Java 并发计数组件Striped64详解

一字马胡 关注

0.674 2017.10.23 20:33:02 字数 2,098 阅读 5,807

> 作者： 一字马胡
> 转载标志 【2017-11-03】

更新日志

日期	更新内容	备注
2017-11-03	添加转载标志	持续更新

Java Striped64

Striped64是在java8中添加用来支持累加器的并发组件，它可以在并发环境下使用来做某种计数，Striped64的设计思路是在竞争激烈的时候尽量分散竞争，在实现上，Striped64维护了一个base Count和一个Cell数组，计数线程会首先试图更新base变量，如果成功则退出计数，否则会认为当前竞争是很激烈的，那么就会通过Cell数组来分散计数，Striped64根据线程来计算哈希，然后将不同的线程分散到不同的Cell数组的index上，然后这个线程的计数内容就会保存在该Cell的位置上面，基于这种设计，最后的总计数需要结合base以及散落在Cell数组中的计数内容。这种设计思路类似于java7的ConcurrentHashMap实现，也就是所谓的分段锁算法，ConcurrentHashMap会将记录根据key的hashCode来分散到不同的segment上，线程想要操作某个记录只需要锁住这个记录对应着的segment就可以了，而其他segment并不会被锁住，其他线程任然可以去操作其他的segment，这样就显著提高了并发度，虽然如此，java8中的ConcurrentHashMap实现已经抛弃了java7中分段锁的设计，而采用更为轻量级的CAS来协调并发，效率更佳。关于java8中的ConcurrentHashMap的分析可以参考文章Java 8 ConcurrentHashMap源码分析。

虽然Striped64的设计类似于分段锁算法，但是任然有其独到之处，本文将分析Striped64的实现细节，并且会分析基于Striped64的计数类LongAdder。Striped64的实现还是较为复杂的，本文会尽量分析，对于没有充分了解的内容，或者分析有误的内容，会在未来不断修改补充。

下面首先展示了Striped64中的Cell类：

```
/**
 * Padded variant of AtomicLong supporting only raw accesses plus CAS.
 *
 * JVM intrinsics note: It would be possible to use a release-only
 * form of CAS here, if it were provided.
 */
@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);
        }
    }
}
```

Cell类中仅有一个保存计数的变量value，并且为该变量提供了CAS操作方法，Cell类的实现虽然看起来很简单，但是它的作用是非常大的，它是Striped64实现分散计数的最为基础的数据结构，当然为了达到并发环境下的线程安全以及高效，Striped64做了很多努力。Striped64中有两个提供计数的api方法，分别为longAccumulate和doubleAccumulate，两者的实现思路是一致的，只是前者对long类型计数，而后者对double类型计数，本文只分析前者的实现，下面是longAccumulate方法的代码：

```
final void longAccumulate(long x, LongBinaryOperator fn,
                          boolean wasUncontended) {
        int h;
        if ((h = getProbe()) == 0) { //获取当前线程的probe值，如果为0，则需要初始化该线程的probe值
            ThreadLocalRandom.current(); // force initialization
            h = getProbe();
            wasUncontended = true;
        }
        boolean collide = false;                // True if last slot nonempty
        for (;;) {
            Cell[] as; Cell a; int n; long v;
            if ((as = cells) != null && (n = as.length) > 0) { //获取cell数组
                if ((a = as[(n - 1) & h]) == null) { // 通过 (hashCode & (length - 1)) 这种算法来实现取模
                    if (cellsBusy == 0) {       // 如果当前位置为null说明需要初始化
                        Cell r = new Cell(x);   // Optimistically create
                        if (cellsBusy == 0 && casCellsBusy()) {
                            boolean created = false;
                            try {               // Recheck under lock
                                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;
                }
                //运行到此说明cell的对应位置上已经有想相应的Cell了，不需要初始化了
                else if (!wasUncontended)       // CAS already known to fail
                    wasUncontended = true;      // Continue after rehash

                //尝试去修改a上的计数，a为Cell数组中index位置上的cell
                else if (a.cas(v = a.value, ((fn == null) ? v + x :
                                             fn.applyAsLong(v, x))))
                    break;

                //cell数组最大为cpu的数量，cells != as表面cells数组已经被更新了
                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]; //Cell数组扩容，每次扩容为原来的两倍
                            for (int i = 0; i < n; ++i)
                                rs[i] = as[i];
                            cells = rs;
                        }
                    } finally {
                        cellsBusy = 0;
                    }
                    collide = false;
                    continue;                   // Retry with expanded table
                }
                h = advanceProbe(h);
            }
            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);
```

```
                } finally {
                    cellsBusy = 0;
                }
                if (init)
                    break;
            }
            else if (casBase(v = base, ((fn == null) ? v + x :
                                        fn.applyAsLong(v, x))))
                break;                          // Fall back on using base
        }
    }

```

仅从代码量上就可以意识到longAccumulate的实现时异常复杂的，下面来梳理一下该方法的运行逻辑：

- longAccumulate会根据当前线程来计算一个哈希值，然后根据算法(hashCode & (length - 1))来达到取模的效果以定位到该线程被分散到的Cell数组中的位置
- 如果Cell数组还没有被创建，那么就去获取cellBusy这个共享变量（相当于锁，但是更为轻量级），如果获取成功，则初始化Cell数组，初始容量为2，初始化完成之后将x保证成一个Cell，哈希计算之后分散到相应的index上。如果获取cellBusy失败，那么会试图将x累计到base上，更新失败会重新尝试直到成功。
- 如果Cell数组以及被初始化过了，那么就根据线程的哈希值分散到一个Cell数组元素上，获取这个位置上的Cell并且赋值给变量a，这个a很重要，如果a为null，说明该位置还没有被初始化，那么就初始化，当然在初始化之前需要竞争cellBusy变量。
- 如果Cell数组的大小已经最大了（CPU的数量），那么就需要重新计算哈希，来重新分散当前线程到另外一个Cell位置上再走一遍该方法的逻辑，否则就需要对Cell数组进行扩容，然后将原来的计数内容迁移过去。这里面需要注意的是，因为Cell里面保存的是计数值，所以在扩容之后没有必要做其他的处理，直接根据index将旧的Cell数组内容直接复制到新的Cell数组中就可以了。

