24 讲一讲公平锁和非公平锁，为什么要“非公平”？

本课时我们主要讲一讲公平锁和非公平锁，以及为什么要“非公平”？

什么是公平和非公平

首先，我们来看下什么是公平锁和非公平锁，公平锁指的是按照线程请求的顺序，来分配锁；而非公平锁指的是不完全按照请求的顺序，在一定情况下，可以允许插队。但需要注意这里的非公平并不是指完全的随机，不是说线程可以任意插队，而是仅仅“在合适的时机”插队。

那么什么时候是合适的时机呢？假设当前线程在请求获取锁的时候，恰巧前一个持有锁的线程释放了这把锁，那么当前申请锁的线程就可以不顾已经等待的线程而选择立刻插队。但是如果当前线程请求的时候，前一个线程并没有在那一时刻释放锁，那么当前线程还是一样会进入等待队列。

为了能够更好的理解公平锁和非公平锁，我们举一个生活中的例子，假设我们还在学校读书，去食堂排队买饭，我排在队列的第二个，我前面还有一位同学，但此时我脑子里想的不是午饭，而是上午的一道数学题并陷入深思，所以当前面的同学打完饭之后轮到我时我走神了，并也没注意到现在轮到我了，此时前面的同学突然又回来插队，说“不好意思，阿姨麻烦给我加个鸡腿”，像这样的行为就可以类比我们的公平锁和非公平锁。

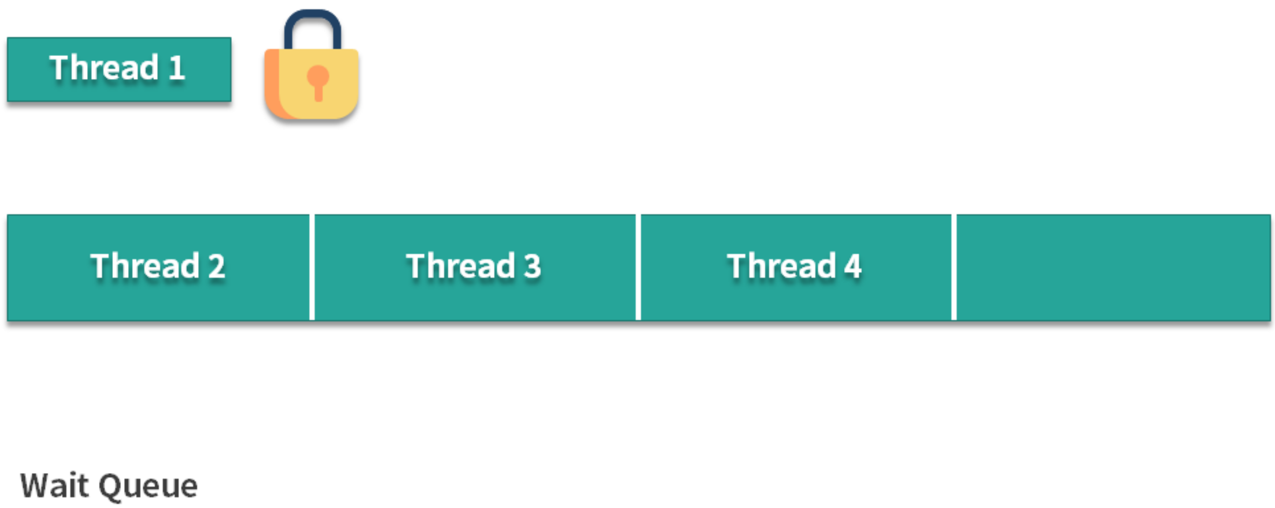
看到这里，你可能不解，为什么要设置非公平策略呢，而且非公平还是 ReentrantLock的默认策略，如果我们不加以设置的话默认就是非公平的，难道我的这些排队的时间都白白浪费了吗，为什么别人比我有优先权呢？毕竟公平是一种很好的行为，而非公平是一种不好的行为。

