✧ 运行时存储组织

运行时存储组织

- ✧ 运行时存储组织的作用与任务
- ✧ 程序运行时存储空间的布局
- ✧ 存储分配策略
- ✧ 活动记录
- ✧ 过程调用与参数传递
- ✧ 垃圾回收（选讲）
- ✧ 面向对象程序运行时组织（选讲）
- ✧ 函数式程序运行时组织（选讲）

运行时存储组织

✧ 运行时存储组织的作用与任务

– 代码生成前如何安排目标机存储资源的使用

– 几个重要问题

- 数据表示 目标机中如何表示源语言中各类数据对象
- 表达式计算 如何组织表达式的计算
- 存储分配策略 如何为不同作用域或不同生命周期的数据对象分配存储
- 过程实现 如何实现过程/函数调用以及参数传递

运行时存储组织

✧ 数据表示

– 源程序中数据对象在内存或寄存器中的表示形式

- 源程序中数据对象的属性

  名字（*name*），类型（*type*），值（*value*），
  复合数据对象（*component*），……

- 数据对象在内存或寄存器中的表示形式

  位、字节、字、字节序列、……

- 有些机器要求数据存放时要按某种方式对齐（*align*）

  如：要求数据存放的起始地址为能够被4整除

运行时存储组织

✧ 数据表示举例

– 基本类型数据

char 数据 1 *byte* *integer* 数据 4 *bytes*
float 数据 8 *bytes* *boolean* 数据 1 *byte*

指针 4 *bytes*

数组 一块连续的存储区（按行/列存放）

结构/记录 所有域（*field*）存放在一块连续的存储区

对象 实例变量像结构的域一样存放在一块连续的存储区，操作例程（方法、成员函数）存放在其所属类的代码区

运行时存储组织

✧ 表达式的计算

– 在何处进行计算

- 在栈区计算

  运算数/中间结果存放于当前活动记录或通用寄存器中

- 在运算数栈计算

  某些目标机采用专门的运算数栈用于表达式计算

  对于普通表达式（无函数调用），一般可以估算出能否在运算数栈上进行

  使用了递归函数的表达式的计算通常在栈区

运行时存储组织

✧ 程序运行时存储空间的布局（*layout*）

– 典型的程序布局

- 保留地址区

  目标机体系结构和操作系统专用

- 代码区 静态存放目标代码

- 静态数据区

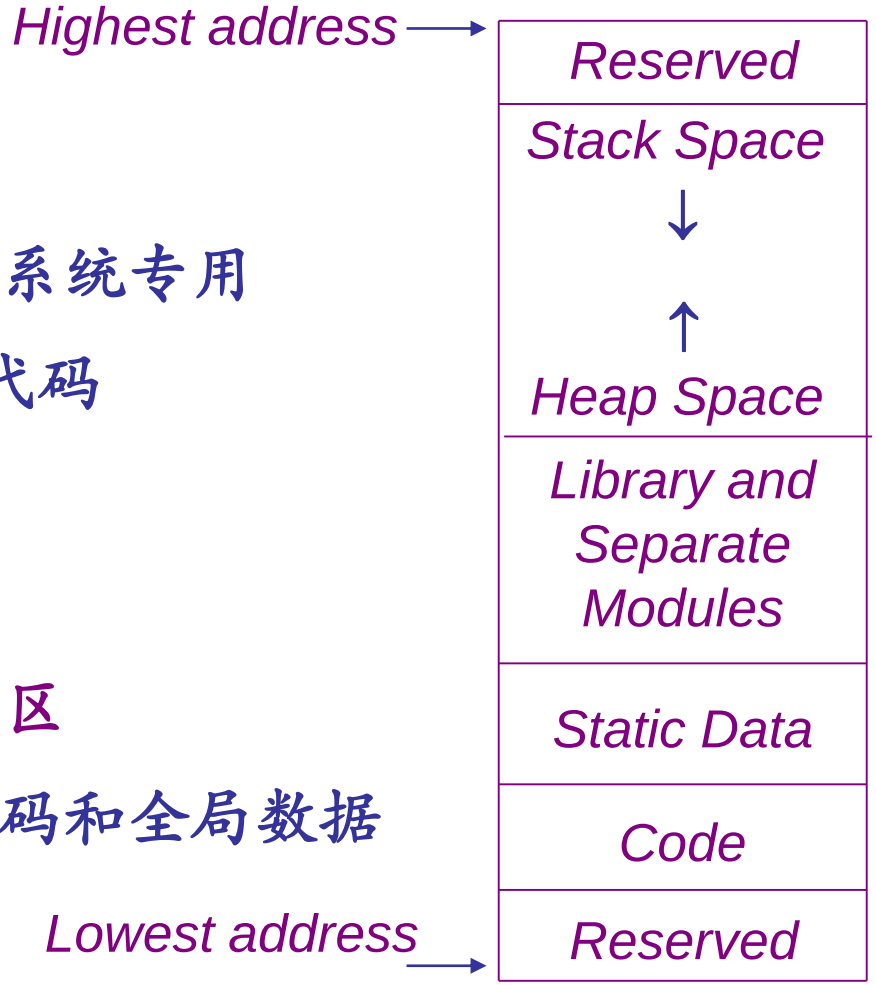
  静态存放全局数据

- 共享库和分别编译模块区

  静态存放这些模块的代码和全局数据

- 动态数据区

  运行时动态变化的堆区和栈区

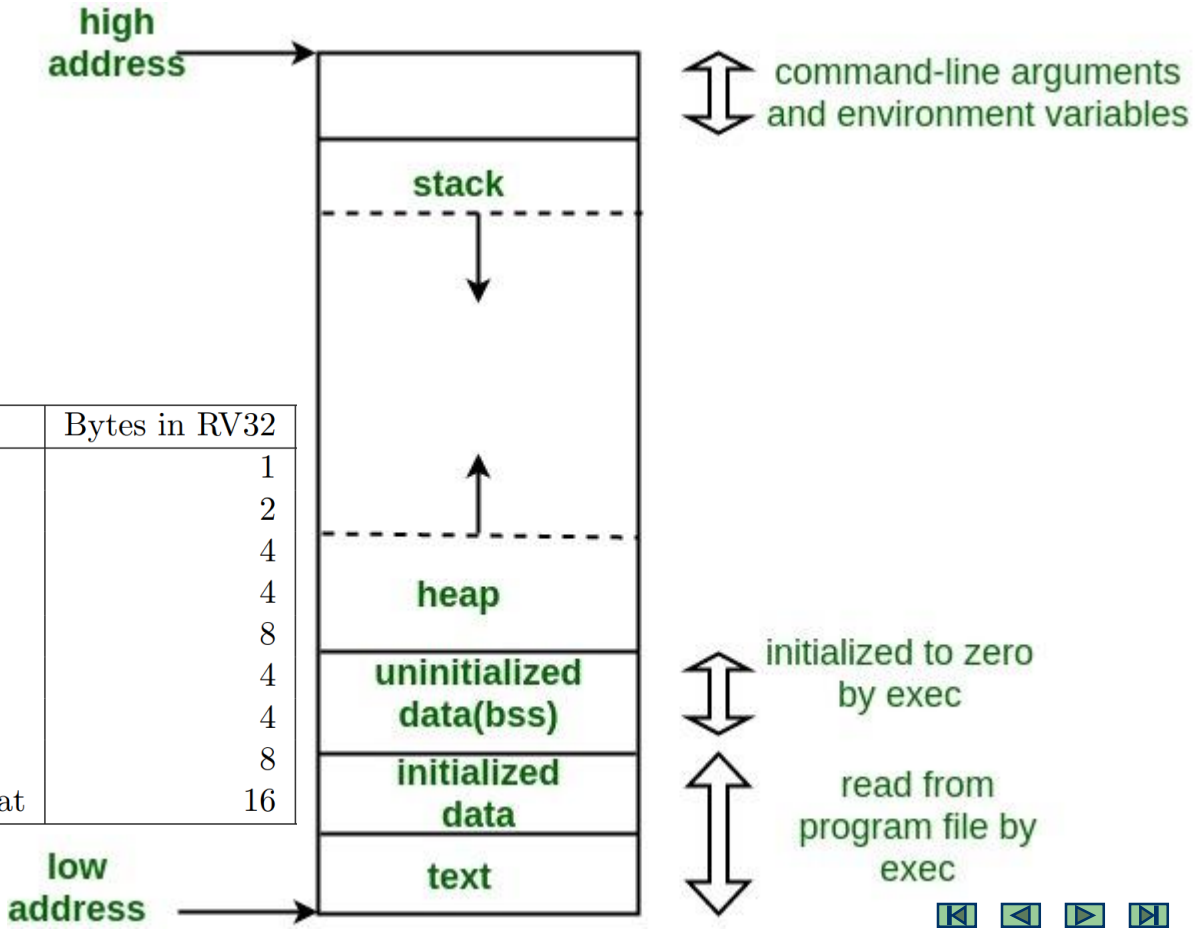


运行时存储组织

✧ 程序运行时存储空间的布局 (*layout*)

– 用户程序运行时虚地址空间布局举例

Linux on x86/MIPS/RISC-V



C type	Description	Bytes in RV32
`char`	Character value/byte	1
`short`	Short integer	2
`int`	Integer	4
`long`	Long integer	4
`long long`	Long long integer	8
`void*`	Pointer	4
`float`	Single-precision float	4
`double`	Double-precision float	8
`long double`	Extended-precision float	16

运行时存储组织

✧ 存储分配策略

– 静态分配

- 在编译期间为数据对象分配存储

– 动态分配

- 栈式分配

  将数据对象的运行时存储按照栈的方式来管理

- 堆式分配

  从数据段的堆空间分配和释放数据对象的运行时存储

运行时存储组织

✧ 静态存储分配

- 在编译期间就可确定数据对象的大小
  - 不宜处理递归过程或函数
- 某些语言中所有存储都是静态分配
  - 如早期的FORTRAN语言，COBOL语言
- 多数语言只有部分存储进行静态分配
  - 可静态分配的数据对象如大小固定且在程序执行期间可全程访问的全局变量，以及程序中的常量（*literals*）
  - 如 C++ 中的 static 变量

运行时存储组织

✧ 栈式存储分配

- 用于有效实现可动态嵌套的程序结构
  - 如实现过程/函数，块层次结构
- 可以实现递归过程/函数
  - 比较：静态分配不宜实现递归过程/函数
- 运行栈中的数据单元是活动记录（*activation record*）（专门介绍）

运行时存储组织

✧ 堆式存储分配

- 从堆空间为数据对象分配/释放存储
  - 灵活 数据对象的存储分配和释放不限时间和次序
- 显式的分配或释放（*explicit allocation / dealocation*）
  - 程序员负责应用程序的（堆）存储空间管理（借助于编译器与（或）运行时系统所提供的默认存储管理机制）
- 隐式的分配或释放（*implicit allocation / dealocation*）
  - （堆）存储空间的分配或释放不需要程序员负责，由编译器与（或）运行时系统自动完成

运行时存储组织

✧ 堆式存储分配

– 某些语言有显式的堆空间分配和释放命令

- 如：**Pascal** 中的 ***new*** , ***deposit***
  C++ 中的 ***new*** , ***delete***
- 比较：**C** 语言没有堆空间管理机制，***malloc()*** 和 ***free()*** 是标准库中的函数，可以由 *library vendor* 提供

– 某些语言支持隐式的堆空间释放

- 采用垃圾回收（*garbage collection*）机制
- 如：**Java** 程序员不需要考虑对象的析构

运行时存储组织

✧ 堆式存储分配

– 不释放堆空间的方法

- 只分配空间，不释放空间，空间耗尽时停止
- 适合于堆数据对象多数为一旦分配，永久使用的情形
- 在虚存很大及无用数据对象不致带来很大零乱的情形也可采用

运行时存储组织

✧ 堆式存储分配

– 显式释放堆空间的方法

- 用户负责清空无用的数据空间（通过执行释放命令）
- 堆管理程序只维护可供分配命令使用的空闲空间
- 问题：可能导致灾难性的 ***dangling pointer*** 错误

