# 运行环境

## 问题的提出：

程序最终要在机器上运行。

如何在内存中运行；

需要什么样的支持。


## 运行环境

目标计算机的寄存器及存储器的结构，用来管理存储器并保存执行过程所需的信息。

函数

- 不同的源语言结构，所需的运行环境和支持不同。本章仅以最简单的、基于过程的、顺序执行的程序为前提讨论，即源程序的基本结构是顺序执行的过程，过程与过程之间仅通过子程序调用的方式进行控制流的转移。

# 第7章 运行环境

7.1 程序运行时的存储组织

7.2 静态运行时环境与存储分配

7.3 基于栈的运行时环境的动态存储分配

7.4 基于堆的运行时环境的动态存储分配

运行时为名字X分配存储空间S，这一过程称为**绑定** (binding)。

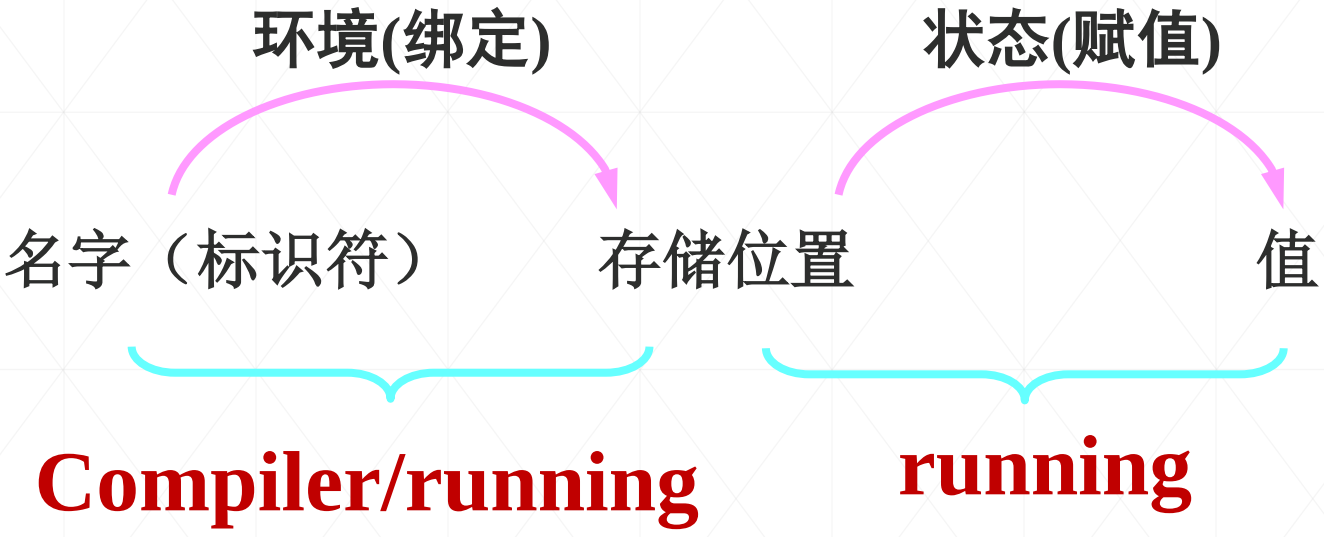
**X是一个对象**：

* 既可以是数据对象，如变量，与之结合的是一个存储单元；

* 也可以是操作对象，如过程。与之结合的是可执行的代码。

讨论仅限于X是一个数据对象。

## 变量名字与值的两步映射




## 常量名字的映射

例： 若有变量声明 float x ；

常量声明 const pi=3.14;

则赋值句中变量和常量的映射关系：


x= 3.14的映射

pi=3.14的映射

常量不存在状态变化，不能被赋值

## 注意：

* **环境、状态的区别：**

赋值改变状态，但不改变环境。

* **环境与语言特性相关：**

FORTRAN完全静态环境；

C，C++，PASCAL，Ada基于栈的环境；

LISP完全动态环境。

## 名字的两种属性：

- **静态：** 如果一个名字的属性通过说明语句或隐或显规则而定义，则称这种属性是“静态”确定的。

编译处理

- **动态：** 如果名字的属性只有在程序运行时才能知道，则称这种属性为“动态”确定的。

运行处理

## 静态与动态属性举例

| 静 态 | 动 态 |
| --- | --- |
| 过程的定义 | 过程的活动 |
| 名字的声明 | 名字的绑定 |
| 声明的作用域 | 绑定的生存期 |

# 名字与存储空间的不同绑定时间

存储分配策略
- 静态分配策略
- 动态分配策略
  - 栈式方案
  - 堆式方案

## 静态存储分配

在编译时能够确定目标程序运行时所需的全部数据空间的大小，即在编译时就可以将程序中的名字关联到存储单元，确定其存储位置，这种分配策略称为静态存储分配。

## 动态存储分配

在编译时不能确定目标程序运行时所需的全部数据空间的大小，而是在目标程序运行时动态确定的。

## 栈式动态存储分配

在内存中开辟一个栈区，按栈的特性进行存储分配。程序运行时，每当进入一个函数或过程，该函数所需的存储空间动态地分配于栈顶，函数返回时，释放所占用的空间。

## 堆式动态存储分配

在内存中开辟一个称为堆的存储区，程序运行每当需要（申请）时就按照某种分配原则在堆的自由区(可占用区)中，分配能满足其需要的存储空间给它，使用后需要释放操作，再将不再占用的存储空间归还给堆的自由区。

运行时存储分配的一个原则是，尽可能对数据对象进行静态分配 。

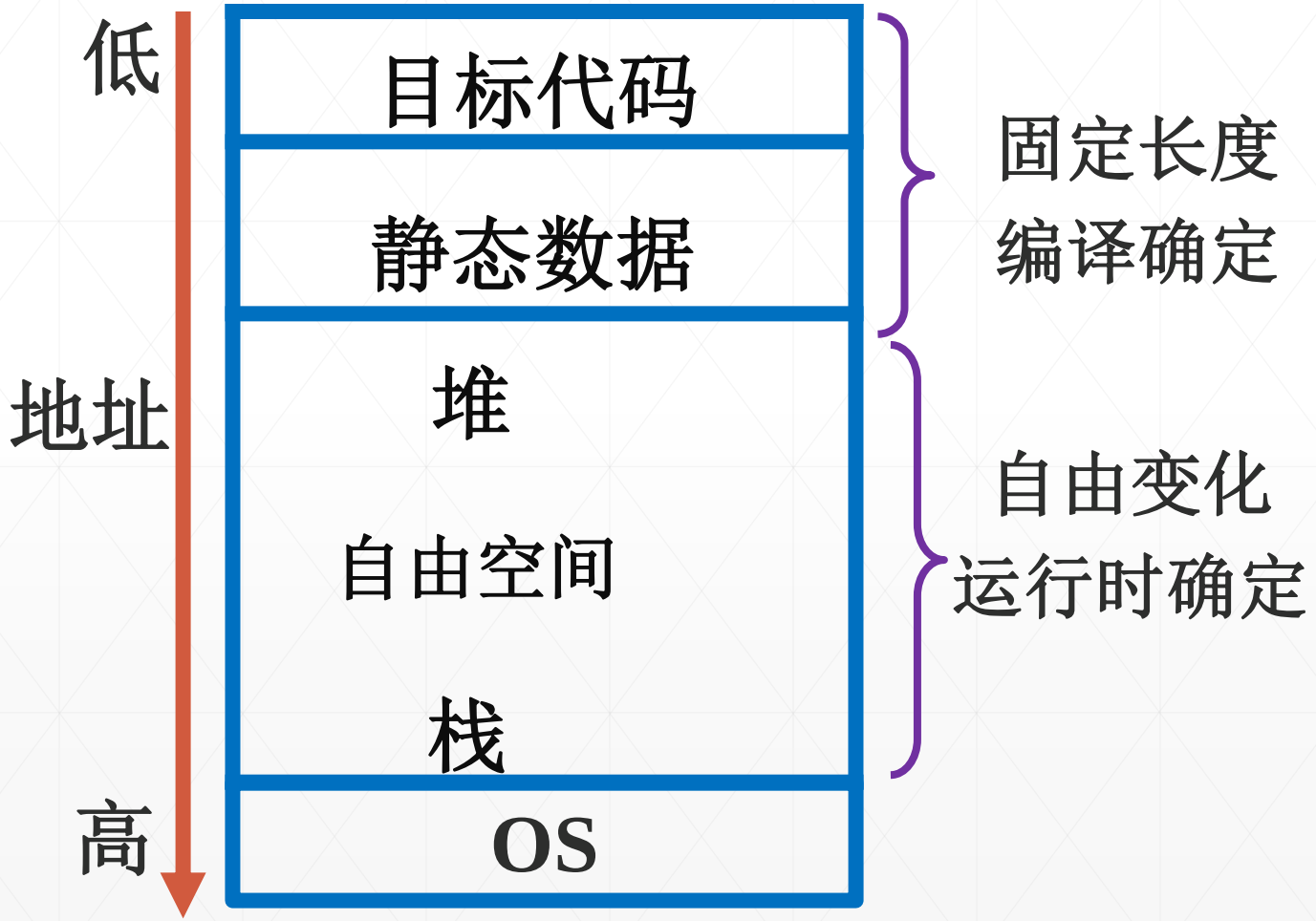
## 存储区保存的对象

(1) 生成的目标代码。(代码区)

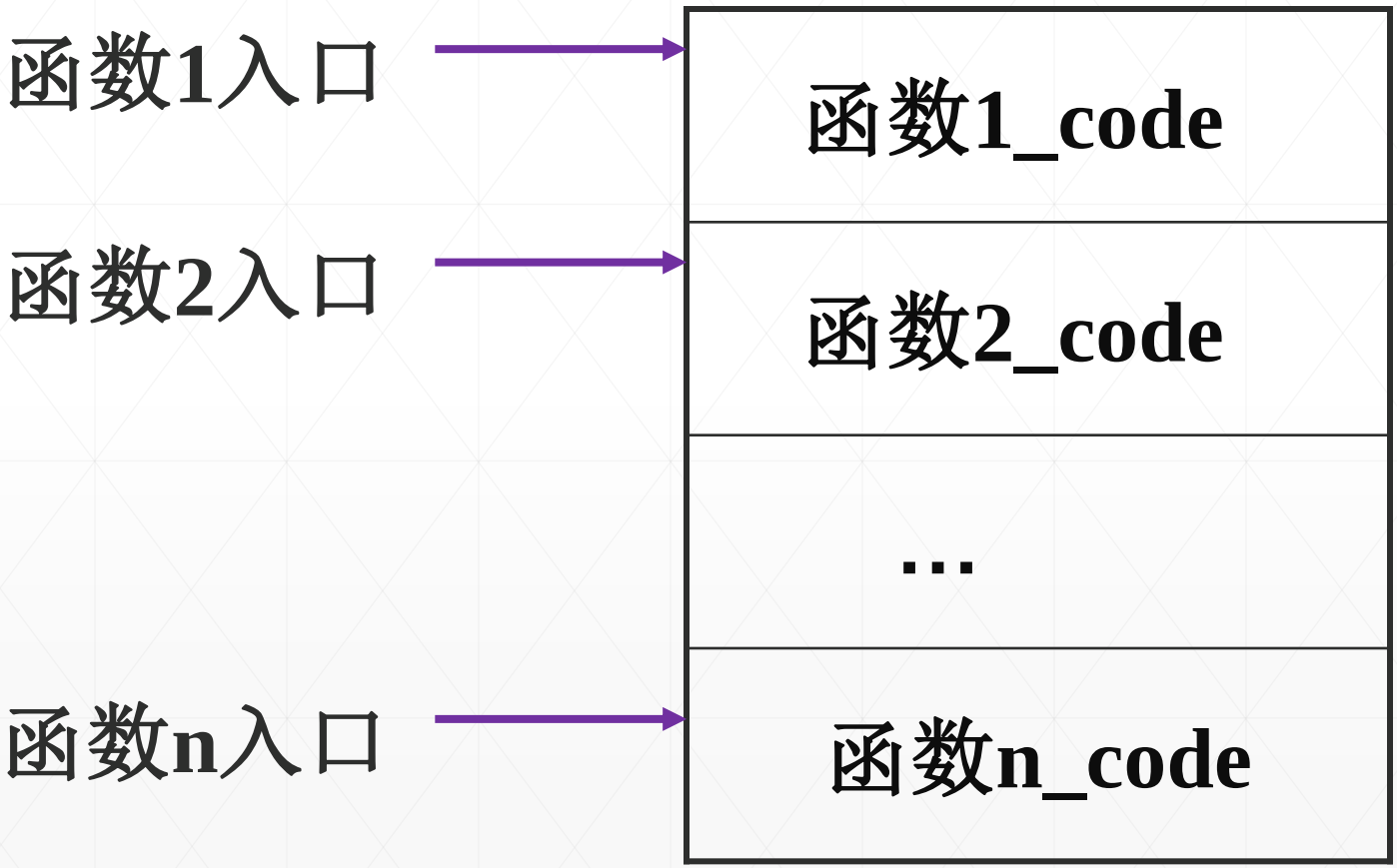
(2) 目标代码运行时的数据空间。包括用户定义的各种类型的数据对象，作为保存中间结果和传递参数的临时工作单元，组织输入、输出所需的缓冲区等。

