的访问链都指向最近的 q 的活动。请注意，q 的嵌套深度一定比 p 的嵌套深度恰巧少 1。访问链形成了一条链路，它从栈顶活动记录开始，经过嵌套深度逐步递减的活动的序列。沿着这条链路找到的活动就是其数据和对应过程可以被当前正在运行的过程访问的所有活动。

假定栈顶的过程 p 的嵌套深度是 n_p 且 p 需要访问 x，而 x 是在某个包围 p 的嵌套深度为 n_q 的过程 q 中定义的一个元素。注意，$n_q \leqslant n_p$，且仅当 p 和 q 是同一个过程时两者相等。为了找到 x，我们从位于栈顶的 p 的活动记录开始，沿着访问链进行 $n_p - n_q$ 次从一个活动记录到另一个活动记录的查找，最终我们找到了 q 的活动记录。这一定是当前出现在在栈中的最近(即最高)的 q 的活动记录。这个活动记录中包含了我们要找的元素 x。因为编译器知道活动记录的布局，所以

我们可以根据最后一个访问链找到 q 的活动记录中的某个位置，而 x 就位于和这个位置具有某个固定偏移量的位置上。

例 7.6 图 7-11 给出了图 7-10 中的函数 *sort* 在执行时可能得到的栈的序列。同以前一样，我们用函数名的第一个字母来表示函数。我们展示了某些可能在不同活动记录中出现的数据，同时显示了每个活动的访问链。在图 7-11a 中，我们看到的是 *sort* 调用 *readArray* 将输入加载到数组 a 上后再调用 *quicksort*(1, 9)对数组进行排序的情形。*quicksort*(1, 9)中的访问链指向 *sort* 的活动记录，这不是因为 *sort* 调用了 *quicksort*，而是因为在图 7-10 的程序中，*sort* 是 *qucksort* 外围的最靠近它的嵌套函数。

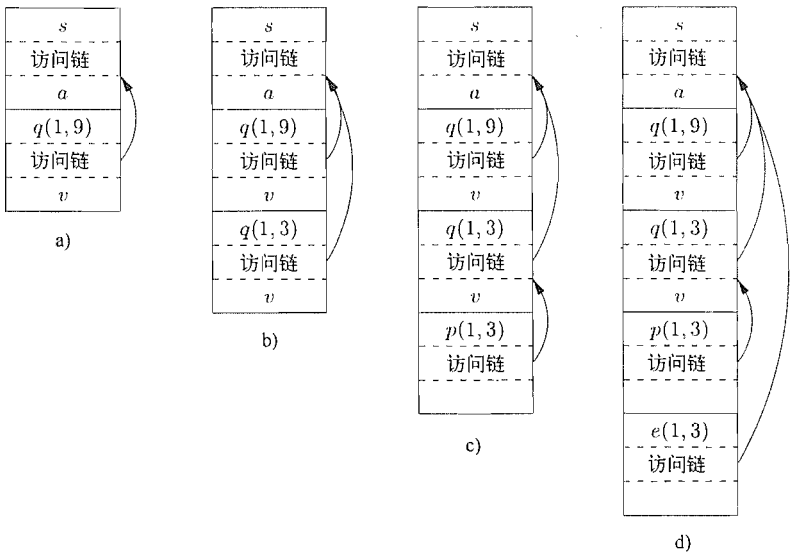


图 7-11 用来查找非局部数据的访问链

在图 7-11 所示的连续步骤中，我们看到对 *quicksort*(1, 3)的一次递归调用，然后是对 *partition* 的调用，而 *partition* 又调用 *exchange*。请注意，*quicksort*(1, 3)的访问链指向 *sort*，其理由和 *quicksort*(1, 9)的访问链指向 *sort* 的理由相同。

在图 7-11d 中，*exchange* 的访问链绕过了 *quicksort* 和 *partition* 的活动记录，因为 *exchange* 直接嵌套在 *sort* 中。这种安排是合理的，因为 *exchange* 只需要访问数组 a，而它要对换的两个元素由其参数 i 和 j 指定。 □

7.3.6 处理访问链

如何确定访问链呢？当一个过程调用另一个特定的过程，而被调用过程的名字在此次调用中明确给出，那么处理方法就很简单。更复杂的情况是当调用的对象是一个过程型参数的时候。在那种情况下，要在运行时刻才能知道被调用的是哪个过程，因此在这个调用的不同执行中，被调用过程的嵌套深度可能有所不同。因此，让我们首先考虑当一个过程 q 显式地调用过程 p 时会发生什么事情。有三种情况：

1）过程 p 的嵌套深度大于 q 的嵌套深度，那么 p 一定是直接在 q 中定义的，否则 q 调用 p 的位置就不可能位于过程名 p 的作用域内。因此，p 的嵌套深度恰好比 q 的嵌套深度大 1，而 p 的访问链一定指向 q。这个问题很简单，只需要在调用代码序列中增加一个步骤，即在 p 的访问链中放置一个指向 q 的活动记录的指针。这样的例子包括 *sort* 对 *quicksort* 的调用，该调用生成了

图 7-11a；以及 *quicksort* 对 *partition* 的调用，该调用产生了图 7-11c。

2）这个调用是递归的，也就是说 $p = q$[⊖]。那么，新的活动记录的访问链和它下面的活动记录的访问链是相同的。例如 *quicksort*(1, 9)对 *quicksort*(1, 3)的调用，该调用形成了图 7-11b。

3）p 的嵌套深度 n_p 小于 q 的嵌套深度 n_q。为了使 q 中的调用位于名字 p 的作用域中，过程 q 必定嵌套在某个过程 r 中，而 p 是一个直接在 r 中定义的过程。因此，从 q 的活动记录开始，沿着访问链经过 $n_q - n_p + 1$ 步就可以找到栈中最高的 r 的活动记录。那么，p 的访问链必须指向 r 的这个活动记录。

例 7.7 作为情况 3 的一个例子，请注意我们是如何从图 7-11c 转变为图 7-11d 的。被调用函数 *exchange* 的嵌套深度为 2，比调用函数 *partition* 的嵌套深度 3 少 1。因此，我们从 *partition* 的活动记录开始，前进 3 − 2 + 1 = 2 个访问链，这使我们从 *partition* 的活动记录到达 *quicksort*(1, 3)的活动记录，再到 *sort* 的活动记录。因此，*exchange* 的访问链指向 *sort* 的这个活动记录，这就是我们在图 7-11d 中看到的。

另一种等价的找到这个访问链的方法是沿着访问链前进 $n_q - n_p$ 步，并拷贝在那个活动记录中找到的访问链。在我们的例子中，我们将经过一步到达 *quicksort*(1, 3)的活动记录，并拷贝出它的指向 *sort* 的访问链。请注意，这个访问链对于 *exchange* 来说是正确的，尽管 *exchange* 不在 *quicksort* 的作用域中，这两个函数是嵌套在 *sort* 中的兄弟函数。 □

7.3.7 过程型参数的访问链

当一个过程 p 作为参数传递给另一个过程 q，并且 q 随后调用了这个参数(因此也就在 q 的这个活动中调用了 p)，有可能 q 并不知道 p 在程序中出现时的上下文。如果是这样，q 就不可能知道如何为 p 设定访问链。这个问题的解决办法如下：当过程被用作参数的时候，调用者除了传递过程参数的名字，同时还需要传递这个参数对应的正确的访问链。

调用者总是知道这个访问链，因为如果 p 被过程 r 当作一个实在参数传递，那么 p 必然是一个可以被 r 访问的名字。因此，r 可以像直接调用 p 那样为 p 确定访问链。也就是说，我们使用 7.3.6 节中给出的有关构造访问链的规则。

例 7.8 在图 7-12 中，我们看到一个 ML 函数 a 的大体描述。函数 a 中嵌套了函数 b 和 c。函数 b 有一个值为函数的参数 f，b 调用了这个参数。函数 c 在它自身中定义了一个函数 d，然后 c 用实在参数 d 调用了 b。

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

图 7-12 使用函数参数的 ML 程序的概要

让我们分析一下在执行 a 的时候发生了什么事情。首先，a 调用 c，因此我们在栈中将 c 的活动记录放在 a 的活动记录之上。因为 c 是直接在 a 中定义的，所以 c 的访问链指向 a 的记录。然后 c 调用 $b(d)$。调用代码序列设置了 b 的活动记录，如图 7-13a 所示。

在这个活动记录中有实在参数 d 和它的访问链，两者结合组成了 b 的活动记录中的形式参数 f 的值。请注意，c 了解 d 的信息，因为 d 是在 c 中定义的，因而 c 传递了一个指向它自己的活动记录的指针作为 d 的访问链。不管 d 在哪里定义，如果 c 在该定义的作用域内，那么必然适用 7.3.6 节中的三

⊖ ML 支持相互递归调用的函数，这种情况可以用同样的方式处理。

条规则之一，因此 c 可以给出这个访问链。

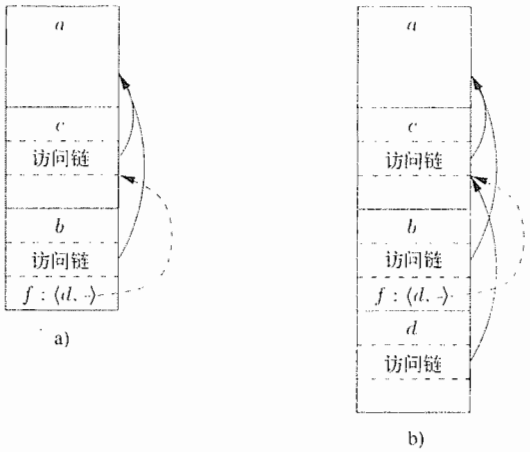


图 7-13 带有它们自己的访问链的实在参数

现在让我们看一下函数 b 所做的工作。我们知道它将在某个点上使用它的参数 f，其效果就是调用了 d。如图 7-13b 所示，d 的一个活动记录出现在栈中。应该放在这个活动记录中的正确的访问链可以在参数 f 的值中找到。该访问链指向 c 的活动记录，因为 c 就在 d 的定义的外围。请注意，b 能够正确地设置这个访问链，尽管 b 不在 c 的定义的作用域内。 □

7.3.8 显示表

使用访问链的方法来访问非局部数据的问题之一是，如果嵌套深度变大，我们就必须沿着一段很长的访问链路才能找到需要的数据。一个更高效的实现方法是使用一个称为显示表(display)的辅助数组 d，它为每个嵌套深度保存了一个指针。我们设法使得在任何时刻，指针 $d[i]$ 指向栈中最高的对应于某个嵌套深度为 i 的过程的活动记录。图 7-14 给出了一个显示表的例子。例如，在图 7-14d 中，我们看到显示表 d 的元素 $d[1]$ 保存了一个指向 *sort* 的活动记录的指针，该活动记录是最高的(也是唯一的)对应于某个嵌套深度为 1 的函数的活动记录。同时，$d[2]$ 保存了指向 *exchange* 的活动记录的指针，该记录是嵌套深度为 2 的最高活动记录。$d[3]$ 指向 *partition*，即嵌套深度为 3 的最高活动记录。

使用显示表的优势在于如果过程 p 正在运行，且它需要访问属于某个过程 q 的元素 x，那么我们只需要查看 $d[i]$ 即可。其中，i 是 q 的嵌套深度。我们沿着指针 $d[i]$ 找到 q 的活动记录，根据已知的偏移量就可以在这个活动记录中找到 x。编译器知道 i 的值，因此它可以产生代码，该代码根据 $d[i]$ 和 x 相对于 q 的活动记录顶部的偏移量来访问 x。因此，该代码不需要经过一段很长的访问链路。

为了正确地维护显示表，我们需要在新的活动记录中保存显示表条目的原来的值。如果嵌套深度为 n_p 的过程 p 被调用，并且它的活动记录不是栈中的对应于某个深度为 n_p 的过程的第一个活动记录，那么 p 的活动记录就需要保存 $d[n_p]$ 原来的值，同时 $d[n_p]$ 本身则被设定指向 p 的这个活动记录。当 p 返回且它的这个活动记录从栈中清除时，我们将 $d[n_p]$ 恢复到对 p 的这次调用之前的值。

例 7.9 图 7-14 给出了操作显示表的若干步骤。在图 7-14a 中，深度为 1 的 *sort* 调用了深度为 2 的 *quicksort*(1, 9)。*quicksort* 的活动记录中有一个用于存放 $d[2]$ 的原值的位置，图中显示为“保

存的 $d[2]$"，尽管在这个例子中因为之前没有深度为 2 的活动记录，这个指针为空。

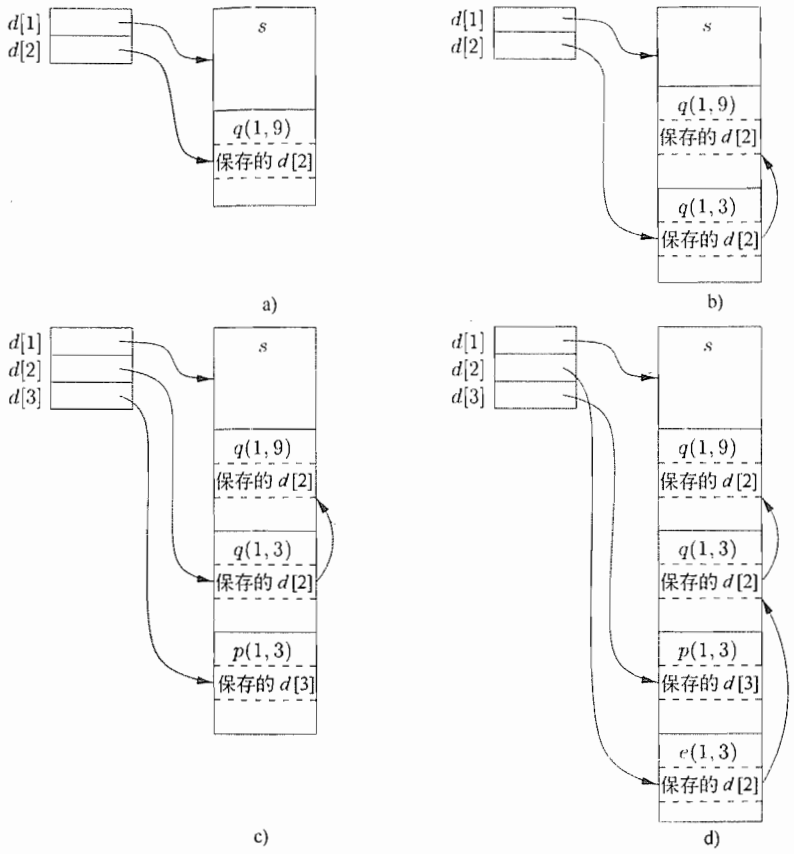


图 7-14 维护显示表

在图 7-14b 中，*quicksort*(1，9)调用 *quicksort*(1，3)。因为这两次调用的活动记录的深度都为 2，所以我们必须首先将 $d[2]$中指向 *quicksort*(1，9)的指针保存到 *quicksort*(1，3)的活动记录中去。然后 $d[2]$被设置为指向 *quicksort*(1，3)。

下一步调用 *partition*。这个函数的嵌套深度为 3，因此我们将首次使用显示表中的 $d[3]$位置，并使它指向 *partition* 的活动记录。*partition* 的记录中有一个存放原来的 $d[3]$值的位置。但是在这个例子中，$d[3]$原先没有值，因此这个位置上的指针为空。此时的显示表和栈如图 7-14c 所示。

然后，*partition* 调用 *exchange*。函数 *exchange* 的嵌套深度为 2，因此它的活动记录保存了旧的 $d[2]$指针，即指向 *quicksort*(1，3)的活动记录的指针。请注意，这里出现了多个显示表指针之间相互交叉的情况。也就是说，$d[3]$指向的位置比 $d[2]$所指位置更低。这是一个正常的情况，因为 *exchange* 只访问它自己的数据和通过 $d[1]$访问的 *sort* 的数据。 □

7.3.9 7.3 节的练习

练习 7.3.1：图 7-15 中给出了一个按照非标准方式计算 Fibonacci 数的 ML 语言的函数 `main`。函数 `fib0` 将计算第 n 个 Fibonacci 数($n \geqslant 0$)。嵌套在 `fib0` 中的是 `fib1`，它假设 $n \geqslant 2$ 并计算第 n 个 Fibonacci 数。嵌套在 `fib1` 中的是 `fib2`，它假设 $n \geqslant 4$。请注意，`fib1` 和 `fib2` 都不需要检查基本情况。我们考虑从对 `main` 的调用开始，直到(对 `fib0(1)` 的)第一次调用即将

返回的时段，请描述出当时的活动记录栈，并给出栈中的各个活动记录的访问链。

练习 7.3.2：假设我们使用显示表来实现图 7-15 中的函数。请给出对 `fib0(1)` 的第一次调用即将返回时的显示表。同时指明那时在栈中的各个活动记录中保存的显示表条目。

```
fun main () {
    let
        fun fib0(n) =
            let
                fun fib1(n) =
                    let
                        fun fib2(n) = fib1(n-1) + fib1(n-2)
                    in
                        if n >= 4 then fib2(n)
                        else fib0(n-1) + fib0(n-2)
                    end
            in
                if n >= 2 then fib1(n)
                else 1
            end
    in
        fib0(4)
    end;
```

