> 点赞再看，养成习惯，微信搜索【**三太子敖丙**】关注这个互联网苟且偷生的工具人。

> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言

上次我们提到了乐观锁和悲观锁，那我们知道锁的类型还有很多种，我们今天简单聊一下，公平锁和非公平锁两口子，以及他们在我们代码中的实践。

正文

开始聊之前，我先大概说一下他们两者的定义，帮大家回顾或者认识一下。

公平锁：多个线程按照申请锁的顺序去获得锁，线程会直接进入队列去排队，永远都是队列的第一位才能得到锁。

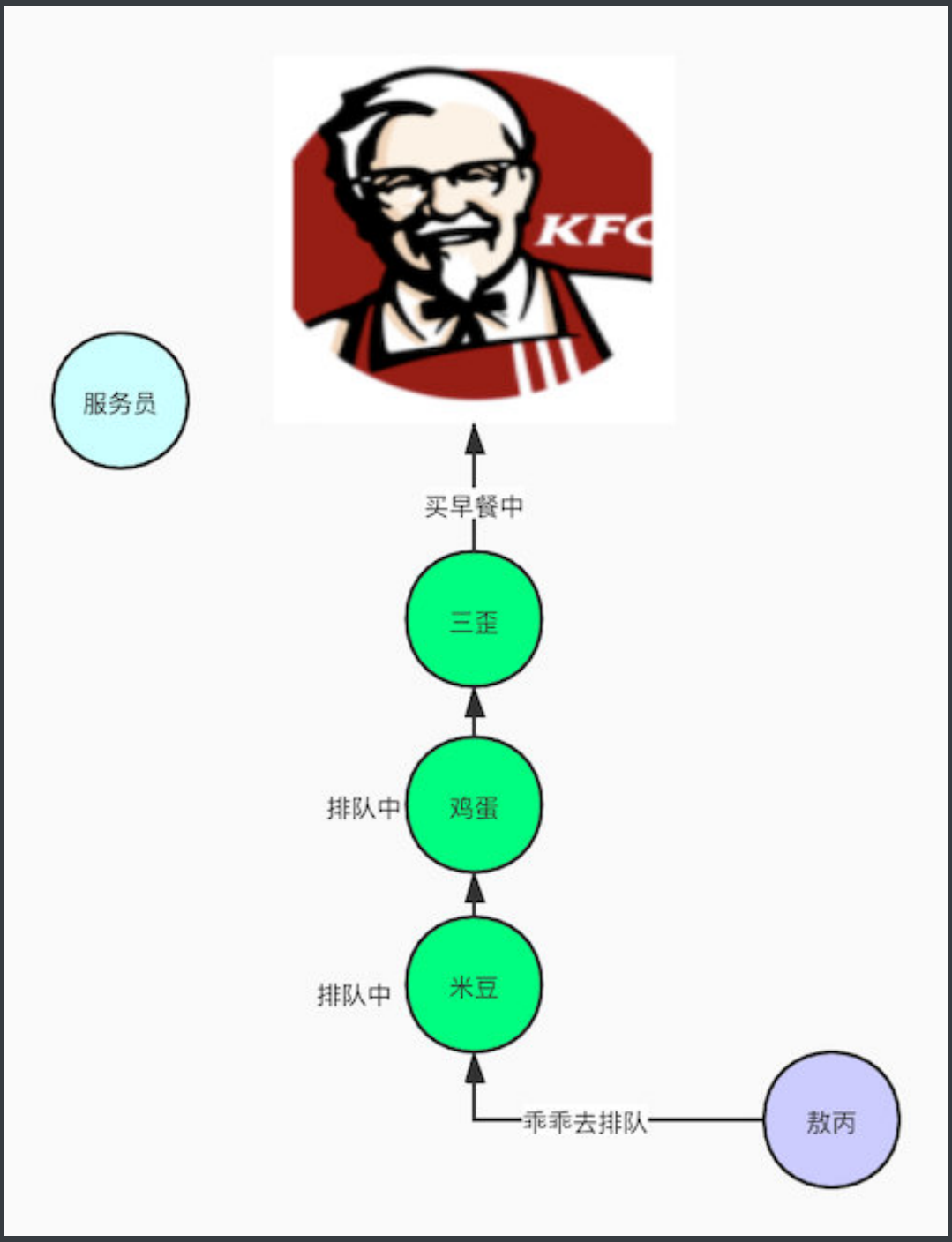
- 优点：所有的线程都能得到资源，不会饿死在队列中。
- 缺点：吞吐量会下降很多，队列里面除了第一个线程，其他的线程都会阻塞，cpu唤醒阻塞线程的开销会很大。

