Java theory and practice: Building a better HashMap

How ConcurrentHashMap offers higher concurrency without compromising thread safety

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

21 August 2003

`ConcurrentHashMap`, part of Doug Lea's `util.concurrent` package, offers a higher degree of concurrency than `Hashtable` or `synchronizedMap`. In addition, it manages to avoid locking completely for most successful `get()` operations, which results in excellent throughput in concurrent applications. This month, Brian Goetz dives into the code of `ConcurrentHashMap` and looks at how this impressive feat is accomplished without compromising thread safety.

View more content in this series

In July's installment of *Java theory and practice* ("Concurrent collections classes"), we reviewed scalability bottlenecks and discussed how to achieve higher concurrency and throughput in shared data structures. Sometimes, the best way to learn is to examine the work of the experts, so this month we're going to look at the implementation of `ConcurrentHashMap` from Doug Lea's `util.concurrent` package. A version of `ConcurrentHashMap` optimized for the new Java Memory Model (JMM), which is being specified by JSR 133, will be included in the `java.util.concurrent` package in JDK 1.5; the version in `util.concurrent` has been audited for thread-safety under both the old and new memory models.

Optimizing for throughput

`ConcurrentHashMap` uses several tricks to achieve a high level of concurrency and avoid locking, including using multiple write locks for different hash buckets and exploiting the uncertainties of the JMM to minimize the time that locks are held -- or avoid acquiring locks at all. It is optimized for the most common usage, which is retrieving a value likely to already exist in the map. In fact, most successful `get()` operations will run without any locking at all. (Warning: don't try this at home! Trying to outsmart the JMM is much harder than it looks and is not to be undertaken lightly. The `util.concurrent` classes were written by concurrency experts and have been extensively peer-reviewed for JMM safety.)

Multiple write locks

Recall that the major scalability impediment of `Hashtable` (or alternatively, `Collections.synchronizedMap`) is that it uses a single map-wide lock, which must be held for the entirety of an insertion, removal, or retrieval operation, and sometimes even for the entirety of an iterator traversal. This prevents other threads from accessing the `Map` at all while the lock is held, even if idle processors are available, significantly limiting concurrency.

`ConcurrentHashMap` dispenses with the single map-wide lock, instead using a collection of 32 locks, each of which guards a subset of the hash buckets. Locks are primarily used by mutative (`put()` and `remove()`) operations. Having 32 separate locks means that a maximum of 32 threads can be modifying the map at once. This doesn't necessarily mean that if there are fewer than 32 threads writing to the map currently, that another write operation will not block -- 32 is the theoretical concurrency limit for writers, but may not always be achieved in practice. Still, 32 is a lot better than one and should be more than adequate for most applications running on the current generation of computer systems.

Map-wide operations

Having 32 separate locks, each guarding a subset of the hash buckets, means that operations that require exclusive access to the map must acquire all 32 locks. Some map-wide operations, such as `size()` and `isEmpty()` may be able to get away without locking the entire map at once (by suitably qualifying the semantics of these operations), but some operations, such as map rehashing (expanding the number of hash buckets and redistributing elements as the map grows) must guarantee exclusive access. The Java language does not provide a simple way to acquire a variable-sized set of locks; in the infrequent cases where this must be done, it is done by recursion.

A JMM overview

Before we jump into the implementation of `put()`, `get()`, and `remove()`, let's briefly review the JMM, which governs how actions on memory (reads and writes) by one thread affect actions on memory by other threads. Because of the performance benefits of using processor registers and per-processor caches to speed up memory access, the Java Language Specification (JLS) permits some memory operations not to be immediately visible to all other threads. There are two language mechanisms for guaranteeing consistency of memory operations across threads -- `synchronized` and `volatile`.

According to the JLS, "In the absence of explicit synchronization, an implementation is free to update the main memory in an order that may be surprising." This means that without synchronization, writes that occur in one order in a given thread may appear to occur in a different order to a different thread, and that updates to memory variables may take an unspecified time to propagate from one thread to another.

