

JVM笔记无特殊说明，基于jdk14

[零、参考资料 3](#_Toc40997592)

[一、对象 3](#_Toc40997593)

[1对象内存布局（HotSpot） 3](#_Toc40997594)

[1.1、概述 3](#_Toc40997595)

[1.2对象头(Header) 3](#_Toc40997596)

[2、New对象() 4](#_Toc40997597)

[4、对象查找 5](#_Toc40997598)

[5、对象生命 5](#_Toc40997599)

[5.1、年龄记录 5](#_Toc40997600)

[5.2、引用计数算法 5](#_Toc40997601)

[5.3、可达性分析算法 5](#_Toc40997602)

[5.4、GC ROOT 5](#_Toc40997603)

[三、内存分配 5](#_Toc40997604)

[1顺序分配(Sequential allocation) 5](#_Toc40997605)

[1.1概述 5](#_Toc40997606)

[1.2代码示例 5](#_Toc40997607)

[2空闲链表 5](#_Toc40997608)

[2.1概述 5](#_Toc40997609)

[2.1首次适应分配(first-fit allocation) 5](#_Toc40997610)

[2.2循环首次适应分配(Next-fit allocation) 7](#_Toc40997611)

[2.3最佳适应分配(best-fit allocation) 7](#_Toc40997612)

[2.4链表加速方案 8](#_Toc40997613)

[2.5笛卡尔树(Cartesian tree) 8](#_Toc40997614)

[3. 分区适应分配(Segregated-fits allocation) 9](#_Toc40997615)

[3.1概述 9](#_Toc40997616)

[3.2空间大小分级填充(Populating size classes) 9](#_Toc40997617)

[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) 9](#_Toc40997618)

[4.内存分配考虑点 9](#_Toc40997619)

[四、堆分区 9](#_Toc40997620)

[1分区目的(Why to partition) 9](#_Toc40997621)

[2如何分区(How to partition) 11](#_Toc40997622)

[3什么时候需要分区(When to partition) 11](#_Toc40997623)

[1.2全区域设计 11](#_Toc40997624)

[1.3大对象空间(Large object space) 11](#_Toc40997625)

[五、分代垃圾回收 11](#_Toc40997626)

[1分代假说 11](#_Toc40997627)

[2对象年龄 11](#_Toc40997628)

[五、基本算法 11](#_Toc40997629)

[1、三色抽象 11](#_Toc40997630)

[2、标记-清除(mark-sweep collection) 12](#_Toc40997631)

[2.1概述 12](#_Toc40997632)

[2.2单线程实现 12](#_Toc40997633)

[2.4、优化方案——位图标记(Bitmap marking) 13](#_Toc40997634)

[2.5、优化方案——懒惰清扫(Lazy sweeping) 13](#_Toc40997635)

[3、标记-复制(mark-copy collection) 14](#_Toc40997636)

[4、标记-整理(mark-compact collection) 17](#_Toc40997637)

[4.1概述 17](#_Toc40997638)

[4.2双指针算法 17](#_Toc40997639)

[4.3 Lisp2 算法 18](#_Toc40997640)

[4.4引线整理算法 19](#_Toc40997641)

[4.5单次遍历算法 20](#_Toc40997642)

[三、GC性能分析 21](#_Toc40997643)

[四、JAVA中GC实现 21](#_Toc40997644)

[JVM GC----G1 21](#_Toc40997645)

[JVM GC----ZGC 21](#_Toc40997646)

# 零、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
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. 限定未

# 一、对象

## 1对象内存布局（HotSpot）

### 1.1、概述

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

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

### 1.2对象头(Header)

1.2.1对象自身的运行时数据(Mark Word)

a、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。

b、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

c、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

|  |  |  |
| --- | --- | --- |
| 存储内容 | 标志位 | 状态 |
| 对象哈希吗、对象分代年龄 | 01 | 未锁定 |
| 指向锁记录的指针 | 00 | 轻量级锁定 |
| 指向重量级锁的指针 | 10 | 膨胀(重量级锁定) |
| 空、不需要记录信息 | 11 | GC标记 |
| 偏向线程ID、偏向时间戳、对象分代年龄 | 01 | 可偏向 |

