中国科学技术大学
University of Science and Technology of China

运行时存储空间的组织与管理

《编译原理和技术》

张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容

术语

- 过程的活动(activation): 过程的一次执行
- 活动记录

  过程的活动需要可执行代码和
  存放所需信息的存储空间, 后者称为活动记录

本章内容

- 一个活动记录中的数据布局
- 程序执行过程中，所有活动记录的组织方式
- 非局部名字的管理、参数传递方式、堆管理
- 几种典型的编译运行时系统（新增）

影响存储分配策略的语言特征

- 过程能否递归
- 当控制从过程的活动返回时，局部变量的值是否要保留
- 过程能否访问非局部变量
- 过程调用的参数传递方式
- 过程能否作为参数被传递
- 过程能否作为结果值传递
- 存储块能否在程序控制下被动态地分配
- 存储块是否必须被显式地释放

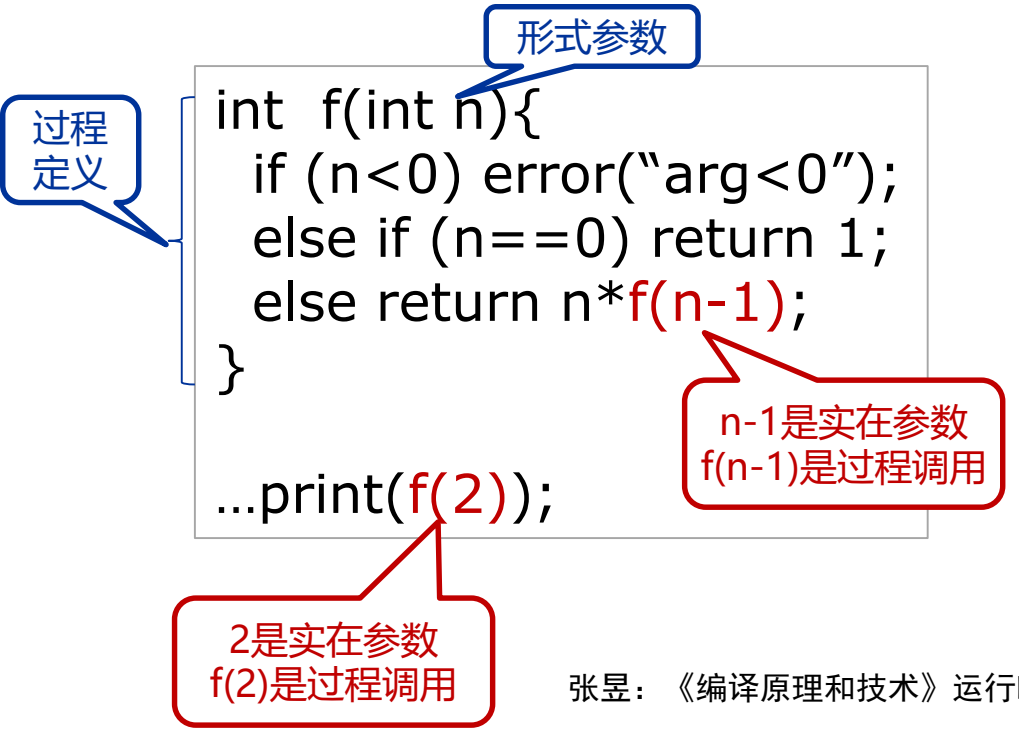
中国科学技术大学
University of Science and Technology of China

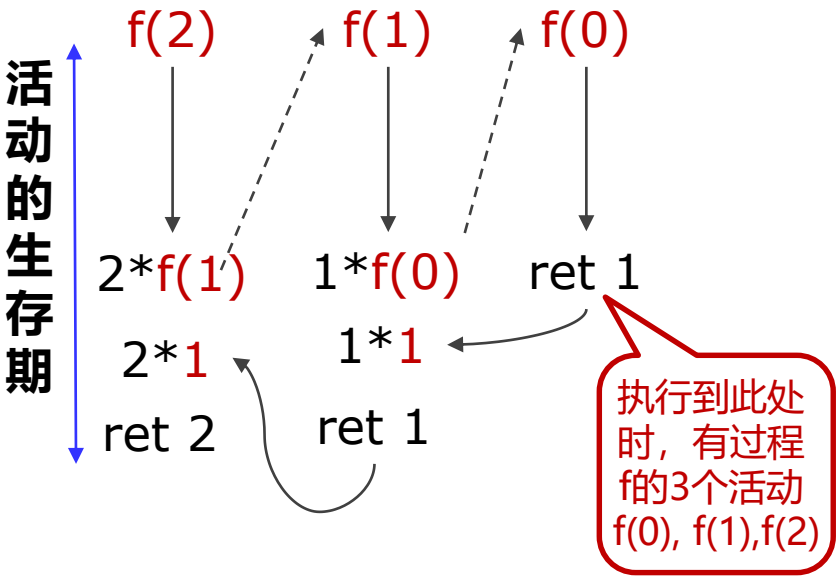
1. 名字、绑定、作用域

- 过程定义、调用、活动
- 名字、绑定、作用域、生存期
- 活动记录的常见布局
  - 字节寻址、类型、次序、对齐
- 同名变量的处理

基本概念：过程/函数

- 过程(包括函数、方法等)
  - 过程定义、过程调用、形式参数、实在参数
  - 活动(过程的一次调用)、活动的生存期
  - **每个活动有一个活动记录➔栈帧**





基本概念：名字

- von Neumann体系：内存、处理器
- 名字≡ 标识符
  - 与程序中的过程、形参、变量等程序构造相关联
  - 命名约定 ➔ 构词规则
  - 关键字和保留字：相同的名字可否有不同的含义？

Fortran：关键字不是保留字

```
Integer Apple
Integer = 4
Integer Real
Real Integer
```

关键字可以作为变量名

C/C++、Java：关键字是保留字

```
int i; /* 合法 */
float int;  /*不合法*/
```

基本概念：变量

- 变量：程序语言中对机器的内存单元的抽象
  - 名字、类型、字宽、地址、作用域、生存期(lifetime)

① 绝大多数的变量都有**名字**

  - 没有名字的变量：临时变量、存储在堆中的变量

② 变量的**地址**≡ 左值　变量的**值**≡ 右值

  - 程序中相同的变量在不同时间关联到不同的地址
  - 环境把名字映射到左值，而状态把左值映射到右值
  - 赋值改变状态，但不改变环境
  - 过程调用改变环境：不同的活动有不同的活动记录
  - 如果环境将名字 x 映射到存储单元 s，则说 x 被绑定

基本概念：绑定

- 绑定(binding)：程序中实体和属性之间的关联
  - 变量和其类型、值
  - 符号和其操作：
    如“+”绑定到整数加add、浮点加fadd(x86)/adf(ARM)
- 绑定时间：绑定被创建的时间点
  - 语言设计时：程序结构、可能的类型
  - 语言实现时：I/O、运算的溢出、类型等价性
  - 程序编写时：算法、名字
  - 编译时：数据布局的规划
  - 链接时：整个程序在内存中的布局
  - 加载时：物理地址的选择
  - 运行时：变量-值的绑定、程序启动时间、过程进入时间、语句执行时间等

基本概念：绑定

绑定时机

语言设计时、语言实现时、程序编写时、编译时、链接时、加载时、运行时

如：count = count + 2;

- count的类型在编译时绑定
- count的可能取值集合在语言设计时绑定
- count的值在运行时绑定
- +的含义在编译时当确定操作数的类型时被绑定
- 2的内部表示在语言设计时被绑定

基本概念：类型绑定

- 静态绑定：运行前发生并且在程序执行期间保持不变
- 动态绑定：运行期间发生或者在程序执行期间改变

类型绑定：变量在被引用前必须绑定到数据类型

- 静态类型绑定≡变量声明
  - 显式声明
  - 隐式声明，如Fortran的命名约定：凡以字母I~N六个字母开头的变量名，如无另外说明则为整型变量
- 动态类型绑定，如JavaScript、Python、PHP、Ruby
  - 不用声明变量的类型，根据对变量的赋值**推断**其**类型**

JavaScript：`list = [2, 4.33, 6, 8];` 此时`list`为数组类型

`list = 17;` 此时`list`为整型

C# 2010：`dynamic any;` `any`可以被赋予任何类型的值[链接]

基本概念：类型绑定

- **静态类型绑定的隐式声明**

  带来方便，但可能会损害可靠性(阻止编译期间检测某些拼写错误和编程错误)，进一步的解决办法有：

  - **Fortran**：引入**implicit none** 声明使隐式声明失效
  - **Perl:**变量名以**$、@、%**开始，代表**标量、数组、散列**

- **动态类型绑定**

  灵活（通用的程序单元），但缺点是：

  - 代价高：动态类型检查、动态存储分配(变量的存储大小是可变的）、解释执行
  - 难以在编译时检测类型错误
  - 这些语言通常用解释器实现
    - **Google的V8(JavaScript引擎)引入hidden class以提升性能**

基本概念：生存期与作用域

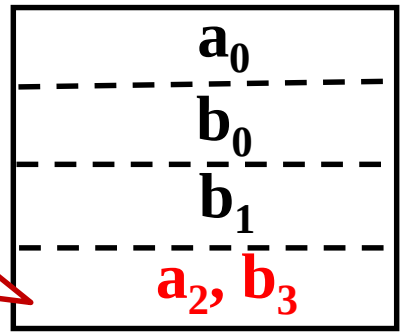
- 存储绑定与生存期(lifetime)
  - **存储绑定**：变量所绑定的内存单元的**分配、回收**
    - **分配机制**：**静态、栈、堆**
  - **生存期**：变量绑定到某个存储单元的时间区间
    C、C++的存储类别：static、extern、auto、register
- 控制绑定与作用域(scope)
  - **作用域**：一个(变量/过程)**声明**起作用的程序部分
  - 局部变量、非局部变量：同名变量的合法性规定
  - **命名空间(namespace)**：不同命名空间的同名符号的含义互不相干

程序块与同名变量的处理

- 现代语言一般可在**程序块**中的任何地方声明
  - C99、C++、 Java: 作用域为从声明处开始到该语句块结尾结束
  - C#：作用域整个语句块
- C/C++中可以嵌套声明同名变量，按**最近(小)嵌套作用域规则**，但在Java和C#中不合法—容易出错
- **并列程序块**不会同时活跃,不同并列块中的变量可以重叠分配

```
main()
{
  int a = 0;                 // B0
  int b = 0;
  {                          // B1
    int b = 1;
    {                        // B2
      int a = 2;
    }
    {                        // B3
      int b = 3;
    }
  }
}
```

a_i: 作用域B_i中声明的变量a



B_2、B_3不会同时运行，故B_2中的a_2和B_3中的b_3复用存储空间

全局作用域

- C、C++、Python等**允许在函数定义外声明变量**
- C、C++有全局变量声明和定义，后者要分配内存单元
- C、C++同名局部变量与全局变量作用域重叠的，**重叠部分按局部变量处理**

C++

```
#include <iostream>
void func( float );
const int a = 17;              // global constant
int b, c;                      // global variable
int main()
{
        b = 4;                 // assignment to global b
        c = 6;                 // assignment to global c
        func(42.8);  return 0;
}
void func( float c)        // prevents access to global c
{
        float b;               // prevent access to global b
        b = 2.3;               // assignment to local b
        cout << " a = " << a;      // output global a (?)
        cout << " b = " << b;      // output local b (?)
        cout << " c = " << c;      // output local c (?)
}
```

Output:
A = 17 b = 2.3 c = 42.8

Python： 全局变量可以在函数中引用，但是只能对在函数中**声明为global**的全局变量**赋值**

```
a = 3
def Fuc():
  print (a)
  a = a + 1
Fuc()
```

```
a = 3
def Fuc():
  print (a)
Fuc()
```

```
a = 3
def Fuc():
  global a
  print (a)
  a = a + 1
Fuc()
```

类与作用域、命名空间

- 类有自己的局部变量
  - 类变量、实例变量
  - 方法中声明的变量
  - 访问属性
    - public
    - protected
    - private

- 命名空间
  - C++允许用户创建自己的命名作用域
  - 如标准的cstdlib头文件包含一些库函数的原型声明

```
namespace std
 {
      ..
      int abs (int );
      ..
 }
```

```
#include <cstdlib>
 int main()
 {
     int alpha;
     int beta;
     ..
     alpha = std::abs(beta);
 }
```

Scope resolution operator

基本概念

静态概念和动态概念的对应

静 态 概 念	动 态 对 应
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

活动记录（activation record）

- 常见布局

局部数据的布局

存储布局的一些因素

- 字节是可编址内存的最小单位
- 变量所需的存储空间可以根据其类型而静态确定
- 一个过程所声明的局部变量，按这些变量声明时出现的次序，在局部数据域中依次分配空间
- 局部数据的地址可以用相对于活动记录中某个位置的地址来表示
- 数据对象的存储布局还需考虑对齐问题

对齐对存储size的影响

例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16，为什么不一样？

```
typedef struct  _a{
    char    c1;
    long    i;
    char    c2;
    double f;
}a;
```

```
typedef struct  _b{
    char c1;
    char    c2;
    long i;
    double f;
}b;
```

对齐：char : 1, long : 4, double : 8

对齐对存储size的影响

例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16，为什么不一样？

```
typedef struct  _a{          typedef struct  _b{
    char    c1;      0           char c1;       0
    long    i;       4           char   c2;     1
    char    c2;      8           long i;        4
    double f;        16          double f;      8
}a;                              }b;
```

对齐：char : 1, long : 4, double : 8

对齐对存储size的影响

例 在X86/Linux工作站上下面两个结构体的size分别是20和16，为什么不一样？

```
typedef struct  _a{
    char    c1;     0
    long    i;      4
    char    c2;     8
    double f;       12
}a;
```

```
typedef struct  _b{
    char c1;        0
    char    c2;     1
    long i;         4
    double f;       8
}b;
```

对齐：char : 1, long : 4, double : 4

例　题　1

一个C语言程序及其在X86/Linux操作系统上的编译结果如下。根据生成的汇编程序来解释程序中四个变量的存储分配、生存期、作用域和置初值方式等方面的区别

```
static long aa = 10;
short bb = 20;
extern int f( );
int func( ) {
    static long cc = 30;
    short dd = 40;
    cc = f(cc,dd);
}
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

分配8字节

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
int func( ) {
    static long cc = 30;
    short dd = 40;
    cc = f(cc, dd);}
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

分配8字节

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

```
int func( ) {
static long cc = 30;
short dd = 40;
cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

实参dd先提升成4字节整型，

再通过寄存器esi传参

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
static long aa = 10;
short bb = 20;
extern int f();
```

```
.data
.align 8
    .type aa,@object
    .size aa,8
aa:
    .quad 10
    .globl bb
    .align 2
    .type bb,@object
    .size bb,2
bb:
    .value 20
```

实参cc先加载到寄存器rax，

再通过寄存器rdi传参

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);}
```

```
    .align 8
    .type   cc.1797,@object
    .size   cc.1797, 8
cc.1797:
    .quad 30
.text
    .globl func
    .type func, @function
func: . . .
    movw $40,-2(%rbp)
    movswl -2(%rbp), %edx
    movq cc.1797(%rip), %rax
    movl  %edx, %esi
    movq %rax, %rdi
    movl  $0, %eax
    call  f
```

在64位系统用gcc –S编译


```
func:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movw    $40, -2(%rbp)
        movswl  -2(%rbp), %edx
        movq    cc.1797(%rip), %rax
        movl    %edx, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    f
        cltq
        movq    %rax, cc.1797(%rip)
        nop
        leave
        ret
```

分配局部变量空间，按16字节对齐

```
int func( ) {
  static long cc = 30;
  short dd = 40;
  cc = f(cc, dd);
}
```

f 函数的返回值通过寄存器eax 返回

cltq等效于movslq %eax, %rax

2. 多个活动记录的组织

- 程序运行时各个活动记录的存储分配策略
  - 静态、栈式、堆式
- 过程的目标代码如何访问名字对应的存储单元

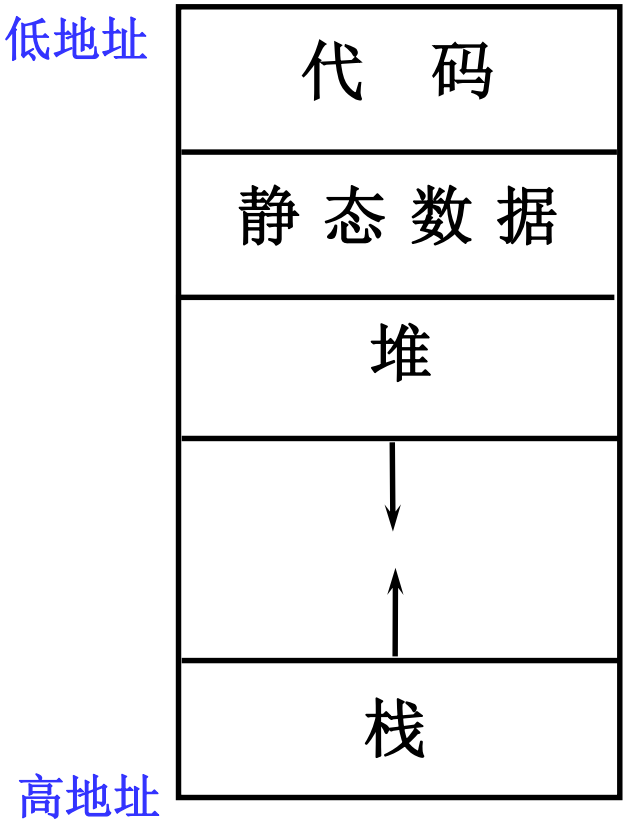
进程地址空间和静态分配

静态分配

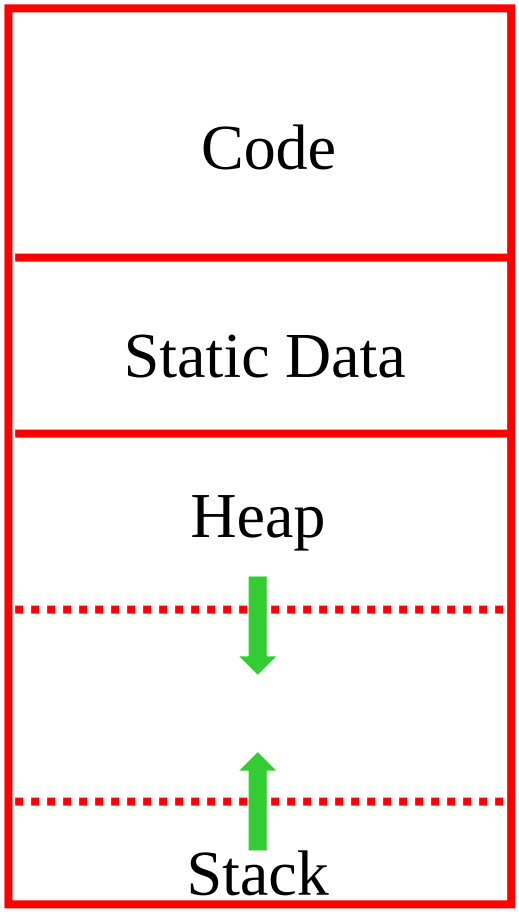
- 名字在程序被编译时绑定到存储单元，不需要运行时的任何支持
- **生存期**是程序的整个运行期间