非公平锁：多个线程去获取锁的时候，会直接去尝试获取，获取不到，再去进入等待队列，如果能获取到，就直接获取到锁。

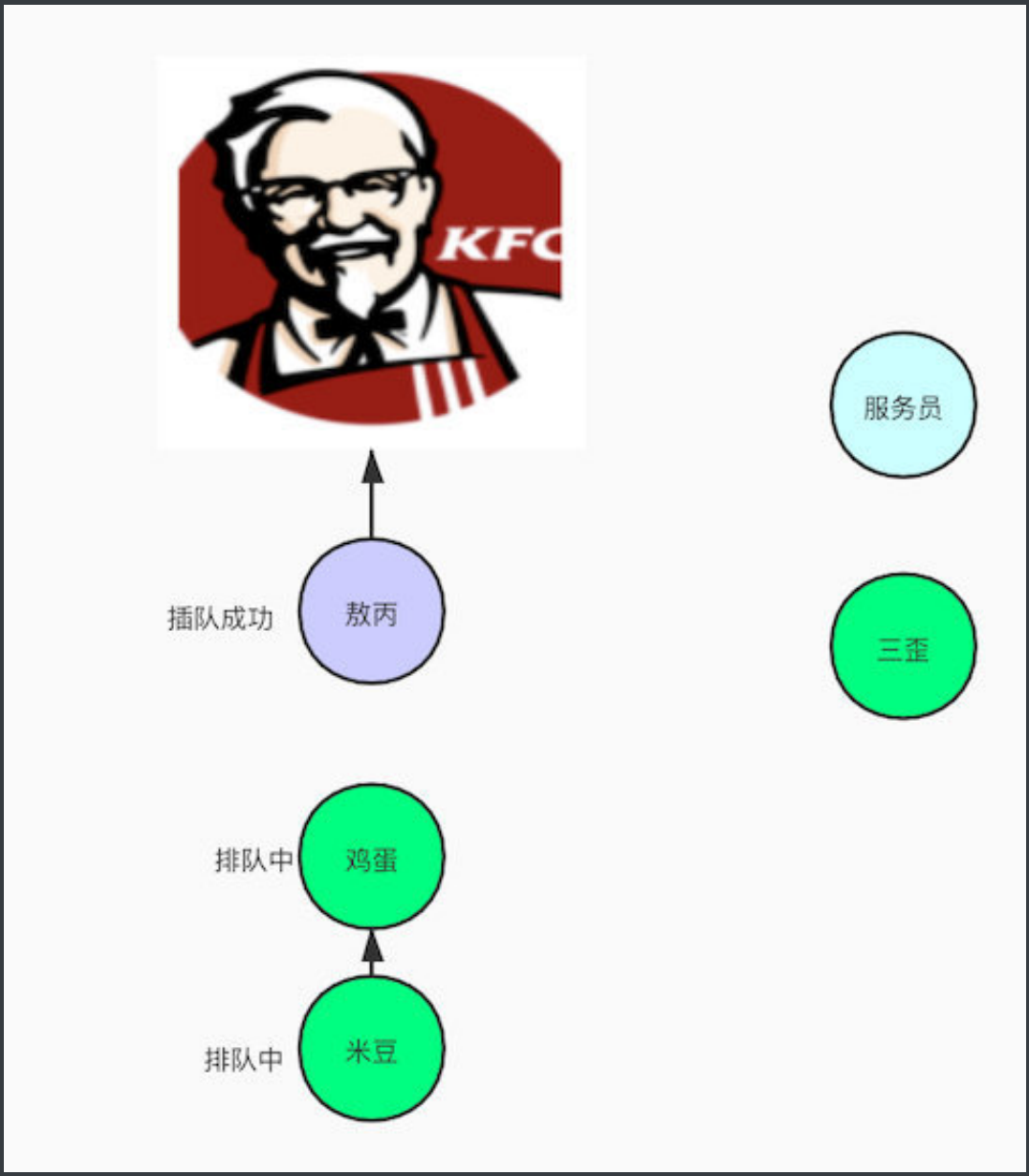
- 优点：可以减少CPU唤醒线程的开销，整体的吞吐效率会高点，CPU也不必取唤醒所有线程，会减少唤起线程的数量。
- 缺点：你们可能也发现了，这样可能导致队列中间的线程一直获取不到锁或者长时间获取不到锁，导致饿死。

我举个例子给他家通俗易懂的讲一下的，想了好几天终于在前天跟三歪去肯德基买早餐排队的时候发现了怎么举例了。

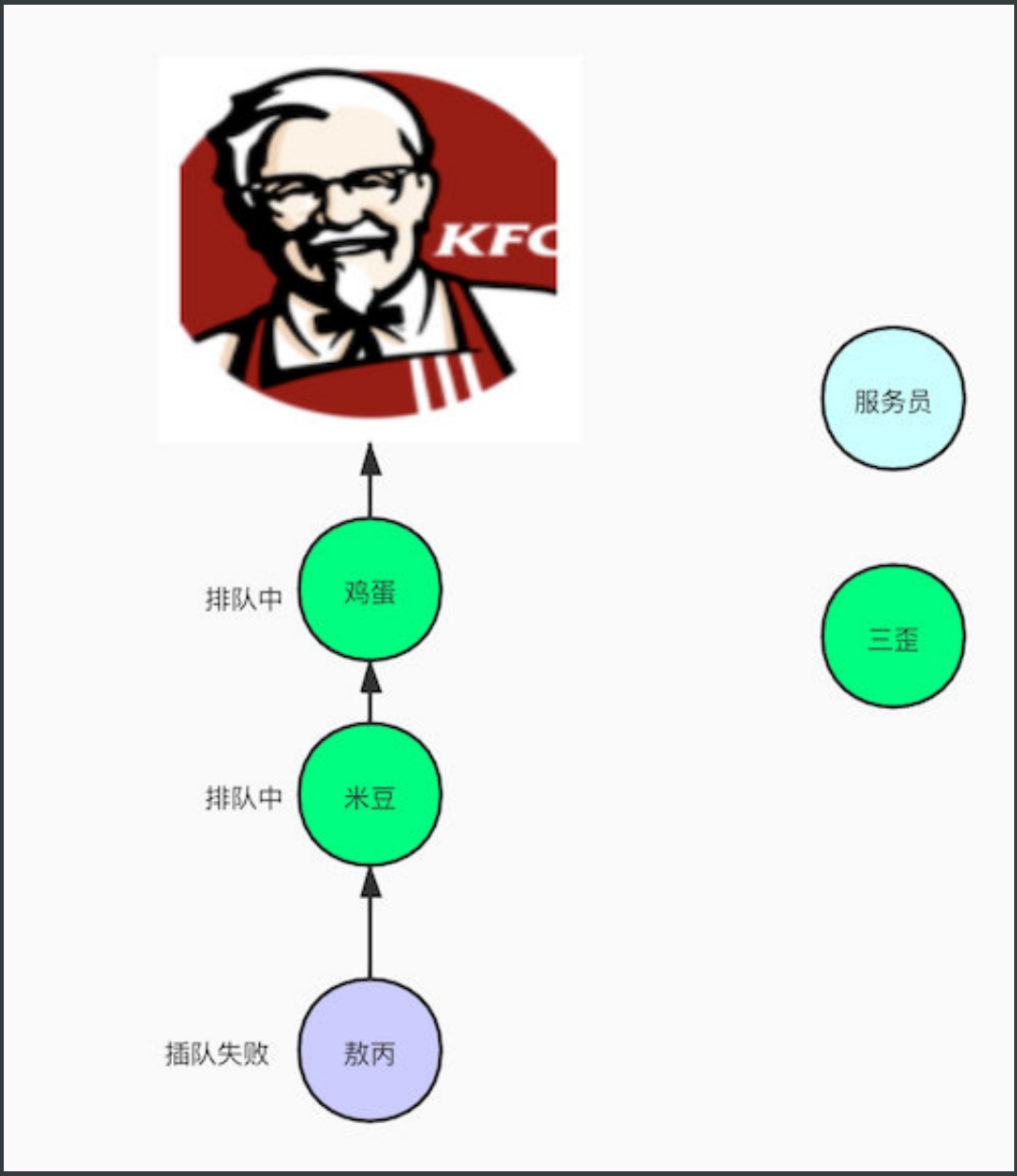
现在是早餐时间，敖丙想去kfc搞个早餐，发现有很多人了，一过去没多想，就乖乖到队尾排队，这样大家都觉得很公平，先到先得，所以这是公平锁咯。



那非公平锁就是，敖丙过去买早餐，发现大家都在排队，但是敖丙这个人有点渣的，就是喜欢插队，那他就直接怼到第一位那去，后面的鸡蛋，米豆都不行，我插队也不敢说什么，只能默默忍受了。



