第七章 运行时刻环境

《编译原理》

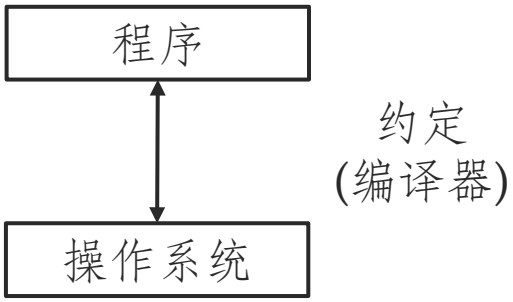
谭添

南京大学计算机系

2026年春季

运行时刻环境

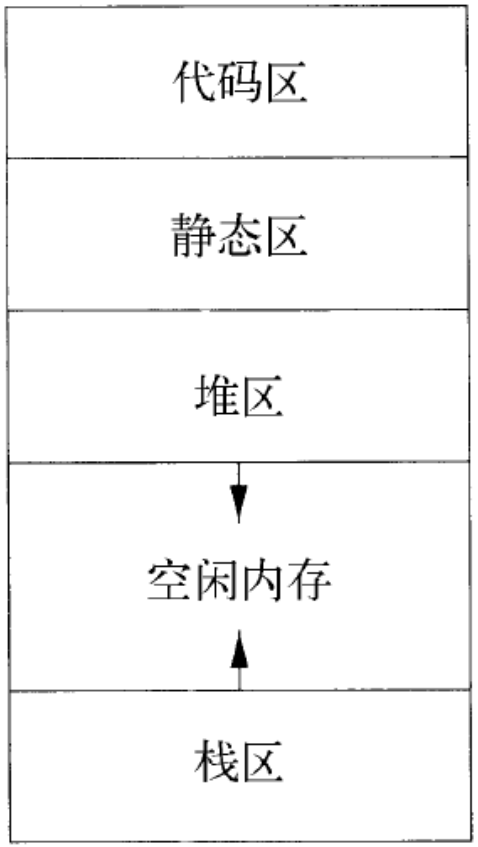
- 运行时刻环境
  - 为数据分配安排存储位置
  - 确定访问变量时使用的机制
  - 过程之间的连接、参数传递
  - 和操作系统、输入输出设备相关的其它接口



- 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



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
  - 全局常量、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，值的生命期与过程生命期相同
  - **堆存储**：数据对象可比创建它的过程调用更长寿
    - 手工进行回收
    - 垃圾回收机制

栈式分配

- 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
    - 表示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
  - 又称为调用树 (Call Tree)

