Atomic原子操作类介绍

在并发编程中很容易出现并发安全的问题，有一个很简单的例子就是多线程更新变量i=1,比如多个线程执行i++操作，就有可能获取不到正确的值，而这个问题，最常用的方法是通过Synchronized进行控制来达到线程安全的目的。但是由于synchronized是采用的是悲观锁策略，并不是特别高效的一种解决方案。实际上，在J.U.C下的atomic包提供了一系列的操作简单，性能高效，并能保证线程安全的类去更新基本类型变量，数组元素，引用类型以及更新对象中的字段类型。atomic包下的这些类都是采用的是乐观锁策略去原子更新数据，在java中则是使用CAS操作具体实现。

在java.util.concurrent.atomic包里提供了一组原子操作类：

基本类型：AtomicInteger、AtomicLong、AtomicBoolean；

引用类型：AtomicReference、AtomicStampedRerence、AtomicMarkableReference；

数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

对象属性原子修改器：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

原子类型累加器（jdk1.8增加的类）：DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64

原子更新基本类型

以AtomicInteger为例总结常用的方法

```
//以原子的方式将实例中的原值加1，返回的是自增前的旧值；
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}

//getAndSet(int newValue)：将实例中的值更新为新值，并返回旧值；
public final boolean getAndSet(boolean newValue) {
    boolean prev;
    do {
        prev = get();
    } while (!compareAndSet(prev, newValue));
    return prev;
}

//incrementAndGet() ：以原子的方式将实例中的原值进行加1操作，并返回最终相加后的结果；
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

```
19
20 //addAndGet(int delta) : 以原子方式将输入的数值与实例中原本的值相加，并返回最后的结果；
21 public final int addAndGet(int delta) {
22     return unsafe.getAndAddInt(this, valueOffset, delta) + delta;
```

测试

```
1  public class AtomicIntegerTest {
2      static AtomicInteger sum = new AtomicInteger(0);
3
4      public static void main(String[] args) {
5
6          for (int i = 0; i < 10; i++) {
7              Thread thread = new Thread(() -> {
8                  for (int j = 0; j < 10000; j++) {
9                      // 原子自增  CAS
10                     sum.incrementAndGet();
11                     //TODO
12                 }
13             });
14             thread.start();
15         }
16
17         try {
18             Thread.sleep(3000);
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22         System.out.println(sum.get());
23
24     }
25
```

```
public final int incrementAndGet() {
    return unsafe.getAndAddInt( o: this, valueOffset,  i: 1) + 1;
}
```

incrementAndGet()方法通过CAS自增实现，如果CAS失败，自旋直到成功+1。

```java
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}
```

cas失败，自旋

思考：这种CAS失败自旋的操作存在什么问题?

原子更新数组类型

AtomicIntegerArray为例总结常用的方法

```java
//addAndGet(int i, int delta)：以原子更新的方式将数组中索引为i的元素与输入值相加；
public final int addAndGet(int i, int delta) {
    return getAndAdd(i, delta) + delta;
}

//getAndIncrement(int i)：以原子更新的方式将数组中索引为i的元素自增加1；
public final int getAndIncrement(int i) {
    return getAndAdd(i, 1);
}

//compareAndSet(int i, int expect, int update)：将数组中索引为i的位置的元素进行更新
public final boolean compareAndSet(int i, int expect, int update) {
    return compareAndSetRaw(checkedByteOffset(i), expect, update);
```

测试

```java
public class AtomicIntegerArrayTest {

    static int[] value = new int[]{ 1, 2, 3, 4, 5 };
    static AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(value);


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

        //设置索引0的元素为100
        atomicIntegerArray.set(0, 100);
        System.out.println(atomicIntegerArray.get(0));
        //以原子更新的方式将数组中索引为1的元素与输入值相加
```

```
        atomicIntegerArray.getAndAdd(1,5);

        System.out.println(atomicIntegerArray);
    }
```

原子更新引用类型

AtomicReference作用是对普通对象的封装，它可以保证你在修改对象引用时的线程安全性。

```
public class AtomicReferenceTest {

    public static void main( String[] args ) {
        User user1 = new User("张三", 23);
        User user2 = new User("李四", 25);
        User user3 = new User("王五", 20);

        //初始化为 user1
        AtomicReference<User> atomicReference = new AtomicReference<>();
        atomicReference.set(user1);

