编译原理

第八章 运行时数据区的管理

方徽星

扬州大学 信息工程学院(505)

fanghuixing@yzu.edu.cn

2018年春季学期

运行时环境的功能

- 为数据分配安排存储位置
- 确定
  - 访问变量时使用的机制
  - 过程间的连接（符号重定位）
  - 参数传递机制
  - 操作系统、输入输出设备相关的其它接口

本章主要内容

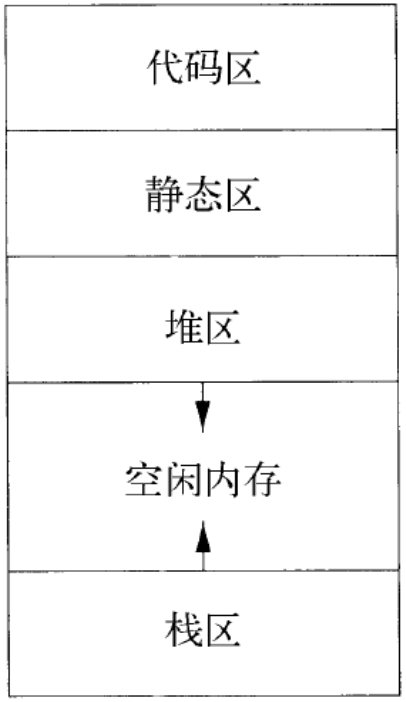
一. 存储组织

二. 空间的栈式分配

三. 栈中非局部数据的访问

1 存储组织

- 目标程序的代码放置在代码区，通常位于存储的低端
- 静态区、堆区、栈区分别放置不同类型生命期的数据值
- 实践中栈向较低地址方向增长堆向较高方向增长
- **约定栈向较高地址方向增长：方便使用正的偏移量**



1 存储组织

- **静态存储**分配
  - 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
  - 静态变量，全局变量
- **动态存储**分配
  - **栈**式存储：
    - 过程的局部名字采用栈式存储
    - 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：
    - 有些数据生命期比相应过程调用的生命期更长，常分配在一个可重用存储的“堆”中

1 存储组织

- 堆是虚拟内存的一个区域
  - 在被创建时获得存储空间
  - 并在变得无效时释放该存储空间
- 垃圾回收
  - 检测出无用的数据元素，释放它们
  - 手工进行回收
  - 垃圾回收机制

2 空间的栈式分配

- 过程调用(过程的活动)在时间上总是嵌套的：
  - 后调用先返回(LIFO: Last In First Out)
  - 可用**栈式分配**来处理过程活动所需要的内存空间
- 程序运行过程可以用树表示
  - **结点对应过程的活动**
  - **根结点**对应**主过程**的活动
  - 过程p的某次活动对应的结点的**子结点**对应**被p**的这次活动**调用的**各个过程活动

2 空间的栈式分配

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns 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);
}
```

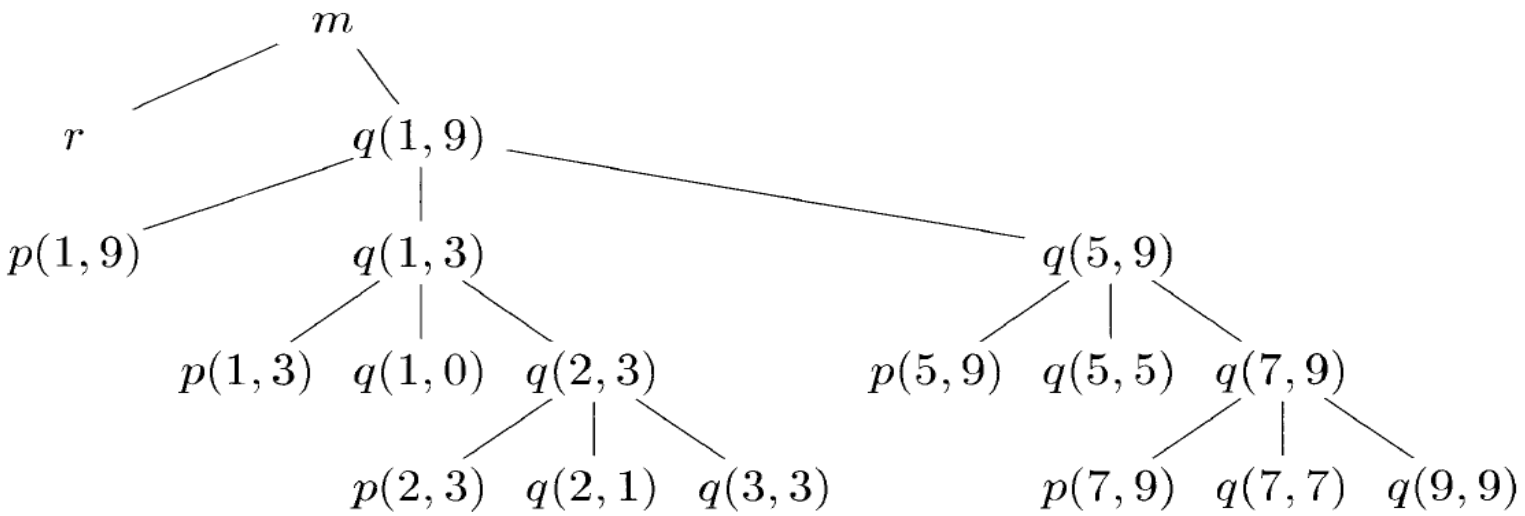
快速排序程序概要

快排程序可能的活动序列

