Java的实时垃圾收集

由于垃圾收集（GC）是造成 Java 应用程序不可预测性的最主要因素之一，实时虚拟机（VM）必须找到一种方法来避免垃圾收集暂停造成任务错过其时间期限。然而，应该注意，RTSJ 并没有定义实时 GC。

可以使用多种不同方法在一个实时环境内调度 GC，每种方法各有利弊。其中包括基于工作的和基于时间的增量式收集方法，旨在最小化 GC 对调度的影响。

尽管这两个过程减少了持续时间并改善了特定暂停的可预测性，但是仍然难以投入使用和调优，这限制了它们在硬实时应用中的可靠性。

**基于工作的方法**

基于工作的方法在每次分配对象时，使每个线程执行一定数量的增量式 GC。这种方法具有良好的公平性，因为分配最多内存的行为将支付最大份额的收集成本，并且将收集成本分散为较小的块。

但是这种方法的作用有限，因为它执行固定数量的增量式收集工作所花费的时间是变化的。就是说，它依赖于整个应用程序的分配行为。因此，仍然很难预测垃圾收集对调度的影响。

图 1 展示了基于工作的 GC 的风险：这种可变性会造成一个高优先级任务在分配内存时错过时间期限。

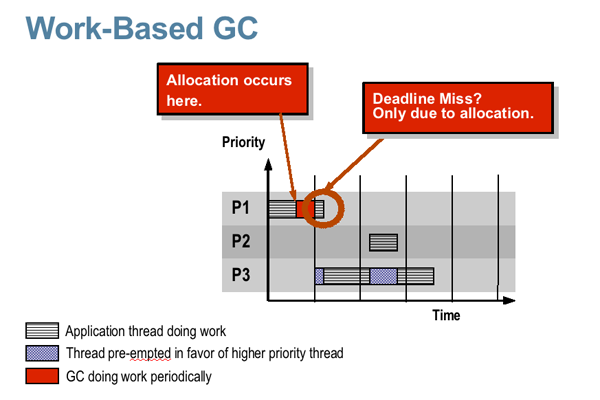


图 1. *基于工作的 GC 的风险*

**基于时间的方法**

和基于工作的方法将增量式收集划分成多个部分不同，基于时间的方法在每次调度期间为 GC 安排固定数量的时间。但是，增量式 GC 运行的时间量和它收回的内存量之间没有直接联系，因此使用固定时间量的方法可能无法跟上应用程序的步调。

图 2 展示了基于时间 GC 的效果，以及它如何造成高优先级线程错过时间期限 —— 即使该线程没有分配内存。

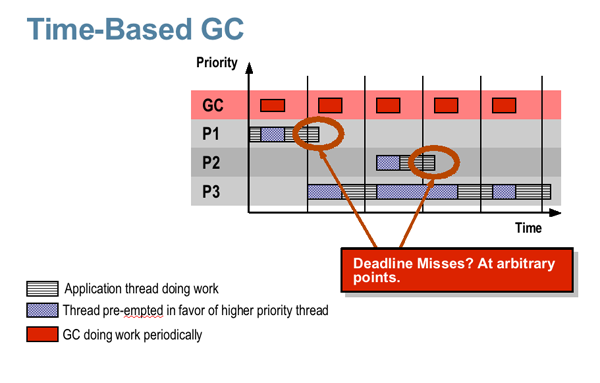


图 2. *基于时间的 GC 和造成的不良后果*

**Henriksson 的 GC**

第三种方法称为 Henriksson 的 GC，它对基于工作的方法进行了修改，这样关键线程中与分配有关的增量式 GC 工作将被推迟到关键任务结束以后。

垃圾收集器运行在比关键线程低的优先级。这使得拥有更低优先级的线程必须承受全部 GC 的代价 —— 假设您在执行高优先级任务期间没有耗光内存。与使用其他实时垃圾收集器相比，拥有更高优先级的线程可以满足更短的时间期限，即使它们分配了内存。

