一，LongAdder和AtomicInteger效率对比

```
/**
 * @author yhd
 * @createtime 2020/10/3 22:35
 * 比较LongAdder和AtomicInteger的效率
 */
public class LongAdderAndAtomicTest {

    public static void main(String[] args) throws Exception {
        test(1, 10000000);

        test(10, 10000000);

        test(20, 10000000);

        test(50, 10000000);
    }

    /**
     * 测试LongAdder和Atomic的效率
     *
     * @param threadNum 线程数
     * @param times     执行时间
     */
    public static void test(Integer threadNum, Integer times) throws Exception {
        System.out.println("线程数为：" + threadNum);
        testAtomic(threadNum, times);
        testLongAdder(threadNum, times);
    }

    public static AtomicInteger a = new AtomicInteger(0);
    public static LongAdder b = new LongAdder();

    /**
     * 测试Atomic的效率
     *
     * @param threadNum
     * @param times
     */
    public static void testAtomic(Integer threadNum, Integer times) throws
InterruptedException {
        //开始时间
        long start = System.currentTimeMillis();

        CountDownLatch countDownLatch = new CountDownLatch(threadNum);
        for (int i = 0; i < threadNum; i++) {

            new Thread(() -> {
                for (int j = 0; j < times; j++) {
                    a.incrementAndGet();
                }
                countDownLatch.countDown();
```

```
            }).start();
        }
        countDownLatch.await();
        //结束时间
        long end = System.currentTimeMillis();

        System.out.println("Atomic 消耗时间: " + (end - start));
    }

    /**
     * 测试LongAdder的效率
     *
     * @param threadNum
     * @param times
     */
    public static void testLongAdder(Integer threadNum, Integer times) throws
InterruptedException {
        //开始时间
        long start = System.currentTimeMillis();

        CountDownLatch countDownLatch = new CountDownLatch(threadNum);
        for (int i = 0; i < threadNum; i++) {

            new Thread(() -> {
                for (int j = 0; j < times; j++) {
                    b.increment();
                }
                countDownLatch.countDown();
            }).start();
        }
        countDownLatch.await();
        //结束时间
        long end = System.currentTimeMillis();

        System.out.println("LongAdder 消耗时间: " + (end - start));
    }
}
```

由图可以看到，线程数越多，LongAdder的效率型对于Atomic越高，由此可以看出，LongAdder更适合于高并发情况下。

二，LongAdder源码解读

1.继承关系

```
public class LongAdder extends Striped64 implements Serializable {
```

LongAdder类继承了Striped64类，Striped64里面声明了一个内部类Cell：

```
    @sun.misc.Contended static final class Cell {
        volatile long value;
        Cell(long x) { value = x; }
        final boolean cas(long cmp, long val) {
            return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val);
        }

        // Unsafe mechanics
        private static final sun.misc.Unsafe UNSAFE;
        private static final long valueOffset;
        static {
            try {
                UNSAFE = sun.misc.Unsafe.getUnsafe();
                Class<?> ak = Cell.class;
                valueOffset = UNSAFE.objectFieldOffset
                    (ak.getDeclaredField("value"));
            } catch (Exception e) {
```

```
                throw new Error(e);
            }
        }
    }

transient volatile Cell[] cells;

transient volatile long base;
```

回到LongAdder，LongAdder里面使用的这两个属性实际上是继承自他的父类的。

2.LongAdder的add()

```
    public void add(long x) {
        /**
         * as: 表示cells数组的引用
         * b:  表示获取的base值
         * v: 表示期望值
         * m: 表示cells数组的长度
         * a: 表示当前线程命中的cell单元格
         */
        Cell[] as; long b, v; int m; Cell a;
        /**
         * 条件一：true->表示cells已经初始化过，当前线程应该将数据写入到对应的cell中
         *           false->表示cells为初始化，当前所有线程应该将数据写入到base中
         * 条件二：false->表示当前线程cas替换数据成功
         *           true->表示发生竞争了，可能需要重试或者扩容
         * 进入if的条件：数组已经初始化 或者  cas交换数据失败，表示有竞争
         *
         */
        if ((as = cells) != null || !casBase(b = base, b + x)) {
            /**
             * uncontended: true -> 未竞争  false->发生竞争
             * 条件一：true->数组没有初始化
             *           false->数组已经初始化
             * 条件二：true->数组没有初始化
             *           false->数组已经初始化
             *
             *     getProbe()：获取当前线程的hash值
             * 条件三：true-> 当前线程对应的cell并没有初始化
             *           false->当前线程对应的cell已经初始化
             * 条件四：true->cas交换失败，表示有竞争
             *           false->cas交换成功
             * 进入if的条件：
             * cells未初始化， 或者 当前线程对应的cell未初始化， 或者 cas交换失败
             */
            boolean uncontended = true;
            if (as == null || (m = as.length - 1) < 0 ||
                (a = as[getProbe() & m]) == null ||
                !(uncontended = a.cas(v = a.value, v + x)))
                longAccumulate(x, null, uncontended);
        }
    }
```