While the most common reason for using synchronization is to guarantee atomic access to critical sections of code, synchronization actually provides three separate functions -- atomicity, visibility, and ordering. Atomicity is straightforward enough -- synchronization enforces a reentrant mutex,

preventing more than one thread from executing a block of code protected by a given monitor at the same time. Unfortunately, most texts focus on the atomicity aspects of synchronization to the exclusion of the other aspects. But synchronization also plays a significant role in the JMM, causing the JVM to execute memory barriers when acquiring and releasing monitors.

When a thread acquires a monitor, it executes a *read barrier* -- invalidating any variables cached in thread-local memory (such as an on-processor cache or processor registers), which will cause the processor to re-read any variables used in the synchronized block from main memory. Similarly, upon monitor release, the thread executes a *write barrier* -- flushing any variables that have been modified back to main memory. The combination of mutual exclusion and memory barriers means that as long as programs follow the correct synchronization rules (that is, synchronize whenever writing a variable that may next be read by another thread or when reading a variable that may have been last written by another thread), each thread will see the correct value of any shared variables it uses.

Some very strange things can happen if you fail to synchronize when accessing shared variables. Some changes may be reflected across threads instantly, while others may take some time (due to the nature of associative caches). As a result, without synchronization you cannot be sure that you have a consistent view of memory (related variables may not be consistent with each other) or a current view of memory (some values may be stale). The common -- and recommended -- way to avoid these hazards is of course to synchronize properly. However, in some cases, such as in very widely used library classes like `ConcurrentHashMap`, it may be worth applying some extra expertise and effort in development (which may well be many times as much effort) to achieve higher performance.

ConcurrentHashMap implementation

As suggested earlier, the data structure used by `ConcurrentHashMap` is similar in implementation to that of `Hashtable` or `HashMap`, with a resizable array of hash buckets, each consisting of a chain of `Map.Entry` elements, as shown in Listing 1. Instead of a single collection lock, `ConcurrentHashMap` uses a fixed pool of locks that form a partition over the collection of buckets.

Listing 1. Map.Entry elements used by ConcurrentHashMap

```
protected static class Entry implements Map.Entry {
    protected final Object key;
    protected volatile Object value;
    protected final int hash;
    protected final Entry next;
    ...
}
```

Traversing data structures without locking

Unlike `Hashtable` or a typical lock-pool `Map` implementation, `ConcurrentHashMap.get()` operations do not necessarily entail acquiring the lock associated with the relevant bucket. In the absence of locking, the implementation must be prepared to deal with stale or inconsistent values of any variables it uses, such as the list head pointer and the fields of the `Map.Entry` elements (including the link pointers that comprise the linked list of entries for each hash bucket).

Most concurrent classes use synchronization to ensure exclusive access to -- and a consistent view of -- a data structure. Instead of assuming exclusivity and consistency, the linked list used by `ConcurrentHashMap` is designed carefully so that the implementation can *detect* that its view of the list is inconsistent or stale. If it detects that its view is inconsistent or stale, or simply does not find the entry it is looking for, it then synchronizes on the appropriate bucket lock and searches the chain again. This optimizes lookup in the common case -- where most retrievals are successful and retrievals outnumber insertions and removals.

Exploiting immutability

One significant source of inconsistency is avoided by making the `Entry` elements nearly immutable -- all fields are final, except for the value field, which is volatile. This means that elements cannot be added to or removed from the middle or end of the hash chain -- elements can only be added at the beginning, and removal involves cloning all or part of the chain and updating the list head pointer. So once you have a reference into a hash chain, while you may not know whether you have a reference to the head of the list, you do know that the rest of the list will not change its structure. Also, since the value field is volatile, you will be able to see updates to the value field immediately, greatly simplifying the process of writing a `Map` implementation that can deal with a potentially stale view of memory.

While the new JMM provides initialization safety for final variables, the old JMM does not, which means that it is possible for another thread to see the default value for a final field, rather than the value placed there by the object's constructor. The implementation must be prepared to detect this as well, which it does by ensuring that the default value for each field of `Entry` is not a valid value. The list is constructed such that if any of the `Entry` fields appear to have their default value (zero or `null`), the search will fail, prompting the `get()` implementation to synchronize and traverse the chain again.