        //把 user2 赋给 atomicReference
        atomicReference.compareAndSet(user1, user2);
        System.out.println(atomicReference.get());

        //把 user3 赋给 atomicReference
        atomicReference.compareAndSet(user1, user3);
        System.out.println(atomicReference.get());

    }

}


@Data
@AllArgsConstructor
class User {
    private String name;
    private Integer age;
```

对象属性原子修改器

AtomicIntegerFieldUpdater可以线程安全地更新对象中的整型变量。

```
public class AtomicIntegerFieldUpdaterTest {

    public static class Candidate {

        volatile int score = 0;

        AtomicInteger score2 = new AtomicInteger();
    }

    public static final AtomicIntegerFieldUpdater<Candidate> scoreUpdater =
            AtomicIntegerFieldUpdater.newUpdater(Candidate.class, "score");

    public static AtomicInteger realScore = new AtomicInteger(0);

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

        final Candidate candidate = new Candidate();

        Thread[] t = new Thread[10000];
        for (int i = 0; i < 10000; i++) {
            t[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    if (Math.random() > 0.4) {
                        candidate.score2.incrementAndGet();
                        scoreUpdater.incrementAndGet(candidate);
                        realScore.incrementAndGet();
                    }
                }
            });
            t[i].start();
        }
        for (int i = 0; i < 10000; i++) {
            t[i].join();
```

```
35          }
36          System.out.println("AtomicIntegerFieldUpdater Score=" + candidate.score);
37          System.out.println("AtomicInteger Score=" + candidate.score2.get());
38          System.out.println("realScore=" + realScore.get());
39
40      }
```

对于AtomicIntegerFieldUpdater 的使用稍微有一些限制和约束，约束如下：

（1）字段必须是volatile类型的，在线程之间共享变量时保证立即可见.eg:volatile int value = 3

（2）字段的描述类型（修饰符public/protected/default/private）与调用者与操作对象字段的关系一致。也就是说调用者能够直接操作对象字段，那么就可以反射进行原子操作。但是对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。

（3）只能是实例变量，不能是类变量，也就是说不能加static关键字。

（4）只能是可修改变量，不能使final变量，因为final的语义就是不可修改。实际上final的语义和volatile是有冲突的，这两个关键字不能同时存在。

（5）对于AtomicIntegerFieldUpdater和AtomicLongFieldUpdater只能修改int/long类型的字段，不能修改其包装类型（Integer/Long）。如果要修改包装类型就需要使用AtomicReferenceFieldUpdater。

LongAdder/DoubleAdder详解

AtomicLong是利用了底层的CAS操作来提供并发性的，比如addAndGet方法：

```
public final long getAndAdd(long delta) {
    return unsafe.getAndAddLong( o: this, valueOffset, delta);
}
```

```
public final long getAndAddLong(Object var1, long var2, long var4) {
    long var6;
    do {
        var6 = this.getLongVolatile(var1, var2);
    } while(!this.compareAndSwapLong(var1, var2, var6, var6: var6 + var4));

    return var6;
}
```

上述方法调用了Unsafe类的getAndAddLong方法，该方法内部是个native方法，它的逻辑是采用自旋的方式不断更新目标值，直到更新成功。

在并发量较低的环境下，线程冲突的概率比较小，自旋的次数不会很多。但是，高并发环境下，N个线程同时进行自旋操作，会出现大量失败并不断自旋的情况，此时AtomicLong的自旋会成为瓶颈。

这就是**LongAdder引入的初衷——解决高并发环境下AtomicInteger，AtomicLong的自旋瓶颈问题。**

性能测试

```
public class LongAdderTest {

    public static void main(String[] args) {
        testAtomicLongVSLongAdder(10, 10000);
        System.out.println("==================");
        testAtomicLongVSLongAdder(10, 200000);
        System.out.println("==================");
        testAtomicLongVSLongAdder(100, 200000);
    }

    static void testAtomicLongVSLongAdder(final int threadCount, final int times) {
        try {
            long start = System.currentTimeMillis();
            testLongAdder(threadCount, times);
            long end = System.currentTimeMillis() - start;
            System.out.println("条件>>>>>>线程数:" + threadCount + ", 单线程操作计数" + tim
            System.out.println("结果>>>>>>LongAdder方式增加计数" + (threadCount * times) +

            long start2 = System.currentTimeMillis();
            testAtomicLong(threadCount, times);
            long end2 = System.currentTimeMillis() - start2;
            System.out.println("条件>>>>>>线程数:" + threadCount + ", 单线程操作计数" + tim
            System.out.println("结果>>>>>>AtomicLong方式增加计数" + (threadCount * times)
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static void testAtomicLong(final int threadCount, final int times) throws Interrupte
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        AtomicLong atomicLong = new AtomicLong();
        for (int i = 0; i < threadCount; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < times; j++) {
                        atomicLong.incrementAndGet();
                    }
```

