
Lecture 14: 运行时刻环境-II

Xiaoyuan Xie 谢晓园

xxie@whu.edu.cn

计算机学院E301



栈中非局部数据的访问

作用域

- 所谓的**作用域**就是指**某段程序文本代码**
 - 一个声明**起作用**的那段程序文本区域，称为**该声明的作用域**
 - 静态作用域：声明的作用域是根据程序正文在**编译时就确定的**，有时也称为词法作用域 - C、PASCAL、ML
 - 动态作用域：程序中某个变量所引用的对象是在程序运行时刻根据程序的控制流信息来确定的 - APL、Snobol和Lisp

注意区分：静态存储分配，静态作用域

静态作用域

■ 平坦结构，允许递归调用：C语言

```
1 int x = 1;
2 int g(int z) { return x + z; }
3 int f(int y)
4 {
5     int x = y + 1;
6     return g(y*x);
7 }
8 f(3);
```

没有任何一个函数属于另一个函数
e.g. f 和 g 是并列的

外层块	x	1
-----	---	---

f(3)	y	3
	x	4

g(12)	z	12
-------	---	----

静态作用域

■ 平坦结构，允许递归调用：C语言

- 每个函数能够访问的变量
 - 函数的局部变量：相对地址已知，且存放在当前活动记录，可以通过 *base_sp* 指针来访问
 - 函数非局部变量：即全局变量，在静态区，地址在编译时刻可知
- 很容易将函数作为参数进行传递，也可以作为结果来返回
 - 参数中只需包括函数代码的开始地址。
 - 在函数中访问非局部变量的模式很简单，不需要考虑过程是如何激活的，即无须深入栈中取数据，无须访问链

静态作用域

■ 嵌套定义，允许递归调用：PASCAL, ML

```
1) fun sort(inputFile, outputFile) =  
    let  
2)      val a = array(11,0);  
3)      fun readArray(inputFile) = ...  
4)        ... a ... ;  
5)      fun exchange(i,j) =  
6)        ... a ... ;  
7)      fun quicksort(m,n) =  
          let  
8)          val v = ... ;  
9)          fun partition(y,z) =  
10)             ... a ... v ... exchange ...  
          in  
11)             ... a ... v ... partition ... quicksort  
          end  
    in  
12)      ... a ... readArray ... quicksort ...  
    end;
```

静态作用域

■ 嵌套层次(深度)

嵌套深度是正文概念，可以根据源程序静态地确定：不内嵌于任何其他过程中的过程，嵌套深度为1；嵌套在深度为i的过程中的过程，深度为i+1

```
1) fun sort(inputFile, outputFile) =  
  let  
2)    val a = array(11,0);  
3)    fun readArray(inputFile) = ...  
4)      ... a ... ;  
5)    fun exchange(i,j) =  
6)      ... a ... ;  
7)    fun quicksort(m,n) =  
      let  
8)        val v = ... ;  
9)        fun partition(y,z) =  
10)          ... a ... v ... exchange ...  
      in  
11)        ... a ... v ... partition ... quicksort  
      end  
    in  
12)    ... a ... readArray ... quicksort ...  
  end;
```

深度为1

sort

深度为2

readArray,
exchange,
quicksort

深度为3

partition

允许嵌套定义的名字作用域规则（静态作用域）

■ 最内嵌套作用域规则：

- 由一个声明引进的标识符在这个声明所在的作用域里可见，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的对同名标识符的另一个声明所掩盖。
 - fun f 可以访问其内部定义的所有名字：use a variable, call a nested function
 - 可以访问包围fun f 的任意外层过程所定义的名字（使用变量，调用函数）

允许嵌套定义的名字作用域规则（静态作用域）

■ 从当前最内层作用域开始查找

- 如找到，则可以找到该标识符所引用的对象；
- 否则，到直接的外层作用域里去查找，并继续向外顺序地检查外层作用域，直到到达程序的最外嵌套层次，也就是全局对象声明所在的作用域
- 如果在所有层次上都没有找到有关声明，那么这个程序就有错误。

```

proc S ( ——— )
{
  var a = array( )
  {
    proc r ( ——— )
    { // 引用 a }
    proc e ( ——— )
    { // 引用 a }
    proc q ( ——— )
    {
      var v
      proc p ( ——— )
      {
        // 引用 a
        // 引用 v
        call e // 调用 e } 调用
      }
    }
  }
}