1.2.2类型指针

a、JVM通过这个指针来确定该对象是哪个类型的实例

b、并不是所有JVM实现都必须在对象数据上保留类型指针。

c、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

1.3、实例数据

## 2、New对象()

2.1、JVM创建对象过程

A close up of a map

Description automatically generated

## 4、对象查找

### 4.2、引用计数算法

### 4.3、可达性分析算法

4.4、GC ROOT

## 二、对象生命周期

### 5.1、

### 

# 三、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

### 1.2代码示例

1. sequentialAllocate(n):
2. //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
3. result <— free
4. //result增加指定空间，作为作为新起始地址
5. newFree <— result + n
6. //判断newFree是否超过了界限指针
7. **if** newFree > limit
8. //如果超过了，则返回空，分配失败
9. **return** null
10. //则更新free指针
11. free <— newFree
12. **return** result

## 2空闲链表

### 2.1概述

a使用某种数据结构记录空闲内存单元的位置和大小

b实际数据结构并非一定是链表结构

c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，P88，

### 2.1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：

b1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

b2分裂后剩余空间小于指定阈值或百分比

c首次分配基本代码示例

1. //首次适应分配
2. //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
3. firstFitAllocate(n):
4. //获取头对象
5. prev <- addressOf(head)
6. //死循环
7. loop
8. //获取下个单元地址
9. curr <— next(prev)
10. **if** curr = null
11. //如果当前单元为空，则分配失败
12. **return** null
13. **else** **if** size(curr) < n
14. //如果当前单元小于所需空间，则继续遍历下个单元
15. prev <- curr
16. **else**
17. //如果当前单元符合所需空间，则开始分配内存
18. **return** listAllocate(prev, curr, n)
20. //策略1
21. //prev：上个单元地址，curr：当前单元空间地址，n：所需空间大小
22. listAllocate(prev, curr, n):
23. //获取当前单元起始地址
24. result <— curr
25. //判断是否需要判断
26. **if** shouldSplit(size(curr), n)
27. //如果需要分裂单元
28. //获取剩余部分起始地址
29. remainder <- result + n
30. //更新remainder下个对象起始地址
31. next(remainder) <— next(curr)
32. //更新remainder大小
33. size(remainder) <- size(curr) — n
34. //将prev单元的下个对象地址指向remainder
35. next(prev) <- remainder
36. **else**
37. //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
38. next(prev) <- next(curr)
39. **return** result
41. //策略2
42. //将单元的尾部分割出来
43. //return the portion at the end of the cell being split
44. //该方案不足之处在于对象对齐方式有所不同
45. //A possible disadvantage of this approach is the different alignment of objects
46. listAllocateAlt(prev, curr, n):
47. //判断是否需要分裂
48. **if** shouldSplit(size(curr), n)
49. //当前单元减去所需大小，重新设定当前单元大小
50. size(curr) <— size(curr) — n;
51. //将当前单元加上新的单元大小，获取分配空间的起始地址
52. result <— curr + size(curr)
53. **else**
54. //如果不需要分裂则当前单元即分配单元
55. next(prev) <- next(curr)
56. result <— curr
57. //返回分配单元
58. **return** result

### 2.2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：

c1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

c2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.)

c3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哦符相同
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)

### 2.3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)
9. loop
10. //获取下个单元地址
11. curr <— next(prev)
12. //判断当前是否为空，单元大小是否符合需求
13. **if** curr = null || size(curr) = n
14. //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
15. //判断当前单元是否为空
16. **if** curr != null
17. //如果当前单元非空，则获取最佳单元上个单元地址
18. bestPrev <— prev
19. //当前单元即为最佳单元
20. best <— curr
21. **else** **if** best = null
22. //如果最佳单元为空，则返回null，分配失败
23. **return** null
24. //分配内存
25. **return** listAllocate(bestPrev, best, n)
26. //判断当前单元是否小于需求或大于最佳大小
27. **else** **if** size(curr) < n || bestSize < size(curr)
28. //如果当前单元小于需求或大于最佳大小，则进入下个循环
29. prev <— curr
30. **else**
31. //如果如果当前单元大于需求且小于等于最佳大小
32. //设置当前单元为最佳单元
33. best <— curr
34. bestPrev <— prev
35. //当前单元大小为最佳大小单元
36. bestSize <— size(curr)

### 2.4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：

c1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。

c2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)

c3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)

c4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

### 2.5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

1. //笛卡尔树--基于首次适应分配
2. firstFitAllocateCartesian(n):
3. parent <— null
4. //从根开始遍历
5. 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<....<sk-1。