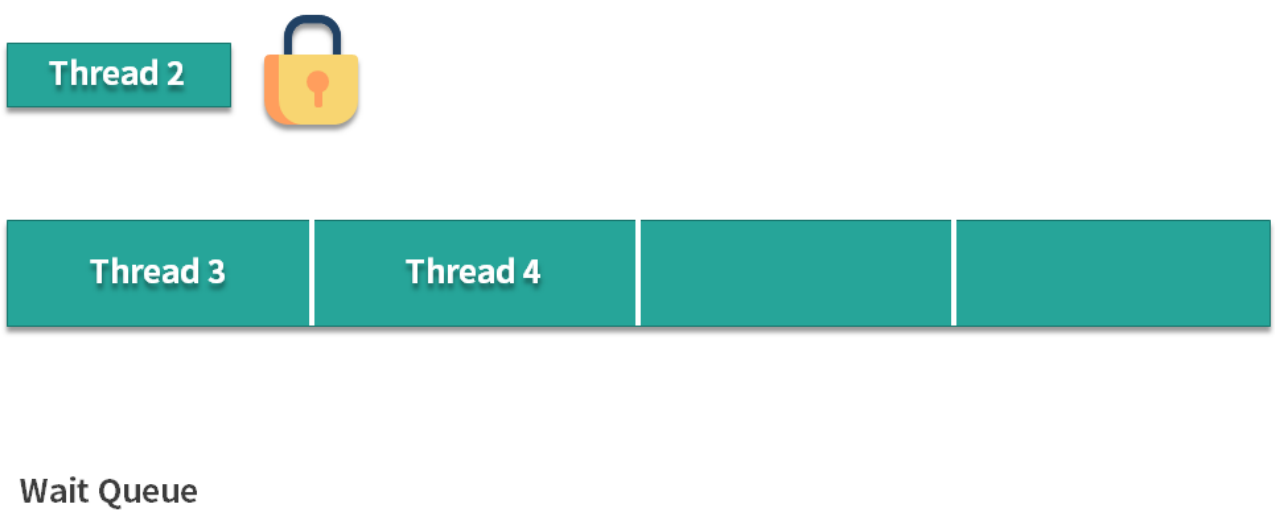
让我们考虑一种情况，假设线程 A 持有一把锁，线程 B 请求这把锁，由于线程 A 已经持有这把锁了，所以线程 B 会陷入等待，在等待的时候线程 B 会被挂起，也就是进入阻塞状态，那么当线程 A 释放锁的时候，本该轮到线程 B 苏醒获取锁，但如果此时突然有一个线程 C 插队请求这把锁，那么根据非公平的策略，会把这把锁给线程 C，这是因为唤醒线程 B 是需要很大开销的，很有可能在唤醒之前，线程 C 已经拿到了这把锁并且执行完任务释放了这把锁。相比于等待唤醒线程 B 的漫长过程，插队的行为会让线程 C 本身跳过陷入阻塞的过程，如果在锁代码中执行的内容不多的话，线程 C 就可以很快完成任务，并且在线

程 B 被完全唤醒之前，就把这个锁交出去，这样是一个双赢的局面，对于线程 C 而言，不需要等待提高了它的效率，而对于线程 B 而言，它获得锁的时间并没有推迟，因为等它被唤醒的时候，线程 C 早就释放锁了，因为线程 C 的执行速度相比于线程 B 的唤醒速度，是很快的，所以 Java 设计者设计非公平锁，是为了提高整体的运行效率。

公平的场景

下面我们用图示来说明公平和非公平的场景，先来看公平的情况。假设我们创建了一个公平锁，此时有 4 个线程按顺序来请求公平锁，线程 1 在拿到这把锁之后，线程 2、3、4 会在等待队列中开始等待，然后等线程 1 释放锁之后，线程 2、3、4 会依次去获取这把锁，线程 2 先获取到的原因是它等待的时间最长。

