



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-II

《编译原理(H)》

张昱

0551-63603804, [yuzhang@ustc.edu.cn](mailto:yuzhang@ustc.edu.cn)

中国科学技术大学  
计算机科学与技术学院



## 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**)



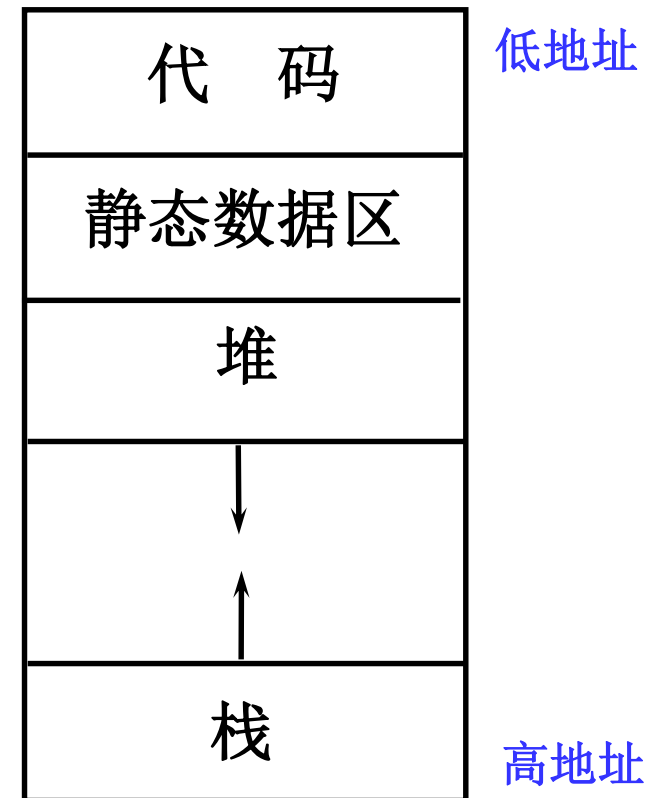
## 6.2 全局栈式存储分配

- 运行时内存的划分
- 活动树和运行栈
- 调用序列
- 栈上可变长度数据、悬空引用



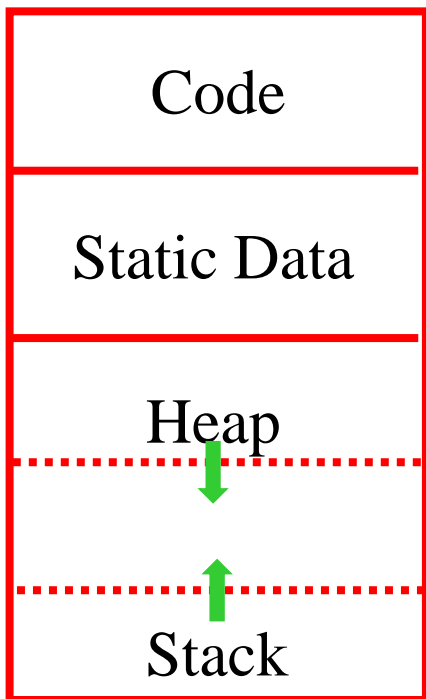
## □ 逻辑地址空间（OS：进程地址空间）

- **代码区：** 存储可执行代码  
加载时静态分配，只读，一般可共享
- **静态数据区：** 分初始化数据、未初始化数据  
加载时静态分配，大小在编译和链接时确定
- **堆：** 用于动态内存分配  
动态分配，从低地址向高地址增长，  
大小在运行期间可变，OS通过sbrk或mmap管理
- **栈：** 用于存储活动记录  
动态分配，从高地址向低地址增长，  
大小根据函数调用的深度在运行期可变
- **共享库：** 加载程序运行时需要的动态链接库  
通过mmap动态加载，随机放置，可与其他进程共享





## 32位Linux系统



低地址 0x080480000

- **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

## 64位Linux系统

低地址 0x0000000000400000

- 使用**低48位**虚拟地址(**256TB**)  
48-63位必须与47位一致
- 用户空间: 低128TB  
0 ~ 0x7FFFFFFFFFFF
- 内核空间: 64TB  
0xFFFF880000000000-  
0xFFFFc7FFFFFFFFFFF

高地址 0x00007FFFFFFFFF0000  
TASK\_SIZE

Linux 通常启用了**ASLR (地址空间布局随机化)**，这会在进程每次运行时随机化堆、栈、共享库等内存区域的起始地址，增加程序安全性，防止攻击者利用已知的内存地址进行缓冲区溢出攻击等漏洞利用。



## □ 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
- 自动变量auto -- 在活动记录中

(改变生存期)



# 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) ); ...
```



## 6.2 全局栈式存储分配

- 运行时内存的划分
- 活动树和运行栈
- 调用序列
- 栈上可变长度数据、悬空引用



