第六章 运行时的存储组织及管理

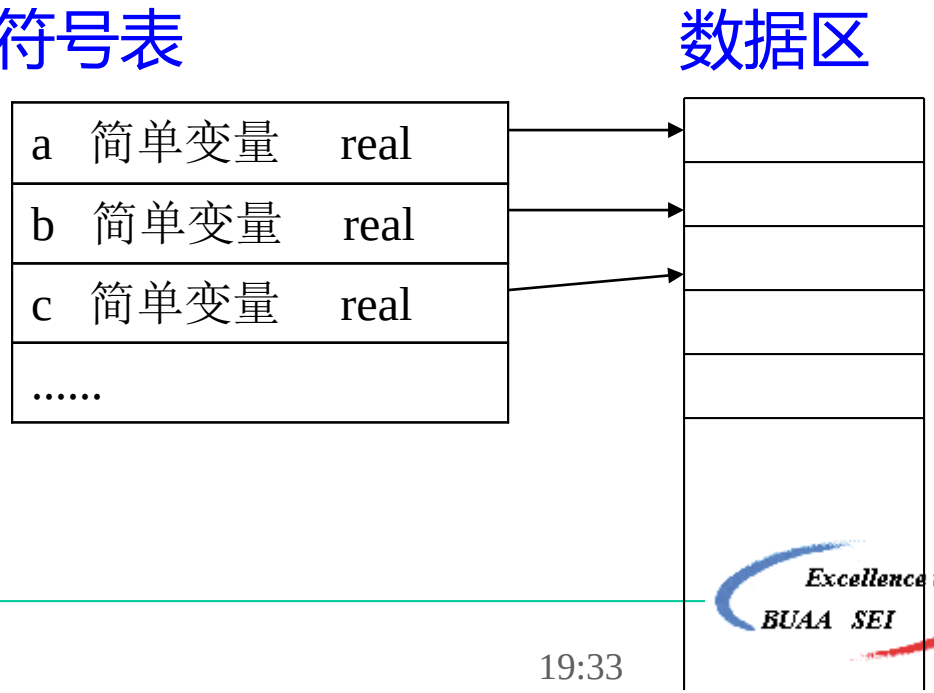
- 概述
- 静态存储分配
- 动态存储分配

6.1 概述

(1) 运行时的存储组织及管理

目标程序运行时所需存储空间的组织与管理以及源程序中变量存储空间的分配。

例：real a, b, c ;
...
a := b*c ;
⇩
取b
* c
送a



(2) 静态存储分配和动态存储分配

静态存储分配

在编译阶段由编译程序实现对存储空间的管理和为源程序中的变量分配存储的方法。

条 件

如果在编译时能够确定源程序中变量在运行时的数据空间大小，且运行时不改变，那么就可以采用静态存储分配方法。

但是并不是所有数据空间大小都能在编译过程中确定

动态存储分配

在目标程序运行阶段由目标程序实现对存储空间的组织与管理，和为源程序中的变量分配存储的方法。

特　点

- 在目标程序运行时进行变量的存储分配。
- 编译时要生成进行动态分配的目标指令。

6.2 静态存储分配

(1) 分配策略

由于每个变量所需空间的大小在编译时已知，因此可以用简单的方法给变量分配目标地址。

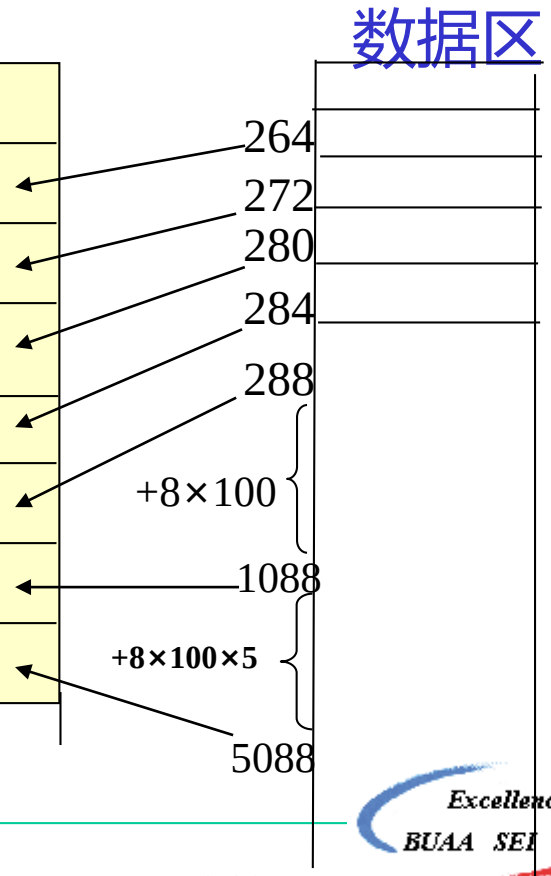
- 开辟一数据区。（首地址在加载时定）
- 按编译顺序给每个模块分配存储空间。
- 在模块内部按顺序给模块的变量分配存储，一般用相对地址，所占数据区的大小由变量类型决定。
- 目标地址填入变量的符号表中。

例：有下列FORTRAN 程序段

```
real        MAXPRN, RATE
integer     IND1, IND2
real        PRINT(100),YPRINT(5,100),TOTINT
```

假设整数占4个字节大小，实数占8个字节大小，则符号表中各变量在数据区中所分配的地址为：

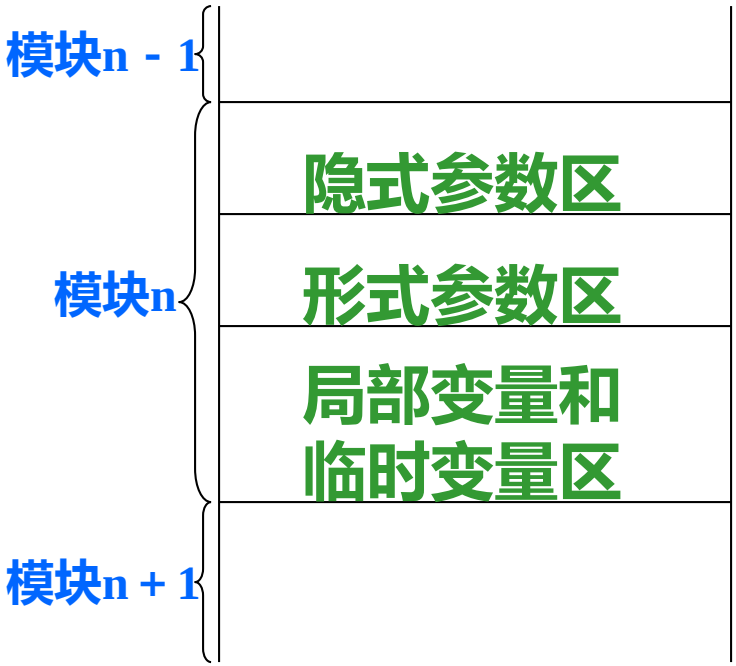
名字	类型	维数	地址
MAXPRN	r	0	264
RATE	r	0	272
IND1	i	0	280
IND2	i	0	284
PRINT	r	1	288
YPRINT	r	2	1088
TOTINT	r	0	5088



(2) 模块(FORTRAN子程序)的完整数据区

- 变量
- 返回地址
- 形式参数
- 临时变量(如表达式计算的中间结果等)

FORTRAN子程序的典型数据区



隐式参数区:返回地址
函数返回值

形式参数区:存放相应实
参信息(值或地址)

6.3 动态存储分配

- 编译时不能具体确定程序所需数据空间
- 编译程序生成有关存储分配的目标代码
- 实际上的分配要在目标程序运行时进行

分程序结构，且允许递归调用的语言：

栈式动态存储分配

分配策略： 整个数据区为一个堆栈，

(1) 当进入一个过程时，在栈顶为其分配一个数据区。

(2) 退出时，撤消过程数据区。

静态存储 vs 动态存储

静态存储

动态存储

(1)当进入一个过程时，在栈顶为其分配一个数据区。
(2)退出时，撤消过程数据区。

例1:

```
1  BBLOCK;                                  (1)
       REAL  X, Y;  STRING   NAME;
2      M1: PBLOCK(INTEGER IND) ;            (2)
           INTEGER   X;
                                            (3)
           CALL  M2(IND+1);
                                            (7)
       EDN  M1;
3      M2:  PBLOCK(INTEGER J);              (3)
4          BBLOCK;                          (4)
               ARRAY  INTEGER F(J);
               LOGICAL TEST1;
                                            (5)
           END
       END  M2;                             (6)
                                            (2)
       CALL  M1(X/Y);
END;                                        (8)
```

