AtomicLong

类的基本字段

```
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();

private static native boolean VMSupportsCS8();

static {
    try {
        // 使用 unsafe 获取AtomicLong 中变量 value 在 AtomicLong 对象中的内存偏移
        valueOffset = unsafe.objectFieldOffset(AtomicLong.class.getDeclaredField("value"));
     } catch (Exception ex) { throw new Error(ex); }
}

private volatile long value;
```

一些计算操作

```
// 通过unsafe 实现硬件级别的自增
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}

// 通过unsafe 实现硬件级别的自减
public final long getAndDecrement() {
    return unsafe.getAndAddLong(this, valueOffset, -1L);
}
```

这里版本是：jdk1.8 ， 是采用的Unsafe 实现的。 而在JDK1.7中是CAS操作通过while循环实现的

LongAdder

与AtomicLong的使用场景小区别

首先说明：**如果要产生自增长序列，那么AtomicLong是非常好的方法，但是如果是一个计数器不需要每次增加都返回当前值选择**

也就是说，LongAdder可以支持原子级别的加减操作，但是不支持执行完毕之后马上获取操作后的结果值，因为没有提供getAndIncrement方法

下面来说明 LongAdder是如何实现原子操作的

先看获取值

```
public long longValue() {return 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;
}
```

将cells数组的值进行求和，那么再看数组中的Cell类。

```
@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);
         }
     }
}
```

类似于AtomicLong，利用CAS来更新变量， @Contended避免value伪共享。

小结：AtomicLong 是 单个值进行cas计算， 而LongAdder是通过一个类似AtomicLong的类Cell的 数组 Cell[] 来将值分散存储，需要的时候

再看自增increment

```
public void increment() {
    add(1L);
```

```
}

public void add(long x) {
    Cell[] as; long b, v; int m; Cell a;

    //想要add一个元素的时候，先看一下cells数组是否为空，如果是空的就尝试去看能不能直接加到base上面，如果线程竞争很小
    //如果cells是空的，并且竞争很大，cas失败，就进入if块内，创建cells
     //如果不是空的就进入到cell数组中看能加到哪个上面去
     if ((as = cells) != null || !casBase(b = base, b + x)) {
         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);
     }
}

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

性能对比

```
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.LongAdder;

public class TestLongAdder {

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

    public static void main(String[] args) {
            // testAtomicLongSpeed();
            testLongAdderSpeed();
    }

    public static void testAtomicLongSpeed() {
            long start = System.currentTimeMillis();
            for(int i = 0 ; i < 5 ;i++) {
                    new Thread(new Runnable() {

                            @Override
                            public void run() {
                                    while(true) {
                                            atomicLong.addAndGet(1);
                                    }
                            }
                    }).start();
            }
            while(true) {
                    if ((System.currentTimeMillis() - start) / 1000 >= 10) {
                            System.out.println(atomicLong.get());
```

```
                                System.exit(0);
                        }
                }
        }

        public static void testLongAdderSpeed() {
                long start = System.currentTimeMillis();
                for(int i = 0 ; i < 5 ;i++) {
                        new Thread(new Runnable() {

                                @Override
                                public void run() {
                                        while(true) {
                                                longAdder.increment();
                                        }
                                }
                        }).start();
                }
                while(true) {
                        if ((System.currentTimeMillis() - start) / 1000 >= 10) {
                                System.out.println(longAdder.longValue());
                                System.exit(0);
                        }
                }
        }
}
```

LongAdder 比 atomicLong性能快十倍，但是要注意使用场景上与 atomicLong的小区别