AtomicReference

Complete guide to understanding and effectively working with AtomicReference class. Learn the differences among atomic assignment of references in Java, volatile variables and the AtomicReference class.

*If you are interviewing, consider buying our **number#1** course for **Java Multithreading Interviews**.*

Overview

A reference type is a data type that represents an instance of a Java class, i.e. it is a type other than one of the Java's primitive types. For instance:

```
Long myLong = new Long(5);
```

In the above snippet the variable `myLong` represents a reference type. When we create the above object `myLong`, Java allocates appropriate memory for the object to be stored. The variable `myLong` points to this address in the memory, which stores the object. The address of the object in the memory is called the *reference*. Consider the below snippet:

```
Long myLong = new Long(5);
Long otherLong = myLong;
```

The two variables `myLong` and `otherLong` both point to the same memory location where the `Long` object is stored or we can say both variables are assigned the same reference (address) to the object but not the object itself.

Given the above context, it becomes easier to understand the `AtomicReference` class. According to the official documentation, `AtomicReference` allows us to atomically update the reference to an object. Before we further delve into the topic, we'll first clarify the distinction between `AtomicReference` and assignment of references atomically in Java.

Atomic assignment of references in Java

Consider the following snippet:

```
class SomethingUseful {
   Thread thread;
   int myInt;
   HashMap<String, String> map;
   double myDouble;
   long myLong;

    void usefulFunction(int passedInt, long passedLong, double passedDouble, HashMap<String, String> pas
    |   thread = Thread.currentThread();
    |   myInt = passedInt;
    |   map = passedMap;
    }

}
```

If the method `usefulFunciton()` is invoked by several threads, can we see garbage values for the variables of an instance of class `SomethingUseful`? In the method `usefulFunciton()` there are two primitive type assignments and two reference assignments. The following holds true for assignments in Java as per the language specification:

- All reference assignments are atomic. This doesn't mean our method `usefulFunction()` is thread-safe. It just means that when a thread assigns the variables `map` or the `thread`, the assignment is atomic, i.e. it can't happen that the variable `thread` or `map` hold some bytes from the assignment operation of one thread and other bytes from the assignment operation of an another thread. Whatever reference the two variables hold will reflect an assignment from one of the threads. Reference reads and writes are always atomic whether the reference itself consists of 32 or 64 bits.
- Assignments and reads for primitive data types except for `double` and `long` are always atomic. If two threads are invoking the method `usefulFunction()` and passing in 5 and 7 for the integer variable then the variable `myInt` will hold either 5 or 7 and not any other value.
- The reads and writes to `double` and `long` primitive types aren't atomic. The JVM specification allows implementations to break-up the write of the 64 bit `double` or `long` primitive type into two writes, one for each 32 bit half. This can result in a situation where two threads are simultaneously writing a `double` or `long` value and a third thread observes the first 32 bits from the write by the first thread and the next 32 bits from the write by the second thread. As a result the third thread reads a value that has neither been written by either of the two threads or is a garbage value. In order to make reads and writes to `double` or `long` primitive types atomic, we must mark them as `volatile`. The specification guarantees writes and reads to volatile `double` and `long` primitive types as atomic. Note that some JVM implementations may make the writes and reads

to `double` and `long` types as atomic but this isn't universally guaranteed across all the JVM implementations.

So far we understood what a reference in Java means and that except for two primitive types all assignments are atomic in Java. This may confuse some readers as to the purpose and intent of the class `AtomicReference` when assignments in Java are mostly atomic and the ones to `double` and `long` can be made atomic using volatile. We'll address this question next.

`AtomicReference` class

The class `AtomicReference` promises assignment of a reference atomically, however, it offers far more in functionality than just an atomic assignment. Consider the snippet below:

```
// Regular variables
Thread thread;
int myInt;

// Assignments are indeed atomic, however, the method isn't thread-safe and the threads
// can potentially cache a stale value of the two variables in their cache.
void myFunction(int passedInt) {
    myInt = passedInt;
    thread = Thread.currentThread();
}
```

The variables `thread` and `myInt` can both be cached and a thread may not observe the latest value for them.

Next, we can mark the variable `thread` as volatile.

```
// Marking Thread reference as volatile
volatile Thread thread;

// No change here.
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. The memory
// visibility guarantees are now different. If a thread executes the function and then
// another thread reads the Thread reference, the reading thread will also see the latest
// value for myInt even though it is not marked as volatile.
void myFunction(int passedInt) {
    myInt = passedInt;
    thread = Thread.currentThread();
}
```

In the above snippet, the threads reading the `thread` variable also see the latest value for the variable `myInt` because the variable `myInt` is assigned to before the `thread` variable. Since `thread` is marked volatile it establishes the *happens-before* relations for the variables that are assigned earlier than it.

We can rewrite the above snippet using the `AtomicReference` class as follows:

```
// Using an atomic reference
AtomicReference<Thread> threadAtomicReference = new AtomicReference<>();
// No change here.
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. The memory
// visibility guarantees are similar to the snippet with the volatile thread
// variable.
void myFunction(int passedInt) {
    myInt = passedInt;
    threadAtomicReference.set(Thread.currentThread());
}
```

The above snippet is equivalent of the snippet with `thread` marked as volatile.