活动树的例子

- 快速排序程序
  - 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()
```

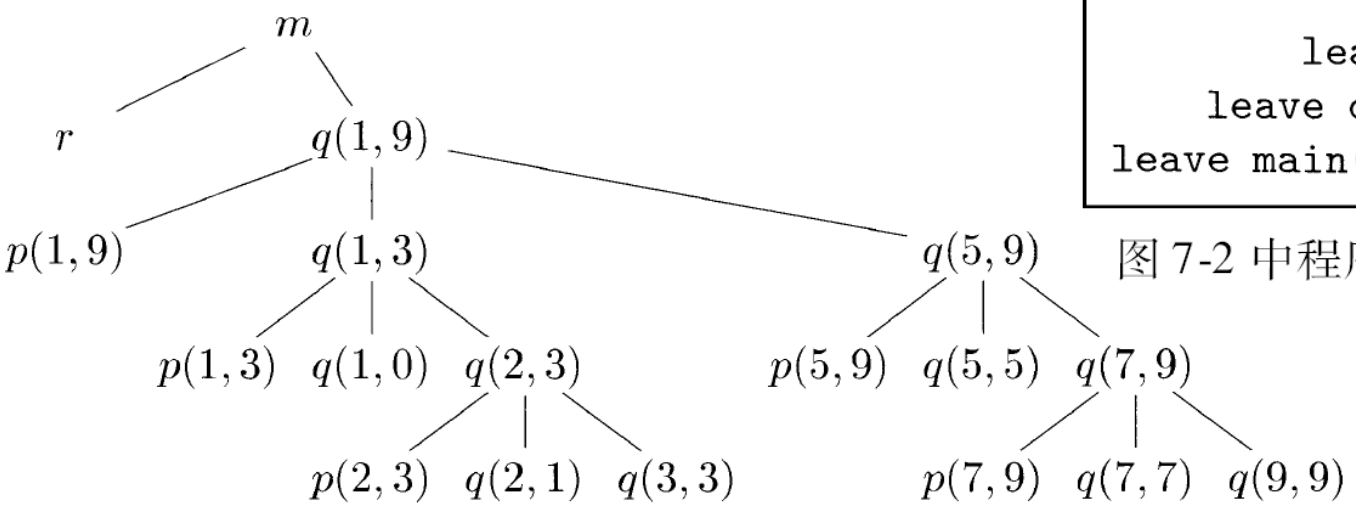
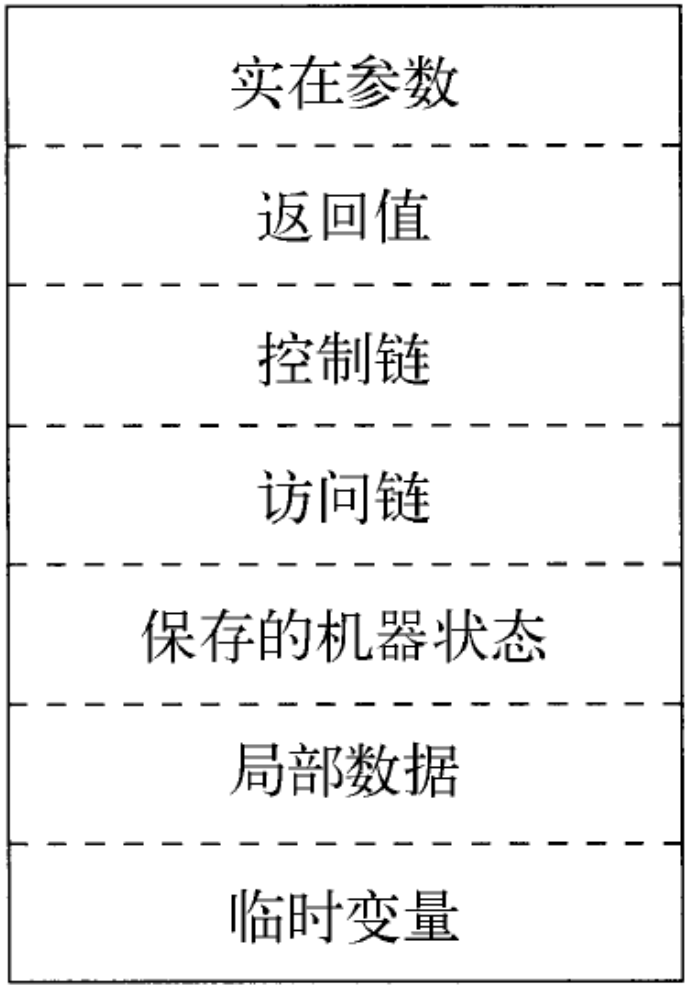


图 7-2 中程序的可能的活动序列

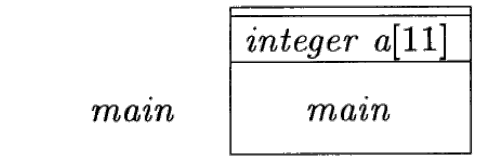
活动记录

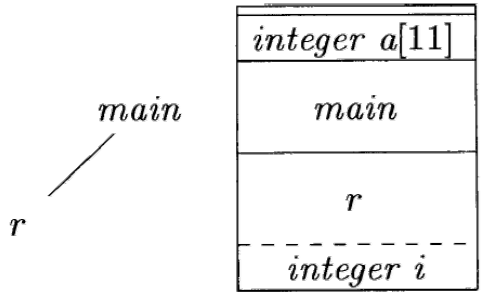
- 过程调用和返回由控制栈进行管理
- 每个活跃的活动对应于栈中的一个**活动记录**
- 活动记录按照活动的开始时间，从栈底到栈顶排列



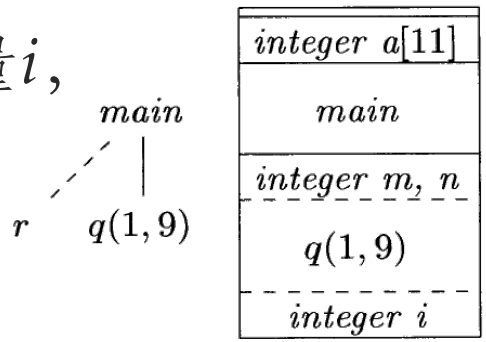
运行时刻栈的例子

- 说明
  - a[11]为全局变量
  - main无局部变量
  - r有局部变量i
  - q有局部变量i, 和参数m, n

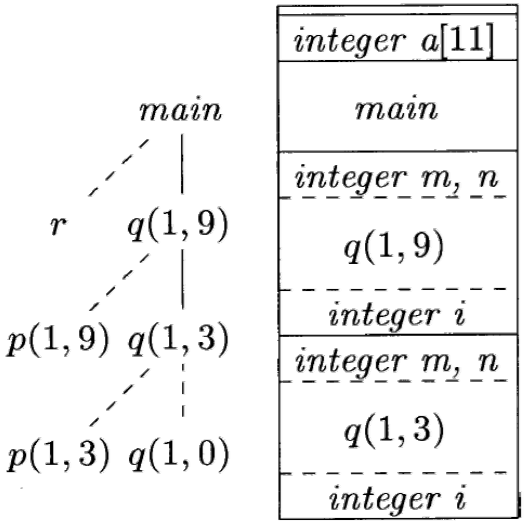




b) r被激活



c) r被弹出栈 ,$q(1,9)$ 被压栈



d) 控制返回到 $q(1, 3)$

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) 恢复调用者状态，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、目标机器和操作系统的限制，可以有不同的分割方案
  - 把代码尽可能放在被调用者中

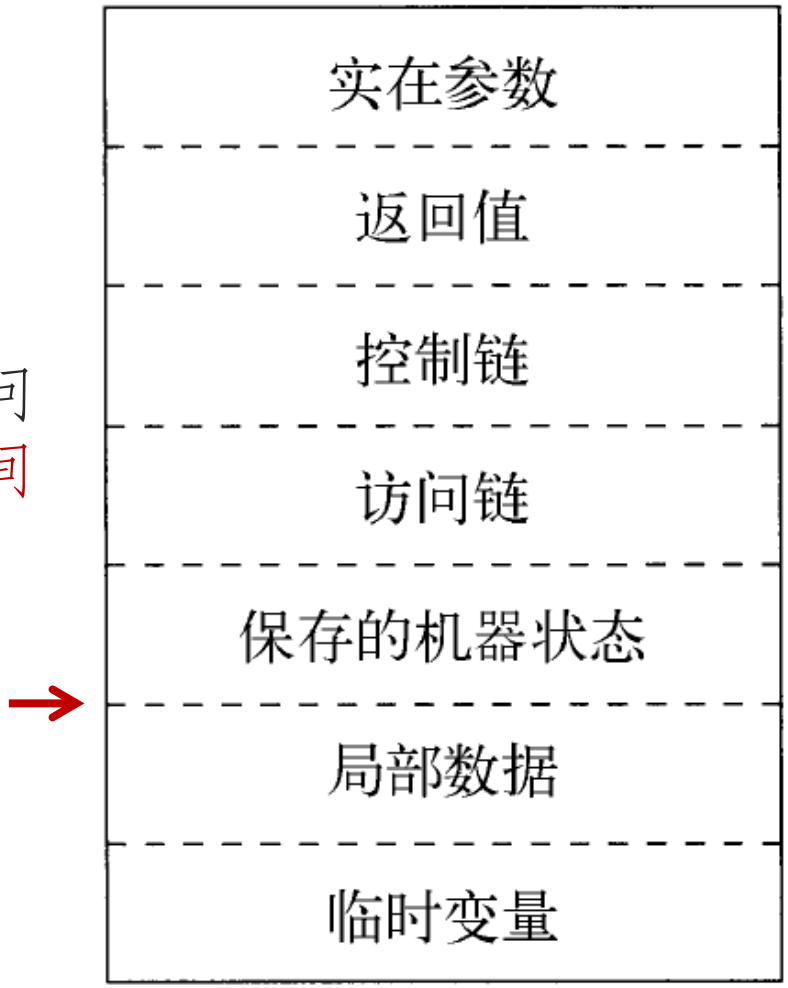
调用/返回代码序列的要求

- 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)

- 调用代码序列与活动记录的布局相关

活动记录的布局原则

- 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 (控制链、访问链和机器状态字段) 放在中间位置
  - 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

- Calling sequence
  - 调用者计算实在参数的值
  - 将返回地址和原*top_sp*存放到被调用者的活动记录中；调用者增加*top_sp*的值 (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)
  - 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
- Return sequence
  - 被调用者将返回值放到与参数相邻的位置