此外，由于延迟的 GC 行为不会影响这些关键线程的响应时间 —— 因而不会影响时间期限，可以为每次分配指定一个对要执行的 GC 工作量的较高估计。这将减少由于糟糕的 GC 配置或分配猛增而引起的内存耗尽的风险。

图 3 展示了这种算法。

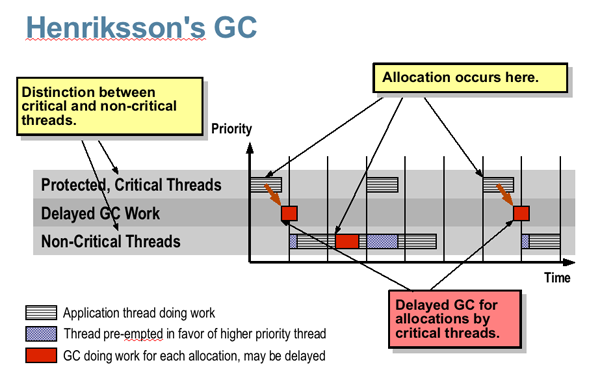


图 3. *Henriksson 的 GC*

**Sun Java 实时系统（Java RTS）中的实时收集**

[Sun Java 实时系统（Java RTS）](http://java.sun.com/javase/technologies/realtime/) 是 Sun 为 Java 实时规范（RTSJ），即 [JSR 1](http://jcp.org/en/jsr/detail?id=1) 提供的商业实现。

从 Sun Java RTS 2.0 开始，引入了一种新的基于 Henriksson 算法的实时垃圾收集器（RTGC）。这种垃圾收集器可以作为一个或多个实时线程（RTT）运行。它所运行的优先级要低于 NoHeapRealtimeThread（[NHRT](http://java.sun.com/developer/technicalArticles/Programming/rt_pt1/index.html#memory_management)）的所有实例，可能还低于某些 RTT，因此关键线程可能会抢占收集器。同这种方式，关键线程就不会受到 GC 的影响。

RTGC 算法的其中一个调优参数是垃圾收集器的最大优先级。这将优先级的范围划分为不同的区间。NHRT 接受最高优先级并且被认为是关键性任务。下一个优先级是关键型实时线程，随后是非关键实时线程，最后一个优先级是非实时线程。默认情况下，垃圾收集器在其 最初的优先级运行，这个优先级要低于非关键型实时线程的优先级。但是，随着内存越来越少，VM 将把垃圾收集器的优先级提升为所配置的最大优先级。

图 4 使用灰色表示不同类型线程的优先级级别，用粉色表示垃圾收集器最初的和最大的优先级级别。

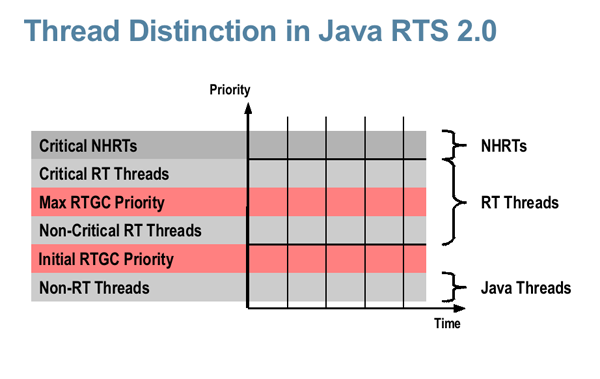


图 4. *Java RTS 中的线程区别*

有关 Java RTS 提供的 RTGC 的一个重点是，它是完全并发性的，因此可以在任何时间被抢占。不需要使用最高优先级运行 RTGC，并且这里不会出现（stop-the-world）阶段，否则在 GC 期间应用程序的所有线程都将被挂停。

在图 5 中，RTGC 使用最初的优先级开始执行，然后被一个高优先级线程抢占。当线程停止后，RTGC 重新运行但仍然被抢占。最后，如果运行的线程正在分配内存，并且剩余内存足够少，那么 RTGC 将被提升为它的最高优先级，此时它只能被关键线程抢占。

