第七章 运行时刻环境

运行时刻环境

运行时刻环境

- 为源程序中命名的对象分配安排存储位置
- 确定目标程序访问变量时使用的机制
- 过程之间的连接
- 参数传递

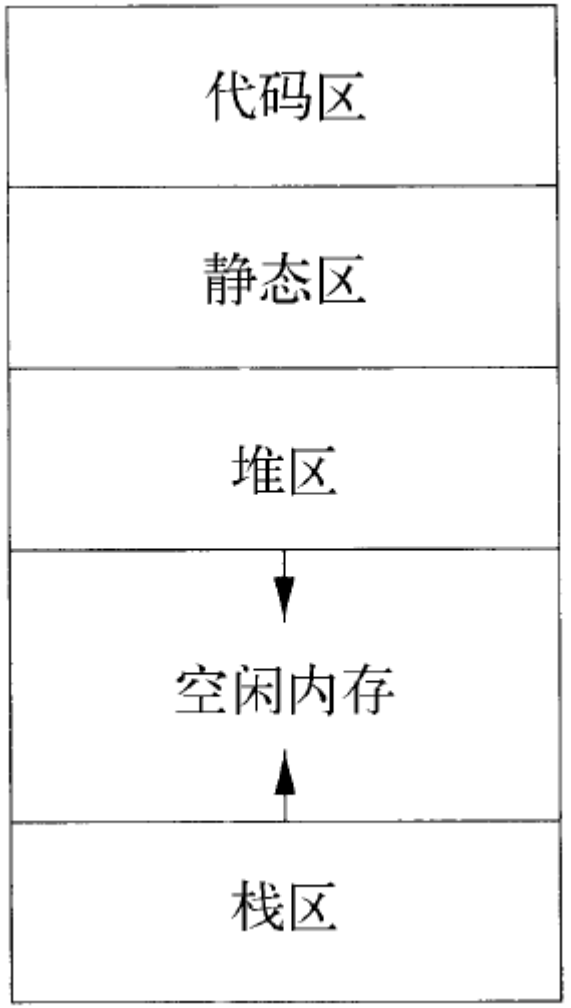
主题

- 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
  - 垃圾回收机制

静态存储分配

静态分配给语言带来限制

- 递归过程不被允许
- 数据对象的长度和它在内存中位置的限制，必须是在编译时可以知道的
- 数据结构不能动态建立

静态存储分配

C语言程序的外部变量和程序中出现的常量都可以静态分配

声明在函数外面

- 外部变量 —— 静态分配
- 静态外部变量 —— 静态分配

声明在函数里面

- 静态局部变量 —— 静态分配
- 自动变量 —— 不能静态分配

过程

过程是一组动作或者计算的抽象

过程体作为一个整体由其名字表示，通过提供一个说明以及一个实体来定义

一个过程通过它的名字和实参被调用

- intswap(a,b)

```
void intswap(int x,  int y);   //说明
void intswap(int x, int y）   // 定义
{
        int t;
        t=x;  x=y; y=t;
}
```

过程调用

调用过程时，控制转移到被调过程的过程体开端

- 当执行到达过程体的末端时，控制返回给主调过程
  - 有些语言可以使用return语句提前返回
- 控制进入过程时，由主调过程的活跃状态转移到被调过程的活跃状态。
  - 从被调过程返回时，控制返回主调过程，转移到主调过程的活跃状态

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

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

过程体的一次执行称为该过程的一个活动

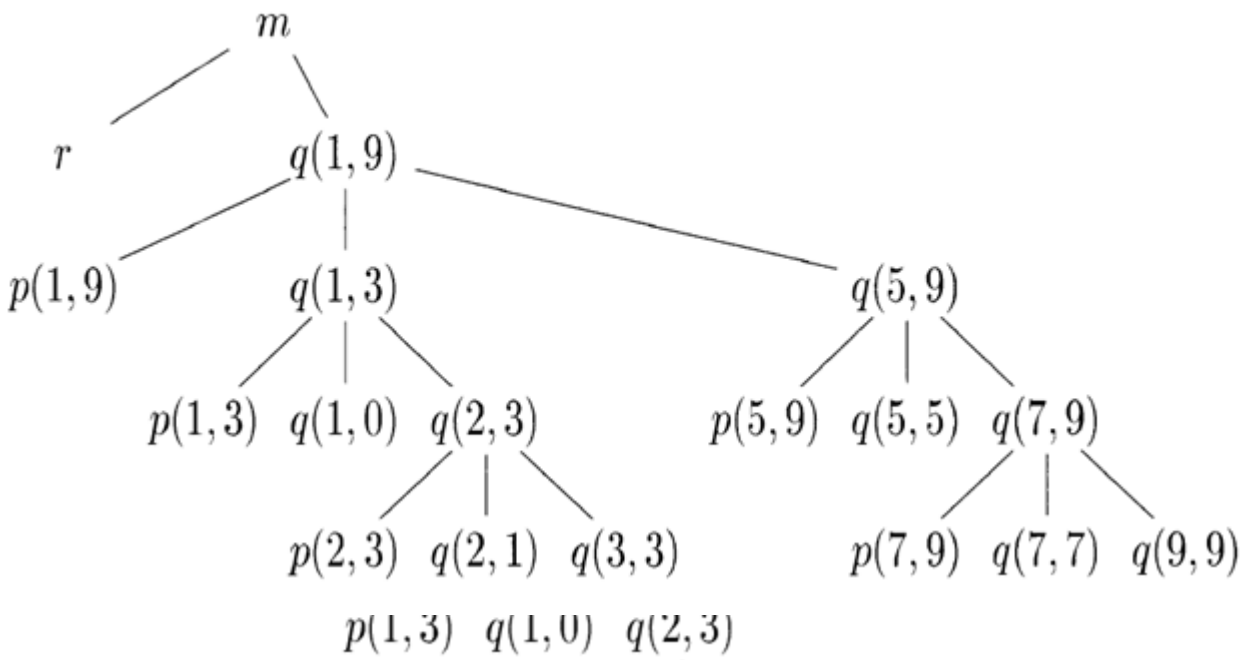
活动树：用来描述程序运行期间控制进入和离开各个活动的情况的树

- 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
- 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。按照这些活动被调用的顺序，自左向右地显示它们。一个子结点必须在其右兄弟结点的活动开始之前结束

一个快排程序的概要

```
int a[11];
void readArray(){int i;}
int partition(int m,int n){ }
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);
}
```

