



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](mailto: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

- 几种典型的编译运行时系统 (新增)
- 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



# 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

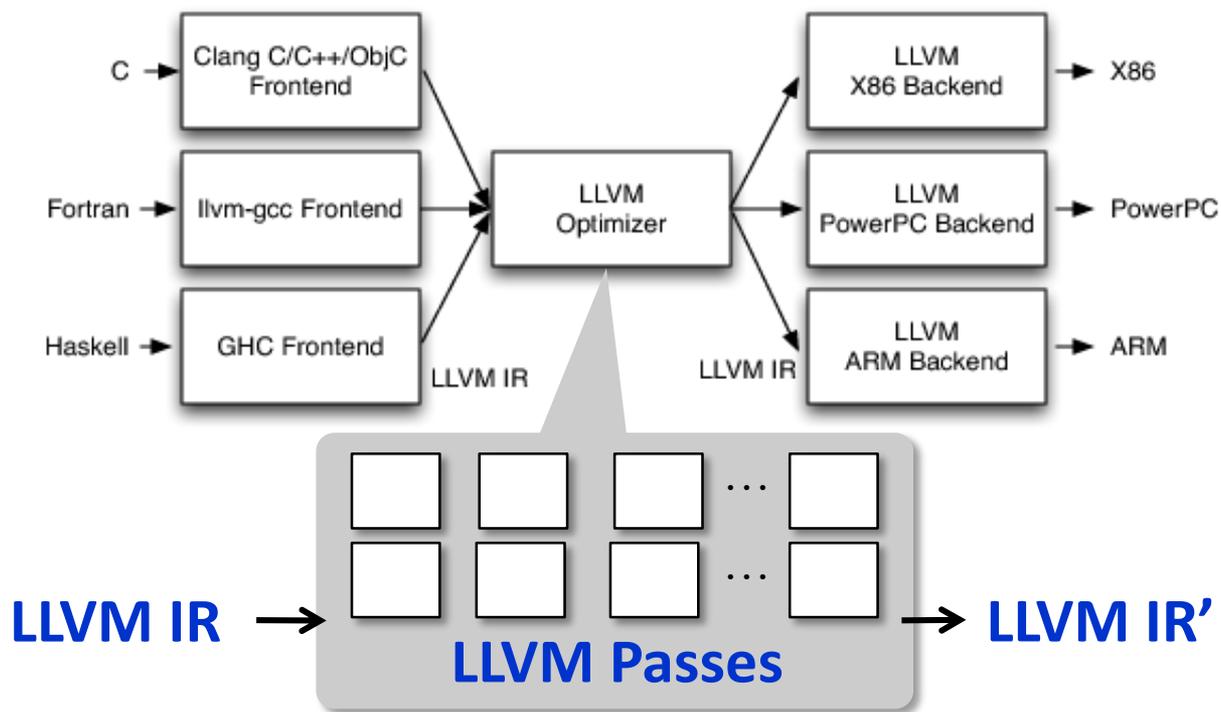
# I. 几种典型的编译和运行时系统

- C/C++编译运行时环境
- 虚拟机：Java、.Net等等
- 求值的机器模型



# C/C++编译系统

## □ Clang + LLVM





# C/C++编译系统: clang

## □ 主要模块

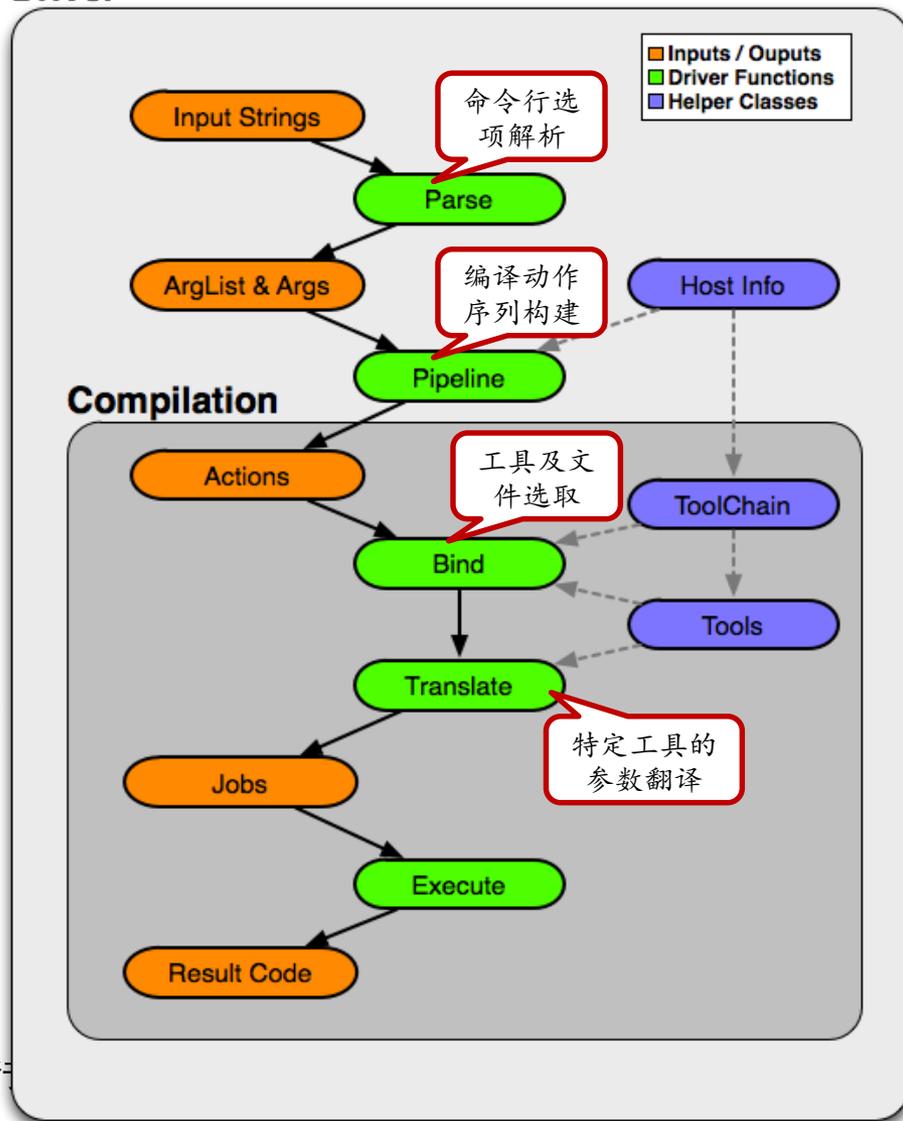
- Driver
- Lexer and Preprocessor
- Parser: 递归下降分析
- AST: [intro](#), [slides](#)
- Sema:
- CodeGen: AST → LLVM IR

## Clang Static Analyzer

Tooling(doc)提供基于clang写独立工具的库

## □ Driver

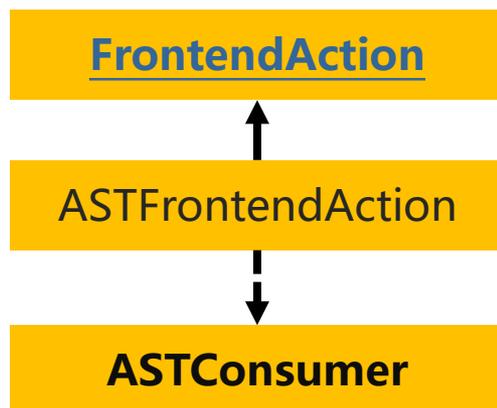
### Driver





# Clang前端要点

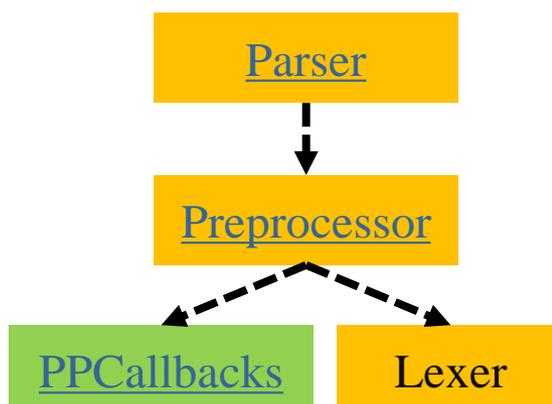
## Frontend



- PrepareToExecuteAction()
- CreateASTConsumer()
- BeginInvocation()
- BeginSourceFileAction()
- ExecuteAction()
- EndSourceFileAction()
- shouldEraseOutputFiles()

张昱：《编

## Lexer and Parser



### PPCallbacks可定制

PPCallbacks回调对象定义了预处理过程中各种可能发生的事件处理接口。继承该类型并重写相应的虚方法，即可定制对感兴趣事件的监听处理。

## AST

### 核心表示

- ASTContext
- AST Node: Decl, Stmt, Type

### 遍历访问

- [RecursiveASTVisitor](#)
- [ASTMatcher](#)

### Dump工具

clang -ast-dump -ast-dump-filter -ast-list

理



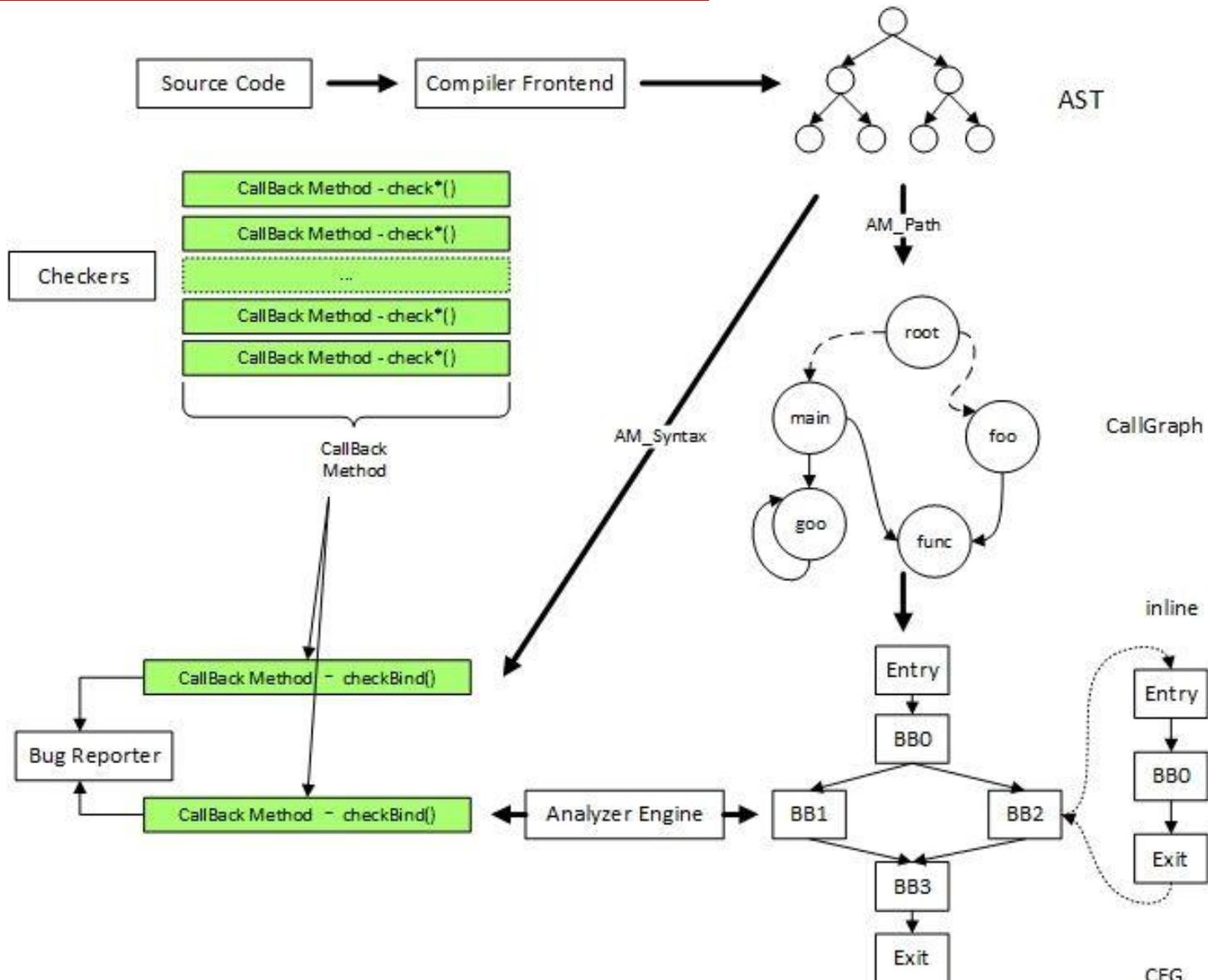
# CSA: Clang Static Analyzer

## □ 相关资料

- [A Memory Model for Static Analysis of C Programs, 2010](#)
- [Clang Static Analyzer: A Checker Developer's Guide](#)
- Building a Checker in 24 hours, [slides](#) and [video](#)
- [Clang/lib/StaticAnalyzer/README.txt](#)
- Clang Static Analyzer源代码, [clang\lib\StaticAnalyzer\Checkers](#)
- [知乎专栏Clang Static Analyzer](#)

