AtomicReferenceFieldUpdater

Guide to understanding and using AtomicReferenceFieldUpdater.

Overview

The class `AtomicReferenceFieldUpdater` is one of the three field updater classes. The field updater classes exist primarily for performance reasons. Instead of using atomic variables, one can use ordinary variables that occasionally need to be get and then set atomically. Another reason can be to avoid having atomic fields in objects that are short-lived and frequently created e.g. the next pointer of nodes in a concurrent linked list.

The atomicity guarantees for the updater classes are weaker than those of regular atomic classes because the underlying fields can still be modified directly i.e. without using the updater object. Additionally, the atomicity guarantees for `compareAndSet` method stand only with respect to other threads using the updater's methods. The atomic fields present a reflection-based view of an existing `volatile` field that an updater can execute the compare and set method against. Note, that the updater instance isn't tied to any one instance of the target class; rather the updater object can be used to update the target field of any instance of the target class.

Example

classes because the underlying fields can still be modified directly i.e. without using the updater object. Additionally, the atomicity guarantees for `compareAndSet` method stand only with respect to other threads using the updater's methods. The atomic fields present a reflection-based view of an existing `volatile` field that an updater can execute the compare and set method against. Note, that the updater instance isn't tied to any one instance of the target class; rather the updater object can be used to update the target field of any instance of the target class.

Example

As an example consider the `NodeWithAtomicReference` class that uses `AtomicReference` to store the next pointer. The class looks as follows:

```
class Node {
    volatile DemoAtomicReferenceFieldUpdater.Node next;
    int val;

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

We can re-write the above class as `Node` class and store the next pointer in an ordinary `Node` variable.

```
class NodeWithAtomicReference {
    AtomicReference<DemoAtomicReferenceFieldUpdater.NodeWithAtomicReference> next;
    int val;

    public NodeWithAtomicReference(int val) {
        this.val = val;
    }
}
```

We can use the `AtomicReferenceFieldUpdater` object to update the next pointer of the `Node` object. The code widget below demonstrates a `Node` object's next pointer using a `AtomicReferenceFieldUpdater` object.

Java

main.java — Saved

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

class Demonstration {

    // Node class
    static class Node {
        volatile Node next;
        int val;

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

Run

Note, that in the code widget above if we remove `volatile` with the `long` variable, the

updater object will throw an error since only `volatile` fields can be updated using the atomic field updater classes.

← Back lesson

AtomicReferenceArray

☑ Mark As Completed Next →

AtomicStampedReference