例：**Pascal** 代码片断

```
var p,q: ^real;
…
new(p);
q:=p;
dispose(p);
q^:=1.0;
```

C++ 代码片断

```
float  * p,*q;
…
p=new  float;
q=p;
delete p;
*q:=1.0;
```

运行时存储组织

✧ 堆式存储分配

– 隐式释放堆空间的方法

- 主要技术： 垃圾回收（*garbage collection*）机制

（可以分专门的话题讨论，选讲）

运行时存储组织

✧ 堆式存储分配

– 堆空间的管理

- 分配算法　面对多个可用的存储块，选择哪一个

  如：最佳适应算法（选择浪费最少的存储块）

  最先适应算法（选择最先找到的足够大的存储块）

  循环最先适应算法（起始点不同的最先适应算法）

- 碎片整理算法　压缩合并小的存储块，使其更可用

（可以分专门的话题讨论，超出本课程范围）
（部分内容可参考数据结构和操作系统课程）

运行时存储组织

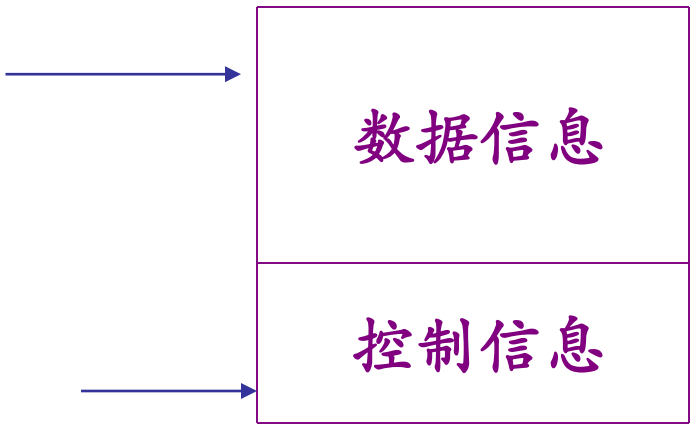
✧ 活动记录（*activation record*）

– 过程活动记录

- 函数/过程调用或返回时，在运行栈上创建或从运行栈上消去的栈帧（*frame*）

  包含局部变量，函数实参，临时值（用于表达式计算的中间单元）等数据信息以及必要的控制信息

某个数据对象的地址=
活动记录起始地址
+ 偏移地址（*offset*）



活动记录起始地址

运行时存储组织

✧ 活动记录（*activation record*）

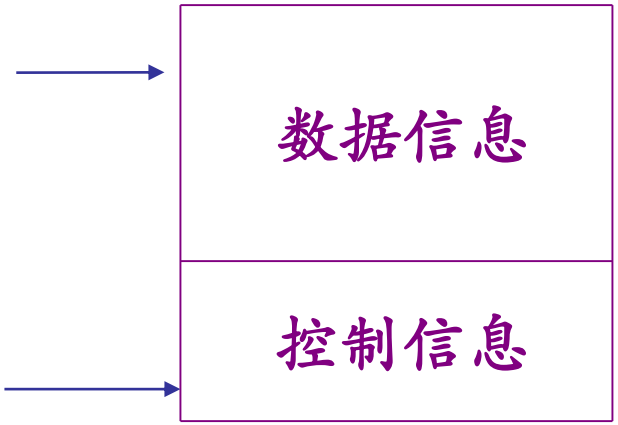
- RV32函数的入口

```
entry_label:
addi sp, sp, -framesize
sw ra, framesize-4(sp)
…
```

- RV32函数的结尾部分

```
lw ra, framesize-4(sp)
addi sp, sp, framesize
…
ret
```

某个数据对象的地址=
活动记录起始地址
+ 偏移地址（*offset*）



参数
局部变量
临时变量

s0:Frame Pointer
ra: Return Addr

活动记录起始地址

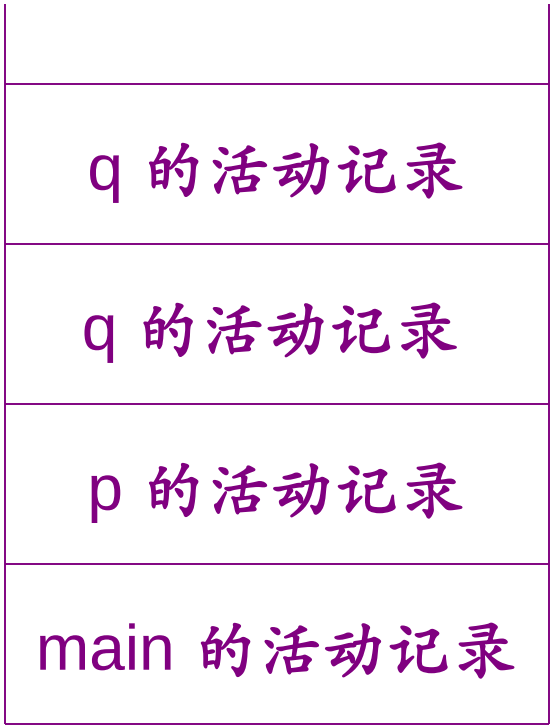
运行时存储组织

✧ 活动记录

– 过程活动记录的栈式分配举例

```
void p( ) {
   …
   q( );
}

void q( ) {
   …
   q( );
}

int main {
   p( );
}
```

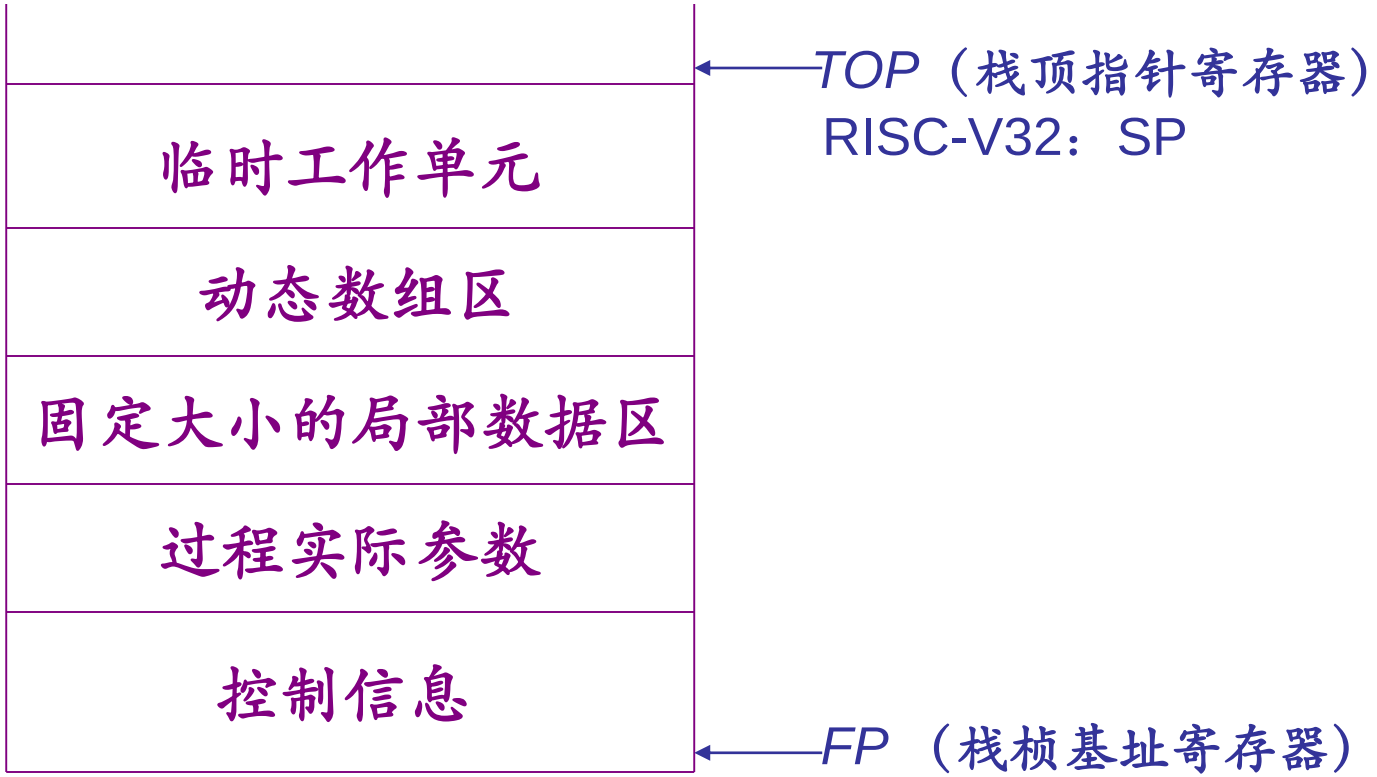
函数 q 被第二次激活时运行栈上活动记录分配情况



运行时存储组织

✧ 活动记录

– 典型的过程活动记录结构



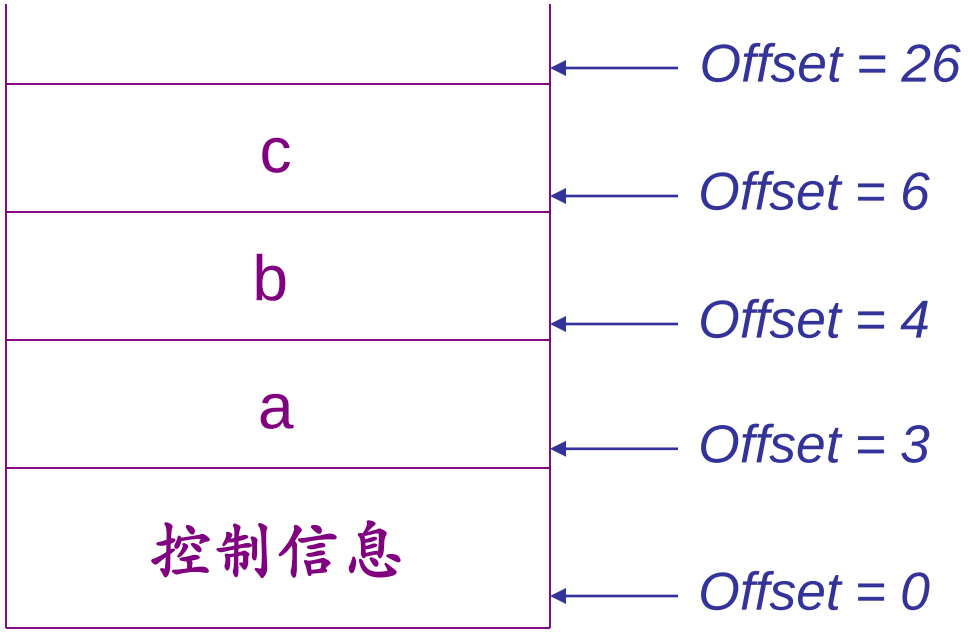
运行时存储组织

✧ 活动记录

– 过程活动记录举例

```
void p( int a) {
   float b;
   float c[10];
   b=c[a];
}
```

函数 p 的活动记录



运行时存储组织

✧ 活动记录

– 过程活动记录举例

```
static int N;

void p( int a) {
   float b;
   float c[10];
   float d[N];
   float e;
   …

}
```

/*d为动态数组*/

函数 p 的活动记录

运行时存储组织

✧ 活动记录

– 含嵌套过程说明语言的栈式分配

- 主要问题

  解决对非局部量的引用（存取）

- 解决方案

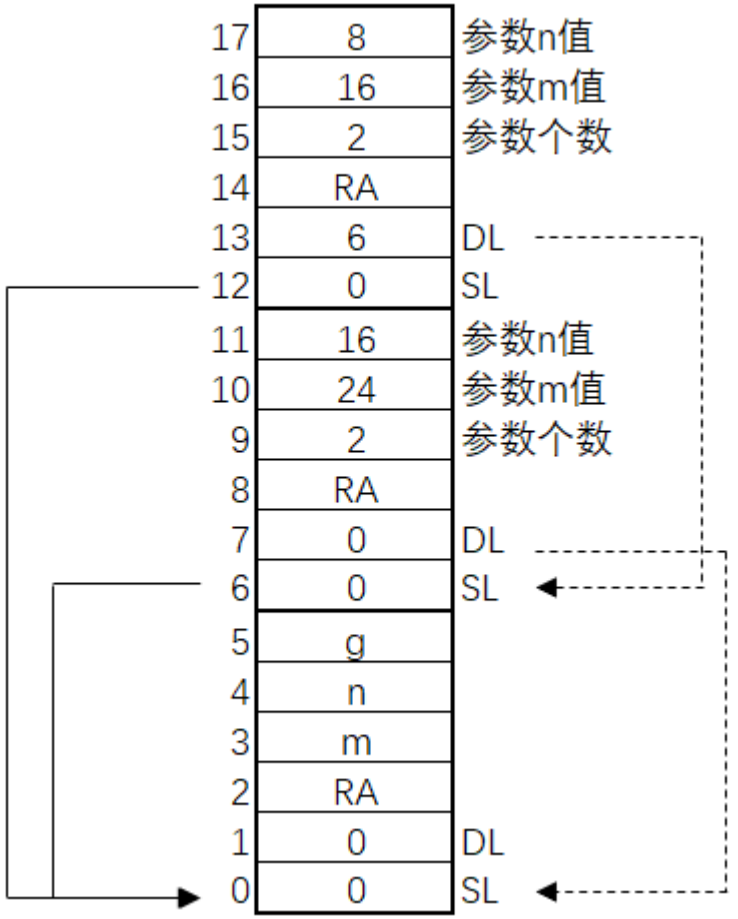
  采用 Display 表

  为活动记录增加静态链域

运行时存储组织

✧ 活动记录

```
var m, n, g:int;
function gcd(m,n:int):int
  begin
   if n = 0 then
        g := m ;
   else g := gcd(n, m mod n);
begin
 m := 24; n := 16;
 g := gcd(m, n)
end
```



