Java concurrency scenario -- do I need synchronization or not?

Asked 16 years, 1 month ago Modified 11 years, 3 months ago Viewed 3k times

16

Here's the deal. I have a hash map containing data I call "program codes", it lives in an object, like so:

```
Class Metadata
{
    private HashMap validProgramCodes;
    public HashMap getValidProgramCodes() { return validProgramCodes; }
    public void setValidProgramCodes(HashMap h) { validProgramCodes = h; }
}
```

I have lots and lots of reader threads each of which will call getValidProgramCodes() once and then use that hashmap as a read-only resource.

So far so good. Here's where we get interesting.

I want to put in a timer which every so often generates a new list of valid program codes (never mind how), and calls setValidProgramCodes.

My theory -- which I need help to validate -- is that I can continue using the code as is, without putting in explicit synchronization. It goes like this: At the time that validProgramCodes are updated, the value of validProgramCodes is always good -- it is a pointer to either the new or the old hashmap. *This is the assumption upon which everything hinges.* A reader who has the old hashmap is okay; he can continue to use the old value, as it will not be garbage collected until he releases it. Each reader is transient; it will die soon and be replaced by a new one who will pick up the new value.

Does this hold water? My main goal is to avoid costly synchronization and blocking in the overwhelming majority of cases where no update is happening. We only update once per hour or so, and readers are constantly flickering in and out.

java concurrency

Share Improve this question Follow

asked Nov 18, 2008 at 22:07

storkman

161 ● 2 ● 3

10 Answers

Sorted by: Highest score (default)

28

Use Volatile

Is this a case where one thread cares what another is doing? Then the JMM FAQ has the answer:

> Most of the time, one thread doesn't care what the other is doing. But when it does, that's what synchronization is for.

In response to those who say that the OP's code is safe as-is, consider this: There is nothing in Java's memory model that guarantees that this field will be flushed to main memory when a new thread is started. Furthermore, a JVM is free to reorder operations as long as the changes aren't detectable within the thread.

Theoretically speaking, the reader threads are not guaranteed to see the "write" to validProgramCodes. In practice, they eventually will, but you can't be sure when.

I recommend declaring the validProgramCodes member as "volatile". The speed difference will be negligible, and it will guarantee the safety of your code now and in future, whatever JVM optimizations might be introduced.

Here's a concrete recommendation:

```
import java.util.Collections;

class Metadata {

    private volatile Map validProgramCodes = Collections.emptyMap();

    public Map getValidProgramCodes() {
      return validProgramCodes;
    }

    public void setValidProgramCodes(Map h) {
      if (h == null)
        throw new NullPointerException("validProgramCodes == null");
      validProgramCodes = Collections.unmodifiableMap(new HashMap(h));
    }

}
```

Immutability

In addition to wrapping it with `unmodifiableMap`, I'm copying the map (`new HashMap(h)`). This makes a snapshot that won't change even if the caller of setter

continues to update the map "h". For example, they might clear the map and add fresh entries.

Depend on Interfaces

On a stylistic note, it's often better to declare APIs with abstract types like `List` and `Map`, rather than a concrete types like `ArrayList` and `HashMap.` This gives flexibility in the future if concrete types need to change (as I did here).

Caching

The result of assigning "h" to "validProgramCodes" may simply be a write to the processor's cache. Even when a new thread starts, "h" will not be visible to a new thread unless it has been flushed to shared memory. A good runtime will avoid flushing unless it's necessary, and using `volatile` is one way to indicate that it's necessary.

Reordering

Assume the following code:

```
HashMap codes = new HashMap();
codes.putAll(source);
meta.setValidProgramCodes(codes);
```

If `setValidCodes` is simply the OP's `validProgramCodes = h;`, the compiler is free to reorder the code something like this:

```
1: meta.validProgramCodes = codes = new HashMap();
2: codes.putAll(source);
```

Suppose after execution of writer line 1, a reader thread starts running this code:

```
1: Map codes = meta.getValidProgramCodes();
2: Iterator i = codes.entrySet().iterator();
3: while (i.hasNext()) {
4:   Map.Entry e = (Map.Entry) i.next();
5:   // Do something with e.
6: }
```

Now suppose that the writer thread calls "putAll" on the map between the reader's line 2 and line 3. The map underlying the Iterator has experienced a concurrent

modification, and throws a runtime exception—a devilishly intermittent, seemingly inexplicable runtime exception that was never produced during testing.

Concurrent Programming

Any time you have one thread that cares what another thread is doing, you *must* have some sort of memory barrier to ensure that actions of one thread are visible to the other. If an event in one thread must happen before an event in another thread, you *must* indicate that explicitly. There are no guarantees otherwise. In practice, this means `volatile` or `synchronized`.

