Java theory and practice: Concurrent collections classes
ConcurrentHashMap and CopyOnWriteArrayList offer thread safety and improved scalability

Level: Introductory
Brian Goetz (brian@quiotix.com), Principal Consultant, Quiotix Corp
23 Jul 2003

In addition to many other useful concurrency building blocks, Doug Lea's util.concurrent package contains high-performance, thread-safe implementations for workhorse collection types List and Map. This month, Brian Goetz shows you how many concurrent programs will benefit from simply replacing Hashtable or synchronizedMap with ConcurrentHashMap.

The first associative collection class to appear in the Java class library was Hashtable, which was part of JDK 1.0. Hashtable provided an easy-to-use, thread-safe, associative map capability, and it was certainly convenient. However, the thread-safety came at a price -- all methods of Hashtable were synchronized. At that time, uncontended synchronization had a measurable performance cost. The successor to Hashtable, HashMap, which appeared as part of the Collections framework in JDK 1.2, addressed thread-safety by providing an unsynchronized base class and a synchronized wrapper, Collections.synchronizedMap. Separating the base functionality from the thread-safety Collections.synchronizedMap allowed users who needed synchronization to have it, but users who didn't need it didn't have to pay for it.

The simple approach to synchronization taken by both Hashtable and synchronizedMap -- synchronizing each method on the Hashtable or the synchronized Map wrapper object -- has two principal deficiencies. It is an impediment to scalability, because only one thread can access the hash table at a time. At the same time, it is insufficient to provide true thread safety, in that many common compound operations still require additional synchronization. While simple operations such as get() and put() can complete safely without additional synchronization, there are several common sequences of operations, such as iteration or put-if-absent, which still require external synchronization to avoid data races.

Conditional thread-safety

The synchronized collections wrappers, synchronizedMap and synchronizedList, are sometimes called conditionally thread-safe -- all individual operations are thread-safe, but sequences of operations where the control flow depends on the results of previous operations may be subject to data races. The first snippet in Listing 1 shows the common put-if-absent idiom -- if an entry does not already exist in the Map, add it. Unfortunately, as written, it is possible for another thread to insert a value with the same key between the time the containsKey() method returns and the time the put() method is called. If you want to ensure exactly-once insertion, you need to wrap the pair of statements with a synchronized block that synchronizes on the Map m.

The other examples in Listing 1 deal with iteration. In the first example, the results of List.size() could become invalid during the execution of the loop, because another thread could delete items from the list. If the timing was unlucky, and an item was deleted by another thread just after entering the last iteration of the loop, List.get() will return null, and doInBackground() will likely throw a NullPointerException. What can you do to avoid this? If another thread may be accessing a List while you are iterating through it, you must lock the entire List while you are iterating by wrapping it with a synchronized block, synchronizing on the List l. This addresses the data race, but has further costs for concurrency, since locking the entire List while iterating could block other threads from accessing the list for a long time.

The Collections framework introduced iterators for traversing a list or other collection, which optimizes the process of iterating through the elements in a collection. However, the iterators implemented in the java.util Collections classes are fail-fast, which means that if one thread changes a collection while another thread is traversing it through an Iterator, the next
Iterator.hasNext() or Iterator.next() call will throw ConcurrentModificationException. Just as with the previous example, if you want to prevent ConcurrentModificationException, you must lock the entire List while you are iterating by wrapping it with a synchronized block that synchronizes on the List. Alternatively, you can invoke List.toArray() and iterate on the array without synchronization, but this could be expensive if the list is large.

Listing 1. Common race conditions in synchronized maps

```java
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());

// put-if-absent idiom -- contains a race condition
// may require external synchronization
if (!map.containsKey(key))
    map.put(key, value);

// ad-hoc iteration -- contains race conditions
// may require external synchronization
for (int i=0; i<list.size(); i++) {
    doSomething(list.get(i));
}

// normal iteration -- can throw ConcurrentModificationException
// may require external synchronization
for (Iterator i=list.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
```

False sense of confidence

The conditional thread safety provided by synchronizedList and synchronizedMap present a hidden hazard -- developers assume that because these collections are synchronized, they are fully thread-safe, and they neglect to synchronize compound operations properly. The result is that while these programs appear to work under light load, under heavy load they may start throwing NullPointerException or ConcurrentModificationException.