(3) 记录过程活动的信息。

- 运行时内存分成代码区和数据区的典型划分



## 代码区



- 存储分配的重要单元—过程的活动记录AR（Procedure Activation Record）

一块连续的存储区，用来存放过程或函数的一次调用执行所需要的信息。

活动记录的组织依赖于目标机体系结构，被编译的语言特性等。

## 与过程运行相关的概念：活动、生存期

**过程**的每一次运行称为一次活动(activation)。

**活动**是一个动态的概念，它有有限的生存期(life time)。

**活动的生存期**是指从进入活动的第一条指令执行到离开此活动前的最后一条指令执行的这段时间，其中包括调用其它过程时其它活动的生存期。

## 函数调用的语义处理：

① 检查所调用的过程或函数是否定义；与所定义的过程或函数的类型、实参与形参的数量及类型是否一致；

② 给被调过程或函数分配活动记录所需的存储空间；

③ 计算并传送实参；

④ 加载调用结果和返回地址，恢复主调用过程或函数的继续执行；

⑤ 转向相应的过程或函数。

## 活动记录基本内容


自变量实参空间
局部数据空间
局部临时变量空间
机器状态信息
返回地址 RA
存取链 SL
控制链 DL
代码地址
活动记录地址

# 第7章 运行环境

7.1 程序运行时的存储组织

7.2 静态运行时环境与存储分配

7.3 基于栈的运行时环境的动态存储分配

7.3 基于堆的运行时环境的动态存储分配

## 静态存储分配

在编译时能够确定目标程序运行时所需的全部数据空间的大小，即在编译时就可以将程序中的名字关联到存储单元，确定其存储位置，这种分配策略称为静态存储分配。

**特点：**

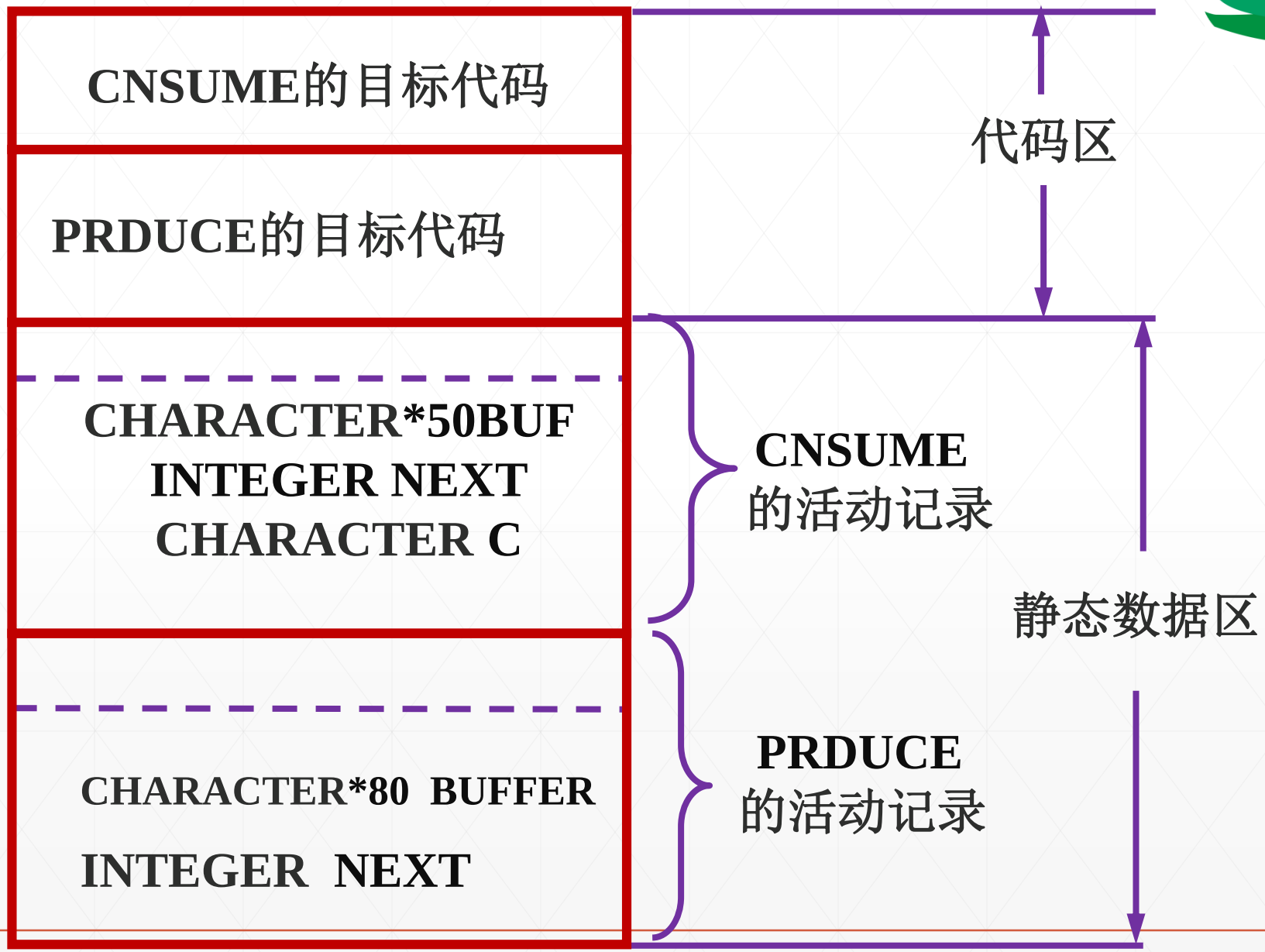
名字在程序编译时与存储空间结合，

每次过程活动时，它的名字映射到同一存储单元。程序运行时不再有对存储空间的分配。

# 例7.1 给出FORTRAN程序如下。

```
(1)        PROGRAM CNSUME
(2)          CHARACTER *50 BUF
(3)           INTEGER   NEXT
(4)           CHARACTER C, PRDUCE
(5)           DATA NEXT / 1 / ,BUF/ ' ' /
(6)        6  C=PRDUCE( )
(7)           BUF(NEXT:NEXT)=C
(8)           NEXT=NEXT+1
(9)           IF(C.NE.' ')  GOTO 6
(10)          WRITE(* , '(A)' ) BUF
(11)        END
```

```
(12)      CHARACTER FUNCTION PRDUCE()
(13)        CHARACTER   *80  BUFFER
(14)        INTEGER NEXT
(15)        SAVE BUFFER , NEXT
(16)        DATA NEXT /81/
(17)        IF (NEXT.GT.80) THEN
(18)        READ ( * , '(A)' ) BUFFER
(19)        NEXT=1
(20)        END IF
(21)        PRDUCE=BUFFER(NEXT:NEXT)
(22)        NEXT=NEXT+1
(23)      END
```


CNSUME的目标代码
PRDUCE的目标代码
CHARACTER*50BUF
INTEGER NEXT
CHARACTER C
CHARACTER*80 BUFFER
INTEGER NEXT
代码区
CNSUME
的活动记录
PRDUCE
的活动记录
静态数据区

## 静态分配对语言的限制

(1) 数据对象的长度和它在内存中的位置的限制必须在编译时知道。

(2) 不允许递归过程，因为一个过程的所有活动使用同样的局部名字绑定。

(3) 数据结构不能动态建立，因为没有运行时的存储分配机制。

# 第7章 运行环境

7.1 程序运行时的存储组织

7.2 静态运行时环境与存储分配

7.3 基于栈的运行时环境的动态存储分配

7.3 基于堆的运行时环境的动态存储分配

## 基于栈的运行时环境的动态存储分配

将整个程序运行时使用的存储空间都安排在一个栈里。每当调用一个函数时，它所需的数据空间就分配在栈顶，每当过程结束时就释放这部分空间。

函数所需的数据空间：

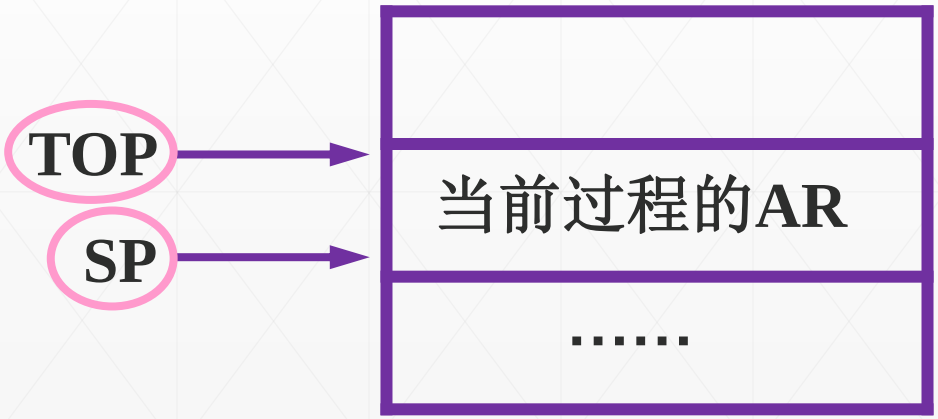
- 生存期在本过程的本次活动中的数据对象 (局部变量、临时变量…)
- 管理过程活动的记录信息 (过程调用中断时当前机器的状态信息)

## 活动记录的栈

运行时存储活动记录的空间，称为运行时栈或调用栈。

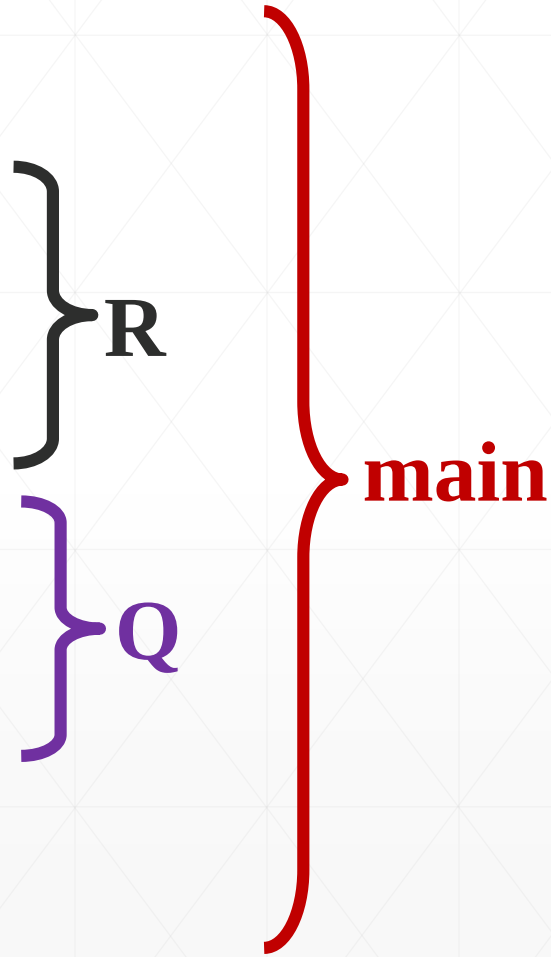
随着程序执行时发生的调用链生长或缩小。

每个过程每次在调用栈上可以有不同的活动记录，每个都代表一个不同的调用。

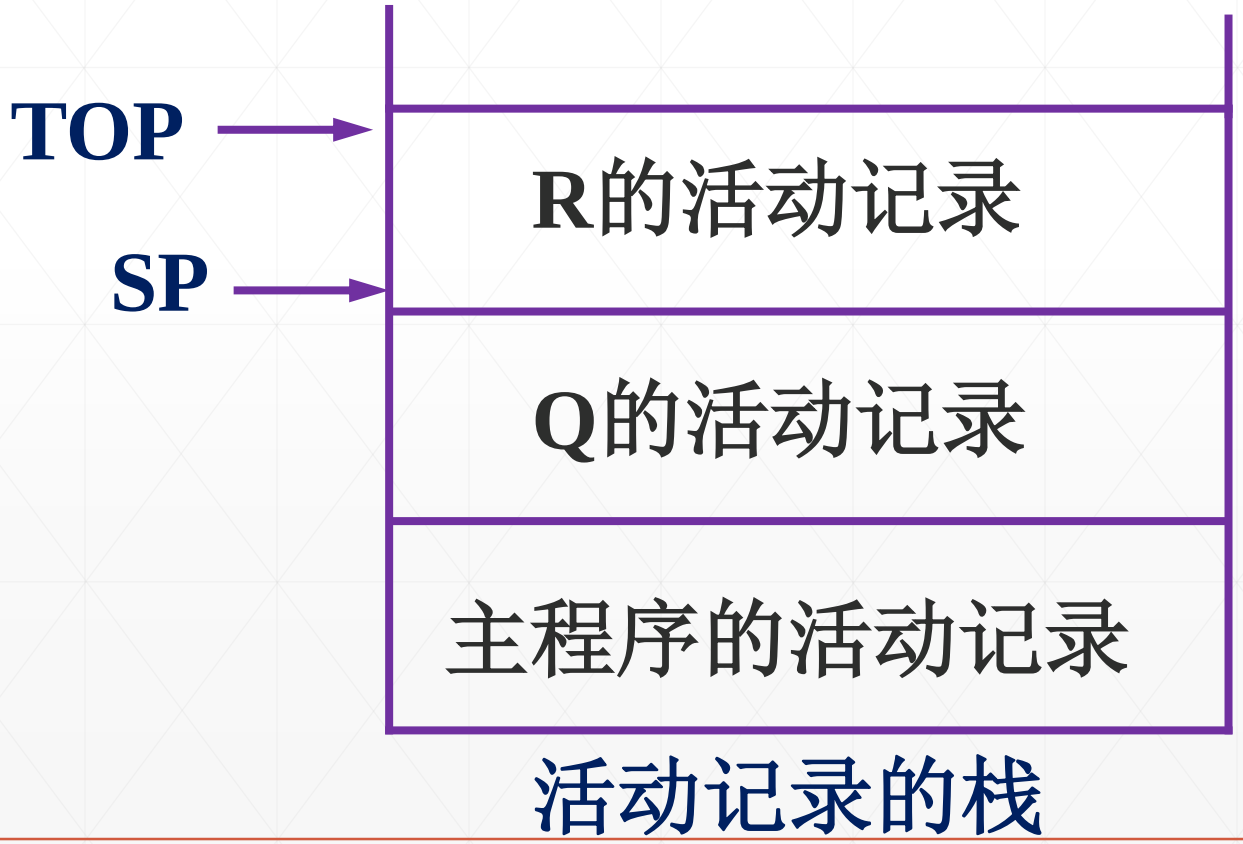


例 设有如下程序(允许嵌套定义过程）

```
main
    全局变量的说明
        proc   R
          ……
        end  R;
        proc  Q
          ……
        end  Q;
    主程序执行语句
end  main
```