纯静态分配给语言带来的**限制**：

- 不允许递归过程
- 数据对象的长度和它在内存中的位置必须是在编译时可以知道的
- 数据结构不能动态建立
  - 可以用静态链模拟实现



Linux的地址布局



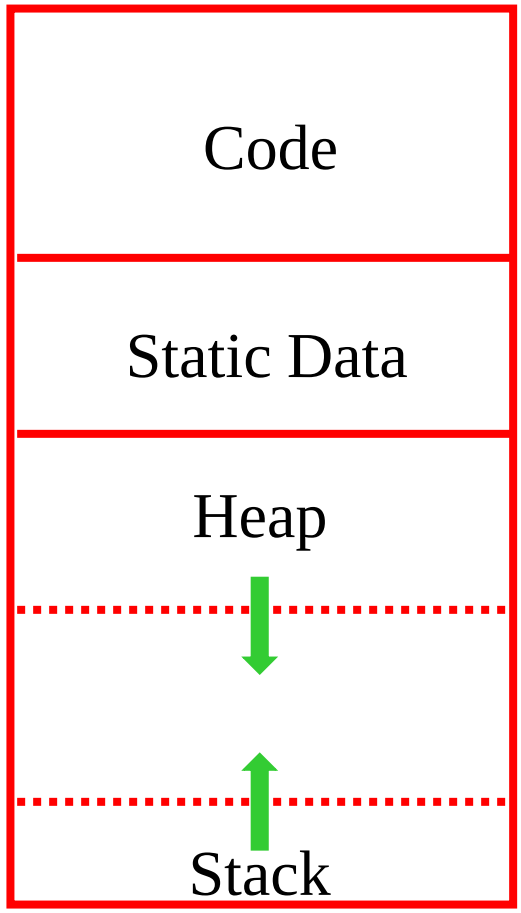
低地址 0x080480000

32位Linux系统：

- **4GB**
- **用户空间：低3GB**
  0-0xBFFFFFFF
- **内核空间：1GB**
  0xc0000000-0xFFFFFFFF
- **栈的大小**
  RLIMIT_STACK 通常为8MB

高地址 0xC0000000
TASK_SIZE

Linux的地址布局



低地址 0x0000000000400000

64位Linux系统：

- **使用低48位虚拟地址， 48位至63位必须与47位一致 （256TB)**
- **用户空间：低128TB 0-0x7FFFFFFFFFFF**
- **内核空间：64TB 0xFFFF880000000000-0xFFFFc7FFFFFFFFFF**

高地址 0x00007FFFFFFF0000
TASK_SIZE

C语言的存储分配

- 声明在函数外面
  - 外部变量extern -- 静态分配
  - 静态外部变量static -- 静态分配

    （改变作用域）
- 声明在函数里面
  - 静态局部变量static -- 也是静态分配

    （改变生存期）
  - 自动变量auto -- 在活动记录中

C程序举例、问题与分析

1. 当执行到f1(0)时，有几个f1的活动记录？
2. f1(3)的值是多少？f2(3)呢？
3. 怎么解释在某些系统下f2(3)为0？
4. 对f3(n)编译会报错吗？为什么？
5. 如果编译不报错，执行f3(n) 运行时会产生什么现象？怎么解释这种现象？

请补齐右边的三段程序，成为三个独立的C程序，然后用gcc –m32 -S编译之，产生汇编码并理解和分析。

```
int  f1(int n){
  if (n==0) return 1;
  else return n*f1(n-1);
}
… print ( f1(3) ); …
```

```
int  f2(int n){
  static int m; m = n;
  if (m==0) return 1;
  else return m*f2(m-1);
}
… print ( f2(3) ); …
```

```
int n=3;
int  f3(){
  if (n==0) return 1;
  else return n*f3(n-1);
}
… print ( f3(n) ); …
```

C程序举例、问题与分析

1. 当执行到f1(0)时，有几个f1的活动记录？

   f1(3), f1(2),f1(1),f1(0) -- 运行栈

2. f1(3)的值是多少？f2(3)呢？

   6；6或0

3. 怎么解释在某些系统下f2(3)为0？

   表达式的代码生成(寄存器分配策略)

4. 对f3(n)编译会报错吗？为什么？

   不会，主要做函数值的类型检查

5. f3(n) 运行时会产生什么现象？

   Segmentation fault

```
int  f1(int n){
  if (n==0) return 1;
  else return n*f1(n-1);
}
… print ( f1(3) ); …
```

```
int  f2(int n){
  static int m; m = n;
  if (m==0) return 1;
  else return m*f2(m-1);
}
… print ( f2(3) ); …
```

```
int n=3;
int  f3(){
  if (n==0) return 1;
  else return n*f3(n-1);
}
… print ( f3(n) ); …
```

活动树和运行栈

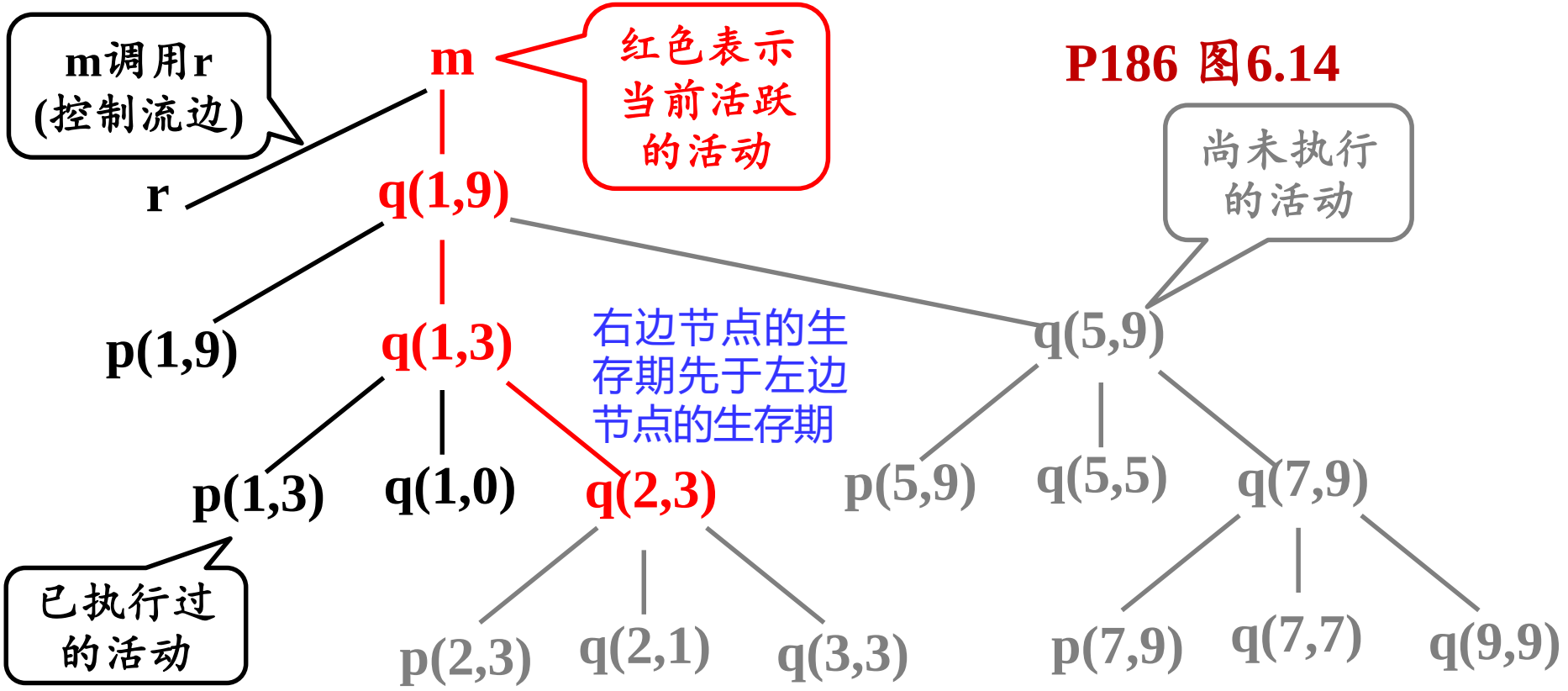
P186 图6.14

```
(1) program sort(input, output);
(2)       var a: array[0..10] of integer;
(3)       x:: integer;
(4)       procedure readArray;
(5)                 var i: integer;
(6)                 begin … a …       end      {readArray};
(7)       procedure exchange(i, j : integer);
(8)       begin
(9)                 x := a[i]; a[i] := a[j]; a[j] := x
(10)      end       {exchange};
(11)      procedure quickSort(m, n: integer);
(12)                var k, v: integer;
(13)                function partition(y, z: integer): integer;
(14)                          var i, j: integer;
(15)                          begin     … a …
(16)                                    … v …
(17)                                    … exchange(i, j); …
(18)                          end       {partition};
(19)                begin … end       {quickSort};
(20)      begin …  end       {sort}.
```

sort
- **readArray**
- **exchange**
- **quicksort**
  - **partition**

活动树和运行栈

- 活动树：用树来描绘控制进入和离开活动的方式
- 运行栈：当前活跃的活动保存在一个栈中

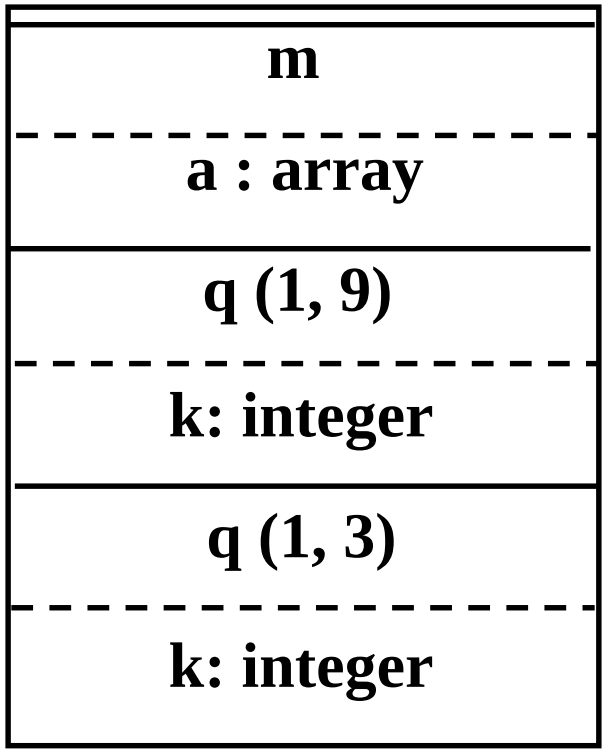


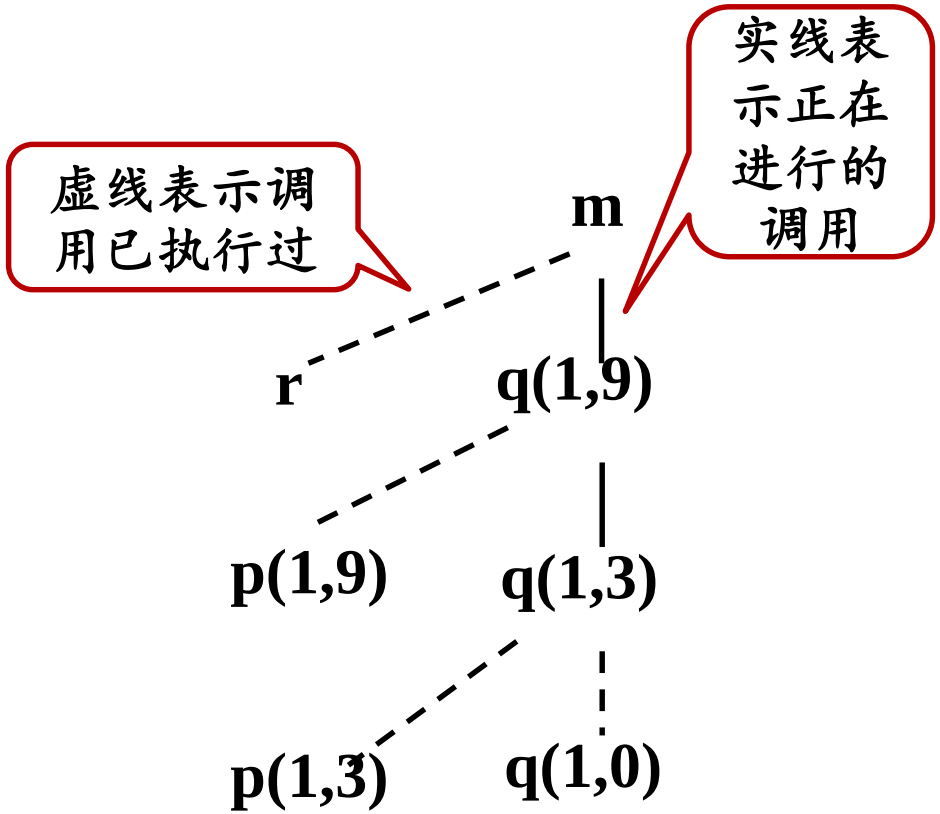
活动树和运行栈

- 活动树的特点
  - 每个**结点**代表某过程的一个**活动**
  - **根结点**代表**主程序的活动**
  - 结点a是结点b的父结点，当且仅当控制流从a的活动进入b的活动
  - 结点a处于结点b的左边，当且仅当a的生存期**先于**b的生存期
- 运行栈
  - 把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即**活动记录**）

运行栈举例

- P186 图6.14



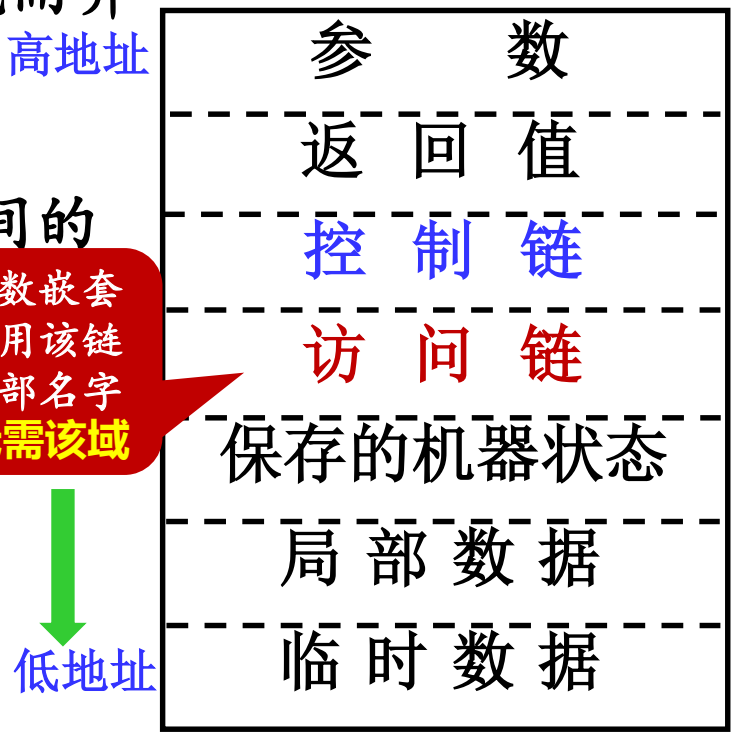


过程调用与返回和活动记录的设计

- 活动记录的具体组织和实现不唯一

  即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异

- 设计的一些原则
  - 以活动记录(大小不确定)中间的某个位置作为基地址（一般是控制链）
  - 长度能较早确定的域放在活动记录的中间
  - 一般把临时数据域放在局部数据域的后面



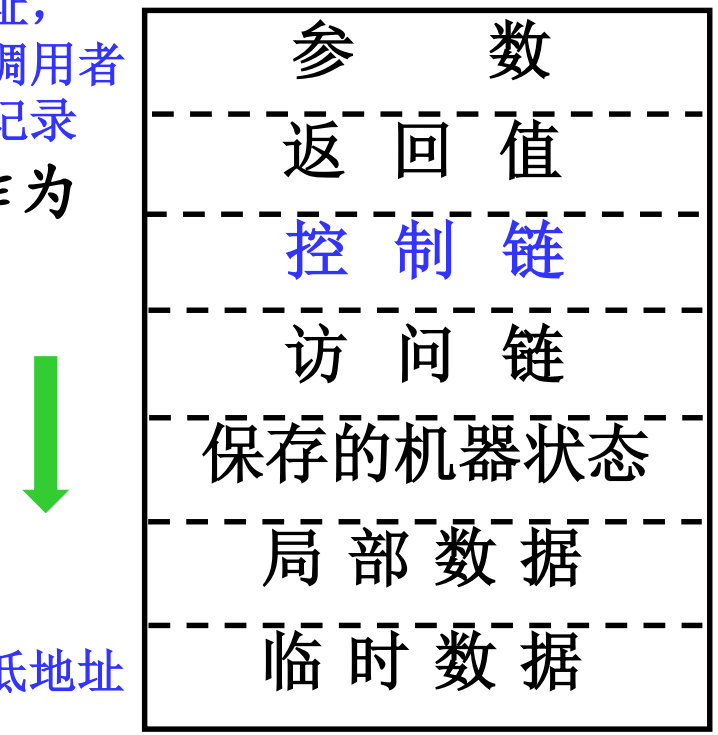
过程调用与返回和活动记录的设计

设计的一些原则

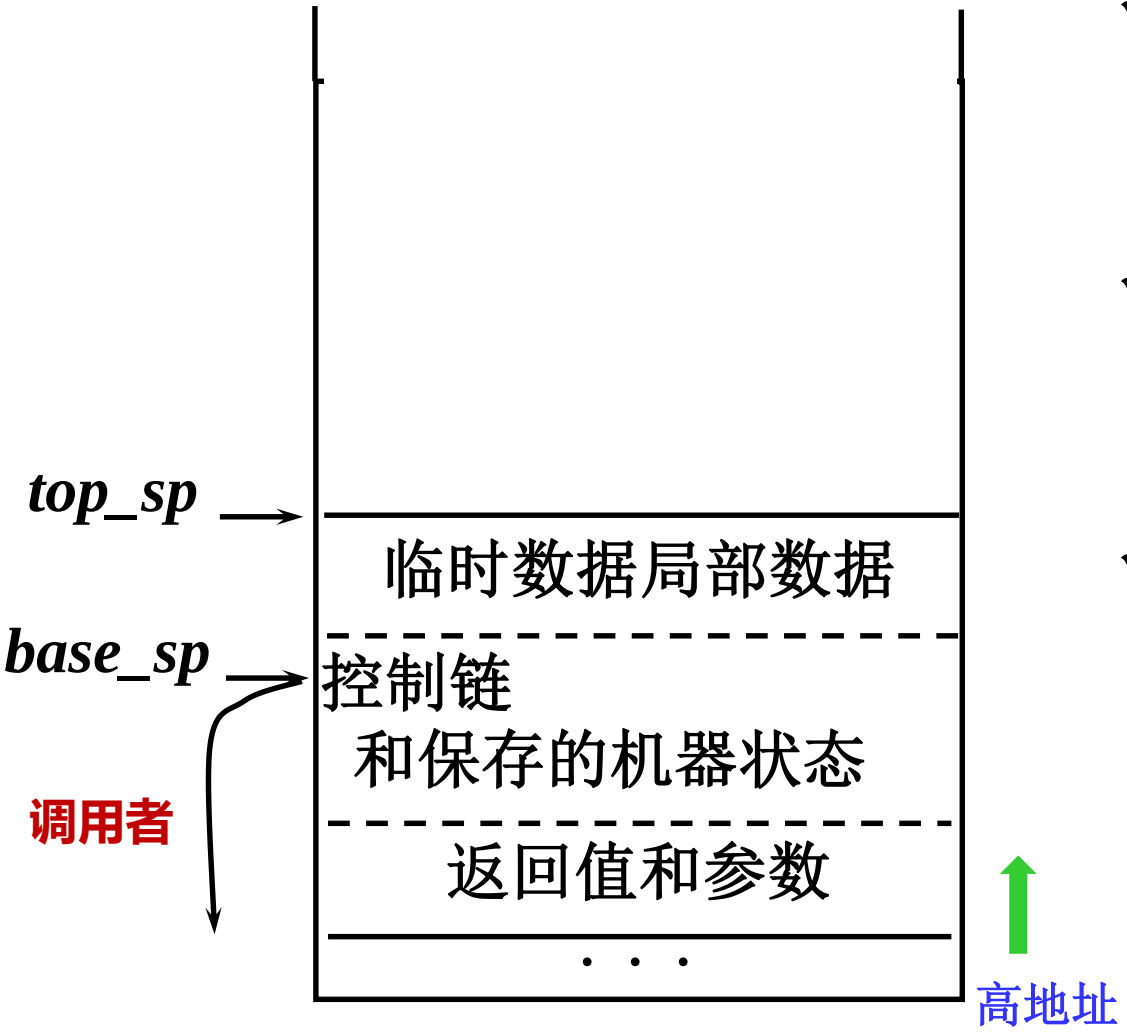
- 以活动记录中间的某个位置作为基地址（一般是控制链）
- 长度能较早确定的域放在活动记录的中间
- 一般把临时数据域放在局部数据域的后面
- 把参数域和可能有的返回值域放在紧靠调用者活动记录的地方