(when we speak of segregated-fits we mean multiple lists being used for allocating for the same (logical)space)

b

### 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)

4.2字节对齐

4.3空间大小限制

4.4边界标签

4.5堆可分析性

4.6局部性

4.7扩展块保护

4.8跨越映射

# 四、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

b1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

b2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回收器需要区分出对象是否被多线程共享。

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。

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

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

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

c物理页的组织方式也是一种堆分区。

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

a新创建的对象往往被修改的更频繁

## 2如何分区(How to partition)

## 3何时分区(When to partition)

### 1.2全区域设计

### 1.3大对象空间(Large object space)

# 五、分代垃圾回收

## 1分代假说

## 2对象年龄

# 六、基本算法

## 1、三色抽象

1.1、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)

1.2、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

1.3、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、标记-清除(mark-sweep collection)

### 2.1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优点

e.1、实现简单，是最基础的收集算法

e.2、与保守式GC算法兼容

f缺点

f.1、碎片化

f.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

f.3、与写时复制(Copy-on-write)技术不兼容

g、要求对布局满足：

g.1、回收器不会移动对象

g.2、回收器能够遍历堆中每个对象

h、标记阶段完成标志是工作列表清空(worklist)

### 2.2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()
10. //再次尝试分配
11. ref <— allocate()
12. //再次分配失败
13. **if** ref = **null**
14. //抛异常
15. error "Out of memory" s
16. //如果成功分配到内存，则返回地址引用
17. **return** ref
19. //分配过程
20. atomic collect():
21. //标记对象
22. markFromRoots()
23. //开始清理，传入参数为HeapStart=堆开始位置，HeapEnd=堆结束位置
24. sweep(HeapStart, HeapEnd)
26. //标记过程
27. markFromRoots():
28. //初始化工作列表，这个工作列表基于栈实现
29. worklist <— empty
30. //遍历根，采用深度优先遍历(depth-first traversal)
31. **for** each fid in Roots
32. //获取当前根对象
33. ref <- \*fld
34. //如果根对象不为空，且未被标记
35. **if** ref != **null** && not isMarked(ref)
36. //标记根对象
37. setMarked(ref)
38. //将根对象压入栈
39. add(worklist, ref)
40. //开始遍历&标记根对象下面的对象
41. mark()

44. //遍历&标记根对象下面的对象过程
45. mark():
46. //循环遍历工作列表，直接工作列表为空
47. **while** not isEmpty(worklist)
48. //弹出栈顶，并获取弹出的对象
49. ref <— remove(worklist)
50. //获取当前对象的所有引用者
51. **for** each fid in Pointers(ref)
52. //获取引用者Z对象
53. child <— \*fld
54. //如果对象非空，且未被标记
55. **if** child != **null** && not isMarked(child)
56. //标记对象
57. setMarked(child)
58. //压入栈
59. add(worklist, child)
61. /\*\*
62. 清理过程
63. @Param start 起始位置
64. @Param end 结束位置
65. \*\*/
66. sweep(start, end):
67. //获取堆第一个对象开始位置
68. scan <— start
69. //判断当前位置是否已经到最后
70. **while** scan < end
71. //判断当前对象是否被标记
72. **if** isMarked(scan)
73. //如果已经被标记，则取消标记
74. unsetMarked(scan)
75. **else**
76. //如果未被标记，则释放对象
77. free(scan)
78. //获取下个对象位置
79. scan <— nextObject(scan)

### 2.4、优化方案——位图标记(Bitmap marking)

### 2.5、优化方案——懒惰清扫(Lazy sweeping)

## 3、标记-复制(mark-copy collection)

3.1概述

3.2基本实现

1. //初始化空间
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
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

28. //开始GC
29. atomic collect():
30. flip()//翻转半区
31. //初始化工作链表
32. initialise(worklist)
33. //遍历根列表
34. **for** each fld in Roots
35. //处理根节点
36. process(fld)
37. //循环工作链表
38. **while** not isEmpty(worklist)
39. //弹出工作链表中对象
40. ref <— remove(worklist)
41. //扫描对象
42. scan(ref)
43. //翻转半区
44. flip():
45. fromspace, tospace <— tospace, fromspace
46. top <- tospace + extent
47. free <— tospace
48. //扫描对象的所有引用
49. scan(ref):
50. //遍历指定对象的引用
51. **for** each fld in Pointers(ref)
52. //处理节点
53. process(fld)
54. //处理节点
55. process(fld):
56. //获取对象源地址
57. fromRef <- \*fld
58. //如果对象非空
59. **if** fromRef != null
60. //移动对象并更新对象引用
61. \*fld <— forward(fromRef)
62. //重定向对象
63. forward(fromRef):
64. //获取对象新地址
65. toRef <— forwardingAddress(fromRef)
66. //判断空间是否获取成功
67. **if** toRef = null
68. //移动对象并获取新地址
69. toRef <— copy(fromRef)
70. **return** toRef

