二

77 AQS 在 CountDownLatch 等类中的应用原理是什么?

本课时我们主要讲解 AQS 在 CountDownLatch 类中的应用原理，即在 CountDownLatch 中如何利用 AQS 去实现 CountDownLatch 自己的线程协作逻辑的。本课时会包含一定的源码分析。

AQS 用法

我们先讲一下 AQS 的用法。如果想使用 AQS 来写一个自己的线程协作工具类，通常而言是分为以下三步，这也是 JDK 里**利用 AQS 类的主要步骤**：

- **第一步**，新建一个自己的线程协作工具类，在内部写一个 Sync 类，该 Sync 类继承 AbstractQueuedSynchronizer，即 AQS；
- **第二步**，想好设计的线程协作工具类的协作逻辑，在 Sync 类里，根据是否是独占，来重写对应的方法。如果是独占，则重写 tryAcquire 和 tryRelease 等方法；如果是非独占，则重写 tryAcquireShared 和 tryReleaseShared 等方法；
- **第三步**，在自己的线程协作工具类中，实现获取/释放的相关方法，并在里面调用 AQS 对应的方法，如果是独占则调用 acquire 或 release 等方法，非独占则调用 acquireShared 或 releaseShared 或 acquireSharedInterruptibly 等方法。

通过这三步就可以实现对 AQS 的利用了。由于这三个步骤是经过浓缩和提炼的，所以现在你可能感觉有些不太容易理解，我们后面会有具体的实例来帮助理解，这里先有一个初步的印象即可。

你可能注意到了，上面的第二步是根据某些条件来重写特定的一部分方法，这个做法好像之前很少遇到过，或者说你可能会想，是不是有更好的做法？比如通过实现接口的方式，因为实现某一个接口之后，自然就知道需要重写其中哪些方法了，为什么要先继承类，然后自己去判断选择哪些方法进行重写呢？这不是自己给自己设置障碍吗？

关于这个问题的答案，其实在 AQS 的原作者 Doug Lea 的论文中已经进行了说明，他认为如果是实现接口的话，那**每一个抽象方法都需要实现。**比如你把整个 AQS 作为接口，那么需要实现的方法有很多，包括 tryAcquire、tryRelease、tryAcquireShared、

tryReleaseShared 等，但是实际上我们并不是每个方法都需要重写，根据需求的不同，有选择的去实现一部分就足以了，所以就设计为不采用实现接口，而采用继承类并重写方法的形式。

那可能你又有疑问了，继承类后，是不强制要求重写方法的，所以如果我们一个方法都不重写，行不行呢？答案是，如果不重写刚才所讲的 tryAcquire 等方法，是不行的，因为在执行的时候会抛出异常，我们来看下 AQS 对这些方法的默认的实现就知道了。

下面有四个方法的代码，分别是 tryAcquire、tryRelease、tryAcquireShared 和 tryReleaseShared 方法：

```
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
protected int tryAcquireShared(int arg) {
  throw new UnsupportedOperationException();
}
protected boolean tryReleaseShared(int arg) {
    throw new UnsupportedOperationException();
}
```

可以看到它们内部只有一行实现代码，就是直接抛出异常，所以要求我们在继承 AQS 之后，必须把相关方法去重写、覆盖，这样未来我们写的线程协作类才能正常的运行。

AQS 在 CountDownLatch 的应用

上面讲了使用 AQS 的基本流程，现在我们用例子来帮助理解，一起来看看 AQS 在 CountDownLatch 中的应用。

在 CountDownLatch 里面有一个子类，该类的类名叫 **Sync，这个类正是继承自 AQS。**下面给出了 CountDownLatch 部分代码的截取：

```
public class CountDownLatch {

    /**
     * Synchronization control For CountDownLatch.
     * Uses AQS state to represent count.
     */
    private static final class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 4982264981922014374L;
        Sync(int count) {
            setState(count);
        }
        int getCount() {
            return getState();
        }
        protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
        protected boolean tryReleaseShared(int releases) {
            // Decrement count; signal when transition to zero
            for (;;) {
                int c = getState();
                if (c == 0)
                    return false;
                int nextc = c-1;
                if (compareAndSetState(c, nextc))
                    return nextc == 0;
            }
        }
    }
```

```
    private final Sync sync;

   //省略其他代码...

}
```