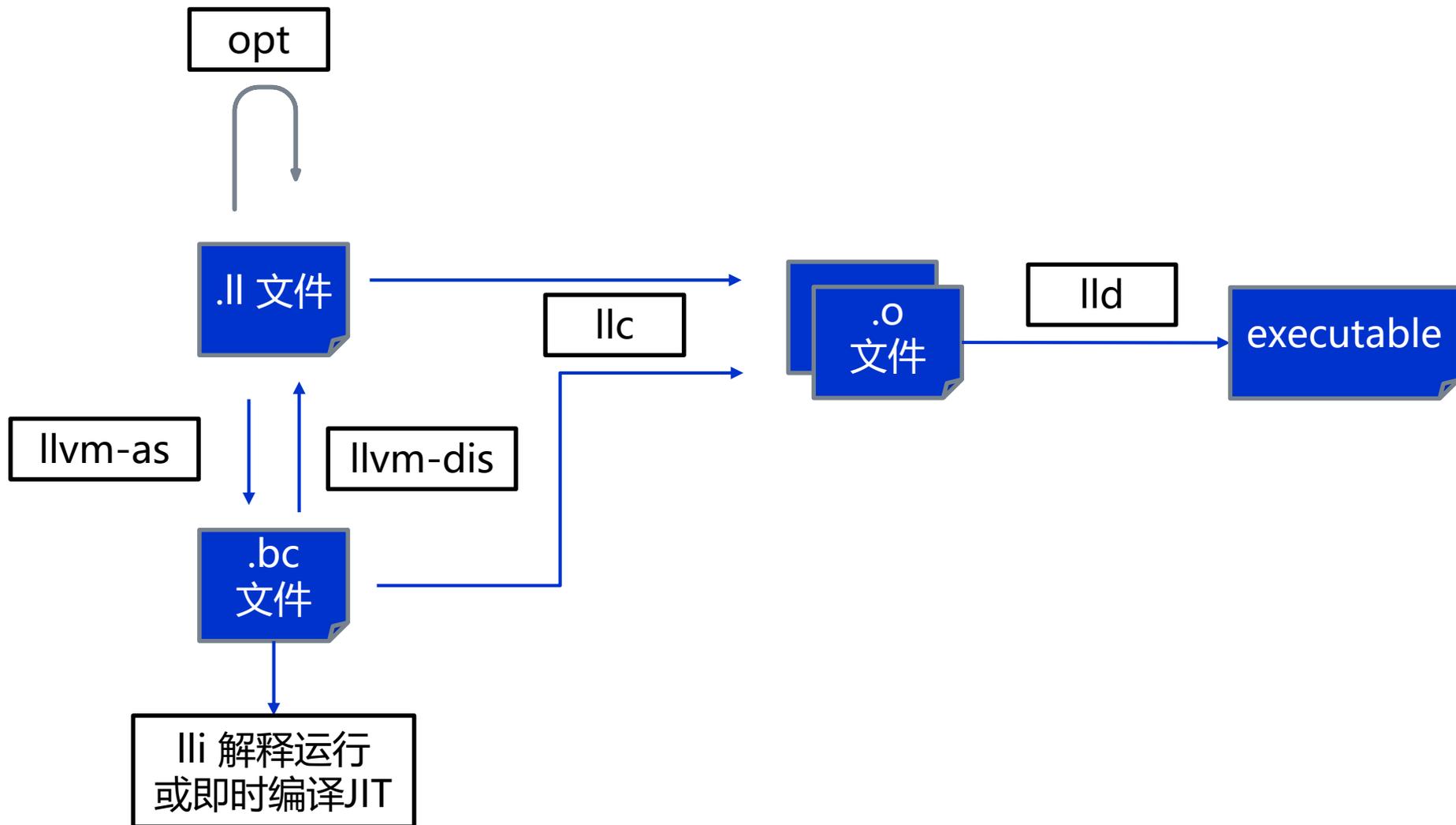


# CSA 工作流程





# LLVM 一些工具





# C/C++的运行时系统

- 代码区
- 数据区
  - 静态数据
  - 栈
  - 堆



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

# I. 几种典型的编译和运行时系统

- C/C++编译运行时环境
- 虚拟机：Java、.Net等等
- 求值的机器模型



# 虚拟机：Apache Harmony

## □ 参考资料

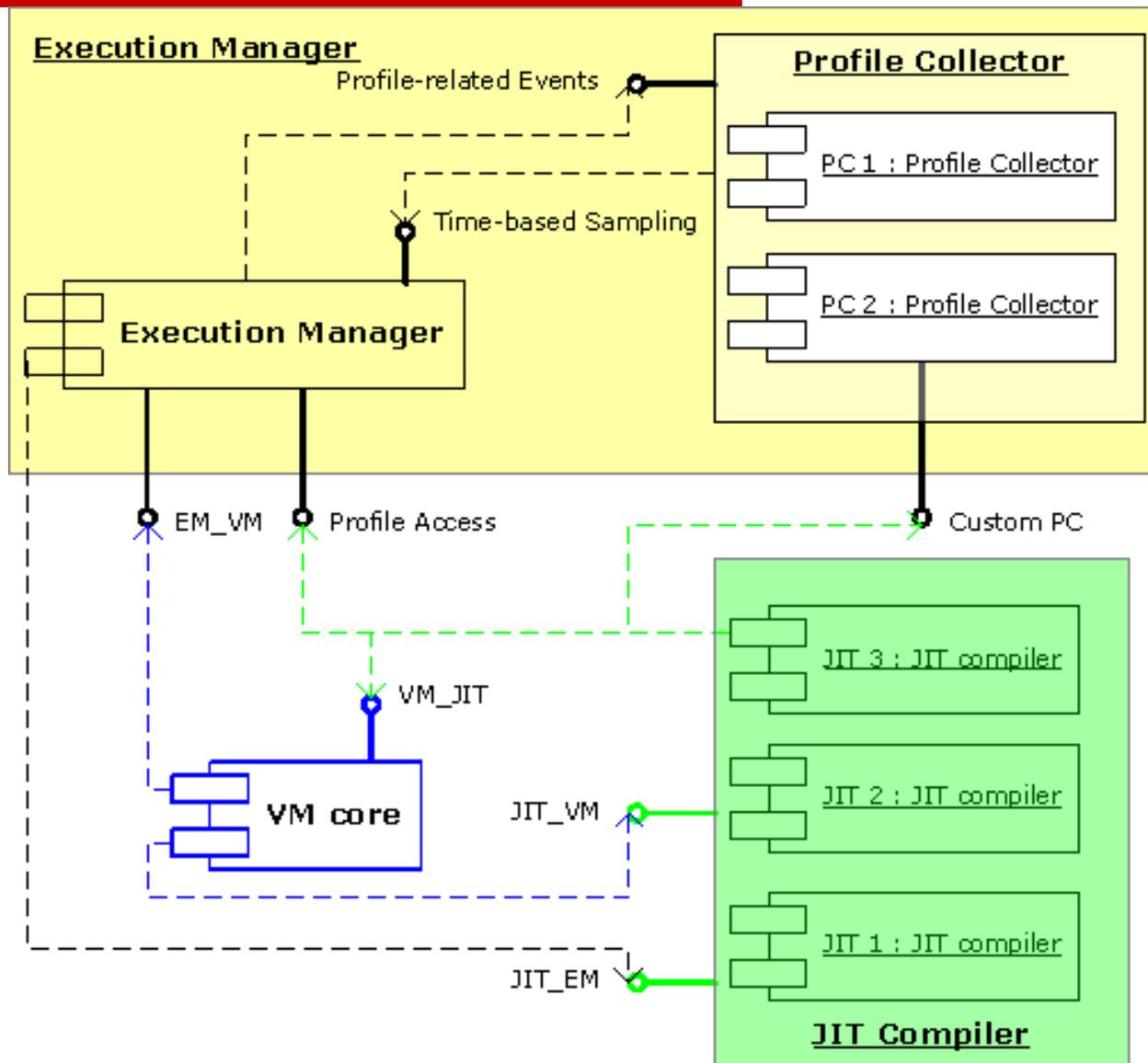
- [Apache Harmony](#): DRLVM
- [Jitrino Just-in-time Compiler](#)([Jitrino即时编译器](#))
- [Execution Manager](#) ([执行管理器](#))
- [Thread Manager](#)([线程管理](#))

## □ 类似的架构

- .Net虚拟机
- JS引擎
- .....

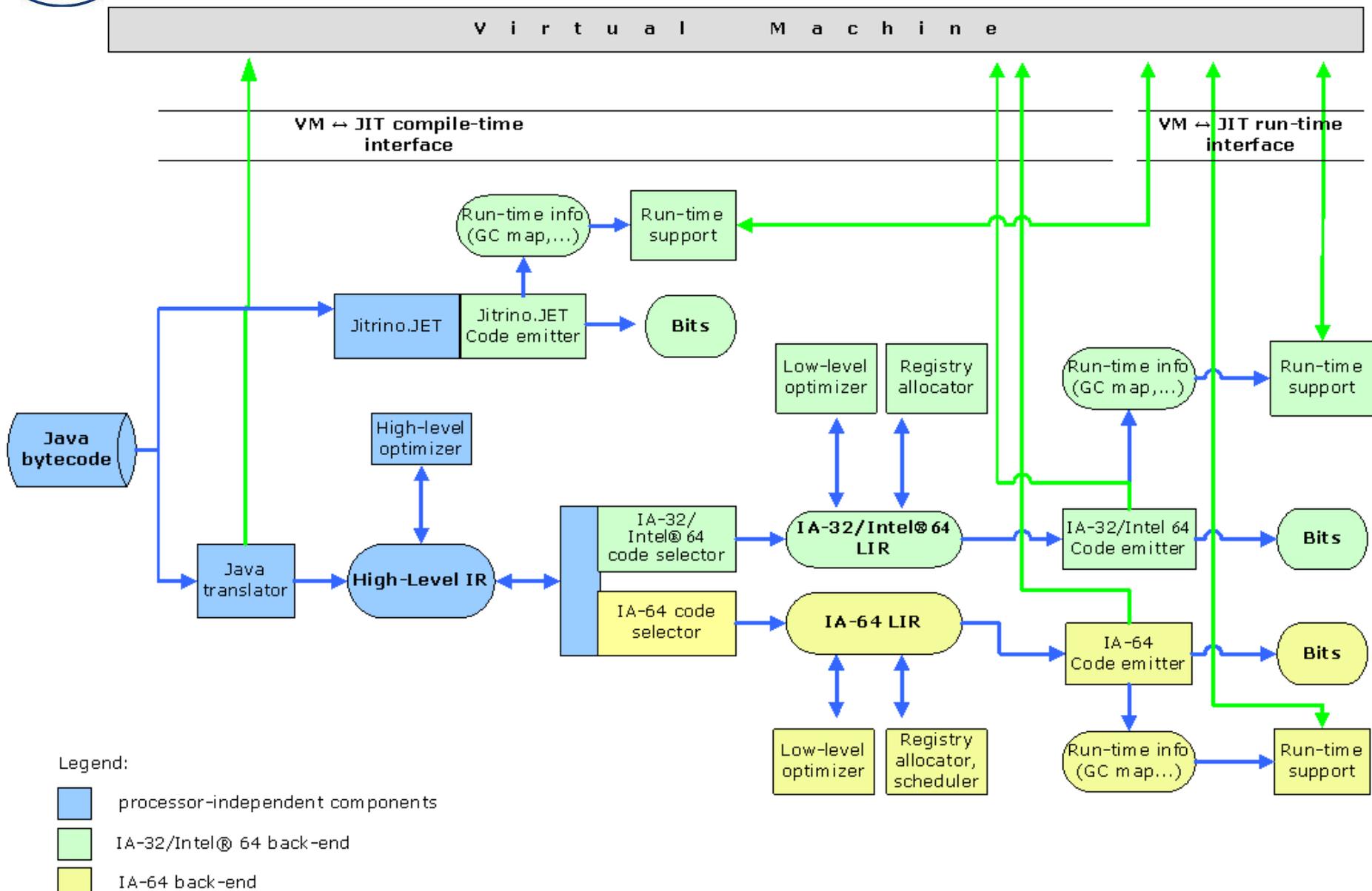


# 虚拟机：执行引擎



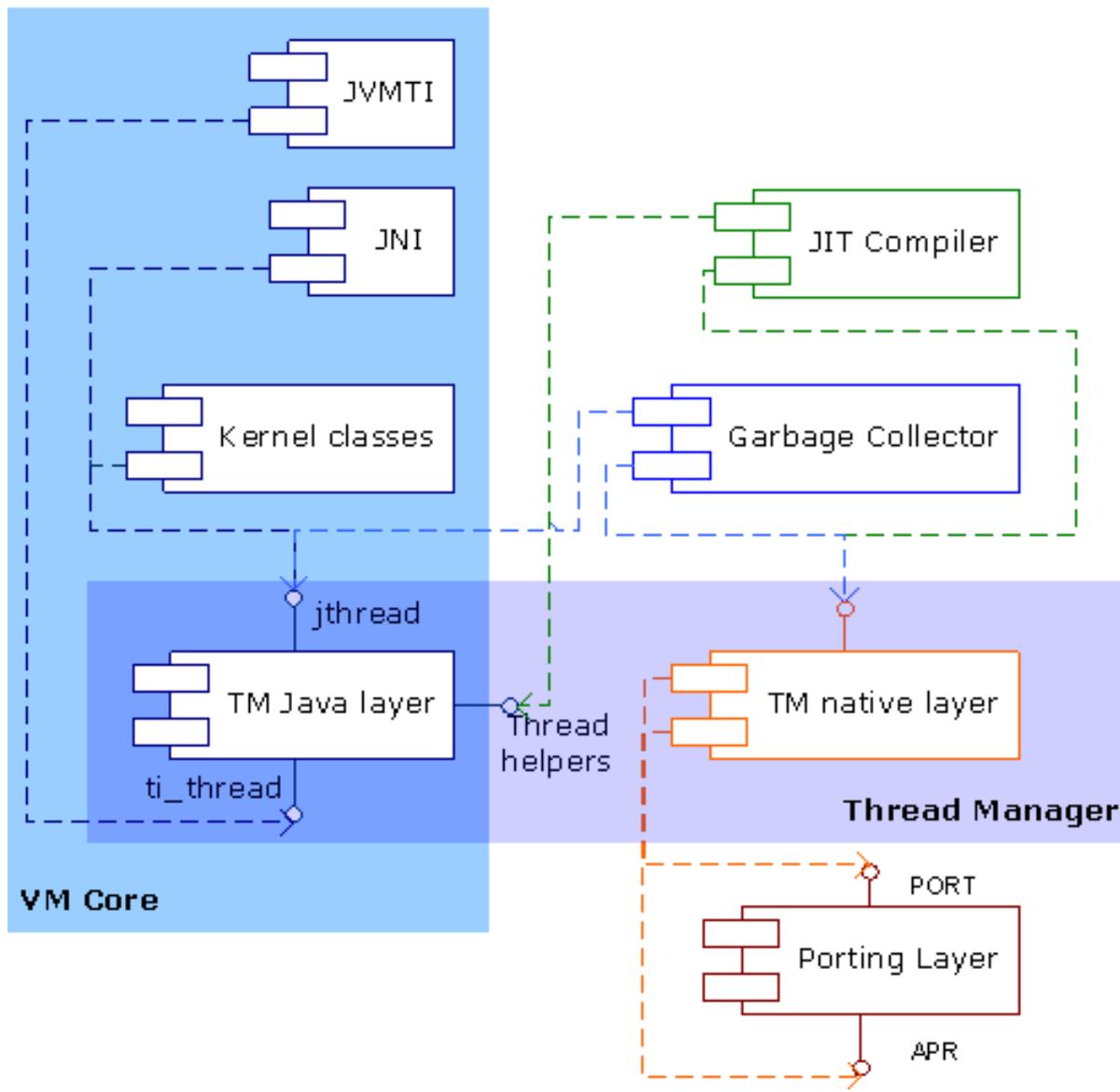


# 虚拟机：即时编译器





# 虚拟机：线程管理





# 虚拟机

## □ 本地代码管理

- 代码、栈、堆、静态数据区

## □ Java字节码管理

- VM core
- 代码cache
- 垃圾收集
- 线程管理
- .....



# I. 几种典型的编译和运行时系统

- C/C++编译运行时环境
- 虚拟机：Java、.Net等等
- 求值的机器模型



# 栈型机器

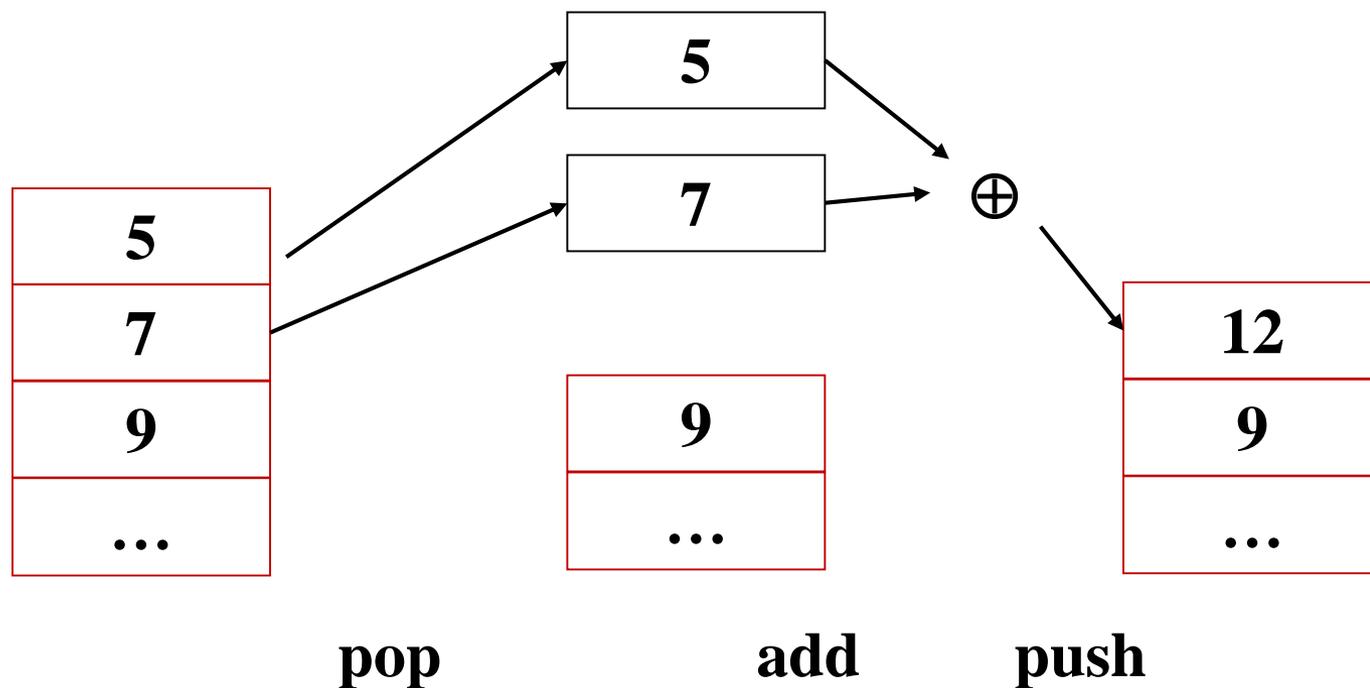
## □ 栈型机器模型

- 是一种简单的求值模型
- 没有变量和寄存器
- 中间结果存放在栈中
- 每条指令：
  - 1) 从栈顶取操作数
  - 2) 将这些操作数从栈中删除
  - 3) 在这些操作数上计算所需要的操作
  - 4) 将结果入栈



# 栈型机器操作示例

例 在栈型机器上执行加法操作





# 栈型机器操作示例

## □ 指令

■ `push i`            将 `i` 置于栈顶

■ `add`                从栈中弹出两个元素，将它们相加，再将结果入栈

## □ 计算 `7+5` 的程序

`push 7`

`push 5`

`add`

`iconst 1`

`iconst 2`

`iadd`

**Java字节码、.Net的 CLR、Pascal的P-code、Perl 5属于此类**



# 栈型机器的优点

## □ 统一的编译机制 => 编译器实现简单

每个操作从**同一地方**取操作数，并把结果放到**同一地方**

## □ 操作数的位置是隐式的

- 总是在栈顶

- 无需显式指定操作数和结果的位置

## □ 代码量小、程序更紧凑

- 如，用 **add**，而不是 **add r1, r2**

—— **Java字节码使用栈求值模型的原因**



# 栈型机器的优化

## □ 存在的问题

- add指令涉及3条访存操作：2 reads、1 write
- 栈顶被频繁访问

## □ 优化思路

- 将栈顶内容保存在寄存器中（寄存器访问速度快）  
——累加器(accumulator)
- add指令改为  $\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$   
这样就只有一条访存指令了

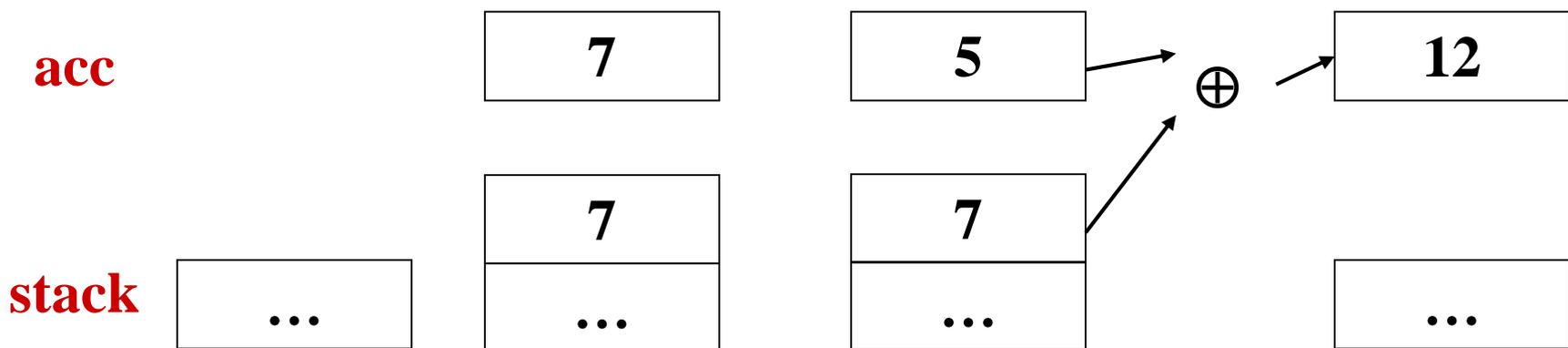


# 带累加器的栈型机器

## □ 不变式(invariants)

- 表达式的结果总是在累加器中
- $op(e_1, \dots, e_n)$ : 在计算 $e_1, \dots, e_{n-1}$ 后, 将累加器的值入栈
- 表达式值保存在栈中

计算 7+5



$acc \leftarrow 7$

push acc

$acc \leftarrow 5$

$acc \leftarrow acc + top\_of\_stack$

pop



# 寄存器型机器

## □ 寄存器型机器模型

- 操作数存在在寄存器上
- 指令中需要参数指定操作数的地址，如 **add t1, t2, t3**  
如，**Dalvik VM、Lua VM、Parrot VM**等均属于此类

## □ 相比栈型机器的优势和劣势

- 劣势：编译器复杂（寄存器分配和指派等、跟踪变量存放的场所）、代码量大、单条指令的代价高
- 优势：指令数少（执行得快）、还可以做一些栈型机器无法实现的优化（如公共表达式的结果保存到寄存器中，不必重复计算）