图 7-15　计算 Fibonacci 数的嵌套函数

7.4　堆管理

堆是存储空间的一部分，它被用来存储那些生命周期不确定，或者将生存到被程序显式删除为止的数据。虽然局部变量通常在它们所属的过程结束之后就变得不可访问，但很多语言支持创建某种对象或其他数据，它们的存在与否和创建它们的过程的活动无关。例如，C ++ 和 Java 语言都为程序员提供了 `new` 语句，该语句创建的对象（或指向对象的指针）可以在过程之间进行传递，因此这些对象在创建它们的过程结束之后仍然可以长期存在。这样的对象被存放在堆区。

在本节中，我们将讨论*存储管理器*（memory manager），即分配和回收堆区空间的子系统，它是应用程序和操作系统之间的一个接口。对于 C 或 C ++ 这样需要手动回收存储块的语言（即通过程序中的显式语句，比如 `free` 或 `delete`，进行回收）而言，存储管理器还负责实现空间回收。

我们将在 7.5 节中讨论*垃圾回收*（garbage collection），即在堆区中找到那些不再被程序使用、因此可以被重新分配以便存放其他数据项的空间的过程。对于 Java 这样的语言，内存的回收是由垃圾回收器完成的。在需要进行垃圾回收时，垃圾回收器是存储管理器的一个重要子系统。

7.4.1　存储管理器

存储管理器总是跟踪堆区中的空闲空间。它具有两个基本的功能：

- *分配*。当程序为一个变量或对象请求内存时[⊖]，存储管理器产生一段连续的具有被请求大小的堆空间。如果有可能，它使用堆中的空闲空间来满足分配请求；如果没有被请求大小的空间块可供分配，它试图从操作系统中获得连续的虚拟内存来增加堆区的存储空

⊖ 在后面的内容中，我们将把需要内存空间的事物称为“对象”，尽管它们并不是“面向对象程序设计”意义上的真正对象。

间。如果空间已经用完，存储管理器将空间耗尽的信息传回给应用程序。

- 回收。存储管理器把被回收的空间返还到空闲空间的缓冲池中，这样它可以复用该空间来满足其他的分配请求。存储管理器通常不会将内存返回给操作系统，即使当这个程序不再需要那么多的堆空间时也不会归还给操作系统。

如果下面的(a)、(b)两个条件都成立，内存的管理就会相对简单：(a)所有分配请求都要求相同大小的存储块，(b)存储空间按照可预见的方式被释放，比如先分配先回收。对于有些语言(比如 Lisp)而言条件 a 成立。纯的 Lisp 语言只使用一种数据元素 ——一个双指针单元，所有的数据结构都在该元素的基础上构建。条件 b 在某些情况下也可能成立，最常见的情况是可以在运行时刻栈中分配的数据。然而，对于大部分的语言而言，这两个条件一般都不成立。相反地，我们需要为不同大小的数据元素分配空间，并且没有好方法可以预测所有已分配对象的生命期。

因此，存储管理器必须准备以任何顺序来处理任何大小的空间分配和回收请求。这些请求小到一个字节，大到该程序的整个地址空间。

下面是我们期望存储管理器具有的特性：

- 空间效率。存储管理器应该能够使一个程序所需的堆区空间的总量达到最小。这样做就可以在一个固定大小的虚拟地址空间中运行更大的程序。空间效率是通过使存储碎片达到最少而得到的，该技术将在 7.4.4 节中讨论。
- 程序效率。存储管理器应该充分利用存储子系统，使程序可以运行得更快。我们将在 7.4.2 节中看到，根据数据对象在存储中所处的不同位置，执行一条指令所花费的时间可能相差很大。幸运的是，程序通常会表现出“局部性”，7.4.3 节将讨论这种现象，它指的是通常的程序在访问内存时具有的非随机性聚集的特性。通过关注对象在存储中的放置方法，存储管理器可以更好地利用空间，并且有希望使程序运行得更快。
- 低开销。因为存储分配和回收在很多程序中是常用的操作，因此使得这些操作尽可能地高效是非常重要的。也就是说，我们希望最小化开销(overhead)，即花费在分配和回收上的执行时间在总运行时间中所占的比例。请注意，分配的开销由小型请求决定，管理大型对象的开销相对不重要，因为通常会在它上面执行大量的计算，这个开销被分摊了。

7.4.2 一台计算机的存储层次结构

存储管理和编译器优化必须在充分了解存储行为的基础上完成。现代机器的设计使得程序员不需要考虑内存子系统的细节就能够写出正确的程序。然而，程序的效率不仅取决于被执行的指令的数量，还取决于执行其中每条指令所花费的时间。不同情况下执行一条指令所花费的时间可能会有明显的不同，因为访问不同的存储区域所花费的时间从几纳秒到几毫秒不等。因此，数据密集型程序可以从能够充分利用存储子系统的优化技术中得到很大的好处。我们将在 7.4.3 节看到，这种优化可以利用程序的“局部性”现象，即一般程序的非随机行为。

内存访问时间上的巨大差异源于硬件技术的根本性局限。我们可以制造出一个小而快的存储器件或者大而慢的存储器件，但是无法制造出既大又快的存储器件。现在，制造一个具有纳秒级访问时间的千兆容量的存储器件仍然是不可能的，而纳秒级正是高性能处理器的运行速度。因此，在实践中，现代计算机都以*存储层次结构*(memory hierarchy)的方式安排它们的存储。如图 7-16 所示的一个存储层次结构由一系列存储元素组成，较小较快的元素“更加接近”处理器，较大但较慢的元素则离存储器比较远。

一个处理器通常具有少量寄存器，寄存器中的内容由软件控制。然后，它具有一层或多层高速缓存，这些高速缓存通常使用静态 RAM 制造，其大小从几千字节到几兆字节不等。层次结构中的下一层是物理(主)内存，它由数百兆到几千兆的动态 RAM 构成。物理内存由下一层的虚拟

内存提供支持，虚拟内存由几千兆字节的磁盘实现。在一次内存访问中，机器首先在最近（最底层的）的存储中寻找数据，如果数据不在那里则到上一层中寻找，以此类推。

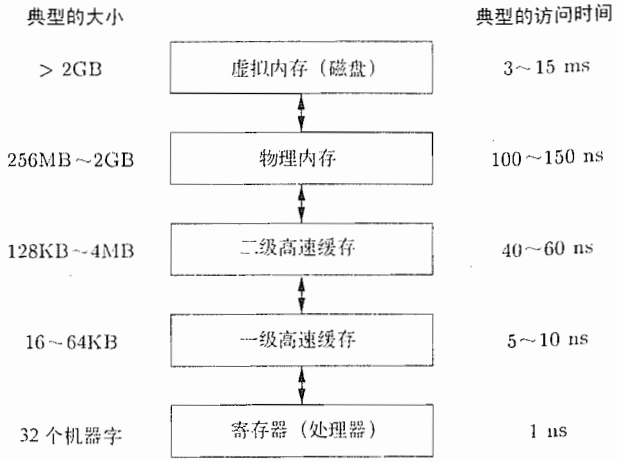


图 7-16 典型的内存层次结构的配置

寄存器个数很少，因此寄存器的使用会根据特定应用进行裁剪，并由编译器生成的代码进行管理。存储层次结构中的所有其他层都是自动管理的。这样做不仅简化了编程任务，并且相同的程序可以在具有不同存储配置的机器上高效工作。对于每次存储访问，机器从最低层开始逐层搜索每一层存储，直到找到数据为止。高速缓存是完全通过硬件进行管理的，这么做是为了能够跟上相对较快的 RAM 访问时间。因为磁盘访问速度相对较慢，虚拟内存是由操作系统进行管理的，辅以一个称为“转换旁视缓冲”的硬件结构。

数据以连续存储块的方式进行传输。为了分摊访问的开销，内存层次结构中较慢的层次通常使用较大的块。在主存和高速缓存之间的数据是按照被称为高速缓存线（cache line）的块进行传输的，高速缓存线的长度通常在 32 ~ 256 字节之间。在虚拟内存（硬盘）和主内存之间的数据是以被称为“页”（page）的内存块进行传输的，页的大小通常在 4 ~ 64 KB 之间。

7.4.3 程序中的局部性

大部分程序表现出高度的*局部性*（locality），也就是说，程序的大部分运行时间花费在相对较小的一部分代码中，此时它们只涉及少部分数据。如果一个程序访问的存储位置很可能将在一个很短的时间段内被再次访问，我们就说这个程序具有*时间局部性*（temporal locality）。如果被访问过的存储位置的临近位置很可能在一个很短的时间段内被访问，我们就说这个程序具有*空间局部性*（spatial locality）。

通常认为程序把 90% 的时间用来执行 10% 的代码。原因如下：

- 程序经常包含很多从来不会被执行的指令。使用组件和库构建得到的程序只使用了它们提供的一小部分功能。同时，随着需求的改变和程序的演化，遗留系统中常常包含很多不再被使用的指令。
- 在程序的一次典型运行中，可能被调用的代码中只有一小部分会被实际执行。例如，虽然处理非法输入和异常情况的指令对于程序的正确性是至关重要的，但是它们在某次运行中很少会被调用。
- 通常的程序往往将大部分时间花费在执行程序中的最内层循环和最紧凑的递归环上。

静态的和动态的 RAM

大部分随机访问内存是动态的(dynamic)，这意味着它们是由简单的电子电路构成的。这些电路会在短时间内丢失电位(因此也就会“忘记”它们原本存储的比特值)。这些电路需要定期刷新，即读出然后重新写入它们的比特。另一方面，在静态(static)RAM 的设计中，每个比特都需要一个更复杂的电路，结果是存储在其中的比特值可以保持任意长时间，直到它被改写为止。显然，一个芯片使用动态 RAM 电路可以比使用静态 RAM 电路存储更多的比特。因此我们通常会看到动态 RAM 类型的大容量主存，而像高速缓存这样的较小存储则使用静态电路构造。

局部性使得我们可以充分利用如图 7-16 所示的现代计算机的存储层次结构。将最常用的指令和数据放在快而小的存储中，而将其余部分放入慢而大的存储中，我们就可以显著地降低一个程序的平均存储访问时间。

人们已经发现，很多程序在对指令和数据的访问方式上既表现出时间局部性，又表现出空间局部性。然而，数据访问模式通常比指令访问模式表现出更大的多样性。将最近使用的数据放在最快的存储层次中的策略可以在普通程序中发挥很好的作用，但是在某些数据密集型程序中的作用并不明显——循环遍历非常大的数组的程序就是这样的例子。

仅仅通过查看代码，我们一般无法看出哪部分代码会被频繁地用到，针对特定输入指出这一点则更加困难。即使我们知道哪些指令会被频繁执行，最快的高速缓存通常也不能够同时存储这些指令。因此，我们必须动态调整最快的存储中的内容，用它们来保存可能很快会被频繁使用的指令。

利用存储层次结构的优化

将最近使用过的指令放入高速缓存的策略通常很有效。换句话说，过去的情况能够很好地预测将来的存储使用情况。当一条新的指令被执行时，其下一条指令也很有可能将被执行。这种现象是空间局部性的一个例子。提高指令的空间局部性的一个有效技术是让编译器把很可能连续执行的多个基本块(即总是顺序执行的指令序列)连续存放，即放在同一个存储页面中，可能的话甚至放在同一高速缓存线中。属于同一个循环或同一个函数的指令很有可能被一起运行[⊖]。

我们还可以改变数据布局或计算顺序，从而改进一个程序中的数据访问的时间局部性和空间局部性。例如，一些程序反复地访问大量数据，而每次访问只完成少量的计算，这样的程序的性能不会很好。我们可以每次将一部分数据从存储层次结构的较慢层次加载到较快层次(比如从磁盘移到主存)，并且在这些数据驻留在较快层中时执行所有针对这些数据的运算，那么程序的性能就会变得更好。这个概念可以递归地应用于物理内存、高速缓存以及寄存器中的数据的复用。

高速缓存体系结构

我们如何知道一个高速缓存线在高速缓存中呢？逐个检查高速缓存中的每一条高速缓存线过于费时，因此在实践中常常会限制一条高速缓存线在高速缓存中的放置位置。这个约束

⊖ 当机器从内存中获得一个存储字时，同时预取(prefetch)出其后的多个连续内存字的开销相对较小。因此，一个常见的存储层次结构的特性是在每次访问某层存储的时候会从该层存储中获取一个包含了多个机器字的块。

称为成组相关性(set associativity)。如果在一个高速缓存中，一条缓存线只能被放在 k 个位置上，那么这个高速缓存就称为 k 路成组相关的(k-way set associative)。最简单的高速缓存是 1 路相关高速缓存，它也称为直接映射高速缓存(direct-mapped cache)。在一个直接映射高速缓存中，存储地址为 n 的数据只能够放在缓存地址 $n \bmod s$ 上，其中 s 是这个高速缓存的大小。类似地，一个 k 路成组相关高速缓存被分为 k 个集合，而一个地址为 n 的数据只能映射到各个集合中的位置 $n \bmod (s/k)$ 上。大部分指令和数据高速缓存的相关性在 1 ~ 8 之间。如果一条缓存线被调入高速缓存，并且所有可能存放这个高速缓存线的位置都已经被占用，那么通常情况下会将最近最少使用的缓存线清除出高速缓存。

7.4.4 碎片整理

在程序开始执行的时候，堆区就是一个连续的空闲空间单元。随着这个程序分配和回收存储工作的进行，空间被分割成若干空闲存储块和已用存储块，而空闲块不一定位于堆区的某个连续区域中。我们将空闲存储块称为“窗口”(hole)。对于每个分配请求，存储管理器必须将请求的存储块放入一个足够大的“窗口”中。除非找到一个大小恰好相等的“窗口”，否则我们必定会切分某个窗口，结果创建出更小的窗口。

对于每个回收请求，被释放的存储块被放回到空闲空间的缓冲池中。我们把连续的窗口接合(coalesce)成为更大的窗口，否则窗口只会越变越小。如果我们不小心，空闲存储最终会变成碎片，即大量的细小且不连续的窗口。此时，就有可能找不到一个足够大的“窗口”来满足某个将来的请求，尽管总的空闲空间可能仍然充足。

best-fit 和 next-fit 对象放置

我们通过控制存储管理器在堆区中放置新对象的方法来减少碎片。经验表明，使现实中的程序中碎片最少的一个良好策略是将请求的存储分配在满足请求的最小可用窗口中。这个 best-fit 算法趋向于将大的窗口保留下来满足后续的更大请求。另一种策略被称为 first-fit。在这个策略中，对象被放置到第一个(即地址最低的)能够容纳请求对象的窗口中。这种策略在放置对象时花费的时间较少，但是人们发现它在总体性能上要比 best-fit 策略差。

为了更有效地实现 best-fit 放置策略，我们可以根据空闲空间块的大小，将它们分在若干个容器中。一个实际可行的想法是为较小的尺寸设置较多的容器，因为小对象的个数通常比较多。例如，在 GNU 的 C 编译器 gcc 中使用的存储管理器 Lea 将所有的存储块对齐到 8 字节的边界。对于 16 字节到 512 字节之间的、每个大小为 8 字节整数倍的存储块，这个存储管理器都设置了一个容器。更大尺寸的容器按照对数值进行划分(即每个容器的最小尺寸是前一个容器的最小尺寸的两倍)。在每一个容器中，存储块按照它们的大小排列。总是存在这样一个空闲空间块，存储管理器可以向操作系统请求更多的页面来扩展这个块。这个块被称为“荒野块”(wilderness chunk)。因为它的可扩展性，Lea 把这个块当作最大尺寸存储块的容器。

容器机制使得寻找 best-fit 块变得容易。

- 如果被请求的尺寸有一个专有容器，即该容器只包含该尺寸的存储块，我们可以从该容器中任意取出一个存储块。Lea 存储管理器在处理小尺寸请求时就是这样做的。
- 如果被请求的尺寸没有专有的容器，我们可以找出一个能够包含该尺寸的存储块的容器。在这个容器中，我们可以使用 first-fit 或 best-fit 策略。也就是说，我们既可以找到并选择第一个足够大的存储块，也可以花更多的时间去寻找最小的满足需求的存储块。注意，如果选择的空闲存储块的大小不是正好合适，通常将该块的剩余部分放到一个对应于更

小尺寸的容器中。

- 不过，这个目标容器可能为空，或者这个容器中的所有存储块都太小，不能满足空间请求。在这种情况下，我们只需要使用对应于下一个较大尺寸的容器重新进行搜索。最后，我们要么找到可以使用的存储块，要么到达“荒野块”。从这个荒野块中我们一定可以得到需要的空间，但有可能需要请求操作系统为堆区增加更多的内存页。

虽然 best-fit 放置策略可以提高空间利用率，但从空间局部性的角度考虑，它可能并不是最好的。程序在同一时间分配的块通常具有类似的访问模式，并具有类似的生命周期。因此将它们放置在一起可以改善程序的空间局部性。对 best-fit 算法的有用改进之一是在找不到恰巧等于请求尺寸的存储块时，使用另一种对象放置方法。在这种情况下，我们使用 next-fit 策略，只要刚刚分割过的存储块中还有足够的空间来容纳这个对象，我们就把这个对象放置在这个存储块中。next-fit 策略还可以提高分配操作的速度。

管理和接合空闲空间

当一个对象通过手工方式回收时，存储管理器必须将该存储块设置为空闲的，以便它可以被再次分配。在某些情况下，还可以将这个块和堆中的相邻块合并（接合）起来，构成一个更大的块。这样做是有好处的。因为我们总能够用一个大的存储块来完成总量相等的多个小存储块所完成的工作，但是不能用很多个小存储块来保存一个大对象，而合并后的存储块就有可能做到。