```
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()
```

2 空间的栈式分配

- 如果当前活动对应结点N，那么所有尚未结束的活动对应结点N及其祖先结点

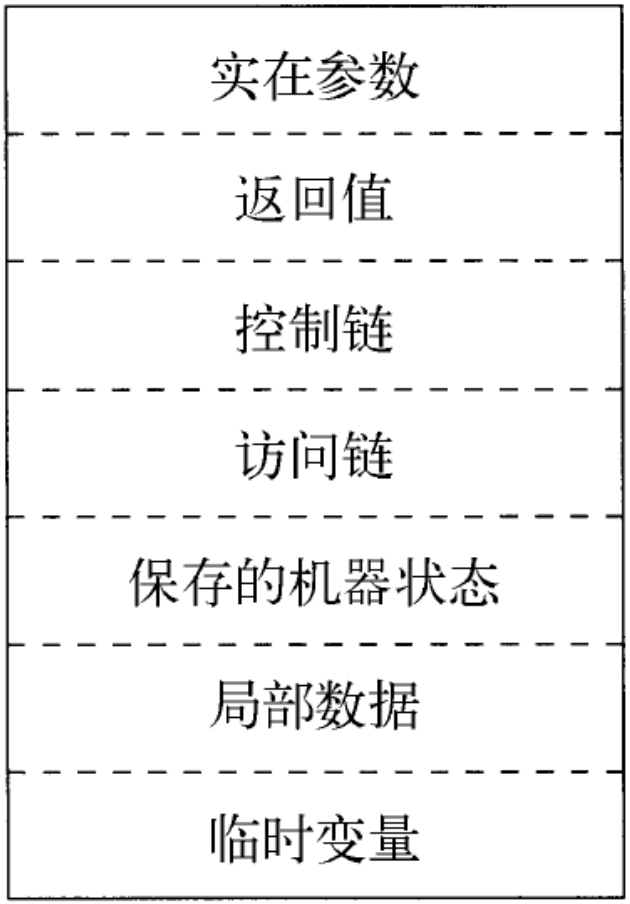


程序的活动树

2 空间的栈式分配

- 过程调用和返回由称为**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该栈中的**活动记录**(Activation Record，也称为帧Frame)
- 活动树的**根在栈底**
- **栈中**全部活动记录的**序列**对应活动树中到达当前控制所在的活动**结点**的**路径**

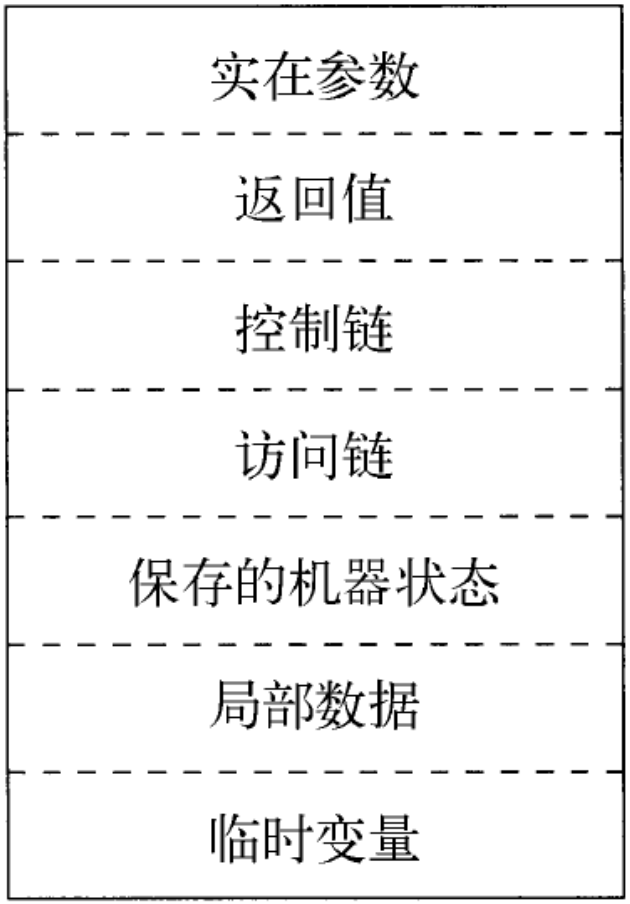
2 空间的栈式分配



活动记录

- 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，生成**临时值**
- 对应这个活动记录的**过程的局部数据**
- 保存的**机器状态**：此次调用前的机器状态信息，如：返回地址及一些寄存器的值

2 空间的栈式分配



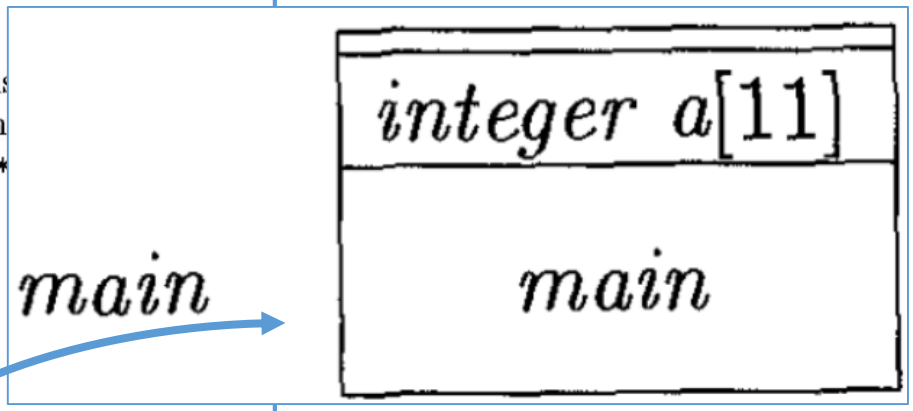
活动记录

- **控制链**(Control Link)：指向调用者的活动记录
- **访问链**：用于访问非局部数据
- **返回值**一般存在寄存器中
- **实参**尽可能放在寄存器中

2 空间的栈式分配

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions
       a[m..p-1] are less than v, a[p] = v, an
       equal to or greater than v. Returns 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);
}
```