可以很明显看到最开始一个 Sync 类继承了 AQS，这正是上一节所讲的“第一步，新建一个自己的线程协作工具类，在内部写一个 Sync 类，该 Sync 类继承 AbstractQueuedSynchronizer，即 AQS”。而在 CountDownLatch 里面还有一个 sync 的变量，正是 Sync 类的一个对象。

同时，我们看到，Sync 不但继承了 AQS 类，而且**还重写了 tryAcquireShared 和 tryReleaseShared 方法**，这正对应了“第二步，想好设计的线程协作工具类的协作逻辑，在 Sync 类里，根据是否是独占，来重写对应的方法。如果是独占，则重写 tryAcquire 或 tryRelease 等方法；如果是非独占，则重写 tryAcquireShared 和 tryReleaseShared 等方法”。

这里的 CountDownLatch 属于非独占的类型，因此它重写了 tryAcquireShared 和 tryReleaseShared 方法，那么这两个方法的具体含义是什么呢？别急，接下来就让我们对 CountDownLatch 类里面最重要的 4 个方法进行分析，逐步揭开它的神秘面纱。

构造函数

首先来看看构造函数。CountDownLatch 只有一个构造方法，传入的参数是需要“倒数”的次数，每次调用 countDown 方法就会倒数 1，直到达到了最开始设定的次数之后，相当于是“打开了门闩”，所以之前在等待的线程可以继续工作了。

我们具体来看下构造函数的代码：

```
public CountDownLatch(int count) {

    if (count < 0) throw new IllegalArgumentException("count < 0");

    this.sync = new Sync(count);

}
```

从代码中可以看到，当 count < 0 时会抛出异常，当 count > = 0，即代码 this.sync = new Sync(count)，往 Sync 中传入了 count，这个里的 Sync 的构造方法如下：

```
Sync(int count) {

     setState(count);
```

```
}
```

该构造函数调用了 AQS 的 setState 方法，并且把 count 传进去了，而 setState 正是给 AQS 中的 state 变量赋值的，代码如下：

```
protected final void setState(int newState) {
    state = newState;
}
```

所以我们通过 CountDownLatch 构造函数将传入的 count **最终传递到 AQS 内部的 state 变量**，给 state 赋值，state 就代表还需要倒数的次数。

getCount

接下来介绍 getCount 方法，该方法的作用是获取当前剩余的还需要“倒数”的数量，getCount 方法的源码如下：

```
public long getCount() {
     return sync.getCount();
}
```

该方法 return 的是 sync 的 getCount：

```
int getCount() {
     return getState();
}
```

我们一步步把源码追踪下去，getCount 方法调用的是 AQS 的 getState：

```
protected final int getState() {
    return state;
}
```

如代码所示，protected final int getState 方法直接 return 的就是 state 的值，所以最终它获取到的就在 AQS 中 state 变量的值。

countDown

我们再来看看 countDown 方法，该方法其实就是 CountDownLatch 的“**释放**”方法，下面来看下源码：

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

在 countDown 方法中调用的是 sync 的 releaseShared 方法：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

可以看出，releaseShared 先进行 if 判断，判断 tryReleaseShared 方法的返回结果，因此先把目光聚焦到 tryReleaseShared 方法中，tryReleaseShared 源码如下所示：

```
protected boolean tryReleaseShared(int releases) {
    // Decrement count; signal when transition to zero
    for (;;) {
        int c = getState();
        if (c == 0)
            return false;
        int nextc = c-1;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
```

```
}
```

方法内是一个 for 的死循环，在循环体中，最开始是通过 getState 拿到当前 state 的值并赋值给变量 c，这个 c 可以理解为是 count 的缩写，如果此时 c = 0，则意味着已经倒数为零了，会直接会执行下面的 return false 语句，一旦 tryReleaseShared 方法返回 false，再往上看上一层的 releaseShared 方法，就会直接跳过整个 if (tryReleaseShared(arg)) 代码块，直接返回 false，相当于 releaseShared 方法不产生效果，也就意味着 countDown 方法不产生效果。