【有的用寄存器传参数和返回值—提升时空性能】

- 用同样的代码来执行各个活动的保存和恢复



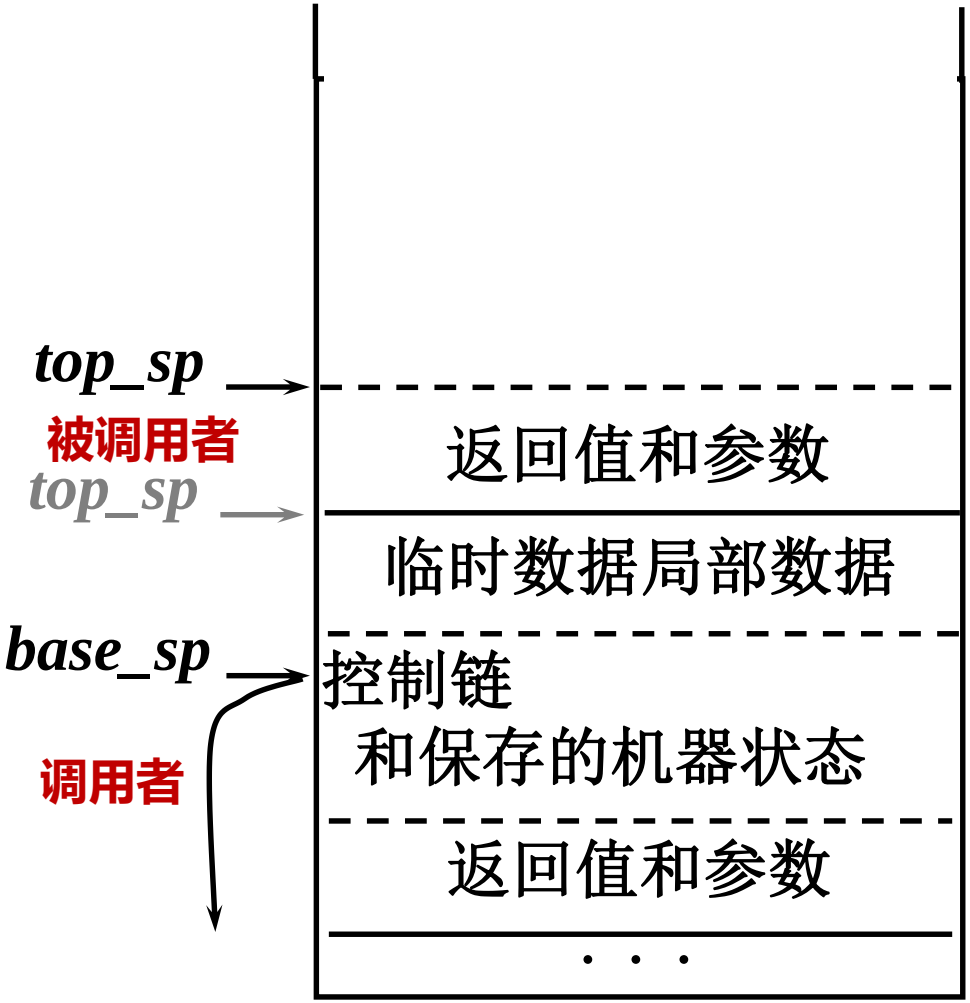
过程调用序列：p调用q



- ✓ *top_sp*: 栈顶寄存器
  - x86: esp、rsp
  - ARM: SP
- ✓ *base_sp*: 基址寄存器
  - x86: ebp、rbp
  - ARM: FP
- ✓ PC: 程序计数器
  - x86: eip、rip
  - ARM: PC

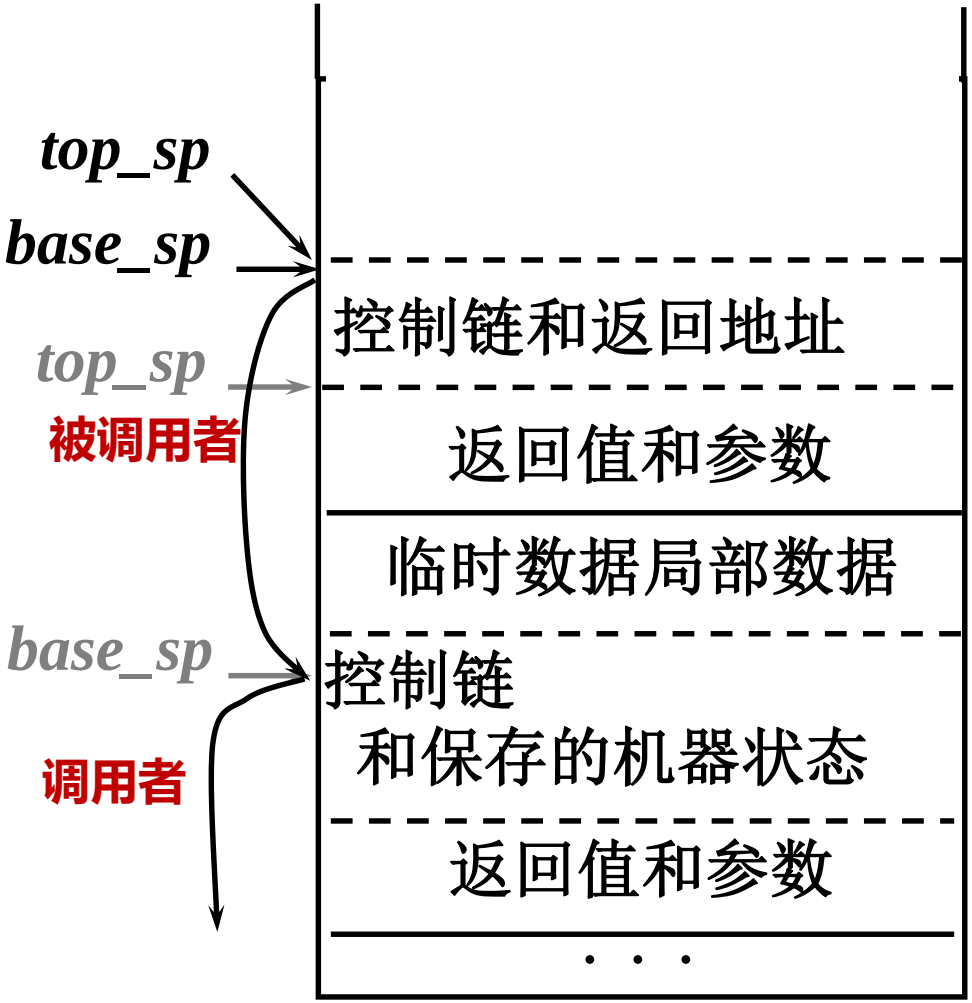
ARM: LR连接寄存器(保存子程序返回地址)

过程调用序列：p调用q



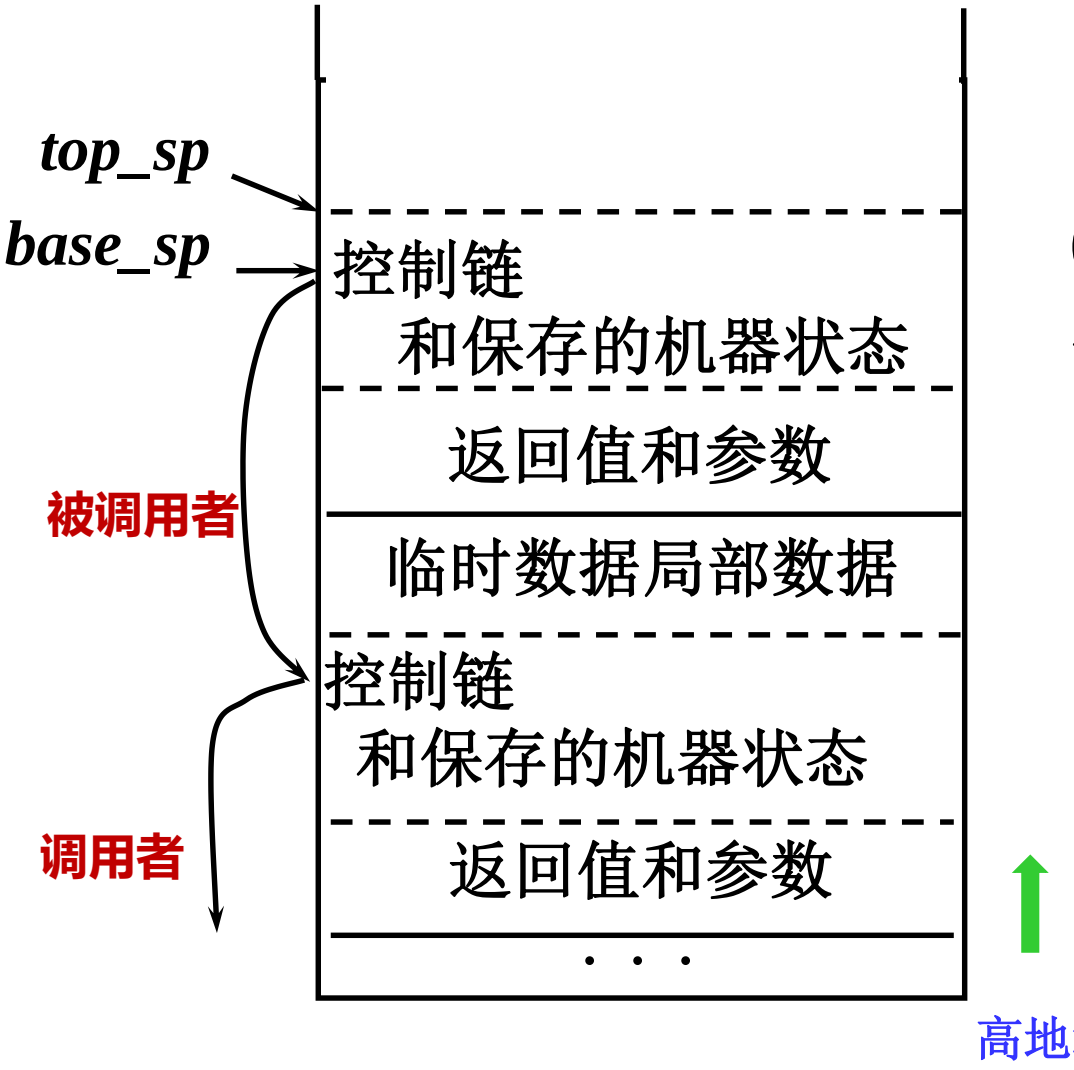
(1) p计算实参，依次放入栈顶，并在栈顶留出放返回值的空间。*top_sp*的值在此过程中被改变

过程调用序列：p调用q



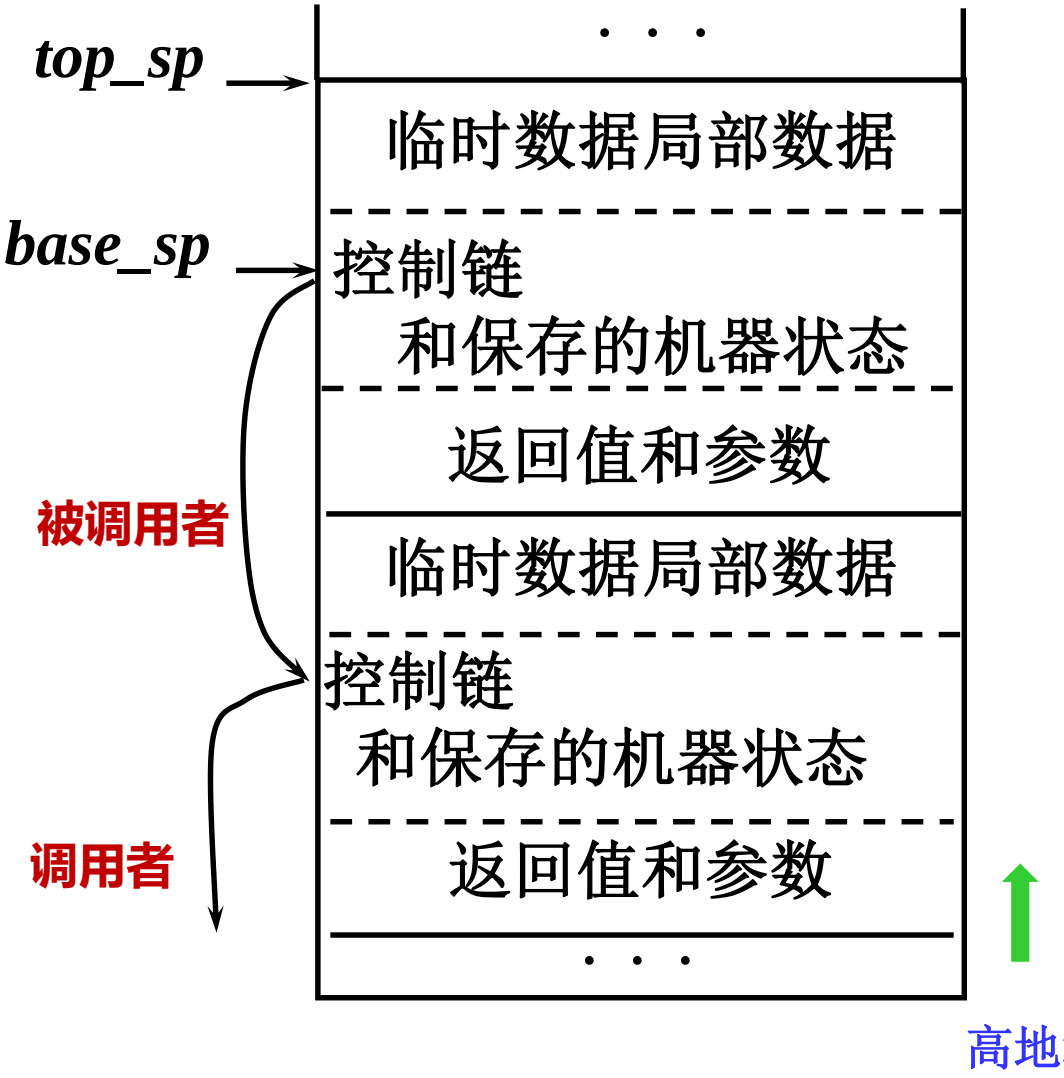
(2) p把返回地址和当前*base_sp*的值存入q的活动记录中，建立q的访问链，增加*base_sp*的值

