Learn › Java development

Java theory and practice

Managing volatility

Guidelines for using volatile variables

Brian Goetz
Published on June 19, 2007

Content series:

+ This content is part of the series: **Java theory and practice**

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

Locks offer two primary features: *mutual exclusion* and *visibility*. Mutual exclusion means that o hold a given lock, and this property can be used to implement protocols for coordinating access one thread at a time will be using the shared data. Visibility is more subtle and has to do with er shared data prior to releasing a lock are made visible to another thread that subsequently acqu visibility guarantees provided by synchronization, threads could see stale or inconsistent values could cause a host of serious problems.

Volatile variables share the visibility features of `synchronized`, but none of the atomicity featur
automatically see the most up-to-date value for volatile variables. They can be used to provide
restricted set of cases: those that do not impose constraints between multiple variables or betv
and its future values. So volatile alone is not strong enough to implement a counter, a mutex, or
that relate multiple variables (such as "start <=end").

You might prefer to use volatile variables instead of locks for one of two principal reasons: simp
idioms are easier to code and read when they use volatile variables instead of locks. In addition
locks) cannot cause a thread to block, so they are less likely to cause scalability problems. In si
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. Bot
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 vari
other program state, including the variable's current state.

The first condition disqualifies volatile variables from being used as thread-safe counters. While
may look like a single operation, it is really a compound read-modify-write sequence of operati
atomically -- and volatile does not provide the necessary atomicity. Correct operation would rec
unchanged for the duration of the operation, which cannot be achieved using volatile variables.
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 volatil
applicable approach to achieving thread-safety than `synchronized`. Listing 1 shows a non-threa
contains an invariant -- that the lower bound is always less than or equal to the upper b d.