```
                    countDownLatch.countDown();
                }
            }, "my-thread" + i).start();
        }
        countDownLatch.await();
    }

    static void testLongAdder(final int threadCount, final int times) throws InterruptedI
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        LongAdder longAdder = new LongAdder();
        for (int i = 0; i < threadCount; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < times; j++) {
                        longAdder.add(1);
                    }
                    countDownLatch.countDown();
                }
            }, "my-thread" + i).start();
        }

        countDownLatch.await();
    }
```

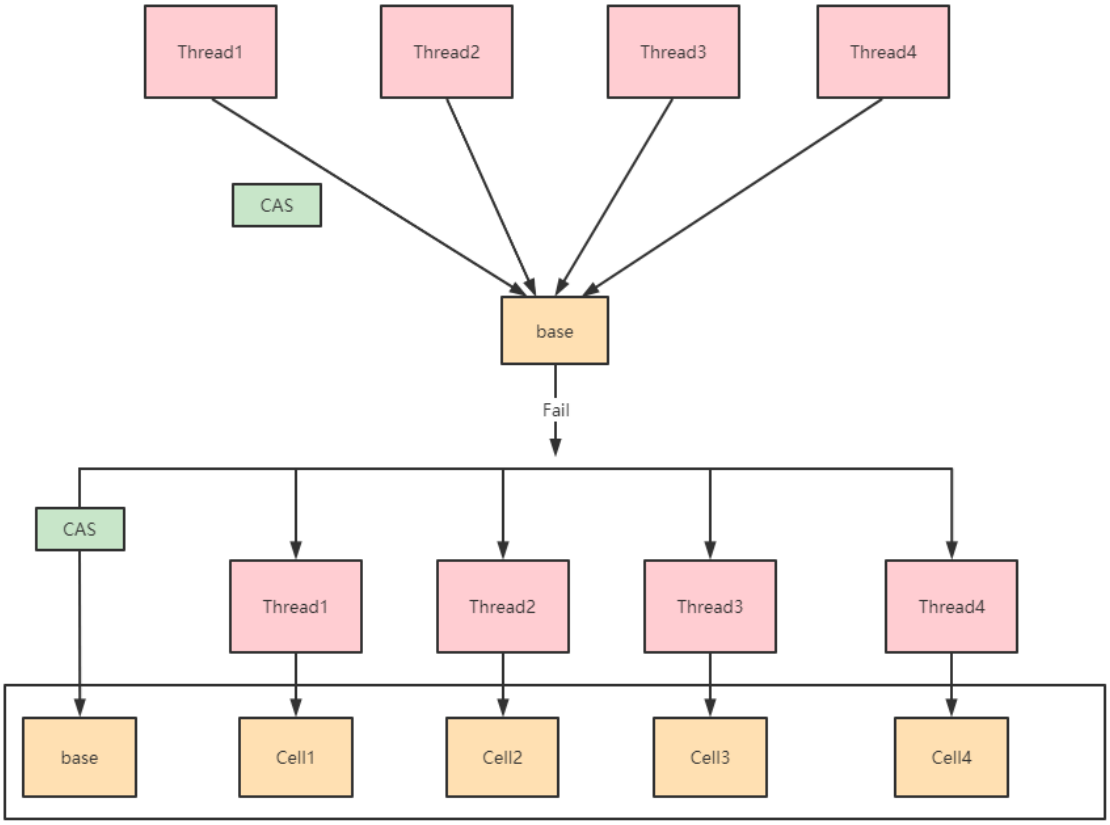
测试结果：线程数越多，并发操作数越大，LongAdder的优势越明显

低并发、一般的业务场景下AtomicLong是足够了。如果并发量很多，存在大量写多读少的情况，那LongAdder可能更合适。

LongAdder原理

设计思路

AtomicLong中有个内部变量value保存着实际的long值，所有的操作都是针对该变量进行。也就是说，高并发环境下，value变量其实是一个热点，也就是N个线程竞争一个热点。LongAdder的基本思路就是分散热点，将value值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的那个值进行CAS操作，这样热点就被分散了，冲突的概率就小很多。如果要获取真正的long值，只要将各个槽中的变量值累加返回。



LongAdder的内部结构

LongAdder内部有一个base变量，一个Cell[]数组：

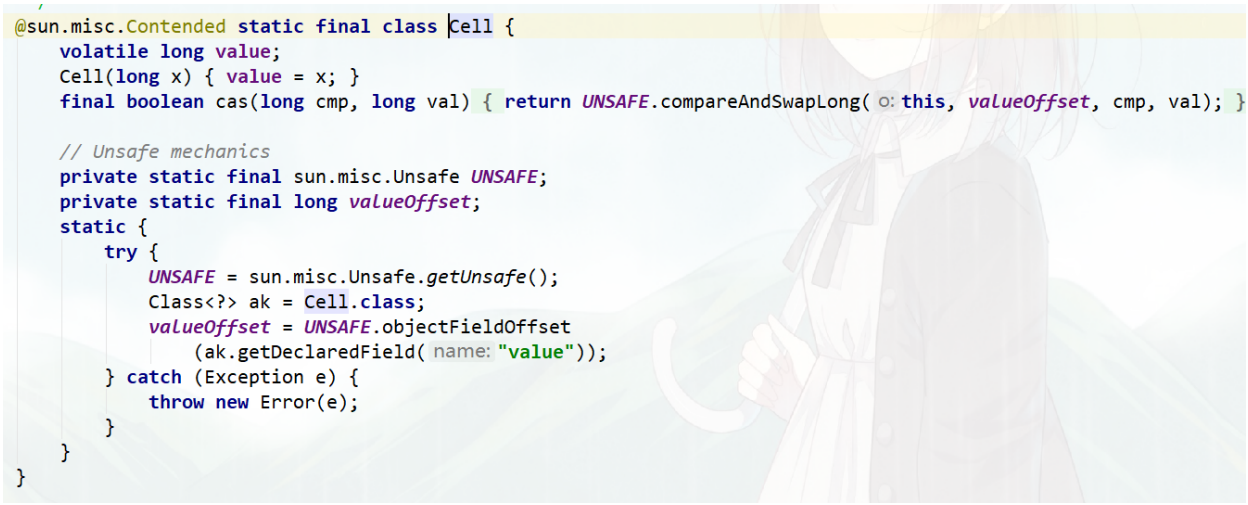
base变量：非竞态条件下，直接累加到该变量上

Cell[]数组：竞态条件下，累加个各个线程自己的槽Cell[i]中

