锁的概念和synchronized关键字

Java中锁的概念

1. 自旋锁：为了不放弃CPU执行时间，循环的使用CAS技术对数据尝试进行更新，直至成功。
2. 悲观锁：假定会发生并发冲突，同步所有对数据的相关操作，从读数据就开始上锁。
3. 乐观锁：假定没有冲突，在修改数据时如果发现数据和之前获取的不一致，则读取最新数据，修改后重试更新。
4. 独享锁(写)：给资源加上锁，线程可以修改资源，其他线程不能再加锁(单写)。
5. 共享锁(读)：给资源加上读锁后只能读不能改，其他线程也只能加读锁，不能加写锁(多读)。
6. 可重入锁，不可重入锁：线程拿到一把锁后，可以自由进入同一把锁同步的其他代码。
7. 公平锁，非公平锁：争抢锁的顺序，如果是按先来后到，则为公平锁。

同步关键字synchronized

1. 属于最基本的线程通信机制，基于对象监视器实现的。Java中的每个对象都与一个监视器相关联，一个线程可以锁定或解锁。一次只能有一个线程可以锁定监视器。试图锁定该监视器的任何其他线程都会被阻塞，直到它们可以获得该监视器上的锁定为止。
2. 特点：可重入，独享，悲观锁。
3. 锁的范围：类锁，对象锁，锁消除，锁粗化
    - 类锁，class对象，静态方法

```
public class ObjectSyncDemo1 {

    static Object temp = new Object();

    public void test1() {
        synchronized (ObjectSyncDemo1.class) {
            try {
                System.out.println(Thread.currentThread() + " 我开始执行");
                Thread.sleep(3000L);
                System.out.println(Thread.currentThread() + " 我执行结束");
            } catch (InterruptedException e) {
            }

        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
            new ObjectSyncDemo1().test1();
        }).start();

        Thread.sleep(1000L); // 等1秒钟,让前一个线程启动起来
        new Thread(() -> {
            new ObjectSyncDemo1().test1();
        }).start();
```

```
    }
}
```

- 对象锁，可重入，this，普通方法

```
public class ObjectSyncDemo2 {

    public synchronized void test1(Object arg) {
        System.out.println(Thread.currentThread() + " 我开始执行 " + arg);
        if (arg == null) {
            test1(new Object());
        }
        System.out.println(Thread.currentThread() + " 我执行结束" + arg);
    }

    public static void main(String[] args) throws InterruptedException {
        new ObjectSyncDemo2().test1(null);
    }
}
```

- 锁粗化

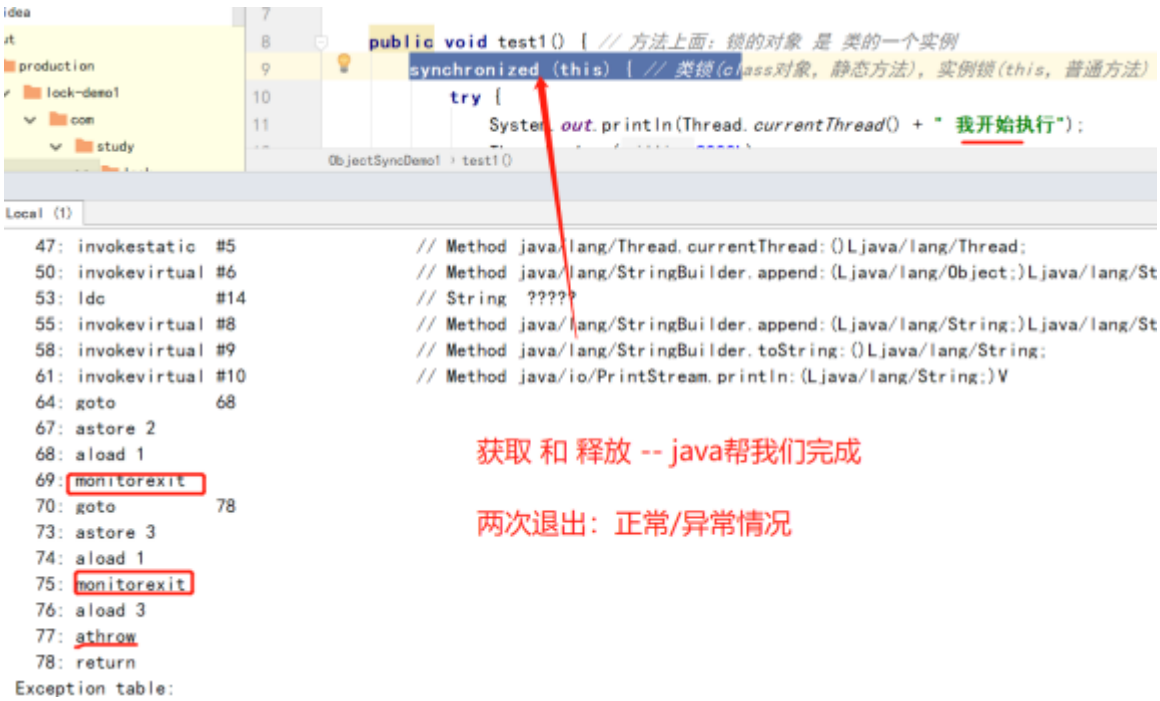
```
public class ObjectSyncDemo3 {
    int i;

    public void test1(Object arg) {
        synchronized (this) {
            i++;
        }
        synchronized (this) {
            i++;
        }

        // 会被优化
//        synchronized (this) {
//            i++;
//            i++;
//        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 10000000; i++) {
            new ObjectSyncDemo3().test1("a");
        }
    }
}
```

