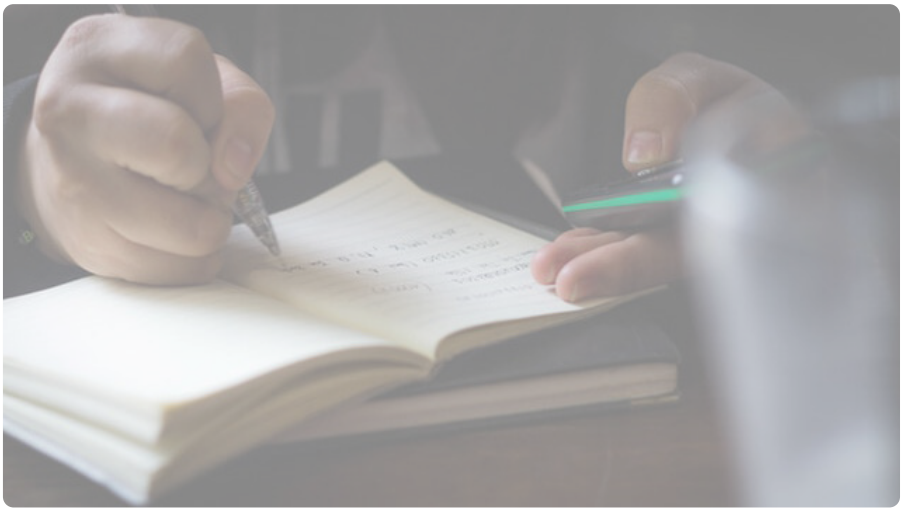
目录

33 CountDownLatch、Atomic 等其它源码解析

更新时间：2019-11-12 09:45:43



每个人的生命都是一只小船，理想是小船的风帆。

——张海迪

引导语

本小节和大家一起来看看 CountDownLatch 和 Atomic 打头的原子操作类，CountDownLatch 的源码非常少，看起来比较简单，但 CountDownLatch 的实际应用却不是很容易；Atomic 原子操作类就比较好理解和应用，接下来我们分别来看一下。

1 CountDownLatch

CountDownLatch 中文有的叫做计数器，也有翻译为计数锁，其最大的作用不是为了加锁，而是通过计数达到等待的功能，主要有两种形式的等待：

1. 让一组线程在全部启动完成之后，再一起执行（先启动的线程需要阻塞等待后启动的线程，直到一组线程全部都启动完成后，再一起执行）；
2. 主线程等待另外一组线程都执行完成之后，再继续执行。

我们会举一个示例来演示这两种情况，但在这之前，我们先来看看 CountDownLatch 的底层源码实现，这样就会清晰一点，不然一开始就来看示例，估计很难理解。

CountDownLatch 有两个比较重要的 API，分别是 await 和 countDown，管理着线程能否获得锁和锁的释放（也可以称为对 state 的计数增加和减少）。

1.1 await

await 我们可以叫做等待，也可以叫做加锁，有两种不同入参的方法，源码如下：

```
}
// 带有超时时间的，最终都会转化成毫秒
public boolean await(long timeout, TimeUnit unit)
    throws InterruptedException {
    return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
}
```

两个方法底层使用的都是 sync，sync 是一个同步器，是 CountDownLatch 的内部类实现的，如下：

```
private static final class Sync extends AbstractQueuedSynchronizer {}
```

可以看出来 Sync 继承了 AbstractQueuedSynchronizer，具备了同步器的通用功能。

无参 await 底层使用的是 acquireSharedInterruptibly 方法，有参的使用的是 tryAcquireSharedNanos 方法，这两个方法都是 AQS 的方法，底层实现很相似，主要分成两步：

1. 使用子类的 tryAcquireShared 方法尝试获得锁，如果获取了锁直接返回，获取不到锁走 2；
2. 获取不到锁，用 Node 封装一下当前线程，追加到同步队列的尾部，等待在合适的时机去获得锁。

第二步是 AQS 已经实现了，第一步 tryAcquireShared 方法是交给 Sync 实现的，源码如下：


```
// 如果当前同步器的状态是 0 的话，表示可获得锁
protected int tryAcquireShared(int acquires) {
    return (getState() == 0) ? 1 : -1;
}
```

获得锁的代码也很简单，直接根据同步器的 state 字段来进行判断，但还是有两点需要注意一下：

1. 获得锁时，state 的值不会发生变化，像 ReentrantLock 在获得锁时，会把 state + 1，但 CountDownLatch 不会；
2. CountDownLatch 的 state 并不是 AQS 的默认值 0，而是可以赋值的，是在 CountDownLatch 初始化的时候赋值的，代码如下：

```
// 初始化,count 代表 state 的初始化值
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    // new Sync 底层代码是 state = count;
    this.sync = new Sync(count);
}
```