AR4 F，TEST1数据区

运行中数据区的分配情况：

(a)

(b)

(c)

(d)

(e)

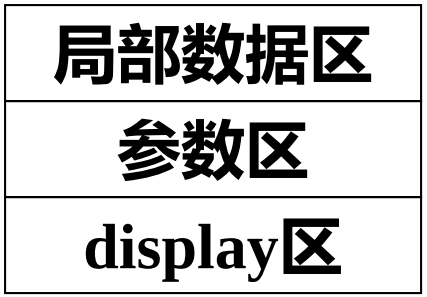
(f)

(g)

```
BBLOCK;
1    REAL X,Y;   STRING   NAME;
   M1: PBLOCK(INTEGER  IND);
         INTEGER   X;
2            CALL  M2(IND+1);
   END  M1;
   M2:  PBLOCK(INTEGER J);
          BBLOCK;
3            ARRAY  INTEGER F(J);
        4    LOGICAL TEST1;
           END ;
   END  M2;
     CALL  M1(X/Y);
END
```

6.3.1 活动记录

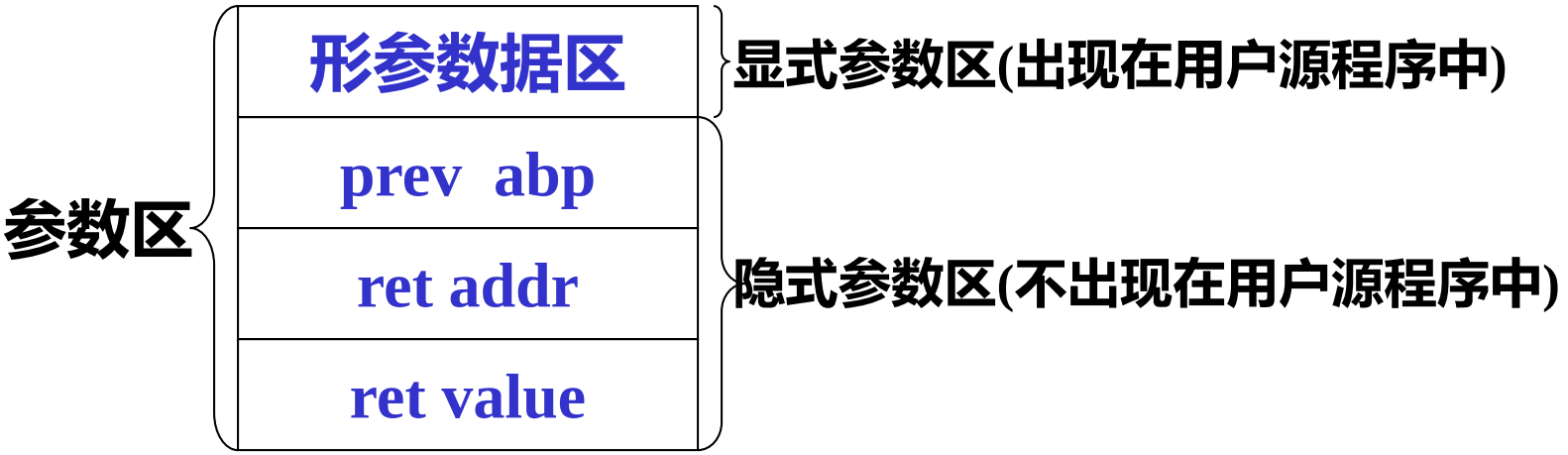
一个典型的活动记录可以分为三部分：



(1) 局部数据区：

存放模块中定义的各个局部变量。

(2) 参数区： 存放隐式参数和显式参数。



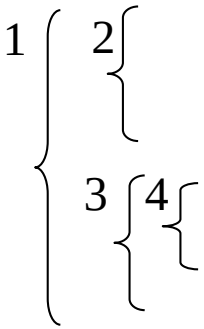
形参数据区： 每一形参都要分配数据空间,形参单元中存放实参值或者实参地址

prev abp： 存放调用模块活动记录基地址,函数执行完时,释放其数据区，数据区指针指向调用前的位置

ret addr：返回地址，即调用语句的下一条执行指令地址

ret value： 函数返回值(无值则空)

(3) display区：存放各外层模块活动记录的基地址。



对于例1中所举的程序段，模块4可以引用模块1和模块3中所定义的变量，故在模块4的display，应包括AR1和AR3的基地址。

变量二元地址(BL、ON)

BL：变量声明所在的层次。

可得到该层数据区开始地址

并列过程具有相同层次

ON：相对于显式参数区的开始位置的位移。

相对地址

例如：程序块1
X：（1，0）
Y：（1，1）
NAME：（1，2）

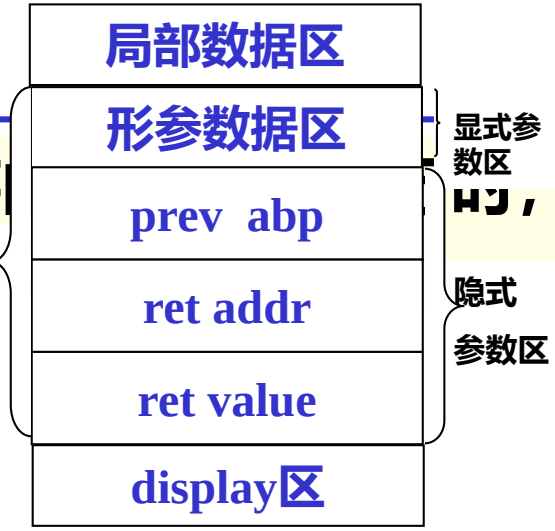
过程块M1
IND：（2，0）
X：（2，1）

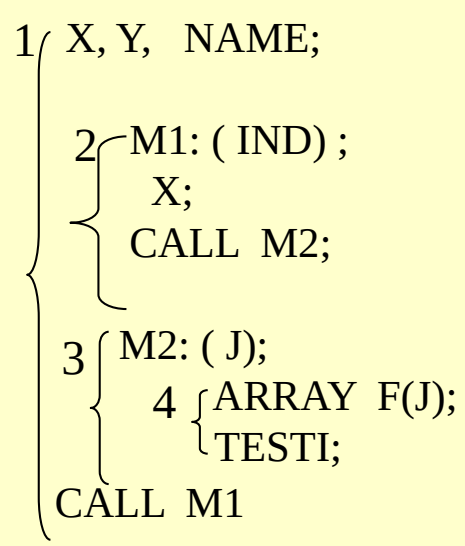
```
BBLOCK;
    REAL X,Y;  STRING  NAME;
(1) M1: PBLOCK(INTEGER  IND);
        INTEGER  X;
(2)         CALL  M2(IND+1);
    END  M1;
    M2:  PBLOCK(INTEGER J);
        BBLOCK;
(3)         ARRAY  INTEGER F(J);
(4)         LOGICAL TEST1;
        END ;
    END  M2;
    CALL  M1(X/Y);
END
```

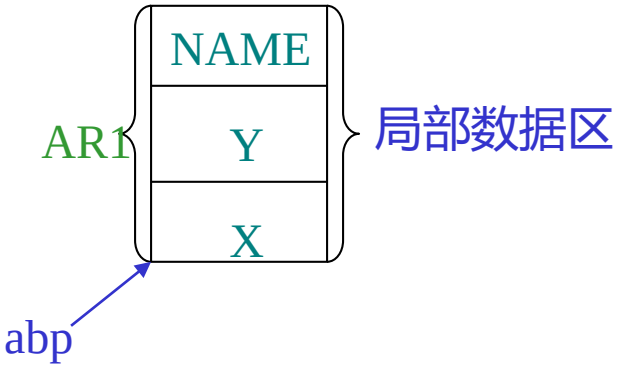
高层(内层)模块可以引用低层(外层)模块中的变量，例如在M1中可引用外层模块中定义的变量Y。