```

S 定义在 1 层

从 S 的形参开始, 进入 2 层

a, r, e, q 都定义在 2 层

从 r, e, q 的形参开始
进入 3 层, v, p 都定义在 3 层

从 p 的形参开始, 进入 4 层
4 层无定义。

```

使用 {
  // 引用 a
  // 引用 v
  call p // 调用 p
  call q // 调用 q 的自身
}

调用 {
  // 引用 a
  call r // 调用 r
  call q // 调用 q
}

```

允许嵌套定义的名字作用域规则（静态作用域）

■ 困难在哪里？

注意区别：嵌套定义 v.s. 嵌套调用
即嵌套定义与活动记录栈位置没有对应关系

例：PASCAL中，如果过程A的声明中包含了过程B的声明，那么B可以使用在A中声明的变量。当B的代码运行时，如果它使用的是A中的变量。那么这个变量指向运行栈中最上层的同名变量。但是，我们不能通过**嵌套层次**直接得到A的活动记录的相对位置，必须通过**访问链**访问

```
void A()
```

```
{
```

```
  int x,y;
```

```
  void B() { int b; x = b+y; }
```

```
  void C(){B();}
```

```
  C();
```

```
  B();
```

```
}
```

当A调用C，C又调用B时：



A的活动记录
C的活动记录
B的活动记录

当A直接调用B时：



A的活动记录
B的活动记录

关键问题：B和C都需要
(可以)访问直系外层函数
A的名字(使用变量或调
用函数)，如何找到A在
栈里的活动记录？

解决方法-访问链

■ 使用访问链进行定位

- 当被调用过程需要其他地方的某个数据时需要使用访问链进行定位
- 如果过程p在声明时嵌套在过程q的声明中，那么p的活动记录中的访问链指向最上层的q的活动记录
- 从栈顶活动记录开始，访问链形成了一个链路，嵌套深度沿着链路逐一递减

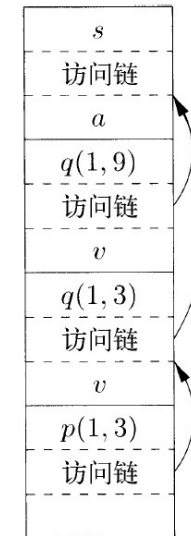
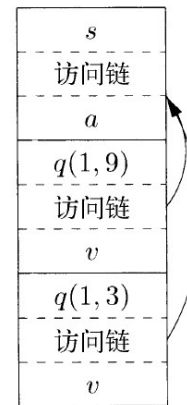
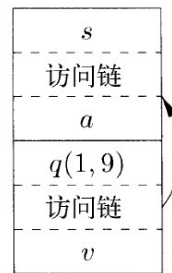
解决方法-访问链

- 设深度为 n_p 的过程 p 访问变量 x ，而变量 x 在深度为 n_q 的过程中声明，那么
 - $n_p - n_q$ 在编译时刻已知
 - 从当前活动记录出发，沿访问链前进 $n_p - n_q$ 次找到的活动记录中的 x 就是要找的变量位置
 - x 相对于这个活动记录的偏移量在编译时刻已知

解决方法-访问链

■ 图解如何实现的访问

```
1) fun sort(inputFile, outputFile) =  
  let  
2)    val a = array(11,0);  
3)    fun readArray(inputFile) = ... ;  
4)    ... a ... ;  
5)    fun exchange(i,j) =  
6)      ... a ... ;  
7)    fun quicksort(m,n) =  
      let  
8)        val v = ... ;  
9)        fun partition(y,z) =  
10)          ... a ... v ... exchange ...  
11)      in  
          ... a ... v ... partition ... quicksort  
        end  
      in  
12)    ... a ... readArray ... quicksort ...  
      end;
```



