

61% completed



Search Course

Java Concurrency Reference

- Setting-up Threads
- Basic Thread Handling
- Executor Framework
- Executor Implementations
- Thread Pools
- Types of Thread Pools
- An Example: Timer vs ScheduledThreadPool
- ThreadPoolExecutor
- Callable Interface
- Future Interface
- CompletionService Interface
- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview



- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview



Java Multithreading for Senior Engineering Interviews / ... / LongAccumulator

LongAccumulator

Comprehensive guide to working with LongAccumulator.

If you are interviewing, consider buying our number#1 course for [Java Multithreading Interviews](#).

Overview

The `LongAccumulator` class is similar to the `LongAdder` class, except that the `LongAccumulator` class allows for a function to be supplied that contains the logic for computing results for accumulation. In contrast to `LongAdder`, we can perform a variety of mathematical operations rather than just addition. The supplied function to a `LongAccumulator` is of type `LongBinaryOperator`. The class `LongAccumulator` extends from the class `Number` but doesn't define the methods `compareTo()`, `equals()`, or `hashCode()` and shouldn't be used as keys in collections such as maps.

An example of creating an accumulator that simply adds long values presented to it

```
// function that will be supplied to an instance of LongAccumulator
LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
    @Override
    public long applyAsLong(long left, long right) {
        | return left + right;
    }
};

// instantiating an instance of LongAccumulator with an initial value of zero
LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator, 0);
```

An example of creating an accumulator that simply adds long values presented to it

```
// function that will be supplied to an instance of LongAccumulator
LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
    @Override
    public long applyAsLong(long left, long right) {
        | return left + right;
    }
};

// instantiating an instance of LongAccumulator with an initial value of zero
LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator, 0);
```

Note that in the above example, we have supplied a function that simply adds the new long value presented to it. The method `applyAsLong` has two operands `left` and `right`. The `left` operand is the current value of the `LongAccumulator`. In the above example, it'll be zero initially, because that is what we are passing-in to the constructor of the `LongAccumulator` instance. The code widget below runs this example and prints the operands and the final sum.

```
Java
main.java
1- import java.util.concurrent.atomic.LongAccumulator;
2- import java.util.function.LongBinaryOperator;
3-
4- class Demonstration {
5-     public static void main( String args[] ) {
6-         // function that will be supplied to an instance of LongAccumulator
7-         LongBinaryOperator longBinaryOperator = new LongBinaryOperator() {
8-             @Override
9-             public long applyAsLong(long left, long right) {
10-                 System.out.println(left + " + " + right);
11-                 return left + right;
12-             }
13-         };
14-
15-         // instantiating an instance of LongAccumulator with an initial value of zero
16-         LongAccumulator longAccumulator = new LongAccumulator(longBinaryOperator, 0);
17-
18-         // adding some values
19-         longAccumulator.accumulate(10L);
20-         longAccumulator.accumulate(20L);
21-         longAccumulator.accumulate(30L);
22-
23-         // printing the final sum
24-         System.out.println("Final sum: " + longAccumulator.get());
25-     }
26- }
```

As you can see we aren't confined to adding long values, rather we can perform as complex operations as desired in the supplied function, which makes the `LongAccumulator` class far more versatile than the `LongAdder` class which is limited to addition. In fact, `LongAdder` can be thought of as a specialized case of `LongAccumulator` for keeping counts and sums.

Distributing contention

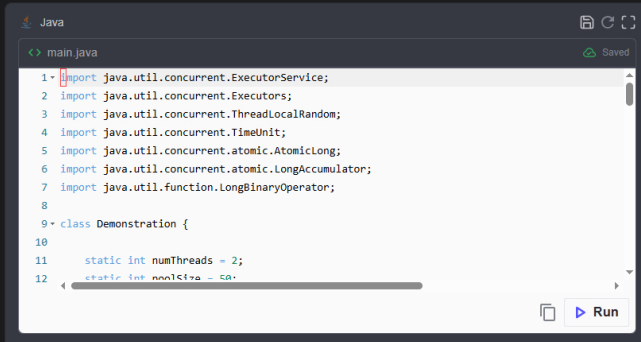
We can achieve the same functionality by using an instance of `AtomicLong` as we can with the `LongAccumulator`, however, the rational for `LongAccumulator` is to distribute contention among threads by maintaining a set of variables that grow dynamically and each one is updated by only a subset of threads. Thus the contention is spread from a single variable to several variables. When the current value is asked for by invoking the `get()` or the `longValue()` methods, all the underlying variables are accumulated by applying the supplied function and the result is returned. The expected throughput of `LongAccumulator` is significantly higher when used in place of `AtomicLong` under high contention. The improved performance comes at the cost of using more space.

Order of accumulation

When multiple threads accumulate an instance of `LongAccumulator`, eventually all the long values in the underlying set are accumulated using the supplied function. The order in which these long values are accumulated isn't guaranteed and the supplied function should produce the same value irrespective of the order in which these values are accumulated. In case, the supplied function isn't commutative i.e., `left + right` isn't the same as `right + left` then the accumulation can produce different results for the same series of accumulated long values.

Example

In the example below, we use the `LongAccumulator` class to keep track of the maximum value observed. There are several threads that use the `ThreadLocalRandom` class to produce a random long value less than 1000, and then attempt to update the instance of `LongAccumulator`. We conduct the same test using `AtomicLong` and time the two tests. Go through the listing which is self-explanatory.



```
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3 import java.util.concurrent.ThreadLocalRandom;
4 import java.util.concurrent.TimeUnit;
5 import java.util.concurrent.atomic.AtomicLong;
6 import java.util.concurrent.atomic.LongAccumulator;
7 import java.util.function.LongBinaryOperator;
8
9 class Demonstration {
10
11     static int numThreads = 2;
12     static int numTries = 50;
```

Note that the above test is crude and imprecise in nature, but does give a general idea of the performance of the two classes under high contention. We can tweak the different parameters such as the number of `iterations` or `numThreads` to produce an environment with different contention characteristics and maybe different results.