微机原理与接口技术

姓名：陈致蓬

单位：中南大学自动化学院

电话： 15200328617

Email ： ZP.Chen@csu.edu.cn

Homepage:

https://www.scholarmate.com/psnweb/homepage

QQ ： 315566683

第 7 章　内存（三）堆原理

7.1　数据结构堆概述

7.2　内存堆概述

7.3　内存管理

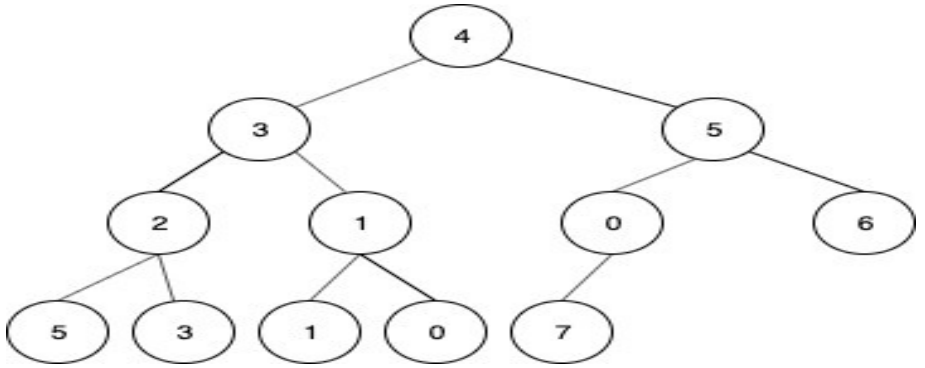
7.4　GC 回收

7.1 数据结构堆概述
7.1.1 堆的定义

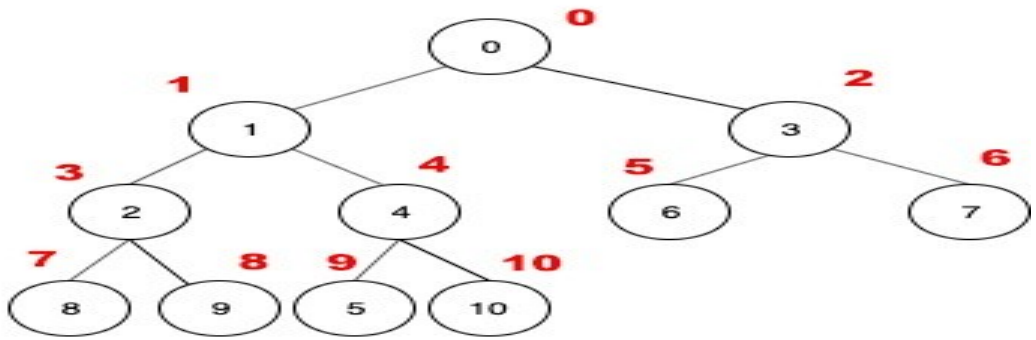
堆是一种可以被看做一棵完全二叉树的数组对象，或被视为优先队列，它满足以下性质：

1、堆按照每个元素的优先级来排序，优先级高的在堆顶，称为堆序性质

2、堆总是一棵完全二叉树，即除了最后一层，其他层的节点数都达到最大，且最后一层的节点都靠左排列

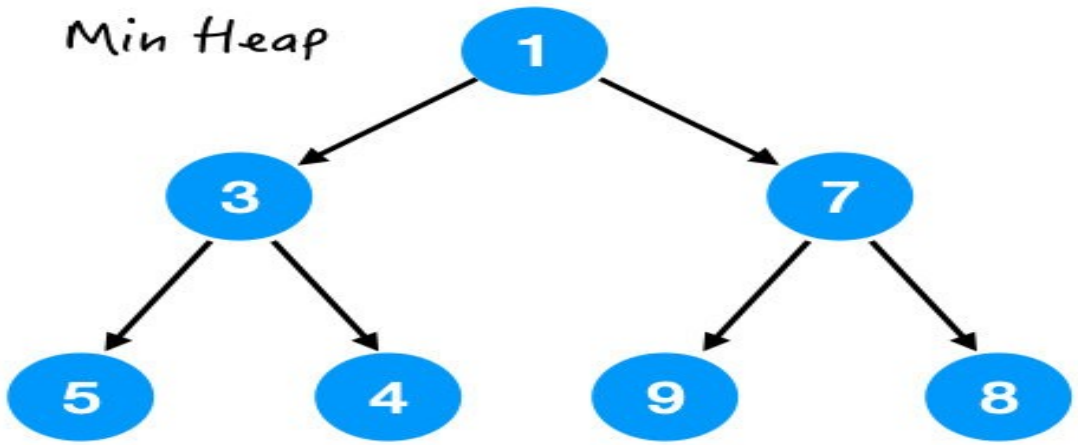


