

# 第七章 运行时刻环境

许畅

南京大学计算机系

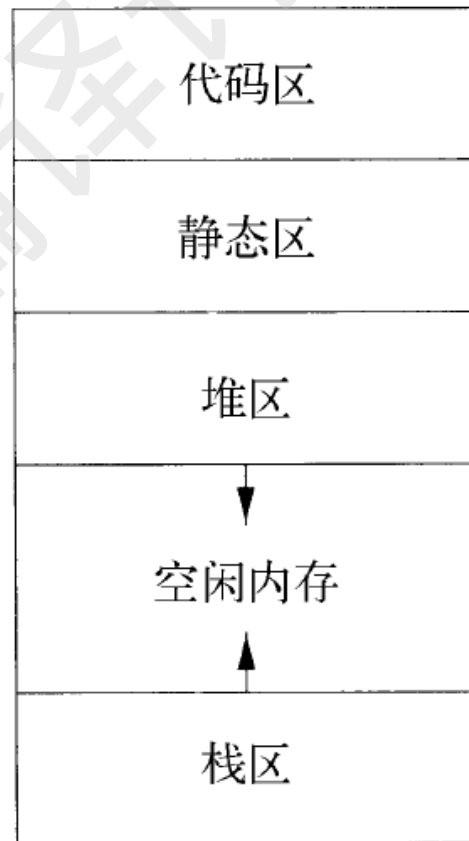
2020年春季

# 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

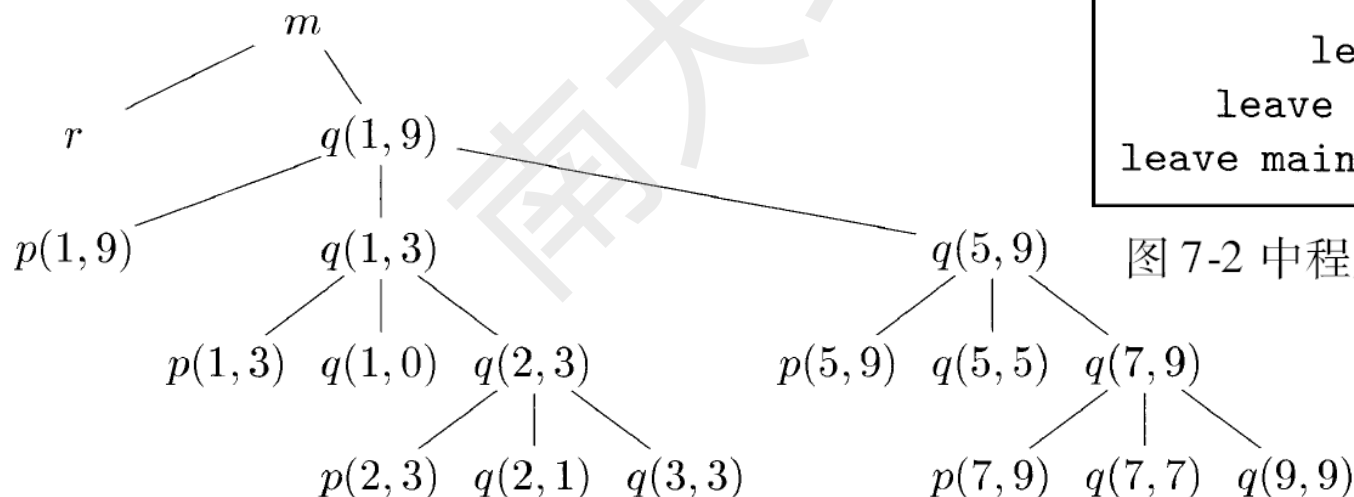
- 内容
  - 活动树
  - 活动记录
  - 调用代码序列
  - 栈中的变长数据