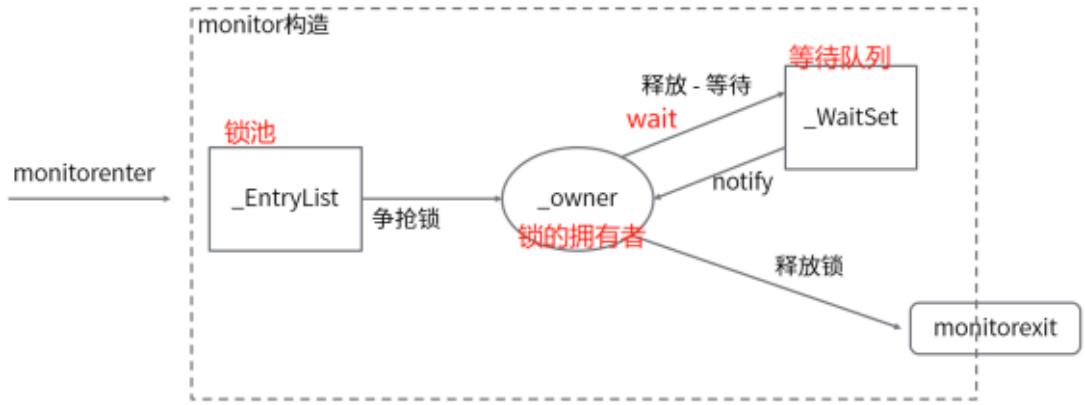
- 锁消除

```
public class ObjectSyncDemo4 {
    public void test1(Object arg) {
```

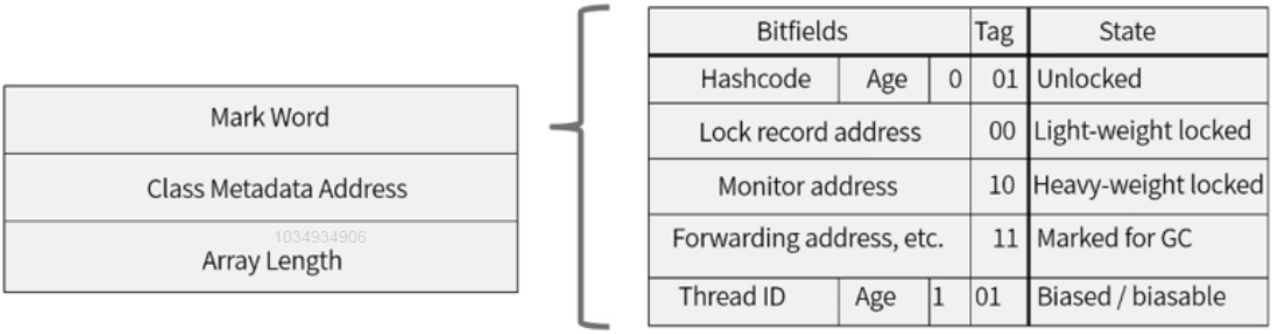
```
        // jit 优化，消除了锁，StringBuffer中的同步锁会被消除
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append("a");
        stringBuffer.append(arg);
        stringBuffer.append("c");
        // System.out.println(stringBuffer.toString());
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000000; i++) {
            new ObjectSyncDemo4().test1("123");
        }
    }
}
```

