developerWorks®

Java theory and practice: Garbage collection and performance

Hints, tips, and myths about writing garbage collection-friendly classes

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

27 January 2004

The past two installments of *Java theory and practice* have discussed various techniques for garbage collection and the basics of the JDK 1.4.1 garbage collectors. This month, columnist Brian Goetz looks at the performance impact of the choice of collector, how various coding idioms interact with the garbage collector, and how allocation and other related costs have changed in Java virtual machines over the past several years.

View more content in this series

In the early days of Java technology, allocating objects got a pretty bad rap. There were lots of articles (including some by this author) advising developers to avoid creating temporary objects unnecessarily because allocation (and the corresponding garbage-collection overhead) was expensive. While this used to be good advice (in situations where performance was significant), it is no longer generally applicable to all but the most performance-critical situations.

How expensive is allocation?

The 1.0 and 1.1 JDKs used a mark-sweep collector, which did compaction on some -- but not all -- collections, meaning that the heap might be fragmented after a garbage collection. Accordingly, memory allocation costs in the 1.0 and 1.1 JVMs were comparable to that in C or C++, where the allocator uses heuristics such as "first-first" or "best-fit" to manage the free heap space. Deallocation costs were also high, since the mark-sweep collector had to sweep the entire heap at every collection. No wonder we were advised to go easy on the allocator.

In HotSpot JVMs (Sun JDK 1.2 and later), things got a *lot* better -- the Sun JDKs moved to a generational collector. Because a copying collector is used for the young generation, the free space in the heap is always contiguous so that allocation of a new object from the heap can be done through a simple pointer addition, as shown in Listing 1. This makes object allocation in Java applications significantly cheaper than it is in C, a possibility that many developers at first have difficulty imagining. Similarly, because copying collectors do not visit dead objects, a heap with a large number of temporary objects, which is a common situation in Java applications, costs very little to collect; simply trace and copy the live objects to a survivor space and reclaim the entire heap in one fell swoop. No free lists, no block coalescing, no compacting -- just wipe the heap clean and start over. So both allocation and deallocation costs per object went way down in JDK 1.2.

Listing 1. Fast allocation in a contiguous heap

```
void *malloc(int n) {
  synchronized (heapLock) {
    if (heapTop - heapStart > n)
      doGarbageCollection();

    void *wasStart = heapStart;
    heapStart += n;
    return wasStart;
  }
}
```

Performance advice often has a short shelf life; while it was once true that allocation was expensive, it is now no longer the case. In fact, it is downright cheap, and with a few very compute-intensive exceptions, performance considerations are generally no longer a good reason to avoid allocation. Sun estimates allocation costs at approximately *ten machine instructions*. That's pretty much free -- certainly no reason to complicate the structure of your program or incur additional maintenance risks for the sake of eliminating a few object creations.

Of course, allocation is only half the story -- most objects that are allocated are eventually garbage collected, which also has costs. But there's good news there, too. The vast majority of objects in most Java applications become garbage before the next collection. The cost of a minor garbage collection is proportional to the number of live objects in the young generation, not the number of objects allocated since the last collection. Because so few young generation objects survive to the next collection, the amortized cost of collection per allocation is fairly small (and can be made even smaller by simply increasing the heap size, subject to the availability of enough memory).

But wait, it gets better

The JIT compiler can perform additional optimizations that can reduce the cost of object allocation to zero. Consider the code in Listing 2, where the `getPosition()` method creates a temporary object to hold the coordinates of a point, and the calling method uses the `Point` object briefly and then discards it. The JIT will likely inline the call to `getPosition()` and, using a technique called *escape analysis*, can recognize that no reference to the `Point` object leaves the `doSomething()` method. Knowing this, the JIT can then allocate the object on the stack instead of the heap or, even better, optimize the allocation away completely and simply hoist the fields of the Point into registers. While the current Sun JVMs do not yet perform this optimization, future JVMs probably will. The fact that allocation can get even cheaper in the future, with no changes to your code, is just one more reason not to compromise the correctness or maintainability of your program for the sake of avoiding a few extra allocations.