活动树的例子




```
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()
```

活动记录

过程调用引发局部变量的内存分配

活动记录：分配给过程体内局部对象的存储空间，它在控制返回主调过程时释放

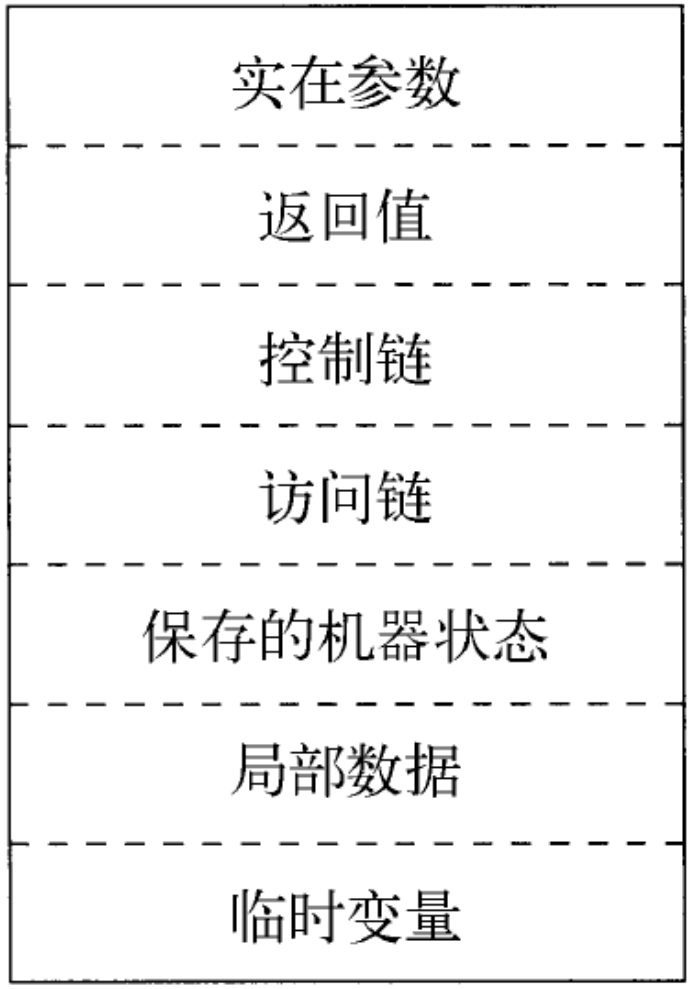
过程调用和返回由控制栈进行管理

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

活动记录

- 访问链:当前过程需要位于其它地方（如另一个活动记录）的某个数据时，通过访问链来定位
- 控制链:源自被调过程的活动记录，指向主调过程的活动记录
- 机器状态信息:过程被调用时的机器状态（处理器的现场信息）
  - 返回地址（程序计数器的值，在被调过程结束时，控制从被调过程返回到该值所指位置）
  - 寄存器的内容：主调过程会使用这些内容，被调过程必须在返回时恢复这些内容



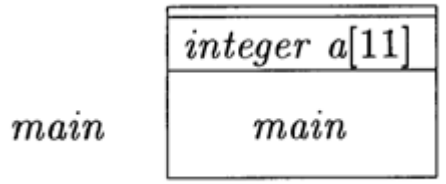
运行时刻栈的例子

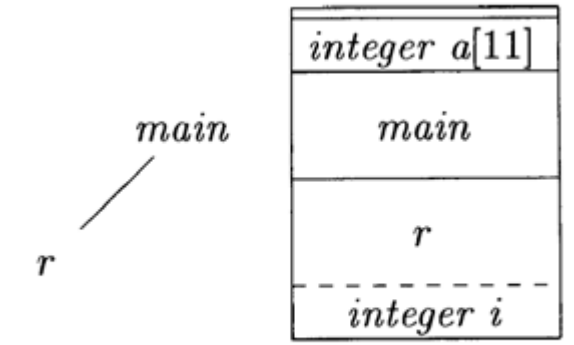
a[11]为全局变量

main没有局部变量

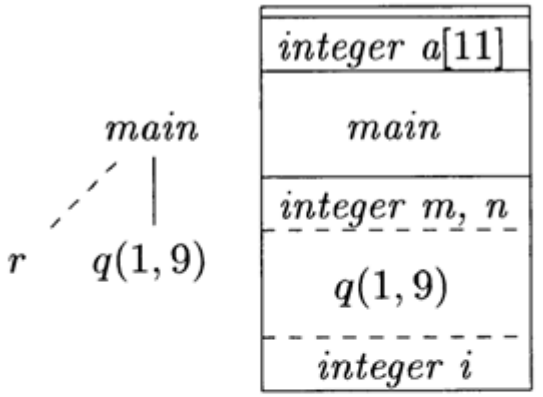
r有局部变量i

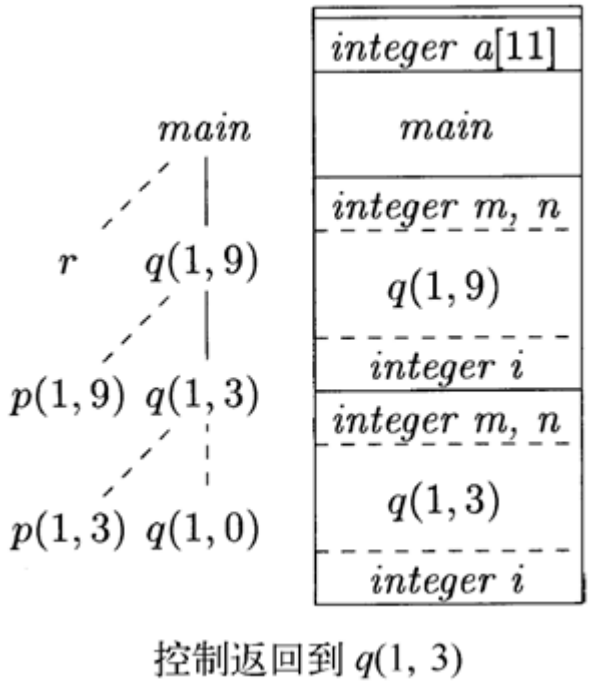
q的局部变量i，和参数m,n。





γ被激活





控制返回到 $q(1, 3)$

γ被弹出栈，$q(1, 9)$ 被压栈

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

过程调用（返回）序列和活动树的前序（后序）遍历对应

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+返回代码序列

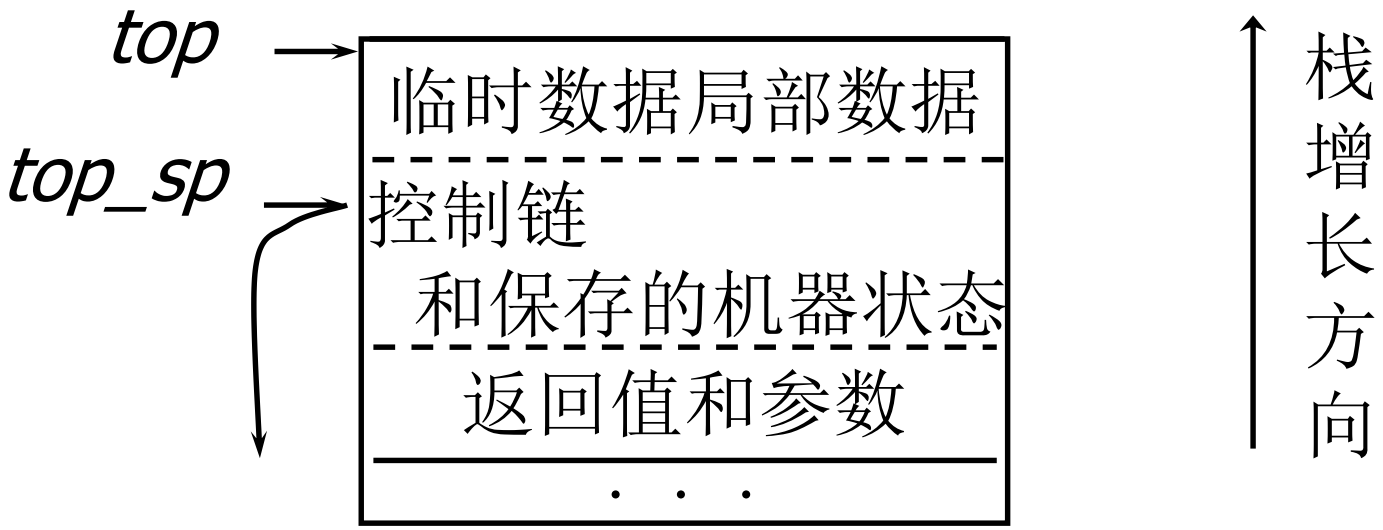
Calling sequence

- 调用者计算实在参数的值
- 将返回地址和原top_sp存放到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