4. 如何实现

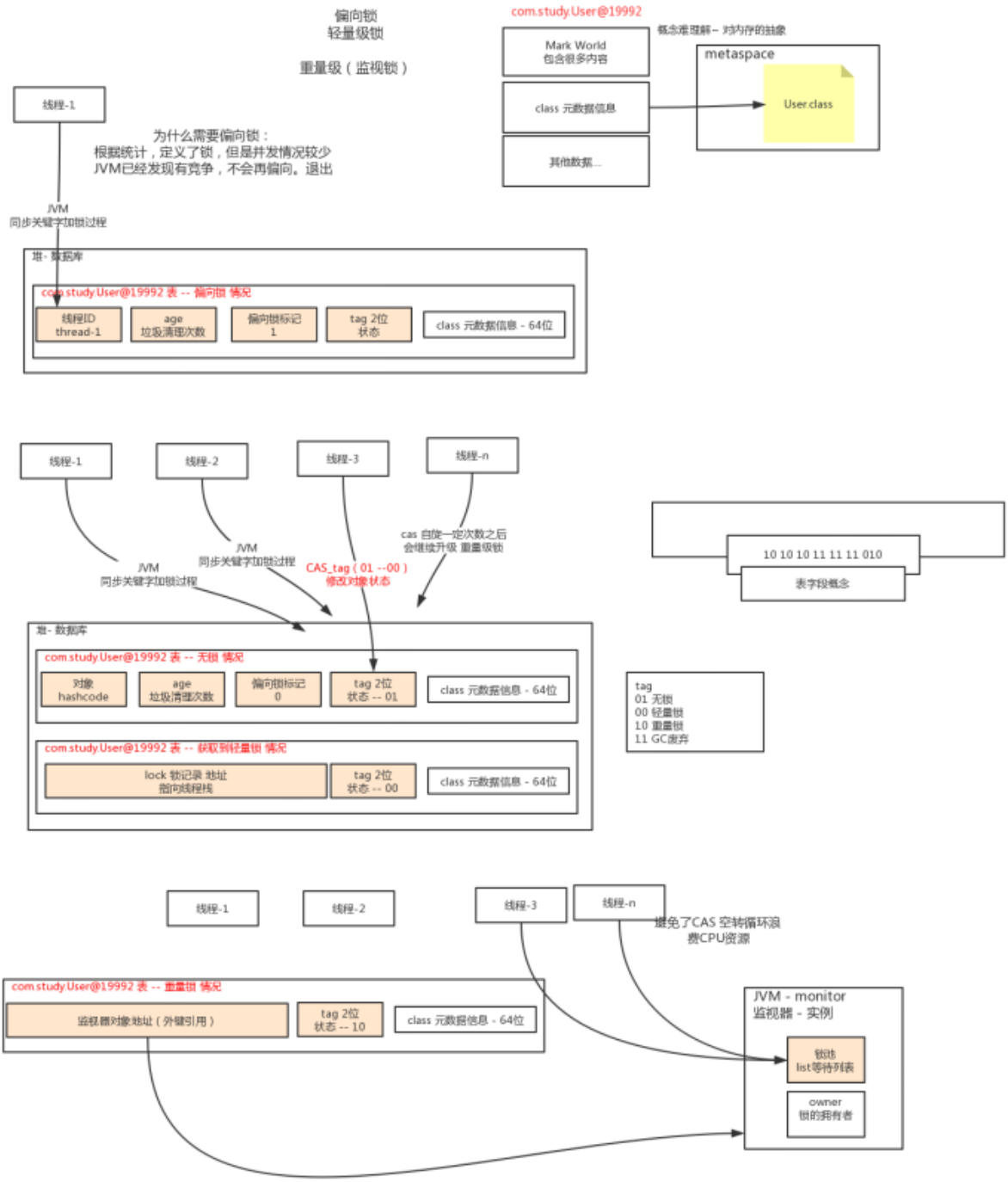




同步关键字加锁原理



1. HotSpot中，对象前面会有一个人类指针和标题，存储哈希码的标题字以及用于分代垃圾回收的年龄和标记位，默认情况下JVM锁会经历：偏向锁-》轻量级锁-》重量级锁这三个状态。
2. 锁升级的过程

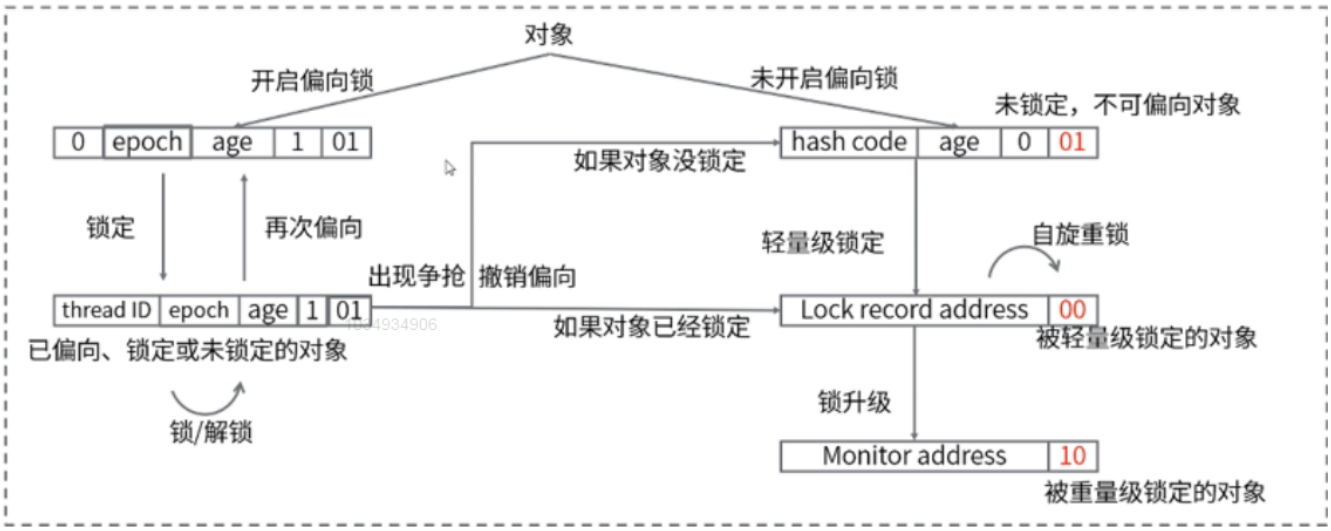


