java performance

Using Java Reference Objects to Implement Caching

by Glen McCluskey
<glenm@glenmccl.com>

Glen McCluskey is a consultant with 15 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ perfor-mance, testing, and technical documenta-tion areas.

When we hear the term "cache," we often think of hardware, such as L1/L2 caches used with a CPU to implement very fast memory access. We also might think of operating-system caches, for example, a cache of recently used disk blocks. Or we might think of customized caching implemented in an application, such as keeping the most recently accessed record from a database around in hopes of avoiding a costly future lookup.

The Java standard libraries and runtime system offer another style of cache support, one that rests above the hardware and operating-system levels, but that requires support within the Java virtual machine (JVM). The underlying mechanism is a simple one. Suppose I have an arbitrary object of some type, and I say:

```
Object obj = ...
SoftReference ref = new SoftReference(obj);
```

`SoftReference` is a class whose instances are used to wrap other objects. In other words, a object of type `SoftReference` has within itself a reference to another object; this is known as the "referent." If I want to get my referred-to object back out, I can say:

```
obj = ref.get();
```

Reference objects are useful in implementing caches, because the garbage collector in the JVM knows about them and implements specific semantics for such objects. For example, the referent of a `SoftReference` may be cleared by the garbage collector if it needs additional memory and the referred-to object is not actually used anywhere in the application. In such a case the referent is said to be "softly reachable" rather than "strongly reachable."

Suppose that at some point in my program, I have:

```
ref = new SoftReference(obj);
```

with `obj` as some memory-resident object, and then at a later point I say:

```
obj = ref.get();
```

and `obj` is null, that is, the garbage collector has cleared the reference because it needs the memory. In this example, using a `SoftReference` object as a wrapper means that the referred-to object will be kept around if possible, but if memory is required elsewhere, then the object may be discarded if it's not being used in the program.

To see how `SoftReference` works in practice, here is a program that implements a file cache. If a requested file is in memory, a byte vector with its contents is returned; otherwise, the file is read from disk. A hash table is used that maps pathnames to `SoftReference` objects.

```
import java.util.HashMap;
import java.lang.ref.SoftReference;
import java.io.*;

public class FileCache {

  // hash table that maps pathnames -> SoftReference objects

  private HashMap map = new HashMap();

  // read a file into a byte vector and put it in the table

  private byte[] readin(String fn) throws IOException {

    // open the file

    FileInputStream fis = new FileInputStream(fn);

    // read all its bytes

    long filelen = new File(fn).length();
    byte[] vec = new byte[(int)filelen];
    fis.read(vec);
    fis.close();

    // put the (pathname, vector) entry into the hash table

    map.put(fn, new SoftReference(vec));

    return vec;

  }

  // get the byte vector for a file

  public byte[] getFile(String fn) throws IOException {

    // look up the pathname in the hash table

    SoftReference ref = (SoftReference)map.get(fn);
    byte[] vec;

    // if the name is not there, or the referent has been cleared,
    // then read from disk

    if (ref == null || (vec = (byte[])ref.get()) == null) {
      System.err.println("read " + fn + " from disk");
      return readin(fn);
    }
    else {
      System.err.println("read " + fn + " from cache");
      return vec;
```

```
    }
  }

  public static void main(String args[]) throws IOException {

    // set up the cache for files

    FileCache cache = new FileCache();
    byte[] vec;

    // read in big files the first time

    vec = cache.getFile("big1");
    vec = cache.getFile("big2");
    //vec = cache.getFile("big3");

    // read the second time from cache or from disk

    vec = cache.getFile("big1");
    vec = cache.getFile("big2");
  }
}
```

This program reads three large files called `big1`, `big2`, and `big3`. You create them using this program:

```
import java.io.*;

public class MakeBig {

  public static void main(String args[]) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: filename length(K)");
      System.exit(1);
    }

    // open output file

    FileOutputStream fos = new FileOutputStream(args[0]);
    int len = Integer.parseInt(args[1]);
    byte[] vec = new byte[1024];

    // write out 1K chunks to the file

    for (int i = 0; i < vec.length; i++)
      vec[i] = (byte)'*';
    for (int i = 0; i < len; i++)
      fos.write(vec);

    fos.close();
  }
}
```

You create the files by saying:

```
$ java MakeBig big1 5000
$ java MakeBig big2 5000
$ java MakeBig big3 5000
```

When the cache program is run on my machine, the results vary depending on whether the third `cache.getFile()` line in the first group is commented. If this line is commented, then the second set of `getFile()` calls will read from cache, but if the line is uncommented, then the second set reads from disk again.

In other words, changing the numbers and sizes of files that the program reads affects memory usage and garbage collection, and because of this, the referents of `SoftReference` objects are affected as well. Your results will vary, depending on what JVM you're using, how much memory you have available, sizes of files, and so on.

There are several other reference types similar to `SoftReference`. For example, the class `java.util.WeakHashMap` is based on `WeakReference` and is a hash table with the property that unused keys are discarded over time, via an interaction between the garbage collector and `WeakHashMap`.