参考资料

Java 虚拟机规范(Java SE 8 版)

深入理解 Java 虚拟机——JVM 高级特性与最佳实践 第三版

实战 Java 虚拟机——JVM 故障诊断与性能优化

垃圾回收的算法与实现

垃圾回收算法手册

深入理解计算机系统

Stackoverflow:

https://stackoverflow.com/questions/16549066/java-major-and-minor-garbage-collections

OpenJdk:

https://openjdk.java.net/jeps/122

http://cr.openjdk.java.net/~sundar/8022483/webrev.01/raw_files/new/src/share/classes/com/sun/tools/hat/resources/oqlhelp.html

Oracle 官方 DOC:

https://docs.oracle.com/javase/8/javase-books.htm

https://docs.oracle.com/en/java/javase/14/

https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf

https://docs.oracle.com/javase/specs/index.html

https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

资料整理说明:

1、参考《深入理解 Java 虚拟机——JVM 高级特性与最佳实践》第三版

2、《垃圾回收算法手册》，这本书有很多探讨 GC 设计时会遇到的问题，特别干燥，我看过的书，水分最少的之一

3、《垃圾回收的算法与实现》

4、基于 JDK14&Hotspot，但因为 JDK8 是目前主流版本，已被抛弃但 JDK8 中部分 GC 收集器内容还是会做简单整理

5、不局限于 JVM 的 GC 内容，还包括垃圾回收算法基本实现理论

6、这里不过多讨论算法效率问题，但会在后续整理《算法导论》时，深入讨论

7、当前进度：未整理并发情况下的 GC 算法

一、JAVA 中对象

1、对象内存布局（HotSpot）

1.1、概述

a、布局划分为三部分：对象头(Header)、实例数据(Instance)、对齐填充(Padding)

b、对象头包含两类信息：用于存储对象自身的运行时数据(Mark Word)和对象指向它的类型元数据的指针(类型指针)

c、实例数据部分时对象真正存储的有效信息，存储顺序受 JVM 分配策略参数和字段在 java 源码中定义顺序的影响,

d、HotSpot 默认存储顺序：long/double,int,short,char,byte/boolean,oop(Ordinary Object Pointer)

e、对齐填充部分不是必须存在，也没有任何含义，仅仅用于占位符

1.2、对象头(Header)

a、对象自身的运行时数据(Mark Word)

a.1、这部分数据的长度在 32 位和 64 位的未开启压缩指针的 JVM 中分别为 32bit 和 64bit。

a.2、Mark Word 是一个有着动态定义的数据结构，目的在于在极小的空间存储更多的数据，根据对象的状态复用自己的存储空间

a.3、32 位 HotSpot 中，如果对象未被同步锁锁定的状态下，Mark Word 的 32bit 存储空间中，25bit 用于存储对象哈希码，4bit 存储对象分代年龄，2bit 存储锁标志位，1bit 固定为 0。其他状态存储内容如下表

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁	对象的 HashCode		分代年龄	0	01
偏向锁	线程 ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC 标记	空				11

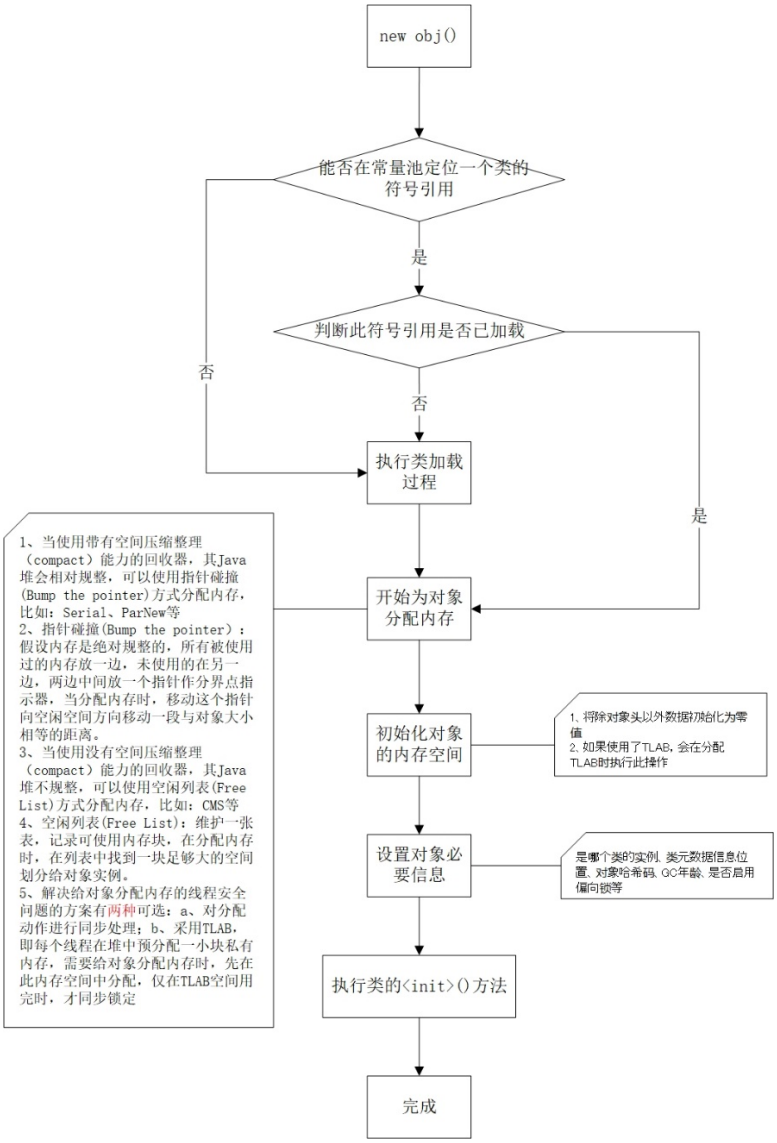
b、类型指针

a.1、JVM 通过这个指针来确定该对象是哪个类型的实例

a.2、并不是所有 JVM 实现都必须在对象数据上保留类型指针。

a.3、如果对象是数组，对象头中还必须有一块用于记录数组长度的数据

2、New 对象()



3、对象查找

3.1、概述

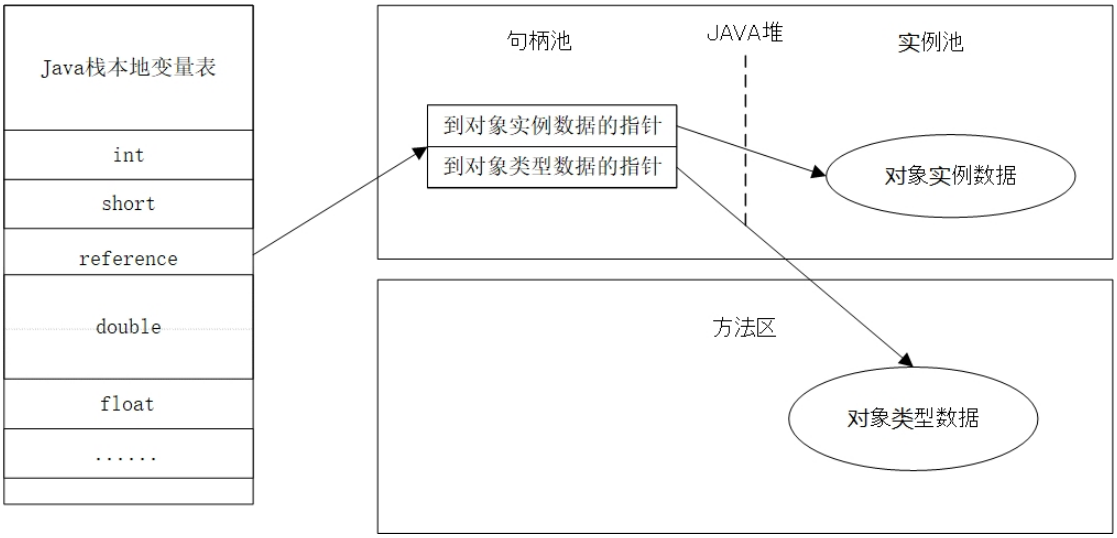
a、reference 类型在 JVM 规范中只规定了它是一个指向对象的引用，并没有定义这个引用应该通过什么方式去定位、访问到堆中对象的具体位置

b、主流访问方式有：使用句柄和直接指针

3.2、通过句柄访问

c、JVM 将在队中划分除一块内存作为句柄池，reference 中存储对象的句柄地址，而句柄包含了对象实例和类型数据各自的地址信息

d、优势是：reference 存储的是稳定句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而本身不用被修改

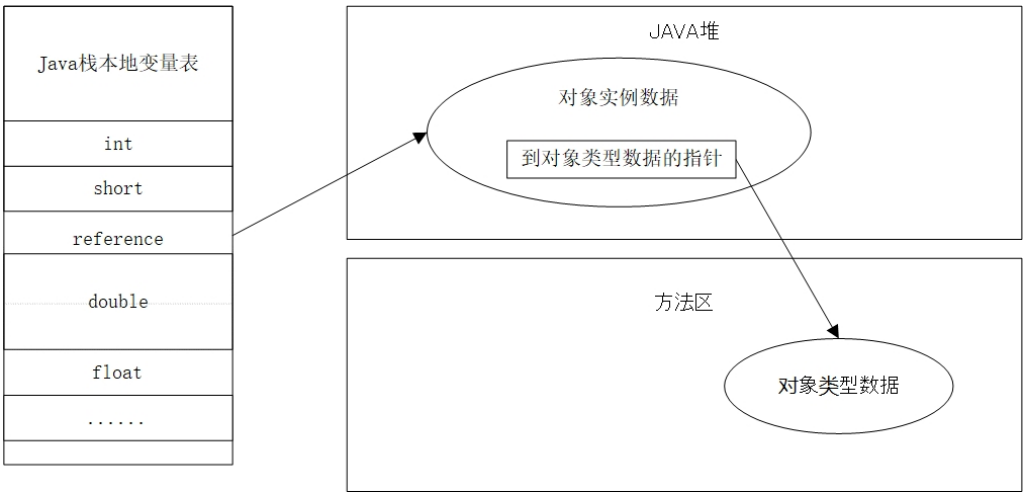


3.3、通过直接指针访问

a、堆中的内存布局必须考虑如何放置访问类型数据的相关信息，reference 存储的就是对象地址

b、访问对象本身，不需要多一次间接访问开销，相对句柄，速度块，节省了一次指针定位的时间开销

c、Hotspot 主要使用此方法访问



二、内存分配

1、顺序分配(Sequential allocation)

