



誠朴雄偉
勵學敦行

第七章 运行时环境

冯 洋



运行时刻环境



■ 运行时刻环境

- 为数据分配安排存储位置
- 确定访问变量时使用的机制
- 过程之间的连接
- 参数传递
- 和操作系统、输入输出设备相关的其它接口

■ 主题

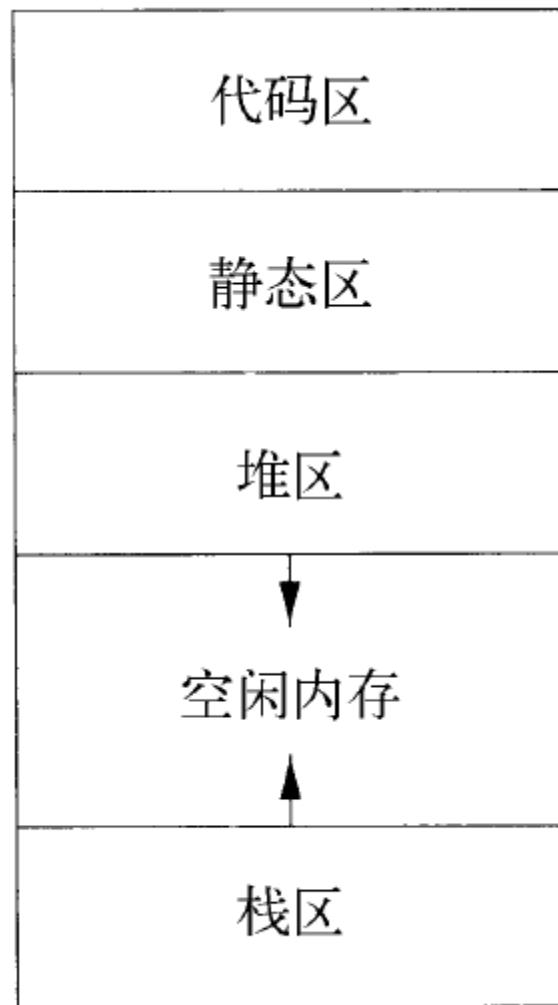
- 存储管理：栈分配、堆管理、垃圾回收
- 对变量、数据的访问



存储分配的典型方式



- 目标程序的代码放置在代码区
- 静态区、堆区、栈区分别放置不同类型生命期的数据值
- 实践中，栈是由低地址向高地址增长，而堆是由高地址向低地址增长





静态和动态存储分配



■ 静态分配

- 编译器在编译时刻就可以做出存储分配决定，不需要考虑程序运行时刻的情形
- 全局变量

■ 动态分配

- 栈式存储：和过程的调用/返回同步进行分配和回收，值的生命期和过程生命期相同
- 堆存储：数据对象比创建它的过程调用更长寿
 - 手工进行回收：C++/C...
 - 垃圾回收机制：Java/C#/Python...



栈式分配



- 主要内容
 - 活动树
 - 活动记录
 - 调用代码序列
 - 栈中的变长数据



活动树



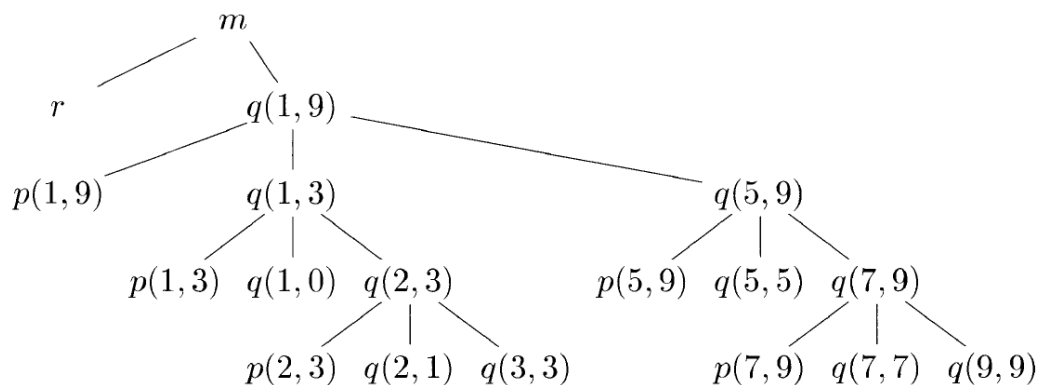
- 过程调用（过程活动）在**时间**上总是嵌套的
 - 后调用的先返回
 - 因此用栈式分配来分配过程活动所需内存空间
- 程序运行的所有过程活动可以用树表示
 - 每个结点对应于一个过程活动
 - 根结点对应于main过程的活动
 - 过程p的某次活动对应的结点的所有子结点：此次活动所调用的各个过程活动（从左向右，表示调用的先后顺序）