再回到 tryReleaseShared 方法中往下看 return false 下面的语句，如果 c 不等于 0，在这里会先把 c-1 的值赋给 nextc，然后再利用 CAS 尝试把 nextc 赋值到 state 上。如果赋值成功就代表本次 countDown 方法操作成功，也就意味着把 AQS 内部的 state 值减了 1。最后，是 return nextc == 0，如果 nextc 为 0，意味着本次倒数后恰好达到了规定的倒数次数，门闩应当在此时打开，所以 tryReleaseShared 方法会返回 true，那么再回到之前的 releaseShared 方法中，可以看到，接下来会调用 doReleaseShared 方法，效果是**对之前阻塞的线程进行唤醒，让它们继续执行。**

如果结合具体的数来分析，可能会更清晰。假设 c = 2，则代表需要倒数的值是 2，nextc = c-1，所以 nextc 就是 1，然后利用 CAS 尝试把 state 设置为 1，假设设置成功，最后会 return nextc == 0，此时 nextc 等于 1，不等于 0，所以返回 false，也就意味着 countDown 之后成功修改了 state 的值，把它减 1 了，但并没有唤醒线程。

下一次执行 countDown时，c 的值就是 1，而 nextc = c - 1，所以 nextc 等于 0，若这时 CAS 操作成功，最后 return nextc == 0，所以方法返回 true，一旦 tryReleaseShared 方法 return true，则 releaseShared 方法会调用 doReleaseShared 方法，把所有之前阻塞的线程都唤醒。

await

接着我们来看看 await 方法，该方法是 CountDownLatch 的“**获取**”方法，调用 await 方法会把线程阻塞，直到倒数为 0 才能继续执行。await 方法和 countDown 是配对的，追踪源码可以看到 await 方法的实现：

```
public void await() throws InterruptedException {

    sync.acquireSharedInterruptibly(1);

}
```

它会调用 sync 的 acquireSharedInterruptibly ，并且传入 1。acquireSharedInterruptibly 方法源码如下所示：

```
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
```

可以看到，它除了对于中断的处理之外，比较重要的就是 tryAcquireShared 方法。这个方法很简单，它会直接判断 getState 的值是不是等于 0，如果等于 0 就返回 1，不等于 0 则返回 -1。

```
protected int tryAcquireShared(int acquires) {
    return (getState() == 0) ? 1 : -1;
}
```

getState 方法获取到的值是剩余需要倒数的次数，如果此时剩余倒数的次数大于 0，那么 getState 的返回值自然不等于 0，因此 tryAcquireShared 方法会返回 -1，一旦返回 -1，再看到 if (tryAcquireShared(arg) < 0) 语句中，就会符合 if 的判断条件，并且去执行 doAcquireSharedInterruptibly 方法，然后会**让线程进入阻塞状态。**

我们再来看下另一种情况，当 state 如果此时已经等于 0 了，那就意味着倒数其实结束了，不需要再去等待了，就是说门闩是打开状态，所以说此时 getState 返回 0，tryAcquireShared 方法返回 1 ，一旦返回 1，对于 acquireSharedInterruptibly 方法而言相当于立刻返回，也就意味着 await 方法会立刻返回，那么此时**线程就不会进入阻塞状态了**，相当于倒数已经结束，立刻放行了。

这里的 await 和 countDown 方法，正对应了本讲一开始所介绍的“第三步，在自己的线程协作工具类中，实现获取/释放的相关方法，并在里面调用 AQS 对应的方法，如果是独占则调用 acquire 或 release 等方法，非独占则调用 acquireShared 或 releaseShared 或 acquireSharedInterruptibly 等方法。”

AQS 在 CountDownLatch 的应用总结

最后对 AQS 在 CountDownLatch 的应用进行总结。当线程调用 CountDownLatch 的 await 方法时，便会尝试获取“共享锁”，不过一开始通常获取不到锁，于是线程被阻塞。“共享锁”

可获取到的条件是“锁计数器”的值为 0，而“锁计数器”的初始值为 count，当每次调用 CountDownLatch 对象的 countDown 方法时，也可以把“锁计数器” -1。通过这种方式，调用 count 次 countDown 方法之后，“锁计数器”就为 0 了，于是之前等待的线程就会继续运行了，并且此时如果再有线程想调用 await 方法时也会被立刻放行，不会再去做任何阻塞操作了。

总结

在本课时中我们主要介绍了 AQS 的用法，通常分为三步，然后以 CountDownLatch 为例，介绍了如何利用 AQS 实现自己的业务逻辑。

上一页 下一页