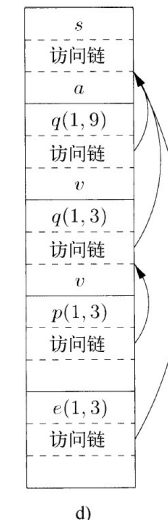
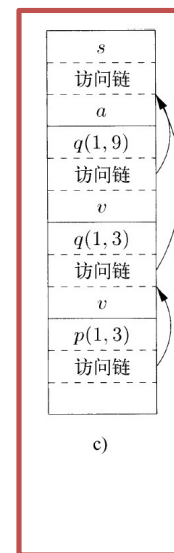
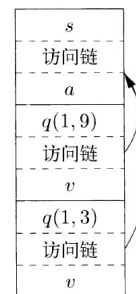
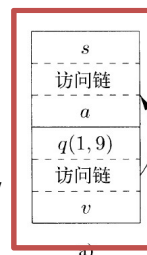
解决方法-访问链

■ 如何构造并维护访问链，以过程q调用过程p为例说明

- Case 1: p的深度大于q: 根据作用域规则, p必然在q中直接定义; 那么p的访问链指向当前活动记录,

对应于规则中 “fun f 可以访问其内部定义的
所有名字use a variable, call a nested function”

此时, 定义的嵌套关系与调用的嵌套关系一致

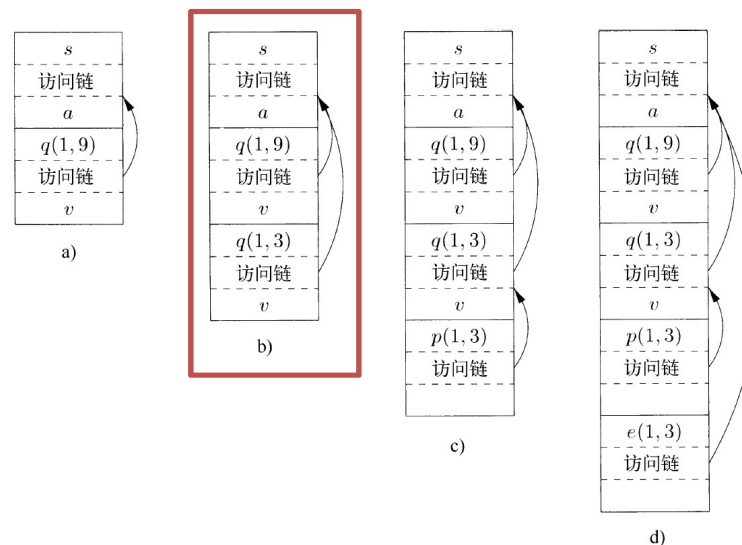


解决方法-访问链

■ 如何构造并维护访问链，以过程q调用过程p为例说明

- Case 2: 递归调用 ($p=q$) : 新活动记录的访问链等于当前记录的访问链

- $q(1,9)$ 调用 $q(1,3)$

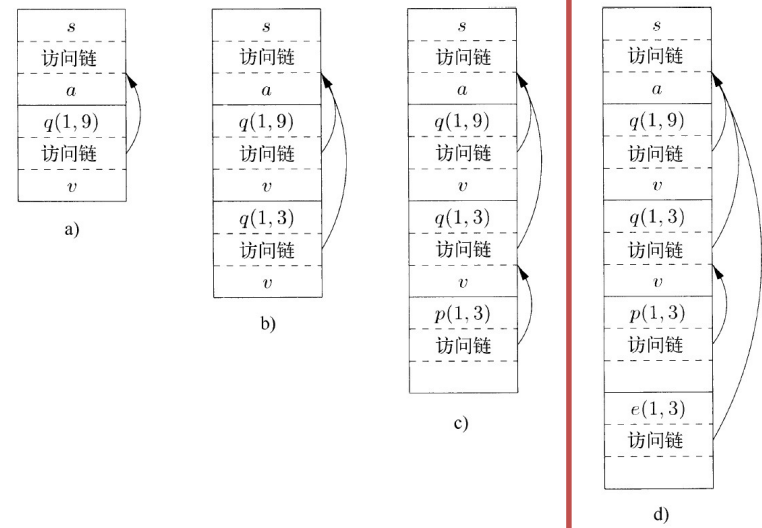


解决方法-访问链

■ 如何构造并维护访问链，以过程q调用过程p为例说明

- Case 3: p的深度小于等于q的深度：此时必然有过程r，p直接在r中定义，而q嵌套在r中；p的访问链指向栈最高的r的活动记录

- p调用exchange



```

proc s ( — )
{
  var a = array( — )
  proc r ( — )
  { // 访问 a }
  proc e ( — )
  { // 访问 a }
  proc q ( — )
  {
    var v
    proc p ( — )
    {
      // 访问 a
      // 访问 v
      call e // 访问 e
    }
  }
}