活动树的例子（1）



- 程序：P277，图7-2
- 过程调用（返回）序列和活动树的前序（后序）遍历对应
- 假定当前活动对应结点N，那么所有尚未结束的活动对应于N及其祖先结点。



```
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 7-2 中程序的可能的活动序列



活动记录



- 过程调用和返回由控制栈进行管理
- 每个活跃的活动对应于栈中的一个活动记录
- 活动记录按照活动的开始时间，从栈底到栈顶排列

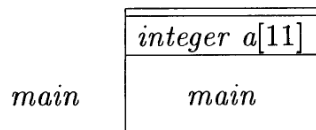
实在参数
返回值
控制链
访问链
保存的机器状态
局部数据
临时变量



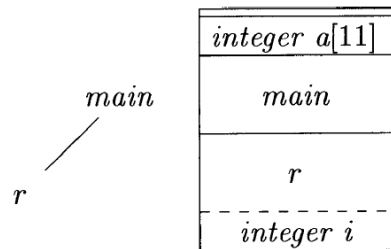
运行时刻栈的例子



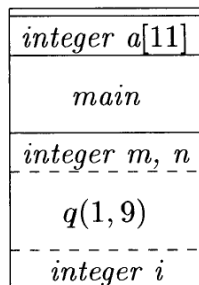
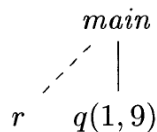
- $a[11]$ 为全局变量
- `main` 没有局部变量
- r 有局部变量 i
- q 的局部变量 i , 和参数 m, n



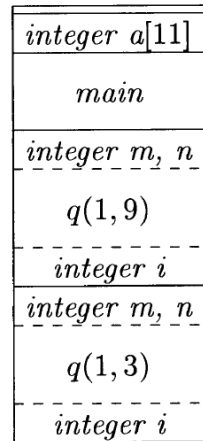
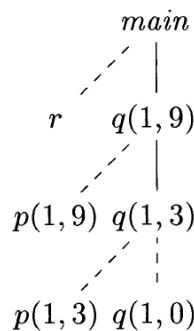
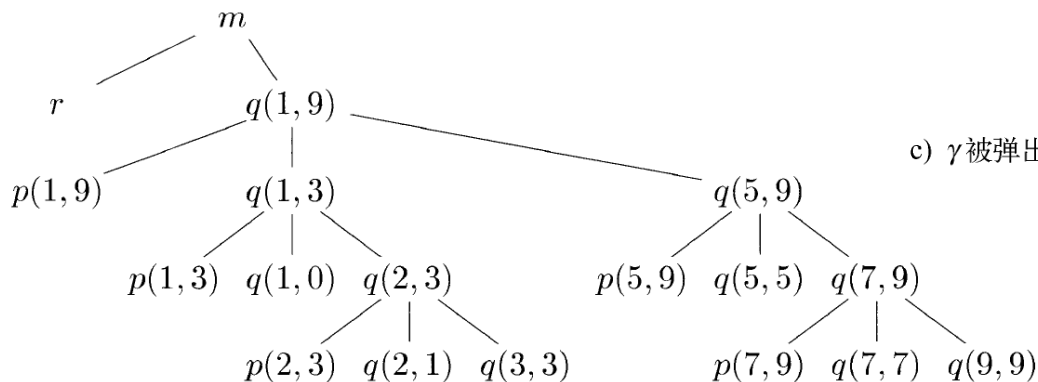
a) γ 被弹出栈



b) γ 被激活



c) γ 被弹出栈, $q(1, 9)$ 被压栈



d) 控制返回到 $q(1, 3)$



调用代码序列



- 调用代码序列(calling sequence)为活动记录分配空间，填写记录中的信息
- 返回代码序列(return sequence)恢复机器状态，使调用者继续运行
- 调用代码序列会分割到调用者和被调用者中
 - 根据源语言、目标机器、操作系统的限制，可以有不同的分割方案
 - 把代码尽可能放在被调用者中



调用/返回代码序列的要求



■ 数据方面

- 能够把参数正确地传递给被调用者
- 能够把返回值传递给调用者

■ 控制方面

- 能够正确转到被调用过程的代码开始位置
- 能够正确转回调用者的调用位置（的下一条指令）

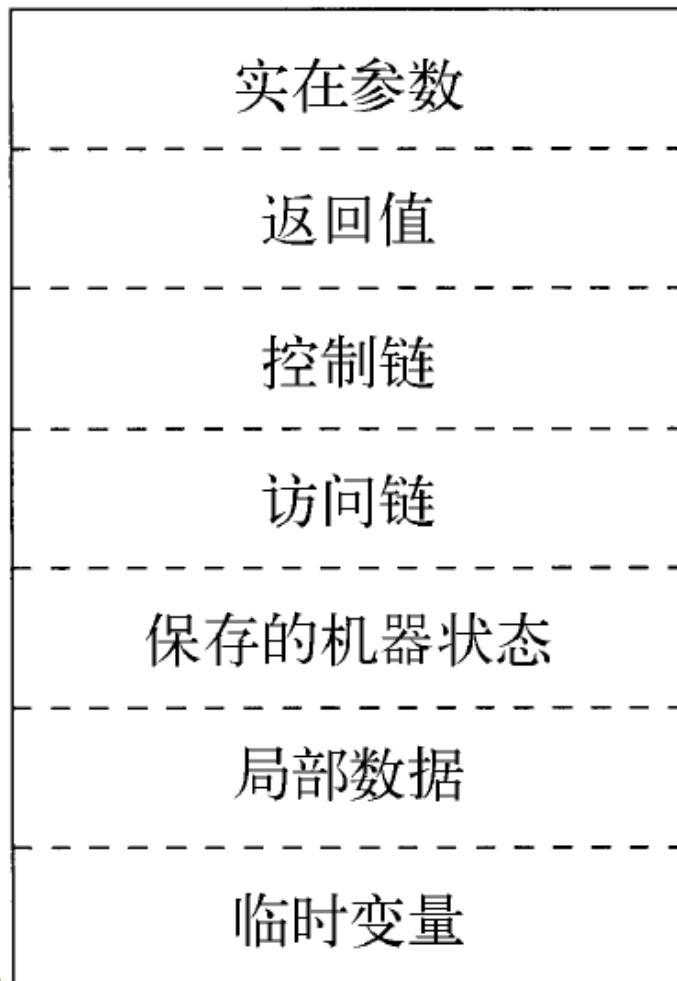
■ 调用代码序列和活动记录的布局相关



活动记录的布局原则



- 调用者和被调用者之间传递的值放在被调用者活动记录的开始位置
- 固定长度的项放在中间位置
 - 控制链、访问链、机器状态字段
- 早期不知道大小的项在活动记录尾部
- 栈顶指针(top_sp)通常指向固定长度字段的末端





调用代码序列的例子



- 调用代码序列 (Calling sequence)
 - 调用者计算实在参数的值
 - 将返回地址和原top_sp存放 to 被调用者的活动记录中。调用者增加top_sp的值 (越过了局部数据、临时变量、被调用者的参数、机器状态字段)
 - 被调用者保存寄存器值和其他状态字段
 - 被调用者初始化局部数据、开始运行
- 返回代码序列 (Return sequence)
 - 被调用者将返回值放到和参数相邻的位置
 - 恢复top_sp和寄存器, 跳转到返回地址



调用者/被调用者的活动记录

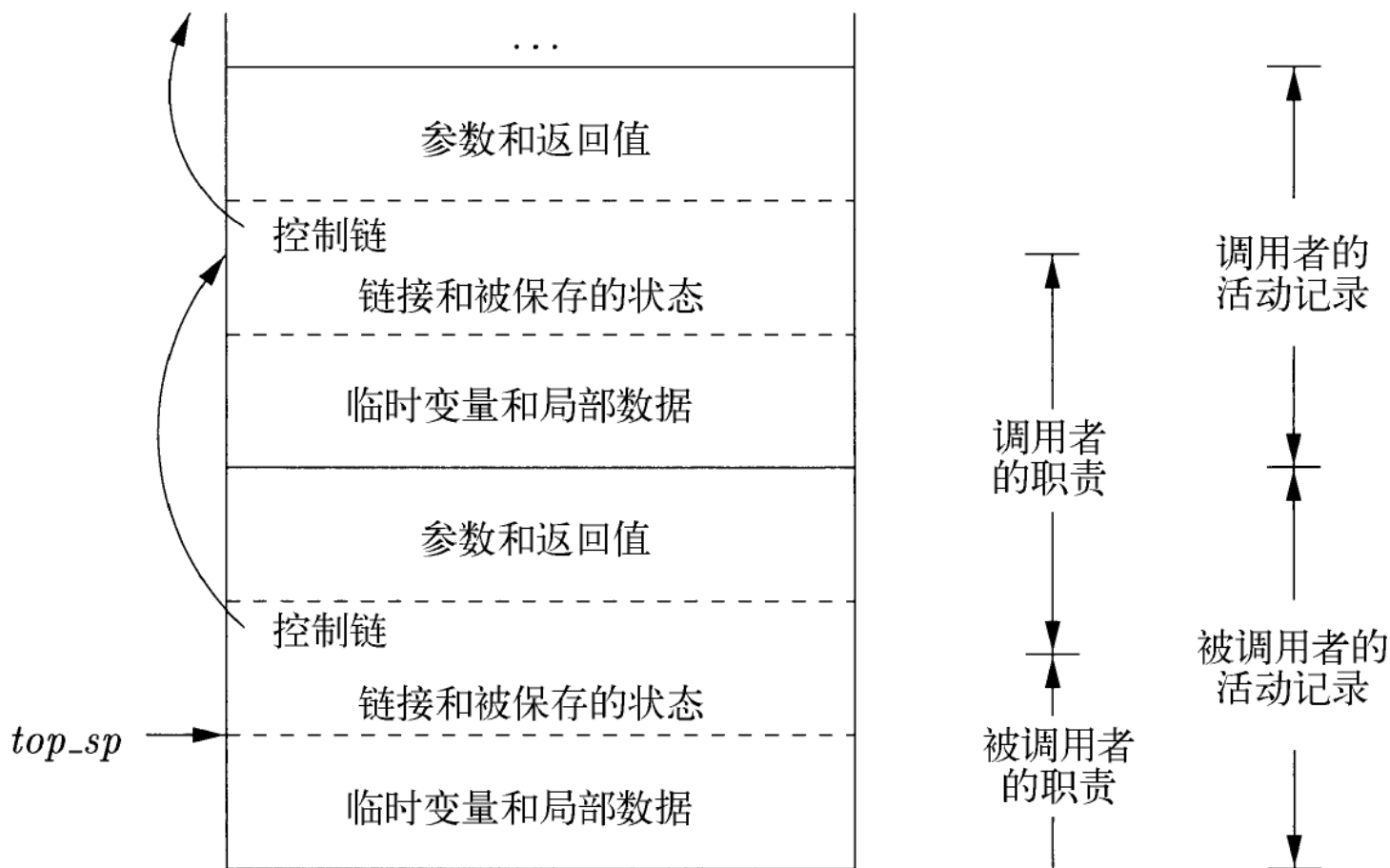


图 7-7 调用者和被调用者之间的任务划分



栈中的变长数据



- 看一个实际的例子



栈中的变长数据



- 如果数据对象的生命期局限于过程活动的生命期，就可以分配在运行时刻栈中
 - 变长数组也可以放在栈中
- `top`指向实际的栈顶
- `top_sp`用于寻找顶层记录的定长字段

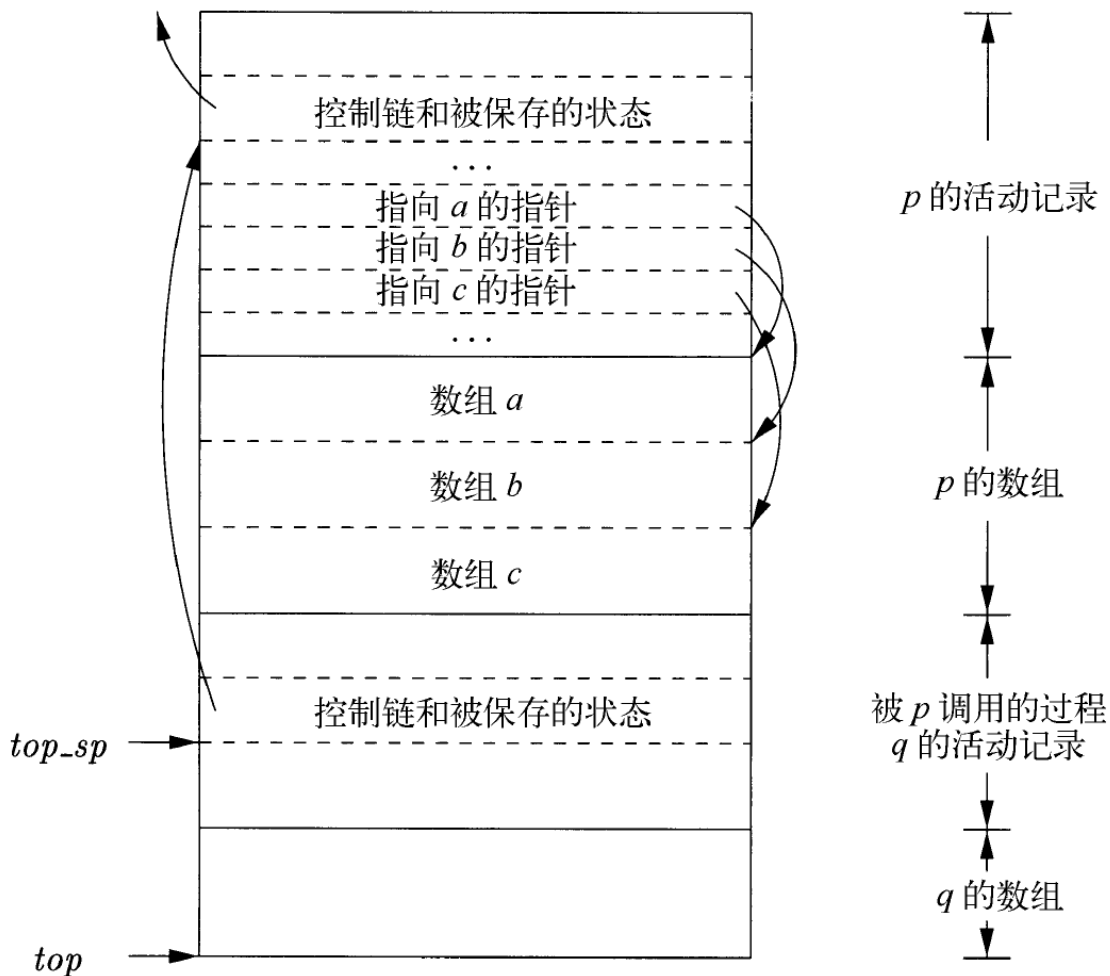


图 7-8 访问动态分配的数组



非局部数据的访问（无嵌套过程）



- 没有嵌套过程时的数据访问
 - C语言中，每个函数能够访问的变量
 - 函数的局部变量：相对地址已知，且存放在当前活动记录内，`top_sp`指针加上相对地址即可访问
 - 全局变量：在静态区，地址在编译时刻可知
 - 很容易将C语言的函数作为参数进行传递
 - 参数中只需包括函数代码的开始地址。
 - 在函数中访问非局部变量的模式很简单，不需要考虑过程是如何激活的



非局部数据的访问（嵌套声明过程）



- PASCAL中，如果过程A的声明中包含了过程B的声明，那么B可以使用在A中声明的变量。
- 当B的代码运行时，如果它使用的是A中的变量。那么这个变量指向运行栈中最上层的同名变量。
- 但是，我们不能通过**嵌套层次**直接得到A的活动记录的相对位置，必须通过**访问链**访问

void A()

{

int **x,y;**

void **B()**

 { **int b;**

x = b+y;

 }

void **C(){B();}**

C();

B();

}

当A调用C，C又调用B时：

A的活动记录
C的活动记录
B的活动记录

当A直接调用B时：

A的活动记录
B的活动记录



一个支持嵌套声明的例子



- ML是一种函数式语言.

变量一旦被声明并初始化就不会在改变, 只有少数几个例外.

- 定义变量并设定它们不可更改的初始值的语句有如下形式

