出色的“清洁工具” — 理解 IBM Java 垃圾收集器，第一部分：：对象分配

Sam Borman (sambo@uk.ibm.com)
软件工程师
EMC

2002 年 9 月 02 日

本文是关于 IBM Java 垃圾收集器（GC）系列文章（共三篇）的第一篇，IBM Java 垃圾收集器是用于 IBM Java 开发工具箱和运行时环境的存储器管理器。该系列文章将包含：GC 使用的存储器区；对象分配；垃圾收集；外部接口是如何工作的；以及 `verbosegc`和其它命令行参数。
本文讨论如何在 Java 堆中分配对象以进行垃圾收集。它描述了对象的布局并研究了某些数据区（如堆和空闲表）。作者还讨论了直接从堆进行分配和线程本地分配，并给出了控制堆大小的一些建议。本文中的信息适用于 Java 1.2.2 版到 1.3.1 版。

概述

来自其它子组件的调用使对象分配使用下列分配接口之一，如 `stCacheAlloc`、`stAllocObject`、`stAllocArray`或 `stAllocClass`。分配接口从堆中分配给定数量的存储器，但具有不同参数和语义。`stCacheAlloc`例程是专门为小对象提供最佳分配性能而设计的。对象是从线程本地分配缓冲区中直接分配的，线程本地分配缓冲区是线程先前从堆中分配的。新对象是从这个高速缓存的末端分配的，而不必获取堆锁，因此效率非常高。如果通过 `stAllocObject`和 `stAllocArray`接口分配的对象足够小（当前是 512 字节），也会从高速缓存进行分配。

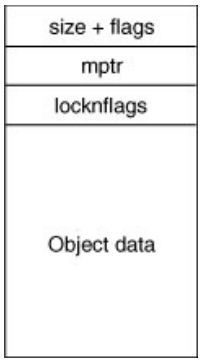
数据区

本节讨论 GC 的对象和堆或存储器方面。

对象

图 1显示了对象在堆上的布局。在图的下方说明了对象的各个部分。

图 1. 对象布局



size + flags size + flags 位置在 32 位体系结构上是 4 个字节，在 64 位体系结构上是 8 个字节。这个位置的主要目的是包含对象的长度。因为所有对象都是从 8 字节边界开始，并且大小可以被 8 整除，所以底部的 3 位不用于表示大小；我们将其作为标志，用于表示对象的不同状态。因为对象大小有限，所以最高 2 位可以用于标志。请注意，mptr 是按 8 字节边界划分颗粒度的，而这一位置不是。标志如下：

- 第 1 位是仅在压缩期间使用的交换位。第 1 位还是仅在标志堆栈溢出期间使用的 NotYetScanned 位。另外，第 1 位还是用于表示这个对象已被多次固定的多固定位。在垃圾收集期间，将除去多固定位并且恢复它以允许对这一重载位的其它使用。
- 第 2 位是 dosed 位。如果从堆栈或寄存器引用对象，则将 dosed 位设置为启用。在这一 GC 周期中不能移动对象；我们不能修改引用，因为它可能不是一个真的引用而只是恰巧与堆上的对象同值的一个整数。
- 第 3 位是固定位。通常不能移动固定对象，因为它们是从堆外部被引用的。示例包含 `Thread`和 `ClassClass`对象。
- 32 位体系结构中的第 31 位或 64 位体系结构中的第 63 位是由锁定（LK）组件使用的平面锁定争用（flc）位。
- 32 位体系结构中的第 32 位或 64 位体系结构中的第 64 位是散列位，用来表示已经返回其散列值的对象。这是必需的，因为散列值是对象的地址，并且如果您移动该对象，则需要维护它。

mptr mptr 位置在 32 位体系结构上是 4 字节，在 64 位体系结构上是 8 字节。mptr 位置是按 8 字节边界划分颗粒度的，而 size + flags 位置不是。mptr 具有以下两种功能之一：

