3 Tips for volatile fields in java

December 12, 2017

Category: Programming concurrent Java
(/categories/multithreaded_programming)

3 Tips for volatile fields in java

Volatile fields are one of built-in mechanism to write multi-threaded java.

> Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread. ... The visibility effects of volatile variables extend beyond the value of the volatile variable itself. When thread A writes to a volatile variable and subsequently thread B reads that same variable, the values of all variables that were visible to A prior to writing to the variable become visible to B after reading the volatile variable.
> — Java Concurrency in Practice - Brian Goetz, et al. (http://jcip.net/)

In the following I collected three tips on when and how to use volatile fields in practice:

1) Use volatile fields when writes do not depend on its current value.

An example is a flag to stop a worker thread from another thread:

```
1  public class WorkerThread extends Thread {
2      private volatile boolean isRunning = true;
3      @Override
4      public void run() {
5          while(isRunning)
6          {
7              // execute a task
8          }
9      }
10     public void stopWorker()
11     {
12         isRunning = false;
13     }
14 }
```

The WorkerThread executes his tasks in a while loop, line 5. It checks the volatile field isRunning in each iteration and stops processing if the field is false. This allows other threads to stop the WorkerThread by calling the method stopWorker which sets the value of the field to false. Since a thread can call the method stopWorker even if the WorkerThread is already stopped, the write to the field can be executed independently of its current value.

By declaring the field volatile we make sure that the WorkerThread sees the update done in another Thread and does not run forever.

2) Use volatile fields for reading and locks for writing

The java.util.concurrent.CopyOnWriteArrayList get and set methods are an example of this tip:

```
1  public class CopyOnWriteArrayList<E>
2      implements List<E>, RandomAccess, Cloneable, java.i
3    private transient volatile Object[] array;
4    final Object[] getArray() {
5          return array;
6     }
7     final void setArray(Object[] a) {
8          array = a;
9     }
10    private E get(Object[] a, int index) {
11         return (E) a[index];
12    }
13    public E get(int index) {
14         return get(getArray(), index);
15    }
16    public E set(int index, E element) {
17         final ReentrantLock lock = this.lock;
18         lock.lock();
19         try {
20             Object[] elements = getArray();
21             E oldValue = get(elements, index);
22             if (oldValue != element) {
23                 int len = elements.length;
24                 Object[] newElements = Arrays.copyOf(ele
25                 newElements[index] = element;
26                 setArray(newElements);
27             } else {
28                 // Not quite a no-op; ensures volatile w
29                 setArray(elements);
30             }
31             return oldValue;
32         } finally {
33             lock.unlock();
34         }
35     }
36     // Other fields and methods omitted
37 }
```

The get method, line 13, simply reads the volatile field array and returns the value at the position index. Writing uses a lock to ensure that only one thread can modify the array at a given time. Line 18 acquires the lock and line 33 releases the lock. Writing requires copying the array when an element is changed, line 24 so that the reading threads do not see an inconsistent state. The writing thread then updates the array, line 25 and set the new array to the volatile field array, line 26.

Using this tip only writes block writes. Compare this to using synchronized set and get methods where each operation block all other operations. Or java.util.concurrent.locks.ReentrantReadWriteLock where too many readers can lead to starvation of writers. (https://www.javaspecialists.eu/archive/Issue165.html)

This is especially a problem for older JDKs. Here are the number from Akhil Mittal in a DZone (https://dzone.com/articles/3-tips-for-volatile-fields-in-java) comment to this article:

```
Java 6
R0= 4, RW= 4, fair=false 4,699,560   584,991
Java 9
R0= 4, RW= 4, fair=false 2,133,904   3,289,220
```

3) Use with JDK 9 VarHandle for atomic operations.

All modern CPU provide instructions to atomically compare and set or increment and get values. Those operations are used internally by the JVM to implement synchronized locks. Prior to JDK 1.9, they were available for Java applications only through classes in the java.util.concurrent.atomic package or by using the private java API sun.misc.Unsafe. With the new JDK 9 VarHandle, it is now possible to execute such operations directly on volatile fields. The following shows the AtomicBoolean compareAndSet method implemented using VarHandles:

```
1  public class AtomicBoolean implements java.io.Serializab
2    private static final VarHandle VALUE;
3    static {
4          try {
5              MethodHandles.Lookup l = MethodHandles.looku
6              VALUE = l.findVarHandle(AtomicBoolean.class,
7          } catch (ReflectiveOperationException e) {
8              throw new Error(e);
9          }
10     }
11     private volatile int value;
12     public final boolean compareAndSet(boolean expectedV
13         return VALUE.compareAndSet(this,
14                                    (expectedValue ? 1 :
15                                    (newValue ? 1 : 0));
16     }
17     // Other fields and methods omitted
18 }
```

The VarHandle works similar to the class java.lang.reflect.Field. You need to lookup a VarHandle from the class which contains the field using the name of the field, line 6. To execute a compareAndSet operation on the field we need to call the VarHandle with the object of the field, the expected and the new value, line 13.

Conclusion

You can use volatile fields if the writes do not depend on the current value as in tip one. Or if you can make sure that only one thread at a time can update the field as in tip two and three. I think that those three tips cover the more common ways to use volatile fields. Read here (http://vmlens.com/articles/lock_free_executor_service/) about a more exotic way to use volatile fields to implement a concurrent queue.

I would be glad to hear from you about other ways to use volatile fields to achieve thread-safety.