# 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

# 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

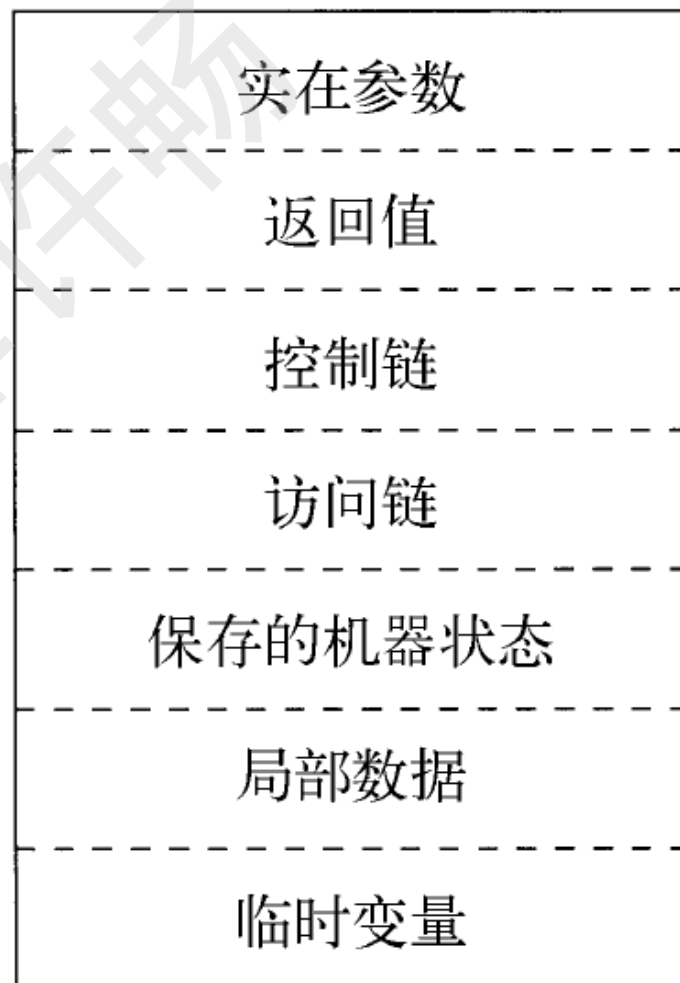


```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
      ...
    leave quicksort(1,3)
    enter quicksort(5,9)
      ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

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

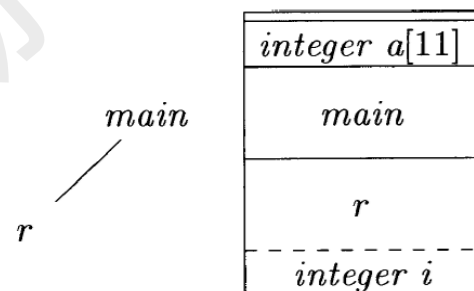
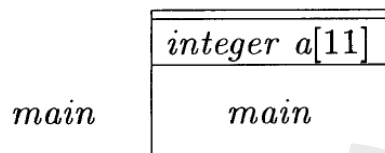




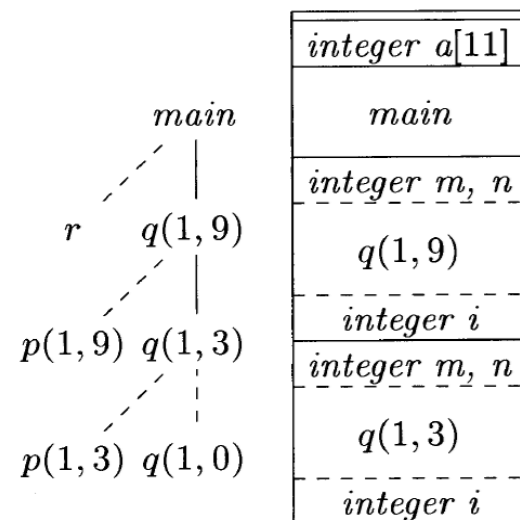
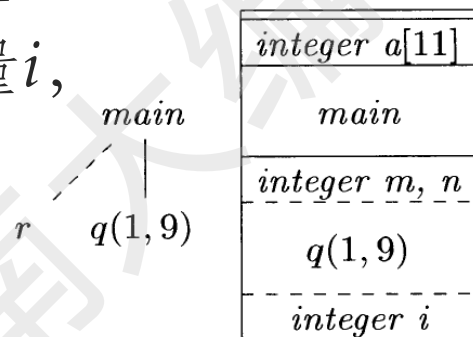
# 运行时刻栈的例子

## 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

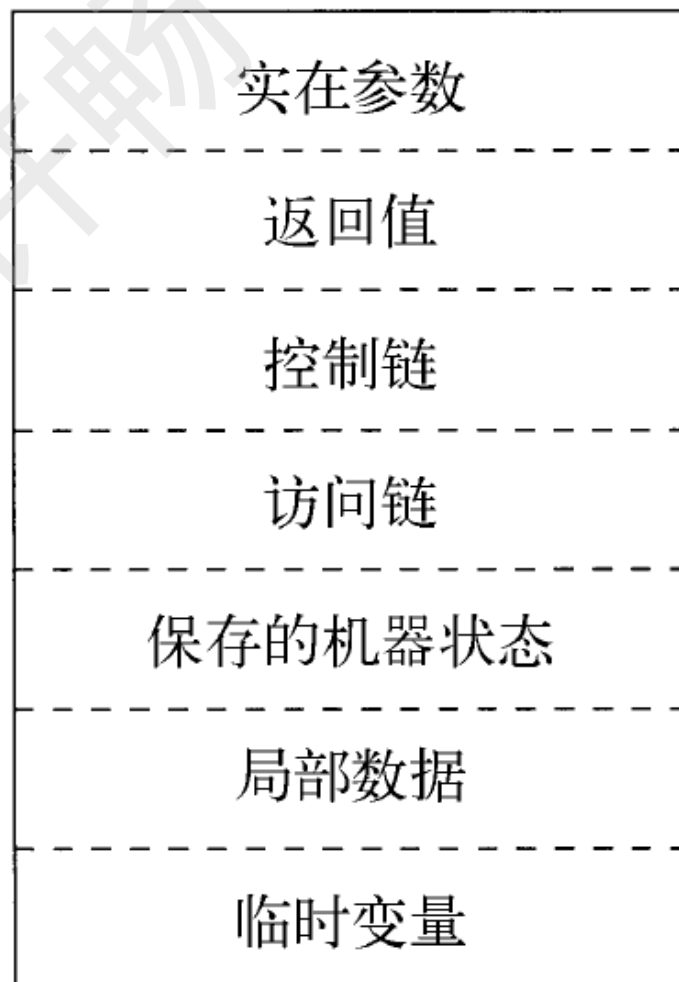
