Custom Search

Java *copy-on-write* collections

Along with `ConcurrentHashMap` (synchronization_concurrency_8_hashmap.shtml), Java 5 introduces a copy-on-write implementation of a list: `CopyOnWriteArrayList`. It also introduces `CopyOnWriteArraySet`, essentially a convenience wrapper around the list.

CopyOnWriteArrayList

An instance of `CopyOnWriteArrayList` behaves as a `List` implementation that allows multiple concurrent reads, and for reads to occur concurrently with a write. The way it does this is to make a brand new copy of the list every time it is altered.

- Reads do not block, and effectively pay only the cost of a `volatile` read;
- Writes do not block reads (or vice versa), *but* **only one write can occur at once**;
- Unlike `ConcurrentHashMap`, write operations that write or access multiple elements in the list (such as `addAll()`, `retainAll()`) will be atomic.

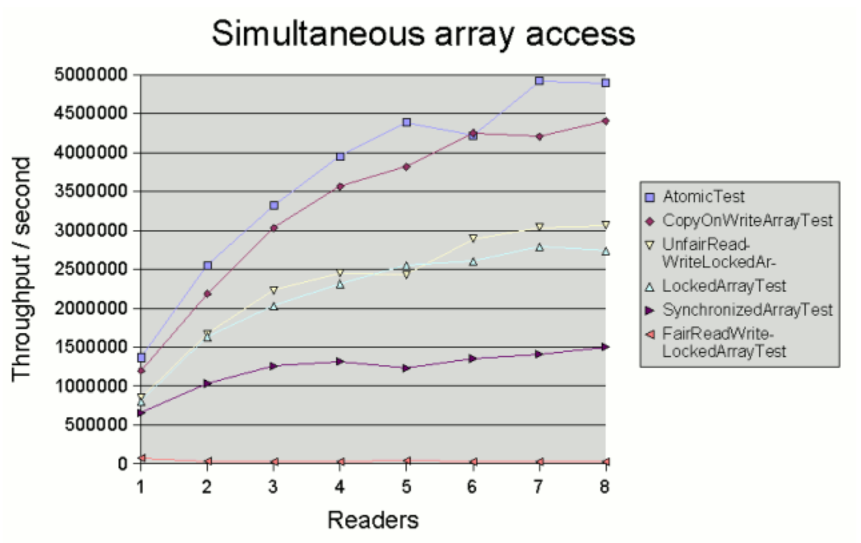
During a write operation, the array must be locked completely against other writes. (The standard implementation uses a `ReentrantLock` (synchronization_concurrency_9_locks_j5b.shtml#reentrantlock).) However, this does mean that as mentioned, operations affecting multiple locations can be atomic. That is, if one thread adds several items to the list with `addAll()` while another thread calls `size()`, the thread reading the size will get a value that either reflects or not the number of elements added in `addAll()`: there'll be no possibility of an intermediate value returned (provided of course that these are the only two threads accessing the list!).

`CopyOnWriteArrayList` is designed for cases where:

- reads hugely outnumber writes;
- the array is small (or writes are very infrequent);
- the caller genuinely needs the functionality of a *list* rather than an *array*.

The first two are fairly obvious: if too great, then either the size of the array or the frequency of writes can outweigh the benefit of not synchrozing.

The last point is possibly more subtle, but what it boils down to is this: if you only need the functionality of an *array* (that is, you don't need the list to grow arbitrarily, you don't need inserts etc), then the **atomic array classes (synchronization_concurrency_7_atomic_arrays.shtml) generally give better throughput**. To illustrate this, the following graph shows overall throughput as we increase the number of reader threads concurrently reading from random positions in a shared 20-element array of integers. (There is always one writer thread, which continually writes to random positions.) Each coloured line represents the test run on an array synchronized in a different way. The "AtomicTest" array was an `AtomicIntegerArray`. The graph shows that throughput is generally a little better with the atomic array (after all, it is being used for what it was designed for).



On the other hand, what this graph also shows is that `CopyOnWriteArrayList` still gives pretty good read throughput, and beats hands down any other method that locks the array on every read.

CopyOnWriteArraySet

Another class, `CopyOnWriteArraySet` is build on top of `CopyOnWriteArrayList`. Like its list counterpart, it is designed for cases where the set contains only a few elements and where reads greatly outnumber writes. Note that the performance characteristics differ from `HashSet` in a couple of ways:

- adding or looking for an element in a `CopyOnWriteArraySet` entails scanning sequentially through the array of elements to check if an equal element is present;
- with a `HashSet`, there's generally little performance penalty for attempting to add an item already there; `CopyOnWriteArraySet` is optimised for the case where the element `isn't` already present.

The second point comes from the fact that `CopyOnWriteArraySet` relies on the **putIfAbsent()** method provided by `CopyOnWriteArrayList`. This, as an atomic operation, checks whether an equal element is already in the list and adds it only if it is not. This method, and thus the set implementation, has the subtle characteristic that a copy of the array is made (and then thrown away) *even if the item isn't added*. Of course, that doesn't make the already-present case any more expensive than the not-present case, but it means that the already-present case isn't generally much cheaper either. (Both cases also perform similarly with `HashSet`, but with the latter class, both are generally much less expensive since only a tiny subset of the set is scanned when inserting or retrieving.)