

JVM笔记无特殊说明，基于jdk14

[零、参考资料 3](#_Toc40553653)

[一、对象 3](#_Toc40553654)

[1对象内存布局（HotSpot） 3](#_Toc40553655)

[1.1、概述 3](#_Toc40553656)

[1.2对象头(Header) 3](#_Toc40553657)

[2、New对象() 4](#_Toc40553658)

[4、对象查找 5](#_Toc40553659)

[5、对象生命 5](#_Toc40553660)

[5.1、年龄记录 5](#_Toc40553661)

[5.2、引用计数算法 5](#_Toc40553662)

[5.3、可达性分析算法 5](#_Toc40553663)

[5.4、GC ROOT 5](#_Toc40553664)

[三、内存分配 5](#_Toc40553665)

[1顺序分配(Sequential allocation) 5](#_Toc40553666)

[1.1概述 5](#_Toc40553667)

[1.2代码示例 5](#_Toc40553668)

[2空闲链表 5](#_Toc40553669)

[2.1概述 5](#_Toc40553670)

[2.1首次适应分配(first-fit allocation) 5](#_Toc40553671)

[2.2循环首次适应分配(Next-fit allocation) 6](#_Toc40553672)

[2.3最佳适应分配(best-fit allocation) 7](#_Toc40553673)

[2.4链表加速方案 8](#_Toc40553674)

[2.5笛卡尔树(Cartesian tree) 8](#_Toc40553675)

[3. 分区适应分配(Segregated-fits allocation) 9](#_Toc40553676)

[3.1概述 9](#_Toc40553677)

[3.2空间大小分级填充(Populating size classes) 9](#_Toc40553678)

[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](#_Toc40553679)

[4.内存分配考虑点 9](#_Toc40553680)

[四、堆分区 9](#_Toc40553681)

[1分区目的(Why to partition) 9](#_Toc40553682)

[2如何分区(How to partition) 10](#_Toc40553683)

[3什么时候需要分区(When to partition) 10](#_Toc40553684)

[4分代垃圾回收 10](#_Toc40553685)

[1.2全区域设计 10](#_Toc40553686)

[1.3大对象空间(Large object space) 10](#_Toc40553687)

[五、基本算法 10](#_Toc40553688)

[1、三色抽象 10](#_Toc40553689)

[2、标记-清除(mark-sweep collection) 10](#_Toc40553690)

[2.1概述 10](#_Toc40553691)

[2.2单线程实现 11](#_Toc40553692)

[2.4、优化方案——位图标记(Bitmap marking) 12](#_Toc40553693)

[2.5、优化方案——懒惰清扫(Lazy sweeping) 12](#_Toc40553694)

[3、标记-复制(mark-copy collection) 12](#_Toc40553695)

[4、标记-整理(mark-compact collection) 13](#_Toc40553696)

[4.1概述 13](#_Toc40553697)

[4.2双指针算法 13](#_Toc40553698)

[4.3 Lisp2 算法 13](#_Toc40553699)

[4.4引线整理算法 13](#_Toc40553700)

[4.5单次遍历算法 13](#_Toc40553701)

[三、GC性能分析 13](#_Toc40553702)

[四、JAVA中GC实现 13](#_Toc40553703)

# 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

# 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

## 5、对象生命

### 5.1、年龄记录

### 5.2、引用计数算法

### 5.3、可达性分析算法

### 5.4、GC ROOT

# 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

### 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)

1.2根据对象大小分区(Partitioning by size)

1.3根据空间大小分区(Partitioning for space)

1.4根据类别进行分区(Partitioning by kind)

1.5为效益分区(Partitioning for yield)

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

1.7为局部性分区(Partitioning for locality)

1.8根据线程分区(Partitioning by thread)

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

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

## 2如何分区(How to partition)

## 3什么时候需要分区(When to partition)

## 4分代垃圾回收

4.1分代假说

4.2对象年龄

### 1.2全区域设计

### 1.3大对象空间(Large object space)

# 五、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

## 4、标记-整理(mark-compact collection)

### 4.1概述

### 4.2双指针算法

### 4.3 Lisp2 算法

### 4.4引线整理算法

### 4.5单次遍历算法

# 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

7 G1

8 Shenanndoah

9 ZGC

10 Epsilon