```

S 能访问所有变量
 再上一层无法访问，
 call 嵌套与定义嵌套一致

(Case 1)

黄色所访问黄色 (Case 2)
 也可访问其所定义的绿色
 变量 (Case 1)

绿色所访问绿色 (Case 2)
 也可访问其所定义的黄色

好 (Case 1)
 还可以访问黄色名字
 (Case 3)

使用
 访问 { // 访问 a
 // 访问 v
 call p // 访问 p
 call q // 访问 q 自身

(Case 3)

访问 { // 访问 a
 call r // 访问 r
 call q // 访问 q

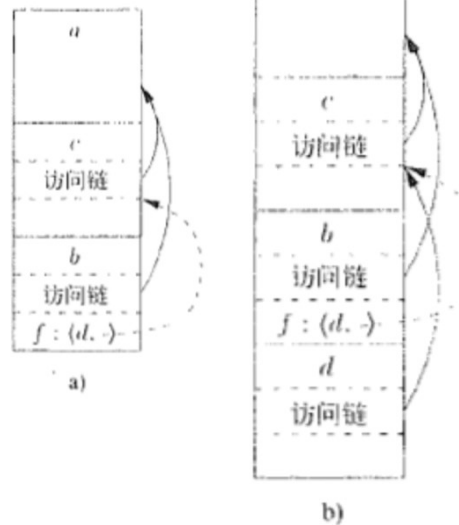
ps: 访问名字显示
 1) 访问变量名
 2) call 函数名

解决方法-访问链

■ 过程指针型参数的访问链

- 在传递过程指针参数时，过程型参数中不仅包含过程的代码指针，还包括正确的访问链

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



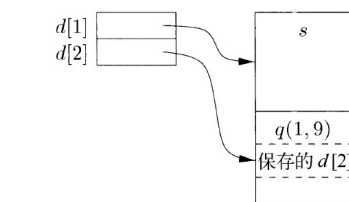
实现方法-display表

- 用访问链访问数据时，访问开销和嵌套深度差有关
- 使用显示表可以提高效率，访问开销为常量
 - 数组d为每个嵌套深度保留一个指针
 - 指针d[i]指向栈中最高的、嵌套深度为i的活动记录
 - 如果程序p中访问嵌套深度为i的过程q中声明的变量x，那么d[i]直接指向相应的（必然是q的）活动记录
 - 注意：i在编译时刻已知

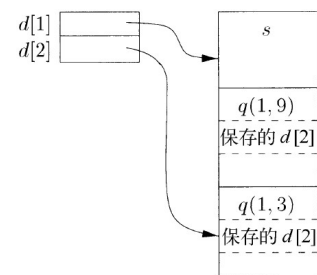
实现方法-display表

■ 显示表维护例子

调用过程p时，在p的活动记录中保存d[np]的值，并将d[np]设置为当前活动记录。从p返回时，恢复d[np]的值。

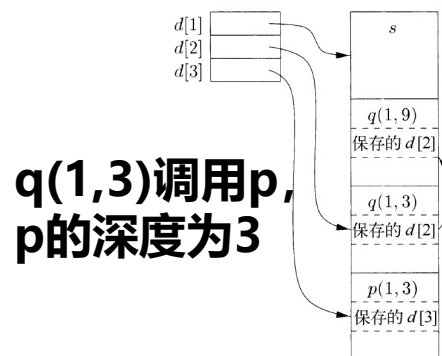


a)