# 栈 vs. 寄存器

- 栈代码容易翻译到寄存器代码，反之则不成立
  - 局部变量直接映射
  - 栈存储单元映射到虚拟寄存器

栈代码	寄存器代码	说明
<code>iload 4</code>	<code>imove r10, r4</code>	; 加载局部变量4
<code>bipush 57</code>	<code>biload r11, 57</code>	; 加载立即数57
<code>iadd</code>	<code>iadd r10, r10, r11</code>	; 整数加
<code>istore 6</code>	<code>imove r6, r10</code>	; 存储结果到局部变量6
<code>iload 6</code>	<code>imove r10, r6</code>	; 加载局部变量6
<code>ifreq 7</code>	<code>ifreq r10, 7</code>	; 如果结果为0, 跳到7



# Virtual machine showdown: Stack versus registers

- VEE '05
- ACM Transactions on Architecture and Code Optimization (TACO): 4(4), 2008  
(slides)



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

## II. 运行时数据的组织与管理



# 1. 局部存储分配

- 基本概念：作用域与生存期
- 活动记录的常见布局
  - 字节寻址、类型、次序、对齐
- 程序块：同名情况的处理



# 基本概念

## □ 过程

- 过程定义、过程调用、形式参数、实在参数
- 活动、活动的生存期

## □ 名字的作用域(scope)

- **作用域**：一个声明起作用的程序部分
- 即使一个名字在程序中只声明一次，该名字在程序运行时也可能表示不同的数据对象

如 右边代码中的**n**

```
int f(int n){  
    if (n<0) error("arg<0");  
    else if (n==0) return 1;  
    else return n*f(n-1);  
}
```



# 基本概念

## □ 环境和状态

- 环境把名字映射到左值，而状态把左值映射到右值（即名字到值有两步映射）
- 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  $x$  被绑定到  $s$







# 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{          typedef struct _b{
    char  c1;                char c1;
    long  i;                 char  c2;
    char  c2;                long i;
    double f;                double f;
}a;                          }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{          typedef struct _b{
    char  c1;      0        char c1;      0
    long  i;      4        char  c2;      1
    char  c2;      8        long i;      4
    double f;    12        double f;     8
}a;                      }b;
```

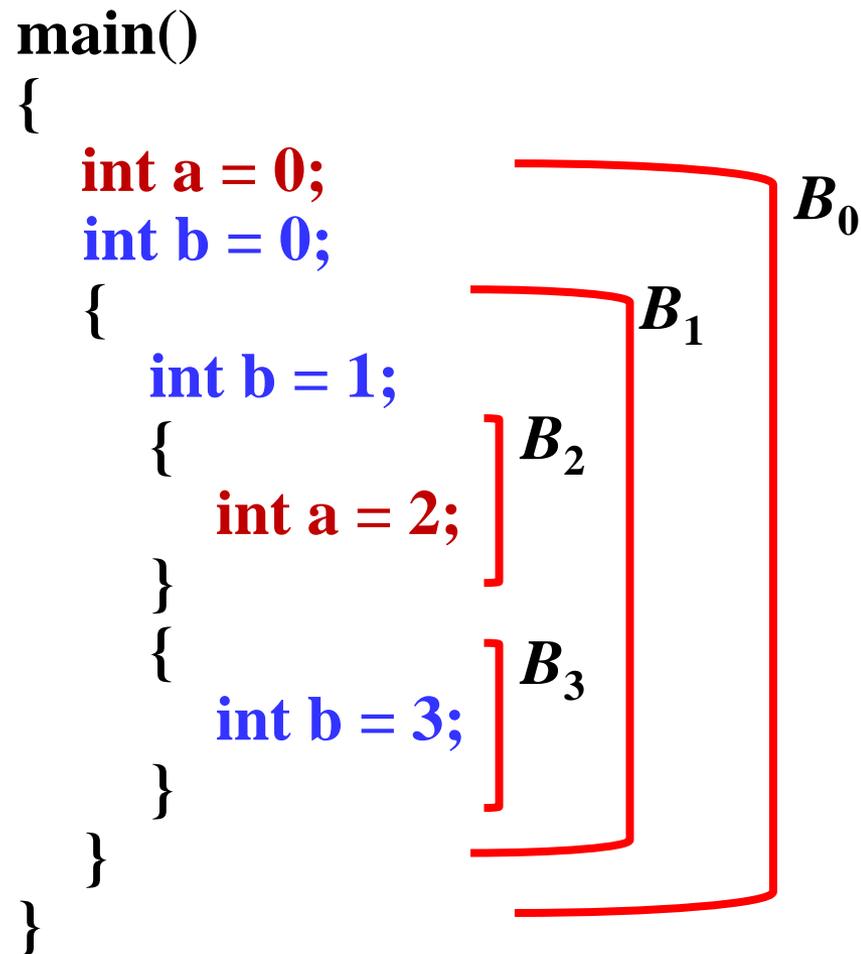
对齐： char : 1, long : 4, double : 4



# 程序块与同名变量的处理

## □ 程序块

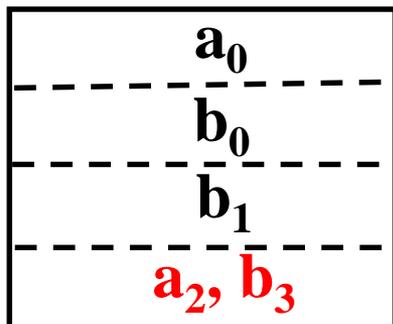
- 本身含有局部变量声明的语句
- 可以嵌套
- 最接近的嵌套作用域规则
- 并列程序块不会同时活跃
- 并列程序块的变量可以重叠分配





# 程序块与同名变量的处理

声明	作用域
<code>int a = 0;</code>	$B_0 - B_2$
<code>int b = 0;</code>	$B_0 - B_1$
<code>int b = 1;</code>	$B_1 - B_3$
<code>int a = 2;</code>	$B_2$
<code>int b = 3;</code>	$B_3$



## 重叠分配存储单元

张昱：《编译原理和技术》运行时存储空间的组织与管理

main()

```

{
  int a = 0;
  int b = 0;
  {
    int b = 1;
    {
      int a = 2;
    }
    {
      int b = 3;
    }
  }
}

```

Diagram illustrating overlapping storage units for the code blocks. Red brackets on the right side group the code blocks into four regions labeled  $B_0$ ,  $B_1$ ,  $B_2$ , and  $B_3$ .  $B_0$  covers the entire main() function.  $B_1$  covers the innermost block containing `int b = 1;` and `int b = 3;`.  $B_2$  covers the block containing `int a = 2;`.  $B_3 covers the block containing `int b = 3;`.$



## 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

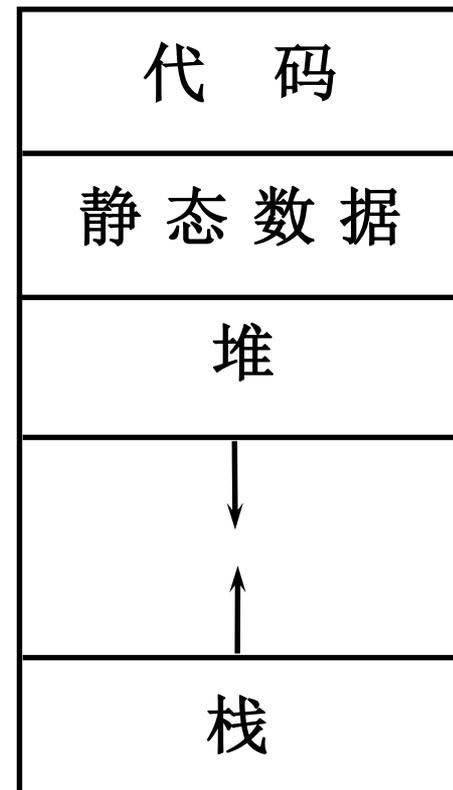
## □ 静态分配

- 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
- 绑定的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

低地址

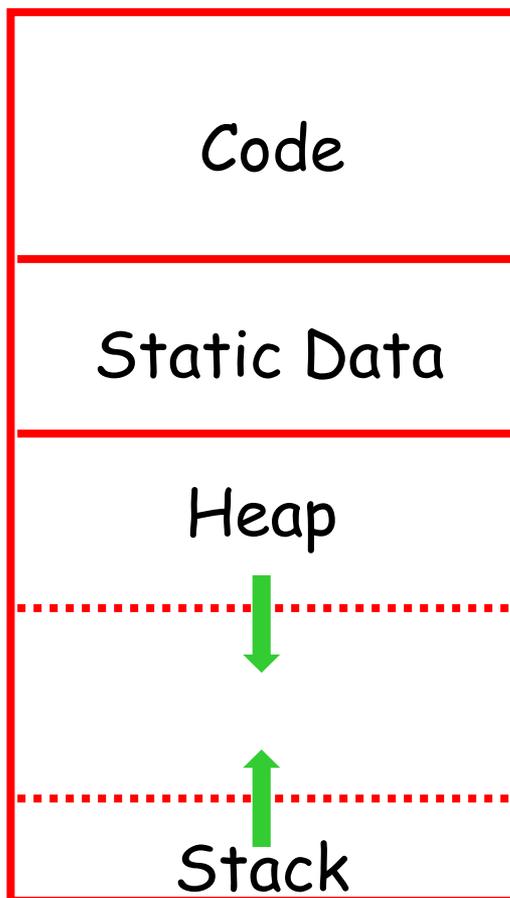


高地址



# Linux的地址布局

Memory



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  $0xC0000000$

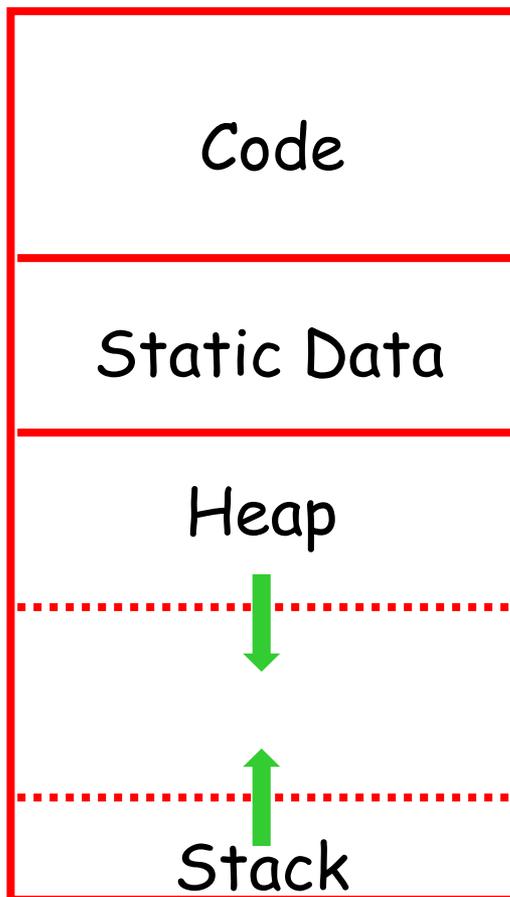
`TASK_SIZE`



# Linux的地址布局

低地址  $0x000000000000400000$

Memory



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  
 $0-0x7FFFFFFFFFFFFFFF$
- 内核空间: 64TB  
 $0xFFFF880000000000-0xFFFFc7FFFFFFFFFFFF$

高地址  $0x00007FFFFFFFFF0000$   
 $TASK\_SIZE$



# C语言的存储分配

## □ 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

(改变作用域)

## □ 声明在函数里面

- 静态局部变量                -- 也是静态分配
- 自动变量                    -- 不能静态分配，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**

张昱：《编译原理和技术》运行时存储空间的组织与