Using `AtomicReference` we establish the *happen-before* relationship similar to when a `volatile` variable is written or read. Reading or writing to a volatile variable guarantees that the variable value is written to and read from the main memory and not the cache. Therefore all threads observing a volatile variable read the most up to date value for that variable and avoid using a stale value for the variable from their cache. The `get()` and `set()` operations on the `AtomicReference` variable promise similar guarantees as the read and write of a volatile variable respectively.

However, also note that the assignment or initialization of an `AtomicReference` doesn't guarantee a happens-before relationship. For instance,

```
class SomeClass {
   AtomicReference<Object> atomicReference;

   public void init(Object obj) {
       atomicReference = new AtomicReference<Object>(obj);
   }

   // ... Rest of class definition
}
```

In the above snippet, the `atomicReference` is being initialized or assigned to and unless the `atomicReference` variable is marked volatile, a happens-before relationship isn't established.

check-then-act

The astute reader would question the usability of the `AtomicReference` class if we can use `volatile`, as we have done in our previous snippets, to achieve the same functionality. The

crucial functionality `AtomicReference` offers is to execute algorithms or actions that check the value of a variable and then act upon it based on the observed value, also known as *check-then-act*, atomically. We slightly modify our previous snippet to create a check-then-act scenario below:

```
// Marking Thread reference as volatile
volatile Thread thread;
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. Now more
// than two threads can enter the if-clause even though the thread variable
// is marked volatile. Remember volatile doesn't mean thread-safety!
void myFunction(int passedInt) {
    myInt = passedInt;
    if (thread == null) {
        thread = Thread.currentThread();
    }
}
```

In the above scenario, we intend for a single thread to find the `thread` variable null and initialize it. We want the two steps - check if `thread` is null and act assign`thread` - to be performed atomically, which `volatile` can't help us with. In fact, `volatile` is only limited to offering stronger memory visibility guarantees and that's about it. This is where `AtomicReference` comes in and allows us to execute the two steps atomically. The same snippet re-written with `AtomicReference` appears below:

```
// Using an atomic reference
AtomicReference<Thread> threadAtomicReference = new AtomicReference<>();
int myInt;

// Assignments are atomic, however, the method isn't thread-safe. Only a
// single thread ever executes the if-clause.
void myFunction(int passedInt) {
    myInt = passedInt;
    if (threadAtomicReference.compareAndSet(null, Thread.currentThread())) {
        thread = Thread.currentThread();
    }
}
```

Take a moment to ponder on the above snippet and observe the following facts:

1. The method as a whole is still not thread-safe. The value of `myInt` may be from a thread that didn't initialize the `threadAtomicReference` variable.
2. If the variable `myInt` is moved **after** the `if-clause` the *happens-before* isn't established between variables `threadAtomicReference` and `myInt` and the value of `myInt` may only be updated in the cache and not the main memory. A thread that subsequently reads the variable `threadAtomicReference` will fetch all the variables visible to it from the main memory as per the volatile variable visibility guarantee but since `myInt` was never updated in the main memory, the reader thread may observe a stale value for the `myInt` variable.
3. The initialization is guaranteed to be atomic and only a single thread ever enters the `if-clause`.

Example

In our code widgets below, we'll construct three versions of the same example. We have several threads all attempting to simultaneously initialize a variable `firstThread` to the current thread reference using `Thread.currentThread()`. If the threads find the `firstThread` variable `null` it simply initializes the variable to itself. Here's the first version using volatile variable:

Java

main.java Saved

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {
    static volatile Thread firstThread = null;

    public static void main( String args[] ) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(50);

        try {
```

Run

If you run the above widget enough times, you'll see some runs show multiple threads entering the `if-clause`. We can make the initialization thread-safe by using a syncrhonized as shown below:

Java

main.java Saved

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;
```

```
class Demonstration {

    static Thread firstThread = null;

    public static void main( String args[] ) throws Exception {

        ExecutorService executor = Executors.newFixedThreadPool(50);
```

Run

In the above code-snippet the initialization is thread-safe and only one thread ever executes the `if-clause`. Finally, we can rewrite the above code using the `AtomicReference` class as shown below:

Java — Press + to interact — main.java — Saved

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {

    static AtomicReference<Thread> atomicReference = new AtomicReference<>();

    public static void main( String args[] ) throws Exception {

        ExecutorService executor = Executors.newFixedThreadPool(50);
```

Run

The above code uses `AtomicReference` to initialize the static variable `atomicReference` in a thread-safe manner. Only a single thread is able to initialize the variable while all others find the variable not null. In terms of performance, under low to moderate contention `AtomicReference` is expected to perform better than the same code written using `synchronized` or other locking mechanisms.

Conclusion

Generally speaking, it is easy to get confused among the three concepts atomic reference/primitive assignments, `volatile` and `AtomicReference`. It is important to understand the distinction between assignment of reference and primitive types being atomic and the class `AtomicReference`. Additionally, `AtomicReference` goes beyond `volatile` and allows for executing check-then-act type scenarios to be executed atomically.

← Back lesson
DoubleAccumulator

Mark As Completed
Next →
AtomicReferenceArray