Listing 1. Non-thread-safe number range class

```

    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 u be sufficient to make the class thread-safe; synchronization would still be needed. Otherwise, v threads executing `setLower` and `setUpper` with inconsistent values could leave the range in an i example, if the initial state is `(0, 5)`, and thread A calls `setLower(4)` at the same time that thre the operations are interleaved just wrong, both could pass the checks that are supposed to prot with the range holding `(4, 3)` -- an invalid value. We need to make the `setLower()` and `setUpp` 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 vola using the corresponding locking. A secondary motivation for using volatile variables is performa 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 th comes to intrinsic JVM operations. (For example, the VM may be able to remove locking entirely makes it hard to talk about the relative cost of `volatile` vs. `synchronized` in the abstract.) That processor architectures, volatile reads are cheap -- nearly as cheap as nonvolatile reads. Volatil expensive than nonvolatile writes because of the memory fencing required to guarantee visibilit than lock acquisition.

Unlike locking, volatile operations will never block, so volatiles offer some scalability advantage where they can be used safely. In cases where reads greatly outnumber writes, volatile variable

Patterns for using volatile correctly

Many concurrency experts tend to guide users away from using volatile variables at all, becaus correctly than locks. However, some well-defined patterns exist, which, if you follow them caref wide variety of situations. Always keep in mind the rules about the limits of where volatile can b state that is truly independent of everything else in your program -- and this should keep you fr patterns into dangerous territory.

Pattern #1: status flags

Perhaps the canonical use of volatile variables is simple boolean status flags, indicating that an 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, 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 -- such, some form of synchronization is required to ensure the proper visibility of the `shutdownRe` called from a JMX listener, an action listener in the GUI event thread, through RMI, through a W However, coding the loop with `synchronized` blocks would be much more cumbersome than co flag as in Listing 2. Because volatile simplifies the coding, and the status flag does not nd c program, this is a good use for volatile.

flags that can change back and forth, but only if it is acceptable for a transition cycle (from `fals` undetected. Otherwise, some sort of atomic state transition mechanism is needed, such as ator

Pattern #2: one-time safe publication

The visibility failures that are possible in the absence of synchronization can get even trickier to object references instead of primitive values. In the absence of synchronization, it is possible tc object reference that was written by another thread and still see stale values for that object's st the problem with the infamous double-checked-locking idiom, where an object reference is rea the risk is that you could see an up-to-date reference but still observe a partially constructed ol

One technique for safely publishing an object is to make the object reference volatile. Listing 3 s during startup, a background thread loads some data from a database. Other code, when it migl 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 seei `Flooble` as it dereferences the `theFlooble` reference.

visibility of the object in its as-published form, but if the state of the object is going to change af synchronization is required.

Pattern #3: independent observations

Another simple pattern for safely using volatile is when observations are periodically "publishec program. For example, say there is an environmental sensor that senses the current temperatur read this sensor every few seconds and update a volatile variable containing the current temper 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 hc mechanism might remember the name of the last user to have logged on. The `lastUser` referen 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 w of publication being a one-time event, it is a series of independent events. This pattern requires be effectively immutable -- that its state not change after publication. Code consuming the valu change at any time.

Pattern #4: the "volatile bean" pattern

volatile bean pattern is that many frameworks provide containers for mutable data holders (for 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 a they must contain no logic other than getting or setting the appropriate property. Further, for da references, the referred-to objects must be effectively immutable. (This prohibits having array-v array reference is declared `volatile`, only the reference, not the elements themselves, have vo volatile variable, there may be no invariants or constraints involving the properties of the JavaBe 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 se This section looks at a more advanced pattern where volatile might offer a performance or scala

The more advanced patterns for using volatile can be extremely fragile. It is critical that your as documented and these patterns strongly encapsulated because very small changes can break y primary motivation for the more advanced volatile use cases is performance, be sure that you a need it through a rigorous measurement program), then it is probably a bad trade because you'ı 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. Beca three operations (read, add, store), with some unlucky timing it is possible for updates to be los increment a volatile counter at once.

However, if reads greatly outnumber modifications, you can combine intrinsic locking and volati on the common code path. Listing 6 shows a thread-safe counter that uses `synchronized` to en operation is atomic and uses `volatile` to guarantee the visibility of the current result. If update may perform better as the overhead on the read path is only a volatile read, which is generally c 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 sy reads and writes. Because the writes in this case violate the first condition for using volatile, you implement the counter -- you must use locking. However, you can use volatile to ensure the *visi* reading, so you use locking for all mutative operations and volatile for read-only operations. Wh to access a value at once, volatile reads allow more than one, so when you use volatile to guard higher degree of sharing than you would were you to use locking for all code paths -- ju e a in mind the fragility of this pattern: With two competing synchronization mechanisms, t can g beyond the most basic application of this pattern.

Volatile variables are a simpler -- but weaker -- form of synchronization than locking, which in s
performance or scalability than intrinsic locking. If you follow the conditions for using volatile sa
independent of both other variables and its own prior values -- you can sometimes simplify cod
`synchronized`. However, code using `volatile` is often more fragile than code using locking. The
the most common cases where `volatile` is a sensible alternative to `synchronized`. Following tl
to push them beyond their limits -- should help you safely cover the majority of cases where vol

Downloadable resources

PDF of this content

Related topics

- *Java Concurrency in Practice*: The how-to manual for developing concurrent programs in Jav and composing thread-safe classes and programs, avoiding liveness hazards, managing per concurrent applications.
- Going Atomic: Describes the atomic variable classes added in Java 5.0, which extend the co support atomic state transitions.
- An introduction to nonblocking algorithms: Describes how concurrent algorithms can be imp atomic variables.
- Volatiles: More about volatile variables from Wikipedia.

Comments

Sign in or register to add and subscribe to comments.

☐ Subscribe me to comment notifications

About

Help

Submit content

Report abuse

Third-party notice

Community

Product feedback

Developer Centers

Follow us

Join

Faculty

Students

Startups

Business Partners

developerWorks®

Learn

Develop

Connect

Select a language

English

中文

日本語

Русский

Português (Brasil)

Español

한글

Tutorials & training

Demos & sample code

Q&A forums