```
1 int a[11];
2 void readArray() { /* Reads 9 integers into a[1],...,a[9].*/
3     int i; ...
4 }
5 }
6 int partition(int k, int n) {
7     /* Picks a separator value v, and partitions a[m..n] so that a[k..p-1] are less than v,
8        a[p]=v, and a[p+1..n] are equal to or great than v. Returns p.*/
9     ...
10 }
11 void quickSort(int k, int n) {
12     int i;
13     if( n>m){
14         i=partition(k,n);
15         quickSort(k,i-1);
16         quickSort(i+1,n);
17     }
18 }
19 main() {
20     readArray();
21     a[0]=-9999;
22     a[10]=9999;
23     quickSort(1,9);
24 }
```

**C程序**

**sort**

**readArray**

**exchange**

**partition**

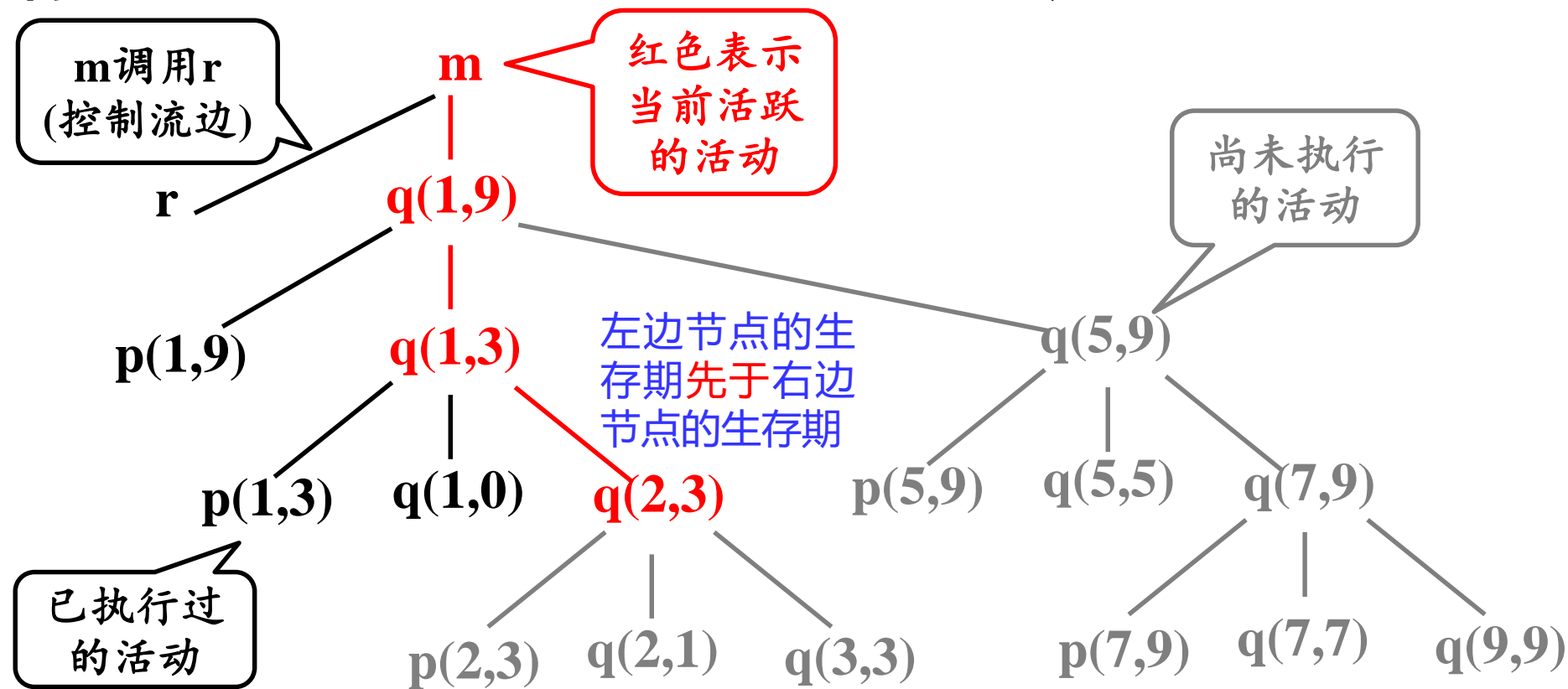
**quicksort**



# 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

对任意两个过程，  
其生存期或者嵌套，  
或者无重叠





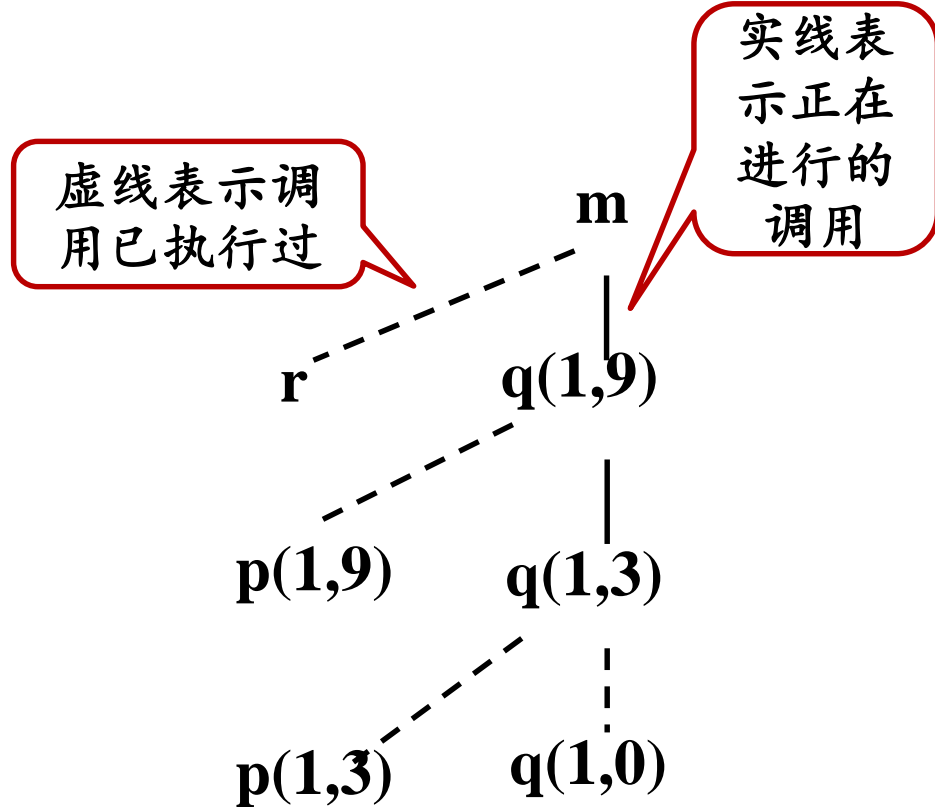