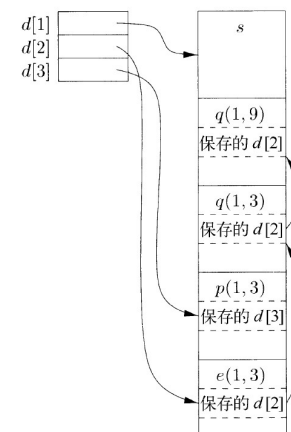
b)

$q(1, 9)$ 调用
 $q(1, 3)$ 时,
 q 的深度为2



c)

$q(1, 3)$ 调用p,
p的深度为3



d)

q 调用e
e的深度为2

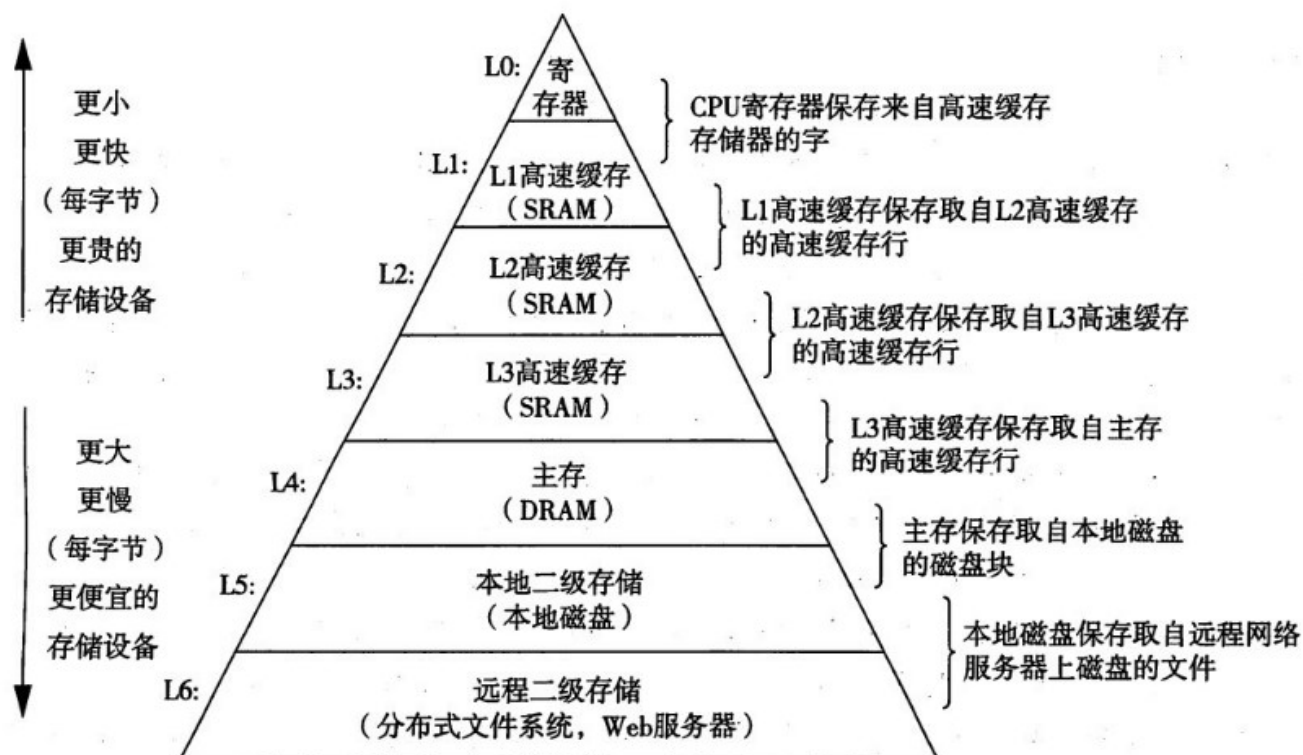
作业

- 教材p290: 7.3.1, 7.3.2



堆管理

计算机内存分层



计算机内存分层

- 现代计算机都设计成程序员不用关心内存子系统的细节就可以写出正确的程序
- 程序的效率不仅取决于被执行的指令数，还取决于执行每条指令需要多长时间
- 执行一条指令的时间区别非常可观
- 差异源于硬件技术的基本局限：构造不了大容量的高速存储器
- 数据以块（缓存行、页）为单位在相邻层次之间进行传送
- 数据密集型程序可从恰当利用内存子系统中获益