但是偶尔，鸡蛋也会崛起，叫我滚到后面排队，我也是欺软怕硬，默默到后面排队，就插队失败了。



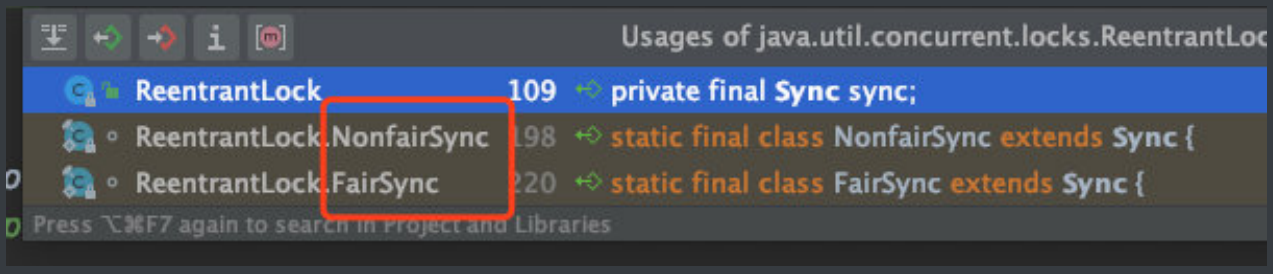
介绍完简单的例子，大家可能会说，渣丙，这个我也知道的啊。

我们是不是应该回归真正的实现了，其实在大家经常使用的ReentrantLock中就有相关公平锁，非公平锁的实现了。

大家还记得我在乐观锁、悲观锁章节提到的Sync类么，是ReentrantLock他本身的一个内部类，他继承了AbstractQueuedSynchronizer，我们在操作锁的大部分操作，都是Sync本身去实现的。

```
 */
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    private final Sync sync;

    /**
     * Base of synchronization control for this lock. Subclassed
     * into fair and nonfair versions below. Uses AQS state to
     * represent the number of holds on the lock.
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = -5179523762034025860L;

        /**
         * Performs {@link Lock#lock}. The main reason for subclassing
```

Sync呢又分别有两个子类：FairSync和NofairSync



他们子类的名字就可以见名知意了，公平和不公平那又是怎么在代码层面体现的呢?

公平锁：

```
/**
 * Sync object for fair locks
 */
static final class FairSync extends Sync {
    private static final long serialVersionUID = -3000897897090466540L;

    final void lock() { acquire( arg: 1); }

    /**
     * Fair version of tryAcquire.  Don't grant access unless
     * recursive call or no waiters or is first.
     */
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (!hasQueuedPredecessors() &&
                compareAndSetState( expect: 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;
    }
}
```