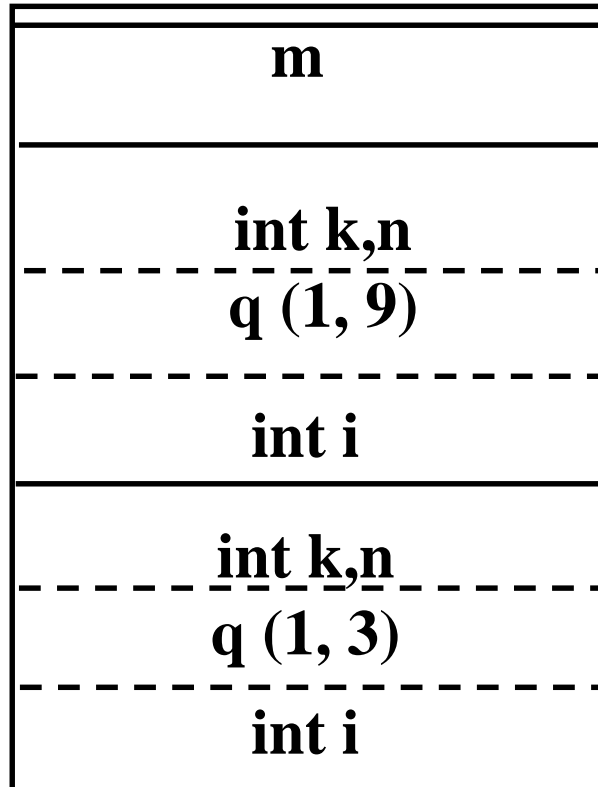
## □ 活动树的特点

- 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**）

静态数据区  
`int a[11];`





## 6.2 全局栈式存储分配

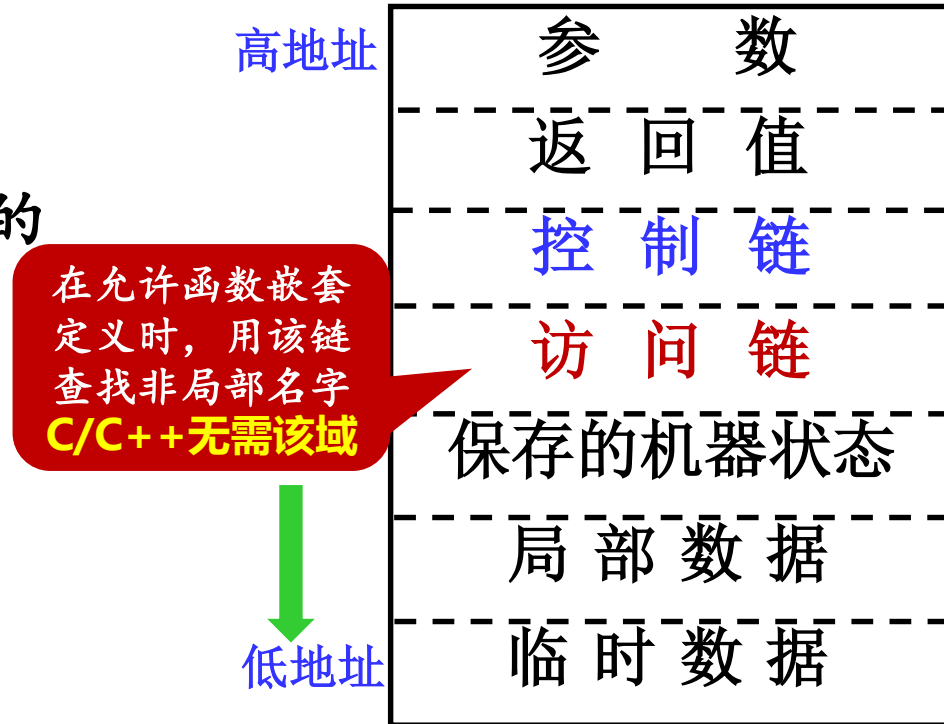
- 运行时内存的划分
- 活动树和运行栈
- 调用序列
- 栈上可变长度数据、悬空引用

## □ 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



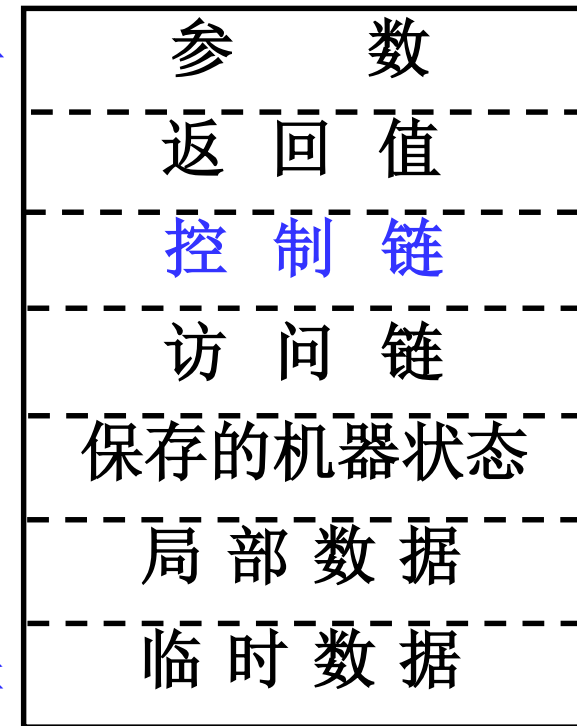
## □ 设计的一些原则

- 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
- 用同样的代码来执行各个活动的保存和恢复

高地址，  
靠近调用者  
活动记录

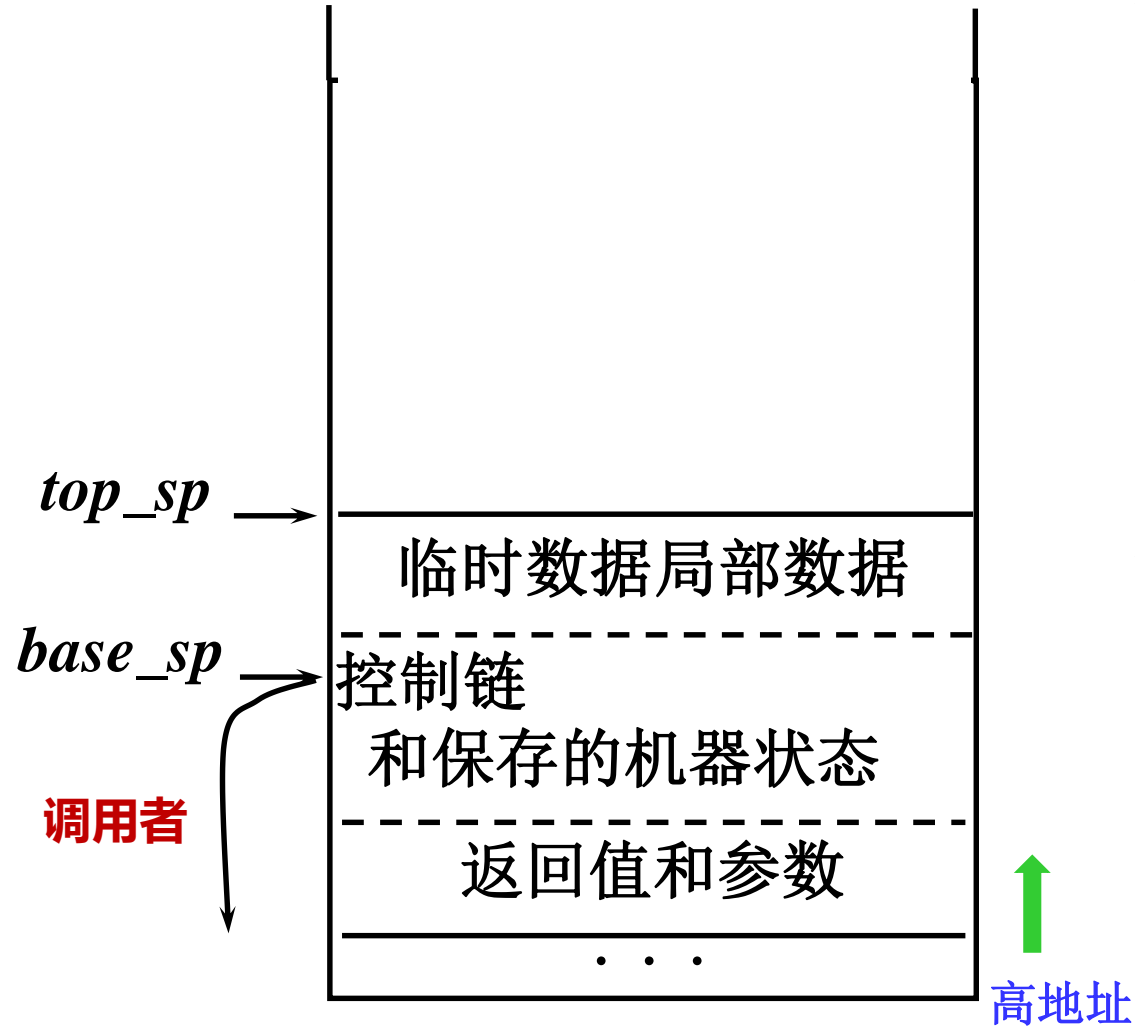


低地址





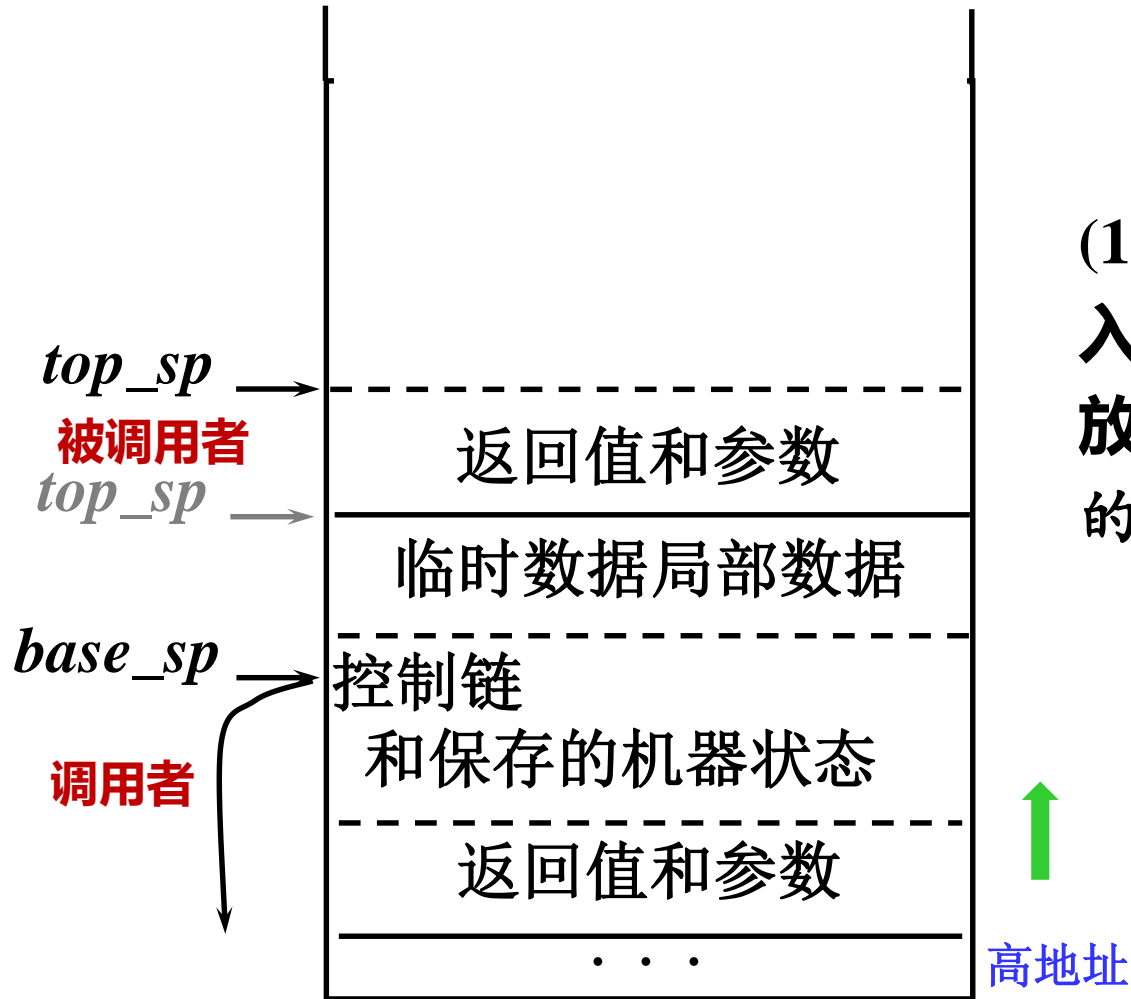
# 过程调用序列：p调用q



- ✓ *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)



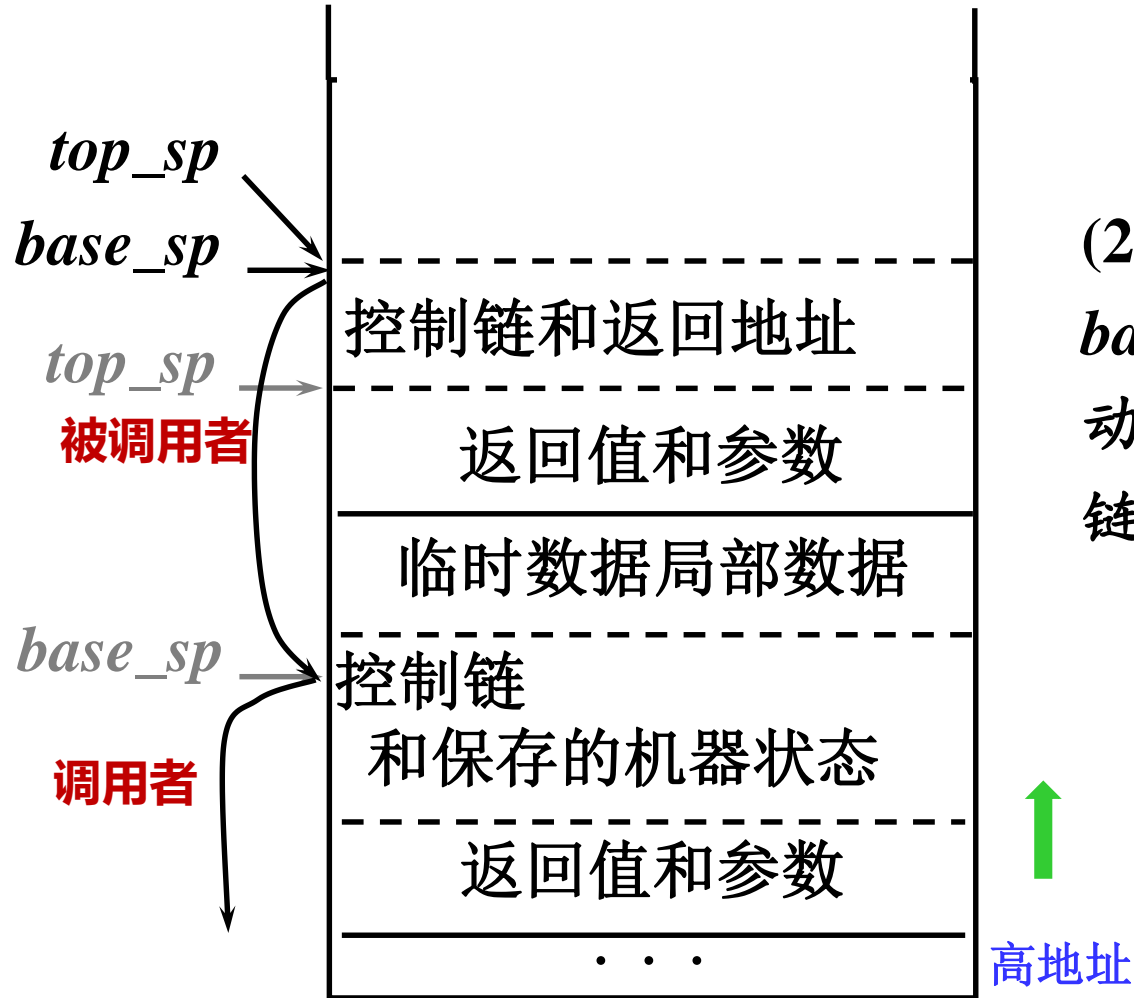
# 过程调用序列：p调用q



(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

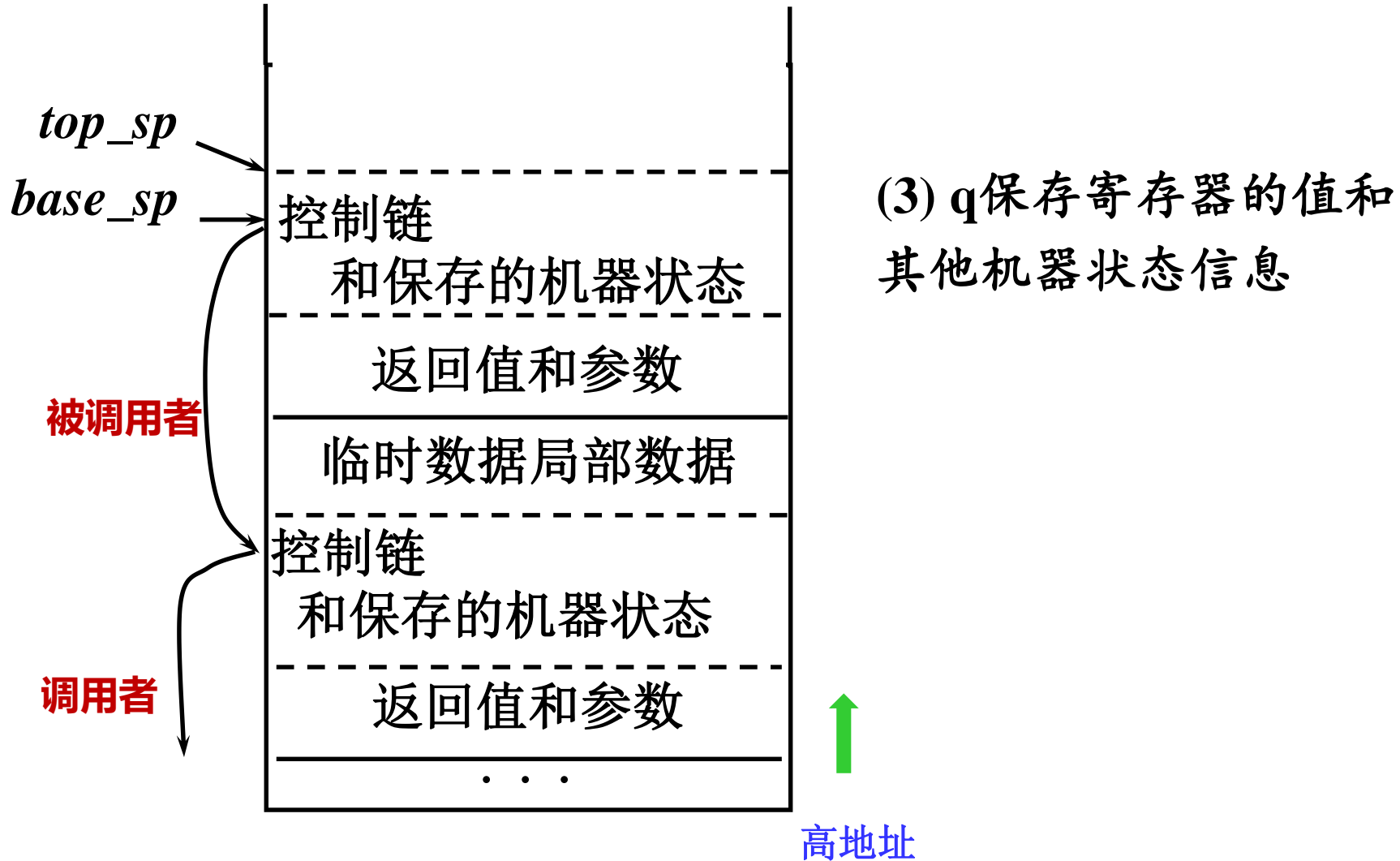


(2) p把返回地址和当前  $base\_sp$  的值存入q的活动记录中，建立q的访问链，增加  $base\_sp$  的值

↑  