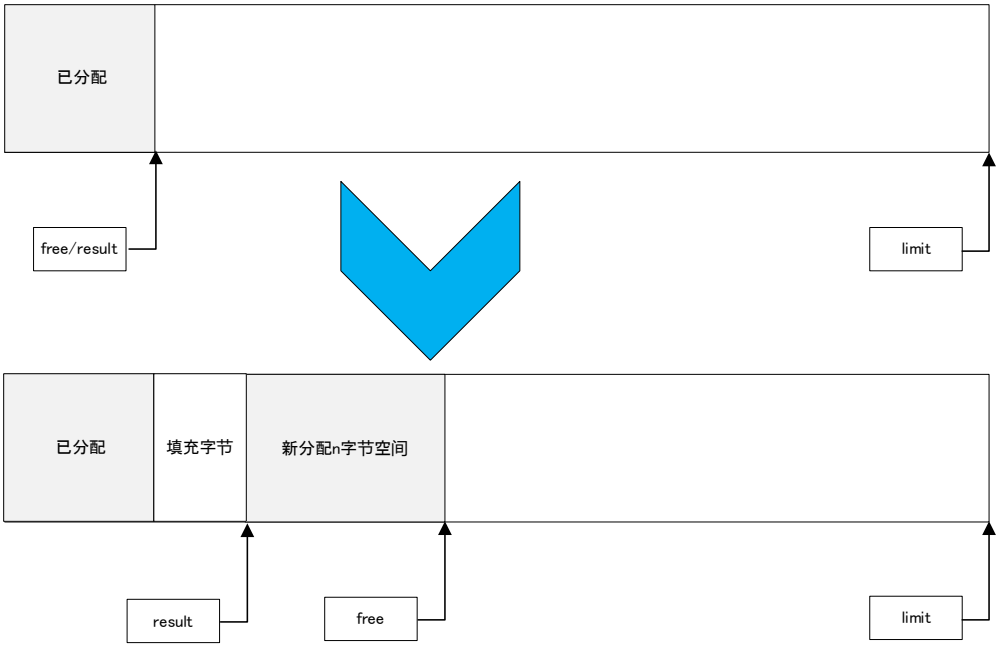
1.1、概述

a、又称为指针碰撞(bump pointer allocation)，或称为线性分配(linear allocation)

b、不适用于非移动式回收器

c、相对于空闲链表分配，顺序分配可以有更好的高速缓存局部性，尤其是移动式回收器(moving collectors)的初始化分配

d、不适用于非移动式回收器(non-moving collectors)



1.2、代码示例

```
sequentialAllocate(n):
        //开始分配，result 和 free 在统一起点，如果需要字节对齐，可能需要增加几个字节对齐
        result <— free
        //result 增加指定空间，作为作为新起始地址
        newFree <— result + n
        //判断 newFree 是否超过了界限指针
        if newFree > limit
            //如果超过了，则返回空，分配失败
            return null
        //则更新 free 指针
        free <— newFree
        return result
```

2、空闲链表

2.1、概述

a、使用某种数据结构(不一定是链表)记录空闲内存单元的位置和大小
b、分配策略一般是顺序适应分配(sequential fits allocation):顺序扫描所有空闲内存单元，知道发现第一个符合条件的内存单元位置
c、典型顺序适应分配策略：首次适应分配(first-fit)、循环首次适应(next-fit)，最佳适应(best-fit)
d、与顺序分配的代价差异十分小，主要差异在于两个分配策略对局部性改进程度的二阶效应决定的

2.2、首次适应分配(first-fit allocation)

a、遍历链表，将发现的第一个满足分配要求的内存单元中进行分配，如果内存单元的空间大于分配空间，在符合一定条件下会执行分裂(split)单元空间操作
b、如果内存单元的空间大于分配空间，但满足以下条件，则不进行分裂操作：
b.1、分裂以后剩余空间小于算法及数据结构规定的最小可分配内存单元的大小
b.2、分裂后剩余空间小于指定阈值或百分比
c、通常靠近空闲链表头部有小碎片
d、首次分配基本代码示例

```
//首次适应分配
//假设每个单元本身都记录了自身大小和下一个空闲内存单元的地址
firstFitAllocate(n):
    //获取头对象
    prev <- addressOf(head)
    //死循环
    loop
        //获取下个单元地址
        curr <— next(prev)
        if curr = null
            //如果当前单元为空，则分配失败
            return null
        else if size(curr) < n
            //如果当前单元小于所需空间，则继续遍历下个单元
            prev <- curr
        else
            //如果当前单元符合所需空间，则开始分配内存
            return listAllocate(prev, curr, n)

//策略 1
//prev：上个单元地址，curr：当前单元空间地址，n：所需空间大小
listAllocate(prev, curr, n):
    //获取当前单元起始地址
    result <— curr
    //判断是否需要分裂判断
    if shouldSplit(size(curr), n)
        //如果需要分裂单元
        //获取剩余部分起始地址
        remainder <- result + n
        //更新 remainder 下个对象起始地址
        next(remainder) <— next(curr)
        //更新 remainder 大小
        size(remainder) <- size(curr) — n
        //将 prev 单元的下个对象地址指向 remainder
        next(prev) <- remainder
    else
        //如果不需要分裂单元，则更新链表中单元指向，将已分配的空间抛离链表
        next(prev) <- next(curr)
    return result

//策略 2
//将单元的尾部分割出来
//return the portion at the end of the cell being split
//该方案不足之处在于对象对齐方式有所不同
//A possible disadvantage of this approach is the different alignment of objects
listAllocateAlt(prev, curr, n):
    //判断是否需要分裂
    if shouldSplit(size(curr), n)
        //当前单元减去所需大小，重新设定当前单元大小
        size(curr) <— size(curr) — n;
        //将当前单元加上新的单元大小，获取分配空间的起始地址
        result <— curr + size(curr)
    else
        //如果不需要分裂则当前单元即分配单元
        next(prev) <- next(curr)
        result <— curr
        //返回分配单元
    return result
```

2.3、循环首次适应分配(Next-fit allocation)

a、每次查找都从上次分配的位置开始查找
b、如果到达表尾，则从头开始找
c、此分配策略存在如下不足：
c.1、在空间上相邻的存活对象很可能不是同一时段分配的，因此回收的时间也不同，加剧了碎片化
c.2、因为单元不断向前迭代，新分配的空间很可能不是刚释放的空间，因此空间局部性比较差(Accesses through the roving pointer have poor locality because the pointer cycles through all the free cells.)
c.3、分配器的局部性也会因为同一时间分配的对象不在连续的位置上而受影响(The allocated objects may also exhibit poor locality, being spread out through memory and interspersed with objects allocated by previous mutator phases.)
d、代码实例

```
1.  //循环首次适应分配
2.  nextFitAllocate(n):
3.          //获取上次单元地址
4.          start  <—  prev
5.          //死循环
6.          loop
7.              //获取下个单元地址
8.              curr  <—  next(prev)
9.              //判断下个单元是否为空
10.             if  curr  =  null
11.                 //如果下个单元为空，则获取重新定位到头地址
12.                 prev  <—  addressOf  (head)
13.                 //再次获取下个单元地址
14.                 curr  <-  next(prev)
15.             //判断上次单元和开始单元是否符相同
16.             if  prev  =  start
17.                 //如果相同，则说明已经轮训一遍都未找到合适的单元，返回空
18.                 return  null
19.             //判断当前单元是否符合容量需求
20.             else  if  size(curr)  <  n
21.                 //如果不符合容量要求，则进入下个单元
22.                 prev  <—  curr
23.             else
24.                 //如果符合分配需求，则
25.                 return  listAllocate(prev,  curr,  n)
```

2.4、最佳适应分配(best-fit allocation)

a、查找最接近分配需求大小的单元(Best-fit allocation finds the cell whose size most closely matches the request)
b、目标是减少空间浪费和单元分裂
c、会导致堆中散布大量很小的碎片
d、代码示例

```
1.  //最佳适应分配
2.  bestFitAllocate(n):
3.          //初始化单元最佳位置
4.          best  <-  null
5.          //初始化最佳单元的大小为无限大
6.          bestSize  <—  oo
7.          //获取头地址
8.          prev  <—  addressOf  (head)
```

```
        loop
            //获取下个单元地址
            curr <— next(prev)
            //判断当前是否为空，单元大小是否符合需求
            if curr = null || size(curr) = n
            //如果当前单元为空或大小正好是需求大小，则进入分配阶段
                //判断当前单元是否为空
                if curr != null
                //如果当前单元非空，则获取最佳单元上个单元地址
                    bestPrev <— prev
                    //当前单元即为最佳单元
                    best <— curr
                else if best = null
                    //如果最佳单元为空，则返回 null，分配失败
                    return null
                //分配内存
                return listAllocate(bestPrev, best, n)
            //判断当前单元是否小于需求或大于最佳大小
            else if size(curr) < n || bestSize < size(curr)
                //如果当前单元小于需求或大于最佳大小，则进入下个循环
                prev <— curr
            else
                //如果如果当前单元大于需求且小于等于最佳大小
                //设置当前单元为最佳单元
                best <— curr
                bestPrev <— prev
                //当前单元大小为最佳大小单元
                bestSize <— size(curr)
```