- 如果 mptr 位置 不是数组，则 mptr 指向方法块，您可以从方法块达到类块。类块告诉您对象是哪一个类的实例。类装入程序分配方法块和类块，这两者都不在堆中。
- 如果 mptr 位置 是数组，则 mptr 包含这个对象中数组项的数量。

locknflags 虽然只使用低位的 4 个字节，但这个位置在 32 位体系结构上是 4 个字节，在 64 位体系结构是 8 个字节。它主要用于在锁定时包含 LK 组件的数据，locknflags 还具有一些标志。下列标志很重要：

- 第 2 位是数组标志。如果启用这个位，则对象是数组并且 mptr 字段将包含数组中元素数量的计数。
- 第 4 位是散列和移动位。如果启用这个位，则它告诉我们对象在被散列后已经被移动，且散列值在对象的最后位置中。

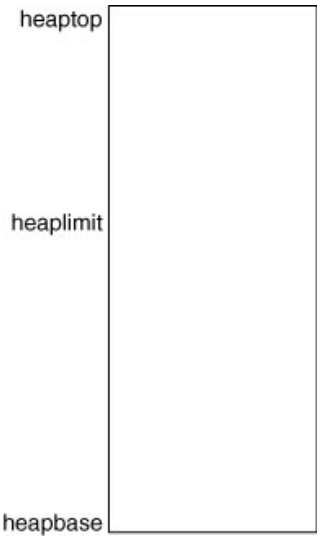
对象数据（Object data） 这是对象数据开始的位置，并且布局取决于对象。

size + flags、mptr 和 locknflags 有时合称为 头。

堆

堆是在 JVM 初始化时从操作系统获取的连续的存储器块。 图 2显示了堆的布局。

图 2. 堆



heapbase是堆的起始地址， heaptop是堆的结束地址。 heaplimit是堆当前使用部分顶部的地址， 可以对它进行缩放。从 heapbase 到 heaptop 的大小是由 `-Xmx`选项控制的。如果没有指定这个选项，则按下列方式应用缺省值：

Java 发行版 1.2.2 和 1.3.0	发行版 1.3.1
对于 Windows 平台，实际存储器的一半，最小 16M 最大 2G-1。	对于 Windows 平台，实际存储器的一半，最小 16M 最大 2G-1。
对于所有其它平台，64M。	对于 OS/390 和 AIX 平台，64M。
	对于 Linux 平台，实际存储器的一半，最小 16M 最大 512M-1。

从 heapbase 到 heaplimit 的初始大小由 `-Xms`选项控制。如果没有指定这个选项，则按下列方式应用缺省值：

Java 发行版 1.2.2 和 1.3.0	发行版 1.3.1
对于 Windows 平台，4M。	对于 Windows、AIX 和 Linux 平台，4M。
对于所有其它平台，1M。	对于 OS/390，1M。

设置堆大小

对于大部分应用程序，缺省设置选项将工作良好。堆将扩充，直到达到稳定状态为止。然后，堆保持这个状态，这会占据堆的 70%（或者任何时间堆上活动数据的数量）。在这一级别上，GC 的频率和 GC 的暂停时间应该处在可接受的级别上。

对于某些应用程序，缺省设置可能不会产生最佳结果。下面是可能出现的一些问题以及一些建议采取的措施。您可以使用 `verbosegc`来帮助监控堆。

问题	建议措施
在堆达到稳定状态以前，GC 的频率太高。	使用 verbosegc确定处于稳定状态的堆大小并将 -Xms设置成这个值。
堆被完全扩展并且占用率大于 70%。	增加 -Xmx值使堆占用率不超过 70%。为了获取最佳性能，尝试确保堆从不换页。物理内存应该能容纳最大的堆大小（如果可能）。
占用率为 70% 时，GC 的频率过大。	更改 -Xminf的设置。缺省值是 0.3，它将通过扩充堆来尝试维持 30% 可用空间。设置为 0.4 将可用空间目标增加到 40%，从而降低 GC 的频率。
暂停时间过长。	尝试使用 -Xgcpolicy:optavgpause（在 1.3.1 中引入），它在堆占用增加时减少暂停时间并且使它们更一致。在吞吐量方面要付出代价。代价是变化的，大约在 5% 左右。