你可以看到，他加了一个hasQueuedPredecessors的判断，那他判断里面有些什么玩意呢?

```
public final boolean hasQueuedPredecessors() {
    // The correctness of this depends on head being initialized
    // before tail and on head.next being accurate if the current
    // thread is first in queue.
    Node t = tail; // Read fields in reverse initialization order
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

代码的大概意思也是判断当前的线程是不是位于同步队列的首位，是就是返回true，否就返回false。

我总觉得写到这里就应该差不多了，但是我坐下来，静静的思考之后发现，还是差了点什么。

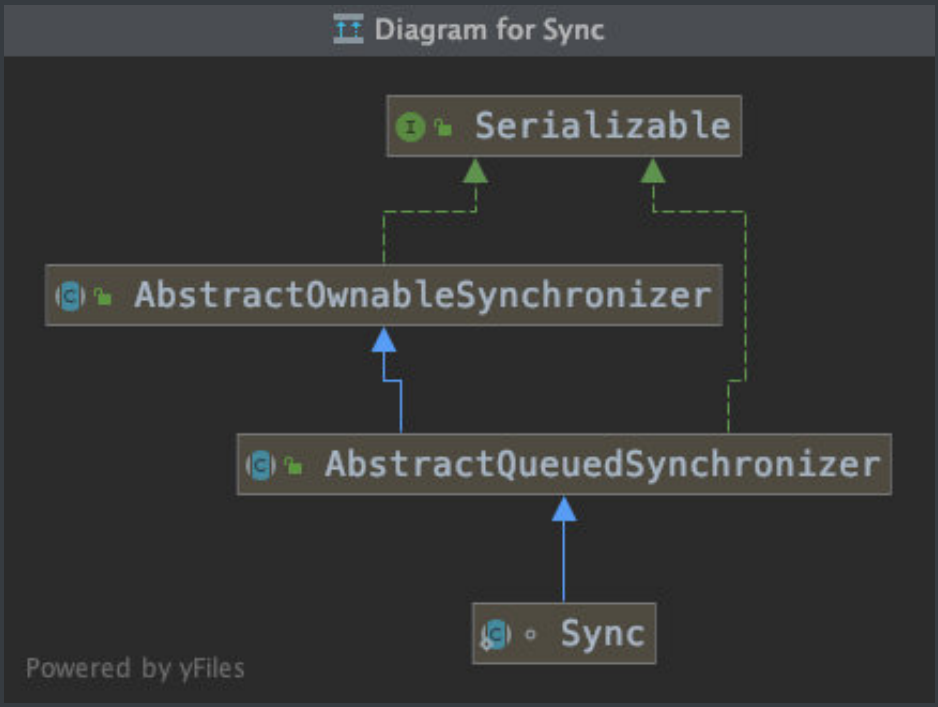
上次聊过ReentrantLock了，但是AQS什么的我都只是提了一嘴，一个线程进来，他整个处理链路到底是怎样的呢?

公平锁到底公平不公平呢？让我们一起跟着丙丙走进ReentrantLock的内心世界。

上面提了这么多，我想你应该是有所了解了，那一个线程进来ReentrantLock这个渣男是怎么不公平的呢？（默认是非公平锁）

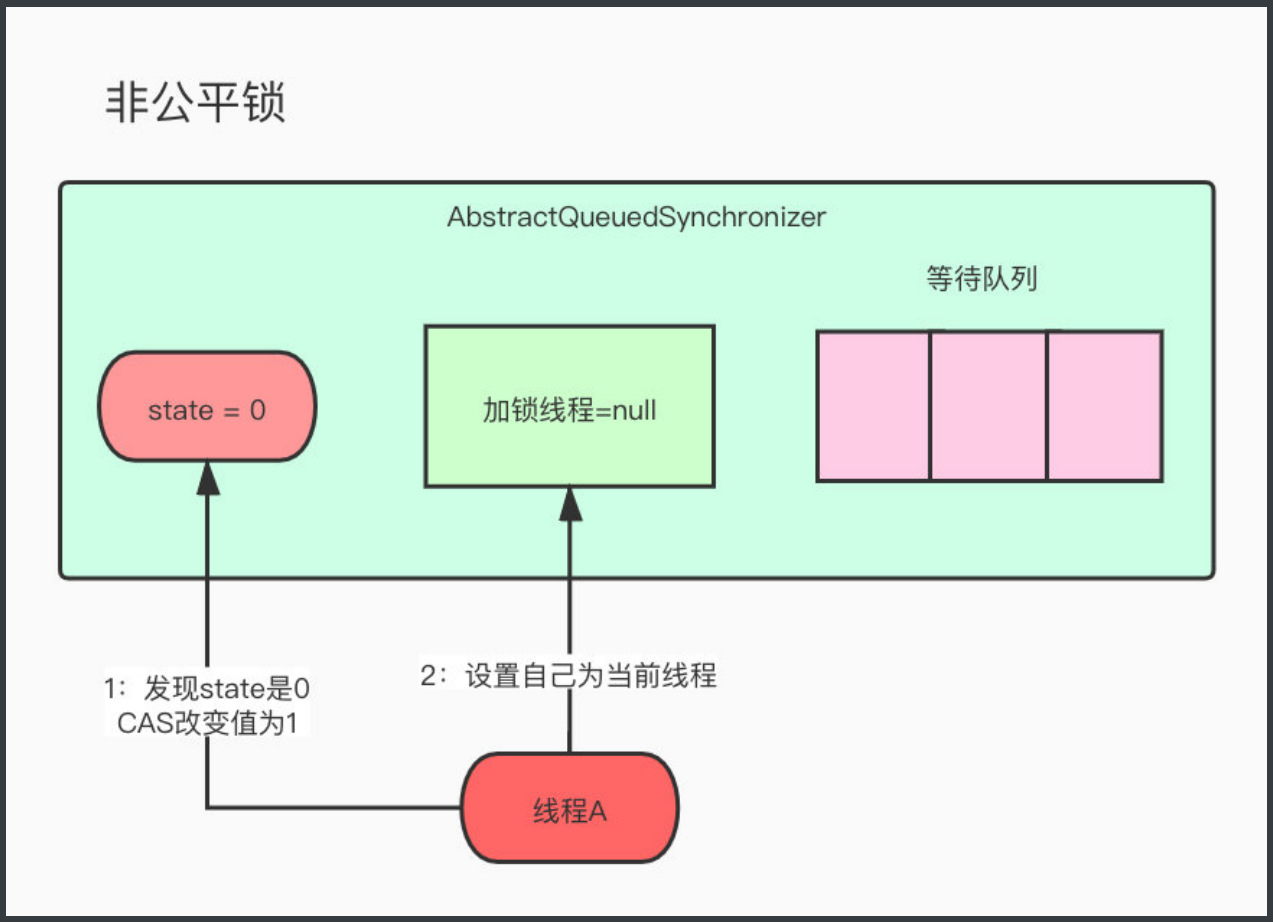
我先画个图，帮助大家了解下细节：

ReentrantLock的Sync继承了AbstractQueuedSynchronizer也就是我们常说的AQS

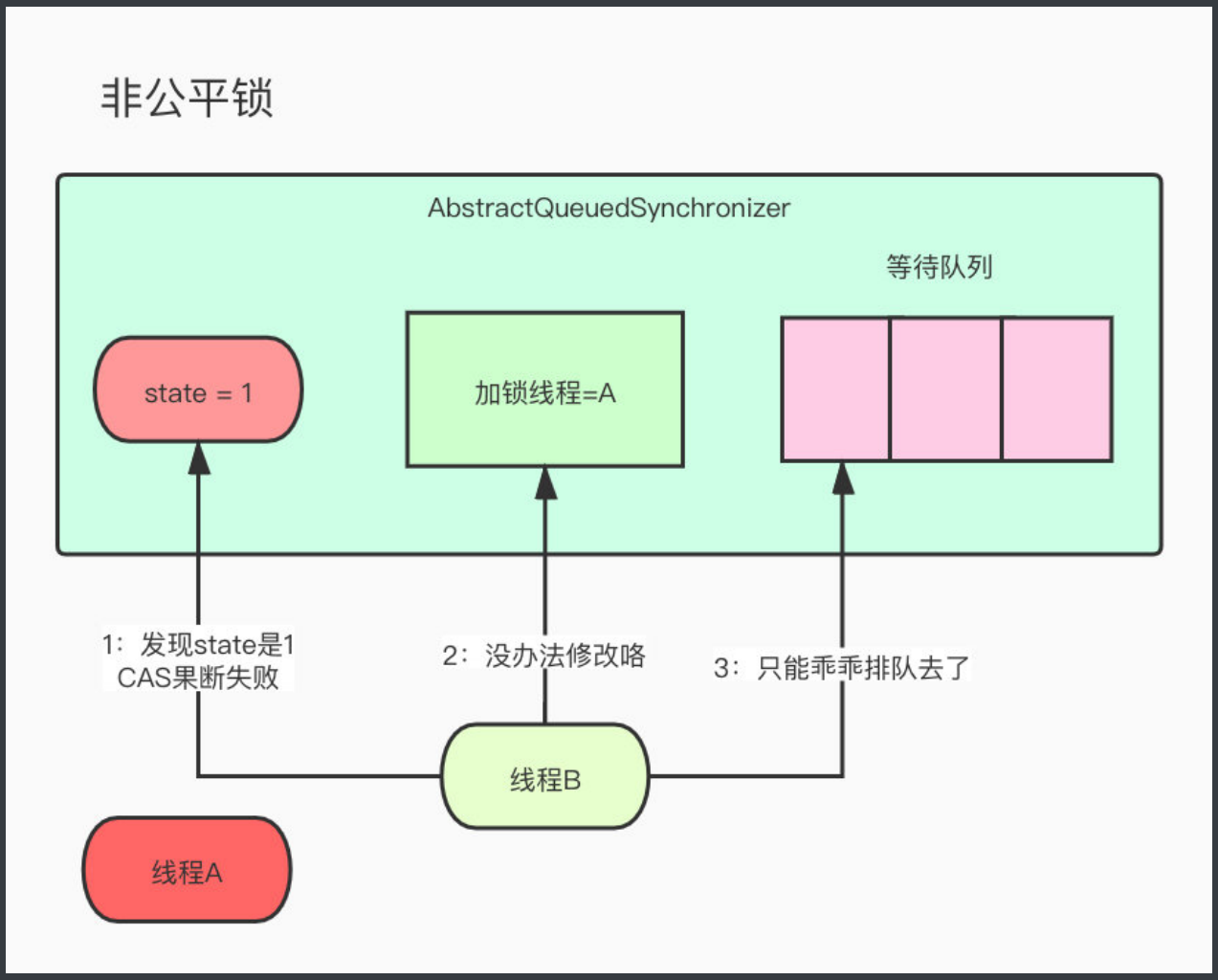


他也是ReentrantLock加锁释放锁的核心，大致的内容我之前一期提到了，我就不过多赘述了，他们看看一次加锁的过程吧。

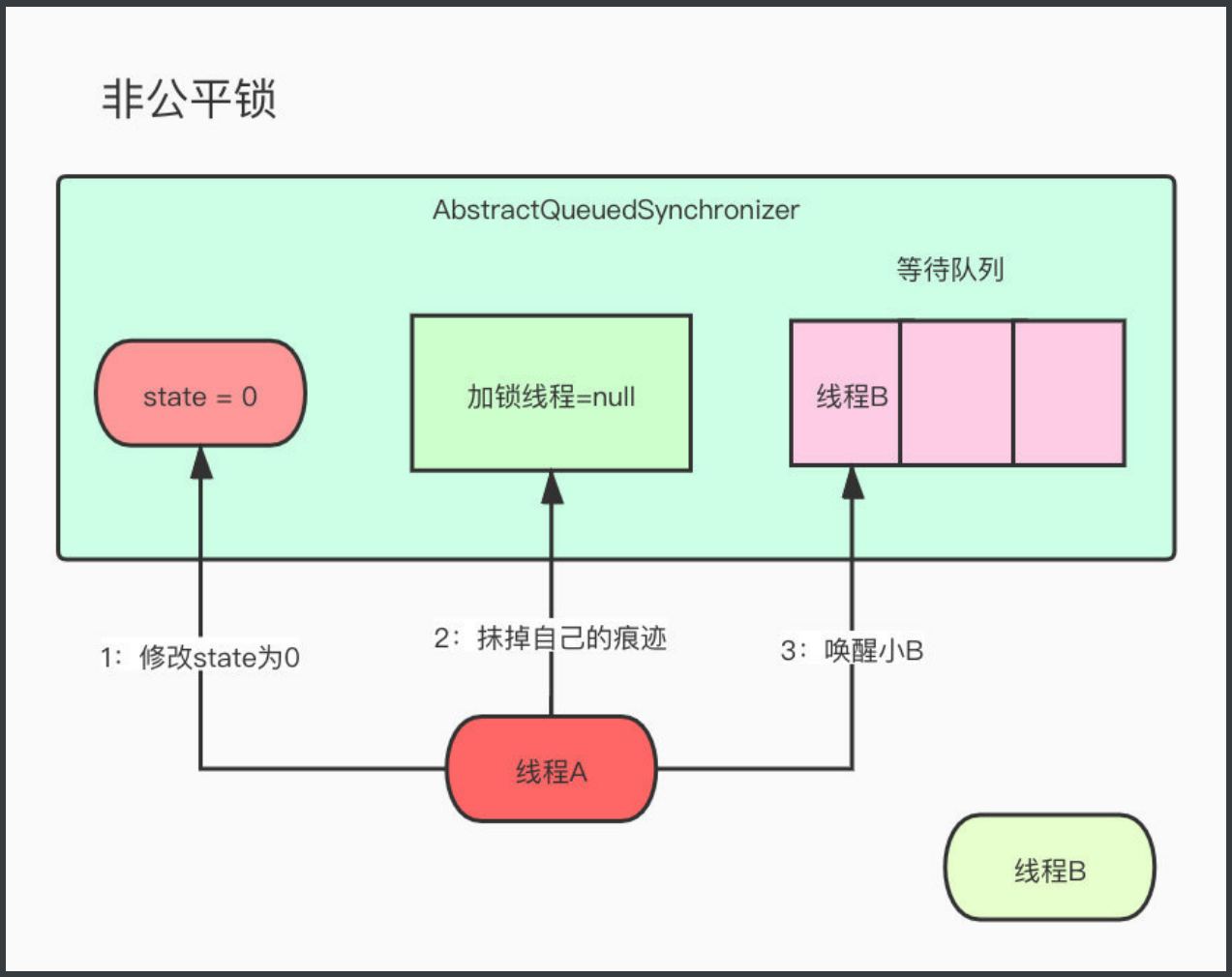
