链滴

详解两种 Java 锁 synchronized ReentrantLock（AQS)

作者：yscxy

原文链接：https://ld246.com/article/1621344059391

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一、synchronized

（推荐阅读 -死磕Synchronized）

（推荐阅读 -ynchronized原理底层）

1.1 三种应用形式

```
// 关键字在实例方法上，锁为当前实例
public synchronized void instanceLock() {
    // code
}

// 关键字在静态方法上，锁为当前Class对象
public static synchronized void classLock() {
    // code
}

// 关键字在代码块上，锁为括号里面的对象
public void blockLock() {
    Object o = new Object();
    synchronized (o) {
        // code
    }
}
```

1.2锁的几种状态

jdk1.6以前，synchronized是一个重量级锁

1. **无锁状态**
2. **偏向锁状态**
3. **轻量级锁状态**
4. **重量级锁状态**

1.3对象头

运行时元数据**
哈希值（HashCode），可以看作是堆中对象的地址
GC分代年龄（年龄计数器）（用于新生代from/to区晋升老年代的标准, 阈值为15）
锁状态标志 (用于JDK1.6对synchronized的优化 -> 轻量级锁)
线程持有的锁
偏向线程ID (用于JDK1.6对synchronized的优化 -> 偏向锁)
****偏向时间戳**

Java的锁都是基于对象的，锁的信息都是储存到对象头里面的（这里分为两种情况）

如果是普通对象——>2个字宽储存对象头(markWord、Klass Word)

如果是 数组类型——>3个字宽储存对象头（主要是多了一个数组的长度）(markWord、Klass Word、arrayLength)

32位处理器一个字宽32位，64位处理器，一个字宽就是64位

表：

长度	内容	说明
32/64bit	Mark Word	存储对的hashCode或锁信息等
32/64bit	Class Metadata Address	储到对象类型数据的指针
32/64bit	Array length	数组的度（如果是数组）

我们主要来看看Mark Word的格式：

锁状态	29 bit 或 61 bit	1 bit 是否是偏向锁?	bit 锁标志位
无锁		0	01
偏向锁	线程ID	1	01
轻量级锁	指向栈中锁记录的指针	此时这一位不用标识偏向锁	00
重量级锁	指向互斥量（重量级锁）的指针	此时这位不用于标识偏向锁	10
GC标记		此时这一位不用于标识偏向锁	

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS）

1

总结：当对象状态为偏向锁时，Mark Word存储的是偏向的线程ID；当状态为轻量级锁时，Mark Word存储的是指向线程栈中 Lock Record的指针；当状态为重量级锁时，Mark Word为指向堆中的monitor对象的指针。

无 锁
是否偏向锁（0）
锁标志位（01）
是否偏移？
偏向锁标志位
当有一个线程访问同步块时
升级成偏向锁
偏向锁
是否偏向锁（1）
有锁竞争时
升级为轻量级锁
锁状态
锁标志位（00）
轻量级锁
自旋十次失败（锁膨胀）
升级为重量级锁
10
锁标志位（）
重量级锁
牧小农的夏天

1.4锁的对比

偏向锁　　　　　　　　　　加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒的差距　　　　　　　　　　如果线程间存在锁竞争，会带来额外的锁撤销的消耗
用于只有一个线程访问同步块场景

轻量级锁　　　　　　　　　　竞争的线程不会阻塞，提高了程序的响应速度
果始终得不到锁竞争的线程，使用自旋会消耗CPU　　　　　　　　　　追求响应速度，同步块行速度非常快

重量级锁　　　　　　　　　　线程竞争不使用自旋，不会消耗CPU　　　　　　　　　　线阻塞，响应时间缓慢　　　　　　　　　　追求吞吐量，同步块执行速度较慢

1.5偏向锁

1.5.1、应用场景

经过大量的实验得出结论，大部分加锁的代码，很大一部分时间只有一个线程运行，所以并不是每次行该段代码都需要借助操作系统进行加锁 ， 只是设置了一个标志位，Mark Word中的线程id

