二

40 AtomicInteger 在高并发下性能不好，如何解决？为什么？

本课时我们主要讲解 AtomicInteger 在高并发下性能不好，如何解决？以及为什么会出现这种情况？

我们知道在 JDK1.5 中新增了并发情况下使用的 Integer/Long 所对应的原子类 AtomicInteger 和 AtomicLong。

在并发的场景下，如果我们需要实现计数器，可以利用 AtomicInteger 和 AtomicLong，这样一来，就可以避免加锁和复杂的代码逻辑，有了它们之后，我们只需要执行对应的封装好的方法，例如对这两个变量进行原子的增操作或原子的减操作，就可以满足大部分业务场景的需求。

不过，虽然它们很好用，但是如果你的业务场景是并发量很大的，那么你也会发现，这两个原子类实际上会有较大的性能问题，这是为什么呢？就让我们从一个例子看起。

AtomicLong 存在的问题

首先我们来看一段代码：

```
/**
* 描述：     在16个线程下使用AtomicLong
*/
public class AtomicLongDemo {
   public static void main(String[] args) throws InterruptedException {
       AtomicLong counter = new AtomicLong(0);
       ExecutorService service = Executors.newFixedThreadPool(16);
       for (int i = 0; i < 100; i++) {
           service.submit(new Task(counter));
```

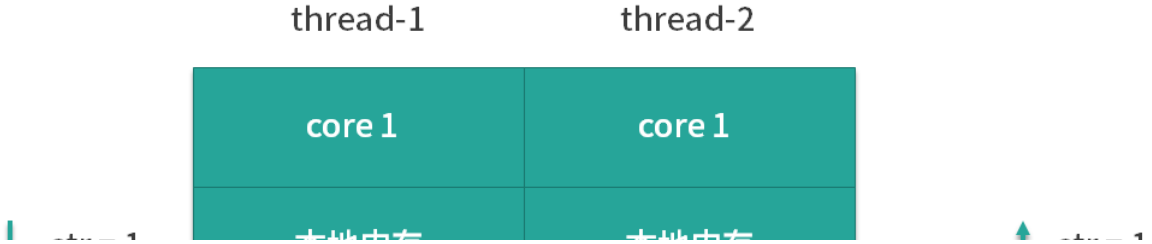
```
        }
        Thread.sleep(2000);
        System.out.println(counter.get());
    }
    static class Task implements Runnable {
        private final AtomicLong counter;
        public Task(AtomicLong counter) {
            this.counter = counter;
        }
        @Override
        public void run() {
            counter.incrementAndGet();
        }
    }
}
```

在这段代码中可以看出，我们新建了一个原始值为 0 的 AtomicLong。然后，有一个线程数为 16 的线程池，并且往这个线程池中添加了 100 次相同的一个任务。

那我们往下看这个任务是什么。在下面的 Task 类中可以看到，这个任务实际上就是每一次去调用 AtomicLong 的 incrementAndGet 方法，相当于一次自加操作。这样一来，整个类的作用就是把这个原子类从 0 开始，添加 100 个任务，每个任务自加一次。

这段代码的运行结果毫无疑问是 100，虽然是多线程并发访问，但是 AtomicLong 依然可以保证 incrementAndGet 操作的原子性，所以不会发生线程安全问题。

不过如果我们深入一步去看内部情景的话，你可能会感到意外。我们把模型简化成只有两个线程在同时工作的并发场景，因为两个线程和更多个线程本质上是一样的。如图所示：



我们可以看到在这个图中，每一个线程是运行在自己的 core 中的，并且它们都有一个本地内存是自己独用的。在本地内存下方，有两个 CPU 核心共用的共享内存。

对于 AtomicLong 内部的 value 属性而言，也就是保存当前 AtomicLong 数值的属性，它是被 volatile 修饰的，所以它需要保证自身可见性。

这样一来，每一次它的数值有变化的时候，它都需要进行 flush 和 refresh。比如说，如果开始时，ctr 的数值为 0 的话，那么如图所示，一旦 core 1 把它改成 1 的话，它首先会在左侧把这个 1 的最新结果给 flush 到下方的共享内存。然后，再到右侧去往上 refresh 到核心 2 的本地内存。这样一来，对于核心 2 而言，它才能感知到这次变化。

由于竞争很激烈，这样的 flush 和 refresh 操作耗费了很多资源，而且 CAS 也会经常失败。

LongAdder 带来的改进和原理

在 JDK 8 中又新增了 LongAdder 这个类，这是一个针对 Long 类型的操作工具类。那么既然已经有了 AtomicLong，为何又要新增 LongAdder 这么一个类呢?