```
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) ); ...
```

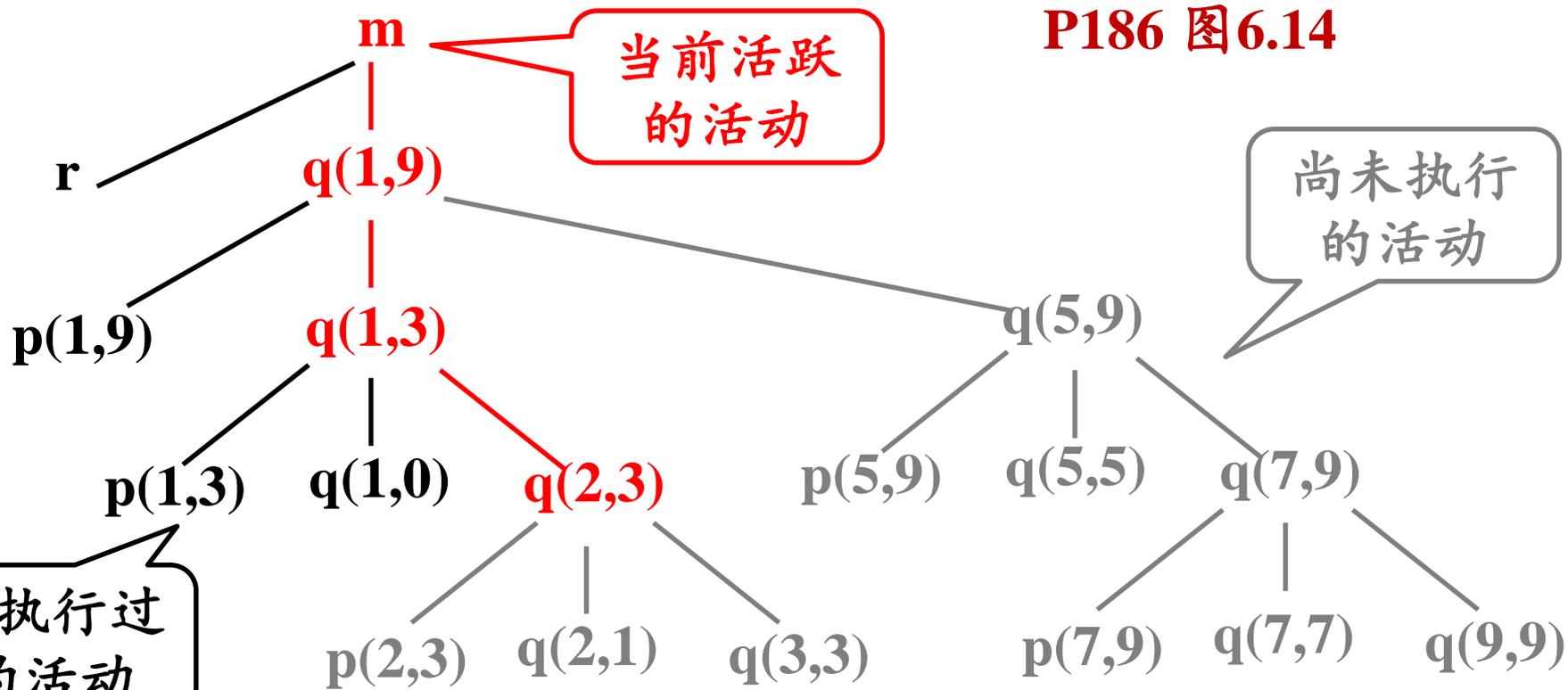
```
int n=3;
int f3(){
    if (n==0) return 1;
    else return n*f3(n-1);
}
... print ( f3(n) ); ...
```



# 活动树和运行栈

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

P186 图6.14





# 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

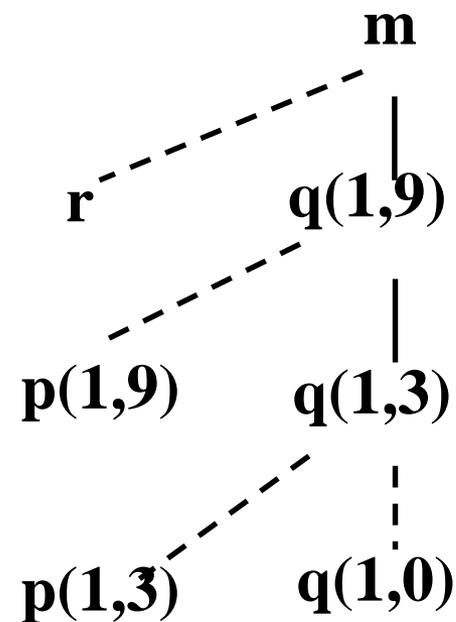
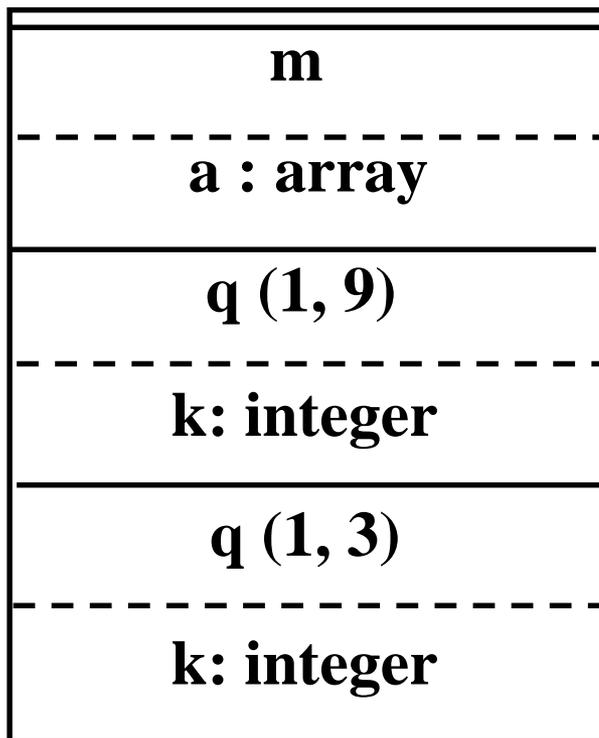
## □ 运行栈

- 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





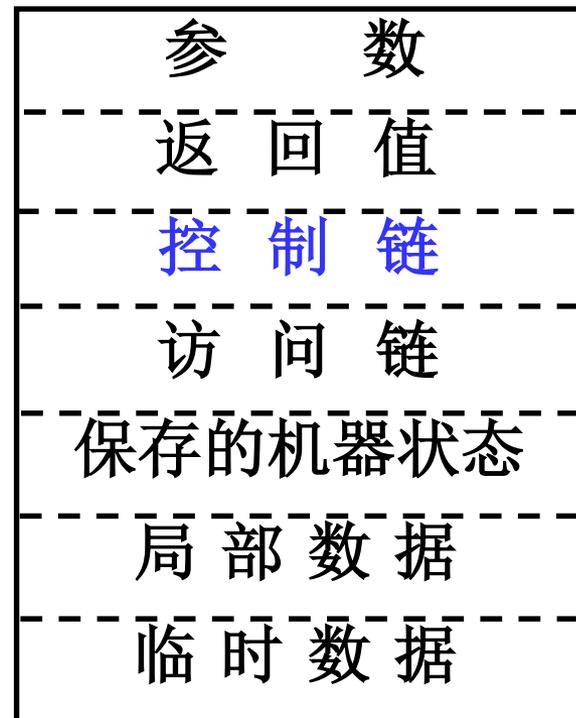
# 过程调用与返回和活动记录的设计

## □ 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

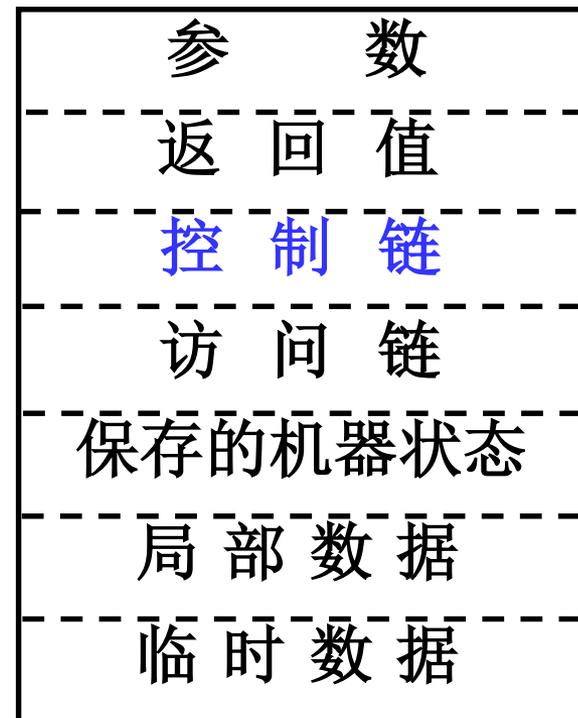




# 过程调用与返回和活动记录的设计

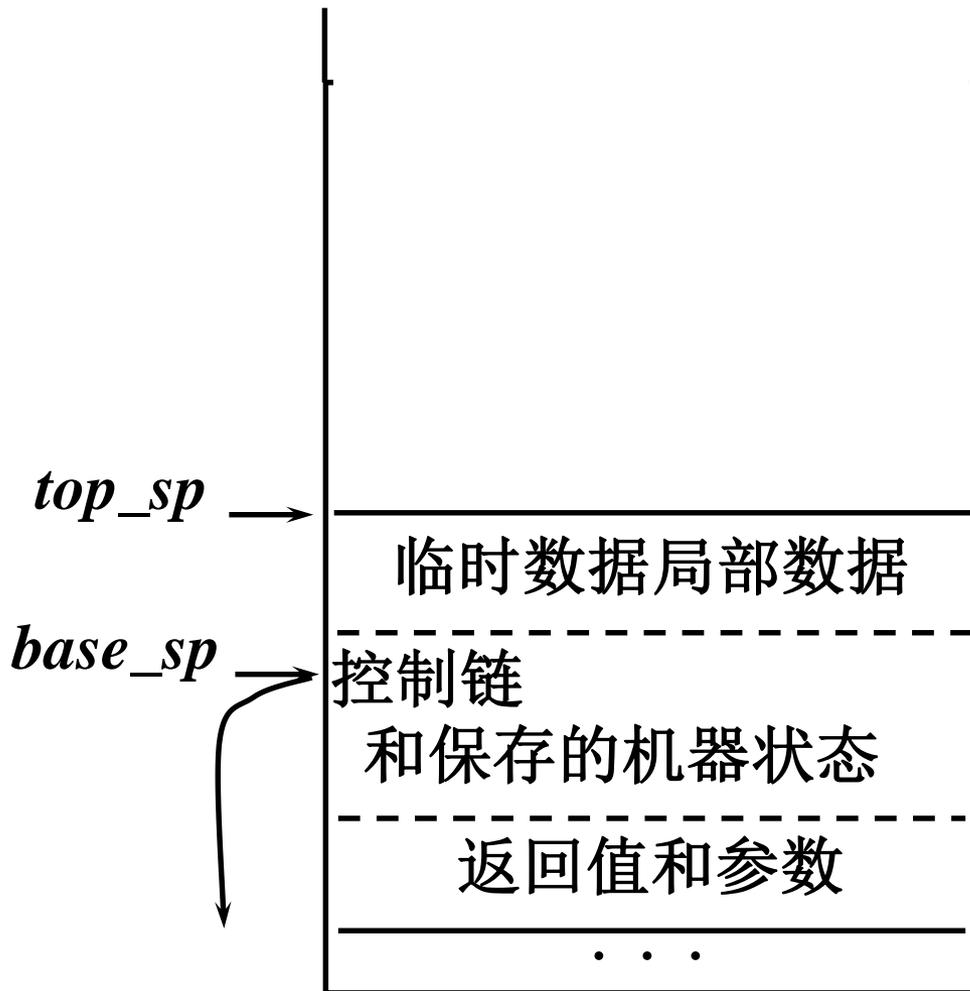
## □ 设计的一些原则

- 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
- 用同样的代码来执行各个活动的保存和恢复





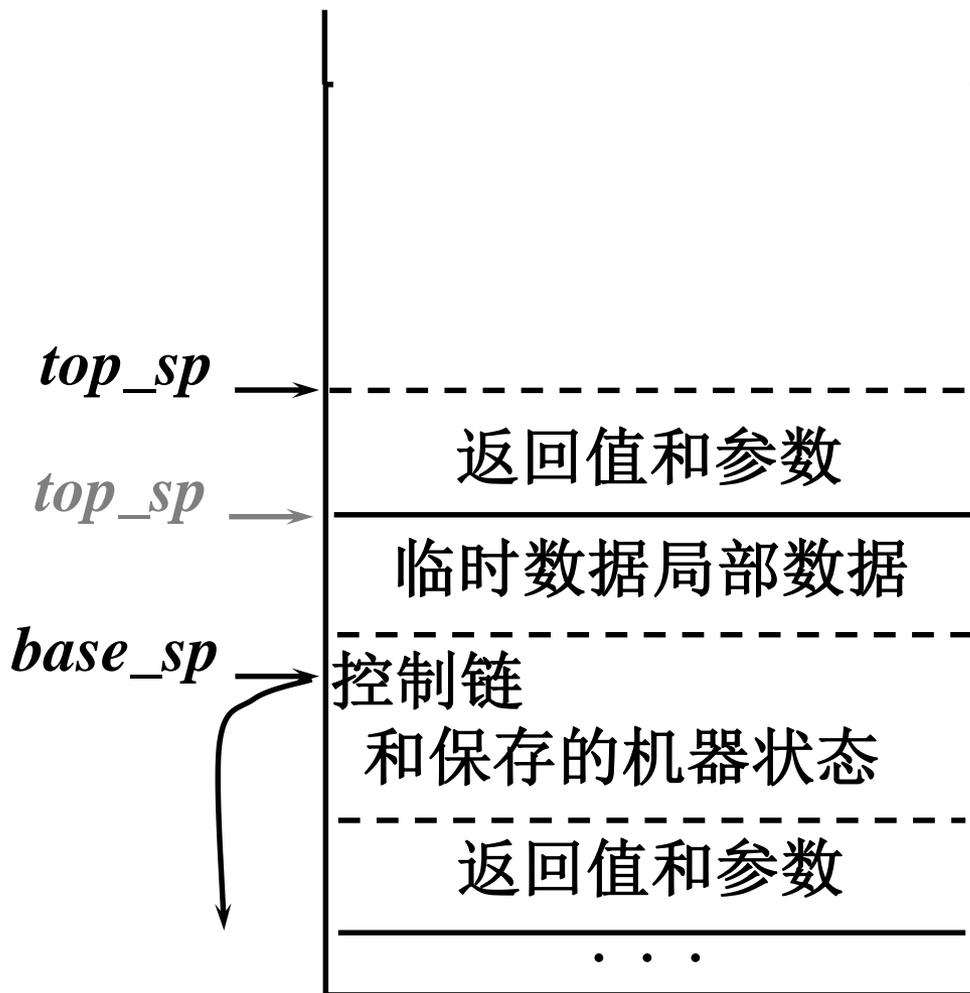
# 过程调用序列：p调用q



- ✓  $top\_sp$ : 栈顶寄存器，如esp、rsp
- ✓  $base\_sp$ : 基址寄存器，如ebp、rbp
- ✓ PC: 程序计数器，如eip、rip



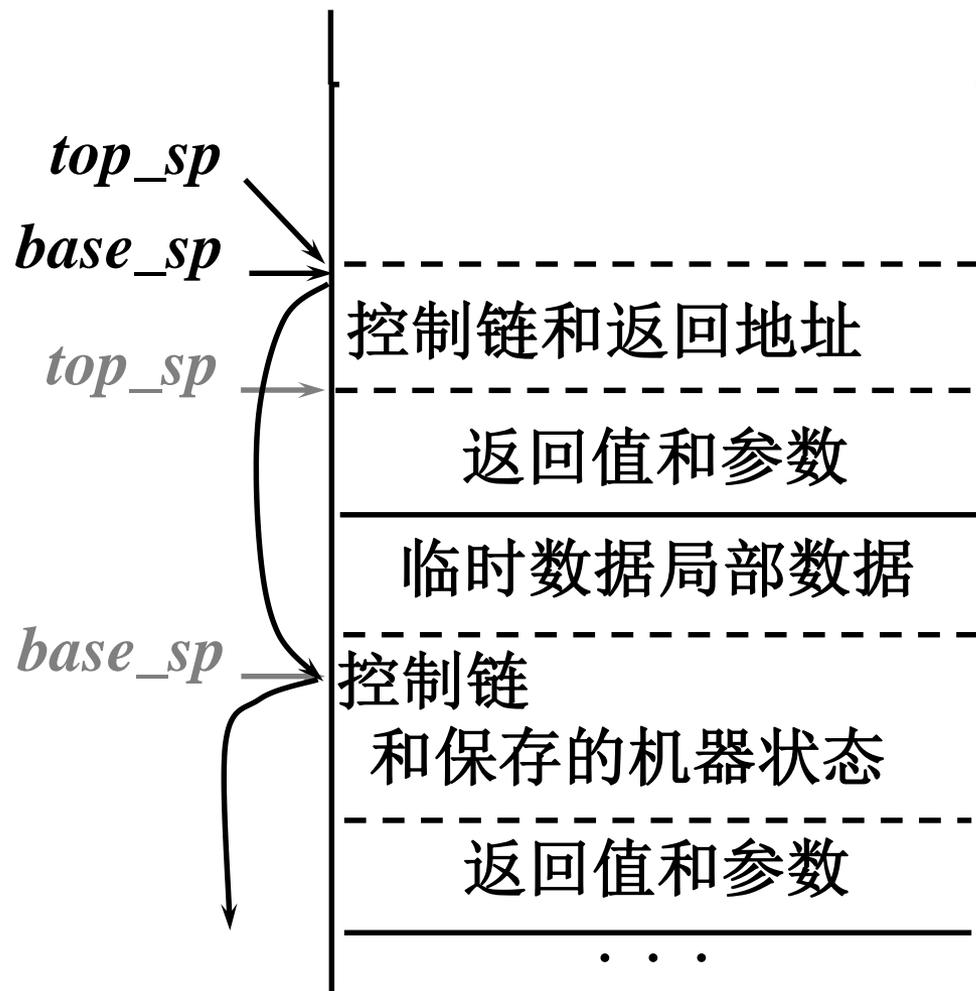
# 过程调用序列：p调用q



(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



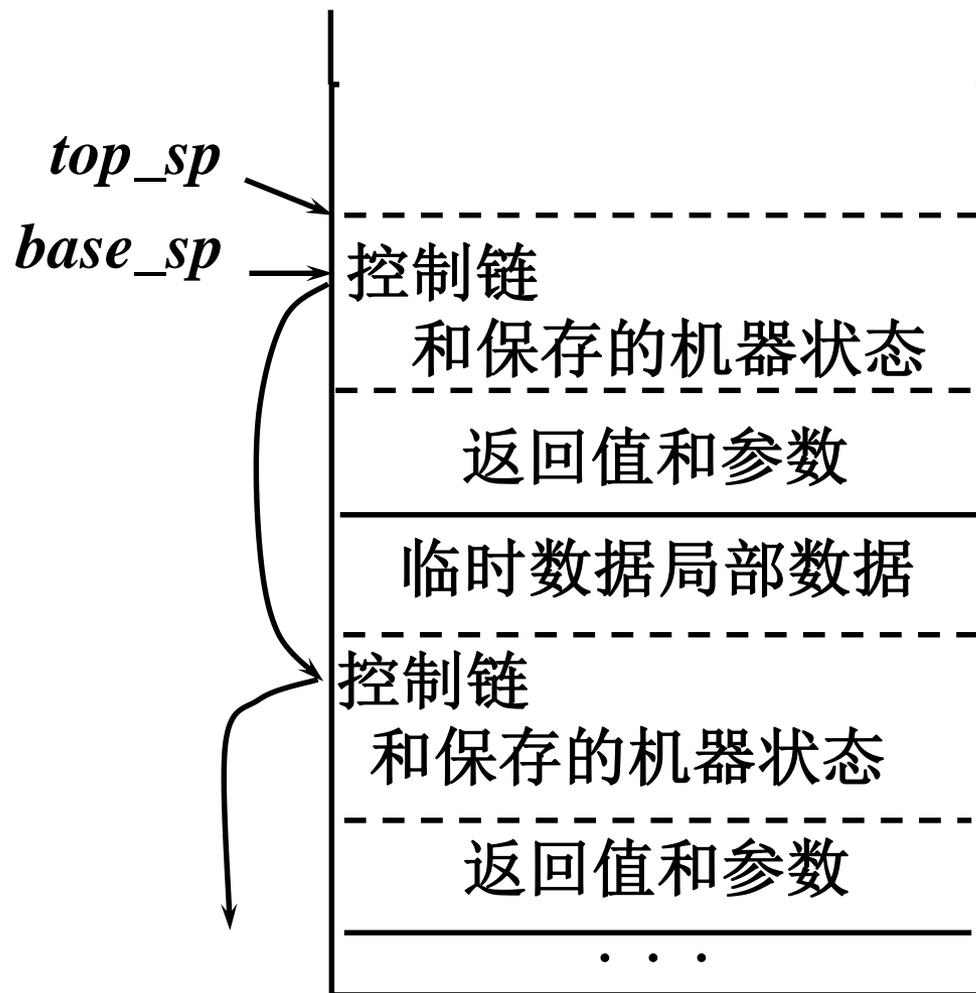
# 过程调用序列：p调用q



(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



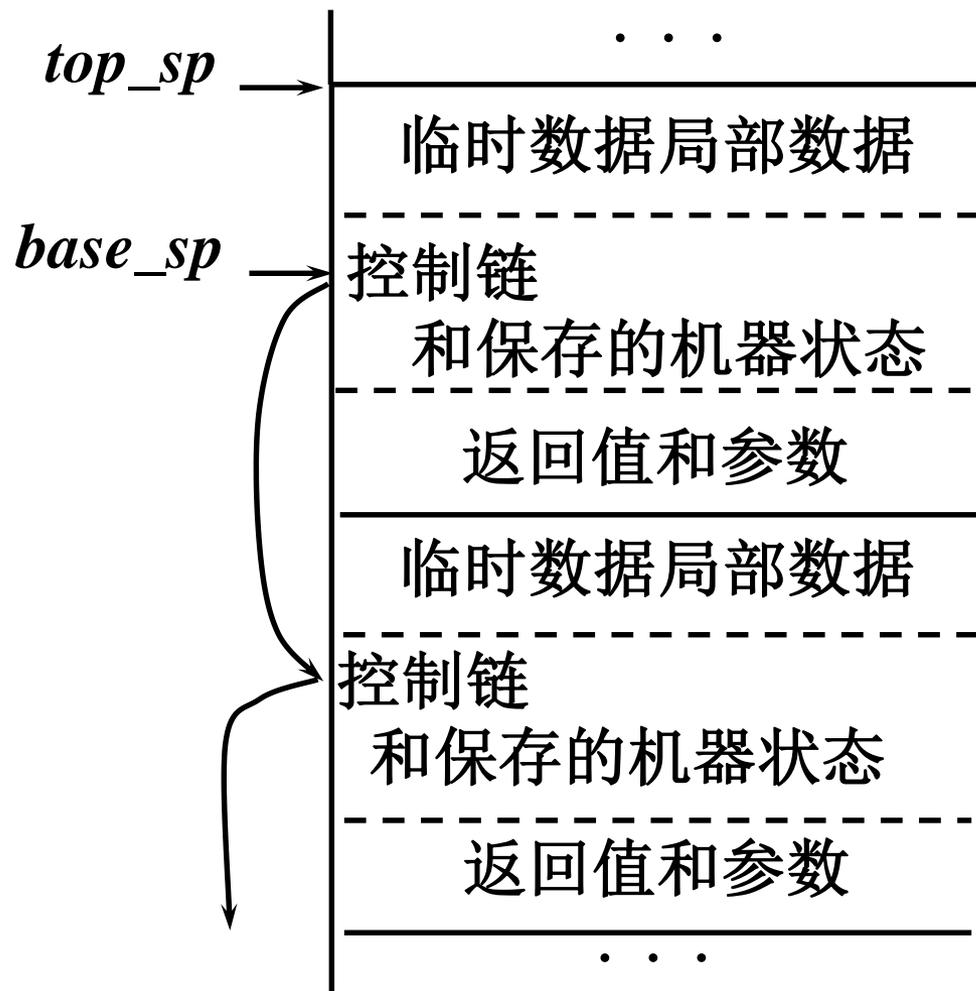
# 过程调用序列：p调用q



(3) q保存寄存器的值和其他机器状态信息