# 调用/返回代码序列的要求

- 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 to 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

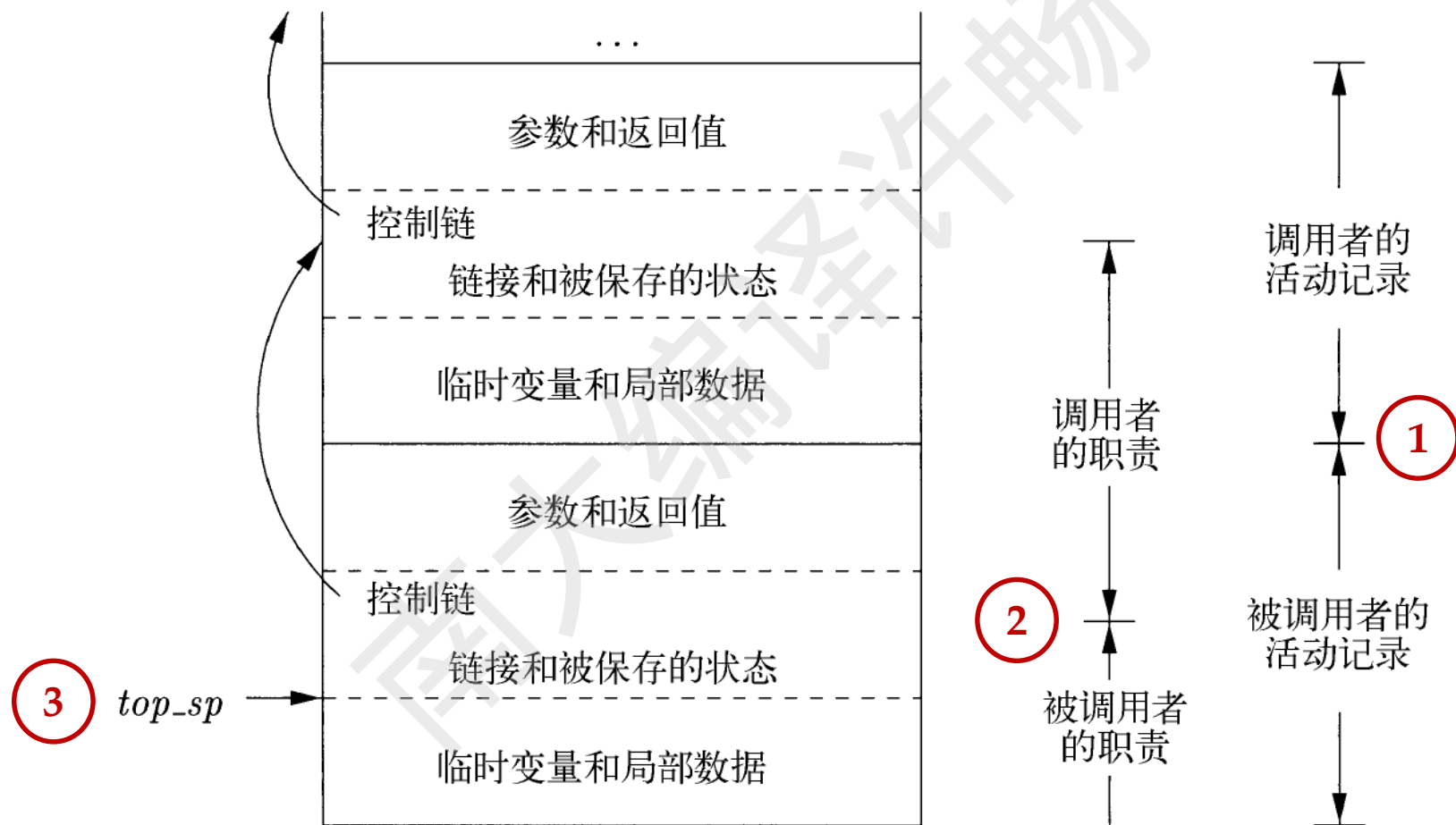


图 7-7 调用者和被调用者之间的任务划分

# 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
- $top$ 指向实际栈顶
- $top\_sp$ 用于寻找顶层记录的定长字段

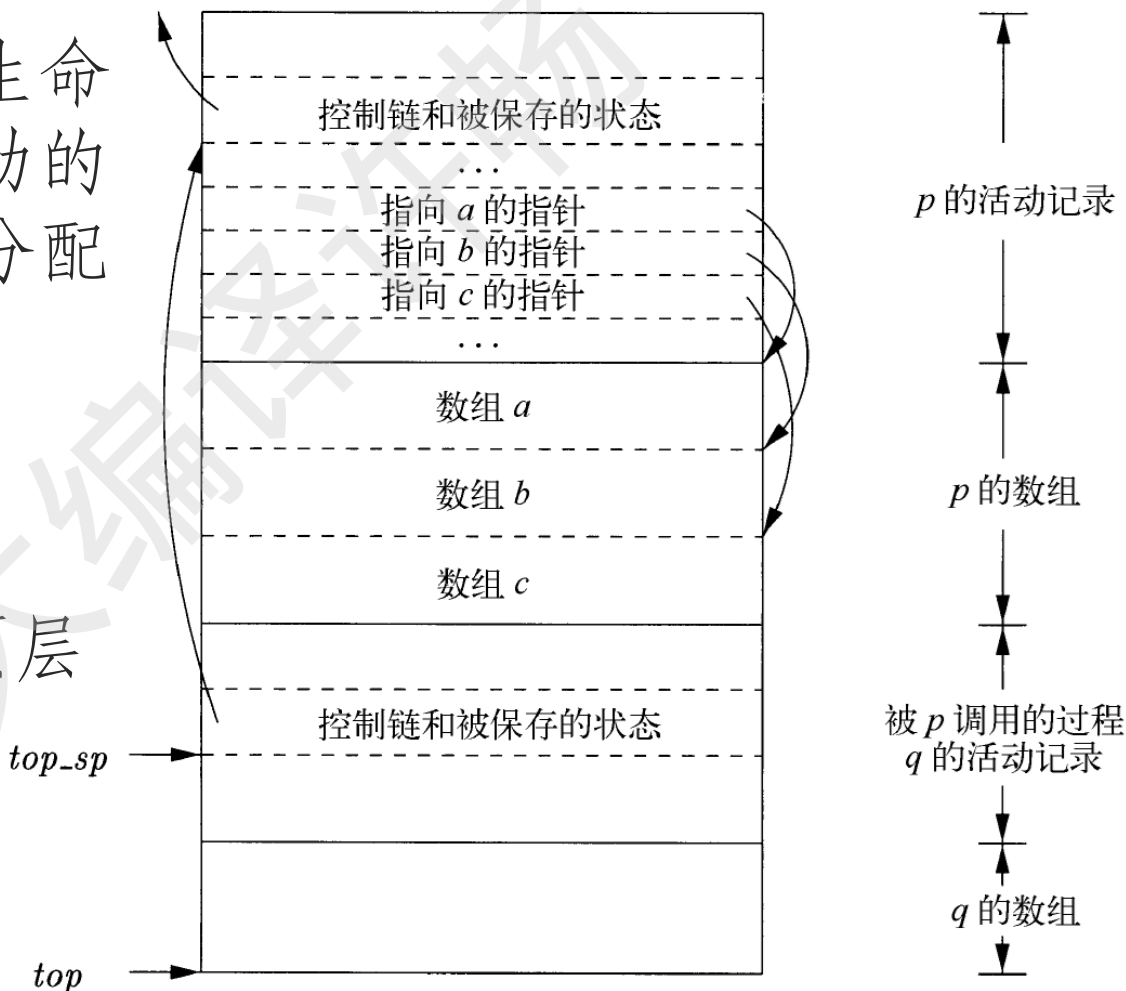


图 7-8 访问动态分配的数组

# 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：

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

# 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 $i+1$

深度1: sort

深度2: readArray, exchange, 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;
```