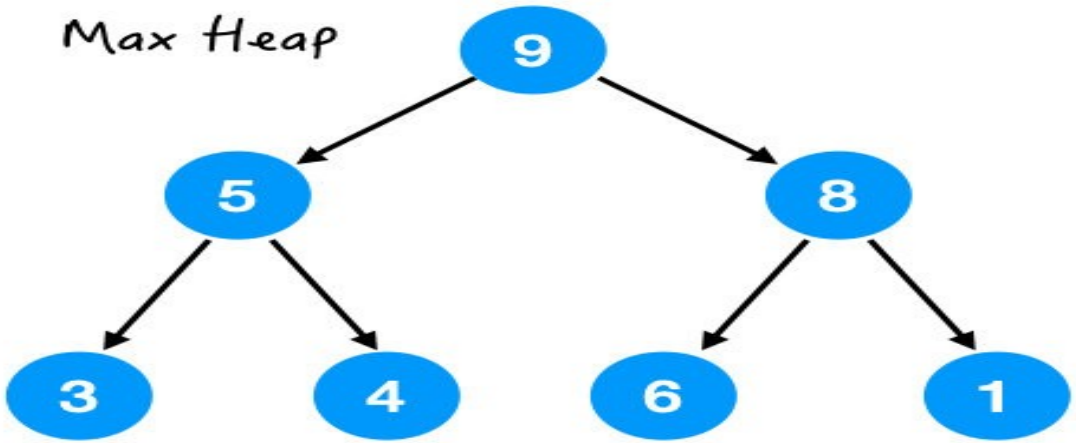
Min-Heap



7.1.2 堆的性质

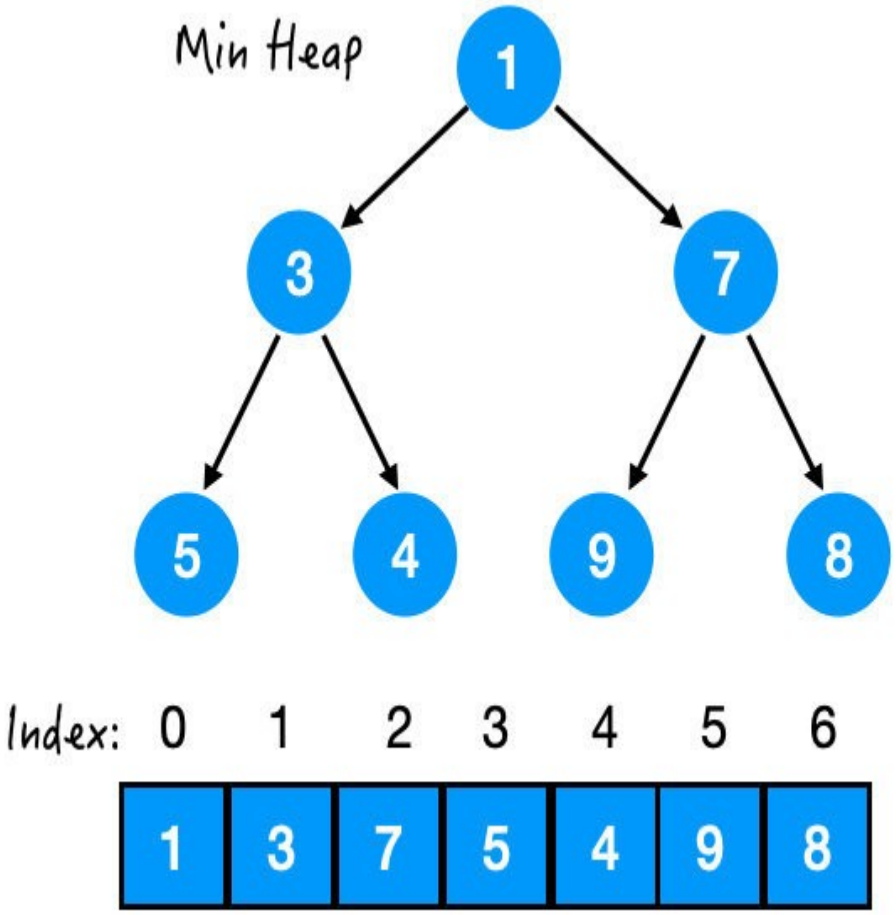
对于堆的优先级，可以按照数据大小划分，从而将堆分为最大堆和最小堆：
最大堆：值越大优先级越大，也就是说父节点的值总是大于或等于子节点的值
最小堆：值越小优先级越大，也就是说父节点的值总是小于或等于子节点的值





7.1.3 堆的定位

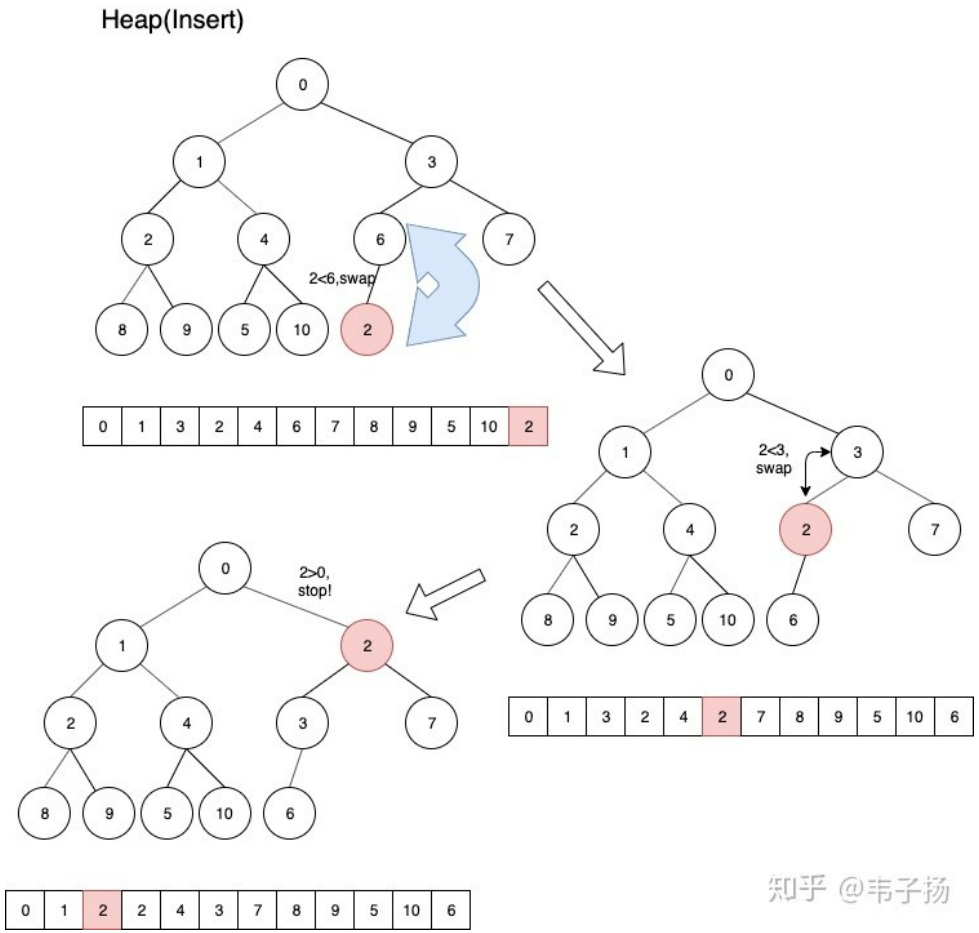
既然堆是用数组来实现的，那么我们可以找到每个节点和它的父母 / 孩子之间的关系，从而可以直接访问到它们。比如对于节点 3 来说，
它的索引 = 1 ，
父母的索引 = 0,
左孩子索引 = 3,
右孩子索引 = 4.
设当前节点的索引 = x ，
父母的索引 = (x-1)/2,
左孩子索引 = 2*x + 1,
右孩子索引 = 2*x + 2.