高地址

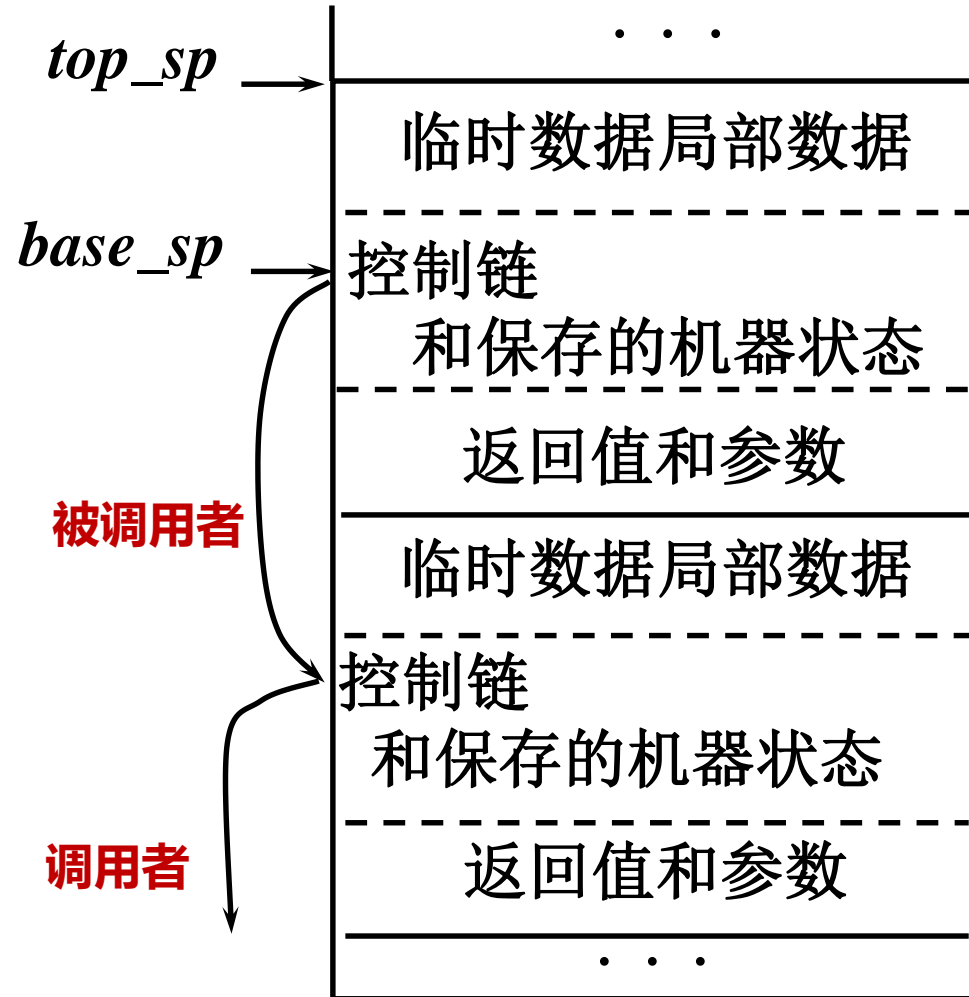


# 过程调用序列：p调用q





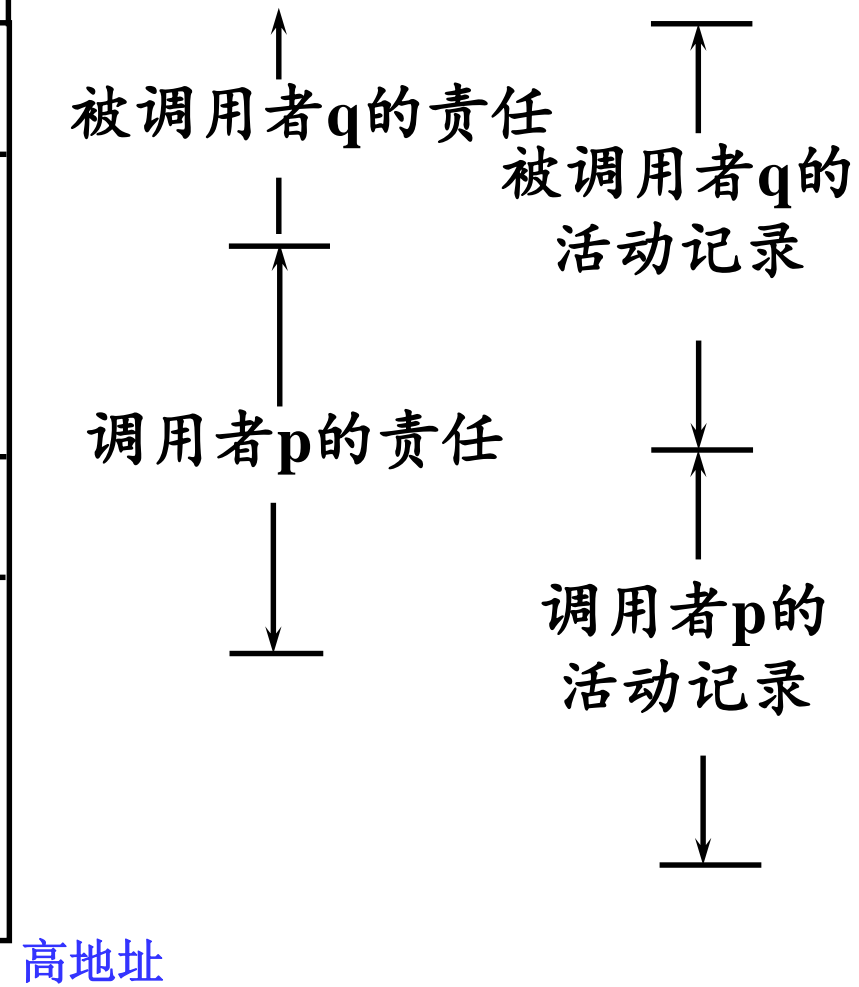
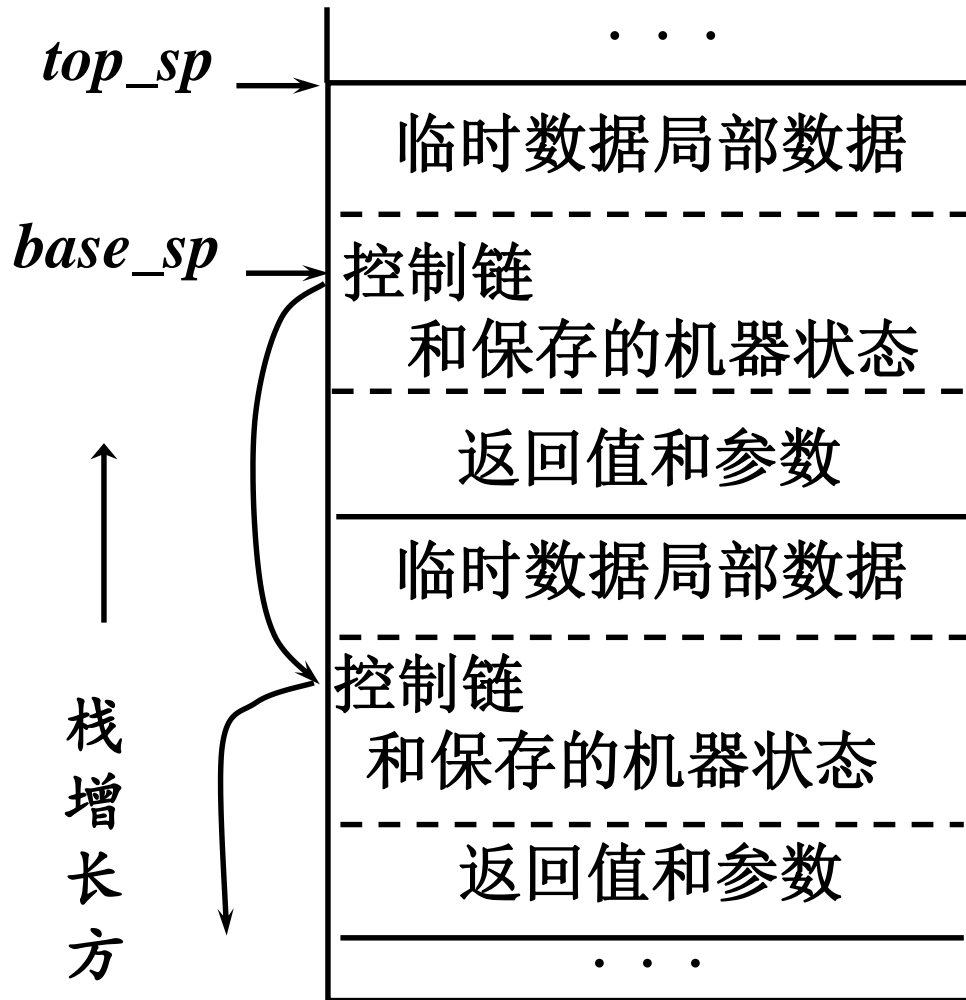
# 过程调用序列：p调用q



(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

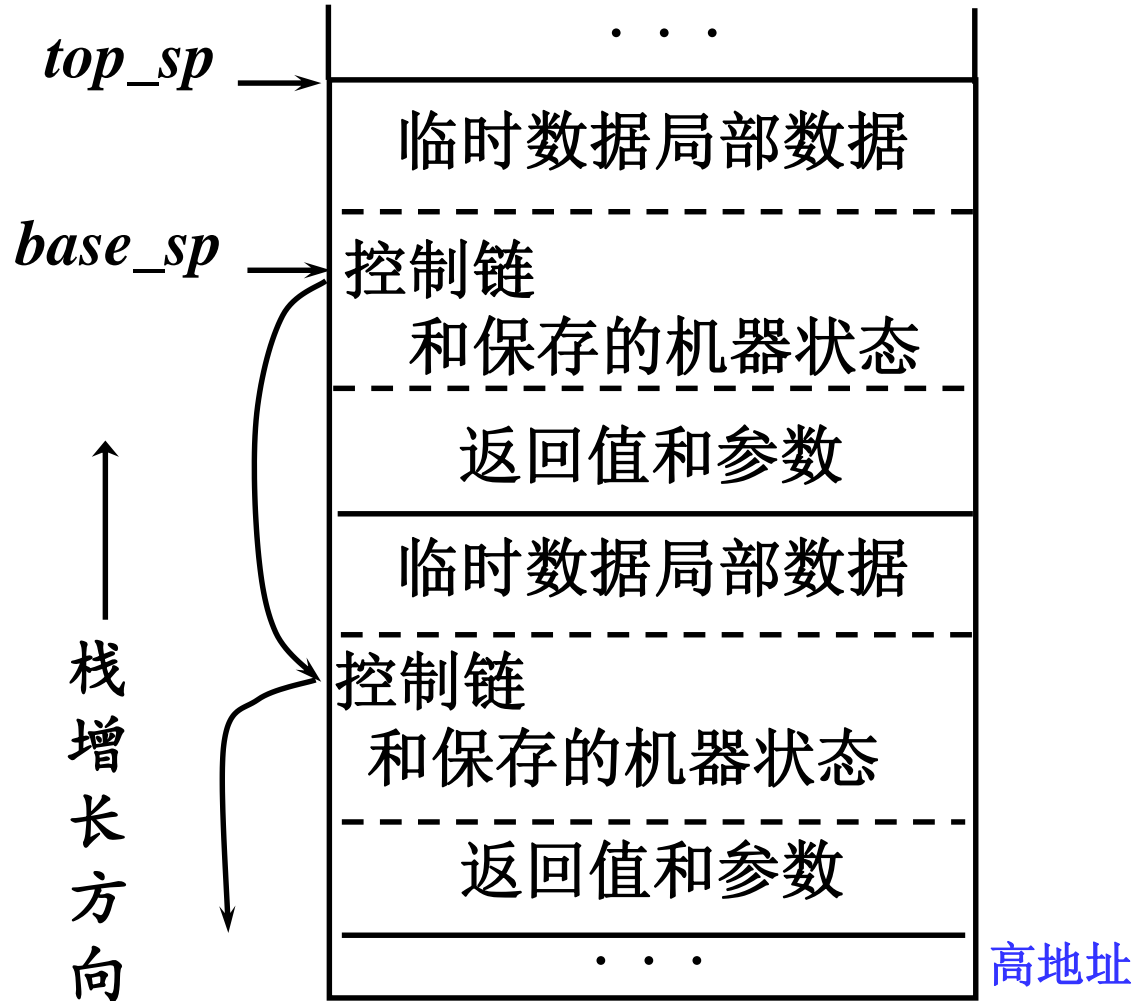


# 调用者p和被调用者q的任务划分



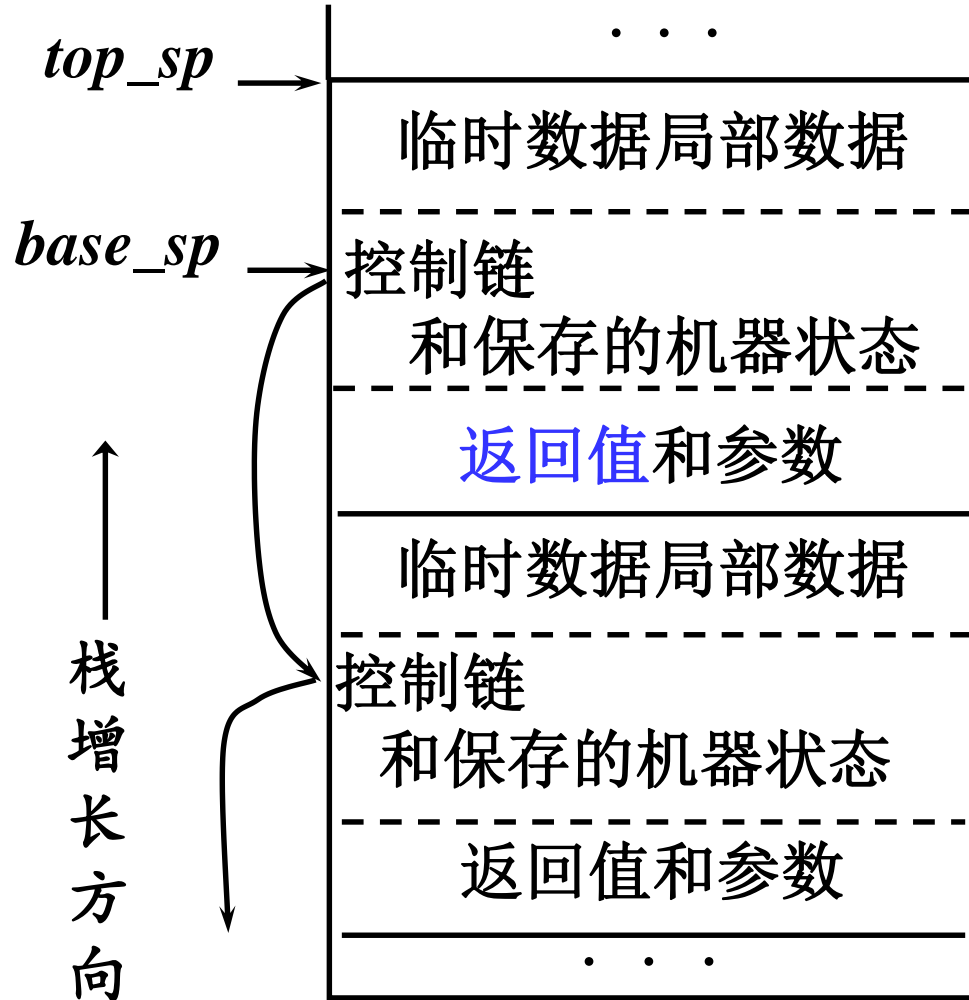


# 过程返回序列：p调用q





# 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

高地址





## □ 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

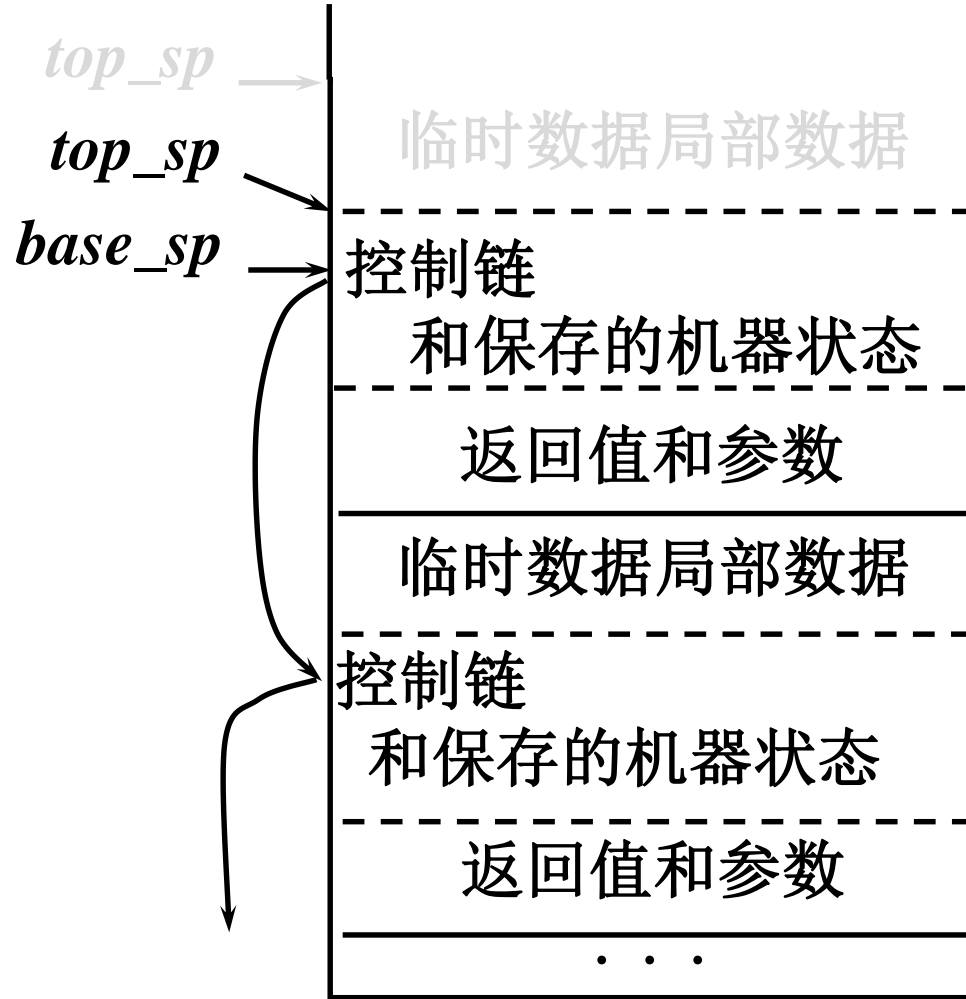
## □ ARM: [AAPCS](#)

■ 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

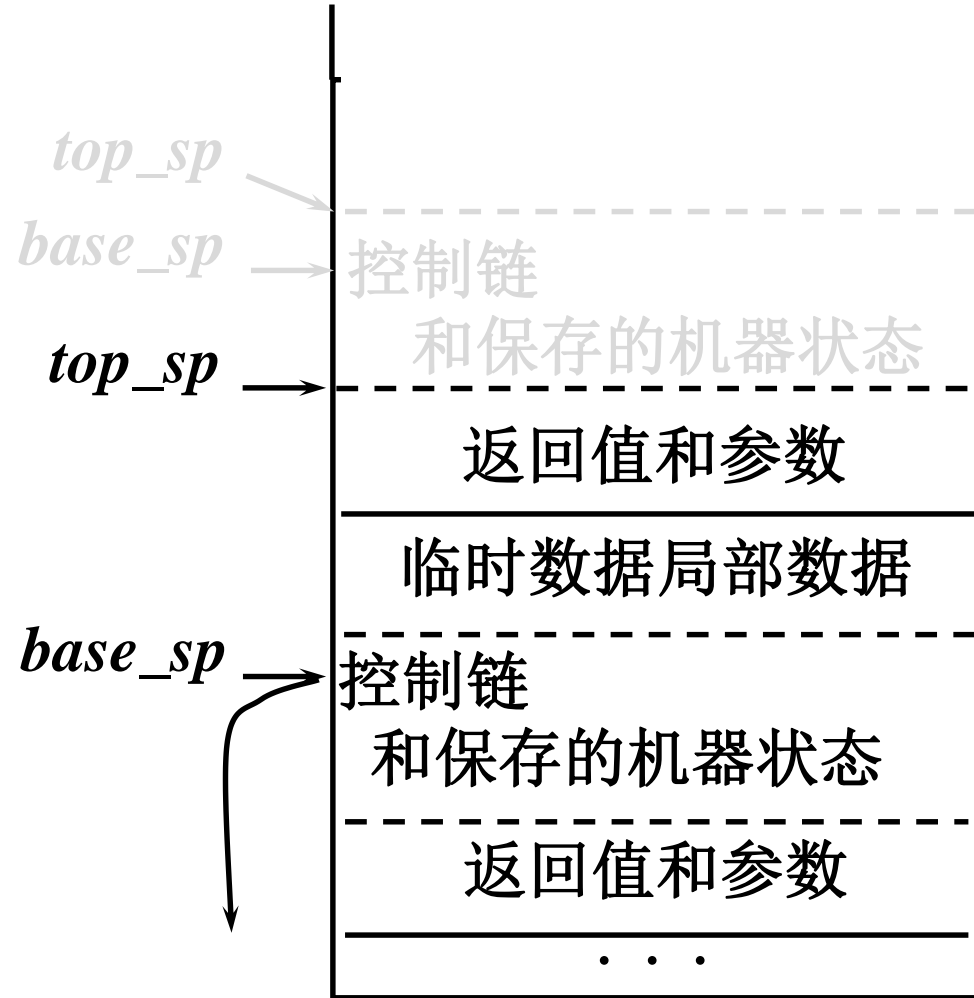


(2) q对应调用序列的步骤(4)，减小 $top\_sp$ 的值  
(释放局部变量和临时数据的空间)

高地址



# 过程返回序列：p调用q

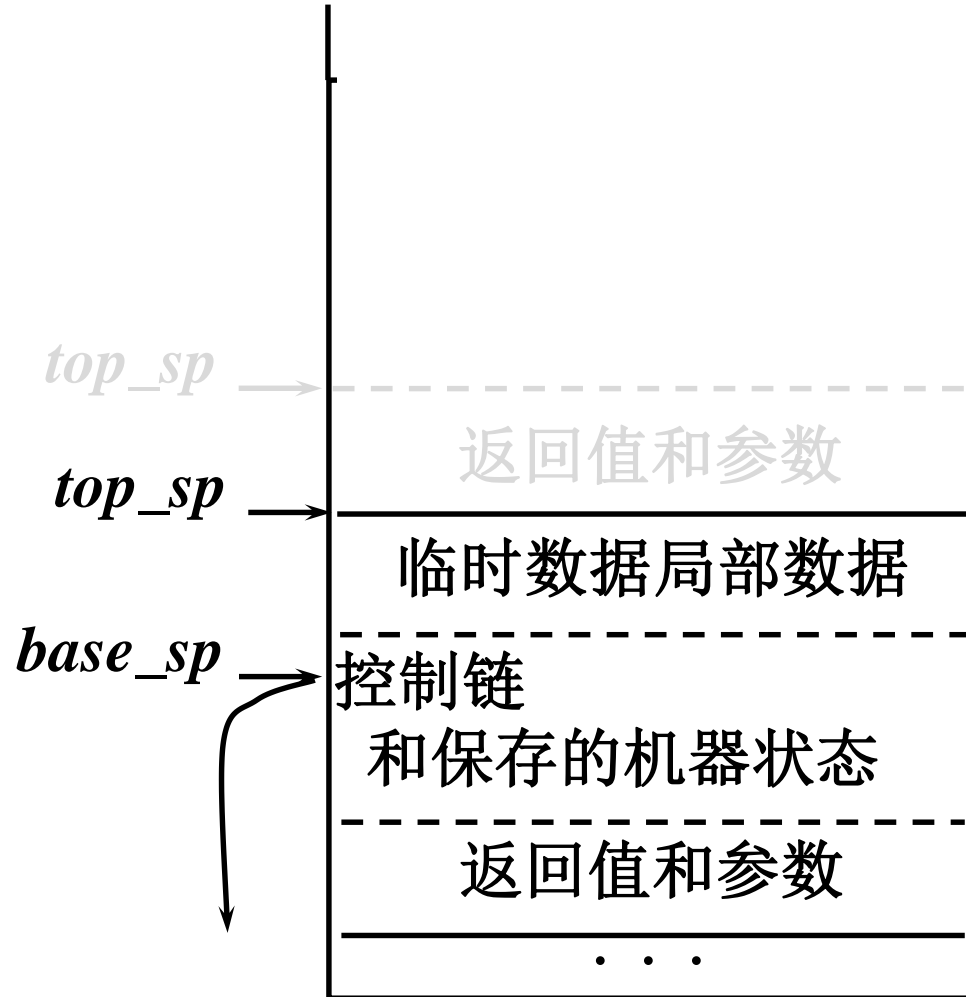


(3) q恢复寄存器(包括  $base\_sp$ )和机器状态, 返回p

高地址



# 过程返回序列：p调用q

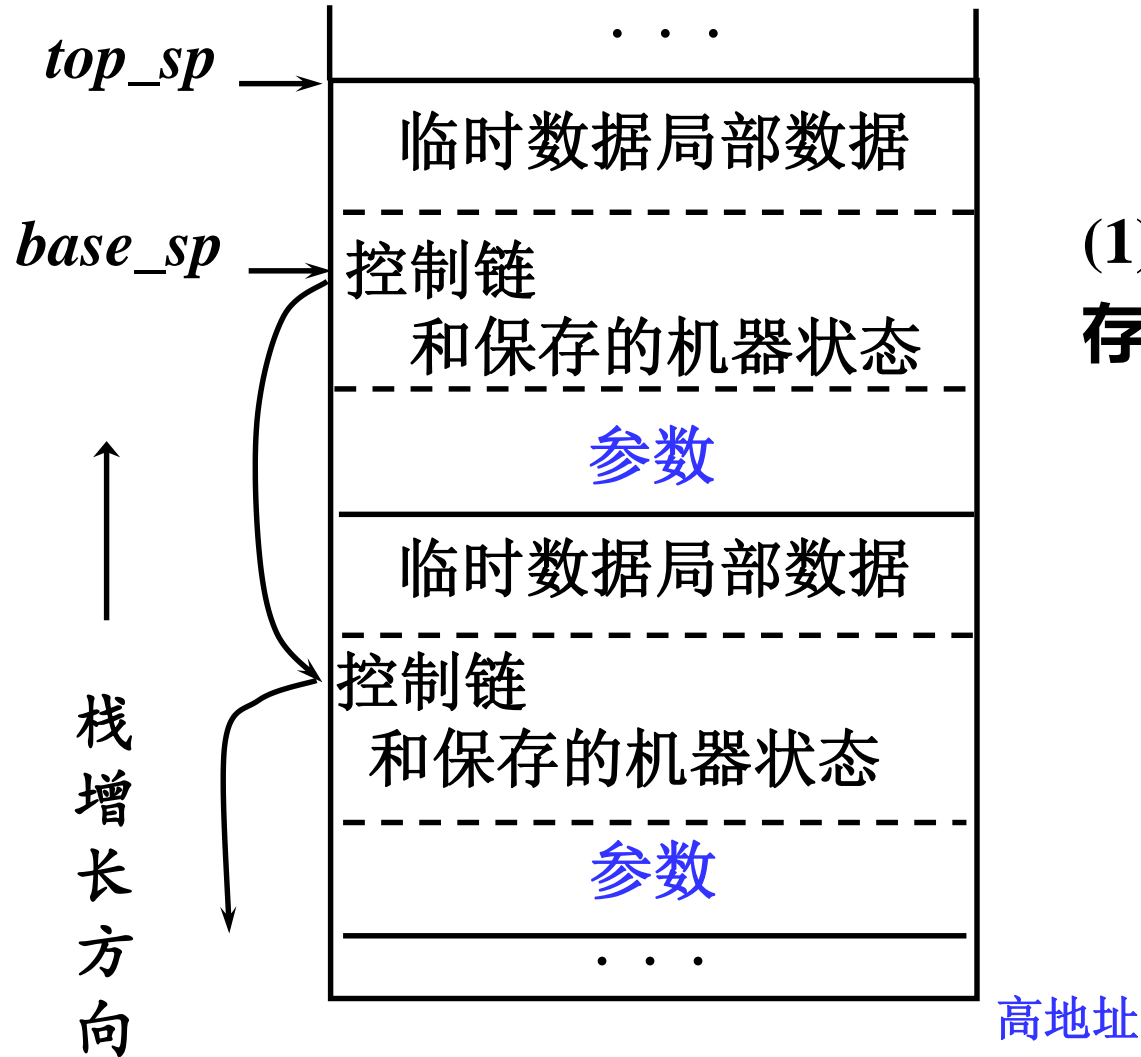