过程调用序列：p调用q



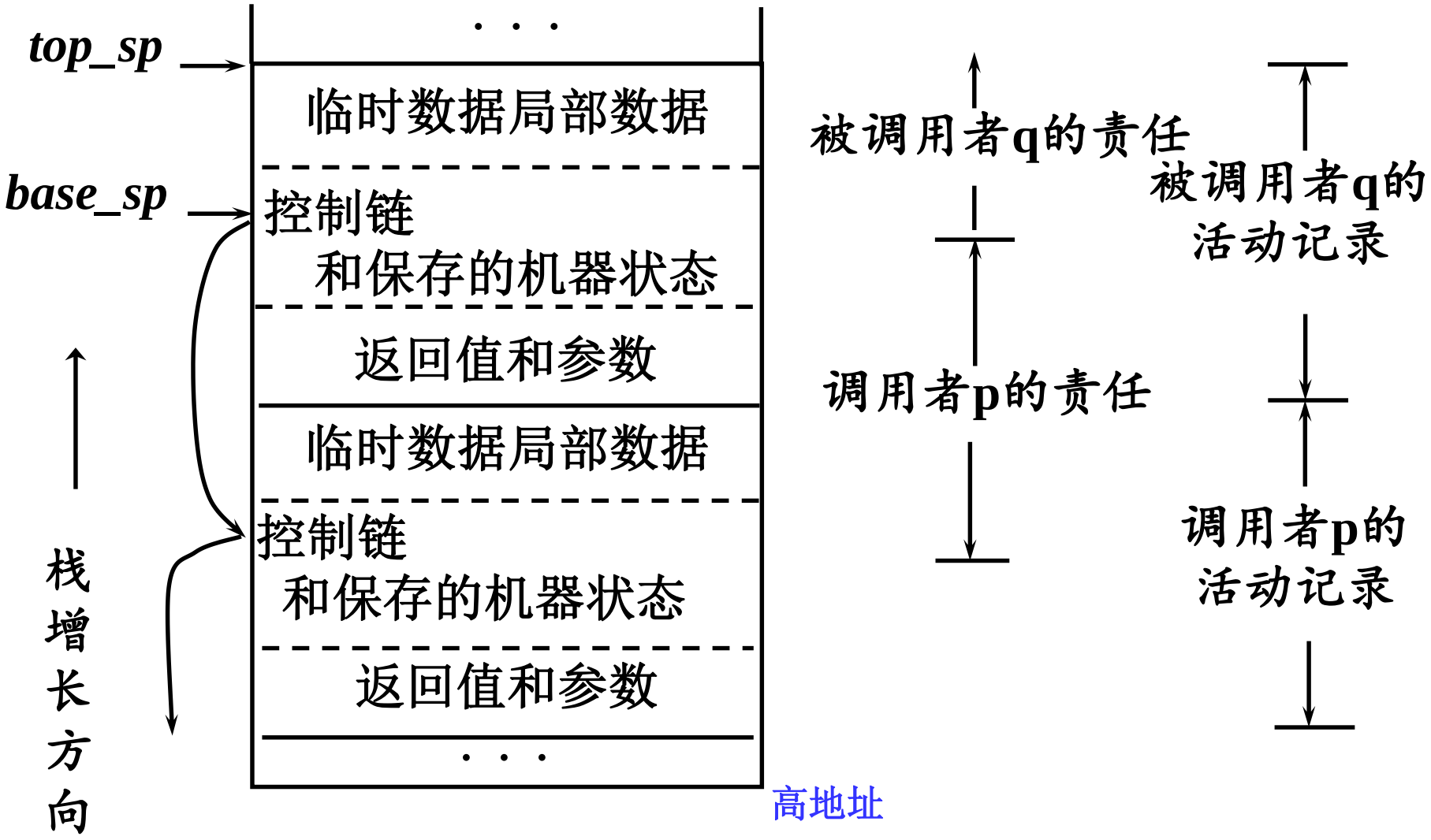
(3) q保存寄存器的值和其他机器状态信息

过程调用序列：p调用q

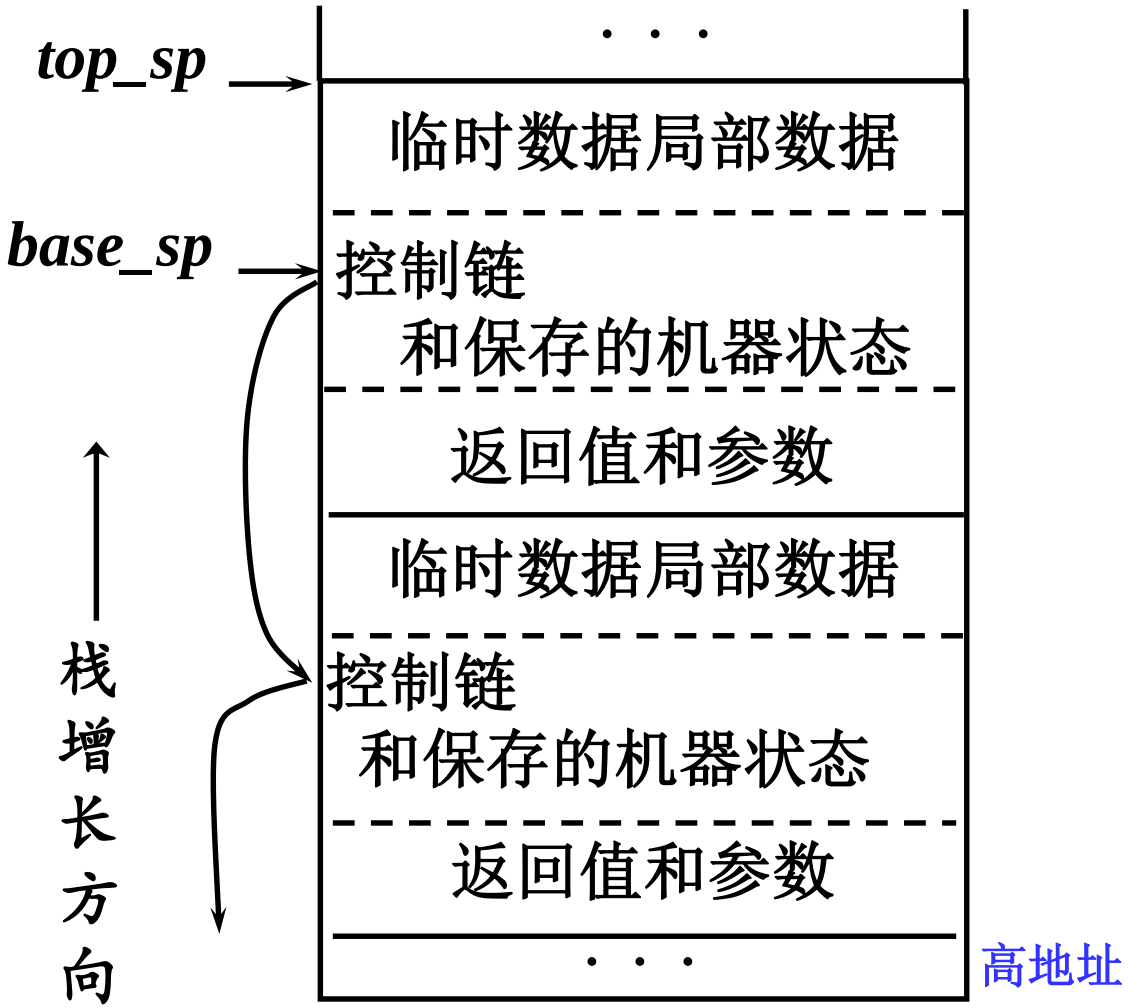


(4) q根据局部数据域和临时数据域的大小增加*top_sp*的值（**分配局部变量和临时数据的空间**），初始化它的局部数据，并开始执行过程体

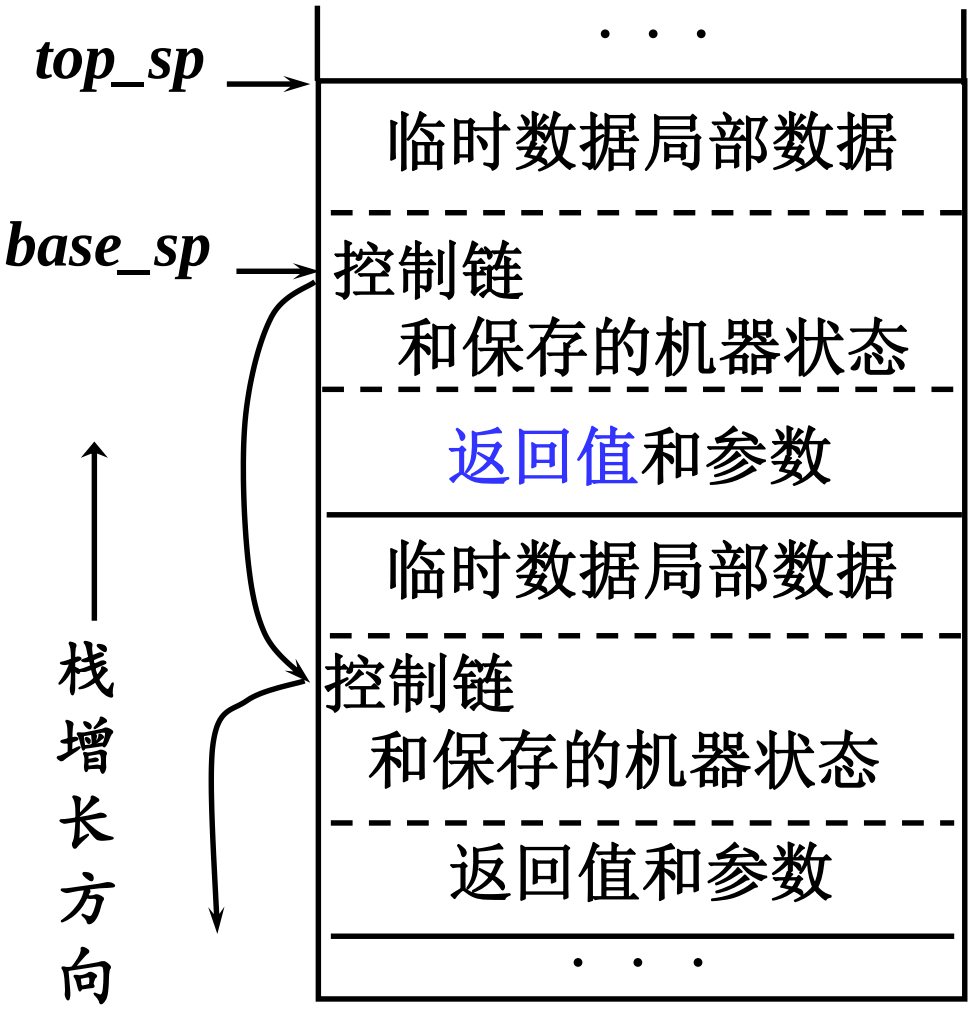
调用者p和被调用者q的任务划分



过程返回序列：p调用q


top_sp
base_sp
. . .
临时数据局部数据
控制链
和保存的机器状态
返回值和参数
临时数据局部数据
控制链
和保存的机器状态
返回值和参数
. . .
栈增长方向
高地址

过程返回序列：p调用q



(1) q把**返回值置入邻近调用者p**的活动记录的地方

参数个数可变场合难以确定存放返回值的位置，因此**通常用寄存器传递返回值**

通过寄存器传递返回值

- X86-32位系统
  - 32位整型返回值： eax
  - 64位整型返回值： 低32位 eax，高32位 edx
  - 浮点类型的返回值：浮点寄存器 st(0)
- X86-64位系统
  - 整型： rax
  - 浮点型：浮点寄存器 st(0)
- ARM 呢？
  - ATPCS：ARM-Thumb procedure call standard
    AAPCS：ARM Architecture procedure call standard， 2007，是ATPCS的改进版
  - 小于或等于4字节的：r0；
  - 双字：r0和r1；　　128位的向量通过r0~r3

通过寄存器传递参数

- **微软x86-64调用约定**

  使用RCX, RDX, R8, R9四个寄存器用于存储函数调用时的4个参数(从左到右)，使用XMM0, XMM1, XMM2, XMM3来传递浮点变量

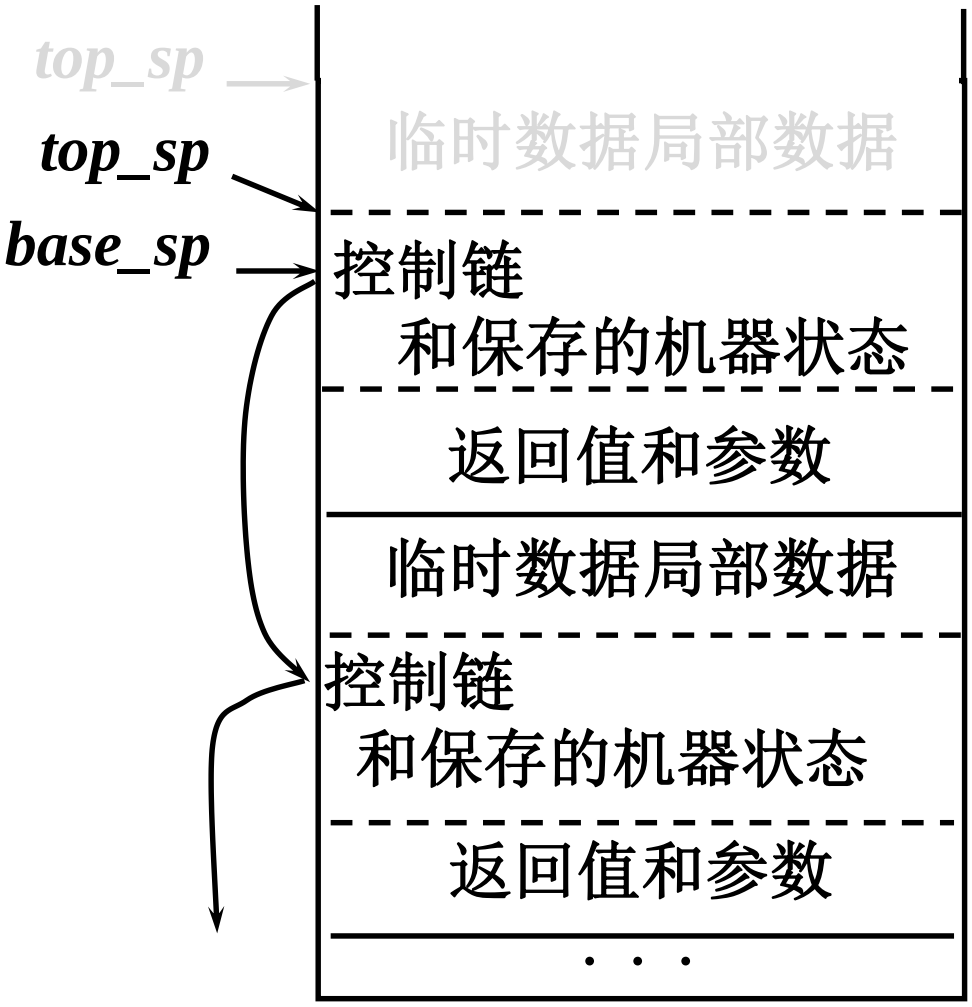
- **Linux等的64位系统调用约定**

  头六个整型参数放在寄存器RDI, RSI, RDX, RCX, R8和R9上；同时XMM0到XMM7用来放置浮点变量。对于系统调用，R10用来替代RCX

- **ARM：AAPCS**
  - **用r0~r3和栈传参**

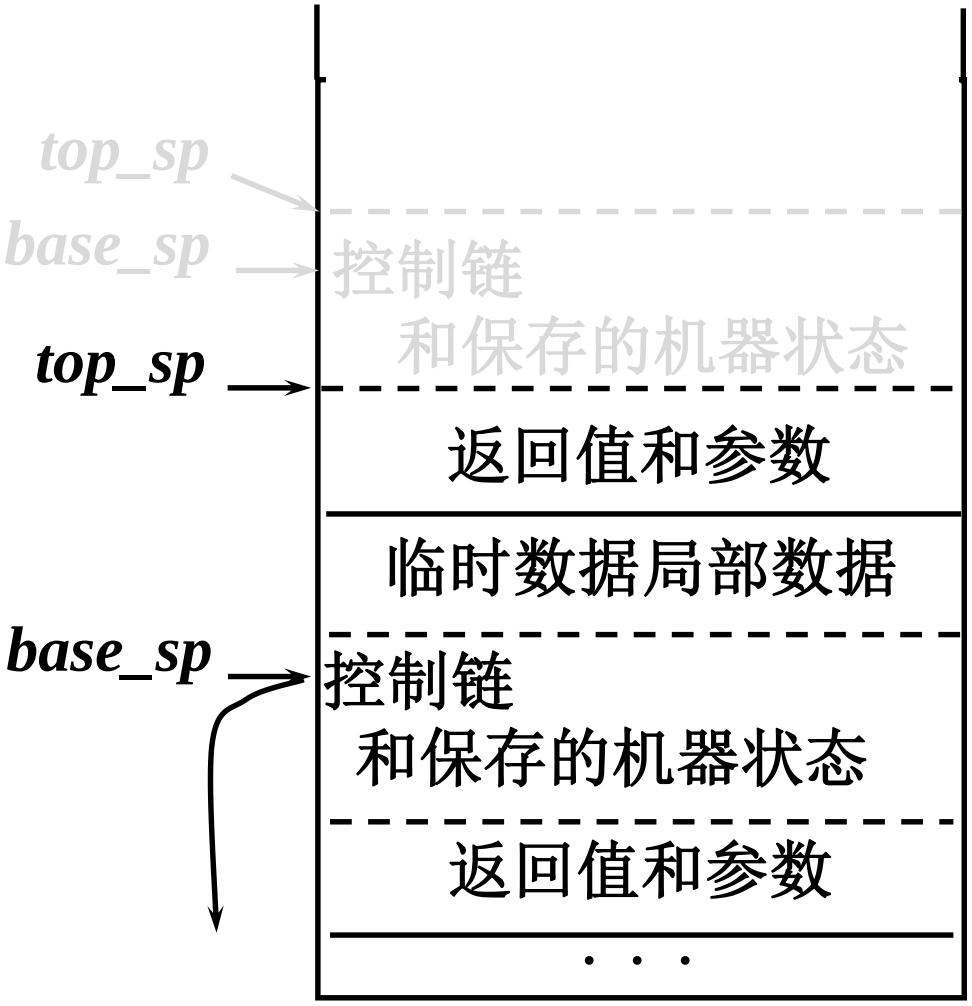
- **gcc 对整型和浮点型参数传递的汇编码生成特点分析**

过程返回序列：p调用q



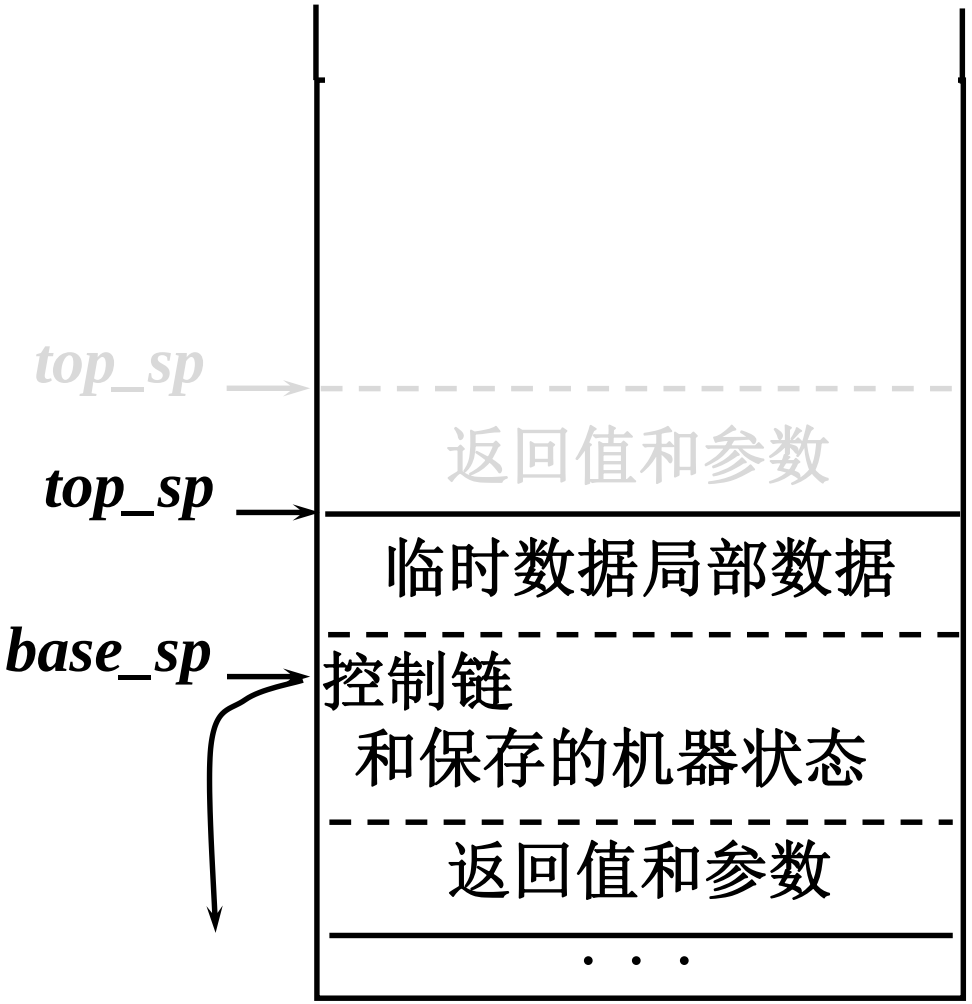
(2) q对应调用序列的步骤(4)，减小*top_sp*的值（**释放局部变量和临时数据的空间**）