2.5、链表加速方案

a、采用平衡二叉树组织内存单元，从而可以按空间大小或地址顺序进行排序。如果按节点大小排序，可将相同大小的空闲节点组织成链表来管理

b、对于首次适应分配和循环首次适用分配，有笛卡尔树(Cartesian tree)和位图适应分配(bitmapped-fits allocation)等

c、位图适应分配，适用额外位图记录每个可分配内存颗粒状态，有以下优点：

c.1、通过映射表，分配时仅需根据位图中一个字节计算既可得知对应的 8 个内存颗粒所能构成的最长可连续空间。

c.2、位图本身和对象隔离，因此不容被破坏(They are 'on the side' and thus less vulnerable to corruption)

c.3、不用记录回收信息，从而降低对内存大小的要求(They do not require information to be recorded in the free and allocated cells, and thus minimise constraints on cell size)

c.4、相对于内存单元，位图更紧凑，因此可提高缓存命中率和局部性

2.6、笛卡尔树(Cartesian tree)

a、适用于优化首次适应分配和循环首次适用分配方案的平衡树

b、依照节点地址排序，同时也将节点按大小组织成堆，从而满足快速查找

c、树节点内容包括：单元地址和大小，左右节点指针，节点自身以及子树的最大空闲单元大小

d、代码示例

```
//笛卡尔树--基于首次适应分配
firstFitAllocateCartesian(n):
        parent <— null
        //从根开始遍历
        curr <— root
```

```
6.          loop
7.              //判断左节点非空且左节点最大单元大小是否满足需求
8.              if  left(curr)  !=  null  &&  max(left(curr))  >=  n
9.                  //如果左节点非空且最大单元大小满足需求，则从左节点遍历
10.                 parent  <—  curr
11.                 curr  <—  left(curr)
12.             else  if  prev  <  curr  &&  size(curr)  >=  n
13.                 //如果当前节点是在上个扫描到节点之后且符合需求大小
14.                 prev  <—  curr
15.                 //分配资源，并增删树
16.                 return  treeAllocate(curr,  parent,  n)
17.             else  if  right(curr)  !=  null  &&  max(right(curr))  >  n
18.                     //如果右节点非空且最大单元大小满足需求，则从右节点遍历
19.                     parent  <—  curr
20.                     curr  <—  right(curr)
21.             else
22.                     //否则 OOM
23.                     return  null
```

3、分区适应分配(Segregated-fits allocation)

3.1、概述

a、基本思想：将相同(逻辑)空间可分配单元大小限制为 k 种，其中 s0<s1<....<s_{k-1}。
(when we speak of segregated-fits we mean multiple lists being used for allocating for the same (logical)space)

3.2、空间大小分级填充(Populating size classes)

a、基于内存卡页簇分配(Big bag of pages block-based allocation)
b、基于内存块分裂(Splitting)

3.3、分区适应分配与空闲链表分配的配合(combining segregated-fits with first-, best- and next-fit)

4、内存分配考虑点

4.1、内存碎片(Fragmentation)

a、对于一次分配请求，尽管空间足够，但所有空闲单元都无法满足分配需求。
b、会导致程序消耗更多的地址空间、更多的常驻内存页以及更多的高速缓存行
c、无法根除碎片化问题
c.1、分配器无法预测程序内存分配序列
c.2、即使可以预测内存分配序列，找出一种最优分配策略也是很困难
d、唯一解决内存碎片方案是使用整理(compaction)式/复制(copying)式垃圾回收
e、外部碎片(external fragmentation):在已分配空间之外的不可用空间
f、内部碎片(internal fragmentation):由于引入空间大小分级后，在已分配的内存单元内部就可能存在空间的浪费，所造成的碎片

4.2、字节对齐

a、要求字节对齐
a.1、底层硬件或机器指令的要求

a.2、有助于提升各个层次存储器的性能
b、代码示例

```
/**
*结合字节对其要求进行内存分配
*n 所需空间
*a 对齐方式
*m 2 的整数次幂
*blk 空闲内存
**/
fits(n, a, m, blk):
    //获取对齐还需空间
    z <- blk - a
    //通过位运算 向上取整
    z <- (z+m- l) & ~(m- l)
    //获取对齐后空间大小
    z <- z + a
    //获取需要补偿字节
    pad <- z - blk
    return n + pad <= size(curr)
```

4.3、空间大小限制
a、某些回收器要求对象的大小必须大于某阈值
b、对象必须要有唯一的地址，大小至少一个字节

4.4、边界标签(Boundary tags)
a、用于确保在释放内存时可以将相邻空闲内存单元合并
b、保存了内存单元的大小与状态(空闲/已分配)，可能还记录上个单元的大小
c、当单元空闲时，也可用于保存构建空闲链表的指针

4.5、堆可解析性(heap pasability)
a、堆可解析性：能够顺序逐个遍历堆中的内存
b、并非不可或缺，通常只需要支持单方向的解析
c、基于内存块的分配可以简化内存块单元的解析

4.6、局部性
a、同等条件下，基于地址顺序的空间链表分配可能提升分配器访问内存的局部性
b、同一时刻分配的对象通常也会在同一时间称为垃圾，因此将连续两次分配的对象连续排列或者尽可能靠近排列的启发式方法是有价值的
c、如果分配是通过拆分一个大块内存分配，则下次内存分配应当尽可能使用同一个内存块

4.7、荒野保留(Wilderness preservation)
a、书中称为拓展块保护，但我个人理解下来，荒野保留更恰当
b、堆通常是由一大块连续的地址空间组成，其低地址边界通常与程序的静态代码段或数据区相邻，高地址边界之外的地址空间，通常会保留用作扩展。

c、高地址边界之外的地址空间通常不会映射到虚拟内存中，因此具有一定可扩展性，此区域称为未使用空间(unoccupied territory)或者荒野(wilderness)
d、荒野保护：如果将荒野用于分配内存的最后手段，有助于降低内存碎片、延缓堆的增长，从而节省系统资源

4.8、跨越映射(Crossing maps)
a、反映了堆中每个已对齐的2^k片段内最后一个起始于该片段的对象的地址(This map indicates, for each aligned segment of the heap of size 2k for some suitable k, the address (or offset within the 2k segment) of the last object that begins in that segment.)

三、堆分区

1、分区目的(Why to partition)

1.1、根据移动性分区(Partitioning by mobility)
a、识别哪些对象可以移动，哪些对象不能移动或移动的代价很大
b、异步移动对象还不利于编译器的优化
c、移动一个对象，必须找到对象的引用，以便将对象的引用指向对象的新位置
d、当对象不能被移动，要么这个对象位置固定(pinned),要么保证GC不在对象所在空间执行

1.2、根据对象大小分区(Partitioning by size)
a、将大小超过指定阈值的对象分配到专门的大对象空间(large object space,LOS)
b、因为大对象通常被放置在独立内存页，因此大对象大小至少是页的一半，通过类似“标记-清理”这样不移动对象的回收器来管理
c、将对象放入专属内存页，可以通过转轮(treadmill)回收器或重新映射虚拟内存页的方式实现虚拟“复制”

1.3、根据空间大小分区(Partitioning for space)
a、采用非复制式回收器来管理大对象空间的更深层次因素是，为大对象预留复制空间的代价非常高
b、不同的内存空间用不同的内存管理器
c、生命周期比较长并且不急于处理内存碎片的对象，可以保存在主要非移动操作、偶尔会进行整理的空间中
d、分配频率、死亡率高的对象，可以保存在分配速度较快且回收代价较小的复制式回收管理器管理的空间中

1.4、根据类别进行分区(Partitioning by kind)
a、根据类别隔离对象，可以通过对象地址识别对象类型，无需获取对象某字段值或追踪一个指针。
b、充分利用高速缓存优势，消除了加载更多字段的必要性
c、可以对空间基于地址快速识别，同时将空间和对象属性关联起来，不用将相同属性在每个对象头复制一份
d、当大型压缩位图成为伪指针的一个经常性来源时，保守式垃圾回收器因将这些位图放到一个不被扫描的区域而受益
e、如果将那些不可能成为垃圾环的备选根的天生非循环对象隔离保存，则可回收循环引用垃圾的追踪回收器将因此受益

1.5、为效益分区(Partitioning for yield)

a、充分利用对象统计信息，新分配的数据很可能要么被“敲定”，要么在相对较短时间内被抛弃

b、在 JAVA 中，大多数分配点所创建的对象的寿命符合双峰(bimodal)寿命分布,满足弱分代假说

c、c 如果对象周期的分布是足够偏斜，则反复清理一个或多个对内存的子集而非全部，是非常值得的

d、d 如果回收器允许一些垃圾不被清理，意味着分配新对象时系统的可用空间小于应有的的大小，导致垃圾回收器调用会更频繁。

1.6、为缩短停顿时间分区(Partitioning to reduce pause time)

a、通过限制回收器需要追踪的定罪空间(condemned space)的大小，我们就可以限定需要清理或标记的数量，从而控制 GC 时间，缩短停顿时间。

b、一般情况下，分区回收不能降低最坏情况下的停顿时间

c、如果一个区域中所有对象都不可达，清理时不需要任何追踪工作，直接一并释放空间

1.7、为局部性分区(Partitioning for locality)

a、随着存储层级变得日益复杂，局部性对性能的影响不断增加

b、分代回收器可以使回收器和赋值器的局部性都得到进一步改善

1.8、根据线程分区(Partitioning by thread)

a、如果每次只中止一个线程，并仅回收由该线程分配的且不共享的对象，可以减少同步的代价，进而减少 STW 时间

b、b 回收器需要区分出对象是否被多线程共享。

c、c 通过将属于不同线程的非共享数据隔离，能简化局部数据回收操作。

d、

e、1.9 根据可用性分区(Partitioning by availability)

f、a 部分回收器会根据对象的物理分布采取不同的处理策略

g、b 高速缓存不命中的代价可能是几百个时钟周期，而内存页不命中的代价将是几百万个时钟周期

h、c 物理页的组织方式也是一种堆分区。

i、

j、1.10 根据易变性分区(Partitioning by mutability)

k、a 新创建的对象往往被修改的更频繁

l、

2、如何分区(How to partition)

a、最普通值观的方法是将堆内存分成互不重叠的地址段

a.1、最简单情况下，每个空间通过一对一映射占据连续的堆内存块，对应的虚拟地址空间必须事先保留

a.2、把内存块按 2 的整数次幂地址边界对齐，对象所属空间信息相当于被编码到地址的最高位，并可以通过位移或掩码操作来找到具体位置。

a.3、回收在获知空间特性后可以决定如何处理其中的对象

a.4、很多情况下 OS 因为安全性等原因趋向于把代码段映射到不可预知的地方，这使得预留连续虚拟地址操作变得困难

a.5、连续的地址空间不灵活且可能导致系统中虽有足够空间但虚拟内存被耗尽

b、另一种方法是将空间实现为非连续内存块的地址空间集合，不连续的空间包含几组连续地址空间，每段地址空间又有固定大小的帧组成

b.1、这个方法的一个明显确点就是获取对象空间时，需要查表
c、对象可以通过头部添加若干位来标识所属空间
c.1、即使在回收器不移动对象的情况下，也能根据年龄或线程可达性这样的运行时属性来划分对象
c.2、有助于处理被钉住(pinned)的对象
c.3、运行时的动态分区可能比静态分区更精确，比如静态逃逸分析(escape analysis)仅提供了一个对象是否被共享的保守评估
c.4、动态分隔的一大缺点，会引入更多的写屏障(write barrier)，一旦一个指针更新操作使其所指对象成为潜在共享(potentially shared)对象，则其所指对象和传递闭包(transitive closure)必须标记为共享