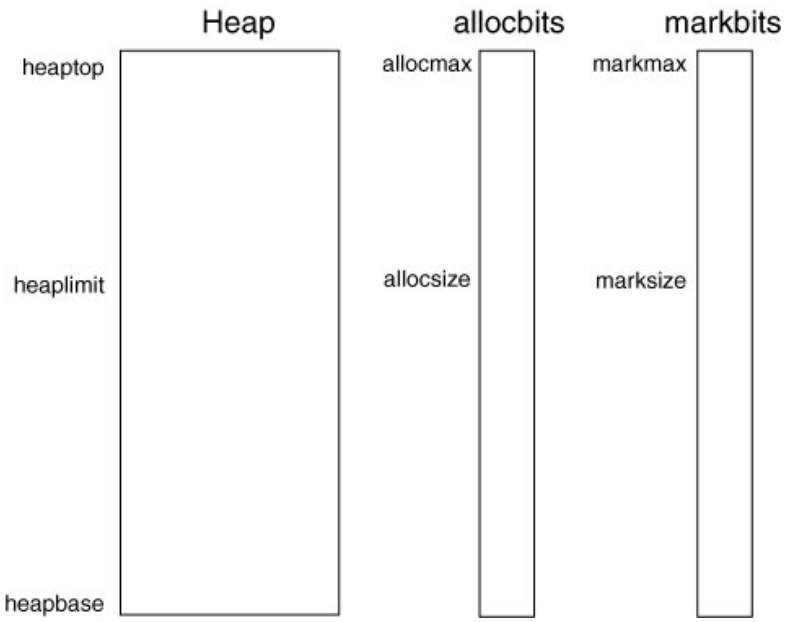
在实际工作中下列技巧很有用：

- 确保堆从不换页；物理内存必须容纳最大的堆大小。
- 避免终止函数 — 您无法确定何时终止函数会运行，并且它们通常会产生问题。如果您确实要使用终止函数，则尝试避免在 finalizer方法中分配对象。 verbosegc跟踪显示是否正在调用终止函数。
- 避免压缩 — verbosegc跟踪将显示是否发生压缩。压缩通常是由对大量内存分配的请求引起的。分析对大量内存分配的请求并尽可能避免它们；如果它们是大数组，则尝试将它们分割成较小的块。

分配位和标志位

图 3显示与 allocbit 和 markbit 有关的堆，后两者是表示堆上对象状态的两个位向量。因为堆上的所有对象都是从 8 字节边界开始的，两个向量都有一位表示堆的 8 个字节，因此每个向量是堆的 64 分之一。

