In last month's Java theory and practice, columnist Brian Goetz reviewed the basic algorithms for garbage collection. This month, he goes a step further and examines how the 1.4.1 JVM actually handles garbage collection, including some of the new garbage collection options for multiprocessor systems.

Last month, we looked at the classic garbage collection techniques of reference counting, copying, mark-sweep, and mark-compact. Each of these approaches has advantages and disadvantages in certain situations. For example, copying does well when a large proportion of objects are garbage, but does poorly with many long-lived objects (copying them repeatedly). Conversely, mark-compact does quite well with long-lived objects (copying them only once), but not so well with many short-lived objects. The technique used by the 1.2 and later JVMs, called generational garbage collection, combines these two techniques to get the best of both worlds, and as a bonus provides very low object allocation overhead.

Old objects, young objects

In any application heap, some objects become garbage shortly after their creation, some survive for a long time and then become garbage, and others can remain live for the entirety of the program's run. Empirical studies have shown that for most object-oriented languages, the Java language included, the vast majority of objects -- as much as 98 percent, depending on your metric for object youth -- die young. One can measure an object's age in wall-clock seconds, in total bytes allocated by the memory management subsystem since the object was allocated, or the number of garbage collections since the object was allocated. But no matter how you measure, the studies show the same thing -- most objects die young. The fact that most objects die young has significance for the choice of collector. In particular, copying collectors perform quite well when the majority of objects die young, since copying collectors do not visit dead objects at all; they simply copy the live objects to another heap region, then reclaim the whole of the remaining space in one fell swoop.
Of the objects that survive past their first collection, a significant portion of those will become long-lived or permanent. The various garbage collection strategies perform very differently depending on the mix of short-lived and long-lived objects. Copying collectors work very well when most objects die young, because objects that die young never need to be copied at all. However, the copying collector deals poorly with long-lived objects, repeatedly copying them back and forth from one semi-space to another. Conversely, mark-compact collectors do very well with long-lived objects, because long-lived objects tend to accumulate at the bottom of the heap and then do not need to be copied again. Mark-sweep and mark-compact collectors, however, expend considerably more effort examining dead objects, because they must examine every object in the heap during the sweep phase.

**Generational collection**

A generational collector divides the heap into multiple generations. Objects are created in the young generation, and objects that meet some promotion criteria, such as having survived a certain number of collections, are then promoted to the next older generation. A generational collector is free to use a different collection strategy for different generations and perform garbage collection on the generations separately.

**Minor collections**

One of the advantages of generational collection is that it can make garbage collection pauses shorter by not collecting all generations at once. When the allocator is unable to fulfill an allocation request, it first triggers a minor collection, which only collects the youngest generation. Since many of the objects in the young generation will already be dead and the copying collector does not need to examine dead objects at all, minor collection pauses can be quite short and can often reclaim significant heap space. If the minor collection frees enough heap space, the user program can resume immediately. If it does not free enough heap space, it proceeds to collect higher generations until enough memory has been reclaimed. (In the event the garbage collector cannot reclaim enough memory after a full collection, it will either expand the heap or it will throw an `OutOfMemoryError`.)

**Intergenerational references**

Tracing garbage collectors, such as copying, mark-sweep, and mark-compact, all start scanning from the root set, traversing references between objects, until all live objects have been visited.

A generational tracing collector starts from the root set, but does not traverse references that lead to objects in the older generation, which reduces the size of the object graph to be traced. But this creates a problem -- what if an object in the older generation references a younger object, which is not reachable through any other chain of references from a root?

To address this problem, generational collectors must explicitly track references from older objects to younger objects and add these old-to-young references into the root set of the minor collection. There are two ways to create a reference from an old object to a young object. Either one of the references contained in an old object is modified to refer to a young object, or a young object that refers to other young objects is promoted into the older generation.
Tracking intergenerational references

Whether an old-to-young reference is created by promotion or a pointer modification, the garbage collector needs to have a comprehensive set of old-to-young references when it wants to perform a minor collection. One way to do this would be to trace the old generation, but this clearly has significant overhead. Somewhat better would be to linearly scan the old generation looking for references to young objects. This approach is faster than tracing and has better locality, but is still considerable work.

