再谈 Urban 性能传言

**简介：** Java™ 语言遭到许多性能方面的攻击。虽然有些攻击可能是名符其实的，但是看看公告板和新闻组上关于这一主题的贴子，可以发现，对于 Java 虚拟机（JVM）实际的工作方式存在许多误解。在本月的 Java 理论与实践 中，Brian Goetz 驳斥了反复重复的有关 JVM 分配慢的传言。

流行问题：哪种语言的原始分配性能更快，Java 语言还是 C/C++？答案可能令人惊讶 —— 现代 JVM 中的分配比执行得最好的 malloc 实现还要快得多。HotSpot 1.4.2 之后虚拟机中的 new Object() 常见代码路径最多 10 条机器指令（Sun 提供的数据；请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#resources)），而用 C 语言实现的执行得最好的 malloc 实现，每个调用平均要求的指令在 60 到 100 条之间（Detlefs 等；请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#resources)）。而且分配性能在整体性能中不是一个微不足道的部分，测评显示：对于许多实际的 C 和 C++ 程序（例如 Perl 和 Ghostscript），整体执行时间中的 20% 到 30% 都花在 malloc 和 free 上，远远多于健康的 Java 应用程序在分配和垃圾收集上的开销（Zorn；请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#resources)）。

**继续，弄得一团糟**

没有必要搜索众多的 blog 或 Slashdot 贴子，去寻找像“垃圾收集永远不会像直接内存管理一样有效”这样能够说服人的陈述。而且，从某个方面来说，这些话说的是对的 —— 动态内存管理并不一样快 —— 而是快得多。malloc/free 技术一次处理一个内存块，而垃圾收集机制则采用大批量方式处理内存管理，从而形成更多的优化机会（以一些可以预见到的损失为代价）。

这条“听起来有理的意见” （以大批量清理垃圾要比一天到晚一点点儿清理垃圾更容易）得到了数据的证实。一项研究（Zorn; 请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#resources)）测量了在许多常见 C++ 应用程序中，用保守的 Boehm-Demers-Weiser（BDW）替换 malloc 的效果，结果是：许多程序在采用垃圾收集而不是传统的分配器运行时，表现出了速度提升。（BDW 是个保守的、不移动的垃圾收集器，严重地限制了对分配和回收进行优化的能力，也限制了改善内存位置的能力；像 JVM 中使用的那些精确的浮动收集器可以做得更好。）

在 JVM 中的分配并不总是这么快，早期 JVM 的分配和垃圾收集性能实际上很差，这当然就是 JVM 分配慢这一说法的起源。在非常早的时候，我们看到过许多“分配慢”的意见 —— 因为就像早期 JVM 中的一切一样，它确实慢 —— 而性能顾问提供了许多避免分配的技巧，例如对象池。（公共服务声明：除了对最重量的对象之外，对象池现在对于所有对象都是严重的性能损失，而且要在不造成并发瓶颈的情况下使用对象池也很需要技巧。）但是，从 JDK 1.0 开始已经发生了许多变化；JDK 1.2 中引入的分代收集器（generational collector）支持简单得多的分配方式，可以极大地提高性能。

**分代垃圾收集**

分代垃圾收集器把堆分成多代；多数 JVM 使用两代，“年轻代”和“年老代”。对象在年轻代中分配；如果它们在一定数量的垃圾收集之后仍然存在，就被当作是”长寿的“，并晋升到年老代。

HotSpot 提供了使用三个年轻代收集器的选择（串行拷贝、并行拷贝和并行清理），它们都采用“拷贝”收集器的形式，有几个重要的公共特征。拷贝收集器把内存空间从中间分成两半，每次只使用一半。开始时，使用中的一半构成了可用内存的一个大块；分配器满足分配请求时，返回它没有使用的空间的前 N 个字节，并把指针（分隔“使用”部分）从“自由”部分移动过来，如清单 1 的伪代码所示。当使用的那一半用满时，垃圾收集器把所有活动对象（不是垃圾的那些对象）拷贝到另一半的底部（把堆压缩成连续的），然后从另一半开始分配。