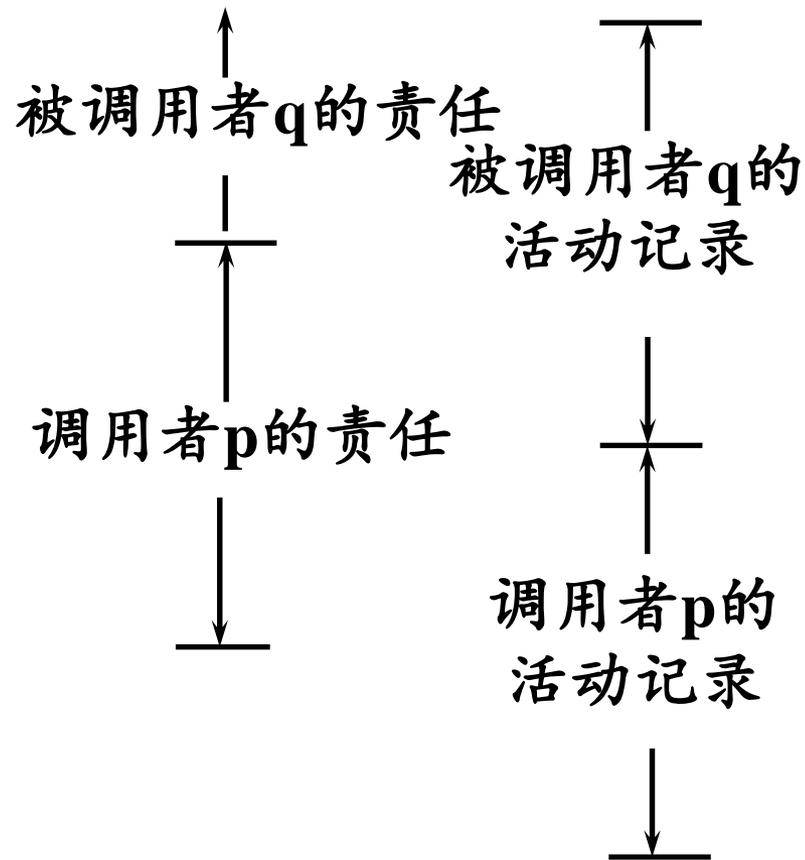
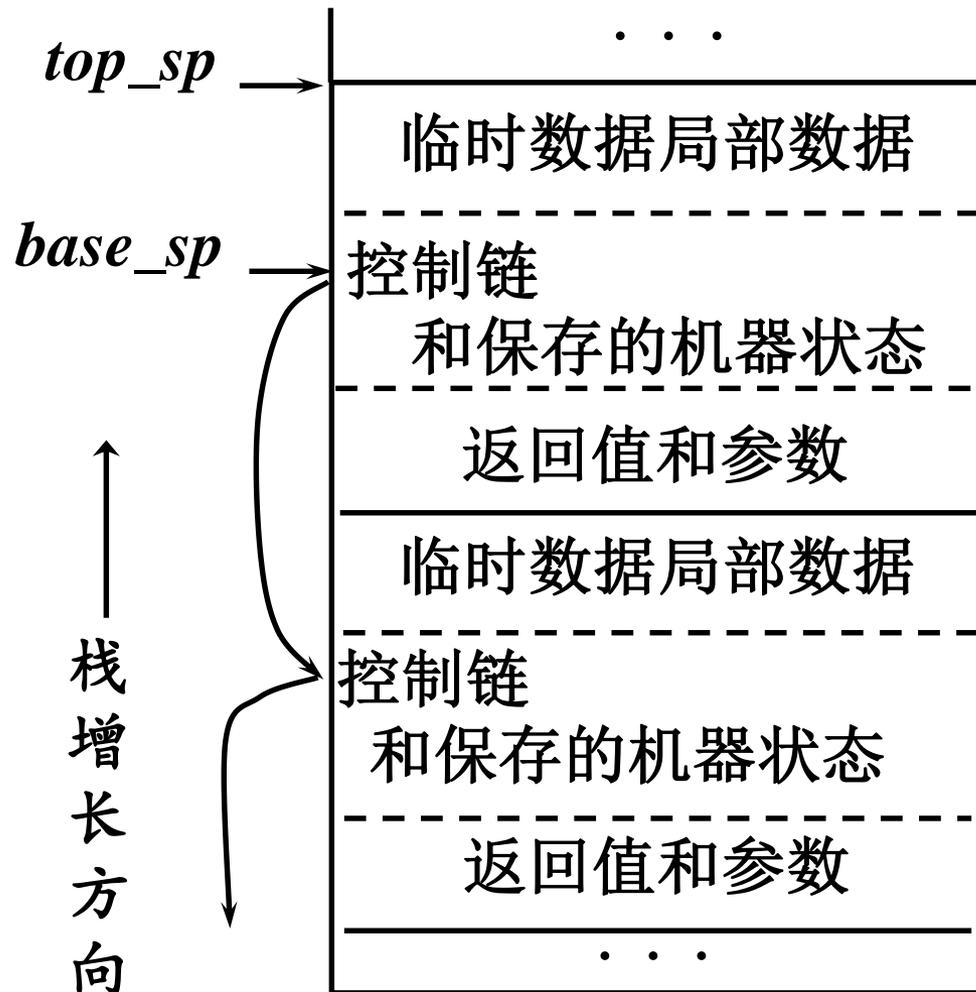
# 过程调用序列：p调用q



(4) q根据局部数据域和临时数据域的大小增加 *top\_sp* 的值，初始化它的局部数据，并开始执行过程体

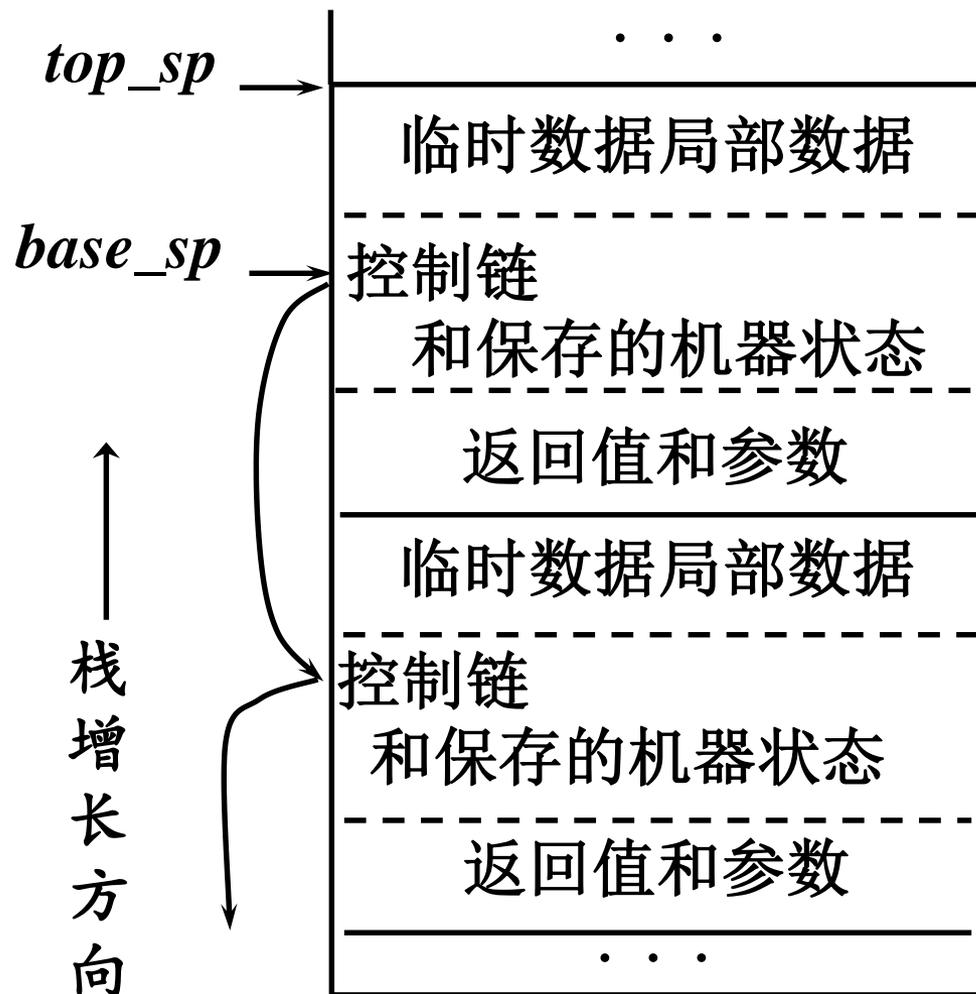


# 调用者p和被调用者q的任务划分



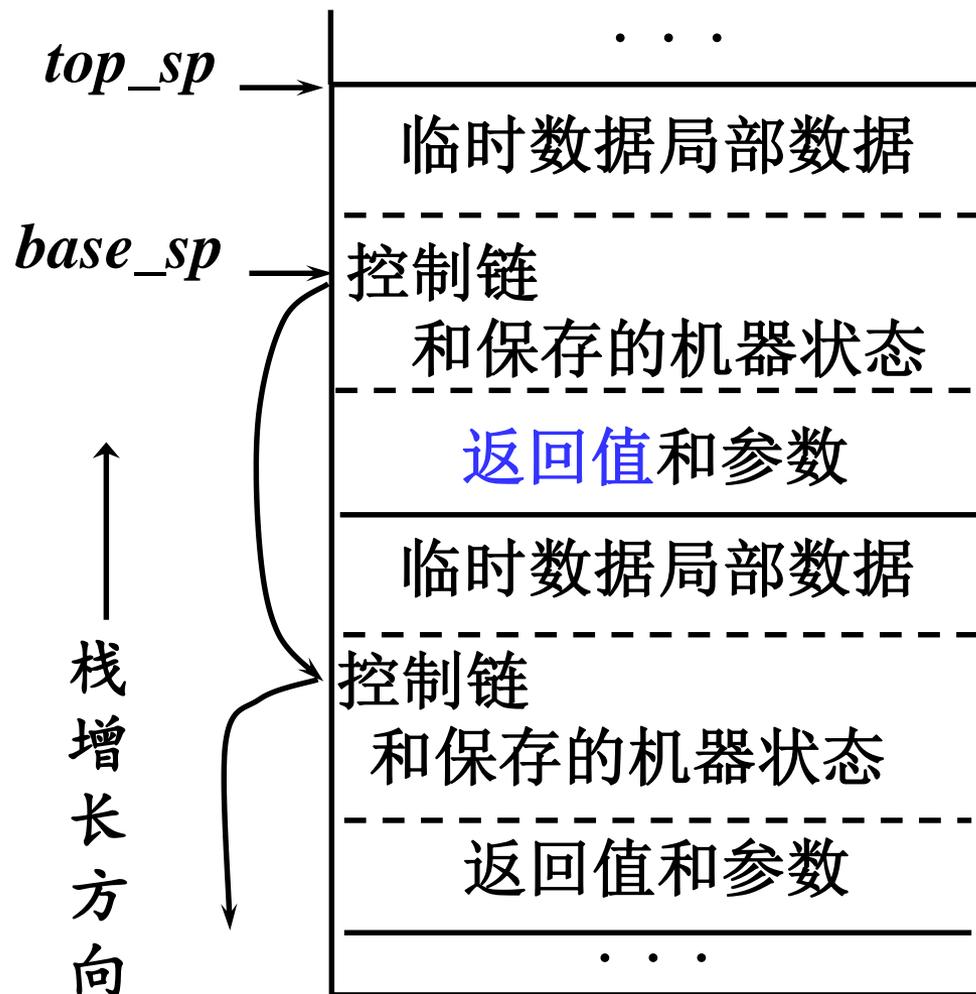


# 过程返回序列：p调用q





# 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

## □ 32位系统

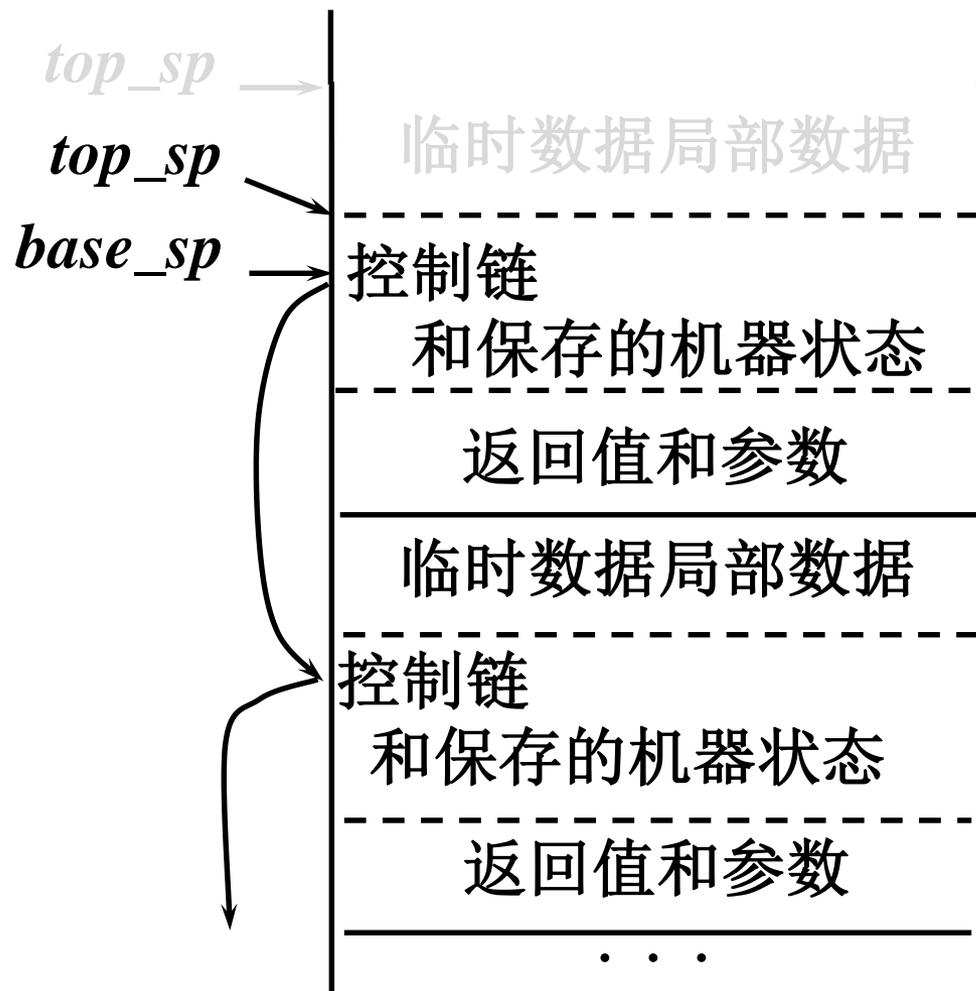
- 32位整型返回值： `eax`
- 64位整型返回值： 低32位 `eax`， 高32位 `edx`
- 浮点类型的返回值： 浮点寄存器 `st(0)`

## □ 64位系统

- 整型： `rax`
- 浮点型： 浮点寄存器 `st(0)`



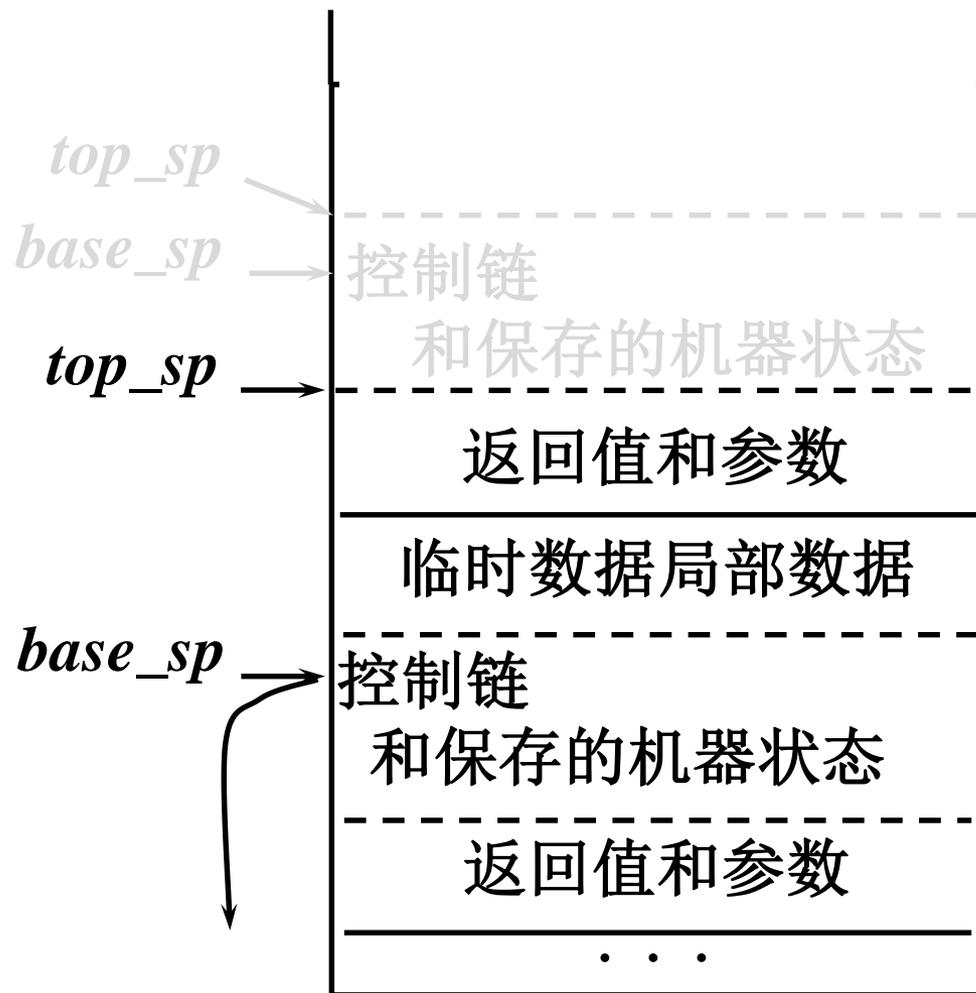
# 过程返回序列：p调用q



(2) q对应调用序列的步骤(4)，减小 $top\_sp$ 的值



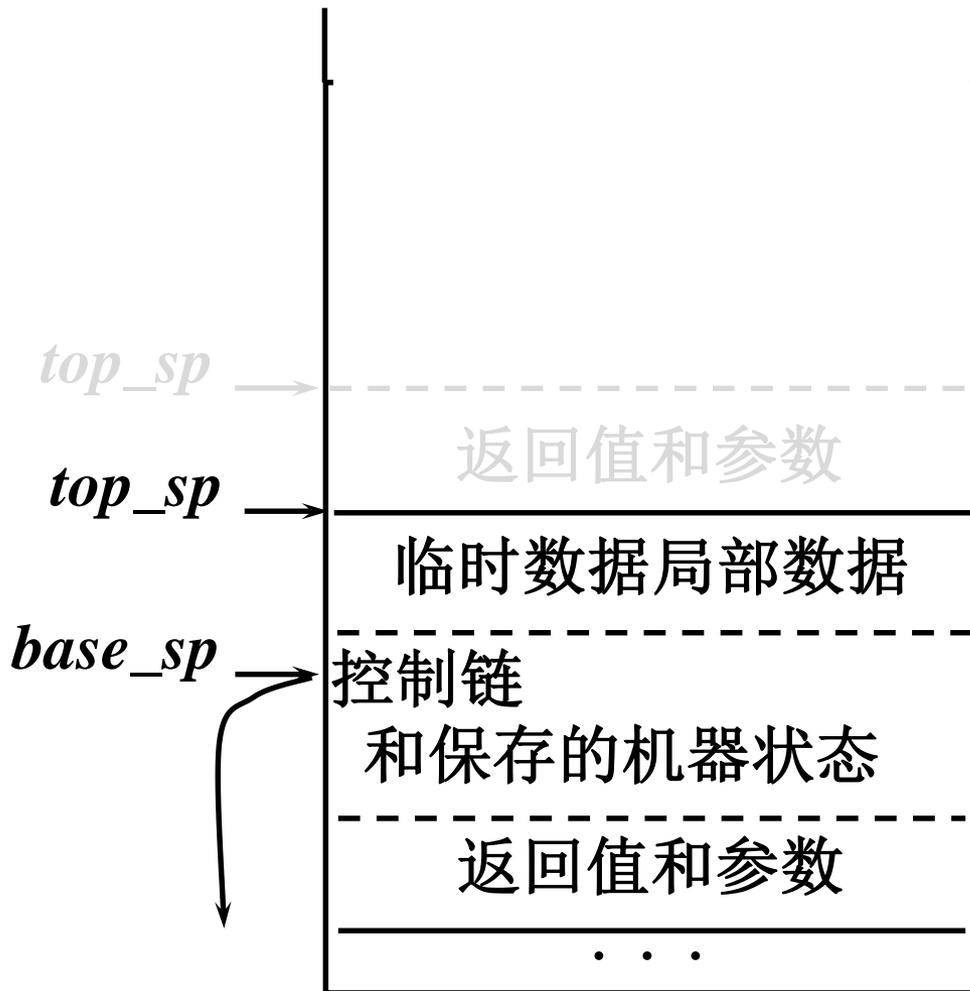
# 过程返回序列：p调用q



(3) q恢复寄存器(包括 *base\_sp*)和机器状态，返回p



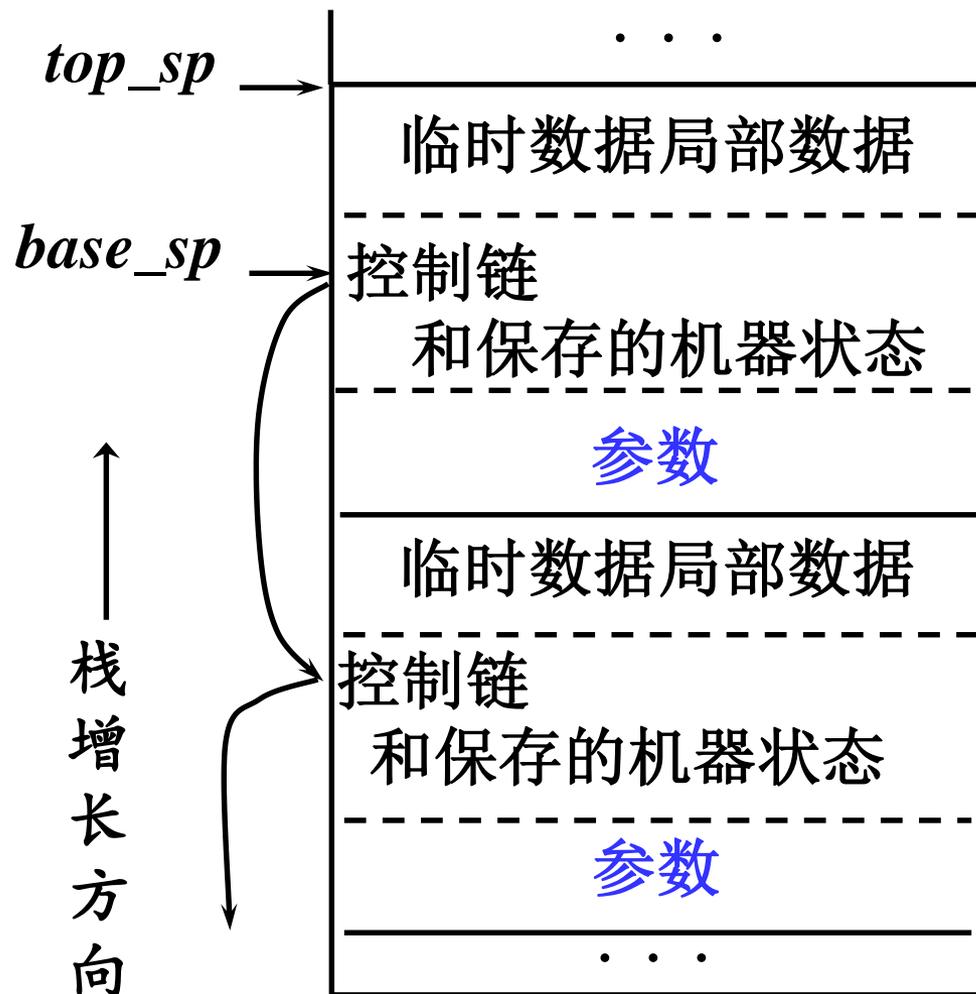
# 过程返回序列：p调用q



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



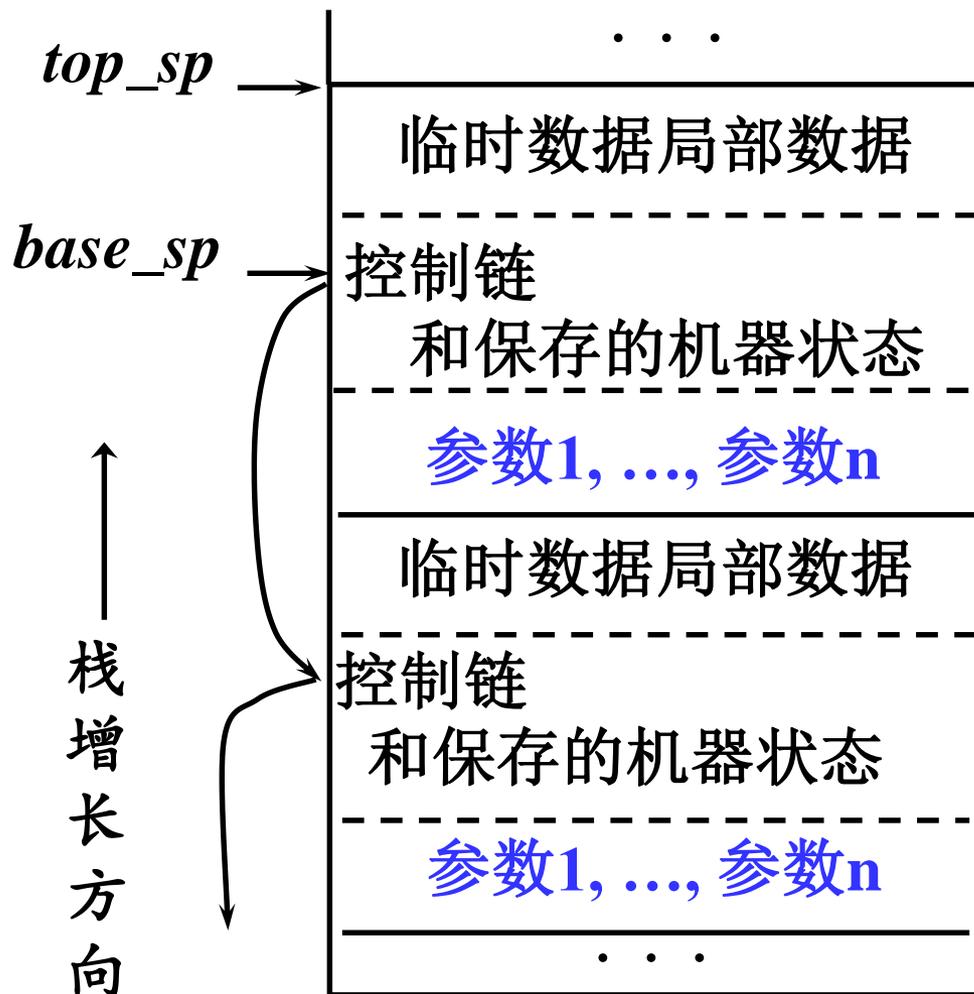
# 过程的参数个数可变的情况



(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？

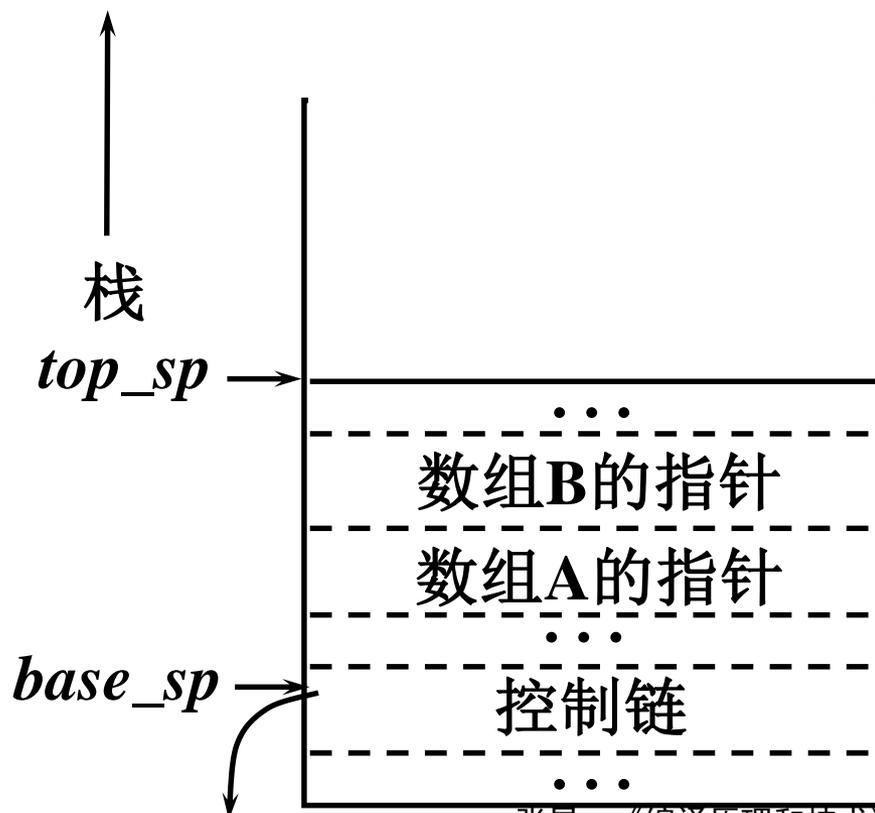
■ 先分配存放数组指针的单元，对数组的访问通过指针间接访问

■ 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

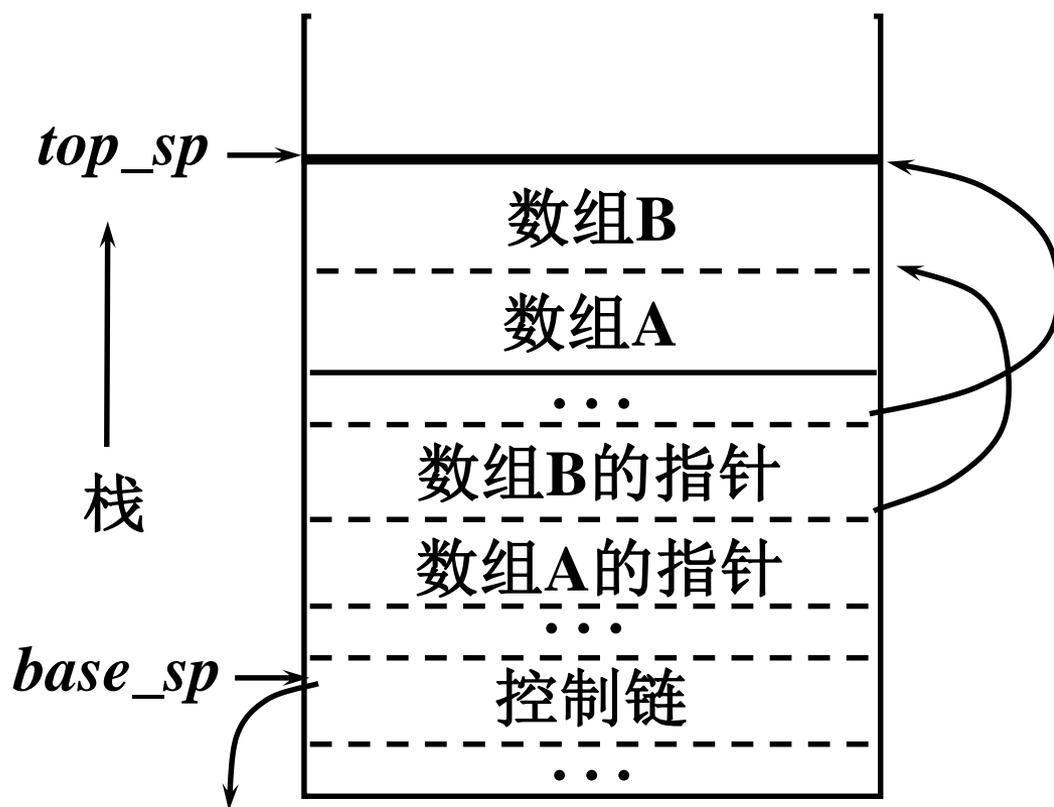


(1) 编译时，在活动记录中为这样的数组分配存放数组指针的单元



# 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

(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



# 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!  
}
```