- 动态链 DL：指向调用该过程前的最新活动记录地址的指针
- 静态链 SL：指向静态直接外层最新活动记录地址的指针，用来访问非局部数据

运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用 Display 表 （或称全局 Display 表）

  Display 表记录各嵌套层当前过程的活动记录在运行栈上的起始位置（基地址）

  当前激活过程的层次为K（主程序的层次设为0），则对应的 Display 表含有 K+1 个单元，依次存放着现行层，直接外层...直至最外层的每一过程的最新活动记录的基地址

  嵌套作用域规则确保每一时刻Display 表内容的唯一性

  Display 表的大小（即最多嵌套的层数）取决于实现

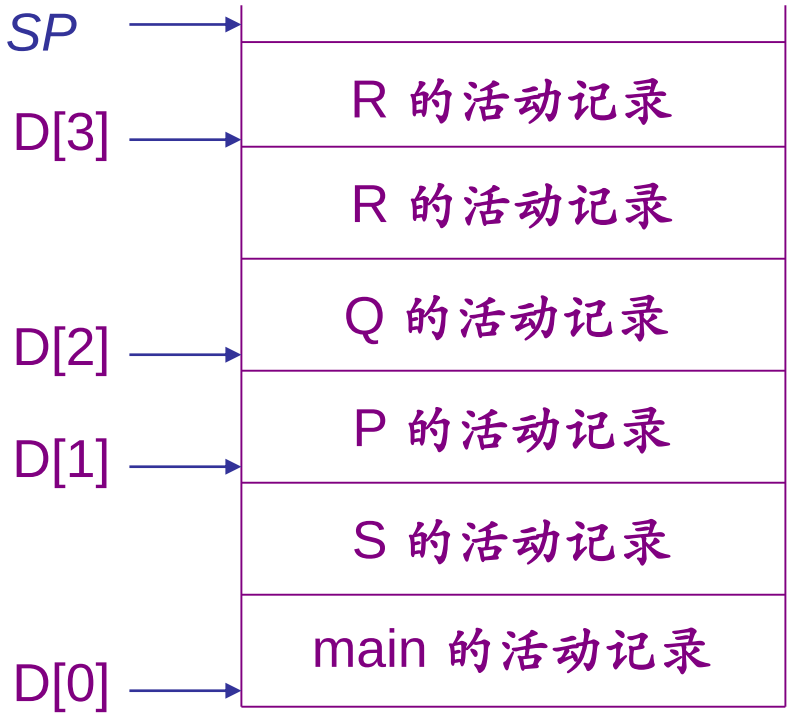
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表方案举例

过程 R 被第二次激活后运行栈和 Display 寄存器 D[i] 的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;   /*R*/
    begin
      … R; …
    end;   /*Q*/
  begin
    … Q; …
  end;   /*P*/
procedure S;
  begin
    … P; …
  end;   /*S*/
begin
  …  S; …
end.   /*main*/
```

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护（过程被调用和返回时的保存和恢复）

方法一 极端的方法是把整个 Display 表存入活动记录
若过程为第 *n* 层，则需要保存 D[0] ~D[*n*] ）

一个过程（处于第 *n* 层）被调用时，从调用过程的 Display 表中自下向上抄录 *n* 个 TOP 值，再加上本层的 TOP 值

方法二 只在活动记录保存一个的 Display 表项，在静态存储区或专用寄存器中维护全局 Display 表

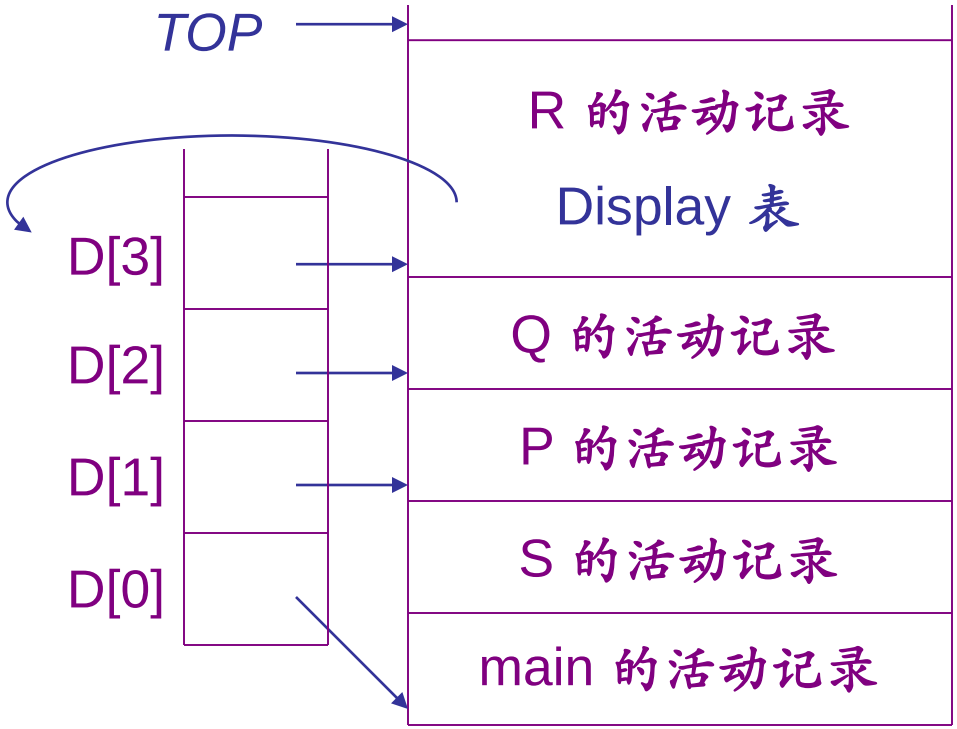
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护举例

活动记录中保存完整的Display 表




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护举例

只保存一个Display 表项

calls	S	P	Q	R	P'	Q'	R'
D[3]	_	_	_	R	R	R	R'
D[2]	_	_	Q	Q	Q	Q'	Q'
D[1]	S	P	P	P	P'	P'	P'
D[0]	Main	Main	Main	Main	Main	Main	Main
saved	_	S	_	_	P	Q	R

```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … P; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用静态链（*static link*）

  Display 表的方法要用到多个存储单元或多个寄存器，有时并不情愿这样做，一种可选的方法是采用静态链

  所有活动记录都增加一个静态链（如在offset 为 0 处）的域，指向定义该过程的**直接外过程**（或主程序）运行时**最新的**活动记录

  在过程返回时当前 AR 要被撤销，为回卷（*unwind*）到调用过程的AR（恢复FP）需要用到**动态链**域

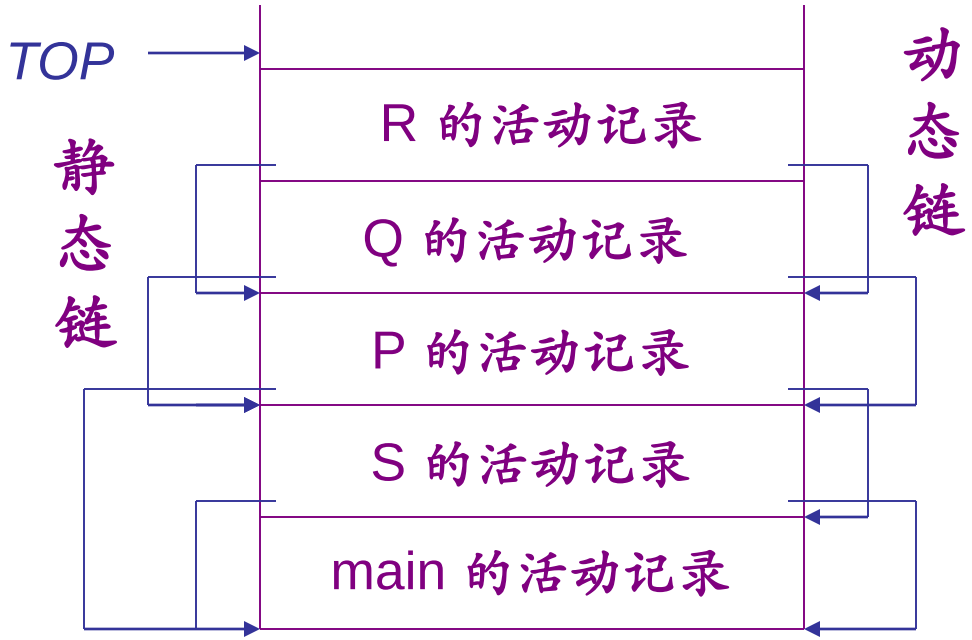
运行时存储组织

✧ 活动记录

– 嵌套过程语言的栈式分配

- 采用静态链的方法举例

过程 R 被第一次激活后运行栈的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;   /*R*/
    begin
      … R; …
    end;   /*Q*/
  begin
    … Q; …
  end;   /*P*/
procedure S;
  begin
    … P; …
  end;   /*S*/
begin
  …  S; …
end.   /*main*/
```

运行时存储组织

✧ 活动记录

– 嵌套程序块的非局部量访问

- 一些语言（如 C 语言）支持嵌套的块，在这些块的内部也允许声明局部变量，同样要解决依嵌套层次规则进行非局部量使用（访问）的问题

  方法一　将每个块看作为内嵌的无参过程，为它创建一个新的活动记录，称为**块级活动记录**

  该方法代价很高

  方法二　由于每个块中变量的相对位置在编译时就能确定下来，因此可以不创建块级活动记录，仅需要过程级的活动记录就可解决问题（见下例）

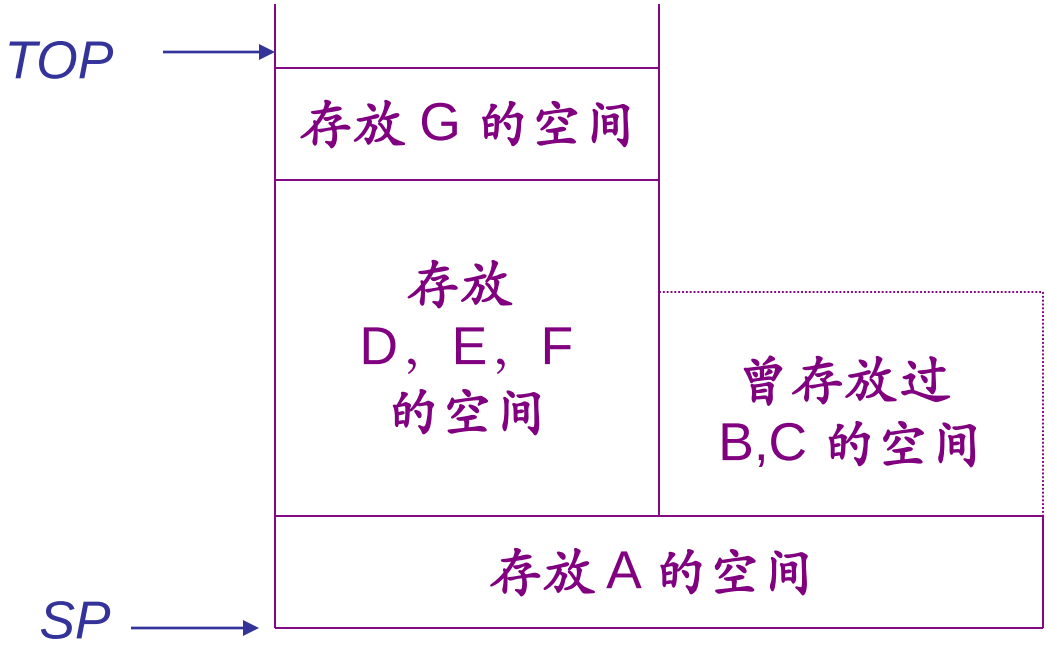
运行时存储组织

✧ 活动记录

– 嵌套程序块的非局部量访问

- 采用过程级活动记录的方法举例

运行至/*here*/时p的活动记录形如：




```
int p（）
{
    int A;
    …
    {
        int B,C;
        …
    }
    {
        int D,E,F;
        …
        {
            int G;
            …  /*here*/
        }
    }
}
```

运行时存储组织

✧ 活动记录

– 动态作用域规则 vs. 静态（词法）作用域规则