A线程准备进去获取锁，首先判断了一下state状态，发现是0，所以可以CAS成功，并且修改了当前持有锁的线程为自己。



这个时候B线程也过来了，也是一上来先去判断了一下state状态，发现是1，那就CAS失败了，真晦气，只能乖乖去等待队列，等着唤醒了，先去睡一觉吧。

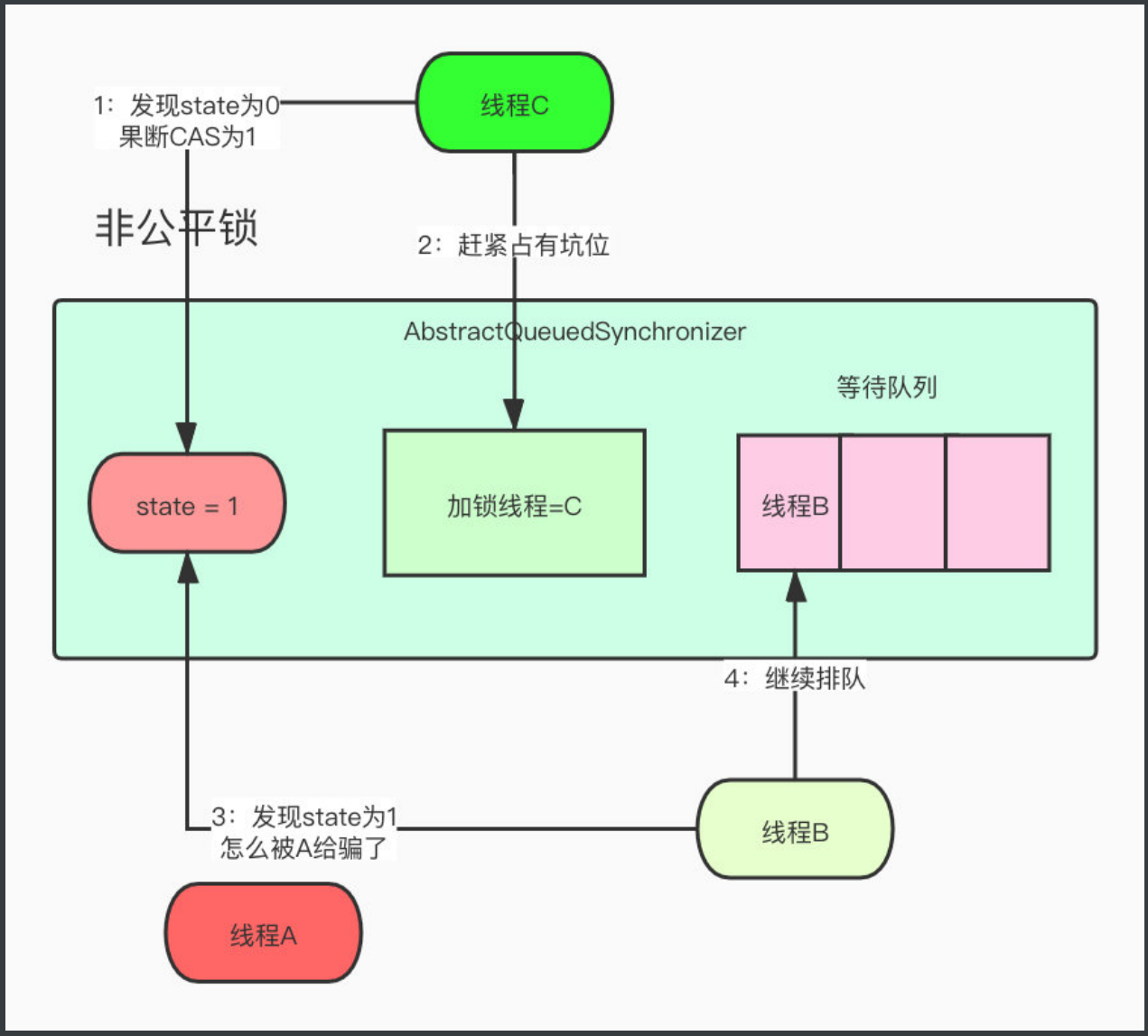


A持有久了，也有点腻了，准备释放掉锁，给别的仔一个机会，所以改了state状态，抹掉了持有锁线程的痕迹，准备去叫醒B。



这个时候有个带绿帽子的仔C过来了，发现state怎么是0啊，果断CAS修改为1，还修改了当前持有锁的线程为自己。

B线程被A叫醒准备去获取锁，发现state居然是1，CAS就失败了，只能失落的继续回去等待队列，路线还不忘骂A渣男，怎么骗自己，欺骗我的感情。



诺以上就是一个非公平锁的线程，这样的情况就有可能像B这样的线程长时间无法得到资源，优点就是可能有的线程减少了等待时间，提高了利用率。

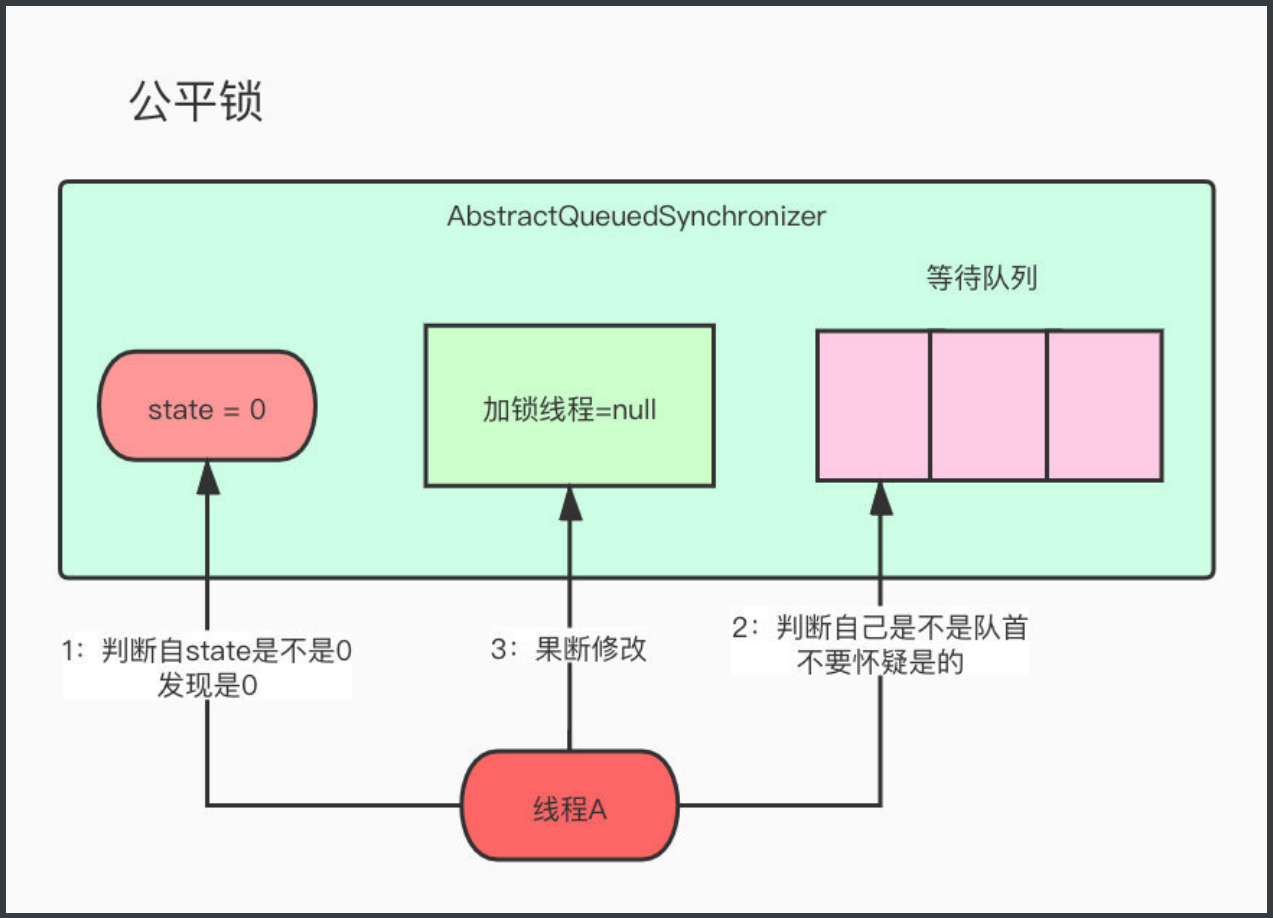
现在都是默认非公平了，想要公平就得给构造器传值true。