Listing 2. Escape analysis can eliminate many temporary allocations entirely

```
void doSomething() {
  Point p = someObject.getPosition();
  System.out.println("Object is at (" + p.x, + ", " + p.y + ")");
}

...

Point getPosition() {
  return new Point(myX, myY);
}
```

Isn't the allocator a scalability bottleneck?

Listing 1 shows that while allocation itself is fast, access to the heap structure must be synchronized across threads. So doesn't that make the allocator a scalability hazard? There are several clever tricks JVMs use to reduce this cost significantly. IBM JVMs use a technique called *thread-local heaps*, by which each thread requests a small block of memory (on the order of 1K) from the allocator, and small object allocations are satisfied out of that block. If the program requests a larger block than can be satisfied using the small thread-local heap, then the global allocator is used to either satisfy the request directly or to allocate a new thread-local heap. By this technique, a large percentage of allocations can be satisfied without contending for the shared heap lock. (Sun JVMs use a similar technique, instead using the term "Local Allocation Blocks.")

Finalizers are not your friend

Objects with finalizers (those that have a non-trivial `finalize()` method) have significant overhead compared to objects without finalizers, and should be used sparingly. Finalizeable objects are both slower to allocate and slower to collect. At allocation time, the JVM must register any finalizeable objects with the garbage collector, and (at least in the HotSpot JVM implementation) finalizeable objects must follow a slower allocation path than most other objects. Similarly, finalizeable objects are slower to collect, too. It takes at least two garbage collection cycles (in the best case) before a finalizeable object can be reclaimed, and the garbage collector has to do extra work to invoke the finalizer. The result is more time spent allocating and collecting objects and more pressure on the garbage collector, because the memory used by unreachable finalizeable objects is retained longer. Combine that with the fact that finalizers are not guaranteed to run in any predictable timeframe, or even at all, and you can see that there are relatively few situations for which finalization is the right tool to use.

If you *must* use finalizers, there are a few guidelines you can follow that will help contain the damage. Limit the number of finalizeable objects, which will minimize the number of objects that have to incur the allocation and collection costs of finalization. Organize your classes so that finalizeable objects hold no other data, which will minimize the amount of memory tied up in finalizeable objects after they become unreachable, as there can be a long delay before they are actually reclaimed. In particular, beware when extending finalizeable classes from standard libraries.

Helping the garbage collector . . . not

Because allocation and garbage collection at one time imposed significant performance costs on Java programs, many clever tricks were developed to reduce these costs, such as object pooling and nulling. Unfortunately, in many cases these techniques can do more harm than good to your program's performance.

Object pooling

Object pooling is a straightforward concept -- maintain a pool of frequently used objects and grab one from the pool instead of creating a new one whenever needed. The theory is that pooling spreads out the allocation costs over many more uses. When the object creation cost is high, such as with database connections or threads, or the pooled object represents a limited and costly resource, such as with database connections, this makes sense. However, the number of situations where these conditions apply is fairly small.

In addition, object pooling has some serious downsides. Because the object pool is generally shared across all threads, allocation from the object pool can be a synchronization bottleneck. Pooling also forces you to manage deallocation explicitly, which reintroduces the risks of dangling pointers. Also, the pool size must be properly tuned to get the desired performance result. If it is too small, it will not prevent allocation; and if it is too large, resources that could get reclaimed will instead sit idle in the pool. By tying up memory that could be reclaimed, the use of object pools places additional pressure on the garbage collector. Writing an effective pool implementation is not simple.

In his "Performance Myths Exposed" talk at JavaOne 2003, Dr. Cliff Click offered concrete benchmarking data showing that object pooling is a performance loss for all but the most heavyweight objects on modern JVMs. Add in the serialization of allocation and the dangling-pointer risks, and it's clear that pooling should be avoided in all but the most extreme cases.

Explicit nulling

Explicit nulling is simply the practice of setting reference objects to null when you are finished with them. The idea behind nulling is that it assists the garbage collector by making objects unreachable earlier. Or at least that's the theory.

There is *one* case where the use of explicit nulling is not only helpful, but virtually required, and that is where a reference to an object is scoped more broadly than it is used or considered valid by the program's specification. This includes cases such as using a static or instance field to store a reference to a temporary buffer, rather than a local variable, or using an array to store references that may remain reachable by the runtime but not by the implied semantics of the program. Consider the class in Listing 3, which is an implementation of a simple bounded stack backed by an array. When `pop()` is called, without the explicit nulling in the example, the class could cause a memory leak (more properly called "unintentional object retention," or sometimes called "object loitering") because the reference stored in `stack[top+1]` is no longer reachable by the program, but still considered reachable by the garbage collector.