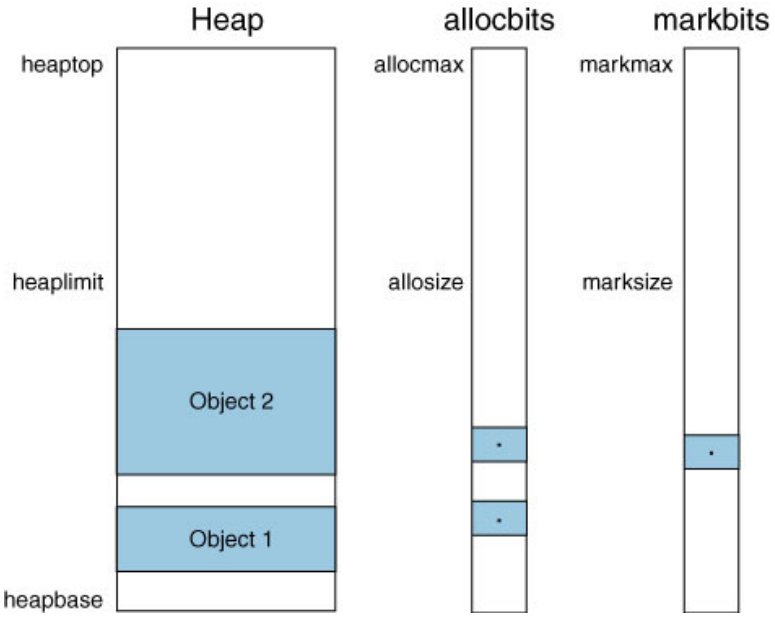
图 3. 带有 allocbit 和 markbit 的堆



因为对象被分配在堆中，所以在 allocbit 中将一位设置为启用以表示对象的起始位置。这个过程告诉我们分配的对象在哪里，但它没有说明对象是否是活动的。在标志阶段，在 markbit 中将一位设置为启用来表示活动对象的起始位置。 图 4显示了了堆上的两个对象，这两个对象的 allocbit 都是启用

的。在标志阶段，发现引用了 Object 2，但未发现引用 Object 1；因此，启用 Object 2 的 markbit。在清理阶段将收集 Object 1。

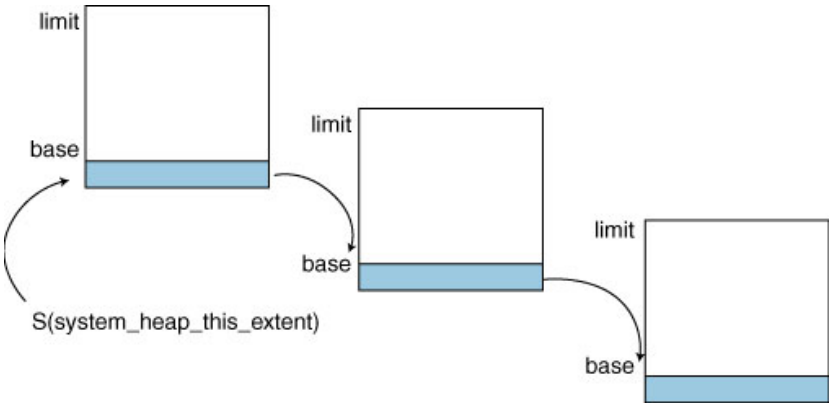
图 4. 堆中的某些对象



系统堆

系统堆是在发行版 1.3.0 中引入的，它只包含期望生命期为 JVM 生命期的对象。这个堆中的对象是系统的类对象和可共享的中间件以及应用程序类。系统堆从不进行垃圾收集，因为其中所有对象要么在 JVM 生命期期间都是可访问的，要么（在可共享的应用程序类情况下）在 JVM 生命期期间已被选中重用。 图 5显示了系统堆布局。

图 5. 系统堆

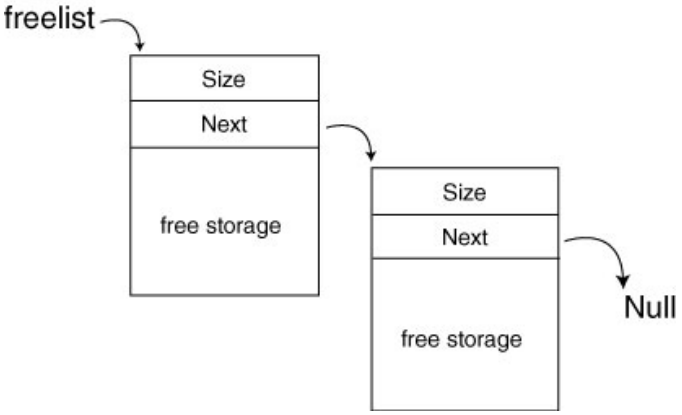


系统堆是一串不连续的存储器块。系统堆的初始大小在 32 位体系结构中是 128k，在 64 位体系结构中是 8M。如果这一范围满了，则系统堆将获取另一个范围然后将各个范围链接起来。

空闲表

图 6显示了空闲表链。表头在全局存储器中并且指向堆上的第一个空闲块。空闲存储器的每一块都有一个 size 字段和一个指向下一个空闲块的指针。空闲块按地址排序。最后一个空闲块具有一个 NULL 指针。