73. copy(fromRef):
74. //获取可分配内存起始位置
75. toRef <— free
76. //更新可分配内存起始位置
77. free <— free + size(fromRef)
78. //移动
79. move(fromRef, toRef)
80. //重定向对象地址
81. forwardingAddress(fromRef) <- toRef
82. //将对象加入工作队列
83. add(worklist, toRef)
84. //返回对象新地址
85. **return** toRef
87. //Cheney's work list
88. //初始化工作列表
89. //这里画图比较直观
90. initialise(worklist):
91. //在这里scan=free=topspace
92. scan <- free
94. isEmpty(worklist):
95. //当scan地址追上free时，即为空列表
96. **return** scan = free
97. //弹出要处理的对象
98. remove(worklist):
99. ref <- scan
100. scan <— scan + size(scan)
101. **return** ref
103. add(worklist, ref): /\* nop \*/

3.3遍历顺序和局部性

3.4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
7. isEmpty(worklist):
8. **return** scan = free
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

21. add(worklist, ref):
22. //优先扫描最后面的对象
23. partialScan <— max(partialScan, startOfPage(ref))

3.5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
29. //移动并更新地址
30. \*fld <— forward(fromRef)
31. //通知扫描
32. adviceScan(obj):
33. //遍历对象所有引用
34. **for** each fld in Pointers(obj)
35. //判断是否热对象
36. **if** isHot(fld)
37. adviceProcess(fld)
38. **else**
39. add(coldList, fld)

## 4、标记-整理(mark-compact collection)

### 4.1概述

### 4.2双指针算法

1. atomic collect():
2. markFromRoots()
3. compact()

6. //双指针
7. compact():
8. //整理
9. relocate(HeapStart, HeapEnd)
10. //重新定位
11. updateReferences(HeapStart, free)
13. relocate(start, end):
14. //free指向开始地址
15. free <— start
16. //scan指向结束地址
17. scan <— end
18. //开始移动
19. **while** free < scan
20. //移动free指针，直到发现可收回对象
21. **while** isMarked(free)
22. //如果发现活对象，则移动free指针
23. unsetMarked(free)
24. free <— free + size(free)
25. //移动scan，直到发现活对象或越过了free指针
26. **while** not isMarked(scan) && scan > free
27. //移动scan指针
28. scan <— scan — size(scan)
29. //这时free指向未分配空间起始地址
30. //scan指向一个活对象
31. //判断scan是否越过free
32. **if** scan > free
33. //取消scan对象的标记
34. unsetMarked(scan)
35. //将scan指向的对象移动到free
36. move(scan, free)
37. //在当前\*scan位置上记录对象被转发到哪个地址上了
38. \*scan <— free
39. //移动free，距离为移入对象的大小
40. free <— free + size(free)
41. //移动scan，距离为移出对象的大小
42. scan <- scan — size(scan)
44. //更新对象引用
45. updateReferences(start, end):
46. //遍历根集合
47. **for** each fid in Roots
48. //获取根对象最初地址
49. ref <- \*fld
50. //判断根对象最初地址是否在end后
51. **if** ref >= end
52. //如果根对象在end后，说明根对象被移动过
53. //获取根对象最新地址，更新引用位置
54. \*fld <— \*ref
55. //遍历所有对象
56. scan <— start
57. **while** scan<end
58. //遍历当前对象地址集合
59. **for** each fld in Pointers(scan)
60. //更新对象地址
61. ref <- \*fld
62. **if** ref >= end
63. \*fld <- \*ref
64. //获取下个对象
65. scan <- scan + size(scan)