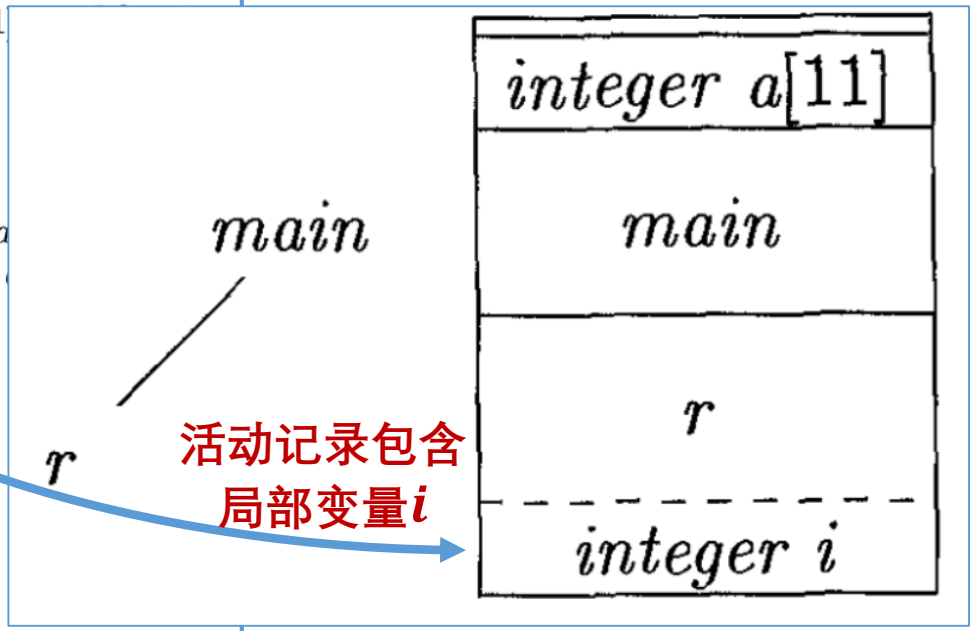
数组a是全局的



程序的执行随着过程main的一次活动开始

2 空间的栈式分配

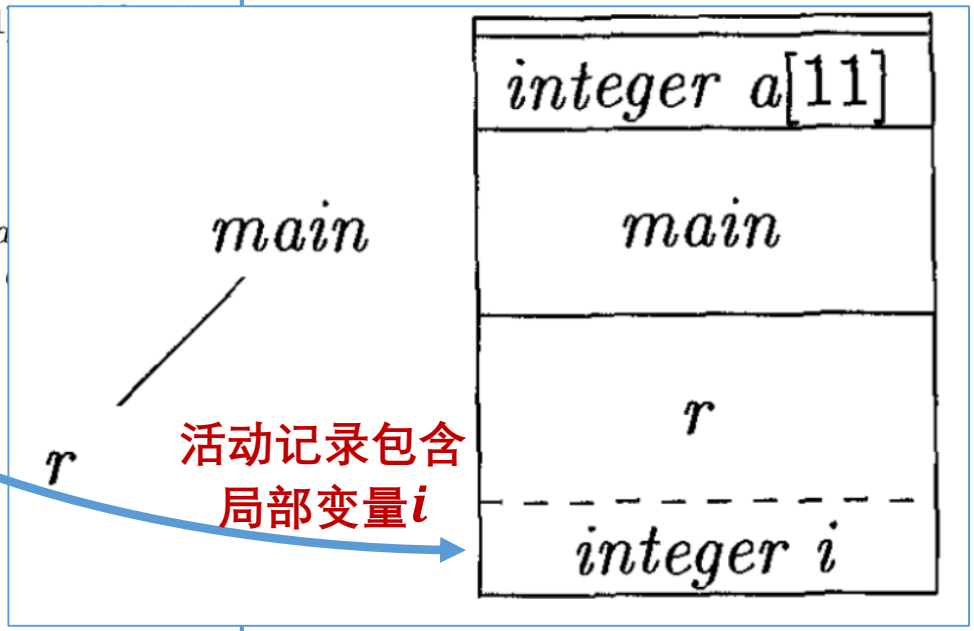
```
int a[11];
void readArray() { /* Reads 9 integers into a[1
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a
       a[m..p-1] are less than v, a[p] = v, and
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    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);
}
```



当控制到达main函数体的第一个函数调用时，过程r被激活

2 空间的栈式分配

```
int a[11];
void readArray() { /* Reads 9 integers into a[1
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a
       a[m..p-1] are less than v, a[p] = v, and
       equal to or greater than v. Returns 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);
}
```