这里的初始化的 count 和一般的锁意义不太一样，count 表示我们希望等待的线程数，在两种不同的等待场景中，count 有不同的含义：

1. 让一组线程在全部启动完成之后，再一起执行的等待场景下， count 代表一组线程的个数；
2. 主线程等待另外一组线程都执行完成之后，再继续执行的等待场景下，count 代表一组线程的个数。

1.2 countDown

countDown 中文翻译为倒计时，每调用一次，都会使 state 减一，底层调用的方法如下：

```
public void countDown() {
    sync.releaseShared(1);
}
```

releaseShared 是 AQS 定义的方法，方法主要分成两步：

1. 尝试释放锁（tryReleaseShared），锁释放失败直接返回，释放成功走 2；
2. 释放当前节点的后置等待节点。

第二步 AQS 已经实现了，第一步是 Sync 实现的，我们一起来看下 tryReleaseShared 方法的实现源码：

```
// 对 state 进行递减，直到 state 变成 0；
// state 递减为 0 时，返回 true，其余返回 false
protected boolean tryReleaseShared(int releases) {
    // 自旋保证 CAS 一定可以成功
    for (;;) {
        int c = getState();
        // state 已经是 0 了，直接返回 false
        if (c == 0)
            return false;
        // 对 state 进行递减
        int nextc = c-1;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
}
```


从源码中可以看到，只有到 count 递减到 0 时，countDown 才会返回 true。

1.3 示例

看完 CountDownLatch 两个重要 API 后，我们来实现文章开头说的两个功能：

1. 让一组线程在全部启动完成之后，再一起执行；
2. 主线程等待另外一组线程都执行完成之后，再继续执行。

代码在 CountDownLatchDemo 类中，大家可以调试看看，源码如下：

```
public class CountDownLatchDemo {

  // 线程任务
  class Worker implements Runnable {
    // 定义计数锁用来实现功能 1
    private final CountDownLatch startSignal;
    // 定义计数锁用来实现功能 2
    private final CountDownLatch doneSignal;

    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
      this.startSignal = startSignal;
```

目录

```
    public void run() {
      try {
        System.out.println(Thread.currentThread().getName()+" begin");
        // await 时有两点需要注意：await 时 state 不会发生变化，2：startSignal 的state初始化是 1，
        startSignal.await();
        doWork();
        // countDown 每次会使 state 减一，doneSignal 初始化为 9，countDown 前 8 次执行都会返
        doneSignal.countDown();
        System.out.println(Thread.currentThread().getName()+" end");
      } catch (InterruptedException ex) {
      } // return;
    }

    void doWork() throws InterruptedException {
      System.out.println(Thread.currentThread().getName()+"sleep 5s .....");
      Thread.sleep(5000l);
    }
  }

  @Test
  public void test() throws InterruptedException {
    // state 初始化为 1 很关键，子线程是不断的 await，await 时 state 是不会变化的，并且发现 sta
    // 造成所有线程都到同步队列中去等待，当主线程执行 countDown 时，就会一起把等待的线程给
    CountDownLatch startSignal = new CountDownLatch(1);
    // state 初始化成 9，表示有 9 个子线程执行完成之后，会唤醒主线程
    CountDownLatch doneSignal = new CountDownLatch(9);

    for (int i = 0; i < 9; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();

    System.out.println("main thread begin");
    // 这行代码唤醒 9 个子线程，开始执行(因为 startSignal 锁的状态是 1，所以调用一次 countDow
    startSignal.countDown();
    // 这行代码使主线程陷入沉睡，等待 9 个子线程执行完成之后才会继续执行(就是等待子线程执行 c
    doneSignal.await();
    System.out.println("main thread end");
  }
}
```

执行结果：

```
Thread-0 begin
Thread-1 begin
Thread-2 begin
Thread-3 begin
Thread-4 begin
Thread-5 begin
Thread-6 begin
Thread-7 begin
Thread-8 begin
main thread begin
Thread-0sleep 5s .....
Thread-1sleep 5s .....
Thread-4sleep 5s .....
Thread-3sleep 5s .....
Thread-2sleep 5s .....
Thread-8sleep 5s .....
Thread-7sleep 5s .....
Thread-6sleep 5s .....
Thread-5sleep 5s .....
Thread-0 end
Thread-1 end
Thread-4 end
```