1.6、轻量级锁

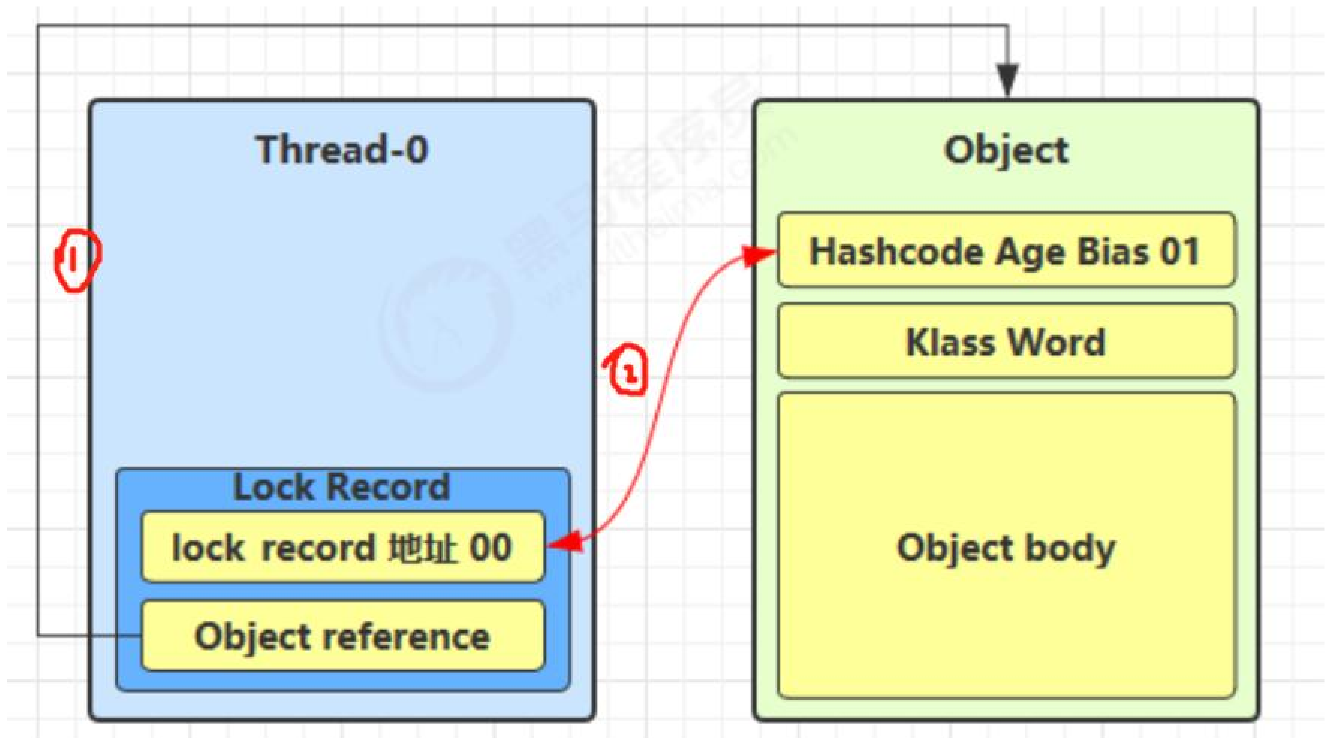
1.6.1 升级过程

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS)

当新加入的线程发现被锁的对象头里面不是自己的线程id，那么久会直接升级为轻量级锁

1.6.2 轻量级锁的具体方案

在轻量级锁状态下，当前线程会在栈帧下创建一个LockRecord ， LockRecord会把MarkWord的息拷贝进去，并且有个Owner指针指向加锁的对象（如下图
oint_down）