```
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.
```

lexical scope

0.250 0.250
0.250 0.250

dynamic scope

0.250 0.125
0.250 0.125

思考：

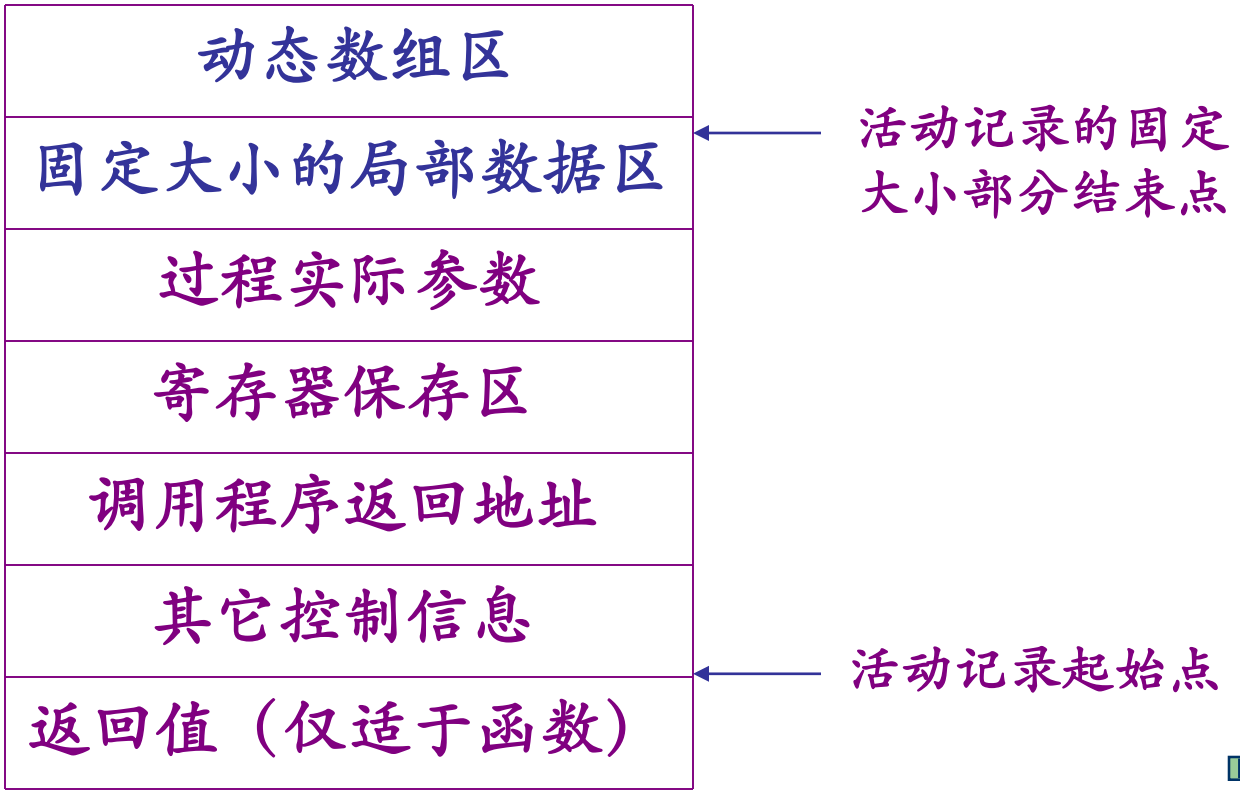
如何实现动态作用域规则？

运行时存储组织

✧ 过程调用与参数传递

– 活动记录中与过程/函数调用相关的信息

- 典型的活动记录形式举例



运行时存储组织

✧ 过程调用与参数传递

– 最常见的参数传递方式

- 传值 **call-by-value**

  传递的是实际参数的右值（*r-value*）

- 传地址 **call-by-reference（-address, -location）**

  传递的是实际参数的左值（*l-value*）

– 注　表达式的左值代表存储该表达式值的地址

表达式的右值代表该表达式的值

运行时存储组织

✧ 过程调用与参数传递

– 参数传递方式

- **call-by-value** 举例
  调用swap(a,b) 过程将不会影响a和b的值，其结果等价于执行下列语句序列：