如果我们为所有具有固定尺寸的存储块保留一个容器，如 Lea 中为小尺寸块所做的那样，那么我们可能倾向于不把相邻的该尺寸的块合并成为双倍大小的块。比较简单的做法是将所有同样大小的块全部按照需要放在多个页中，而不必接合。那么，一个简单的分配/回收方案是维护一个位映射，其中的每个比特对应于容器中的一个块。1 代表该块已被占用，0 表示它是空闲的。当一个块被回收时，我们将它对应的 1 改为 0。当我们需要分配一个存储块时，便找出任意一个相应比特为 0 的块，将这个位改为 1，然后就可以使用该内存块了。如果没有空闲块，我们就获取一个新的页，将其分割成适当大小的存储块，同时扩展用于存储管理的位向量。

在有些情况下问题会变得比较复杂。比如，我们不使用容器而把堆区作为一个整体进行管理；或者我们想要接合相邻的块，并在必要的时候将合并得到的块移动到另一个容器中。有两种数据结构可以用于支持相邻空闲块的接合：

- *边界标记*。在每个（不管是空闲的还是已分配的）存储块的高低两端，我们都存放了重要的信息。在块的两端都设置了一个 free/used 位，用来标识当前该块是已用的（used）还是空闲的（free）。在与每一个 free/used 位相邻的位置上存放了该块中的字节总数。
- *一个双重链接的、嵌入式的空闲列表*。各个空闲块（而不是已分配的块）还使用一个双重链表进行链接。这个链表的指针就存放在这些块中，比如说存放在紧挨着某一端边界标记的位置上。因此，不需要额外的空间来存放这个空闲块列表，尽管它的存在为块的大小设置了一个下界。即使数据对象只有一个字节，存储块也必须提供存放两个边界标记和两个指针的空间。空闲列表中的存储块的顺序没有确定。例如，这个列表可以按块的大小排序，因此可以支持 best-fit 放置策略。

例 7.10 图 7-17 给出堆区的一个部分，其中包含三个相邻的存储块 A、B 和 C。B 块的大小为 100，它刚刚被回收并回到了空闲列表中。因为我们知道 B 的开始位置（左端），也就知道了紧靠在 B 的左边的存储块的末端，在这个例子中就是 A。A 右端的 free/used 位当前为 0，因此 A 也是空闲的。于是我们可以将 A 和 B 接合成一个 300 字节的存储块。

有可能出现这样的情况，即紧靠在 B 的右端的存储块 C 也是空闲的。在这种情况下，我们可

以把 A、B 和 C 全部合并起来。请注意，如果我们总是尽可能地把存储块接合起来，那么就不会有两个连续的空闲块。因此我们总是只需要查看与正被回收的块相邻的两个块。在当前例子中，我们按照下面的步骤找到 C 的开始位置。我们从已知的 B 的左端开始，在 B 的左边界标记中知道 B 块的总字节数为 100 字节。根据这个信息，我们可以找到 B 的右端和紧靠在 B 右边的存储块的起始位置。在该点上，我们检查 C 的 free/used 位，发现其值为 1，表明 C 正在被使用，因此 C 不可以被接合。

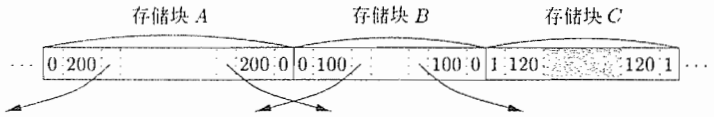


图 7-17 堆的片段和一个双重链接的空闲列表

因为我们必须接合 A 和 B，所以需要从空闲列表中删除它们中的一个。空闲列表的双重链接结构使得我们可以找到 A 和 B 中的前驱和后继结点。请注意，不应该假定在物理上相邻的 A 和 B 在空闲列表中也相邻。知道了 A 和 B 在空闲列表中的前驱和后继的存储块，就可以操作列表中的指针并将 A 和 B 替换为一个接合后的存储块。 □

如果自动垃圾回收过程将所有已分配的存储块移动到一段连续的存储中，它同时还可以消除所有的碎片。在 7.6.4 节中将更详细地讨论垃圾回收机制和存储管理之间的相互影响。

7.4.5 人工回收请求

我们在本节的最后讨论人工存储管理。此时，程序员必须像在 C 和 C++ 语言中那样显式地安排数据的回收。在理想情况下，任何不会再被访问的存储都应该删除。反过来，任何可能还会被引用的空间都不能删除。遗憾的是，这两个性质都很难保证。除了考虑人工回收的困难之处以外，我们还将描述一些被程序员用于处理这些难点的技术。

人工回收带来的问题

人工存储管理很容易出错。常见的错误有两种形式：一直未能删除不能被引用的数据，这称为*内存泄漏*(memory-leak)错误；引用已经被删除的数据，这称为*悬空指针引用*(dangling-pointer-dereference)错误。

程序员不能保证一个程序是否永远不会在将来引用某块存储，因此第一个常见的错误是没有删除那些不会被再次引用的数据。请注意，尽管内存泄漏可能由于占用的存储增多而降低程序运行的速度，但是只要机器没有用完全部存储，它们就不会影响程序的正确性。很多程序可以容忍内存泄漏，当泄漏比较缓慢时尤其如此。然而，对于长期运行的程序，特别是像操作系统和服务器代码这样不间断运行的程序，保证它们没有内存泄漏是非常关键的。

自动垃圾回收通过回收所有的垃圾而消除了内存泄漏问题。即使使用自动垃圾回收机制，程序可能仍然耗费了过多的内存。有时尽管在某处还存在着对某个对象的引用，但程序员可能已经知道该对象不会再被引用。在那种情况下，程序员可以主动地删除指向那些不会再被引用的对象的引用，使得这些对象可以被自动回收。

一个工具实例：Purify

Rational 的 Purify 是帮助程序员寻找程序中的内存访问错误和内存泄漏的最常用的商业工具之一。Purify 对二进制代码进行插装，加入在程序运行时检查程序错误的附加指令。它维护了一个存储的映像图，指明所有空闲的和已用的空间的分布。每个已分配空间的对象都被一段额外空间包围；对未分配空间的访问，或对数据对象之间的间隙空间的访问都被标记

为错误。通过这种方法可以找到一些悬空指针引用，但是当该内存已经被重新分配且该位置上已经存在一个有效对象时，这种方法就无能为力了。这种方法还可以找到一些越界的数组访问，前提是它们恰巧落在这些对象之后，由 Purify 插入的空间中。

Purify 也可以在程序运行结束时发现内存泄漏。它搜索所有的已分配的对象中的内容，找出所有可能的指针值。任何没有指针指向的对象都是一块泄漏的存储块。Purify 可以报告泄漏内存的大小和泄漏对象的位置。我们可以将 Purify 和一个“保守的垃圾回收器”相比较，后者将在 7.8.3 节中讨论。

过度热衷于删除对象可能引起比内存泄漏更严重的问题。第二个常见的错误是删除了某个存储空间，然后又试图去引用这个已回收空间中的数据。指向已回收空间的指针称为悬空指针(dangling pointer)。一旦这个已释放的空间被重新分配给另一个变量，通过该悬空指针进行的任何读、写或回收操作都可能产生看起来不可捉摸的结果。我们把诸如读、写、回收等沿着一个指针试图使用该指针所指对象的所有操作称为对这个指针的“解引用”(dereferencing)。

注意，通过一个悬空指针读取数据可能会返回不确定的值。通过一个悬空指针进行写操作则可能不确定地改变新变量的值。回收一个悬空指针的存储空间意味着这个新变量的存储空间可能被分配给另一个变量。新旧变量上的动作可能会相互冲突。

和内存泄漏不一样，在释放的空间被重新分配之后再对相应的悬空指针进行解引用总是会带来难以调试的程序错误。因而，当程序员不能确定一个变量是否还会被引用时，他们更倾向于不回收该变量。

另一个相关的编程错误形式是访问非法地址。这种错误的常见例子包括对空指针的解引用和访问一个数组界限之外的元素。探测出这种错误要好过任由程序产生错误结果。实际上，很多安全危害就是利用了这种类型的程序错误。其中，某个程序输入会导致意想不到的数据访问，使得一个黑客取得这个程序和机器的控制权。解决办法之一是让编译器在每次访问中插入检查代码，以保证该次访问在数组界限之内。一些编译器的优化器可以发现并删除那些不必要的检查代码，因为这些优化器能够推导出相应的访问必然在区间之内。

编程规范和工具

现在我们给出几个最流行的编程规范和工具，开发它们的目的是帮助程序员来应对的存储管理的复杂性：

- 当一个对象的生命周期能够被静态推导出来时，*对象所有者*(object ownership)的概念是很有用的。它的基本思想是在任何时候都给每个对象关联上一个*所有者*(owner)。这个所有者是指向该对象的一个指针，通常属于某个函数调用。所有者(也就是这个函数)负责删除这个对象或者把这个对象传递给另一个所有者。可能会有其他的指针也指向同一个对象，但是这些指针不代表拥有关系。这些指针可在任何时刻被覆盖，但是绝对不应该通过它们进行删除操作。这个规范可以消除内存泄漏，同时也可以避免将同一对象删除两次。然而，它对解决悬空指针引用问题没有帮助，因为有可能沿着一个不代表拥有关系的指针访问一个已经被删除的对象。
- 当一个对象的生命周期需要动态确定时，*引用计数*(reference counting)会有所帮助。它的基本思想是给每个动态分配的对象附上一个计数。在指向这个对象的引用被创建时，我们将此对象的引用计数加一；当一个引用被删除时，我们将此引用计数减一。当计数变成 0 时，这个对象就不会再被引用，因此可以被删除。然而，这个技术不能发现无用的循环数据结构，其中的一组对象不能再被访问，但是因为它们之间互相引用，导致它们的引

用计数不为0。在例子7.11中可以看到这个问题的一个示例。引用计数技术确实可以根除所有的悬空指针引用，因为不存在指向已删除对象的引用。因为引用计数在存储一个指针的每次运算上增加了额外开销，因此引用计数的运行时刻代价很大。

- 对于其生命周期局限于计算过程中的某个特定阶段的一组对象，可以使用基于区域的分配(region-based allocation)方法。当被创建的对象只在一个计算过程的某个步骤中使用时，我们可以把这些对象分配在同一个区域中。一旦这个计算步骤完成，我们就删除整个区域。基于区域的分配方法有一定的局限性。然而当可以使用它时，它又非常高效。因为该技术以成批的方式一次性删除区域中的所有对象，而不是每次回收一个对象。

7.4.6 7.4节的练习

练习7.4.1：假设堆区从0地址开始编址，由几个存储块组成。按照地址顺序，这些存储块的大小分别是80，30，60，50，70，20，40个字节。当我们在一个存储块中放入一个对象时，如果该块中的剩余空间仍然足以形成一个较小的块，我们就将此对象放置在块的高端(这样可以比较容易地把较小的块保存在空闲空间的链表中)。然而，我们不能使用小于8个字节的存储块，因此如果一个对象和被选中的存储块差不多大，我们就把整个块分配给它，并将这个对象放置在这个块的低端。如果我们按顺序为大小分别为32、64、48、16的对象申请空间，在满足了这些请求之后的空闲空间列表是什么样子的？假设选择存储块的方法是：

1) First-fit

2) Best-fit

7.5 垃圾回收概述

不能被引用的数据通常称为垃圾(garbage)。很多高级程序设计语言提供了用以回收不可达数据的自动垃圾回收机制，从而解除了程序员进行手工存储管理的负担。垃圾回收最早出现在1958年的Lisp语言的初次实现中。其他提供垃圾回收机制的主要语言包括Java、Perl、ML、Modula-3、Prolog和Smalltalk。

在本节中，我们将介绍多个和垃圾回收相关的概念。对象“可达”这个概念是很直观的，但是我们仍需要精确地定义，准确的规则将在7.5.2节中讨论。我们将在7.5.3节中讨论一种简单但是有缺陷的自动垃圾回收方法：引用计数。它基于如下的思想：一旦一个程序失去了指向一个对象的所有引用，它就不能并且也不会再引用该对象的存储空间。

7.6节将讨论基于跟踪的回收器。它包含多个算法，用以找出所有仍然有用的对象，然后将堆区中所有的其他存储块变成空闲空间。

7.5.1 垃圾回收器的设计目标

垃圾回收是重新收回那些存放了不能再被程序访问的对象的存储块。我们假定这些对象的类型可以由垃圾回收器在运行时刻确定。基于这个类型信息，我们可以知道该对象有多大，以及该对象的哪些分量包含指向其他对象的引用(指针)。我们还假定对对象的引用总是指向该对象的起始位置，而不会指向该对象中间的位置。因此，对同一个对象的所有引用具有相同的值，可以被很容易地识别。

我们把一个用户程序称为*增变者*(mutator)，它会修改堆区中的对象集合。增变者从存储管理器处获取空间，创建对象，它还可以引入和消除对已有对象的引用。当增变者程序不能“到达”某些对象时，这些对象就变成了垃圾。在7.5.2节中将给出“到达”的准确定义。垃圾回收器找到这些不可达对象，并将这些对象交给跟踪空闲空间的存储管理器，收回它们所占的空间。

一个基本要求：类型安全

不是所有的语言都适合进行自动垃圾回收。为了使垃圾回收器能够工作，它必须知道任何给定的数据元素或一个数据元素的分量是否为(或可否被用作)一个指向某块已分配存储空间的指针。在一种语言中，如果任何数据分量的类型都是可确定的，那么这种语言就称为*类型安全*(typesafe)的。对于某些类型安全的语言，比如 ML，我们可以在编译时刻确定数据的类型。另外一些类型安全语言，比如 Java，其类型不能在编译时刻确定，但是可以在运行时刻确定。后者称为*动态类型*(dynamically typed)语言。如果一个语言既不是静态类型安全的，也不是动态类型安全的，它就被称为*不安全的*(unsafe)。

类型不安全的语言不适合使用自动垃圾回收机制。遗憾的是，有些最重要语言却是类型不安全的，比如 C 和 C++。在不安全语言中，存储地址可以进行任意操作：可以将任意的算术运算应用于指针，创建出一个新的指针，并且任何整数都可以被强制转化为指针。因此，从理论上来说，一个程序可以在任何时候引用内存中的任何位置。这样，没有哪个内存位置可以被认为是不可访问的，也就无法安全地收回任何存储空间。

在实践中，大部分 C 和 C++ 程序并没有随意地生成指针。因此人们开发了一个在理论上不正确，但是实践经验表明很有效的垃圾回收器。我们将在 7.8.3 节中讨论用于 C 和 C++ 语言的保守的垃圾回收技术。

性能度量

尽管在几十年前就发明了垃圾回收机制，并且它能够完全防止内存泄漏，但是垃圾回收的代价是如此高昂，所以至今没有被很多主流的程序设计语言使用。在多年的研究中，很多不同的回收方法被提出来，但是还没有一种无可争议的最好的垃圾回收算法。在讨论这些方法之前，我们首先列举一些在设计垃圾回收器时必须考虑的性能度量标准。

- *总体运行时间*。垃圾回收的速度可能会很慢。使它不会显著增加一个应用程序的总运行时间是很重要的。因为垃圾回收器必须要访问很多数据，它的性能很大程度上决定于它能否充分利用存储子系统。
- *空间使用*。重要之处在于垃圾回收机制避免了内存碎片，并最大限度地利用了可用内存。
- *停顿时间*。简单的垃圾回收器有一个众所周知的问题，即垃圾回收过程会在没有任何预警的情况下突然启动，导致程序(即增变者)突然长时间停顿。因此，除了最小化总体运行时间之外，人们还希望将最长停顿时间最小化。作为一个重要的特例，实时应用要求某些计算在一个时间界限内完成。我们要么在执行实时任务时压制住垃圾回收过程，要么限定最长停顿时间。因此，垃圾回收机制很少在实时应用中使用。
- *程序局部性*。我们不能只通过一个垃圾回收器的运行时间来评价它的速度。垃圾回收器控制了数据的放置，因此影响了增变者程序的数据局部性。它可以通过释放空间并复用该空间来改善增变者程序的时间局部性；它也可以将那些一起使用的数据重新放置在同一个高速缓存线或内存页上，从而改善程序的空间局部性。

这些设计目标中的某些目标可能互相冲突，设计者必须在认真考虑程序的典型行为之后作出权衡。不同特性的对象可能适会使用不同的处理方式，这就要求垃圾回收器使用不同的技术来处理不同类型的对象。

例如，已分配的对象数量中小对象的数量很大比例，那么对小对象的分配不能产生大的开销。另一方面，考虑一下对可达对象进行重定位的垃圾回收器。在处理大对象时重新定位是非常昂贵的，但在处理小对象时代价就比较小。

考虑另一个例子。一般来说，在基于跟踪的回收器中，我们等待垃圾回收的时间越长，可回

收对象的比例就越大。原因在于很多对象常常"英年早逝"，因此如果我们等一段时间，很多新分配的对象就会变成不可达的。这样的回收器平均花在每个被回收对象上的开销就会变小。另一方面，降低回收频率会增加程序的内存使用要求，降低数据局部性，并增加停顿时间。

