DoubleAdder

Guide to using DoubleAdder class.

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

Overview

The `DoubleAdder` class offers an alternative mechanism to using atomic classes or lock-based counters in situations where a variable is repeatedly updated to compute stats or summary counts. In such scenarios throughput suffers significantly when several threads attempt to update a single variable. In case of locks, threads get suspended and resumed while in case of atomic classes threads spin until successful. With `DoubleAdder` the idea is to have several variables that maintain the count in a thread-safe manner rather than a single variable, thus effectively reducing contention. Reducing contention improves throughput though at the cost of using more space.

The `DoubleAdder` class is very similar to `LongAdder`, in that both classes maintain a set of variables, of a package-private type `java.util.concurrent.atomic.Striped64.Cell` which is a variant of `AtomicLong`. As the number of threads manipulating an instance of `DoubleAdder` increases, this underlying table of `Cell`s keeps growing upto the maximum number of CPUs. The long to double conversions are handled internally. The array of counts helps to spread contention among threads rather than all threads competing to increment/decrement a single instance.

When the cumulative or final count is desired, the `sum()` method can be invoked. Floating point arithmetic isn't associative and when `sum()` is invoked it doesn't guarantee the order in which the accumulation happens. This appears as a disclaimer in the class's documentation stating *The order of accumulation within or across threads is not guaranteed. Thus, this class may not be applicable if numerical stability is required, especially when combining values of substantially different orders of magnitude.*. Additionally, `sum()` is not an atomic snapshot, and if concurrent updates by threads continue while `sum()` is executing, the newer updates may not be reflected in the result. However, when `sum()` is invoked in the absence of concurrent updates i.e. no thread is performing a write operation, then `sum()` returns an accurate result.

`DoubleAdder` should be used in scenarios where updates to an instance of `DoubleAdder` are frequent and reads are rare. For instance, calculating summary statistics where several threads update a common variable is a good candidate for use of `DoubleAdder`.

Example

In the example below, we have ten threads that attempt to increment an instance of `DoubleAdder` by one a million times each. Though we don't have an equivalent for double in the atomic classes but for purposes of our crude test, we use an `AtomicLong` that is also incremented by one and then time the two runs, one with `AtomicLong` and one with

frequent and reads are rare. For instance, calculating summary statistics where several threads update a common variable is a good candidate for use of `DoubleAdder`.

Example

In the example below, we have ten threads that attempt to increment an instance of `DoubleAdder` by one a million times each. Though we don't have an equivalent for double in the atomic classes but for purposes of our crude test, we use an `AtomicLong` that is also incremented by one and then time the two runs, one with `AtomicLong` and one with `DoubleAdder`. The test is setup so that the counter variables in both scenarios experience high contention and the results show that `DoubleAdder` outperforms `AtomicLong`.

Back lesson
LongAccumulator
Mark As Completed
Next
DoubleAccumulator