```
x :=a;
y :=b;
temp :=x;
x :=y;
y :=temp
```

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

✧ 过程调用与参数传递

– 参数传递方式

- 实现 **call-by-value**

  形式参数当作过程的局部变量处理，即在被调过程的活动记录中开辟了形参的存储空间，这些存储位置用以存放实参

  调用过程计算实参的值，将其放于对应的存储空间

  被调用过程执行时，就像使用局部变量一样使用这些形式单元

✧ 过程调用与参数传递

– 参数传递方式

- **call-by-reference** 举例

  调用swap(a,b) 过程将交换 a 和 b 的值

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

✧ 过程调用与参数传递

– 参数传递方式

- 实现 call-by-reference

把实在参数的地址传递给相应的形参，即调用过程把一个指向实参的存储地址的指针传递给被调用过程相应的形参:

若实在参数是一个名字，或具有左值的表达式，则传递左值

若实在参数是无左值的表达式，则计算该表达式的值，放入一存储单元，传此存储单元地址

✧ 过程调用与参数传递

– 过程/函数参数

- **不含嵌套过程/函数声明**

  如 C 语言，任何过程/函数内部访问的非局部量只有全局量
  可以将所有全局量分配在静态区

  这种情况下，无论采取什么方式激活一个过程/函数，活动记录没有什么差异，局部数据在活动记录中访问，而非局部数据只有全局量，均在静态区访问

- **包含嵌套过程/函数声明**

  介绍龙书中的一种解决方案

✧ 过程调用与参数传递

– 过程/函数参数

- 一个 ML 程序的例子（包含嵌套函数声明，选自龙书）

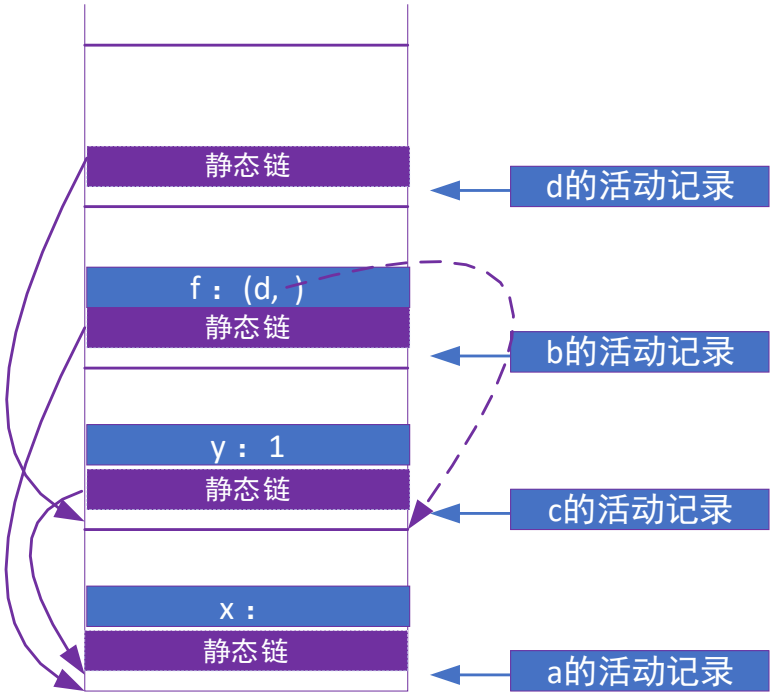
```
fun a(x) =
    let
        fun b(f) =
            … f … ;
        fun c(y) =
            let
                fun d(z) = …
            in
                … b(d) …
            end
    in
        … c(1) …
    end;
```

运行时存储组织

✧ 过程调用与参数传递

– 过程/函数参数

- 解决方案：使用带静态链的函数实参 （*call-by-closure*）




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

垃圾回收（这讲）

✧ 垃圾回收

– 垃圾回收机制

- 维护不变量：任何活跃的对象是可达
  （理解：活跃对象和可达对象）
- 更变程序（*mutator*）用于维护对象引用链的可达性
- 与运行时系统的交互：堆空间分配与堆空间释放

垃圾回收（这讲）

✧ 垃圾回收

– 垃圾回收算法（两大类）

- 观察当前可达的对象是否转变为不可达的，如引用计数（*reference counting*）法
- 重要语言范型相基于跟踪的垃圾回收（*trace-based garbage collection*）

基本方法：标记-清除（*mark-and-sweep*），拷贝回收（*copying collection*）

短停顿（*short-pause*）方法：

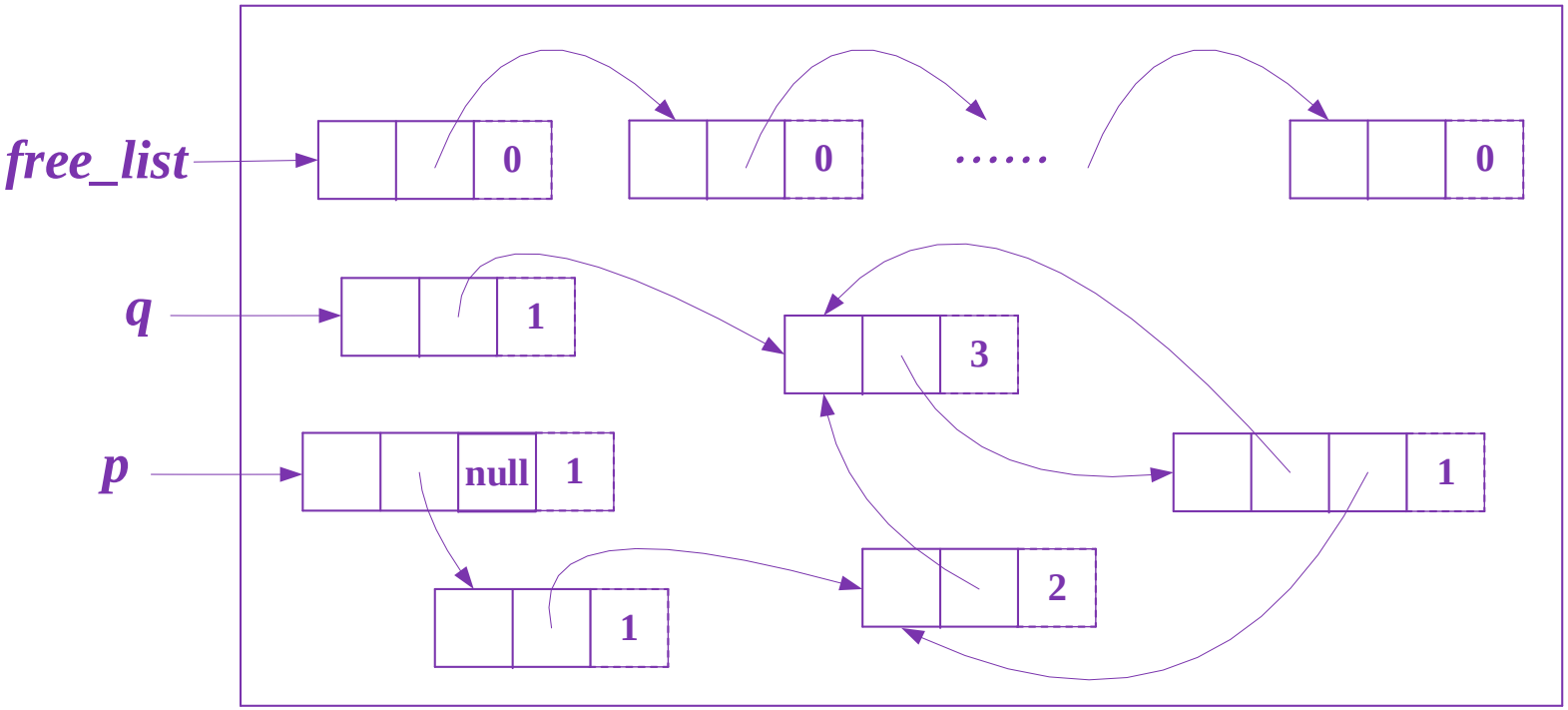
分代回收（*generational collection*）
增量回收（*incremental collection*）

垃圾回收（这讲）

✧ 垃圾回收算法

– 引用计数（*reference counting*）

- 示例 （执行 *q* = *p* 之前）

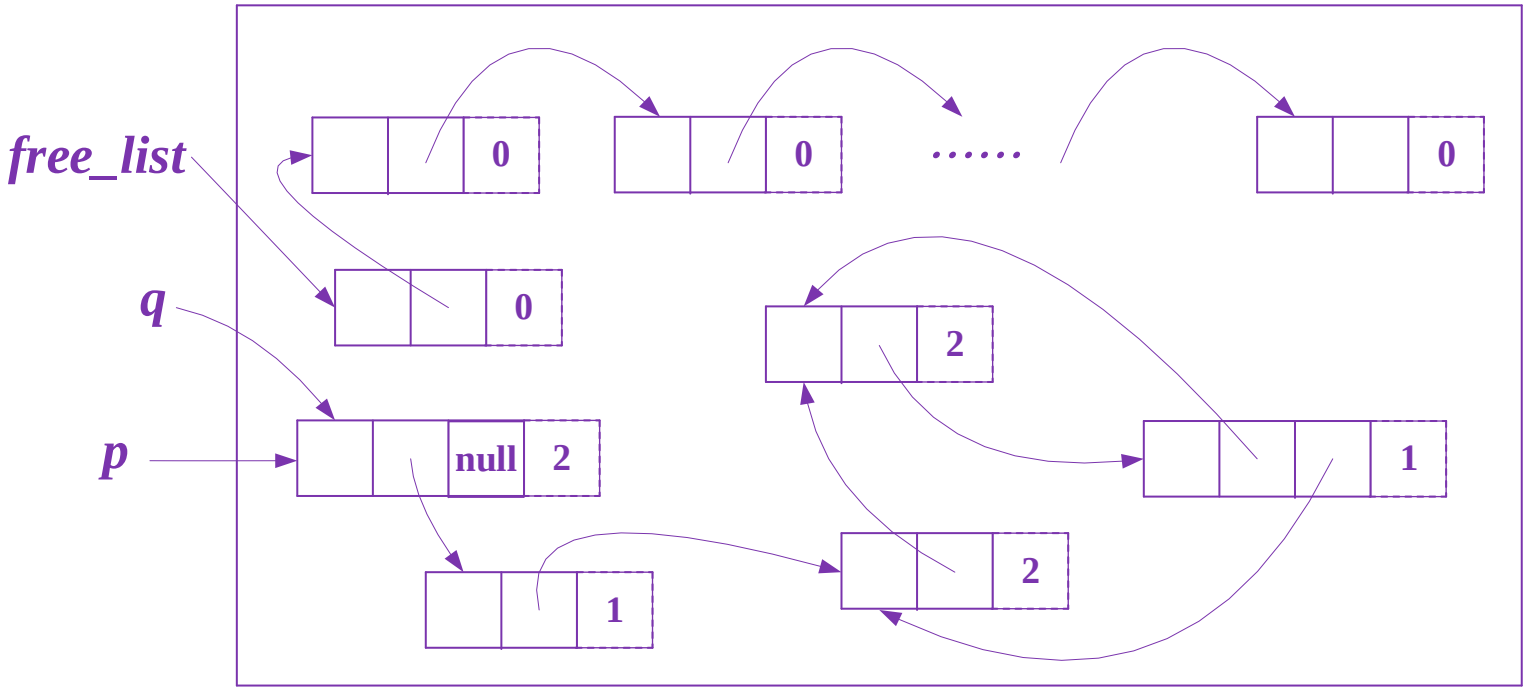


垃圾回收（这讲）

✧ 垃圾回收算法

– 引用计数（*reference counting*）

- 示例 （执行 *q* = *p*之后）

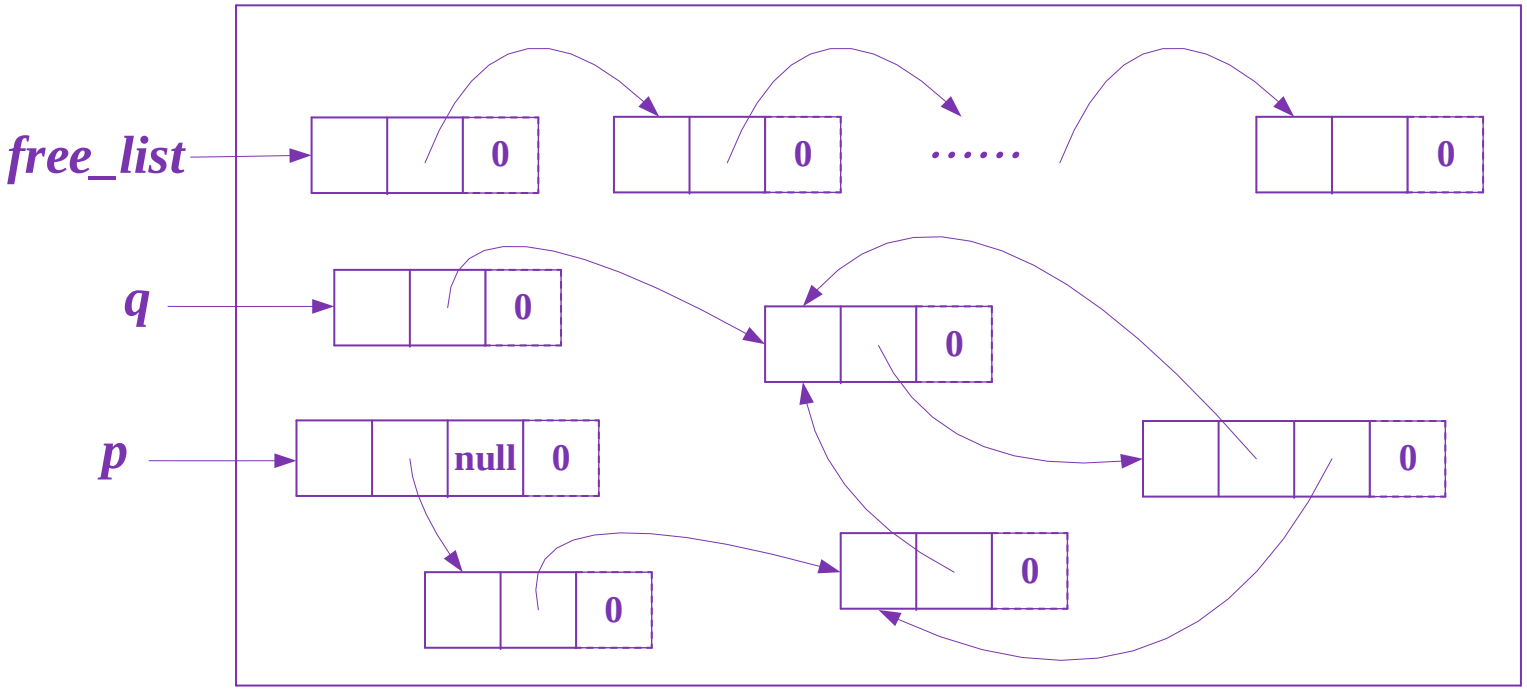


垃圾回收（这讲）

✧ 垃圾回收算法

– 标记-清除（*mark-and-sweep*）

- 示例 （触发前）

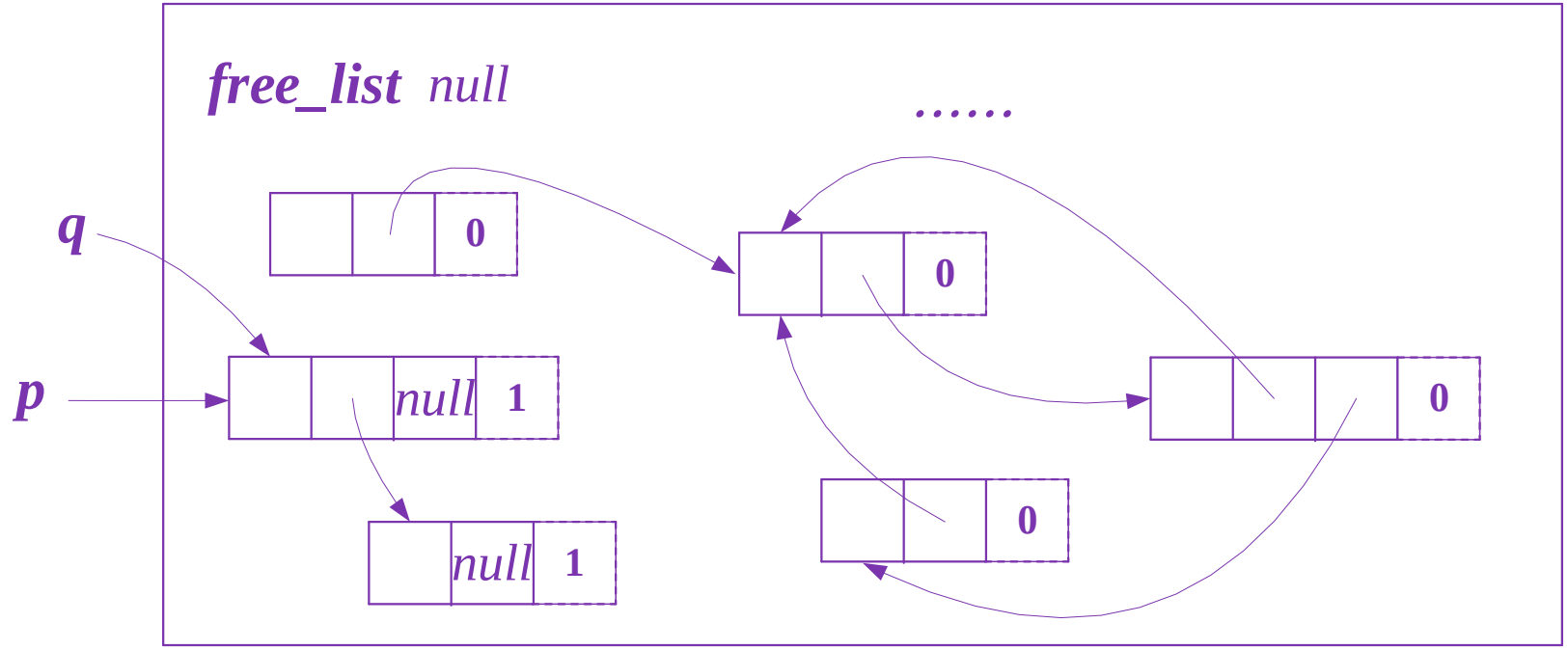


垃圾回收（这讲）

✧ 垃圾回收算法

– 标记-清除（*mark-and-sweep*）

- 示例 （触发并标记）

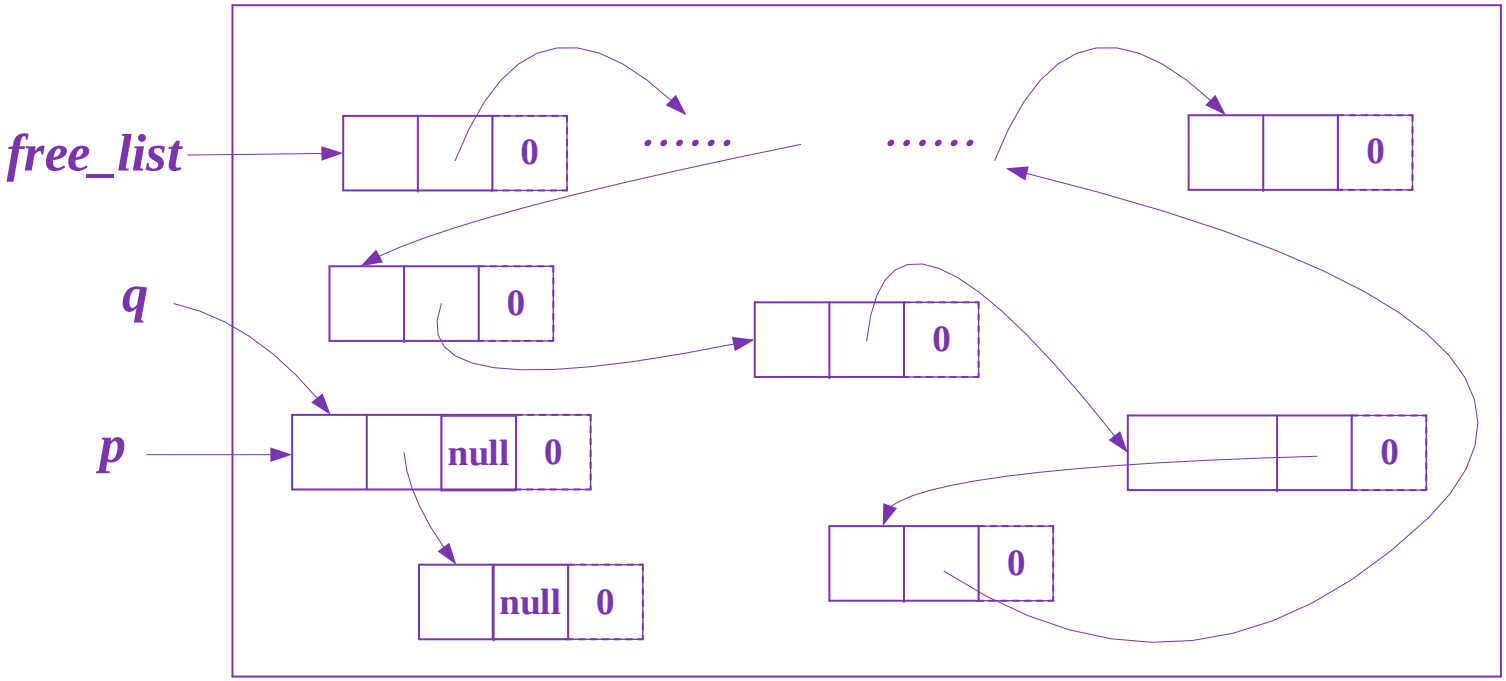


垃圾回收（这讲）

✧ 垃圾回收算法

– 标记-清除（*mark-and-sweep*）

- 示例 （清除后）



垃圾回收（这讲）

✧ 垃圾回收算法

– 拷贝回收 (*copying collection*)

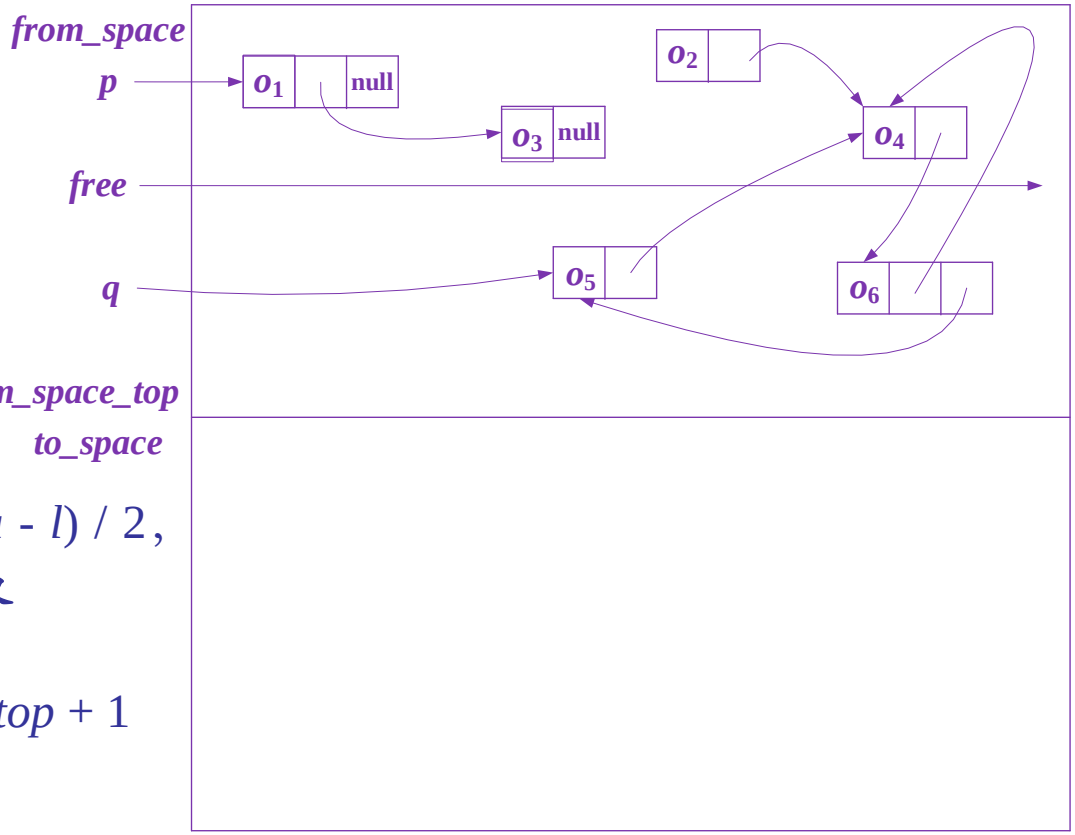
- 堆空间初始化

（1）令 *from_space* = *l*，为可用堆空间的下界

（2）置 *from_space_top* = *l* + (*u* - *l*) / 2，这里*u*为可用堆空间的上界

（3）令 *to_space* = *from_space_top* + 1

（4）置 *free* = *from_space*

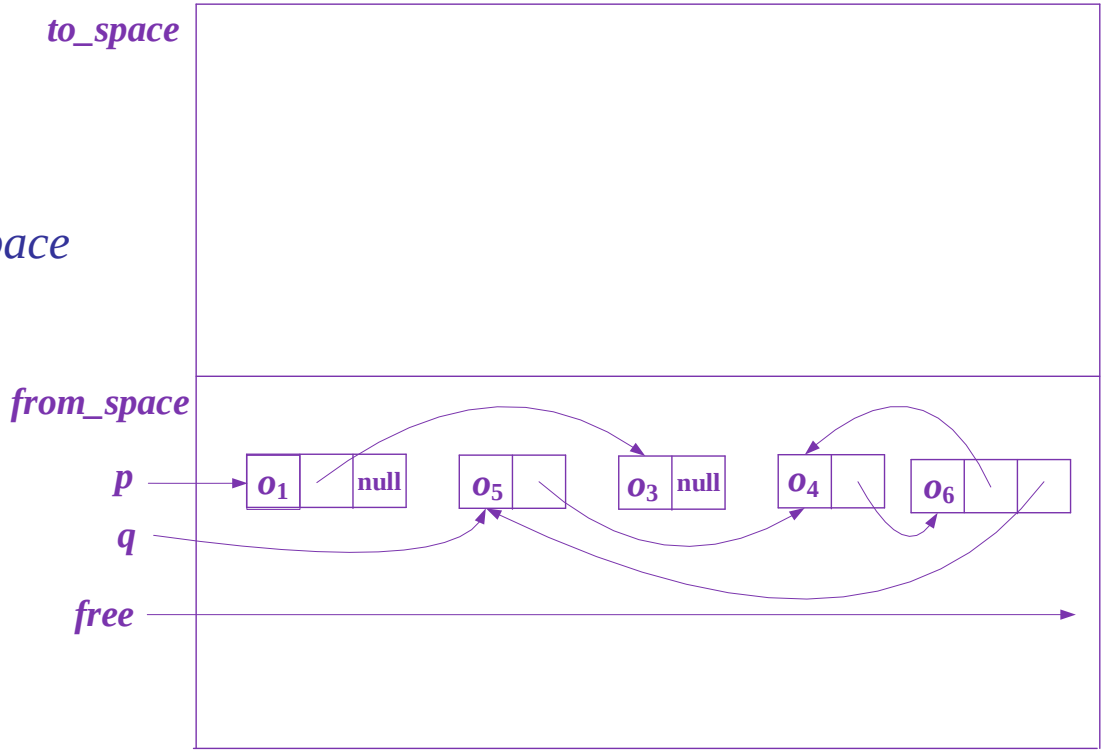


垃圾回收（这讲）

✧ 垃圾回收算法

– 拷贝回收 (*copying collection*)