深度3: partition

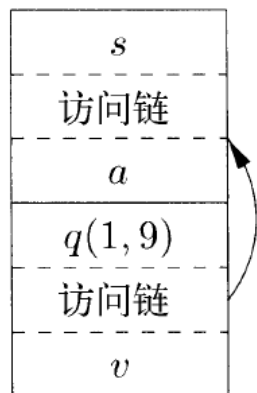
# 访问链和访问链的使用

- 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

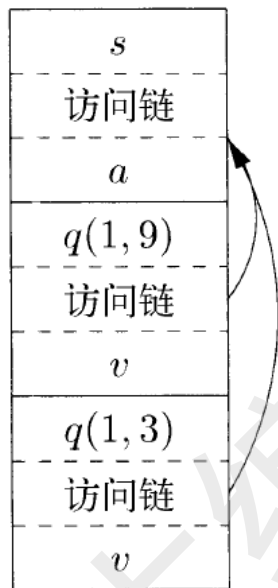
# 访问链的维护

- 当过程 $q$ 调用过程 $p$ 时
  - $p$ 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)
    - 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 $exchange$

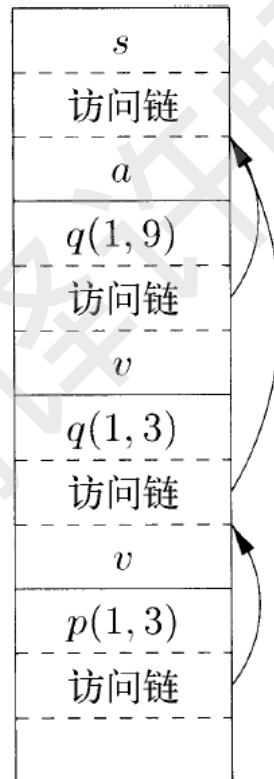
# 访问链的例子



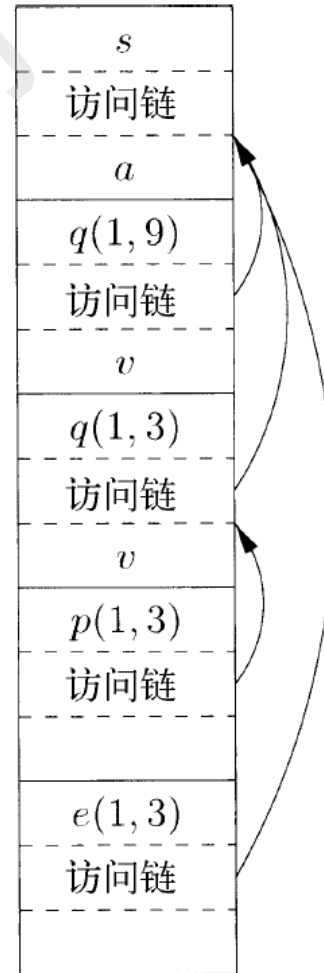
a)



b)



c)



d)