(4) p根据参数个数与类型和返回值调整  $top\_sp$  (释放参数空间), 然后取出返回值

高地址



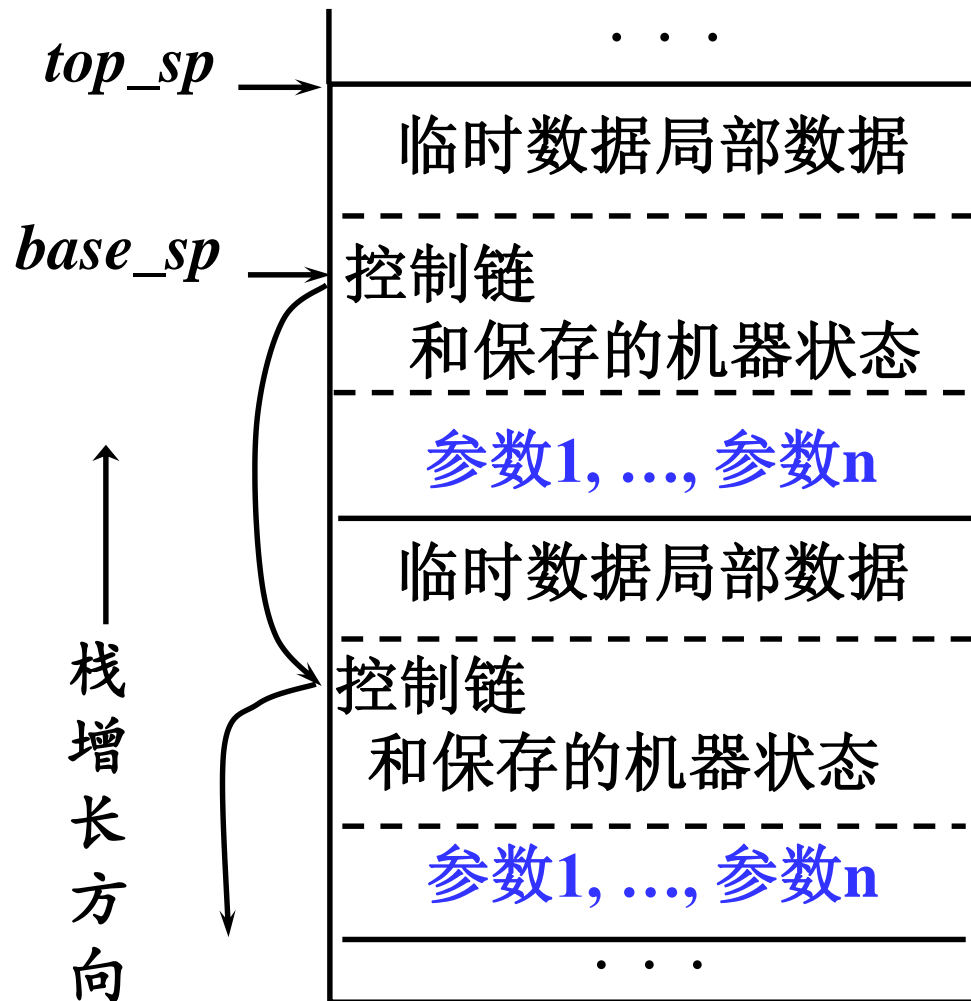
# 过程的参数个数可变的情况



(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”);`



## 6.2 全局栈式存储分配

- 运行时内存的划分
- 活动树和运行栈
- 调用序列
- 栈上可变长度数据、悬空引用



## □ 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

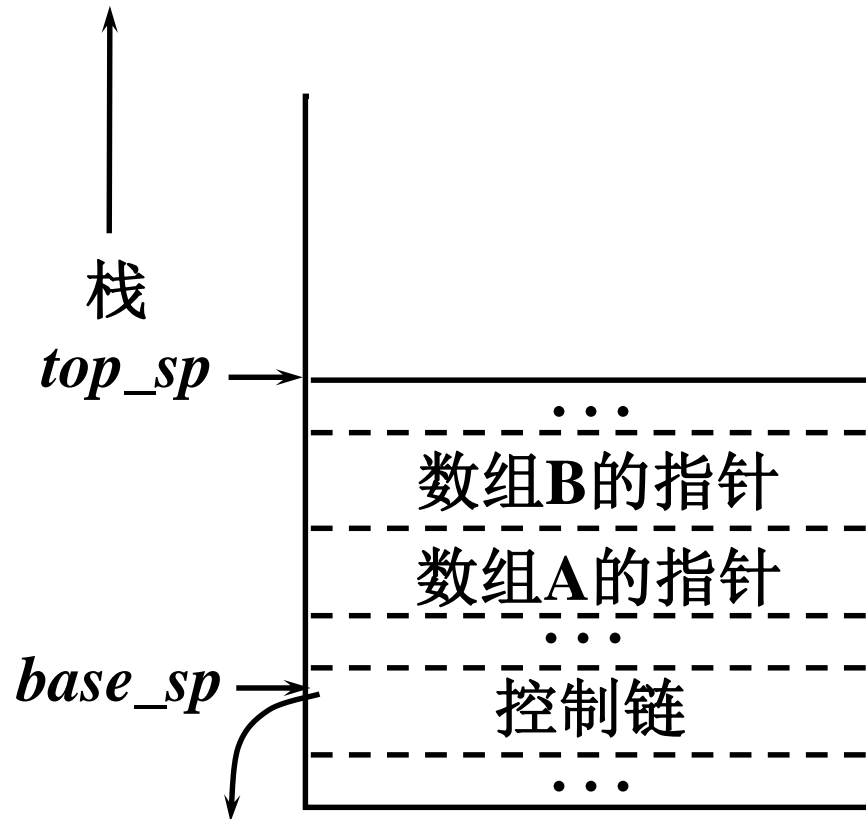
## □ 如何在栈上布局可变长的数组?

■ 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

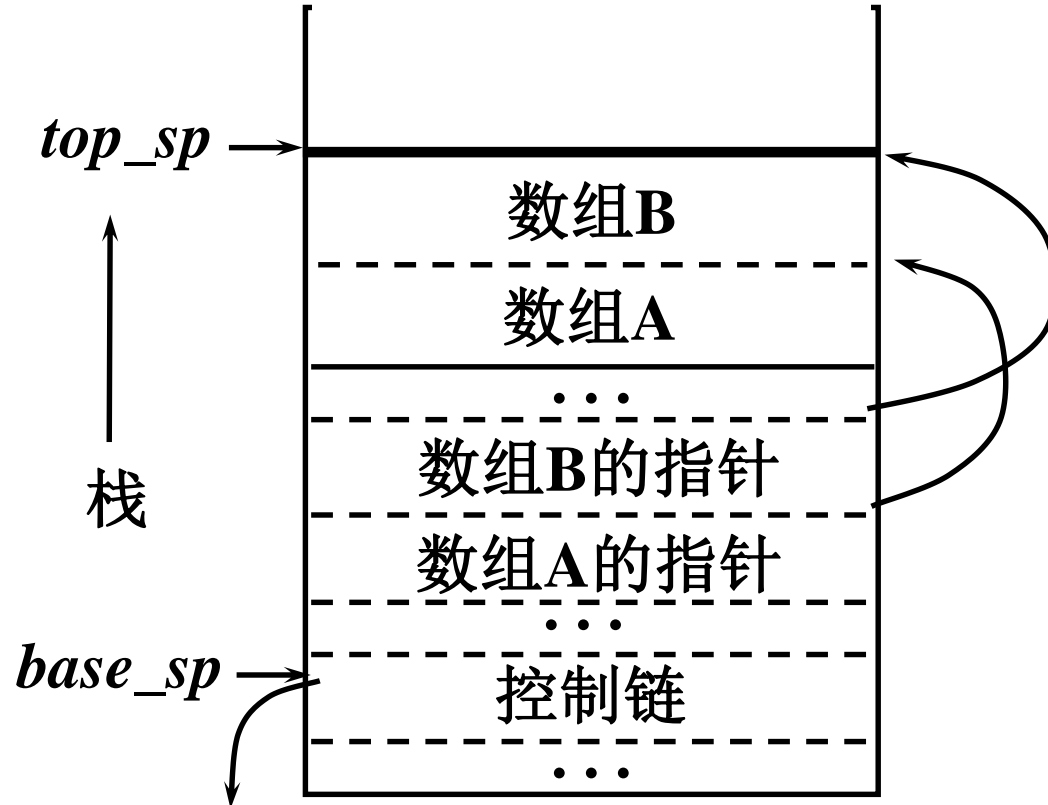


## □ 访问动态分配的数组



(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

## ■ 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）



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

## C程序应用举例

□ 缺省用 gcc v7.5.0



# 例题2 函数调用与返回

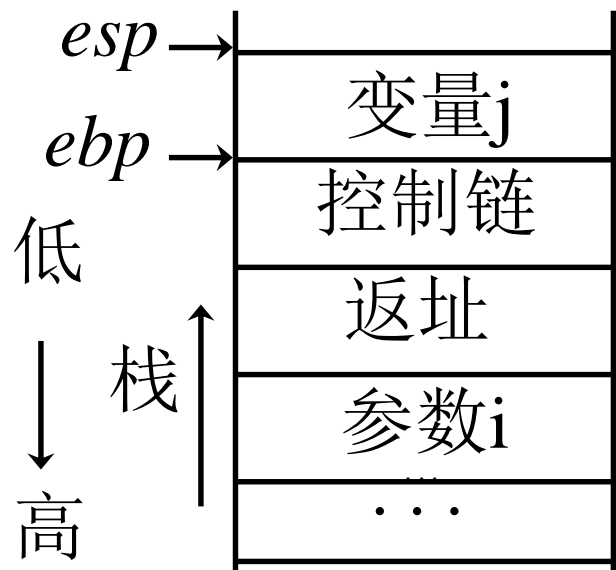
gcc -S  
注意早期的  
编译器版本:  
通过栈传参

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

func:

```
pushl %ebp      老的基地址指针压栈
movl %esp,%ebp  修改基地址指针
subl $4,%esp    为j分配空间
movl 8(%ebp),%edx 取i到寄存器
decl %edx       i - 1
movl %edx,-4(%ebp)  i - 1 => j
movl -4(%ebp),%eax
pushl %eax      把实参j的值压栈
call func       函数调用
addl $4,%esp    恢复栈顶指针

L1:
leave           即 mov ebp, esp; pop ebp
ret             即 pop eip(下条指令地址)
```





# 例题2 函数调用与返回

```

void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}

```

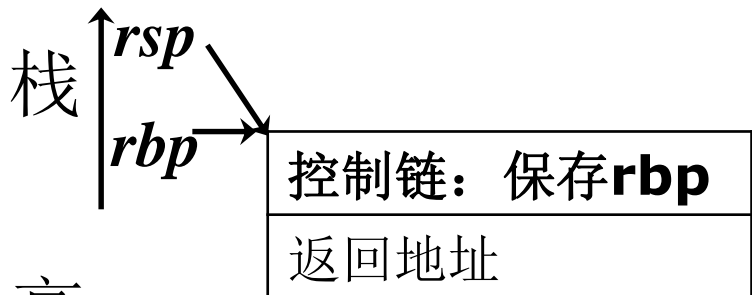
**gcc -S -fno-pie**  
-fno-pie 用于关闭地址随机化

```

pushq   %rbp      老的基址指针压栈
movq    %rsp, %rbp 修改基址指针
subq    $32, %rsp
movq    %rdi, -24(%rbp)
movq    -24(%rbp), %rax
subq    $1, %rax
movq    %rax, -8(%rbp)
movq    -8(%rbp), %rax
movq    %rax, %rdi
call    func
nop
leave
ret

```

低



高



# 例题2 函数调用与返回

```

void func(long i)  func:
{
    long j;
    j = i - 1;
    func(j);
}

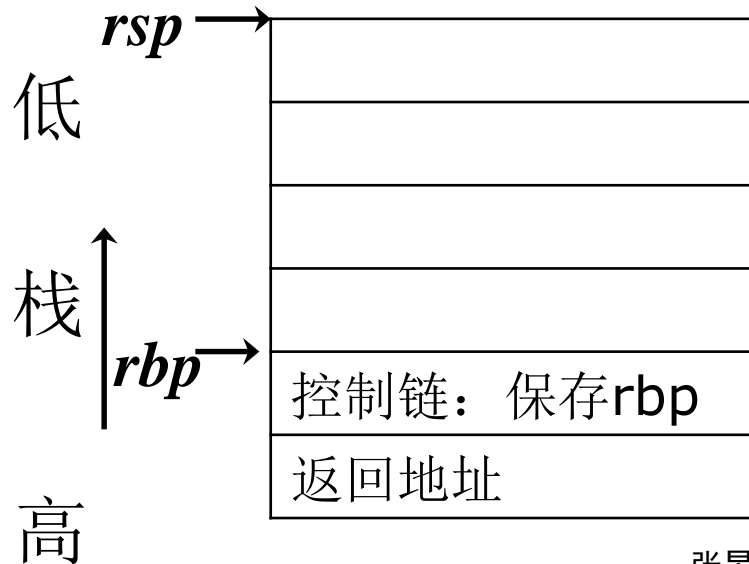
```

```

pushq  %rbp      老的基址指针压栈
movq   %rsp, %rbp  修改基址指针
subq  $32, %rsp  分配32字节空间
movq   %rdi, -24(%rbp)
movq   -24(%rbp), %rax
subq   $1, %rax
movq   %rax, -8(%rbp)
movq   -8(%rbp), %rax
movq   %rax, %rdi
call   func
nop
leave
ret

```

**gcc -S**



**GCC 自4.5 版本开始, 栈上的数据必须按16字节对齐, 之前按4字节对齐**

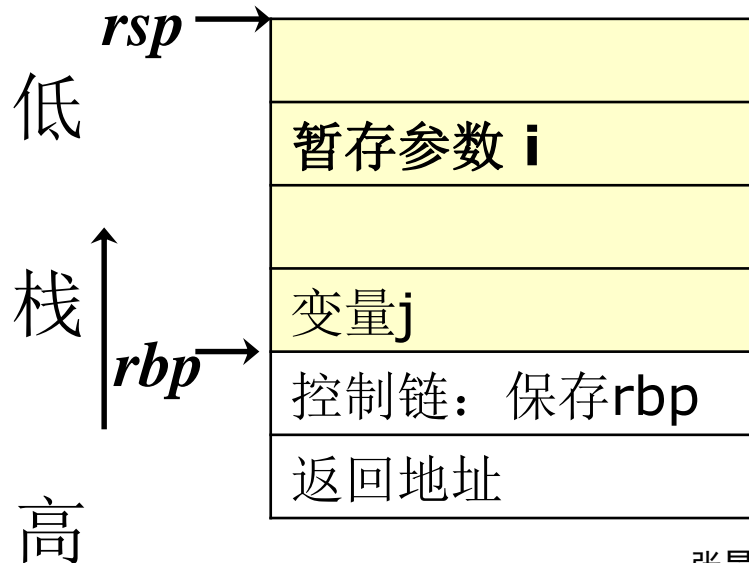


# 例题2 函数调用与返回

```
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}
```

```
pushq %rbp      老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp  分配32字节空间
movq %rdi, -24(%rbp) 参数i 暂存到栈
movq -24(%rbp), %rax
subq $1, %rax
movq %rax, -8(%rbp)
movq -8(%rbp), %rax
movq %rax, %rdi
call func
nop
leave
ret
```

gcc -S



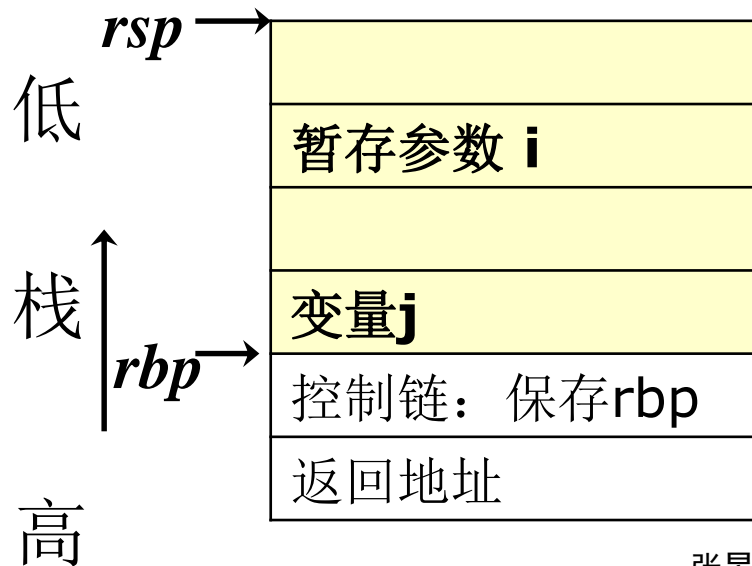
参数i通过寄存器rdi传递



# 例题2 函数调用与返回