过程返回序列：p调用q



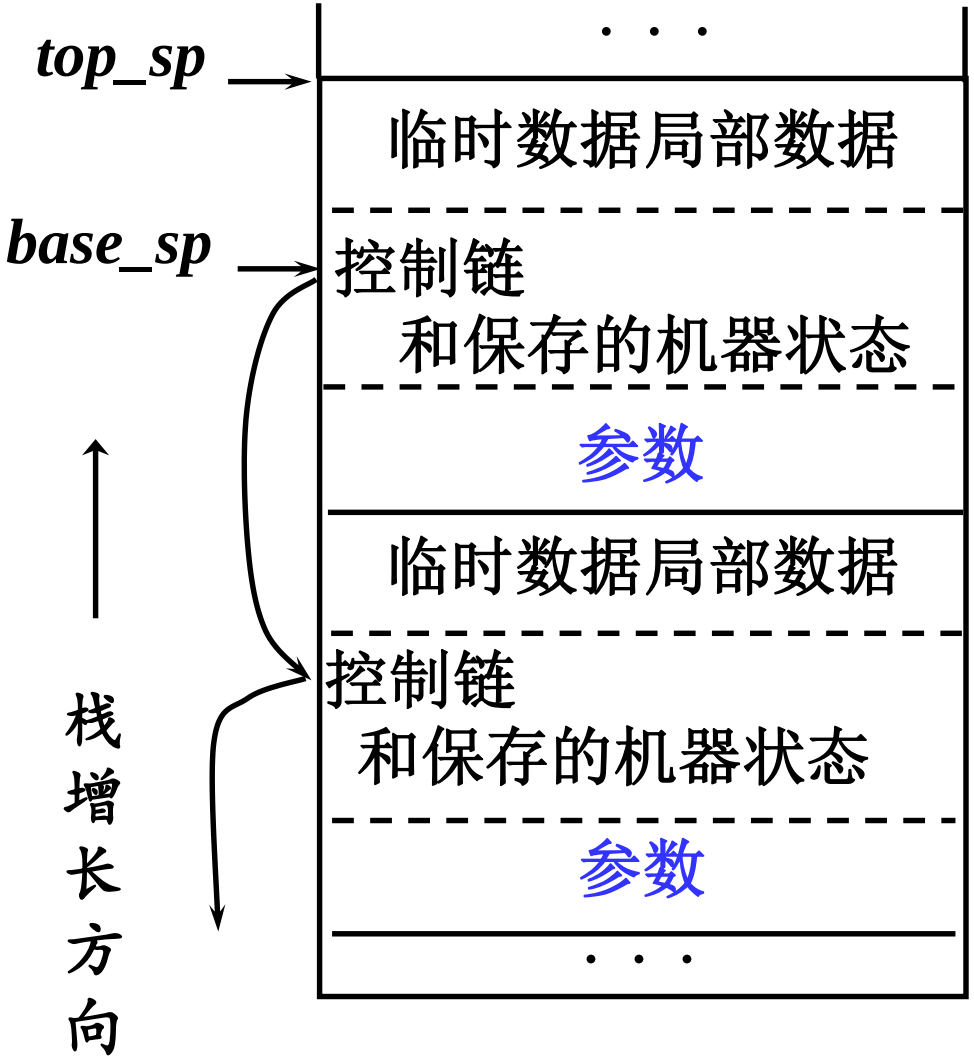
(3) q恢复寄存器(包括*base_sp*)和机器状态，返回p

过程返回序列：p调用q



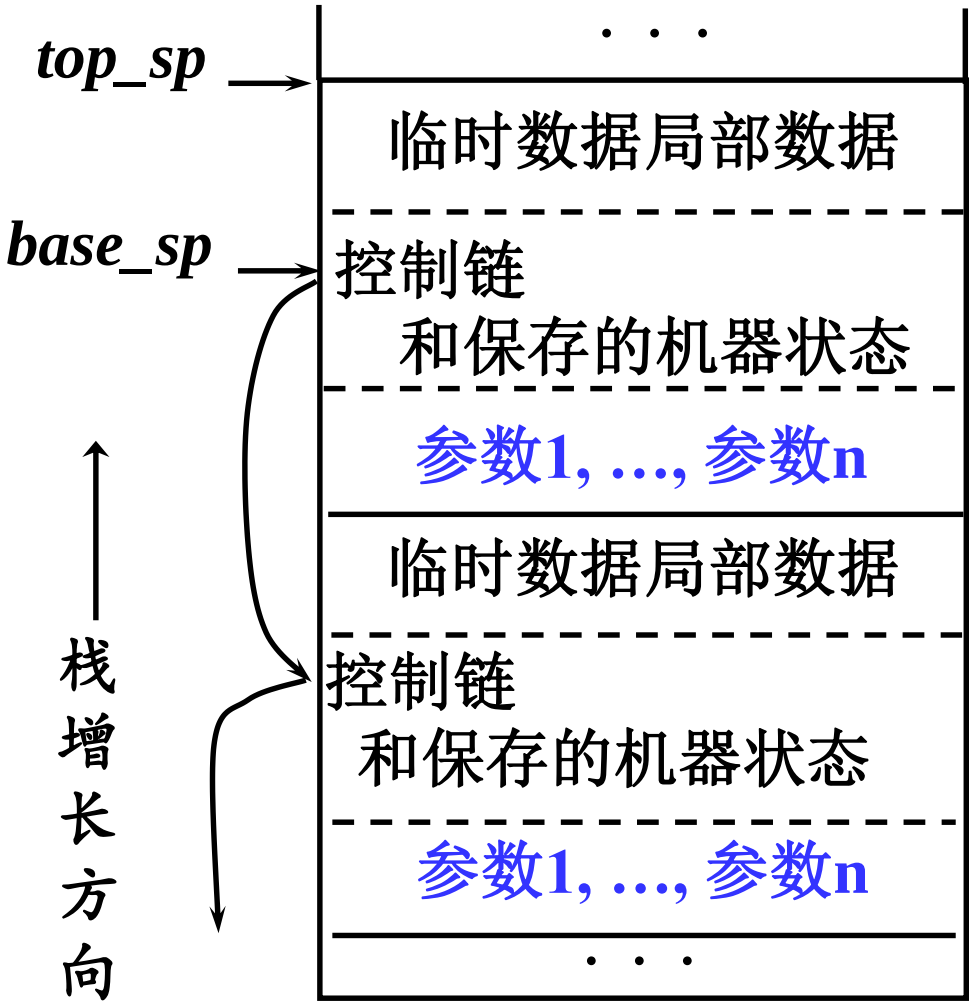
(4) p根据参数个数与类型和返回值调整 *top_sp*（**释放参数空间**）,然后**取出返回值**

过程的参数个数可变的情况



(1) 函数返回值改成**用寄存器传递**

过程的参数个数可变的情况



(2) 编译器产生将**实参表达式逆序计算**并将结果进栈的代码

自上而下依次是参数1, …, 参数n

(3)被调用者能准确地知道第一个参数的位置

(4)被调用函数根据第一个参数到栈中取第二、第三个参数等等

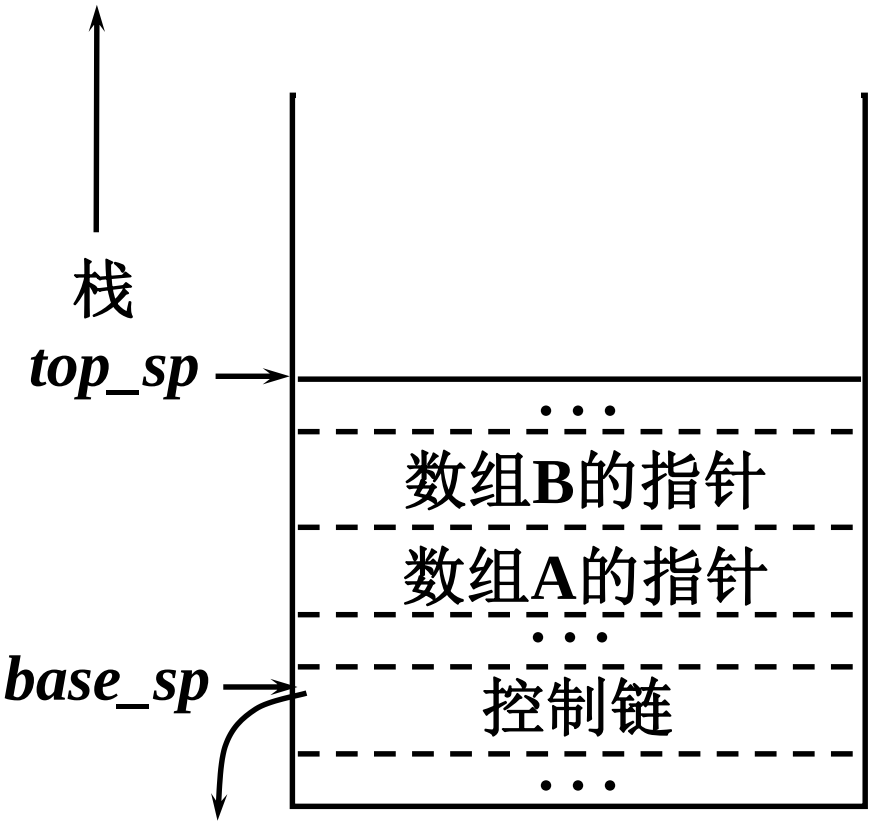
例：printf(“%d, %d,\n”);

栈上存储可变长的数据

- 可变长度的数组
  - C ISO/IEC9899：2011 n1570.pdf
    (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf)
    - 6.7.6.2(P132)： int a[n][6][m];
    - 6.10.8.3(P177)： __STDC_NO_VLA__宏为1时不支持可变长数组
    - 局部数组：在栈上分配
  - Java
    - int[ ] a = new int[n];
    - 在堆上分配
- 如何在栈上布局可变长的数组？
  - 先分配存放数组指针的单元，对数组的访问通过指针间接访问
  - 运行时，这些指针指向分配在栈顶的数组存储空间

栈上可变长数据

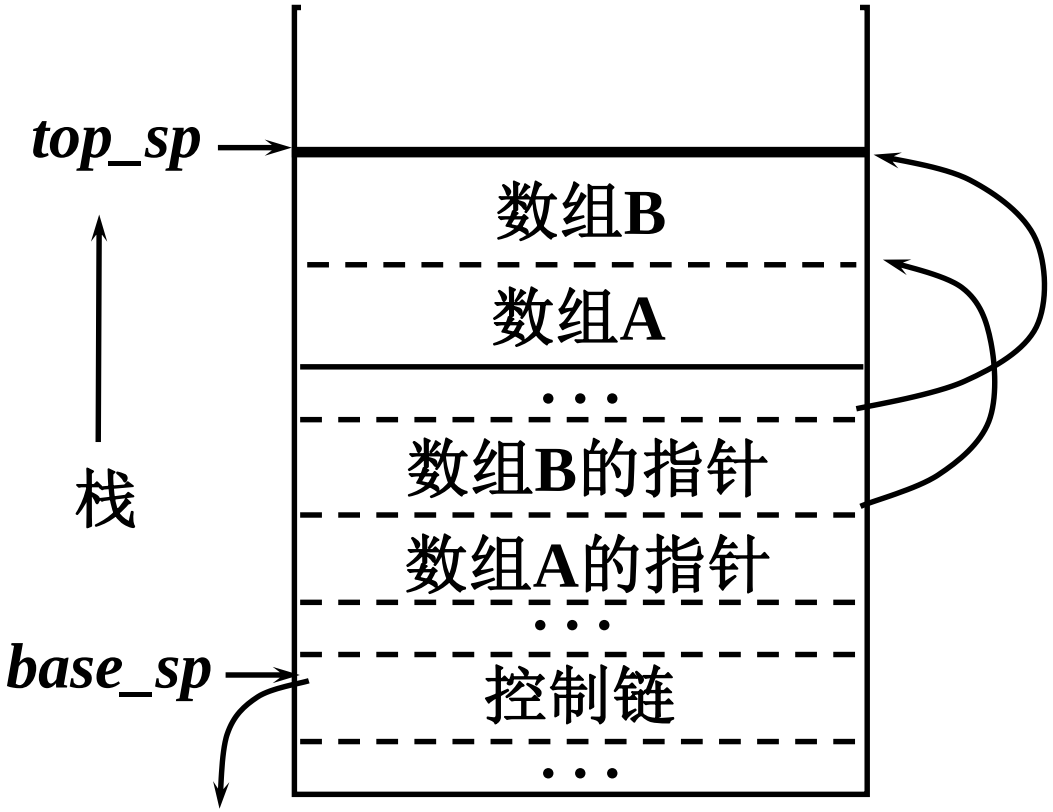
访问动态分配的数组



(1) 编译时，在活动记录中为这样的数组分配存放**数组指针**的单元

栈上可变长数据

- 访问动态分配的数组



(2) 运行时，这些指针指向分配在栈顶的数组存储空间（**数组实际空间在运行时分配**）

(3) 运行时，对数组A和B的访问都要**通过相应指针来间接访问**（数组访问指令是编译时生成）

悬空引用

- **悬空引用**：程序执行中的某个时刻，处于活动状态的某个指针变量或引用变量没有引用到合法的对象

```
T *p;
...
p = (T*)malloc(sizeof(T));
...
free(p);
... *p ... // 危险!
{
  T n = ...;
  p = &n;   // p生存期比n长
}
... *p ... // 危险!
```

```
T* fun(...) {
  T n;
  ...
  return &n;
}
... { ...
  T *p = fun(...);
  .. *p .. // 危险!
}
```

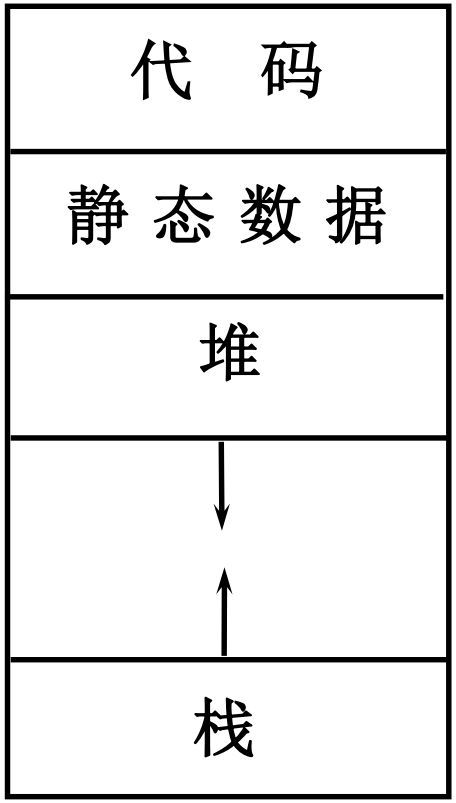
中国科学技术大学
University of Science and Technology of China

3. 非局部名字的访问

- 静态数据区、堆
- 静态作用域：无过程嵌套的（C）、有过程嵌套的（Pascal）
- 动态作用域（Lisp、JavaScript）

非局部数据的存储

- 静态数据区
  - 全局变量、静态局部变量
- 堆
  - C：malloc、free
    glibc 的ptmalloc, Doug Lea's dlmalloc
    高效的并发内存分配器 jemalloc,
    TBBmalloc,TCMalloc (gperftools)
  - Java: new、Garbage Collection
    Richard Jones's the Garbage Collection Page
  - JavaScript等动态类型绑定的语言
    变量的空间采用隐式的堆分配



静态作用域

- 无过程嵌套时，如C语言
  - 非静态的局部变量的访问：位于栈顶的活动记录，通过基址指针 *base_sp* 来访问
  - 过程体中的非局部引用、静态局部变量：直接用静态确定的地址（位于静态数据区中的数据）
  - 过程可以作为参数来传递，也可以作为结果来返回
  - 无须访问链
- 有过程嵌套时

  如Ada、JavaScript、Pascal、Python、Fortran 2003+
  - 需要构建访问链，并通过访问链访问在外围过程中声明的非局部名字

中国科学技术大学
University of Science and Technology of China

有过程嵌套的静态作用域

- 非局部名字的访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题

过程嵌套定义程序举例

图6.14， P186

sort

readarray

exchange

quicksort

partition

过程嵌套定义程序举例

图6.14， P186

- 过程嵌套深度

sort	1
readarray	2
exchange	2
quicksort	2
partition	3

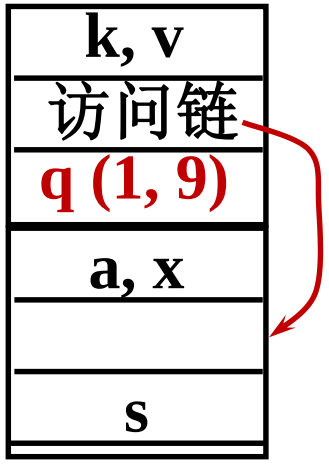
- 变量的嵌套深度：以它的声明所在的过程的嵌套深度作为该名字的嵌套深度

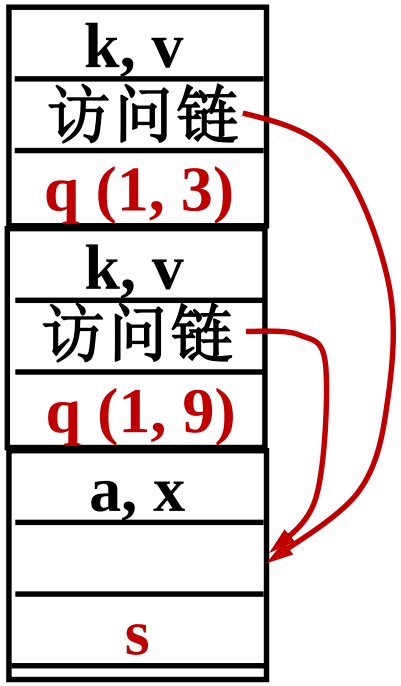
访问链

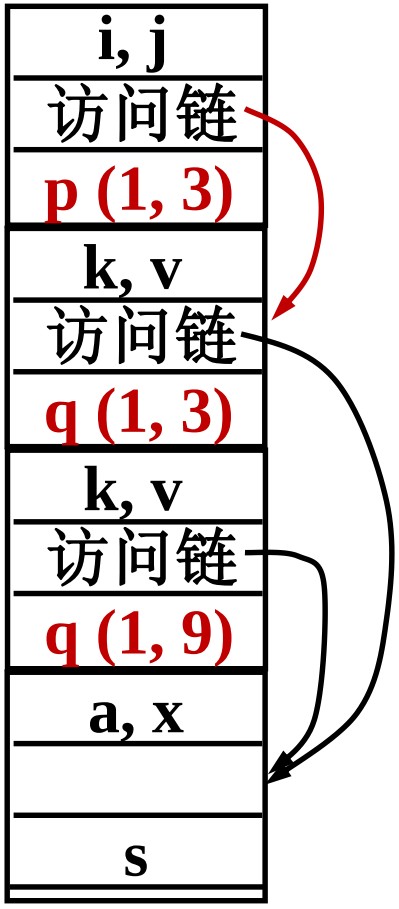
- 用来寻找非局部名字的存储单元

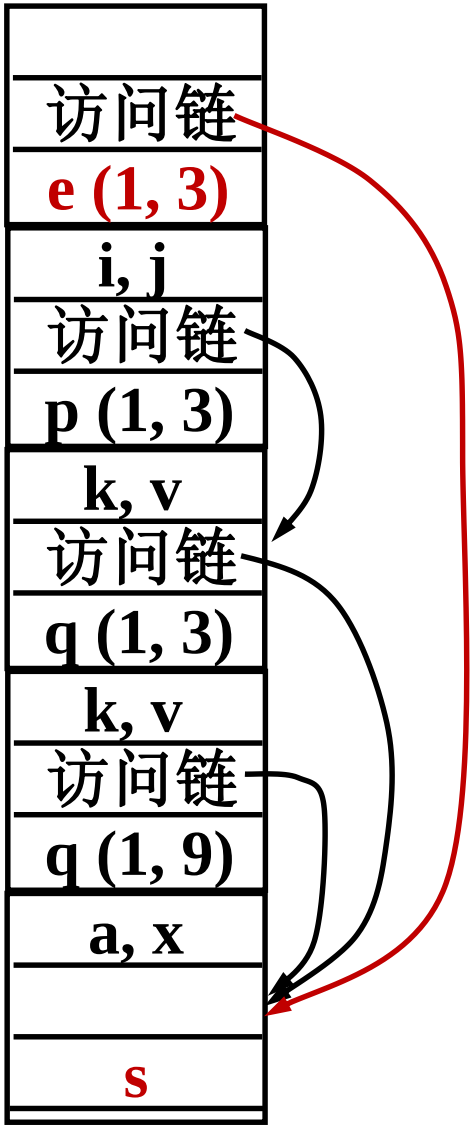
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