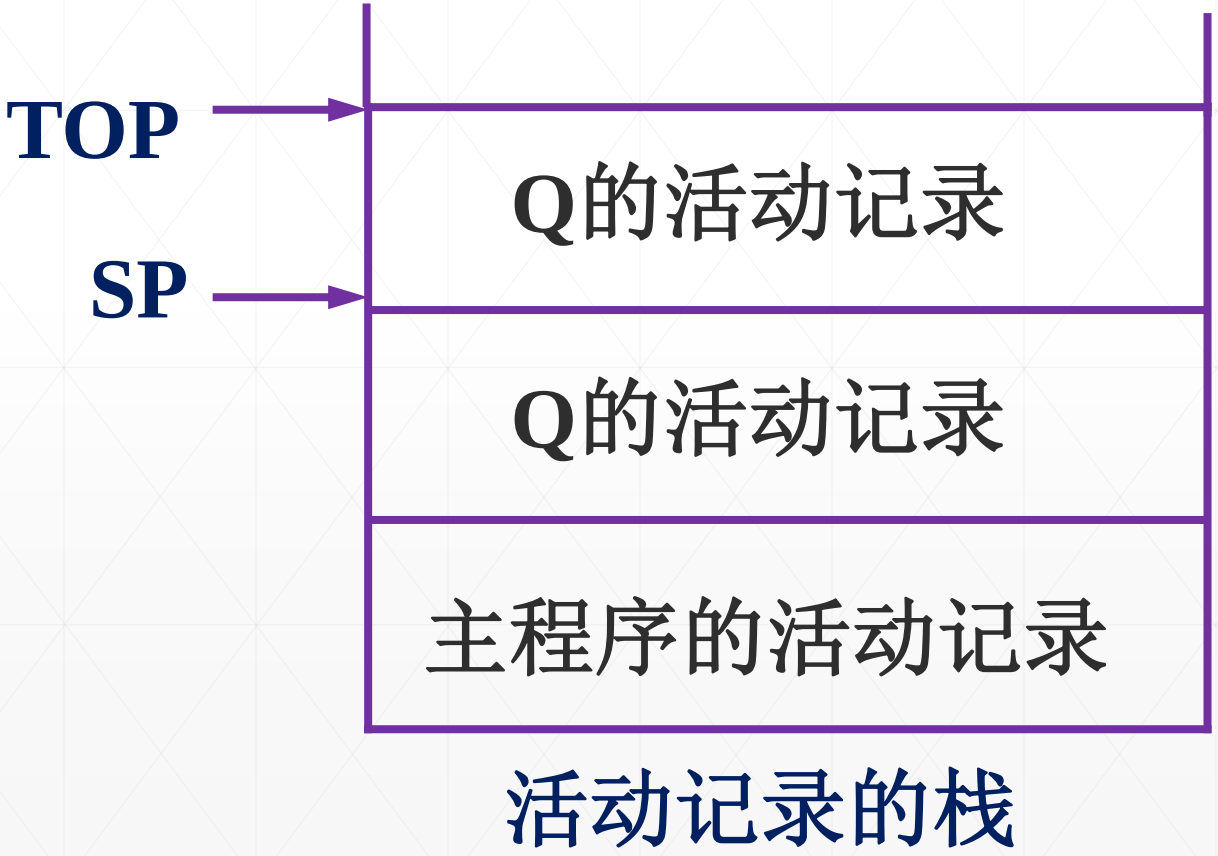


程序运行顺序： Main $\xrightarrow{\text{调用}}$ Q $\xrightarrow{\text{调用}}$ R




活动记录的栈

程序运行顺序： Main $\xrightarrow{\text{调用}}$ Q $\xrightarrow{\text{调用}}$ Q



## 嵌套定义的过程可使用的数据：

1、本过程定义的数据；

2、外层过程中定义的数据。

## 引入问题：

外层过程中定义的数据怎样找到？

- **关键技术：**

  解决对非局部量的引用（存取）。

  设法跟踪每个外层过程的最新活动记录AR的位置。

- **跟踪办法：**

  1. 用静态(访问、存取）链。
  2. 用DISPLAY表。

AR（活动记录）

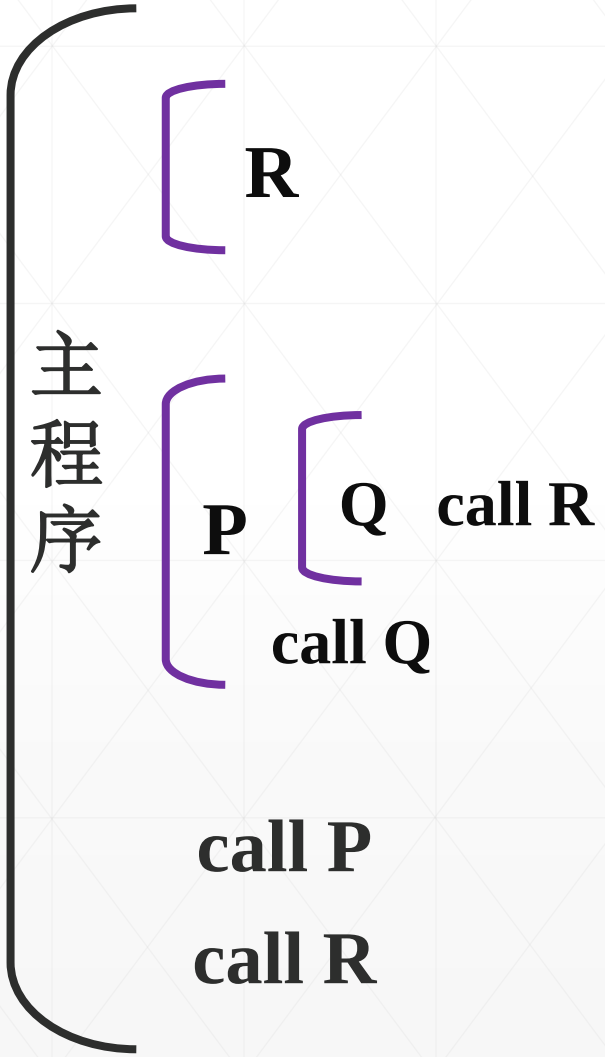

**SL：存取链。**指向定义该过程的直接外层过程运行时最新活动记录的基地址。

**DL：控制链。** 指向调用该过程前正在运行过程的活动记录基地址。

例

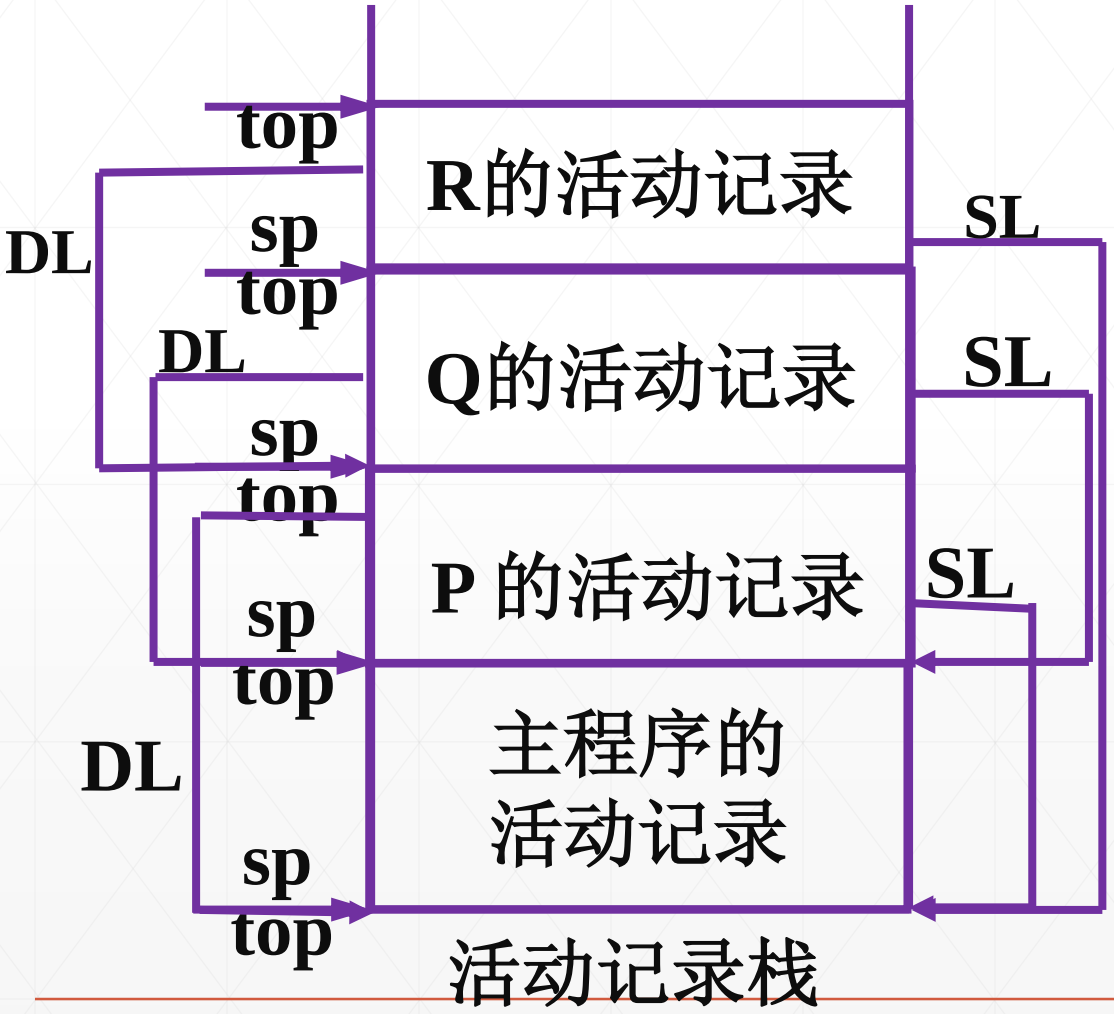
```
program  main( i,0) ;
        ……
    proc  R(c,d) ;
        ……
    end  /*R*/
   proc     P (a);
     ……
        proc   Q (b);
            ……
          R(x,y);
        end    /* Q*/
     ……
      Q(z);
    end   /* P*/
     ……
   P(W);
     ……
   R(U,V);
     ……
end   /* main*/
```

## 程序结构图



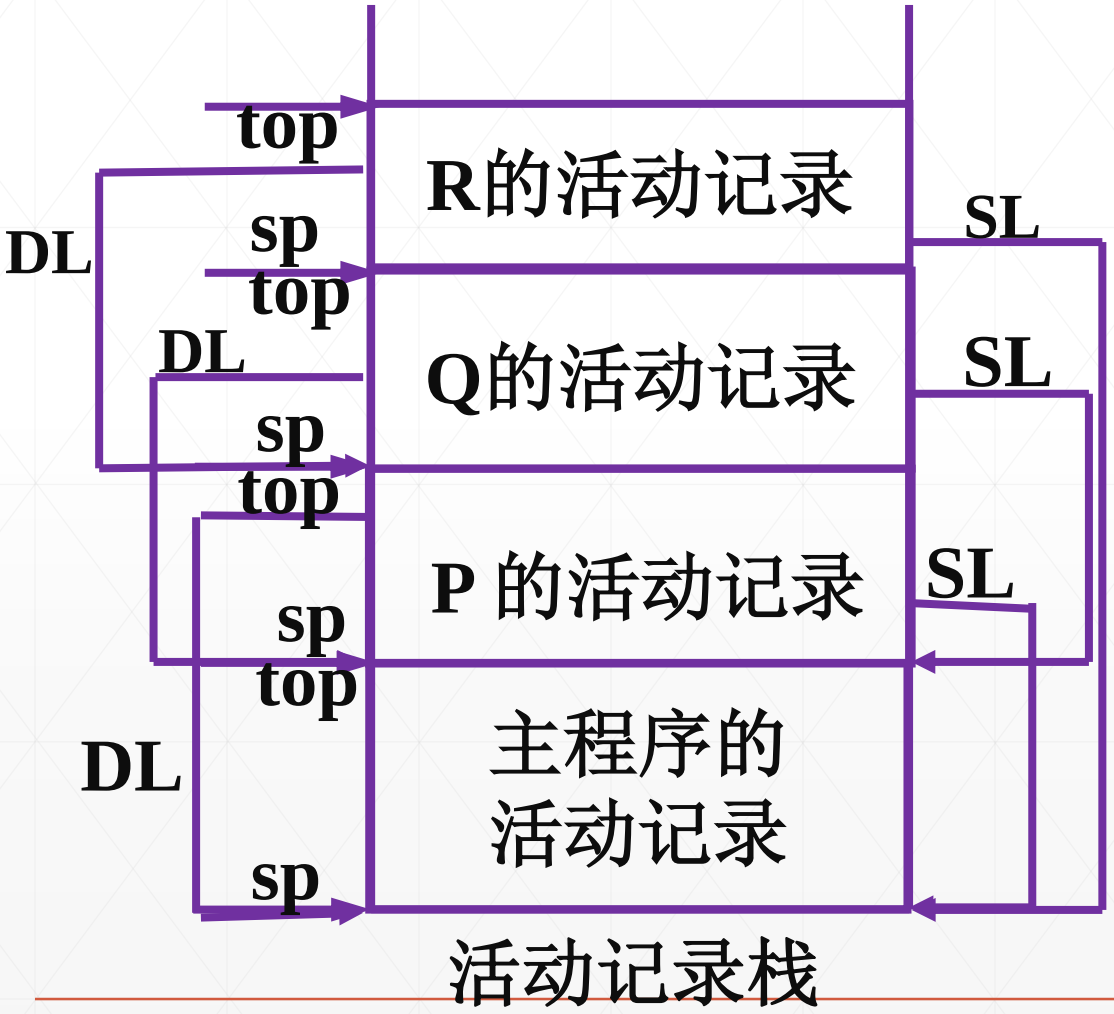
# 用SL(存取链）的方案

主程序 ➔ P ➔ Q ➔ R



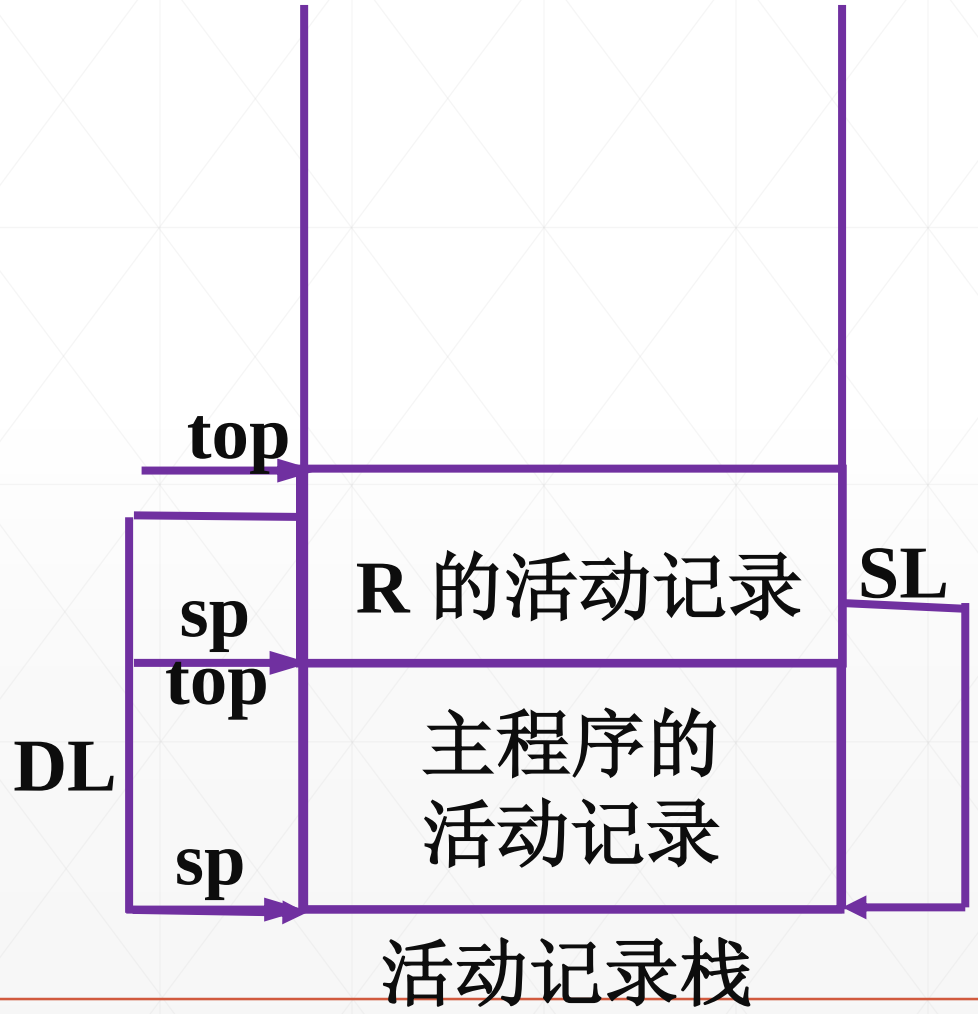

# 用SL(存取链）的方案

主程序 ➔ P ➔ Q ➔ R



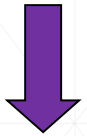

# 用SL(存取链）的方案

主程序 ➔ R

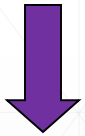



例 设有C程序

gcd(15,10)



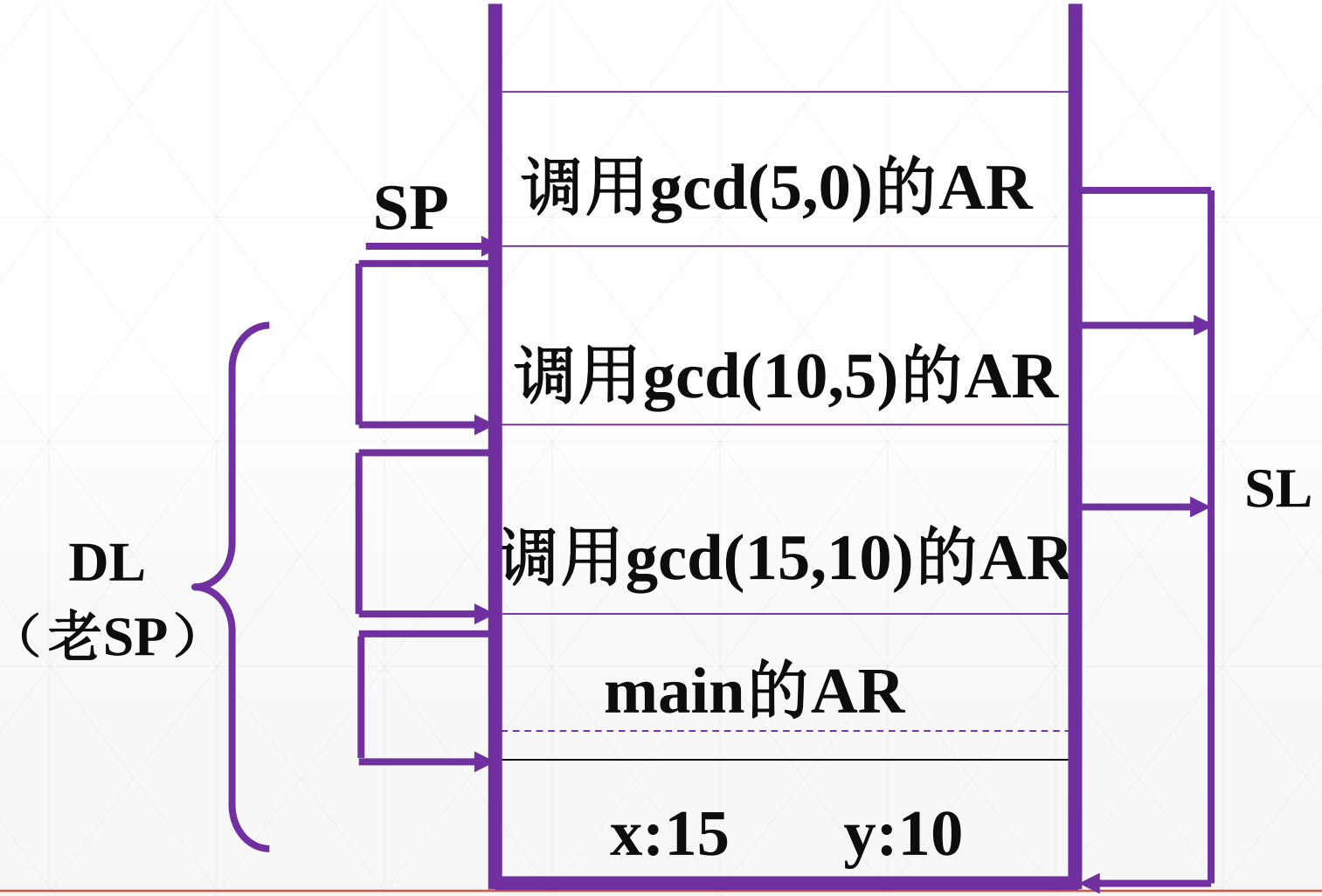
gcd(10,5)



gcd(5,0)

```
#include <stdio.h>
int x,y ;
int gcd(int u,int v)
{  if (v==0) return u;
    else return gcd(v, u%v);
}
main()
{ scanf("%d%d",&x,&y);
   printf("%d\n",gcd(x,y));
   return;
}
```