```
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}
```

**gcc -S**



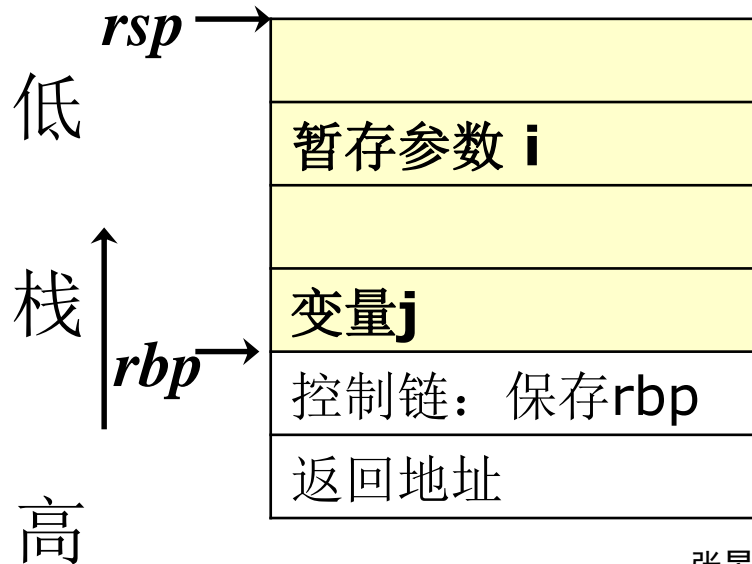
```
pushq %rbp    老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp 分配32字节空间
movq %rdi, -24(%rbp) 参数i 暂存到栈
movq -24(%rbp), %rax i加载到寄存器rax
subq $1, %rax    i-1=>rax
movq %rax, -8(%rbp) i-1存入变量j
movq -8(%rbp), %rax
movq %rax, %rdi
call func
nop
leave
ret
```



# 例题2 函数调用与返回

```
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}
```

**gcc -S**



```
pushq %rbp    老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp 分配32字节空间
movq %rdi, -24(%rbp) 参数i 暂存到栈
movq -24(%rbp), %rax i加载到寄存器rax
subq $1, %rax    i-1=>rax
movq %rax, -8(%rbp) i-1存入变量j
movq -8(%rbp), %rax 加载j到寄存器rax
movq %rax, %rdi    通过rdi传参
call func
nop
leave
ret
```



# 例题2 函数调用与返回

```

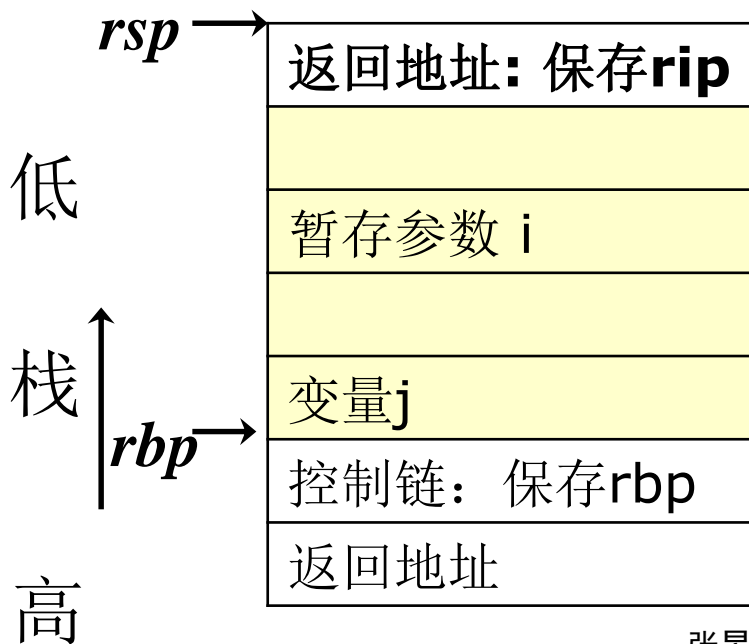
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}

```

```

pushq %rbp      老的基址指针压栈
movq %rsp, %rbp  修改基址指针
subq $32, %rsp  分配32字节空间
movq %rdi, -24(%rbp)  参数i 暂存到栈
movq -24(%rbp), %rax  i加载到寄存器rax
subq $1, %rax      i-1=>rax
movq %rax, -8(%rbp)  i-1存入变量j
movq -8(%rbp), %rax  加载j到寄存器rax
movq %rax, %rdi      通过rdi传参
call func           保存返回地址并跳转到func
nop
leave
ret

```





# 例题2 函数调用与返回

```

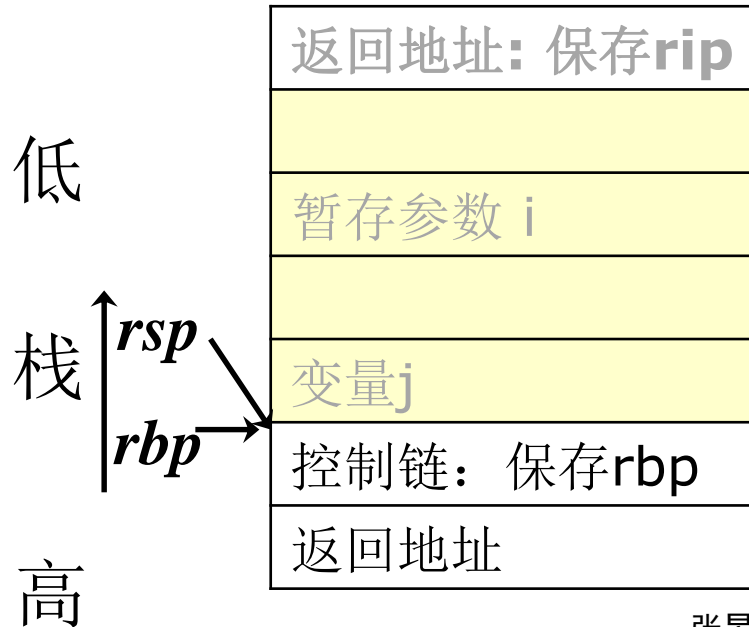
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}

```

```

pushq %rbp      老的基址指针压栈
movq %rsp, %rbp  修改基址指针
subq $32, %rsp  分配32字节空间
movq %rdi, -24(%rbp)  参数i 暂存到栈
movq -24(%rbp), %rax  i加载到寄存器rax
subq $1, %rax      i-1=>rax
movq %rax, -8(%rbp)  i-1存入变量j
movq -8(%rbp), %rax  加载j到寄存器rax
movq %rax, %rdi      通过rdi传参
call func            保存返回地址并跳转到func
nop
leave  即 movq %rbp, %rsp; popq %rbp
ret

```





# 例题2 函数调用与返回

```

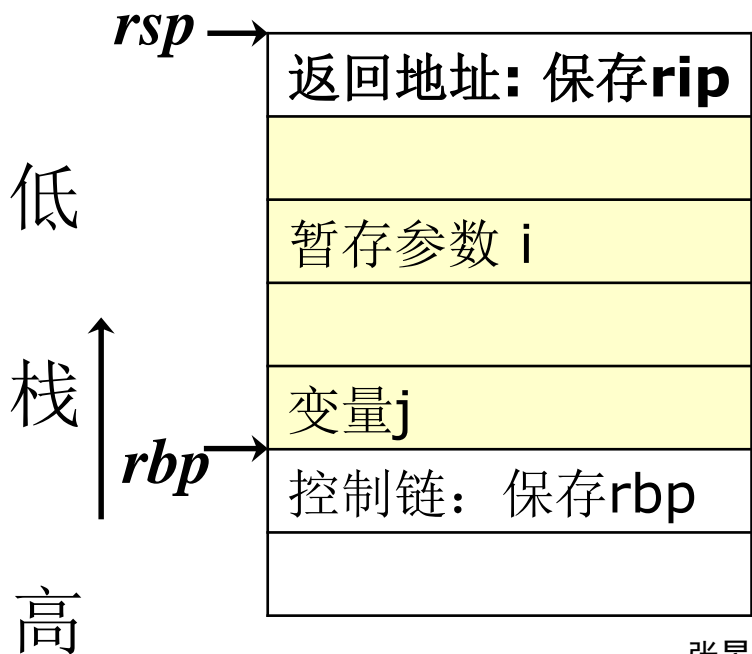
void func(long i) func:
{
    long j;
    j = i - 1;
    func(j);
}

```

```

pushq %rbp      老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp  分配32字节空间
movq %rdi, -24(%rbp) 参数i 暂存到栈
movq -24(%rbp), %rax i加载到寄存器rax
subq $1, %rax    i-1=>rax
movq %rax, -8(%rbp) i-1存入变量j
movq -8(%rbp), %rax 加载j到寄存器rax
movq %rax, %rdi    通过rdi传参
call func          保存返回地址并跳转到func
nop
leave  即 movq %rbp, %rsp; popq %rbp
ret

```





# 例题2 函数调用与返回

```

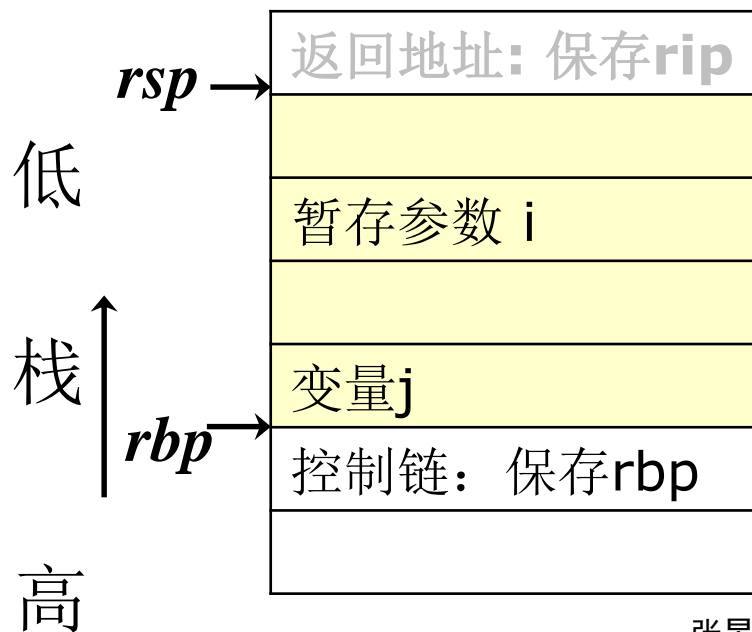
void func(long i)  func:
{
    long j;
    j = i - 1;
    func(j);
}

```

```

pushq  %rbp      老的基址指针压栈
movq   %rsp, %rbp  修改基址指针
subq   $32, %rsp  分配32字节空间
movq   %rdi, -24(%rbp)  参数i 暂存到栈
movq   -24(%rbp), %rax  i加载到寄存器rax
subq   $1, %rax    i-1=>rax
movq   %rax, -8(%rbp)  i-1存入变量j
movq   -8(%rbp), %rax  加载j到寄存器rax
movq   %rax, %rdi    通过rdi传参
call   func        保存返回地址并跳转到func
nop
leave  即 movq %rbp, %rsp; popq %rbp
ret    即 popq %rip

```





# 例题2 函数调用与返回

```
void func(int i)
{
    int j;
    j = i - 1;
    func(j);
}
```

func:

```
pushq %rbp      老的基址指针压栈
movq %rsp, %rbp  修改基址指针
subq $32, %rsp  分配32字节空间
movq %edi, -20(%rbp)  参数i 暂存到栈
movq -20(%rbp), %eax  i加载到寄存器rax
subq $1, %eax      i-1=>eax
movq %eax, -4(%rbp)  i-1存入变量j
movq -4(%rbp), %eax  加载j到寄存器eax
movq %eax, %edi      通过edi传参
call func            保存返回地址并跳转到func
nop
leave  即 movq %rbp, %rsp; popq %rbp
ret    即 popq %rip
```

参数i的类型由long改为int

- 1) rdi => edi
- 2) -24(%rbp) => -20(...)
- 3) rax => eax
- 4) -8(%rbp) => -4(...)



# 例题2 函数调用与返回

```
int func(long i)
{
    long j;
    j = i - 1;
    return i+func(j);
}
```

func:

```
pushq %rbp      老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp  分配32字节空间
movq %rdi, -24(%rbp) 参数i 暂存到栈
movq -24(%rbp), %rax i加载到寄存器rax
subq $1, %rax    i-1=>rax
movq %rax, -8(%rbp) i-1存入变量j
movq -8(%rbp), %rax 加载j到寄存器rax
movq %rax, %rdi    通过rdi传参
call func         保存返回地址并跳转到func
movl %eax, %edx   返回值保存到edx
movq -24(%rbp), %rax 加载 i 的值得到rax
addl %edx, %eax   eax存放i+func(j)
nop
leave 即 movq %rbp, %rsp; popq %rbp
ret 即 popq %rip
```

**返回类型由 void 改为int**  
**返回值为表达式**

- 1) rax传递返回值
- 2) 加式：先计算函数调用
- 3) 去除nop



# 例题2 函数调用与返回