**清单 1. 在存在拷贝收集器的情况下，分配器的行为**

|  |
| --- |
| void \*malloc(int n) {  if (heapTop - heapStart < n)  doGarbageCollection();  void \*wasStart = heapStart;  heapStart += n;  return wasStart;  } |

从这个伪代码可以看出为什么拷贝收集器可以实现这么快的分配 —— 分配新对象只是检查在堆中是否还有足够的剩余空间，如果还有，就移动指针。不需要搜索自由列表、最佳匹配、第一匹配、lookaside 列表 ，只要从堆中取出前 N 个字节，就成功了。

**如何回收？**

但是分配仅仅是内存管理的一半，回收是另一半。对于多数对象来说，直接垃圾收集的成本为零。这是因为，拷贝收集器不需要访问或拷贝死对象，只处理活动对象。所以在分配之后很快就变成垃圾的对象，不会造成收集周期的工作量。

在典型的面向对象程序中，绝大多数对象（根据不同的研究，在 92% 到 98% 之间）“死于年轻”，这意味着它们在分配之后，通常在下一次垃圾收集之前，很快就变成垃圾。（这个属性叫作 *分代假设*，对于许多面向对象语言已经得到实际测试，证明为真。）所以，不仅分配要快，对于多数对象来说，回收也要自由。

**线程本地分配**

如果分配器完全像 [清单 1](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#listing1) 所示的那样实现，那么共享的 heapStart 字段会迅速变成显著的并发瓶颈，因为每个分配都要取得保护这个字段的锁。为了避免这个问题，多数 JVM 采用了 *线程本地分配块*，这时每个线程都从堆中分配一个更大的内存块，然后顺序地用这个线程本地块为小的分配请求提供服务。所以，线程花在获得共享堆锁的大量时间被大大减少，从而提高了并发性。（在传统的 malloc 实现的情况下要解决这个问题更困难，成本更高；把线程支持和垃圾收集都构建进平台促进了这类协作。）

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#ibm-pcon)

**堆栈分配**

C++ 向程序员提供了在堆或堆栈中分配对象的选择。基于堆栈的分配更有效：分配更便宜，回收成本真正为零，而且语言提供了隔离对象生命周期的帮助，减少了忘记释放对象的风险。另一方面，在 C++ 中，在发布或共享基于堆栈的对象的引用时，必须非常小心，因为在堆栈帧整理时，基于堆栈的对象会被自动释放，从而造成孤悬的指针。

基于堆栈的分配的另一个优势是它对高速缓存更加友好。在现代的处理器上，缓存遗漏的成本非常显著，所以如果语言和运行时能够帮助程序实现更好的数据位置，就会提高性能。堆栈的顶部通常在高速缓存中是“热”的，而堆的顶部通常是“冷”的（因为从这部分内存使用之后可能过了很长时间）。所以，在堆上分配对象，比起在堆栈上分配对象，会带来更多缓存遗漏。

更糟的是，在堆上分配对象时，缓存遗漏还有一个特别讨厌的内存交互。在从堆中分配内存时，不管上次使用内存之后留下了什么内容，内存中的内容都被当作垃圾。如果在堆的顶部分配的内存块不在缓存中，执行会在内存内容装入缓存的过程中出现延迟。然后，还要用 0 或其他初始值覆盖掉刚刚费时费力装入缓存的那些值，从而造成大量内存活动的浪费。（有些处理器，例如 Azul 的 Vega，包含加速堆分配的硬件支持。）

**escape 分析**

Java 语句没有提供任何明确地在堆栈上分配对象的方式，但是这个事实并不影响 JVM 仍然可以在适当的地方使用堆栈分配。JVM 可以使用叫作 *escape 分析* 的技术，通过这项技术，JVM 可以发现某些对象在它们的整个生命周期中都限制在单一线程内，还会发现这个生命周期绑定到指定堆栈帧的生命周期上。这样的对象可以安全地在堆栈上而不是在堆上分配。更好的是，对于小型对象，JVM 可以把分配工作完全优化掉，只把对象的字段放入寄存器。

清单 2 显示了一个可以用 escape 分析把堆分配优化掉的示例。Component.getLocation() 方法对组件的位置做了一个保护性的拷贝，这样调用者就无法在不经意间改变组件的实际位置。先调用 getDistanceFrom() 得到另一个组件的位置，其中包括对象的分配，然后用 getLocation() 返回的 Point 的 x 和 y 字段计算两个组件之间的距离。