7.1.4 堆的插入

插入是堆的基本操作：

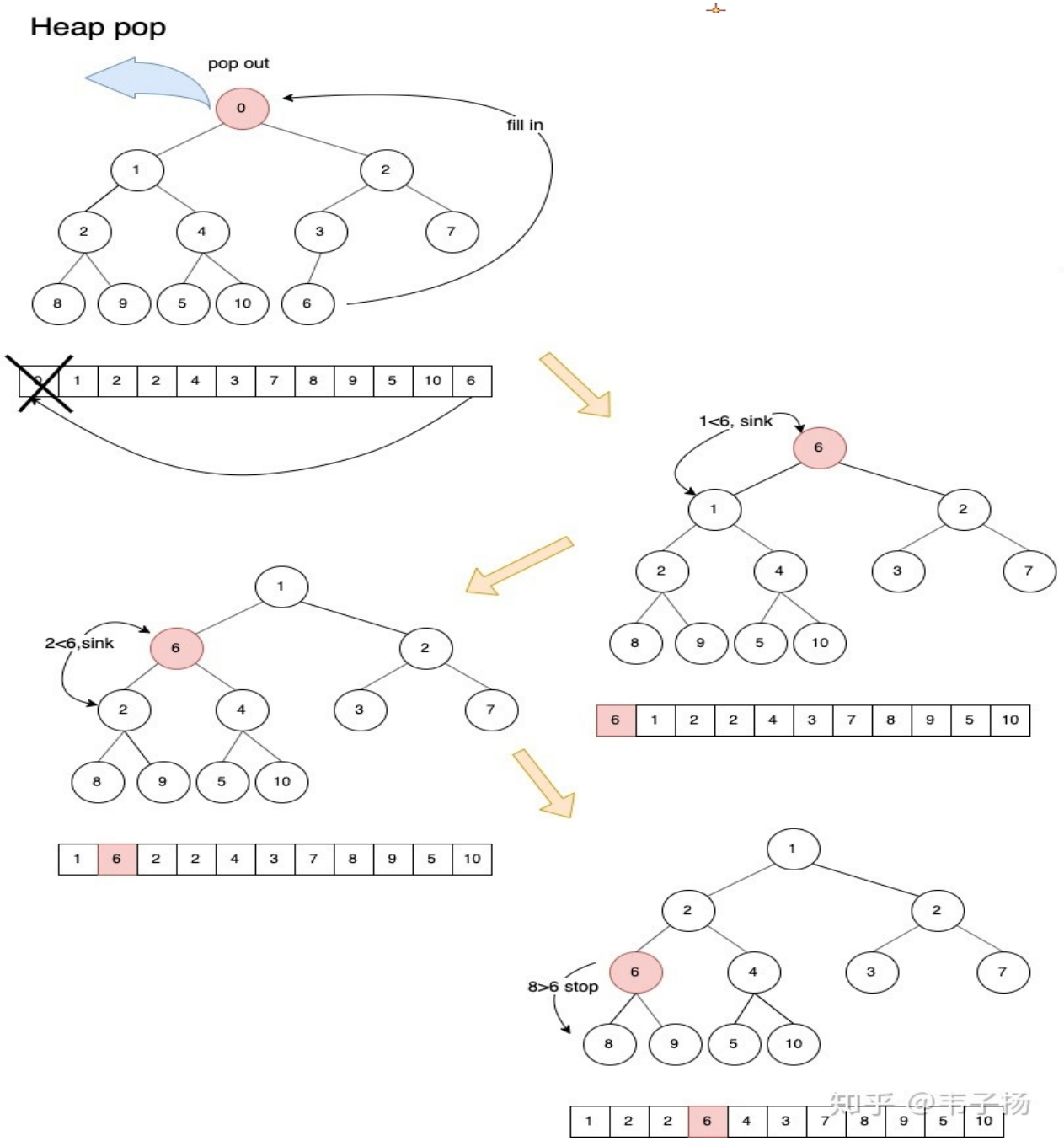
插入：将新元素放在堆的最后一个位置，然后从下往上与父节点比较，如果破坏了堆性质，就交换它们的位置，直到满足堆性质或到达根节点



7.1.5 堆的删除

删除是堆的基本操作：

删除：将堆顶元素（最大或最小）删除，并用最后一个元素替换，然后从上往下与子节点比较，如果破坏了堆性质，就交换它们的位置，直到满足堆性质或到达叶节点



7.2 内存堆概述
7.2.1 堆定义

内存中的堆是一种动态分配和释放的内存空间，由程序员手动申请和管理，大小和位置不固定，可以灵活地分配和回收。内存中的堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址