访问链反映过程之间的嵌套定义关系
控制链反映过程之间的调用关系









针对访问链的两个关键问题

- 通过访问链访问非局部引用

  假定过程p 的嵌套深度为n_p，它引用嵌套深度为n_a的变量a，$n_a \le n_p$，如何访问a的存储单元

- 访问链的建立

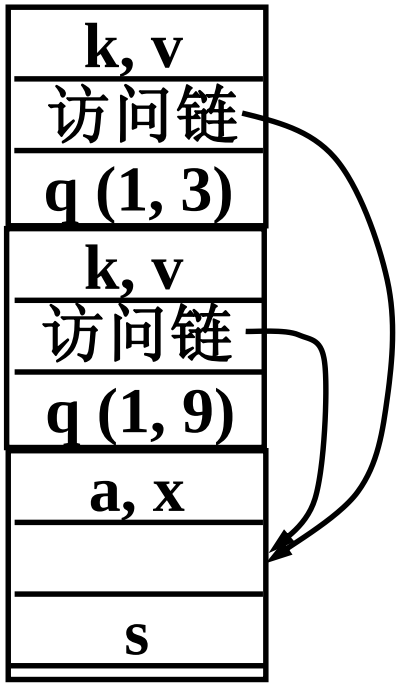
  假定嵌套深度为n_p的过程p调用嵌套深度为n_x的过程x，分别考虑 (1) $n_p < n_x$，(2) $n_p \ge n_x$ 的情况

通过访问链访问非局部引用

假定过程 p 的嵌套深度为 n_p，它引用嵌套深度为 n_a 的变量 a，$n_a \le n_p$，如何访问 a 的存储单元

- 从栈顶的活动记录开始，追踪访问链 $n_p - n_a$ 次
- 到达 a 的声明所在过程的活动记录
- 访问链的追踪用间接操作就可完成

```
sort              1
   readarray      2
   exchange       2
   quicksort      2
      partition   3
```



非局部名字引用的表示

过程p对变量a访问时，a的地址由下面的二元组表示：

（$n_p - n_a$，a在活动记录中的偏移）

访问链的建立

假定嵌套深度为n_p的过程 p调用嵌套深度为n_x的过程 x

1. $n_p < n_x$的情况　　这时 x 肯定就声明在p中

- 被调用过程的访问链须指向调用过程的活动记录的访问链
- sort调用quicksort、quicksort调用partition

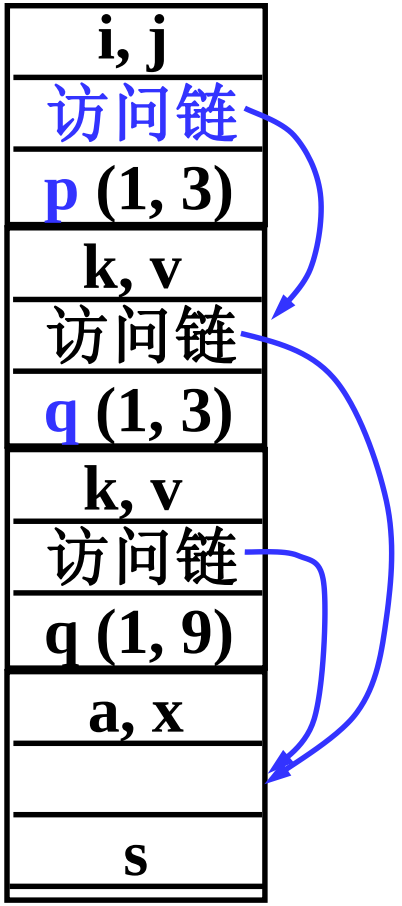
sort　　1

　　readarray　　2

　　exchange　　2

　　quicksort　　2

　　　　partition　　3

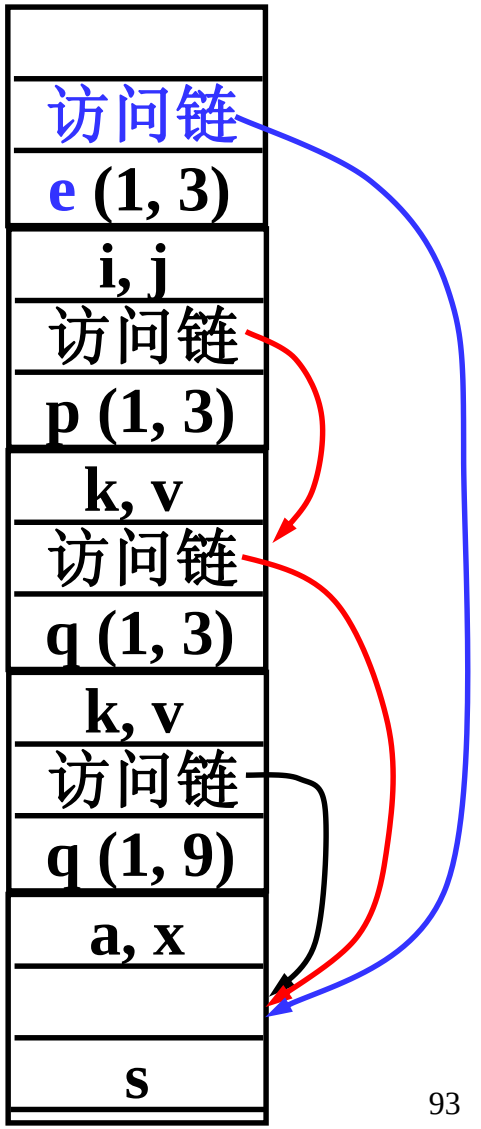


访问链的建立

2. $n_p \geq n_x$的情况　*p*和 *x* 有公共的外围过程

- 追踪访问链 $n_p - n_x + 1$次,到达静态包围x和p且离它们最近的那个过程的最新活动记录
- 所到达的活动记录就是*x*的活动记录中的访问链应该指向的那个活动记录
- partition调用exchange

```
sort                1
    readarray       2
    exchange        2
    quicksort       2
        partition   3
```



中国科学技术大学
University of Science and Technology of China

有过程嵌套的静态作用域

- 非局部名字的访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题

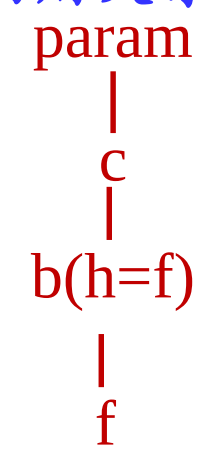
过程作为参数

```
program param(input, output);（过程作为参数）
    procedure b(function h(n: integer): integer);
        begin writeln(h(2)) end {b};
    procedure c;
        var m: integer;
        function f(n: integer): integer;
          begin f := m+n end {f};
        begin m := 0; b(f) end {c};
begin
    c
end.
```

静态的嵌套定义关系

param	1
b	2
c[m]	2
f	3

调用关系



要先理解每个过程和函数的类型

b: (integer→integer) →void

c: void →void

f: integer →integer

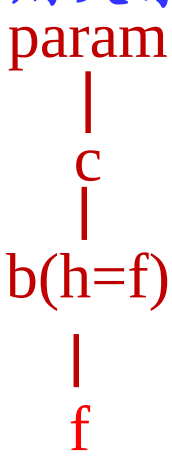
过程作为参数

```
program param(input, output);（过程作为参数）
    procedure b(function h(n: integer): integer);
        begin writeln(h(2)) end {b};
    procedure c;
        var m: integer;
        function f(n: integer): integer;
          begin f := m+n end {f};
        begin m := 0; b(f) end {c};
begin
    c
end.
```

静态的嵌套定义关系

param	1
b	2
c[m]	2
f	3

调用关系



作为参数传递时，怎样在 f 被激活时建立它的访问链，以便访问非局部名字m？

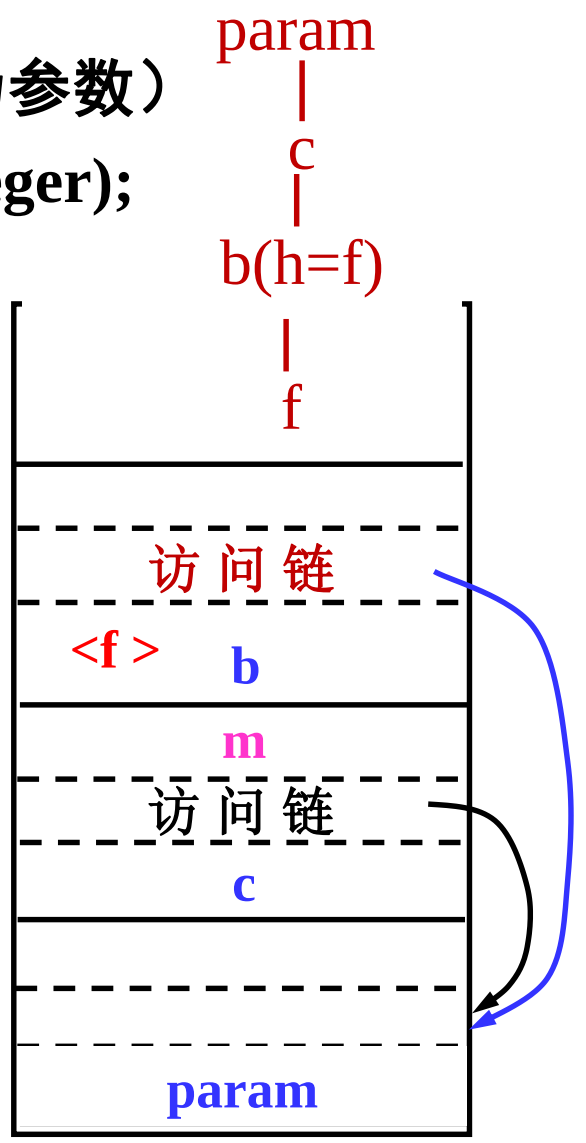
过程作为参数


```
program param(input, output);（过程作为参数）
    procedure b(function h(n: integer): integer);
        begin writeln(h(2)) end {b};
    procedure c;
        var m: integer;
        function f(n: integer): integer;
            begin f := m+n end {f};
        begin m := 0; b(f) end {c};
begin
    c
end.
```

b：(integer→integer)→void

param
  b
  c[m]
    f

从b的访问链难以建立f的访问链

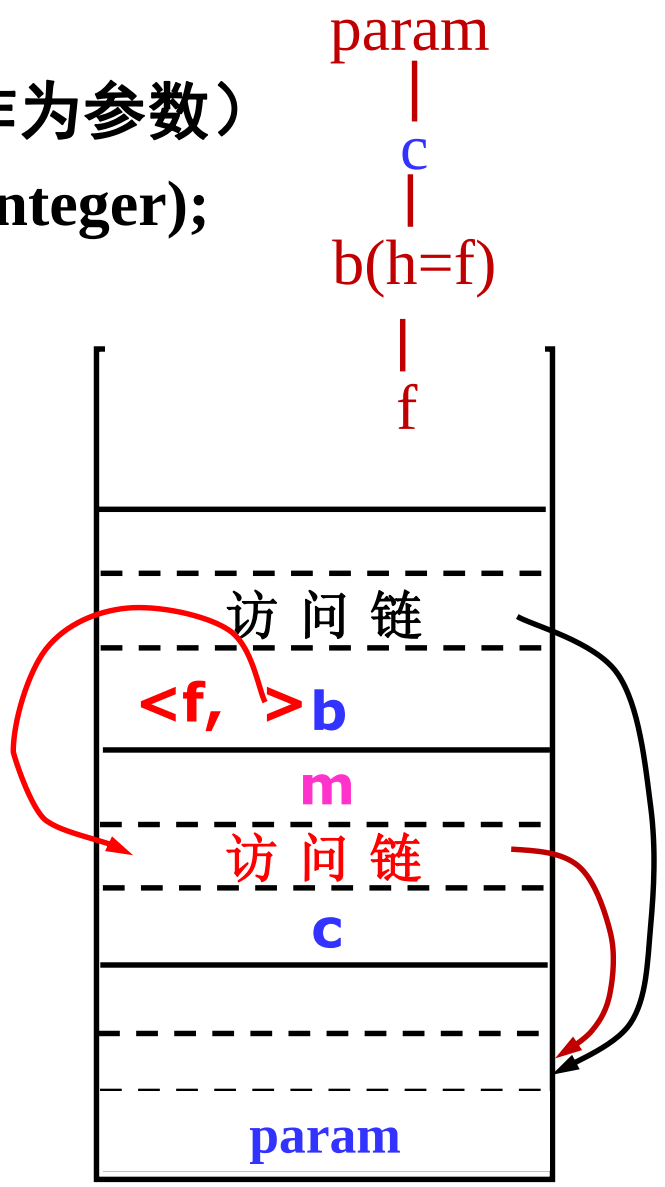


过程作为参数

```
program param(input, output);（过程作为参数）
    procedure b(function h(n: integer): integer);
        begin writeln(h(2)) end {b};
    procedure c;
        var m: integer;
        function f(n: integer): integer;
            begin f := m+n end {f};
        begin m := 0; b(f) end {c};
begin
    c
end.
```

param
b
c[m]
f

f作为参数传递时，它的起始地址连同它的访问链一起传递

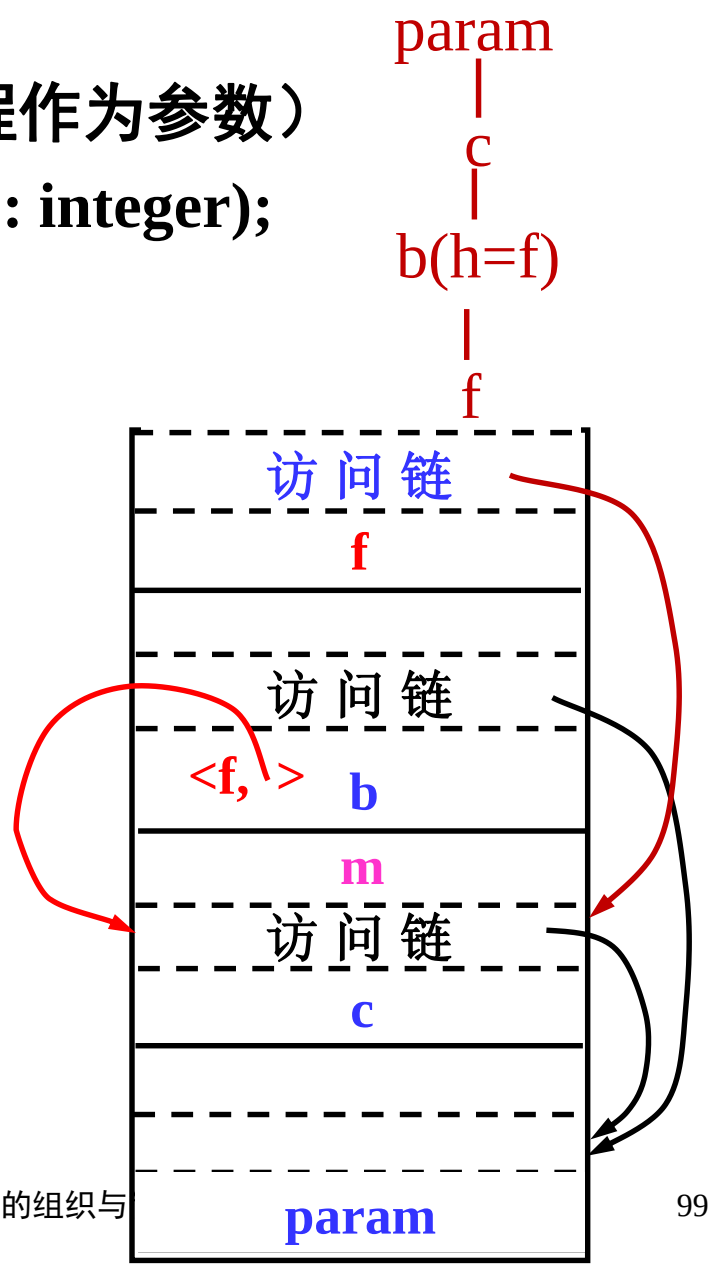


过程作为参数


```
program param(input, output);（过程作为参数）
    procedure b(function h(n: integer): integer);
        begin writeln(h(2)) end {b};
    procedure c;
        var m: integer;
        function f(n: integer): integer;
            begin f := m+n end {f};
        begin m := 0; b(f) end {c};
begin
    c
end.
```

param
b
c[m]
f

b调用f 时，用传递过来的访问链来建立f 的访问链



中国科学技术大学
University of Science and Technology of China

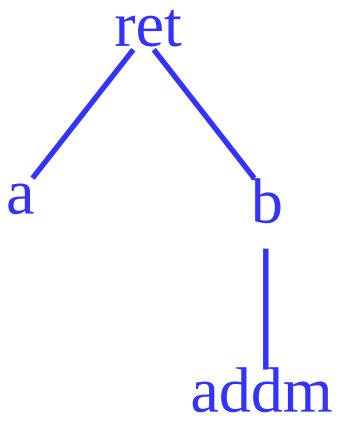
有过程嵌套的静态作用域

- 非局部名字的访问：访问链
- 过程作为参数产生的问题和解决
- 过程作为返回值产生的问题

过程作为返回值

```
program ret (input, output);（过程作为返回值）
    var f: function (integer): integer;
      function a: function (integer): integer;
         var m: integer;
        function addm (n: integer): integer;
           begin return m+n end;
        begin m:= 0; return addm end;
    procedure b (g: function (integer): integer);
        begin writeln (g(2)) end;
    begin
        f := a; b(f)
    end.
```

这里是对a的调用



要理解每个过程和函数的类型

a: integer→integer ?

addm: integer→integer

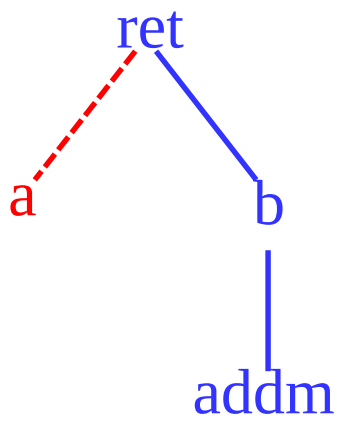
b: (integer→integer) →void

a: void →(integer→integer)

过程作为返回值

```
program ret (input, output);（过程作为返回值）
  var f: function (integer): integer;
    function a: function (integer): integer;
      var m: integer;
      function addm (n: integer): integer;
        begin return m+n end;
      begin m:= 0; return addm end;
  procedure b (g: function (integer): integer);
    begin writeln (g(2)) end;
  begin
    f := a; b(f)
  end.
```