Don't skimp. It doesn't matter how fast an incorrect program fails to do its job. The examples shown here are simple and contrived, but rest assured, they illustrate real-world concurrency bugs that are incredibly difficult to identify and resolve due to their unpredictability and platform-sensitivity.

Additional Resources

- The Java Language Specification - 17 Threads and Locks sections: §17.3 and §17.4
- The JMM FAQ
- Doug Lea's concurrency books

Share
Improve this answer
Follow

edited Jun 14, 2012 at 18:22

reevesy
3,472 ● 1 ● 27 ● 23

answered Nov 18, 2008 at 22:14

erickson
269k ● 59 ● 401 ● 497

4

No, the code example is not safe, because there is no **safe publication** of any new HashMap instances. Without any synchronization, there is a possibility that a reader thread will see a *partially initialized* HashMap.

Check out @erickson's explanation under "Reordering" in his answer. Also I can't recommend Brian Goetz's book Java Concurrency in Practice enough!

Whether or not it is okay with you that reader threads might see old (stale) HashMap references, or might even never see a new reference, is beside the point. The worst thing that can happen is that a reader thread might obtain reference to and attempt to access a HashMap instance that is not yet initialized and not ready to be accessed.

Share Improve this answer Follow

answered Nov 20, 2008 at 7:23

Scott Bale
10.8k ● 5 ● 35 ● 36

No, by the Java Memory Model (JMM), this is not thread-safe.

There is no *happens-before* relation between writing and reading the `HashMap` implementation objects. So, although the writer thread appears to write out the object first and then the reference, a reader thread may not see the same order.

As also mentioned there is no guarantee that the reaer thread will ever see the new value. In practice with current compilers on existing hardware the value should get updated, unless the loop body is sufficiently small that it can be sufficiently inlined.

So, making the reference `volatile` is adequate under the new JMM. It is unlikely to make a substantial difference to system performance.

The moral of this story: Threading is difficult. Don't try to be clever, because sometimes (may be not on your test system) you wont be clever enough.

Share Improve this answer Follow

answered Nov 19, 2008 at 15:09

Tom Hawtin - tackline

147k ● 30 ● 221 ● 312

3

As others have already noted, this is not safe and you shouldn't do this. You need either volatile or synchronized here to force other threads to see the change.

What hasn't been mentioned is that synchronized and especially volatile are probably a lot faster than you think. If it's actually a performance bottleneck in your app, then I'll eat this web page.

Another option (probably slower than volatile, but YMMV) is to use a ReentrantReadWriteLock to protect access so that multiple concurrent readers can read it. And if that's still a performance bottleneck, I'll eat this whole web site.

```
public class Metadata
{
  private HashMap validProgramCodes;
  private ReadWriteLock lock = new ReentrantReadWriteLock();

  public HashMap getValidProgramCodes() {
    lock.readLock().lock();
    try {
      return validProgramCodes;
    } finally {
      lock.readLock().unlock();
    }
  }

  public void setValidProgramCodes(HashMap h) {
```

```
      lock.writeLock().lock();
      try {
        validProgramCodes = h;
      } finally {
        lock.writeLock().unlock();
      }
    }
  }
```

Share Improve this answer Follow

answered Nov 20, 2008 at 15:44

Alex Miller

70.1k ● 25 ● 124 ● 168

2

I think your assumptions are correct. The only thing I would do is set the `validProgramCodes` volatile.

```
private volatile HashMap validProgramCodes;
```

This way, when you update the "pointer" of `validProgramCodes` you guaranty that all threads access the same latest `HasMap` "pointer" because they don't rely on local thread cache and go directly to memory.

Share Improve this answer Follow

answered Nov 18, 2008 at 22:23

bruno conde

48.2k ● 15 ● 102 ● 118

1

The assignment will work as long as you're not concerned about reading stale values, and as long as you can guarantee that your hashmap is properly populated on initialization. You should at the least create the hashMap with Collections.unmodifiableMap on the Hashmap to guarantee that your readers won't be changing/deleting objects from the map, and to avoid multiple threads stepping on each others toes and invalidating iterators when other threads destroy.

(writer above is right about the volatile, should've seen that)

Share Improve this answer Follow

answered Nov 18, 2008 at 22:18

Steve B.