相比之下，一个使用引用计数的回收器给增变者的每次运算引入一个常量开销，从而明显地减慢程序的整体运行速度。但是另一方面，引用计数技术不会产生长时间的停顿，并且能够有效地利用内存，因为它可以在垃圾产生时立刻发现它们(除了 7.5.3 节中将讨论的特定的循环结构)。

语言的设计同样会影响内存使用的特性。有些语言提倡的程序设计风格会产生很多垃圾。比如，函数式(或者几乎函数式)的程序设计语言为了避免改变已存在的对象，会创建出更多的对象。在 Java 中，除了整型和引用这样的基本类型，所有的对象都被分配在堆区而不是栈区。即使这些对象的生命周期被限制在一次函数调用的生命周期内，它们仍然被分到堆区中。这种设计使得程序员不需要关注变量的生命周期，但是其代价是产生更多的垃圾。已经有一些编译器优化技术可以分析变量的生命周期，并尽可能地将它们分配到栈区。

7.5.2 可达性

我们把所有不需要对任何指针解引用就可以被程序直接访问的数据称为根集(root set)。例如，在 Java 中，一个程序的根集由所有的静态字段成员和栈中的所有变量组成。显然，程序可以在任何时候访问根集中的任何成员。递归地，对于任意一个对象，如果指向它的一个引用被保存在任何可达对象的字段成员或数组元素中，那么这个对象本身也是可达的。

当程序被编译器优化之后，可达性问题会变得更加复杂。首先，编译器可能会把引用变量放在寄存器中。这些引用也必须被看做是根集的一部分。其次，尽管在一个类型安全语言中，程序员不能直接操作内存地址，但是编译器常常会为了提高代码速度而这么做。因此，编译得到的代码中的寄存器可能会指向一个对象或数组的中间位置，或者程序可能把一个偏移量加到这些寄存器中的值上，计算得到一个合法地址。为了使得垃圾回收器能够找到正确的根集，优化编译器可以做如下的处理：

- 编译器可以限制垃圾回收机制只能在程序中的某些代码点上被激活。在这些点上没有"隐藏"的引用。
- 编译器可以写出一些信息供垃圾回收器恢复所有的引用。比如，指出哪些寄存器中包含了引用，或者如何根据给定的某个对象的内部地址来计算该对象的基地址。
- 编译器可以确保当垃圾回收器被激活时每个可达对象都有一个引用指向它的基地址。

可达对象的集合随着程序的执行而变化。当新对象被创建时该集合会增长，当某些对象变得不可达时该集合就缩小。重要的是记住一旦某个对象变得不可达，它就不可能再次变得可达。下面是一个增变者程序改变可达对象集合的四种基本操作：

- *对象分配*。这些操作由存储管理器完成。它返回一个指向新创建的存储区域的引用。这个操作向可达对象集中添加成员。
- *参数传递和返回值*。对象引用从实在输入参数传递到相应的形式参数，也可以从返回结果传回给调用者。这些引用指向的对象仍然是可达的。
- *引用赋值*。对于引用 u 和 v，形如 $u = v$ 的赋值语句有两个效果。首先，u 现在是 v 所指对象的一个引用。只要 u 是可达的，那么它指向的对象当然也是可达的。其次，u 中原来的引用丢失了。如果这个引用是指向某一可达对象的最后一个引用，那么那个对象就变成不可达的。当某个对象变得不可达时，所有只能通过这个对象中的引用到达的对象都会变成不可达的。

- 过程返回。当一个过程退出时，保存其局部变量的活动记录将被弹出栈。如果这个活动记录中保存了某个对象的唯一引用，那个对象就变得不可达。同样，如果这个刚刚变得不可达的对象保存了指向其他对象的唯一引用，那么那些对象也将变得不可达，以此类推。

总而言之，新的对象通过对象分配被引入。参数传递和赋值可以传递可达性；赋值和过程结束可能结束对象的可达性。当一个对象变得不可达时，可能会导致更多的对象变得不可达。

栈对象的残存问题

当一个过程被调用时，一个局部变量 v 的对象被分配在栈中。可能会有一些指向 v 的指针被放置在非局部变量中。这些指针将在这个过程返回之后继续存在，但是存放 v 的空间消失了，从而产生了一个悬空指针的情况。我们是否应该象 C 所作的那样将象 v 这样的局部变量分配在栈中呢？答案是很多语言的语义要求局部变量在它们的过程返回后不再存在。保留一个指向这样的变量的引用是一个编程错误，不会要求编译器去改正程序中的这个错误。

有两种寻找不可达对象的基本方法。我们可以捕获可达对象变得不可达的转变时刻，也可以周期性地定位出所有可达对象，然后推出所有其他对象都是不可达的。7.4.5 节中介绍的引用计数技术是一种著名的近似实现第一种方法的技术。我们在增变者执行可能改变可达对象集合的动作时，维护了指向各个对象的引用的计数。当计数器变成 0 时，相应的对象变得不可达。我们将在 7.5.3 节中更详细地讨论这个方法。

第二种方法传递地跟踪所有的引用，从而计算可达性。一个基于跟踪的垃圾回收器首先为根集中的所有对象加上“可达的”标号，然后重复地检查可达对象中的所有引用，找到更多的可达对象，并为它们加上同样的标号。这个方法必须首先跟踪所有的引用，然后才能决定哪些对象是不可达的。但是一旦计算得到可达集合，它就可以立刻找到很多不可达对象，并同时确定大量的空闲存储空间。因为所有的引用都必须在同一时刻进行分析，所以我们还可以选择将可达对象重新定位，从而减少碎片。有很多种不同的基于跟踪的算法，我们将在 7.6 节和 7.7.1 节中讨论这些可选算法。

7.5.3 引用计数垃圾回收器

现在，我们考虑一个简单但有缺陷的基于引用计数的垃圾回收器。当一个对象从可达转变为不可达的时候，该回收器就可以将该对象确认为垃圾；当一个对象的引用计数为 0 时，该对象就会被删除。使用引用计数的垃圾回收器时，每个对象必须有一个用于存放引用计数的字段。引用计数可以按照下面的方法进行维护：

1）对象分配。新对象的引用计数被设置为 1。

2）参数传递。被传递给一个过程的每个对象的引用计数加一。

3）引用赋值。如果 u 和 v 都是引用，对于语句 $u=v$，v 指向的对象的引用计数加 1，u 本来指向的原对象的引用计数减 1。

4）过程返回。当一个过程退出时，该过程活动记录的局部变量中所指向的对象的引用数必须减一。如果多个局部变量存放了指向同一对象的引用，那么对每个这样的引用，该对象的引用计数都要减 1。

5）可达性的传递丢失。当一个对象的引用计数变成 0 时，我们必须将该对象中的各个引用所指向的每个对象的引用计数减 1。

引用计数有两个主要的缺点：它不能回收不可达的循环数据结构，并且它的开销较大。循环数据结构的出现都是有理由的：数据结构常常会指回到它们的父结点，也可能相互指向对方，从

而形成交叉引用。

例 7.11 图 7-18 给出了三个对象以及它们之间的引用，但是没有来自其他部分的引用。如果这些对象都不是根集的成员，那么它们都是垃圾，但是它们的引用计数都大于 0。如果我们在垃圾回收中使用引用计数技术，这个情况就等同于一次内存泄漏，因为这种垃圾以及任何类似的结构永远不会被回收。 □

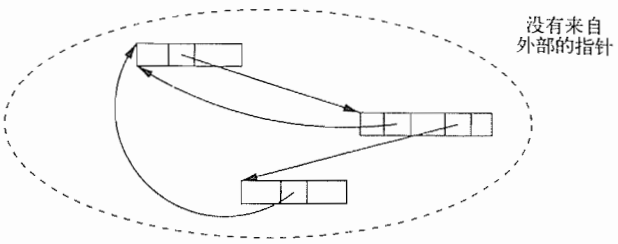


图 7-18 一个不可达的循环数据结构

引用计数的开销比较大，因为每一次引用赋值，以及在每个过程的入口和出口处，都会增加一个额外运算。这个开销和程序中的计算量成正比关系，而不仅仅和系统中的对象数目相关。需要特别考虑的是对一个程序的根集中的引用的更新。局部栈访问会引起引用计数的更新，为了消除因这种更新而引起的时间开销，人们提出了*延期引用计数*的概念。也就是说，引用计数不包括来自程序根集的引用。除非扫描整个根集仍没有找到指向某一对象的引用，否则这个对象不会被当作垃圾。

另一方面，引用计数的优势在于垃圾回收是以*增量方式*完成的。尽管总的开销可能很大，但这些运算分布在增变者的整个计算过程中。尽管删除一个引用可能致使大量对象变得不可达，我们可以很容易地延期执行递归地修改引用计数的运算，并在不同的时间点上逐步完成修改。因此，当应用必须满足某个时间期限时，或者对于不能接受长时间突然停顿的交互式系统而言，引用计数是一种特别有吸引力的算法。这个方法的另一种优势是垃圾被及时回收，从而保持了较低的空间使用量。

7.5.4 7.5 节的练习

练习 7.5.1：当下列事件发生时，图 7-19 中的对象的引用计数会发生哪些改变？

1）从 A 指向 B 的指针被删除。

2）从 X 指向 A 的指针被删除。

3）结点 C 被删除。

练习 7.5.2：当图 7-20 中的从 A 到 D 的指针被删除时，引用计数会发生什么样的改变？

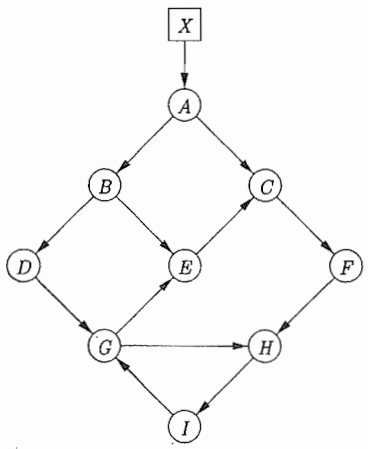


图 7-19 一个对象网络

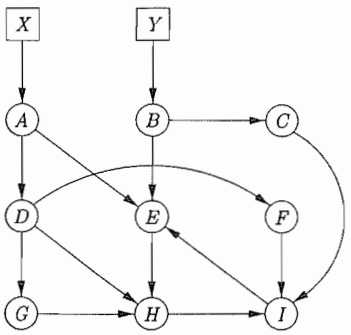


图 7-20 另一个对象网络

7.6 基于跟踪的回收的介绍

基于跟踪的回收器并不在垃圾产生的时候就进行回收，而是会周期性地运行，寻找不可达对象并收回它们的空间。通常的做法是在空闲空间被耗尽或者空闲空间数量低于某个阈值时启动垃圾回收器。

在本节中，我们首先介绍最简单的“标记 - 清扫式”垃圾回收算法。然后我们将通过存储块可能具有的四种状态来描述多个基于跟踪的算法。这一节中还包含了一些对基本算法的改进，包括那些将对象重定位加入到垃圾回收功能中的算法。

7.6.1 基本的标记 - 清扫式回收器

标记 - 清扫式(mark-and-sweep)垃圾回收算法是一种直接的全面停顿的算法。它们找出所有不可达的对象，并将它们放入空闲空间列表。算法 7.12 在一开始的跟踪步骤中访问并“标记”所有的可达对象，然后“清扫”整个堆区并释放不可达对象。在介绍了基于跟踪的算法的一个一般性框架之后，我们将考虑算法 7.14，它是算法 7.12 的一个优化。算法 7.14 使用一个附加的列表来保存所有已分配对象，使得它对每个可达对象只访问一次。

算法 7.12 标记 - 清扫式垃圾回收。

输入：一个由对象组成的根集，一个堆和一个被称为 *Free* 的包含了堆中所有未分配存储块的空闲空间列表(free list)。和 7.4.4 节中一样，所有空间块都用边界标记进行标识，指明它们的空闲/已用状态和大小。

输出：在删除了所有垃圾之后的经过修改的 *Free* 列表。

方法：在图 7-21 中显示的算法使用了几个简单的数据结构。列表 *Free* 保存了已知的空闲对象。一个名为 *Unscanned* 的列表保存了我们已经确定可达的对象，但是我们还没有考虑这些对象的后继对象的可达性。也就是说，我们还没有扫描这些对象来确定通过它们能够到达哪些对象。列表 *Unscanned* 最初为空。另外，每个对象包括一个比特，用来指明该对象是否可达(即 *reached* 位)。在算法开始之前，所有已分配对象的 *reached* 位都被设定为 0。

```
    /* 标记阶段 */
1)  /* 把被根集引用的每个对象的 reached 位设置为 1，并把它加入
       到 Unscanned 列表中；*/
2)  while (Unscanned ≠ ∅) {
3)        从 Unscanned 列表中删除某个对象 o；
4)        for (在 o 中引用的每个对象 o′) {
5)              if (o′ 尚未被访问到，即它的 reached 位为 0) {
6)                    将 o′ 的 reached 位设置为 1；
7)                    将 o′ 放到 Unscanned 中；
                  }
          }
    }
    /* 清扫阶段 */
8)  Free = ∅;
9)  for (堆区中的每个内存块 o) {
10)       if (o 未被访问到，即它的 reached 位为 0) 将 o 加入到 Free 中；
11)       else 将 o 的 reached 位设置为 0；
    }
```

图 7-21 一个标记 - 清扫式垃圾回收器

在图 7-21 的第(1)行，我们初始化 *Unscanned* 列表，在其中放入所有被根集引用的对象。同时这些对象的 *reached* 位被设置为 1。第(2)行到第(7)行是一个循环，在此循环中我们逐个检查

每个已经被放入 *Unscanned* 列表中的对象 *o*。

从第(4)行到第(7)行的 for 循环实现了对对象 *o* 的扫描。我们检查每个在 *o* 中被引用的对象 *o'*。如果 *o'* 已经被访问过(其 *reached* 位为 1)，那么就不需要对 *o'* 做任何处理；它要么已经在之前被扫描过，要么已经在 *Unscanned* 列表中等待扫描。然而，如果 *o'* 还没有被访问到，那么我们需要在第(6)行将它的 *reached* 位设置为 1，并在第(7)行中将 *o'* 加入到 *Unscanned* 列表中。图 7-22 说明了这个过程。它显示了一个带有四个对象的 *Unscanned* 列表。列表中的第一个对象对应于上述讨论中的对象 *o*。它正在被扫描。虚线对应于可能从 *o* 到达的三种类型的对象：

1) 之前扫描过的对象，它不需要被再次扫描。

2) 当前在 *Unscanned* 列表中的对象。

3) 一个可达的数据项，但是之前它被认为是未被访问的。

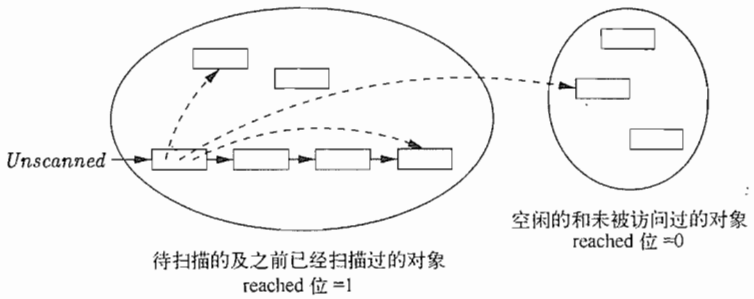


图 7-22 一个标记 - 清扫式垃圾回收器的标记阶段中对象之间的关系

第(8)行到第(11)行是清扫阶段，它收回所有那些在标记阶段结束之后仍然未被访问到的对象的空间。请注意，这些对象将包括所有原本就在 *Free* 列表中的对象。因为无法直接枚举不可达对象的集合，这个算法将清扫整个堆区。第(10)行将空闲且不可达的对象逐个放入 *Free* 列表。第(11)行处理可达对象。我们将它们的 *reached* 位设为 0，以便在这个垃圾回收算法下一次运行时，其前置条件得到满足。 □

7.6.2 基本抽象

所有基于跟踪的算法都计算可达对象集合，然后取这个集合的补集。因此，内存是按照下列方式循环使用的：

1) 程序(或者说增变者)运行并发出分配请求。

2) 垃圾回收器通过跟踪揭示可达性。

3) 垃圾回收器收回不可达对象的存储空间。

图 7-23 按照存储块的四种状态(空闲的、未被访问的、待扫描的和已扫描的)说明这个循环。一个存储块的状态可以存储在该块内部，也可以使用垃圾回收算法的某个数据结构隐含地表示。

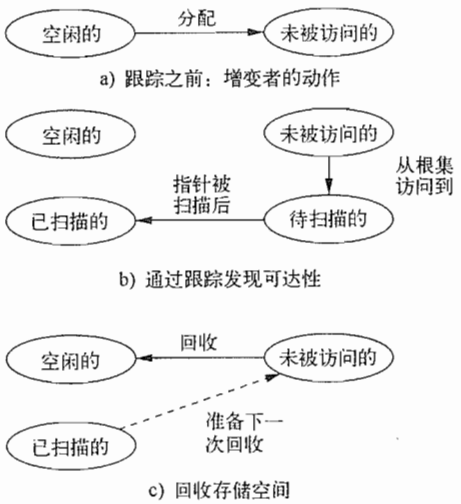


图 7-23 在一个垃圾回收循环中的存储块的状态

虽然不同的基于跟踪的算法可能在实现方法上有所不同，但是它们都可以通过下列状态进行描述：

1）空闲的。存储块处于空闲状态表示它可以被分配。因此，一个空闲块内不会存放任何可达对象。

2）未被访问的。除非通过跟踪证明存储块可达，否则它被默认为是不可达的。在垃圾回收过程中的任何时刻，如果还没有确定一个块的可达性，该块就处于未被访问的状态。如图 7-23a 所示，当一个存储块被存储管理器分配出去时，它的状态就被设置为未被访问的。一轮垃圾回收之后，可达对象的状态仍然会被重置为未被访问状态，以准备下一轮处理，参见图中从已扫描状态到未被访问状态的转换。这个转换用虚线显示，以强调它是为下一轮处理做准备。