当然，上面的流程是高度概括的，longAccumulate的实际分支还要更多，并且为了保证线程安全做的判断更多。longAccumulate会根据不同的状态来执行不同的分支，比如在线程竞争非常激烈的时候，会通过对cells数组扩容或者从新计算哈希值来重新分散线程，这些做法的目的是将多个线程的计数请求分散到不同的cells的index上，其实这和java7中的ConcurrentHashMap的设计思路是完全一致的，但是java7中的ConcurrentHashMap实现在segment加锁使用了比较重的synchronized，而Striped64使用了java中较为底层的Unsafe类的CAS操作来进行并发操作，这种方式更为轻量级，因为它会不停的尝试，失败会返回，而加锁的方式会阻塞线程，线程需要被唤醒，这涉及到了线程的状态的改变，需要上下文切换，所以是比较重量级的。

Unsafe

在这里添加一点关于java中底层操作的类Unsafe类的使用方法，首先看下面的代码：

```
// Unsafe mechanics
private static final sun.misc.Unsafe UNSAFE;
private static final long BASE;
private static final long CELLSBUSY;
private static final long PROBE;
static {
    try {
        UNSAFE = sun.misc.Unsafe.getUnsafe();
        Class<?> sk = Striped64.class;
        BASE = UNSAFE.objectFieldOffset
            (sk.getDeclaredField( name: "base"));
        CELLSBUSY = UNSAFE.objectFieldOffset
            (sk.getDeclaredField( name: "cellsBusy"));
        Class<?> tk = Thread.class;
        PROBE = UNSAFE.objectFieldOffset
            (tk.getDeclaredField( name: "threadLocalRandomProbe"));
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

Unsafe需要关注的是Field的offset，然后在CAS的时候需要oldValue和expectValue以及newValue，它会在比较了oldValue == exceptValue的时候将oldValue设置为newValue，否则不会改变。这也是CAS的定义，（compare And set）下面的代码展示了CAS操作的示例：

```
UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val)

this是需要改变的对象，valueOffset为需要修改的Field在该对象中的offset，这个值的获取可以参考上面展示的
图片，cmp为exceptValue，也就是我们希望他的旧值为cmp值，如果相等，则将该Field设置为val，否则别修改。
```

LongAdder实现细节

上文中分析了Striped64的实现细节，下面来分析一下LongAdder的实现细节，LongAdder的实现基于Striped64，理解了Striped64就很好理解LongAdder了。下面先来看一下LongAdder的add方法：

```
/**
 * Adds the given value.
 *
 * @param x the value to add
 */
public void add(long x) {
    Cell[] as; long b, v; int m; Cell a;
    if ((as = cells) != null || !casBase(b = base, val: b + x)) {
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[getProbe() & m]) == null ||
            !(uncontended = a.cas(v = a.value, val: v + x)))
            longAccumulate(x, fn: null, uncontended);
    }
}
```

首先判断cells是否为null，如果为null，则会尝试将本次计数累计到base上，如果cells不为null，或者操作base失败，那么就会通过哈希值来获取当前线程对应的cells数组中的位置，获取该位置上的cell，如果该cell不为null，那么就试图将本次计数累计到该cell上，如果不成功，那么就需要借助Striped64类的longAccumulate方法来进行计数累计，关于longAccumulate的分析见上文。

当我们想要获得当前的总计数的时候，需要调用sum方法来获取，下面展示了该方法的细节：

```
/**
 * Returns the current sum.  The returned value is <em>NOT</em> an
 * atomic snapshot; invocation in the absence of concurrent
 * updates returns an accurate result, but concurrent updates that
 * occur while the sum is being calculated might not be
 * incorporated.
 *
 * @return the 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;
}
```

它需要累计base和Cell数组中的Cell中的计数，base中的计数为线程竞争不是很激烈的时候累计的数，而在线程竞争比较激烈的时候就会将计数的任务分散到Cell数组中，所以在sum方法里，需要合并两处的计数值。

除了获取总计数，我们有时候想reset一下，下面的代码展示了这种操作：

```
    public void reset() {
        Cell[] as = cells; Cell a;
        base = 0L;
        if (as != null) {
```

```
                    a.value = 0L;
            }
        }
    }
```

同样注意点在于需要同时将base和Cell数组都reset。

Striped64在ConcurrentHashMap中的使用

Striped64的计数方法在java8的ConcurrentHashMap中也有使用，具体的实现细节可以参考addCount方法，下面来看一下ConcurrentHashMap的size方法的实现细节：

```
    public int size() {
        long n = sumCount();
        return ((n < 0L) ? 0 :
                (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
                (int)n);
    }

    final long sumCount() {
        CounterCell[] as = counterCells; CounterCell a;
        long sum = baseCount;
        if (as != null) {
            for (int i = 0; i < as.length; ++i) {
                if ((a = as[i]) != null)
                    sum += a.value;
            }
        }
        return sum;
    }
```

ConcurrentHashMap中的baseCount对应着Striped64中的base变量，而counterCells则对应着Striped64中的cells数组，他们的实现时一样的，更为详细的内容可以参考java8中的ConcurrentHashMap实现。