```
ReentrantLock lock = new ReentrantLock(true);
```

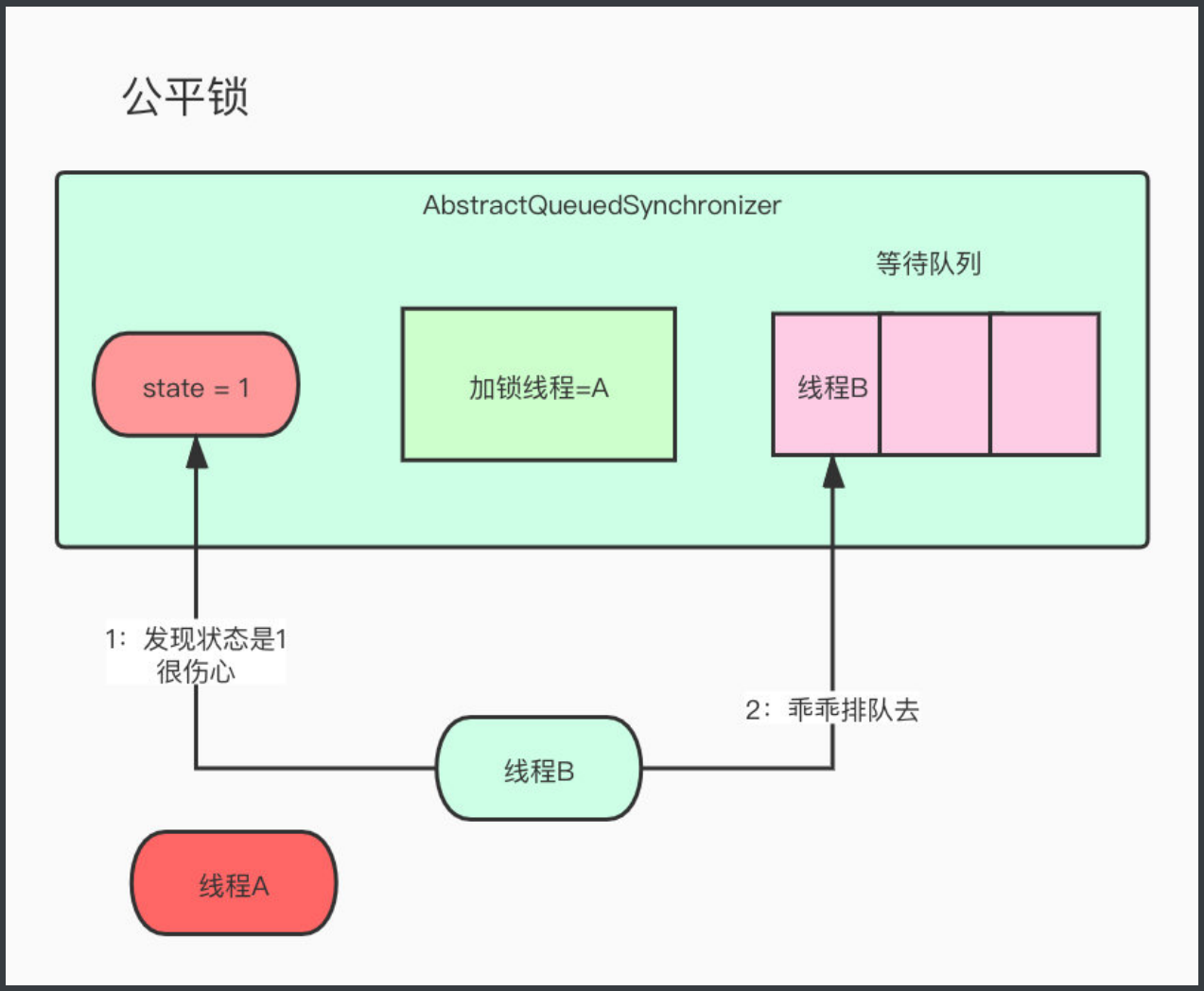
```java
/**
 * Creates an instance of {@code ReentrantLock}.
 * This is equivalent to using {@code ReentrantLock(false)}.
 */
public ReentrantLock() { sync = new NonfairSync(); }