```
val <name> = (expression)
```

- 函数使用如下语法进行定义

```
fun<name> (<arguments>) = <body>
```

- 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, 及直到end为止的所有语句. 函数可嵌套地定义. 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函数q的定义.



嵌套深度



- 嵌套深度是正文概念，可以根据源程序静态地确定
 - 不内嵌于任何其他过程中的过程，嵌套深度为1
 - 嵌套在深度为 i 的过程中的过程，深度为 $i+1$

深度为1

sort

深度为2

readArray,
exchange,

quicksort

深度为3

partition

```
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;
```



访问链



■ 访问链

- 当被调用过程需要其他地方的某个数据时需要使用访问链进行定位
- 如果过程 p 在声明时嵌套在过程 q 的声明中，那么 p 的活动记录中的访问链指向最上层的 q 的活动记录
- 从栈顶活动记录开始，访问链形成了一个链路，嵌套深度沿着链路逐一递减



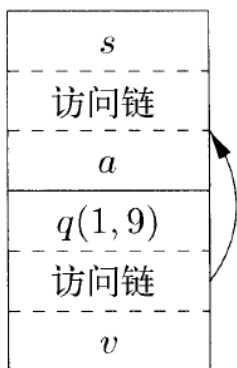
访问链



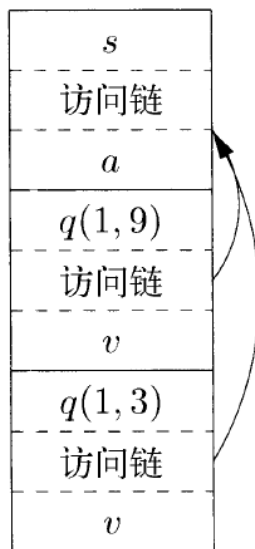
- 设深度为 n_p 的过程 p 访问变量 x ，而变量 x 在深度为 n_q 的过程中声明，那么
 - $n_p - n_q$ 在编译时刻已知
 - 从当前活动记录出发，沿访问链前进 $n_p - n_q$ 次找到的活动记录中的 x 就是要找的变量位置
 - x 相对于这个活动记录的偏移量在编译时刻已知



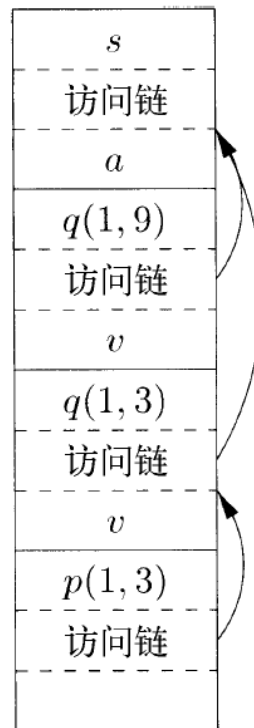
访问链的例子



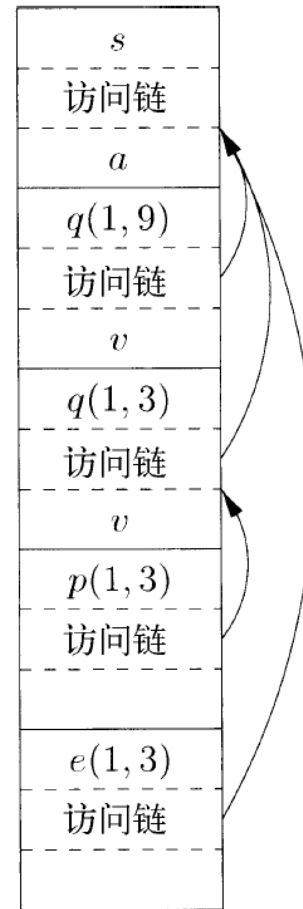
a)



b)



c)



d)

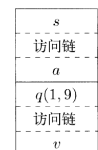
```
1) fun sort(inputFile, outputFile) =  
    let  
2)     val a = array(11,0);  
3)     fun readArray(inputFile) = ... ;  
4)         ... a ... ;  
5)     fun exchange(i,j) =  
6)         ... a ... ;  
7)     fun quicksort(m,n) =  
        let  
8)         val v = ... ;  
9)         fun partition(y,z)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



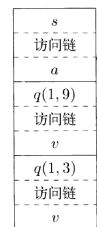
访问链的维护（直接调用过程）



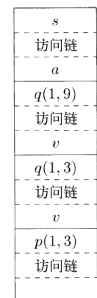
- 当过程 q 调用过程 p 时，访问链的变化
 - p 的深度大于 q ：根据作用域规则， p 必然在 q 中直接定义；那么 p 的访问链指向当前活动记录
 - s 调用 $q(1, 9)$
 - 递归调用（ $p=q$ ）：新活动记录的访问链等于当前记录的访问链
 - $q(1, 9)$ 调用 $q(1, 3)$
 - p 的深度小于等于 q 的深度：此时必然有过程 r ， p 直接在 r 中定义，而 q 嵌套在 r 中； p 的访问链指向栈最高的 r 的活动记录
 - p 调用 $exchange$



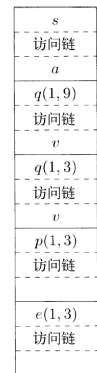
a)



b)



c)



d)

访问链的维护：访问链的定义+作用域规则



访问链的维护（过程指针型参数）



- 在传递过程指针参数时，过程型参数中不仅包含过程的代码指针，还包括正确的访问链



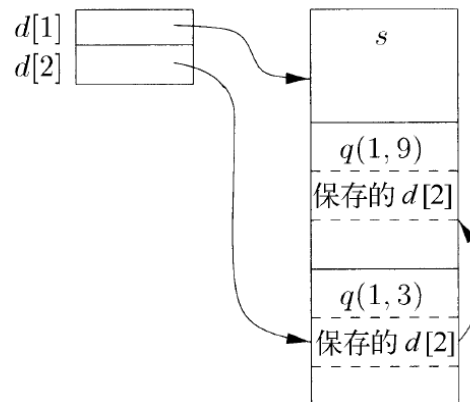
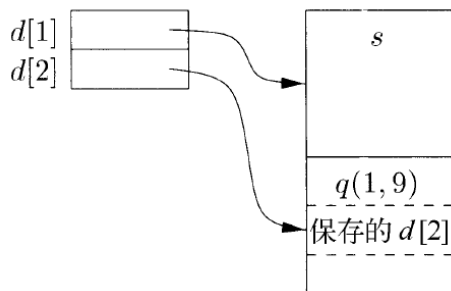
显示表



- 用访问链访问数据时，访问开销和嵌套深度差有关
- 使用显示表可以提高效率，访问开销为常量
- 显示表：数组d为每个嵌套深度保留一个指针
 - 指针d[i]指向栈中最高的、嵌套深度为i的活动记录。
 - 如果程序p中访问嵌套深度为i的过程q中声明的变量x，那么d[i]直接指向相应的（必然是q的）活动记录
 - 注意：i在编译时刻已知
- 显示表的维护
 - 调用过程p时，在p的活动记录中保存d[n_p]的值，并将d[n_p]设置为当前活动记录。
 - 从p返回时，恢复d[n_p]的值。

