

JVM笔记无特殊说明，基于jdk14

[零、参考资料 2](#_Toc39607097)

[一、对象 2](#_Toc39607098)

[1对象内存布局（HotSpot） 2](#_Toc39607099)

[1.1、概述 2](#_Toc39607100)

[1.2对象头(Header) 2](#_Toc39607101)

[2、New对象() 3](#_Toc39607102)

[3、内存分配 3](#_Toc39607103)

[4、对象查找 3](#_Toc39607104)

[5、对象已死？ 3](#_Toc39607105)

[5.1、年龄记录 3](#_Toc39607106)

[5.2、引用计数算法 3](#_Toc39607107)

[5.3、可达性分析算法 3](#_Toc39607108)

[5.4、GC ROOT 3](#_Toc39607109)

[三、分区策略 3](#_Toc39607110)

[1.1分代设计 3](#_Toc39607111)

[1.2全区域设计 3](#_Toc39607112)

[1.3大对象空间 3](#_Toc39607113)

[四、基本算法 3](#_Toc39607114)

[1、三色抽象 3](#_Toc39607115)

[2、标记-清除(mark-sweep collection) 4](#_Toc39607116)

[2.1概述 4](#_Toc39607117)

[2.2基础实现 5](#_Toc39607118)

[2.4、优化方案——位图标记(Bitmap marking) 5](#_Toc39607119)

[2.5、优化方案——懒惰清扫(Lazy sweeping) 5](#_Toc39607120)

[3、标记-复制(mark-copy collection) 6](#_Toc39607121)

[4、标记-整理(mark-compact collection) 6](#_Toc39607122)

[三、GC性能分析 6](#_Toc39607123)

[四、JAVA中GC实现 6](#_Toc39607124)

# 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

## 3、内存分配

## 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、分区策略

### 1.1分代设计

### 1.2全区域设计

### 1.3大对象空间(Large object spaceå)

# 四、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、除了Root，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)

2.3多线程实现

### 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双指针算法

3.3 Lisp2 算法

3.4引线整理算法

3.5单次遍历算法

## 4、标记-整理(mark-compact collection)

# 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