程序局部性

- 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
 - 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
 - 毗邻被访问单元的内存单元在很短的时间内可能被访问

程序局部性

- 即使知道哪些指令会被频繁执行，最快的缓存也可能没有大到足以把它们同时放在其中，因此必须动态调整最快缓存的内容
- 把最近使用的指令保存在缓存是一种较好的最优化利用内存分层的策略
- 改变数据布局或计算次序也可以改进程序数据访问的时间和空间局部性

堆 管 理

■ 堆用来存放生存期不确定的数据

- C++和Java允许程序员用new创建对象，它们的生存期没有被约束在创建它们的过程活动的生成期之内
- 实现内存回收是内存管理器的责任

■ 堆空间的回收有两种不同方式

- 程序显式释放空间：new生成的对象可以生存到被delete为止，malloc申请的空间生存到被free为止（C/C++）
- 垃圾收集器自动收集（Java）

堆 管 理

- **内存管理器把握的基本信息是堆中空闲空间**
 - 分配/回收堆区空间的子系统
 - 根据语言而定
 - C、C++需要手动回收空间
 - Java可以自动回收空间（垃圾收集）

存储管理器

- **基本功能**

- 分配：为每个内存请求分配一段连续的、适当大小的堆空间
 - 首先从空闲的堆空间分配
 - 如果不行则从操作系统中获取内存、增加堆空间
- 回收：把被回收的空间返回空闲空间缓冲池，以满足其他内存需求

- **评价存储管理器的特性**

- 空间效率：使程序需要的堆空间最小，即减小碎片
- 程序效率：充分运用内存系统的层次，提高效率
- 低开销：使分配/收回内存的操作尽可能高效

堆空间的碎片问题

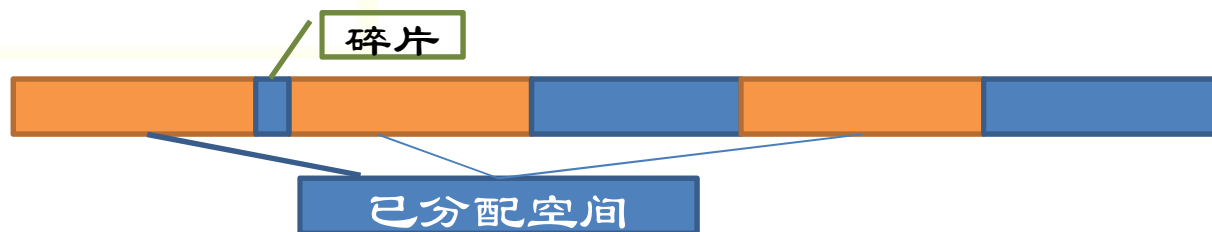
- 随着程序分配/回收内存，堆区逐渐被割裂成为若干空闲存储块（窗口，hole）和已用存储块的交错
- 分配一块内存时，通常是把一个窗口的一部分分配出去，其余部分成为更小的块
- 回收时，被释放的存储块被放回缓冲池。通常要把连续的窗口接合成为更大的窗口

堆空间的碎片问题

an allocated chunk	size/status=inuse
	... user data space ...
	size
a freed chunk	size/status=free
	pointer to next chunk in bin
	pointer to previous chunk in bin
	...unused space ...
	size
an allocated chunk	size/status=inuse
	user data
	size
other chunks	...
wilderness chunk	size/status=free