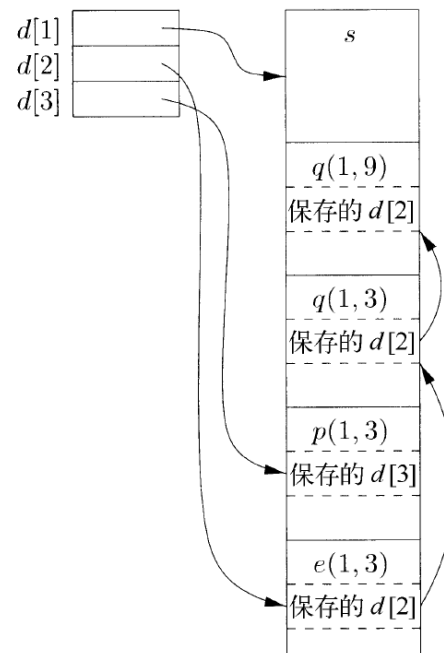
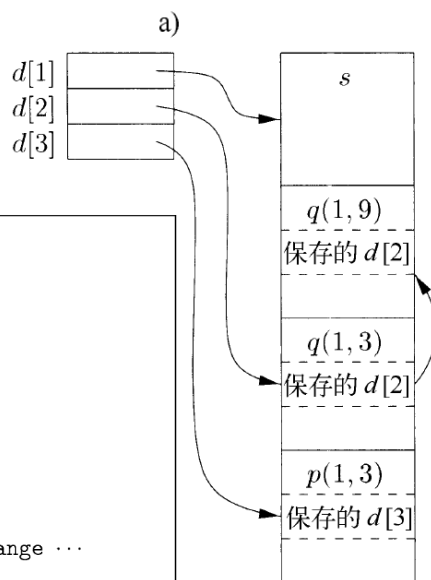


显示表的例子



q(1, 9) 调用
q(1, 3) 时,
q 的深度为 2

q(1, 3) 调用 p,
p 的深度为 3



q 调用 e, e
的深度为 2

c)

d)

```
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;
```



堆管理



■ 堆空间

- 用于存放生命周期不确定、或生存到被明确删除为止的数据对象
- 例如：new生成的对象可以生存到被delete为止
- malloc申请的空间生存到被free为止

■ 存储管理器

- 分配/回收堆区空间的子系统
- 根据语言而定
 - C、C++需要手动回收空间
 - Java可以自动回收空间（垃圾收集）



存储管理器



■ 基本功能

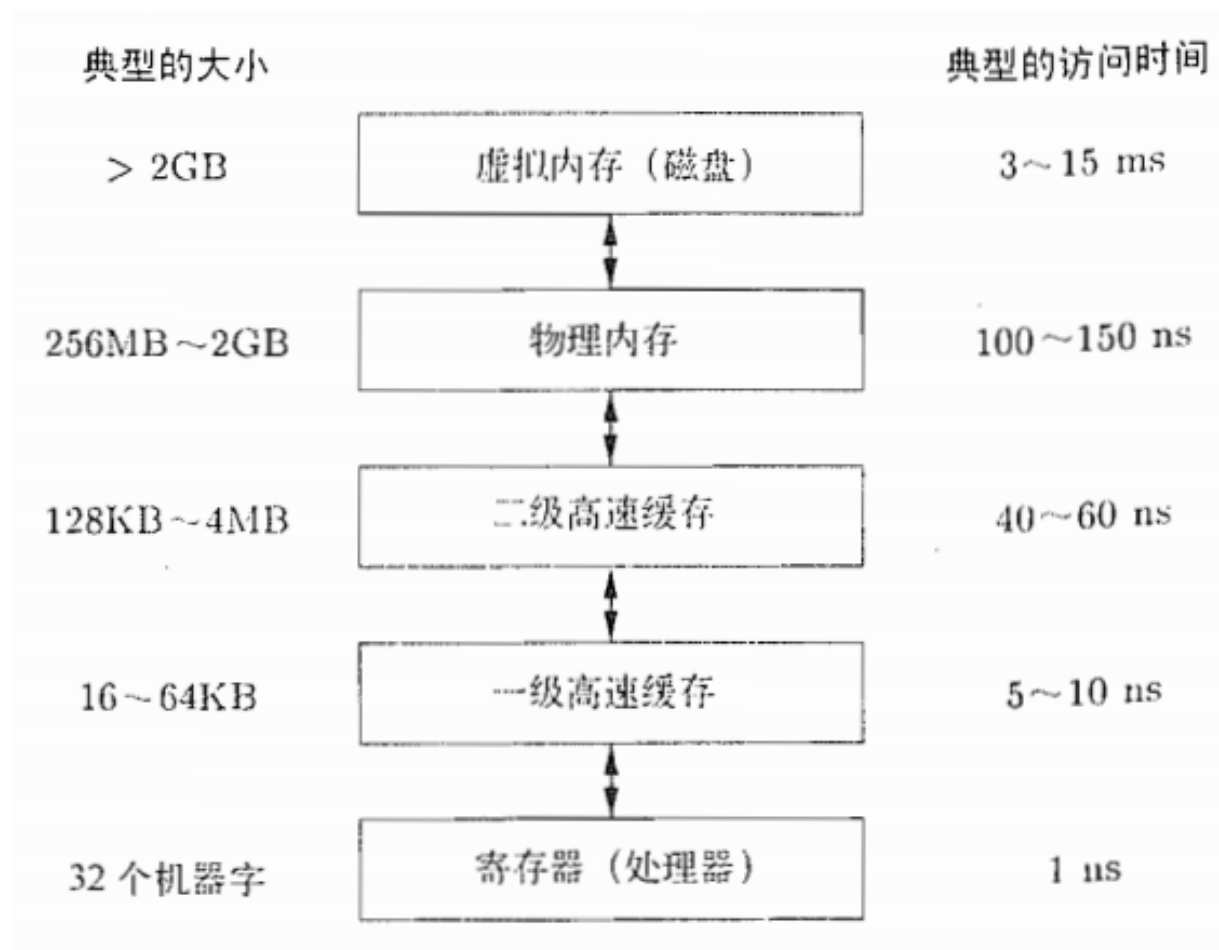
- 分配：为每个内存请求分配一段连续的、适当大小的堆空间
 - 首先从空闲的堆空间分配
 - 如果不行则从操作系统中获取内存、增加堆空间
- 回收：把被回收的空间返回空闲空间缓冲池，以满足其他内存需求

■ 评价存储管理器的特性

- 空间效率：使程序需要的堆空间最小，即减小碎片
- 程序效率：充分运用内存系统的层次，提高效率
- 低开销：使分配/收回内存的操作尽可能高效



存储管理器





程序中的局部性

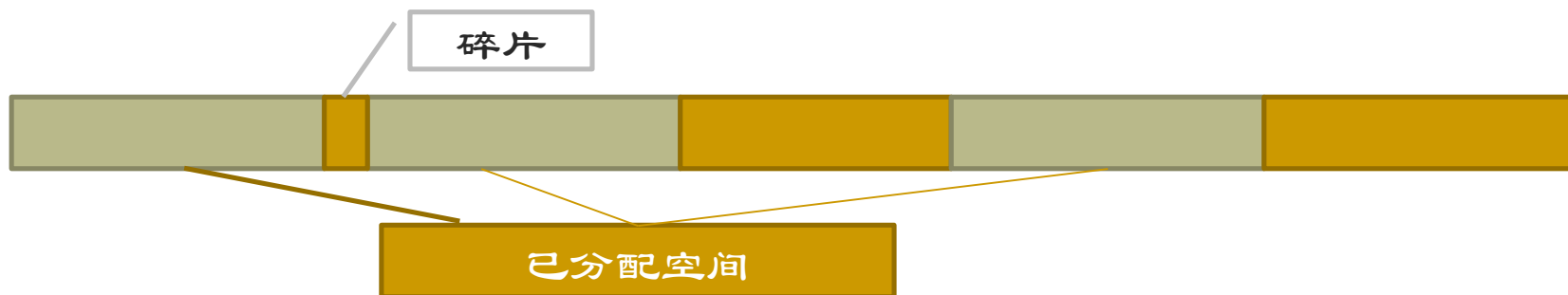


■ 局部性

大部分程序表现出高度的局部性 - 程序的大部分运行时间花费在相对较小的一部分代码中（此时可能只涉及固定的一小部分数据）。



堆空间的碎片问题



- 随着程序分配/回收内存，堆区逐渐被割裂成为若干空闲存储块（窗口，hole）和已用存储块的交错
- 分配一块内存时，通常是把一个窗口的一部分分配出去，其余部分成为更小的块
- 回收时，被释放的存储块被放回缓冲池。通常要把连续的窗口接合成为更大的窗口



堆空间分配方法



■ Best-Fit

- 总是将请求的内存分配在满足请求的最小的窗口中
- 好处：可以将大的窗口保留下来，应对更大的请求

■ First-Fit

- 总是将对象放置在第一个能够容纳请求的窗口中
- 放置对象时花费时间较少，但是总体性能较差
- 但是first-fit的分配方法通常具有较好的数据局部性
 - 同一时间段内生成的对象经常被分配在连续的空间内