## 3. 非局部名字访问

- 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](#)

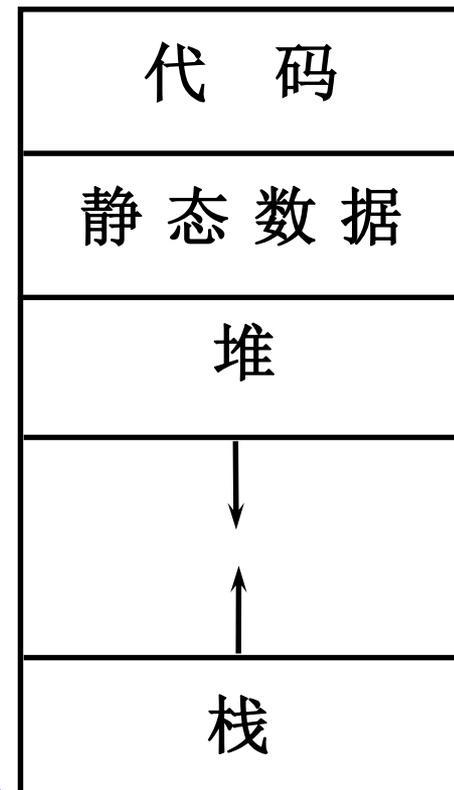
高效的并发内存分配器 [jemalloc](#),

[TBBmalloc](#), [TCMalloc \(gperftools\)](#)

- Java: new、Garbage Collection 高地址

[Richard Jones's the Garbage Collection Page](#)

低地址





# 静态作用域

## □ 无过程嵌套时

- 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

- 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



# 有过程嵌套的静态作用域

- 非局部名字访问：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

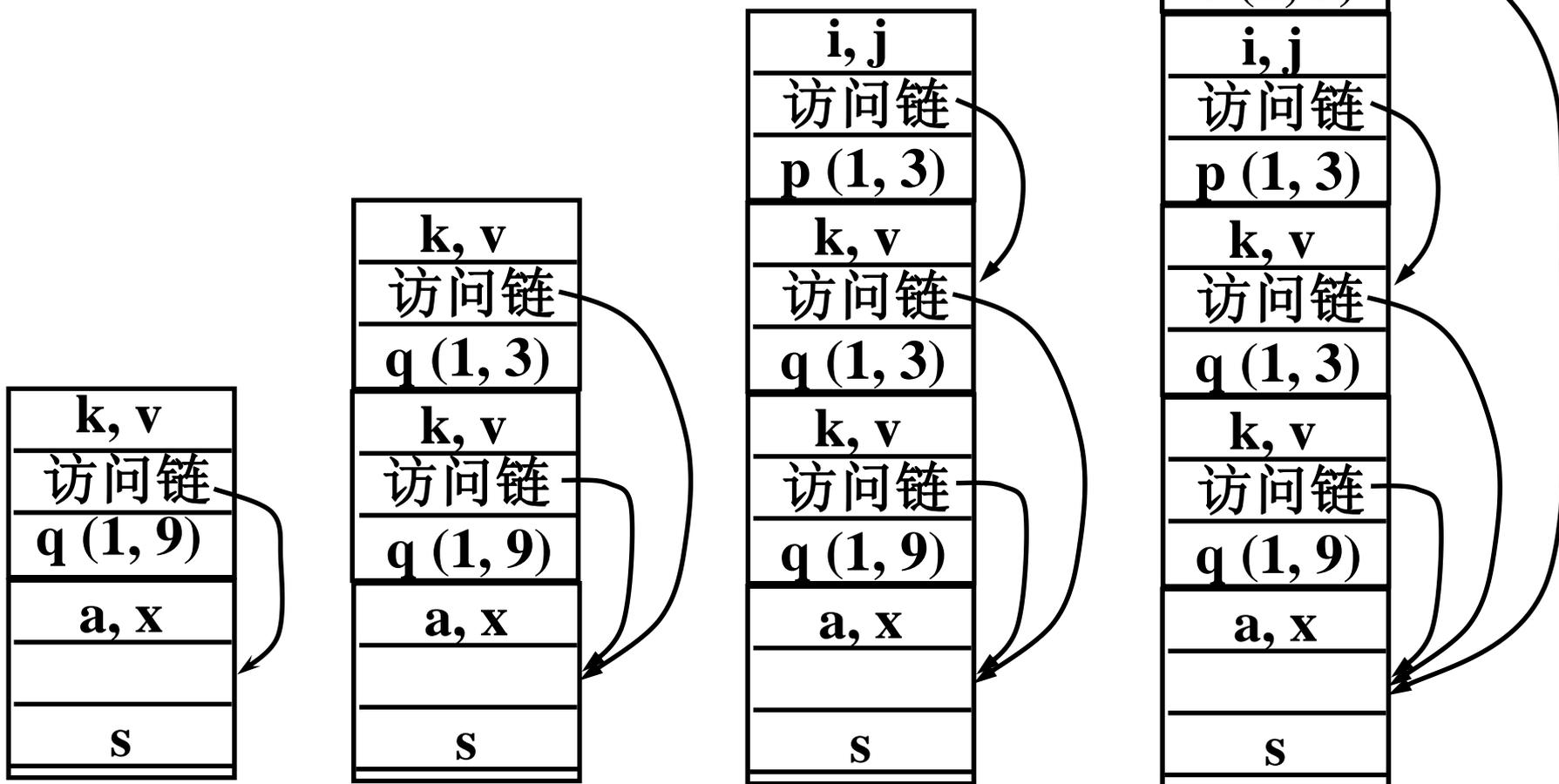
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