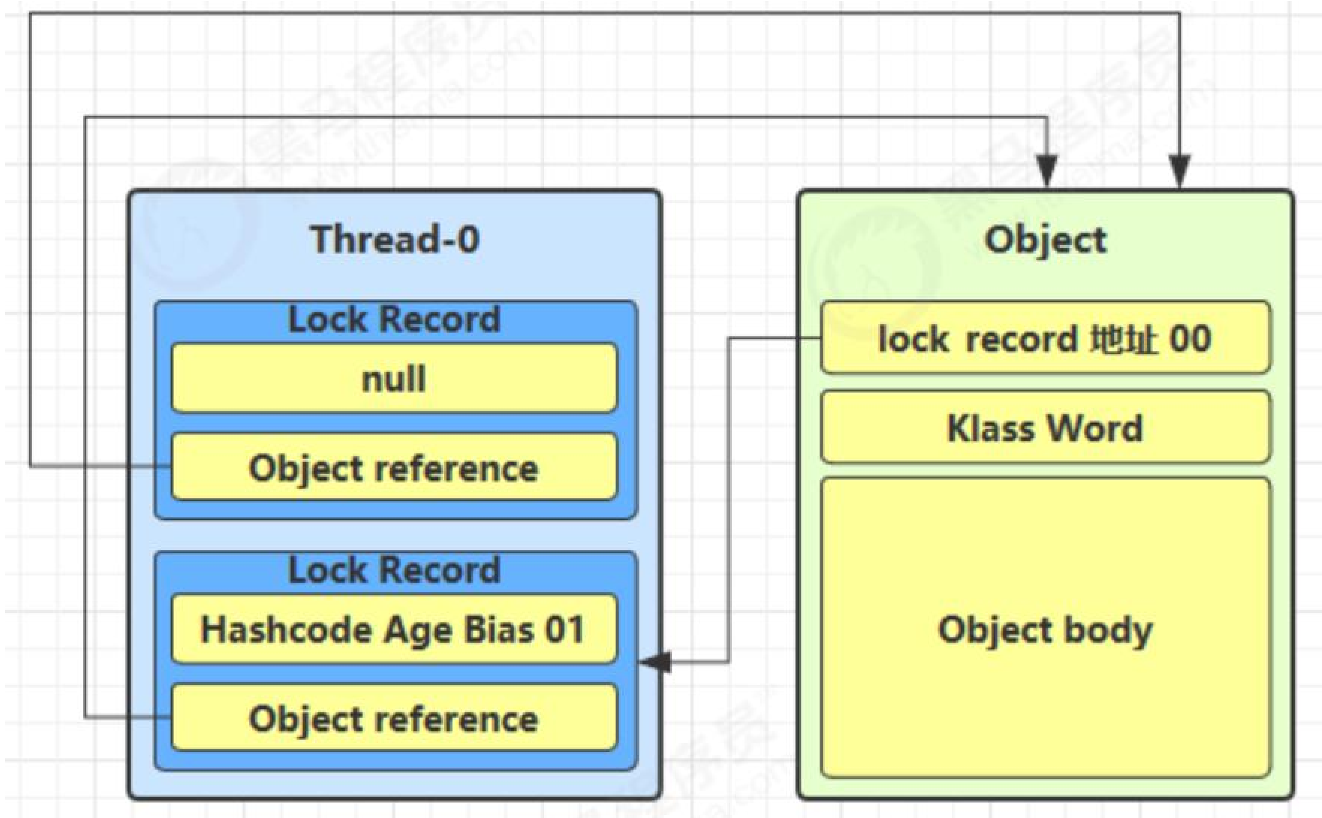
如果这个时候出现另外 线程进行竞争锁，那么会出现三种情况

情况一（其他线程来竞争，cas没有超过10次获取到锁）

情况二（其他线程来竞争，cas超过10次没有获取到锁——此时升级为重量级锁）

情况三（该线程本身来再次此执行Synchronized代码块）那么会再添加一条LockRecord作为重入的计数器（每次获得锁都添加一个 **Lock Record**来表示锁的重入如下图po
nt_down）

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS)



1.6.3、解锁过程

当 线程退出synchronized代码块的时候，如果获取的是 取值为 null 的锁记录，表示有锁重入，这时置锁记录，表示重入计数减一

轻量级锁解锁时，会使用CAS将之前复制在栈桢中的 Displaced Mard Word 替换回 Mark Word 。如果替换成功，则说明整个过程都成功执行，期间没有其他线程访问同步代码块。

