既然选择了吞吐量，谈何公平 —— ReentrantLock

大家好，我是徒手敲代码。

今天来介绍一下 ReentrantLock

它属于`java.util.concurrent.locks`包下的一个类，是一种可重入的互斥锁，允许同一个线程多次获取同一把锁而不会导致死锁。

与`synchronized`相比，ReentrantLock 允许尝试非阻塞地获取锁、支持锁的超时获取以及在等待锁时可被中断。另外，ReentrantLock 可以通过构造函数参数指定锁的获取是否公平，而`synchronized`总是采用非公平策略。

需要注意，Reentrantlock 不会像`synchronized`那样，在出现异常或者线程结束时自动释放锁，所以我们需要在`finaly`块中手动 释放锁，以避免发生死锁。

下面来探讨一下其中的底层原理。

可重入的实现

这个特性的实现，需要解决两个问题：

* **线程再次获取锁**：锁需要去识别获取锁的线程，是否为当前占据锁的线程，如果是，则再次成功获取
* **锁的最终释放**：线程重复 n 次获取了锁，随后在第 n 次释放该锁后，其他线程能够获取到该锁

具体实现看`nonfairTryAcquire`这个方法，这是非公平锁的加锁逻辑：

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
```

```
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

先判断当前锁的状态，0 表示还没有任何线程占用锁，CAS 操作申请获取锁；state 是非 0，表示有线程已经占用了锁，判断当前线程是否就是占用锁的线程，如果是，那么同步状态值递增。

从这个方法可以看出，成功获取锁的线程再次获取锁，只是增加了同步状态值，那么在释放的时候，也要减少同步状态值。具体的释放逻辑看`tryRelease`方法：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

当状态值为 0 的时候，才会将占有线程设置为`null`，锁才算是真正的被释放。

公平锁的实现

下面来说说公平锁是如何实现的。直接看`tryAcquire`这个方法：

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

跟非公平锁的加锁逻辑对比，很明显看出，多了`!hasQueuedPredecessors()`这个东西，再点进去这个方法里面看：

```
public final boolean hasQueuedPredecessors() {
    Node t = tail;
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

综合来看就是，当前线程的前面，没有其他线程在排队（当前线程节点就是队列第一个等待的节点），那么才会执行 CAS 和 加锁操作。

实际上，公平性锁每次都是从同步队列中的 第一个节点获取到锁，而非公平性锁则可能出现一个线程连续获取锁的情况。

既然非公平锁会造成线程饥饿的现象，那么为什么要搞成是默认实现呢?

这是因为，公平性锁保证了锁的获取是按照 FIFO 原则，而代价是进行大量的**线程切换**。非公平性锁虽然可能造成线程饥饿，但极少的线程切换，保证了其更大的**吞吐量**。

今天的分享到这里结束了。

公众号“**徒手敲代码**”，免费领取腾讯大佬推荐的Java电子书！

原文链接: https://juejin.cn/post/7374665776538861622