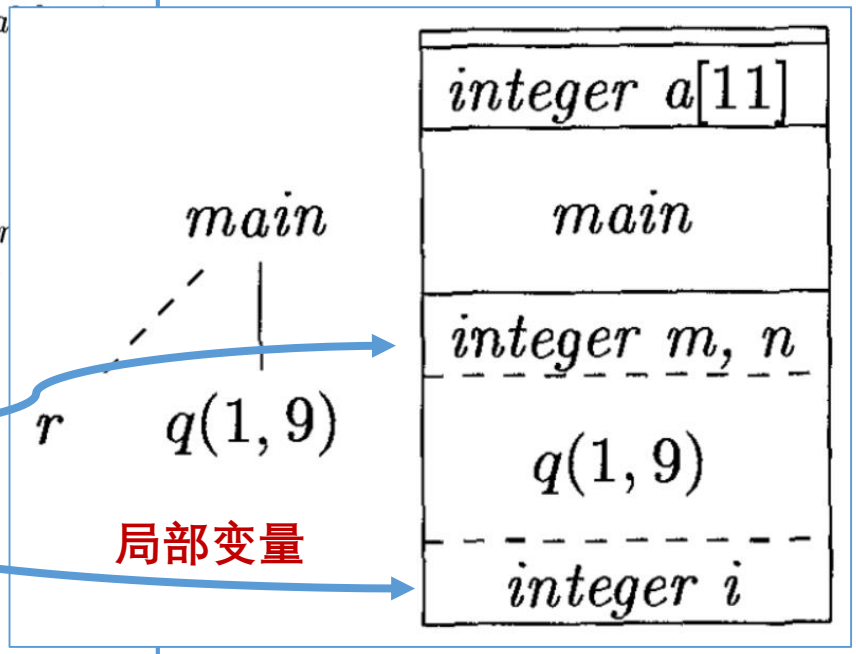
从过程r的活动中返回时，活动记录出栈，栈中只剩下main的记录

2 空间的栈式分配

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..r
       a[m..p-1] are less than v, a[p] = v, and a[p+
       equal to or greater than v. Returns 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);
}
```

参数

局部变量

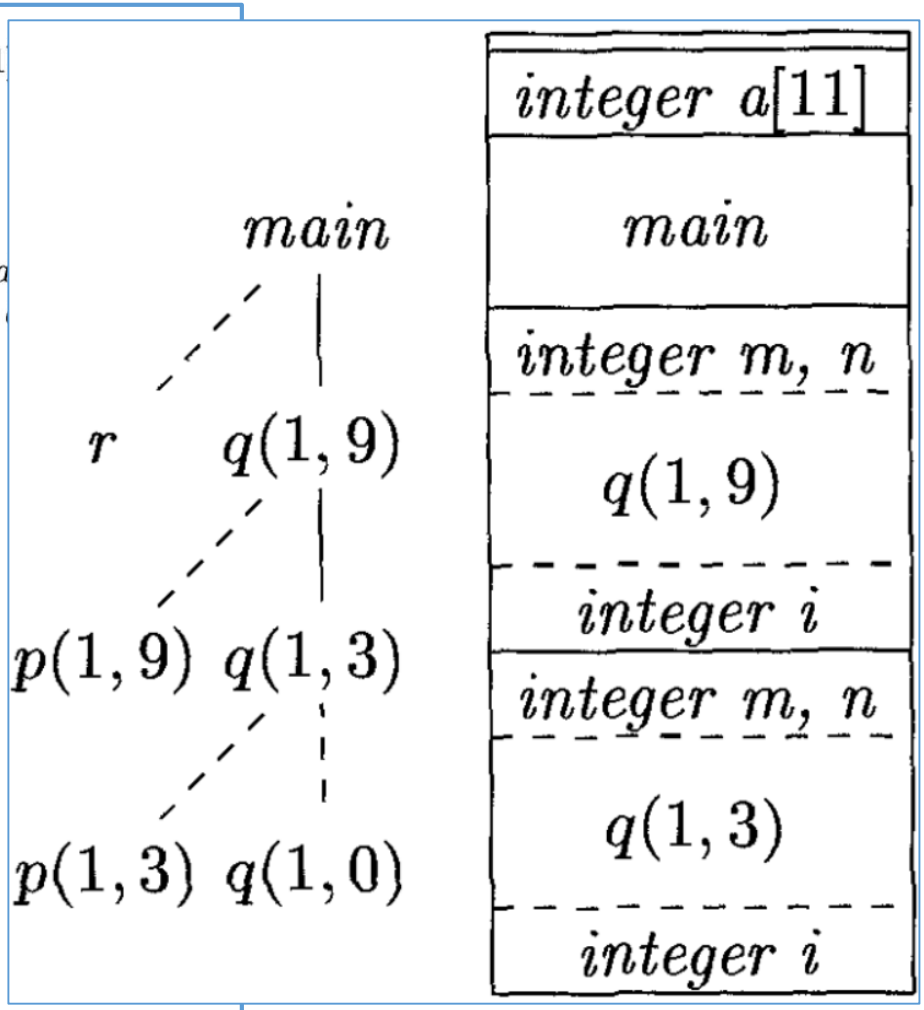


控制到达参数为1和9的对快速排序过程的调用

当此次活动返回时，栈中将再次只有main的活动记录

2 空间的栈式分配