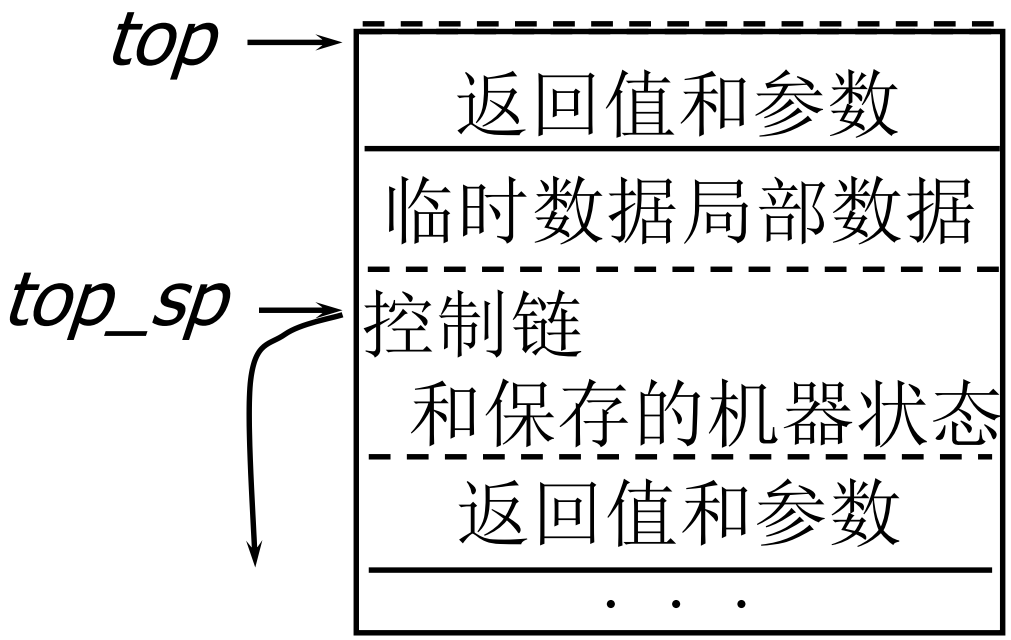
Return sequence

- 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

过程p调用过程q的调用序列

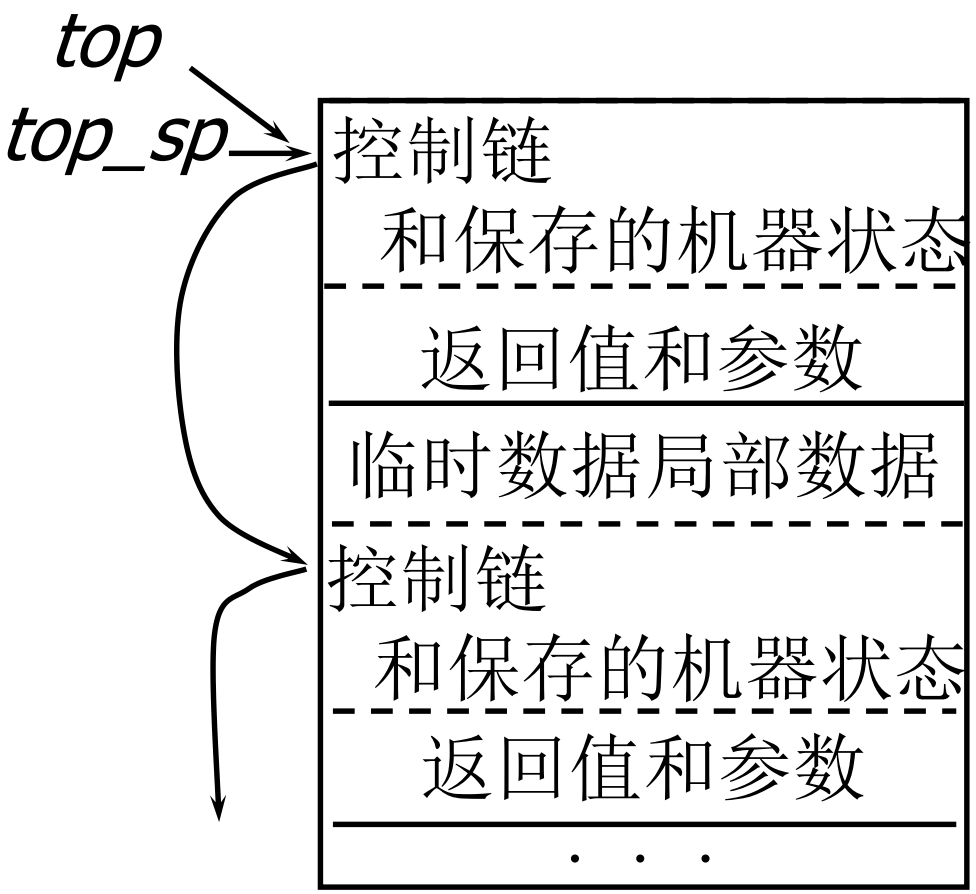


过程p调用过程q的调用序列



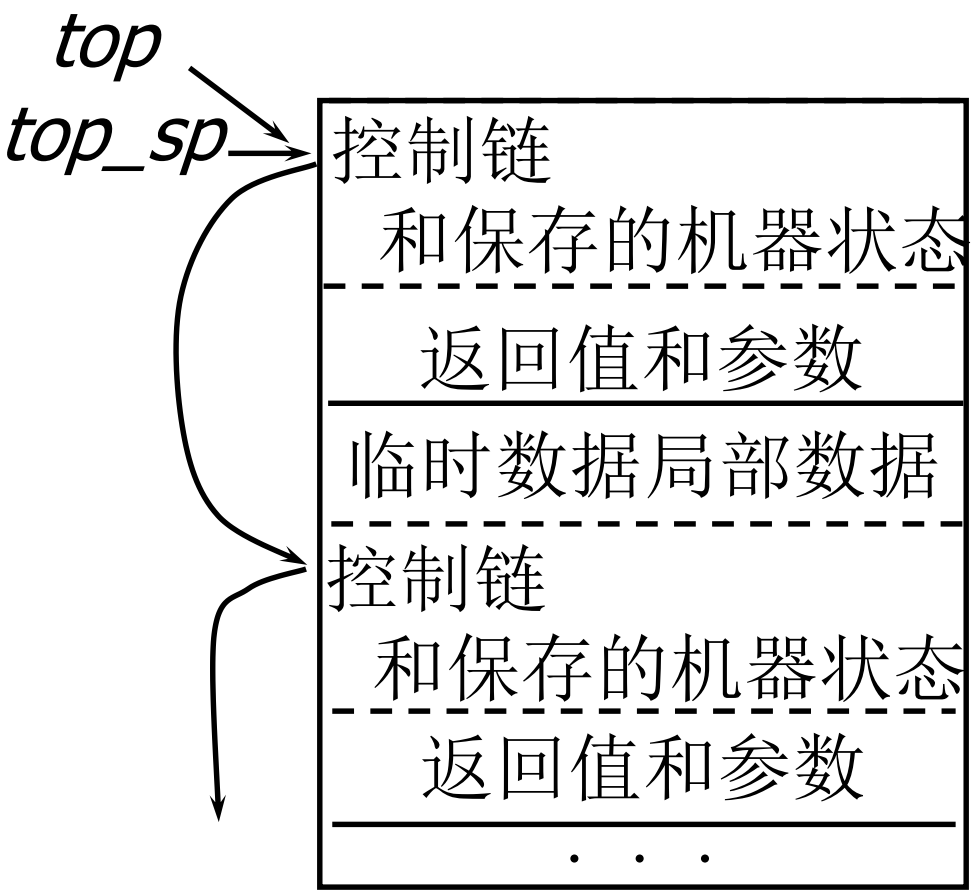
(1)p过程计算实在参数的值，依次放入栈顶，并在栈顶留出放返回值的空间，修改top指针

