Java theory and practice: Plugging memory leaks with soft references

Soft references provide for quick-and-dirty caching

Level: Intermediate

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

24 Jan 2006

In the previous installment of *Java theory and practice*, Java™ sanitation engineer Brian Goetz examined *weak references*, which let you put the garbage collector on notice that you want to maintain a reference to an object without preventing it from being garbage collected. This month, he explains another form of `Reference` object, *soft references*, which can also enlist the aid of the garbage collector in managing memory usage and eliminating potential memory leaks.

Garbage collection might make Java programs immune to memory leaks, at least for a sufficiently narrow definition of "memory leak," but that doesn't mean we can totally ignore the issue of object lifetime in Java programs. Memory leaks in Java programs usually arise when we pay insufficient attention to object lifecycle or subvert the standard mechanisms for managing object lifecycle. For example, last time we saw how failing to demarcate an object's lifecycle could cause unintentional object retention when trying to associate metadata with transient objects. There are other idioms that can similarly ignore or subvert object lifecycle management and can also lead to memory leaks.

Object loitering

A form of memory leak, sometimes called *object loitering*, is illustrated by the `LeakyChecksum` class in Listing 1, which provides a `getFileChecksum()` method to calculate the checksum of a file's contents. The `getFileChecksum()` method reads the contents of the file into a buffer to compute the checksum. A more straightforward implementation would simply allocate the buffer as a local variable within `getFileChecksum()`, but this version is more "clever" than that, instead caching the buffer in an instance field to reduce memory churn. This "optimization" often does not deliver the expected savings; object allocation is cheaper than many give it credit for. (Also note that promoting the buffer from a local variable to an instance variable renders the class no longer thread-safe without additional synchronization; the straightforward implementation would not require `getFileChecksum()` to be declared `synchronized` and would offer better scalability when called concurrently.)

Listing 1. Class that exhibits "object loitering"

```
// BAD CODE - DO NOT EMULATE
public class LeakyChecksum {
    private byte[] byteArray;

    public synchronized int getFileChecksum(String fileName) {
        int len = getFileSize(fileName);
        if (byteArray == null || byteArray.length < len)
            byteArray = new byte[len];
        readFileContents(fileName, byteArray);
        // calculate checksum and return it
    }
}
```

This class has plenty of problems, but let's focus on the memory leak. The decision to cache the buffer most likely followed from the assumption that it would be called many times within a program and that it would therefore be more efficient to reuse the buffer rather than reallocate it. But as a result, the buffer is never

released because it is always reachable by the program (unless the `LeakyChecksum` object is garbage collected). Worse, while it can grow, it cannot shrink, so `LeakyChecksum` permanently retains a buffer as large as the largest file processed. At the very least, this puts pressure on the garbage collector and requires more frequent collections; keeping a large buffer around for the purpose of computing future checksums may not be the most efficient use of available memory.

The cause of the problem in `LeakyChecksum` is that the buffer is logically local to the `getFileChecksum()` operation, but its lifecycle has been artificially prolonged by promoting it to an instance field. As a result, the class has to manage the lifecycle of the buffer itself rather than letting the JVM do it.

Soft references

In the previous installment, we saw how weak references can provide an application with an alternate means of reaching an object while it is used by the program, but without prolonging its lifetime. Another subclass of `Reference`, soft references, fulfills a different but related purpose. Where weak references allow the application to create references that do not interfere with garbage collection, soft references allow the application to enlist the aid of the garbage collector by designating some objects as "expendable." While the garbage collector does a good job at figuring out what memory the application is using and not using, it is up to the application to decide the most appropriate use for the available memory. If the application makes poor decisions as to what objects to retain, performance can suffer as the garbage collector has to work harder to prevent the application from running out of memory.

Caching is a common performance optimization, allowing the application to reuse the results of a previous calculation instead of recalculating it. Caching is a trade-off between CPU utilization and memory usage, and the ideal balance for this trade-off depends on how much memory is available. With too little caching, the desired performance benefit will not be achieved; with too much, performance may suffer because too much memory is being expended on caching and therefore not enough is available for other purposes. Because the garbage collector is generally in a better position than the application to determine the demand for memory, it makes sense to enlist the garbage collector's help in making these decisions, which is what soft references do for us.

If the only remaining references to an object are weak or soft references, then that object is said to be *softly reachable*. The garbage collector does not aggressively collect softly reachable objects the way it does with weakly reachable ones -- instead it only collects softly reachable objects if it really "needs" the memory. Soft references are a way of saying to the garbage collector, "As long as memory isn't too tight, I'd like to keep this object around. But if memory gets really tight, go ahead and collect it and I'll deal with that." The garbage collector is required to clear all soft references before it can throw `OutOfMemoryError`.

We can fix the problems in `LeakyChecksum` by using a soft reference to manage the cached buffer, as shown in Listing 2. Now the buffer is retained as long as the memory is not badly needed, but it can be reclaimed by the garbage collector if necessary:

Listing 2. Fixing LeakyChecksum with a soft reference

```
public class CachingChecksum {
    private SoftReference<byte[]> bufferRef;

    public synchronized int getFileChecksum(String fileName) {
        int len = getFileSize(fileName);
        byte[] byteArray = bufferRef.get();
        if (byteArray == null || byteArray.length < len) {
            byteArray = new byte[len];
            bufferRef.set(byteArray);
        }
        readFileContents(fileName, byteArray);
```

```
        // calculate checksum and return it
    }
}
```

A poor man's cache

`CachingChecksum` used a soft reference to cache a single object and let the JVM handle the details of when to evict the object from the cache. Similarly, soft references are also often used in GUI applications for caching bitmap graphics. The key to whether a soft reference can be used is whether an application can recover from the loss of the data being cached.