  - 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

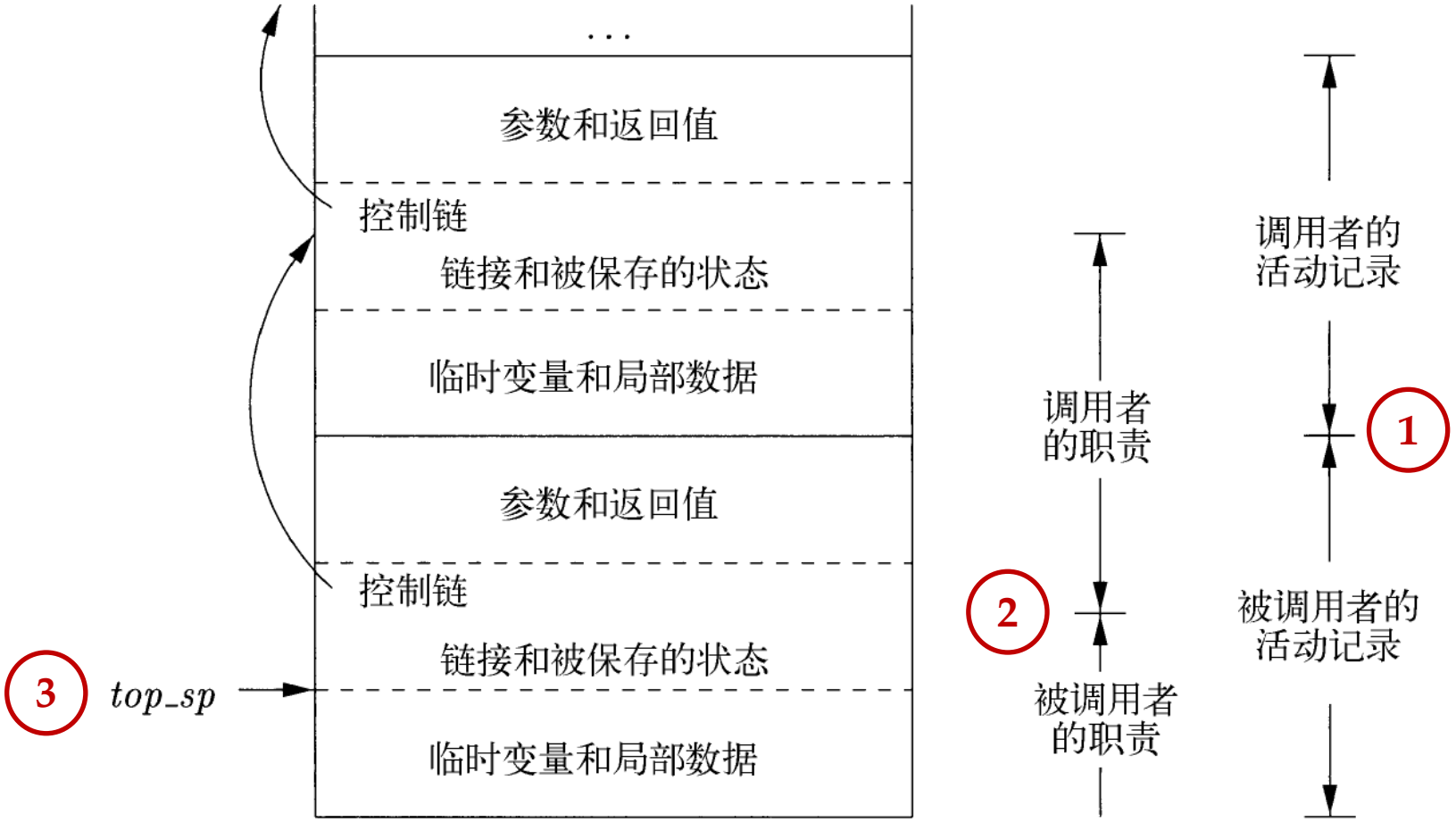


图 7-7　调用者和被调用者之间的任务划分

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 (有嵌套过程)

- 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，必须通过访问链访问

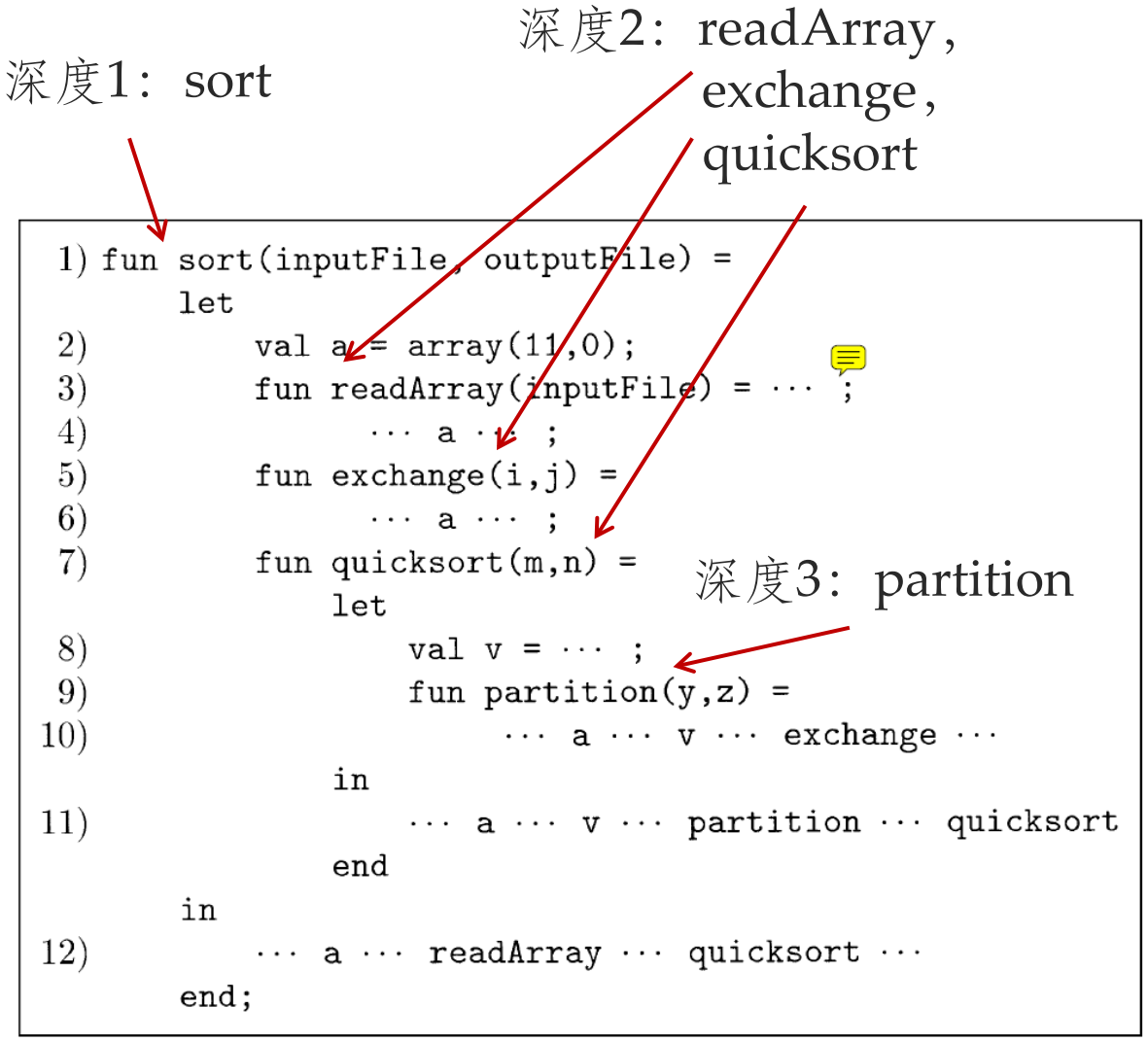
```
void A() {
    int x, y;
    void B() {
        int b;
        x = b + y;
    }
    void C() { B(); }
    C();
    B();
}
```

当*A*调用*C*，*C*又调用*B*时:

当*A*直接调用*B*时:

嵌套深度

- 嵌套深度可根据源程序静态确定
  - 不内嵌于任何其它过程的过程，深度为1
  - 嵌套于深度为i的过程的过程，深度为$i+1$



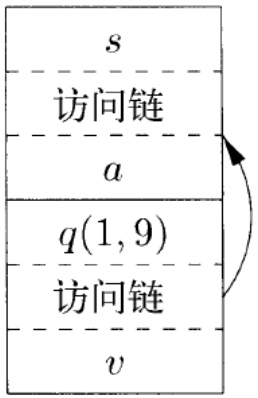
访问链和访问链的使用

- 访问链被用于访问非局部的数据
  - 如果过程p在声明时(直接)嵌套在过程q中，那么p活动记录中的访问链指向上层最近的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
- 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q中声明
  - $n_p - n_q$在编译时刻已知；从当前活动记录出发，沿访问链前进$n_p - n_q$次找到活动记录
  - x相对于这个活动记录的偏移量在编译时刻已知

访问链的维护

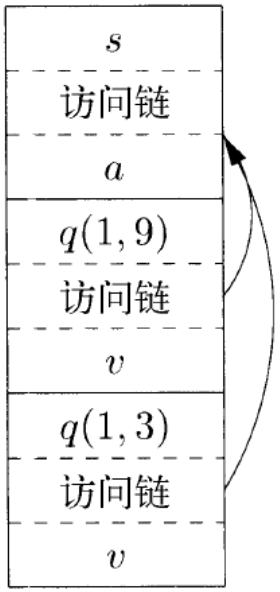
- 当过程q调用过程p时　　$p.access = q$
  - p的深度大于q：根据作用域规则，p必然在q中直接定义；那么p的访问链指向当前活动记录(即q)
    - 例如：*sort*调用*quicksort*(1, 9)
  - 递归调用$p = q$：新活动记录的访问链等于当前记录的访问链(即和前一个q指向同一目标)　　$p.access = q.access$
    - 例如：*quicksort*(1, 9)调用*quicksort*(1, 3)
  - p的深度小于等于q的深度：必然有过程r，p直接在r中定义，而q嵌套在r中；p的访问链指向栈中r的活动记录
    - 例如：*partition*调用*exchange*　　$p.access = p.outer$

访问链的例子



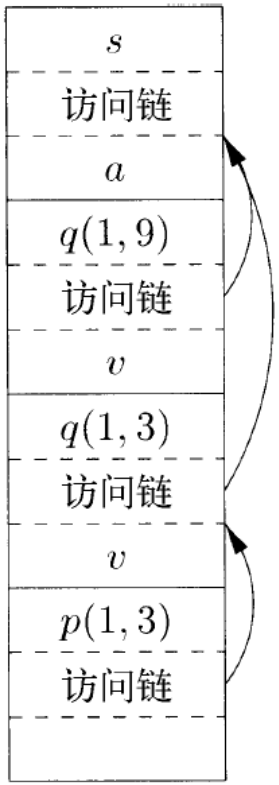
