Package java.util.concurrent.atomic

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running `double` value updated using a supplied function.

DoubleAdder	One or more variables that together maintain an initially zero `double` sum.
LongAccumulator	One or more variables that together maintain a running `long` value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero `long` sum.

Package java.util.concurrent.atomic Description

A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of `volatile` values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

```
boolean compareAndSet(expectedValue, updateValue);
```

This method (which varies in argument types across different classes) atomically sets a variable to the `updateValue` if it currently holds the `expectedValue`, reporting `true` on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation `weakCompareAndSet` described below.

The specifications of these methods enable implementations to employ efficient machine-level atomic instructions that are available on contemporary processors. However on some platforms, support may entail some form of internal locking. Thus the methods are not strictly guaranteed to be non-blocking -- a thread may block transiently before performing the operation.

Instances of classes `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, and `AtomicReference` each provide access and updates to a single variable of the corresponding type. Each class also provides appropriate utility methods for that type. For example, classes `AtomicLong` and `AtomicInteger` provide atomic increment methods. One application is to generate sequence numbers, as in:

```
class Sequencer {
  private final AtomicLong sequenceNumber
    = new AtomicLong(0);
  public long next() {
    return sequenceNumber.getAndIncrement();
  }
}
```

It is straightforward to define new utility functions that, like `getAndIncrement`, apply a function to a value atomically. For example, given some transformation

```
long transform(long input)
```

write your utility method as follows:

```
long getAndTransform(AtomicLong var) {
  long prev, next;
  do {
    prev = var.get();
    next = transform(prev);
```

```
   } while (!var.compareAndSet(prev, next));
   return prev; // return next; for transformAndGet
 }
```

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in The Java Language Specification (17.4 Memory Model):

- `get` has the memory effects of reading a `volatile` variable.
- `set` has the memory effects of writing (assigning) a `volatile` variable.
- `lazySet` has the memory effects of writing (assigning) a `volatile` variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-`volatile` writes. Among other usage contexts, `lazySet` may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.
- `weakCompareAndSet` atomically reads and conditionally writes a variable but does *not* create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the `weakCompareAndSet`.
- `compareAndSet` and all other read-and-update operations such as `getAndIncrement` have the memory effects of both reading and writing `volatile` variables.

In addition to classes representing single values, this package contains *Updater* classes that can be used to obtain `compareAndSet` operations on any selected `volatile` field of any selected class. `AtomicReferenceFieldUpdater`, `AtomicIntegerFieldUpdater`, and `AtomicLongFieldUpdater` are reflection-based utilities that provide access to the associated field types. These are mainly of use in atomic data structures in which several `volatile` fields of the same node (for example, the links of a tree node) are independently subject to atomic updates. These classes enable greater flexibility in how and when to use atomic updates, at the expense of more awkward reflection-based setup, less convenient usage, and weaker guarantees.

The `AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReferenceArray` classes further extend atomic operation support to arrays of these types. These classes are also notable in providing `volatile` access semantics for their array elements, which is not supported for ordinary arrays.

The atomic classes also support method `weakCompareAndSet`, which has limited applicability. On some platforms, the weak version may be more efficient than `compareAndSet` in the normal case, but differs in that any given invocation of the `weakCompareAndSet` method may return `false` *spuriously* (that is, for no apparent reason). A `false` return means only that the operation may be retried if desired, relying on the guarantee that repeated invocation when the variable holds `expectedValue` and no other thread is also attempting to set the variable will eventually succeed. (Such spurious failures may for example be due to memory contention effects that are unrelated to whether the expected and current values are equal.) Additionally `weakCompareAndSet` does not provide ordering guarantees that are usually needed for synchronization control. However, the method may be useful for updating counters and statistics when such updates are unrelated to the other happens-before orderings of a program. When a thread sees an update to an atomic variable caused by a `weakCompareAndSet`, it does not necessarily see updates to any *other* variables that occurred before the `weakCompareAndSet`. This may be acceptable when, for example, updating performance statistics, but rarely otherwise.

The `AtomicMarkableReference` class associates a single boolean with a reference. For example, this bit might be used inside a data structure to mean that the object being referenced has logically been deleted. The `AtomicStampedReference` class associates an integer value with a reference. This may be used for example, to represent version numbers corresponding to series of updates.

Atomic classes are designed primarily as building blocks for implementing non-blocking data structures and related infrastructure classes. The `compareAndSet` method is not a general replacement for locking. It applies only when critical updates for an object are confined to a *single* variable.

Atomic classes are not general purpose replacements for `java.lang.Integer` and related classes. They do *not* define methods such as `equals`, `hashCode` and `compareTo`. (Because atomic variables are expected to be mutated, they are poor choices for hash table keys.) Additionally, classes are provided only for those types that are commonly useful in intended applications. For example, there is no atomic class for representing `byte`. In those infrequent cases where you would like to do so, you can use an `AtomicInteger` to hold `byte` values, and cast appropriately. You can also hold floats using `Float.floatToRawIntBits(float)` and `Float.intBitsToFloat(int)` conversions, and doubles using `Double.doubleToRawLongBits(double)` and `Double.longBitsToDouble(long)` conversions.

Since:

1.5