图 6. 空闲表



分配

在本节中讨论两种分配类型： 堆锁和 高速缓存。

堆锁分配

当分配请求大于 512 字节或者现有高速缓存没有足够空间用于分配时发生堆锁分配。顾名思义，堆锁分配需要一个锁，应该通过使用高速缓存来避免它（如果有可能）。下面是发行版 1.3.1 中堆锁分配的一些伪代码。

堆锁分配

```
If size greater than 512 or enough space in cache
 try cacheAlloc
 return if OK
HEAP_LOCK
Do forever
 If there is a big enough chunk on freelist
  Take it
  goto Gotit
 else
  manageAllocFailure
  If any error
   goto GetOut
End do
Gotit:
Initialise object
GetOut:
HEAP_UNLOCK
```

首先检查分配请求的大小，如果它小于 512 或现有的高速缓存有足够空间，则设法使用高速缓存分配进行分配。如果不是使用高速缓存分配，或者如果应用程序无法找到空闲的空间，则会得到 HEAP_LOCK。应用程序现在会搜索空闲表以查找足以满足分配请求的空闲存储器块。如果它发现空闲存储器，则采用之并初始化对象，并将所有余下的空闲存储器返回空闲表。如果余下的空闲存储器少于 512 字节加上头大小（在 32 位体系结构上是 12 字节，在 64 位体系结构上是 24 字节），则不要将它放入空闲表。这些不在空闲表中的存储器小块称为 暗物质，当邻近暗物质的对象变成空闲或者当您压缩堆时，将恢复这些暗物质。如果您无法找到一个足够大的空闲存储块，则分配失败并需要执行 GC。如果 GC 创建了足够的空闲存储器，则您将再次搜索空闲表然后获取空闲块。如果 GC 没有发现足够的空闲存储器，则返回内存不足信息。在我们分配了对象或无法找到足够可用空间之后会释放 HEAP_LOCK。

提示

提示（hint）是在发行版 1.3.0 中引入的。在某些情况下，空闲表在一个应用程序中具有许多小的可用空间，而该应用程序进行许多较大的分配，在这样的大堆中，堆锁分配方案会遇到问题。它总是从头开始搜索大部分长列表以查找足够大的可用空间来满足分配。引入了快速空闲表提示算法来解决这个问题。

对于所有遍历空闲表的堆锁分配尝试，收集了下列数据：

- 在找到足够满足期望的分配大小 n的可用空间之前在空闲表上检查的块数量的搜索计数。
- 在可用空间用于分配之前还记录了空闲表中最大的可用空间块的大小。（换言之，最大块不足以满足需求。）

当找到了能够满足分配的可用空间而且搜索计数大于 20 时，您应该创建一个指向空闲表的新的活动提示。

现在可以使用活动提示在非起始位置来启动搜索空闲表，这取决于分配需求的大小。从空闲表分配块时会动态更新提示。

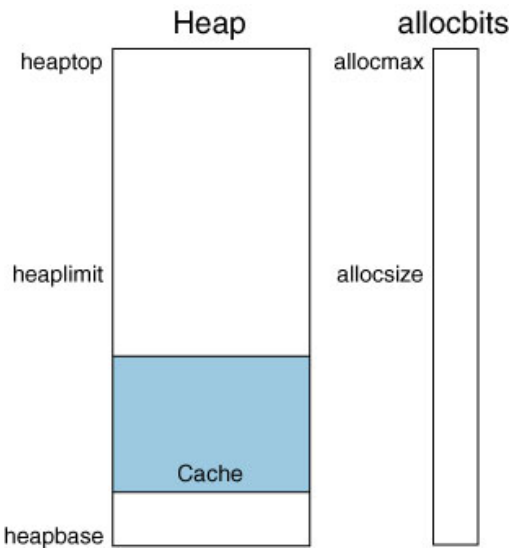
高速缓存分配

高速缓存分配是专门设计来为小对象提供最佳分配性能的。对象是从线程本地分配缓冲区中直接分配的，线程本地分配缓冲区是线程先前从堆中分配的。新对象是从这个高速缓存的末端分配的而不必获取堆锁，因此效率非常高。在各发行版中已经对使用高速缓存分配的标准作了如下更改：