- 激活拷贝回收算法（如 *Cheney*算法），将 *from_space* 空间中分配的所有可达结点拷贝至 *to_space* 空间相邻的存储块依次排放，且令指针 *free* 指向紧其后的空闲空间起始位置，然后对调 *from_space* 与 *to_space* 的角色，即令 *from_space,to_space = to_space,from_space*以及 *from_space_top=from_space+(u-l)/2*



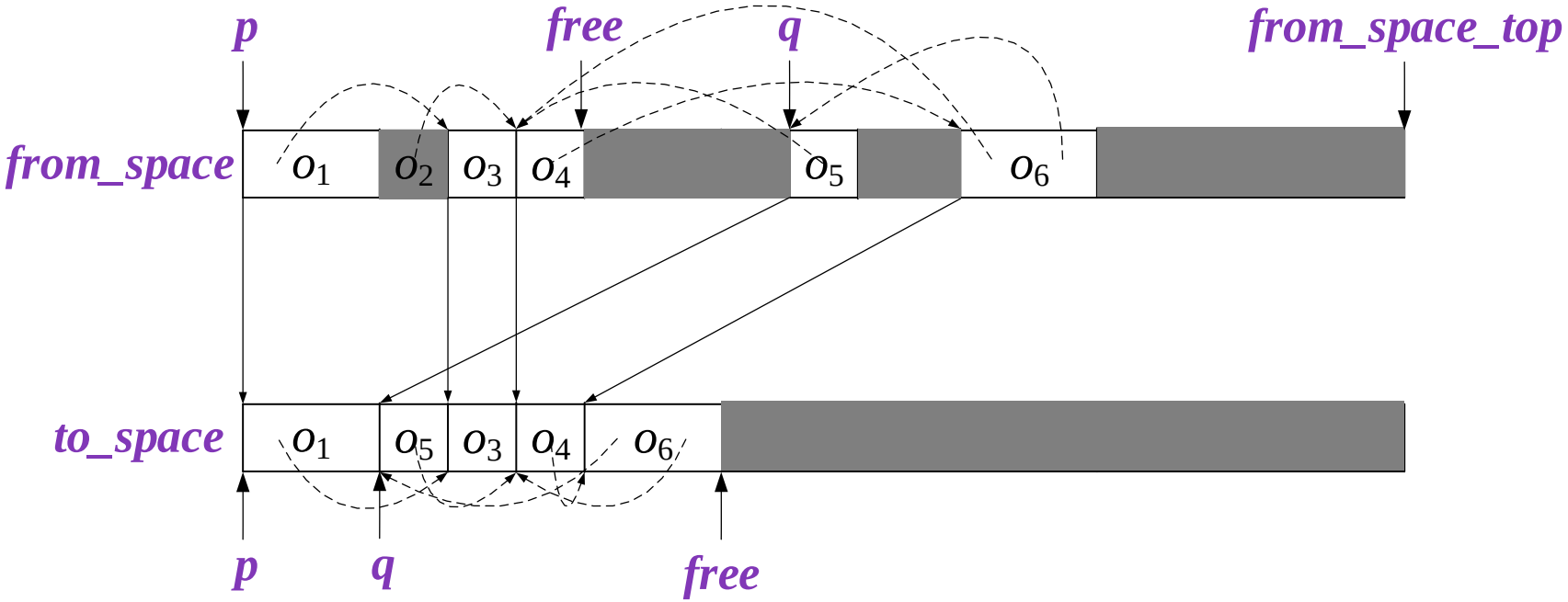
（*from_space* 为调换前的 *to_space* ）

垃圾回收（这讲）

✧ 垃圾回收算法

– 拷贝回收（*copying collection*）

- *Cheney*算法



垃圾回收（这讲）

✧ 垃圾回收算法

– 分代回收（*generational collection*）

- 核心思路：优先回收年轻“短命”数据对象

（1）堆空间被划分成多个分区，称为“代”，假设分编号为 G_0，G_1，G_2，…，G_n；最年轻的对象首先分配在 G_0；G_1中的所有对象比 G_0 中的任何对象都“年长”；等等

（2）每次新产生的数据对象均是在G_0分区分配，当空间不足时触发垃圾回收算法（如标记/清除或拷贝回收）；有所不同的是，根对象集合还包含 G_1，…，G_n中可能指向 G_0中对象的年长对象

垃圾回收（续讲）

✧ 垃圾回收算法

– 分代回收（*generational collection*）

- 核心思路：优先回收年轻“短命”数据对象 （续）

（3）将经历 G_0 的若干轮回收后仍能存活下来的对象“提升”至 G_1；若是某个时刻，G_1 已拥挤到不能再接纳从 G_0 提升的对象时，则启动对 G_0 和 G_1 相联合的回收算法；等等

（4）依此类推，“长寿”对象被逐级筛选出来，被重复处理的代价将大大降低

（5）缺陷：“长寿”对象的回收代价相当昂贵；
改进：与专注于“长寿”对象的垃圾回收方法联用，
如列车算法（*train algorithm*）

垃圾回收（选讲）

✧ 垃圾回收算法

– 增量回收（*incremental collection*）

- 核心思路

（1）每一轮次仅回收部分垃圾，但要确保遗留的垃圾（漂流垃圾）可以在后面轮次中回收

（2）在确保正确性和满足实时性要求的前提下，每一轮次所遗留的漂流垃圾越少越好

（3）关键技术：允许增量回收程序和更变程序交互或并发执行（如基于交叉执行来实现）

- 三色标记（*tricolor marking*）方案
- 例：*Baker* 增量回收算法（虎书）

面向对象程序运行时组织（选讲）

✧ 理解“类”和“对象”的角色

- 类扮演的角色是程序的静态定义
- 对象扮演的角色是程序运行时的动态结构
- 类是一组运行时对象的共同性质的静态描述

  类的**特征**（*feature*）成员：

  属性（*attribute*）和 **例程**（*routine*）
- 每个对象都必定是某个类的一个**实例**（*instance*），而一个类可以创建有许多个对象
- 实例对象是在程序运行时，根据该对象所属类的属性动态地构造的

面向对象程序运行时组织（选讲）

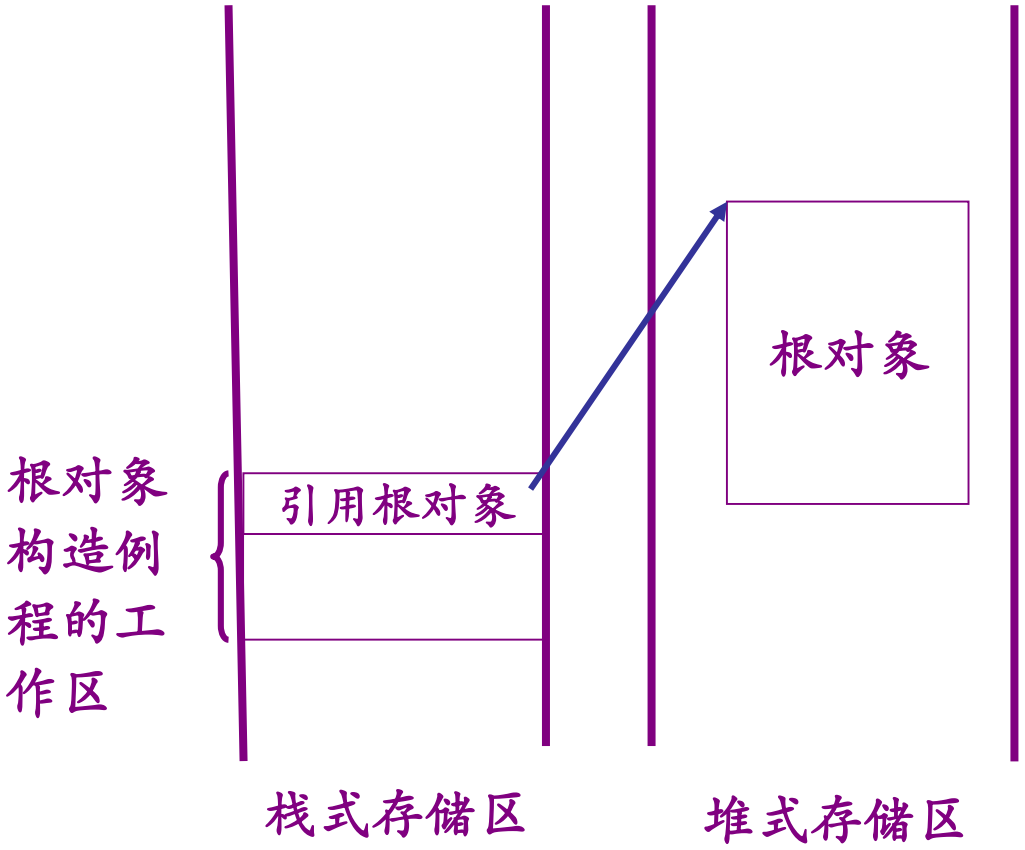
✧ 面向对象程序运行时的特征

- 对象是类的一个实例，是系统动态运行时一个物理结构的模块，是按需要创建、而不是预先分配的
- 对象是在类实例化过程中，由类的属性定义所确定的一组域动态地组成，每个域对应类中的一个属性
- 执行一个面向对象程序就是创建系统根类的一个实例，并调用该实例的创建过程
- 创建对象的过程即为实现该对象初始化，对于根类而言，创建其对象即执行该系统
- 创建根对象相当于通常软件启动 main，在非纯面向对象方式下，通常也用启动 main的方式创建根对象

✧ 面向对象程序运行时的特征

— 创建根对象时的存储结构

- 根对象的函数工作区中主要是运行该程序的启动参数，它们大都是根对象的成员。因而，根对象的函数工作区中主要存放对根对象的引用。
- 创建的根对象存放在堆式存储区中。



✧ 面向对象程序运行时的特征

– 例程运行时的特征

- 每个例程都必定是某个类的成员，且每个例程都只能把它的计算施加在它所属类所创建的对象上。因而在一个例程执行前，首先要求它所施加计算的对象已经存在，否则要求先创建该对象。
- 一个例程执行时，其参数除实参外，还用到它所施加计算的对象，它们与该例程的局部量及返回值一起组成一个该例程的工作区（放在栈式存储区中）。
- 例程工作区中的局部量若是较为复杂数据结构，则在工作区中存放对该复杂数据结构的一个引用，并在堆式存储区中创建一个该复杂数据结构的对象。

✧ 对象的存储组织

– 一个简单机制是，初始化代码将所有当前的继承特征（属性和例程）直接地复制到对象存储区中（将例程当作代码指针）。

但这样做较浪费空间。

✧ 对象的存储组织

– 另一种方法是在执行时将类结构的一个完整的描述保存在每个类的存储中，由超类指针维护继承性（形成所谓的继承图）。每个对象保存一个指向其定义类的指针，作为一个附加的域和它的属性变量放在一起，通过这个类就可找到所有（局部和继承的）的例程。此时，只记录一次例程指针（在类结构中），且对于每个对象并不将其复制到存储器中。

其缺点在于：虽然属性变量具有可预测的偏移量（如在标准环境中的局部变量一样），但例程却没有，它们必须由带有查询功能的符号表结构中的名字来维护。这是对于诸如 *Smalltalk* 等强动态性语言的合理的结构，因为类结构可以在执行中改变。

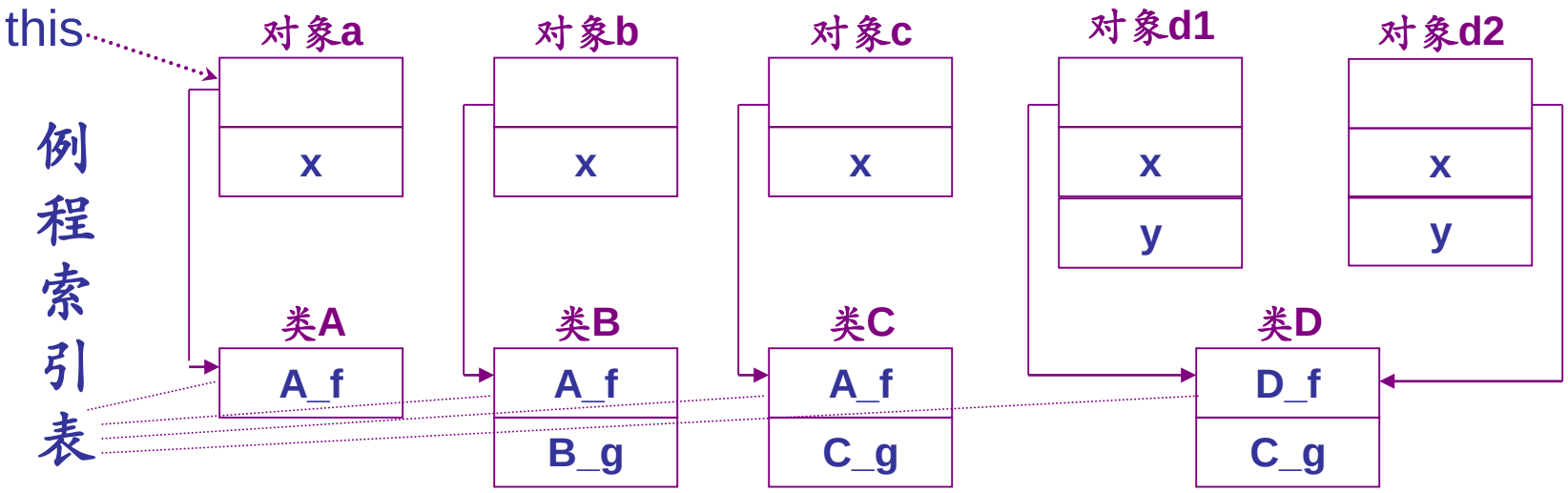
✧ 对象的存储组织

– 一种折衷方案：计算出每个类的可用例程的代码指针列表（称为例程索引表，如 C++ 的 *Vtable*，简称虚表）。

其优点在于：可做出安排以使每个例程都有一个可预测的偏移量，而且也不再需要用一系列表查询遍历类的层次结构。这样，每个对象不仅包括属性变量，还包括了一个相应的例程索引表的指针（不是类结构的指针）。

✧ 某个单继承O-O语言的对象存储示例