```
/** Number of CPUS, to place bound on table size */
// CPU核数，用来决定槽数组的大小
static final int NCPU = Runtime.getRuntime().availableProcessors();

/**
 * Table of cells. When non-null, size is a power of 2.
 */
 // 数组槽，大小为2的次幂
transient volatile Cell[] cells;

/**
 * Base value, used mainly when there is no contention, but also as
 * a fallback during table initialization races. Updated via CAS.
 */
 /**
 *  基数，在两种情况下会使用：
 *  1. 没有遇到并发竞争时，直接使用base累加数值
```

```
 *  2. 初始化cells数组时，必须要保证cells数组只能被初始化一次（即只有一个线程能对cells初始化），
 *  其他竞争失败的线程会讲数值累加到base上
 */
transient volatile long base;

/**
 * Spinlock (locked via CAS) used when resizing and/or creating Cells.
 */
```

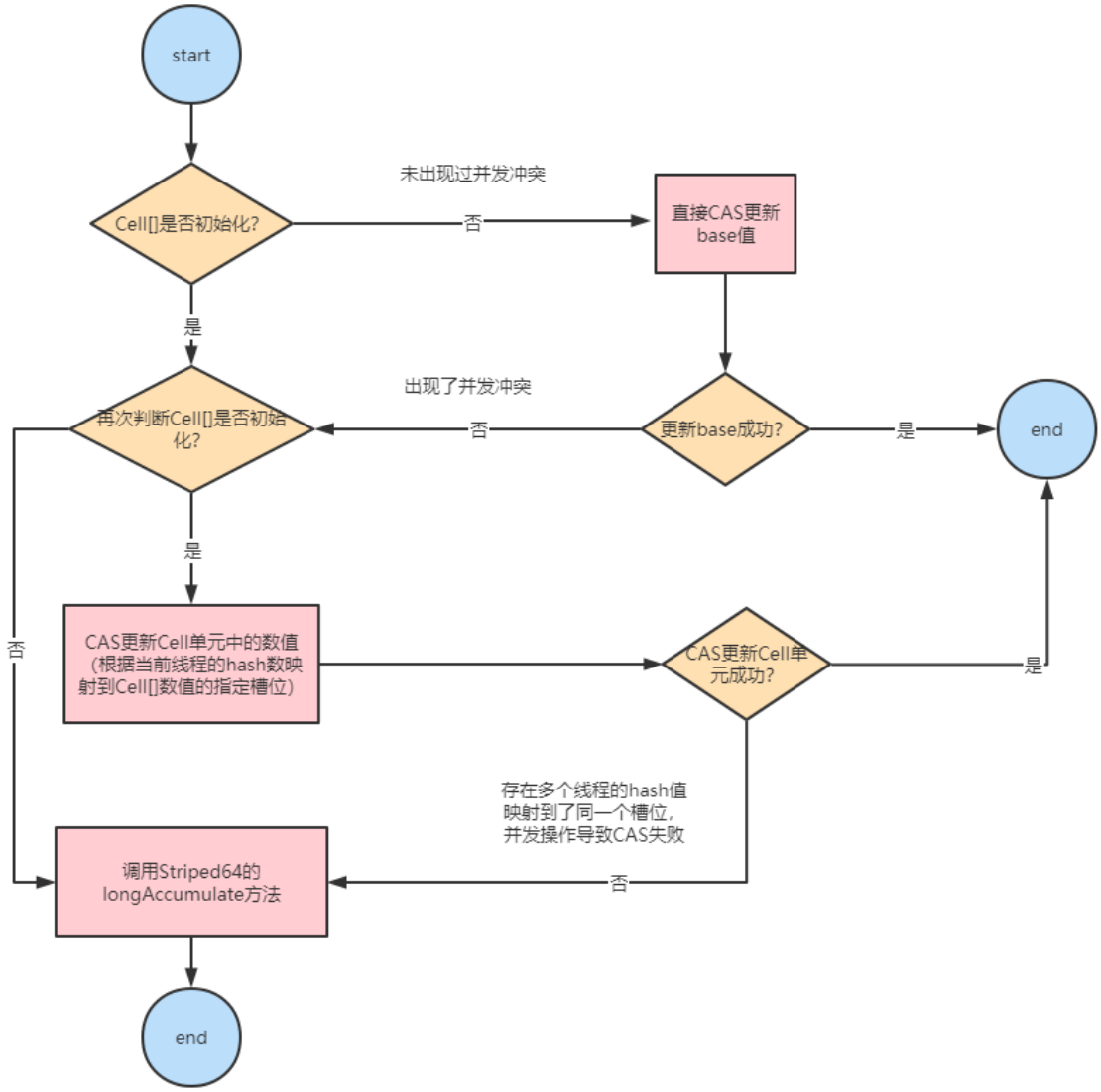
定义了一个内部Cell类，这就是我们之前所说的槽，每个Cell对象存有一个value值，可以通过Unsafe来CAS操作它的值：



```
@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( o: 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( name: "value"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

