61% completed

Search Course

AtomicLongFieldUpdater
LongAdder
LongAccumulator
DoubleAdder
DoubleAccumulator
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
Practice Mock Interview →

LongAdder

Learn why LongAdder can be a better choice for computing statistics in a multithreaded environment than AtomicLong.

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

Overview

Consider a scenario where you are hosting an API service and need to maintain a running count of all the invocations of a particular API endpoint. You could use `AtomicLong` for the occasion as a counter but if you expect several concurrent requests hitting your endpoint, then multiple threads will attempt to increment the `AtomicLong` instance. In such a scenario where the intention is to compute statistics a better approach is to use the `LongAdder` class rather than the `AtomicLong`.

`AtomicLong` internally uses the compare and set instruction to perform the increment operation. In case of high contention, that is multiple threads attempting to increment an instance, a single thread wins the race to perform the increment operation while the rest have to retry until all of them succeed one by one. `AtomicLong` doesn't use locks and threads don't suspend accessing it. We discuss non-blocking synchronization to implement thread-safe algorithms and cons of locking in the Atomic Classes section.

`LongAdder` overcomes high contention by keeping an array of counts, which can be thought
rather than the `AtomicLong`.

`AtomicLong` internally uses the compare and set instruction to perform the increment operation. In case of high contention, that is multiple threads attempting to increment an instance, a single thread wins the race to perform the increment operation while the rest have to retry until all of them succeed one by one. `AtomicLong` doesn't use locks and threads don't suspend accessing it. We discuss non-blocking synchronization to implement thread-safe algorithms and cons of locking in the Atomic Classes section.

`LongAdder` overcomes high contention by keeping an array of counts, which can be thought of as variants of `AtomicLong`, and each one of them can be incremented atomically and independently of the other. The contention is spread out across the array resulting in increased throughput. Though the increased throughput comes at the cost of using more space. When the final value is requested, the sum of the array is calculated and returned by invoking the `sum()` method. Note that `sum()` isn't an atomic snapshot and any writes by threads while `sum()` is executing may not be reflected in the result. An accurate result is returned when concurrent updates don't occur.

The counts are of a non-public type `Cell` which is a variant of `AtomicLong`. The array is referred to as the *table of Cells* and it grows in size as contention increases but its maximum size is capped by the number of CPUs.

Example

The widget below runs a crude performance test between `LongAdder` and `AtomicLong`. We create ten threads that increment an instance of either class a million times. The sum at the end should come to be ten million. From the test set-up we can see that the instance of each class is highly contended for. From the test output we can see that `LongAdder` performs much better than `AtomicLong`.

Java — Press ⇧ + ↵ to interact

main.java — Saved

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

class Demonstration {

    static AtomicLong atomicLong = new AtomicLong(0);
    static LongAdder longAdder = new LongAdder();

    public static void main( String args[] ) throws Exception {
        withAtomicLong();
        withLongAddr();
```

Run

← Back lesson
AtomicLongFieldUpdater

Mark As Completed
Next →
LongAccumulator