```
void func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

func:

```
pushq %rbp      老的基址指针压栈
movq %rsp, %rbp  修改基址指针
subq $32, %rsp   分配32字节空间
movq %rdi, -24(%rbp)  参数i 暂存到栈
movq -24(%rbp), %rax  i加载到寄存器rax
subq $1, %rax     i-1=>rax
movq %rax, -8(%rbp)  i-1存入变量j
movq -8(%rbp), %rax  加载j到寄存器rax
movq %rax, %rdi     通过rdi传参
movl $0, %eax
call func          保存返回地址并跳转到func
nop
leave  即 movq %rbp, %rsp; popq %rbp
ret    即 popq %rip
```

按老的参数声明方式

将该函数视为返回 int 型  
事先将存放返回值的eax清0



# 例题2 函数调用与返回

```
void func(i)
int i;
{
    long j;
    j = i - 1;
    func(j);
}
```

func:

```
pushq %rbp      老的基址指针压栈
movq %rsp, %rbp 修改基址指针
subq $32, %rsp  分配32字节空间
movq %edi, -20(%rbp) 参数i 暂存到栈
movq -20(%rbp), %eax i加载到寄存器rax
subq $1, %eax      i-1=>eax
cltq           类型提升 int=>long
movq %rax, -8(%rbp) i-1存入变量j
movq -8(%rbp), %rax 加载j到寄存器rax
movq %rax, %rdi     通过rdi传参
movl $0, %eax
call func保存返回地址并跳转到func
nop
leave 即 movq %rbp, %rsp; popq %rbp
ret   即 popq %rip
```

按老的参数声明方式

参数i的类型由long改为int

- 1) cltq: 类型提升
- 2) 事先将存放返回值的eax清0



# 例题2 函数调用与返回

"GCC 11.4.0: (ubuntu1~22.04)"

```
void func(long i)
{
    long j;
    j = i - 1;
    func(j);
}
```

func:

.arch armv8-a

....

.align 2

.global func .type func, %function

stp x29, x30, [sp, -48]!

将x29和x30的值压栈，sp减去 48 并更新sp

mov x29, sp

设置x29为新的sp

str x0, [sp, 24]

将寄存器x0(参数i)的值存入sp+24单元

ldr x0, [sp, 24]

将sp+24单元的值加载到x0

sub x0, x0, #1

x0=x0-1, 即x0=i-1

str x0, [sp, 40]

存入sp+40 (局部变量j)

ldr x0, [sp, 40]

加载sp+40单元的值到x0

bl func

跳转到func函数，返回地址保存到x30

nop

ldp x29, x30, [sp], 48 从栈中恢复FP和LR

ret

.size func, .-func

.section .note.GNU-stack,"",@progbits

**在x86-64机器上用交叉编译器产生ARM代码**

- `sudo apt install gcc-aarch64-linux-gnu`
- `aarch64-linux-gnu-gcc -S`

x29	帧指针寄存器	FP, Frame Pointer	保存函数调用的栈帧指针
x30	链接寄存器	LR, Link Register	保存函数调用的返回地址
sp	栈顶寄存器	Stack Pointer	保存栈顶指针



# 例题3 参数数目可变的函数

```
void print()
{
    printf("%d,%d,%d");
}
```

程序运行时会输出**3**个整数

## X86-64

```
.LC0:
    .string "%d,%d,%d"
print:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $.LC0, %edi
    movl    $0, %eax
    call   printf
    nop
    popq   %rbp
    ret
```

## ARMv8-a

```
.LC0:
    .string "%d,%d,%d"
print:
    stp    x29, x30, [sp, -16]!
    mov    x29, sp
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
    bl    printf
    nop
    ldp    x29, x30, [sp], 16
    ret
```

### \$ gcc -S print.c

```
print.c: In function ‘print’ :
print.c:3:5: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-
declaration]
    printf("%d,%d,%d");
    ^~~~~~
print.c:3:5: warning: incompatible implicit declaration of built-in function ‘printf’
print.c:3:5: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
print.c:3:14: warning: format ‘%d’ expects a matching ‘int’ argument [-
Wformat=]
    printf("%d,%d,%d");
           ~^
.....
```



## 例题4 参数与局部变量

```
func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

gcc v11.4.0

ubuntu1~22.04

**输出:**

```
0x7ffdd906376c,0x7ffdd9063768,0x7ffdd9063764,0x7ffdd9063760
0x7ffdd906377c,0x7ffdd906377e,0x7ffdd9063780,0x7ffdd9063784
```



# 例题4 参数与局部变量

```

func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:

0x7fdd906376c,  
0x7fdd9063768,  
0x7fdd9063764,  
0x7fdd9063760

```

subq    $48, %rsp
movl    %edi, %edx
movl    %esi, %eax
movss   %xmm0, -44(%rbp)
movss   %xmm1, -48(%rbp)
movw    %dx, -36(%rbp)
movw    %ax, -40(%rbp)
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
leaq    -48(%rbp), %rsi
leaq    -44(%rbp), %rcx
leaq    -40(%rbp), %rdx
leaq    -36(%rbp), %rax
movq    %rsi, %r8

```

寄存器传参;  
short提升为  
int传递



# 例题4 参数与局部变量

```

func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:

0x7fdd906376c,  
0x7fdd9063768,  
0x7fdd9063764,  
0x7fdd9063760

```

subq    $48, %rsp
movl    %edi, %edx
movl    %esi, %eax
movss   %xmm0, -44(%rbp)
movss   %xmm1, -48(%rbp)
movw    %dx, -36(%rbp)
movw    %ax, -40(%rbp)
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
leaq    -48(%rbp), %rsi
leaq    -44(%rbp), %rcx
leaq    -40(%rbp), %rdx
leaq    -36(%rbp), %rax
movq    %rsi, %r8

```

**取short参数  
压入栈中,  
按4字节对齐**

相对于rbp的  
偏移地址及其  
暂存的变量值

-36 i  
-40 j  
-44 f  
-48 e



# 例题4 参数与局部变量

```

func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:

0x7fdd906376c,  
0x7fdd9063768,  
0x7fdd9063764,  
0x7fdd9063760

```

movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
leaq    -48(%rbp), %rsi
leaq    -44(%rbp), %rcx
leaq    -40(%rbp), %rdx
leaq    -36(%rbp), %rax
movq    %rsi, %r8
movq    %rax, %rsi
movl    $.LC0, %edi
movl    $0, %eax
call   printf

```

**参数逆序  
存入寄存器**

相对于rbp的  
偏移地址及其  
暂存的变量值

-36 i  
-40 j  
-44 f  
-48 e



# 例题4 参数与局部变量

```

func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:  
0x7ffdd906376c,  
0x7ffdd9063768,  
0x7ffdd9063764,  
0x7ffdd9063760

```

movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
leaq    -48(%rbp), %rsi
leaq    -44(%rbp), %rcx
leaq    -40(%rbp), %rdx
leaq    -36(%rbp), %rax
movq    %rsi, %r8
movq    %rax, %rsi
movl    $.LC0, %edi
movl    $0, %eax
call    printf

```

存放返回值的  
寄存器清0,  
调用printf



# 例题4 参数与局部变量

```
func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
```

```
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

输出:

0x7fdd906376c,  
0x7fdd9063768,  
0x7fdd9063764,  
0x7fdd9063760

```
leaq -12(%rbp), %rsi
leaq -16(%rbp), %rcx
leaq -18(%rbp), %rdx
leaq -20(%rbp), %rax
movq %rsi, %r8
movq %rax, %rsi
movl $.LC0, %edi
movl $0, %eax
call printf
```

**参数逆序  
存入寄存器**

相对于rbp的  
偏移地址及其  
暂存的变量值

-12 e1  
-16 f1  
-18 j1  
-20 i1



# 例题4 参数与局部变量

```
func(i, j, f, e)
short i; short j; float f; float e;
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

gcc v11.4.0

ubuntu1~22.04

输出:

```
0x7ffd5859463c,0x7ffd58594638,0x7ffd58594630,0x7ffd58594628
0x7ffd5859464c,0x7ffd5859464e,0x7ffd58594650,0x7ffd58594654
0x7ffdd906376c,0x7ffdd9063768,0x7ffdd9063764,0x7ffdd9063760
0x7ffdd906377c,0x7ffdd906377e,0x7ffdd9063780,0x7ffdd9063784
```



# 例题4 参数与局部变量

```

func(i, j, f, e)
short i; short j; float f; float e;
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:  
0x7ffd5859463c,  
0x7ffd58594638,  
0x7ffd58594630,  
0x7ffd58594628

```

subq    $64, %rsp
movl    %edi, %edx
movl    %esi, %eax
movw    %dx, -36(%rbp)
movw    %ax, -40(%rbp)
cvtsd2ss %xmm0, %xmm0
movss   %xmm0, -48(%rbp)
cvtsd2ss %xmm1, %xmm0
movss   %xmm0, -56(%rbp)
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
leaq    -56(%rbp), %rsi
leaq    -48(%rbp), %rcx
leaq    -40(%rbp), %rdx

```

寄存器传参;  
short提升为  
int传递

寄存器传参;  
float提升  
为double  
传递

cvtsd2ss将双  
精度转换成单  
精度类型



# 例题4 参数与局部变量

```

func(i, j, f, e)
short i, short j, float f, float e;
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

输出:  
0x7ffd5859463c,  
0x7ffd58594638,  
0x7ffd58594630,  
0x7ffd58594628

```

cvtsd2ss %xmm0, %xmm0
movss %xmm0, -48(%rbp)
cvtsd2ss %xmm1, %xmm0
movss %xmm0, -56(%rbp)
movq %fs:40, %rax
movq %rax, -8(%rbp)
xorl %eax, %eax
leaq -56(%rbp), %rsi
leaq -48(%rbp), %rcx
leaq -40(%rbp), %rdx
leaq -36(%rbp), %rax
movq %rsi, %r8
movq %rax, %rsi
movl $.LC0, %edi
movl $0, %eax

```

相对于rbp的  
偏移地址及其  
暂存的变量值

-36 i  
-40 j  
-48 f  
-56 e

寄存器传参



# 低版本的gcc (如3.x, 2.x)

```
func(i,j,f,e)
short i,j; float f,e;
{
    short i1,j1; float f1,e1;
    printf(&I,&j,&f,&e);
    printf(&i1,&j1,&f1,&e1);
}
int main()
{
    short I,j; float f,e;
    func(I,j,f,e);
}
```

Sizes of short, int, long, float,  
double = 2, 4, 4, 4, 8  
(在SPARC/SUN工作站上)

- 1、参数通过栈传递，由左到右逆序入栈， i, j, f, e 地址升序
- 2、局部变量按声明的先后次序排列， i1, j1, f1, e1 地址降序

Address of i,j,f,e = ...36, ...42, ...44, ...54 (八进制数)

Address of i1,j1,f1,e1 = ...26, ...24, ...20, ...14



# 例题4 参数与局部变量

## □ 现代GCC编译器如何布局局部变量？

```
func(short i, short j, float f, float e)
{
    short i1,j1; float f1,e1;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

- 1、参数通过寄存器传递，暂存在栈中，暂存地址在局部变量之后（地址值比局部变量的地址小）
- 2、局部变量 **未** 按声明的先后次序排列？ i1, j1, f1, e1 地址升序

输出：

```
0x7ffdd906376c,0x7ffdd9063768,0x7ffdd9063764,0x7ffdd9063760
0x7ffdd906377c,0x7ffdd906377e,0x7ffdd9063780,0x7ffdd9063784
```



# 例题4 参数与局部变量

## □ 现代GCC编译器如何布局局部变量？

```

func(short i, short j, float f, float e)
{
    short i1=1, j1=2, j3[4]={5,6,7,8};
    float f1=9, e1=10;
    printf("%p,%p,%p,%p\n", &i,&j,&f,&e);
    short i2=3, j2=4;
    printf("%p,%p,%p,%p\n", &i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}

```

```

movw $1, -32(%rbp)
movw $2, -30(%rbp)
movw $5, -16(%rbp)
movw $6, -14(%rbp)
movw $7, -12(%rbp)
movw $8, -10(%rbp)
movw $3, -28(%rbp)
movw $4, -26(%rbp)
movss .LC0(%rip), %xmm0
movss %xmm0, -24(%rbp)
movss .LC1(%rip), %xmm0
movss %xmm0, -20(%rbp)
.align 4
.LC0:
.long 1091567616
.align 4
.LC1:
.long 1092616192

```

局部变量按类型分别排列，  
 相同类型的逆序布局→  
 类型在编译器中日趋重要



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

**下期预告：非局部名字访问**