	size

end of available memory



堆空间分配方法

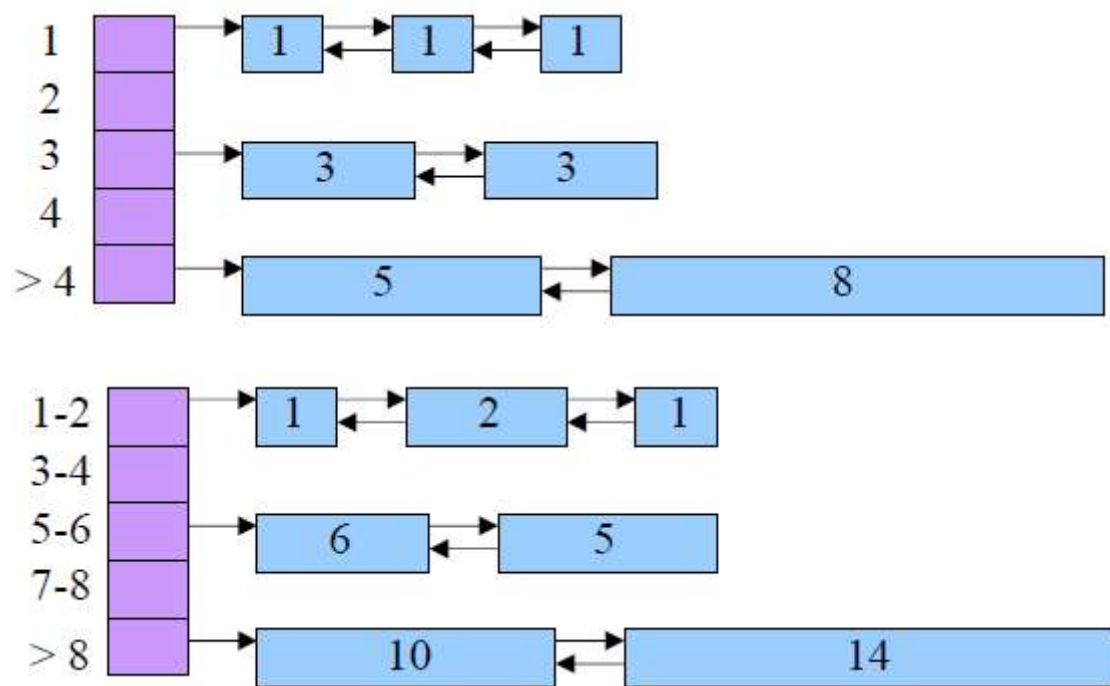
- **Best-Fit**

- 总是将请求的内存分配在满足请求的最小的窗口中
- 好处：可以将大的窗口保留下来，应对更大的请求

- **First-Fit**

- 总是将对象放置在第一个能够容纳请求的窗口中
- 放置对象时花费时间较少，但是总体性能较差
- 但是first-fit的分配方法通常具有较好的数据局部性
 - 同一时间段内生成的对象经常被分配在连续的空间内

有效实现best-fit

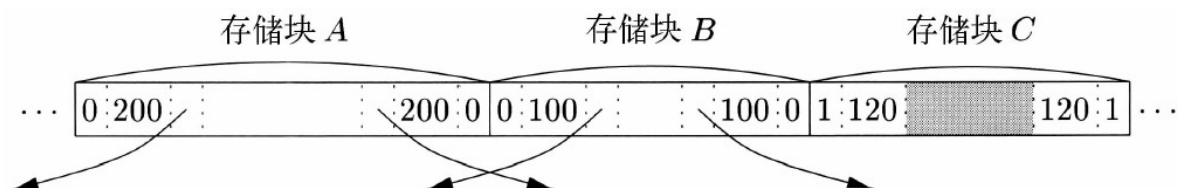


管理和接合空闲空间

- 当回收一个块时，可以把这个块和相邻的块接合起来，构成更大的块
- 支持相邻块接合的数据结构
 - 边界标记：在每一块存储块的两端，分别设置一个free/used位；相邻的位置上存放字节总数
 - 双重链接的、嵌入式的空闲块列表：列表的指针存放在空闲块中、用双向指针的方式记录了有哪些空闲块

例子

- **相邻的存储块A、B、C**
 - 当回收B时，通过对free/used位的查询，可以知道B左边的A是空闲的，而C不空闲。
 - 同时还可以知道A、B合并为长度为300的块
 - 修改双重链表，把A替换为A、B接合后的空闲块
- **注意：双重链表中一个结点的前驱并不一定是它邻近的块**