a)

$q.access = s$



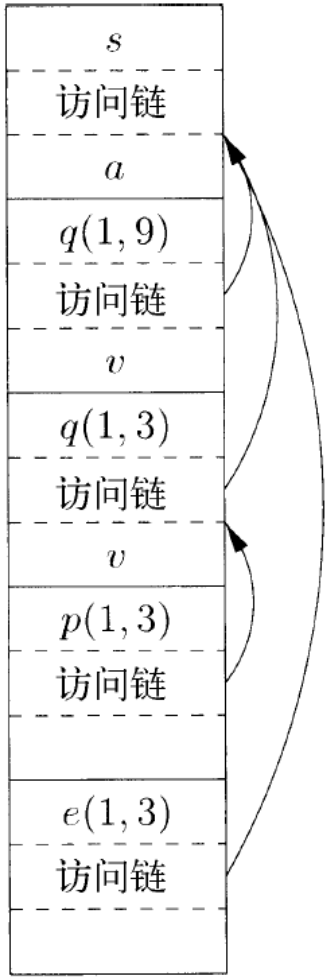
b)

$q_2.access = q_1.access\ (s)$



c)

$p.access = q_2$



d)

$e.access = e.outer$

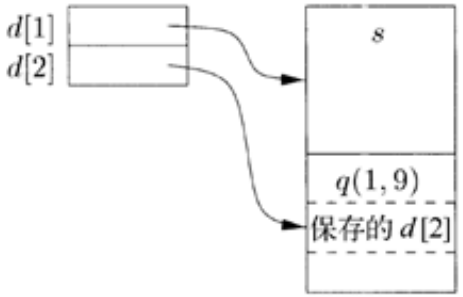
访问链的维护 (过程指针型参数)

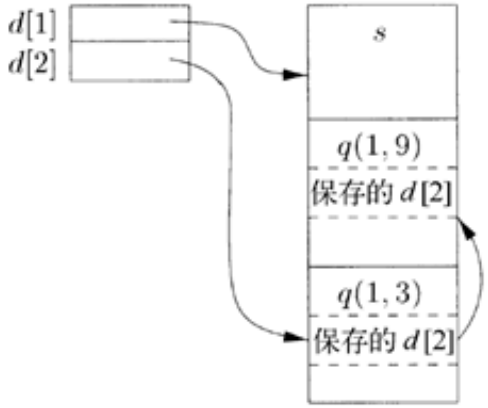
- 在传递过程指针参数时，过程型参数中不仅包含过程的代码指针 (开始地址)，还包括正确的访问链

显示表

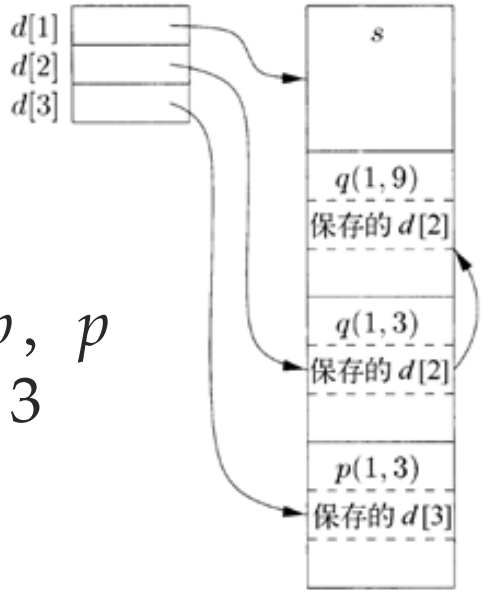
- 用访问链访问数据，访问开销和嵌套深度差有关
  - 使用显示表可以提高效率，访问开销为常量
- **显示表**：数组d为每个嵌套深度保留一个指针
  - 指针$d[i]$指向栈中最近的、嵌套深度为i的活动记录
  - 如果过程p访问嵌套深度为i的过程q中声明的变量x，那么$d[i]$直接指向相应的活动记录 (i在编译时刻已知)
- 显示表的维护
  - 调用过程p时，在p的活动记录中保存$d[n_p]$的值，并将$d[n_p]$设置为当前活动记录 (即p)
  - 从p返回时，恢复$d[n_p]$的值

显示表的例子

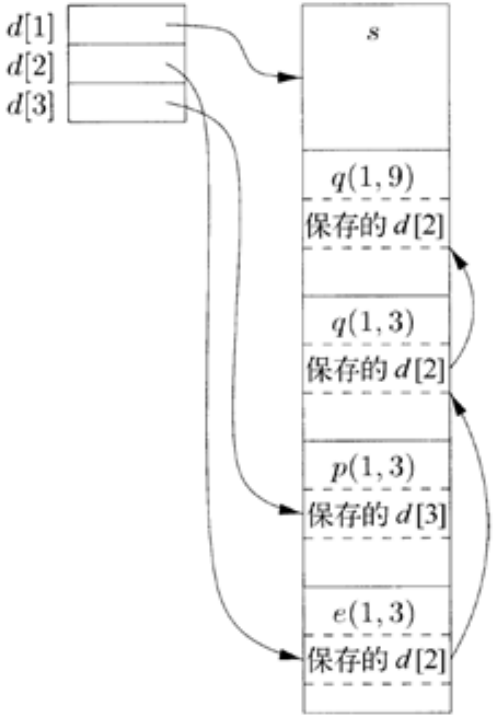




$q(1,9)$调用$q(1,3)$时，q的深度为2



$q(1,3)$调用p，p的深度为3



p调用e，e的深度为2

非局部变量的作用域

- 静态作用域 (Static Scoping)
  - 又称词法作用域 (Lexical Scoping)
  - 根据函数声明时的位置寻找变量定义（静态时决定）
  - 优点：程序行为易于推理

- 动态作用域 (Dynamic Scoping)
  - 根据函数运行时的上下文寻找变量定义（动态时决定）
  - 优点：易于实现 (低情商：偷懒)

闭包 (Closure)

- 一个过程如果记录了其创建时的环境 (该过程引用的变量以及它们的值)，则称之为一个闭包
  - 若在过程p内定义了过程q，则过程p的调用退出后，其局部变量并不会立即释放，因为后续有可能被q访问

- 不同语言有不同程度的支持
  - 无闭包：C
  - 半吊子闭包：Java、C++
  - 完全体闭包：Scheme、Python、JavaScript

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)
    - Rust也可以自动回收空间 (Ownership)

存储管理器

- 基本功能
  - **分配**：为内存请求分配一段连续、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，以满足其它内存需求