3）待扫描的。已知可达的存储块要么处于待扫描状态，要么处于已扫描状态。如果已知一个存储块是可达的，但是该块中的指针还没被扫描，那么该块就处于待扫描状态。当我们发现某个块可达时，就会发生一个从未被访问状态到待扫描状态的转换，如图 7-23b 所示。

4）已扫描的。每个待扫描对象最终都将被扫描并转换到已扫描状态。在扫描一个对象时，我们检查其内部的各个指针，并且沿着这些指针找到它们引用的对象。如果引用指向一个未被访问的对象，那么该对象将被设为待扫描状态。当对一个对象的扫描结束时，这个对象被放入已扫描状态，见 7-23b 中下面的转换。一个已扫描的对象只能包含指向其他已扫描或待扫描对象的引用，决不会包含指向未被访问对象的引用。

当不再有对象处于待扫描状态时，可达性的计算就完成了。到最后仍然处于未被访问状态的对象确实是不可达的。垃圾回收器收回它们占用的空间，并将这些存储块置于空闲的状态，如图 7-23c 中实线转换所示。为了准备下一轮垃圾回收，处于已扫描状态中的对象将回到未被访问状态，见图 7-23c 中的虚线转换。再次提醒大家，这些对象现在确实是可达的。将它们设定为未被访问状态是正确的，因为当下一轮垃圾回收开始时，我们将要求所有对象都从这个状态出发。在那个时候，当前可达的某些对象可能实际上已经被变成了不可达的。

例 7.13 我们看一下算法 7.12 中的数据结构与上面介绍的四种状态有什么关系。使用 reached 位，以及是否在列表 *Free* 和 *Unscanned* 中，我们可以区分全部四种状态。图 7-24 中的表格归纳了用算法 7.12 中的数据结构来刻画四种状态的方式。 □

状态	在列表*Free*中	在*Unscanned*列表中	Reached 位
空闲	是	否	0
未被访问的	否	否	0
待扫描	否	是	1
已扫描	否	否	1

图 7-24 算法 7.12 中状态的表示方式

7.6.3 标记－清扫式算法的优化

基本的标记－清扫式算法的最后一步的代价很大，因为没有一个容易的方法可以不用检查整个堆区就找到所有不可达对象。由 Baker 提出的一个优化算法用一个列表记录了所有已分配的对象。我们必须将不可达对象的存储返回给空闲空间。为了找出不可达对象的集合，我们可以求已分配对象和可达对象之间的差集。

算法 7.14 Baker 的标记－清扫式回收器。

输入： 一个由对象组成的根集，一个堆区，一个空闲列表 *Free*，一个名为 *Unreached* 的已分配对象的列表。

输出： 经过修改的 *Free* 列表和 *Unreached* 列表。*Unreached* 列表保存了被分配的对象。

方法： 这个算法如图 7-25 所示。算法中用于垃圾回收的数据结构是名字分别为 *Free*、

Unreached、*Unscanned*、*Scanned* 的四个列表。这些列表分别保存了处于空闲、未被访问、待扫描和已扫描状态上的所有对象。像 7.4.4 节中讨论的那样，这些列表可以通过嵌入式的双重链表来实现。对象中的 reached 位没有被使用，但是我们假定每个对象中都包含了一些二进制位，指明该对象处于上述四个状态的哪一个。最初，*Free* 就是由存储管理器维护的空闲列表，所有已分配的对象都在 *Unreached* 列表中(这个表同时也由存储管理器在为对象分配存储块时维护)。

```
1)  Scanned = ∅;
2)  Unscanned = 在根集中引用的对象的集合；并将这些对象从Unreached中删除；
3)  while (Unscanned ≠ ∅) {
4)      将对象从 Unscanned 移动到 Scanned;
5)      for (在 o 中引用的每个对象 o′) {
6)          if (o′ 在 Unreached 中)
7)              将o′从 Unreached 移动到Unscanned 中;
        }
    }
8)  Free = Free ∪ Unreached;
9)  Unreached = Scanned;
```

图 7-25 Baker 的标记 - 清扫式算法

第(1)、(2)行将 *Scanned* 列表初始化为空列表，并将 *Unscanned* 列表初始化为仅包含那些可以从根集访问的对象。值得注意的是，这些对象本来都在列表 *Unreached* 中，现在它们必须从该列表中删除。第(3)行到第(7)行是一个使用这些列表的基本标记 - 清扫式算法的简单实现。也就是说，第(5)行到第(7)行的 for 循环检查了一个待扫描对象 o 中的所有引用，如果这些引用中的某一个 o'还没有被访问过，则第(7)行将 o'改变为*待扫描*状态。

然后，第(8)行处理所有仍然在 *Unreached* 列表中的对象，将它们移到 *Free* 列表中，从而回收它们的存储块。然后，第(9)行处理所有处于*已扫描*状态的对象，即所有的可达对象，并将 *Unreached* 列表重新初始化，使之恰好包含这些对象。我们假设，当存储管理器创建新对象时，它们同样会被移出 *Free* 列表，加入到 *Unreached* 列表中。 □

在本节介绍的两个算法中，我们都假设返回给空闲列表的存储块仍然保持被回收前的样子。然而，如 7.4.4 节中讨论的，将相邻的空闲块合并成较大的块常常会带来好处。如果我们想这样做，那么在图 7-21 的第(10)行或图 7-25 的第(8)行上，每次我们将一个存储块放入空闲列表时，我们检查该块的左端和右端，如果有一端为空闲就进行合并。

7.6.4 标记并压缩的垃圾回收器

进行重新定位(relocating)的垃圾回收器会在堆区内移动可达对象以消除存储碎片。通常，可达对象占用的空间要大大小于空闲空间。因此，在标记出所有的“窗口”之后并不一定要逐个释放这些空间，另一个有吸引力的做法是将所有可达对象重新定位到堆区的一端，使得堆区的所有空闲空间成为一个块。毕竟垃圾回收器已经分析了可达对象中的每个引用，因此更新这些引用使之指向新的存储位置并不需要增加很多工作量。我们需要改变的全部引用包括可达对象中的引用和根集中的引用。

将所有可达对象放在一段连续的位置上可以减少内存空间的碎片，使得它更容易存储较大的对象。同时，通过使数据占用更少的缓存线和内存页，重新定位可以提高程序的时间局部性和空间局部性，因为几乎同时创建的对象将被分配在相邻的存储块中。如果这些相邻的块中的对象一起使用，那么就可以从数据预取中得到好处。不仅如此，用以维护空闲空间的数据结构也可以得到简化。我们不再需要一个空闲空间列表，需要的只是一个指向唯一空闲块的起始位置的指针 *free*。

存在多种进行重新定位的回收器，其不同之处在于它们是在本地进行重新定位，还是在重新定位之前预留了空间：

- 本节描述的*标记并压缩回收器*(mark-and-compact collector)在本地压缩对象。在本地重新定位可以降低存储需求。
- 7.6.5 节中给出了更高效、更流行的*拷贝回收器*(copying collector)，它把对象从内存的一个区域移到另一个区域。保留额外的空间用于重新定位可以使得一发现可达对象就立刻移动它。

算法 7.15 中的标记并压缩垃圾回收器有 3 个阶段：

1）首先是标记阶段，它和前面描述的标记－清扫式算法的标记阶段类似。

2）在第二阶段，算法扫描堆区中的已分配内存段，并为每个可达对象计算新的地址。新地址从堆的最低端开始分配，因此在可达对象之间没有空闲存储窗口。每个对象的新地址记录在一个名为 *NewLocation* 的结构中。

3）最后，算法将对象拷贝到它们的新地址，更新对象中的所有引用，使之指向相应的新地址。新的地址可以在 *NewLocation* 中找到。

算法 7.15 一个标记并压缩的垃圾回收器。

输入：一个由对象组成的根集，一个堆，以及一个标记空闲空间的起始位置的指针 *free*。

输出：指针 *free* 的新值。

方法：图 7-26 给出了这个算法，此算法使用下列的数据结构：

1）一个 *Unscanned* 列表，同算法 7.12 中的 *Unscanned* 列表。

2）所有对象的 reached 位也和算法 7.12 中相同。为了使我们的描述简单，当我们要说一个对象的 reached 位为 1 或 0 时，我们分别称它们为“已被访问的”或“未被访问的”。在初始时刻，所有的对象都是未被访问的。

3）指针 *free*，标记了堆区中未分配空间的开始位置。

4）*NewLocation* 表。这个结构可以是任意一个实现了如下两个操作的散列表、搜索树或其他数据结构：

① 将 *NewLocation*(*o*)设为对象 *o* 的新地址。

② 给定对象 *o*，得到 *NewLocation*(*o*)的值。

```
    /* 标记 */
1)  Unscanned = 根集引用的对象的集合;
2)  while (Unscanned ≠ ∅) {
3)      从 Unscanned 中移除对象o;
4)      for (在o中引用的每个对象o') {
5)          if (o'是未被访问的) {
6)              将o'标记为已被访问的;
7)              将o'加入到列表Unscanned中;
            }
        }
    }
    /* 计算新的位置 */
8)  free = 堆区的开始位置;
9)  for (从低端开始，遍历堆区中的每个存储块o) {
10)     if (o是已被访问的{
11)         NewLocation(o) = free;
12)         free = free + sizeof(o);
        }
    }
    /* 重新设置引用目标并移动已被访问的对象 */
13) for (从低端开始，堆区中的每个存储块o) {
14)     if (o是已被访问的) {
15)         for (o中的每个引用o.r)
16)             o.r = NewLocation(o.r);
17)         将o拷贝到NewLocation(o);
        }
    }
18) for (根集中的每个引用r)
19)     r = NewLocation(r);
```

图 7-26 一个标记并压缩回收器

我们不会关心到底使用了什么样的数据结构，虽然你可以假设 *NewLocation* 是一个散列表，因此“set”和“get”操作所需要的平均时间为某个常量，这个时间和堆区内的对象数量无关。

第(1)行到第(7)行的第一(或标记)阶段在本质上和算法 7.12 的第一阶段相同。第二阶段是从第(8)行到第(12)行。该阶段从左边(或者说从低地址端)开始访问堆中的已分配部分的每一个存储块。结果，被分配给存储块的新地址与它们的老地址按照同样的顺序增长。这个顺序很重要，它可以保证我们在重新定位对象时总是将对象向左移，那么在移动时，原来占据目标空间的对象已经被我们移走了。

第(8)行首先将 *free* 指针设定为指向堆区的低端。在这个阶段，我们使用 *free* 来指示第一个可用的新地址。我们只会为标记为已被访问的对象 *o* 创建新的地址。在第(10)行中，对象 *o* 被赋予下一个可用地址；在第(11)行，我们根据对象 *o* 需要的存储数量增加 *free* 指针，因此 *free* 仍然指向空闲空间的开始位置。

从第(13)行到第(17)行是最后阶段，此时我们再次按照第二阶段中的自左向右的顺序访问可达对象。第(15)、(16)行将一个已被访问到的对象 *o* 的所有内部指针替换为它们的新地址，*NewLocation*表用来确定这个新的地址。然后，第(17)行将内部引用已被更新的对象 *o* 移动到新的位置。最后，第(18)和(19)行重新确定根集元素中的指针指向的目标，这些元素本身不是堆区对象，它们可能是静态分配对象或栈分配对象。图 7-27 说明了如何将可达对象(图中无阴影的对象)移动到堆区的底部，同时内部指针被修改，指向已被访问对象的新位置。 □

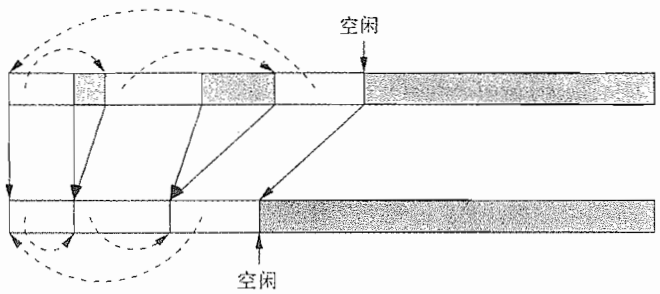


图 7-27 将已被访问对象移动到堆的前部，同时保持内部指针的指向关系

7.6.5 拷贝回收器

拷贝回收器预先保留了可以将对象移入的空间，因而解除了跟踪和发现空闲空间之间的依赖关系。整个存储空间被划分为两个半空间(semispace)A 和 B。增变者在半空间之一(比如 A)内分配内存，直到它被填满。此时增变者停止，垃圾回收器将可达对象拷贝到另一个半空间，比如说 B。当垃圾回收完成时，两个半空间的角色进行对换。增变者可以继续运行，并在半空间 B 中分配对象。下一轮垃圾回收将把可达对象移动到 A。下面的算法是由 C. J. Cheney 提出的。

算法 7.16 Cheney 的拷贝回收器。

输入： 一个由对象组成的根集，一个包含了 *From* 半空间和 *To* 半空间的堆区，其中 *From* 半空间包含了已分配对象，*To* 半空间全部是空闲的。

输出： 最后，*To* 半空间保存已分配的对象。*free* 指针指明了 *To* 半空间中剩余空闲空间的开始位置。*From* 半空间此时全部空闲。

方法： 图 7-28 显示了这个算法。Cheney 算法在 *From* 半空间中找出可达对象，并且访问到它们时立刻把它们拷贝到 *To* 半空间。这种放置方法将相关对象放在一起，从而提高空间局部性。

在探讨算法本身(即图 7-28 中的函数 *CopyingCollector*)之前，首先考虑第(11)行到第(16)行的辅助函数 *LookupNewLocation*。该函数的输入是一个对象 *o*，如果 *o* 在 *To* 空间中还没有对应的位置，则为其分配一个 *To* 空间中的新地址。所有新地址都被记录在一个结构 *NewLocation* 中，特殊

值 Null 用来表示还没有为 o 分配空间[⊖]。和算法 7.15 一样，*NewLocation* 结构的具体形式可以变化，但是现在假设它是一个哈希表就行了。

如果我们在第(12)行发现 o 没有存储位置，那么在第(13)行上它将被赋予 *To* 半空间中空闲空间的开始位置。第(14)行使 *free* 指针增加 o 所占的空间数量。在第(15)行，我们将 o 从 *From* 空间拷贝到 *To* 空间。因此，对象从一个半空间到另一个半空间的移动实际上是一个函数的副作用。这个副作用发生在我们第一次为这个对象寻找新地址的时候。不管之前有没有设定对象 o 的位置，第(16)行返回 o 在 *To* 空间中的位置。

```
 1)  CopyingCollector () {
 2)        for (From空间中的所有对象o) NewLocation(o) =NULL;
 3)        unscanned = free = To空间的开始地址;
 4)        for (根集中的每个引用r)
 5)              将r替换为LookupNewLocations(r);
 6)        while (unscanned ≠ free) {
 7)              o = 在unscanned所指位置上的对象;
 8)              for (o中的每个引用o.r)
 9)                    o.r = LookupNewLocation(o.r);
10)             unscanned = unscanned + sizeof(o);
          }
    }

    /* 如果一个对象已经被移动过了，查找这个对象的新位置 */
    /* 否则将对象设置为待扫描状态 */
11) LookupNewLocation(o) {
12)       if (NewLocation(o) = NULL) {
13)             NewLocation(o) = free;
14)             free = free + sizeof(o);
15)             将对象o拷贝到NewLocation(o);
          }
16)       return NewLocation(o);
    }
```

图 7-28 一个拷贝垃圾回收器

现在我们可以考虑这个算法本身了。第(2)行确保 *From* 空间中的所有对象都还没有新地址。在第(3)行中，我们初始化两个指针 *unscanned* 和 *free*，使它们都指向 *To* 半空间的开始位置。指针 *free* 将总是指向 *To* 半空间中空闲空间的起始位置。当我们往 *To* 空间加入对象时，那些地址低于 *unscanned* 的对象将处于*已扫描*状态，而那些位于 *unscanned* 和 *free* 之间的对象则处于*待扫描*状态。因此，*free* 总是在 *unscanned* 的前面。当后者追上前者时就表示不存在更多的*待扫描*对象了，我们就完成了垃圾回收工作。请注意，我们是在 *To* 空间中完成垃圾回收工作的，尽管在第(8)行中检查的对象中的所有引用都是指向 *From* 空间的。

第(4)行和第(5)行处理可以从根集访问到的对象。请注意，因为函数副作用，在第(5)行中对 *LookupNewLocation* 的某些调用会在 *To* 中为这些对象分配存储块，同时增加 *free* 指针的值。因此，除非没有被根集引用的对象(在这种情况下，整个堆区都是垃圾)，当程序第一次运行到这里时将进入第(6)行到第(10)行的循环。然后，这个循环扫描所有已经被加入到 *To* 空间中并处于*待扫描*状态的对象。第(7)行处理下一个待扫描的对象 o。在第(8)、(9)行，对于 o 中的每个引用，从它在 *From* 半空间中的原值被翻译为在 *To* 半空间中的值。请注意，因为函数副作用，如果