The mutator and garbage collector can work together to maintain a comprehensive list of old-to-young references as they are created. When objects are promoted into the older generation, the garbage collector can note any old-to-young references that are created as a result of the promotion, which leaves only the tracking of intergenerational references created by pointer modifications.

The garbage collector can track old-to-young references that arise through modifying references held within existing objects in several ways. It could track them in the same manner as maintaining reference counts in reference-counting collectors (the compiler could generate additional instructions surrounding pointer assignments) or could use virtual memory protection on the old generation heap to trap writes to older objects. Another potentially more efficient virtual memory approach would be to use the page modification dirty bits in the old generation heap to identify blocks to scan for objects containing old-to-young pointers.

With a little cleverness, it is possible to avoid the overhead of tracking every pointer modification and inspecting it to see if it crosses generational boundaries. For example, it is not necessary to track stores to local or static variables, because they will already be part of the root set. It may also be possible to avoid tracking pointer stores within constructors that simply initialize fields of newly created objects (so-called initializing stores), as (almost) all objects are allocated in the young generation. In any case, the runtime must maintain an explicit set of references from old objects to young ones and add these references to the root set when collecting the young generation.

In Figure 1, the arrows represent references between objects in the heap. The red arrows represent old-to-young references that must be added to the root set for a minor collection. The blue arrows represent references to old objects, either from the root set or from the young generation, which don't need to be traced when collecting only the young generation.
Card marking

The Sun JDKs use an optimized variant of an algorithm called *card marking* to identify modifications to pointers held in fields of old-generation objects. In this approach, the heap is divided into a set of *cards*, each of which is usually smaller than a memory page. The JVM maintains a card map, with one bit (or byte, in some implementations) corresponding to each card in the heap. Each time a pointer field in an object in the heap is modified, the corresponding bit in the card map for that card is set. At garbage collection time, the mark bits associated with cards in the old generation are examined, and dirty cards are scanned for objects containing references into the younger generation. Then the mark bits are cleared. There are several costs to card marking – additional space for the card map, additional work to be done on each pointer store, and additional work to be done at garbage collection time. Card marking algorithms can add as little as two or three machine instructions per non-initializing heap pointer store, and entails scanning any objects on dirty cards at minor collection time.

The JDK 1.4.1 default collector

By default, the 1.4.1 JDK divides the heap into two sections, a young generation and an old generation. (Actually, there is also a third section, the permanent space, which is used for storing loaded class and method objects.) The young generation is divided into a creation space, often called *Eden*, and two survivor semi-spaces, using a copying collector.

The old generation uses a mark-compact collector. Objects are promoted from the young generation to the old generation after they have survived copying a certain number of times. A minor collection will copy live objects from Eden and one of the survivor semi-spaces into the other survivor space, potentially promoting some objects to the older generation. A major collection will collect both the young and old generation. The `System.gc()` method always triggers a major collection, which is one of the reasons you should use `System.gc()` sparingly, if at all, because major collections can take much longer than a minor collection. There is no way to programmatically trigger a minor collection.

Other collection options

In addition to the copying and mark-compact collectors used by default, the 1.4.1 JDK also contains four other garbage collection algorithms, each of which is suited to a different purpose. JDK 1.4.1 includes an incremental collector (which has been around since JDK 1.2) and three new collectors for more efficient collection on multiprocessor systems -- the parallel copying collector, the parallel scavenging collector, and the concurrent mark-sweep collector. These new collectors address the problem of the garbage collector being a scalability bottleneck on multiprocessor systems. Figure 2 shows some guidelines on when to choose alternate collection options.
Incremental collection

The incremental collection option has been part of the JDK since 1.2. Incremental collection reduces garbage collection pauses at the expense of throughput, making it desirable only in cases where shorter collection pauses are very important, such as near-realtime systems.