Scalability problems

Scalability describes how an application's throughput behaves as its workload and available computing resources increase. A scalable program can handle a proportionally larger workload with more processors, memory, or I/O bandwidth. Locking a shared resource for exclusive access is a scalability bottleneck -- it prevents other threads from being able to access that resource, even if idle processors are available to schedule those threads. To achieve scalability, we must eliminate or reduce our dependence on exclusive resource locks.

The bigger problem with the synchronized Collections wrappers, and the earlier Hashtable and Vector classes, is that they synchronize on a single lock. This means that only one thread may access the collection at once, and if one thread is in the process of reading from a Map, all other threads that want to either read from it or write to it must wait. The most common Map operations, get() and put(), may involve more computation than is obvious -- when traversing a hash bucket to find a specific key, get() may have to call Object.equals() on a large number of candidates. If the hashCode() function used by the key class does not spread values evenly over the hash range or has a large number of hash collisions, certain bucket chains may be much longer than others, and traversing a long hash chain and calling equals() on some percentage of its elements could be slow. The problem with get() and put() being expensive under these conditions is not only that access will be slow, but that all other threads are locked out from accessing the Map while that hash chain is being traversed.

The fact that get() may take significant time to execute in some cases is made significantly worse by the conditional thread safety problem discussed above. The race conditions illustrated in Listing 1 often make it necessary to hold the single collection lock for much longer than it takes to execute a single operation. If you are going to hold the lock on the collection during an entire iteration, then other threads may be stalled waiting for the collection lock for a long time.

Example: A simple cache
One of the most common applications for Map in server applications is to implement a cache. Server applications may cache file contents, generated pages, results of database queries, DOM trees associated with parsed XML files, and many other types of data. The primary purpose of a cache is to reduce service time and increase throughput by reusing the results of a previous computation. A typical characteristic of cache workload is that retrievals are much more common than updates, so (ideally) a cache would offer very good get() performance. A cache that impedes application performance is worse than no cache at all.

If you use synchronizedMap to implement a cache, you are introducing a potential scalability bottleneck into your application. Only one thread can access the Map at once, and this includes all the threads that might be retrieving a value out of the Map as well as threads that want to install a new (key, value) pair into the map.

Reducing lock granularity

One approach to improving the concurrency of a HashMap while providing thread safety is to dispense with the single lock for the entire table, and use a lock for each hash bucket (or more commonly, a pool of locks where each lock protects several buckets). This means that multiple threads can access different portions of the Map simultaneously, without contending for the single collection-wide lock. This approach immediately improves the scalability of insertion, retrieval, and removal operations. Unfortunately, this concurrency has a cost -- it becomes harder to implement methods that operate on the collection as a whole, such as size() or isEmpty(), because this may require acquiring many locks at once or risk returning an inaccurate result. However, for situations such as implementing caches, this is a very sensible trade-off -- retrieval and insertion operations are frequent, and size() and isEmpty() operations are considerably less frequent.

ConcurrentHashMap

The ConcurrentHashMap class from util.concurrent (which will also appear in the java.util.concurrent package in JDK 1.5) is a thread-safe implementation of Map that offers far better concurrency than synchronizedMap. Multiple reads can almost always execute concurrently, simultaneous reads and writes can usually execute concurrently, and multiple simultaneous writes can often execute concurrently. (The related ConcurrentReaderHashMap class offers similar multiple-reader concurrency, but allows only a single active writer.) ConcurrentHashMap is designed to optimize retrieval operations; in fact, successful get() operations usually succeed with no locking at all. Achieving thread-safety without locking is tricky and requires a deep understanding of the details of the Java Memory Model. The ConcurrentHashMap implementation, along with the rest of util.concurrent, has been extensively peer-reviewed by concurrency experts for correctness and thread safety. We will look at the implementation details of ConcurrentHashMap in next month's article.

ConcurrentHashMap achieves higher concurrency by slightly relaxing the promises it makes to callers. A retrieval operation will return the value inserted by the most recent completed insert operation, and may also return a value added by an insertion operation that is concurrently in progress (but in no case will it return a nonsense result). Iterators returned by ConcurrentHashMap.iterator() will return each element once at most and will not ever throw ConcurrentHashMapException, but may or may not reflect insertions or removals that occurred since the iterator was constructed. No table-wide locking is needed (or even possible) to provide thread-safety when iterating the collection. ConcurrentHashMap may be used as a replacement for synchronizedMap or Hashtable in any application that does not rely on the ability to lock the entire table to prevent updates.