Retrieval operations

Retrieval operations proceed by first finding the head pointer for the desired bucket (which is done without locking, so it could be stale), and traversing the bucket chain without acquiring the lock for that bucket. If it doesn't find the value it is looking for, it synchronizes and tries to find the entry again, as shown in Listing 2:

Listing 2. ConcurrentHashMap.get() implementation

```
public Object get(Object key) {
  int hash = hash(key);     // throws null pointer exception if key is null

  // Try first without locking...
  Entry[] tab = table;
  int index = hash & (tab.length - 1);
  Entry first = tab[index];
  Entry e;

  for (e = first; e != null; e = e.next) {
    if (e.hash == hash && eq(key, e.key)) {
      Object value = e.value;
      // null values means that the element has been removed
      if (value != null)
```

```
        return value;
      else
        break;
    }
  }

  // Recheck under synch if key apparently not there or interference
  Segment seg = segments[hash & SEGMENT_MASK];
  synchronized(seg) {
    tab = table;
    index = hash & (tab.length - 1);
    Entry newFirst = tab[index];
    if (e != null || first != newFirst) {
      for (e = newFirst; e != null; e = e.next) {
        if (e.hash == hash && eq(key, e.key))
          return e.value;
      }
    }
    return null;
  }
}
```

Removal operations

Because a thread could see stale values for the link pointers in a hash chain, simply removing an element from the chain would not be sufficient to ensure that other threads will not continue to see the removed value when performing a lookup. Instead, as Listing 3 illustrates, removal is a two-step process -- first the appropriate `Entry` object is found and its value field is set to `null`, and then the portion of the chain from the head to the removed element is cloned and joined to the remainder of the chain following the removed element. Since the value field is volatile, if another thread is traversing the stale chain looking for the removed element, it will see the null value field immediately, and know to retry the retrieval with synchronization. Eventually, the initial part of the hash chain ending in the removed element will be garbage collected.

Listing 3. ConcurrentHashMap.remove() implementation

```
protected Object remove(Object key, Object value) {
  /*
    Find the entry, then
      1. Set value field to null, to force get() to retry
      2. Rebuild the list without this entry.
         All entries following removed node can stay in list, but
         all preceding ones need to be cloned.  Traversals rely
         on this strategy to ensure that elements will not be
        repeated during iteration.
  */

  int hash = hash(key);
  Segment seg = segments[hash & SEGMENT_MASK];

  synchronized(seg) {
    Entry[] tab = table;
    int index = hash & (tab.length-1);
    Entry first = tab[index];
    Entry e = first;

    for (;;) {
      if (e == null)
        return null;
      if (e.hash == hash && eq(key, e.key))
        break;
```

```
        e = e.next;
      }

      Object oldValue = e.value;
      if (value != null && !value.equals(oldValue))
        return null;

      e.value = null;

      Entry head = e.next;
      for (Entry p = first; p != e; p = p.next)
        head = new Entry(p.hash, p.key, p.value, head);
      tab[index] = head;
      seg.count--;
      return oldValue;
    }
  }
```

Figure 1 illustrates a hash chain before an element is removed:

Figure 1. Hash chain

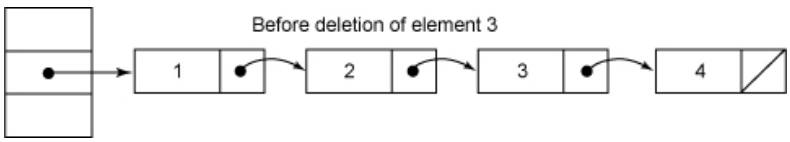
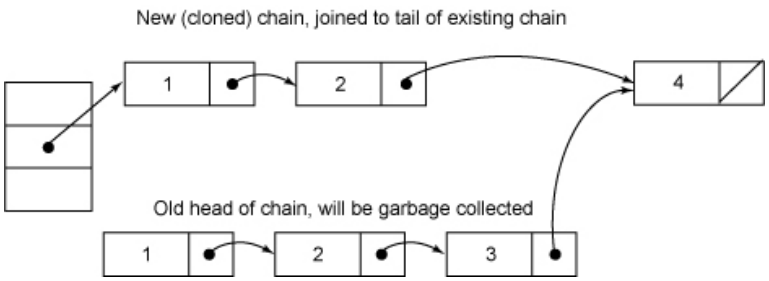