使用容器的堆管理方法



- 设定不同大小的空闲块规格，相同规格的块放在同一容器中
- 较小的（较常用的）尺寸设置较多的容器
- 比如GNU的C编译器将所有存储块对齐到8字节边界
 - 空闲块的尺寸大小
 - 16, 24, 32, 40, ..., 512
 - 大于512的按照对数划分：每个容器的最小尺寸是前一个容器的最小尺寸的两倍
 - 荒野块：可以扩展的内存块
 - 分配方法
 - 对于小尺寸的请求，直接在相应容器中找
 - 大尺寸的请求，在适当的容器中寻找适当的空闲块
 - 可能需要分割内存块
 - 可能需要从荒野块中分割出更多的块



管理和接合空闲空间



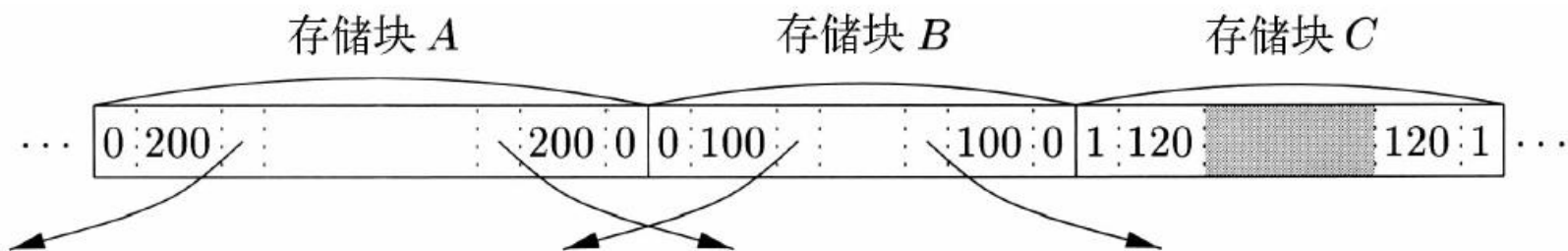
- 当回收一个块时，可以把这个块和相邻的块接合起来，构成更大的块
- 支持相邻块接合的数据结构
 - 边界标记：在每一块存储块的两端，分别设置一个free/used位；相邻的位置上存放字节总数
 - 双重链接的、嵌入式的空闲块列表：列表的指针存放在空闲块中、用双向指针的方式记录了有哪些空闲块



例子



- 相邻的存储块A、B、C
 - 当回收B时，通过对free/used位的查询，可以知道B左边的A是空闲的，而C不空闲。
 - 同时还可以知道A、B合并为长度为300的块
 - 修改双重链表，把A替换为A、B接合后的空闲块
- 注意：双重链表中一个结点的前驱并不一定是它邻近的块





处理手工存储管理



■ 两大问题

- 内存泄露：未能删除不可能再被引用的数据
- 悬空指针引用：引用已被删除的数据

■ 其他问题

- 空指针访问/数组越界访问

■ 解决方法

- 自动存储管理
- 正确的编程模式



正确的编程模式（1）



- 对象所有者（Object ownership）
 - 每个对象总是有且只有一个所有者（指向此对象的指针）；只有通过Owner才能够删除这个对象
 - 当Owner消亡时，这个对象要么也被删除，要么已经被传递给另一个owner
 - 语句`v=new ClassA`；创建的对象的所有者为v
 - 即将对v进行赋值的时刻（v的值即将消亡）
 - 要么v已经不是它所指对象的所有者；比如`g=v`可以把v的ownership传递给g
 - 要么需要在返回/赋值之前，执行`delete v`操作
 - 编程时需要了解各个指针在不同时刻是否owner
 - 防止内存泄漏，避免多次删除对象。不能解决悬空指针问题



正确的编程模式（2）



■ 引用计数

- 每个动态分配的对象附上一个计数：记录有多少个指针指向这个对象
- 在赋值/返回/参数传递时维护引用计数的一致性
- 在计数变成0之时删除这个对象
- 可以解决悬空指针问题；但是在递归数据结构中仍然可能引起内存泄漏
- 需要较大的运行时刻开销

■ 基于区域的分配

- 将一些生命期相同的对象分配在同一个区域中
- 整个区域同时删除



垃圾回收



- 垃圾
 - 狭义：不能被引用（不可达）的数据
 - 广义：不需要再被引用的数据
- 垃圾回收：自动回收不可达数据的机制，解除了程序员的负担
- 使用的语言
 - Java、Perl、ML、Modula-3、Prolog、Smalltalk



垃圾回收器的设计目标



■ 基本要求:

- 语言必须是类型安全的: 保证回收器能够知道数据元素是否为一个指向某内存块的指针
- 类型不安全的语言: C, C++

■ 性能目标

- 总体运行时间: 不显著增加应用程序的总运行时间
- 空间使用: 最大限度地利用可用内存
- 停顿时间: 当垃圾回收机制启动时, 可能引起应用程序的停顿。这个停顿应该比较短
- 程序局部性: 改善空间局部性和时间局部性



可达性



- 直观地讲，可达性就是指一个存储块可以被程序访问到
- 根集：不需要指针解引用就可以直接访问的数据
 - Java：静态成员、栈中变量
- 可达性
 - 根集的成员都是可达的
 - 对于任意一个对象，如果指向它的一个指针被保存在可达对象的某字段中、或数组元素中，那么这个对象也是可达的
- 性质
 - 一旦一个对象变得不可达，它就不会再变成可达的



改变可达对象集合的操作



- 对象分配：返回一个指向新存储块的引用
- 参数传递/返回值：对象引用从实在参数传递到形式参数，从返回值传递给调用者
- 引用赋值： $u=v$ ； v 的引用被复制到 u 中， u 中原来的引用丢失。可能使得 u 原来指向的对象变得不可达，并且递归地使得更多对象变得不可达
- 过程返回：活动记录出栈，局部变量消失，根集变小；可能使得一些对象变得不可达



垃圾回收方法分类



- 跟踪相关操作，捕获对象变得不可达的时刻，回收对象占用的空间
- 在需要时，标记出所有可达对象、回收其它对象



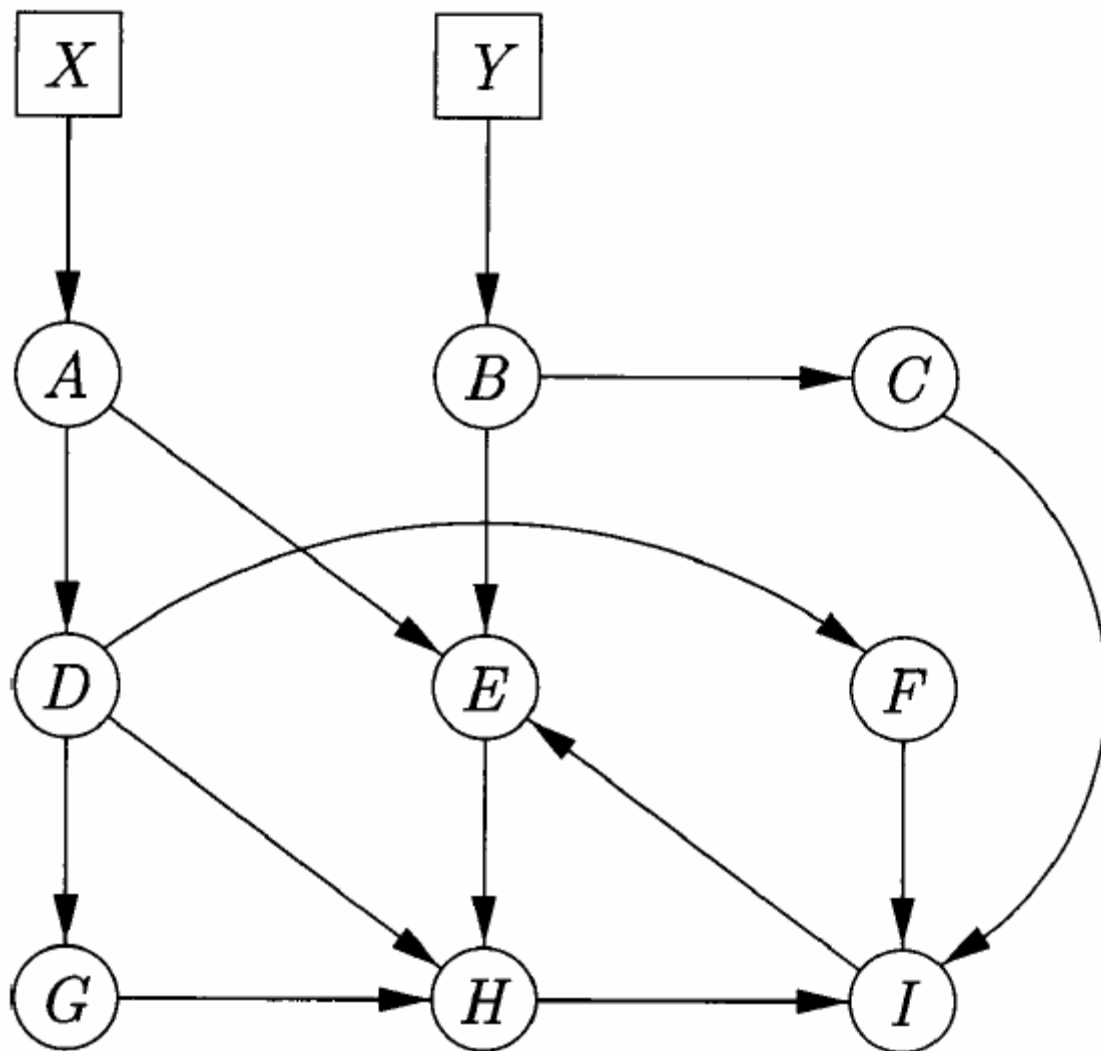
基于引用计数的垃圾回收器



- 每个对象有一个用于存放引用计数的字段，并按照如下方式维护
 - 对象分配：引用计数设为1
 - 参数传递：引用计数加1
 - 引用赋值： $u=v$ ： u 指向的对象引用减1、 v 指向的对象引用加1
 - 过程返回：局部变量指向对象的引用计数减1
- 如果一个对象的引用计数为0，在删除对象之前，此对象中各个指针所指对象的引用计数减1
- 回收器有缺陷，可能引起内存泄漏
- 开销较大、但是不会引起停顿