在M1的display区中可找到程序块1的活动记录基地址, 加上Y在数据区的相对地址就可以求得Y的绝对地址。

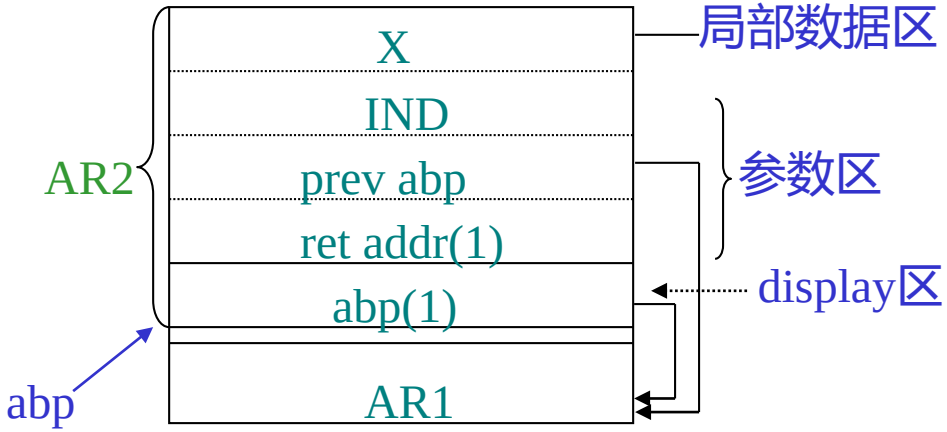
```
1 BBLOCK;
    REAL X,Y;  STRING  NAME;
  M1: PBLOCK(INTEGER  IND);
2       INTEGER  X;
            CALL  M2(IND+1);
  END  M1;
  M2:  PBLOCK(INTEGER J);
3       BBLOCK;
4           ARRAY  INTEGER F(J);
            LOGICAL TEST1;
        END ;
  END  M2;
    CALL  M1(X/Y);
END
```







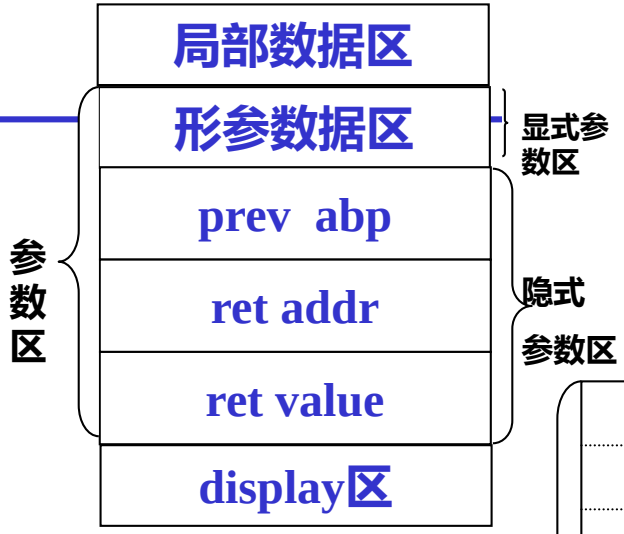
(a) 进入模块1

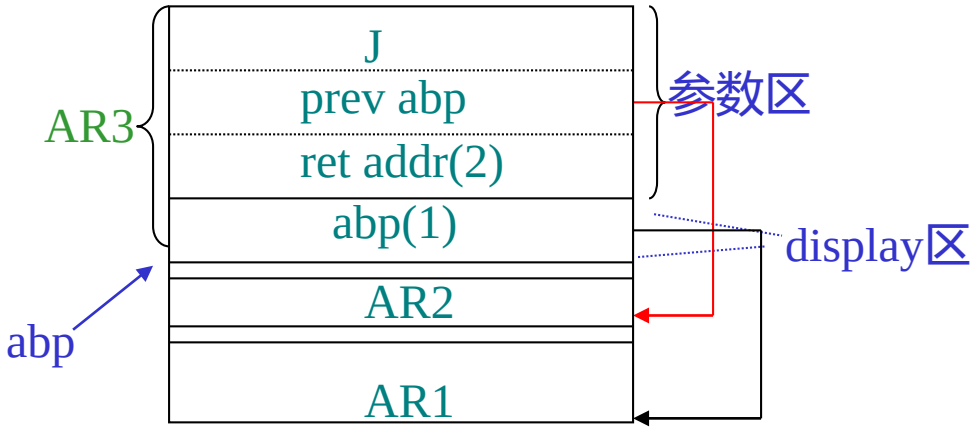


(b) M1被调用

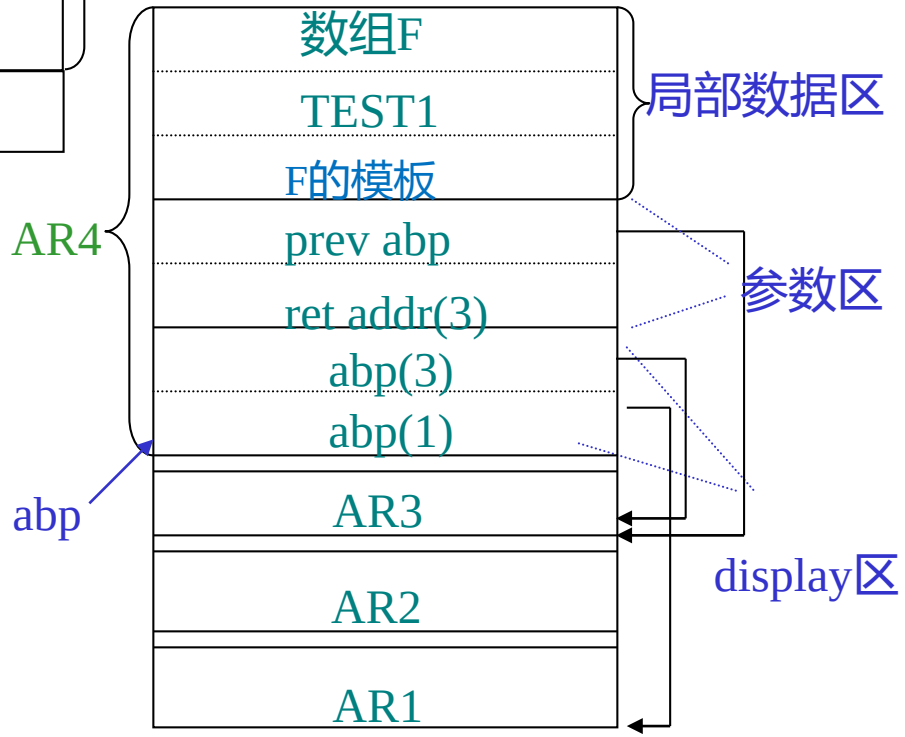
函数无返回值，所以没有ret value

```
1  X, Y,   NAME;

   2  M1: ( IND) ;
        X;
      CALL  M2;

   3  M2: ( J);
        4  ARRAY  F(J);
           TESTI;
   CALL  M1
```





(c)M2被调用



(d) 进入内模块4

（e）当模块4执行完，则abp:=prev abp，这样abp恢复到进入模块4时的情况，运行栈情况如（c）

（f）当M2执行完，则abp:=prev abp，这样abp恢复到进入M2时的情况，运行栈情况如（b）

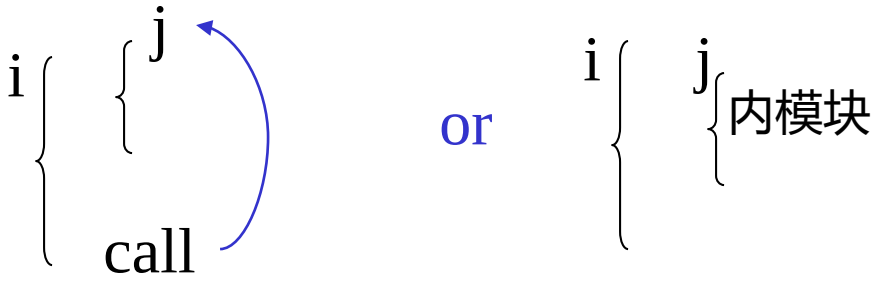
（g）当M1执行完，则abp:=prev abp，这样abp恢复到进入M1时的情况，运行栈情况如（a）