、

3. 同步关键字，优化内容

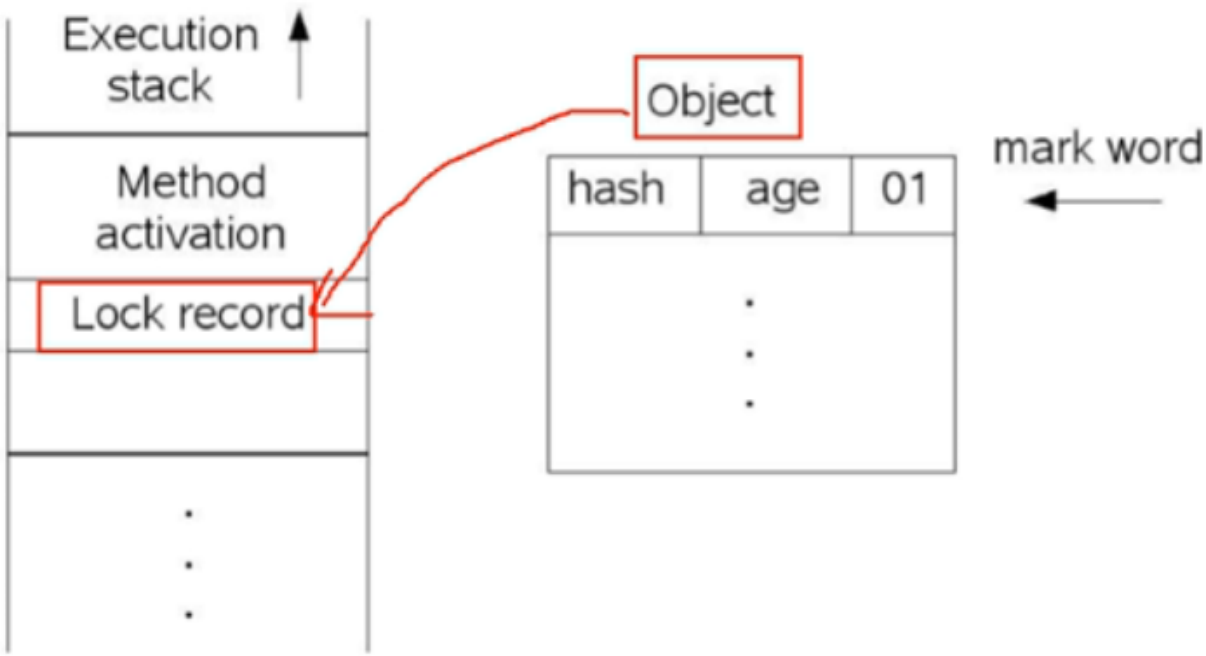
- 偏向锁，减少在无竞争情况下，JVM资源消耗。
- 出现两个及以上线程争抢->轻量级锁，CAS修改状态。
- 线程CAS自旋一定次数以后，升级为重量级锁，对象的mark work内部会保存一个监视器锁的地址。mark word里面包含四种状态：01无锁，00轻量级锁，10重量级锁，11等待垃圾回收。