引用计数的例子



- 考虑如下操作：
 - $y=x$
 - y 是当前函数 f 的局部变量, 且 f 返回
- 修改计数后总是先考虑是否释放
- 释放一个对象之前总是先处理对象内部的指针



循环垃圾的例子

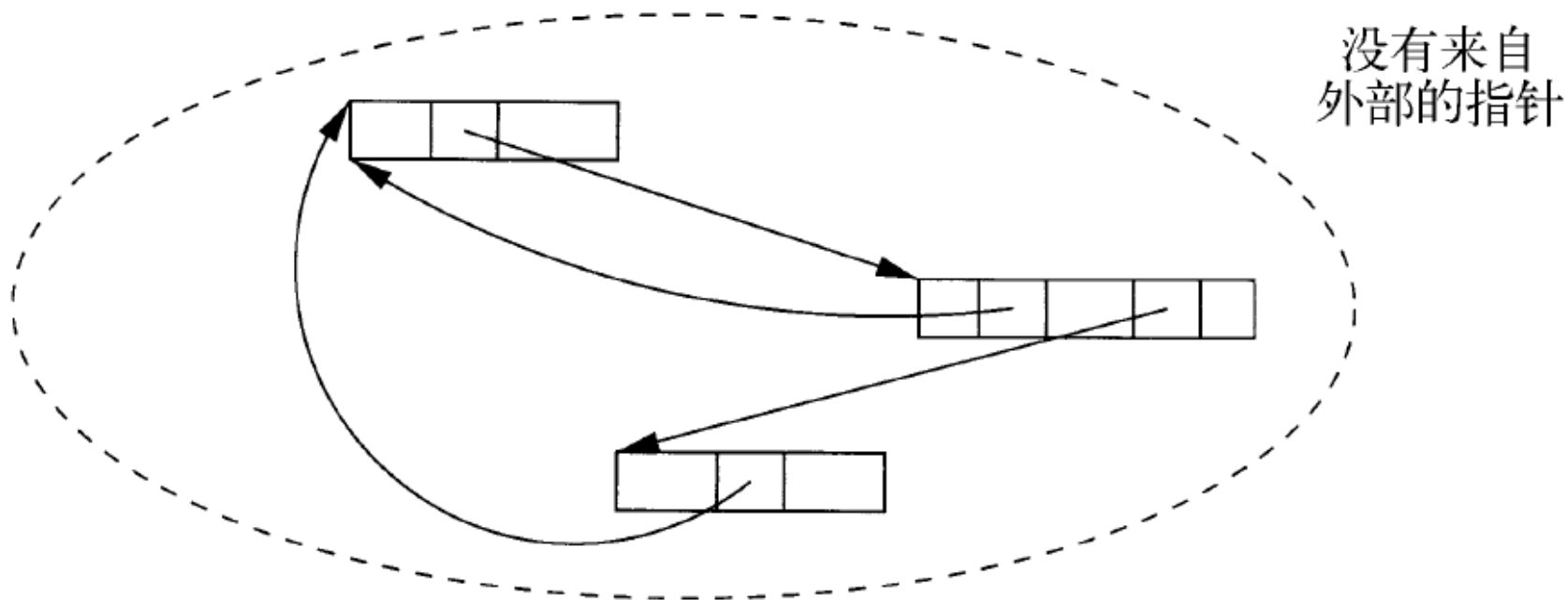


图 7-18 一个不可达的循环数据结构



基于跟踪的垃圾回收



- 标记-清扫式垃圾回收
- 标记-清扫式垃圾回收的优化
- 标记并压缩垃圾回收



标记-清扫式垃圾回收



- 一种直接的、全面停顿的算法
- 分成两个阶段
 - 标记：从根集开始，跟踪并标记出所有可达对象；
 - 清扫：遍历整个堆区，释放不可达对象；
- 如果我们把数据对象看作顶点，引用看作有向边，那么标记的过程实际上是从根集开始的图遍历的过程



标记-清扫式垃圾回收



- 该算法主要包括两步，
 - mark, 从 root 开始进行树遍历, 每个访问的对象标注为「使用中」
 - sweep, 扫描整个内存区域, 对于标注为「使用中」的对象去掉该标志, 对于没有该标注的对象直接回收掉



标记-清扫式垃圾回收



// 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。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



标记-清扫垃圾回收算法



```
/* 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  $\neq \emptyset$ ) {
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 =  $\emptyset$ ；
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；
    }
```

因为语言是强类型的，所以垃圾回收机制可以知道每个数据对象的类型，以及这个对象有哪些字段是指针



标记-清扫式垃圾回收

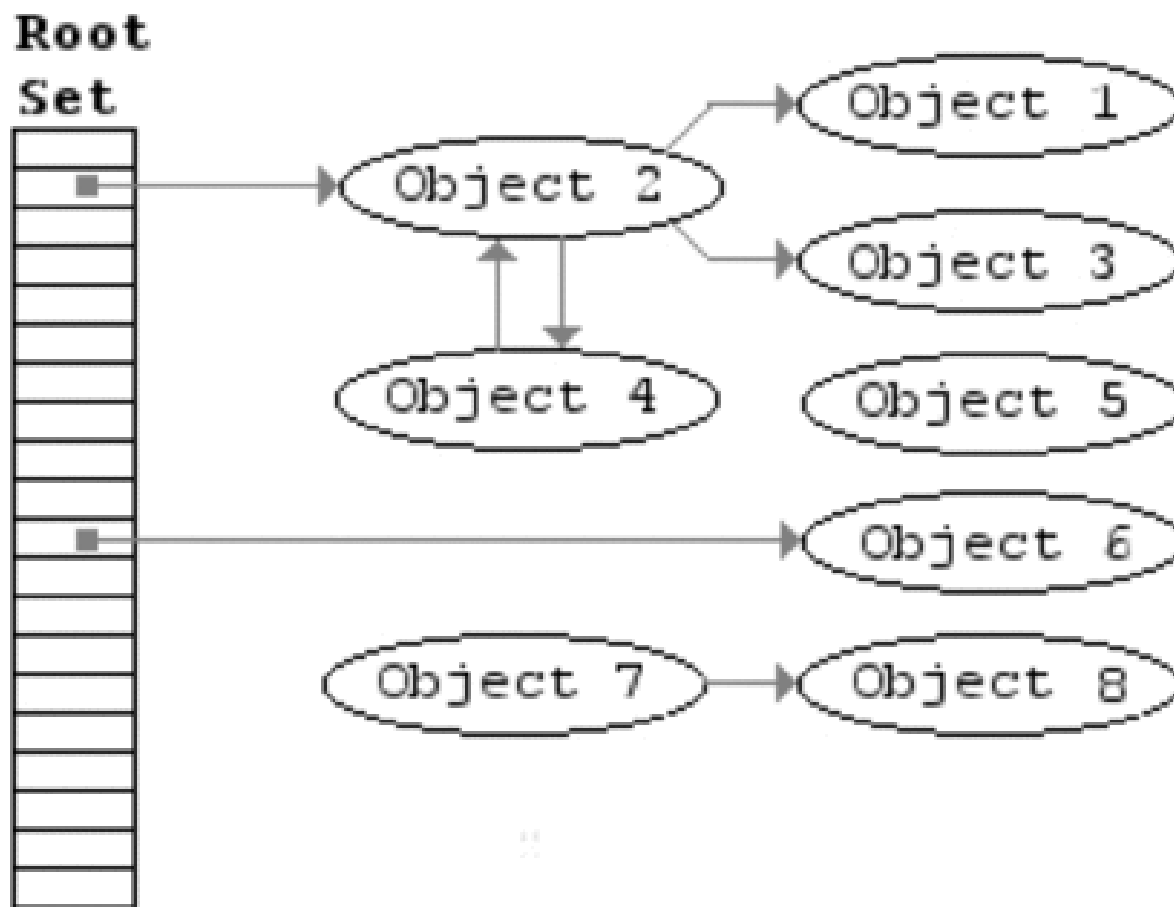


■ 该算法的缺点有：

- 在进行 GC 期间，整个系统会被挂起（暂停，Stop-the-world），所以在一些实现中，会采用各种措施来减少这个暂停时间
- heap 容易出现碎片。实现中一般会进行 move 或 compact。（需要说明一点，所有 heap 回收机制都会这个问题）
- 在 GC 工作一段时间后，heap 中连续地址上存在 age 不同的对象，这非常不利于引用的本地化（locality of reference）
- 回收时间与 heap 大小成正比