C语言中无过程（函数）的嵌套定义，所以数据链简单，同指向全局数据。



## 解决对非局部量的引用：用Display表

### Display表 — 嵌套层次显示表

主程序设为第0层；

主程序中定义的过程为第1层；

第i层过程中定义的过程为i＋1层；

假设当前激活过程的层次为K，它的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。

# 构造display区的规则：

假定从i层过程进入第j层过程，则有：

① 如果j=i+1 (即调用当前块局部说明的过程块)，则复制第i层的display;转③。

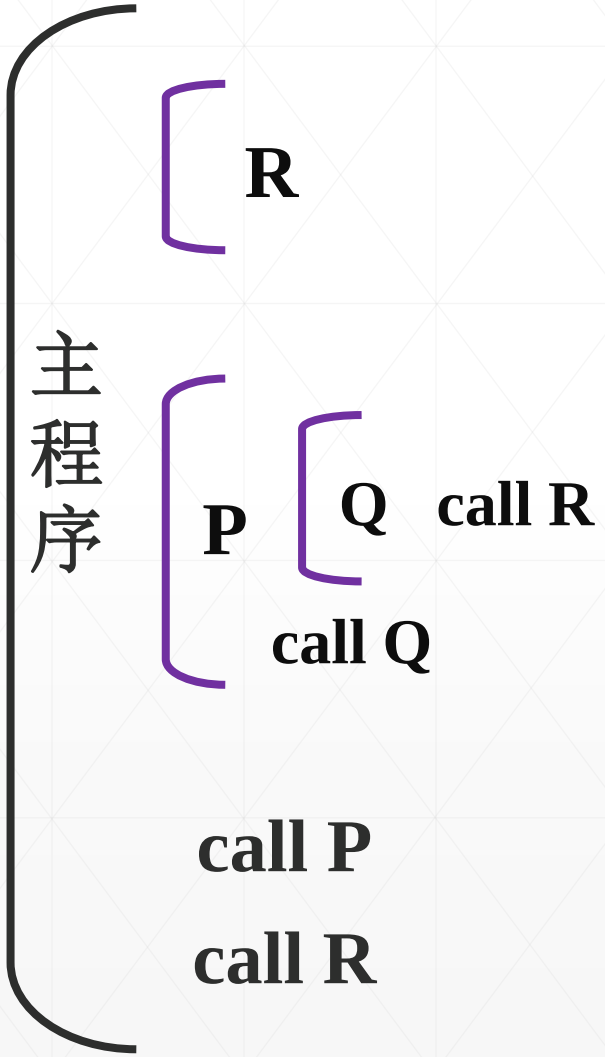
② 如果 j≤i (即调用对当前过程来说属于全局说明的过程)，则复制来自i层过程活动记录中的display区前面j个记录；转③。

③ 再增加一个指向j层过程活动记录的指针；

例

```
program  main( i,0) ;
        ……
    proc  R(c,d) ;
        ……
    end  /*R*/
   proc     P (a);
     ……
        proc   Q (b);
            ……
          R(x,y);
        end    /* Q*/
     ……
      Q(z);
    end   /* P*/
     ……
   P(W);
     ……
   R(U,V);
     ……
end   /* main*/
```

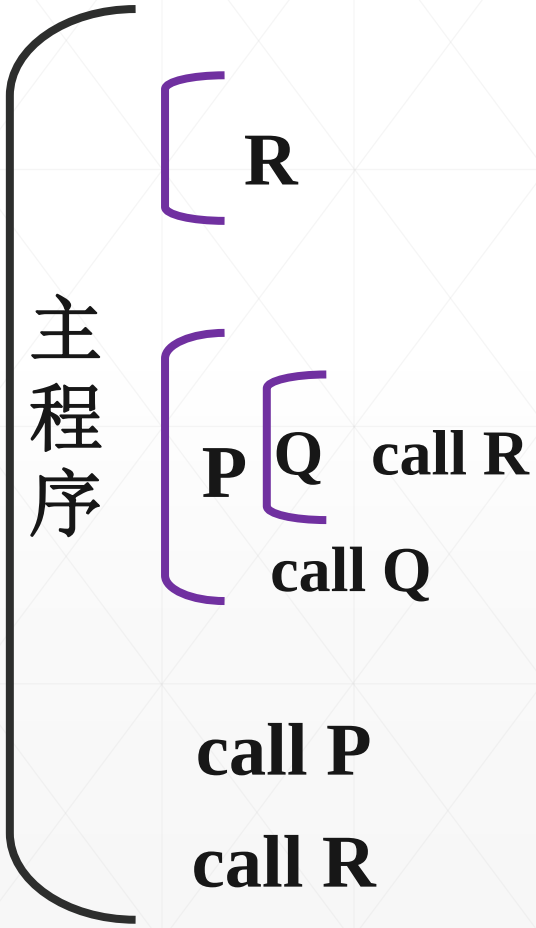
程序结构图



# 用Display表的方案

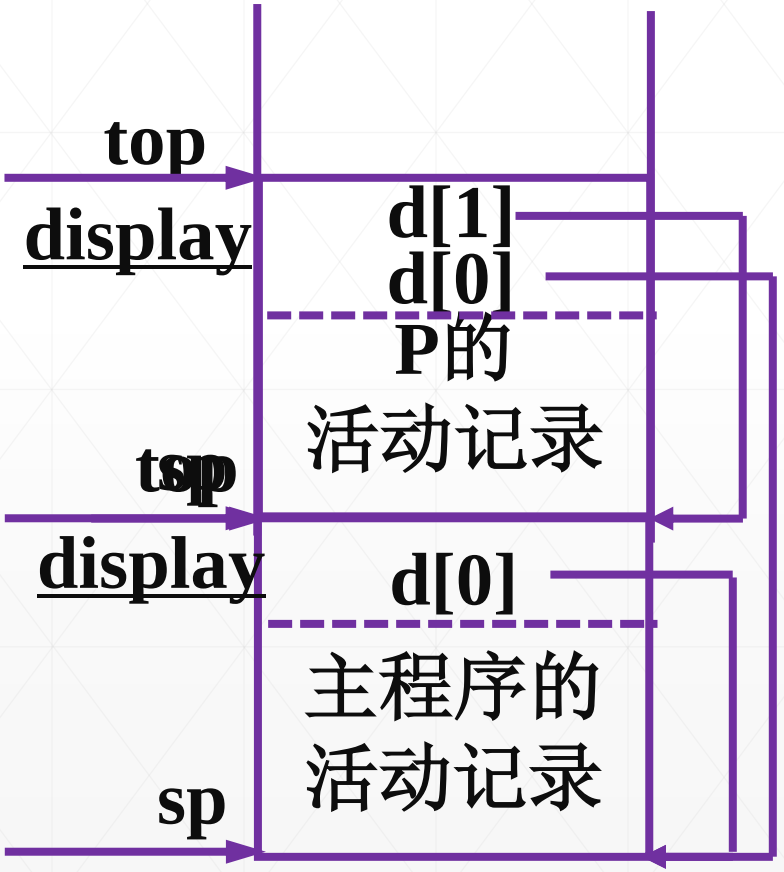
主程序 (1) ➔ P(2) ➔ Q(3) ➔ R(4) ➔ …


（1）

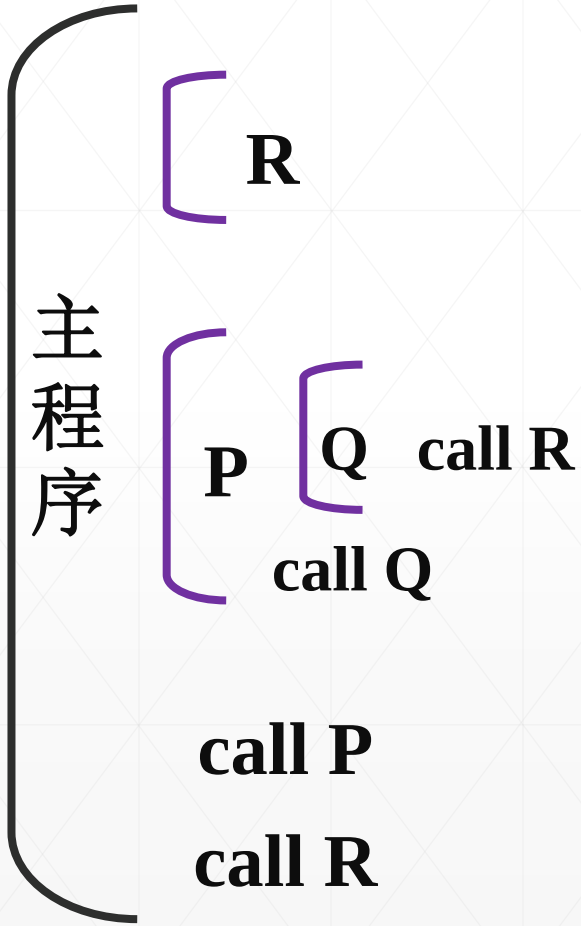


# 用Display表的方案

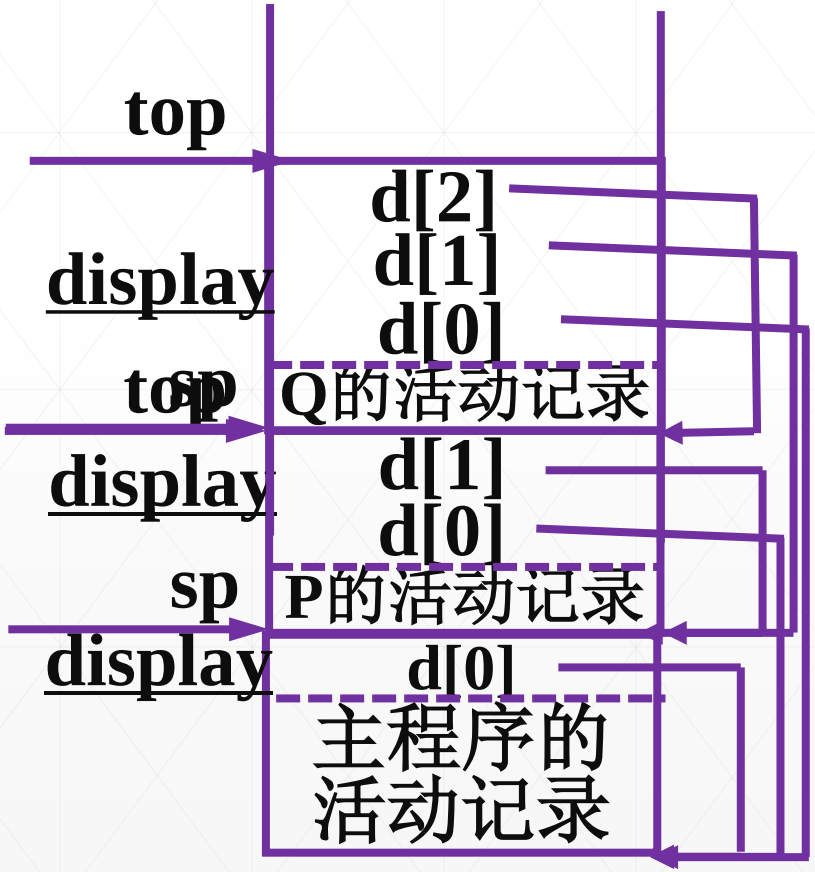
主程序 (1) ➔ P(2) ➔ Q(3) ➔ R(4) ➔ …




（2）

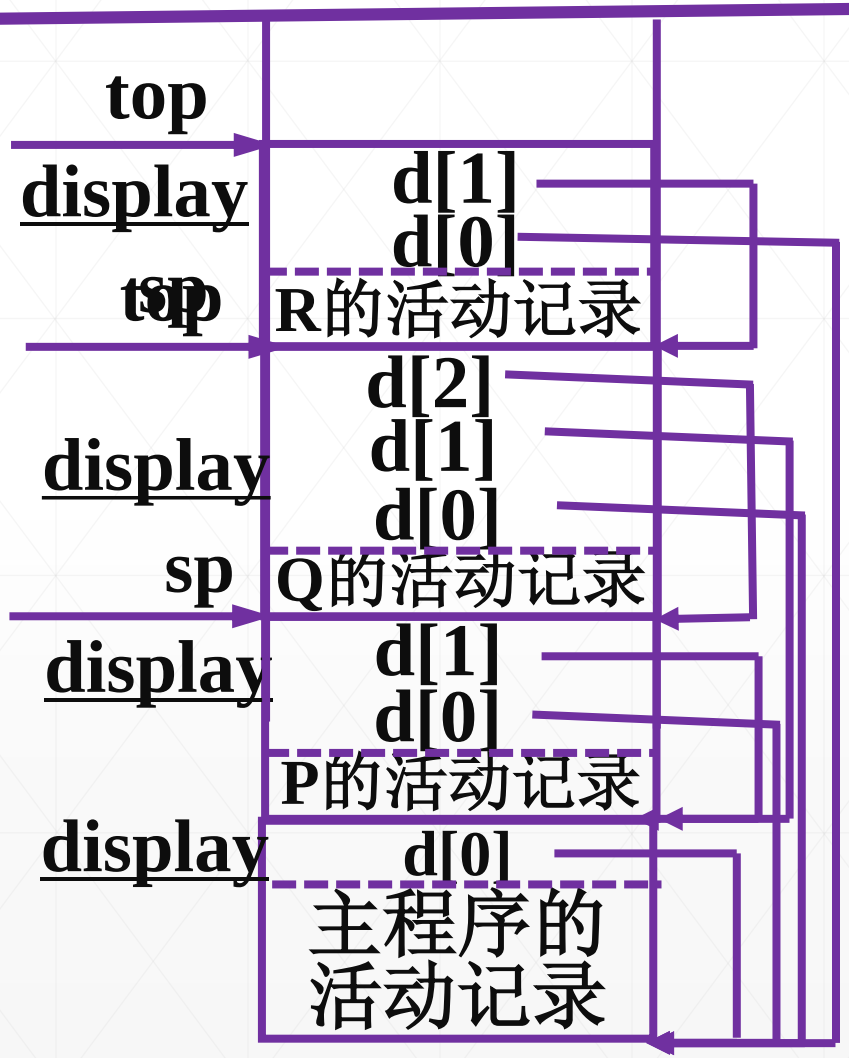


# 主程序 (1) ➔ P(2) ➔ Q(3) ➔ R(4)




（3）

主程序 (1) ➔ P(2) ➔ Q(3) ➔ R(4)





（4）

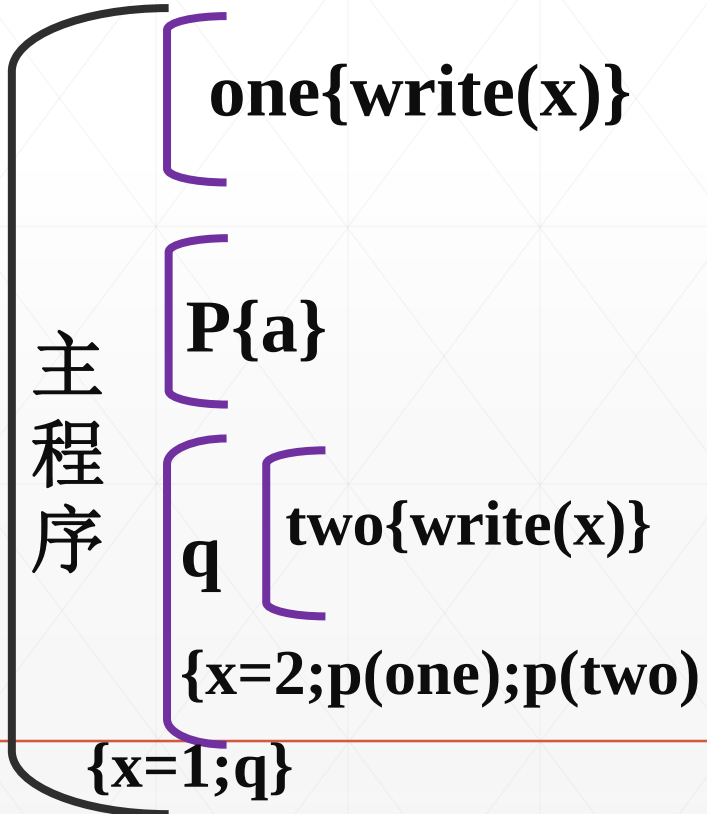
**P219:7-7**为以下的Pascal程序画出活动记录的栈。

(1)对p的第一次调用中对a的调用之后。

(2)在对p的第二次调用中对a的调用之后。

(3)程序打印出什么？为什么？

程序结构图




```
PROGRAM closureEx (Output);
  VAR x:Integer;
PROCEDURE one;
BEGIN
   Writeln(x);
END;
PROCEDURE p(PROCEDURE a);
BEGIN
  a;
END;
PROCEDURE q;
VAR x:Integer;
    PROCEDURE two;
    BEGIN
      Writeln(x);
    END;
BEGIN
  x : = 2;
  p (one);
  p (two);
END;  (* q *)
BEGIN  (* main *)
  x : = 1;
  q;
END.
```