过程p调用过程q的调用序列



(2)p将返回地址和原top_sp存放到q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）

过程p调用过程q的调用序列

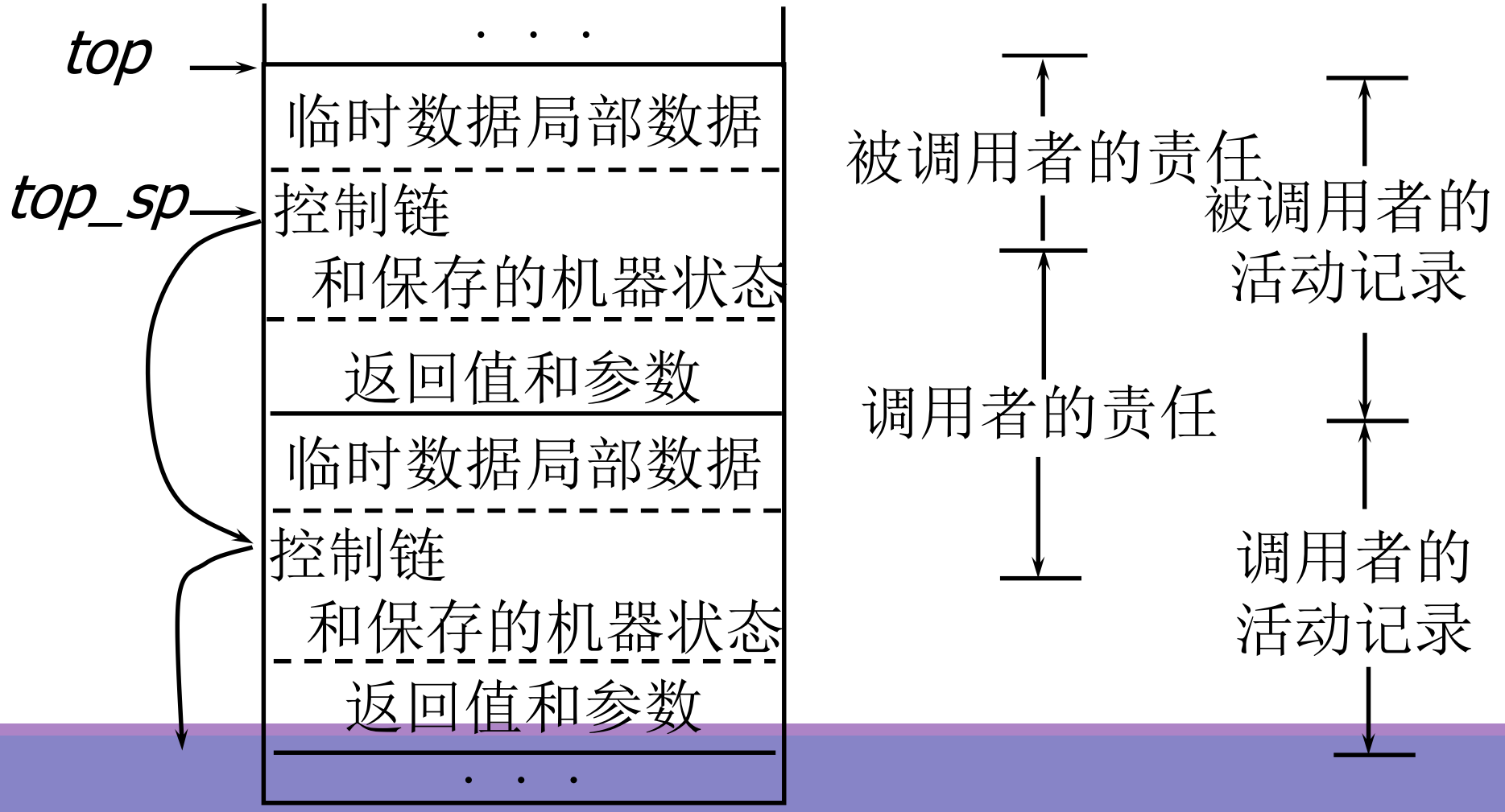


(3) q保存寄存器值和其他状态字段

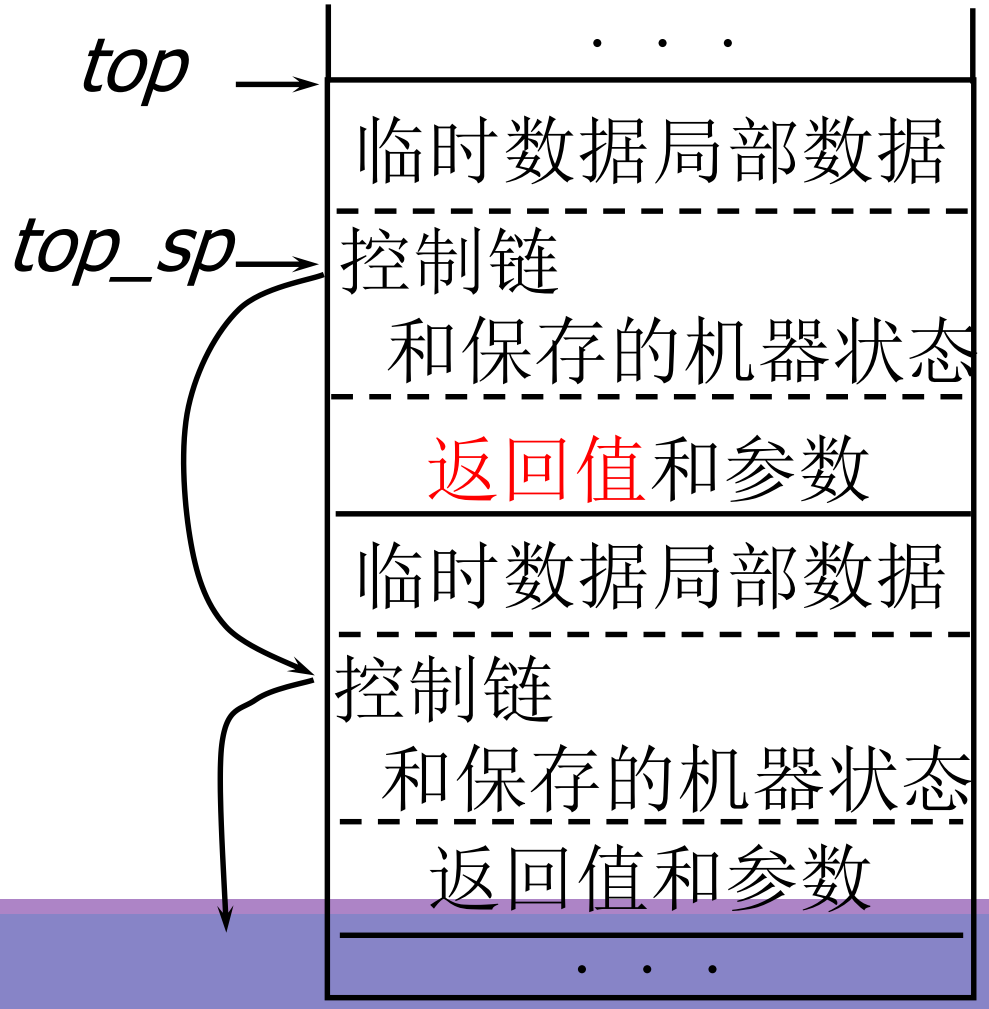
过程p调用过程q的调用序列

(4) q初始化局部数据、开始运行。

调用者和被调用者之间的任务划分

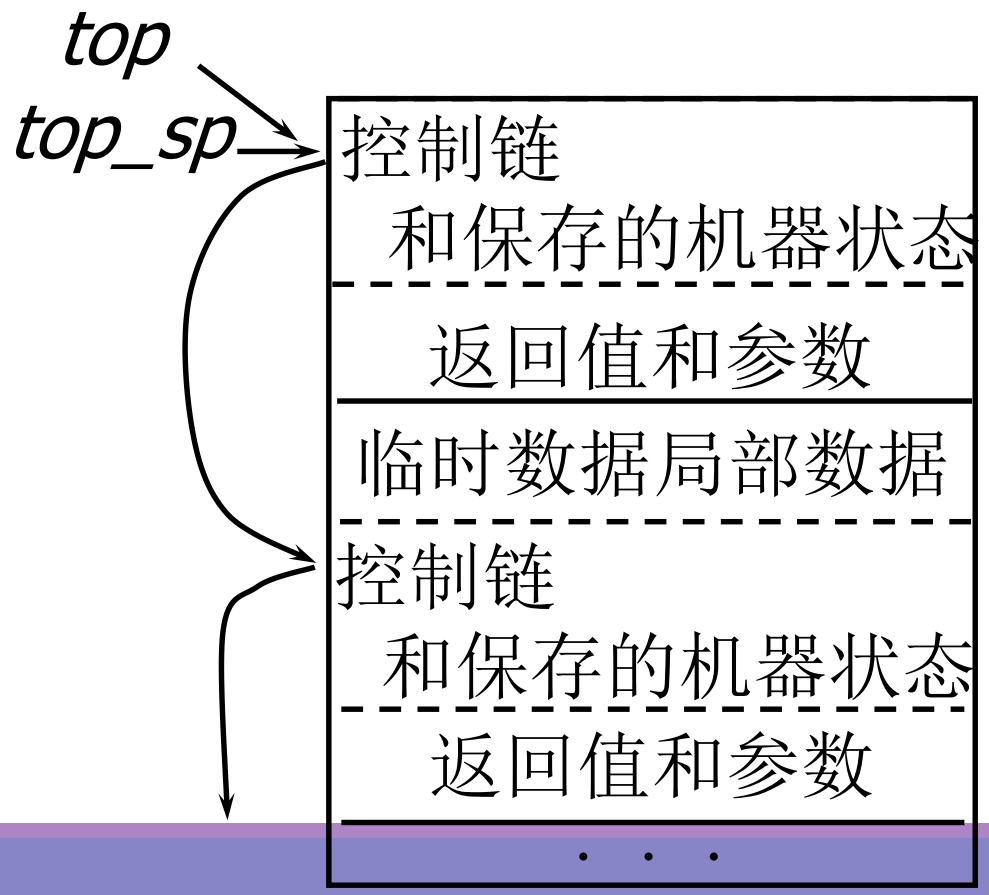


过程p调用过程q的返回序列



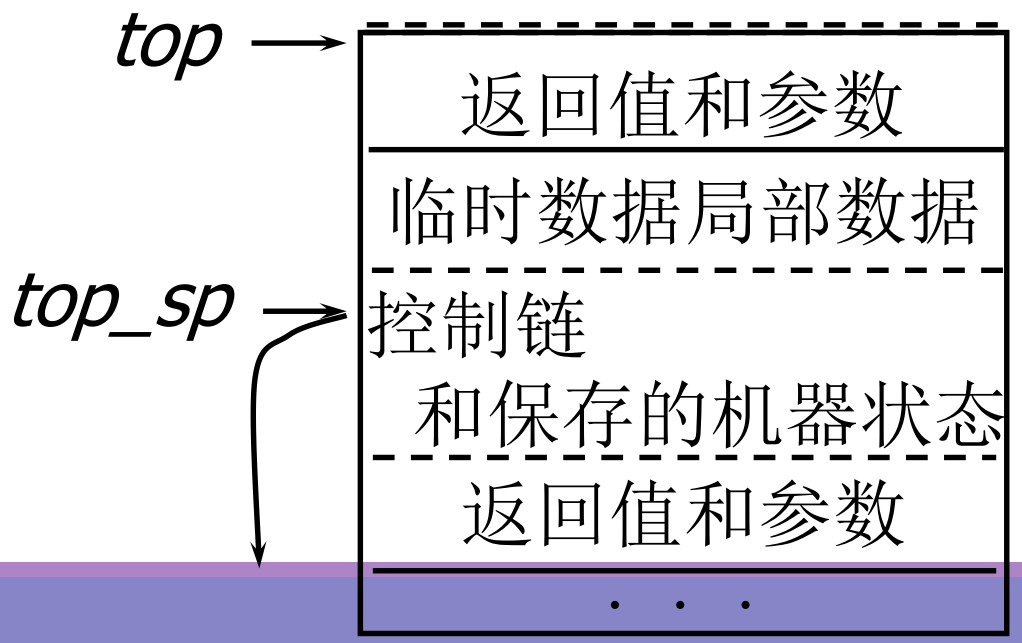
（1）q把返回值置入邻近p的活动记录的地方

过程p调用过程q的返回序列



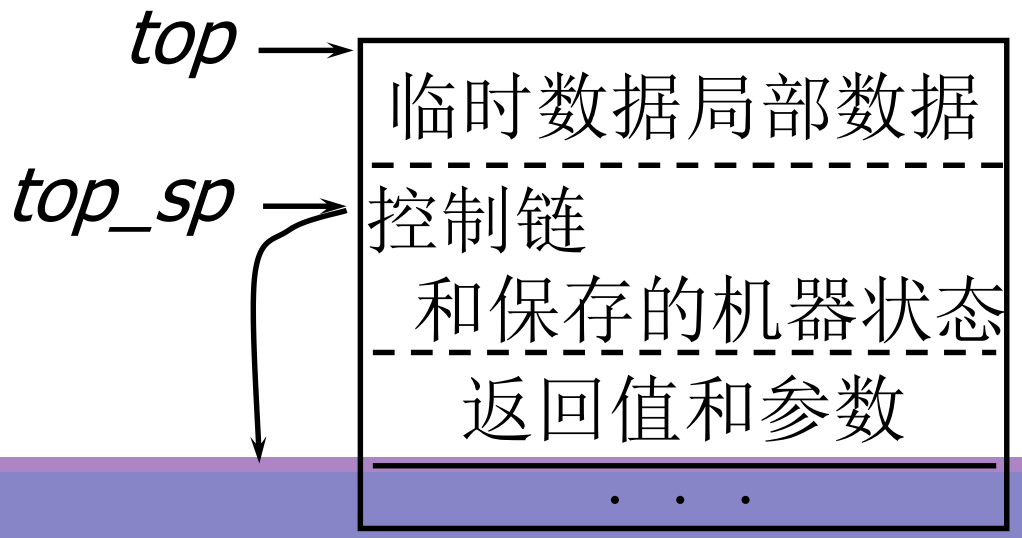
(2) q根据局部数据域和临时数据域的大小减小top的值

过程p调用过程q的返回序列



(3) q恢复寄存器（包括top_sp）和机器状态，返回p

过程p调用过程q的返回序列