- **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





## 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 $a$ ， $n_a \leq n_p$ ，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)  $n_p \geq n_x$  的情况

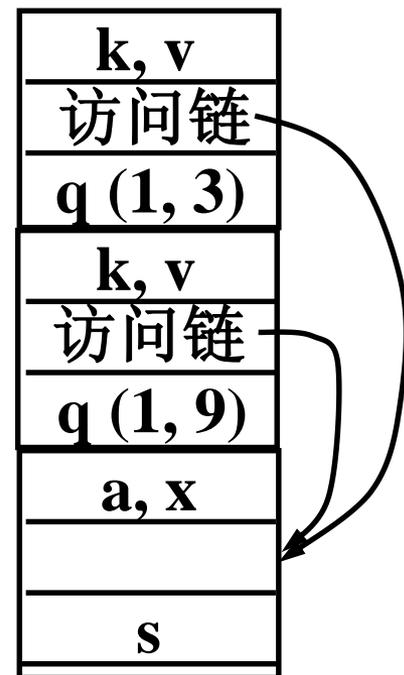


# 通过访问链访问非局部引用

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  $a$ ， $n_a \leq n_p$ ，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

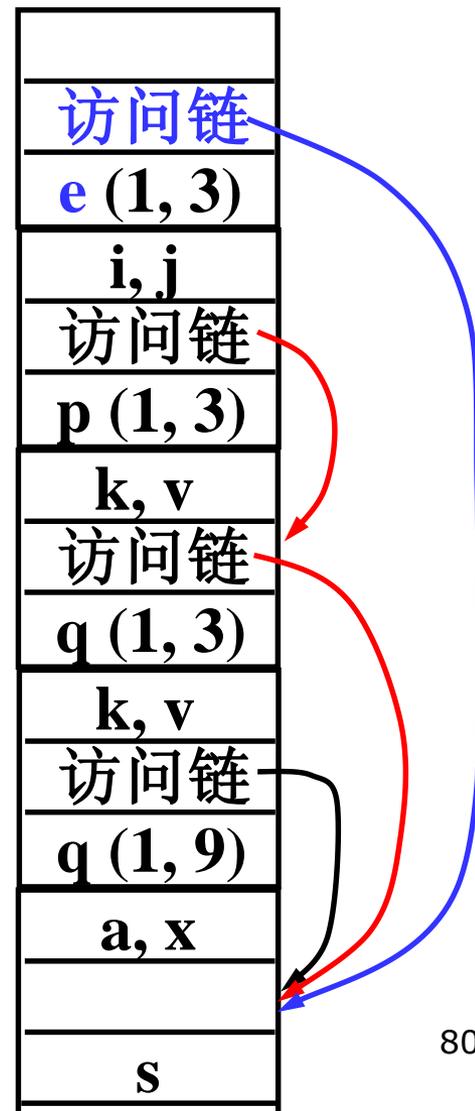




# 访问链的建立

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





# 有过程嵌套的静态作用域

- 非局部名字访问：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};**

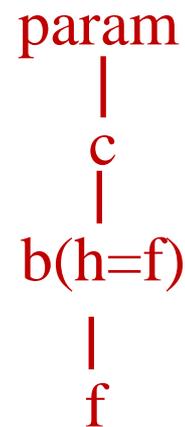
**begin**

**c**

**end.**

作为参数传递时，怎样在  
f 被激活时建立它的访问链

param	1
b	2
c[m]	2
f	2



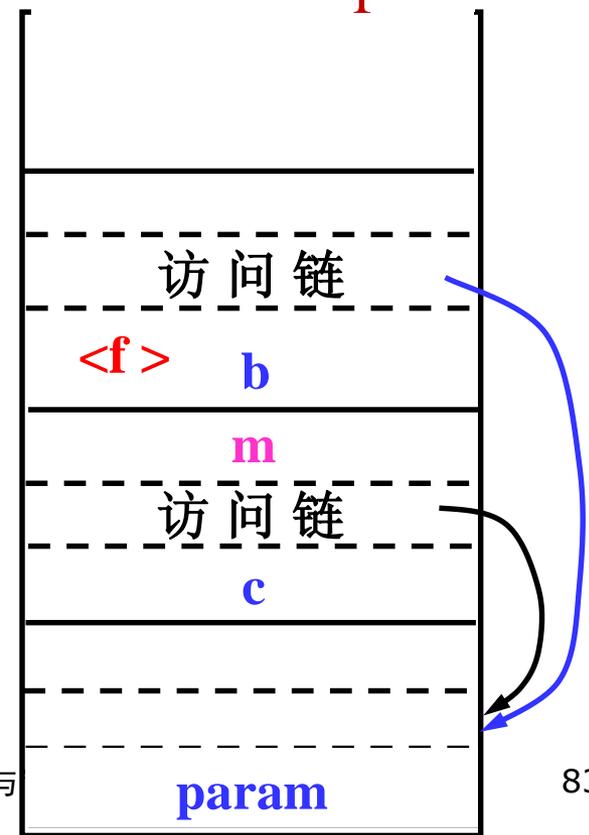
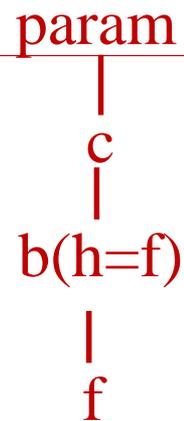


# 过程作为参数

```

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;  b: (integer→integer)→void
    var m: integer;
    function f(n: integer): integer;
      begin f := m+n end {f};
    begin m := 0; b(f) end {c};
begin
  c
end.

```



从b的访问链难以建立f的访问链

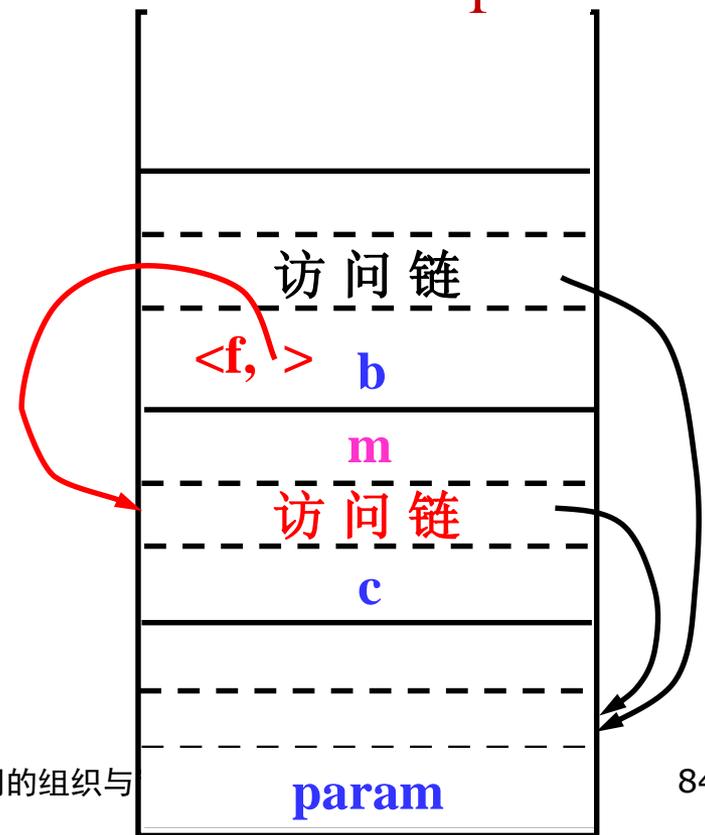
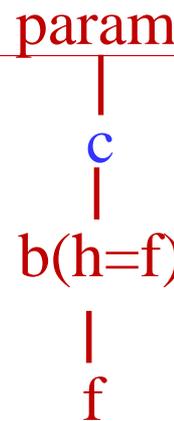


# 过程作为参数

```

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.

```



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

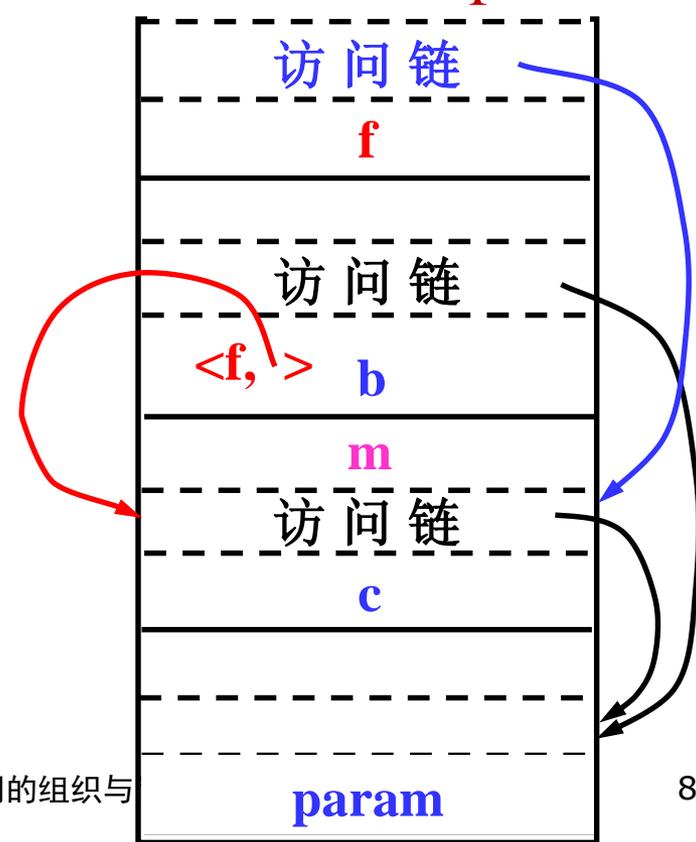
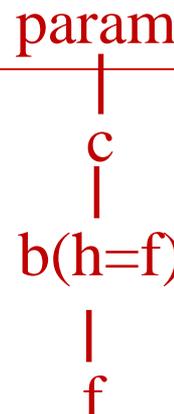


# 过程作为参数

```

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.

```



**b调用f时，用传递过来的访问链来建立f的访问链**



# 有过程嵌套的静态作用域

- 非局部名字访问：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);**

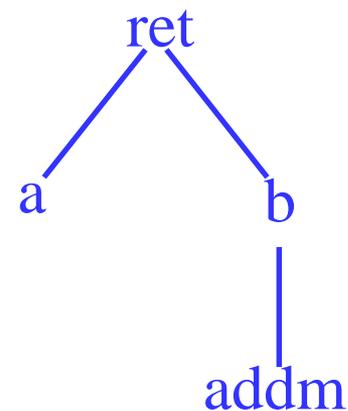
**begin writeln (g(2)) end;**

**begin**

**这里是对a的调用**

**f := a; b(f)**

**end.**





# 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);

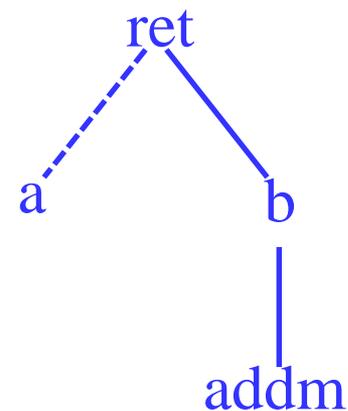
begin writeln (**g(2)**) end;

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，包含函数和声明函数时的环境  
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;
```

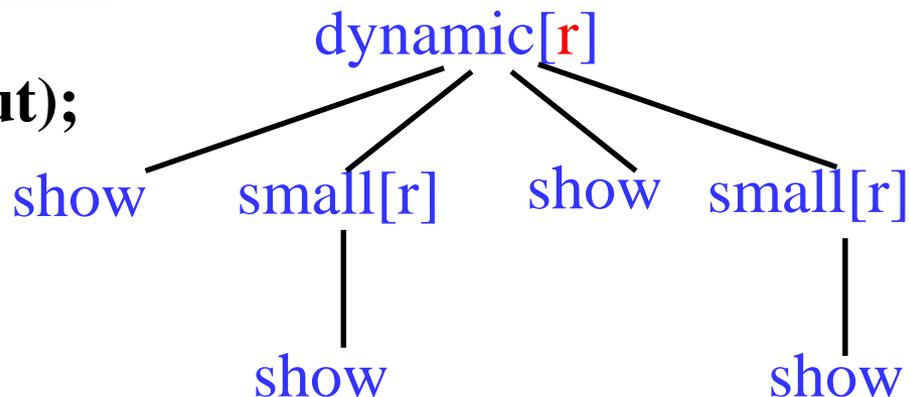
```
  begin
```

```
    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;
```

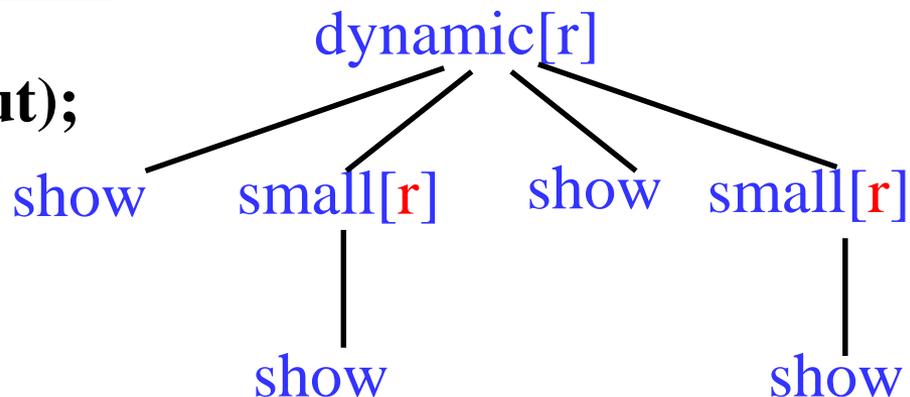
```
begin
```

```
  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\\_\(computer\\_science\)](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;
```

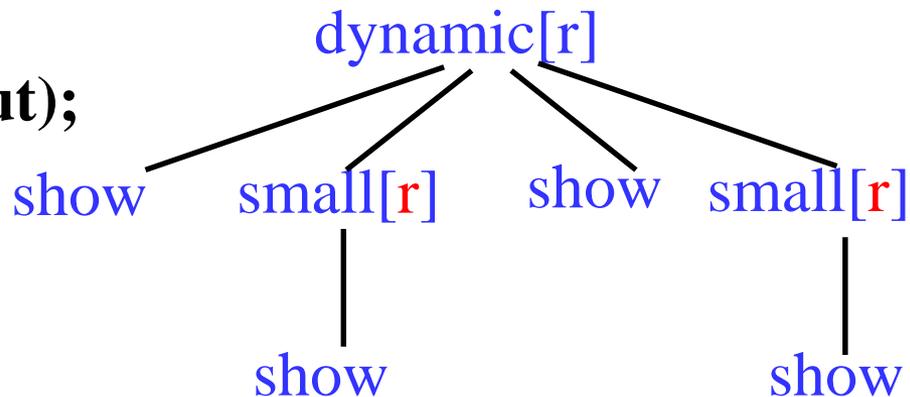
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

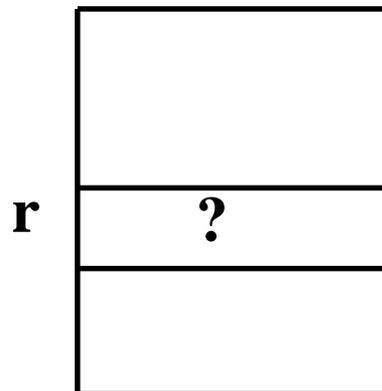
```
  show; small; writeln
```

```
end.
```



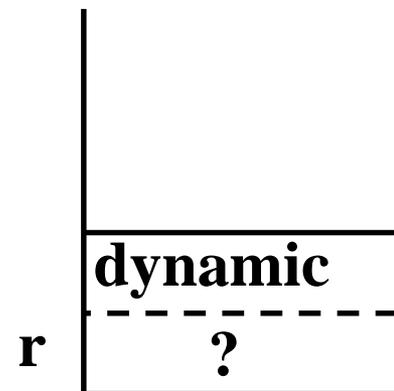
静态区

使用值的地方



栈区

暂存值的地方





# 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;
```

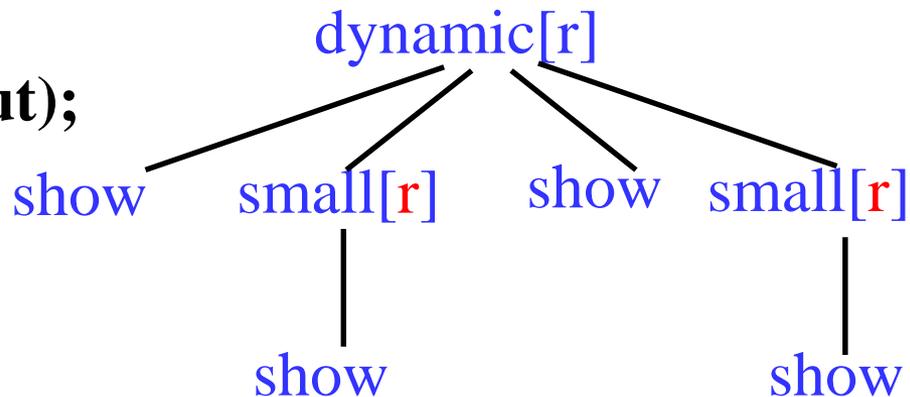
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