```
class A { int x; void f (){…} }
class B extends A {void g(){…} }
class C entends B {void g(){…} }
class D extends C{bool y; void f (){…}}
class A a; class B b; class C c; class D d1,d2;
```

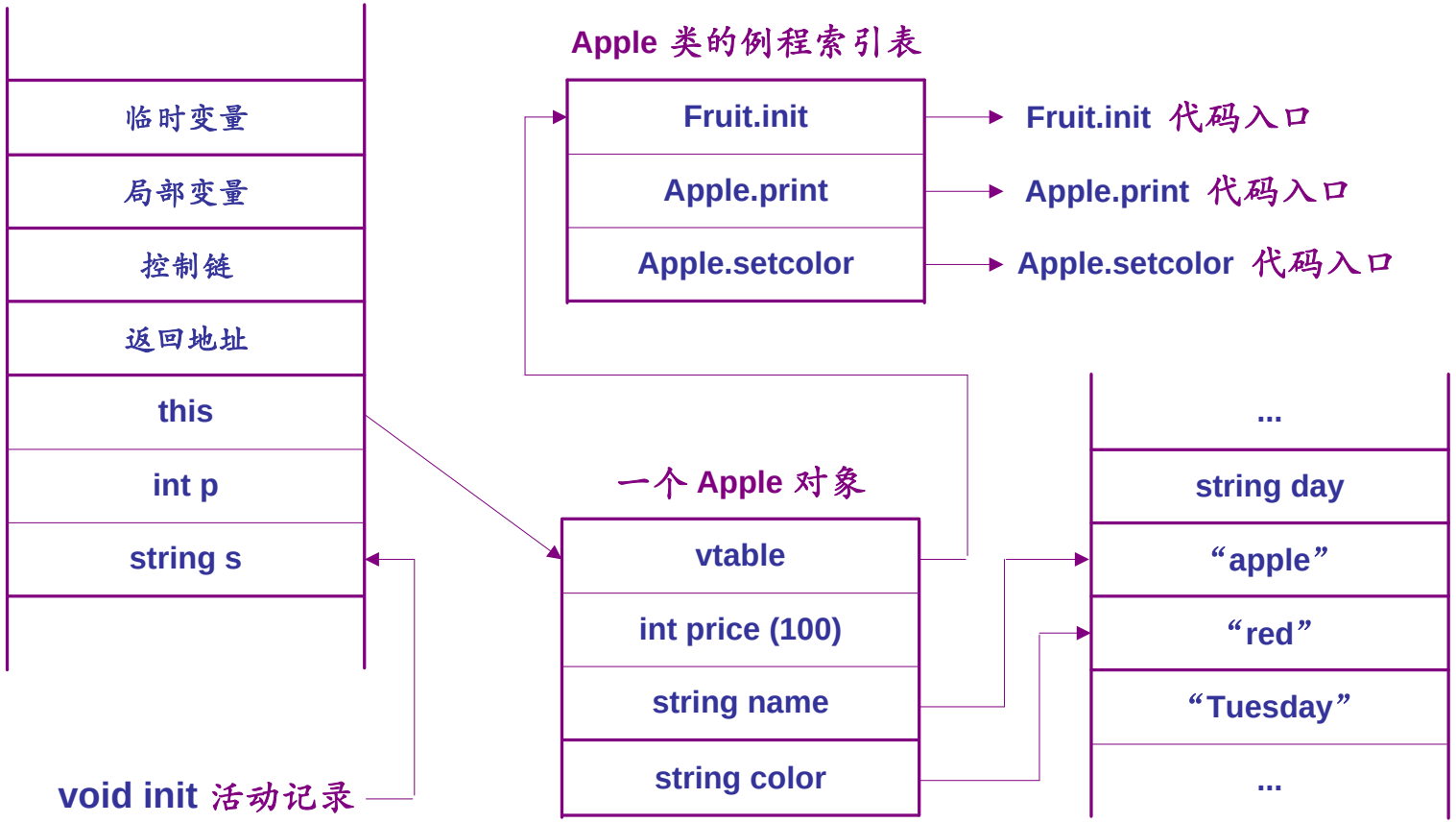


✧ 某个单继承O-O语言的对象存储示例