3、何时分区(When to partition)

a、分区决策可以静态(statically)完成(在编译期)
b、也可以动态完成：当对象创建时、或垃圾回收时、或赋值器访问对象时
c、与年龄相关的回收器按对象年龄分区，根据对象的年龄增长是否超过特定阈值，动态决定是否将对象提升到下个分代中
d、分区决策也可由分配器完成，最常见的，分配器会根据请求空间大小决定是否将对象分配到大对象空间(large object space)中
d.1、在系统支持的，对程序员可见的显示内存或由编译器推测得到的区域内，分配器或分配器可以将对象放到指定区域中
d.2、除非对象被明确告知可共享，否正有限分配到当前线程子堆中
e、通过对象的类型、代码、一些其他分析，对象所属空间也可被静态划定。
e.1、如果可以提前预知某种类型的对象存在某种公共属性，编译器就可以决定将其放入哪个空间
e.2、分代垃圾回收器通常在为新对象预留的空间中创建对象，然后回收器会将其中一些特定对象直接在较老的分代中分配。
f、如果堆是被并发回收器管理时，对象还可以被赋值器重新划分
g、通过与操作系统协作，运行时系统可以在对象所在页被换入换出时对对象重新划分
h、对象的颜色以及持有对象的新/旧空间都可以被认为时一个空间
i、分配器可以根据一些额外属性来划分对象，比如当对象逃逸出它诞生的线程时，通过写屏障可以把它们从逻辑上隔离

四、分代垃圾回收

1、概述

a、弱分代假说(weak generational hypothesis)：大多数对象都在年轻时死亡(most objects die young)
b、强分代假说(strong generational hypothesis):越老的对象越不容易死亡(even for objects that are not newly-created, younger objects will have a lower survival rate than older ones)
c、主要设计目标时减少回收过程的停顿时间，同时提升空间吞吐量
d、标记/构造(mark/cons)率 ,回收器在一次回收过程中所需处理的数据量与分配器在相邻两次回收之间所分配的数量之间的比例(between the volume of data processed by the collector and the volume allocated for that collection)
e、如果被处理的分代中存活对象比较少，则标记/构造率会降低
f、一个配置合理的 GC，完成一次年轻代回收的时间通常 10ms 级别
g、分代 GC 最多只能改善期望停顿时间,而非最大停顿时间(Generational collection therefore

h、improves only expected pause times, not the worst case)
i、为了能够独立回收某一分代，分代回收器必须在赋值器上维护一个额外的记忆集来记录分代间指针

2、分代间指针(inter-generational pointer)

2.1、概述

a、为达到不扫描整个堆，只扫描正在进行回收的空间中的对象的目的，而额外维护的一个指针集合
b、导致分代间指针出现的情况:
b.1、赋值器在一个老年代对象中写入一个指向年轻代对象的指针
b.2、回收器将一个年轻代对象提升到老年代
c、创建方式：在对象创建时写入、赋值器更新指针槽时写入、将对象移动到其他分代时产生
d、为捕获分代间指针赋值器，必须通过写屏障来进行指针写操作，回收器也必须依赖一个简单的赋值写屏障来对提升过程中出现的分代间指针进行探测
e、引导印象(boot image)中的对象存在于堆之外，且从程序启动之后便一直存在

2.2、记忆集

a、用于记录分代间指针的数据结构(The data structures used to record inter-generational pointers are called remembered sets)
b、记录了从堆中一个空间指向另一个空间的指针来源(并非目标)(Remembered sets record the location of possible sources of pointers from one space of the heap to another)
c、不同的记忆集实现方式在来源地址的记录方面所能达到的精度都不同

3、对象提升

3.1、集体提升(En masse promotion)

a、将回收后存活下来的对象集体提升到下一代
b、回收器不仅无需单独记录每个对象年龄，也无需为除最老代外，每一代预留专门复制保留区

3.2、衰老半区(Aging semispaces)

a、要求对象在得到提升之前必须已经在其所属分代的来源空间和目标空间之间复制多次(This allows objects to be copied between the fromspace and tospace an arbitrary number of times within the generation before they are promoted)
b、在对象头中记录年龄，虽可以避免提升最年轻的对象，但会造成一些额外开销
c、Shaw 的桶组(bucket brigade)策略，即可以进行年龄鉴别又不需要记录对象年龄
d、分阶(steps)和分代(generations)不能混淆
d.1、不同分代的回收频率不同，但同代的所有分阶的回收频率相同
d.2、分代因为老年代的回收通常比年轻代晚，所以需要跨代指针，而分阶不需要
d.3、分阶可以降低写屏障的开销

3.3、存活对象空间和柔性提升(Servivor spaces and flexibility)

a、Ungar(1984)进一步将年轻代划分成一个较大的诞生空间(creation space/eden space)和两个较小的存活对象半区或桶(Ungar [1984] organised the young generation as one large creation space (sometimes called eden) and two smaller buckets or survivor semispaces)
a.1、对象在诞生空间分配，次级回收(minor collection)会将诞生空间中的存活对象提升到目标空间(survivor tospace)

a.2、对于来源空间(survivor fromspace)的对象，次级回收根据其年龄决定是否提升到下一代还是移动到目标空间
a.3、HotSpot 虚拟机就是这套方案
b、Opportunistic 垃圾回收器使用桶组(bucket brigade)系统和一个较小(parsimony)的存活对象空间
b.1、可以在不单独记录和操作对象年龄的前提下实现对象级别的提升
b.2、将年轻代划分为一个诞生空间和两个衰老空间(the young generation is divided into a creation space and a pair of aging spaces)，衰老区用分阶策略
b.3、在诞生空间用一个高水位标记(high water mark)，通过一次地址判断区分较为年轻的对象
b.4、该策略将对象的提升年龄标准限定为最多经历两个次级回收，同时也不必显示记录对象年龄或对其进行分区组织(This scheme limits the promotion age to a maximum of two minor collections, and so does not offer as wide a range of promotion age as those that explicitly store ages in objects or associate them with spaces)

4、对程序行为的自适应(adapting to program behaviour)

4.1、Appel 式垃圾回收(Appel-style garbage collection):

a、年轻代具有收缩能力，可以在对内存总量不变的情况下占用尽可能多的空间
b、将堆分为 3 个区域：年老代、复制保留区(copy reserve)、年轻代。
c、必须保证复制保留区在最差情况下仍可以容纳所有存活对象

4.2、基于反馈的统计分析提升(Demographic feedback-mediated tenurin)

a、对刚提升不久便立即死亡的对象进行控制以减缓停顿时间过长的情况
b、使用一次回收所提升对象的总量来预测下次回收将要提升的对象总量，并据此加减提升率
c、此策略虽然可以控制对象提升率，但无法将年老代降级到年轻代

4.3、Hotspot 的 Ergonomics 机制，1.5 版本引入

a、根据不同用户需求调整个分代大小
b、首先尝试满足最大停顿时间要求，在此基础上尝试提升吞吐量，最后在前两个条件都达到情况下减少程序占用时间
c、优化停顿时间方案：对每次回收停顿时间进行统计分析，找出时间最长的分代，缩小此分代的空间
d、提升吞吐量：增加整个堆或某个分代空间来实现，但增加分代空间也会导致此分代回收时间加长
e、ThruMax 算法提供了一种可交替调整年轻代空间大小以及提升率的协同演化框架

五、对象访问&分配

1、概述

a、对象分配&初始化的三个阶段：
a.1、内存管理器的分配子系统分配一块大小合适的，符合字节对齐要求的内存单元
a.2、系统级初始化(system initialisation)：在对象被用户程序访问之前，其所有域都必须初始化到适当的值(By this we mean the initialisation of fields that must be properly set before the object is usable in any way)
a.3、次级初始化(secondary initialisation)：在对象已经“脱离”分配子系统，并可以潜在被程序的其他部分、线程访问时，进一步设置(或更新)其某些域(By this we mean to set (or update)

fields of the new object after the new object reference has 'escaped' from the allocation subsystem and has become potentially visible to the rest of the program, other threads and so on.)
b、JAVA 的三个阶段：
b.1、阶段 1 和 2 共同完成新对象的方法分派向量、哈希值、同步信息的初始化，同时将所有其他域设置为某一默认值。
b.2、数组的长度域也在阶段 1 和 2 完成初始化
b.3、在阶段 1 和 2，字节码 new 所返回的对象即使满足类型安全要求，但仍为“空白”对象
b.4、阶段 3 中 java 将对象构造函数或者静态初始化程序中的代码，或者在对象创建完成后将某些域设置为非零值(Step 3 in Java happens in code provided inside a constructor or static initialiser, or even afterwards, to set fields to non-zero value)
b.5、final fields 的初始化也是在阶段 3 完成。
c、分配过程需要满足一个关键要求：阶段 1 和 2 中所有操作对于其他线程以及回收器都应当是原子化的。
d、分配过程中可能需要的参数：待分配空间大小 (The size requested) 、字节对齐要求(An alignment constraint)、待分配对象的类别(The kind of object to allocat)、待分配对象的具体类型(The specific type of object to allocat)
d.1、当分配数组时，可以将元素大小和个数作为独立参数
d.2、任何需要分配器特殊对待的需求都需要在分配接口中体现

2、清零(Zeroing)

2.1、概述

a、清零：某些系统为了安全要求将其空闲内存设置为指定的值，通常为 0，也有可能式其他值。
b、java 需要将空闲内存清零

2.2、如何进行清零

a、一次性堆较大空间进行清零将更高效，在对以 2 的整数次幂对齐的大内存块清零通常会达到最佳性能
b、因为读操作必须阻塞到硬件写缓冲区中的清零操作全部执行完为止，所以大量的清零操作会影响读操作
c、Hotspot 会在顺序分配中对位于阶跃指针之前的数据进行精确预取，并以此掩盖新分配数据从内存加载到高速缓存的延迟
d、从分配器角度看，最佳清零方式式调用运行时库提供的清零函数
e、使用虚拟内存的请求二进制零页(demand-zero page),通常更适合程序启动场景