（h）当最外层模块执行完，运行栈恢复到进入模块时的情况，运行栈空

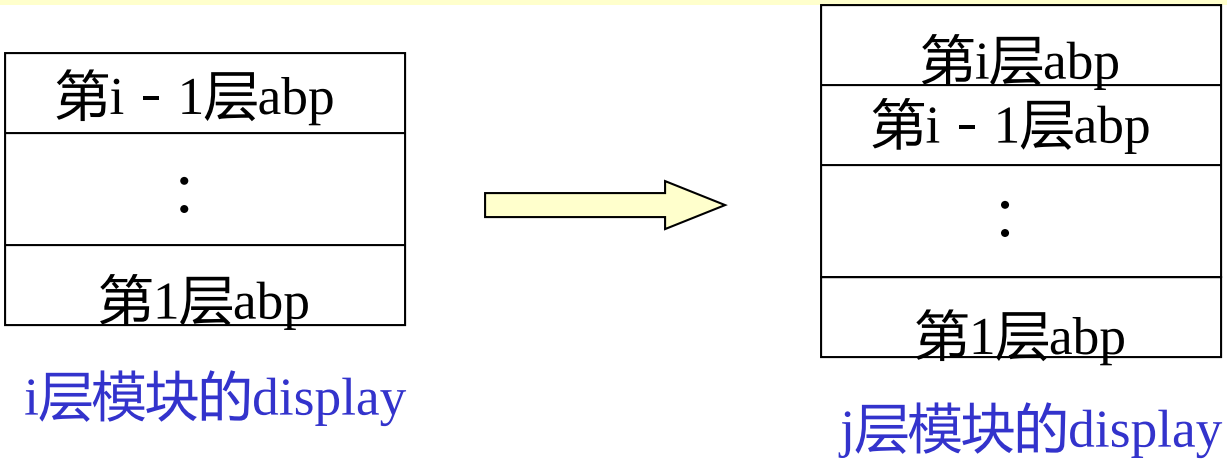
6.3.2 建造display区的规则

从i层模块进入(调用)j层模块，则：

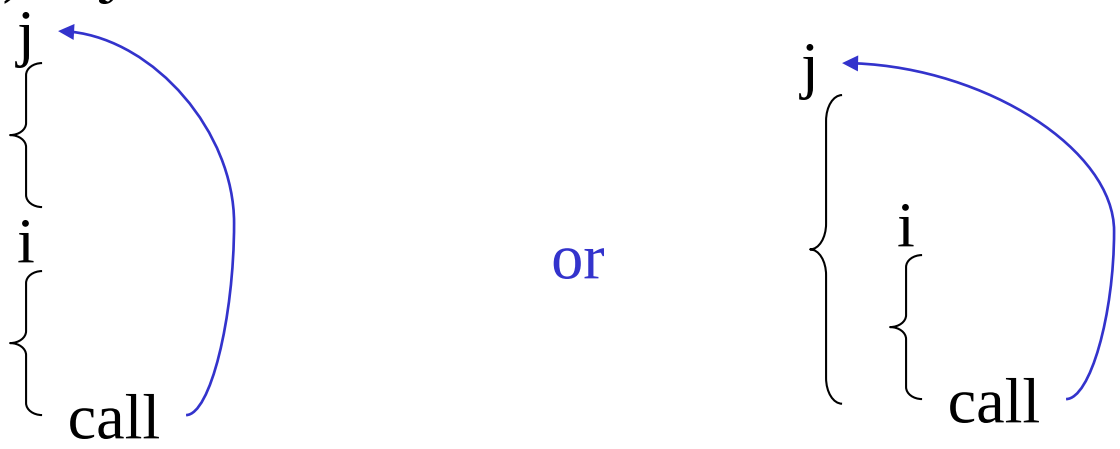
(1) 若j = i + 1



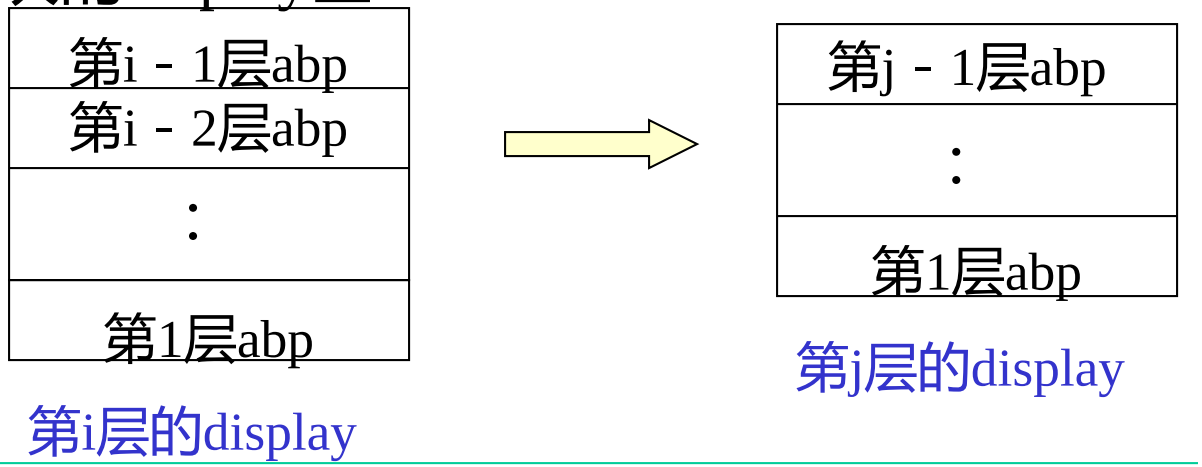
复制i层的display，然后增加一个指向i层模块记录基地址的指针



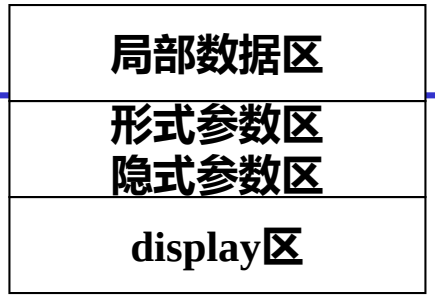
(2) 若j≤i 即调用外层模块或同层模块



将i层模块的display区中的前面j－1个入口复制到第j层模块的display区



6.3.3 运行时的地址计算



设要访问的变量的二元地址为：（BL，ON）

该变量在LEV层模块中引用

地址计算公式：

Display区大小

隐式参数区大小

```
if  BL = LEV  then
            addr := abp + (BL-1) + nip + ON
else  if  BL < LEV  then
            addr := display[BL] + (BL-1) + nip + ON
else
     write(“地址错，不合法的模块层次”）
```

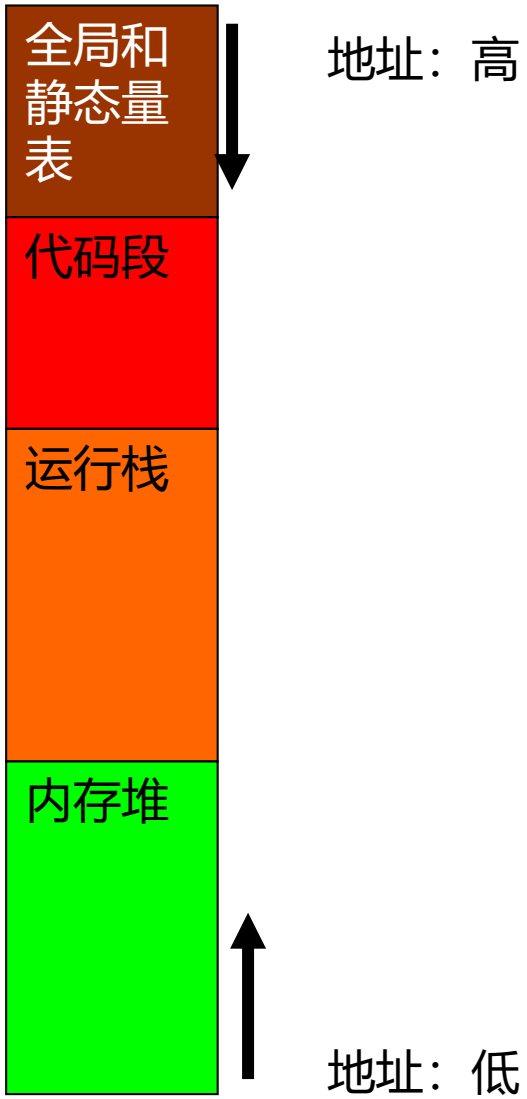
作业：P119　1
P133 2,3

注：P133-2中，PROCEDURE的名字为EXAMPLEPROC,去掉前面“RE”

补充：运行时的存储管理

- 全局和静态量表
- 代码段
- 运行栈
- 内存堆

- 以MS-WIN+VS+X86下的可执行程序为例，从高地址到低地址，自上而下的是：
  - 全局和静态量表
  - 代码段
  - 运行栈
  - 内存堆



全局和静态量表

```
int global_c = 0 ;

void foo(int a)
{
    static int s_c = 0 ;

    s_c += a ;

    global_c = s_c ;
}
```

```
00427e34        global_c
00427e38        s_c
...
```

```
...
12:     s_c += a ;
00401028  mov     eax,[global_c+4 (00427e38)]
0040102D  add     eax,dword ptr [ebp+8]
00401030  mov     [global_c+4 (00427e38)],eax
13:
14:     global_c = s_c ;
00401035  mov     ecx,dword ptr [global_c+4 (00427e38)]
0040103B  mov     dword ptr [global_c (00427e34)],ecx
...
```

运行栈

- 子程序/函数运行时所需的基本空间
- 进入子程序/函数时分配，地址空间向下生长（从高地址到低地址）
- 从子程序/函数返回时，当前运行栈将被废弃
- 递归调用的同一个子程序/函数，每次调用都将获得独立的运行栈空间