```
string day;
class Fruit
{
   int price;
   string name;
   void init(int p,string s){price=p; name=s;}
   void print(){
   Print("On ",day,", the price of ",name,
      " is ",price,"\n");}
}
class Apple extends Fruit
{
   string color;
   void setcolor(string c){color=c;}
   void print(){
      Print("On ",day,", the price of ",color,
      " ",name," is ", price,"\n");}
}
```

```
void foo()
{
   class Apple a;
   a=New (Apple);
   a.setcolor("red");
   a.init(100,"apple");
   day="Tuesday";
   a.print();
}
```

✧ 某个单继承O-O语言的对象存储示例



面向对象程序运行时组织（选讲）

✧ 其他话题

- 类成员测试（*Testing Class Membership*）
- 对象的创建和撤消
  - 构造函数和析构函数（执行次序）
  - 垃圾回收
- 对象的操作
  - 对象的赋值、克隆、比较、持久存储
- 多重继承
- 例外处理（*Exception Handling*）

……

✧ 函数式程序的运行时特征

- 函数式语言特别是纯函数式语言，所涉及的对象主体是数学对象，具有不可更变的（*immutable*）性质，即只能被定义（初始化）一次

  纯函数式语言支持等式推理

✧ 函数式程序的运行时特征（续）

- 数学对象的增长速率一般会很快，运算中间结果占比居多数，需要高效的垃圾回收机制

  （垃圾回收机制：参考前面一节）

✧ 函数式程序的运行时特征（续）

- 函数式语言中函数是一类对象（即高阶函数，函数本身可以用作其他函数的参数或返回值）

  函数对象的代码是函数对象的重要部分，函数对象的求值需要一个求值环境

  函数代码及其求值环境的组合称之为闭包（*closure*）

  核心技术之一：闭包的运行时存储组织

✧ 函数式程序的运行时特征（续）

- 某些编译优化技术、以及良好的编程风格和技巧，可改善运行时存储组织，从而提高性能，例如：

（1）尽可能使用尾调用和尾递归，可以简化活动记录的设计，优化调用代码序列

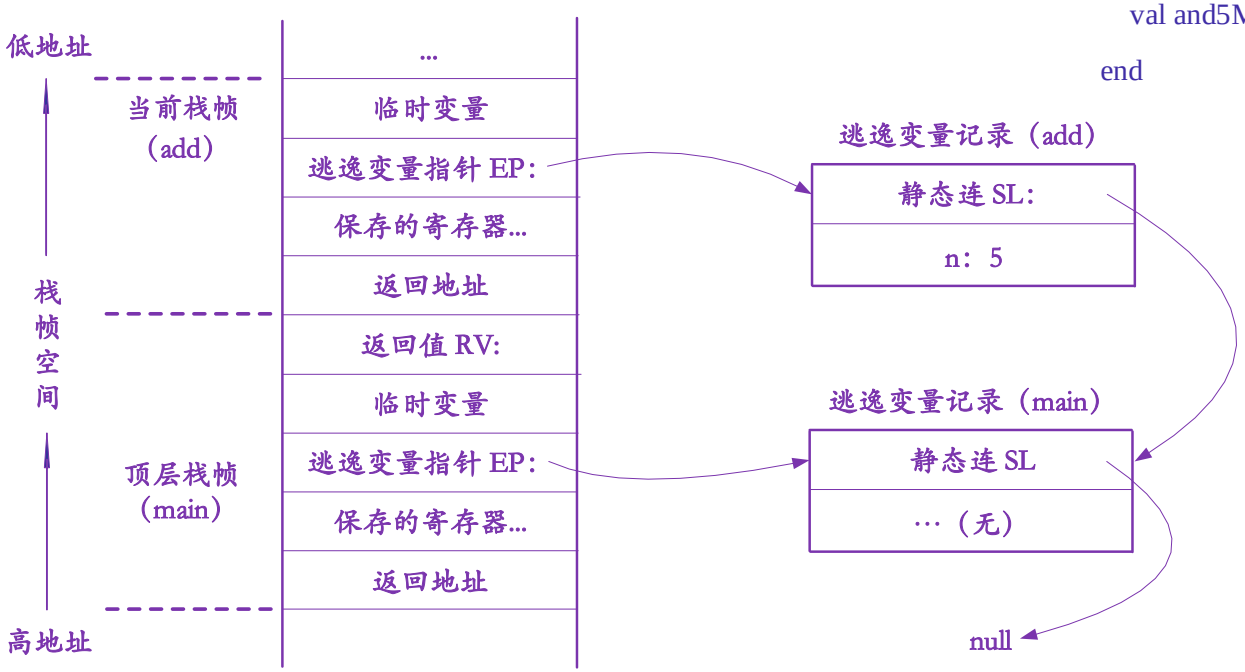
（2）对于纯函数式语言，可以通过延迟求值（或传名调用）来支持函数的某些非严格定义，从而可拓展等式推理的内涵；为避免延迟求值可能带来的重复计算，可借助创建和维护函数记忆簿（*memoization*），从而实现以存储代价换取优化性能的惰性求值机制

✧ 闭包的存储组织

– 逃逸变量（*escaping variables*）

- 示例（进入add函数时）

```
let
   type intfun = int -> int
   fun add (n: int) : intfun =
      let fun h (m: int) : int =
         n+m
      in h
      end
   val addFive : intfun = add (5)
   fun map f nil = nil
      | map f (h::t) = (f h) :: (map f t)
   val and5Map = map (addFive)
in
   val and5Map([1, 2, 3, 4])
end
```

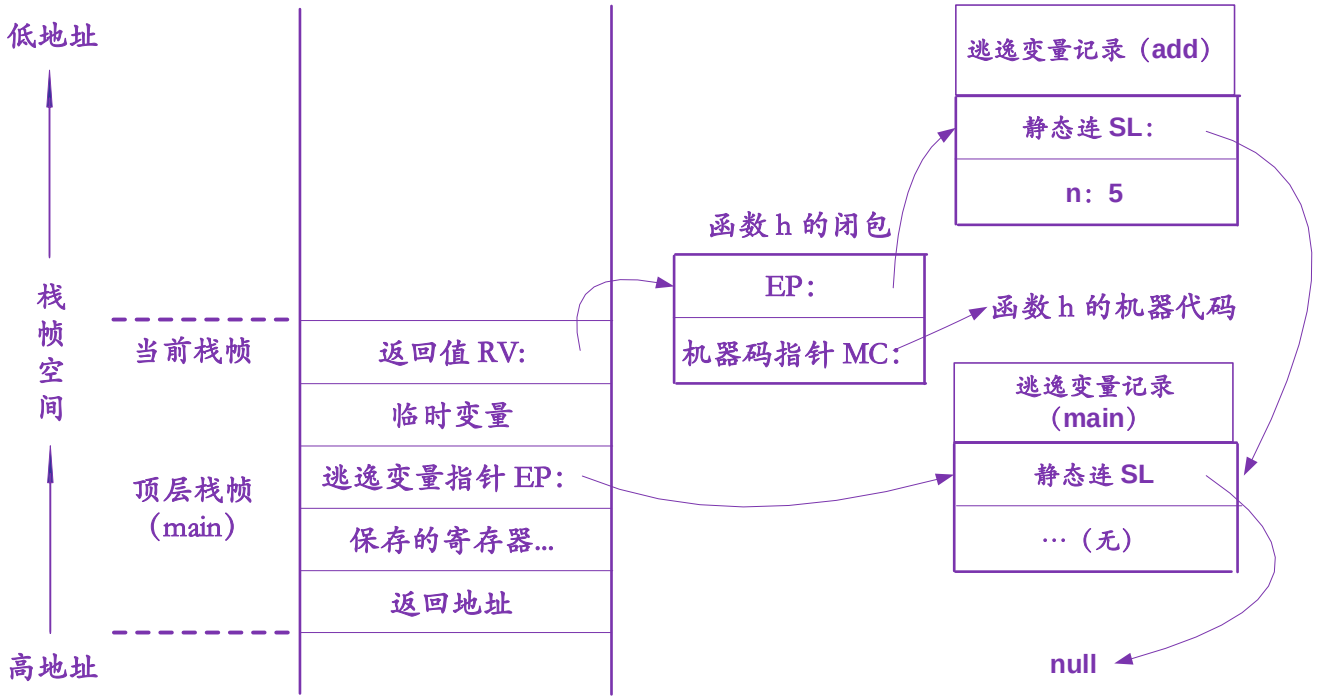


函数式程序运行时组织（续讲）

✧ 闭包的存储组织（续）

– 逃逸变量（*escaping variables*）

- 示例（撤销 add 函数栈帧并返回 main时）



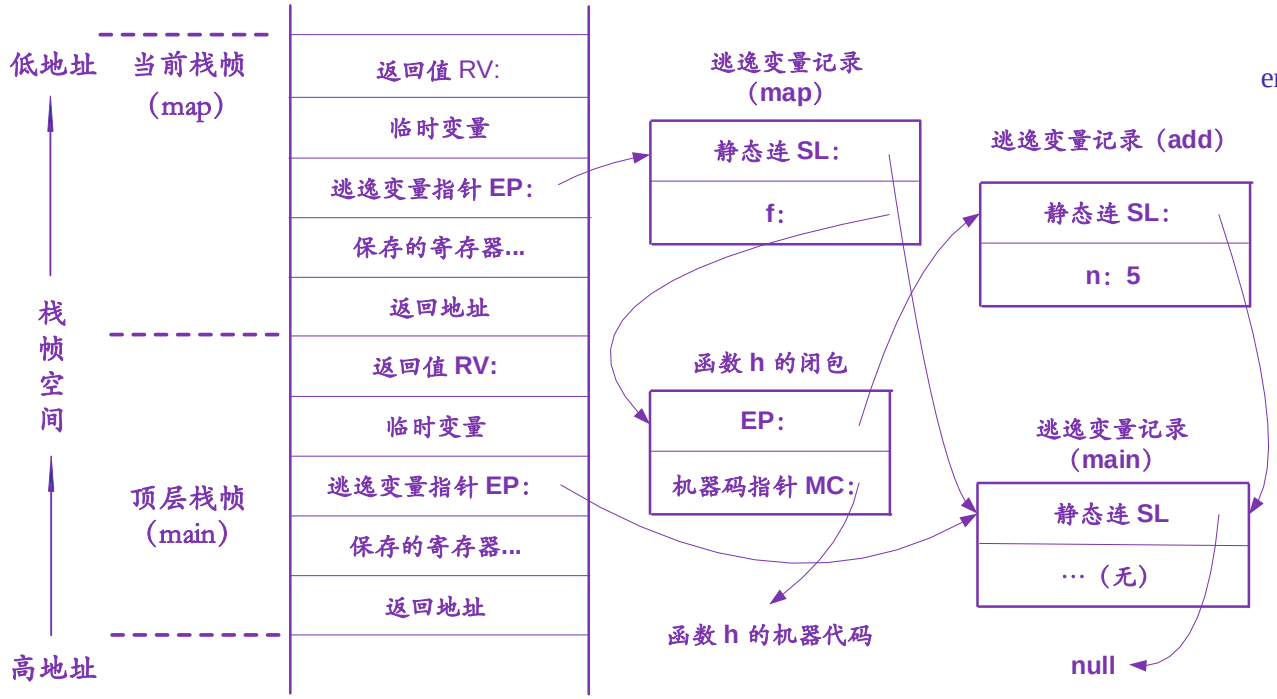

```
let
   type intfun = int -> int

   fun add (n: int) : intfun =
      let fun h (m: int) : int =
          n+m
      in h
      end

   val addFive : intfun = add (5)

   fun map f nil = nil
     | map f (h::t) = (f h) :: (map f t)

   val and5Map = map (addFive)
in
   val and5Map([1, 2, 3, 4])

end
```

✧ 闭包的存储组织（续）

– 逃逸变量（*escaping variables*）

- 示例（在调用map (add (5))后，
  并执行至map函数内部时）

```
let
   type intfun = int -> int

   fun add (n: int) : intfun =
      let fun h (m: int) : int =
         n+m
      in h
      end

   val addFive : intfun = add (5)

   fun map f nil = nil
      | map f (h::t) = (f h) :: (map f t)

   val and5Map = map (addFive)
in
   val and5Map([1, 2, 3, 4])

end
```



课后作业

参见网络学堂公告：“第四次书面作业”

（非书面作业）

1. 结合本讲稿，以及课外参考书籍，更加深刻体会面向对象语言实现中的存储机制（编译时/运行时）。
2. 如有时间和兴趣，最好能找机会自学函数式语言实现中存储组织相关的更多内容。

That's all for today.

Thank You