# P219:7-7为以下的Pascal程序画

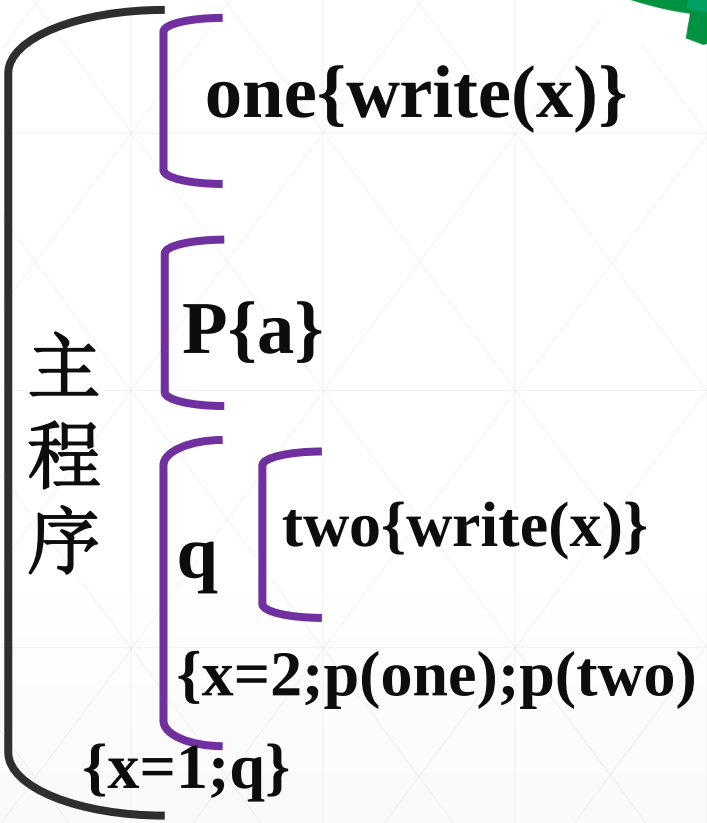
出活动记录的栈。

(1)对p的第一次调用中对a的调用之后。

(2)在对p的第二次调用中对a的调用之后。

(3)程序打印出什么？为什么？


## 程序结构图



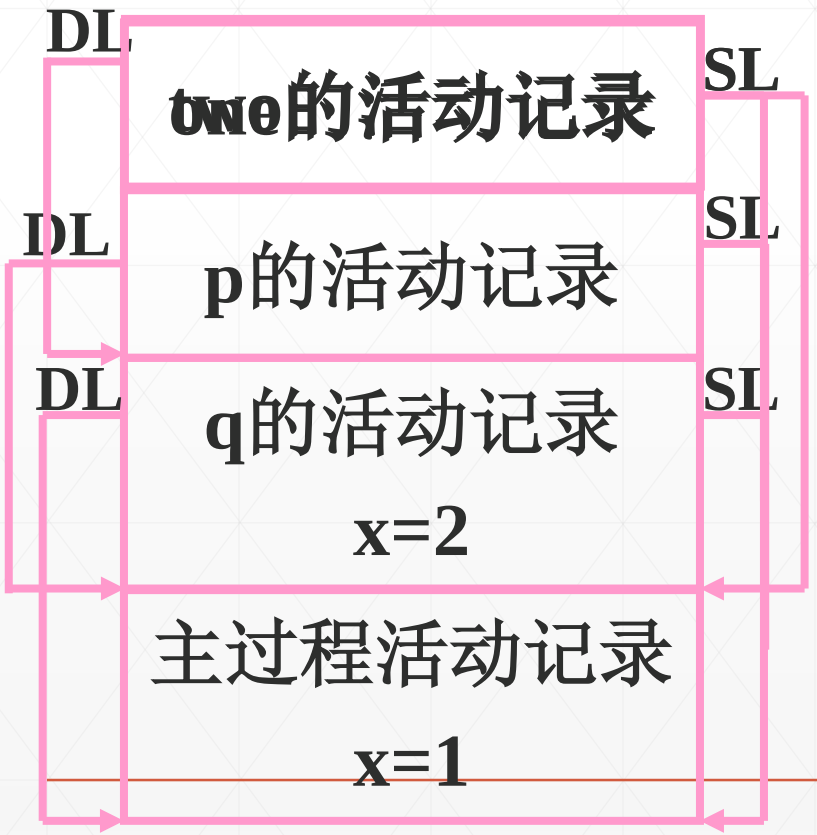
# P219:7-7为以下的Pascal程序画

出活动记录的栈。

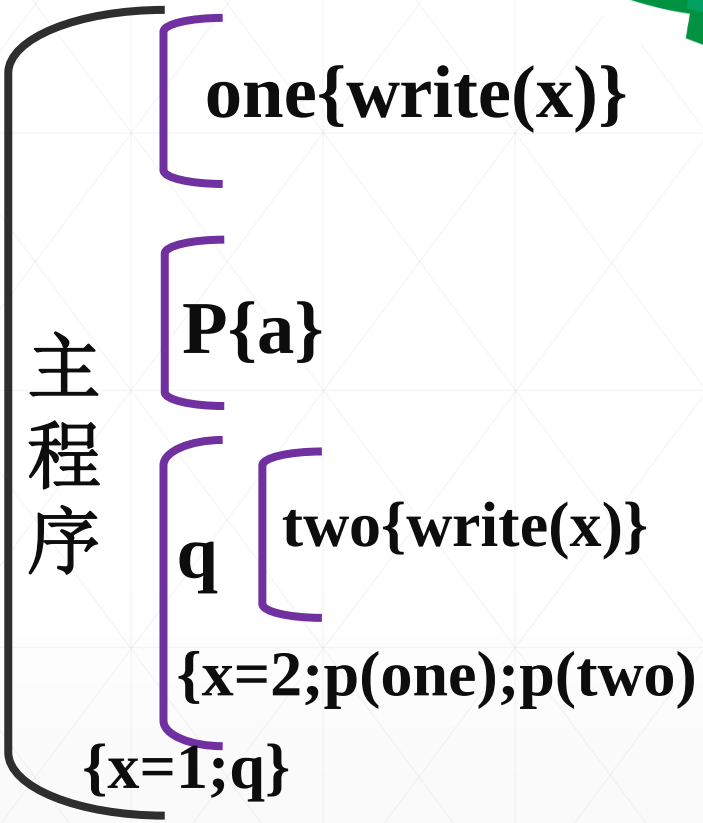
(1)对p的第一次调用中对a的调用之后。

(2)在对p的第二次调用中对a的调用之后。

(3)程序打印出什么？为什么？




## 程序结构图



# 第7章 运行环境

7.1 程序运行时的存储组织

7.2 静态运行时环境与存储分配

7.3 基于栈的运行时环境的动态存储分配

7.4 基于堆的运行时环境的动态存储分配

# 堆/Heap

存储数据对象特点：存活期比较自由

优点：按需灵活分配，

缺点：管理起来非常复杂

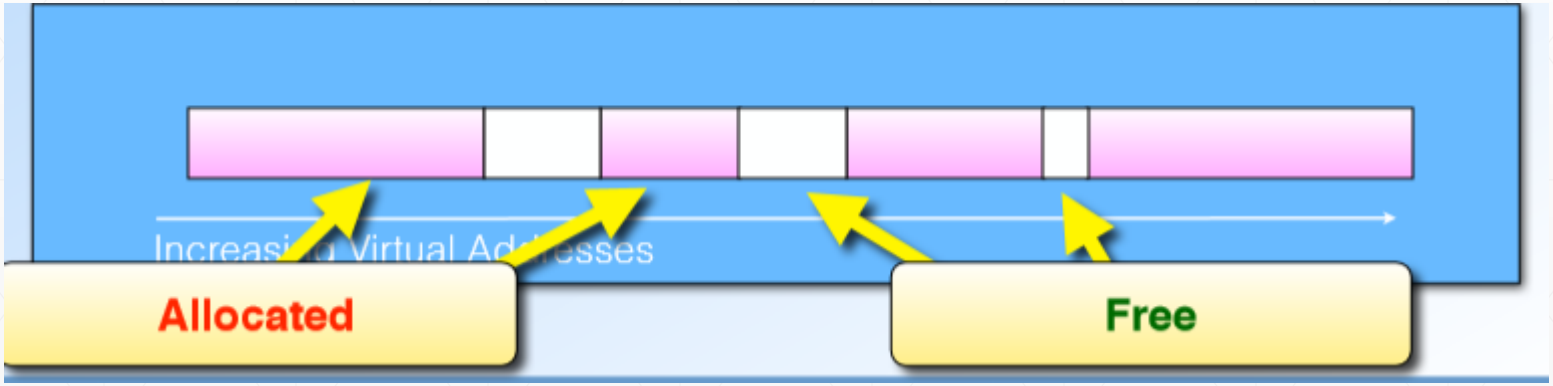
# 相关工作

- 内存分配
  - 多数显式分配
    - C: malloc/alloc
    - C++/Java: new
  - 编译器隐式调用分配器
    - Prolog
- **内存回收**
  - **部分显式**
    - **C: free**
    - **C++: delete**
  - **部分自动回收**
    - **Java、Python**

# 内存分配

- 任务：对于给定的大小S，找到连续的内存区域，大小至少为S
- 问题：

随着分配和回收的执行，出现大量细小的不连续的可用空间——碎片



# 影响1：要寻找合适空闲块

策略：

**first-fit,** 使用第一次找到的≥所需大小的空闲块

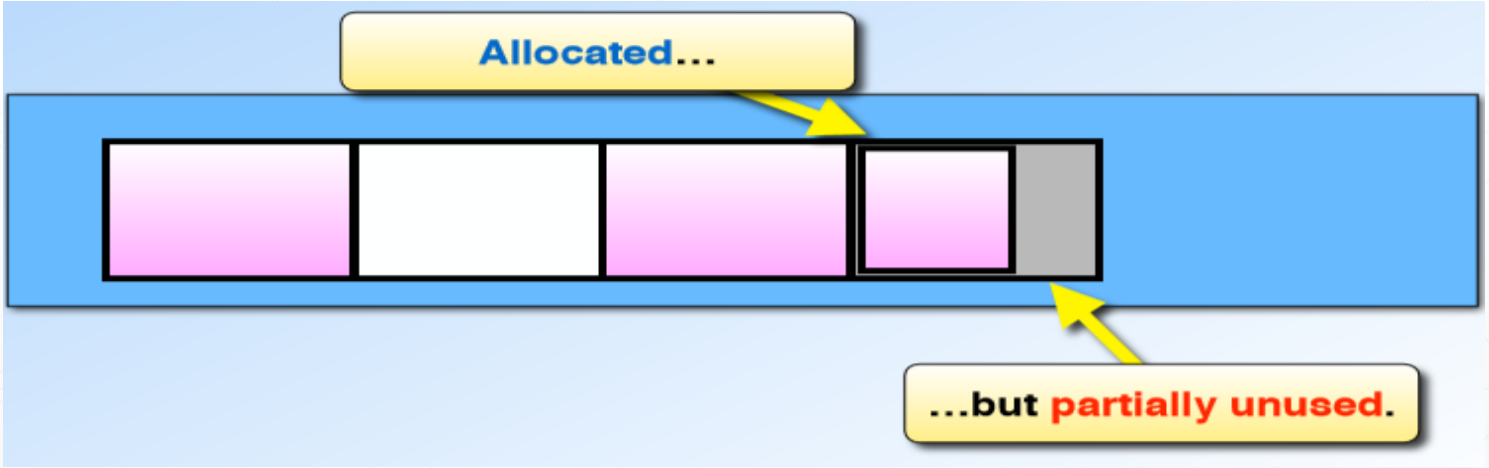
**best-fit,** 使用最接近所需大小的空闲块

**next-fit** 使用刚分配位置的紧邻的空闲块

# 空闲块的组织形式：

## 实际使用

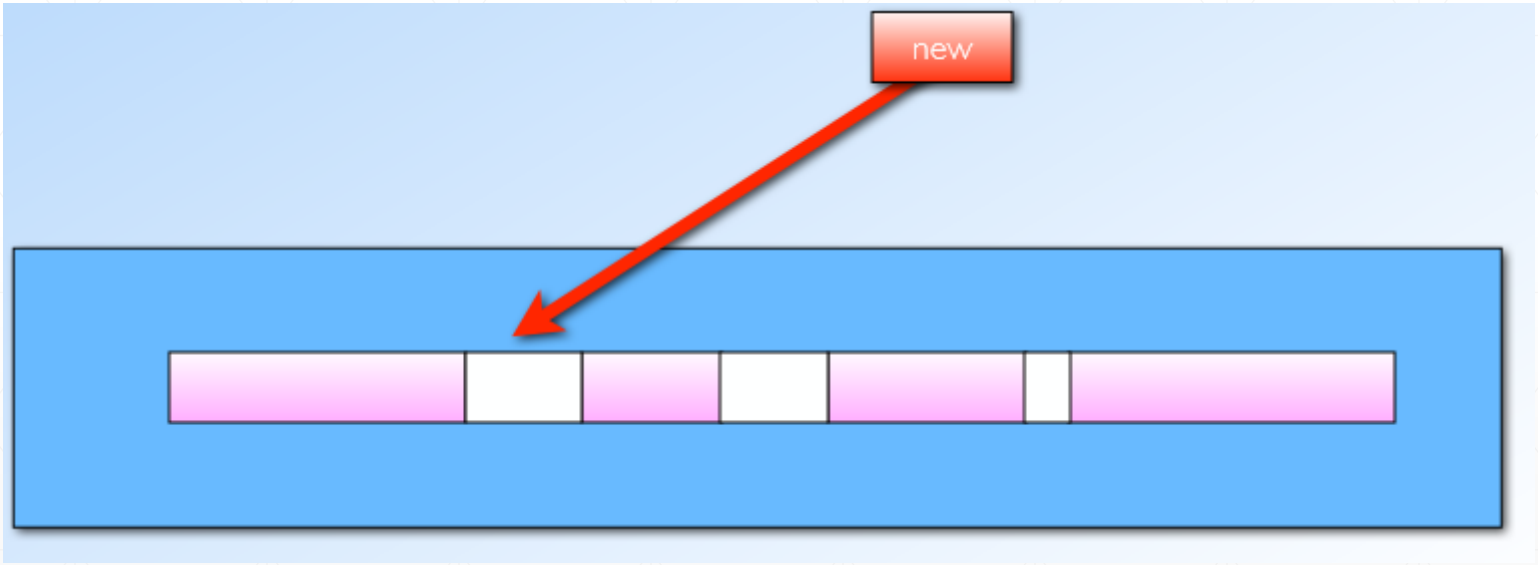
| 可变大小 | 不同的固定大小 |
| --- | --- |
| • 分配和释放均比较耗时 | • 分配和释放比较高效 |