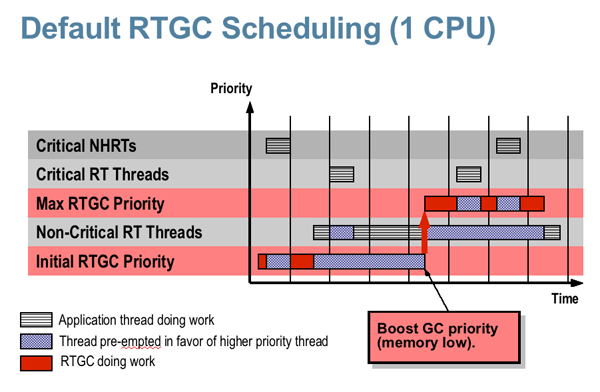


图 5. *默认 RTGC 调度（1 个 CPU）*

在一个多处理器中，一个 CPU 可以执行一些 GC 工作，而另一个应用程序线程在另一个 CPU 中处理。在图 6 中，关键的 NHRT 运行在一个单独的 CPU 中，因此不会抢占 RTGC。RTGC 可以运行至结束，而不会发生优先级提升事件。

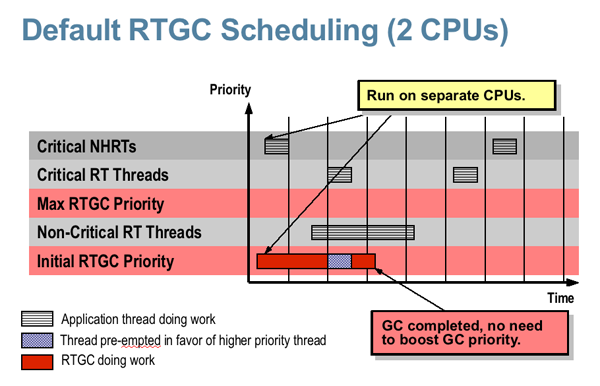


图 6. *默认的 RTGC 调度（2 个 CPU）*

因此，Java RTS 提供的 RTGC 非常灵活。尽管其他实时系统中的垃圾收集器通常必须使用增量方式执行，或者周期性高优先级行为会引入分配时间方面的开销，但是 Java RTS RTGC 可以根据多种不同的调度策略执行。

此外，相比于设法确保应用程序所有线程的确定性，Sun Java RTS 团队在 Henriksson 算法的基础上使用默认的调度策略。

RTGC 认为应用程序线程的关键程度依赖于线程各自的优先级，确保硬实时行为只针对关键的实时线程，而同时设法为关键级别以下的实时线程提供软实时行为。

这减少了 RTGC 的总开销，并确保增加新的低优先级应用程序线程不会影响确定性。这使得配置更加简单，因为配置 RTGC 不需要全面地学习应用程序的分配行为。通常，开发人员通过只查看关键任务设计确定性。

通过只设置两个参数，即内存阈值和优先级，您可以确保 GC 暂停不会中断使用关键优先级运行的线程。这种方法的一大优点是这些参数是独立于应用程序的非关键部分。在添加新的非关键组件或在机器加载类时，您不需要重新配置 RTGC。

RTGC 提供了一直自动调优机制，尝试寻找确定性和吞吐量之间的最佳平衡点。它还尝试在其初始优先级为非关键型实时线程快速重新循环内存，但是不会为它们提供任何保证。

如果非关键型加载增加，RTGC 可能无法为所有线程足够快地重新循环内存，并将提升至它的最大优先级。不过，这并不会打断关键线程，只要为应用程序设置了正确的内存阈值。只有非关键型实时线程将被抢占，并发生暂时性抖动，或内存再循环引起的延迟变化。

然而，请注意，RTGC 并不是一种可以毫无察觉地执行 GC 的 [解决方法](http://en.wikipedia.org/wiki/Silver_bullet)。更确切地说，它是一种可以用来调优调度参数、使关键线程免受 GC 影响的机制。通过选择线程优先级和 RTGC 参数对应用程序调优，可以有效地保护关键线程不受应用程序其他活动的影响。