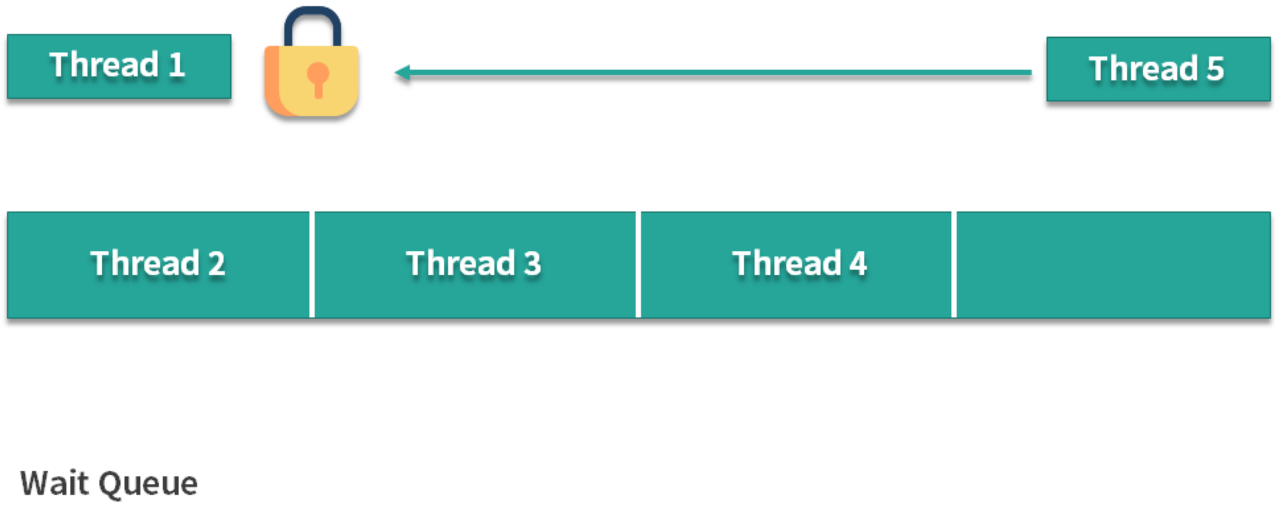




不公平的场景

下面我们再来看看非公平的情况，假设线程 1 在解锁的时候，突然有线程 5 尝试获取这把锁，那么根据我们的非公平策略，线程 5 是可以拿到这把锁的，尽管它没有进入等待队

列，而且线程 2、3、4 等待的时间都比线程 5 要长，但是从整体效率考虑，这把锁此时还是会交给线程 5 持有。



代码案例：演示公平和非公平的效果

下面我们来用代码演示看下公平和非公平的实际效果，代码如下：

```
/**
 * 描述：演示公平锁，分别展示公平和不公平的情况，非公平锁会让现在持有锁的线程优先再次获取到
 */
public class FairAndUnfair {
    public static void main(String args[]) {
        PrintQueue printQueue = new PrintQueue();
        Thread thread[] = new Thread[10];
        for (int i = 0; i < 10; i++) {
            thread[i] = new Thread(new Job(printQueue), "Thread " + i);
        }
        for (int i = 0; i < 10; i++) {
            thread[i].start();
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}
class Job implements Runnable {
    private PrintQueue printQueue;
    public Job(PrintQueue printQueue) {
        this.printQueue = printQueue;
    }
    @Override
    public void run() {
        System.out.printf("%s: Going to print a job\n", Thread.currentThread().getN
        printQueue.printJob(new Object());
        System.out.printf("%s: The document has been printed\n", Thread.currentThre
    }
}
class PrintQueue {
    private final Lock queueLock = new ReentrantLock(false);
    public void printJob(Object document) {
        queueLock.lock();
        try {
            Long duration = (long) (Math.random() * 10000);
            System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",
                    Thread.currentThread().getName(), (duration / 1000));
            Thread.sleep(duration);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        } finally {
            queueLock.unlock();
        }
        queueLock.lock();
        try {
            Long duration = (long) (Math.random() * 10000);
            System.out.printf("%s: PrintQueue: Printing a Job during %d seconds\n",
                    Thread.currentThread().getName(), (duration / 1000));
            Thread.sleep(duration);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            queueLock.unlock();
            }
    }
}
```