- 固定大小的问题：造成一定空间的浪费

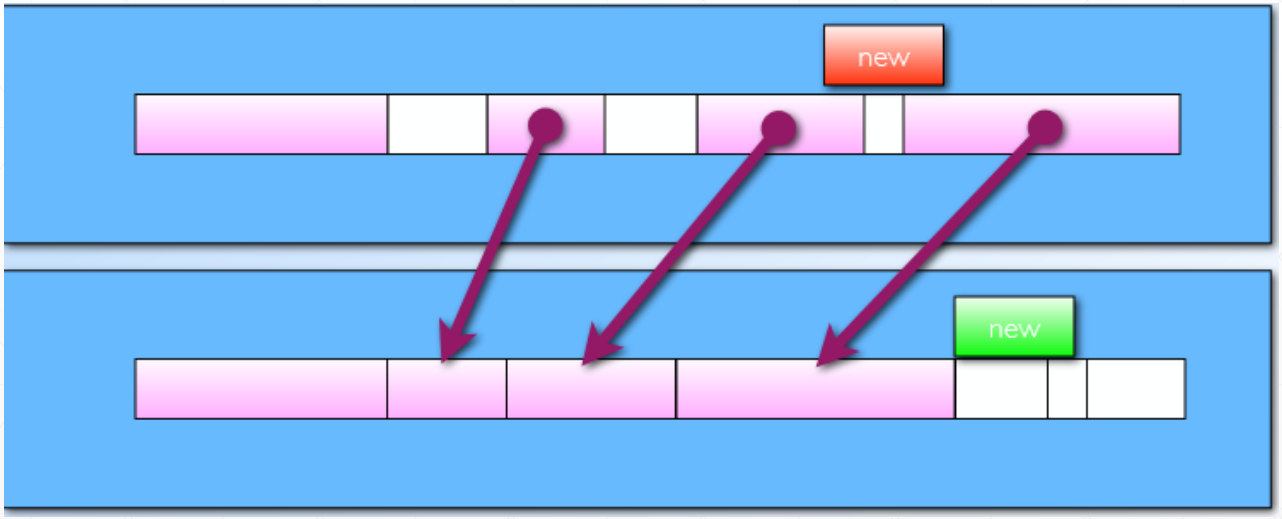
## 影响2：

- 空闲的总空间够，但是找不到合适的空间



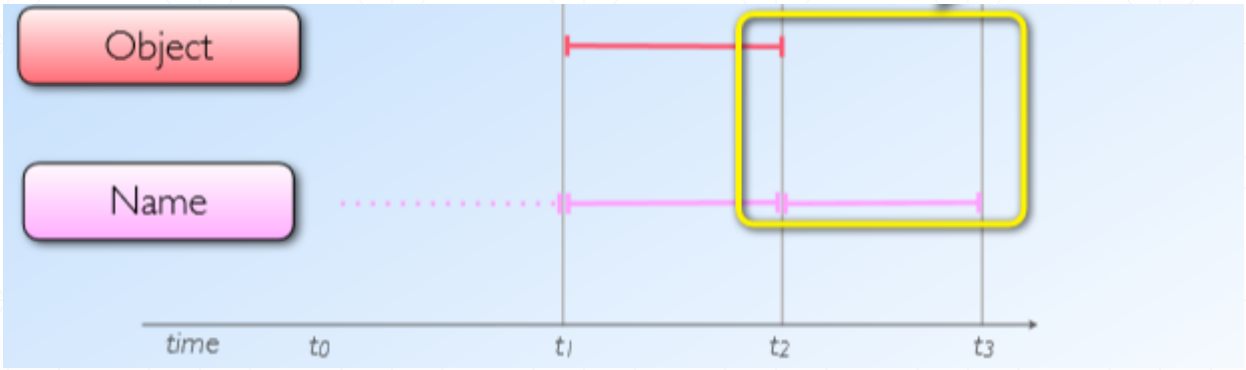
# 解决办法

- 内存空间整理：空闲块合并



# 手动内存管理的常见问题

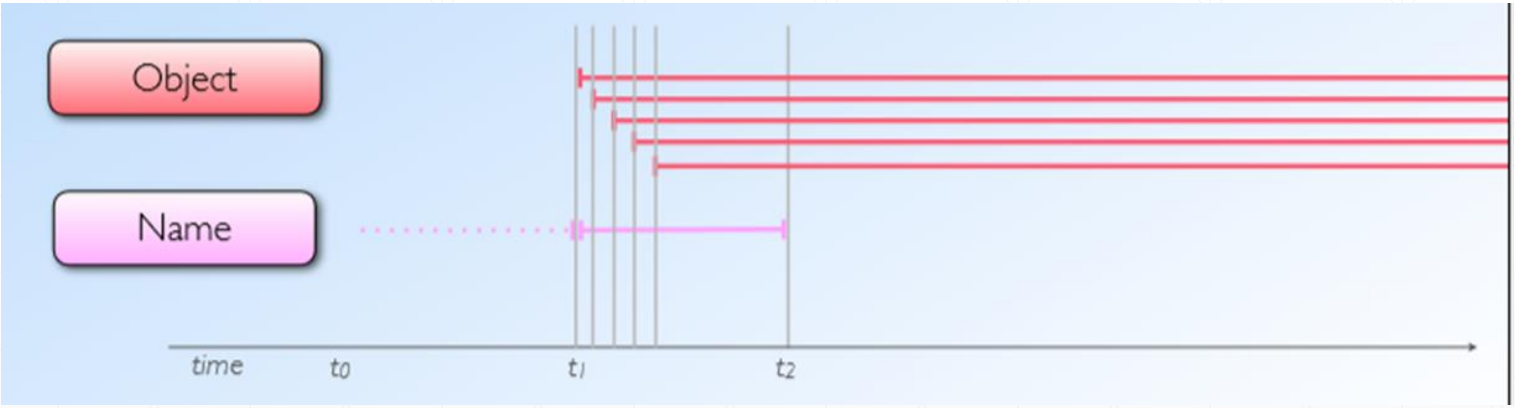
- 问题1：悬空指针（Dangling pointer）
  - 绑定的声明周期长于对象的声明周期




- 对象释放过早，访问变为非法
- 导致“use-after-free”的问题

# 手动内存管理的常见问题

- 问题2：内存泄露（Memory leaks）
  - 忘记释放，对象永远存在




  - 长时间运行导致内存耗尽
  - 进程结束时才会释放

# 垃圾回收器——存储管理的一个子系统

- 运行时系统安全时自动回收对象
- 自动内存管理技术，用户无需关心内存释放
- 使用垃圾回收的语言
  - 函数式编程语言：Haskell、ML
  - 命令式语言：Python、Ruby、Java、C#

# 垃圾收集——找出可以回收的对象

- 第1步：停止程序运行（Stop the world）
- 第2步：找出寄存器中和运行时栈里面的对象引用集合（Root set）
- 第3步：根据Root set找到它们引用的其他对象
- 第4步：不断迭代，直到找不到新对象
- 第5步：从Root Set出发不可达的对象就是可以回收的对象

# 主要的垃圾回收算法

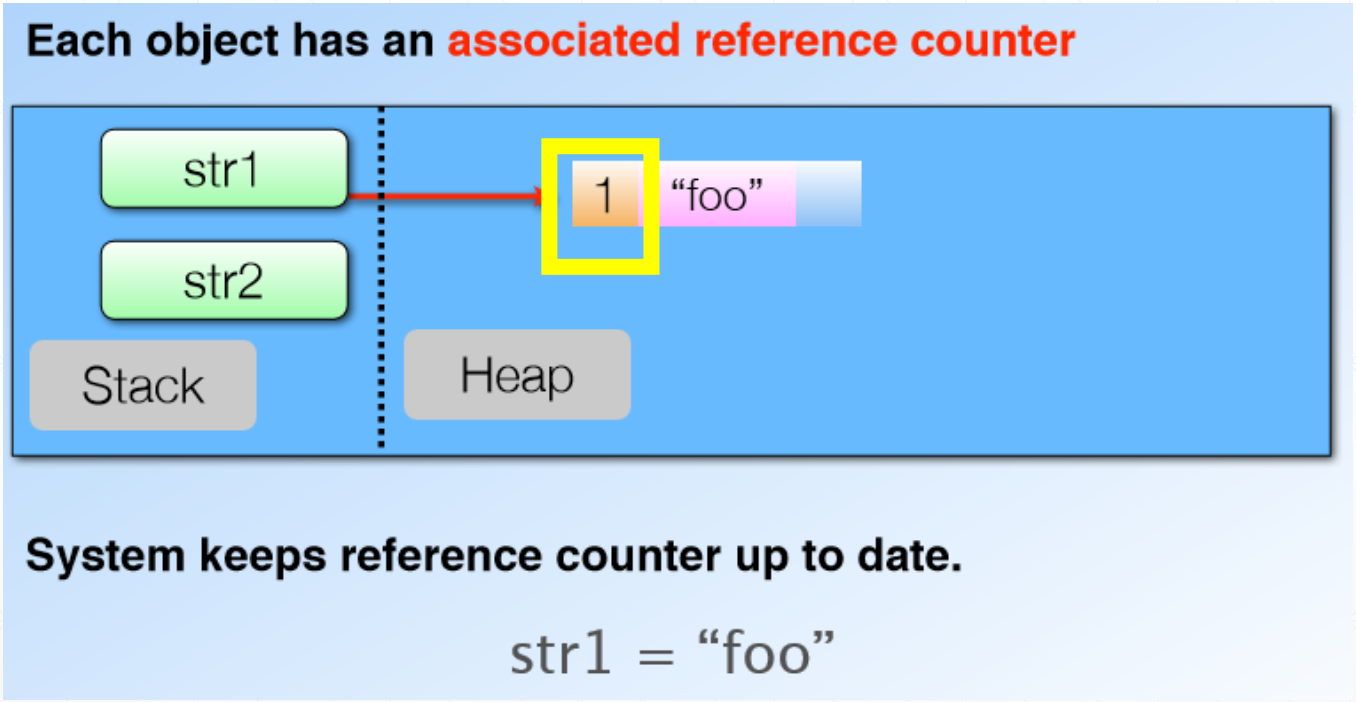
- 引用计数算法
- 拷贝算法
- 标记清除算法
- 按代垃圾收集算法

# 1. 引用计数算法

- 基本思想
  - 每个对象设置一个引用计数器
  - 每增加一次引用时计数器加1
  - 每减少一次引用时计数器减1
  - 从Root set可达的对象引用计数>0
  - 引用计数=0时可以回收
- 特点
  - 实现简单，在很多项目中得到应用

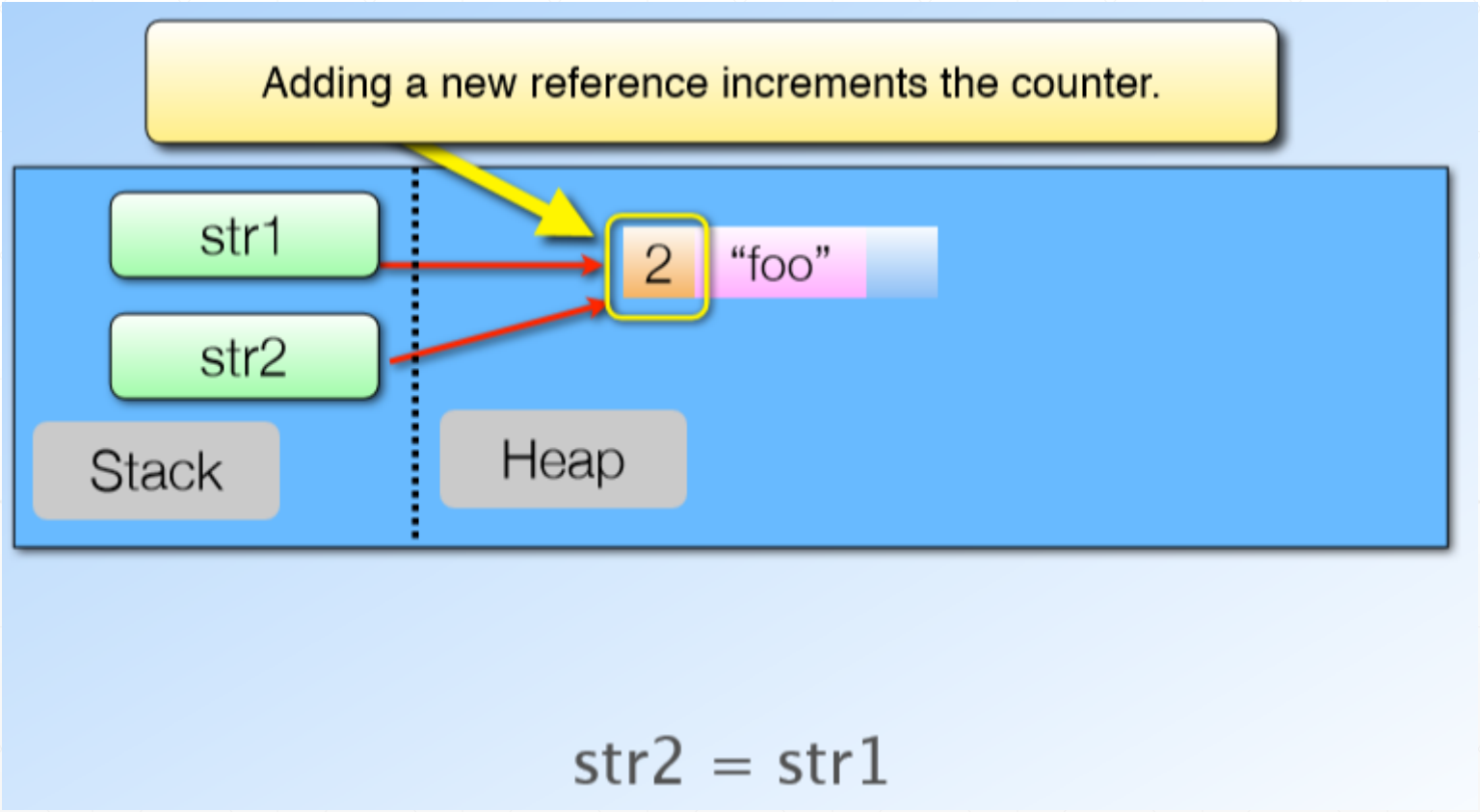
# 1. 引用计数算法

- 例子


Each object has an associated reference counter
str1
str2
Stack
Heap
1
“foo”
System keeps reference counter up to date.
str1 = “foo”

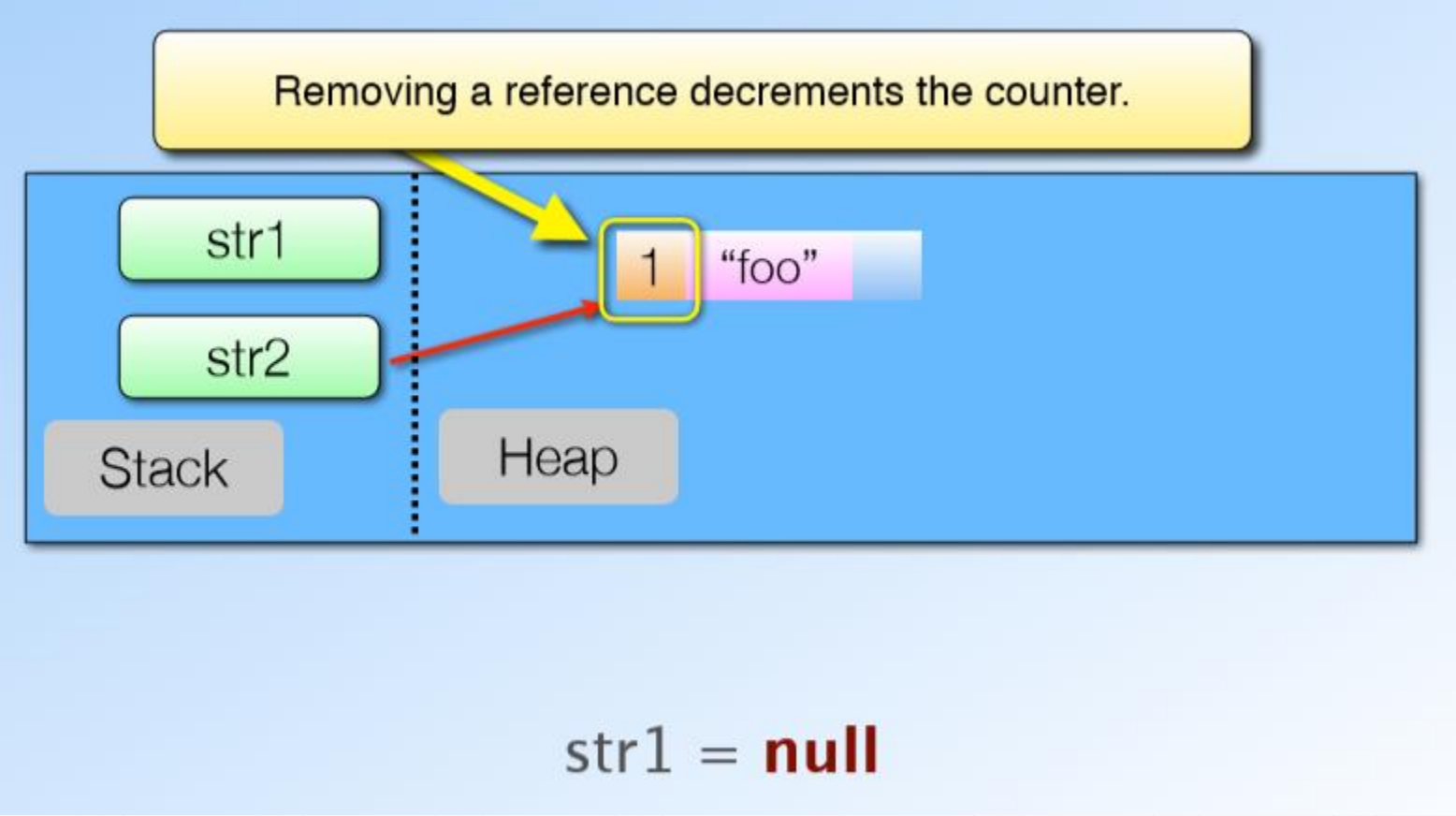
# 1. 引用计数算法

- 例子


Adding a new reference increments the counter.
str1
str2
2
“foo”
Stack
Heap
str2 = str1