执行addm时，a的活动记录已不存在，取不到m的值



C语言中的函数

- 不能嵌套定义
- 当前激活的函数要访问的数据分成两种情况
  - 非静态局部变量（包括形式参数）：分配在活动记录栈顶的那个活动记录中
  - 外部变量（包括定义在其它源文件之中的外部变量）和静态的局部变量：都分配在静态数据区
  - C语言允许函数（的指针）作为返回值

其他语言

- 采用静态作用域的语言
  - 无嵌套过程：如 C++、Java、C#
  - 嵌套过程：如 Python、JavaScript、Ruby
- 闭包 （closure）
  - 解决过程作为返回值时要面对的问题

```
function add(x) {
    return function(y) {
        return x + y;
    }
}
var add3 = add(3);  //add3是闭包对象，包含函数及其声明时的环境
alert(add3(4));
```

其他语言

闭包 （closure）

- 解决过程作为返回值时要面对的问题
- 过程作为参数时也存在相似的问题

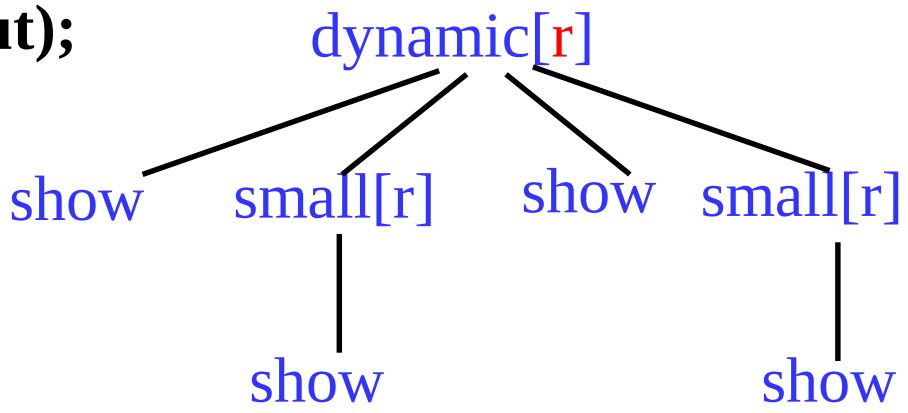
```
function do10times(fn)
   for i = 0,9 do
   fn(i)
   end
end
sum = 0
function addsum(i)
   sum = sum + i
end
do10times(addsum)
print(sum)
```

动态作用域

- 基于运行时的调用关系来确定非局部名字引用的存储单元
- 过程调用时，仅为被调用过程的局部名字建立新的绑定（在活动记录中）
- 实现动态作用域的方法
  - 深访问、浅访问

示例：基于静态作用域时


```
program dynamic(input, output);
   var r: real;
   procedure show;
      begin write(r: 5: 3) end;
   procedure small;
      var r: real;
      begin r := 0.125; show end;
   begin
      r := 0.25;
      show; small; writeln;
      show; small; writeln
   end.
```



show在dynamic中定义，show中访问的r 是dynamic中定义的

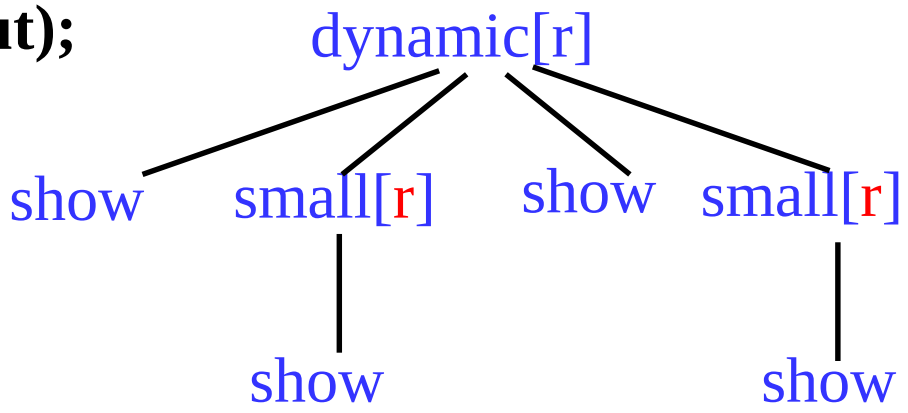
执行后输出：

0.250 0.250

0.250 0.250

示例：基于动态作用域时


```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```



dynamic中调用的show所访问的r 是dynamic中定义的；small中调用的show所访问的r是small中定义的

执行后输出：

0.250 0.125

0.250 0.125

动态作用域

- 使用动态作用域的语言
  - Pascal、Emacs Lisp、Common Lisp(兼有静态作用域)、Perl（兼有静态作用域）、Shell语言（bash, dash, PowerShell）
- 其他
  - 宏展开

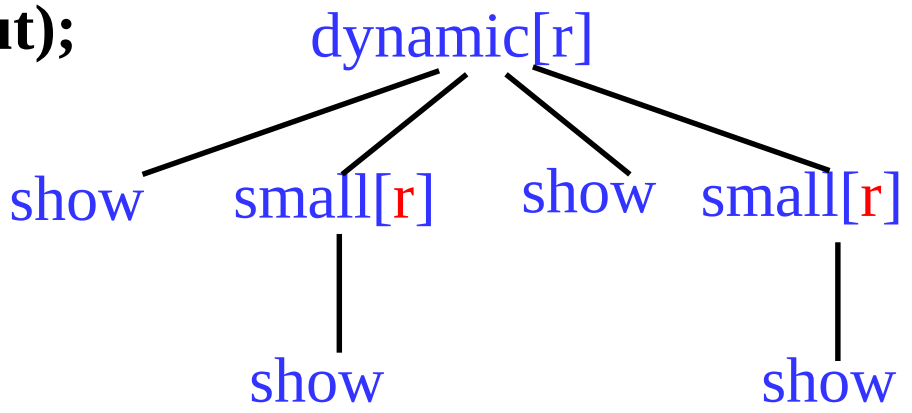
https://en.wikipedia.org/wiki/Scope_(computer_science)

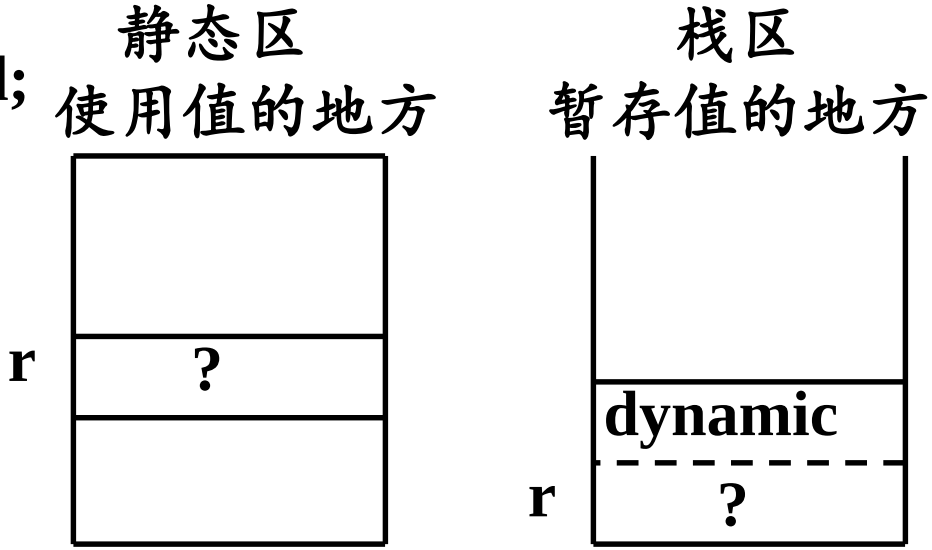
实现动态作用域的方法

- 深访问
  - 用控制链搜索运行栈，寻找包含该非局部名字的第一个活动记录
- 浅访问
  - 为每个名字在静态分配的存储空间中保存它的当前值
  - 当过程p的新活动出现时，p的局部名字*n*使用在静态数据区分配给*n*的存储单元。*n*的先前值保存在p的活动记录中，当p的活动结束时再恢复

基于浅访问实现动态作用域

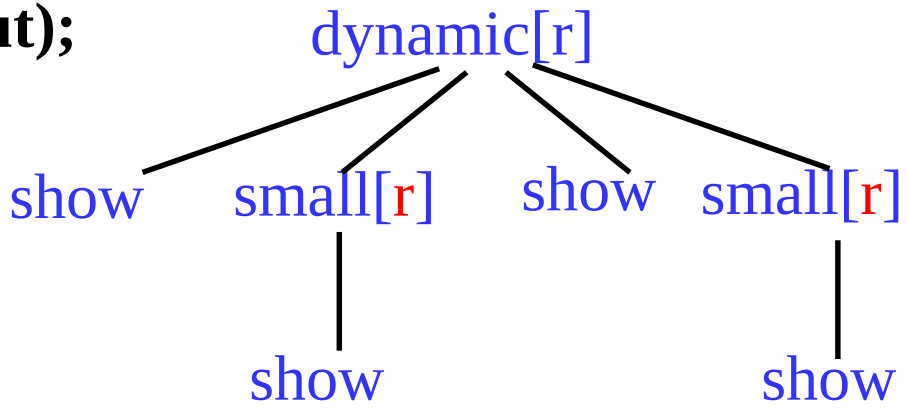
```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
begin(绿色表示已执行部分)
    r := 0.25;
    show; small; writeln;
    show; small; writeln
end.
```

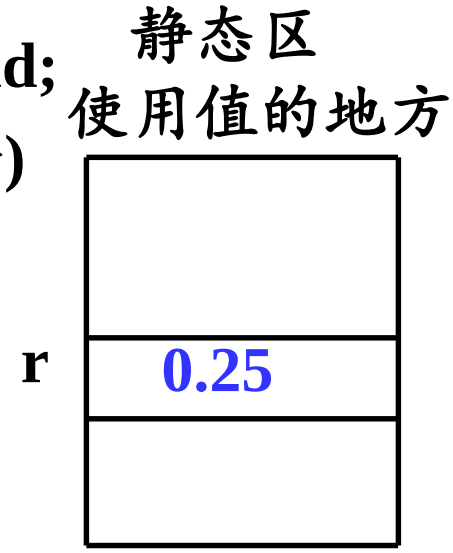


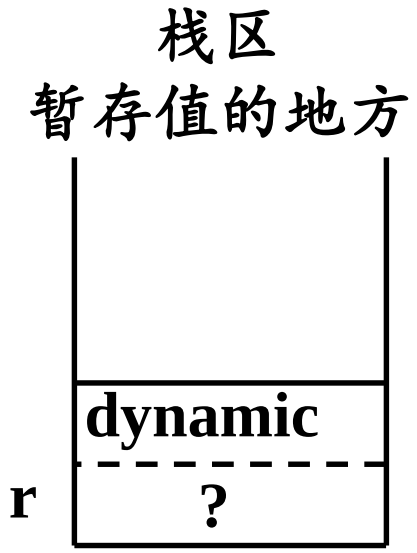


基于浅访问实现动态作用域

```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin(绿色表示已执行部分)
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```

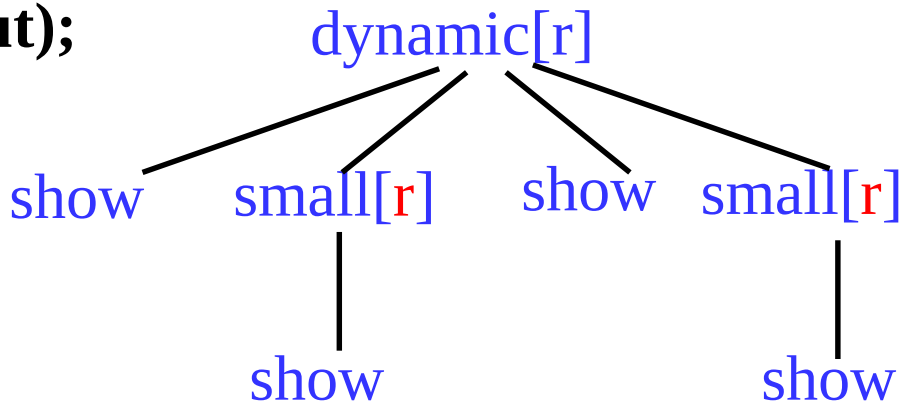


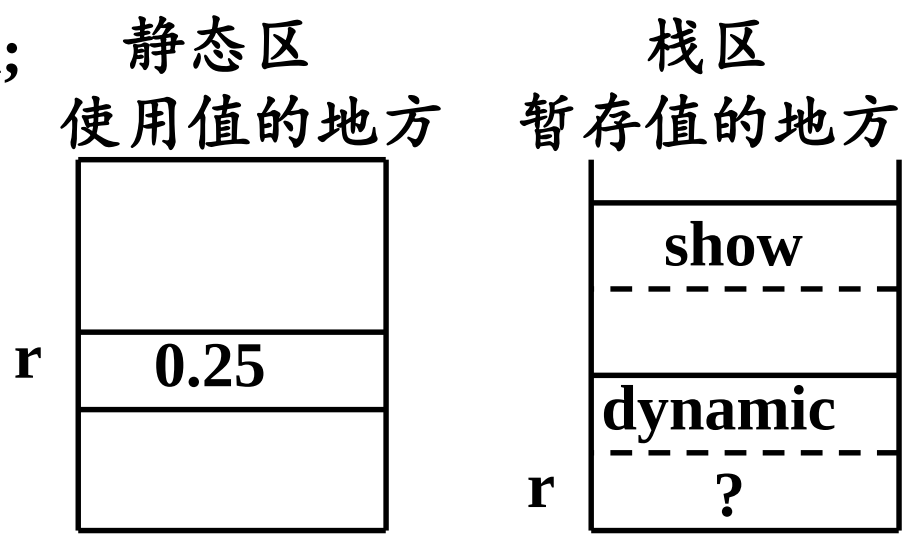




基于浅访问实现动态作用域

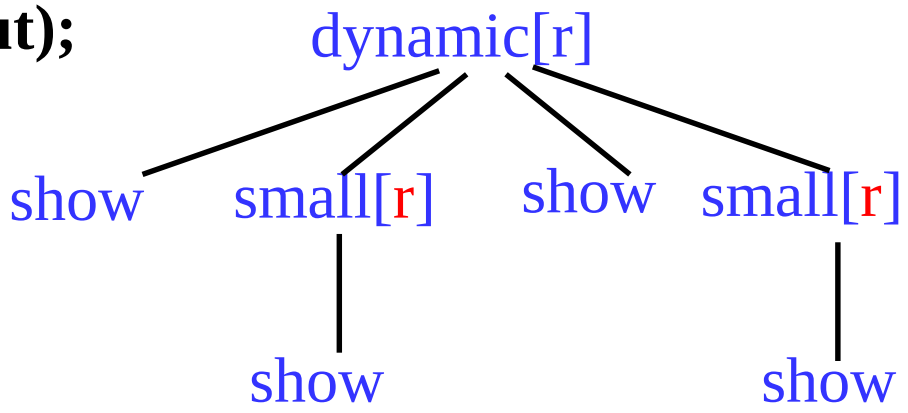
```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin(绿色表示已执行部分)
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```

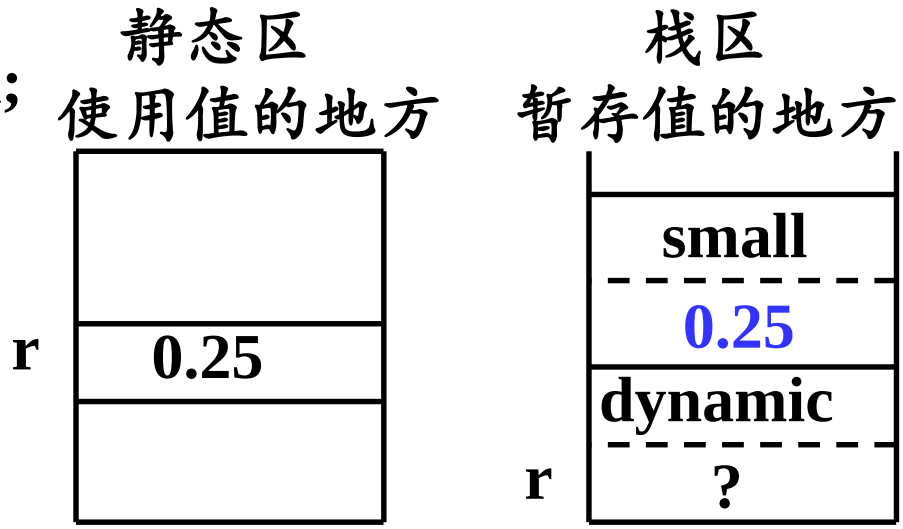




基于浅访问实现动态作用域

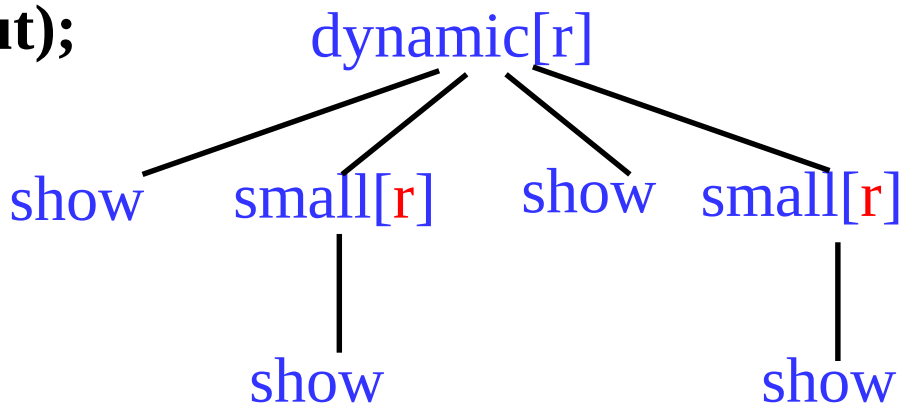
```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin(绿色表示已执行部分)
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```