- 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：运用内存系统的层次，使程序运行更快
  - 低开销：使分配/收回内存的操作尽可能高效

计算机的存储层次结构

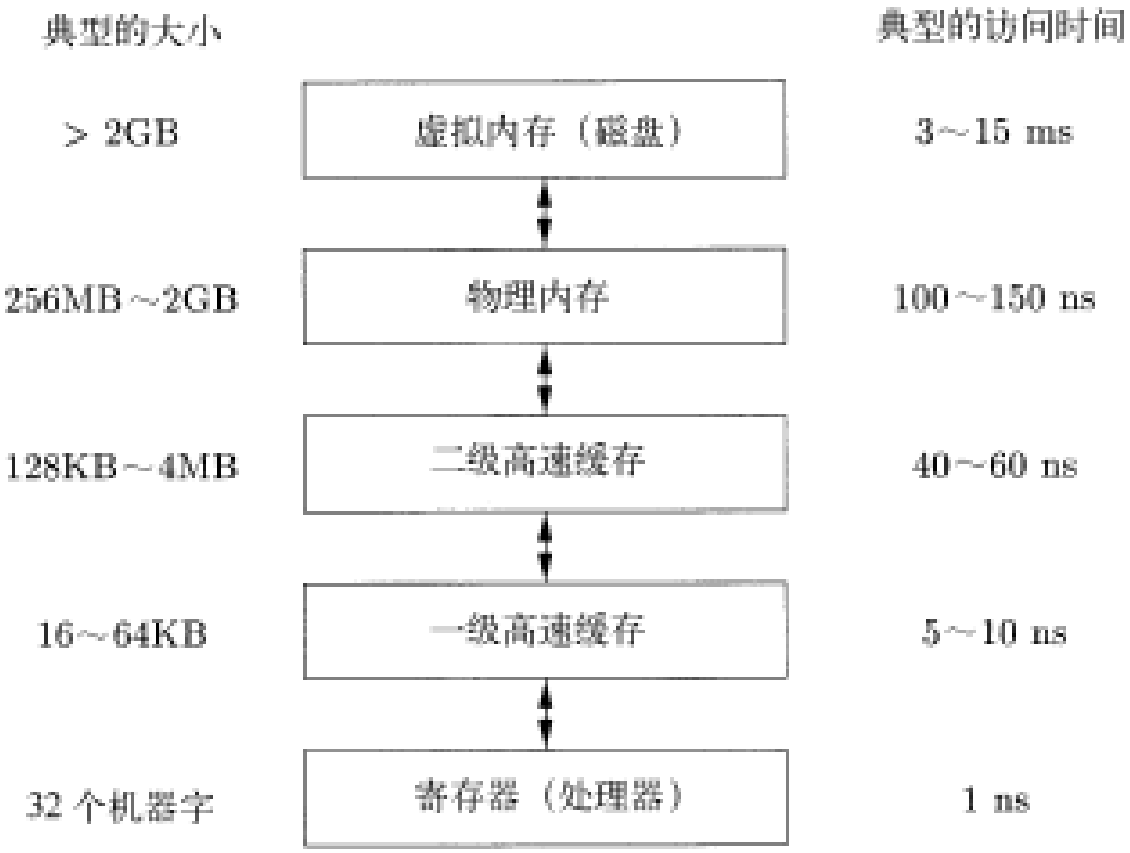


图 7-16　典型的内存层次结构的配置

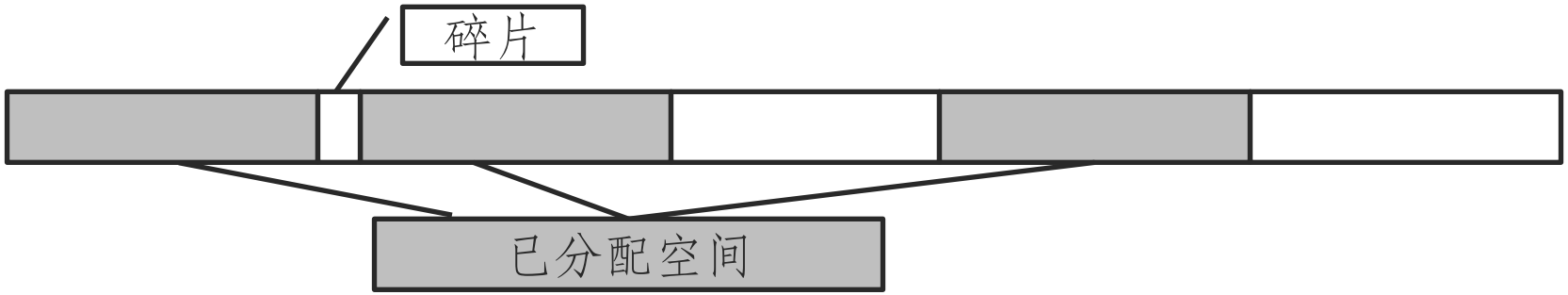
程序中的局部性

- 程序具有高度的**局部性** (Locality)
  - **时间局部性**：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
  - **空间局部性**：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

- 90%的时间用来执行10%的代码
- 局部性这一特性恰好可以充分利用计算机的层次存储结构

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)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
  - 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

- 设定不同大小的块规格，相同的块放入同一容器
- 较小的 (较常用的) 尺寸设置较多的容器
- 如GNU的C编译器将所有存储块对齐到8字节边界
  - 空闲块的大小：
    - 16, 24, 32, 40, … , 512
    - 大于512的按对数划分：每个容器的尺寸是前一容器的两倍
    - **荒野块**：可以扩展的内存块
  - 分配方法
    - 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

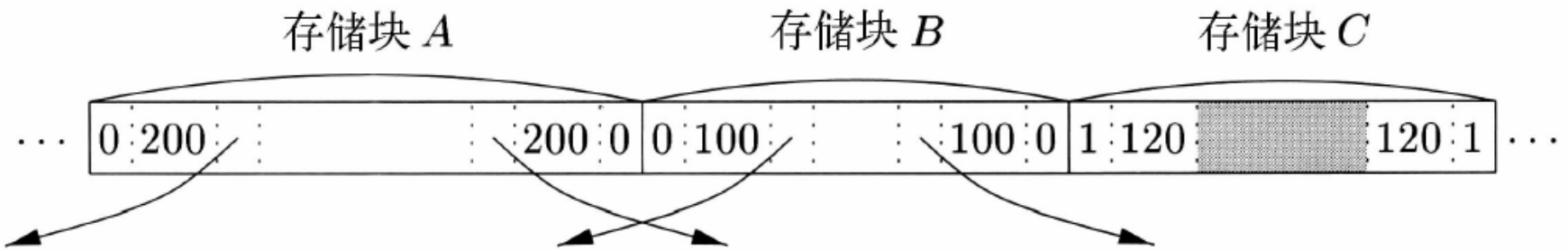
管理和接合空闲空间

- 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
  - 有些管理方法不需要进行接合

- 支持相邻块接合的数据结构
  - **边界标记**：在每个存储块的两端，分别设置一个*free/used*位，并在相邻的位置上存放字节总数
  - 双重链接的空闲块列表：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** (Memory leak)：未能删除不可能再被引用的数据
  - **悬空指针引用** (Dangling pointer)：引用已被删除的数据

- 其他问题
  - 空指针访问/数组越界访问

垃圾回收

- **垃圾**