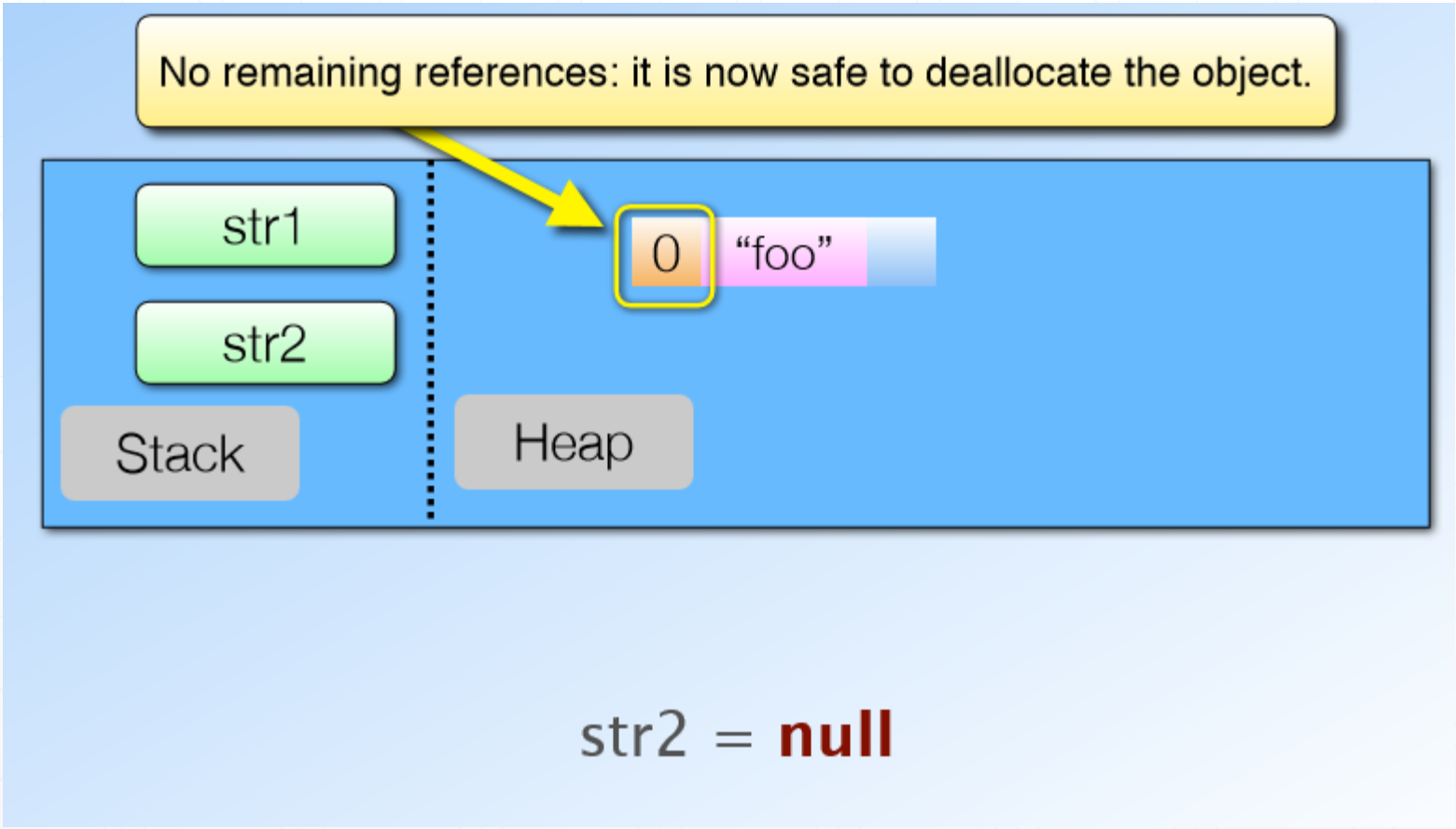
# 1. 引用计数算法

- 例子


Removing a reference decrements the counter.
str1
str2
1
“foo”
Stack
Heap
str1 = null

# 1. 引用计数算法

- 例子

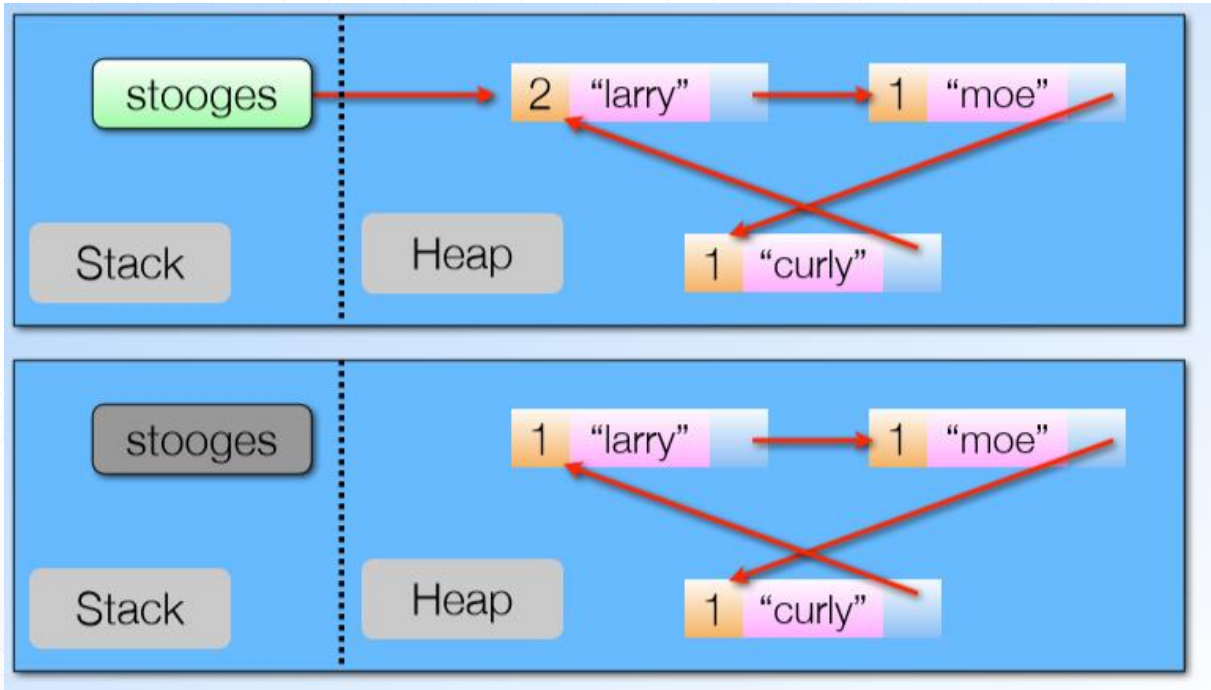

No remaining references: it is now safe to deallocate the object.
str1
str2
Stack
0
"foo"
Heap
str2 = null

# 1. 引用计数算法

- 存在的主要问题
  - 每次引用操作都需要更新计数器，程序性能受影响
  - 多线程环境下，引用计数器更新需要原子性操作保护
  - 循环引用问题

# 1. 引用计数算法

- 循环引用问题




- 不可达，但是引用计数不是0
- 造成内存泄露，需要额外算法检测

# 2. 标记清除/Mark & Sweep算法

## 基本思想

- 为每个对象设置一个标记flag

## 步骤

1. 清除所有对象的标记flag
2. 从Root set出发遍历所有可达对象
   对可达对象进行标记（Mark）
3. 没有被标记的对象就是垃圾
4. 回收所有没有被标记的对象（Sweep）

## 内存空间耗尽时触发

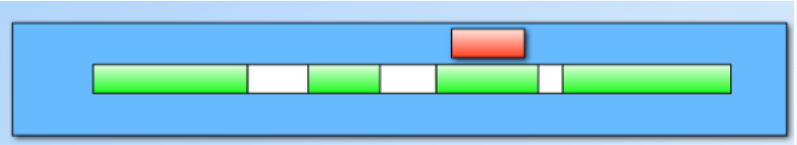
## 2. 标记清除/Mark & Sweep算法

- 主要挑战
  - 停止程序运行：Stop the world
  - 防止遍历对象的过程中对象引用关系被修改
  - 内存空间较大时，用户界面无法响应
  - 算法运行时本身需要空间
- 解决办法
  - 并发垃圾回收，垃圾回收与程序交替运行，期间监控引用操作
  - 增量垃圾收集，更加复杂

# 垃圾回收：标记清除和引用计数比较

| 标记清除 | 引用计数 |
| --- | --- |
| • Stop the world<br>• 其余时间运行比较高效<br>• 精确算法，不会导致内存泄露<br>• 实现复杂 | • 应用无停顿<br>• 引用写操作比较慢<br>• 存在环路引用的问题<br>• 实现简单 |

**共同的问题：**



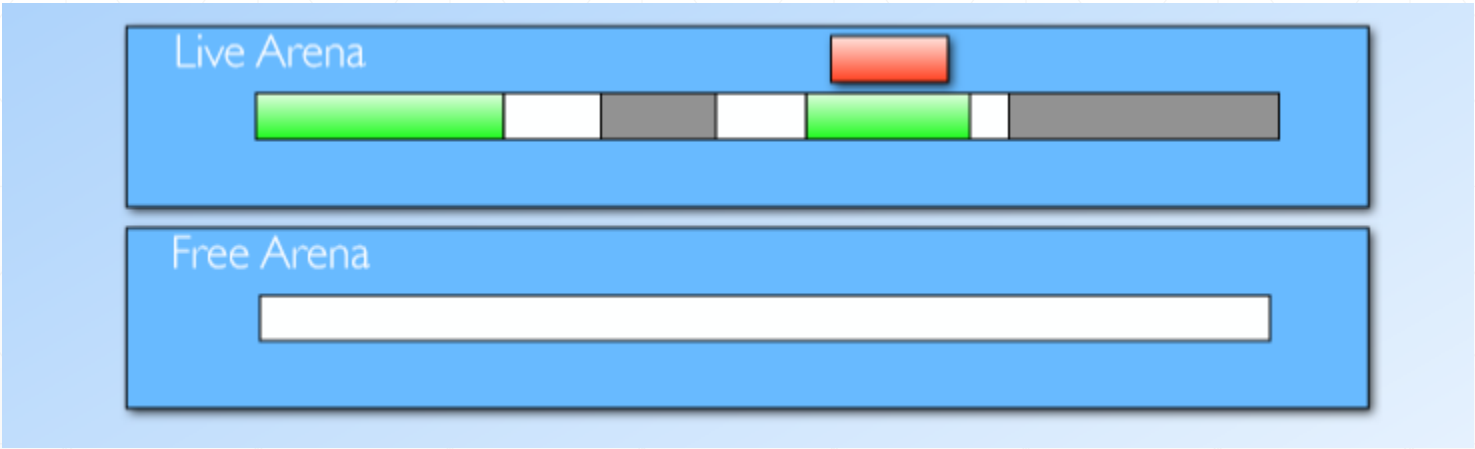
找到无用对象并释放掉并不能解决内存碎片问题

# 3. 拷贝算法

- 基本想法
- 将内存空间分为两个半区A和B
- 首先在A半区内紧挨着创建对象
- A半区满时将存活对象拷贝到B半区
- 从B半区空闲位置开始创建对象
- B半区满时重复以上步骤

