Java theory and practice: Managing volatility

Guidelines for using volatile variables

Brian Goetz June 19, 2007

The Java™ language contains two intrinsic synchronization mechanisms: synchronized blocks (and methods) and volatile variables. Both are provided for the purpose of rendering code thread-safe. Volatile variables are the weaker (but sometimes simpler or less expensive) of the two -- but also easier to use incorrectly. In this installment of *Java theory and practice*, Brian Goetz explores some patterns for using volatile variables correctly and offers some warnings about the limits of its applicability.

View more content in this series

Volatile variables in the Java language can be thought of as "`synchronized` lite"; they require less coding to use than `synchronized` blocks and often have less runtime overhead, but they can only be used to do a subset of the things that `synchronized` can. This article presents some patterns for using volatile variables effectively -- and some warnings about when not to use them.

Locks offer two primary features: *mutual exclusion* and *visibility*. Mutual exclusion means that only one thread at a time may hold a given lock, and this property can be used to implement protocols for coordinating access to shared data such that only one thread at a time will be using the shared data. Visibility is more subtle and has to do with ensuring that changes made to shared data prior to releasing a lock are made visible to another thread that subsequently acquires that lock -- without the visibility guarantees provided by synchronization, threads could see stale or inconsistent values for shared variables, which could cause a host of serious problems.

Volatile variables

Volatile variables share the visibility features of `synchronized`, but none of the atomicity features. This means that threads will automatically see the most up-to-date value for volatile variables. They can be used to provide thread safety, but only in a very restricted set of cases: those that do not impose constraints between multiple variables or between a variable's current value and its future values. So volatile alone is not strong enough to implement a counter, a mutex, or any class that has invariants that relate multiple variables (such as "start <=end").

You might prefer to use volatile variables instead of locks for one of two principal reasons: simplicity or scalability. Some idioms are easier to code and read when they use volatile variables

instead of locks. In addition, volatile variables (unlike locks) cannot cause a thread to block, so they are less likely to cause scalability problems. In situations where reads greatly outnumber writes, volatile variables may also provide a performance advantage over locking.

Conditions for correct use of volatile

You can use volatile variables instead of locks only under a restricted set of circumstances. Both of the following criteria must be met for volatile variables to provide the desired thread-safety:

- Writes to the variable do not depend on its current value.
- The variable does not participate in invariants with other variables.

Basically, these conditions state that the set of valid values that can be written to a volatile variable is independent of any other program state, including the variable's current state.

The first condition disqualifies volatile variables from being used as thread-safe counters. While the increment operation (`x++`) may look like a single operation, it is really a compound read-modify-write sequence of operations that must execute atomically -- and volatile does not provide the necessary atomicity. Correct operation would require that the value of `x` stay unchanged for the duration of the operation, which cannot be achieved using volatile variables. (However, if you can arrange that the value is only ever written from a single thread, then you can ignore the first condition.)

Most programming situations will fall afoul of either the first or second condition, making volatile variables a less commonly applicable approach to achieving thread-safety than `synchronized`. Listing 1 shows a non-thread-safe number range class. It contains an invariant -- that the lower bound is always less than or equal to the upper bound.

Listing 1. Non-thread-safe number range class

```
@NotThreadSafe
public class NumberRange {
    private int lower, upper;

    public int getLower() { return lower; }
    public int getUpper() { return upper; }

    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }

    public void setUpper(int value) {
        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
    }
}
```

Because the state variables of the range are constrained in this manner, making the `lower` and upper fields volatile would not be sufficient to make the class thread-safe; synchronization would still be needed. Otherwise, with some unlucky timing, two threads executing `setLower` and `setUpper` with inconsistent values could leave the range in an inconsistent state. For example,

if the initial state is `(0, 5)`, and thread A calls `setLower(4)` at the same time that thread B calls `setUpper(3)`, and the operations are interleaved just wrong, both could pass the checks that are supposed to protect the invariant and end up with the range holding `(4, 3)` -- an invalid value. We need to make the `setLower()` and `setUpper()` operations atomic with respect to other operations on the range -- and making the fields volatile can't do this for us.

Performance considerations

The primary motivation for using volatile variables is simplicity: In some situations, using a volatile variable is just simpler than using the corresponding locking. A secondary motivation for using volatile variables is performance: In some situations, volatile variables may be a better-performing synchronization mechanism than locking.

It is exceedingly difficult to make accurate, general statements of the form "X is always faster than Y," especially when it comes to intrinsic JVM operations. (For example, the VM may be able to remove locking entirely in some situations, which makes it hard to talk about the relative cost of `volatile` vs. `synchronized` in the abstract.) That said, on most current processor architectures, volatile reads are cheap -- nearly as cheap as nonvolatile reads. Volatile writes are considerably more expensive than nonvolatile writes because of the memory fencing required to guarantee visibility but still generally cheaper than lock acquisition.