  - 广义：不需要再被引用的数据
  - 狭义：不能被引用 (不可达) 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：Lisp、Java、C#、ML、Python、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
  - 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
  - 空间使用：最大限度地利用可用内存
  - 程序局部性：改善空间局部性和时间局部性

可达性

- **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
- 引用赋值：$u = v$ $(u: x / o.f / a[i])$
  - v的引用被复制到u中，u中原有引用丢失；(可能) 使u原来指向的对象变得不可达，并递归使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，使一些对象变得不可达

垃圾回收方法

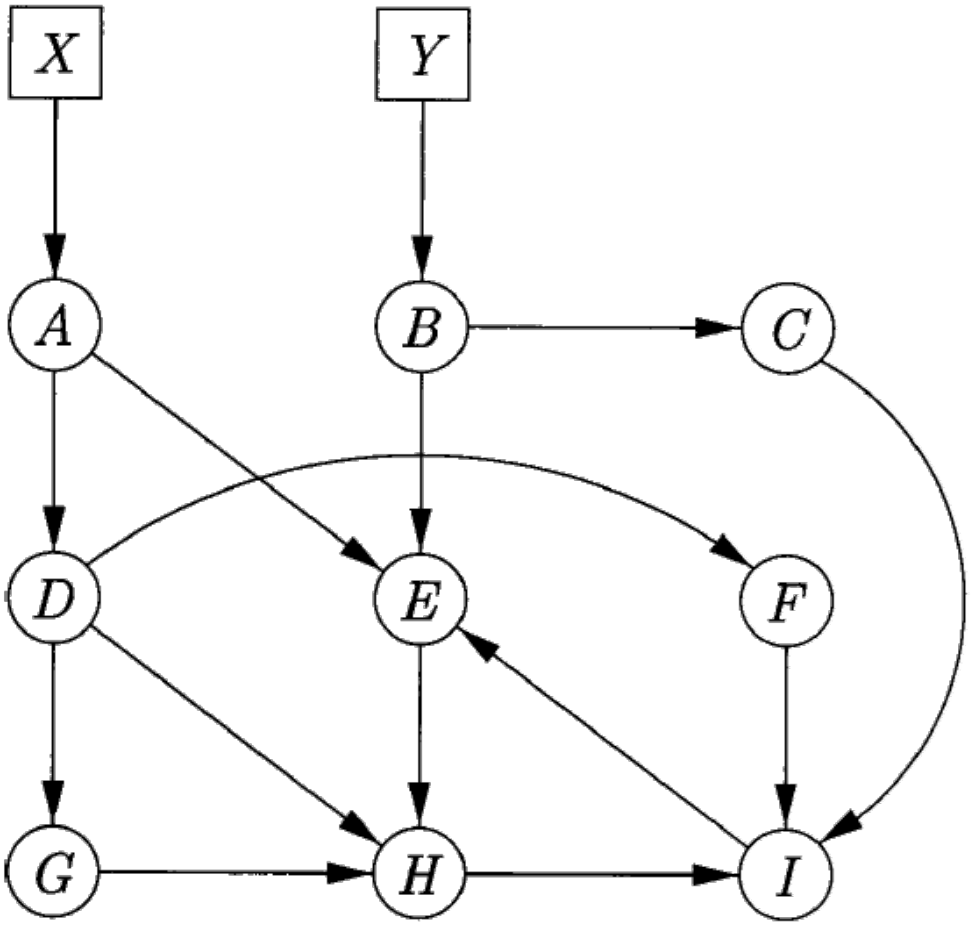
- 关注不可达
  - 跟踪相关操作，捕获对象变得不可达的时刻，回收对象占用的空间
- 关注可达
  - 在需要时，标记出所有可达对象，回收其它对象

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
- 开销较大，但不会引起停顿

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

- 三个对象相互引用，没有来自外部的指针，又不是根集成员，都是垃圾，但是引用计数都大于0

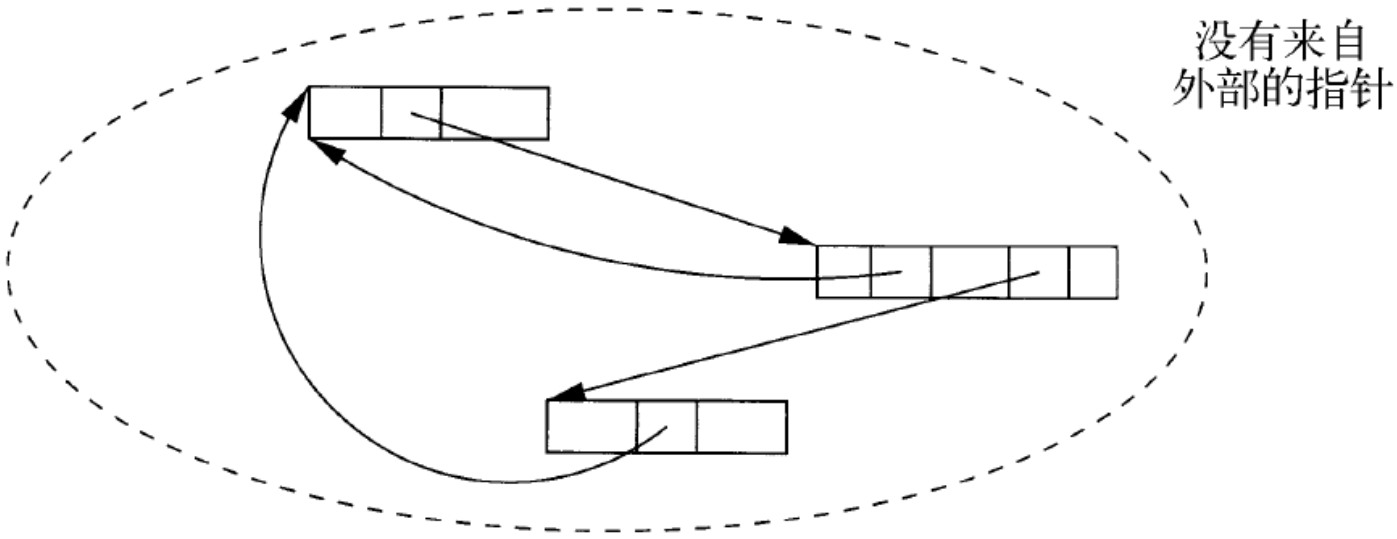


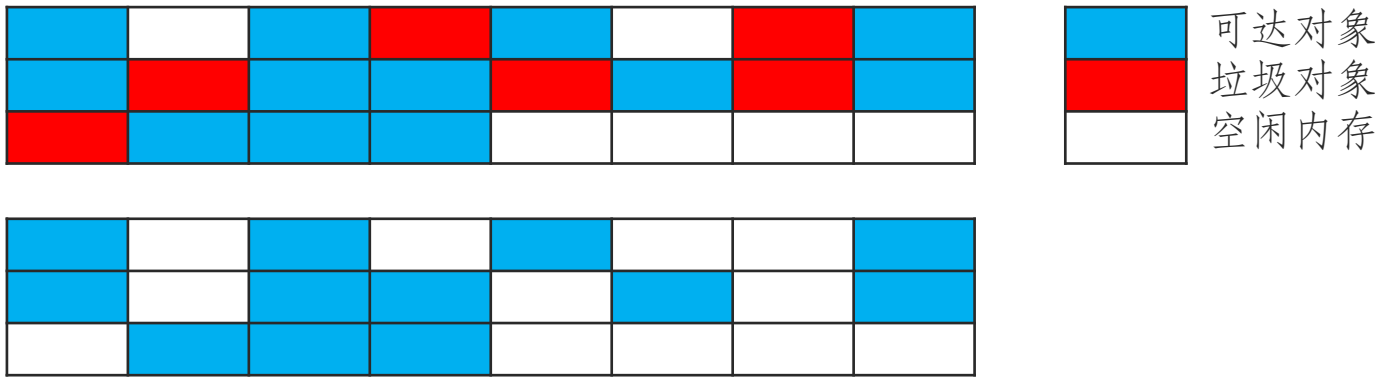
图 7-18　一个不可达的循环数据结构

基于跟踪的垃圾回收

- 标记-清扫式垃圾回收
- 标记-拷贝式垃圾回收
- 标记-整理式垃圾回收
- 分代式垃圾回收

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，跟踪并标记出所有的可达对象
  - **清扫**：遍历整个堆区，释放不可达对象



- 如果把数据对象看作结点，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过程

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 ≠ ∅) {
3)          从 Unscanned 列表中取出某个对象 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；
    }
```