/**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy
 */
public ReentrantLock(boolean fair) { sync = fair ? new FairSync() : new NonfairSync(); }

/**
 * Acquires the lock
```

说完非公平，那我也说一下公平的过程吧：

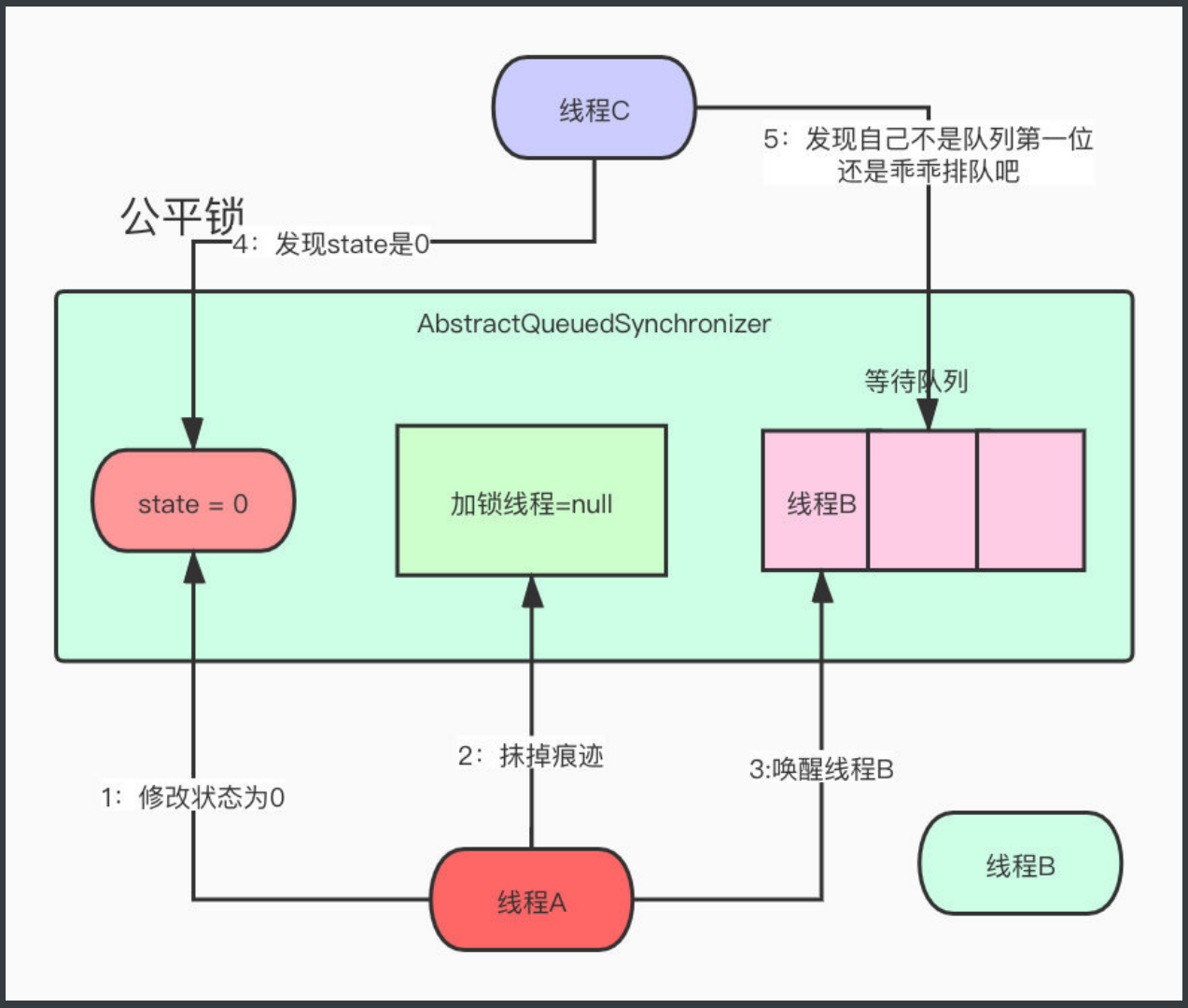
线A现在想要获得锁，先去判断下state，发现也是0，去看了看队列，自己居然是第一位，果断修改了持有线程为自己。



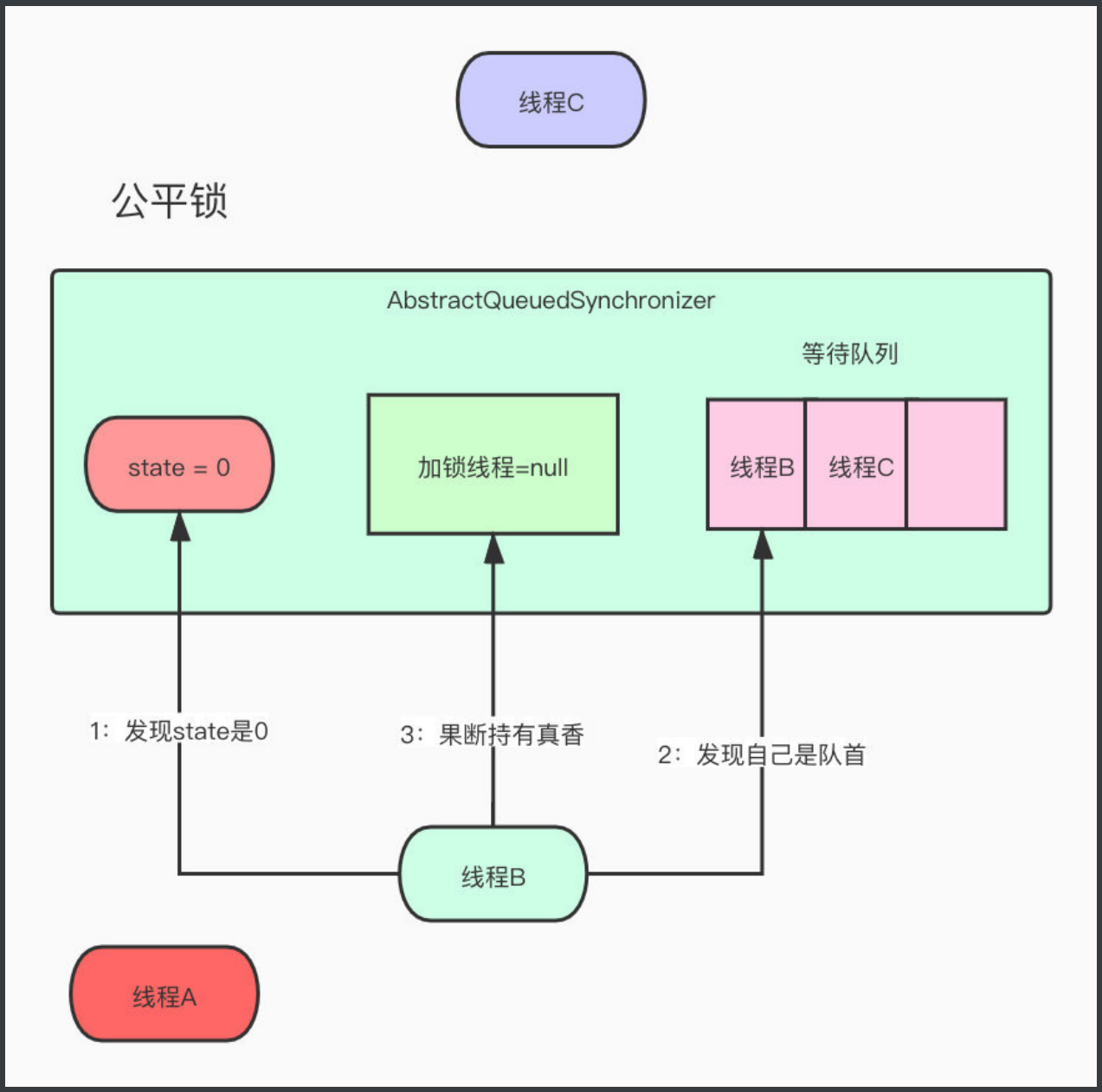
线程b过来了，去判断一下state，嗯哼？居然是state=1，那cas就失败了呀，所以只能乖乖去排队了。



线程A暖男来了，持有没多久就释放了，改掉了所有的状态就去唤醒线程B了，这个时候线程C进来了，但是他先判断了下state发现是0，以为有戏，然后去看了看队列，发现前面有人了，作为新时代的良好市民，果断排队去了。



线程B得到A的召唤，去判断state了，发现值为0，自己也是队列的第一位，那很香呀，可以得到了。



总结：

总结我不说话了，但是去获取锁判断的源码，箭头所指的位置，现在是不是都被我合理的解释了，当前线程，state，是否是0，是否是当前线程等等，都去思考下。

```java
/**
 * Fair version of tryAcquire.  Don't grant access unless
 * recursive call or no waiters or is first.
 */
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 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;
}
```

鬼知道我为了画图，画了多少费稿，点个赞过分么？

课后作业

公平锁真的公平么？那什么层面不是绝对的公平，什么层面才能算公平？

我是敖丙，一个在互联网苟且偷生的工具人。

最好的关系是互相成就，各位的**「三连」**就是丙丙创作的最大动力，我们下期见！

> 文章持续更新，可以微信搜索「 **三太子敖丙** 」第一时间阅读，回复**【资料】【面试】【简历】**有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。