标记-清扫垃圾回收算法





标记-清扫垃圾回收算法



回收前状态：

存活对象	存活对象	可回收	可回收	存活对象	未使用	可回收	存活对象
未使用	可回收	可回收	存活对象	未使用	存活对象	可回收	未使用
未使用	可回收	存活对象	可回收	可回收	存活对象	未使用	未使用

回收后状态：

存活对象	存活对象	未使用	未使用	存活对象	未使用	未使用	存活对象
未使用	未使用	未使用	存活对象	未使用	存活对象	未使用	未使用
未使用	未使用	存活对象	未使用	未使用	存活对象	未使用	未使用

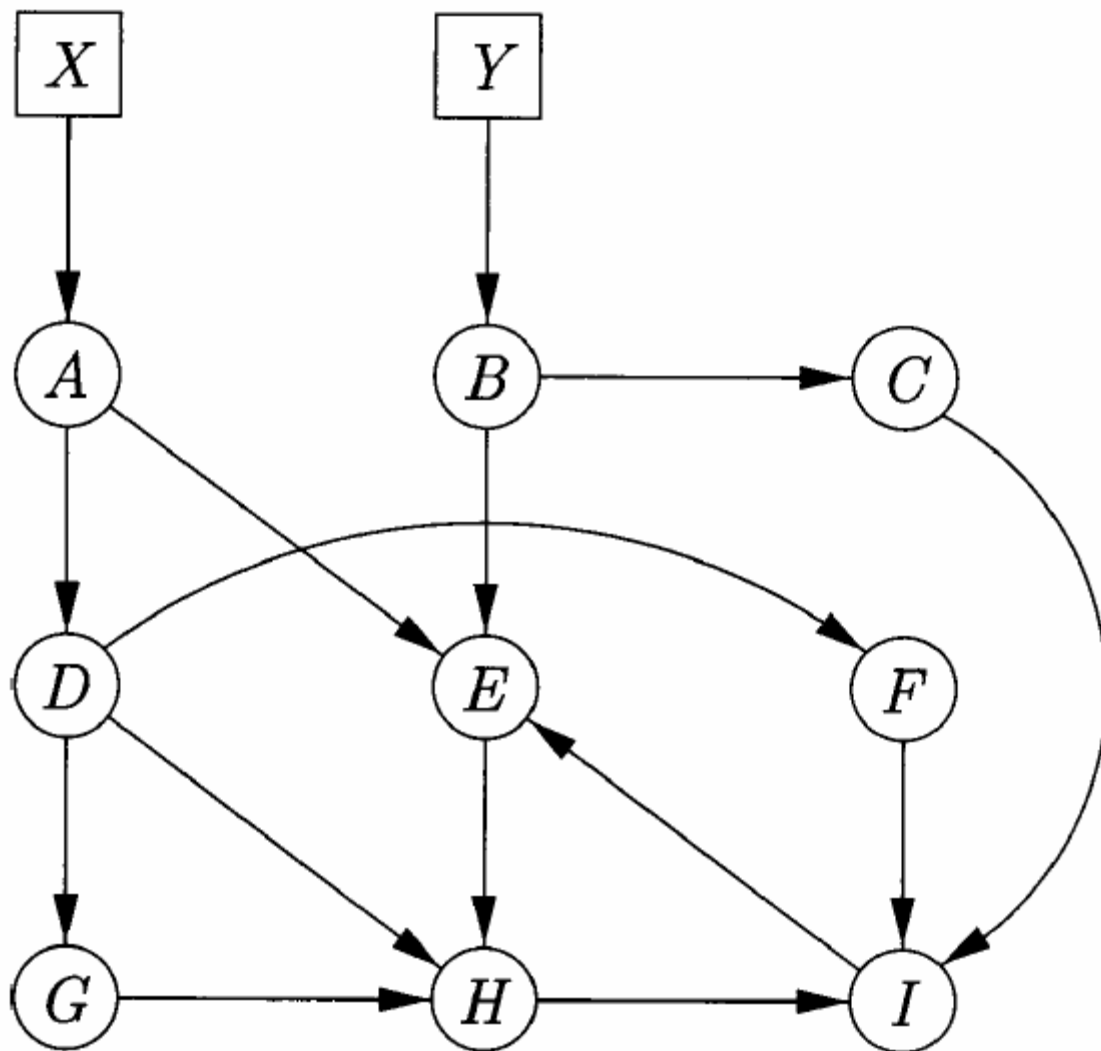
存活对象

可回收

未使用



例子



- 假设 x 是全局变量， y 是当前的函数活动的局部变量
- 当前活动返回之后，进行标记清扫
 - A, D, E, F, I, G, H
 - B, C 不可达



讨论：垃圾收集的代价



■ 效率！

- 标记-清扫垃圾回收算法采用一种类似于深度优先遍历的算法，完成整个标记的过程。
- 深度优先搜索所需的时间与它标记的节点个数成正比，即与可到达数据的数量成正比，可以推知，清扫阶段所需的时间，与堆的大小成正比。假设大小为H的堆中有R个字的可达数据。
- 垃圾回收一次的代价则为： $aR+bH$ ，其中a与b是常数。
- 回收一次得到的可用字空间大小为： $H-R$ 。
- 那么，回收到一个可用字空间的代价为：
$$(aR+bH) / (H-R)$$



基本抽象分类

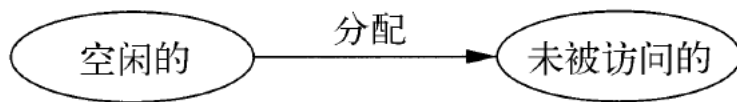


■ 每个存储块处于四种状态之一

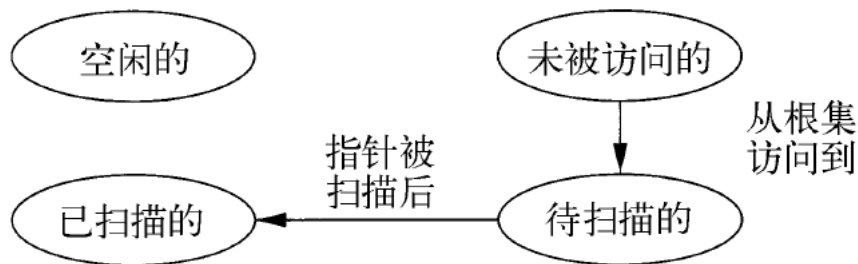
- 空闲
- 未被访问
- 待扫描
- 已扫描

■ 对存储块的操作会改变存储块的状态

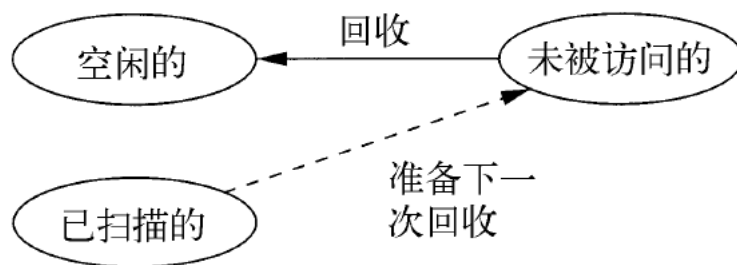
- 应用程序分配
- 垃圾回收器访问、扫描
- 收回



a) 跟踪之前：增变者的动作



b) 通过跟踪发现可达性



c) 回收存储空间



标记-清扫垃圾回收算法的优化



- 基本算法需要扫描整个堆
- 优化
 - 用一个列表记录所有已经分配的对象
 - 不可达对象等于已分配对象减去可达对象
- 好处
 - 只需要扫描这个列表就可以完成清扫
- 坏处
 - 需要维护这个列表



标记-清扫垃圾回收算法的优化



// 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。



标记-清扫垃圾回收算法的优化



方法:

- **Free**, **Unreached**, **Unscanned**, **Scanned**的四个列表, 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.
- 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]



优化后的算法



- 使用了四个列表: Scanned, Unscanned, Unreached, Free;

```
1)  Scanned =  $\emptyset$ ;  
2)  Unscanned = 在根集中引用的对象的集合;  
3)  while (Unscanned  $\neq \emptyset$ ) {  
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  $\cup$  Unreached;  
9)  Unreached = Scanned;
```



压缩并标记垃圾回收



- 对可达对象进行重定位可以消除存储碎片
 - 把可达对象移动到堆区的一端，另一端则是空闲空间
 - 空闲空间合并成单一块，分配内存时高效率
- 整个过程分成三个步骤
 - 标记
 - 计算新位置
 - 移动并设置新的引用



压缩并标记垃圾回收



/* 标记 */

```
1) Unscanned = 根集引用的对象的集合;  
2) while (Unscanned  $\neq \emptyset$ ) {  
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' 是未访问的) {  
6)       将 o' 标记为已访问的;  
7)       将 o' 加入到列表 Unscanned 中;
```