(4) p根据参数个数与类型和返回值类型调整top，然后取出返回值

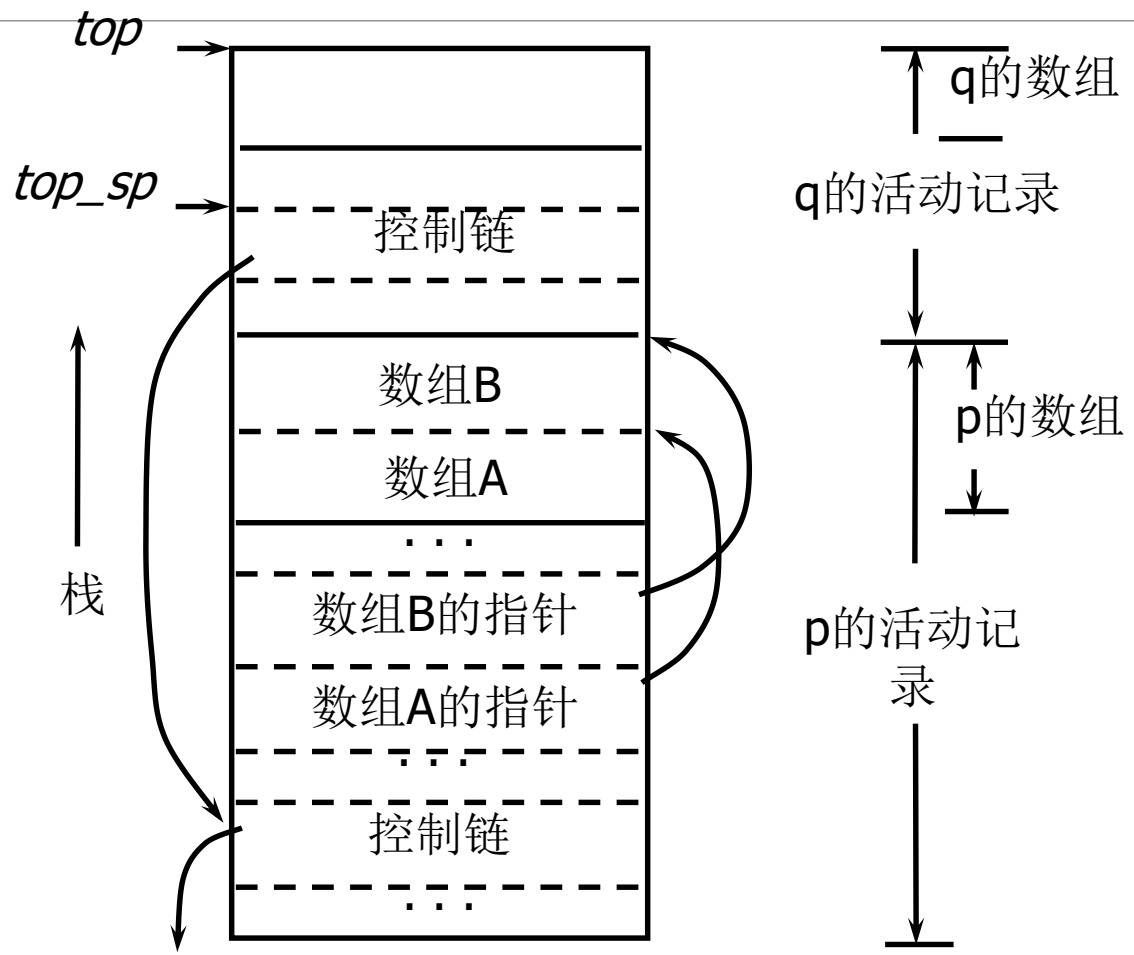
栈中的变长数据

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



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

非局部数据的访问（嵌套声明过程）

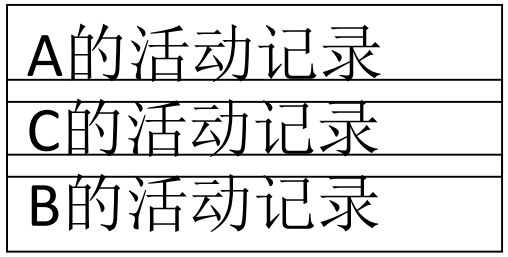
PASCAL中，如果过程A的声明中包含了过程B的声明，那么B可以使用在A中声明的变量。

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。

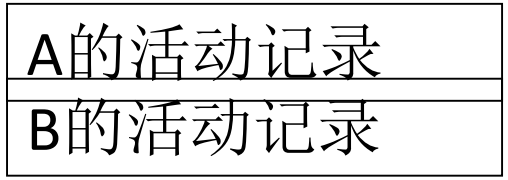
但是，我们不能通过嵌套层次直接得到A的活动记录的相对位置。必须通过访问链访问

```
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：



当A直接调用B时：