运行栈实例分析

- 一个典型的运行栈包括
  - 函数的返回地址
  - 全局寄存器的保存区
  - 临时变量的保存区
  - 未分配到全局寄存器的局部变量的保存区
  - 其他辅助信息的保存区
    - 例，PASCAL类语言的DISPLAY区

一个XScale(ARM)上的Java/C/C++函数运行栈的示意图

内存堆

- 内存堆用来存放哪些数据？
  - 函数/子程序活动结束时仍需保持的数据
  - 程序运行前无法得知所需空间大小的数据
  - 并非全局量或者静态量

内存堆实例分析

```
25:             p = (char*)malloc(len) ;
004010A9    mov         eax,dword ptr [ebp+8]
004010AC    push        eax
004010AD    call        malloc (00401150)
004010B2    add         esp,4
004010B5    mov         dword ptr [ebp-4],eax
26:
27:             return p ;
004010B8    mov         eax,dword ptr [ebp-4]
```

eax = 0x00031000

栈式分配和堆式分配的比较

栈	堆
解决了函数的递归调用等问题	解决了动态申请空间的问题
由编译器自动管理	由程序员控制空间的申请和释放工作
向内存地址减少的方向增长	向内存地址增加的方向增长
不会产生碎片	会产生碎片
计算机底层支持，分配效率高	C函数库支持，分配效率低

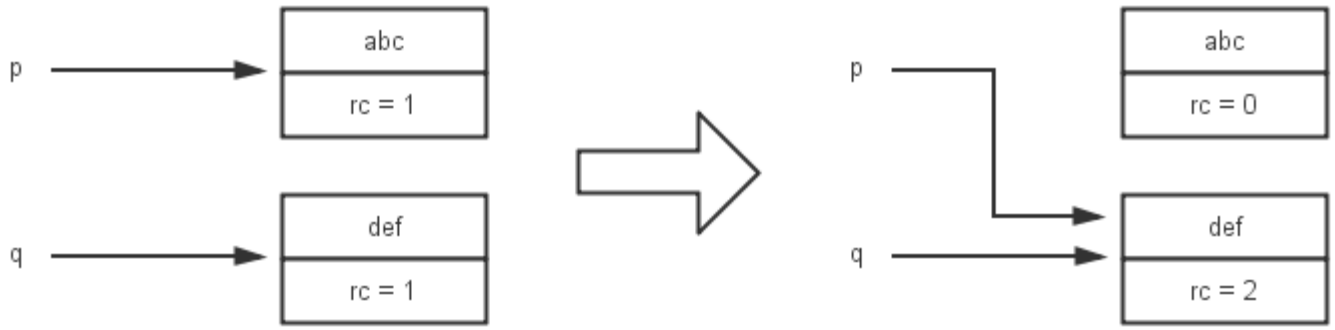
内存垃圾收集器

- 内存垃圾回收器(Garbage Collector, 简称GC)，是一种自动内存管理机制
- 第一次出现于1958年，由John McCarthy首先实现，使之作为Lisp实现的一部分
- 包括Java、Python、Perl、Modula-3、Prolog、ML和Smalltalk等语言的运行时系统都含有GC

GC的典型技术

- 无需标记和“stop-the-world”回收
  - 引用计数
- 需要标记和“stop-the-world”回收
  - 标记和清除
  - 标记紧缩
  - 拷贝回收
  - 分代回收