3、保守式指针查找(Conservative pointer finding)

a、模糊指针(ambiguous pointer):将每个指针大小相同的已对齐字节序列当作可能是指针的值
b、鉴别指针过程：
b.1、阶段 1：回收器先过滤掉为指向任何堆空间地址的值，可以根据模糊指针的高位地址计算出其所对应的内存块编号
b.2、阶段 2：回收器需要鉴别出模糊指针所指向的地址是否真正被分配出去，只有以下步骤全为真，才对模糊指针的目标对象标记：
(a、使用堆边界判定

(b、判断其所引用的内存块是否已经被分配
(c、进啊册其所指向的内存党员是否已经被分配

4、精确指针查找(Accurate pointer finding)

4.1、使用带标签值(Accurate pointer finding using tagged values)

a、标签的基本实现策略：位窃取(bit stealing)和页簇(big bags of pages)
b、位窃取：需要在每个值中预留出一个或多个位，同时要求可能包含指针的对象必须以面向字的方式进行布局(Bit stealing reserves one or more bits, generally at the low or high end of each word, and lays out objects that can contain pointers in a word-oriented fashion)

六、基本算法

1、三色抽象

1.2、概述

a、很简洁的描述 GC 过程对象状态的变化(It is very convenient to have a concise way to describe the state of objects during a collection)

1.3、三色定义

a、黑色：存活对象，已被回收器处理完成；可以表示对象域被处理过；也可表示回收器已完成对其根的扫描，且不需要再次扫描
b、灰色：已被回收器扫描到，但未完成处理或需要被再次处理；可表示对象域正在被处理；也可表示回收器尚未完成对其根的扫描
c、白色：可能死亡的对象

1.4、基本流程

a、任意对象初始化状态为白色
b、当回收器初次扫描到当前对象时，置为灰色
c、当完成当前对象以及子节点扫描，置为黑色

2、可达性分析算法

2.1、概述

a、可达性(pointer reachability):如果从对象 M 的域 f 出发，经过一条指针链可以到达对象 N，则称对象 N 从对象 M 可达
b、数学定义：
a、对于堆中任意两个节点 M 和 N，当且仅当 Pointer(M)中存在一个域 f=&M[i]，且*f=N 时，才有 $M->_fN$。
b、同理，当且仅当根集合中存在域 f，且*f=N 时，才有 $Root->_fN$。
c、如果 Pointer(M)中存在域 f，使得 $M->_fN$，则可以说对象 N 从对象 M 直接可达，记作:M->N。
d、reachable={N ∈ Nodes | (∃ r ∈ Root: r->) V (∃ M ∈ reachable: M-N)}
e、赋值器能访问的只可能是堆中可达对象，不可达对象必然是死亡对象
f、在概念上可以将存活与可达等价，将死亡与不可达等价，将垃圾与不可达对象等价，但严格意义上并不准确

2.2、Java 中 GC Roots 对象

a、JVM 栈中引用的对象
b、方法区中类静态属性引用的对象
c、方法区中常量引用的对象
d、本地方法栈中 JNI 引用的对象
e、JVM 内部的引用、系统类加载器
f、所有被同步锁(synchronized)持有的对象
g、反映 JVM 内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等
h、根据用户所选的 GC 以及当前回收的内存区域不同，还可以有其他对象”临时”加入 GC Root

3、标记-清除(mark-sweep collection)

3.1、概述

a、共两阶段：标记 清理
b、是一种间接回收(indirect collection)算法
c、是可达性分析算法下最直接的回收方案(It is a straightforward embodiment of the recursive definition of pointer reachability)
d、每次执行都需要重新计算存活对象集合(it needs to recalculate its estimate of the set of live objects at each invocation)
e、分配器通常会将大小相同的对象分布在连续空间内，此时回收器可以依照固定步幅对大小相同的对象进行清扫
f、优点
f.1、实现简单，是最基础的收集算法
f.2、与保守式 GC 算法兼容
f.3、不会给赋值器带来任何额外的读写开销
f.4、使用懒惰清扫策略的标记清理回收器通常有较高的吞吐量
g、缺点
g.1、碎片化
g.2、执行效率差，最差情况每次分配都要将空闲链表全遍历一边
g.3、与写时复制(Copy-on-write)技术不兼容
h、要求对布局满足：
h.1、回收器不会移动对象
h.2、回收器能够遍历堆中每个对象
i、标记阶段完成标志是工作列表清空(worklist)

3.2、单线程实现

```
1.  //标记清除
2.  //新建对象
3.  new():
4.      //尝试给对象分配内存
5.      ref  <—  allocate()
6.      //分配失败
7.      if  ref  =  null
8.          //开始收集
9.          collect()
```

```
        //再次尝试分配
        ref <— allocate()
        //再次分配失败
        if ref = null
            //抛异常
            error "Out of memory" s
        //如果成功分配到内存，则返回地址引用
        return ref

//分配过程
atomic collect():
    //标记对象
    markFromRoots()
    //开始清理，传入参数为HeapStart=堆开始位置，HeapEnd=堆结束位置
    sweep(HeapStart, HeapEnd)

//标记过程
markFromRoots():
    //初始化工作列表，这个工作列表基于栈实现
    worklist <— empty
    //遍历根，采用深度优先遍历(depth-first traversal)
    for each fid in Roots
        //获取当前根对象
        ref <- *fld
        //如果根对象不为空，且未被标记
        if ref != null && not isMarked(ref)
            //标记根对象
            setMarked(ref)
            //将根对象压入栈
            add(worklist, ref)
            //开始遍历&标记根对象下面的对象
            mark()


//遍历&标记根对象下面的对象过程
mark():
    //循环遍历工作列表，直接工作列表为空
    while not isEmpty(worklist)
        //弹出栈顶，并获取弹出的对象
        ref <— remove(worklist)
        //获取当前对象的所有引用者
        for each fid in Pointers(ref)
            //获取引用者Z对象
            child <— *fld
            //如果对象非空，且未被标记
            if child != null && not isMarked(child)
                //标记对象
                setMarked(child)
                //压入栈
                add(worklist, child)

/**
清理过程
@Param start 起始位置
@Param end 结束位置
**/
sweep(start, end):
    //获取堆第一个对象开始位置
    scan <— start
    //判断当前位置是否已经到最后
    while scan < end
```

```
    //判断当前对象是否被标记
    if  isMarked(scan)
        //如果已经被标记，则取消标记
        unsetMarked(scan)
    else
        //如果未被标记，则释放对象
        free(scan)
    //获取下个对象位置
    scan  <—  nextObject(scan)


```

3.3、位图标记(Bitmap marking)

a、使用独立位图来维护标记位：为途中的每个位关联堆中可能分配对象的地址。
b、位图所需空间取决于 VM 的字节对齐要求，可以存在多个
c、可以为每个内存块维护独立的位图，可以避免由于堆不连续导致的内存浪费
d、独立位图可以置于对应的内存块中，但所有内存块的位图相对位置相同，可能会因为不同位图争用高速缓存导致性能下降
e、位图标记通常仅适用于单线程
f、使用位图引导清扫的好处：
f.1、回收器可以批量读取/清空一批对象的标记位
f.2、可以更简单的判定某内存块中的所有对象是否都是垃圾，从而可能一次回收整个块
g、Garner 提出了一种混合标记策略：将分区适应分配器(segregated fits allocator)所管理的每个数据块与字节图中的一个字节相关联，同时依然保留对象头部的标记位。
g.1、再并发情况下，无需使用同步操作来设置字节图中的标记字节以及对象头部的字节位
g.2、写操作没有数据依赖，且堆字节图中标记字节的写操作也是条件的
h、位图标记代码示例

```
//Printezis 和 Detlefs 的位图标记
mark()
    //获取下个位图标记
    cur <- nextInBitmap()
    //循环判断是否超过堆
    while cur < HeapEnd
        //当前位，加入 worklist
        add(worklist, cur)
        //标记当前位
        markStep(cur)
        //获取下个位
        cur <- nextInBitmap()
//标记 位
markStep(start):
    //循环遍历 worklist，直到空
    while not isEmpty(worklist)
        //弹出 worklist 元素
        ref <- remove(worklist)
        //遍历弹出元素的子节点
        for each fld in Pointers(ref)
            child <- *fld
            //判断子节点是否不为空且未被标记
            if child != null && not isMarked(child)
                //子节点不为空且未被标记，标记子节点
                setMarked(child)
                //如果子节点所在位置小于 start，则加入当前 worklist 循环
```

```
                if child < start
                    add(worklist, child)
```

3.4、懒惰清扫(Lazy sweeping)

a、清扫一个对象比追踪一个对象开销少

b、此算法基于两个对象与其标记位的特征

b.1、一旦某对象称为垃圾，将都一直是垃圾，不可能被赋值器访问或复活(once an object is garbage, it remains garbage: it can neither be seen nor be resurrected by a mutator)

b.2、赋值器不能操作标记位

c、利用分配器分担清理成本，将寻找可用空间的任务交给分配器

d、回收器依然需要需要将堆中所有存活对象标记，但不立即清扫整个堆，仅简单地将完全为空的内存块还给分配器，同时将其他内存块添加到其所对应空间大小分级的队列中

```
//
//收集
atomic collect():
    //标记根
    markFromRoots()
    //遍历内存块
    for each block in Blocks
        //判断块是否标记
        if not isMarked(block)
             //将未被标记的块归还给分配器
            add(blockAllocator, block)
        else
            //将标记标记的块，加入惰性队列
            add(reclaimList, block)

//分配
//sz: 指定的分配空间
atomic allocate(sz):
      //尝试分配
    result <- remove(sz)
    //判断是否分配成功
    if result = null
         //执行清理
        lazySweep(sz)
        //再次尝试分配
        result <- remove(sz)
    //返回分配结果
    return result
//惰性清理
lazySweep(sz) :
    //do while
    repeat
        //从标记块列表弹出一个符合 sz 大小的块
        block <- nextBlock(reclaimList, sz)
        if block != null
            //清理块
            sweep(start(block), end(block))
            if spaceFound(block)
                return
    until block = null
    //尝试分配空间
    allocSlow(sz)

allocSlow(sz) :
```

```
45.       //分配块
46.     block <- allocateBlock()
47.     if block != null
48.           //分配成功，初始化块
49.         initialise(block, sz)
```