Unlike locking, volatile operations will never block, so volatiles offer some scalability advantages over locking in the cases where they can be used safely. In cases where reads greatly outnumber writes, volatile variables can often reduce the performance cost of synchronization compared to locking.

Patterns for using volatile correctly

Many concurrency experts tend to guide users away from using volatile variables at all, because they are harder to use correctly than locks. However, some well-defined patterns exist, which, if you follow them carefully, can be used safely in a wide variety of situations. Always keep in mind the rules about the limits of where volatile can be used -- only use volatile for state that is truly independent of everything else in your program -- and this should keep you from trying to extend these patterns into dangerous territory.

Pattern #1: status flags

Perhaps the canonical use of volatile variables is simple boolean status flags, indicating that an important one-time life-cycle event has happened, such as initialization has completed or shutdown has been requested.

Many applications include a control construct of the form, "While we're not ready to shut down, do more work," as shown in Listing 2:

Listing 2. Using a volatile variable as a status flag

```
volatile boolean shutdownRequested;

...

public void shutdown() { shutdownRequested = true; }

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

It is likely that the `shutdown()` method is going to be called from somewhere outside the loop -- in another thread -- and as such, some form of synchronization is required to ensure the proper visibility of the `shutdownRequested` variable. (It might be called from a JMX listener, an action listener in the GUI event thread, through RMI, through a Web service, and so on.) However, coding the loop with `synchronized` blocks would be much more cumbersome than coding it with a volatile status flag as in Listing 2. Because volatile simplifies the coding, and the status flag does not depend on any other state in the program, this is a good use for volatile.

One common characteristic of status flags of this type is that there is typically only one state transition; the `shutdownRequested` flag goes from `false` to `true` and then the program shuts down. This pattern can be extended to state flags that can change back and forth, but only if it is acceptable for a transition cycle (from `false` to `true` to `false`) to go undetected. Otherwise, some sort of atomic state transition mechanism is needed, such as atomic variables.

Pattern #2: one-time safe publication

The visibility failures that are possible in the absence of synchronization can get even trickier to reason about when writing to object references instead of primitive values. In the absence of synchronization, it is possible to see an up-to-date value for an object reference that was written by another thread and still see stale values for that object's state. (This hazard is the root of the problem with the infamous double-checked-locking idiom, where an object reference is read without synchronization, and the risk is that you could see an up-to-date reference but still observe a partially constructed object through that reference.)

One technique for safely publishing an object is to make the object reference volatile. Listing 3 shows an example where during startup, a background thread loads some data from a database. Other code, when it might be able to make use of this data, checks to see if it has been published before trying to use it.

Listing 3. Using a volatile variable for safe one-time publication

```
public class BackgroundFloobleLoader {
    public volatile Flooble theFlooble;

    public void initInBackground() {
        // do lots of stuff
        theFlooble = new Flooble();  // this is the only write to theFlooble
    }
}

public class SomeOtherClass {
    public void doWork() {
        while (true) {
            // do some stuff...
            // use the Flooble, but only if it is ready
            if (floobleLoader.theFlooble != null)
                doSomething(floobleLoader.theFlooble);
        }
    }
}
```

Without the `theFlooble` reference being volatile, the code in `doWork()` would be at risk for seeing a partially constructed `Flooble` as it dereferences the `theFlooble` reference.

A key requirement for this pattern is that the object being published must either be thread-safe or effectively immutable (effectively immutable means that its state is never modified after its publication). The volatile reference may guarantee the visibility of the object in its as-published form, but if the state of the object is going to change after publication, then additional synchronization is required.

Pattern #3: independent observations

Another simple pattern for safely using volatile is when observations are periodically "published" for consumption within the program. For example, say there is an environmental sensor that senses the current temperature. A background thread might read this sensor every few seconds and update a volatile variable containing the current temperature. Then, other threads can read this variable knowing that they will always see the most up-to-date value.

Another application for this pattern is gathering statistics about the program. Listing 4 shows how an authentication mechanism might remember the name of the last user to have logged on. The `lastUser` reference will be repeatedly used to publish a value for consumption by the rest of the program.

Listing 4. Using a volatile variable for multiple publications of independent observations

