How AtomicReference Works in Java

by Sankar Banerjee · May. 16, 18 · Java Zone · Tutorial

AtomicReference is still not clear to some people, so I would like to say a few words about it and provide a GitHub link with full-fledged running code.

AtomicReference refers to an object reference. This reference is a volatile member variable in the AtomicReference instance as below.

```
1  private volatile V value;
```

get() simply returns the latest value of the variable (as volatiles do in a "happens before" manner).

```
1  public final V get()
```

```
1  public final boolean  compareAndSet(V expect, V update) {
2          return unsafe.compareAndSwapObject(this, valueOffset, expect, update);
3  }
```

The compareAndSet(expect,update) method calls the compareAndSwapObject() method of the unsafe class of Java. This method call of unsafe invokes the native call, which invokes a single instruction to the processor. "expect" and "update" each reference an object.

If and only if the AtomicReference instance member variable "value" refers to the same object is referred to by "expect", "update" is assigned to this instance variable now, and "true" is returned. Or else, false is returned. The whole thing is done atomically. No other thread can intercept in between.

The main advantage is that we do not need to use the resource consuming synchronized keyword. As we call synchronized, the following happens.

1. The cache and registers are flushed for the running thread, which will eventually have the monitor.

2. We create a memory barrier, and only this thread has the monitor of the object we are synchronizing.

3. After the synchronized block ends, the variables are written into memory.

But in the case of compareAndSet(...,...) all of the above do not happen.

I have created a very small example of a ticket booking program and posted it to GitHub. The single file application can be downloaded and run in Eclipse. It is self-explanatory, and here, I provide the snippet that will clarify what the program is trying to do.

```
for (int i = 0; i < 4; i++) {// 4 seats, user threads will try to reserve seats
    seats.add(new AtomicReference<Integer>());
}
Thread[] ths = new Thread[8];// 8 users, each is a thread
for (int i = 0; i < ths.length; i++) {
    ths[i] = new MyTh(seats, i);
    ths[i].start();
}
//as the number of users is greater, everyone cannot reserve a seat.
```

Here is the GitHub link again. Just download the single source code file, add it to some Java project in Eclipse, resolve any errors due to import- or package-related issues, and run it.

Cheers!!

Like This Article? Read More From DZone

How Synchronization Works in Java (Part 1)

Java: Using Immutable Classes for Concurrent Programming

PostgreSQL Rocks, Except When it Blocks: Understanding Locks

Free DZone Refcard
Getting Started With Kotlin

Topics: ATOMICREFERENCE , LOCK FREE , SYNCHRONIZED BLOCK IN JAVA , JAVA , TUTORIAL