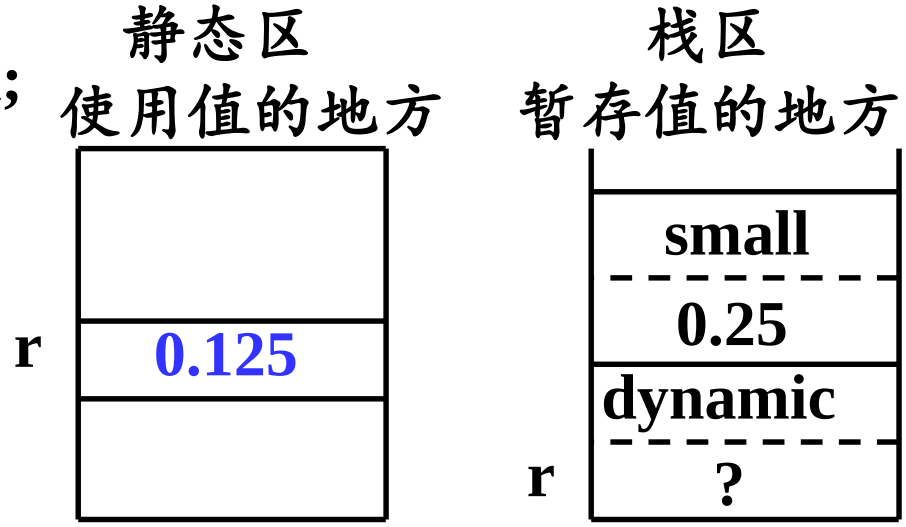




基于浅访问实现动态作用域

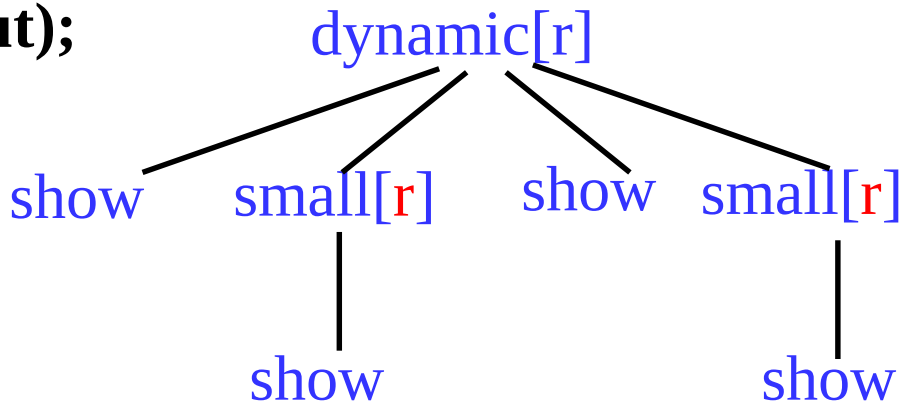
```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin(绿色表示已执行部分)
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```

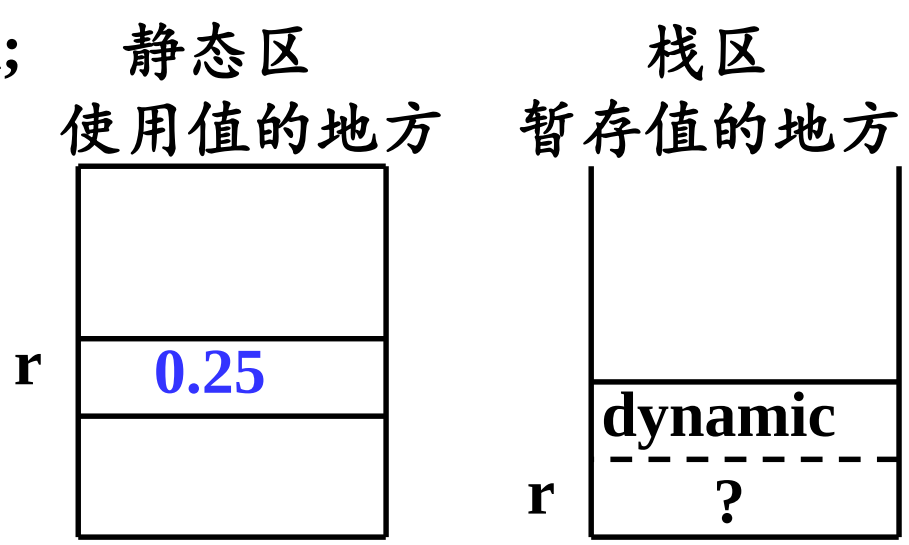




基于浅访问实现动态作用域

```
program dynamic(input, output);
    var r: real;
    procedure show;
        begin write(r: 5: 3) end;
    procedure small;
        var r: real;
        begin r := 0.125; show end;
    begin(绿色表示已执行部分)
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```





中国科学技术大学
University of Science and Technology of China

4. 参数传递

- 值调用
- 引用调用
- 换名调用

值调用(call by value)

- 特点
  - 实参的右值传给被调用过程
- 值调用的可能实现方法
  - 把形参当作所在过程的局部名看待，形参的存储单元在该过程的活动记录中
  - 调用过程计算实参，并把其右值放入被调用过程形参的存储单元中

引用调用(call by reference)

- 特点
  - 实参的左值传给被调用过程
- 引用调用的可能实现方法
  - 把形参当作所在过程的局部名看待，形参的存储单元在该过程的活动记录中
  - 调用过程计算实参，把实参的左值放入被调用过程形参的存储单元
  - 在被调用过程的目标代码中，任何对形参的引用都是通过传给该过程的指针来间接引用实参

换名调用(call by name)

特点

- 用实参表达式对形参进行正文替换,然后再执行

```
procedure swap(var x, y: integer);
    var temp: integer;
      begin
        temp := x;
        x := y;
        y := temp
      end
```

例如：调用swap(i, a[i])

替换结果：

```
temp := i;
i := a[i];
a[i] := temp
```

交换两个数据的程序并非总是正确

中国科学技术大学
University of Science and Technology of China

5. 其他

- 堆：分配与回收
- 计算机内存分层
- 局部性：时间、空间

堆管理

堆

存放生存期不确定的数据

- C：malloc、free

  glibc 的ptmalloc, Doug Lea’s dlmalloc

  高效的并发内存分配器 jemalloc,

  TBBmalloc,TCMalloc (gperftools)

- Java: new、Garbage Collection

  Richard Jones's the Garbage Collection Page

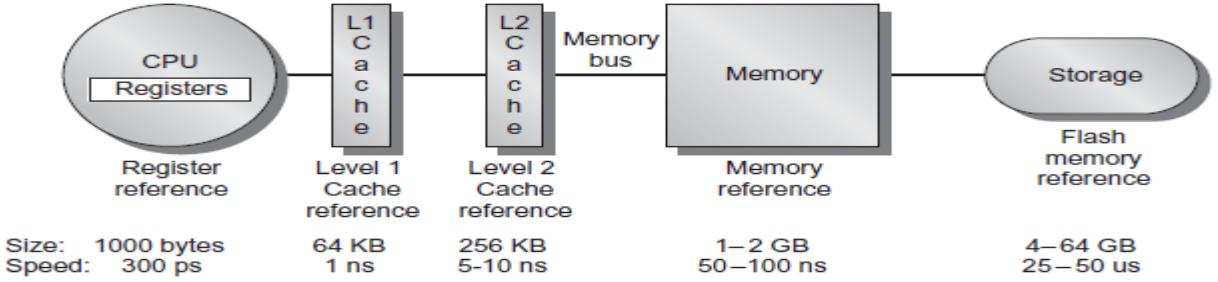
内存管理器

- 内存管理器,也称内存分配器(allocator)
  - 维护的基本信息：堆中空闲空间、…
  - 重点要实现的函数：分配、回收
- 内存管理器应具有下列性质
  - 空间有效性：极小化程序需要的堆空间总量
  - 程序有效性：较好地利用内存子系统，使得程序能运行得快一些
  - 低开销：分配和回收操作所花时间在整个程序执行时间中的比例尽量小

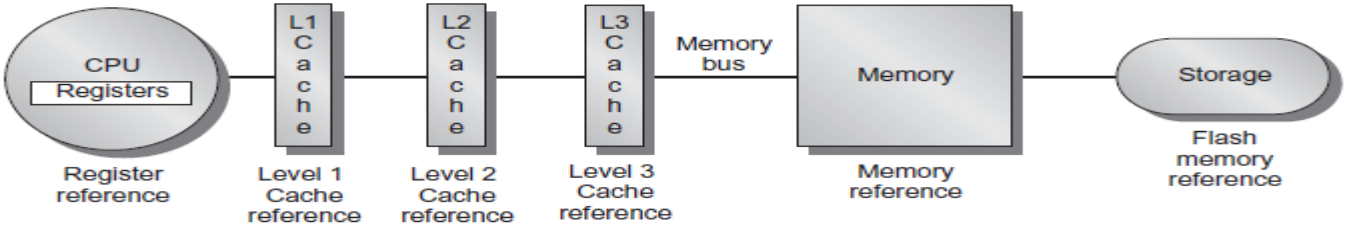
计算机内存分层

https://item.jd.com/12553439.html 第6版

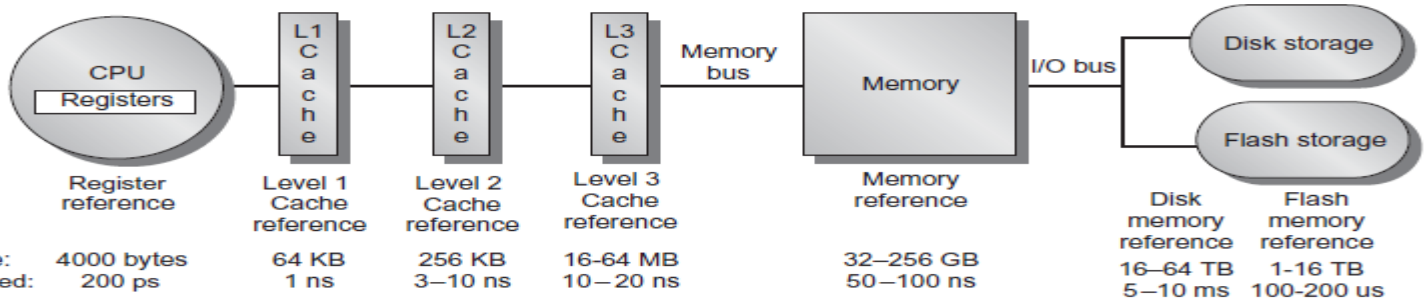
Computer Architecture, Sixth Edition: A Quantitative Approach, 2019



(A) Memory hierarchy for a personal mobile device

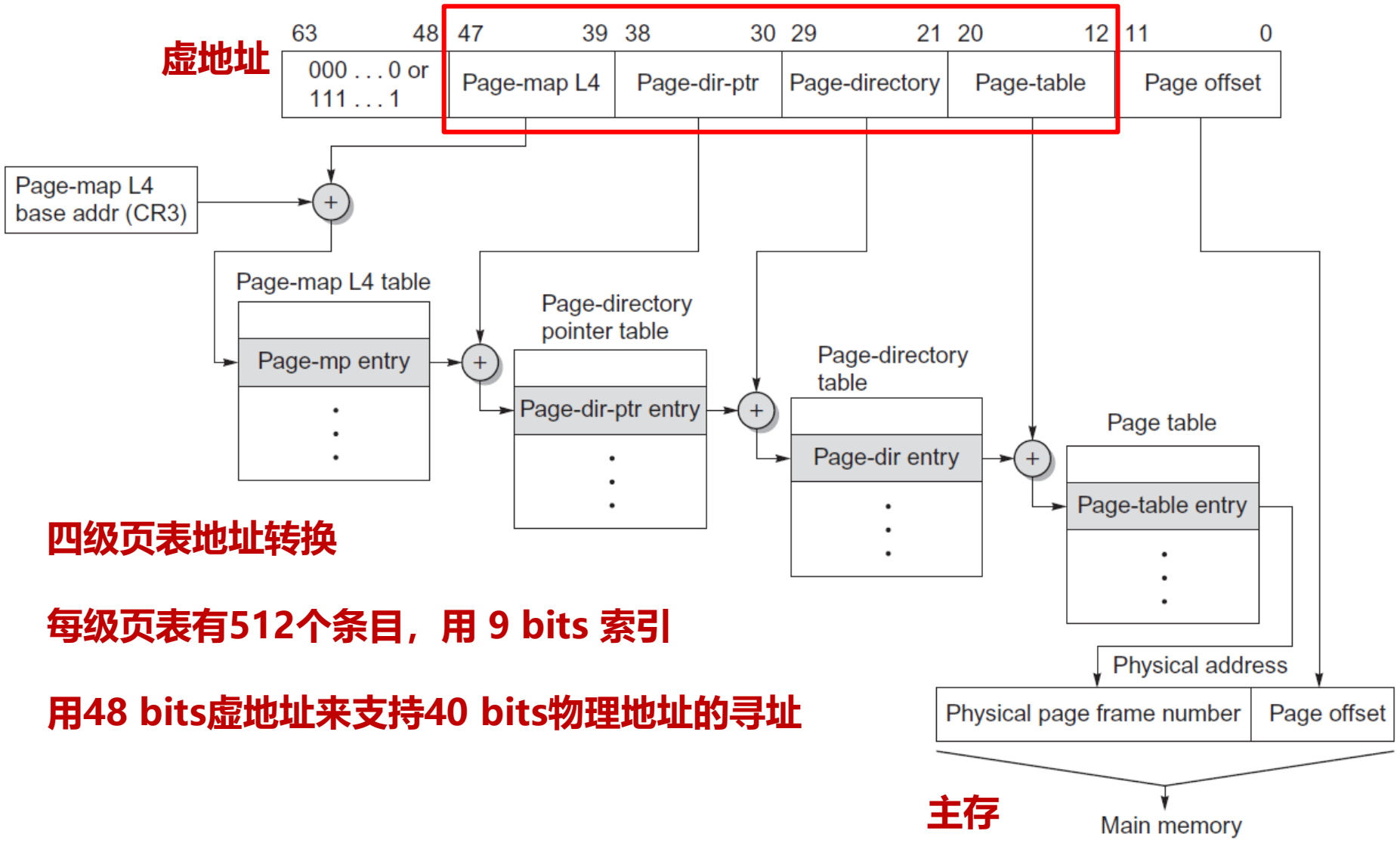


(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

AMD Opteron 皓龙虚地址映射



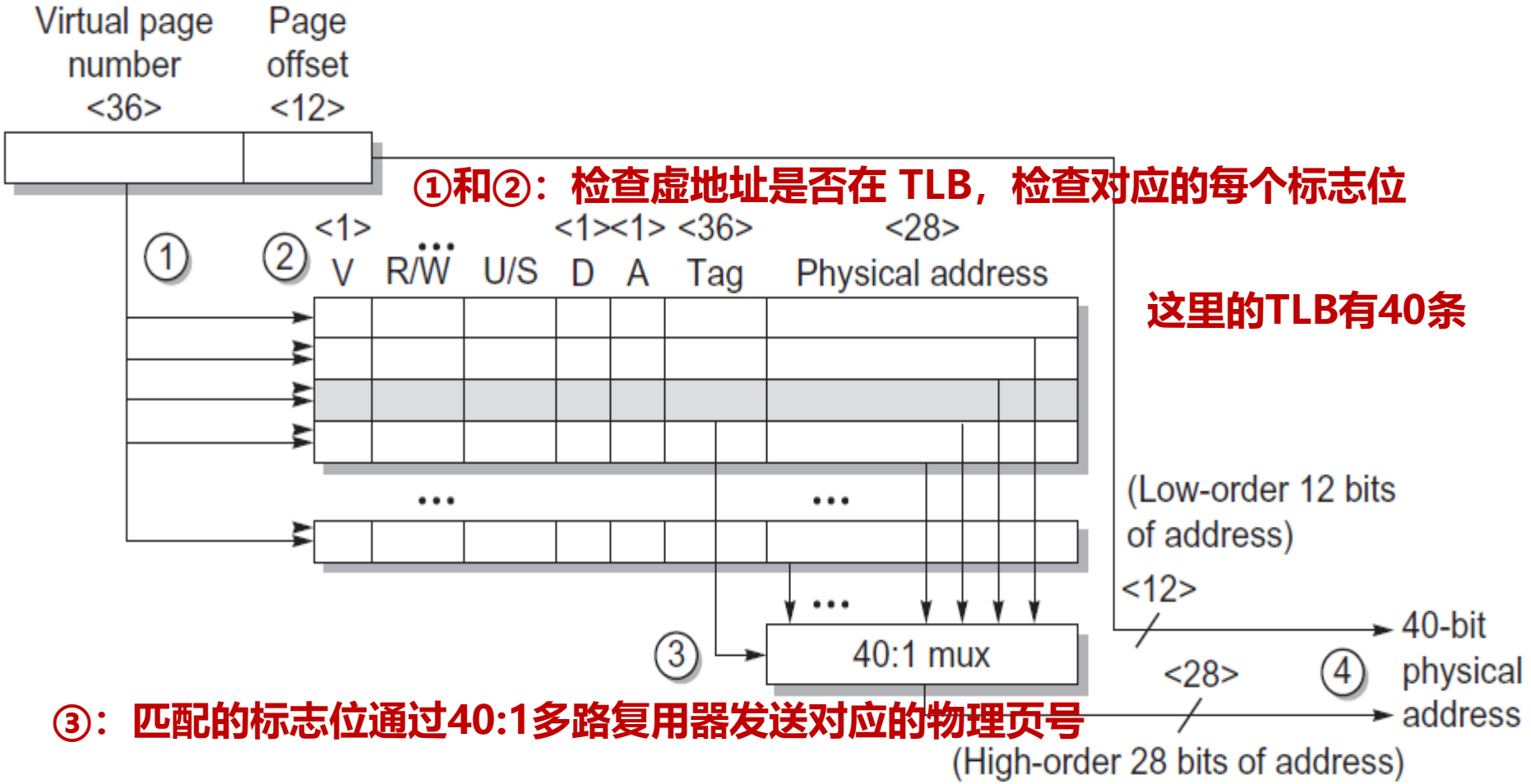
四级页表地址转换

每级页表有512个条目，用 9 bits 索引

用48 bits虚地址来支持40 bits物理地址的寻址

Opteron数据TLB的操作

- 快表（TLB, Translation look aside buffer）



Intel Core i7

峰值内存带宽：25 GB/s；使用48位虚拟地址、36位物理地址；两级 TLB

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Size	128	64	512
Associativity	4-way	4-way	4-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	6 cycles
Miss	7 cycles	7 cycles	Hundreds of cycles to access page table

TLB 结构

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

三级 cache 结构

程序局部性(locality)

大多数程序的大部分时间在执行一小部分代码，并且仅涉及一小部分数据

- **时间局部性(temporal locality)**

  程序访问的内存单元在很短的时间内可能再次被程序访问

- **空间局部性(spatial locality)**

  毗邻被访问单元的内存单元在很短的时间内可能被访问

- **有效利用缓存**
  - Cache容量有限、最近使用的指令保存在cache中
  - 改变数据布局或计算次序=>改进程序局部性

举例: 改变计算次序

```
void copyij (int src[2048][2048],
             int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji (int src[2048][2048],
             int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

上述两个函数功能、算法一样，但执行时间一样吗?

```
int a[2048][2048] = {1, 1};
int b[2048][2048];
int main( )
{
  copyij(a, b);
  // copyji(a, b);
}
```

```
$ time ./copyij
real    0m0.046s
user    0m0.037s
sys     0m0.008s
```

```
$ time ./copyji
real    0m0.404s
user    0m0.384s
sys     0m0.020s
```

举例: 改变计算次序


```
int a[2048][2048] = {1, 1};
int b[2048][2048];
int main( )
{
  copyij(a, b);
  // copyji(a, b);
}
```

```
int a[2048][2048] = {1, 1};
int main( )
{
  int b[2048][2048];
  copyij(a, b);
  // copyji(a, b);
}
```

上述功能一样，但执行会有什么变化呢?

Segmentation fault (core dumped) ☹

Why ?

操作系统、编译器： 进程地址空间布局: 栈大小有限, 如为8MB

2048*2048*4 = 16MB

举例: 改变数据布局

例： 一个结构体大数组分拆成若干个数组

```
struct student {
int num;
char name[20];
…     …
}
struct  student st[10000];
```

```
int num[10000];
char name[10000][20];
…     …
```

- 若是顺序处理每个结构体的多个域，左边的数据局部性较好
- 若是先顺序处理每个结构的num域，再处理每个结构的name域，…，则右边的数据局部性较好
- 最好是按左边编程，由编译器决定是否需要将数据按右边布局