我们同样是用一个例子来说明。下面这个例子和刚才的例子很相似，只不过我们把工具类从 AtomicLong 变成了 LongAdder。其他的不同之处还在于最终打印结果的时候，调用的方法从原来的 get 变成了现在的 sum 方法。而其他的逻辑都一样。

我们来看一下使用 LongAdder 的代码示例：

```
/**
* 描述：     在16个线程下使用LongAdder
*/
public class LongAdderDemo {

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

       LongAdder counter = new LongAdder();

       ExecutorService service = Executors.newFixedThreadPool(16);

       for (int i = 0; i < 100; i++) {
```

```
            service.submit(new Task(counter));
        }
        Thread.sleep(2000);
        System.out.println(counter.sum());
    }
    static class Task implements Runnable {
        private final LongAdder counter;
        public Task(LongAdder counter) {
            this.counter = counter;
        }
        @Override
        public void run() {
            counter.increment();
        }
    }
}
```

代码的运行结果同样是 100，但是运行速度比刚才 AtomicLong 的实现要快。下面我们解释一下，为什么高并发下 LongAdder 比 AtomicLong 效率更高。

因为 LongAdder 引入了分段累加的概念，内部一共有两个参数参与计数：第一个叫作 base，它是一个变量，第二个是 Cell[]，是一个数组。

其中的 base 是用在竞争不激烈的情况下的，可以直接把累加结果改到 base 变量上。

那么，当竞争激烈的时候，就要用到我们的 Cell[] 数组了。一旦竞争激烈，各个线程会分散累加到自己所对应的那个 Cell[] 数组的某一个对象中，而不会大家共用同一个。

这样一来，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，这是一种分段的理念，提高了并发性，这就和 Java 7 的 ConcurrentHashMap 的 16 个 Segment 的思想类似。

竞争激烈的时候，LongAdder 会通过计算出每个线程的 hash 值来给线程分配到不同的 Cell 上去，每个 Cell 相当于是一个独立的计数器，这样一来就不会和其他的计数器干扰，Cell

之间并不存在竞争关系，所以在自加的过程中，就大大减少了刚才的 flush 和 refresh，以及降低了冲突的概率，这就是为什么 LongAdder 的吞吐量比 AtomicLong 大的原因，本质是空间换时间，因为它有多个计数器同时在工作，所以占用的内存也要相对更大一些。

那么 LongAdder 最终是如何实现多线程计数的呢？答案就在最后一步的求和 sum 方法，执行 LongAdder.sum() 的时候，会把各个线程里的 Cell 累计求和，并加上 base，形成最终的总和。代码如下：

```
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;
}
```

在这个 sum 方法中可以看到，思路非常清晰。先取 base 的值，然后遍历所有 Cell，把每个 Cell 的值都加上去，形成最终的总和。由于在统计的时候并没有进行加锁操作，所以这里得出的 sum 不一定是完全准确的，因为有可能在计算 sum 的过程中 Cell 的值被修改了。

那么我们已经了解了，为什么 AtomicLong 或者说 AtomicInteger 它在高并发下性能不好，也同时看到了性能更好的 LongAdder。下面我们就分析一下，对它们应该如何选择。

如何选择

在低竞争的情况下，AtomicLong 和 LongAdder 这两个类具有相似的特征，吞吐量也是相似的，因为竞争不高。但是在竞争激烈的情况下，LongAdder 的预期吞吐量要高得多，经过试验，LongAdder 的吞吐量大约是 AtomicLong 的十倍，不过凡事总要付出代价，LongAdder 在保证高效的同时，也需要消耗更多的空间。

AtomicLong 可否被 LongAdder 替代?

那么我们就要考虑了，有了更高效的 LongAdder，那 AtomicLong 可否不使用了呢？是否凡是用到 AtomicLong 的地方，都可以用 LongAdder 替换掉呢？答案是不是的，这需要区分场景。

LongAdder 只提供了 add、increment 等简单的方法，适合的是统计求和计数的场景，场景比较单一，而 AtomicLong 还具有 compareAndSet 等高级方法，可以应对除了加减之外的更复杂的需要 CAS 的场景。

结论：如果我们的场景仅仅是需要用到加和减操作的话，那么可以直接使用更高效的 LongAdder，但如果我们需要利用 CAS 比如 compareAndSet 等操作的话，就需要使用 AtomicLong 来完成。

上一页 下一页