偏向锁到轻量级锁



1. 偏向锁第一次有用，出现过争用后就没用了。-XX：-UseBiasedLocking禁止使用偏置锁定。偏向锁本质就是无锁，如果没有发生过任何多线程争抢锁的情况，JVM就认为是单线程，无需做同步，JVM为了少干活，同步在JVM底层是由很多操作来实现的，如果是没有争用，就不需要去做同步操作。

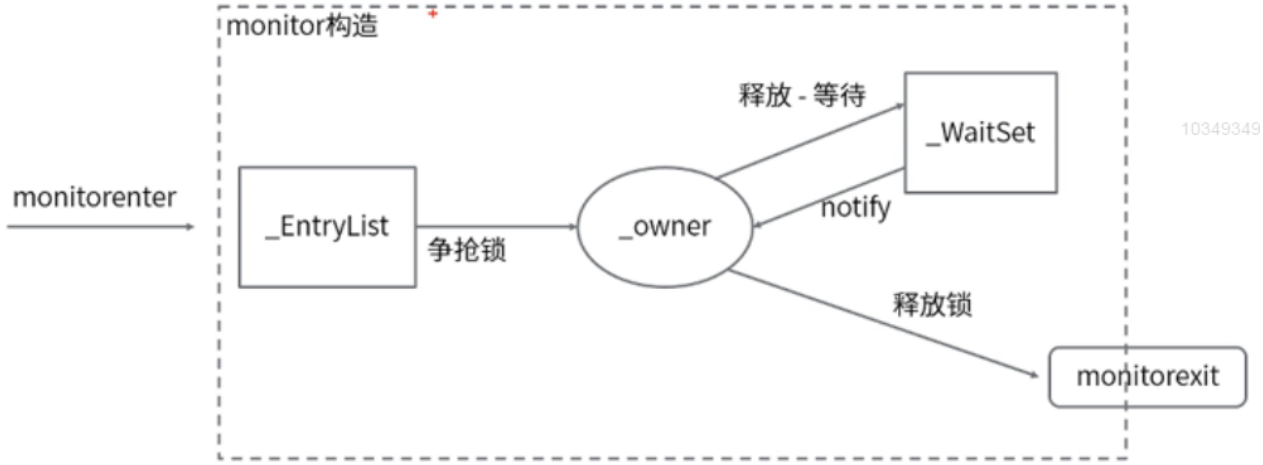
轻量级锁原理



1. 线程栈开辟一个空间，存储当前锁定对象的Mark Word信息。Lock Record可以存储多个锁定对象信息，将当前获取到锁的线程和对象绑定起来。
2. 使用CAS修改mark word完毕，加锁成功。则mark word中的tag进入00状态。解锁的过程，则是一个逆向恢复mark word的过程。

重量级锁

1. 修改mark word如果失败，则会自旋CAS一定次数，该次数可以通过参数配置，超过次数，仍未抢到锁，则锁升级为重量级锁，进入阻塞。monitor也叫做管程，一个对象会有一个对象的monitor。