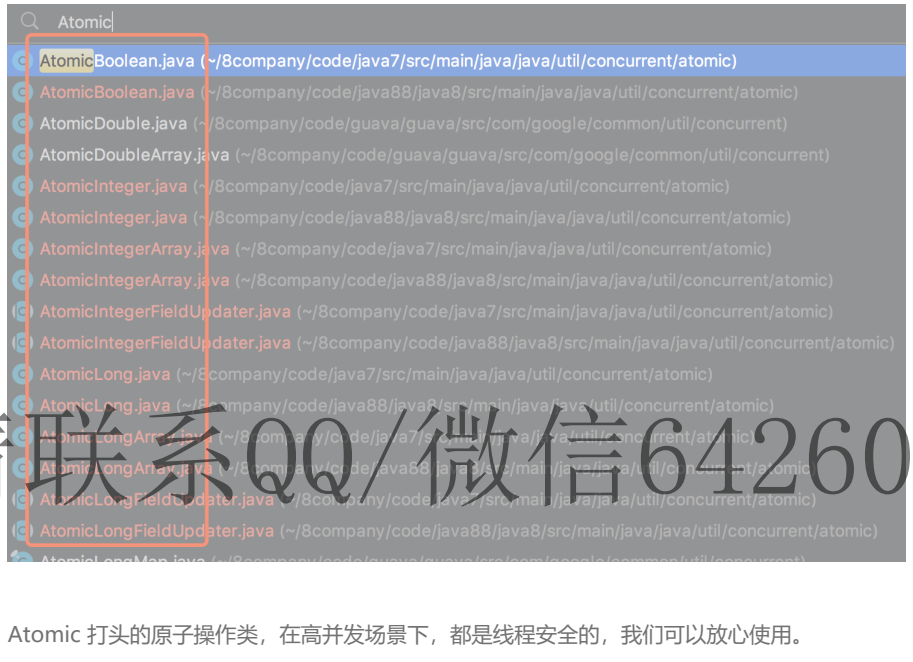
目录

```
Thread-7 end
Thread-6 end
Thread-5 end
main thread end
```

从执行结果中，可以看出已经实现了以上两个功能，实现比较绕，大家可以根据注释，debug 看一看。

2 Atomic 原子操作类

Atomic 打头的原子操作类有很多，涉及到 Java 常用的数字类型的，基本都有相应的 Atomic 原子操作类，如下图所示：



Atomic 打头的原子操作类，在高并发场景下，都是线程安全的，我们可以放心使用。

我们以 AtomicInteger 为例子，来看下主要的底层实现：

```
private volatile int value;

// 初始化
public AtomicInteger(int initialValue) {
    value = initialValue;
}
// 得到当前值
public final int get() {
    return value;
}
// 自增 1，并返回自增之前的值
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
// 自减 1，并返回自增之前的值
public final int getAndDecrement() {
    return unsafe.getAndAddInt(this, valueOffset, -1);
}
```

从源码中，我们可以看到，线程安全的操作方法，底层都是使用 unsafe 方法实现，以上几个 unsafe 方法不是使用 Java 实现的，都是线程安全的。

可操作的对象是个泛型，所以支持自定义类，其底层是没有自增方法的，操作的方法可以作为函数入参传递，源码如下：

```
// 对 x 执行 accumulatorFunction 操作
// accumulatorFunction 是个函数，可以自定义想做的事情
// 返回老值
public final V getAndAccumulate(V x,
                                BinaryOperator<V> accumulatorFunction) {
    // prev 是老值，next 是新值
    V prev, next;
    // 自旋 + CAS 保证一定可以替换老值
    do {
        prev = get();
        // 执行自定义操作
        next = accumulatorFunction.apply(prev, x);
    } while (!compareAndSet(prev, next));
    return prev;
}
```

3 总结

CountDownLatch 的源码实现简单，但真的要用好还是不简单的，其使用场景比较复杂，建议同学们可以 debug 一下 CountDownLatchDemo，在增加实战能力基础上，增加底层的理解能力。

32 ReentrantLock 源码解析 34 只求问例：连环相扣系列锁面试题

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

qs九点九分

github 给个地址

0 回复 2019-11-12

举个栗子啊丶 回复 **qs九点九分**

老师，您好，我也想要github地址

回复 2019-11-13 17:29:41

文贺 回复 **qs九点九分**

https://github.com/luanqiu/java8

回复 2019-11-17 10:52:24

文贺 回复 **举个栗子啊丶**

https://github.com/luanqiu/java8

回复 2019-11-17 10:52:27

目录