Figure 2 illustrates the chain with element 3 removed:

Figure 2. Removal of an element



Insertion and update operations

The implementation of `put()` is straightforward. Like `remove()`, `put()` holds the bucket lock for the duration of its execution, but because `get()` doesn't always need to acquire the lock, this doesn't necessarily block readers from executing (nor does it block other writers from accessing other buckets). First, `put()` searches the appropriate hash chain for the desired key. If it is found, then the `value` field (which is volatile) is simply updated. If it is not found, a new `Entry` object is created to describe the new mapping and is inserted at the head of the list for that bucket.

Weakly consistent iterators

The semantics of iterators returned by `ConcurrentHashMap` differ from those of `java.util` collections; rather than being *fail-fast* (throwing an exception if the underlying collection is modified while an iterator is being used), they are instead *weakly consistent*. When a user

calls `keySet().iterator()` to retrieve an iterator for the set of hash keys, the implementation briefly synchronizes to make sure the head pointers for each chain are current. The `next()` and `hasNext()` operations are defined in the obvious way, traversing each hash chain and then moving on to the next chain until all the chains have been traversed. Weakly consistent iterators may or may not reflect insertions made during iteration, but they will definitely reflect updates or removals for keys that have not yet been reached by the iterator, and will not return any value more than once. The iterators returned by `ConcurrentHashMap` will not throw `ConcurrentModificationException`.

Dynamic resizing

As the number of elements in the map grows, the hash chains will grow longer and retrieval time will increase. At some point, it makes sense to increase the number of buckets and rehash the values. In classes like `Hashtable`, this is easy because it is possible to hold an exclusive lock on the entire map. In `ConcurrentHashMap`, each time an entry is inserted, if the length of that chain exceeds some threshold, that chain is marked as needing to be resized. When enough chains vote that resizing is needed, `ConcurrentHashMap` will use recursion to acquire the lock on each bucket and rehash the elements from each bucket into a new, larger hash table. Most of the time, this will occur automatically and transparently to the caller.

No locking?

To say that successful `get()` operations proceed without locking is a bit of an overstatement, as the `value` field of `Entry` is volatile, and this is used to detect updates and removals. At the machine level, volatile and synchronization often end up getting translated into the same cache coherency primitives, so there effectively is *some* locking going on here, albeit at a finer granularity and without the scheduling or JVM overhead of acquiring and releasing monitors. But, semantics aside, the concurrency achieved by `ConcurrentHashMap` in many common situations, where retrievals outnumber insertions and removals, is quite impressive.

Conclusion

`ConcurrentHashMap` is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the JMM to achieve higher performance. `ConcurrentHashMap` is an impressive feat of coding, one that requires a deep understanding of concurrency and the JMM. Use it, learn from it, enjoy it -- but unless you're an expert on Java concurrency, you probably shouldn't try this on your own.

Resources

- Read the complete *Java theory and practice* series by Brian Goetz. Of particular interest are:
  - "Concurrency made simple (sort of)" (November 2002), which introduced the `util.concurrent` package
  - "To mutate, or not to mutate?" (February 2003), which discussed thread-safety benefits of immutability
  - "Concurrent collections classes" (July 2003), which examined scalability bottlenecks and how to achieve higher concurrency and throughput in shared data structures
- Doug Lea's *Concurrent Programming in Java, Second Edition* is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.
- This excerpt from Doug Lea's book describes what synchronization *really* means.
- Download the `util.concurrent` package.
- The javadoc page for `ConcurrentHashMap` explains in greater detail the differences between `ConcurrentHashMap` and `Hashtable`.
- Take a look at the source code for ConcurrentHashMap.
- JSR 166 is standardizing the `util.concurrent` library for JDK 1.5.
- Bill Pugh maintains a collection of resources on the Java Memory Model.
- JSR 133 addresses a version of `ConcurrentHashMap` optimized for the new Java Memory Model.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

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.