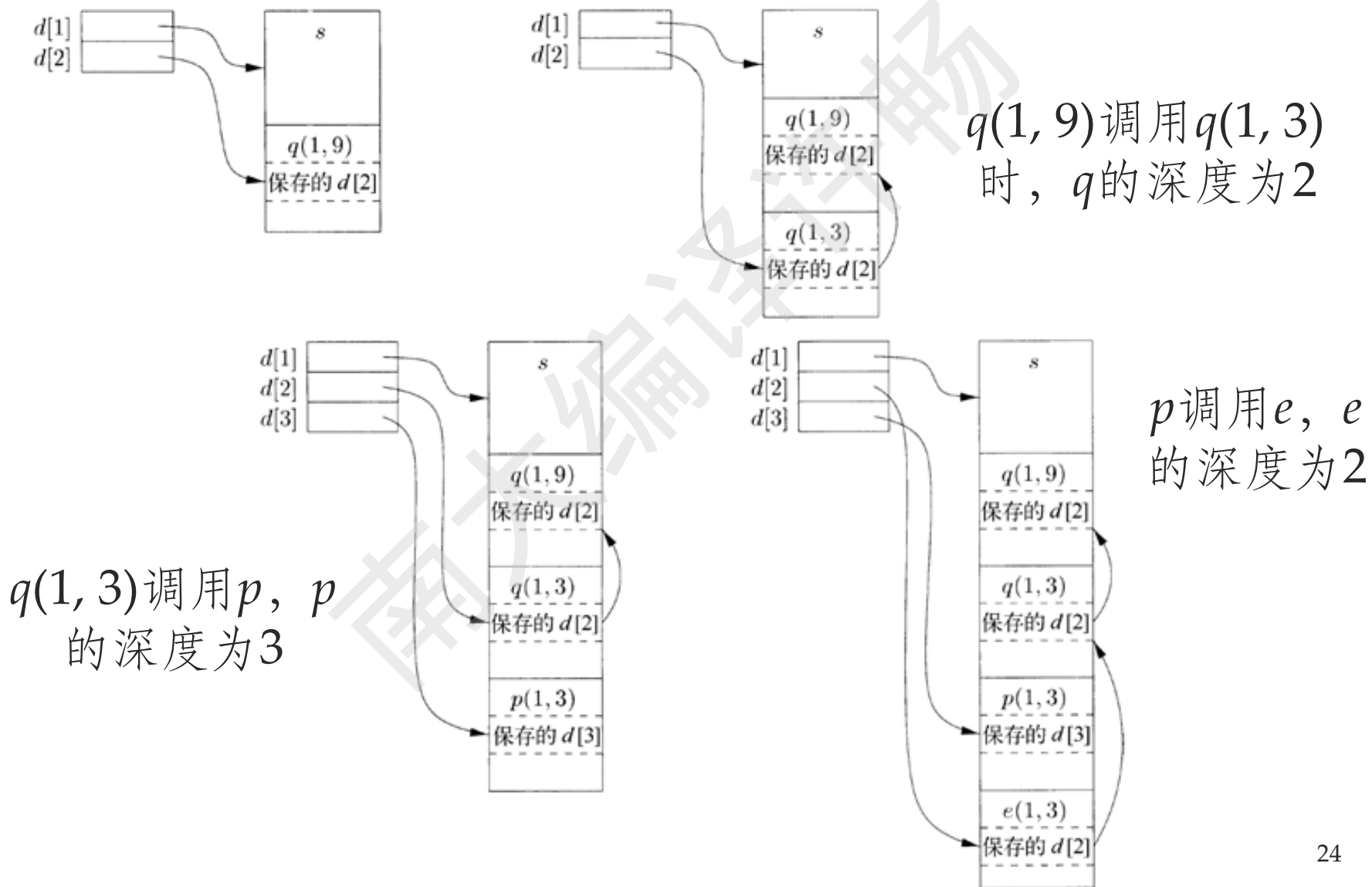
# 访问链的维护 (过程指针型参数)