但如果替换失败了，表示当前线程在执行同步代码块期间，有其他线程也在访问，当前锁资源是存在争的，那么锁将会膨胀成重量级锁

1.5重量级锁

它加锁就是依赖底层操作系统的 **mutex**相关指令实现, 所以会造成 **用户态内核态之间的切换**, 非常耗性能！

1.5.1 触发情况

当对象的锁为轻量级锁的时候，出现锁的竞争并且多次CAS尝试获取锁失败

1、当Thread2访问到synchronized(obj)中的共享资源的时候

首先会将synchronized中的锁对象中对象头的MarkWord去尝试指向操作系统的Monitor对象（每个对象对应的都有一个）. 让锁对象中的MarkWord和Monitor对象相关联. 如果关联成功, 将obj对头中的MarkWord的对象状态从01改为10。**

**因为Monitor没有和其他的obj的MarkWord相关联, 所以Thread2就成为了该Monitor的Owner所有者)。

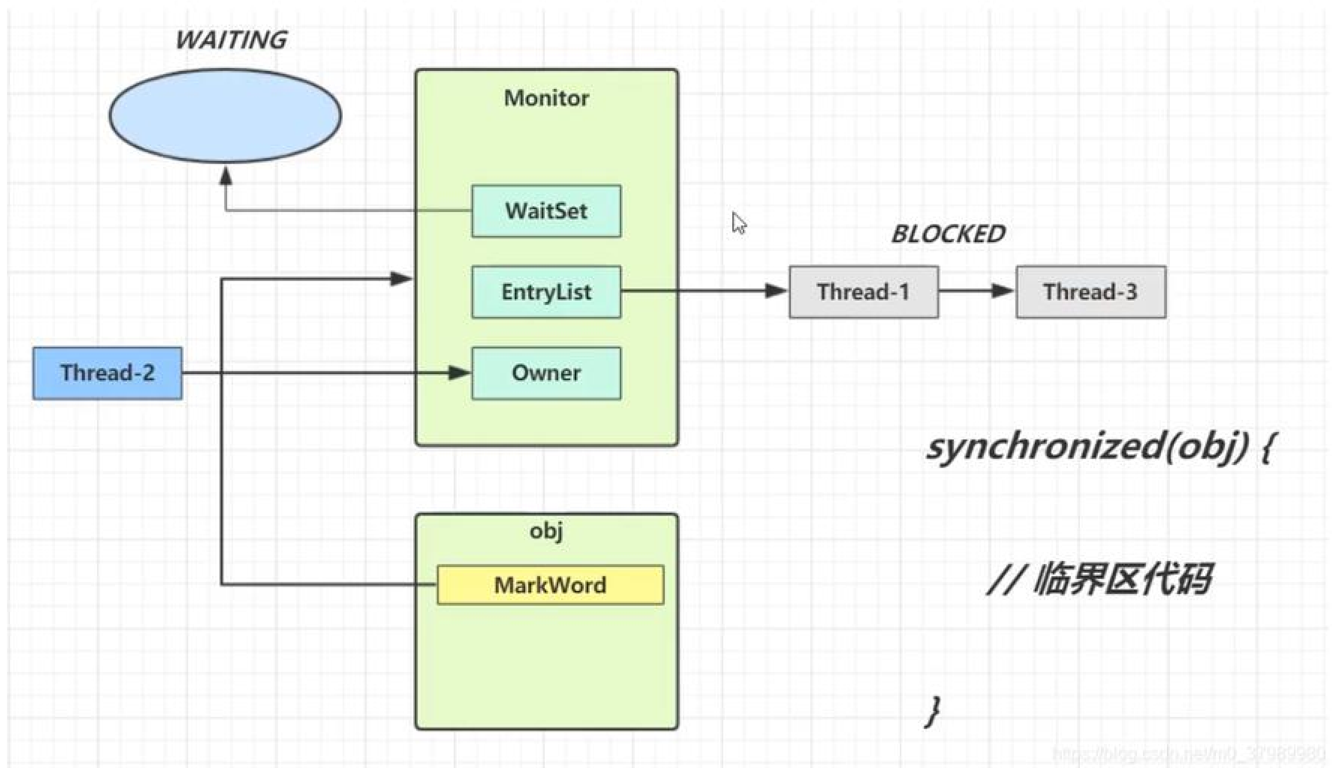
原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS)