我们可以通过改变 new ReentrantLock(false) 中的参数来设置公平/非公平锁。以上代码在公平的情况下的输出：

```
Thread 0: Going to print a job
Thread 0: PrintQueue: Printing a Job during 5 seconds
Thread 1: Going to print a job
Thread 2: Going to print a job
Thread 3: Going to print a job
Thread 4: Going to print a job
Thread 5: Going to print a job
Thread 6: Going to print a job
Thread 7: Going to print a job
```

```
Thread 8: Going to print a job
Thread 9: Going to print a job
Thread 1: PrintQueue: Printing a Job during 3 seconds
Thread 2: PrintQueue: Printing a Job during 4 seconds
Thread 3: PrintQueue: Printing a Job during 3 seconds
Thread 4: PrintQueue: Printing a Job during 9 seconds
Thread 5: PrintQueue: Printing a Job during 5 seconds
Thread 6: PrintQueue: Printing a Job during 7 seconds
Thread 7: PrintQueue: Printing a Job during 3 seconds
Thread 8: PrintQueue: Printing a Job during 9 seconds
Thread 9: PrintQueue: Printing a Job during 5 seconds
Thread 0: PrintQueue: Printing a Job during 8 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 1 seconds
Thread 1: The document has been printed
Thread 2: PrintQueue: Printing a Job during 8 seconds
Thread 2: The document has been printed
Thread 3: PrintQueue: Printing a Job during 2 seconds
Thread 3: The document has been printed
Thread 4: PrintQueue: Printing a Job during 0 seconds
Thread 4: The document has been printed
Thread 5: PrintQueue: Printing a Job during 7 seconds
Thread 5: The document has been printed
Thread 6: PrintQueue: Printing a Job during 3 seconds
Thread 6: The document has been printed
Thread 7: PrintQueue: Printing a Job during 9 seconds
Thread 7: The document has been printed
Thread 8: PrintQueue: Printing a Job during 5 seconds
```

```
Thread 8: The document has been printed

Thread 9: PrintQueue: Printing a Job during 9 seconds

Thread 9: The document has been printed
```

可以看出，线程直接获取锁的顺序是完全公平的，先到先得。

而以上代码在非公平的情况下的输出是这样的：

```
Thread 0: Going to print a job

Thread 0: PrintQueue: Printing a Job during 6 seconds

Thread 1: Going to print a job

Thread 2: Going to print a job

Thread 3: Going to print a job

Thread 4: Going to print a job

Thread 5: Going to print a job

Thread 6: Going to print a job

Thread 7: Going to print a job

Thread 8: Going to print a job

Thread 9: Going to print a job

Thread 0: PrintQueue: Printing a Job during 8 seconds

Thread 0: The document has been printed

Thread 1: PrintQueue: Printing a Job during 9 seconds

Thread 1: PrintQueue: Printing a Job during 8 seconds

Thread 1: The document has been printed

Thread 2: PrintQueue: Printing a Job during 6 seconds

Thread 2: PrintQueue: Printing a Job during 4 seconds

Thread 2: The document has been printed

Thread 3: PrintQueue: Printing a Job during 9 seconds

Thread 3: PrintQueue: Printing a Job during 8 seconds

Thread 3: The document has been printed
```

```
Thread 4: PrintQueue: Printing a Job during 4 seconds

Thread 4: PrintQueue: Printing a Job during 2 seconds

Thread 4: The document has been printed

Thread 5: PrintQueue: Printing a Job during 2 seconds

Thread 5: PrintQueue: Printing a Job during 5 seconds

Thread 5: The document has been printed

Thread 6: PrintQueue: Printing a Job during 2 seconds

Thread 6: PrintQueue: Printing a Job during 6 seconds

Thread 6: The document has been printed

Thread 7: PrintQueue: Printing a Job during 6 seconds

Thread 7: PrintQueue: Printing a Job during 4 seconds

Thread 7: The document has been printed

Thread 8: PrintQueue: Printing a Job during 3 seconds

Thread 8: PrintQueue: Printing a Job during 6 seconds

Thread 8: The document has been printed

Thread 9: PrintQueue: Printing a Job during 3 seconds

Thread 9: PrintQueue: Printing a Job during 5 seconds

Thread 9: The document has been printed
```