57.2k ● 12 ● 97 ● 134

Just to be clear, the way you guarantee that "your hashmap is properly populated on initialization" is to use synchronization :) Either synchronized, volatile, or something from java.util.concurrent. – Scott Bale Nov 20, 2008 at 17:15

1

While this is not the best solution for this particular problem (erickson's idea of a new unmodifiableMap is), I'd like to take a moment to mention the java.util.concurrent.ConcurrentHashMap class introduced in Java 5, a version of HashMap specifically built with concurrency in mind. This construct does **not** block on reads.

Share Improve this answer Follow

answered Nov 21, 2008 at 13:45

Powerlord
88.7k ● 17 ● 129 ● 177

0

Check this post about concurrency basics. It should be able to answer your question satisfactorily.

http://walivi.wordpress.com/2013/08/24/concurrency-in-java-a-beginners-introduction/

Share Improve this answer Follow

answered Aug 28, 2013 at 11:44

Nazgul
1,902 ● 1 ● 11 ● 15

-1

I think it's risky. Threading results in all kinds of subtly issues that are a giant pain to debug. You might want to look at `FastHashMap`, which is intended for read-only threading cases like this.

At the least, I'd also declare `validProgramCodes` to be `volatile` so that the reference won't get optimized into a register or something.

Share Improve this answer Follow

answered Nov 18, 2008 at 22:13

sblundy
61.4k ● 22 ● 123 ● 124

1 FastHashMap is absurdly risky! – Tom Hawtin - tackline Nov 19, 2008 at 15:00

-3

If I read the JLS correctly (no guarantees there!), accesses to *references* are always atomic, period. See Section 17.7 Non-atomic Treatment of double and long

So, if the access to a reference is always atomic and it doesn't matter what instance of the returned `Hashmap` the threads see, you should be OK. You won't see partial writes to the reference, ever.

Edit: After review of the discussion in the comments below and other answers, here are references/quotes from

Doug Lea's book (Concurrent Programming in Java, 2nd Ed), p 94, section 2.2.7.2 *Visibility*, item #3: "

> The first time a thread access a field of an object, it sees either the initial value of the field or the value since written by some other thread."

On p. 94, Lea goes on to describe risks associated with this approach:

> The memory model guarantees that, given the eventual occurrence of the above operations, a particular update to a particular field made by one thread will eventually be visible to another. But *eventually* can be an arbitrarily long time.

So when it ***absolutely, positively, must be visible*** to any calling thread, `volatile` or some other synchronization barrier is required, especially in long running threads or threads that access the value in a loop (as Lea says).

However, in the case where there is a *short lived thread*, as implied by the question, with new threads for new readers and it *does not impact* the application to read stale data, synchronization *is not* ***required.***

@erickson's answer is the safest in this situation, guaranteeing that other threads will see the changes to the `HashMap` reference as they occur. I'd suggest following that advice simply to avoid the confusion over the requirements and implementation that resulted in the "down votes" on this answer and the discussion below.

I'm not deleting the answer in the hope that it will be useful. I'm **not** looking for the "Peer Pressure" badge... ;-)

Share

Improve this answer

Follow

edited May 23, 2017 at 10:27

Community Bot

1 ● 1

answered Nov 18, 2008 at 22:35

Ken Gentle

13.3k ● 2 ● 43 ● 49

---

1 As I answered and commented elsewhere, the bigger potential problem hasn't been mentioned here yet, which is that w/o synchronization a new HashMap is not safely published, and so may become visible to reader threads IN AN INVALID STATE (i.e. it's content not yet visible to the reader threads). – Scott Bale Nov 20, 2008 at 15:31

---

1 The assignment of a new HashMap instance to the "validProgramCodes" field is not the issue. Yes, assignment is atomic. In the absence of proper synchronization, the assignment of that

field AND THE POPULATION OF THAT MAP may appear reordered from the POV of a reader thread. Google "Safe Publication" – Scott Bale Nov 21, 2008 at 17:15

1 @Scott: I've already acknowledged that @erickson's answer is the safest. Yes, you're correct, partial initialization of the HashMap is possible. So do I delete this answer or not? It is not incorrect - just different. – Ken Gentle Nov 21, 2008 at 17:49

1 @Ken "validProgramCodes" type is HashMap, not Map, so it cannot refer to anything other than a HashMap (or HashMap subclass). But aside from that, Collections$SynchronizedMap is only safe if it is safely published, it's constructor does not have any synchronization, so even that idea would not work. – Scott Bale Nov 21, 2008 at 20:47

1 @Scott: DOH! on the Map/syncrhonizedMap/HashMap - however, the assumption about initialization stands. On whether I would approve the API, it depends on the requirements and the entire class implementation. I'm assuming the example is intentionally incomplete and would not be committed as is. – Ken Gentle Nov 21, 2008 at 21:41