⊖ 在一个典型的数据结构中(如散列表)，如果 o 没有被赋予一个位置，那么在这个结构中就没有相关信息。

o 内的某个引用所指向的对象之前还没有被访问过，那么第(9)行中对 *LookupNewLocation* 的调用将在 *To* 空间中为这个对象分配空间并将它移到该空间中。最后，第(10)行增加指针 *unscanned* 的值，使之指向下一个对象，即 *To* 空间中 o 之后的对象。 □

7.6.6 开销的比较

Cheney 算法的优势在于它不会涉及任何不可达对象。另一方面，拷贝垃圾回收器必须移动所有可达对象的内容。对于大型对象，或者那些经历了多轮垃圾收集过程的生命周期长的对象而言，这个过程的开销特别高。我们对本节给出的四种算法的运行时间进行总结。下面的每个估算都忽略了处理根集的开销。

- 基本的标记－清扫式算法(算法 7.12)：与堆区中存储块的数目成正比。
- Baker 的标记－清扫式算法(算法 7.14)：与可达对象的数目成正比。
- 基本的标记并压缩算法(算法 7.15)：与堆区中存储块的数目和可达对象的总大小成正比。
- Cheney 的拷贝回收器(算法 7.16)：与可达对象的总大小成正比。

7.6.7 7.6 节的练习

练习 7.6.1：当下列事件发生时，给出标记－清扫式垃圾回收器的处理步骤。

1) 图 7-19 中指针 $A \rightarrow B$ 被删除。

2) 图 7-19 中指针 $A \rightarrow C$ 被删除。

3) 图 7-20 中指针 $A \rightarrow D$ 被删除。

4) 图 7-20 中对象 B 被删除。

练习 7.6.2：Baker 的标记－清扫式算法在四个列表 *Free*、*Unreached*、*Unscanned* 和 *Scanned* 之间移动对象。对于练习 7.6.1 中的每个对象网络中的每个对象，指出从垃圾回收过程刚开始到该过程刚结束的时间段内，该对象所经历的列表的序列。

练习 7.6.3：假设我们在练习 7.6.1 中的各个网络上执行了一个标记并压缩垃圾回收过程。同时假设

1) 每个对象的大小是 100 个字节。

2) 在开始时刻，堆区中的 9 个对象按照字母顺序从堆区的第 0 个字节开始排列。

在垃圾回收过程结束之后，各个对象的地址是什么？

练习 7.6.4：假设我们在练习 7.6.1 中的各个网络上执行了 Cheney 的拷贝垃圾回收算法。同时假设

1) 每个对象的大小为 100 字节。

2) 待扫描的列表按照队列的方式进行管理，并且当一个对象具有多个指针时，被访问到的对象按照字母顺序被加入到队列中。

3) *From* 半空间从位置 0 开始，*To* 半空间从位置 10 000 开始。

在垃圾回收完成之后，每个保留下来的对象 o 的 *NewLocation*(o)的值是什么？

7.7 短停顿垃圾回收

简单的基于跟踪的回收器是以全面停顿的方式进行垃圾回收的，它可能造成用户程序的运行的长时间的停顿。我们可以每次只做部分垃圾回收工作，从而减少一次停顿的长度。我们可以按照时间来分割工作任务，使垃圾回收和增变者的运行交错进行。我们也可以按照空间来分割工作任务，每次只完成一部分垃圾的回收。前者称为*增量式回收*(incremental collection)，后者称为*部分回收*(partial collection)。

增量式回收器将可达性分析任务分割成为若干个较小单元，并允许增变者和这些任务单元交错运行。可达集合会随着增变者的运行发生变化，因此增量式回收是很复杂的。我们将在7.7.1节看到，寻找一个稍微保守的解决方法将使得跟踪更加高效。

最有名的部分回收算法是*世代垃圾回收*(generational garbage collection)。它根据对象已分配时间的长短来划分对象，并且较频繁地回收新创建的对象，因为这些对象的生命周期往往较短。另一种可选的算法是*列车算法*(train algorithm)，也是每次只回收一部分垃圾。它最适合回收较成熟的对象。这两个算法可以联合使用，构成一个部分回收器。这个回收器使用不同的方法来处理较新的和较成熟的对象。我们将在7.7.3节讨论有关部分回收的基本算法，然后详细地描述世代算法和列车算法的工作原理。

来自于增量回收算法和部分回收算法的思想经过修改，可以用于构造一个在多处理器系统中并行回收对象的算法，见7.8.1节。

7.7.1 增量式垃圾回收

增量式回收器是保守的。虽然垃圾回收器一定不能回收不是垃圾的对象，但是它并不一定要在每一轮中回收所有的垃圾。我们将每次回收之后留下的垃圾称为*漂浮垃圾*(floating garbage)。我们当然期望漂浮垃圾越少越好。明确地说，增量式回收器不应该遗漏那些在回收周期开始时就已经不可达的垃圾。如果我们能够保证做到这一点，那么在某一轮中没有被回收的垃圾一定会在下一轮中被回收。因此不会因为这个垃圾回收方法而产生内存泄漏问题。

换句话说，增量式垃圾回收器会过多地估算可达对象集合，从而保证安全性。它们首先以不可中断的方式处理程序的根集，此时没有来自增变者的干扰。在找到了待扫描对象的初始集合之后，增变者的动作与跟踪步骤交错进行。在这个阶段，任何可能改变可达性的增变者动作都被简洁地记录在一个副表中，使得回收器可以在继续执行时做出必要的调整。如果在跟踪完成之前空间就被耗尽，那么回收器将不再允许增变者执行，并完成全部跟踪过程。在任何情况下，当跟踪完成后，空间回收以原语的方式完成。

增量回收的准确性

一旦对象成为不可达的，该对象就不可能再变成可达的。因此，在垃圾回收和增变者运行时，可达对象的集合只可能：

1）因为垃圾回收开始之后的某个新对象的分配而增长。

2）因为失去了指向已分配对象的引用而缩小。

令垃圾回收开始时的可达对象集合为 R，令 New 表示在垃圾回收期间创建并分配的对象集合，并令 $Lost$ 表示在跟踪开始之后因为引用丢失而变得不可达的对象的集合。那么当跟踪完成之后，可达对象的集合为：

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

如果在每次增变者丢失了一个指向某个对象的引用之后都重新确定该对象的可达性，那么开销会变得很大，因此增量式回收器并不试图在跟踪结束时回收所有的垃圾。任何遗留下的垃圾——漂浮垃圾——应该是 $Lost$ 对象的一个子集。如果形式化地描述，那通过跟踪找到的对象集合 S 必须满足

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

简单的增量式跟踪

我们首先描述一种用来找到集合 $R \cup New$ 的上界的简单跟踪算法。在跟踪期间，增变者的行为更改如下：

- 在垃圾回收开始之前已经存在的所有引用都被保留。也就是说，在增变者覆写一个引用

之前，它原来的值被记住，并被当作一个只包含这个引用的附加待扫描对象。

- 所有新创建的对象立即就被认为是可达的，并被放置在*待扫描*状态中。

这种方案是保守且正确的，因为它找出了 R 和 New。R 是在垃圾回收之前可达的所有对象的集合，New 是所有新分配的对象的集合。然而，这种方案付出的代价也很高，因为算法需要拦截所有的写运算，并记住所有被覆写的引用。这些工作中的一部分是不必要的，因为它涉及的对象在垃圾回收结束时可能已经是不可达的。如果我们能够探测到哪些被覆写的引用所指的对象在本轮垃圾回收结束时不可达，我们就可以避免这部分工作，同时还可以提高算法的准确性。下一个算法在这两个方面都做了很好的改进。

7.7.2 增量式可达性分析

如果我们让增变者和一个像算法 7.12 那样的基本跟踪算法交替执行，那么一些可达对象可能会被错认为是不可达的。问题的根源在于增变者的动作可能会违反这个算法的一个关键不变式，即一个*已扫描*对象中的引用只能指向*已扫描*或*待扫描*的对象，这些引用不可以指向*未被访问*对象。考虑下面的场景：

1）垃圾回收器发现对象 o_1 可达并扫描 o_1 中的指针，因而将 o_1 置于*已扫描*状态。

2）增变者将一个指向*未被访问*（但可达）的对象 o 的引用存放到*已扫描*对象 o_1 中。它从当前处于*未被访问*或*待扫描*状态的对象 o_2 中将一个指向 o 的引用拷贝到 o_1 中。

3）增变者失去了对象 o_2 中指向 o 的引用。它可能已经在扫描 o_2 中指向 o 的引用之前就覆写了这个指针；也可能 o_2 已经变得不可达，因此一直没有进入*待扫描*状态，因此它内部的指针没有被扫描过。

现在，o 可以通过对象 o_1 到达，但是垃圾回收器可能既没有看到 o_1 中指向 o 的引用，也没有看到 o_2 中指向 o 的引用。

要得到一个更加准确且正确的增量式跟踪方法，关键在于我们必须注意所有将一个指向当前未被访问对象的引用从一个尚未扫描的对象中拷贝到已扫描对象中的动作。为了截获可能有问题的引用传递，算法可以在跟踪过程中按照下列方式修改增变者的动作：

- *写关卡*。截获把一个指向*未被访问*的对象 o 的引用写入一个*已扫描*对象 o_1 的运算。在这种情况下，将 o 作为可达对象并将其放入*待扫描*集合。另一种方法是将被写对象 o_1 放回到*待扫描*集合中，使得我们可以再次扫描它。
- *读关卡*。截获对*未被访问*或*待扫描*对象中的引用的读运算。只要增变者从一个处于*未被访问*或*待扫描*状态中的对象读取一个指向对象 o 的引用时，就将 o 设为可达的，并将其放入*待扫描*对象的集合。
- *传递关卡*。截获在*未被访问*或*待扫描*对象中原引用丢失的情况。只要增变者覆写一个*未被访问*或*待扫描*对象中的引用时，保存即将被覆写的引用并将其设为可达的，然后将这个引用本身放入*待扫描*集合。

上述几种做法都不能找到最小的可达对象集合。如果跟踪过程确定一个对象是可达的，那么这个对象就一直被认为是可达的。即使在跟踪过程结束之前所有指向它的引用都被覆写，它仍然被认为是可达的。也就是说，找到的可达对象集合介于 $(R \cup New) - Lost$ 与 $(R \cup New)$ 之间。

上面给出的可选算法中写关卡方法是最有效的。读关卡方法的代价较高，因为一般来说读运算要比写运算多得多。转换关卡没有什么竞争力，因为很多对象“英年早逝”，这种方法会保留很多的不可达对象。

写关卡的实现

我们可以用两种方式来实现写关卡。第一种方式是在增变阶段记录下所有被写入到*已扫描*

对象中的新引用。我们可以将这些引用放入一个列表。如果不考虑从列表中剔除重复引用，列表的大小和对*已扫描*对象的写运算的数量成正比。注意，列表中的引用本身可能在后来又被覆写掉，因此可能被忽略。

第二种，也是更有效的方式是记住写运算发生的位置。我们可以用被写位置的列表来记录它们，其中可能会消除重复的位置。请注意，只要所有被写的位置都被重新扫描，那么是否精确记录被写的位置并不重要。因此，有多种技术支持我们记录较少的有关被覆写的确切位置的细节。

- 我们可以只记录包含了被写字段的对象，而不需要记录被写的精确地址或者被写的对象及字段。
- 我们可以将地址空间分成固定大小的块，这些块被称为卡片(card)，并使用一个位数组来记录曾经被写入的卡片。
- 我们可以选择记录下包含了被写位置的页。我们可以只将那些包含了*已扫描*对象的页置为被保护状态。那么，不需执行任何显式的指令就可以检测到任何对*已扫描*对象的写运算。因为这样的写运算会引发一个保护错误，操作系统将引发一个程序异常。

一般来说，通过增大被覆写位置的记录粒度就可以减少所需的存储空间，但代价是增加了需要再次执行的扫描工作量。在第一种方案中，无论实际上修改了被修改对象中的哪个引用，该对象中的所有引用都要进行重新扫描。在后两种方案中，在被修改的卡片或页中的所有可达对象都要在跟踪过程的最后进行重新扫描。

结合增量和拷贝技术

上述的方法对于标记－清扫式垃圾回收来说已经足够了。因为拷贝回收和增变者的相互影响，它的实现要稍微复杂一点。处于*已扫描*或*待扫描*状态中的对象有两个地址，一个位于 *From* 半空间，另一个位于 *To* 半空间。和算法 7.16 一样，我们必须保存一个从对象的旧地址到其重新定位之后的地址的映射。

我们可以选择两种更新引用的方法。第一种方法是，我们可以让增变者在 *From* 空间中完成所有的运算，只是在垃圾回收结束的时候才更新所有的指针，并将所有的内容都拷贝到 *To* 空间。第二种方法是，我们可以让程序直接改变 *To* 空间中的表示。当增变者对一个指向 *From* 空间的指针解引用时，如果在 *To* 空间中存在对应于该指针的新位置，那么这个指针就被翻译成这个新位置。所有这些指针在最后都需要被转换成指向 *To* 空间的新位置。

7.7.3 部分回收概述

一个基本的事实是，对象通常“英年早逝”。人们发现，通常 80% ~98% 的新分配对象在几百万条指令之内，或者在再分配了另外的几兆字节之前就消亡了。也就是说，对象通常在垃圾回收过程启动之前就已经变得不可达了。因此，频繁地对新对象进行垃圾具有相当高的性价比。

然而，经历了一次回收的对象很可能在多次回收之后依然存在。在迄今为止描述的垃圾回收器中，同一个成熟对象会在各轮垃圾回收中被发现是可达的。如果使用拷贝回收器，这些对象会在各轮垃圾回收中被一次次地拷贝。世代回收在包含最年轻对象的堆区域中的回收工作最为频繁，所以它通常可以用相对较少的工作量回收大量的垃圾。另一方面，列车算法没有在年轻对象上花费太多的时间，但是它能够有效限制因垃圾回收而造成的程序停顿时间。因此，将这两个策略合并的好方法是对年轻对象使用世代回收，而一旦一个对象变得相当成熟，则将它“提升”到一个由列车算法管理的独立堆区中。

我们把将在一轮部分回收中被回收的对象集合称为目标(target)集，而将其他对象称为稳定(stable)集。在理想状态下，一个部分回收器应该回收目标集中所有无法从根集到达的对象。然而，这么做需要跟踪所有的对象，而这正是我们首先要试图避免的事情。实际上，部分回收器只

是保守地回收那些无法从根集和稳定集到达的对象。因为稳定集中的一些对象自身也是不可达的，我们可能会把目标集中一些实际上不存在从根集开始的路径的对象当成可达对象。

我们可以修改7.6.1节和7.6.4节中描述的垃圾回收器，改变“根集”的定义，使之以部分回收的方式工作。现在根集指的不仅是存放在寄存器、栈和全局变量中的对象，它还包括所有指向目标集对象的稳定集中的对象。从一个目标对象指向其他目标对象的引用按照以前的方法进行跟踪，以找到所有的可达对象。我们可以忽略所有指向稳定对象的指针，因为在本轮部分回收中这些对象被认为是可达的。

为了找出那些引用了目标对象的稳定对象，我们可以采用和增量垃圾回收所用技术类似的方法。在增量回收中，我们需要在跟踪过程中记录所有对从已扫描对象到未被访问对象的引用的写运算。在这里，我们需要记录下增变者的整个运行过程中对从稳定对象到目标对象的引用的写运算。只要增变者将一个指向某个目标对象的引用保存到稳定对象中时，我们要么记录下这个引用，要么记录下写入的位置。我们把保存了从稳定对象到目标对象的引用的对象集合称为被记忆集合(remembered set)。如7.7.2节中讨论的，我们可以只记录下包含了被写入对象所在的卡片或页，以压缩被记忆集合的表示。

部分垃圾回收器通常被实现为拷贝垃圾回收器。通过使用链表来跟踪可达对象，也可以实现成为非拷贝回收器。下面描述的“世代”方案是一个关于如何将拷贝和部分回收相结合的例子。

7.7.4 世代垃圾回收

世代垃圾回收(generational garbage collection)是一种充分利用了大多数对象“英年早逝”的特性的有效方法。在世代垃圾回收中，堆区被分成一系列小的区域。我们将用0，1，2，…，n对它们进行编号，序号越小的区域存放的对象越年轻。对象首先在0区域被创建。当这个区域被填满时，它的垃圾被回收，且其中的可达对象被移到1区。现在，0区又成为空的，我们继续把新对象分配到这个区域。当0区再次被填满[⊖]，它的垃圾又被回收，且它的可达对象被拷贝到1区，与之前被拷贝的对象合在一起。这个模式一直被重复，直到1区也被填满为止。此时应对0区和1区应用垃圾回收。

一般来说，每一轮垃圾回收都是针对序号小于等于某个i的区域进行的，应该将i选择为当前被填满区域的最高编号。每当一个对象经历了一轮回收(即它被确定为可达的)，它就从它当前所在区域被提升到下一个较高的区域，直到它到达最老的区域，即序号为n的区域。