7.2.2 程序内存分配

一个程序占用的内存分为以下几个部分

1、栈区（stack）：由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。

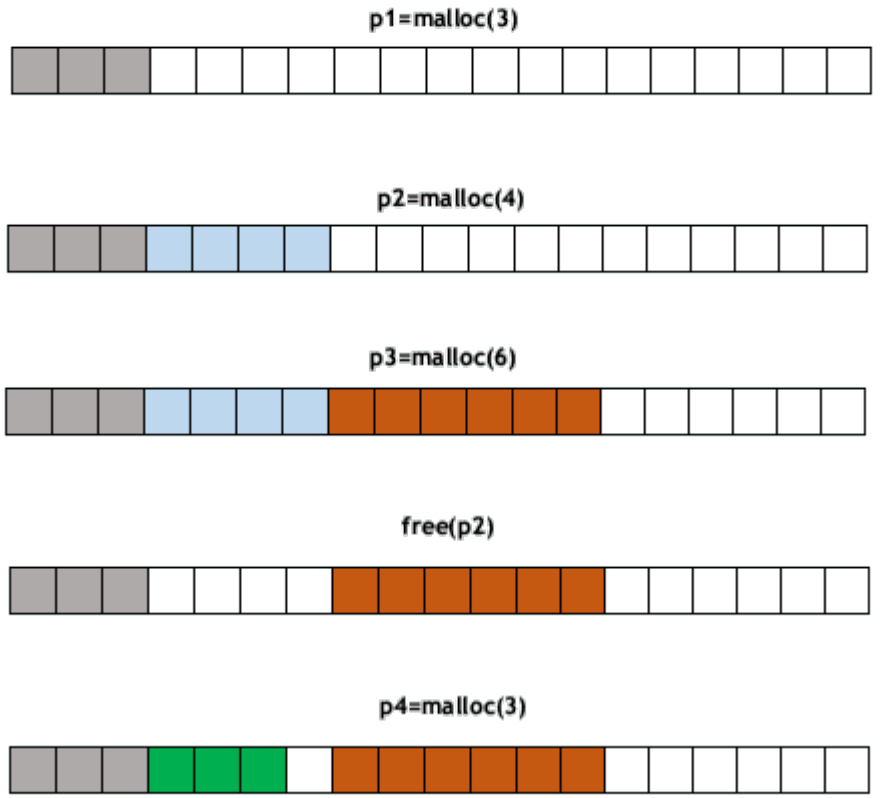
3、全局区（静态区）（static）：全局变量和静态变量的存储

4、文字常量区：常量字符串就是放在这里的。程序结束后由系统释放

5、程序代码区：存放函数体的二进制代码。

7.2.3 堆存取示意

- 堆的申请在 C 语言中用 malloc 函数，在 C++ 中用 new 运算符。
- 堆是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的。
- 堆中的具体内容由程序员安排。每次分配内存，系统会返回一个指向该内存的指针。每次释放内存，系统会将释放指针所指向的内存空间归还给空闲链表。



7.2.4 代码示例

```
#include "stdio.h"
int a = 0;// 全局初始化区
 char *p1;// 全局未初始化区
void main(void)
{ int b;// 栈
 char s[] = "abc";// 栈
 char *p2;// 栈
char *p3 = "123456";// 123456/0 在常量区，p3 在栈上。
 static int c =0;// 全局（静态）初始化区
p1 = (char *)malloc(10); p2 = (char *)malloc(20); // 分配得来得 10 和 20 字节的区域就在堆区。
 strcpy(p1, "123456");// 123456/0 放在常量区，编译器可能会将它与 p3 所指向的"123456" 优化成一个地方。 }
```

7.2.5 堆栈存储差异

存储内容不同：栈存放的是函数的参数值，局部变量的值等，堆存放的是程序员动态分配的内存空间。

堆存储的是数组和对象（头字节存放大小，剩余字节存放数据），凡是 new 建立的都是在堆中。堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性）。当一个数据消失时，整个实体仍然保留，所以堆不会随时释放，但栈不同，栈里存放的都是单个变量，变量被释放即消失。堆里的实体虽然不会被释放，但是会被当成垃圾。

7.2.6 堆栈申请差异

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：操作系统存在记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点的空间分配给程序。由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的部分重新放入空闲链表中。由于这一特性，堆的大小受限于计算机系统中有效的虚拟内存

7.2.7 堆栈总体差异

- 存储内容不同：栈存放的是函数的参数值，局部变量的值等，堆存放的是程序员动态分配的内存空间。
- 管理方式不同：栈由系统自动分配和释放，堆由程序员手动申请和释放。
- 空间大小不同：栈的空间大小是有限的，一般在 Windows 下是 1M 或 2M ，堆的空间大小是相对无限的，受限于系统的有效虚拟内存。
- 能否产生碎片不同：栈不会产生碎片，空间连续，堆会产生碎片，空间不连续。
- 生长方向不同：栈向低地址扩展，堆向高地址扩展。
- 分配方式不同：栈有静态分配和动态分配两种方式，堆都是动态分配的。
- 分配效率不同：栈由系统自动分配，速度较快，堆由 new 或 malloc 分配，速度较慢。

7.3 内存管理
7.3.1 内存分析

内存组成可以分为内核空间和用户空间，其中内核空间存放操作系统，由操作系统管理，而用户空间则一般包含以下几个部分：

- 代码段：存放可执行的指令，一般是只读的，可以被多个进程共享。
- 数据段：存放已经初始化的全局变量和静态变量，可以分为只读区和读写区。
- BSS 段：存放未初始化的全局变量和静态变量，由系统初始化为 0 。
- 堆：存放动态分配的内存，由程序员手动申请和释放，向高地址扩展。
- 栈：存放函数调用的上下文信息，包括局部变量，函数参数，返回地址等，由系统自动分配和释放，向低地址扩展。