These compromises enable ConcurrentHashMap to provide far superior scalability over Hashtable, without compromising its effectiveness in a wide variety of common-use cases, such as shared caches.

How much better?

Table 1 gives a rough idea of the scalability differences between Hashtable and ConcurrentHashMap. In each run, n threads concurrently executed a tight loop where they retrieved random key values values from either a Hashtable or a ConcurrentHashMap, with 80 percent of the failed retrievals performing a put() operation and 1 percent of the successful retrievals performing a remove(). Tests were performed on a dual-processor Xeon system running Linux. The data shows run time in milliseconds for 10,000,000 iterations, normalized to the 1-thread case for ConcurrentHashMap. You can see that the performance of ConcurrentHashMap remains scalable up to many threads, whereas the performance of Hashtable degrades almost immediately in the presence of lock contention.
The number of threads in this test may look small compared to typical server applications. However, because each thread is doing nothing but repeatedly hitting on the table, this simulates the contention of a much larger number of threads using the table in the context of doing some amount of real work.

Table 1. Scalability of Hashtable versus ConcurrentHashMap

<table>
<thead>
<tr>
<th>Threads</th>
<th>ConcurrentHashMap</th>
<th>Hashtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.03</td>
</tr>
<tr>
<td>2</td>
<td>2.59</td>
<td>32.40</td>
</tr>
<tr>
<td>4</td>
<td>5.58</td>
<td>78.23</td>
</tr>
<tr>
<td>8</td>
<td>13.21</td>
<td>163.48</td>
</tr>
<tr>
<td>16</td>
<td>27.58</td>
<td>341.21</td>
</tr>
<tr>
<td>32</td>
<td>57.27</td>
<td>778.41</td>
</tr>
</tbody>
</table>

CopyOnWriteArrayList

The CopyOnWriteArrayList class is intended as a replacement for ArrayList in concurrent applications where traversals greatly outnumber insertions or removals. This is quite common when ArrayList is used to store a list of listeners, such as in AWT or Swing applications, or in JavaBean classes in general. (The related CopyOnWriteArraySet uses a CopyOnWriteArrayList to implement the Set interface.)

If you are using an ordinary ArrayList to store a list of listeners, as long as the list remains mutable and may be accessed by multiple threads, you must either lock the entire list during iteration or clone it before iteration, both of which have a significant cost. CopyOnWriteArrayList instead creates a fresh copy of the list whenever a mutative operation is performed, and its iterators are guaranteed to return the state of the list at the time the iterator was constructed and not throw a ConcurrentModificationException. It is not necessary to clone the list before iteration or lock it during iteration because the copy of the list that the iterator sees will not change. In other words, CopyOnWriteArrayList contains a mutable reference to an immutable array, so as long as that reference is held fixed, you get all the thread-safety benefits of immutability without the need for locking.

Summary

The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes, Collections.synchronizedMap and Collections.synchronizedList, provide a basic conditionally thread-safe implementation of Map and List. However, several factors make them unsuitable for use in highly concurrent applications -- their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationExceptions. The ConcurrentHashMap and CopyOnWriteArrayList implementations provide much higher concurrency while preserving thread safety, with some minor compromises in their promises to callers. ConcurrentHashMap and CopyOnWriteArrayList are not necessarily useful everywhere you might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.

Resources

- Participate in the discussion forum.
- Read the complete Java theory and practice series by Brian Goetz. In particular, the thread-safety benefits of immutability are discussed in "To mutate, or not to mutate?" (developerworks, February 2003).
Doug Lea's *Concurrent Programming in Java, Second Edition* is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.

Download the `util.concurrent` package.

The javadoc page for `ConcurrentHashMap` explains in greater details the differences between `ConcurrentHashMap` and `Hashtable`.

JSR 166 is standardizing the `util.concurrent` library for JDK 1.5.

November 2002's *Java theory and practice* column dealt with the executor facility of `util.concurrent`.


Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

**About the author**

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.