使用7.7.3节中介绍的术语，当区域i及更低区域中的垃圾被回收时，从0到i的区域组成了目标集，所有序号大于i的区域组成了稳定集。为了为各种可能的部分回收找到根集，我们为每个区域i保持了一个被记忆集，该集合由指向区域i中对象且位于大于i的区域中的所有对象组成。在i上激活的一次部分回收的根集包括了区域i及更低区域的被记忆集。

在这个方案中，只要我们对i进行回收，所有序号小于i的区域也将进行垃圾回收。有两个原因促使我们采用这个策略：

1）因为较年轻的世代往往包含较多的垃圾，也就更频繁地被回收。所以，我们可以将它们和较老的世代一起回收。

2）根据这种策略，我们只需要记录从较老世代指向较新世代的引用。也就是说，对最年轻世代的对象进行写运算，以及将对象提升到下一世代时都不需要更新任何被记忆集。如果我们

⊖ 从技术上来说，区域不会被填满，因为如果需要，存储管理器可以使用附加的磁盘块对它们进行扩展。然而，除了最后一个区域，其他区域的尺寸通常都有一个界限。我们将把到达这一界限称为“填满”。

对某个区域进行回收，但是不回收某个较年轻的世代，那么后者将成为稳定集的一部分。我们将不得不同时记录从较年轻世代指向较年老世代的引用。

总而言之，这种方案更频繁地回收较年轻的世代，并且因为“对象英年早逝”，对于这些世代进行垃圾回收的效费比特别高。对较老世代的垃圾回收则要花更多的时间，因为它包括了对所有较年轻世代的回收，同时它们包含的垃圾也相应减少。虽然如此，较老世代还是需要每过一段时间进行一次回收，以删除不可达对象。最老的世代保存了最成熟的对象，对这些对象的回收是最昂贵的，因为它相当于一次完整的回收。也就是说，世代回收器偶尔也需要执行完整的跟踪步骤，因此也会在程序运行时引入较长时间的停顿。接下来将讨论另一种只处理成熟对象的方法。

7.7.5 列车算法

尽管世代方法在处理年轻对象时非常高效，但它在处理成熟对象时却相对低效，因为每当一个垃圾回收过程涉及某个成熟对象时，该对象都会被移动，而且它们不太可能变成垃圾。另一种被称为*列车算法*的增量式回收方法用于改进对成熟对象的处理。它可以用来回收所有的垃圾。但是更好的方法是使用世代方法来处理年轻的对象，只有当这些对象经历了几轮世代回收之后仍然存在，才将它们提升到另一个由列车算法管理的堆区。列车算法的另一个优点是我们永远不需要进行全面的垃圾回收过程，而在世代垃圾回收中却仍然必须偶尔那样做。

为了描述列车算法的动机，我们首先看一个简单的例子。该例子告诉我们为什么在世代方法中必须偶尔进行一轮全面的垃圾回收。图 7-29 给出了位于两个区域 i 和 j 中的两个相互连接的对象，其中 $j > i$。因为这两个对象都有来自其区域之外的指针，只对区域 i 或只对区域 j 进行回收都不能回收这两个对象。然而，它们可能实际上是一个循环垃圾结构中的一部分，没有外部链接指向该垃圾结构。一般来说，这里显示的对象之间的“链接”可能涉及很多对象和一条很长的引用链。

图 7-29 一个跨越区域的可能是循环垃圾的环状结构

在世代垃圾回收中，我们最终会回收区域 j，并且因为 $i < j$，我们同时还会回收 i 区域。那么这个循环结构将被完全包含在正在被回收的堆区中，我们就可以确定它是否真的是垃圾。然而，如果我们从没有进行过一轮包括了 i 和 j 的回收，那么我们就会碰到循环垃圾的问题，也就是我们在使用引用计数进行垃圾回收时碰到的问题。

列车算法使用固定大小的被称为*车厢*(car)的区域。当没有对象比磁盘块更大时，一节车厢可以是一个磁盘块，否则可以将车厢的尺寸设得更大。但是车厢的大小一旦确定就不再变化。多节车厢被组织成*列车*(train)。一辆列车中的车厢数量没有限制，且列车的数量也没有限制。车厢之间按照词典顺序进行排序：首先以列车号排序，在同一列车中则以车厢号排序，如图 7-30 所示。

图 7-30 列车算法中的堆区组织

列车算法有两种回收垃圾的方式：

- 在一个增量式垃圾回收步骤中，按照词典顺序排列的第一节车厢（即尚存的第一辆列车中尚存的第一节车厢）首先被回收。因为我们保留了一个来自该车厢之外的所有指针的“被记忆”列表，所以这一步类似于世代算法中针对第一个区域的回收步骤。这里我们确定出没有任何引用的对象，以及完全包含在这节车厢里的垃圾循环。该车厢中的可达对象总是被移至其他的某个车厢中，因此每个被回收过的车厢都变成空车厢，可以从这辆列车中删除。
- 有时，第一辆列车没有外部引用。也就是说，没有从根集指向该列车中任何车厢的指针，并且各节车厢中的被记忆集中只有来自本列车的其他车厢的引用，没有来自其他列车的引用。在这种情况下，该列车就是一个巨大的循环垃圾集合，我们可以删除整辆列车。

被记忆集

现在我们给出列车算法的细节。每节车厢有一个被记忆集，它由指向该车厢中对象的引用组成，这些引用来自：

1）同一辆列车中序号较高的车厢中的对象，以及

2）序号较高的列车中的对象。

此外，每辆列车有一个被记忆集，它由来自较高序号列车中的引用组成。也就是说，一个列车的被记忆集是它内部的所有车厢的被记忆集的并集，但是不包含列车内部的引用。因此，可以将车厢的被记忆集划分成“内部”（同一列车）和“外部”（其他列车）两个部分，同时表示这两种不同类型的被记忆集。

注意，指向这些对象的引用可以来自各个地方，不只是来自按字典顺序排列的序号较高的车厢。然而，算法中的两种垃圾回收过程分别处理第一辆列车的第一节车厢和整个第一辆列车。因此，当在垃圾回收中需要使用被记忆集的时候，已经没有更早的地方可以有引用到达被处理的车厢或者列车。因此记录下指向较高序号车厢的引用没有什么意义。当然，我们必须认真、正确地管理被记忆集，只要增变者改变了任何对象中的引用，就需要相应地改变被记忆集。

管理列车

我们的目标是找出第一辆列车中所有非循环垃圾的对象。此时，第一辆列车要么只包含了循环垃圾，因此将在下一轮垃圾回收时被回收；要么其中的垃圾不是循环的，那么它的车厢就可以被逐个回收。

因为对一辆列车中的车厢数目没有限制，每当我们需要更多空间时，在原则上我们可以直接向一辆列车中加入新的车厢。但是，我们偶尔也需要创建出新的列车。例如，我们可以设定每创建 k 个对象之后就新建一辆列车。也就是说，当最后一辆列车的最后车厢中还有足够的空间时，新创建的对象一般会被放置在这节车厢中；如果该车厢中没有足够空间，该对象就会被放到一个即将被加到最后一个车厢之后的新车厢中。然而，我们会定期新建一列只有一节车厢的列车，并将新对象放入其中。

单节车厢的垃圾回收

列车算法的核心是我们如何在一轮垃圾回收中处理第一辆列车的第一节车厢。一开始，可达集包括了该车厢中被来自根集的引用指向的对象，以及被该车厢的被记忆集中的引用指向的对象。然后，我们像标记－清扫式回收器那样扫描这些对象，但是不会扫描任何可达的位于被回收车厢之外的对象。在这次跟踪之后，该车厢中的某些对象可能被确定为垃圾。因为无论如何整节车厢都将消失，因此不必回收它们的空间。

然而，该车厢中很可能还有一些可达对象，这些对象必须被移到其他地方。移动一个对象的规则如下：

- 如果被记忆集中有一个来自其他列车的引用(该列车的序号高于被回收车厢所在列车的序号)，那么将这个对象移到这些列车中的某一辆中。如果在发出一个引用的某辆列车中能够找到足够的空间，就将该对象移动这辆列车的某节车厢中。如果找不到空间，它就进入一个新的、最末端的车厢。
- 如果没有来自其他列车的引用，但是存在来自根集或第一辆列车的引用，那么就将此对象移到同一列车中的其他车厢中。如果没有足够空间，就创建一个新的车厢放到列车的末端。如果有可能，挑选有一个指向该对象的引用的车厢，以尽快把循环结构放到同一个车厢中。

在从第一节车厢中移出了所有可达对象之后，我们就可以删除这节车厢。

恐慌模式

上面的规则还存在一个问题。为了保证所有的垃圾最终都会被回收，我们需要保证每辆列车迟早会变成第一辆列车，并且如果这辆列车不是循环垃圾，那么此列车中的所有车厢最后都会被删除，且该列车每次至少会减少一节车厢。然而，根据上面的第二个规则，回收第一辆列车的第一节车厢时可能会产生一个位于最后的新车厢。这个过程不会创建出两个或更多的新车厢，因为第一节车厢中的所有对象一定能够被一起放到最后的新车厢中。然而，是否会出现这种情况，一辆列车的每一个回收步骤都产生一节新车厢，以致于我们永远不能回收完这辆列车，结果永远不能继续处理另一辆列车？

遗憾的是，这种情况是可能出现的。如果我们有一个大型的、循环的非垃圾的结构，并且增变者改变引用的方式使得我们在回收一节车厢时一直没有在被记忆集中看到任何来自较高序号列车的引用，就会出现上述问题。只要在回收一节车厢时有一个对象从这个列车中移出，问题就解决了，因为没有新的对象会被加入到第一辆列车中，所以第一辆列车中的所有对象最终一定会被全部移出。然而，有可能在某个阶段我们根本回收不到任何垃圾，这样就会存在出现循环的风险：有可能一直只对当前的第一辆列车进行垃圾回收。

为了避免出现这个问题，只要我们遇到一个*无效*(futile)垃圾回收，我们就需要改变做法。所谓无效垃圾回收是指，在回收一节车厢时没有一个对象可以作为垃圾删除或者被移动到另一辆列车中。在这种“恐慌模式”下，我们做出两个变化：

1）当指向第一辆列车中的某个对象的某个引用被覆写时，我们将这个引用保留为根集的一个新成员。

2）在进行垃圾回收时，如果第一节车厢中的一个对象有来自根集的引用，其中包括在第1点中设置的哑引用，那么即使该对象没有来自其他列车的引用，我们还是将它移至另一辆列车。只要不是移到第一辆列车，移到哪辆列车并不重要。

按照这个方法，如果有一个指向第一辆列车的对象的引用来自该列车之外，在我们回收每节车厢时都会考虑这些引用，并且最终必然会有一些对象从那辆列车移除。然后，我们就可以脱离恐慌模式，继续正常处理，确保当前的第一辆列车一定要比以前小。

7.7.6 7.7 节的练习

练习 7.7.1：假设图 7-20 中的对象网络由一个增量式算法进行管理。该算法和 Baker 算法一样使用四个列表 *Unreached*、*Unscanned*、*Scanned* 和 *Free*。更明确地说，列表 *Unscanned* 按照队列进行管理。当扫描一个对象时，如果有多个对象要被放进这个列表中，我们按照字母顺序加入它们。同时假设我们使用写关卡来保证没有可达对象被当作垃圾。在开始时，*A* 和 *B* 在 *Unscanned*

列表中，假设下列事件发生：

1）A 被扫描。

2）指针 $A \rightarrow D$ 被覆写为 $A \rightarrow H$。

3）B 被扫描。

4）D 被扫描。

5）指针 $B \rightarrow C$ 被覆写为 $B \rightarrow I$。

假设没有更多的指针被覆写，模拟整个增量式垃圾回收过程。哪些对象是垃圾？哪些对象被放在了列表 *Free* 中？

练习 7.7.2：按照如下假设重复练习 7.7.1：

1）事件(2)和(5)的顺序互换。

2）事件(2)和(5)在(1)、(3)和(4)之前发生。

练习 7.7.3：假设堆区恰好由图 7-30 中显示的三辆列车(共九节车厢)组成(即忽略其中的省略号)。有来自车厢 12、23 和 32 的引用指向车厢 11 中的对象 o。当我们对车厢 11 进行垃圾回收，对象 o 最后在什么地方？

练习 7.7.4：在下列情况下重复练习 7.7.3。假设对象 o

1）只有来自车厢 22 和 31 的引用。

2）没有来自车厢 11 之外的指针。

练习 7.7.5：假设堆区恰好由图 7-30 中显示的三辆列车(共九节车厢)组成(即忽略其中的省略号)。当前我们处于恐慌模式。车厢 11 中的对象 o_1 只有一个来自车厢 12 中的对象 o_2 的引用。这个引用被覆写了。当我们对车厢 11 进行垃圾回收时，o_1 会发生什么事情？

7.8 垃圾回收中的高级论题

我们简要地介绍下面的四个论题，结束我们对垃圾回收的研究：

1）并行环境下的垃圾回收。

2）对象的部分重定位。

3）针对类型不安全的语言的垃圾回收。

4）程序员控制的垃圾回收和自动垃圾回收之间的交互。

7.8.1 并行和并发垃圾回收

当将垃圾回收应用到并发或多处理器机器上运行的应用程序时，这一工作变得更具有挑战性。对于服务器应用，在同一时刻运行成千上万个线程是常有的事情；其中的每个线程都是一个增变者。堆区通常会包含几千兆的存储。

可处理大规模系统的垃圾回收算法必须充分利用系统的多个处理器。如果一个垃圾回收器使用多个线程，我们就称其为*并行的*(parallel)。如果回收器和增变者同时运行，就说它是*并发的*(concurrent)。

我们将描述一个并行的且基本上并发的垃圾回收器。它使用一个并发且并行的阶段来完成大部分的跟踪工作，然后执行一个全面停顿式的步骤来保证找到所有的可达对象并回收存储空间。这个算法在本质上并没有引入新的有关垃圾回收的基本概念，它说明了我们如何将曾经描述的思想组合起来，创造出一个解决并发、并行的垃圾回收问题的完整解决方案。然而，并行执行的本质会带来一些新的实现问题。我们将讨论这个算法如何使用一个相当常见的工作队列模型，在并行计算过程中协调多个线程。

为了理解这个算法的设计思想，我们必须牢记这个问题的规模。即使一个并行应用的根集

也要比普通的应用大很多，它由每个线程的栈、寄存器集和全局可访问变量组成。堆区存储的数量也非常大，可达数据的数量同样也很大。增变过程发生速率也比一般的应用高很多。

为了减少停顿时间，我们可以采用原本为增量式分析而设计的基本思想，使垃圾回收和状态增变过程重叠执行。请回顾一下，正如7.7节所讨论的，一个增量式分析完成下列三个步骤：

1）找到根集。这个步骤通常是以原语方式完成的，即增变者暂时停止运行。

2）增变者的执行和对可达对象的跟踪交替进行。在这个阶段，每次有一个增变者写入一个从已扫描对象指向未被访问对象的引用时，我们都会记录这个引用。如7.7.2节中讨论的，我们可以选择多种粒度来记录这些引用。在本节中，我们将假定使用基于卡片的方案。我们将堆区分成若干被称为“卡片”的区段，并维护一个位映射来指明哪个卡片是脏的(即其中有一个或多个引用被覆写)。

3）再次暂停增变者的运行，重新扫描所有可能保存了指向未被访问对象的引用的卡片。

对于一个大型多线程应用，从根集到达的对象的集合可能非常大。终止所有增变者的执行，然后花费很多时间和空间去访问所有这样的对象是不可行的。同时，因为堆的规模巨大，并且增变线程数量巨大，在将所有对象扫描一次之后，很多卡片都需要重新扫描。此时，值得推荐的做法是并行地扫描其中的某些卡片，同时允许增变者继续并发执行。

为了并行地实现上面第(2)步中的跟踪过程，我们将使用多个垃圾回收线程。这些线程和各个增变者线程并发地运行，以跟踪得到大部分可达对象。然后，为了实现第(3)步，我们暂停执行各个增变者，使用并行线程来保证找到所有的可达对象。

完成第(2)步中跟踪过程的方法是让每个增变者线程在完成其自身工作的同时执行部分垃圾回收工作。另外，我们也使用一些专门用于回收垃圾的线程。一旦垃圾回收过程启动，只要增变者线程执行了某个内存分配操作，它同时也会执行一些跟踪计算。只有当计算机中有空闲的时钟周期时，专用的垃圾回收线程才会投入使用。和增量式分析一样，只要增变者写入了一个从已扫描对象指向未被访问对象的引用，存放这个引用的卡片就被标记为脏的，需要重新扫描。

下面给出一个并行、并发垃圾回收算法的大概描述：

1）扫描每个增变者线程的根集，将所有可以从根集中直接到达的对象设为待扫描状态。完成这一步的最简单的增量式做法是等待一个增变者线程调用内存管理器，如果那时它的根集还没有被扫描，就让它扫描自己的根集。如果所有其他跟踪工作都已经完成，而某个增变者线程还没有调用内存分配函数，那么必须暂停这个线程，扫描它的根集。

2）扫描处于待扫描状态的对象。为了支持并行计算，我们使用一个由固定大小的工作包(work packet)组成的工作队列。每个工作包保存了一些待扫描对象。当发现待扫描对象时，它们就被放置到工作包中。等待工作的线程将从队列中取出这些工作包，并跟踪其中的待扫描对象。这种策略允许在跟踪过程中把工作量平均分配给各个工作线程。如果系统用完了存储空间，使得我们无法找到创建这些工作包所需的空间，就直接为保存这些对象的卡片加上标记，使它们将在以后被扫描。后一种处理方法总是可行的，因为存放卡片标记的位数组已经预先分配好了。