对于上述段，栈大小通常为 1~2M ，可以忽略，代码段、数据段和 BSS 段通常较小，因此内存管理的主要内容是堆内存的管理

7.3.2 内存分配前提

从应用程序的角度来看，对堆内存接口会产生如下两个事实：

- 会随机发无序的 malloc 请求或 free 请求。
- free 请求必须根据之前 malloc 请求返回的指针发出。

从堆内存分配的角度来看，分配器的内存分配受到以下客观因素影响：

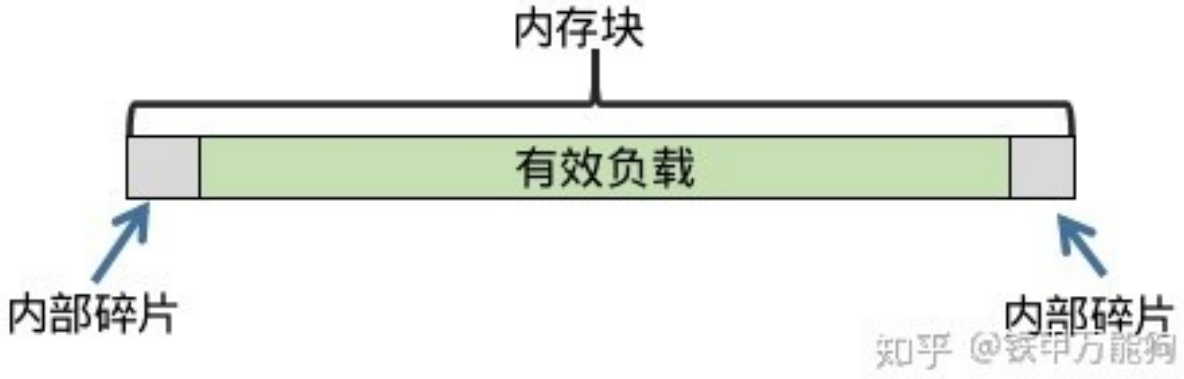
- 无法控制已分配区域的尺寸或数量。
- 必须对 malloc 请求立即响应。
- 内存块的分配必须从闲置内存块中分配。
- 一经分配的内存区域无法移动
- 必须满足内存对齐条件

上述条件导致内存管理极为困难

7.3.3 内部碎片

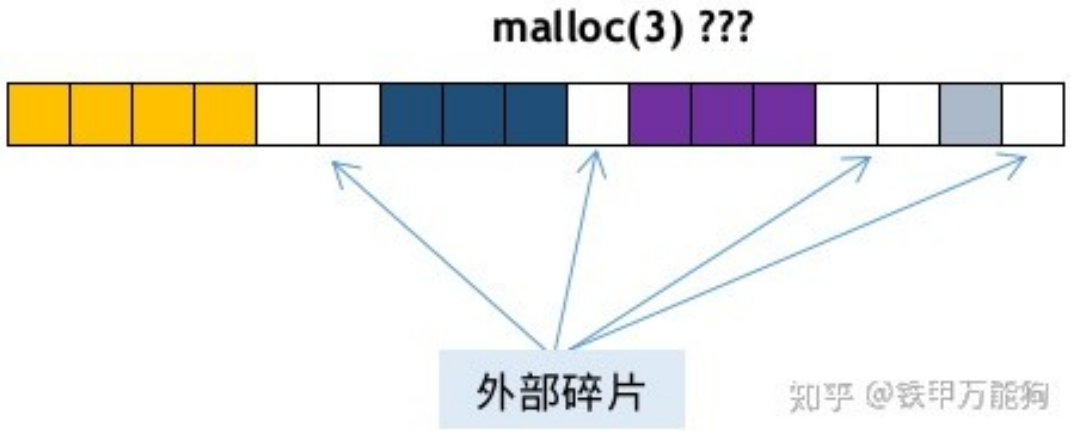
在为程序分配堆内存时，通常会分配大于程序所需的内存，从而产生不会被使用的内存，这种内存碎片被称为**内部碎片**。内部碎片产生原因有：

- 用于维护堆基本数据结构产生的开销。
- 作为内存对齐操作的额外填充位置。
- 内存块的预缓存策略，例如为程序的小尺寸的内存块申请预留更多已分配但尚未使用的内存块。



7.3.4 外部碎片

在为程序分配堆内存时，会生成或减少空闲内存块，当空闲内存块过小时，由于太小无法分配，其将很难被利用，对于这种内存块，我们称之为外部碎片



7.3.5 内存管理需求

malloc 分配器的实现必须遵循严格的规则，并且其实现必须考虑两个性能指标

- **最大的数据吞吐量**：这意味着在应用程序给定一系列 malloc 请求以及 free 请求后，最大程度地提高数据**吞吐量**。
- **内存利用的峰值**：应当尽可能的利用闲置内存，最大限度地减少内、外部碎片的产生