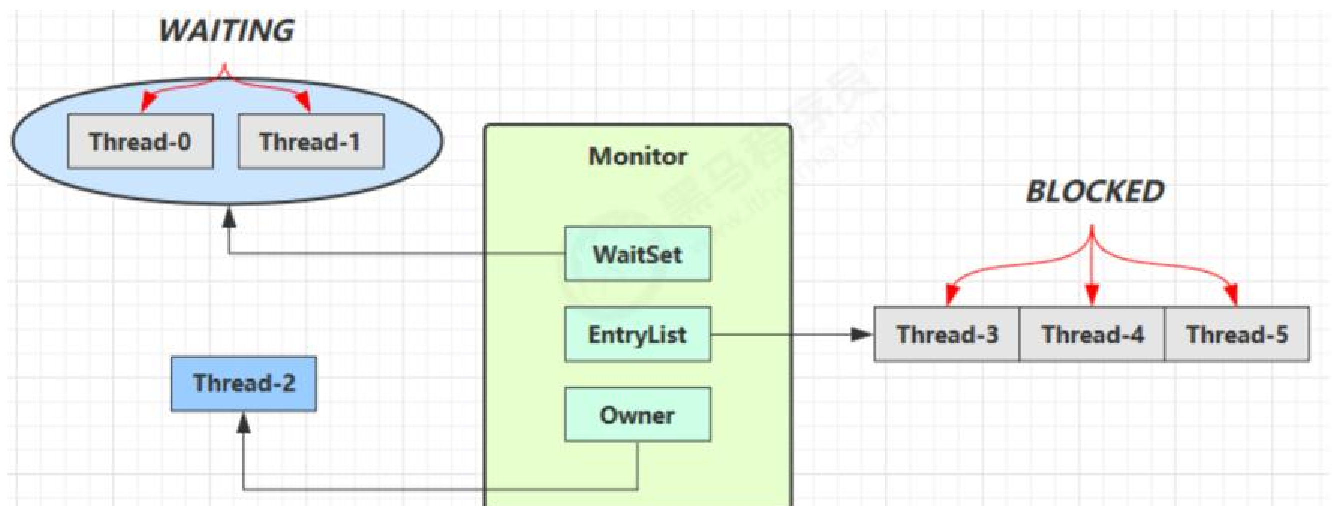
**2、又来了个Thread1执行synchronized(obj)代码, **

它首先会看看能不能执行该临界区的代码; 它会检查obj是否关联了Montior, 此时已经有关联了, 它会去看看该Montior有没有所有者(Owner), 发现有所有者了(Thread2); Thread1也会和该Monito关联, 该线程就会进入到它的EntryList(阻塞队列);

**3、当Thread2执行完临界区代码后, **

Monitor的Owner(所有者)就空出来了. 此时就会通知Monitor中的EntryList阻塞队列中的线程, 这线程通过竞争, 成为新的所有者





图中 WaitSet 中的Thread-0，Thread-1 是之前获得过锁，但条件不满足进入 WAITING 状态的线

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS）

，后面讲wait-notify 时会分析

二、乐观锁-悲观锁

乐观锁：

乐观锁又称为“无锁”，顾名思义，它是乐观派。乐观锁总是假设对共享资源的访问没有冲突，线程以不停地执行，无需加锁也无需等待。而一旦多个线程发生冲突，乐观锁通常是使用一种称为CAS的术来保证线程执行的安全性。