4、标记-复制(mark-copy collection)

4.1、概述

a、将堆划分成两个相等半区(semispace):来源空间(fromspace)、目标空间(tospace)

b、无需再对象头部引入额外空间

c、每个槽都可以用于记录转发地址

d、对于较大的堆，相比标记清理，标记复制的性能更好

4.2、基本实现

```
1.  //初始化空间，假定堆是一块连续的内存空间
2.  createSemispaces():
3.          //目标空间起始位置
4.          tospace  <—  HeapStart
5.          //半区大小
6.          extent  <-  (HeapEnd  -  HeapStart)  /  2
7.          //半区最大地址=来源空间=堆开始地址+半区大小
8.          top  <—  fromspace  <—  HeapStart  +  extent
9.          //分配游标
10.         free  <—  tospace
11.
12. //分配内存
13. atomic  allocate(size):
14.             //获取当前可分配内存起始地址
15.         result  <—  free
16.         //获取新的可分配内存游标地址
17.         newfree  <—  result  +  size
18.         //如果新的游标地址大于 top
19.         if  newfree  >  top
20.                 //分配失败，内存溢出
21.                 return  null
22.         //更新可分配空间起始地址
23.         free  <—  newfree
24.         //返回分配内存的起始地址
25.         return  result
26.
27. //开始 GC
28. atomic  collect():
29.         flip()//翻转半区
30.         //初始化工作链表
31.         initialise(worklist)
32.         //遍历根列表，先将根对象复制到目标空间
33.         for  each  fld  in  Roots
34.                 //处理根节点
35.                 process(fld)
36.         //循环工作链表
37.         while  not  isEmpty(worklist)
38.                 //弹出工作链表中对象
39.                 ref  <—  remove(worklist)
40.                 //扫描对象
```

```
                scan(ref)
//翻转半区
flip():
        fromspace, tospace <— tospace, fromspace
        top <- tospace + extent
        free <— tospace
//扫描对象的所有引用
scan(ref):
            //遍历指定对象的引用
        for each fld in Pointers(ref)
                //处理节点
                process(fld)
//处理节点
process(fld):
            //获取对象源地址
        fromRef <- *fld
        //如果对象非空
        if fromRef != null
                    //移动对象并更新对象引用
                *fld <— forward(fromRef)
//重定向对象
forward(fromRef):
            //获取对象新地址
        toRef <— forwardingAddress(fromRef)
        //判断空间是否获取成功
        if toRef = null
                    //移动对象并获取新地址
                toRef <— copy(fromRef)
        return toRef

//复制对象
copy(fromRef):
        //获取可分配内存起始位置
        toRef <— free
        //更新可分配内存起始位置
        free <— free + size(fromRef)
        //移动
        move(fromRef, toRef)
        //重定向对象地址
        forwardingAddress(fromRef) <- toRef
        //将对象加入工作队列
        add(worklist, toRef)
        //返回对象新地址
        return toRef

//Cheney's work list
//初始化工作列表
//这里画图比较直观
initialise(worklist):
    //在这里 scan=free=topspace
    scan <- free

isEmpty(worklist):
    //当 scan 地址追上 free 时，即为空列表
    return scan = free
//弹出要处理的对象
remove(worklist):
        ref <- scan
        scan <— scan + size(scan)
        return ref

```

```
102. add(worklist,  ref):  /*  nop  */
```

4.3、准深度优先复制

```
1.  //Approximately  depth-first  copying
2.  //假设对象不跨页
3.  initialise(worklist):
4.          scan  <-  free
5.          parrtialScan  <—  free
6.
7.  isEmpty(worklist):
8.          return  scan  =  free
9.
10. remove(worklist):
11.             //优先弹出后面的对象
12.         if(partialScan  <  free)
13.                 ref  <—  partialScan
14.                 partialScan  <—  partialScan  +  size(partialScan)
15.         else
16.                 ref  <—  scan
17.                 scan  <-  scan  +  size(scan)
18.         return  ref
19.
20.
21. add(worklist,  ref):
22.         //优先扫描最后面的对象
23.         partialScan  <—  max(partialScan,  startOfPage(ref))
```

4.4、在线对象记录法(Online Object Recordering)

```
1.  //Online  object  reordering
2.  atomic  collect():
3.              //对调区域
4.          flip()
5.          //初始化，冷热队列
6.          initialise(hotList,  coldList)
7.          //遍历根节点
8.          for  each  fld  in  Roots
9.                  adviceProcess(fld)
10.                 //do  while
11.         repeat
12.                 //遍历热列表
13.                 while  not  isEmpty(hotList)
14.                         adviceScan(remove(hotList))
15.                 //遍历冷列表
16.                 while  not  isEmpty(coldList)
17.                         adviceProcess(remove(coldList))
18.         //直到热列表为空
19.         until  isEmpty(hotList)
20. //初始化
21. initialise(hotList,  coldList):
22.         hotList  <—  empty
23.         coldList  <—  empty
24. //通知处理
25. adviceProcess(fld):
26.             //获取当前对象旧地址
27.         fromRef  <—  *fld
28.         if  fromRef  !=  null
```

```
                    //移动并更新地址
                *fld  <—  forward(fromRef)
//通知扫描
adviceScan(obj):
            //遍历对象所有引用
        for  each  fld  in  Pointers(obj)
                //判断是否热对象
                if  isHot(fld)
                        adviceProcess(fld)
                else
                        add(coldList,  fld)
```

5、标记-整理(mark-compact collection)

5.1、概述

a、三种重新整理(rearrange)顺序：

a.1、任意顺序(Arbitrary)：对象迁移与其原始排列顺序和引用无关，会降低应用的吞吐量

a.2、线性顺序(Linearising)：将具有关联关系的对象排在一起

a.3、滑动顺序(Sliding)：将对象滑动到堆的一端，“挤出”垃圾，从而保持原有顺序，因此不影响赋值器局部性

5.2、Edwards 双指针算法

a、优势

a.1、简单快速

a.2、每次遍历过程的操作较少

a.3、转发地址是在对象移动后才写入，不会存在任何信息丢失

a.4、无需额外空间记录转发地址

a.5、支持内部指针，可预测内存访问模式，支持预取(prefetching)

```
atomic  collect():
    markFromRoots()
    compact()

//双指针，属于任意顺序整理
compact():
    //整理
    relocate(HeapStart,  HeapEnd)
    //重新定位
    updateReferences(HeapStart,  free)

relocate(start,  end):
    //free 指向开始地址
    free  <—  start
    //scan 指向结束地址
    scan  <—  end
    //开始移动
    while  free  <  scan
        //移动 free 指针，直到发现可收回对象
        while  isMarked(free)
            //如果发现活对象，则移动 free 指针
```

```
            unsetMarked(free)
            free <— free + size(free)
        //移动 scan，直到发现活对象或越过了 free 指针
        while not isMarked(scan) && scan > free
            //移动 scan 指针
            scan <— scan — size(scan)
        //这时 free 指向未分配空间起始地址
        //scan 指向一个活对象
        //判断 scan 是否越过 free
        if scan > free
            //取消 scan 对象的标记
            unsetMarked(scan)
            //将 scan 指向的对象移动到 free
            move(scan, free)
            //在当前*scan 位置上记录对象被转发到哪个地址上了
            *scan <— free
            //移动 free，距离为移入对象的大小
            free <— free + size(free)
            //移动 scan，距离为移出对象的大小
            scan <- scan — size(scan)

//更新对象引用
updateReferences(start, end):
    //遍历根集合
    for each fid in Roots
        //获取根对象最初地址
        ref <- *fld
        //判断根对象最初地址是否在 end 后
        if ref >= end
            //如果根对象在 end 后，说明根对象被移动过
            //获取根对象最新地址，更新引用位置
            *fld <— *ref
    //遍历所有对象
    scan <— start
    while scan<end
        //遍历当前对象地址集合
        for each fld in Pointers(scan)
            //更新对象地址
            ref <- *fld
            if ref >= end
                *fld <- *ref
        //获取下个对象
        scan <- scan + size(scan)
```

5.3、Lisp2 算法

a、主要缺陷在于需要在每个对象头部额外增加一个完整的字段记录转发地址

b、需要三次堆遍历

```
//Lisp 2
compact():
    //计算整理
    computeLocations(HeapStart, HeapEnd, HeapStart)
    //更新引用
    updateReferences(HeapStart, HeapEnd)
    //移动对象
    relocate(HeapStart, HeapEnd)

```

```
//start：待整理起始位置， end：待整理结束位置， toRegion；整理目标区域起始位置
(the start of the region into which the compacted objects are to be moved.)
computeLocations(start, end, toRegion):
    scan <- start
    free <- toRegion
    //遍历对象
    while scan < end
        //判断是否活对象
        if isMarked(scan)
            //如果是活对象，则在对象头记录转发地址
            forwardingAddress(scan) <- free
            //将目标区域预留指定大小
            free <- free + size(scan)
        //遍历下个对象
        scan <- scan + size(scan)

//更新引用
updateReferences(start, end):
                //遍历根集合
        for each fid in Roots
                        //获取根对象最初地址
                ref <- *fld
                //根对象非空
                if ref != null
                            //更新根对象引用地址
                        *fld <— forwardingAddress(ref)
        //遍历活对象
        scan <- start
        while scan < end
                //判断是否活对象
                if isMarked(scan)
                            //如果是活对象，则遍历对象地址集合
                        for each fld in Pointers(scan)
                                //如果非空
                                if *fld != null
                                            //更新地址
                                        *fld <— forwardingAddress(*fld)
                //获取下个对象
                scan <- scan + size(scan)

//移动对象
relocate(start, end):
        scan <- start
        //开始遍历
        while scan < end
                //判断是否活对象
                if isMarked(scan)
                            //获取新地址
                        dest <- forwardingAddress(scan)
                        //移动对象到新地址
                        move(scan, dest)
                        //取消对象标记
                        unsetMarked(dest)
                //下个对象
                scan <- scan + size(scan)
```