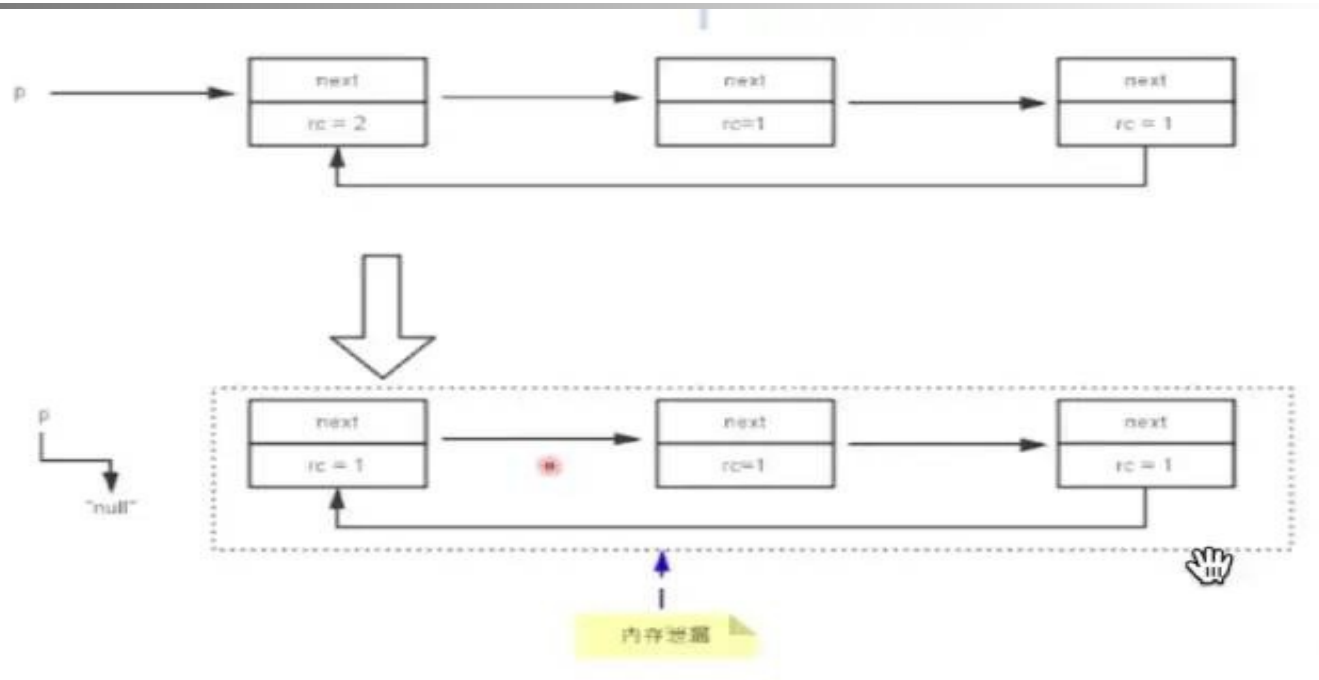
7.4 GC 回收
7.4.1 GC 定义

我们在程序中定义一个变量，会在内存中开辟相应内存空间进行存储，当不需要此变量后，需要手动销毁此对象，并释放内存。而这种对不再使用的内存资源进行自动管理、回收的功能即为**垃圾回收（Garbage Collection，缩写为 GC）**，是一种**自动内存管理机制。**

7.4.2 垃圾识别 - 引用计数

为实现垃圾回收，应当首先识别哪些数据是垃圾，而**引用回收算法(reference counting)**就是常用的算法。

引用计数通过在对象上增加自己被引用的次数，被其他对象**引用时加1**，引用自己的对象被**回收时减1**，引用数为0的对象即为可以被回收的对象，这种算法在内存比较紧张和实时性比较高的系统中使用比较广泛，如php，Python等。