悲观锁：

悲观锁就是我们常说的锁。对于悲观锁来说，它总是认为每次访问共享资源时会发生冲突，所以必须每次数据操作加上锁，以保证临界区的程序同一时间只能有一个线程在执行。

三、什么是CAS（Compare And Swap）

推荐阅读——详解CAS

CAS机制中使用了三个基本操作数：内存地址V，旧的预期值 A，需要修改的新值B

更新一个变量的时候只有当内存值V中的实际值 和 变量的预期值A相同时，才会将内存地址V对应的修改为B

```
package net.yscyx.lk_3.syn_lock;

import java.util.concurrent.atomic.AtomicInteger;

public class CAS {
    static int a = 0;
    public static void main(String[] args) throws InterruptedException {
        testUnSafe();
        testAtomic();
    }

    //线程不安全的情况
    public static void testUnSafe() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) {
                a++;
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) {
                a--;
            }
        });
        t1.start();
        t2.start();
        //让父线程等待子线程结束之后才能继续运行。
        t1.join();
        t2.join();
        System.out.println(a);
```

```
        System.out.println(t2.isAlive());
        System.out.println(t1.isAlive());
    }

    //原子类
    public static void testAtomic() throws InterruptedException {
        AtomicInteger atomicInteger = new AtomicInteger();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) {
                atomicInteger.incrementAndGet();
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 100000; i++) {
                atomicInteger.decrementAndGet();
            }
        });
        t1.start();
        t2.start();
        System.out.println(t2.isAlive());
        System.out.println(t1.isAlive());
        System.out.println(atomicInteger.get());
    }
}
```

三、ReentrantLock

Java 除了使用Synchronized外，还可以使用ReentrantLock实现独占锁功能，并且ReentrantLoc相比于Synchronized而言功能更加丰富，且使用起来更为灵活，也更适合复杂的并发场所。

3.1、ReentrantLock和 Synchronized对比

- **两者都是独占锁，但是Syn加锁解锁是自动，Reen加锁解锁需要手动不易操作，但是灵活**
- **两者都可重入Syn假锁解锁都是自动不必担心最后是否释放锁了；Reen加锁解锁次数需要相同**
- **Syn不可响应中断，一个线程获取不到就需要一直等着，Reen可以响应中断**
- **Syn是非公平锁，Reen可以设置是否为公平锁，默认为非公平**

3.2、公平锁——非公平锁

公平锁：谁等的时间长那么久谁先获取锁

非公平锁：非公平锁那就随机的获取，谁运气好，cpu时间片轮到哪个线程

3.3、响应中断

含义：一个线程获取不到锁，不会一直 等下去。ReentrantLock会给与一个中断回应

3.4、interrupt、interrupted 、isInterrupted

1. **interrupt** ：中断此线程，此线程不一定是当前线程，儿是值调用该方法的Thread实例所代

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS)

的线程例如point_down**

```
public static void main(String[] args) {
      Thread thread1 = new Thread(()->>{
          ....
      });
      thread1.start();
      thread1.interrupt();
  }
```

2. **interrupted**** :测试当前线程是否被中断 ， 返回一个boolean并清除中断状态，第二次再次调的时候中断状态已经被清除，返回一个false。**

3. **isinterrupted**** ：只测试此线程是否被中断，不清除中断状态。**

3.5、lock、tryLock与lockInterruptibly的区别

**lock **

优先考虑获取锁，待获取锁成功后，才响应中断。

**lockInterruptibly **

优先考虑响应中断，而不是响应锁的普通获取或重入获取。

详细区别： **

****ReentrantLock.lockInterruptibly允许在等待时由其它线程调用等待线程的Thread.interrupt方来中断等待线程的等待而直接返回，这时不用获取锁，而会抛出一个InterruptedException。 ReentrantLock.lock方法不允许Thread.interrupt中断,即使检测到Thread.isInterrupted,一样会继续尝获取锁，失败则继续休眠。只是在最后获取锁成功后再把当前线程置为interrupted状态,然后再中断程。**

tryLock

分三种情况进行讨论

- **获取到锁**
- **在指定时间内获取到锁**
- **在指定时间内没有获取到锁**

3.6、打扰机制

1. **线程在sleep或wait,join， 此时如果别的进程调用此进程的 interrupt（）方法，此线程会被唤醒被要求处理InterruptedException；(thread在做IO操作时也可能有类似行为，见java thread api)**

2. **此线程在运行中， 则不会收到提醒。但是 此线程的 “打扰标志”会被设置， 可以通过isInterruptd()查看并 作出处理。**

lockInterruptibly()和上面的第一种情况是一样的， 线程在请求lock并被阻塞时，如果被interrupt则“此线程会被唤醒并被要求处理InterruptedException”。并且如果线程已经被interrupt，再使lockInterruptibly的时候，此线程也会被要求处理interruptedException