Listing 3. Avoiding object loitering in a stack implementation

```
public class SimpleBoundedStack {
  private static final int MAXLEN = 100;
  private Object stack[] = new Object[MAXLEN];
  private int top = -1;

  public void push(Object p) { stack [++top] = p;}

  public Object pop() {
    Object p = stack [top];
    stack [top--] = null;  // explicit null
    return p;
  }
}
```

In the September 1997 "Java Developer Connection Tech Tips" column (see Resources), Sun warned of this risk and explained how explicit nulling was needed in cases like the `pop()` example above. Unfortunately, programmers often take this advice too far, using explicit nulling in the hope of helping the garbage collector. But in most cases, it doesn't help the garbage collector at all, and in some cases, it can actually hurt your program's performance.

Consider the code in Listing 4, which combines several really bad ideas. The listing is a linked list implementation that uses a finalizer to walk the list and null out all the forward links. We've already discussed why finalizers are bad. This case is even worse because now the class is doing extra work, ostensibly to help the garbage collector, but that will not actually help -- and might even hurt. Walking the list takes CPU cycles and will have the effect of visiting all those dead objects and pulling them into the cache -- work that the garbage collector might be able to avoid entirely, because copying collectors do not visit dead objects at all. Nulling the references doesn't help a tracing garbage collector anyway; if the head of the list is unreachable, the rest of the list won't be traced anyway.

Listing 4. Combining finalizers and explicit nulling for a total performance disaster -- don't do this!

```
public class LinkedList {

  private static class ListElement {
    private ListElement nextElement;
    private Object value;
  }

  private ListElement head;

  ...

  public void finalize() {
    try {
      ListElement p = head;
      while (p != null) {
        p.value = null;
        ListElement q = p.nextElement;
        p.nextElement = null;
        p = q;
      }
      head = null;
    }
    finally {
      super.finalize();
    }
  }
}
```

Explicit nulling should be saved for cases where your program is subverting normal scoping rules for performance reasons, such as the stack example in Listing 3 (a more correct -- but poorly performing -- implementation would be to reallocate and copy the stack array each time it is changed).

Explicit garbage collection

A third category where developers often mistakenly think they are helping the garbage collector is the use of `System.gc()`, which triggers a garbage collection (actually, it merely suggests that this might be a good time for a garbage collection). Unfortunately, `System.gc()` triggers a full collection, which includes tracing all live objects in the heap and sweeping and compacting the old generation. This can be a lot of work. In general, it is better to let the system decide when it needs to collect the heap, and whether or not to do a full collection. Most of the time, a minor collection will do the job. Worse, calls to `System.gc()` are often deeply buried where developers may be unaware of their presence, and where they might get triggered far more often than necessary. If you are concerned that your application might have hidden calls to `System.gc()` buried in libraries, you can invoke the JVM with the `-XX:+DisableExplicitGC` option to prevent calls to `System.gc()` and triggering a garbage collection.

Immutability, again

No installment of *Java theory and practice* would be complete without some sort of plug for immutability. Making objects immutable eliminates entire classes of programming errors. One of the most common reasons given for not making a class immutable is the belief that doing so would compromise performance. While this is true sometimes, it is often not -- and sometimes the use of immutable objects has significant, and perhaps surprising, performance advantages.

Many objects function as containers for references to other objects. When the referenced object needs to change, we have two choices: update the reference (as we would in a mutable container class) or re-create the container to hold a new reference (as we would in an immutable container class). Listing 5 shows two ways to implement a simple holder class. Assuming the containing object is small, which is often the case (such as a `Map.Entry` element in a `Map` or a linked list element), allocating a new immutable object has some hidden performance advantages that come from the way generational garbage collectors work, having to do with the relative age of objects.