此时将要执行的是Striped64类里面的longAccumulate（）方法。

3.Striped64类

1.内部类Cell

```
@sun.misc.Contended static final class Cell {
    volatile long value;
    Cell(long x) { value = x; }
    final boolean cas(long cmp, long val) {
        return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val);
    }

    // Unsafe mechanics
    private static final sun.misc.Unsafe UNSAFE;
    private static final long valueOffset;
    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> ak = Cell.class;
            valueOffset = UNSAFE.objectFieldOffset
                (ak.getDeclaredField("value"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

这个内部类只有一个value属性，用来存放base中获取到的值。他的cas方法也是执行的UNSAFE类的compareAndSwapLong()方法。

2.变量

```
//获取当前系统的cpu数  控制cells数组长度的一个关键条件
static final int NCPU = Runtime.getRuntime().availableProcessors();

//数组的长度，只要数组不为空，一定是2的倍数，这样-1转化为二进制的时候一定是一大堆1
transient volatile Cell[] cells;

//没有发生过竞争时，数据会累加到 base上 | 当cells扩容时，需要将数据写到base中
transient volatile long base;

//初始化cells或者扩容cells都需要获取锁，0 表示无锁状态，1 表示其他线程已经持有锁了
transient volatile int cellsBusy;
```

3.方法

casBase()

cas的方式改变base的值。

```
final boolean casBase(long cmp, long val) {
    return UNSAFE.compareAndSwapLong(this, BASE, cmp, val);
}
```

casCellsBusy()

cas的方式获取锁

```
final boolean casCellsBusy() {
        return UNSAFE.compareAndSwapInt(this, CELLSBUSY, 0, 1);
    }
```

getProbe()

获取当前线程的hash值

```
static final int getProbe() {
        return UNSAFE.getInt(Thread.currentThread(), PROBE);
    }
```

advanceProbe()

重置当前线程的hash值

```
    static final int advanceProbe(int probe) {
        probe ^= probe << 13;   // xorshift
        probe ^= probe >>> 17;
        probe ^= probe << 5;
        UNSAFE.putInt(Thread.currentThread(), PROBE, probe);
        return probe;
    }
```

longAccumulate()

这个类的核心无非就是这个方法，接下来进行分析：

```
    /**
     * 首先：哪些情况会进入当前方法？
     * cells未初始化， 或者 当前线程对应的cell未初始化， 或者 cas交换失败
     * @param x 新值
     * @param fn 没用上。一个扩展接口
     * @param wasUncontended  只有cells初始化之后，并且当前线程 竞争修改失败，才会是false
     */
    final void longAccumulate(long x, LongBinaryOperator fn,
                              boolean wasUncontended) {
        //h：代表当前线程hash值
        int h;
        /**
         * 如果当前线程hash值等于0，条件成立
         * 给当前线程分配hash值
         * 将当前线程的hash值重新赋值给h
         * 设置为未竞争或者竞争修改成功状态。
         *            为什么？
         * 因为默认所有线程进来操做的都是cells[0]的位置，所以不把它当作一次真正的竞争。
         */
        if ((h = getProbe()) == 0) {
            //给当前线程分配hash值
            ThreadLocalRandom.current(); // force initialization
            //将当前线程的hash值重新赋值给h
```