3.7、线程之间的通信

wait、notifyAll、notify

3.8、生产者/消费者模式

```
package net.yscyx.lk_3.syn_lock;

import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
 * 多生产者、多消费者
 */
public class ProducerConsumer {
    //定义一个队列缓冲区，数据为Integer
    private final Queue<Integer> queue = new LinkedList<>();

    //设置缓冲区最大容量
    private static final int MAX_SIZE = 100;

    /**
     * 生产者。
     *
     * <p>生产者进行V原语操作</p>
     * <ul>
     * <li>如果缓冲区没有达到MAX_SIZE，则生产一个产品（n个也行）放入缓冲区，并唤醒所有线
</li>
     * <li>否则使自己进入缓冲区的等待池</li>
     * </ul>
     */
    class Producer implements Runnable {
        @Override
        public void run() {
            while (true) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (queue) {
                    if (queue.size() < MAX_SIZE) {
                        int num = new Random().nextInt(100);
                        queue.offer(num);
                        queue.notifyAll();
                        System.out.println("生产者" + Thread.currentThread().getName() + "生产了产
：" + num + "，此时缓冲区数据量为：" + queue.size());
                    } else {
                        try {
                            queue.wait();
                        } catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

/**
 * 消费者。
 * <p>消费者进行P原语操作</p>
 * <ul>
 * <li>如果缓冲区有数据，则从缓冲区取出一个产品（n个也行），并唤醒所有线程</li>
 * <li>否则使自己进入缓冲区的等待池</li>
 * </ul>
 */
class Consumer implements Runnable {
    @Override
    public void run() {
        while (true) {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (queue) {
                if (queue.size() > 0) {
                    int num = queue.poll();
                    System.out.println("消费者" + Thread.currentThread().getName() + "消费了产：" + num + "，此时缓冲区数据量为：" + queue.size());
                    queue.notifyAll();
                } else {
                    try {
                        queue.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}


public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();
    //Thread构造函数需要一个Runnable对象即可构造一个新的线程，Runnable对象可以重复利，不必new多个
    //一个消费者，一个生产者
    Consumer c = pc.new Consumer();
    Producer p = pc.new Producer();
    //生产者和消费者谁先start都一样
    new Thread(c).start();
    new Thread(p).start();
```

```
    }
}
```

3.8详解AQS

（推荐阅读——ReentrantLock原理）

3.8.1、两种构造器

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

默认构造器初始化为NonfairSync对象，即非公平锁，而带参数的构造器可以指定使用公平锁和非公锁。由lock()和unlock的源码可以看到，它们只是分别调用了sync对象的 lock()和release(1)方法。

3.8.2、NonfairSync**

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

首先用一个CAS操作，判断state是否是0（表示当前锁未被占用），如果是0则把它置为1，并且设置前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS操作只能保一个线程操作成功，剩下的只能乖乖的去排队啦。

** “非公平”即体现在这里，如果占用锁的线程刚释放锁，state置为0，而排队等待锁的线程还未醒时，新来的线程就直接抢占了该锁，那么就“插队”了。**

** 若当前有三个线程去竞争锁，假设线程A的CAS操作成功了，拿到了锁开开心心的返回了，那么线B和C则设置state失败，走到了else里面。我们往下看acquire。**

3.8.3AQS 队列几个重要的属性

```
private transient volatile Node head;//队列首
private transient volatile Node tail;//队列尾
private volatile int state;//锁状态，加索成功1，解锁成功0，重入+1
private transient Thread exclusiveOwnerThread;//持有锁的那个线程
```

3.8.4、Node对象属性和信息

```
static final class Node {
    static final Node SHARED = new Node();
```

```
    static final Node EXCLUSIVE = null;
    static final int CANCELLED =  1;
    static final int SIGNAL    = -1;
    static final int CONDITION = -2;
    static final int PROPAGATE = -3;
    volatile int waitStatus;
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;
    Node nextWaiter;
    final boolean isShared() {
        return nextWaiter == SHARED;
    }
    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
        else
            return p;
    }
    Node() {    // Used to establish initial head or SHARED marker
    }
    Node(Thread thread, Node mode) {     // Used by addWaiter
        this.nextWaiter = mode;
        this.thread = thread;
    }
    Node(Thread thread, int waitStatus) { // Used by Condition
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}
```