优点：

1.方式简单，回收速度快。

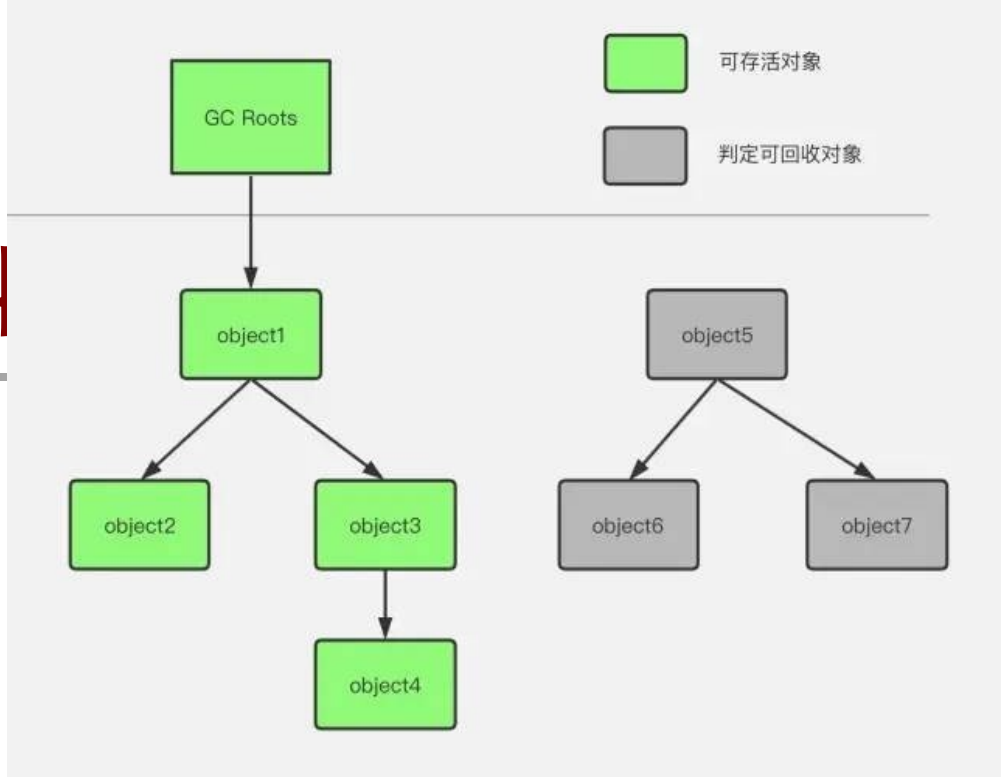
缺点：

2.需要额外的空间存放计数。

3.无法处理循环引用(如a.b=b; b.a=a)。

4.频繁更新引用计数降低了性能。

7.4.3 垃圾识别 - 追踪式回收

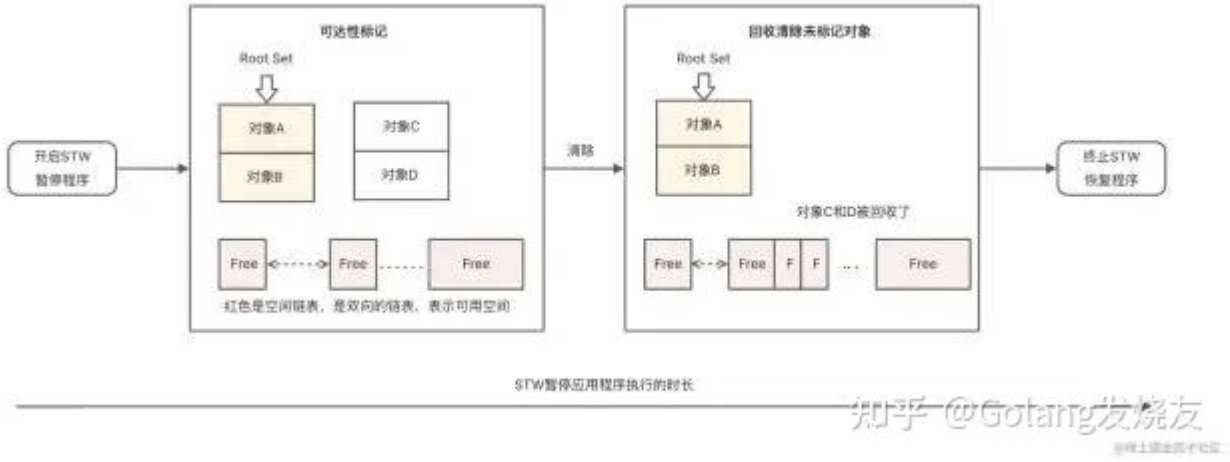


追踪式算法（可达性分析）的核心思想是判断一个对象是否可达，如果这个对象一旦不可达就可以立刻被 GC 回收了，
第一步从根节点开始找出所有的全局变量和当前函数栈里的变量，标记为可达。第二步，从已经标记的数据开始，进一步标记它们可访问的变量，以此类推，专业术语叫传递闭包。当追踪结束时，没有被打上标记的对象就被判定是不可触达。

优点：
1.解决了循环引用的问题
2.占用的空间少了
缺点：
3.无法立刻识别出垃圾对象，需要依赖 GC 线程
4.算法在标记时必须暂停整个程序，即 STW(stop the world)，否则其他线程有可能会修改对象的状态从而回收不该回收的对象

7.4.4 标记清除算法 (Mark Sweep)

标记清除算法是最常见的垃圾收集算法，标记清除收集器是跟踪式垃圾收集器，其执行过程可以分成标记 (Mark) 和清除 (Sweep) 两个阶段：
标记阶段：暂停应用程序的执行，从根对象触发查找并标记堆中所有存活的对象；
清除阶段：遍历堆中的全部对象，回收未被标记的垃圾对象并将回收的内存加入空闲链表，恢复应用程序的执行；



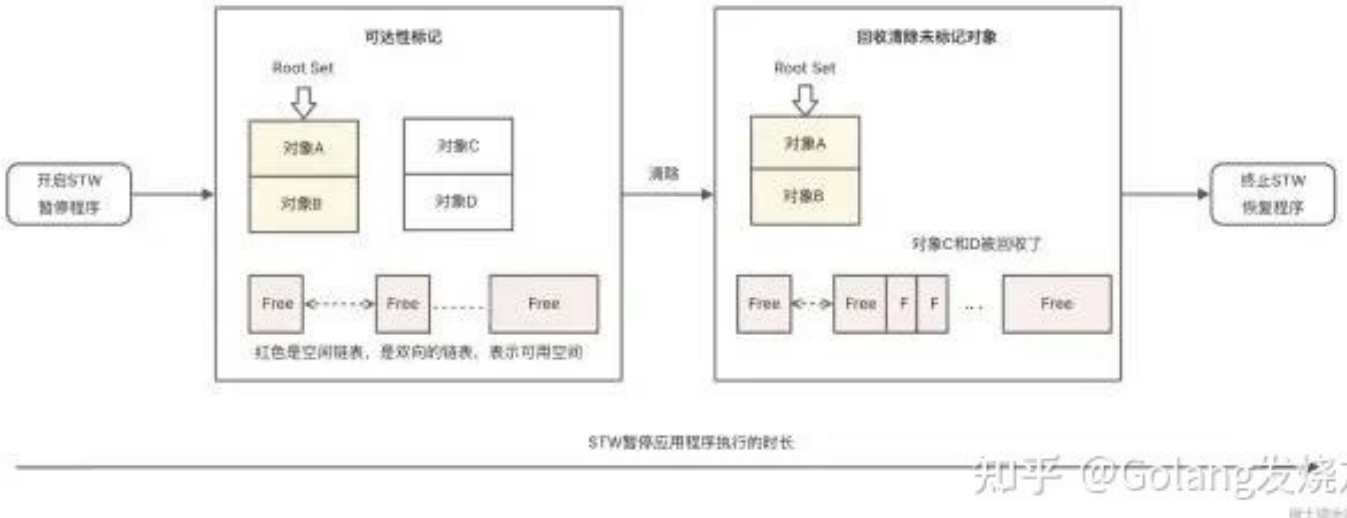
优点：
实现简单。
缺点：
执行期间需要把整个程序完全暂停，不能异步的进行垃圾回收。
容易产生大量不连续的内存碎片，从而提前触发新的一次垃圾收集动作。

7.4.5 标记复制算法

它把内存空间划分为两个相等的区域，每次只使用其中一个区域。在垃圾收集时，遍历当前使用的区域，把存活对象复制到另一个区域中，最后将当前使用的区域的可回收对象进行回收。