自己实现一把简单的锁

```
public class MyLockDemo {
    private volatile int i = 0;

    private MyLock myLock = new MyLock();

    public void incr() {
        myLock.lock();
        try {
            i++;
        } finally {
            myLock.unlock();
        }
    }

    public int getI() {
        return i;
    }

    public static void main(String[] args) throws InterruptedException {
        MyLockDemo unsafeThreadDemo = new MyLockDemo();
        for (int i = 0; i < 7; i++) {
            new Thread(){
                @Override
                public void run() {
                    for (int j = 0; j < 10000; j++) {
                        unsafeThreadDemo.incr();
                    }
                }
            }.start();
        }
        Thread.sleep(2000);
        System.out.println("result: " + unsafeThreadDemo.getI());
    }
}

public class MyLock {
```

```
    private AtomicReference<Thread> owner = new AtomicReference<>();

    private BlockingQueue<Thread> waiters = new LinkedBlockingDeque<>();

    public void lock() {
        Thread currentThread = Thread.currentThread();
        while (!owner.compareAndSet(null, currentThread)) {
            waiters.add(currentThread);
            LockSupport.park(currentThread);
        }
    }

    public void unlock() {
        Thread currentThread = Thread.currentThread();
        if (owner.compareAndSet(currentThread, null)) {
            Thread thread = waiters.poll();
            LockSupport.unpark(thread);
        }
    }
}
```

响应中断

```
public class LockInterruptiblyDemo1 {
    private Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        LockInterruptiblyDemo1 demo1 = new LockInterruptiblyDemo1();
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                try {
                    demo1.test(Thread.currentThread());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        Thread thread0 = new Thread(runnable);
        Thread thread1 = new Thread(runnable);
        thread0.start();
        Thread.sleep(500); // 等待0.5秒，让thread1先执行

        thread1.start();
        Thread.sleep(2000); // 两秒后，中断thread2

        thread1.interrupt(); // 应该被中断
    }

    public void test(Thread thread) throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + ", 想获取锁");
```

```
        // lock.lockInterruptibly();
        lock.lock();   //注意，如果需要正确中断等待锁的线程，必须将获取锁放在外面，然后将
InterruptedException抛出
        try {
            System.out.println(thread.getName() + "~~运行了~~");
            Thread.sleep(10000); // 抢到锁，10秒不释放
        } finally {
            System.out.println(Thread.currentThread().getName() + "执行finally");
            lock.unlock();
            System.out.println(thread.getName() + "释放了锁");
        }
    }
}
结果1：
Thread-0， 想获取锁
Thread-0~~运行了~~
Thread-1， 想获取锁
Thread-0执行finally
Thread-0释放了锁
Thread-1~~运行了~~
Thread-1执行finally
Thread-1释放了锁
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at
com.study.lock.reentrantLock.LockInterruptiblyDemo1.test(LockInterruptiblyDemo1.java:38)
    at
com.study.lock.reentrantLock.LockInterruptiblyDemo1$1.run(LockInterruptiblyDemo1.java:16)
    at java.lang.Thread.run(Thread.java:745)

结果2：
Thread-0， 想获取锁
Thread-0~~运行了~~
Thread-1， 想获取锁
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireInterruptibly(AbstractQueued
Synchronizer.java:898)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly(AbstractQueuedSy
nchronizer.java:1222)
    at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.java:335)
    at
com.study.lock.reentrantLock.LockInterruptiblyDemo1.test(LockInterruptiblyDemo1.java:35)
    at
com.study.lock.reentrantLock.LockInterruptiblyDemo1$1.run(LockInterruptiblyDemo1.java:16)
    at java.lang.Thread.run(Thread.java:745)
Thread-0执行finally
Thread-0释放了锁
```

从结果1中我们可以看出使用锁的lock()方法无法响应中断，主线程在调用thread1.interrupt()方法后，线程1任然在执行，知道调用sleep()方法时看到线程已经是中断状态而抛出异常。当使用锁的lockInterruptibly()方法时，从结果2中可以看出thread1可以响应中断，不再执行，thread0执行正常。