The algorithm used by the JDK for incremental collection, the Train algorithm, creates a new section of the heap between the old and young generations. These heap sections are divided into "trains," each of which is divided into a sequence of "cars." Each car can be collected separately. Effectively, each train car constitutes a separate generation, which means that not only must old-to-young references be tracked, but also references from older trains to younger trains and older cars to younger cars. This creates significant extra work for the mutator and garbage collector, but permits much shorter collection pauses.

Parallel and concurrent collectors

The new collectors in JDK 1.4.1 are all designed to address the issue of garbage collection on multiprocessor systems. Because most garbage collection algorithms stop the world for some period of time, a single-threaded garbage collector can quickly become a scalability bottleneck, because all but one processor are idle while the garbage collector has suspended the user program threads. Two of the new collectors are designed to reduce collection pause time -- the parallel copying collector and the concurrent mark-sweep collector. The other, the parallel scavenging collector, is designed for higher throughput on large heaps.

The parallel copying collector, enabled by the -XX:+UseParNewGC JVM option, is a young-generation copying collector that divides the work of garbage collection among as many threads as there are CPUs. The concurrent mark-sweep collector, enabled by the -XX:+UseConcMarkSweepGC option, is an old-generation mark-sweep collector that stops the world briefly for an initial mark phase (and again later for a brief re-mark phase) and then allows the user program to resume while the garbage collector threads execute concurrently with the user program. The parallel copying collector and concurrent mark-sweep collector are basically concurrent versions of the default copying and mark-compact collectors. The parallel scavenging collector, enabled by -XX:+UseParallelGC, is a young-generation collector optimized for very large (gigabyte and larger) heaps on multiprocessor systems.
Selecting an algorithm

With six algorithms to choose from, you're probably wondering which one to use. Figure 2 offers some guidance, dividing collectors into single-threaded and concurrent, and into low-pause and high-throughput. Given what you know about your application and deployment environment, this may be enough to select the appropriate algorithm. For many applications, the default collectors work just fine -- so if you don't have a performance problem, there's no point in inviting more complexity. However, if your application is deployed on a multiprocessor system or uses a very large heap, you may get some performance boost from changing collector options.

Tuning the garbage collector

JDK 1.4.1 also includes numerous options for tuning garbage collection. You can spend considerable time tweaking these options and measuring their effect, so before you try and tune the garbage collector, you'll probably get a better return on your tuning investment by making sure your application has been thoroughly profiled and optimized first.

The first place to start with tuning garbage collection is to examine the verbose GC output. This will give you information on the frequency, timing, and duration of garbage collection operations. The simplest form of garbage collection tuning is simply expanding the maximum heap size (-Xmx). Copying collection becomes more efficient as the heap grows, so by expanding the heap, you reduce the cost of collection on a per-object basis. In addition to increasing the maximum heap size, you can also use the -XX:NewRatio option to increase the proportion of space assigned to the young generation. You also can specify the size of the young generation explicitly with the -Xmn option. See Related topics for a number of articles offering more detailed garbage collection tuning advice.

Summary

As the JVM has evolved, the default garbage collector has gotten better and better. The generational collector employed by JDK 1.2 and later offers far better allocation and collection performance than the mark-sweep-compact collector used by earlier JDKs. The 1.4.1 JDK further improves the effectiveness of garbage collection by adding new multithreaded collection options for multiprocessor systems and very large heaps.

Next month, we'll wrap up our exploration of garbage collection by looking at some performance hints and myths about garbage collection, including the true cost of object allocation, the costs and benefits of explicit nulling, and the cost of finalization.
**Related topics**

- The Garbage Collection mailing list maintains a garbage collection FAQ.
- This article from *IBM Systems Journal* describes some of the lessons learned in building the IBM 1.1.x Developer Kits for the Java platform, including the details of mark-sweep and mark-sweep-compact garbage collection.
- This document from Sun offers an introduction to the process of garbage collection performance tuning.
- In the paper "A fast write barrier for generational garbage collectors", Urs Hoeltze covers both the classical card-marking algorithm and an improvement that can reduce the cost of marking significantly by slightly increasing the cost of scanning dirty cards at collection time.
- This paper describes the details of the Train incremental garbage collection algorithm.

© Copyright IBM Corporation 2003
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)