二

# 43 Java 8 中 Adder 和 Accumulator 有什么区别?

本课时主要介绍在 Java 8 中 Adder 和 Accumulator 有什么区别。

## Adder 的介绍

我们要知道 Adder 和 Accumulator 都是 Java 8 引入的，是相对比较新的类。对于 Adder 而言，比如最典型的 LongAdder，我们在第 40 讲的时候已经讲解过了，**在高并发下 LongAdder 比 AtomicLong 效率更高**，因为对于 AtomicLong 而言，它只适合用于低并发场景，否则在高并发的场景下，由于 CAS 的冲突概率大，会导致经常自旋，影响整体效率。

而 LongAdder 引入了分段锁的概念，当竞争不激烈的时候，所有线程都是通过 CAS 对同一个 Base 变量进行修改，但是当竞争激烈的时候，LongAdder 会把不同线程对应到不同的 Cell 上进行修改，降低了冲突的概率，从而提高了并发性。

## Accumulator 的介绍

那么 Accumulator 又是做什么的呢？Accumulator 和 Adder 非常相似，**实际上 Accumulator 就是一个更通用版本的 Adder**，比如 LongAccumulator 是 LongAdder 的功能增强版，因为 LongAdder 的 API 只有对数值的加减，而 LongAccumulator 提供了自定义的函数操作。

我这样讲解可能有些同学还是不太理解，那就让我们用一个非常直观的代码来举例说明一下，代码如下：

```
public class LongAccumulatorDemo {

    public static void main(String[] args) throws InterruptedException {

        LongAccumulator accumulator = new LongAccumulator((x, y) -> x + y, 0);

        ExecutorService executor = Executors.newFixedThreadPool(8);

        IntStream.range(1, 10).forEach(i -> executor.submit(() -> accumulator.accum
```

```
            Thread.sleep(2000);
            System.out.println(accumulator.getThenReset());
        }
    }
```

在这段代码中：

- 首先新建了一个 LongAccumulator，同时给它传入了两个参数；
- 然后又新建了一个 8 线程的线程池，并且利用整形流也就是 IntStream 往线程池中提交了从 1 ~ 9 这 9 个任务；
- 之后等待了两秒钟，这两秒钟的作用是等待线程池的任务执行完毕；
- 最后把 accumulator 的值打印出来。

这段代码的运行结果是 45，代表 0+1+2+3+...+8+9=45 的结果，这个结果怎么理解呢？我们先重点看看新建的 LongAccumulator 的这一行语句：

```
LongAccumulator accumulator = new LongAccumulator((x, y) -> x + y, 0);
```

在这个语句中，我们传入了两个参数：LongAccumulator 的构造函数的第一个参数是二元表达式；第二个参数是 x 的初始值，传入的是 0。在二元表达式中，x 是上一次计算的结果（除了第一次的时候需要传入），y 是本次新传入的值。

## 案例分析

我们来看一下上面这段代码执行的过程，当执行 accumulator.accumulate(1) 的时候，首先要知道这时候 x 和 y 是什么，第一次执行时， x 是 LongAccumulator 构造函数中的第二个参数，也就是 0，而第一次执行时的 y 值就是本次 accumulator.accumulate(1) 方法所传入的 1；然后根据表达式 x+y，计算出 0+1=1，这个结果会赋值给下一次计算的 x，而下一次计算的 y 值就是 accumulator.accumulate(2) 传入的 2，所以下一次的计算结果是 1+2=3。

我们在 IntStream.range(1, 10).forEach(i -> executor.submit(() -> accumulator.accumulate(i))); 这一行语句中实际上利用了整型流，分别给线程池提交了从 1 ~ 9 这 9 个任务，相当于执行了：

```
accumulator.accumulate(1);
accumulator.accumulate(2);
```

```
accumulator.accumulate(3);

...

accumulator.accumulate(8);

accumulator.accumulate(9);
```

那么根据上面的这个推演，就可以得出它的内部运行，这也就意味着，LongAccumulator 执行了：

```
0+1=1;

1+2=3;

3+3=6;

6+4=10;

10+5=15;

15+6=21;

21+7=28;

28+8=36;

36+9=45;
```

这里需要指出的是，这里的加的顺序是不固定的，并不是说会按照顺序从 1 开始逐步往上累加，它也有可能会变，比如说先加 5、再加 3、再加 6。但总之，由于加法有交换律，所以最终加出来的结果会保证是 45。这就是这个类的一个基本的作用和用法。

## 拓展功能

我们继续看一下它的功能强大之处。举几个例子，刚才我们给出的表达式是 x + y，其实同样也可以传入 x * y，或者写一个 Math.min(x, y)，相当于求 x 和 y 的最小值。同理，也可以去求 Math.max(x, y)，相当于求一个最大值。根据业务的需求来选择就可以了。代码如下：

```
LongAccumulator counter = new LongAccumulator((x, y) -> x + y, 0);

LongAccumulator result = new LongAccumulator((x, y) -> x * y, 0);

LongAccumulator min = new LongAccumulator((x, y) -> Math.min(x, y), 0);

LongAccumulator max = new LongAccumulator((x, y) -> Math.max(x, y), 0);
```

这时你可能会有一个疑问：在这里为什么不用 for 循环呢？比如说我们之前的例子，从 0 加到 9，我们直接写一个 for 循环不就可以了吗？

确实，用 for 循环也能满足需求，但是用 for 循环的话，它执行的时候是串行，它一定是按照 0+1+2+3+...+8+9 这样的顺序相加的，但是 LongAccumulator 的一大优势就是可以利用线程池来为它工作。**一旦使用了线程池，那么多个线程之间是可以并行计算的，效率要比之前的串行高得多。**这也是为什么刚才说它加的顺序是不固定的，因为我们并不能保证各个线程之间的执行顺序，所能保证的就是最终的结果是确定的。

## 适用场景

接下来我们说一下 LongAccumulator 的适用场景。

第一点需要满足的条件，就是需要大量的计算，并且当需要并行计算的时候，我们可以考虑使用 LongAccumulator。

当计算量不大，或者串行计算就可以满足需求的时候，可以使用 for 循环；如果计算量大，需要提高计算的效率时，我们则可以利用线程池，再加上 LongAccumulator 来配合的话，就可以达到并行计算的效果，效率非常高。

第二点需要满足的要求，就是计算的执行顺序并不关键，也就是说它不要求各个计算之间的执行顺序，也就是说线程 1 可能在线程 5 之后执行，也可能在线程 5 之前执行，但是执行的先后并不影响最终的结果。

一些非常典型的满足这个条件的计算，就是类似于加法或者乘法，因为它们是有交换律的。同样，求最大值和最小值对于顺序也是没有要求的，因为最终只会得出所有数字中的最大值或者最小值，无论先提交哪个或后提交哪个，都不会影响到最终的结果。

上一页 下一页