```
public class UserManager {
    public volatile String lastUser;

    public boolean authenticate(String user, String password) {
        boolean valid = passwordIsValid(user, password);
        if (valid) {
            User u = new User();
            activeUsers.add(u);
            lastUser = user;
        }
        return valid;
    }
}
```

This pattern is an extension of the previous one; a value is being published for use elsewhere within the program, but instead of publication being a one-time event, it is a series of independent events. This pattern requires that the value being published be effectively immutable -- that its state not change after publication. Code consuming the value should be aware that it might change at any time.

Pattern #4: the "volatile bean" pattern

The volatile bean pattern is applicable in frameworks that use JavaBeans as "glorified structs." In the volatile bean pattern, a JavaBean is used as a container for a group of independent properties with getters and/or setters. The rationale for the volatile bean pattern is that many frameworks provide containers for mutable data holders (for instance, `HttpSession`), but the objects placed in those containers must be thread safe.

In the volatile bean pattern, all the data members of the JavaBean are volatile, and the getters and setters must be trivial -- they must contain no logic other than getting or setting the appropriate property. Further, for data members that are object references, the referred-to objects must be effectively immutable. (This prohibits having array-valued properties, as when an array reference is declared `volatile`, only the reference, not the elements themselves, have volatile semantics.) As with any volatile variable, there may be no invariants or constraints involving the properties of the JavaBean. An example of a JavaBean obeying the volatile bean pattern is shown in Listing 5:

Listing 5. A Person object obeying the volatile bean pattern

```
@ThreadSafe
public class Person {
    private volatile String firstName;
    private volatile String lastName;
    private volatile int age;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Advanced patterns for volatile

The patterns in the previous section cover most of the basic cases where the use of volatile is sensible and straightforward. This section looks at a more advanced pattern where volatile might offer a performance or scalability benefit.

The more advanced patterns for using volatile can be extremely fragile. It is critical that your assumptions be carefully documented and these patterns strongly encapsulated because very small changes can break your code! Also, given that the primary motivation for the more advanced volatile use cases is performance, be sure that you actually have a demonstrated need for the purported performance gain before you start applying them. These patterns are trade-offs that give up readability or maintainability in exchange for a possible performance boost -- if you don't need the performance boost (or can't prove you need it through a rigorous measurement program), then it is probably a bad trade because you're giving up something of value and getting something of lesser value in return.

Pattern #5: The cheap read-write lock trick

By now, it should be well-known that volatile is not strong enough to implement a counter. Because `++x` is really shorthand for three operations (read, add, store), with some unlucky timing it is possible for updates to be lost if multiple threads tried to increment a volatile counter at once.

However, if reads greatly outnumber modifications, you can combine intrinsic locking and volatile variables to reduce the cost on the common code path. Listing 6 shows a thread-safe counter that uses `synchronized` to ensure that the increment operation is atomic and uses `volatile` to guarantee the visibility of the current result. If updates are infrequent, this approach may perform better as the overhead on the read path is only a volatile read, which is generally cheaper than an uncontended lock acquisition.

Listing 6. Combining volatile and synchronized to form a "cheap read-write lock"

```
@ThreadSafe
public class CheesyCounter {
    // Employs the cheap read-write lock trick
    // All mutative operations MUST be done with the 'this' lock held
    @GuardedBy("this") private volatile int value;

    public int getValue() { return value; }

    public synchronized int increment() {
        return value++;
    }
}
```

The reason this technique is called the "cheap read-write lock" is that you are using different synchronization mechanisms for reads and writes. Because the writes in this case violate the first condition for using volatile, you cannot use volatile to safely implement the counter -- you must use locking. However, you can use volatile to ensure the *visibility* of the current value when reading, so you use locking for all mutative operations and volatile for read-only operations. Where locks only allow one thread to access a value at once, volatile reads allow more than one, so when you use volatile to guard the read code path, you get a higher degree of sharing than you would were you to use locking for all code paths -- just like a read-write lock. However, bear in mind the fragility of this pattern: With two competing synchronization mechanisms, this can get very tricky if you branch out beyond the most basic application of this pattern.

Summary

Volatile variables are a simpler -- but weaker -- form of synchronization than locking, which in some cases offers better performance or scalability than intrinsic locking. If you follow the conditions for using volatile safely -- that the variable is truly independent of both other variables and its own prior values -- you can sometimes simplify code by using `volatile` instead of `synchronized`. However, code using `volatile` is often more fragile than code using locking. The patterns offered here cover the most common cases where `volatile` is a sensible alternative to `synchronized`. Following these patterns -- taking care not to push them beyond their limits -- should help you safely cover the majority of cases where volatile variables are a win.

Related topics

- *Java Concurrency in Practice*: The how-to manual for developing concurrent programs in Java code, including constructing and composing thread-safe classes and programs, avoiding liveness hazards, managing performance, and testing concurrent applications.
- Going Atomic: Describes the atomic variable classes added in Java 5.0, which extend the concept of volatile variables to support atomic state transitions.
- An introduction to nonblocking algorithms: Describes how concurrent algorithms can be implemented without locks, using atomic variables.
- Volatiles: More about volatile variables from Wikipedia.