AQS队列种队列头种 Thread 为空

3.8.5、ReentrantLock的方法

```
private static final long serialVersionUID = 7373984872572414699L;
//属性变量
private final Sync sync;
// 静态内部类
abstract static class Sync extends AbstractQueuedSynchronizer
//非公平锁-继承上面的Sync
static final class NonfairSync extends Sync
//公平锁-继承上面的Sync
static final class FairSync extends Sync
// 构造方法
public ReentrantLock()
public ReentrantLock(boolean fair)
// 方法
public void lock()
public void unlock()
public boolean isLocked()
//获取不到锁就直接返回
public boolean tryLock()
```

```
public boolean tryLock(long timeout, TimeUnit unit)
//可以进行中断响应的lock方法
public void lockInterruptibly()
public Condition newCondition()
public int getHoldCount()
public boolean isHeldByCurrentThread()
public final boolean isFair()
protected Thread getOwner()
public final boolean hasQueuedThreads()
public final boolean hasQueuedThread(Thread thread)
public final int getQueueLength()
protected Collection<Thread> getQueuedThreads()
public boolean hasWaiters(Condition condition)
public int getWaitQueueLength(Condition condition)
protected Collection<Thread> getWaitingThreads(Condition condition)
public String toString()
```

3.8.6 ReentrantLock加锁的具体过程

3.8.6.1首先讲一下类的结构

lock对象有一个sync属性，我们创建对象的方式有两种 NonfairSync 、FairSync 非公平锁和公平锁

他们两个又都继承Sync 而sync又继承AbstractQueuedSynchronizer，所以说讲ReentrantLock实就是讲AQS

也就是，lock里面有一个抽象队列同步器、队列里面几个比较重要的属性 head、tail、state、exclusveOwnerThread 且里面有个静态内部类Node也就是线程节点，里面有waitStatus、prev、next、hread

3.8.6.2 然后我们来讲一下加锁的过程

**当我们调用lock方法的时候，其实执行的是 **acquire(1);

```
public final void acquire(int arg) {

    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

1、tryAcquire(arg) 有两种执行情况

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    //如果当前没有线程持有锁
    if (c == 0) {
        //是否需要排队
        if (!hasQueuedPredecessors() &&
            //cas尝试获取锁
            compareAndSetState(0, acquires)) {
            //设置持有锁的为当前线程
```

```
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //判断持有锁的线程是否为当前线程，如果是state ++ （可重入锁）
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

** **1.1、会先去判断当前锁是否被占用如果没有被占用 也就是state==0**

** **1.2、判断当前线程和持有锁的线程是否一致**

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

这时候又来了一个线程，或者第一个线程tryAcquire失败，那么将执行 addWaiter(Node.EXCLUSIE) 节点关系

** 然后执行 acquireQueued 、尝试添加到队列**

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            //上一个节点 p
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

如果当前线程的pre节点为首节点的话，会进行一次自旋（再次调用 tryAcquire（））如果获取 锁功，设置当前 节点为头节点，并且将原来头节点的next置空，帮助GC，置空当前节点的返回false，法结束。

如果当前线程的pre节点不是头节点，或者是头节点，但是竞争锁失败，那么会执行shouldParkAfteFailedAcquire(p, node)（竞争锁失败后是否需要Park）如果是，那么就进行park如果不需要，继循环。

3.8.6.2 再讲一下3种加锁方法

Synchronized和ReentrantLock的区别

- **synchronized是 JVM 直接支持的，ReentrantLock是java类**
- **使用方式不同，synchronized是隐式加锁和释放锁，ReentrantLock是通过调用Java API使用的**
- **等待是否可中断，Synchronized不可中断，ReentrantLock可以中断**
- **Synchronized非公平锁，ReentrantLock可以构造公平锁和非公平锁**
- **Synchronized不能指定唤醒线程，ReentrantLock有多个条件队列，能指定唤醒线程**
- **在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维常态**

原文链接：详解两种 Java 锁 synchronized ReentrantLock（AQS）