**清单 2. 返回复合值的典型的保护性拷贝方式**

|  |
| --- |
| public class Point {  private int x, y;  public Point(int x, int y) {  this.x = x; this.y = y;  }  public Point(Point p) { this(p.x, p.y); }  public int getX() { return x; }  public int getY() { return y; }  }  public class Component {  private Point location;  public Point getLocation() { return new Point(location); }  public double getDistanceFrom(Component other) {  Point otherLocation = other.getLocation();  int deltaX = otherLocation.getX() - location.getX();  int deltaY = otherLocation.getY() - location.getY();  return Math.sqrt(deltaX\*deltaX + deltaY\*deltaY);  }  } |

getLocation() 方法不知道它的调用者要如何处理它返回的 Point；有可能得到一个指向 Point 的引用，比如把它放在集合中，所以 getLocation() 采用了保护性的编码方式。但是，在这个示例中，getDistanceFrom() 并不会这么做，它只会使用Point 很短的时间，然后释放它，这看起来像是对完美对象的浪费。

聪明的 JVM 会看出将要进行的工作，并把保护性拷贝的分配优化掉。首先，对 getLocation() 的调用会变成内联的，对getX() 和 getY() 的调用也同样处理，从而导致 getDistanceFrom() 的表现会像清单 3 一样有效。

**清单 3. 伪代码描述了把内联优化应用到 getDistanceFrom() 的结果**

|  |
| --- |
| public double getDistanceFrom(Component other) {  Point otherLocation = new Point(other.x, other.y);  int deltaX = otherLocation.x - location.x;  int deltaY = otherLocation.y - location.y;  return Math.sqrt(deltaX\*deltaX + deltaY\*deltaY);  } |

在这一点上，escape 分析可以显示在第一行分配的对象永远不会脱离它的基本块，而 getDistanceFrom() 也永远不会修改other 组件的状态。（*escape* 指的是对象引用没有保存到堆中，或者传递给可能保留一份拷贝的未知代码。）如果 Point真的是线程本地的，而且也清楚它的生命周期限制在分配它的基本块内，那么它既可以进行堆栈分配，也可以完全优化掉，如清单 4 所示。

**清单 4. 伪代码描述了从 getDistanceFrom() 优化掉分配后的结果**

|  |
| --- |
| public double getDistanceFrom(Component other) {  int tempX = other.x, tempY = other.y;  int deltaX = tempX - location.x;  int deltaY = tempY - location.y;  return Math.sqrt(deltaX\*deltaX + deltaY\*deltaY);  } |

结果就是得到了与所有字段都是 public 时能够得到的相同的性能，同时保持了封装和保护性拷贝（在其他安全编码技术之中）提供的安全性。

**Mustang 中的 escape 分析**

escape 分析是一项被议论了很久的优化，它最后终于出现了：Mustang（Java SE 6）的当前构建中可以做 escape 分析，并在适当的地方把堆分配转换成堆栈分析（或者不分配）。用 escape 分析清除一些分配，会带来更快的平均分配时间，简化的内存工作，更少的缓存遗漏。而且，优化掉一些分配，可以降低垃圾收集器的压力，从而让收集运行得更少。

即使在源代码中进行堆栈分配不太现实的地方，即使语言提供了分配的选项，escape 分析也能找到堆栈分配的机会，因为特定的分配是否会被优化掉，是根据特定代码路径中实际上如何使用对象返回方法的结果而决定的。getLocation() 返回的 Point 可能不是在所有情况下都适合进行堆栈分配，但是一旦 JVM 内联了 getLocation()，它就可以自由而且独立地优化每个调用，从而在两方面都提供了最好的结果：最优的性能，最少的时间花在进行低级的性能调整决策上。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp09275.html#ibm-pcon)

**结束语**

JVM 擅长发现我们一直以为只有开发人员才能知道的事情，这令人震惊。让 JVM 根据具体情况在堆栈分配和堆分配之间进行选择，我们就能得到堆栈分配的性能好处，却不必让程序员在进行堆栈分配还是进行堆分配上费脑筋。