引用计数

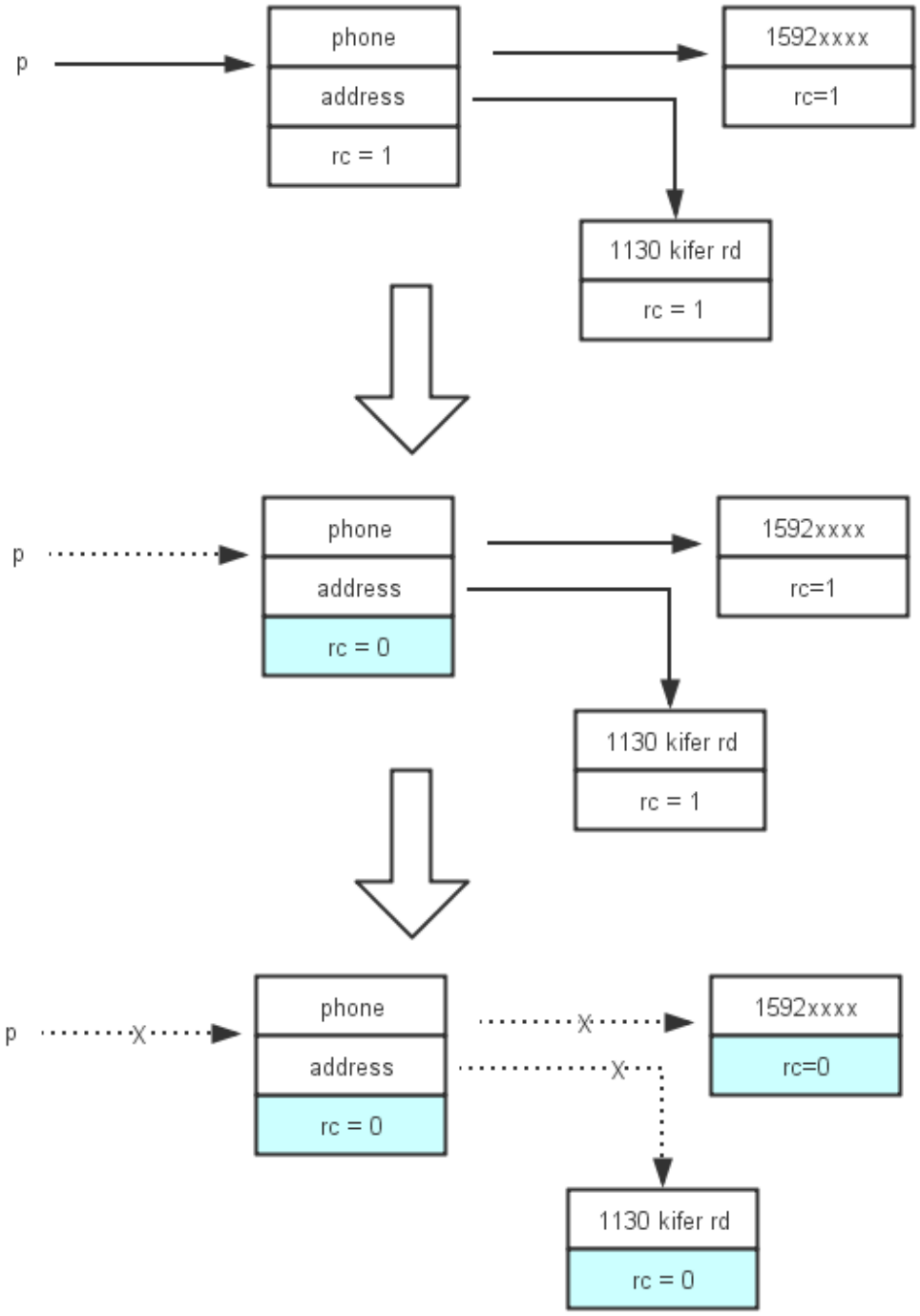


- p指针指向q的对象时，p原指向的对象计数减1，q的计数加1

- **计数减为0的对象，系统随时可以回收**

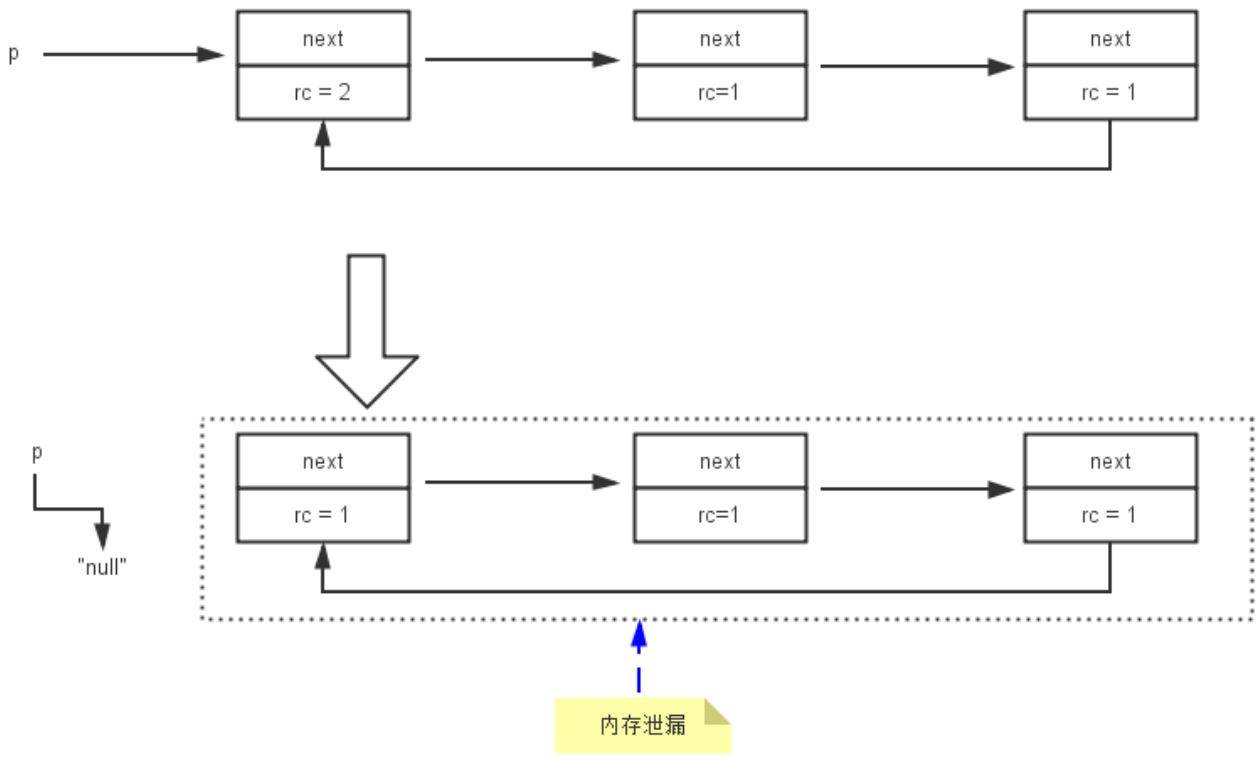
引用计数的局限和问题

- 效率问题
  - 每次改变对象指针均需要递归改写计数（出现引用为0的情况）

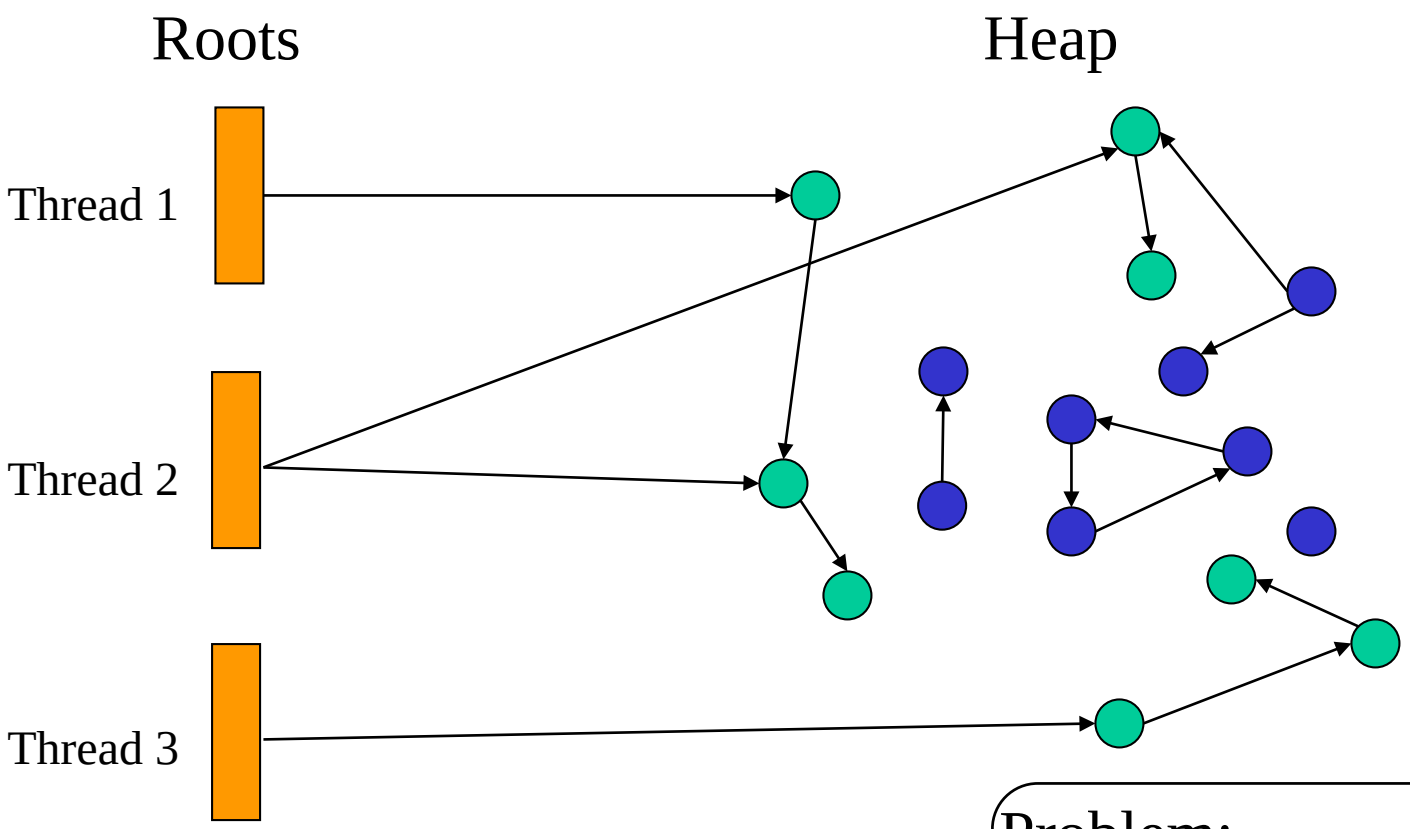


引用环问题

- 导致内存泄漏
- 通过“弱引用”解决
  - SWIFT
  - 华为方舟



How Mark/Sweep Works

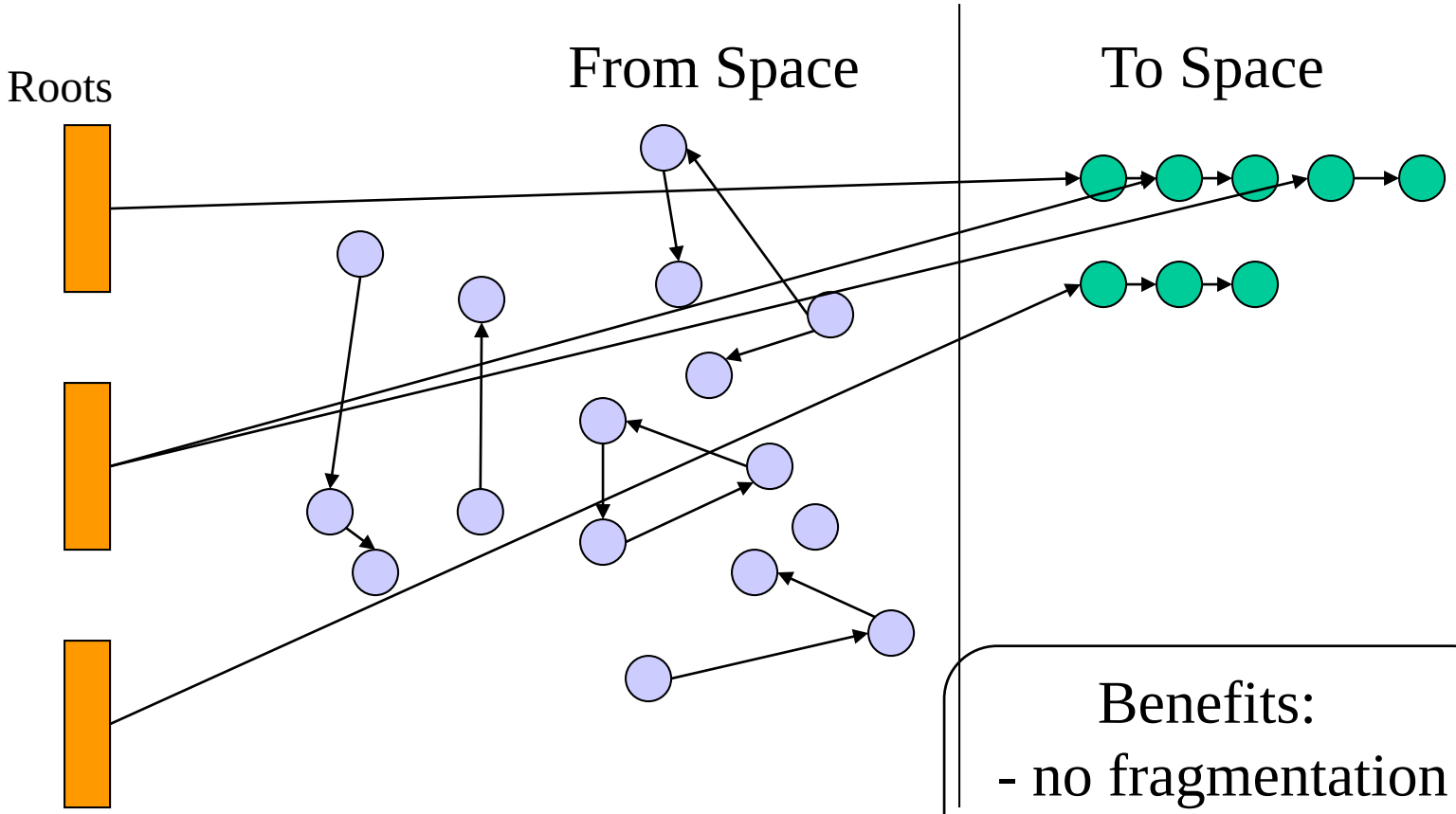


Problem:
- Fragmentation
- Locality

Reachable Objects

Garbage

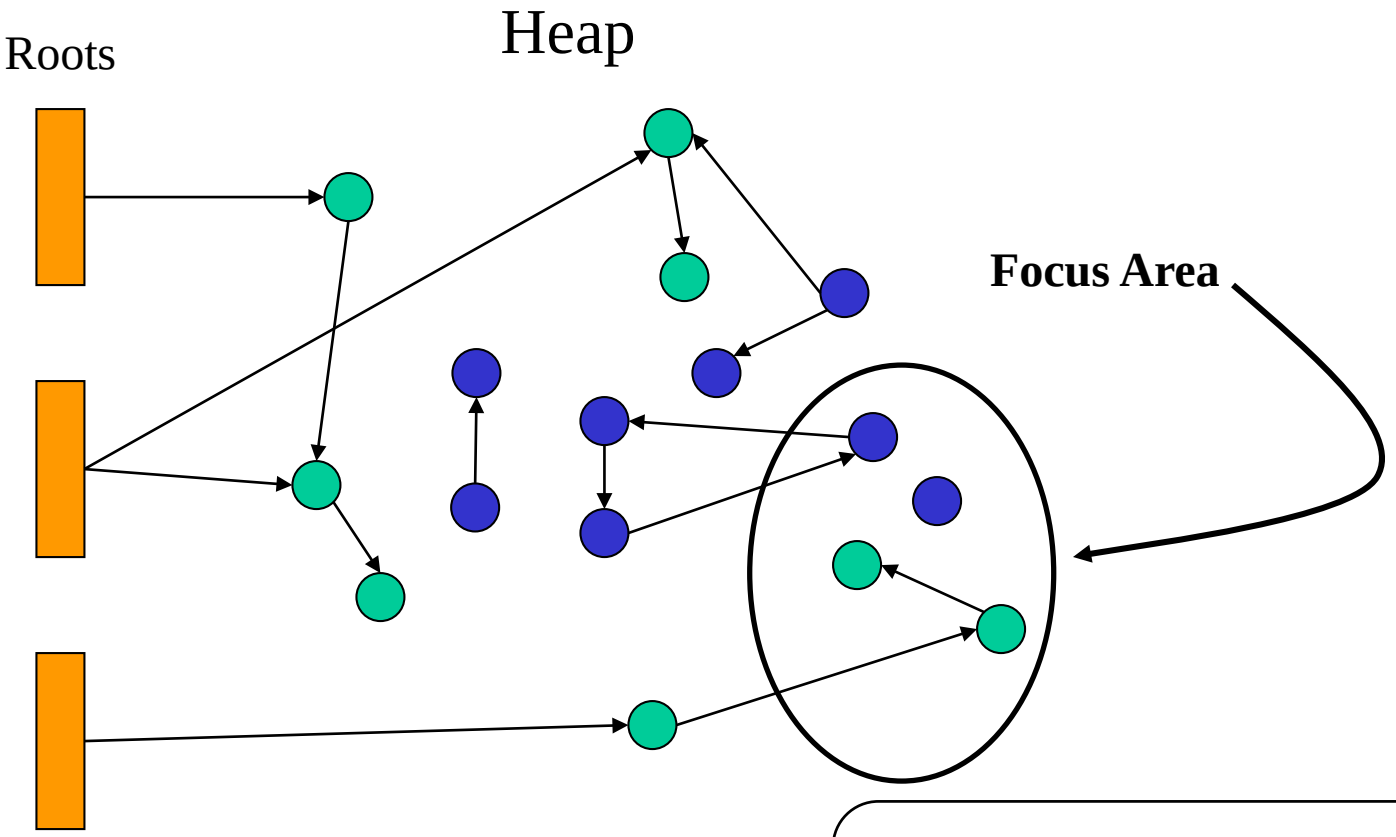
How Copying Works



Benefits:
- no fragmentation
- cache utilization
- fast allocation

Reachable Objects
Reclaimed Objects

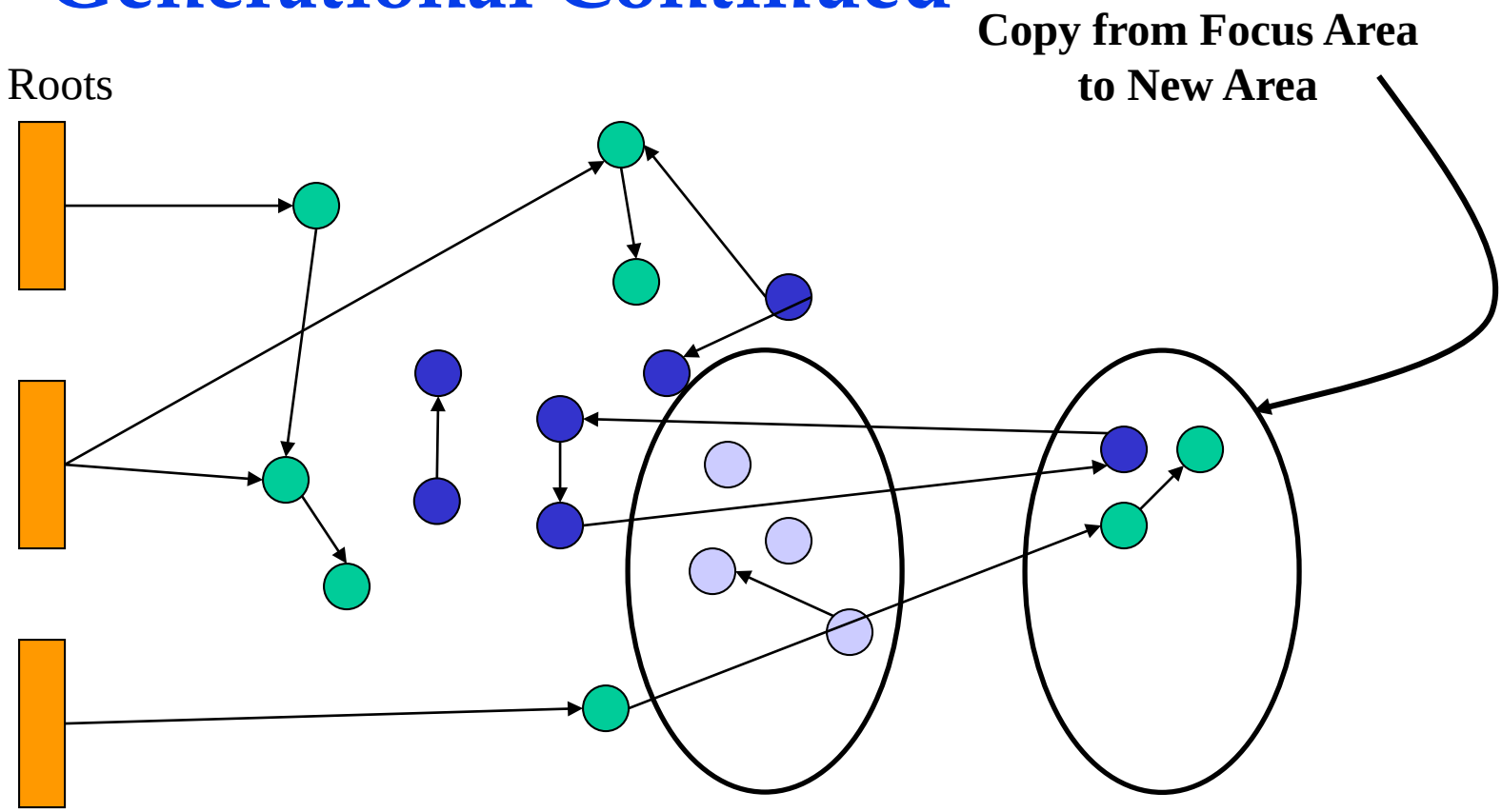
How Generational Collection works



Observation:
- very high infant mortality rate

Reachable Objects
Garbage

Generational Continued



Reachable Objects

Reclaimed

Garbage

Benefits:
- less copying
- lower average latency

分程序结构&
Lambda表达式的编译技术

SWIFT语言是一种分程序结构语言

- **Functions can be nested**. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html