```
int a[11];
void readArray() { /* Reads 9 integers into a[1
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a
       a[m..p-1] are less than v, a[p] = v, and
       equal to or greater than v. Returns 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);
}
```



2 空间的栈式分配

- 实现过程调用的代码段称为**调用代码序列**(Calling Sequence):
  - 为活动记录在栈中分配空间，并填写记录中的信息
- **返回代码序列**(Return Sequence)
  - 恢复机器状态，使调用者能够在调用结束后继续运行。
- 调用序列中的代码会分配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，可以有不同的分配方案
  - 把代码尽可能放在被调用者中

2 空间的栈式分配

- 设计调用代码序列和活动记录布局的原则：
  - 调用者和被调用者之间传递的值放在被调用者记录的开始位置
    - 不用创建整个被调用者的活动记录
    - 不用知道被调用者的活动记录布局
    - 可以使用参数个数或类型可变的过程
  - 固定长度的项放在中间位置
    - 包括控制链、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（干脆将固定长度的局部变量也放入该段）
  - 栈顶指针通常指向固定长度字段的末端

2 空间的栈式分配

- 调用序列例子
  - 调用者计算实在参数的值
  - 将返回地址和原来的top_sp值（控制链）存放到被调用者的活动记录中
  - 然后增加top_sp的值：越过调用者的局部数据、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

2 空间的栈式分配

- 返回序列例子
  - 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
  - 调用者知道返回值相对于当前top_sp值的位置，调用者可以得到返回值

2 空间的栈式分配

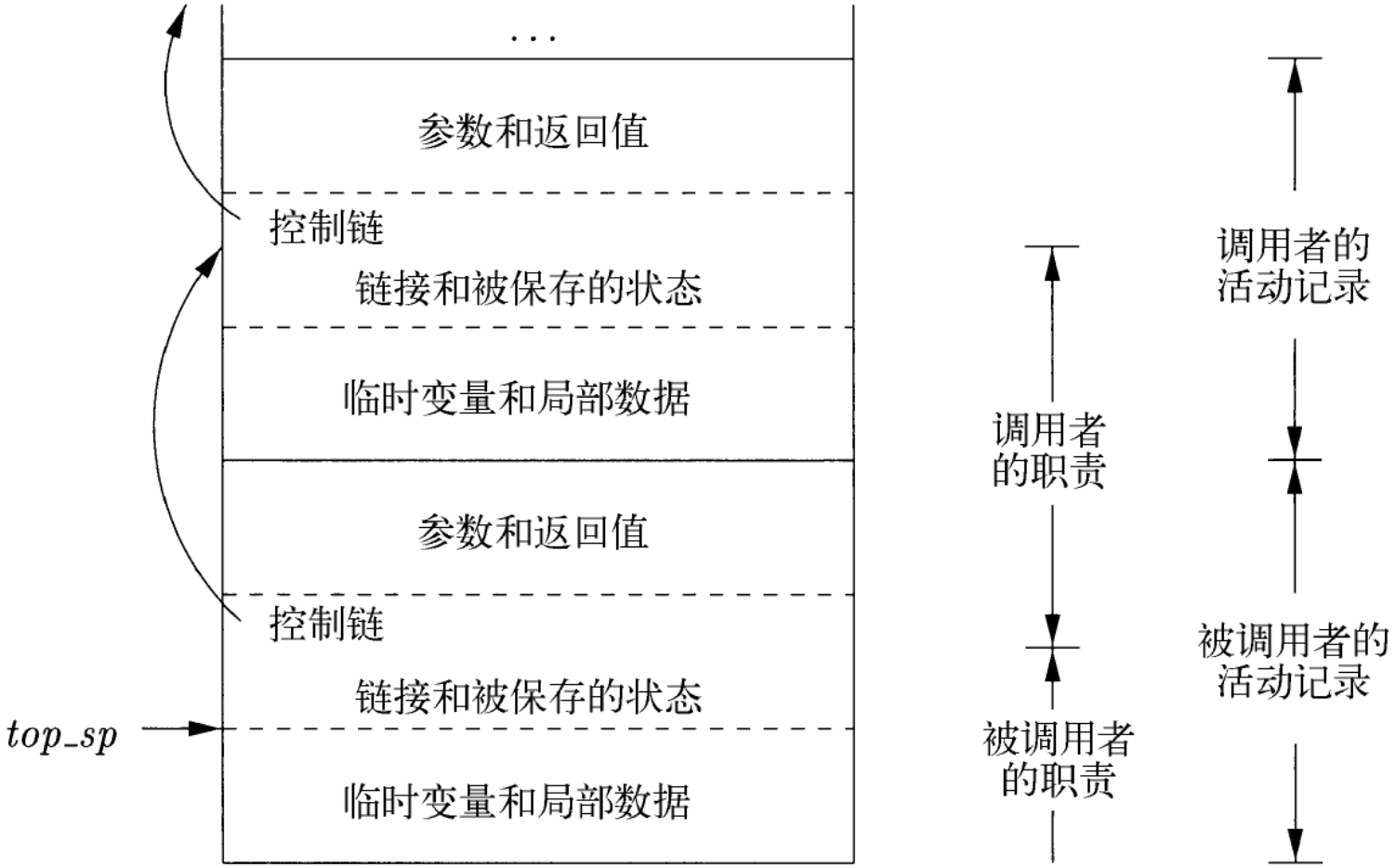


图 7-7　调用者和被调用者之间的任务划分

2 空间的栈式分配

- 当编译时未知大小的数据对象