If you need to cache more than a single object, you might use a `Map`, but you have a choice as to how to employ soft references. You could manage the cache as a `Map<K, SoftReference<V>>` or as a `SoftReference<Map<K,V>>`. The latter option is usually preferable, as it makes less work for the collector and allows the entire cache to be reclaimed with less effort when memory is in high demand. Weak references are sometimes mistakenly used instead of soft references for building caches, but this will result in poor caching performance. In practice, weak references will get cleared fairly quickly after the object becomes weakly reachable -- usually before the cached object is needed again -- as minor garbage collections run frequently.

For applications that rely heavily on caching for performance, soft references may be too blunt an instrument -- it certainly is not meant as a replacement for sophisticated caching frameworks that provide flexible expiration, replication, and transactional caching. But as a "cheap and dirty" caching mechanism, it is an attractive price-performer.

Just as with weak references, a soft reference can be created with an associated reference queue, and the reference is enqueued when it is cleared by the garbage collector. Reference queues are not as useful with soft references as with weak references, but they could be used to raise a management alert that the application is starting to run low on memory.

How the garbage collector handles References

Weak and soft references both extend the abstract `Reference` class (as do *phantom references*, which will be discussed in a future article). Reference objects are treated specially by the garbage collector. When the garbage collector encounters a `Reference` in the course of tracing the heap, it does not mark or trace the referent object, but instead places the `Reference` on a queue of known live `Reference` objects. After the trace, the collector identifies the softly reachable objects -- these objects for which no strong references exist but soft references do. The garbage collector then assesses whether soft references need to be cleared at this time, based on the amount of memory reclaimed by the current collection and other policy considerations. Soft references that are to be cleared are enqueued if they have a corresponding reference queue; the remaining softly reachable objects (the ones that are not cleared) are then treated as a root set and the heap trace is continued using these new roots so that the objects reachable through live soft references can be marked.

After processing the soft references, the set of weakly reachable objects is identified -- objects for which no strong or soft references exist. These are cleared and enqueued. All `Reference` types are cleared before they are enqueued, so the thread handling the post-mortem cleanup never has access to the referent object, only the `Reference` object. For this reason, when `References` are used in conjunction with a reference queue, it is common to subclass the appropriate reference type and either use it directly in your design (as with `WeakHashMap`, whose `Map.Entry` extends `WeakReference`) or to store references to entities that require cleanup.

Performance costs of reference processing

Reference objects introduce some additional costs into the garbage collection process. At each garbage collection, a list of live `Reference` objects must be constructed, and each reference must be processed appropriately, which adds some per-`Reference` overhead to each collection regardless of whether the referent is being collected at that time. `Reference` objects themselves are subject to garbage collection and can be collected before the referent, in which case they are not enqueued.

Array-based collections

Another form of object loitering arises when arrays are used to implement data structures such as stacks or circular buffers. The `LeakyStack` class in Listing 3 shows an implementation of a stack backed by an array. In the `pop()` method, after the top pointer is decremented, `elements` still maintains a reference to the object being popped off the stack. This means that a reference to that object is still reachable by the program even though the program never actually uses that reference again, which prevents that object from being garbage collected until that location is reused by a future `push()`.

Listing 3. Object loitering in array-based collection

```
public class LeakyStack {
    private Object[] elements = new Object[MAX_ELEMENTS];
    private int size = 0;

    public void push(Object o) { elements[size++] = o; }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        else {
            Object result = elements[--size];
            // elements[size+1] = null;
            return result;
        }
    }
}
```

The cure for object loitering in this case is to null out the reference after popping it from the stack, as shown in the commented-out line of code in Listing 3. However, this case -- where a class manages its own memory -- is one of the very few situations where explicitly nulling objects that are no longer needed is a good idea. Most of the time, aggressively nulling references believed to be unused results in no performance or memory utilization gain at all; it usually results in either worse performance or `NullPointerException`s. A linked implementation of this algorithm would not have this problem; in a linked implementation, the lifetime of the link node (and therefore the reference to the object being stored) would be automatically tied to the duration that the object is stored in the collection. Weak references could be used to fix this problem -- maintaining an array of weak references instead of strong references -- but in reality, `LeakyStack` manages its own memory and is therefore responsible for ensuring that references to objects no longer needed are cleared. Using arrays to implement a stack or a buffer is an optimization that reduces allocation but imposes a greater burden for the implementer to carefully manage the lifetime of references stored in the array.

Summary

Soft references, like weak references, can help applications prevent object loitering by enlisting the aid of the garbage collector in making cache eviction decisions. Soft references are suitable for use only if the application can tolerate the loss of the softly referenced object.

Resources

Learn

- "Plugging memory leaks with weak references": Last month's *Java theory and practice* introduced `SoftReference`'s cousin, `WeakReference`.
- "Tuning garbage collection in the HotSpot JVM": Kirk Pepperdine and Jack Shirazi demonstrate how even a slow memory leak eventually puts overwhelming pressure on the garbage collector.
- Reference objects and garbage collection: This paper from Sun, written shortly after Reference objects were added to the class library, describes how the garbage collector treats Reference objects.
- *Java theory and practice*: The complete series by Brian Goetz.
- The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- JTune: The free JTune tool can take GC logs and graphically display heap size, GC duration, and other useful memory management data.

Discuss

- Participate in the discussion forum.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Brian Goetz has been a professional software developer for over 18 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. Brian's book, *Java Concurrency In Practice*, will be published in late 2005 by Addison-Wesley. See Brian's published and upcoming articles in popular industry publications.