- 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





# 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

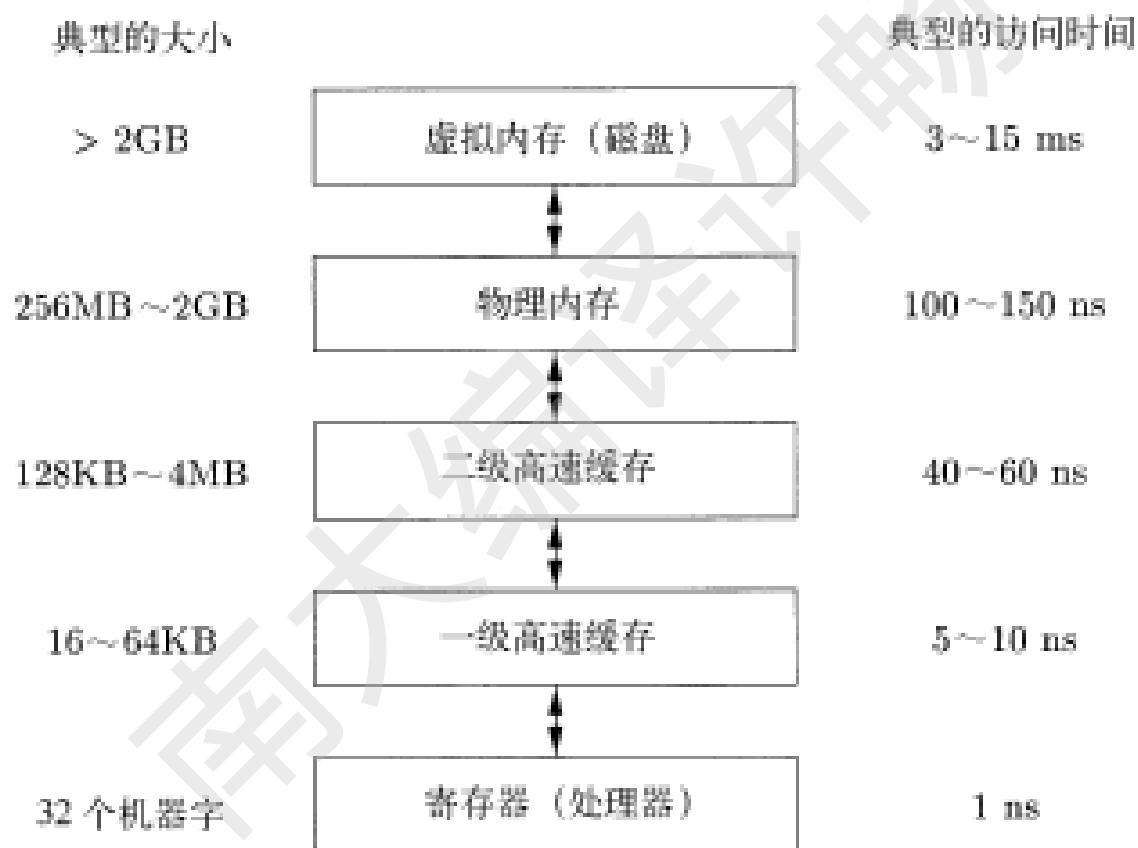


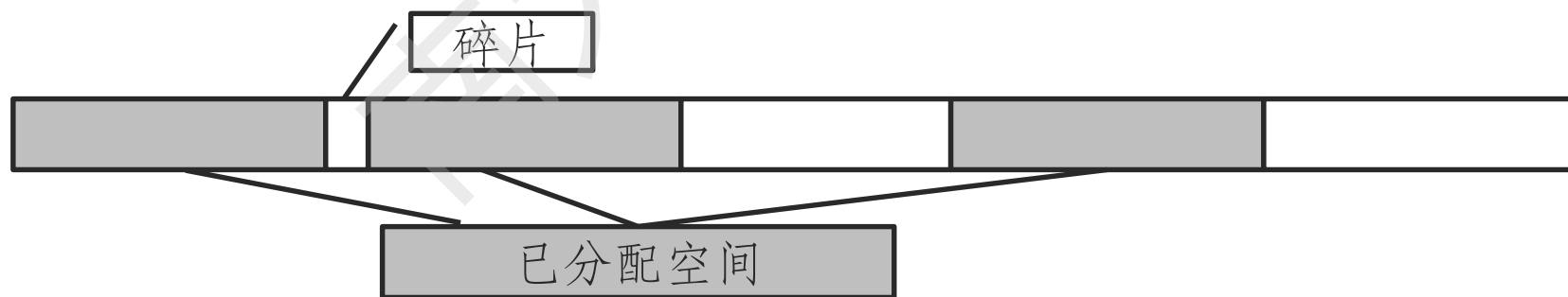
图 7-16 典型的内存层次结构的配置

# 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到
    - 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

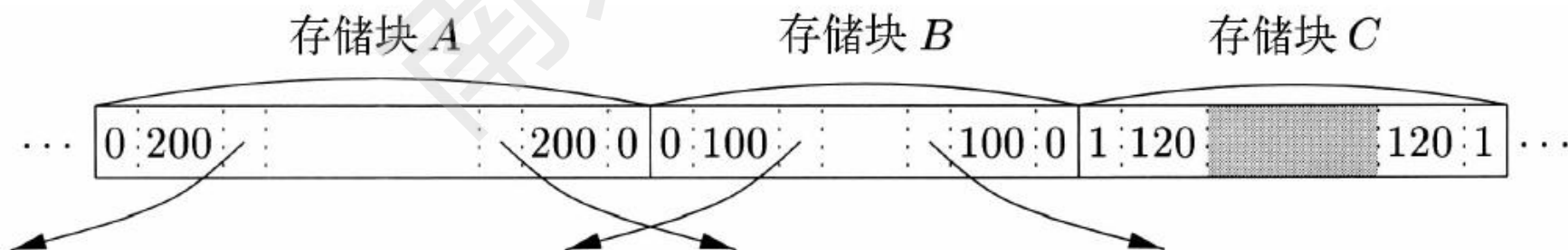
# 管理和接合空闲空间

- 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：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

- 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
- 参数传递/返回值
  - 对象引用从实参传递到形参，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， $u$ 中原有引用丢失；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

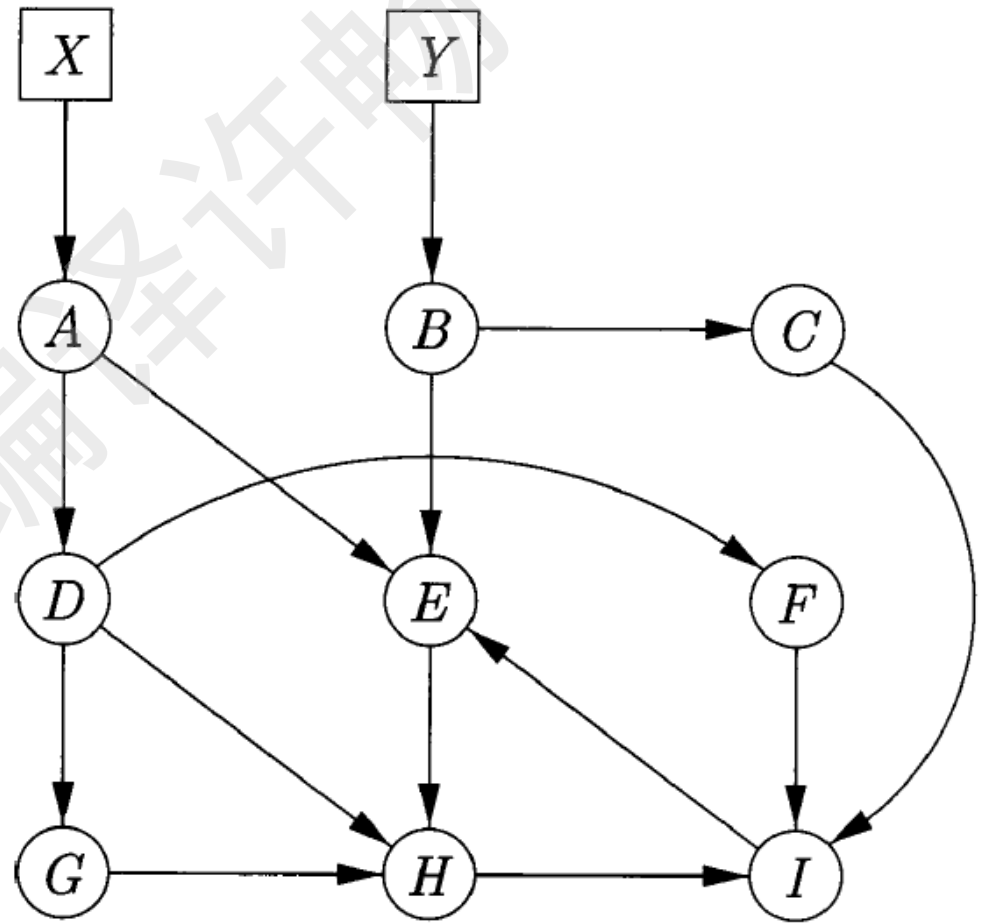
# 基于引用计数的垃圾回收器

- 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

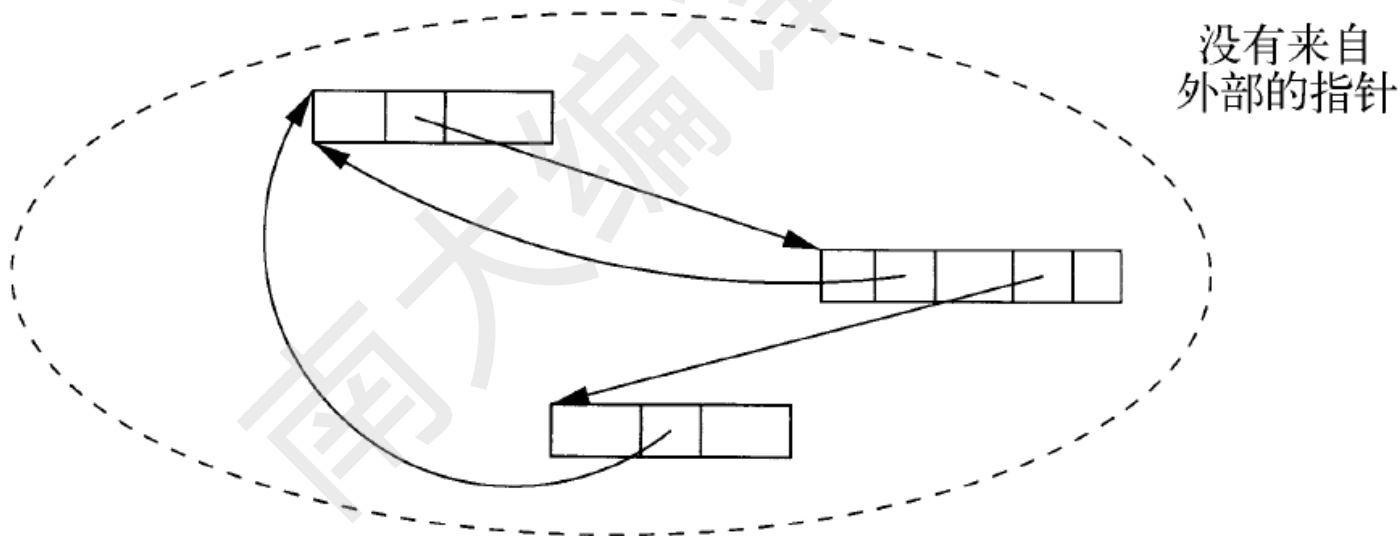


图 7-18 一个不可达的循环数据结构

# 基于跟踪的垃圾回收

- 标记-清扫式垃圾回收
- 标记并压缩垃圾回收
- 拷贝垃圾回收

# 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

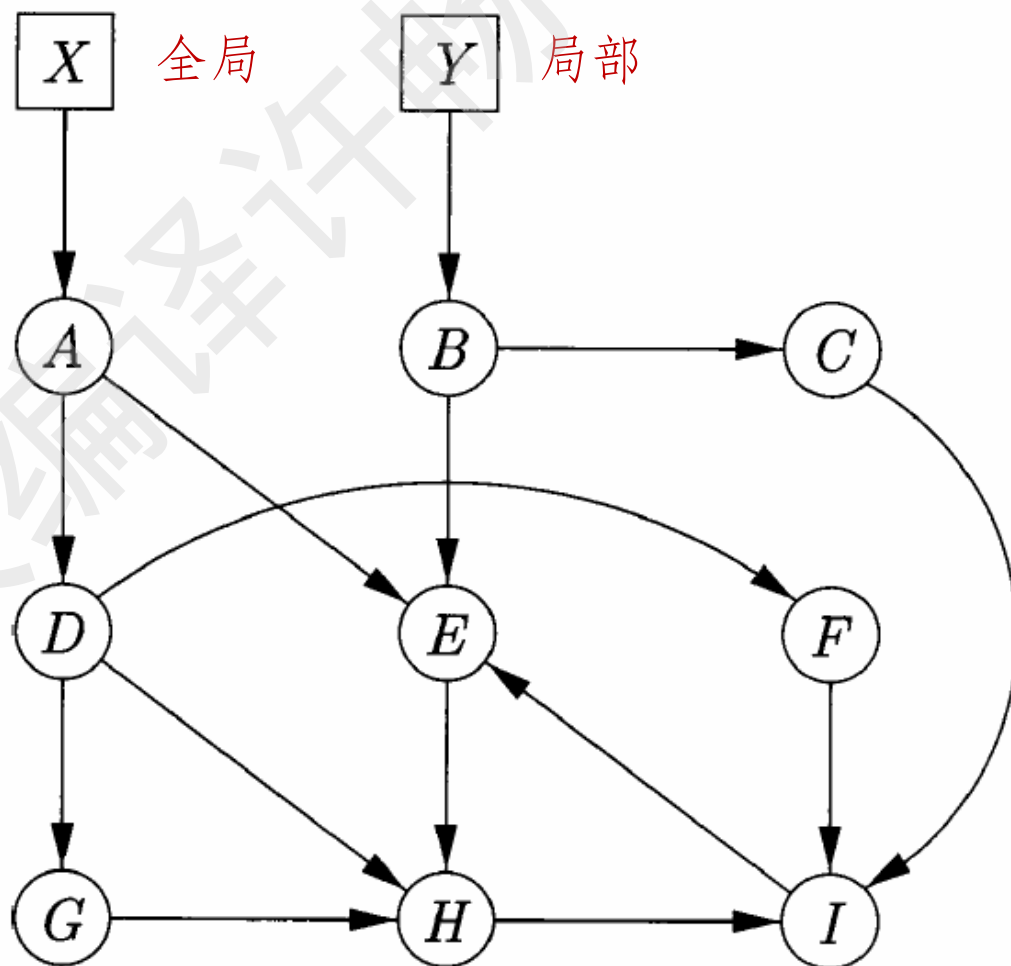
# 回收算法

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

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

# 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



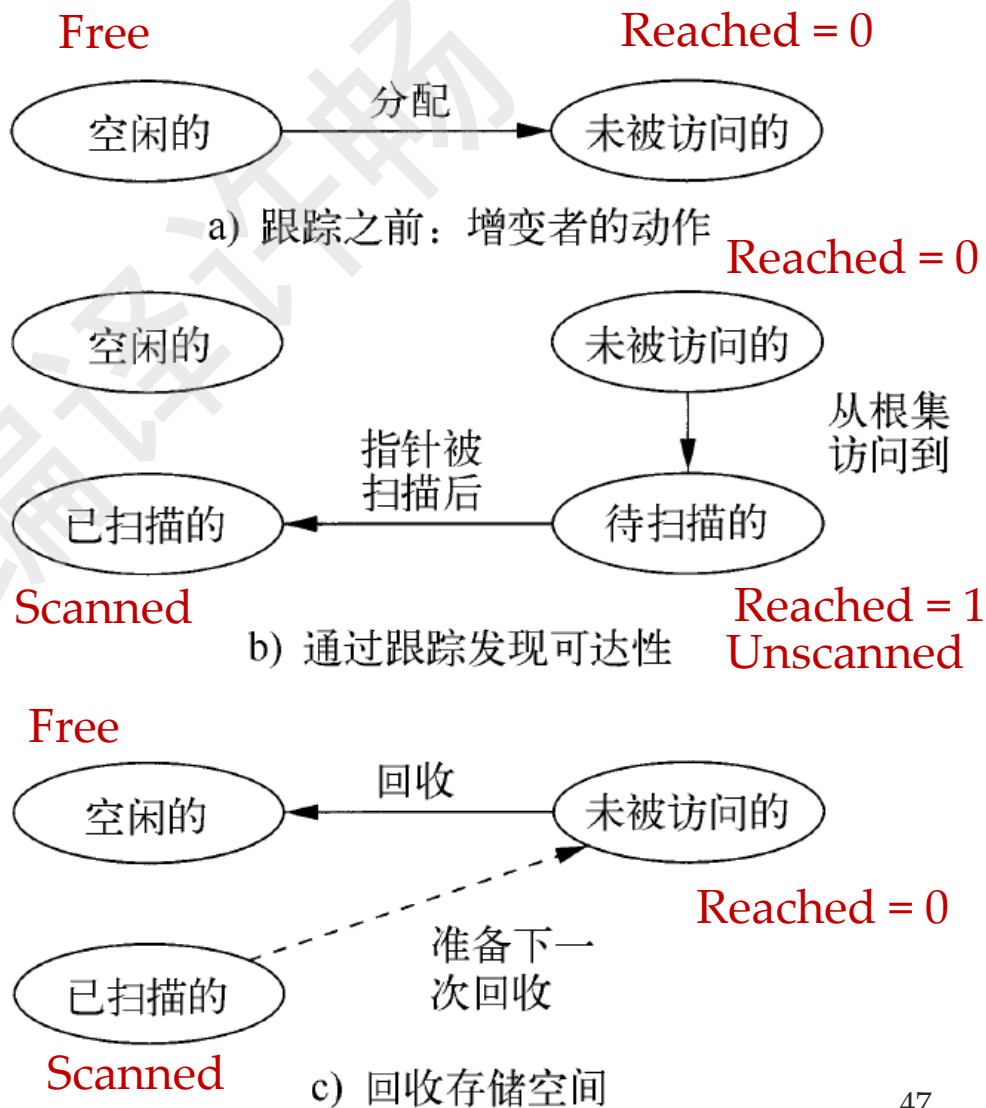
# 基本抽象分类

- 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
- 垃圾回收器扫描
- 回收



# 标记并压缩垃圾回收

- 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**，提高分配内存时的效率
- 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



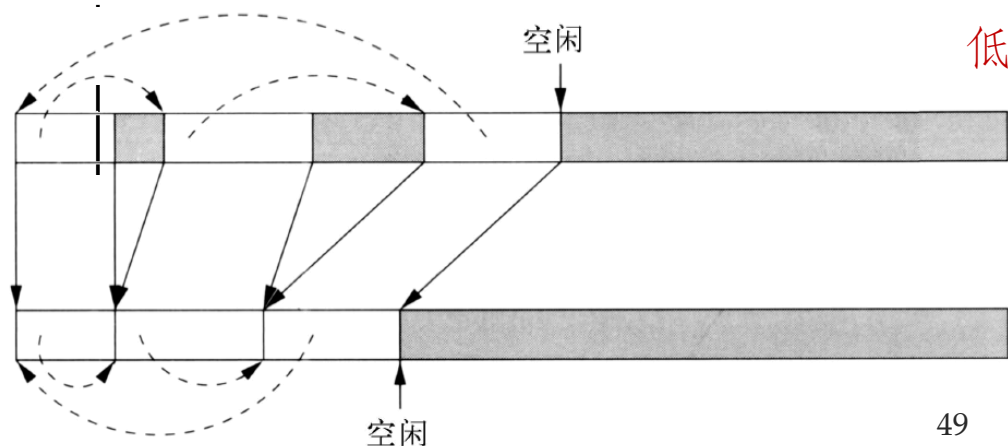
# 标记并压缩垃圾回收

```
/* 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);
```



# 拷贝垃圾回收

- 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
- 优点
  - 不涉及任何不可达对象