- 生命期局限于过程活动的生命期时
- 可分配在栈中
- **指向数组的指针放在活动记录中**
- **重新设置top和top_sp的代码可在编译时生成**

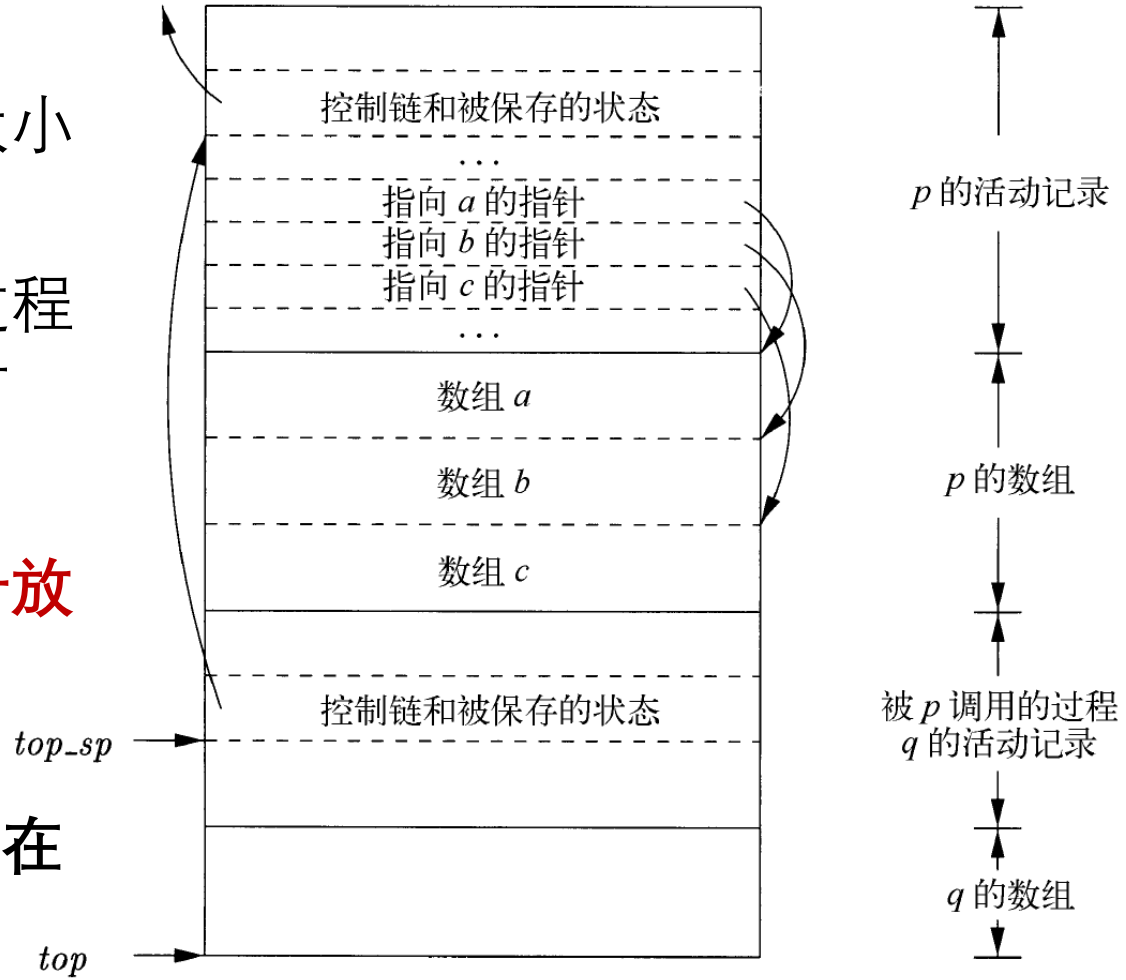


图 7-8　访问动态分配的数组

3 栈中非局部数据的访问

- 先考虑无嵌套过程时的数据访问
  - 例：C语言
    - 不允许嵌套过程声明，变量要么在函数内定义，要么全局地定义
  - C语言中函数对变量的访问
    - 局部变量：在当前活动记录内，可通过top_sp指针加上相对地址来访问
    - 全局/静态变量：在静态区，地址在编译时可知
  - C语言允许函数参数
    - 参数中只需要包括函数代码的起始地址
    - 在函数中访问变量的模式很简单，不需要考虑过程是如何激活的

C语言中活动记录中无需访问链

3 栈中非局部数据的访问

- 在基于栈的环境中，要访问参数和局部变量，可用当前框架指针(top_sp)的偏移量实现
- 在大多数的语言中，每个局部声明的偏移量可由编译程序静态地计算出来
- 因为过程的声明在编译时是固定的，而且为每个声明分配的存储器大小也根据其数据类型而固定

3 栈中非局部数据的访问

- 支持嵌套过程的典型语言**函数式语言ML**
  - 变量一旦声明并初始化就不会再改变
  - 有少数例外，如：数组元素可通过特殊的函数调用改变
  - 变量定义并初始化为不可更改的值的语句形式：
    - **val** <name> = <expression>
  - 函数定义的语法：
    - **fun** <name> (<arguments>) = <body>
  - 函数体定义的语法：
    - **let** <list of definitions>
      in <statements>
      end

- 定义(definition)通常是**val**和**fun**语句
- 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

3 栈中非局部数据的访问

- **嵌套深度**可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，所有C函数的嵌套深度为1
  - 定义在嵌套深度为i的过程中的过程，深度为$i+1$

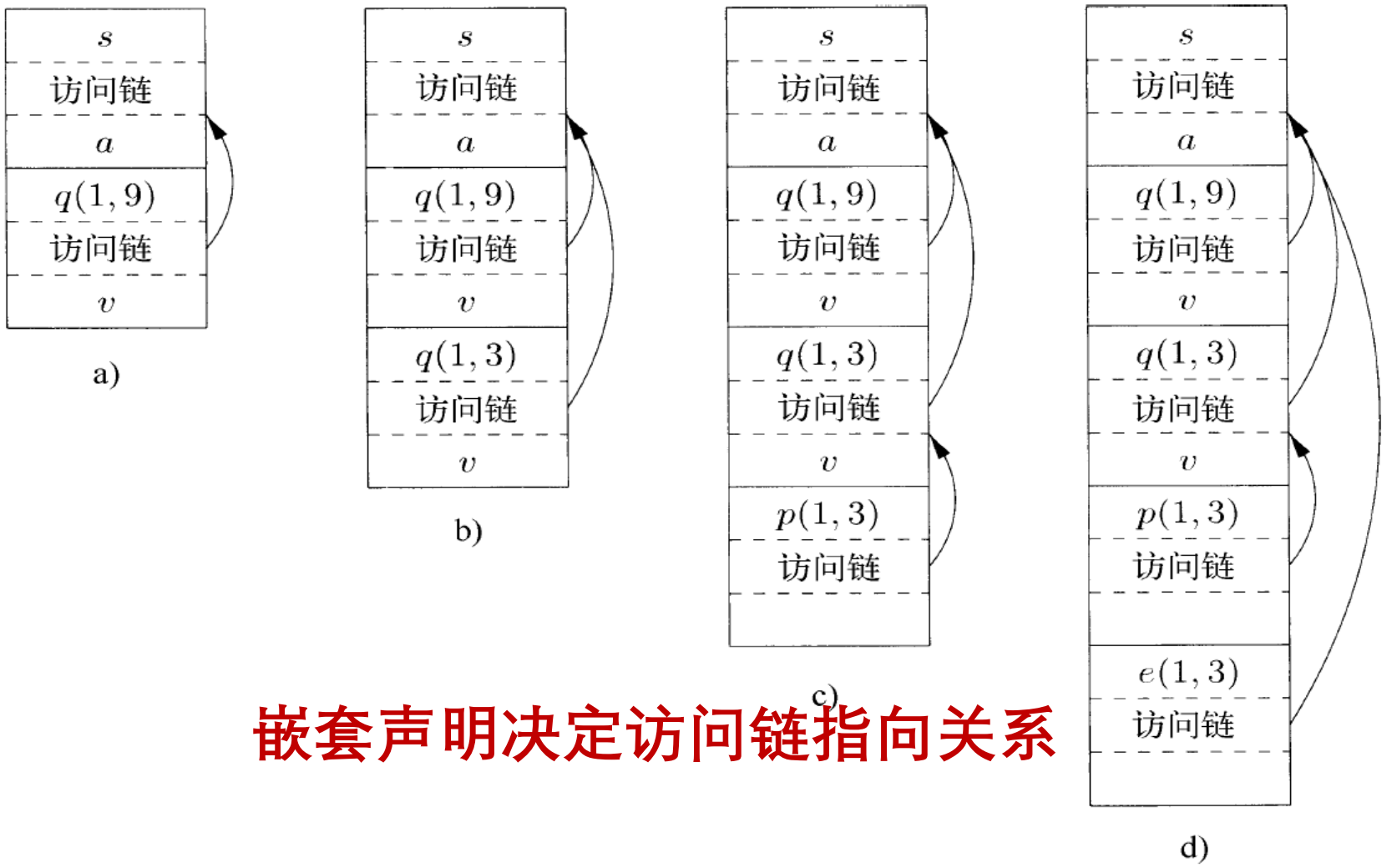
嵌套深度	函数
1	L-1: sort
2	L-3: readArray L-5: exchange L-7: quicksort
3	L-9: partition