```
            h = getProbe();
            wasUncontended = true;
        }
        //表示扩容意向 false 一定不会扩容，true 可能会扩容。
        boolean collide = false;
        //自旋
        for (;;) {
            /**
             * as 代表cells的引用
             * a 当前线程对应的cell
             * n cells的长度
             * v 期望值
             */
            Cell[] as; Cell a; int n; long v;
            /**
             * case1:
             *       cells已经初始化
             *
             *        case1.1:
             *       if(当前线程对应的cell还没有初始化 && 当前处于无锁状态){
             *           创建一个新的cell对象 r
             *
             *           if（当前锁状态未0并且获取到了锁）{
             *               created : 标记是否创建成功
             *               if（cells已经被初始化 && 当前线程对应的cell为空）{
             *                   将当前线程对应位置的cell初始化为新创建的cell r
             *                   create=true 表示创建成功，最终在释放锁。
             *               }
             *           }
             *           将扩容意向改成false
             *       }
             *
             *       case1.2:
             *       if(如果当前线程竞争修改失败){
             *           状态改为true;
             *           //默认所有线程一开始都在cell[0]的位置，所以一定会发生竞争，
             *           //这次竞争就不当作一次真正的竞争。
             *       }
             *
             *       case1.3:
             *       if(当前线程rehash过hash值 && 新命中的cell不为空){
             *           尝试cas一次
             *       }
             *
             *       case1.4:
             *       if(如果cells的长度>cpu数 || cells和as不一致){
             *           //cells和as不一致 说明其他线程已经扩容过了，当前线程只需要rehash
重试即可
             *           扩容意向强制改为false。
             *       }
             *
             *       case1.5:
             *       //!collide = true 设置扩容意向 为true 但是不一定真的发生扩容
             *
             *       case1.6:
             *       if(锁状态为0 && 获取到了锁){
             *       //第二层判断为了防止当前线程在对第一层id的条件判断一半的时候，又进来一个
线程，将所有业务已经执行一遍了。
```