# 3. 拷贝算法

- 某时刻的内存占用情况
  - 绿色表示存活对象，灰色表示无用对象



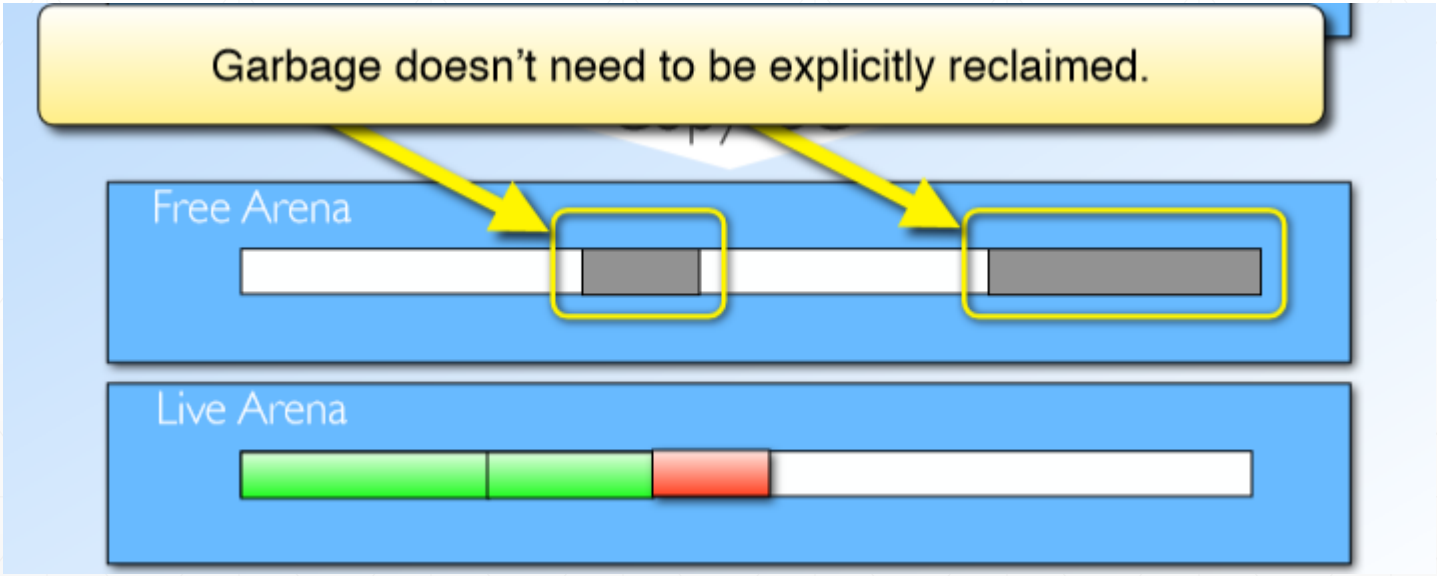
# 3. 拷贝算法

- 具体过程
  - 从Live区拷贝存活对象到Free区
  - 交换Live和Free
  - 在新的Live区创建新对象


# 3. 拷贝算法

- 优点

1.无需显式回收无用对象，自然被抛弃



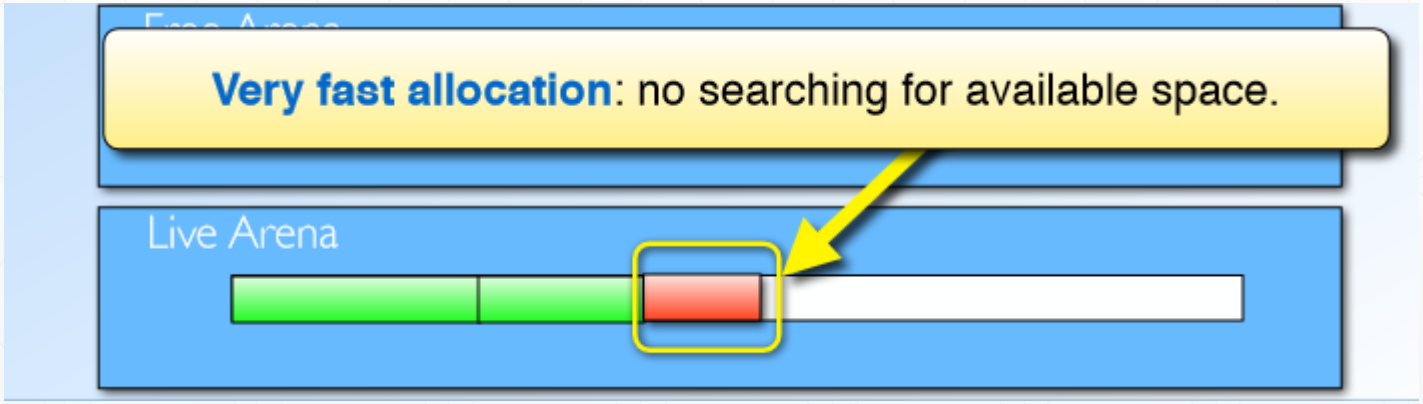
# 3. 拷贝算法

- 优点

1. 无需显式回收无用对象，自然被抛弃
2. 分配空间非常快，地址做加法

- 缺点

任何时刻只有一半空间可以用

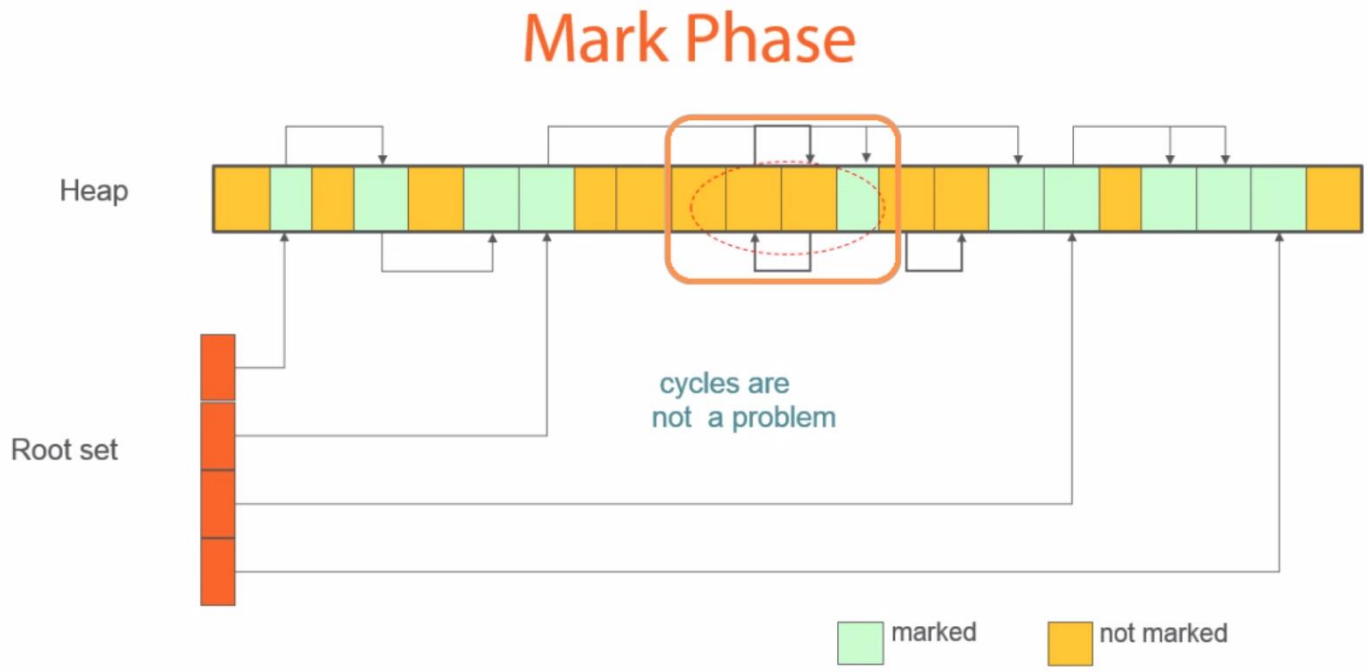


# 标记清除算法的改进——标记压缩

- 主要思想
  - 利用标记算法标记完成后
  - 存活对象拷贝集中移动到一端
  - 无用对象自然抛弃
  - 从新的空闲空间开始位置创建对象
- 优点：整个存储空间可用
- 缺点：每次垃圾收集耗时较长

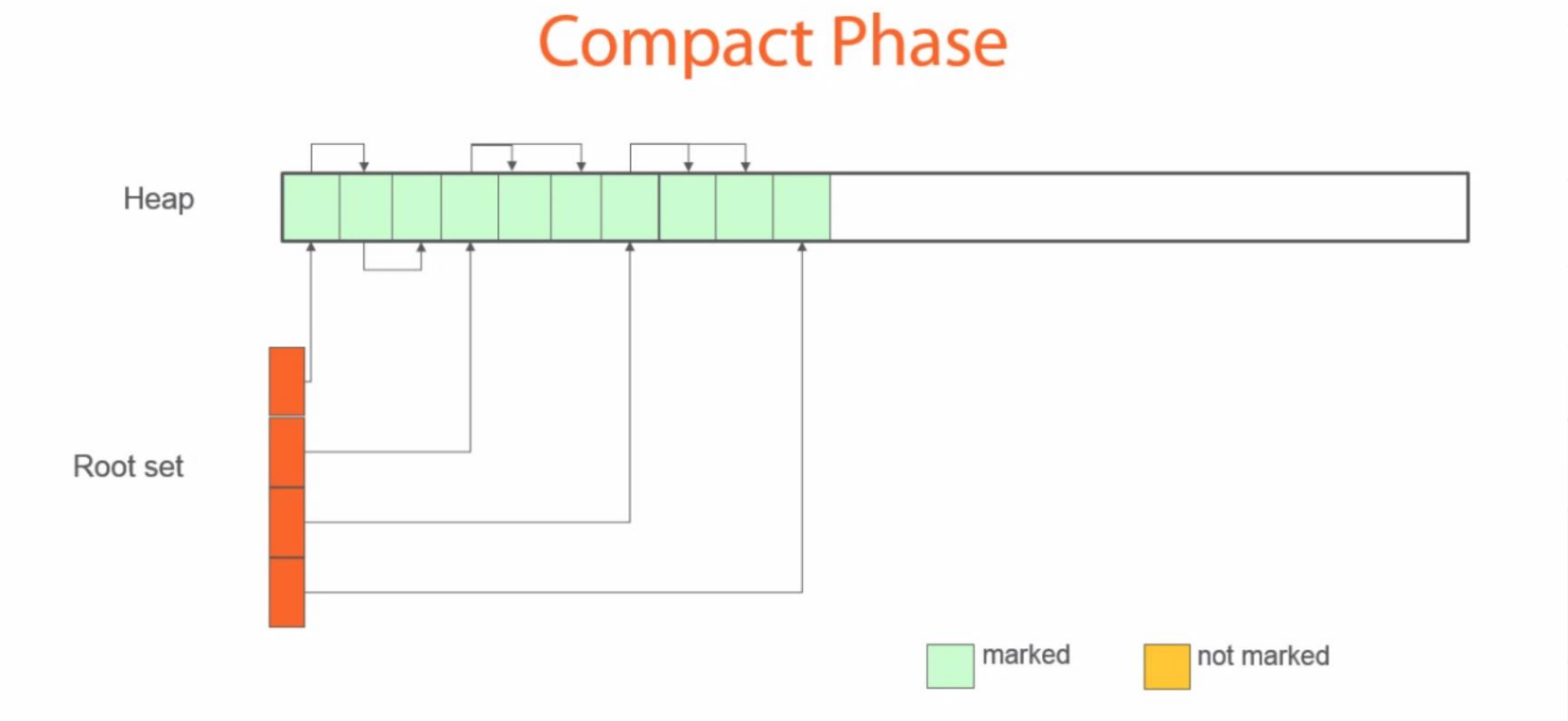
# 标记清除算法的改进——标记压缩

- 标记阶段



# 标记清除算法的改进——标记压缩

- 标记压缩



# 比较：标记清除与标记压缩

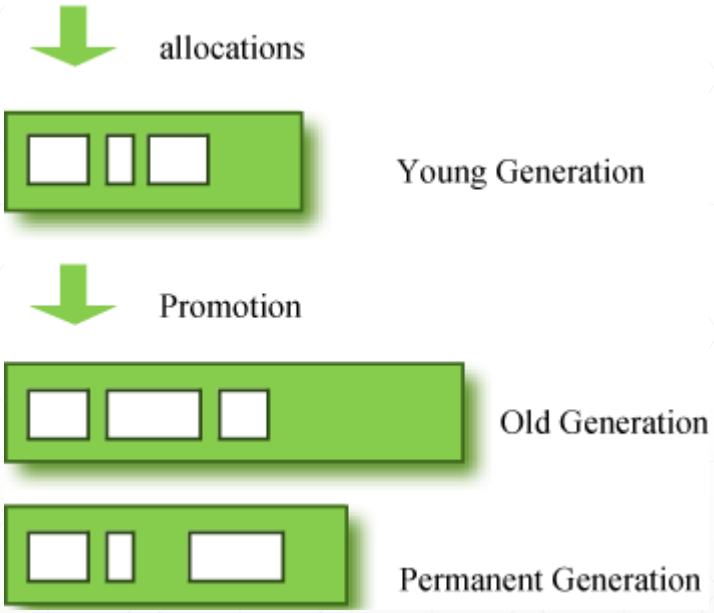
# 4. 按代垃圾回收

- 基于如下的观察结果
  - 大部分对象的生命周期比较短
  - 大部分对象在创建不久就变成无用的了
  - 存活时间越久的对象越不可能是垃圾
- 基本思路
  - 将内存划分为不同区域：新生代和旧生代
  - 新生代存活对象拷贝到旧生代
  - 新生代垃圾收集频率较高，旧生代较低

大部分虚拟机采用该算法

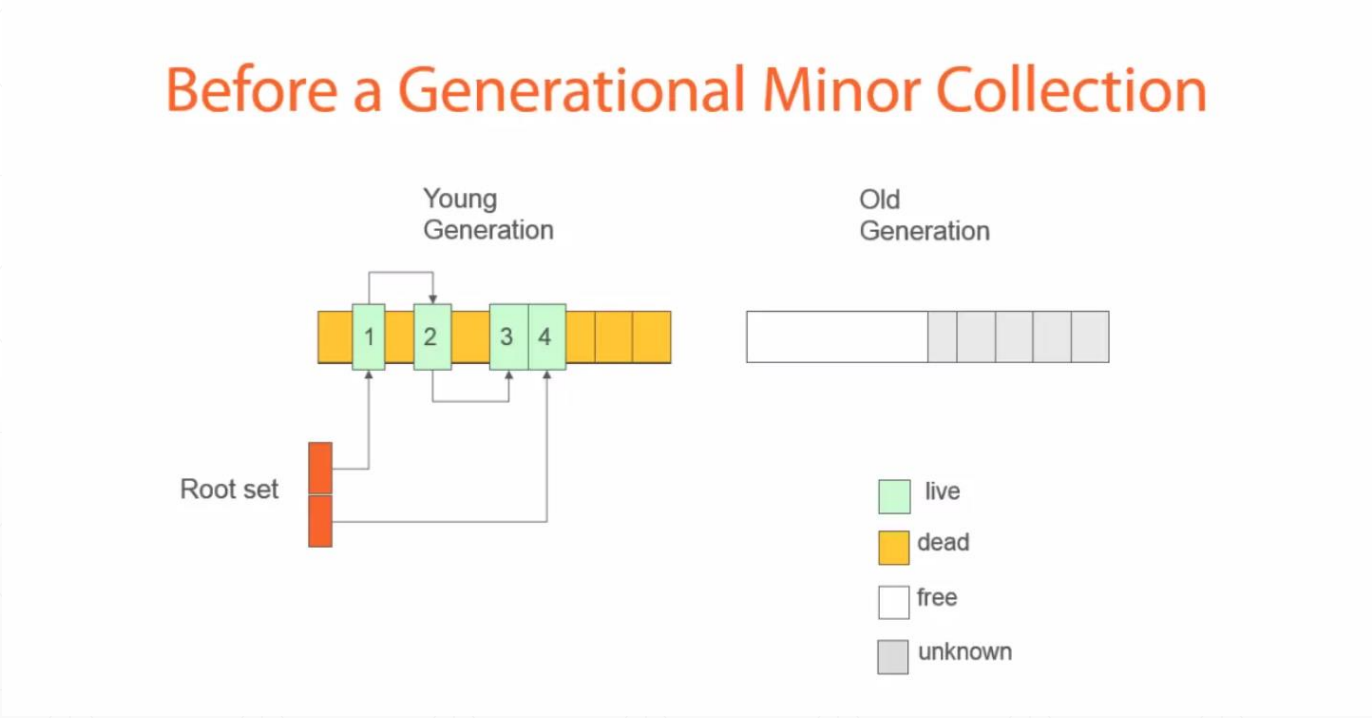
# 4. 按代垃圾收集

- 对象首先分配在新生代
- 经过新生代区域垃圾收集后存活下来的进入旧生代
- 永久存储区存放代码（类）



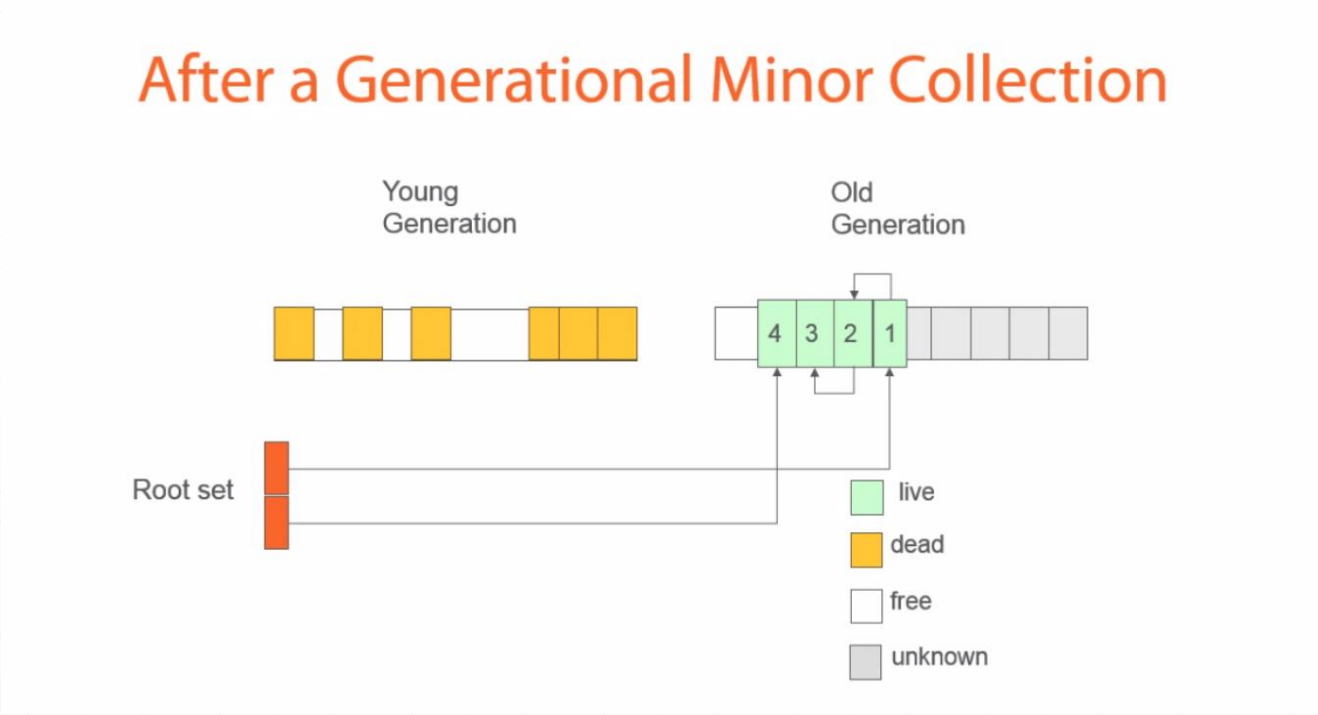
# 垃圾收集：按代垃圾收集

- 例子



# 垃圾收集：按代垃圾收集

- 新生代收集后



# 运行环境　　总结

三种运行环境及三种分配方案

概念：活动记录

以过程为单位的栈式动态分配方案

解决非局部变量的引用查找的两种方式：

数据链SL；

嵌套层次显示表Display。

堆式分配的碎片问题

# 第七章 结 束