}

}

}

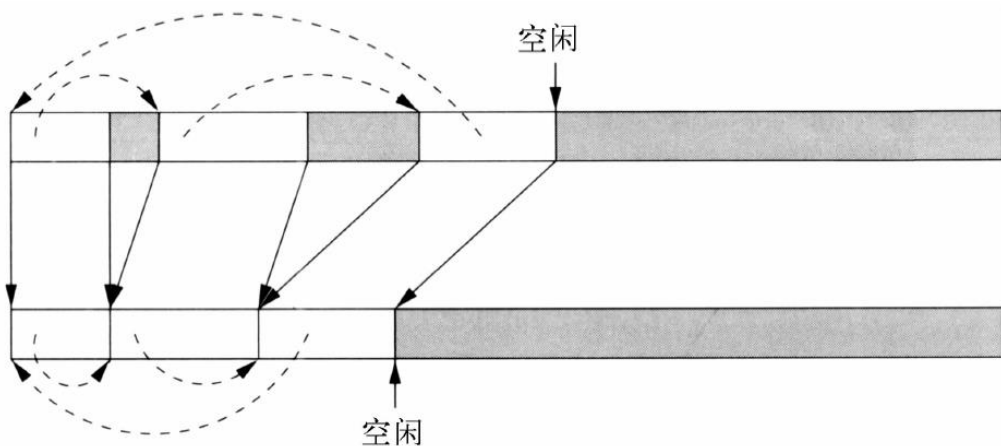
/* 计算新的位置 */

```
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);
```





压缩并标记垃圾回收



回收前状态：

存活对象	存活对象	可回收	可回收	存活对象	未使用	可回收	存活对象
未使用	可回收	可回收	存活对象	未使用	存活对象	可回收	未使用
未使用	可回收	存活对象	可回收	可回收	存活对象	未使用	未使用

回收后状态：

存活对象	存活对象	存活对象	存活对象	存活对象	存活对象	存活对象	存活对象
未使用	未使用	未使用	未使用	未使用	未使用	未使用	未使用

存活对象

可回收

未使用



拷贝回收器



- 堆空间被分为两个半空间
 - 应用程序在某个半空间内分配存储，当充满这个半空间时，开始垃圾回收
 - 回收时，可达对象被拷贝到另一个半空间
 - 回收完成后，两个半空间角色对调



拷贝回收器



- 拷贝回收算法虽然能解决内存碎片问题，但是需要多次遍历heap空间，这会导致较大性能损耗
- 拷贝回收算法采用空间换时间的方式来提升性能
- 这类 GC 并不会真正去“回收”不可到达对象，而是会把所有可到达对象移动到一个区域，heap 中剩余的空间就是可用的了（因为这里面都是垃圾）



拷贝回收器



- 典型的代表半空间拷贝回收器（**semispace collector**）。其工作过程是这样的：
 - heap 被分成2份相邻的空间（**semispace**）：**fromspace** 与 **tospace**
 - 在程序运行时，只有 **fromspace** 会被使用（分配新对象）
 - 在 **fromspace** 没有足够空间容纳新对象时，程序会被挂起，然后把 **fromspace** 的可到达对象拷贝到 **tospace**
 - 在拷贝完成时，之前的2个空间交换身份，**tospace** 成了新一轮的 **fromspace**



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。 **Fron**半空间此时全部空闲。
- 方法：
- 算法在**From**半空间找出可达对象，访问它们时立刻把它们拷贝到**To**半空间。



```
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);
11)    }
12)
13)    /* 如果一个对象已经被移动过了, 查找这个对象的新位置 */
14)    /* 否则将对象设置为待扫描状态 */
15)    LookupNewLocation(o) {
16)        if (NewLocation(o) = NULL) {
17)            NewLocation(o) = free;
18)            free = free + sizeof(o);
19)            将对象 o 拷贝到 NewLocation(o);
20)        }
21)    }
22)    return NewLocation(o);
23) }
```



拷贝回收器



回收前状态：

存活对象	存活对象	可回收	可回收	保留区域	保留区域	保留区域	保留区域
未使用	可回收	可回收	存活对象	保留区域	保留区域	保留区域	保留区域
未使用	可回收	存活对象	可回收	保留区域	保留区域	保留区域	保留区域

回收后状态：

保留区域	保留区域	保留区域	保留区域	存活对象	存活对象	存活对象	存活对象
保留区域	保留区域	保留区域	保留区域	未使用	未使用	未使用	未使用
保留区域	保留区域	保留区域	保留区域	未使用	未使用	未使用	未使用

存活对象

可回收

未使用

保留区域



分代回收



- 针对很多变量存在时间较短的特点设计
- 基本思想：将堆区进行切割成块并标号，垃圾回收技术每次在较低标号的块上进行
- 具体执行方案：
 - 新对象在编号较低的堆块上创建；
 - 每次在0号堆块被填满时刻，就对0号堆块进行垃圾回收；回收完毕后，将 i 号堆块往后移动到 $i+1$ 号堆块中
 - 维护一个记忆集，用以存储较高号的堆块对较低号堆块中对象的引用；



分代回收



- 采用该策略的基本原因：
 - 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；
 - 编程实践中，极少出现较老世代对较新世代的引用。因此，我们只需要记录该种情况，并定期采用其他策略进行处理即可。



GC的扩充内容



■ 为什么需要 GC?

- 计算机诞生初期，在程序运行过程中没有栈帧（**stack frame**）需要去维护
- 早期内存采取的是静态分配策略，这虽然比动态分配要快，但是其一明显的缺点是程序所需的数据结构大小必须在编译期确定，而且不具备运行时分配的能力



GC的扩充内容



■ GC的历史

- Algol-58 语言首次提出了块结构（**block-structured**），通过在内存中申请栈帧来实现按需分配的动态策略
- 在过程被调用时，帧（**frame**）会被压到栈的最上面，调用结束时弹出。栈分配策略赋予程序员极大的自由度，局部变量在不同的调用过程中具有不同的值，这为递归提供了基础
- 后进先出（**Last-In-First-Out, LIFO**）的栈限制了栈帧的生命周期不能超过其调用者，而且由于每个栈帧是固定大小，所以一个过程的返回值类型也必须在编译期确定。所以诞生了新的内存管理策略——堆管理



GC的扩充内容



■ GC的历史

- C/C++/Pascal 把堆管理这个任务交给了程序员，但事实证明这非常容易出错，野指针（wild pointer）、悬挂指针（dangling pointer）是比较典型的错误
- 实际应用的场景中，动态分配的对象传入了其他过程中，这时程序员或编译器就无法预测这个对象什么时刻不再需要，现如今的面向对象语言，这种场景更是频繁，这也就间接促进了自动内存管理技术的发展
- 实际上，即便是 C/C++，也有类似 Boehm GC 这样的第三方库来实现内存的自动管理



GC的实现常见策略



- 追踪（Tracing）
 - 标记清扫（Mark-Sweep）
 - 标记压缩（Mark-Compact）
 - 拷贝（Copying）
- 引用计数（Reference counting）



JAVA的GC设计



引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止



JAVA的GC设计



- **JAVA** 通过定义四种引用，提供了更为灵活的内存管理机制
 - 强引用(**StrongReference**)
 - 软引用(**SoftReference**)
 - 弱引用(**WeakReference**)
 - 虚引用(**PhantomReference**)



JAVA的GC设计



- **强引用(StrongReference):** 强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。

```
Object strongReference = new Object();  
public void test() {  
    Object strongReference = new Object();  
    // 省略其他操作  
}
```

显式地设置**strongReference**对象为null，或让其超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于GC算法。



JAVA的GC设计



■ 软引用(SoftReference):

- 如果一个对象只具有软引用，则内存空间充足时，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用
- 软引用可用来实现内存敏感的高速缓存

// 强引用

```
String strongReference = new String("abc");
```

// 软引用

```
String str = new String("abc");
```

```
SoftReference<String> softReference = new SoftReference<String>(str);
```



JAVA的GC设计



■ 软引用(SoftReference):

- 注意：软引用对象是在jvm内存不够的时候才会被回收，我们调用**System.gc()**方法只是起通知作用，**JVM**什么时候扫描回收对象是**JVM**自己的状态决定的。就算扫描到软引用对象也不一定会回收它，只有内存不够的时候才会回收。
- 垃圾收集线程会在虚拟机抛出**OutOfMemoryError**之前回收软引用对象，而且虚拟机会尽可能优先回收长时间闲置不用的软引用对象。对那些刚构建的或刚使用过的较新的软对象会被虚拟机尽可能保留。



JAVA的GC设计



- 软引用应用场景：**浏览器的后退按钮**
 - 如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建；
 - 如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出。



JAVA的GC设计



■ 软引用应用场景：浏览器的后退按钮

```
// 获取浏览器对象进行浏览
Browser browser = new Browser();
// 从后台程序加载浏览页面
BrowserPage page = browser.getPage();
// 将浏览完毕的页面置为软引用
SoftReference softReference = new SoftReference(page);

// 回退或者再次浏览此页面时
if(softReference.get() != null) {
    // 内存充足，还没有被回收器回收，直接获取缓存
    page = softReference.get();
} else {
    // 内存不足，软引用的对象已经回收
    page = browser.getPage();
    // 重新构建软引用
    softReference = new SoftReference(page);
}
```



JAVA的GC设计



■ 弱引用(WeakReference):

- 弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存
- 由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象

```
String str = new String("abc");  
WeakReference<String> weakReference = new WeakReference<>(str);  
String strongReference = weakReference.get();
```



JAVA的GC设计



■ 虚引用(PhantomReference):

- 与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收
- 虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列(**ReferenceQueue**)联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中



JAVA的GC设计



■ 虚引用应用场景：虚引用主要用来跟踪对象被垃圾回收器回收的活动

- 与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收

```
String str = new String("abc");  
ReferenceQueue queue = new ReferenceQueue();  
// 创建虚引用，要求必须与一个引用队列关联  
PhantomReference pr = new PhantomReference(str, queue);
```