```
                *       //只有当cells==as才能说明，当前线程在第一层if执行条件的过程中，没有其他线程进来破坏。
                *           if(cells==as){
                *               扩容为原来的二倍
                *               重置当前线程Hash值
                *           }
                *       }
                */
               if ((as = cells) != null && (n = as.length) > 0) {
                   if ((a = as[(n - 1) & h]) == null) {
                       if (cellsBusy == 0) {       // Try to attach new Cell
                           Cell r = new Cell(x);   // Optimistically create
                           if (cellsBusy == 0 && casCellsBusy()) {
                               //标记是否创建成功
                               boolean created = false;
                               try {
                                   /**
                                    * rs：cells的引用
                                    * m：cells的长度
                                    * j：当前线程对应的cells下标
                                    */
                                   Cell[] rs; int m, j;
                                   if ((rs = cells) != null &&
                                       (m = rs.length) > 0 &&
                                       rs[j = (m - 1) & h] == null) {
                                       rs[j] = r;
                                       created = true;
                                   }
                               } finally {
                                   cellsBusy = 0;
                               }
                               if (created)
                                   break;
                               continue;           // Slot is now non-empty
                           }
                       }
                       collide = false;
                   }
                   else if (!wasUncontended)       // CAS already known to fail
                       wasUncontended = true;      // Continue after rehash
                   else if (a.cas(v = a.value, ((fn == null) ? v + x :
                                                fn.applyAsLong(v, x))))
                       break;
                   else if (n >= NCPU || cells != as)
                       collide = false;            // At max size or stale
                   else if (!collide)
                       collide = true;
                   else if (cellsBusy == 0 && casCellsBusy()) {
                       try {
                           if (cells == as) {      // Expand table unless stale
                               Cell[] rs = new Cell[n << 1];
                               for (int i = 0; i < n; ++i)
                                   rs[i] = as[i];
                               cells = rs;
                           }
                       } finally {
                           cellsBusy = 0;
                       }
```

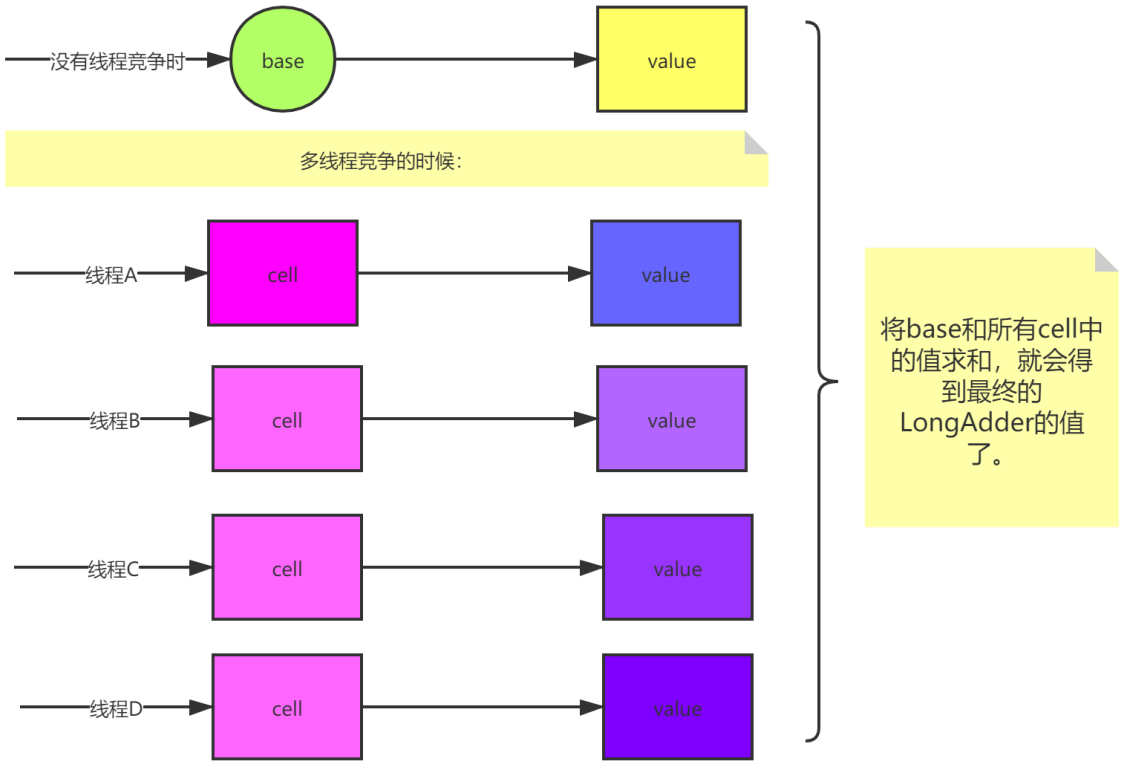
```
                    collide = false;
                    continue;                       // Retry with expanded table
                }
                //重置当前线程Hash值
                h = advanceProbe(h);
            }

            /**
             * case2:
             *      cells并未初始化
             *      锁状态为0
             *      cells==as   ?  因为其它线程可能会在你给as赋值之后修改了 cells
             *      获取锁成功
             * 里面再次判断cells==as是因为防止其他线程在判断第一层if的中间被其他线程先进来修改了一次
             *          初始化cells
             */
            else if (cellsBusy == 0 && cells == as && casCellsBusy()) {
                boolean init = false;
                try {                               // Initialize table
                    if (cells == as) {
                        Cell[] rs = new Cell[2];
                        rs[h & 1] = new Cell(x);
                        cells = rs;
                        init = true;
                    }
                } finally {
                    cellsBusy = 0;
                }
                if (init)
                    break;
            }

            /**
             * case3:
             *      cellsBusy处于加锁状态，表示其他线程正在初始化cells，
             *      那么当前线程就应该将数据累加到base
             */
            else if (casBase(v = base, ((fn == null) ? v + x :
                                        fn.applyAsLong(v, x))))
                break;                              // Fall back on using base
        }
    }
```

三，LongAdder执行流程

LongAdder的执行流程实际上就是
当没有线程竞争的时候，线程会直接操做base里面的值。
当有线程竞争的时候，会将base的值拷贝成一个cells数组，每个线程都来操作一个cell数组中的桶位，最终将cells各个桶位和base求和，就可以得到LongAdder的最终值。



为什么比cas的效率高?

cas是多线程竞争，只有拿到锁的线程才能去操做资源，其他线程不断的自旋重试，相当于线程排队。LongAdder是多线程竞争的时候，他会将共享资源拷贝多份，采用分支合并的思想，提升效率。