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

访问链被用于访问非局部的数据

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

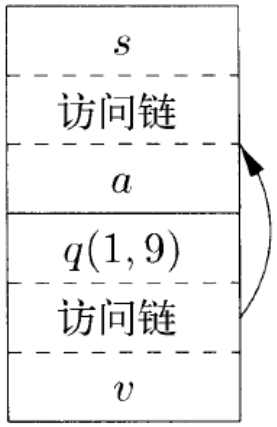
访问链的维护（直接调用过程）

当过程q调用过程p时，访问链的变化

- q的深度小于p：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q的活动记录
  - 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)
- q的深度大于等于p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

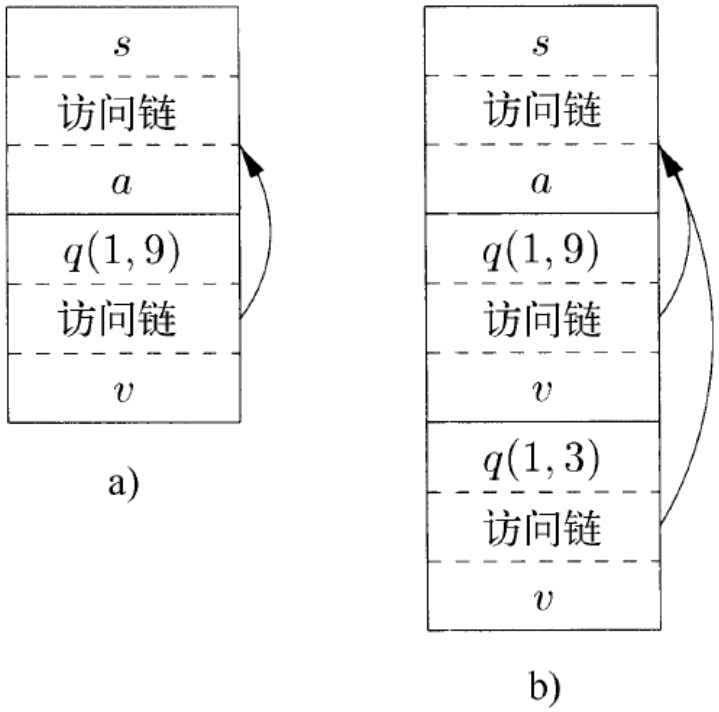
```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;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