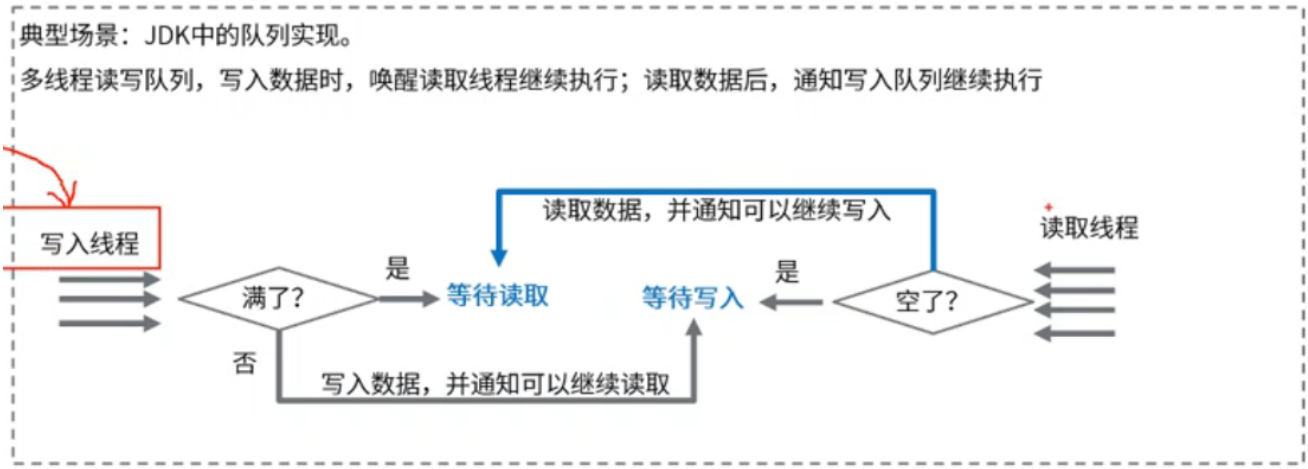
可重入锁

```
public class ReentrantDemo1 {
    private static final ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) {
        lock.lock();  // block until condition holds
        try {
            System.out.println("第一次获取锁");
            System.out.println("当前线程获取锁的次数" + lock.getHoldCount());
            lock.lock(); // 可重入的概念 // 如果时不可重入，这个代码应该阻塞
            System.out.println("第二次获取锁了");
            System.out.println("当前线程获取锁的次数" + lock.getHoldCount());
        } finally {
            lock.unlock();
            lock.unlock();
        }
        System.out.println("当前线程获取锁的次数" + lock.getHoldCount());

        // 如果不释放，此时其他线程是拿不到锁的
        new Thread(() -> {
            System.out.println(Thread.currentThread() + " 期望抢到锁");
            lock.lock();
            System.out.println(Thread.currentThread() + " 线程拿到了锁");
        }).start();


    }
}

结果1：
第一次获取锁
当前线程获取锁的次数1
第二次获取锁了
当前线程获取锁的次数2
当前线程获取锁的次数0
Thread[Thread-0,5,main] 期望抢到锁
Thread[Thread-0,5,main] 线程拿到了锁

结果2：
第一次获取锁
当前线程获取锁的次数1
第二次获取锁了
当前线程获取锁的次数2
当前线程获取锁的次数1
Thread[Thread-0,5,main] 期望抢到锁
```

从结果1中可以看出，锁的lock()方法支持可重入。当我们注释掉finally中的一个解锁操作后，结果为2，这次Thread0-5无法获取到锁，因为主线程加锁两次但只释放了一次。

Condition

1. 用于替代wait/notify。
2. Object中的wait()，notify()，notifyAll()方法是synchronized配合使用的，可以唤醒一个或者全部(单个等待集)。
3. Condition是需要和Lock配合使用的，提供多个等待集合，更精确的控制(底层是park/unpark机制)。




```
public class QueueDemo {
    final Lock lock = new ReentrantLock();
    // 指定条件的等待 - 等待有空位
    final Condition notFull = lock.newCondition();
    // 指定条件的等待 - 等待不为空
    final Condition notEmpty = lock.newCondition();

    // 定义数组存储数据
    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    // 写入数据的线程,写入进来
    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length) // 数据写满了
                notFull.await(); // 写入数据的线程,进入阻塞
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal(); // 唤醒指定的读取线程
        } finally {
            lock.unlock();
        }
    }
    // 读取数据的线程,调用take
    public Object take() throws InterruptedException {
        lock.lock();
        try {
```

```
            while (count == 0)
                notEmpty.await(); // 线程阻塞在这里,等待被唤醒
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal(); // 通知写入数据的线程,告诉他们取走了数据,继续写入
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

参考资料

https://www.oracle.com/technetwork/java/6-performance-137236.html

https://jdk.java.net/

https://wiki.openjdk.java.net/display/HotSpot/Synchronization