为什么要使用DISPLAY区处理分程序结构对外层变量的引用？

```
func A(){
        var int localA ;
        func B(){
                ... C() ; ...
                localA = ....
        }
        func C(){
                ... B() ; ...
                localA = ...
        }
        ...B()...
}
```

- 如果不使用**DISPLAY**区，在**B()**或者**C()**中如何访问**localA?**

Lambda表达式(Closures)是分程序吗？

- Lambda表达式可以理解为一种“匿名”的内联函数
  - 不同语言的Lambda表达式的设计和实现不同，有的（有时）可以内联，有的需要动态调用
- C++为例

```
auto ptr = []() {cout << "hello" << endl; };        //lambda表达式
ptr();//调用函数

 //带参数的lambda表达式
auto fun = [](int x, int y)->int {cout << x + y << endl; return y;};
//  这里的->后面写的是返回值类型
auto z=fun(3, 4);
```

C++Lambda表达式变量作用域

- **[captures]** (params) mutable-> type{...}
  - 在 lambda 表达式引出操作符[ ]里的“**captures**”称为“捕获列表”，可以捕获表达式外部作用域的变量，在函数体内部直接使用。**这种语言设计在编译时可以规避DISPLAY区**
  - Java语言支持的Lambda表达式在访问外部变量时要求被访问变量为**final**类型，同样可以起到规避DISPLAY的作用
- 捕获列表里可以有多个捕获选项，以逗号分隔，使用了略微“新奇”的语法，规则如下
  - [ ] ：无捕获，函数体内不能访问任何外部变量
  - [=] ：以值（拷贝）的方式捕获所有外部变量，函数体内可以访问，但是不能修改。
  - [&] ：以引用的方式捕获所有外部变量，函数体内可以访问并修改（需要当心无效的引用）；
  - [var] ：以值（拷贝）的方式捕获某个外部变量，函数体可以访问但不能修改。
  - [&var] ：以引用的方式获取某个外部变量，函数体可以访问并修改
  - [this] ：捕获this指针，可以访问类的成员变量和函数，
  - [=，&var] ：引用捕获变量var，其他外部变量使用值捕获。
  - [&，var]：只捕获变量var，其他外部变量使用引用捕获。

分程序结构和Lambda表达式

- lambda表达式是分程序结构在软件工程和编程方法意义上的进阶版，但是**lambda表达式不完全等同于传统分程序结构**
  - 它们对外层变量的引用方式上有较大不同，编译方法也不一样

- lambda表达式在不同语言中的设计不同，这也会导致编译方法的差异较大，但一般倾向于低负载的调用机制
  - **规避DISPLAY**
  - **内联**

面向对象语言的编译技术

类的封装、继承、多态

对象的封装

- 隐藏对象的属性和实现细节，仅对外提供公共访问方式
- 编译器对对象的数据管理机制

对象的继承

- 成员和方法都可以继承

NSView (Leopard)	
0	Class isa
4	NSRect bounds
20	NSView *superview
24	NSColor *bgColor

PetShopView
Class isa
NSRect bounds
NSView *superview
NSColor *bgColor
NSArray *kittens
NSArray *puppies

对象成员

NSView (Leopard) Vtable	
0	method0
4	method1
8	method2
12	method3

PetShopView VTable
method0
method1_OV
method2
method3_OV
method4
method5

方法虚表

多态的实现

- 基于虚表的实现

NSView (Leopard) Vtable	
0	method0
4	method1
8	method2
12	method3

PetShopView VTable
method0
method1_OV
method2
method3_OV
method4
method5

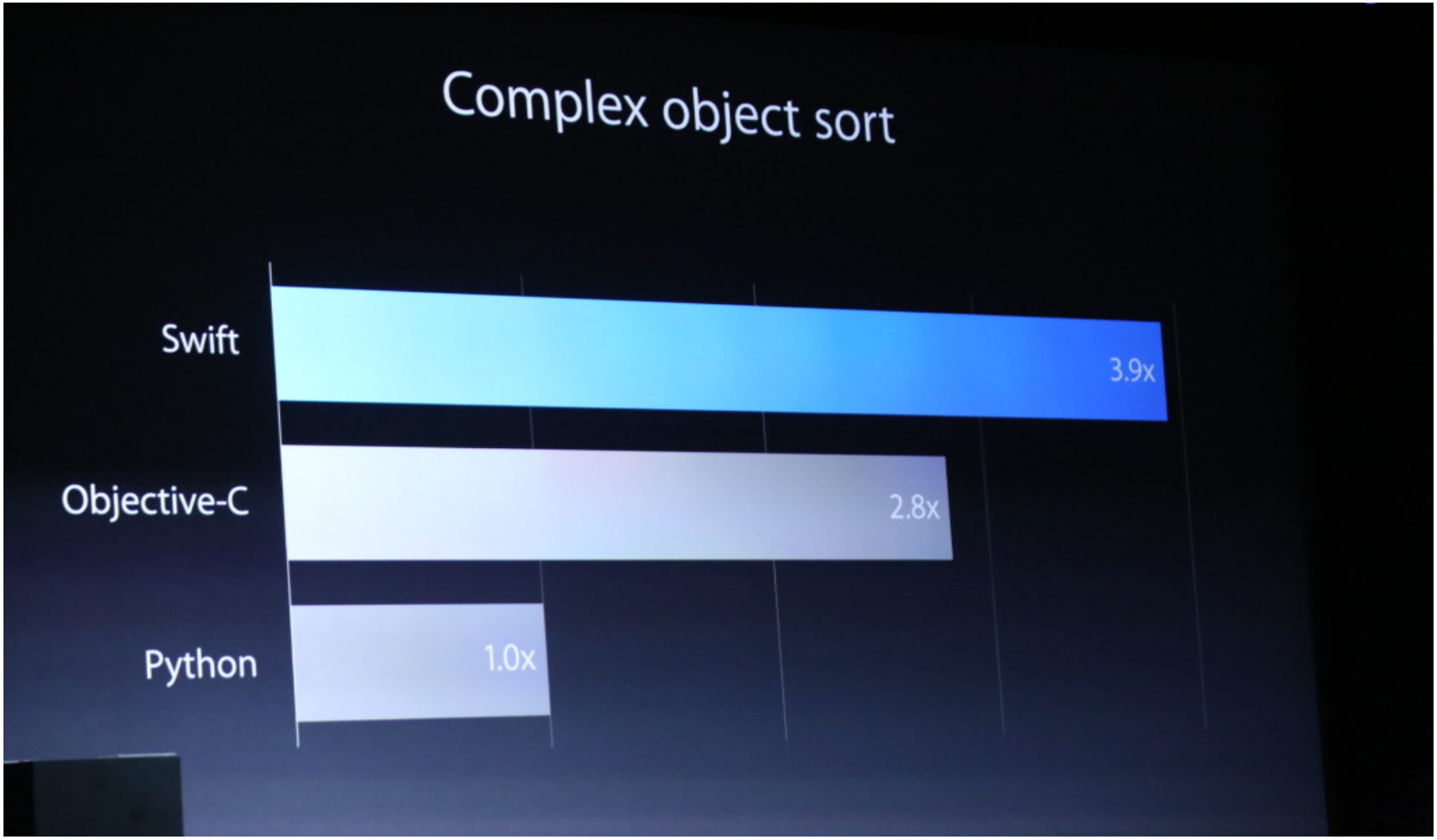
虚表

```
//call obj.method3()
mov ecx, [ebp+8] //ecx = obj
mov eax, [ecx]   //eax = obj.vtable
call [eax+12]    //call
obj.vtable[3]
```

Objective-C Method Dispatching

- Clang编译器将所有函数调用转为对 *objc_msgSend()*的调用
  - passing an object, method selector and parameters as arguments
  - 例, *[object message: param]* 将变为: *objc_msgSend(object, @selector(message:), param, nil)*
  - ***objc_msgSend()* 处理Object-C中所有method dispatching**
    - 汇编实现、开源

```
id objc_msgSend ( id obj, SEL op, ... ){
    Class c = object_getClass(obj);
    IMP imp = CacheLookup(c, op);
    if (!imp) {
        imp = class_getMethodImplementation(c, op);
    }
    jump imp(obj, op, ...);
}
```


Complex object sort
Swift
3.9x
Objective-C
2.8x
Python
1.0x