5.4、引线整理算法

a、这个算法还没搞明白。。。。

```
//Jonkers 引线算法
compact():
        updateForwardReferences()
        updateBackwardReferences()

thread(ref):
        if  *ref  !=  null
                *ref,  **ref  <-  **ref,  ref

update(ref,  addr):
        tmp  <-  *ref
        while  isReference(tmp)
                *tmp,  tmp  <-  addr,  *tmp
        *ref  <-  tmp

updateForwardReferences():
        for  each  fid  in  Roots
                thread(*fld)

        free  <—  HeapStart
        scan  <—  HeapStart
        while  scan  <  HeapEnd
                if  isMarked(scan)
                        update(scan,  free)
                        for  each  fid  in  Pointers(scan)
                                thread(fld)
                        free  <-  free  +  size(scan)
                scan  <r-  scan  +  size(scan)  28

updateBackwardReferences():
        free  <-  HeapStart
        scan  <-  HeapStart
        while  scan  <  HeapEnd
                if  isMarked(scan)
                        update(scan,  free)
                        move(scan,  free)
                        free  <-  free  +  size(scan)
                scan  <-  scan  +  size(scan)
```

5.5、单次遍历算法

```
//单次遍历----Compressor 遍历
compact():
          //计算位置
        computeLocations(HeapStart,  HeapEnd,  HeapStart)
        //更新引用
        updateReferencesRelocate(HeapStart,  HeapEnd)

//计算存活对象偏移量
computeLocations(start,  end,  toRegion):
                //获取新对象存放空间的开始下标
        loc  <—  toRegion
        //获取起始位置块下标
        block  <—  getBlockNum(start)
        //遍历位图，0  to  开始结束位置之间 bit 数量-1
        for  b  <—  0  to  numBits(start,  end)  —1
                //判断当前 bit 下标是否超过一个块的大小
                if  b  %  BITS_IN_BLOCK  =  0
                                //记录当前块对应的起始地址
```

```
                        offset[block] <— loc
                        //指向下个块
                        block <— block + 1
                //判断当前 bit 对应的对象是否存活
                if bitmap[b] = MARKED
                            //增加位移，进量为每 bit 对应的大小
                        loc <- loc + BYTES_PER_BIT

//通过块偏移量和块内容偏移量计算新地址
newAddress(old):
            //获取对象所在的块
        block <— getBlockNum(old)
        //块地址偏移量+对象在块内的偏移量，获取对象新地址
        return offset[block] + offsetInBlock(old)

//更新引用
updateReferencesRelocate(start, end):
            //遍历根路径
        for each fld in Roots
                //获取旧的引用
                ref <— *fld
                //判断是否有存活对象
                if ref != null
                            //更新引用地址
                        *fld <— newAddress(ref)
        //开始移动对象
        scan <— start
        while scan < end
                //获取下个活对象
                scan <— nextMarkedObject(scan)
                //遍历活对象的引用指针
                for each fid in Pointers(scan)
                        //更新引用指针
                        ref <— *fld
                        if ref != null
                                *fld <— newAddress(ref)
                //获取对象本身的新地址位置
                dest <— newAddress(scan)
                //移动对象至新位置
                move(scan, dest)
```

七、引用计数(Referenct counting)

1、概述

a、a、对象的存活性可以通过引用关系的创建或删除直接判定，从而无需像追踪式回收器那样需要遍历堆找出存活对象，再反向确定垃圾对象

b、b、当且仅当指向某个对象的引用数量大于 0 或持有某对象的客户端集合不为空(refernce listing 算法)时，对象才有可能是存活。

c、c、通常在对象头部的某个槽(slot)中保存着引用计数器(reference count)

d、d、对引用计数的修改要遵循先增后减的顺序，否正可能回导致对象过早回收

e、e、编程语言：Smalltalk、Lisp、awk、perl 和 python

f、f、应用：Photoshop、RealNetworks 的 Rhapsody 以及 OS 的文件管理模块

g、g、对于 C++等不支持自动内存管理的语言里，有许多支持对象安全回收的库通过使用智能指针(smart pointer)来访问对象,而智能指针通常会对构造函数以及赋值操作进行重载(overload)，从而实现对象所有权的独占或提供引用计数能力(Libraries to support safe reclamation of objects are

widely available for languages like C++ that do not yet require automatic memory management. Such
h、libraries often use smart pointers to access objects.Smart pointers typically overload constructors and operators such as assignment, either to enforce unique ownership of objects or to provide reference counting.)
h、提升效率方案:
i、h1、延迟计数引用(deferral reference counting)：将某些垃圾对象的鉴别推迟到某一时段结束时的回收段中，从而避免了某些屏障操作。即牺牲回收的时效性换取效率的提升
j、h2、合并引用计数(coalescing referenct counting):在单个时段内，只关注对象是否第一次被修改。
k、h3、缓冲引用计数(buffered reference counting):与延迟或合并不同，该方案将所有引用计数增减操作缓冲起来以便后续处理，同时只有回收线程可以执行引用计数变更操作。其关注的是“何时”执行计数变更操作，而不是“是否”需要进行变更操作。
l、

2、优点

a、在一些场合，在程序运行中，某一对象成为垃圾便可以立即回收，因此可以持续操作即将填满的堆，不用预留一定空间
b、引用计数算法直接操作指针来源和目标，所以局部性良好
c、当确定某对象非共享对象时，可以直接进行破坏性操作而不用事先创建副本(Client programs can use destructive updates rather than copying objects if they can prove that an object is not share)
d、引用计数算法的实现无需运行时系统的支持，特别是无需确定程序的 root(Reference counting can be implemented without assistance from or knowledge of the run-time system.In particular, it is not necessary to know the roots of the program)
e、即使系统部分不能使用，也能回收部分内存，这特性对于分布式系统十分有益

3、缺点

a、引用计数给赋值器带来了额外的时间开销，这一因素决定了引用计数算法不适用于通用的大容量、高性能内存管理器
b、为避免多线程竞争可能导致对象过早释放，不仅引用计数的增减操作时原子化，连加载和存储指针的操作都必须是原子化
c、在简单引用计数算法中，即使是只是读操作也需要引发一次内存写请求用以更新引用计数。这里的写操作会“污染(pollute)”高速缓存，同时可能引发额外的内存冲突
d、即使对象成为孤岛，引用计数算法也无法回收环状引用数据结构，即包含自引用的数据结构(data structures that contain references to themselves)
e、最差的情况下，某对象的引用计数可能等于堆中对象总数，即引用计数所占用空间一定是跟一个指针的大小:一个完整的槽(slot)
f、引用计数算法仍然有可能导致 STW 出现，比如删除某大对象最后引用时，递归删除子节点时
g、引用计数算法在效率方面的一个基本障碍是：对象的引用计数是程序的全局特征之一，但通常只有在（线程）局部状态下的操作才更加高效(One fundamental obstacle to efficiency is that object reference counts are part of the global state of the program, but operations on (thread) local state are usually more efficient)

4、基本引用计数算法

```
New():
        //分配空间
        ref <— allocate()
        if ref = null
                //OOM
                error "Out of memory"
        //初始化对象引用数为0
        rc(ref) <- 0
        return ref

//增加新目标对象计数，同时减少旧目标对象计数，即使对象是局部变量
//这个方法也是一个写屏障(wirte barrier)的例子
//参数 src：计数器集合，i：对象在计数器集合中的位置，ref：目标对象引用
atomic Write(src, i, ref):
        //增加对象的引用
        addReference(ref)
        //减少计数器的引用
        deleteReference(src[i])
        //更新计数器集合
        src[i] <— ref

//增加对象引用计数
addReference(ref):
        if ref != null
                rc(ref) <— rc(ref) + 1

//减少对象引用
deleteReference(ref):
        if ref != null
                //减少对象引用
                rc(ref) <— rc(ref) — 1
                //如果对象引用减少为0，即当前对象可能已死
                if rc(ref) = 0
                        //减少对象的所有引用对象的引用计数
                        for each fld in Pointers(ref)
                                deleteReference(*fld)
                        //释放对象
                        free(ref)
```

5、延迟引用计数

5.1、概述

a、只有当赋值器操作堆中对象时产生的引用计数变更才会立即执行，而操作栈或寄存器所产生的引用计数变更则会被延迟执行

b、如果忽略局部变量的引用计数操作，则计数便不再准确，因此立即回收引用计数为 0 的对象将不再安全

c、延迟引用计数必须引入 STW 来定期修正引用计数

5.2、代码示例

```
//延迟引用
/**
```

```
3.  *零引用表(zero  count  table,ZCT):表中对包含的对象都是引用计数为 0 但可能依旧存活的对象，实现方式包括但不限于位图和哈希表
4.  **/
5.  New():
6.          //分配
7.          ref  <-  allocate()
8.          if  ref  =  null
9.                  //分配失败，尝试 GC
10.                 collect()
11.                 //再次分配
12.                 ref  <—  allocate()
13.                 if  ref  =  null
14.                         //OOM
15.                         error  "Out  of  memory"
16.         //初始化引用数为 0
17.         rc(ref)  <-  0
18.         //加入 zct
19.         add(zct,  ref)
20.         return  ref
21. //写入对象
22. Write(src,  i,  ref):
23.         //列表是根对象集合
24.         if  src  =  Roots
25.                 //直接更新对应位置的引用计数
26.                 src[i]  <-  ref
27.         else
28.                 //原子操作
29.                 atomic
30.                         //增加引用
31.                         addReference(ref)
32.                         //因为增加了计数，所以从 zct 中移除
33.                         remove(zct,  ref)
34.                         //减少引用计数
35.                         deleteReferenceToZCT(src[i])
36.                         //更新计数器集合
37.                         src[i]  <—  ref
38.
39. //减少引用计数
40. deleteReferenceToZCT(ref):
41.         if  ref  !=  null
42.                 //减少引用计数
43.                 rc(ref)  <—  rc(ref)  —  1
44.                 //引用计数为 0
45.                 if  rc(ref)  =  0
46.                         //加入 zct
47.                         add(zct,  ref)
48.
49. //收集垃圾
50. atomic  collect():
51.         //遍历根集合
52.         for  each  fld  in  Roots
53.                 //将根的引用计数+1
54.                 addReference(*fld)
55.         //上面循环结束后引用计数为 0 的是垃圾
56.         //开始释放资源
57.         sweepZCT()
58.         //还原根引用计数
59.         for  each  fid  in  Roots
60.                 deleteReferenceToZCT(*fld)
61.
62. //清理零引用对象
```

```
sweepZCT():
        //遍历 zct
        while  not  isEmpty(zct)
                //弹出一个对象
                ref  <-  remove(zct)
                //判断对象引用是否为 0
                if  rc(ref)  =  0
                        //遍历对象的子引用
                        for  each  fld  in  Pointers(ref)
                                //减少子引用的计数
                                deleteReference(*fld)
                        //释放对象
                        free(ref)
```

