**Java 5’s AtomicLong**

**Vs.**

**Java 8’s LongAdder and LongAccumulator**

<https://www.youtube.com/watch?v=Q_0_1mKTlnY>

AtomicLong holds volatile long variable, so that read and write of that variable happens in main memory instead of thread’s own memory. You can read JavaAtomic.java.

But it has one problem as explained below.

<http://psy-lob-saw.blogspot.com/2014/06/jdk8-update-on-scalable-counters.html>

public final long incrementAndGet() {

|  |
| --- |
| return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L; |
| } |

Which is same as

public final long incrementAndGet() {

|  |
| --- |
| for (;;) { |
| long current = get(); |
| long next = current + 1; |
| if (compareAndSet(current, next)) |
| return next; |
| } |
| } |

Many methods of AtomicLong goes in a loop as shown above. When you call incrementAndGet(), before setting a new incremented value, it will check whether the original value is not changed by some other thread. If it is changed then go in a loop until it is sure.

It uses CompareAndSet (CAS) algorithm.

**AtomicLong** has methods like incrementAndGet() or getAndIncrement(). Using these methods you are expecting the result of increment right after doing it. To maintain concurrency, it needs to iterate through compareAndSet (CAS) algorithm. Before setting a new incremented value, it will check whether the original value is not changed by some other thread. If it is changed then go in a loop until it is sure.

When there are lots of threads updating the value, it takes long time to retrieve the value for one thread. This is a big problem !!!

LongAdder addresses this problem.

**LongAdder** has two different methods increment() and sum().

It mantains cell[]. When different threads ask to increment the original value, it just keeps accumulating the puts values in cell array.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ov=1 | T1=1 | T2=2 | T3=5 | T1=5 | T2=3 | … |  |  |  |

When any thread asks for the sum(), it sums the values from cell[] and returns it and updates the original value (ov).

Increment() method calls longAccumulator() method that does this operation of maintaining cell[].

sum() looks like this. Read the comment carefully. It says that when sum is happening, meanwhile if some thread adds a value to cells[], it will try to catch up that value by checking i < as.length every time in for loop, but it may happen that i < as.length turns to false (might have reached to max) and then some other thread adds a value to cells[], then that value won’t be accounted for.

*/\*\*  
 \* Returns the current sum. The returned value is <em>NOT</em> an  
 \* atomic snapshot; invocation in the absence of concurrent  
 \* updates returns an accurate result, but concurrent updates that  
 \* occur while the sum is being calculated might not be  
 \* incorporated.  
 \*  
 \** ***@return*** *the sum  
 \*/*

**public long** sum() {  
 Cell[] as = **cells**; Cell a;  
 **long** sum = **base**;  
 **if** (as != **null**) {  
 **for** (**int** i = 0; i < as.**length**; ++i) {  
 **if** ((a = as[i]) != **null**)  
 sum += a.**value**;  
 }  
 }  
 **return** sum;  
}

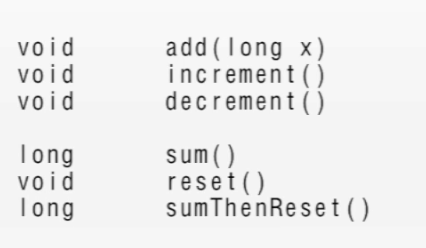
**Good to know**

Why ‘**for** (**int** i = 0; i < as.**length**; ++i)’ is better than ‘for(long asValue : as)’?

Earlier one always check for i < as.length. In this case, if as reference is changed to longer range array then it will check a new reference.

But later one uses the older as reference only till for loop finishes.

LongAdder has following methods



**LongAccumulator**

It is just like LongAdder. It is a generalization of LongAdder. Instead of just adding the values (sum), you can do whatever operation you want to do (like multiplication, subtraction etc).

new LongAccumulator((x, y) -> x \*y, 1L)