LongAdder#add方法

LongAdder#add方法的逻辑如下图：



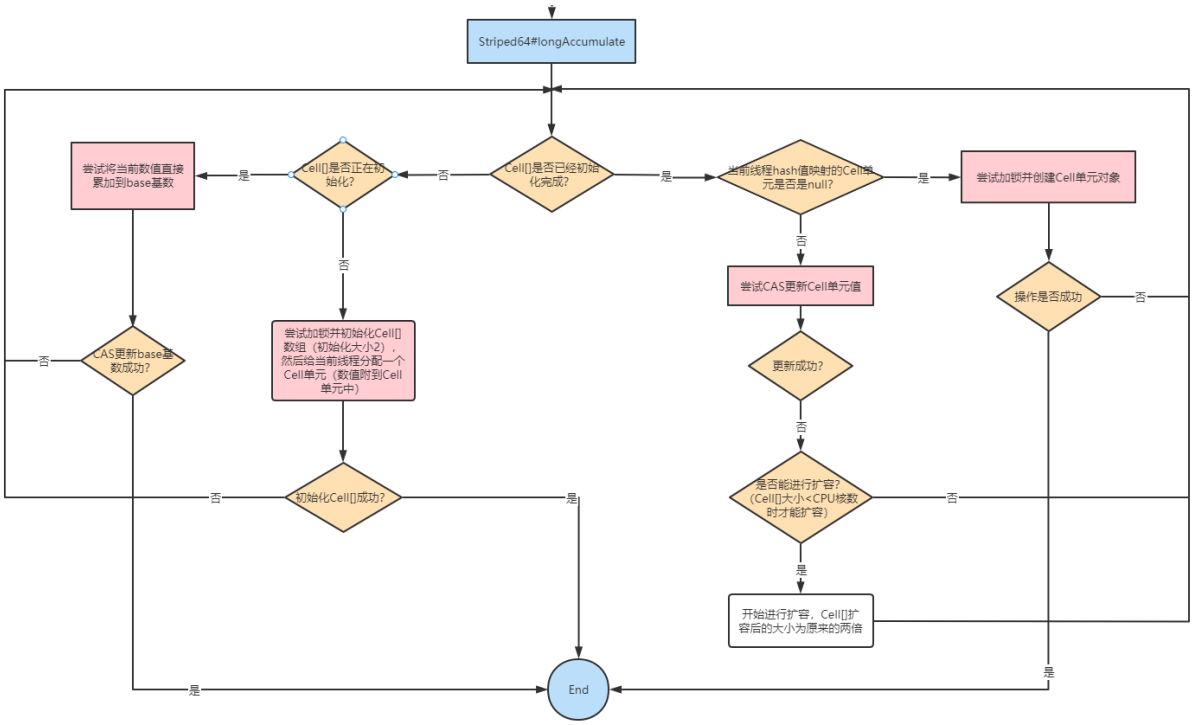
只有从未出现过并发冲突的时候，base基数才会使用到，一旦出现了并发冲突，之后所有的操作都只针对Cell[]数组中的单元Cell。

如果Cell[]数组未初始化，会调用父类的longAccumelate去初始化Cell[]，如果Cell[]已经初始化但是冲突发生在Cell单元内，则也调用父类的longAccumelate，此时可能就需要对Cell[]扩容了。

这也是LongAdder设计的精妙之处：尽量减少热点冲突，不到最后万不得已，尽量将CAS操作延迟。

Striped64#longAccumulate方法

整个Striped64#longAccumulate的流程图如下：



LongAdder#sum方法

```java
/**
* 返回累加的和，也就是"当前时刻"的计数值
* 注意： 高并发时，除非全局加锁，否则得不到程序运行中某个时刻绝对准确的值
*  此返回值可能不是绝对准确的，因为调用这个方法时还有其他线程可能正在进行计数累加，
*  方法的返回时刻和调用时刻不是同一个点，在有并发的情况下，这个值只是近似准确的计数值
*/
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;
```

由于计算总和时没有对Cell数组进行加锁，所以在累加过程中可能有其他线程对Cell中的值进行了修改，也有可能对数组进行了扩容，所以sum返回的值并不是非常精确的，其返回值并不是一个调用sum方法时的原子快照值。

LongAccumulator

LongAccumulator是LongAdder的增强版。LongAdder只能针对数值的进行加减运算，而LongAccumulator提供了自定义的函数操作。其构造函数如下：

```
public LongAccumulator(LongBinaryOperator accumulatorFunction,
                       long identity) {
    this.function = accumulatorFunction;
    base = this.identity = identity;
}
```

通过LongBinaryOperator，可以自定义对入参的任意操作，并返回结果（LongBinaryOperator接收2个long作为参数，并返回1个long）。LongAccumulator内部原理和LongAdder几乎完全一样，都是利用了父类Striped64的longAccumulate方法。

```
public class LongAccumulatorTest {

public static void main(String[] args) throws InterruptedException {
// 累加 x+y
LongAccumulator accumulator = new LongAccumulator((x, y) -> x + y, 0);

ExecutorService executor = Executors.newFixedThreadPool(8);
// 1到9累加
IntStream.range(1, 10).forEach(i -> executor.submit(() -> accumulator.accumulate(i)));

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

}
}
```