```
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;
```

3 栈中非局部数据的访问

- 如果过程p在声明时嵌套在过程q的声明中，那么p的任何活动中的访问链都指向最新的过程q的活动记录
- 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，则
  - 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相对于该活动记录的偏移量在编译时刻已知

3 栈中非局部数据的访问



嵌套声明决定访问链指向关系

3 栈中非局部数据的访问

- 过程q显式地调用过程p时，**如何确定访问链：**
  - **如果p的声明嵌套深度大于q：**p必然在q中直接定义，否则不满足ML语言的作用域规则
  - p的嵌套深度恰好比q的嵌套深度大1
  - 在调用代码序列里面增加一个步骤：在p访问链中放置一个指向q的活动记录的指针

3 栈中非局部数据的访问

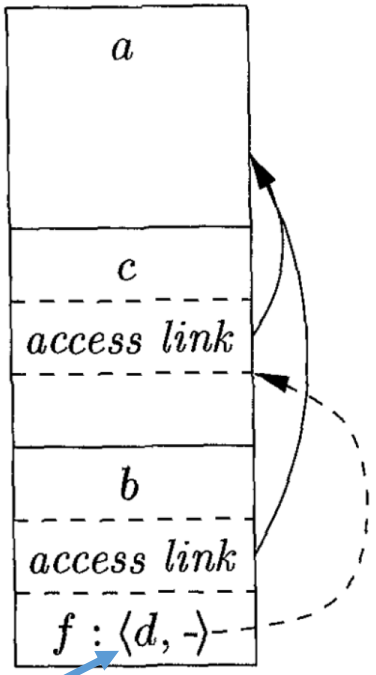
- 过程q显式地调用过程p时，**如何确定访问链：**
  - **如果是递归调用：**p=q，p的活动记录的访问链等于q当前记录的访问链
  - **如果p的声明嵌套深度小于q的深度：**此时必然有过程r，p直接在r中定义，而q嵌套在r中，此时应让p的访问链指向r的活动记录

3 栈中非局部数据的访问

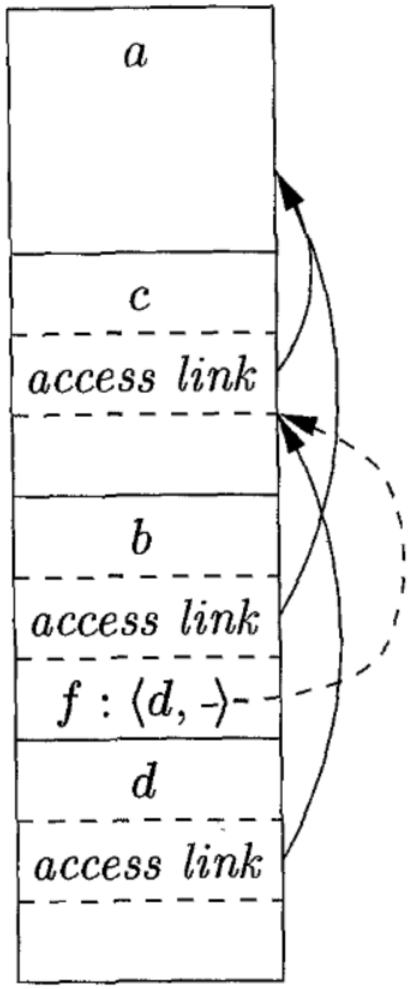
- **允许过程作为参数时**
  - 当一个过程p作为参数传递给另一个过程q，并且q随后调用了这个参数，**有可能q并不知道**p在程序中出现的上下文