访问链的例子

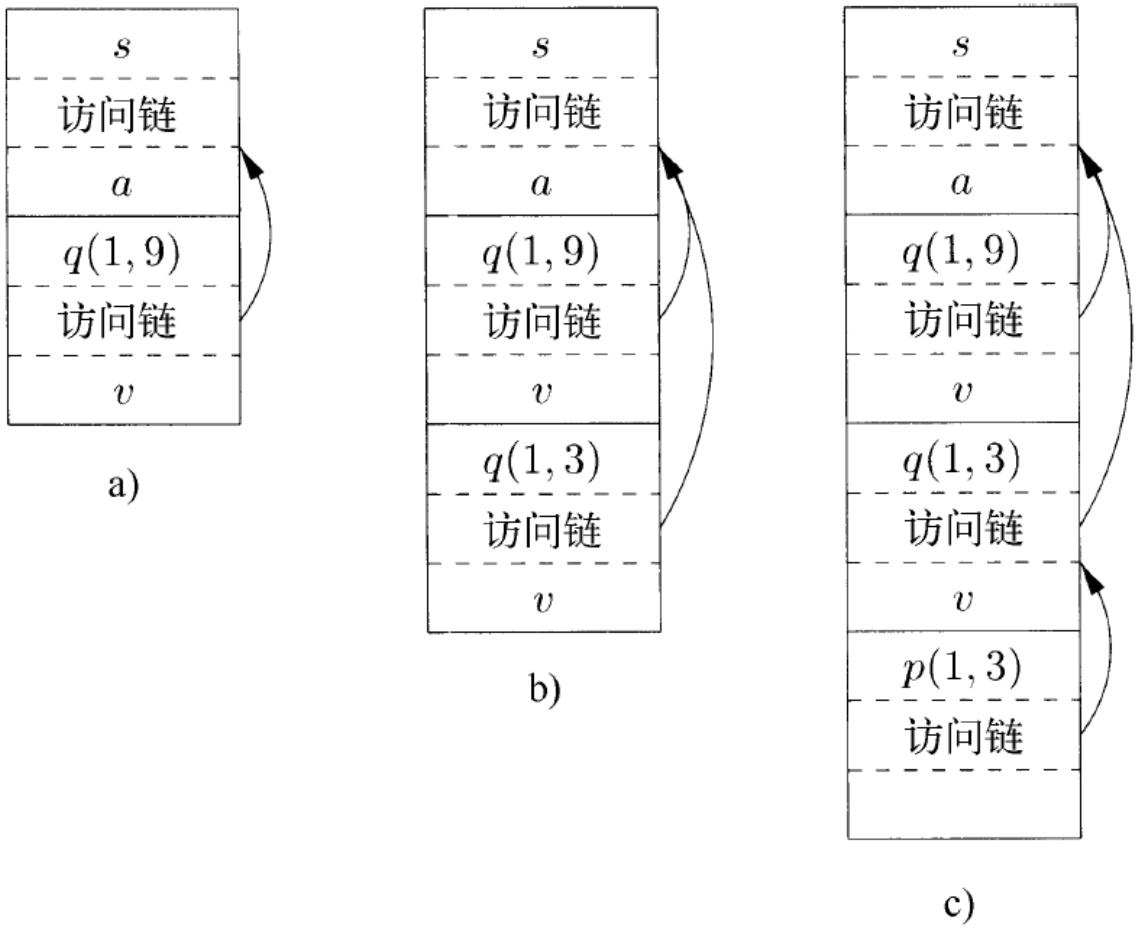


a)

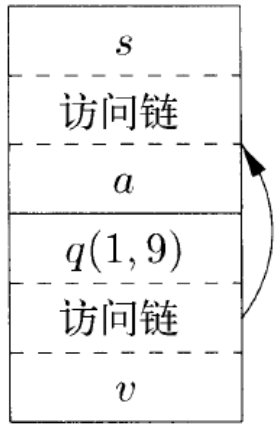
访问链的例子



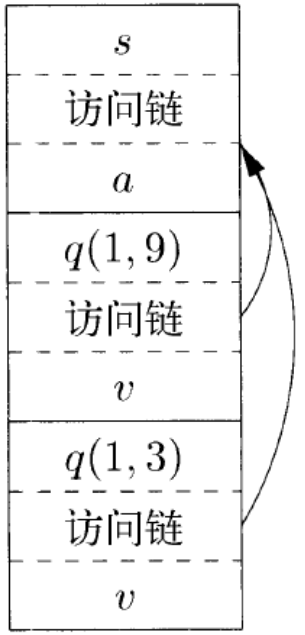
访问链的例子


s
访问链
a
q(1,9)
访问链
v
a)
s
访问链
a
q(1,9)
访问链
v
q(1,3)
访问链
v
b)
s
访问链
a
q(1,9)
访问链
v
q(1,3)
访问链
v
p(1,3)
访问链
c)

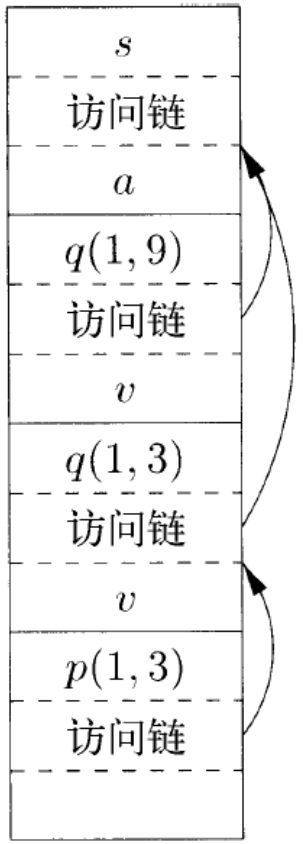
访问链例子


s
访问链
a
q(1,9)
访问链
v

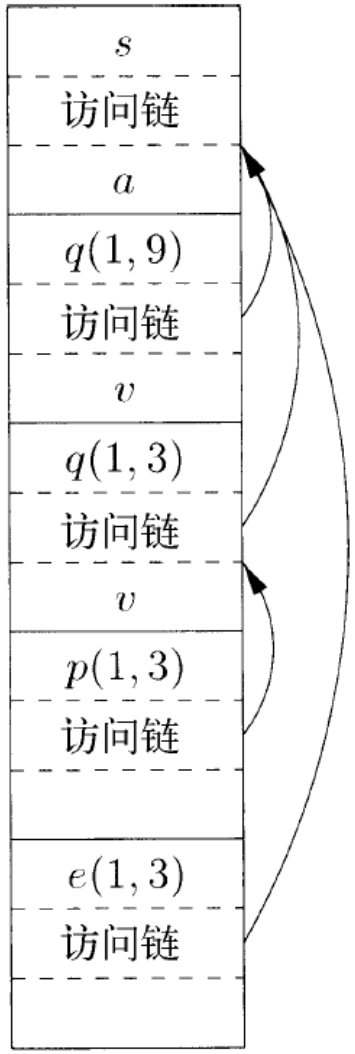
a)


s
访问链
a
q(1,9)
访问链
v
q(1,3)
访问链
v

b)


s
访问链
a
q(1,9)
访问链
v
q(1,3)
访问链
v
p(1,3)
访问链

c)