### 4.3 Lisp2 算法

1. //Lisp 2
2. compact():
3. //计算整理
4. computeLocations(HeapStart, HeapEnd, HeapStart)
5. //更新引用
6. updateReferences(HeapStart, HeapEnd)
7. //移动对象
8. relocate(HeapStart, HeapEnd)
10. //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.)
11. computeLocations(start, end, toRegion):
12. scan <- start
13. free <- toRegion
14. //遍历对象
15. **while** scan < end
16. //判断是否活对象
17. **if** isMarked(scan)
18. //如果是活对象，则在对象头记录转发地址
19. forwardingAddress(scan) <- free
20. //将目标区域预留指定大小
21. free <- free + size(scan)
22. //遍历下个对象
23. scan <- scan + size(scan)
25. //更新引用
26. updateReferences(start, end):
27. //遍历根集合
28. **for** each fid in Roots
29. //获取根对象最初地址
30. ref <- \*fld
31. //根对象非空
32. **if** ref != null
33. //更新根对象引用地址
34. \*fld <— forwardingAddress(ref)
35. //遍历活对象
36. scan <- start
37. **while** scan < end
38. //判断是否活对象
39. **if** isMarked(scan)
40. //如果是活对象，则遍历对象地址集合
41. **for** each fld in Pointers(scan)
42. //如果非空
43. **if** \*fld != null
44. //更新地址
45. \*fld <— forwardingAddress(\*fld)
46. //获取下个对象
47. scan <- scan + size(scan)
49. //移动对象
50. relocate(start, end):
51. scan <- start
52. //开始遍历
53. **while** scan < end
54. //判断是否活对象
55. **if** isMarked(scan)
56. //获取新地址
57. dest <- forwardingAddress(scan)
58. //移动对象到新地址
59. move(scan, dest)
60. //取消对象标记
61. unsetMarked(dest)
62. //下个对象
63. scan <- scan + size(scan)

### 4.4引线整理算法

1. //Jonkers引线算法
2. compact():
3. updateForwardReferences()
4. updateBackwardReferences()
6. **thread**(ref):
7. **if** \*ref != null
8. \*ref, \*\*ref <- \*\*ref, ref
10. update(ref, addr):
11. tmp <- \*ref
12. **while** isReference(tmp)
13. \*tmp, tmp <- addr, \*tmp
14. \*ref <- tmp
16. updateForwardReferences():
17. **for** each fid in Roots
18. **thread**(\*fld)
20. free <— HeapStart
21. scan <— HeapStart
22. **while** scan < HeapEnd
23. **if** isMarked(scan)
24. update(scan, free)
25. **for** each fid in Pointers(scan)
26. **thread**(fld)
27. free <- free + size(scan)
28. scan <r- scan + size(scan) 28
30. updateBackwardReferences():
31. free <- HeapStart
32. scan <- HeapStart
33. **while** scan < HeapEnd
34. **if** isMarked(scan)
35. update(scan, free)
36. move(scan, free)
37. free <- free + size(scan)
38. scan <- scan + size(scan)

### 4.5单次遍历算法

1. //单次遍历----Compressor遍历
2. compact():
3. //计算位置
4. computeLocations(HeapStart, HeapEnd, HeapStart)
5. //更新引用
6. updateReferencesRelocate(HeapStart, HeapEnd)
8. //计算存活对象偏移量
9. computeLocations(start, end, toRegion):
10. //获取新对象存放空间的开始下标
11. loc <— toRegion
12. //获取起始位置块下标
13. block <— getBlockNum(start)
14. //遍历位图，0 to 开始结束位置之间bit数量-1
15. **for** b <— 0 to numBits(start, end) —1
16. //判断当前bit下标是否超过一个块的大小
17. **if** b % BITS\_IN\_BLOCK = 0
18. //记录当前块对应的起始地址
19. offset[block] <— loc
20. //指向下个块
21. block <— block + 1
22. //判断当前bit对应的对象是否存活
23. **if** bitmap[b] = MARKED
24. //增加位移，进量为每bit对应的大小
25. loc <- loc + BYTES\_PER\_BIT
27. //通过块偏移量和块内容偏移量计算新地址
28. newAddress(old):
29. //获取对象所在的块
30. block <— getBlockNum(old)
31. //块地址偏移量+对象在块内的偏移量，获取对象新地址
32. **return** offset[block] + offsetInBlock(old)
34. //更新引用
35. updateReferencesRelocate(start, end):
36. //遍历根路径
37. **for** each fld in Roots
38. //获取旧的引用
39. ref <— \*fld
40. //判断是否有存活对象
41. **if** ref != null
42. //更新引用地址
43. \*fld <— newAddress(ref)
44. //开始移动对象
45. scan <— start
46. **while** scan < end
47. //获取下个活对象
48. scan <— nextMarkedObject(scan)
49. //遍历活对象的引用指针
50. **for** each fid in Pointers(scan)
51. //更新引用指针
52. ref <— \*fld
53. **if** ref != null
54. \*fld <— newAddress(ref)
55. //获取对象本身的新地址位置
56. dest <— newAddress(scan)
57. //移动对象至新位置
58. move(scan, dest)

# 三、GC性能分析

1、

# 四、JAVA中GC实现

1 Serial收集器

2 ParNew

3 Parallel Scavennge

4 Serial Old

5 Parallel Old

6 CMS

8 Shenanndoah

10 Epsilon

# JVM GC----G1

# JVM GC----ZGC