可以看出，非公平情况下，存在抢锁“插队”的现象，比如Thread 0 在释放锁后又能优先获取到锁，虽然此时在等待队列中已经有 Thread 1 ~ Thread 9 在排队了。

对比公平和非公平的优缺点

我们接下来对比公平和非公平的优缺点，如表格所示。

	优势	劣势
公平锁	各线程公平平等，每个线程在等待一段时间后，总有执行的机会	更慢，吞吐量更小
不公平锁	更快，吞吐量更大	有可能产生线程饥饿，也

		就是某些线程在长时间内，始终得不到执行

公平锁的优点在于各个线程公平平等，每个线程等待一段时间后，都有执行的机会，而它的缺点就在于整体执行速度更慢，吞吐量更小，相反非公平锁的优势就在于整体执行速度更快，吞吐量更大，但同时也可能产生线程饥饿问题，也就是说如果一直有线程插队，那么在等待队列中的线程可能长时间得不到运行。

源码分析

下面我们来分析公平和非公平锁的源码，具体看下它们是怎样实现的，可以看到在 ReentrantLock 类包含一个 Sync 类，这个类继承自 AQS（AbstractQueuedSynchronizer），代码如下：

```
public class ReentrantLock implements Lock, java.io.Serializable {

private static final long serialVersionUID = 7373984872572414699L;

/** Synchronizer providing all implementation mechanics */

private final Sync sync;
```

Sync 类的代码：

```
abstract static class Sync extends AbstractQueuedSynchronizer {...}
```

根据代码可知，Sync 有公平锁 FairSync 和非公平锁 NonfairSync两个子类：

```
static final class NonfairSync extends Sync {...}

static final class FairSync extends Sync {...}
```

下面我们来看一下公平锁与非公平锁的加锁方法的源码。

公平锁的锁获取源码如下：

```
protected final boolean tryAcquire(int acquires) {

    final Thread current = Thread.currentThread();

    int c = getState();

    if (c == 0) {
```

```
        if (!hasQueuedPredecessors() && //这里判断了 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;
}
```

非公平锁的锁获取源码如下：

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) { //这里没有判断      hasQueuedPredeces
            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;

    }
```

通过对比，我们可以明显的看出公平锁与非公平锁的 lock() 方法唯一的区别就在于公平锁在获取锁时多了一个限制条件：hasQueuedPredecessors() 为 false，这个方法就是判断在等待队列中是否已经有线程在排队了。这也就是公平锁和非公平锁的核心区别，如果是公平锁，那么一旦已经有线程在排队了，当前线程就不再尝试获取锁；对于非公平锁而言，无论是否已经有线程在排队，都会尝试获取一下锁，获取不到的话，再去排队。

这里有一个特例需要我们注意，针对 tryLock() 方法，它不遵守设定的公平原则。

例如，当有线程执行 tryLock() 方法的时候，一旦有线程释放了锁，那么这个正在 tryLock 的线程就能获取到锁，即使设置的是公平锁模式，即使在它之前已经有其他正在等待队列中等待的线程，简单地说就是 tryLock 可以插队。

看它的源码就会发现：

```
    public boolean tryLock() {

        return sync.nonfairTryAcquire(1);

    }
```

这里调用的就是 nonfairTryAcquire()，表明了是不公平的，和锁本身是否是公平锁无关。

综上所述，公平锁就是会按照多个线程申请锁的顺序来获取锁，从而实现公平的特性。非公平锁加锁时不考虑排队等待情况，直接尝试获取锁，所以存在后申请却先获得锁的情况，但由此也提高了整体的效率。

上一页 下一页