66% completed

Search Course

AtomicStampedReference

Learn how the AtomicStampedReference class can be used to address the ABA problem that can manifest when classes manage their own memory.

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

Overview

Similar to the `AtomicReference` class, the `AtomicStampedReference` class contains a reference to an object and additionally an integer stamp. The integer stamp is usually incremented whenever the object pointed to by the reference is modified. The `AtomicStampedReference` class can be thought of as a generalization of the `AtomicMarkableReference` class, replacing the boolean with an integer. The integer stamp stored alongside the reference can also be used for other purposes such as maintaining one of the finite states a data structure can be in . The stamp and reference fields can be updated atomically, either together or individually.

The code widget below demonstrates the use of `AtomicStampedReference` and its common APIs.

Java

main.java

Saved

```java
import java.util.concurrent.atomic.AtomicStampedReference;

class Demonstration {

    public static void main( String args[] ) {

        Long myLong = new Long(3);
        Long anotherLong = new Long(7);

        // set the initial stamp to 1 and reference to myLong
        AtomicStampedReference<Long> atomicStampedReference = new AtomicStampedReference<>(m

```

Run

Note that the current reference and stamp can be retrieved using the `get(int[] stampHolder )` call. The reference is the return value from the method invocation and the passed-in array is populated with the current stamp. The `AtomicStampedReference` solves the ABA problem that can occur when using the plain `AtomicReference` class. We'll discuss them next.

ABA problem

The ABA problem is described in detail in the Atomic lesson of the course. But briefly, algorithms that dynamically manage their own memory and use conditional synchronization operations such as `compareAndSet()` can run into this issue. Usually, the pattern involves the `compareAndSet()` method about to modify a reference from say *A* to some other reference but before `compareAndSet()` gets a chance to execute, another thread modifies *A* to *B* and then back to *A*. The `compareAndSet()` proceeds forward and succeeds when its effect on the data structure is not what was initially desired. We'll see a concrete set-up of the ABA problem next.

Example

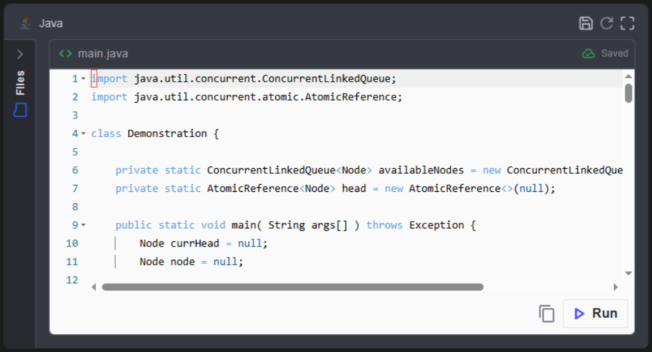
Consider the following example where the `Demonstration` class maintains a concurrent list of free `Node`-s that allows it to recycle nodes and manage its own memory. This is an instructional example to demonstrate the ABA problem and lacks purpose, but the described scenario is applicable to data structures that may choose to conduct memory management on their own for efficiency purposes. In this example, threads either append to or remove the head of a LinkedList of sorts. The setup is as follows:

1. The `head` of the list is stored as an `AtomicReference` and threads can update the head using compare and set API.
2. At the start of the program, the main thread appends ten nodes to the list.
3. Thread1 starts and prepares to dequeue the current `head`. In doing so it has to set the `head` variable to the next of the current head. But just before invoking `compareAndSet()`, we sleep Thread1 to simulate Thread1 being context-switched. Note that at this point, Thread1 has read the current head and the next of current head in local variables.
4. Thread2 starts and immediately sleeps to allow Thread1 to hit its sleep statement and complete the previous step.
5. Thread2 proceeds to dequeue five nodes and places them in the list of free nodes.

Including the head and its next node that was read-in by Thread1 before Thread1 went to sleep.

6. Thread2 now appends a new value to the list but it reuses the node from the free list. The reused Node carries a different value than before but it is the same head value that was read-in by Thread1
7. Thread1 wakes-up and proceeds to execute `compareAndSet()` and since the current `head` is the same reference as the one read-in by Thread1 in step number 3, the `compareAndSet()` succeeds when it should fail. Also the new `head` is set to node that is in the free nodes list. Note that when Thread1 read the head value its value was 10 and when Thread1 executed `compareAndSet()` its value was 99.

The complete example with comments appears in the widget below. Note that we have used `Thread.sleep()` for manifesting the ABA problem, but in practise `Thread.sleep()` should never be used in production code for synchronization among threads.

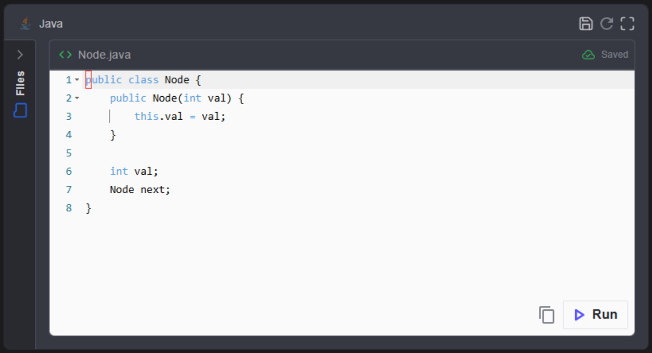


```java
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.atomic.AtomicReference;

class Demonstration {

    private static ConcurrentLinkedQueue<Node> availableNodes = new ConcurrentLinkedQue
    private static AtomicReference<Node> head = new AtomicReference<>(null);

    public static void main( String args[] ) throws Exception {
        Node currHead = null;
        Node node = null;

```

The above result is very interesting, note that the entire code snippet is thread-safe since the operations on the list are always done using `compareAndSet()`, but because of ABA the snippet brings the data structure into a state where the head is set to a reference pointing to a node in the recycle node list.

Fixing the ABA problem with `AtomicStampedReference`

The above example is fixed using the `AtomicStampedReference` class which uses an integer stamp alongside to detect if a node was reused. The same scenario from the previous widget is repeated and the `compareAndSet()` in Thread1 correctly fails.



```java
public class Node {
    public Node(int val) {
        this.val = val;
    }

    int val;
    Node next;
}
```

The above code repeats the same sequence of events as in the previous widget but because we switched to `AtomicStampedReference` class for storing the head of the queue, the `compareAndSet()` call fails in Thread1 and the list remains unmodified from what it looked like after Thread2's operations, when the program exits.

← Back lesson
AtomicReferenceFieldUpdater

Mark As Completed

Next →
AtomicMarkableReference