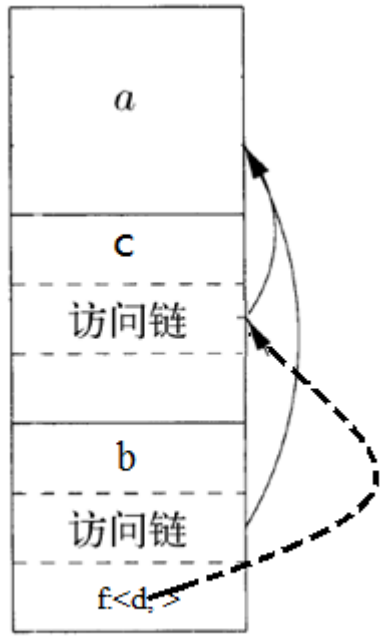

s
访问链
a
q(1,9)
访问链
v
q(1,3)
访问链
v
p(1,3)
访问链
e(1,3)
访问链

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，还包括正确的访问链。

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



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

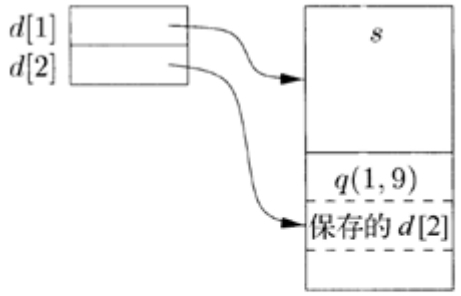
显示表：数组d为每个嵌套深度保留一个指针

- 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

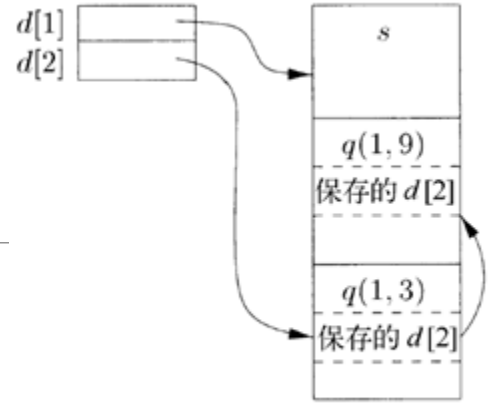
显示表的维护

- 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。

显示表例子



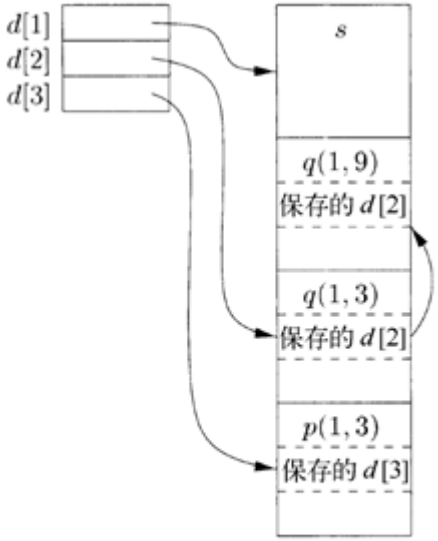
a)



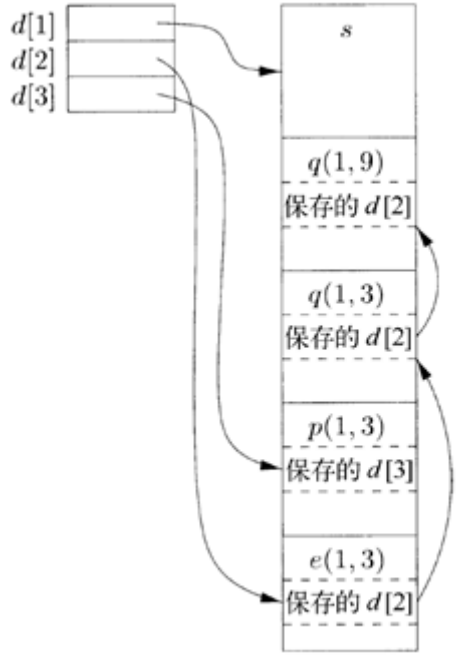
b)

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



c)



d)

q调用e，e的深度为2

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

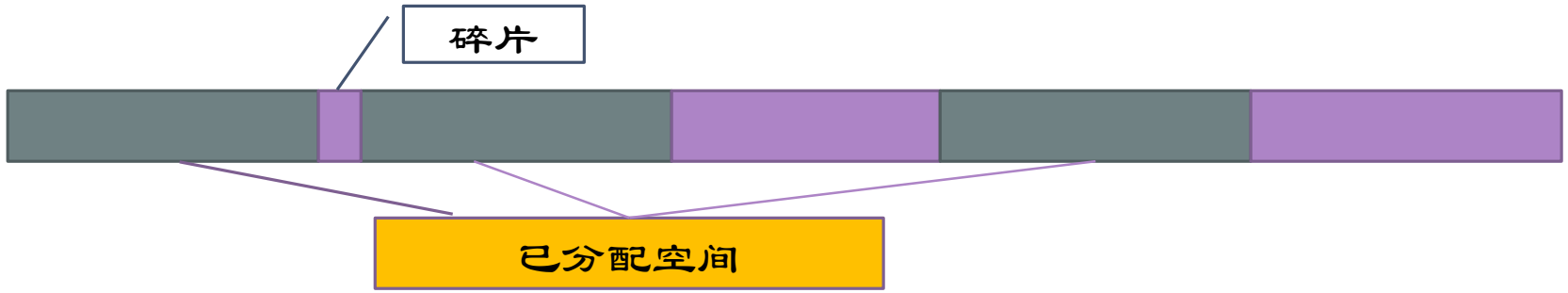
基本功能

- 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

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
  - 同一时间段内的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。

- 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

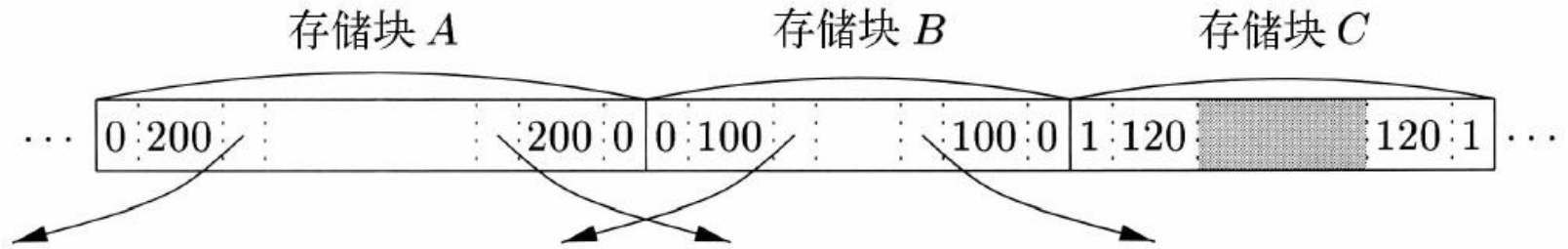
- 边界标记：在每一块存储块的两端，分别设置一个free/used位；相邻的位置上存放字节总数。
- 双重链接的、嵌入式的空闲块列表：列表的指针存放在空闲块中、用双向指针的方式记录了有哪些空闲块。

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：

- 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。

主要内容

运行时刻环境

- 为源程序中命名的对象分配安排存储位置
- 确定目标程序访问变量时使用的机制
- 过程之间的连接
- 参数传递

主题

- 存储管理：栈分配、堆管理、垃圾回收
- 对变量、数据的访问