1.首先这个算法会把对分成两块，一块是 From 、一块是 To

2.对象只会在 From 上生成，发生 GC 之后会找到所有的存活对象，然后将其复制到 To 区，然后整体回收 From 区。



优点

1.不用进行二次扫描

2.解决了内存碎片问题

缺点：

3.复制成本问题：在可达对象占用内存高的时候，复制成本会很高。

4.内存利用率低：相当于可利用的内存仅有一半。

7.4.6 标记压缩算法

在标记可回收的对象后将所有存活的对象压缩到内存的一端，使他们紧凑地排列在一起，然后对边界以外的内存进行回收，回收后，已用和未用的内存都各自一边。

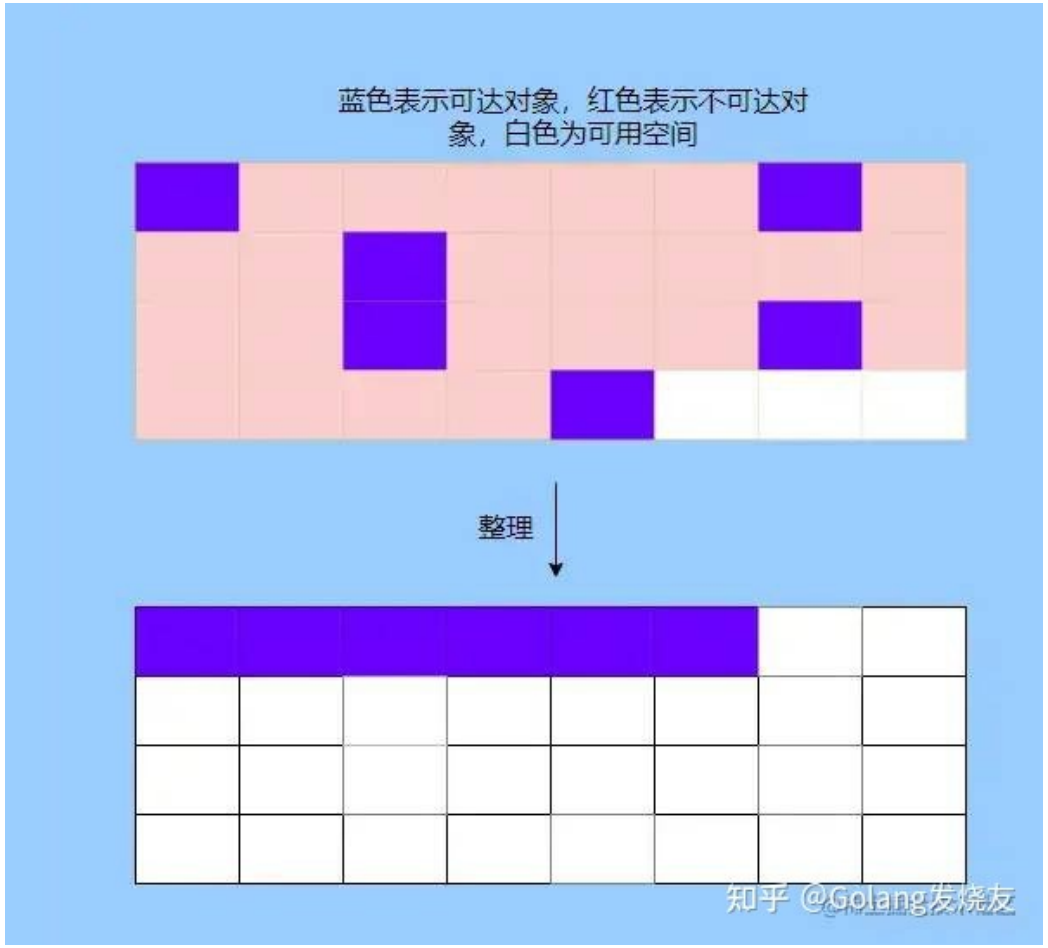
优点：

1.避免了内存碎片化的问题。

2.适合老年代算法，对象存活率高的情况下，不需要复制对象，效率更高。

缺点：

3.需要多次遍历内存，导致 STW 时间比标记清除算法高。



7.4.6 GC 回收流程

JAVA 中　GC 回收的四个阶段可以介绍如下：

•**初始标记**：这个阶段会暂停所有的用户线程，标记出所有直接可达的对象，也就是根集合中的对象，然后恢复用户线程的执行。

•**并发标记**：这个阶段会和用户线程并发执行，从初始标记的对象开始，遍历它们的引用，标记出所有间接可达的对象，也就是存活的对象。这个阶段需要使用写屏障（write barrier）来处理并发过程中对象引用关系的变化。

•**最终标记**：这个阶段会再次暂停所有的用户线程，处理并发标记期间因为用户线程继续运行而导致的标记遗漏的对象。这个阶段一般会很快完成，因为只需要处理少量的对象。

•**筛选回收**：这个阶段会根据每个分区（region）的回收价值和成本，计算出一个回收优先级，然后根据用户设定的 GC 停顿时间来制定一个回收计划，回收那些价值高、成本低的分区。这个阶段也是和用户线程并发执行的。

7.4.7 GC 触发时机

当满足触发垃圾收集的基本条件：允许垃圾收集、程序没有崩溃并且没有处于垃圾循环后，在下述情况触发 GC ：

- 超过内存大小阙值，分配内存时，当前已分配内存与上一次 GC 结束时存活对象的内存达到某个比例时就触发 GC 。（默认配置会在堆内存达到上一次垃圾收集的 2 倍时，触发新一轮的垃圾收集）；
- 如果一直达不到内存大小的阙值，检测出一段时间内没有触发过 GC （默认为 2 分钟），就会触发新的 GC 。
- 调用接口强制触发 GC

谢谢大家！