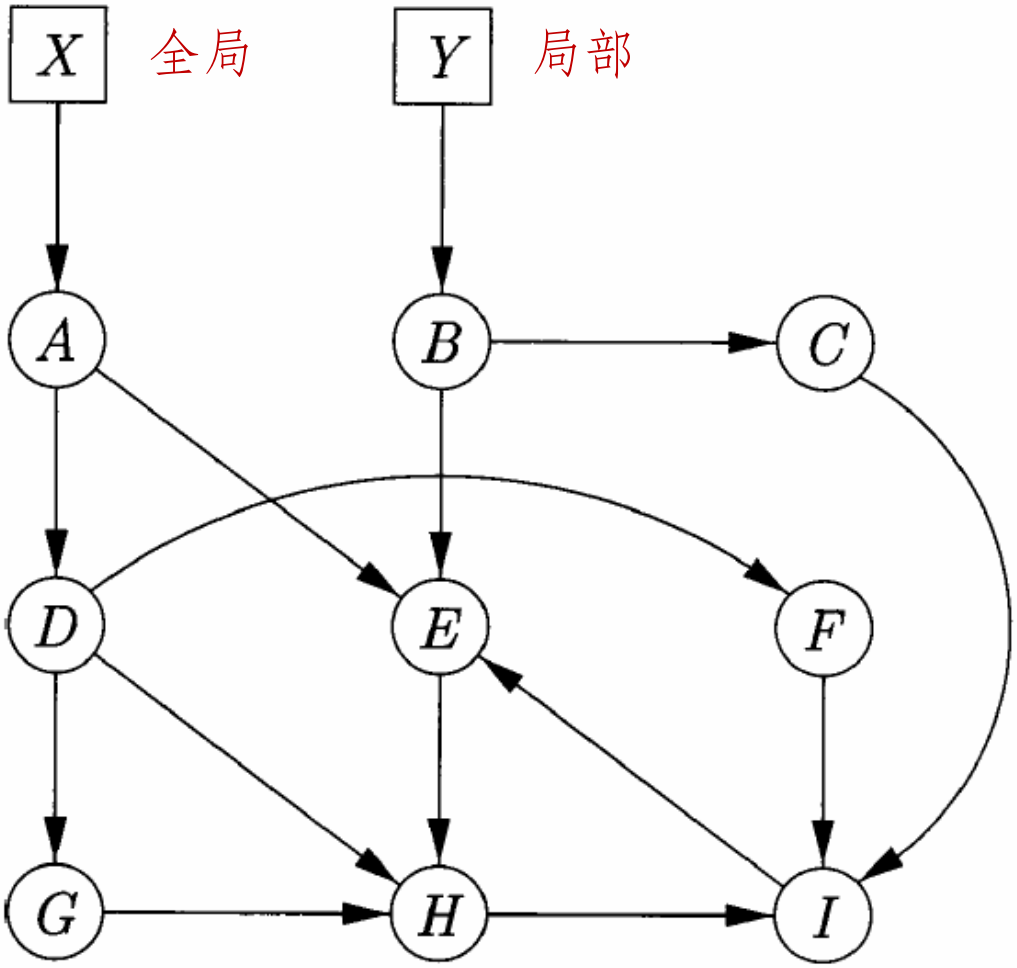
初始化，根集已知可达

为下一次标记-清扫做准备

垃圾回收机制需要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

例子

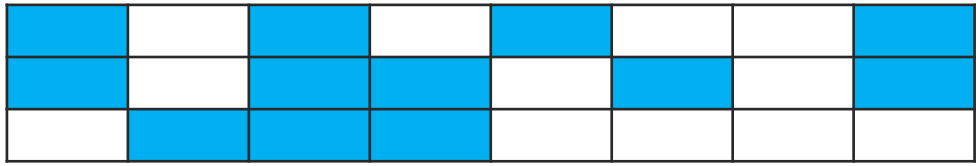
- 假设
  - X是全局变量
  - Y是当前函数的局部变量
- 函数返回后进行标记清扫
  - A, D, E, F, G, H, I可达
  - B, C不可达



标记-清扫式垃圾回收

- 优点
  - 实现简单，无需移动对象 (修改引用地址)
- 缺点
  - 效率堪忧，清扫阶段总是需要遍历所有内存
  - 易产生内存碎片

```
     /* 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;
     }
```

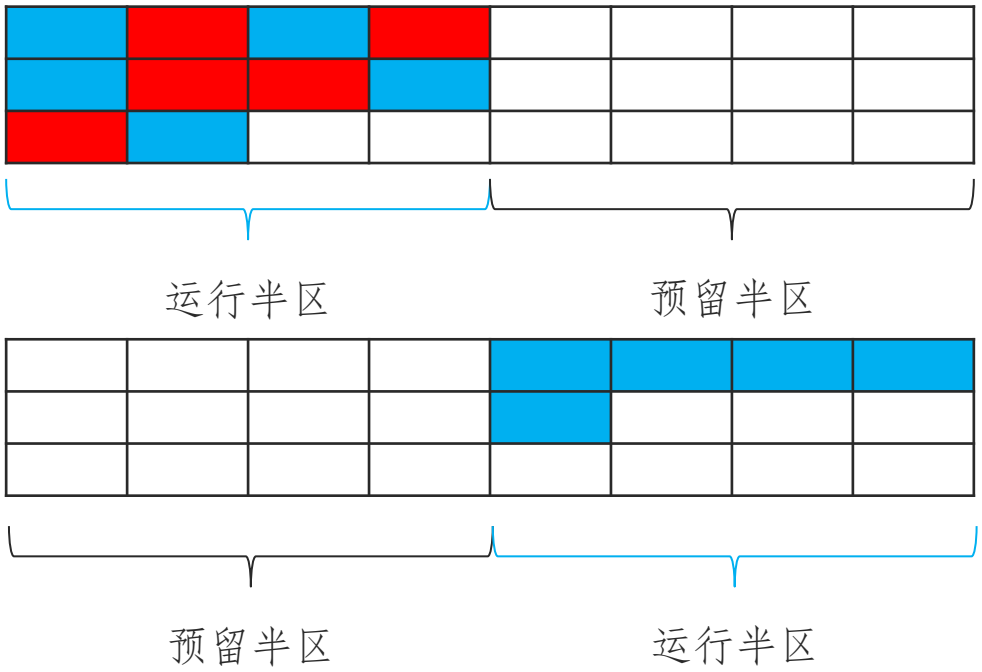


标记-复制式垃圾回收

- 目标
  - 只处理可达对象 (减少回收时间)
  - 将可达对象安排到一起 (减少内存碎片)

- 堆空间被分为两个半区
  - 应用程序在某个半区内分配存储，当充满这个半区时，开始垃圾回收
  - 回收时，复制可达对象到另一个半区 (需修改引用地址)
  - 回收完成后，两个半区角色对调