- 1.2.2 和 1.3.0 — 如果对象的大小小于 384，则使用高速缓存分配。
- 1.3.1 — 如果对象的大小小于 512 或者当前高速缓存块中有足够空间可用于对象，则使用高速缓存分配。

下面的 图 7显示了堆上的高速缓存块。

图 7. 堆上的高速缓存块

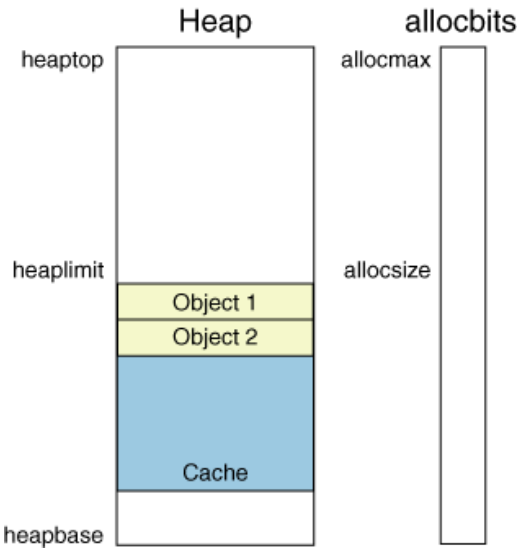


高速缓存块有时称为线程本地堆（TLH）。当为线程分配 TLH 时，我们遍历堆锁定的分配并保留将由单线程专用的堆的一部分。然后所有高速缓存分配就可以转变成 TLH 而不需要任何锁。请注意，对于 TLH，未启用 allocbit。当 TLH 已满或发生 GC 时，我们为 TLH 设置 allocbit。为了提高分配 TLH 的效率，我们总是使空闲表上的下一个空闲块达到最大的大小，各发行版作了如下更改：

- 1.2.2 和 1.3.0 — 6K
- 1.3.1 — 根据 TLH 的使用在 2K 到 164K 之间变化

图 8显示了已被分配到 TLH 中的某些对象。请注意，对象是从 TLH 的后端分配的，这比从前端分配效率更高。仍然没有设置 allocbit；当高速缓存已满或发生 GC 时，将为 TLH 中的所有对象设置它们。

图 8. 高速缓存中的某些对象



请继续关注垃圾收集……

本系列文章的 下一篇文章描述了垃圾收集是如何工作的。它研究了标志、清理、压缩收集程序的功能以及并行标志和并行清理是如何工作的。该文章还讨论了如何缩放堆，并描述了在发行版 1.3.1 中引入的新的 Concurrent Mark 收集程序。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 请通过单击本文顶部或底部的 讨论来参与本文的 论坛。
- Richard Jones 和 Rafael Lins 合著的 Garbage Collection - Algorithms for Automatic Dynamic Memory Management（John Wiley & Son Ltd，1996，ISBN:0-471-94148-4）。
- 在 developerWorks Java 技术专区中可以找到更多 Java 参考资料，该专区包含到
- 请阅读介绍文章 Living with the Garbage Collector（developerWorks，2002 年 1 月）。

关于作者

Sam Borman

Sam Borman 于 1984 年加入 IBM，在此之前他在英国、新西兰和法国的另外七家公司担任程序员和系统程序员。他担任 CICS 领域的开发人员，后来担任开发经理。1990 年，他回到了技术领域从事 CICSPlex/SM 的研究后来又从事 DirectTalk 的研究。1999 年，他加入了 Java 技术中心（Java Technology Centre），在那里他负责“垃圾收集”。可以通过 sambo@uk.ibm.com与 Sam 联系。