6、合并引用计数

```
//合并引用计数
//写屏障
//指定缓存区标识
me  <-  myThreadld

Write(src,  i,  ref):
        //判断是否为脏(dirty)对象
        if  not  dirty(src)
                //如果非脏对象，则缓存到本地
                log(src)
        //记录引用计数
        src[i]  <—  ref
//缓存对象
log(obj):
        //遍历对象的子引用
        for  each  fld  in  Pointers(obj)
                if  *fld  !=  null
                        //将非空引用加入到本地缓存
                        append(updates[me],  *fld)
        //判断是否为脏对象
        if  not  dirty(obj)
                //如果非脏对象，则将对象本身加入本地缓存并塞入 slot 中
                slot  <—  appendAndCommit(updates[me],  obj)
                //设置对象日志标识
                setDirty(obj,  slot)
//判断是否脏对象
dirty(obj):
        //判断依据是对象在缓存中的地址
        return  logPointer(obj)  !=  CLEAN
//记录对象在缓存中的地址
setDirty(obj,  slot)
        logPointer(obj)  <—  slot


//更新引用计数
//STW，挂起所有当前程序线程
atomic  collect():
        //合并线程本地缓冲区
        collectBuffers()
        //处理对象引用计数
        processReferenceCounts()
```

```
        //清理零引用对象
        sweepZCT()
//合并线程本地缓冲区
collectBuffers():
        //初始化合并数组
        collectorLog <— []
        //遍历所有线程
        for each t in Threads
                //合并缓冲区
                collectorLog <— collectorLog + updates[t]

//处理对象引用计数
processReferenceCounts():
        //遍历缓冲区
        for each entry in collectorLog
                obj <— objFromLog(entry)
                //判断是否为脏对象
                if dirty(obj)
                        //去除脏对象标记
                        logPointer(obj) <- CLEAN
                        //增加最新引用的子节点计数
                        incrementNew(obj)
                        //减少副本中对象的旧子节点计数
                        decrementOld(entry)

//减少副本中对象的子节点计数
decrementOld(entry):
        for each fid in Pointers(entry)
                child <- *fld
                if child != null
                        //减少子节点计数
                        rc(child) <- rc(child) - 1
                        //子节点为计数 0
                        if rc(child) = 0
                                //加入 zct 等待清理
                                add(zct, child)
//增加计数
incrementNew(obj):
        //遍历子节点
        for each fid in Pointers(obj)
                child <- *fld
                if child != null
                        //增加子节点计数
                        rc(child) <- rc(child) + 1
```

7、环状引用计数

7.1、概述

a、对于环状数据结构而言，其内部对象引用至少为 1，因此仅靠引用计数本身无法回收环状垃圾

b、最简单优化策略，就是使用追踪式回收作为补充

c、可以把一组对象作为整体进行引用计数操作，当整理引用计数为 0 时将其集体收回

d、Brownbridge 算法：

d.1、强引用(strong reference)：普通引用。弱引用(weak reference)：将导致闭环出现的引用

d.2、每个对象需要包含一个强引用指针计数和一个弱引用指针计数

d.3、进行写操作时，写屏障会检测指针以及目标对象的强弱，并将所有可能产生环的引用设置为弱引用

d.4、赋值器在删除引用时可能需要改变指针的强弱属性，以保证“所有可达对象均为，且强引用不产生环”(Reference deletion may require the strength of pointers to be modified in order to preserve the invariants that all reachable objects are strongly reachable without creating any cycles of strong references.)
e、常见的优化算法还有实验删除算法(trial deletion)和部分追踪算法(partial tracing)以及Recycler 算法

7.2、Recycler 算法

```
1.  //环状引用计数  复杂度 O(N+E)
2.  // N is the number of node and E the number of edges
3.  /**
4.  *黑色表示对象肯定存活或已释放，即处理完成
5.  *白色表示垃圾，即待回收
6.  *灰色表示可能时环状垃圾中的一员
7.  *紫色表示可能是环状垃圾的备选根
8.  **/
9.  New():
10.         //分配空间
11.         ref  <—  allocate()
12.         if  ref  =  null
13.                 //回收
14.                 collect()
15.                 //再次尝试分配
16.                 ref  <—  allocate()
17.                 if  ref  =  null
18.                         //OOM
19.                         error  "Out  of  memory"
20.         //初始化引用
21.         rc(ref)  <-  0
22.         return  ref
23.
24. //增加引用计数
25. addReference(ref):
26.         if  ref  !=  null
27.                 //增加引用计数
28.                 rc(ref)  <—  rc(ref)  +  1
29.                 //不可能为垃圾，标记为黑色
30.                 colour(ref)  <—  black
31.
32. //删除引用
33. deleteReference(ref):
34.         if  ref  !=  null
35.                 //引用计数-1
36.                 rc(ref)  <—  rc(ref)-1
37.                 if  rc(ref)  =  0
38.                         //如果引用计数为 0，则准备释放对象
39.                         release(ref)
40.                 else
41.                         //作为环状结构的备选根
42.                         candidate(ref)
43. //释放对象
44. release(ref):
45.         //遍历子节点
46.         for  each  fld  in  Pointers(ref)
47.                 //删除字节点
48.                 deleteReference(fld)
49.         //将对象标记为黑色，处理完成
```

```
        colour(ref) <— black
        //判断对象是否环状结构的备选根
        if not ref in candidates
                //释放对象
                free(ref)

//加入环状结构的备选根
candidate(ref):
        if colour(ref) != purple
                //设置为紫色
                colour(ref) <— purple
                //与当前备选根集合合并
                candidates <- candidates U {ref}

//回收垃圾
atomic collect():
        //标记范围
        markCandidates()
        //扫描备选根集合，回收灰色对象
        for each ref in candidates
                scan(ref)
        //回收白色对象
        collectCandidates()

//通过备选环状根集合限定环状垃圾对象范围
markCandidates()
        //遍历环状垃圾对象备选根集合
        for ref in candidates
                //判断对象是否紫色
                if colour(ref) = purple
                        //如果对象仍为紫色，则说明该对象被添加到集合中后，没有新的引用指向该对象
                        //标记为灰色
                        markGrey(ref)
                else
                        //如果非紫色，从集合中移除
                        remove(candidates, ref)
                        //判断移除后对象的颜色和引用计数值
                        if colour(ref) = black && rc(ref) = 0
                                //如果为黑色且计数为 0 则释放资源
                                free(ref)

//判断对象是否为灰色
markGrey(ref):
        if colour(ref) != grey
                //标记为灰色
                colour(ref) <- grey
                //遍历子节点
                for each fld in Pointers(ref)
                        child <- *fld
                        if child != null
                                //引用计数-1
                rc(child) <— rc(child) — 1
                //递归遍历子节点
                markGrey(child)

scan(ref):
        if colour(ref) = grey
                if rc(ref) > 0
                        scanBlack(ref)
                else
                        //标记为白色
```

```
                        colour(ref) <— white
                        //遍历子节点
                        for each fld in Pointers(ref)
                                child <- *fld
                                if child != null
                                        //递归扫描子节点
                                        scan (child)

scanBlack(ref):
        //标记为黑色
        colour(ref) <- black
        for each fld in Pointers(ref)
                child <- *fld
                if child != null
                        //子引用的对象引用计数+1
                        rc(child) <— rc(child) + 1
                        //判断子节点是否为黑色
                        if colour(child) != black
                                //如果字节点非黑色，则递归扫描
                                scanBlack(child)


collectCandidates():
        while not is Empty (candidates)
                //弹出对象
                ref <— remove(candidates)
                collectWhite(ref)

collectWhite(ref):
        //判断对象是否为白色且不在备用根中
        if colour(ref) = white && not ref in candidates
                colour(ref) <— black
                for each fld in Pointers(ref)
                        child <- *fld
                        if child != null
                                collectWhite(child)
                free(ref)
```

八、JAVA 中 GC 实现

1、Serial/ Serial Old 收集器

a、单线程的收集器，执行收集时，必定 STW，但额外内存消耗最小
b、新生代采用复制算法，老年代采用标记整理算法
c、迄今为止，Serial 依然是客户端模式下的默认新生代收集器

2、ParNew

a、实质上是 Serial 的多线程版本，共用了大量代码，新生代采用复制算法
b、除了 Serial 外，只有它能与 CMS 配合
c、默认收集线程数与 CPU 核心数相同

3、Parallel Scavennge

a、新生代收集器，基于标记复制算法的并行收集器
b、目标是达到一个可控的吞吐量：处理器用于运行用户代码的时间与处理器总消耗时间(运行用户代码时间 + 运行垃圾收集时间)的比值
c、支持 GC 自适应调节策略(GC Ergonomics)

4、Parallel Old

a、Parallel Scavenge 老年版，基于标记整理算法，

5、Epsilon

a、一个不能够进行 GC 行为的垃圾收集器
b、用于需要剥离 GC 影响的性能测试和压力测试
c、应用只要运行数分钟甚至几秒，且只要 JVM 正确分配内存，就会在堆耗尽前退出，这时 Epsilon 可以提高效率

6、CMS(concurrent mark sweep)

a、以获取最短停顿时间为目的的收集器，基于标记清除算法，默认回收线程数=(CPU 核心数+3)/4
b、GC 过程分为四个步骤：
b.1、初始标记(CMS initial mark)：需要 STW，仅仅标记 GC Roots 能直接关联到的对象
b.2、并发标记(CMS concurrent mark):从 GC Roots 的直接关联对象遍历整个对象图，不需要 STW
b.3、重新标记(CMS remark)：会 STW，为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录
b.4、并发清除(CMS concurrent sweep):清理删除标记阶段判断已死对象
c、缺点：
c.1、对处理器资源非常敏感
c.2、无法处理浮动垃圾(Floating Garbage)，即出现在标记过程结束之后的垃圾，CMS 无法在当次处理，只能留在下次再处理，有可能出现“Concurrent Mode Failure“失败，导致 FullGC
c.3、GC 结束后，会有大量空间碎片产生

7、Shenanndoah

8、G1

9、ZGC

a、