处理手工存储管理

■ 两大问题

- 内存泄露：未能删除不可能再被引用的数据
- 悬空指针引用：引用已被删除的数据

■ 其他问题

- 空指针访问/数组越界访问

■ 解决方法

- 自动存储管理
- 正确的编程模式

正确的编程模式（1）

• 对象所有者（Object ownership）

- 每个对象总是有且只有一个所有者（指向此对象的指针）；只有通过Owner才能够删除这个对象
- 当Owner消亡时，这个对象要么也被删除，要么已经被传递给另一个owner
 - 语句`v=new ClassA`；创建的对象的所有者为v
 - 即将对v进行赋值的时刻（v的值即将消亡）
 - 要么v已经不是它所指对象的所有者；比如`g=v`可以把v的ownership传递给g
 - 要么需要在返回/赋值之前，执行`delete v`操作
- 编程时需要了解各个指针在不同时刻是否owner
- 防止内存泄漏，避免多次删除对象。不能解决悬空指针问题

正确的编程模式（2）

- **引用计数**

- 每个动态分配的对象附上一个计数：记录有多少个指针指向这个对象
- 在赋值/返回/参数传递时维护引用计数的一致性
- 在计数变成0之时删除这个对象
- 可以解决悬空指针问题；但是在递归数据结构中仍然可能引起内存泄漏
- 需要较大的运行时刻开销

- **基于区域的分配**

- 将一些生命期相同的对象分配在同一个区域中
- 整个区域同时删除



垃圾回收

垃圾回收

- **垃圾**

- 狭义：不能被引用（不可达）的数据
- 广义：不需要再被引用的数据

- **垃圾回收：自动回收不可达数据的机制，解除了程序员的负担**

- **使用的语言**

- Java、Perl、ML、Modula-3、Prolog、Smalltalk

垃圾回收器的设计目标

- **基本要求:**

- 语言必须是类型安全的: 保证回收器能够知道数据元素是否为一个指向某内存块的指针
- 类型不安全的语言: C, C++

- **性能目标**

- 总体运行时间: 不显著增加应用程序的总运行时间
- 空间使用: 最大限度地利用可用内存
- 停顿时间: 当垃圾回收机制启动时, 可能引起应用程序的停顿。这个停顿应该比较短
- 程序局部性: 改善空间局部性和时间局部性

可达性

- 直观地讲，可达性就是指一个存储块可以被程序访问到
- 根集：不需要指针解引用就可以直接访问的数据
 - Java：静态成员、栈中变量
- 可达性
 - 根集的成员都是可达的
 - 对于任意一个对象，如果指向它的一个指针被保存在可达对象的某字段中、或数组元素中，那么这个对象也是可达的
- 性质
 - 一旦一个对象变得不可达，它就不会再变成可达的

垃圾检测的算法

■ 引用计数法

- 给对象一个引用计数器，有引用+1，引用消失-1
- 如果一个对象没有引用值，那么这个对象就是垃圾，需要回收
- **存在问题：循环垃圾**，即如果a、b两个对象相互引用，并没有其他的引用。那么他们的计数器即不为0，也属于需要回收的垃圾

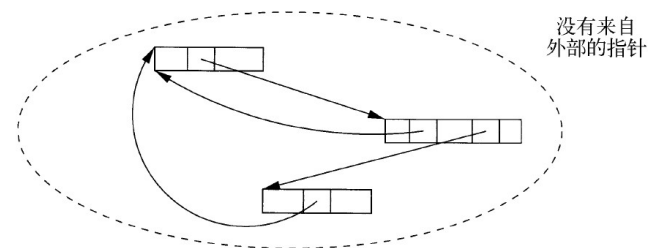


图 7-18 一个不可达的循环数据结构

垃圾检测的算法

■ 可达性分析法

- 以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象
- 这里的根集一般包括java栈中引用的对象、方法区常量池中引用的对象，本地方法中引用的对象等

垃圾回收的算法

■ 标记-清除 (Mark-sweep)

- 标记垃圾对象，统一回收
- 优点：基础算法，简单实现
- 缺点：效率低，产生大量碎片
- 以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象

垃圾回收的算法

■ 复制 (Copying)

- 内存分为平等2分，复制使用对象到另外一块
- 优点:复制成本小、解决了碎片的问题 缺点: 内存需求更多

■ 标记-整理 (Mark-Compact)

- 标记存活对象，清除为标记对象，并压缩存活对象

■ 分代收集算法

- 分代的垃圾回收策略



Thank you!