  - 当过程被用作参数的时候，调用者除了传递过程参数名字，同时还需**传递**这个参数对应的**正确的访问链**
  - 调用者总是知道这个访问链的，可以**像直接调用p**那样为p**确定访问链**

3 栈中非局部数据的访问

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



(a)



(b)

实在参数d和它的访问链

3 栈中非局部数据的访问

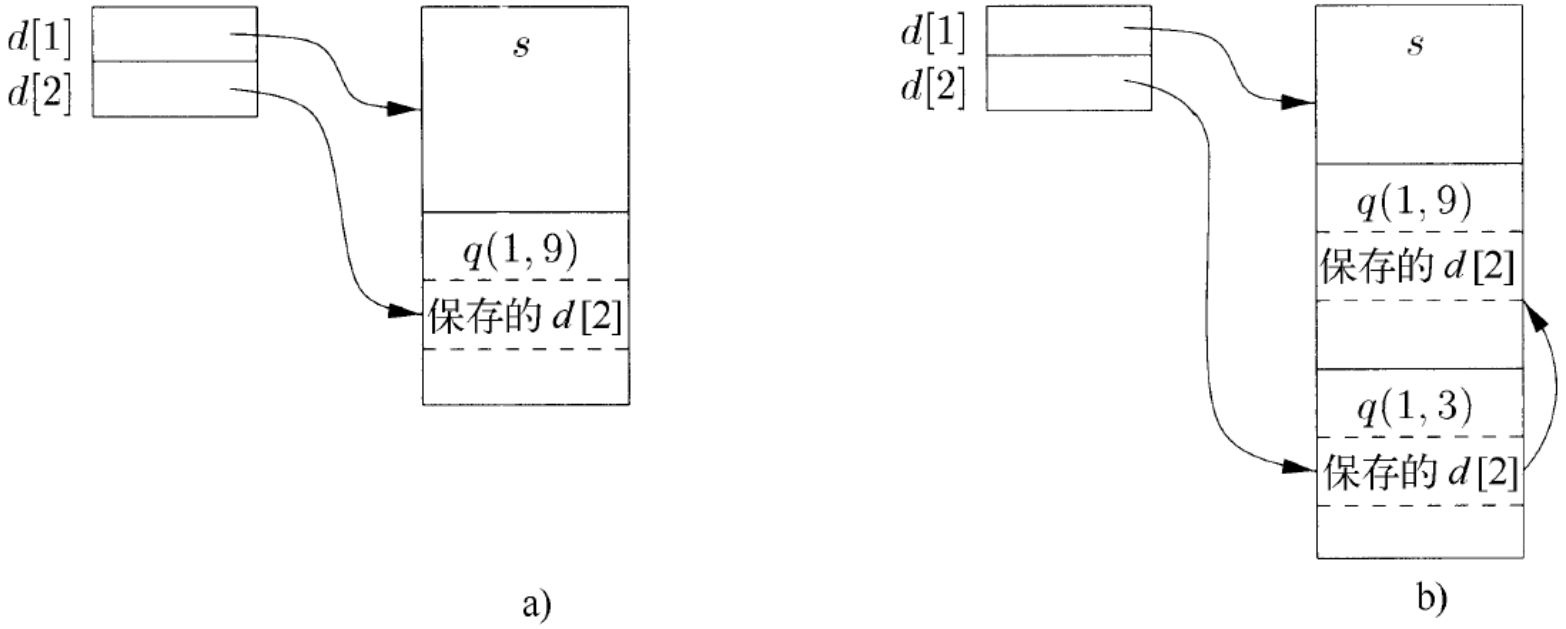
- 用访问链访问数据时，如果嵌套深度大，则访问的效率差
- **显示表：**使用数组d为每个嵌套深度保留一个指针
  - 指针d[i]指向栈中最高的对应于嵌套深度为i的活动记录
  - 如果程序p中访问嵌套深度为i的过程q中变量x，那么d[i]直接指向相应的q活动记录

3 栈中非局部数据的访问

- 显示表的维护
  - 调用过程p时，在p的活动记录中保留**原**$d[n_p]$的值，并将$d[n_p]$的新值设置为p的本次活动记录
  - 当从p返回时，恢复$d[n_p]$的值

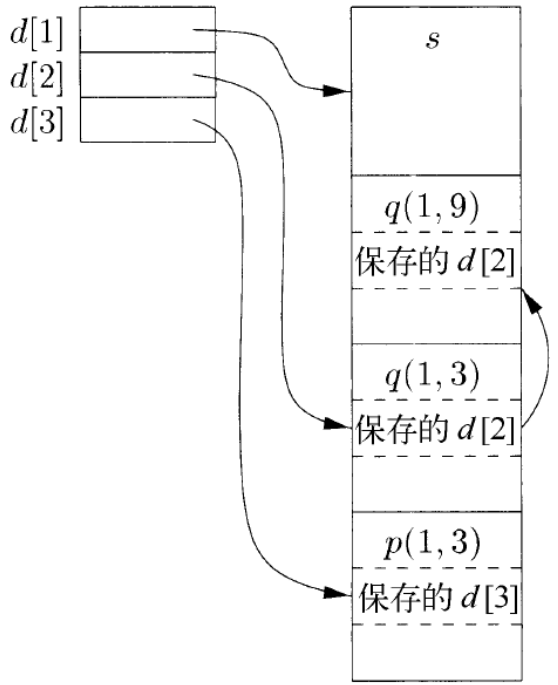
3 栈中非局部数据的访问

快速排序的例子

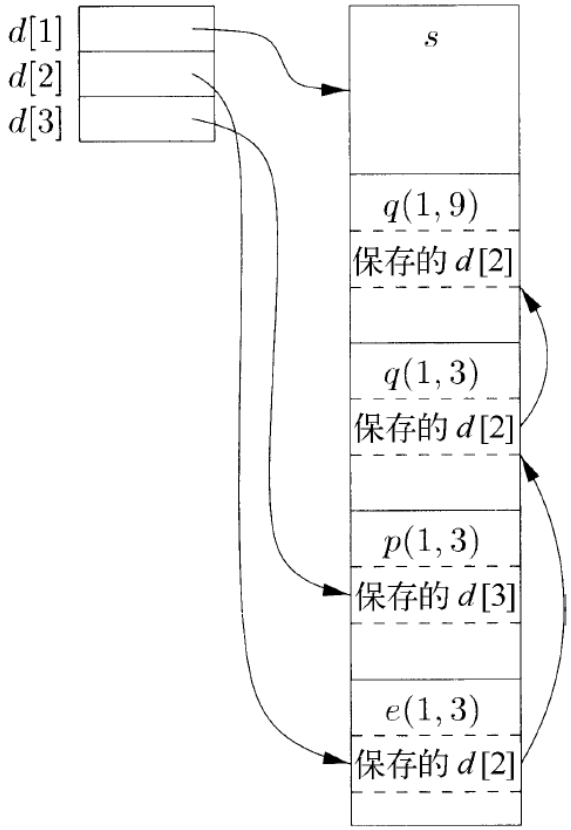


3 栈中非局部数据的访问

快速排序的例子



c)



d)

本章小结

- 存储组织：静态数据区、动态的栈区、堆区
- 控制栈：过程调用、返回
- 局部变量放在栈中存储，控制栈中使用活动记录保存过程调用信息
- 嵌套过程，通过访问链找到变量声明过程对应的活动记录，使用显示表可高效确定目标活动记录