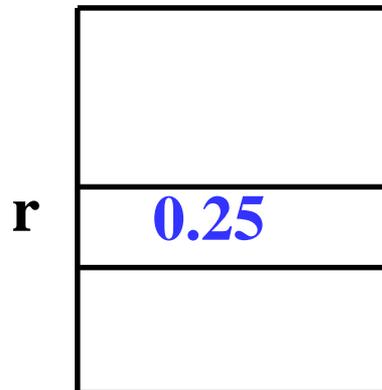
```
  show; small; writeln
```

```
end.
```



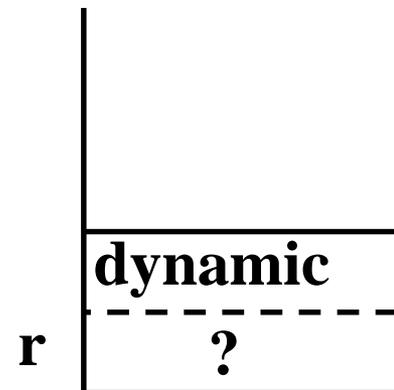
静态区

使用值的地方



栈区

暂存值的地方





# 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;
```

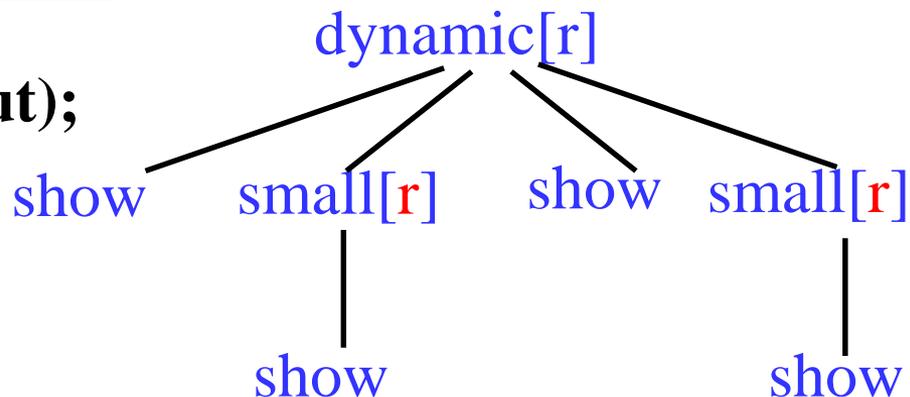
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

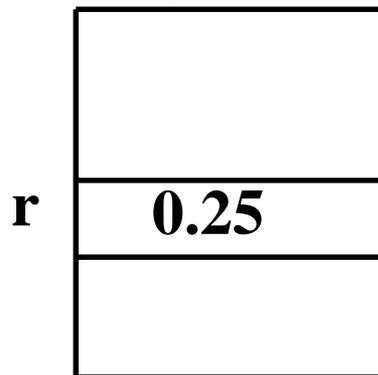
```
  show; small; writeln;
```

```
  show; small; writeln
```

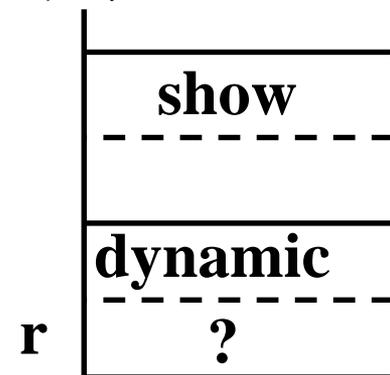
```
end.
```



静态区  
使用值的地方



栈区  
暂存值的地方





# 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;
```

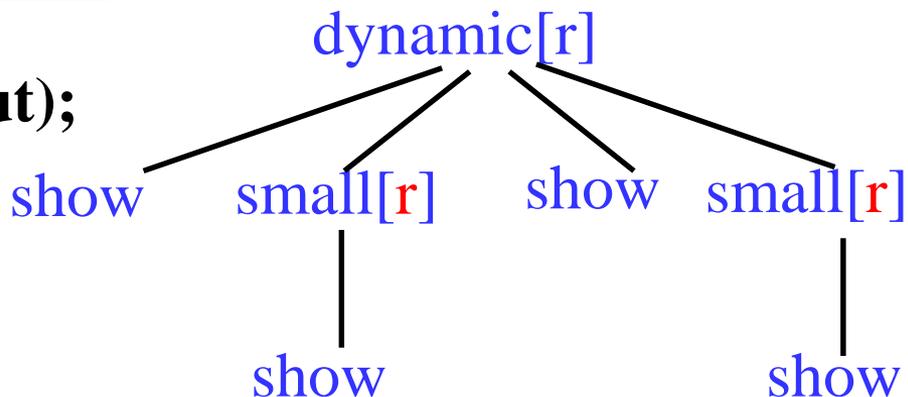
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

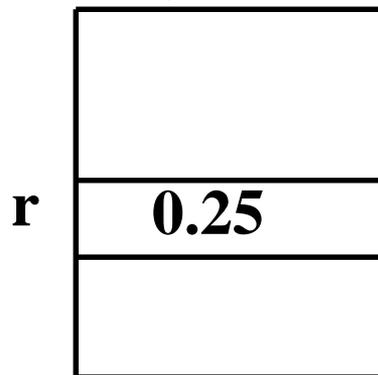
```
  show; small; writeln
```

```
end.
```



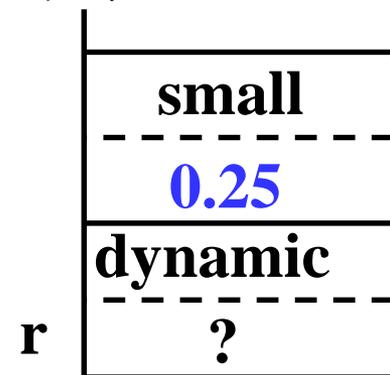
静态区

使用值的地方



栈区

暂存值的地方





# 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;
```

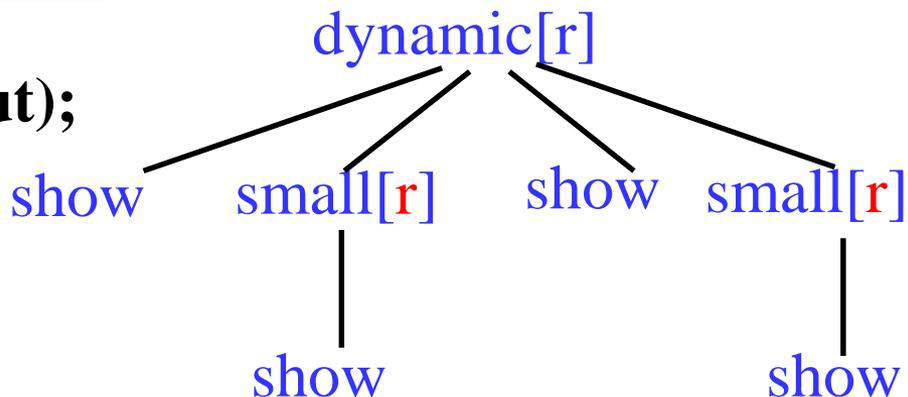
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

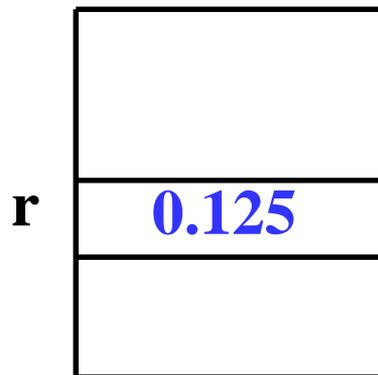
```
  show; small; writeln
```

```
end.
```



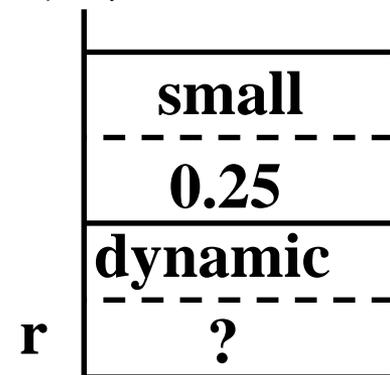
静态区

使用值的地方



栈区

暂存值的地方





# 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;
```

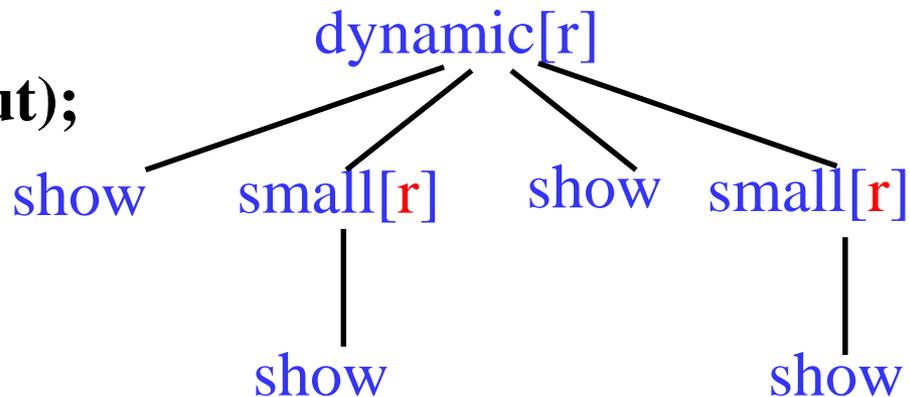
```
begin(绿色表示已执行部分)
```

```
  r := 0.25;
```

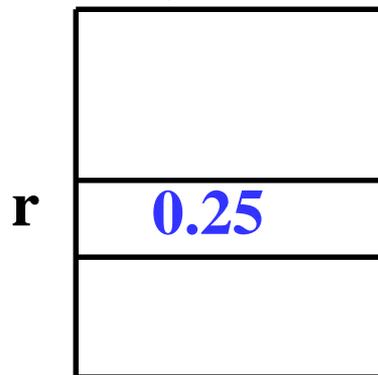
```
  show; small; writeln;
```

```
  show; small; writeln
```

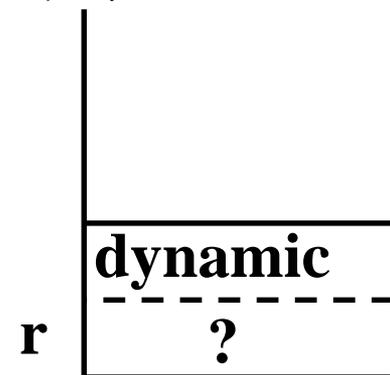
```
end.
```



静态区  
使用值的地方



栈区  
暂存值的地方





## 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



## 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

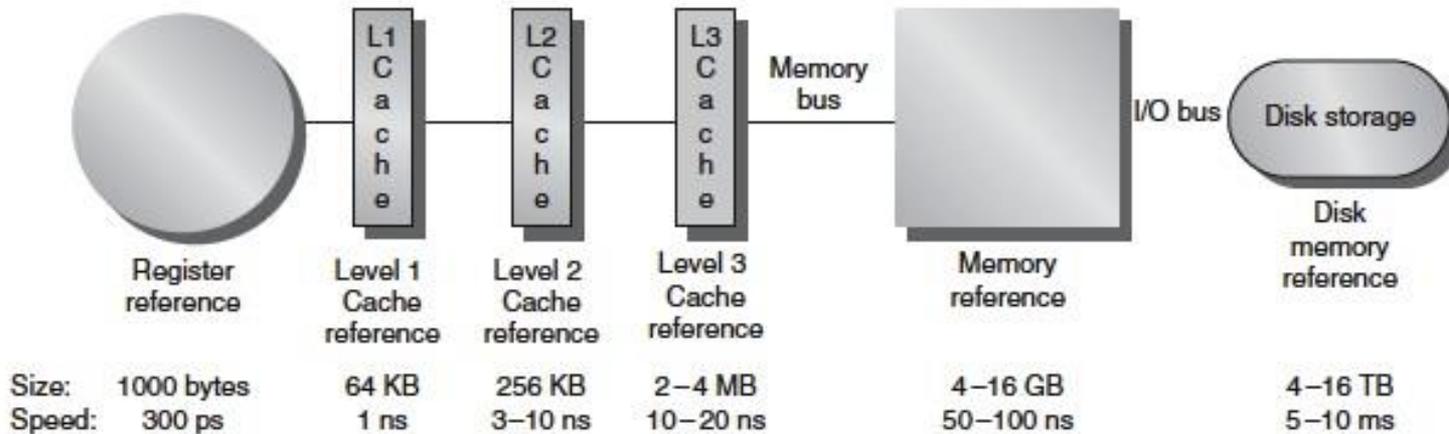
## □ 内存管理器应具有下列性质

- 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個程序执行时间中的比例尽量小

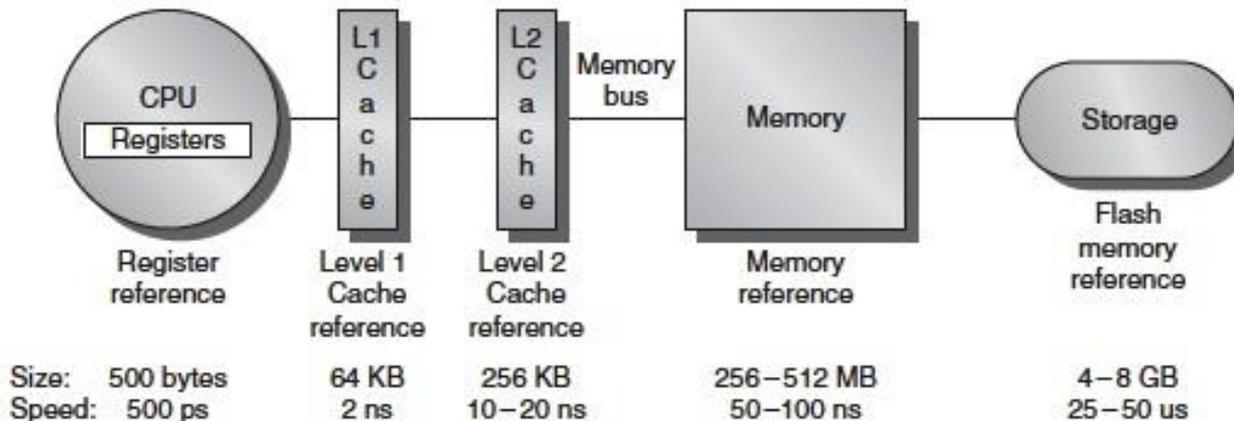


# 计算机内存分层

Computer Architecture, Fifth Edition: A Quantitative Approach, John Hennessy and David Patterson, 2012



(a) Memory hierarchy for server

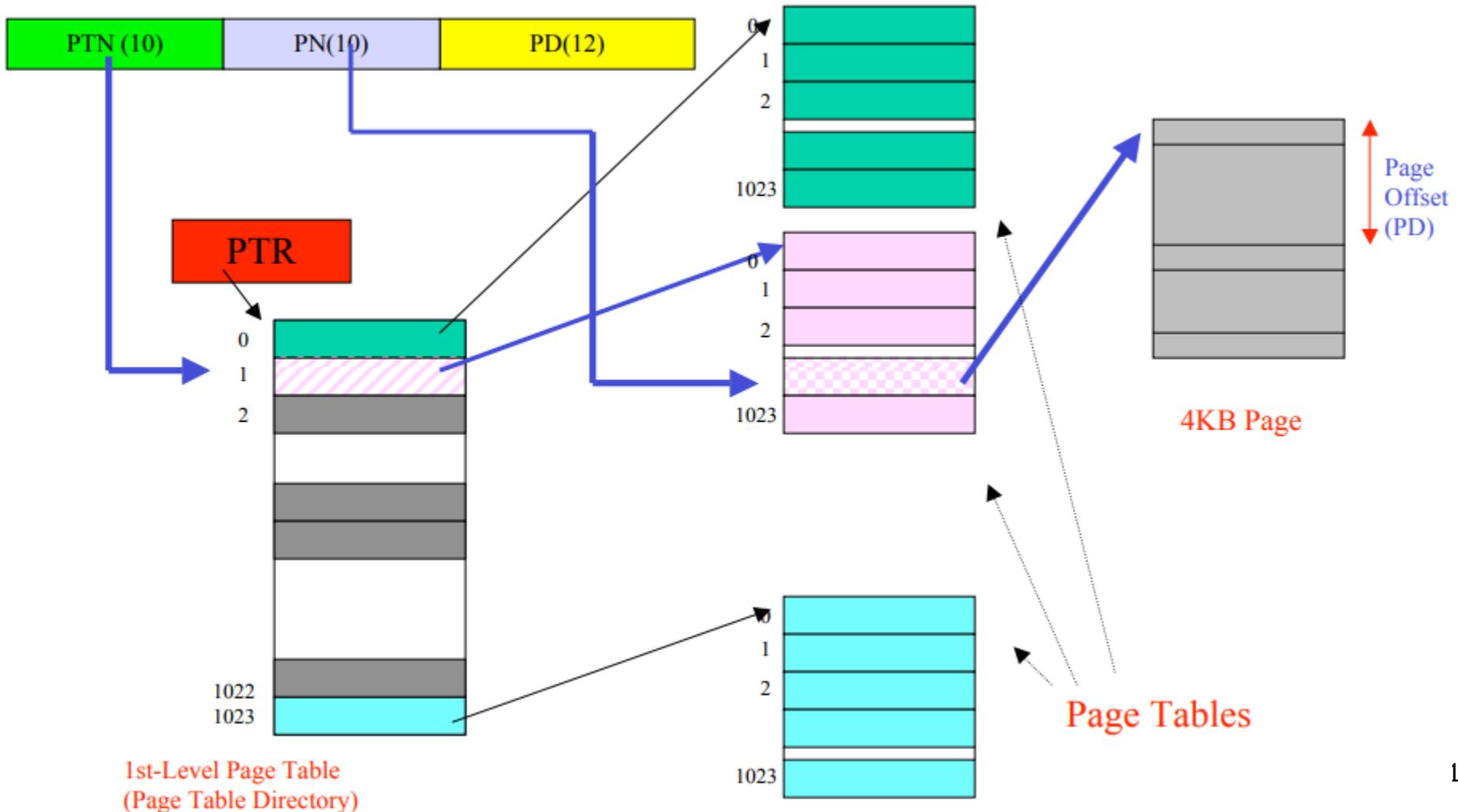


(b) Memory hierarchy for a personal mobile device



# 虚拟地址=>物理地址

## □ 两级快表 (TLB, Translation lookaside 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 algorithm

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

- 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