3）扫描脏卡片中的对象。当工作队列中不再有待扫描对象，并且所有线程的根集都已经被扫描过之后，我们重新扫描这些卡片以寻找可达对象。只要增变者继续执行，脏卡片就会不断产生。因此，我们需要依照某种标准来停止跟踪过程。比如只允许卡片被再次扫描一次或固定的次数，或者当未完成扫描的卡片数量减少到某个阈值时停止跟踪。这么做的结果是使得并行和并发步骤通常会在完成全部跟踪工作之前就停止。剩下的工作将在下面介绍的最后一步中完成。

4）最后一步保证所有的可达对象都被标记为已被访问的。随着所有增变者停止执行，使用系统中的所有处理器就可以快速找到所有线程的根集。因为大部分可达对象已经被跟踪确定，

预计只有少量的对象会被放在待扫描状态中。所有的线程都参与了对其余可达对象的跟踪和对所有卡片的重新扫描。

我们必须控制启动跟踪过程的频率,这很重要。跟踪步骤就像是一场赛跑。增变者创建出必须被扫描的新对象和新引用,而跟踪过程则试图扫描所有可达对象,并重新扫描同时产生的脏卡片。在需要进行垃圾回收之前过分频繁地启动跟踪过程是没有必要的,因为这样做将会增加漂浮垃圾的数量。另一方面,我们又不能等到存储耗尽时才开始跟踪过程。因为这时增变者将不能继续运行,此时的情况就退化为使用全面停顿式回收器的情形。因此,算法必须适当地选择启动回收的时机和跟踪的频率。对前面的各轮垃圾回收中的对象增变速率的估算可以帮助我们在这方面做出决策。根据专用垃圾回收线程所做的工作量,可以动态调整跟踪频率。

7.8.2 部分对象重新定位

就像从7.6.4节开始讨论的,拷贝或压缩回收器的优势在于消除碎片。然而,这些回收器需要不小的开销。压缩回收器需要在垃圾回收结束时移动所有的对象并更新所有的引用。拷贝回收器在跟踪过程中就找出可达对象的位置。如果跟踪采用增量式执行方式,我们要么对增变者的每个引用进行转换,要么到最后才移动所有的对象并更新它们的引用。这两种做法都是比较昂贵的,对大型堆区来说尤其如此。

我们可以改用一个拷贝世代垃圾回收器。它在回收年轻对象并减少碎片方面很有效,但是在回收成熟对象时比较昂贵。我们可以使用列车算法来限制每次分析时处理的成熟数据的数量。然而,列车算法的代价和每个区域的被记忆集的大小相关。

有一种混合型的回收方案,它使用并发跟踪来回收所有不可达对象,同时只移动部分对象。这种方法减少了碎片,又不会因为在每个回收循环中进行重新定位而引起额外的开销。

1)在跟踪开始之前,选择将被清空的一部分堆区。

2)当标记可达对象时,记住所有指向指定区域内的对象的引用。

3)当跟踪完成时,并行地清扫存储空间以回收被不可达对象占用的空间。

4)最后,清空占据指定区域的可达对象,并修正指向被清空对象的引用。

7.8.3 类型不安全的语言的保守垃圾回收

如7.5.1节中讨论的,我们不可能构造出一个可以处理所有C和C++程序的垃圾回收器。因为我们总是可以通过算术运算来计算地址,所以在C和C++中,没有任何内存位置可被认为是不可达的。然而,很多C或C++程序从不按照这种方式随意地构造地址。已经证明,人们可以为这一类程序构造出一种保守的垃圾回收器(也就是不一定回收所有垃圾的回收器),在实践中它能够很好地完成任务。

保守的垃圾回收器假定我们不可以随意构造出一个地址,或者在没有指向某已分配存储块中某处的地址的情况下得到该存储块的地址。我们可以在程序中找出所有满足这一假设的垃圾。方法是,对于在任意可达存储区域中找到的一个二进制位模式,如果该模式可以被构造成一个内存位置,我们就认为它是一个有效地址。这种方案可能会把有些数据错当作地址。然而,这么做是正确的,因为这只会使得垃圾回收器保守地回收垃圾,留下的数据包含了所有必要的数据。

对象重定位需要更新所有指向旧地址的引用,使之指向新地址,因此它和保守的垃圾回收方法是不兼容的。因为保守的垃圾回收器并不能确认某个位模式是否真的指向某个实际地址,所以它不能修改这些模式并使之指向新的地址。

下面是一个保守的垃圾回收器的工作方式。首先修改内存管理器,使之为所有已分配内存块保存一个数据映射(data map)。这个映射使我们很容易地找到一个内存块的起止位置。这两

个起止位置跨越了多个地址。跟踪过程开始时，首先扫描程序的根集，找出所有看起来像内存位置的位模式，此时我们不考虑它的类型。通过在数据映射中查找这些可能的地址，我们可以找出所有可能通过这些位模式到达的内存块的开始位置，并将它们置为*待扫描*状态。然后，我们扫描所有待扫描的内存块，找出更多(很可能)可达的内存块，并且将它们放入工作列表。重复扫描过程，直到工作列表为空。在完成跟踪工作之后，我们使用上述数据映射来清扫整个堆区，定位并释放所有不可达的内存块。

7.8.4 弱引用

有时候，虽然程序员使用了带有垃圾回收机制的语言，但是仍然希望自己管理内存，或者管理部分内存。也就是说，尽管仍然存在一些引用指向某些对象，但程序员知道这些对象不会再被访问。一个来自编译的例子可以说明这一问题。

例 7.17 我们已经看到，词法分析器通常会管理一个符号表，为它碰到的每个标识符创建一个对象。比如，这些对象可能作为词法值被附加于语法分析树中代表这些标识符的叶子结点上。然而，以这些标识符的字符串作为键值构造一个散列表有助于对这些对象进行定位。这个散列表可以在词法分析器碰到一个标识符词法单元时更容易找到对应的对象。

当编译器扫描完标识符 I 的作用域时，I 的符号表对象不再有任何来自语法分析树的引用，也没有来自可能被编译器使用的其他中间结构的引用。然而，在散列表中仍然存在一个指向这个对象的引用。因为散列表是编译器的根集的一部分，所以这个对象不能作为垃圾被回收。如果碰到了另一个词素和 I 相同的标识符，编译器就会发现 I 已经过时了，指向 I 的对象的引用将被删除。然而，如果没有遇到词素相同的其他标识符，那么 I 的对象仍然是不可回收的，尽管在之后的整个编译过程中它都是无用的。 □

如果例子 7.17 中提出的问题很重要，那么编译器的作者可以设法在标识符的作用域一结束时就在散列表中删除对相应对象的所有引用。然而，一种被称为*弱引用*(weak reference)的技术支持程序员依靠自动垃圾回收来解决问题，并且不会因为那些实际不再使用的可达对象而给堆区存储带来负担。在这样的系统中，允许将某些引用声明为“弱”引用。弱引用的一个例子是我们刚刚讨论的散列表中的所有引用。当垃圾回收器扫描一个对象时，它不会沿着该对象内的弱引用前进，也不会将它们指向的对象设置为可达的。当然，如果另有一个不弱的引用指向这一个对象，这个对象可能仍然是可达的。

7.8.5 7.8 节的练习

！练习 7.8.1：在 7.8.3 节中，我们说如果一个 C 语言程序只会在已存在某个指向某存储块中某个位置的地址时构造出指向这块内存中某个位置的地址，我们就可以对这个程序进行垃圾回收。因此我们将形如

```
p = 12345;
x = *p;
```

的代码排除在外，因为即使没有指针指向某个存储块，p 仍然可能碰巧指向该存储块。另一方面，对于上面的代码，更可能发生的情况是 p 什么地方都不指，执行那个代码会引起一个内存分段错误。然而，用 C 语言可能写出一段代码，使得一个像 p 这样的变量一定指向某个存储块，且没有其他指针同时指向该存储块。写出一个这样的程序。

7.9 第 7 章总结

- *运行时刻组织*。为了实现源语言中的抽象概念，编译器与操作系统及目标机器协同，创建并管理了一个运行时刻环境。该运行时刻环境有一个静态数据区，用于存放对象代码

和在编译时刻创建的静态数据对象。同时它还有动态的栈区和堆区，用来管理在目标代码执行时创建和销毁的对象。

- 控制栈。过程调用和返回通常由称为控制栈的运行时刻栈管理。我们可以使用栈结构的原因是过程调用(或者说活动)在时间上是嵌套的。也就是说，如果 p 调用 q，那么 q 的活动就嵌套在 p 的活动之内。
- 栈分配。对于那些允许或要求局部变量在它们的过程结束之后就不可访问的语言而言，局部变量的存储空间可以在运行时刻栈中分配。对于这样的语言，每一个活跃的活动都在控制栈中有一个活动记录(或者说帧)。活动树的根结点位于栈底，而栈中的全部活动记录对应于活动树中到达当前控制所在活动的路径。当前活动的记录位于栈顶。
- 访问栈中的非局部数据。像 C 这样的语言不支持嵌套的过程声明，因此一个变量的位置要么是全局的，要么可以在运行时刻栈顶的活动记录中找到。对于带有嵌套过程的语言而言，我们可以通过访问链来访问栈中的非局部数据。访问链是加在各个活动记录中的指针。可以顺着访问链组成的链路到达正确的活动记录，从而找到期待的非局部数据。显示表是一个和访问链联合使用的辅助数组，它提供了一个不需要使用访问链链路的高效捷径。
- 堆管理。堆是用来存放生命周期不确定的，或者可以生存到被明确删除时刻的数据的存储区域。存储管理器分配和回收堆区中的空间。垃圾回收在堆区中找出不再被使用的空间，这些空间可以回收并用于存放其他数据项。对于要求垃圾回收的语言，垃圾回收器是存储管理器的一个重要子系统。
- 利用局部性。通过更好地利用存储的层次结构，存储管理器可以影响程序的运行时间。访问存储的不同区域所花的时间可能从几纳秒到几毫秒不等。幸运的是，大部分程序将它们的大部分时间用于执行相对较小的一部分代码，并且此时只会访问一小部分数据。如果一个程序很可能在短期内再次访问刚刚访问过的存储位置，该程序就具有时间局部性。如果一个程序很可能访问刚刚访问的存储区域附近的位置，该程序就具有空间局部性。
- 减少碎片。随着程序分配和回收存储，堆区可能会变得破碎，或者说被分割成大量细小且不连续的空闲空间(或称为“窗口”)。best-fit 策略(分配能够满足空间请求的最小可用“窗口”)经实践证明是有效的。尽管 best-fit 策略提高了空间利用率，但对于空间局部性而言它可能并不是最好的。可以通过合并或者说接合相邻的“窗口”来减少碎片。
- 人工回收。人工存储管理有两个常见的问题：没有删除那些不可能再被引用的数据，这称为内存泄漏错误；引用已经被删除的数据，这称为悬空指针引用错误。
- 可达性。垃圾就是不能被引用或者说到达的数据。有两种寻找不可达对象的基本方法：要么截获一个对象从可达变成不可达的转换，要么周期性地定位所有可达对象，并推导出其余对象都是不可达的。
- 引用计数回收器维护了指向一个对象的引用的计数。当这个计数变为 0 时，该对象就变成不可达的。这样的回收器带来了维护引用的开销，并且可能无法找出“循环”的垃圾，即由相互引用的不可达对象组成的垃圾。这些垃圾也可能通过由引用组成的链路相互引用。
- 基于跟踪的垃圾回收器从根集出发，迭代地检查或跟踪所有的引用，找出所有可达对象。根集包括了所有不需要对任何指针解引用就可直接访问的对象。
- 标记 - 清扫式回收器在一开始的跟踪阶段访问并标记所有可达对象，然后清扫堆区，回

收不可达对象。

- *标记并压缩回收器*改进了标记并清扫算法。它们把堆区中的可达对象重新定位，从而消除存储碎片。
- *拷贝回收器*将跟踪过程和发现空闲空间过程之间的依赖关系打破。它将存储分为两个半空间 A 和 B。首先使用某个半空间，比如说 A，来满足分配请求，直到它被填满。此时垃圾回收器开始工作，将可达对象拷贝到另一个半空间，也就是 B，然后对换两个半空间的角色。
- *增量式回收器*。简单的基于跟踪的回收器在垃圾回收期间会停止用户程序的执行。*增量式回收器*让垃圾回收过程和*用户程序*(或者说*增变者*)交错运行。增变者可能干扰增量式可达性分析，因为它可能改变之前已扫描对象中的引用。因此，增量式回收器通过超量估计可达对象集合，达到安全工作的目标。所有的“漂浮垃圾”可以在下一轮回收中被删除。
- *部分回收器*同样可以减少停顿时间。它们每次只回收一部分垃圾。最有名的部分回收算法是*世代垃圾回收方法*，它根据对象已分配时间的长短对对象分区，对新建对象进行更频繁的回收操作，因为它们的生命期通常较短。另一个算法*列车算法*使用固定长度的被称为*车厢*的区域。这些车厢被组织成*列车*。每一个回收步骤都处理尚存的第一辆列车中的当前的第一节车厢。当一节车厢被回收时，可达对象被移动到其他车厢中，这节车厢中最终只剩下垃圾，因此可以将其从该列车中删除。这两种算法可以一起使用，创建出一个部分回收器。该回收器对较年轻对象使用世代算法，对较成熟的对象使用列车算法。

7.10 第 7 章参考文献

在数理逻辑中，作用域规则和通过替换进行参数传递最早由 Frege[8]提出。Church 的 lamda 演算[3]使用词法作用域。这个方法曾被用作研究程序设计语言的模型。Algol 60 及其后续语言，包括 C 和 Java，使用词法作用域。动态作用域首先由 Lisp 语言引入，随后成为该语言的一个重要特征。McCarthy[14]介绍了这段历史。

很多与栈分配相关的概念来源于 Algol60 中的块和递归。在词法作用域语言中使用显示表来访问非局部数据的思想来源于 Dijkstra[5]。在 Randell 和 Russell[16]中更具体地描述了栈分配、显示表的使用、数组动态分配等概念。Johnson 和 Ritchie[10]讨论了一个调用代码序列的设计，该设计支持一个过程在不同的调用中使用不同数量的参数。

垃圾回收的研究一直是一个活跃的研究领域，例如 Wilson[17]。引用计数技术可以追溯到 Collin[4]。基于跟踪的回收技术则最早由 McCarthy[13]提出。他描述了一个针对固定长度单元的标记 - 清扫式算法。管理空闲空间的边界标记由 Knuth 在 1962 年提出并在[11]中出版。

算法 7.14 基于 Baker[1]的算法。算法 7.16 基于 Cheney[2]提出的 Fenichel 和 Yochelson[7]拷贝算法的非递归版本。

增量式可达性分析由 Dijkstra 等[6]进行了详细研究。Lieberman 和 Hewitt[12]给出了一个世代回收器，它是拷贝回收方法的一个扩展。列车算法由 Hudson 和 Moss[9]首先提出。

1. Baker, H. G. Jr., “The treadmill: real-time garbage collection without motion sickness,” *ACM SIGPLAN Notices* **27**:3 (Mar., 1992), pp. 66–70.

2. Cheney, C. J., “A nonrecursive list compacting algorithm,” *Comm. ACM* **13**:11 (Nov., 1970), pp. 677–678.

3. Church, A., *The Calculi of Lambda Conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton, N. J., 1941.

4. Collins, G. E., "A method for overlapping and erasure of lists," *Comm. ACM* **2**:12 (Dec., 1960), pp. 655–657.

5. Dijkstra, E. W., "Recursive programming," *Numerische Math.* **2** (1960), pp. 312–318.

6. Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Comm. ACM* **21**:11 (1978), pp. 966–975.

7. Fenichel, R. R. and J. C. Yochelson, "A Lisp garbage-collector for virtual-memory computer systems", *Comm. ACM* **12**:11 (1969), pp. 611–612.

8. Frege, G., "Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought," (1879). In J. van Heijenoort, *From Frege to Gödel*, Harvard Univ. Press, Cambridge MA, 1967.

9. Hudson, R. L. and J. E. B. Moss, "Incremental Collection of Mature Objects", *Proc. Intl. Workshop on Memory Management*, Lecture Notes In Computer Science **637** (1992), pp. 388-403.

10. Johnson, S. C. and D. M. Ritchie, "The C language calling sequence," Computing Science Technical Report 102, Bell Laboratories, Murray Hill NJ, 1981.

11. Knuth, D. E., *Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Boston MA, 1968.

12. Lieberman, H. and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Comm. ACM* **26**:6 (June 1983), pp. 419–429.

13. McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine," *Comm. ACM* **3**:4 (Apr., 1960), pp. 184–195.

14. McCarthy, J., "History of Lisp." See pp. 173–185 in R. L. Wexelblat (ed.), *History of Programming Languages*, Academic Press, New York, 1981.

15. Minsky, M., "A LISP garbage collector algorithm using secondary storage," A. I. Memo 58, MIT Project MAC, Cambridge MA, 1963.

16. Randell, B. and L. J. Russell, *Algol 60 Implementation*, Academic Press, New York, 1964.

17. Wilson, P. R., "Uniprocessor garbage collection techniques,"

`ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`