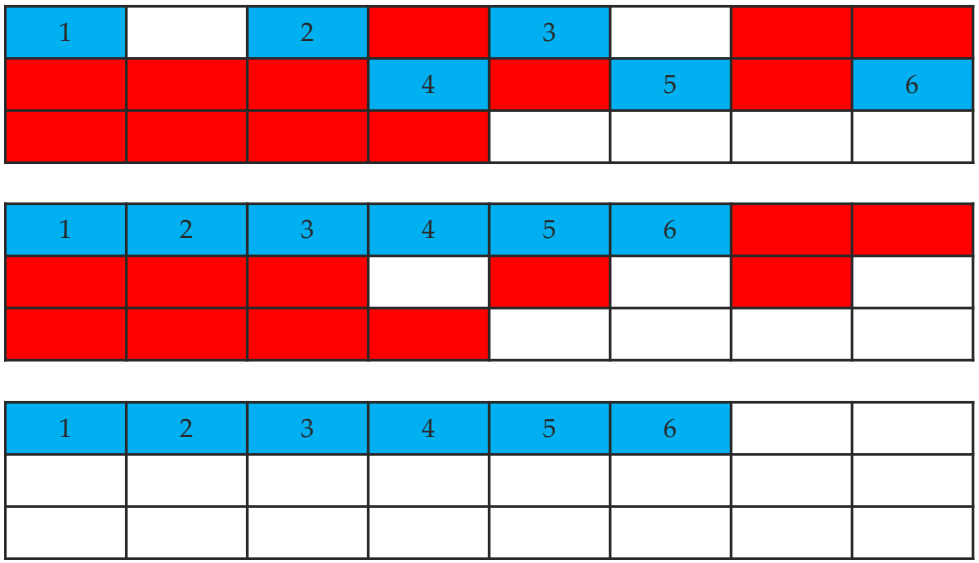
标记-复制式垃圾回收



- 优点
  - 只处理可达对象(减少回收时间)
  - 将可达对象安排到一起(减少内存碎片)
- 缺点
  - 一半预留区域的内存无法使用

标记-整理式垃圾回收

- 对可达对象进行**重定位**可以消除存储碎片
  - 与标记-复制法将对象移动到预留另一半区不同，标记-整理法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，提高分配内存时的效率



标记-整理式垃圾回收

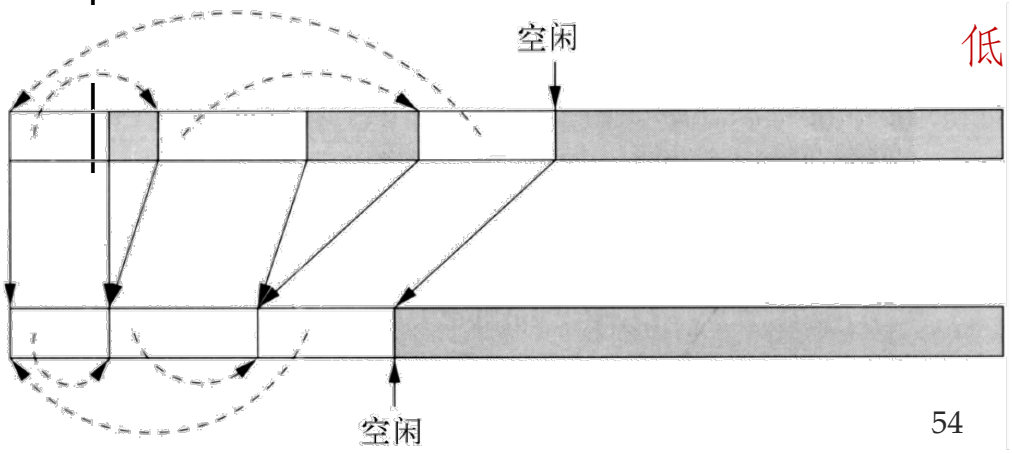
```
    /* 标记 */
1)  Unscanned = 根集引用的对象的集合;
2)  while (Unscanned ≠ ∅) {
3)      从 Unscanned 中移除对象o;
4)      for (在o 中引用的每个对象 o′ ) {
5)          if (o′ 是未访问的) {
6)              将o′ 标记为已访问的;
7)              将o′ 加入到列表 Unscanned 中;
            }
        }
    }
    /* 计算新的位置 */
8)  free = 堆区的开始位置;
9)  for (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);
        }
    }
```

已访问的即可达的

```
    /* 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);
```

整个过程分成三个步骤

- 标记(与前述算法相同)
- 计算新位置
- 移动并设置新的引用

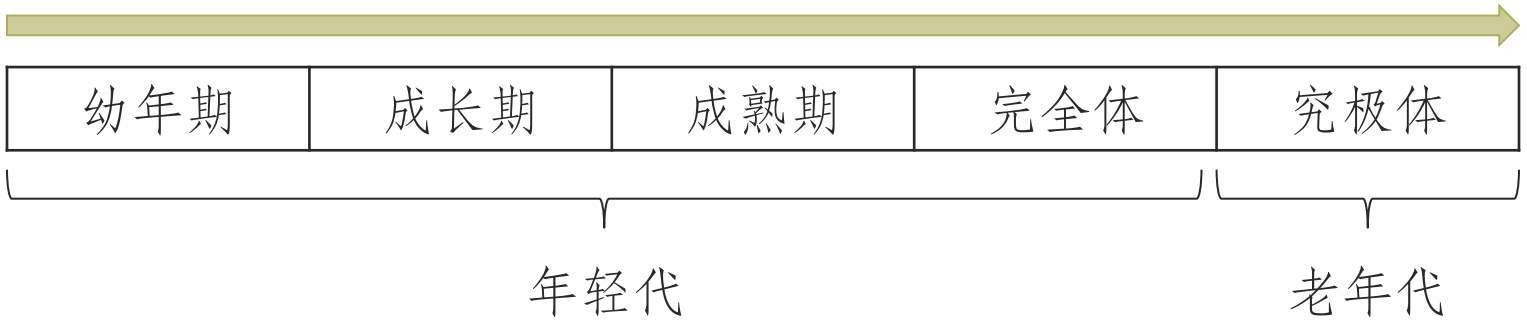


垃圾回收算法比较

- 标记-清扫式垃圾回收
  - 回收效率不高，易造成碎片
- 标记-复制式垃圾回收
  - 将可达对象复制到预留半区，只需遍历可达对象
  - 管理区域内大部分对象为垃圾对象时效率高 (只需移动少量可达对象)，反之则效率低
- 标记-整理式垃圾回收
  - 将可达对象移动到堆区的一端，需遍历整个区域
  - 管理区域内大部分对象为可达对象时效率高 (经过之前的整理、大部分对象已经到位)，反之则效率低

分代式垃圾回收

- 对象生存周期特征
  - 大多数对象刚创建不久后就不再使用 (短命)
  - 存在时间越长的对象，通常被回收的几率越小 (命硬)
- 根据生存周期，对对象分代管理
  - 新创建的对象均放入年轻代区域 (从幼年期开始)
  - 年轻代空间不足时，对该区域执行回收 (标记-复制)
  - 熬过回收的对象移入下一区域 (进化)
  - 老年代空间不足时，对该区域执行回收 (标记-整理)



总结

- 内存空间布局
  - 代码区、静态区、栈区、堆区
- 栈区内存管理
  - 活动记录：函数调用的核心数据结构
  - 进入/返回函数的操作
- 堆区内存管理
  - 手动管理
  - 自动管理 (垃圾回收)