BrokenBarrierException
Annotations

Revision & Quizzes

Course Certificate

Practice Mock Interview →

DoubleAccumulator

Comprehensive guide to working with DoubleAccumulator class.

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

Overview

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

An example of creating an accumulator that simply adds double values presented to it appears below:

```java
// function that will be supplied to an instance of DoubleAccumulator
DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
    @Override
    public double applyAsDouble(double left, double right) {
        return left + right;
    }
};
```

and instances of the class shouldn't be used as keys in collections such as maps.

An example of creating an accumulator that simply adds double values presented to it appears below:

```java
// function that will be supplied to an instance of DoubleAccumulator
DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
    @Override
    public double applyAsDouble(double left, double right) {
        return left + right;
    }
};

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

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

Java

main.java

Saved

```java
import java.util.concurrent.atomic.DoubleAccumulator;
import java.util.function.DoubleBinaryOperator;

class Demonstration {

    public static void main( String args[] ) {

        // function that will be supplied to an instance of DoubleAccumulator
        DoubleBinaryOperator doubleBinaryOperator = new DoubleBinaryOperator() {
            @Override
            public double applyAsDouble(double left, double right) {
                System.out.println(left + " " + right);
```

Run

As you can see we aren't confined to adding double values, rather we can perform as complex operations as desired in the supplied function, which makes the `DoubleAccumulator` class far more versatile than the `DoubleAdder` class which is limited to addition. In fact, `DoubleAdder` can be thought of as a specialized case of `DoubleAccumulator` which adds double values.

Distributing contention

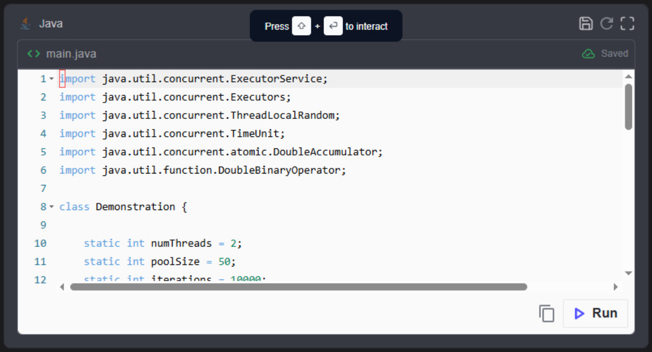
We can achieve the same functionality by using an instance of `AtomicLong` as we can with the `DoubleAccumulator`, however, the rationale for `DoubleAccumulator` is to distribute contention among threads by maintaining a set of variables that grow dynamically and each one is updated by only a subset 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 `doubleValue()` methods, all the underlying variables are accumulated by applying the supplied function and the result is returned. The expected throughput of `DoubleAccumulator` is significantly higher when used in place of `AtomicLong` for maintaining a double value under high contention. The improved performance comes at the cost of using more space. Additionally, we'll also have the overhead of converting double bytes to long bytes and back when using `AtomicLong` for maintaining the running double value. See the `AtomicLong` lesson where we discuss using `AtomicLong` for double values.

Order of accumulation

When multiple threads accumulate an instance of `DoubleAccumulator`, eventually all the double values in the underlying set are accumulated using the supplied function. The order in which these double 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 double values. The accumulation taking place in an arbitrary order may make this class unsuitable for scenarios where numerical stability is required, especially when combining values of very different orders of magnitude.

Example

In the example below, we use the `DoubleAccumulator` class to keep track of the maximum value observed. There are several threads that use the `ThreadLocalRandom` class to produce a random double value less than 1000, and then attempt to update the instance of `DoubleAccumulator`. Go through the listing which is self-explanatory.



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.DoubleAccumulator;
import java.util.function.DoubleBinaryOperator;

class Demonstration {

    static int numThreads = 2;
    static int poolSize = 50;
    static int iterations = 10000;
```

Back lesson
DoubleAdder

Mark As Completed

Next
AtomicReference