Listing 5. Mutable and immutable object holders

```
public class MutableHolder {
  private Object value;
  public Object getValue() { return value; }
  public void setValue(Object o) { value = o; }
}

public class ImmutableHolder {
  private final Object value;
  public ImmutableHolder(Object o) { value = o; }
  public Object getValue() { return value; }
}
```

In most cases, when a holder object is updated to reference a different object, the new referent is a young object. If we update a `MutableHolder` by calling `setValue()`, we have created a situation where an older object references a younger one. On the other hand, by creating a new `ImmutableHolder` object instead, a younger object is referencing an older one. The latter situation, where most objects point to older objects, is much more gentle on a generational garbage collector. If a `MutableHolder` that lives in the old generation is mutated, all the objects on the card that contain the `MutableHolder` must be scanned for old-to-young references at the next minor collection. The use of mutable references for long-lived container objects increases the work done to track old-to-young references at collection time. (See last month's article and this month's Resources, which explain the card-marking algorithm used to implement the write barrier in the generational collector used by current Sun JVMs).

When good performance advice goes bad

A cover story in the July 2003 *Java Developer's Journal* illustrates how easy it is for good performance advice to become bad performance advice by simply failing to adequately identify the conditions under which the advice should be applied or the problem it was intended to solve. While the article contains some useful analysis, it will likely do more harm than good (and, unfortunately, far too much performance-oriented advice falls into this same trap).

The article opens with a set of requirements from a realtime environment, where unpredictable garbage collection pauses are unacceptable and there are strict operational requirements on how long a pause can be tolerated. The authors then recommend nulling references, object pooling, and scheduling explicit garbage collection to meet the performance goals. So far, so good -- they had a problem and they figured out what they had to do to solve it (although they appear to have failed to identify what the costs of these practices were or explore some less intrusive alternatives, such as concurrent collection). Unfortunately, the article's title ("Avoid Bothersome Garbage Collection Pauses") and presentation suggest that this advice would be useful for a wide range of applications -- perhaps *all* Java applications. This is terrible, dangerous performance advice!

For most applications, explicit nulling, object pooling, and explicit garbage collection will harm the throughput of your application, not improve it -- not to mention the intrusiveness of these techniques on your program design. In certain situations, it may be acceptable to trade throughput for predictability -- such as real-time or embedded applications. But for many Java applications, including most server-side applications, you probably would rather have the throughput.

The moral of the story is that performance advice is highly situational (and has a short shelf life). Performance advice is by definition reactive -- it is designed to address a particular problem that occurred in a particular set of circumstances. If the underlying circumstances change, or they are simply not applicable to your situation, the advice may not be applicable, either. Before you muck up your program's design to improve its performance, first make sure you have a performance problem and that following the advice will solve that problem.

Summary

Garbage collection has come a long way in the last several years. Modern JVMs offer fast allocation and do their job fairly well on their own, with shorter garbage collection pauses than in previous JVMs. Tricks such as object pooling or explicit nulling, which were once considered sensible techniques for improving performance, are no longer necessary or helpful (and may even be harmful) as the cost of allocation and garbage collection has been reduced considerably.

Resources

- The previous two installments of *Java theory and practice*, "A brief history of garbage collection" and "Garbage collection in the 1.4.1 JVM," cover some of the basics of garbage collection in Java virtual machines.
- *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* (John Wiley & Sons, 1997) is a comprehensive survey of garbage collection algorithms, with an extensive bibliography. The author, Richard Jones, maintains an updated bibliography of nearly 2000 papers on garbage collection on his Garbage Collection Page.
- The Garbage Collection mailing list maintains a GC FAQ.
- The IBM 1.4 SDK for the Java plaform uses a mark-sweep-compact collector, which supports incremental compaction to reduce pause times.
- The three-part series, Sensible sanitation by Sam Borman (*developerWorks*, August 2002), describes the garbage collection strategy employed by the IBM 1.2 and 1.3 SDKs for the Java platform.
- This article from the *IBM Systems Journal* describes some of the lessons learned building the IBM 1.1.x JDKs, including the details of mark-sweep and mark-sweep-compact garbage collection.
- The example in Listing 3 was raised by Sun in a 1997 Tech Tip.
- The paper "Removing GC Sychronisation" is a nice survey of potential scalability bottlenecks in garbage collection implementations.
- 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.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.
- Browse for books on these and other technical topics.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications.