注意：笔记中所有被我加了注释的代码都被我添加到笔记中了，没有添加到笔记中的代码基本上都能看懂，有实在看不懂的地方去回顾下视频

1. 课程内容

- Concurrent同步工具类
- CountDownLatch
- CyclicBarrier
- Semaphore
- Exchanger
- ReentrantLock
- ReentrantReadWriteLock
- 四种线程池与自定义线程池、底层代码
- 设计模式：单例、Future、Master-Worker、Producer- Consumer模式原理和实现

2. CountDownLatch

- CountDownLatch 是一个辅助工具类，它允许一个或多个线程等待一系列指定操作的完成。
- CountDownLatch 以一个给定的数量初始化，每个这个类的实例对象可以被理解为一个计数器，每个实例对象的初始化的值可以被理解为计数器的初始值。
- 一个线程调用这个CountDownLatch类的某个实例对象（记作Q）中的await()方法之后就会进入阻塞状态，直到这个计数器的值为0
- 当其他线程调用Q对象的countDown() 方法，这个Q对象上的计数器的数量就减一。当实例化Q对象的计数器的数量被减为0的时候所有因为这个Q对象的计数器不是0而被阻塞的线程会同时停止阻塞。
- 在并发编程中可以使用多个CountDownLatch的实例对象（计数器）进行编程
- 示例：
  CountDownLatchTest1
  CountDownLatchTest2

3. CyclicBarrier

- CyclicBarrier一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。
- 需要所有的子任务都完成时，才执行主任务，这个时候就可以选择使用CyclicBarrier。

- 在程序中如果调用await方法的线程数量小于创建CyclicBarrier实例对象的时候指定的数量，在不加限制的情况下会永远等待下去
- 示例：
  CyclicBarrierTest1

```
public class CyclicBarrierTest1 {

    public static void main(String[] args) throws
IOException, InterruptedException {
        //如果将参数改为4，但是下面只加入了3个选手，这永
远等待下去，这里的参数3指的是有三个线程
        //Waits until all parties have invoked
await on this barrier.
        CyclicBarrier barrier = new
CyclicBarrier(3);

        ExecutorService executor =
Executors.newFixedThreadPool(3);
        executor.submit(new Thread(new
Runner(barrier, "1号选手")));
        executor.submit(new Thread(new
Runner(barrier, "2号选手")));
        executor.submit(new Thread(new
Runner(barrier, "3号选手")));

        executor.shutdown();
    }
}

class Runner implements Runnable {
    // 一个同步辅助类，它允许一组线程互相等待，直到到达某
个公共屏障点 (common barrier point)
    private CyclicBarrier barrier;

    private String name;

    public Runner(CyclicBarrier barrier, String
name) {
        super();
        this.barrier = barrier;
        this.name = name;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000 * (new
Random()).nextInt(8));
            System.out.println(name + " 准备好
了...");
            // barrier的await方法，在所有参与者都已经
在此 barrier 上调用 await 方法之前，将一直等待。
```

```
                barrier.await();
                //设置等待时间,如果等待了1秒,最后一个线程还没有就位,则自己继续运行,但是会导致Barrier被标记为一个已经破坏的Barrier
                //barrier.await(1,TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                System.out.println(name + " 中断异常！");
            } catch (BrokenBarrierException e) {
                System.out.println(name + " Barrier损坏异常！");
            } catch (TimeoutException e){
                //当在调用barrier.await(1,TimeUnit.SECONDS);方法的时候在1s内没有等到其他线程就会抛出这个异常异常，同时当其他线程在1s之后再次调用await方法的时候就会抛出BrokenBarrierException异常
                System.out.println(name+"等待超时！");
            }
            System.out.println(name + " 起跑！");
        }
    }
```

```
运行结果：
3号选手 准备好了...
3号选手等待超时!
3号选手 起跑!
2号选手 准备好了...
2号选手 Barrier损坏异常!
2号选手 起跑!
1号选手 准备好了...
1号选手 Barrier损坏异常!
1号选手 起跑!
```

CyclicBarrierTest2（目的是表达一种情况：某个线程执行的操作的前提是其他线程都运行完毕）

4. Semaphore

- Semaphore一个计数信号量。信号量维护了一个许可集合（一个集合中存放着许可证）;通过acquire()和release()来获取和释放访问许可证。只有通过acquire获取了许可证的线程才能执行,否则阻塞。通过release释放许可证其他线程才能进行获取。
- 公平性：没有办法保证线程能够公平地可从信号量中获得许可。也就是说，无法担保掉第一个调用 acquire() 的线程会是第一个获得一个许可的线程。如果第一个线程在等待一个许可时发生阻塞，而第二个线程前来索要一个许可的时候刚好有一个许可被释放出来，那么它就可能会在第一个线程之前获得许可。如果你想要强制公平， Semaphore

类有一个具有一个布尔类型的参数的构造子，通过这个参数以告知 Semaphore 是否要强制公平。强制公平会影响到并发性能，所以除非你确实需要它否则不要启用它。

- 示例： SemaphoreTest1

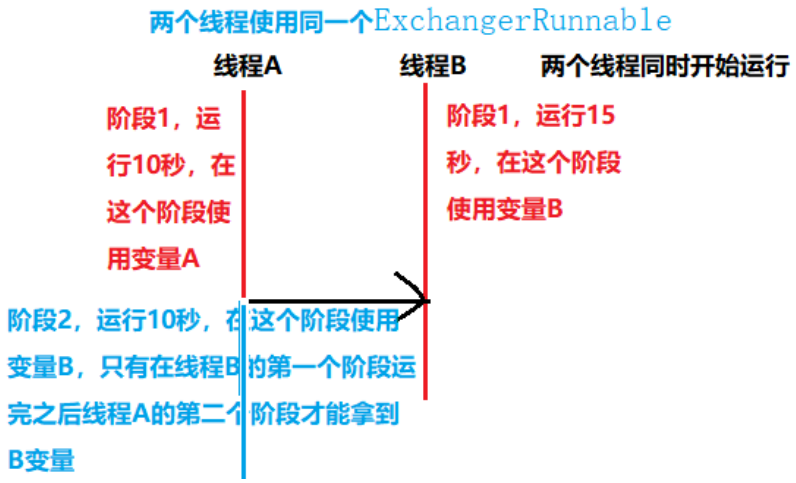
```
public static void main(String[] args) {
    // 线程池
    ExecutorService exec =
Executors.newCachedThreadPool();
    // 只能2个线程同时访问（设置许可证的数量），默认是不公平的,fair值为false
    //final Semaphore semp = new Semaphore(2);
    // 强制公平
    final Semaphore semp = new Semaphore(2,true);
    // 模拟多个客户端并发访问
    for (int index = 0; index < 5; index++) {
        Runnable run = new Runnable() {
            public void run() {
                try {

 System.out.println(Thread.currentThread().getNam
e() + "尝试获取许可证");
                    // 获取许可
                    semp.acquire();

 System.out.println(Thread.currentThread().getNam
e() + "获取许可证");
                    Thread.sleep(1000);
                    // 访问完后，释放 ，如果屏蔽下面的语句，则在控制台只能打印5条记录，之后线程一直阻塞

 System.out.println(Thread.currentThread().getNam
e() + "释放许可证");
                    semp.release();
                } catch (InterruptedException e)
{
                }
            }
        };
        exec.execute(run);
    }
    // 退出线程池
    exec.shutdown();
}
```

5.Exchanger

- Exchanger 类表示一种两个线程可以进行互相交换对象的会和点。
- 只能用于两个线程之间，并且两个线程必须都到达汇合点才会进行数据交换（两个线程使用同一个交换器）
- 

- 示例： ExchangerTest1

```
public class ExchangerTest1 {
    public static void main(String[] args) {
        Exchanger<String> exchanger = new
Exchanger<String>();
        ExchangerRunnable exchangerRunnable1 =
new ExchangerRunnable(exchanger, "A");
        ExchangerRunnable exchangerRunnable2 =
new ExchangerRunnable(exchanger, "B");

        //      new
Thread(exchangerRunnable1).start();
        //      new
Thread(exchangerRunnable2).start();

        Exchanger<String> exchanger2 = new
Exchanger<String>();
        //这里的新创建的两个线程可以都使用交换器1也可以都是用交换器2，但是不能不成对出现，否则会导致程序执
        //在调用exchange方法的时候被阻塞住
        ExchangerRunnable exchangerRunnable3 =
new ExchangerRunnable(exchanger, "C");
        ExchangerRunnable exchangerRunnable4 =
new ExchangerRunnable(exchanger, "D");
        new Thread(exchangerRunnable1).start();
        new Thread(exchangerRunnable2).start();
        new Thread(exchangerRunnable3).start();
        new Thread(exchangerRunnable4).start();
    }
}

class ExchangerRunnable implements Runnable {
    //交换器
    Exchanger<String> exchanger = null;
    //该变量指向：要被交换的对象、以及交换后得到的对象
    String object = null;
```

```
    public ExchangerRunnable(Exchanger<String>
exchanger, String object) {
        this.exchanger = exchanger;
        this.object = object;
    }

    public void run() {
        try {
            Object previous = this.object;

 System.out.println(Thread.currentThread().getNam
e() +"交换前："+this.object);
            if("C".equals(previous)){
                Thread.sleep(1000);

 System.out.println(Thread.currentThread().getNam
e() +"对数据C的处理耗时1s");
            }else if("D".equals(previous)){
                Thread.sleep(2000);

 System.out.println(Thread.currentThread().getNam
e() +"对数据D的处理耗时2s");
            }else if("A".equals(previous)){
                Thread.sleep(3000);

 System.out.println(Thread.currentThread().getNam
e() +"对数据A的处理耗时3s");
            }else if("B".equals(previous)){
                Thread.sleep(4000);

 System.out.println(Thread.currentThread().getNam
e() +"对数据B的处理耗时4s");
            }
            //两个对象必须在此处汇合,只有一个线程调用
exchange方法是不会进行数据交换的，会一直等待
            //方法中的参数是要交换的对象，得到的是交换后
的对象
            this.object =
this.exchanger.exchange(this.object);

 System.out.println(Thread.currentThread().getNam
e() + " 交换后: " + this.object);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

6. ReentrantLock-1

- ReentrantLock可以用来替代Synchronized，在需要同步的代码块加上锁，最后一定要释放锁，否则其他线程永远进不来。
- 示例：com.mimaxueyuan.demo.high.lock1.ReentrantLockTest1（在代码中使用lock.loak()+lock.unlock()取代了synchronized关键字的作用域）

- 可以使用Condition来替换wait和notify来进行线程间的通讯，Condition只针对某一把锁（需要通过锁来创建Condition对象，使用condition.await()+signal()/signalAll()取代了Object的wait()+notify()/notifyAll()的阻塞和唤醒操作）。

- 示例：com.mimaxueyuan.demo.high.lock1.ConditionTest1

```
public class ConditionTest1 {

    private Lock lock = new ReentrantLock();
    private Condition condition =
lock.newCondition();

    public void run1(){
        try {
            lock.lock();
            System.out.println("当前线程：" +
Thread.currentThread().getName() + "进入等待状
态..");
            Thread.sleep(3000);
            System.out.println("当前线程：" +
Thread.currentThread().getName() + "释放锁..");
            condition.await();  // 相当于 Object
wait,释放锁
            System.out.println("当前线程：" +
Thread.currentThread().getName() +"继续执行...");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void run2(){
        try {
            lock.lock();
            System.out.println("当前线程：" +
Thread.currentThread().getName() + "进入..");
            Thread.sleep(3000);
            System.out.println("当前线程：" +
Thread.currentThread().getName() + "发出唤醒..");
```

```
            condition.signal();     // 相当于
Object notify/notifyAll, 唤醒锁
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {

        final ConditionTest1 uc = new
ConditionTest1();
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                uc.run1();
            }
        }, "t1");
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                uc.run2();
            }
        }, "t2");

        t1.start();
        t2.start();
    }
}
```

- 一个Lock可以创建多个Condition，更加灵活
- 示例：com.mimaxueyuan.demo.high.lock1.ConditionTest2（使用同一锁的多个Condition实例对象）

7. ReentrantLock-2

- ReentrantLock的构造函数可以如传入一个boolean参数，用来指定公平/非公平模式，默认是false非公平的。非公平的效率更高。
- 可重入锁
- Lock的其他方法：
    - tryLock()：尝试获得锁，返回true/false
    - tryLock(timeout, unit)：在给定的时间内尝试获得锁
    - isFair()：是否为公平锁
    - isLocked()：当前线程是否持有锁

- lock.getHoldCount()：持有锁的数量，只能在当前调用线程内部使用，不能再其他线程中使用
- 示例：com.mimaxueyuan.demo.high.lock1.HoldCountTest（解释重入锁和getHoldCount方法）

8. ReentrantReadWriteLock

分为：写锁、悲观读锁，其中读锁是共享锁，写锁是排他锁（独占锁）

- ReentrantReadWriteLock读写锁，采用读写分离机制，高并发下读多写少时性能优于ReentrantLock。
- 读锁和写锁都是可重入锁
- 读读共享(不互斥)，写写互斥，读写互斥
- 示例：com.mimaxueyuan.demo.high.lock2.ReadWriteLockTest1

```
public class ReadWriteLockTest1{
    private ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();
    //得到读锁
    private ReadLock readLock =
rwLock.readLock();
    //得到写锁
    private WriteLock writeLock =
rwLock.writeLock();

    public void read() {
        try {
            readLock.lock();
            System.out.println("当前线程:" +
Thread.currentThread().getName() + "read进入...");
            Thread.sleep(3000);
            System.out.println("当前线程:" +
Thread.currentThread().getName() + "read退出...");
        } catch (Exception e) {
            e.printStackTrace();
        } finally{
            readLock.unlock();
        }
    }

    public void write() {
        try {
            writeLock.lock();
            System.out.println("当前线程:" +
Thread.currentThread().getName() + "write进
入...");
            Thread.sleep(3000);
```

```
            System.out.println("当前线程:" +
Thread.currentThread().getName() + "write退
出...");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            writeLock.unlock();
        }
    }

    public static void main(String[] args) {
        final ReadWriteLockTest1 urrw = new
ReadWriteLockTest1();

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                urrw.read();
            }
        }, "t1");
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                urrw.read();
            }
        }, "t2");
        Thread t3 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                urrw.write();
            }
        }, "t3");
        Thread t4 = new Thread(new Runnable() {
            @Override
            public void run()
            {
                urrw.write();
            }
        }, "t4");
        //测试读读
        //t1.start(); // Read
        //t2.start(); // Read
        //测试读写
        //t1.start(); // Read
        //t3.start(); // Write
        //测试写写
        t3.start(); //write
        t4.start(); //write
    }
```

```
}
```

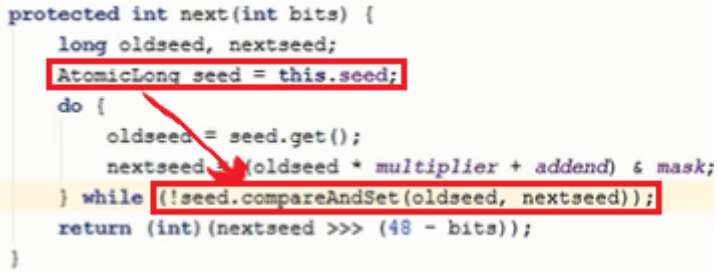
15. 高性能随机数

Random 与 ThreadLocalRandom在高并发场景下的性能差异和原理

高并发随机数ThreadLocalRandom与Random分析

知识点

Random存在性能缺陷，主要原因是要不断的计算新的种子更新原种子，使用CAS方法。高并发的情况下会造成大量的线程自旋，而只有一个线程会更新成功，当一个线程更新成功之后其他线程才可以继续更新，如果某个线程还在执行更新的时候其他线程就在效用seed.compareAndSet方法尝试着更新，那么其他线程会得到一个false，重新进入这个while循环获得新的oldseed和nextseed；
Random类中的next方法源码照片：



ThreadLocalRandom采用ThreadLocal的机制，每一个线程都是用自己的种子去进行计算下一个种子，规避CAS在并发下的问题。

源码阅读

java.util.Random java.util.concurrent.ThreadLocalRandom 继承自*Random*

代码演示：
com.mkevin.demo4.RandomDemo0
com.mkevin.demo4.RandomDemo1 两者性能对比

16. 高性能累加器

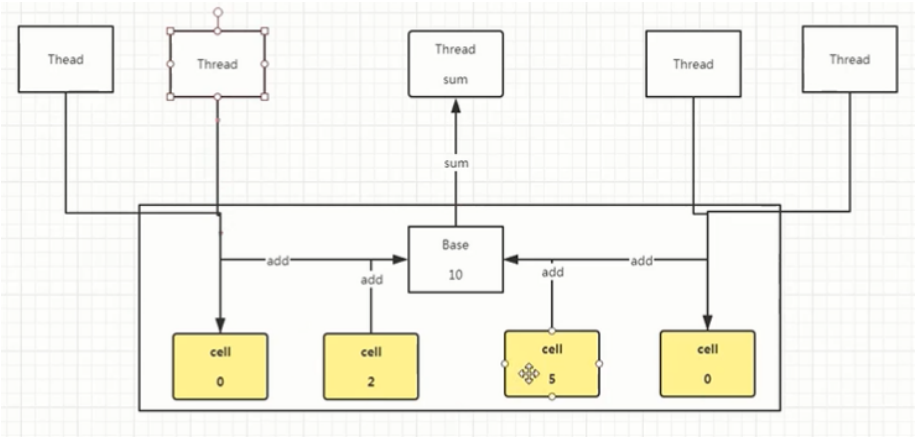
LongAdder、DoubleAdder、LongAccumulator、DoubleAccumulator

16.1 高性能累加器LongAddr

知识点

AtomicLong存在性能瓶颈，由于使用CAS方法。高并发的情况下会造成大量的线程自旋，而只有一个线程会更新成功，浪费CPU资源（和Randm类的使用时产生的问题是同样的）。

LongAdder的思想是将单一的原子变量拆分为多个变量，当需要的时候才计算它们的和，从而降低高并发下的资源争抢。



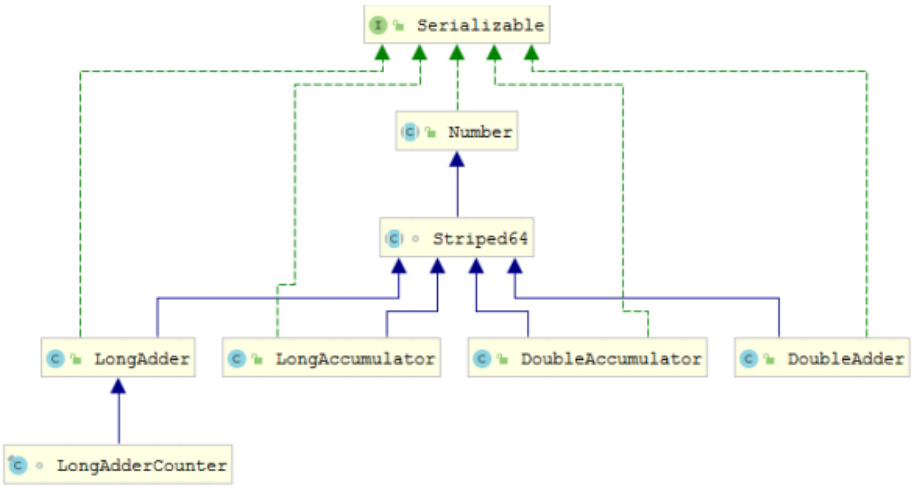
源码阅读

> *java.util.concurrent.atomic.AtomicLong*
> *java.util.concurrent.atomic.LongAdder 继承自*
> *java.util.concurrent.atomic.Striped64*
>
> *具体源码的阅读过程去看视频*

代码演示

> *com.mkevin.demo5.LongAdderDemo0*
> *com.mkevin.demo5.LongAdderDemo1 两者性能对比*

16.2 类图



没有incrementAndGet、decrementAndGet这种方法，只有单独的increment、longValue这种方法，如果组合使用则需要自己做同步控制，否则无法保证原子性。

LongAddr本质上是一种空间换时间的策略，累加器家族还有以下3种
java.util.concurrent.atomic.DoubleAdder
java.util.concurrent.atomic.LongAccumulator
java.util.concurrent.atomic.DoubleAccumulator

LongAdder是LongAccumulator的特例，DoubleAdder是DoubleAccumulator的特列；
Accumulator的特点是可以设置初始值、自定义累加算法

示例：
com.mkevin.demo5.LongAccumulatorDemo1

17. COW迭代器的弱一致性

java.util.concurrent.CopyOnWriteArrayList
java.util.concurrent.CopyOnWriteArraySet

17.1 COWIterator的弱一致性

知识点

使用COW容器的iterator方法实际返回的是COWIterator实例，遍历的数据为快照数据，其他线程对于容器元素增加、删除、修改不对快照产生影响。对java.util.concurrent.CopyOnWriteArrayList、java.util.concurrent.CopyOnWriteArraySet 均适用。

源码阅读

> *java.util.concurrent.CopyOnWriteArrayList*
> *java.util.concurrent.CopyOnWriteArraySet*

代码演示

> *com.mkevin.demo7.COWDemo0*
>
> *com.mkevin.demo7.COWDemo1*

18. LockSupport

java.util.concurrent.locks.LockSupport

知识点

1、**LockSupport的底层采用Unsafe类来实现，他是其他同步类的阻塞与唤醒的基础。**

2、park（阻塞当前正在运行的线程，无参数）与unpark（解除参数中指定线程的阻塞）需要成对使用，这两个都是静态方法；
parkUntil与parkNanos可以单独使用，parkUntil方法的作用是将当前正在运行的成成阻塞到参数指定的时候，这个方法的参数指的就是要阻塞到的时间，

3、先调用unpark再调用park会导致park失效

4、线程中断interrupte会导致park失效（也就是不阻塞）并且不抛异常

5、例如blocker可以对堆栈进行追踪，官方推荐，例如结合jstack进行使用

源码阅读

> *java.util.concurrent.locks.LockSupport #知识点1*

代码演示

> *com.mkevin.demo6.LockSupportDemo0 #知识点*2 *(简单介绍这几个方法的使用方式)*

```
public static void main(String[] args) throws
InterruptedException {
 Thread t = new Thread(new Runnable() {
     @Override
     public void run() {

System.out.println(Thread.currentThread().getName()
+" start run");
         //KEVIN知识点1: park()与unpark()需要成对儿使
用，这个方法的作用是：阻塞当前正在运行的线程，无参数
         //LockSupport.park();
         //KEVIN知识点2：parkUntil(deadline) 可以单独使
用，从当前时间开始，阻塞当前正在运行的线程到两秒钟之后

//LockSupport.parkUntil(System.currentTimeMillis()+
2000);
         //KEVIN知识点3：parkNanos(nanos) 可以单独使
用，作用和parkUntil类似，不过这里的参数
         //              是：阻塞当前正在运行的线程的时间的
纳秒值
         LockSupport.parkNanos(1000000000);

System.out.println(Thread.currentThread().getName()
+" stop run");
     }
 });
 t.start();
 //Thread.sleep(2000);
 //KEVIN知识点1: park()与unpark()需要成对儿使用，解除参数
指定的线程的阻塞，静态方法
 //LockSupport.unpark(t);
```

```
 t.join();
 System.out.println(Thread.currentThread().getName(
)+" stop run");
}
```

*com.mkevin.demo6.LockSupportDemo1 #知识点*4 *（一个线程被中断导致该线程中所有的park方法都失效）*

```
public static void main(String[] args) throws
InterruptedException {
 Thread t = new Thread(new Runnable() {
     @Override
     public void run() {

System.out.println(Thread.currentThread().getName()
+" start run");
         //KEVIN知识点1:调用线程的interrupt也会导致park
被中断，但是差别是它不抛出InterruptedException
         LockSupport.park();
         if(Thread.currentThread().isInterrupted())
{
             //KEVIN知识点1: 通常通过判断来弥补不抛出异常
带来的疑问，当发生中断时执行自定义操作

System.out.println(Thread.currentThread().getName()
+" is interrupted, so do something");
             //测试线程被中断之后是所有的park都失效了，还
是只有已经工作过的park失效
             //测试的结果是：当一个线程被打断之后，该线程中
所有的park都会失效
             LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" repark");
             LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" repark");
         }

System.out.println(Thread.currentThread().getName()
+" stop run");
     }
 });
 t.start();
 Thread.sleep(2000);
 //KEVIN知识点1:调用线程的interrupt也会导致park被中断，即
不再阻塞，但是差别是它不抛出InterruptedException
 //也就是如果不注释这行代码，那么程序不会被阻塞住，如果注释了
这行代码，程序就会被阻塞住
 //t.interrupt();
```

```
 t.join();
 System.out.println(Thread.currentThread().getName(
)+" stop run");
}
```

**com.mkevin.demo6.LockSupportDemo2 #知识点*3 *（目的是为了说明：在使用park+unpark的时候没有计数的概念）

```
System.out.println(Thread.currentThread().getName()
+" start run");

      LockSupport.unpark(Thread.currentThread());
      //即使这三个被注释的代码打开，下面倒数三个park方法的调用也不会成功的不被阻塞
      //LockSupport.unpark(Thread.currentThread());
      //LockSupport.unpark(Thread.currentThread());
      //LockSupport.unpark(Thread.currentThread());
      //先调用unpark方法，之后第一次调用park方法的时候不会被阻塞
      LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" park1 already run");
      //这三个park方法都会被阻塞
      LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" park2 already run");
      LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" park3 already run");
      LockSupport.park();

System.out.println(Thread.currentThread().getName()
+" park4 already run");


System.out.println(Thread.currentThread().getName()
+" stop run");
```

com.mkevin.demo6.LockSupportDemo3 #知识点5*

```
```

19. AQS

java.util.concurrent.locks.AbstractQueuedSynchronizer
简称*AQS*

知识点 *java.util.concurrent.locks.AbstractQueuedSynchronizer*

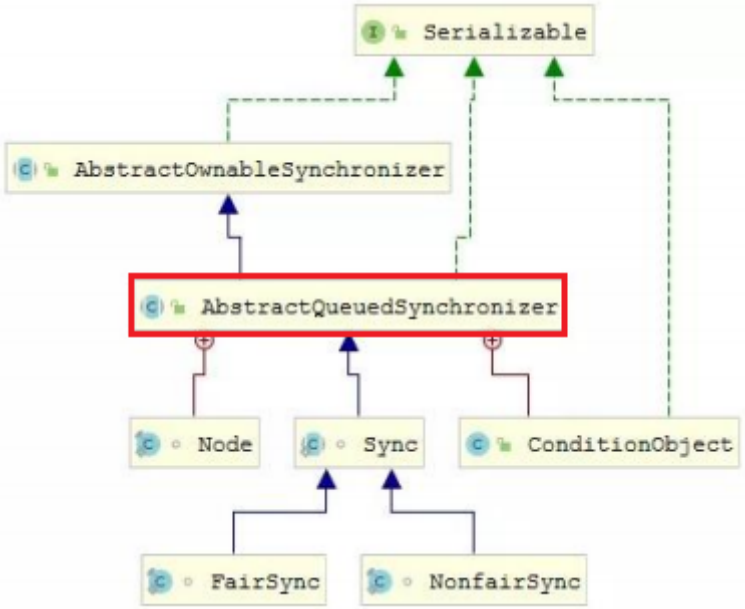
1、**抽象队列同步器简称AQS，它是同步器的基础组件，JUC种锁的底层实现均依赖于AQS**，开发不需要使用（但是需要了解它的原理和构成）；AQS的实现类的队列中存放的是线程（被阻塞的线程）。

2、采用FIFO的双向队列实现，队列元素为Node（静态内部类），Node内的thread变量用于存储进入队列的线程。

3、Node节点内部的SHARED用来标记该线程是获取共享资源时被阻塞挂起后放入AQS队列的， EXCLUSIVE用来标记线程是获取独占资源时被挂起后放入AQS队列的。waitStatus记录当前线程等待状态，可以为CANCELLED（线程被取消了）、SIGNAL（线程需要被唤醒）、CONDITION（线程在条件队列里 面等待）、PROPAGATE（释放共享资源时需要通知其他节点）；prev记录当前节点的前驱节点，next记录当前节点的后继节点。

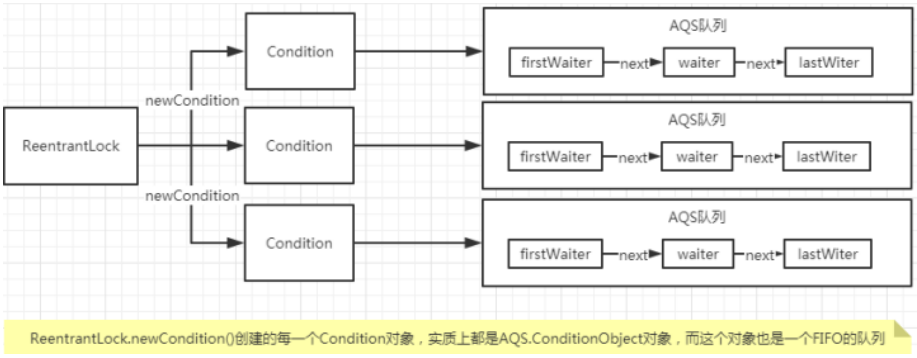
4、在AQS中维持了一个单一的状态信息state，可以通过getState、setState、compareAndSetState函数 修改其值。对于ReentrantLock的实现来说，state可以用来表示当前线程获取锁的可重入次数；对于读写锁ReentrantReadWriteLock来说，state的高16位表示读状态，也就是获取该读锁的次数，低16位表示获取到写锁的线程的可重入次数；对于semaphore来说，state用来表示当前可用信号的个数；对于 CountDownlatch来说，state用来表示计数器当前的值。

19.1类图



Demo: com.mkevin.demo8.AQSDemo0（视频中讲这个示例代码的时候配合着源码以及执行流程介绍了一次，可以去看看；视频中介绍是可重入锁）
Demo: com.mkevin.demo8.AQSDemo1（视频中讲这个示例代码的时候配合着源码以及执行流程介绍了一次，可以去看看；视频中介绍的是可重入锁的Condition）

19.2AQS.ConditionObject



20. Phaser

移相器/阶段器

Phaser，中文为移相器，是电子专业中使用的术语。此类在JDK7中加入的并发工具类全路径为java.util.concurrent.Phaser

这个类的功能类似于CountDownLatch、CyclicBarrier

20.1 相关术语

可以对照着下面的图片理解

1、**party**

通过phaser同步的线程被称为party（参与者）. **所有需要同步的party必须持有同一个phaser对象.** party需要向phaser注册,执行phaser.register()方法注册,该方法仅仅是增加phaser中的线程数量.（不常用方式）

也可以通过构造器注册,比如new Phaser(3)就会在创建phaser对象时注册3个party. (常用方式)

这3个party只要持有该phaser对象并调用该对象的api就能实现同步.

2、unarrived
party到达一个phaser（阶段）之前处于unarrived状态

3、arrived
到达时处于arrived状态.一个arrived的party也被称为arrival

4、deregister

一个线程可以在arrive某个phase后退出(deregister),与参赛者中途退赛相同，可以使用arriveAndDeregister()方法来实现.（到达并注销）

5、phase计数

Phaser类有一个phase计数,初始阶段为0。当一个阶段的所有线程arrive时,会将phase计数加1,这个动作被称为advance. 当这个计数达到Integer.MAX_VALUE时,会被重置为0,开始下一轮循环

advace这个词出现在Phaser类的很多api里，比如arriveAndAwaitAdvance()、awaitAdvance(int phase)等.

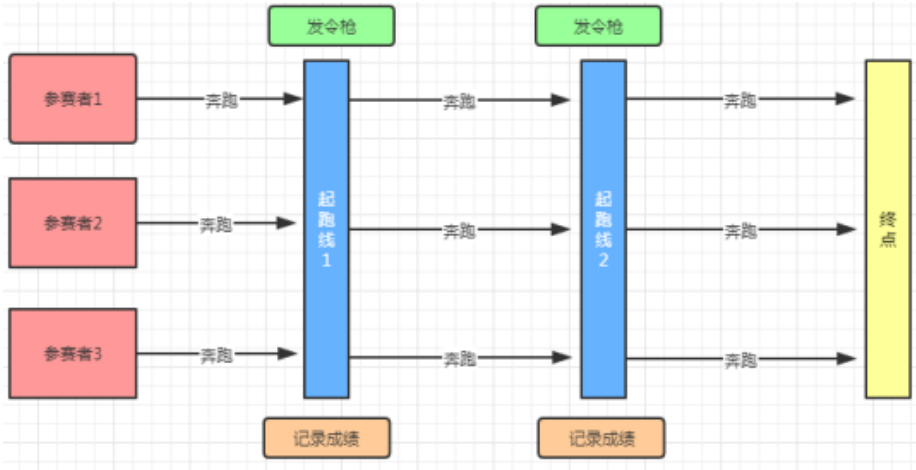
在advance时,会触发onAdvance(int phase, int registeredParties)方法的执行.

6、onAdvance(int phase, int registeredParties)

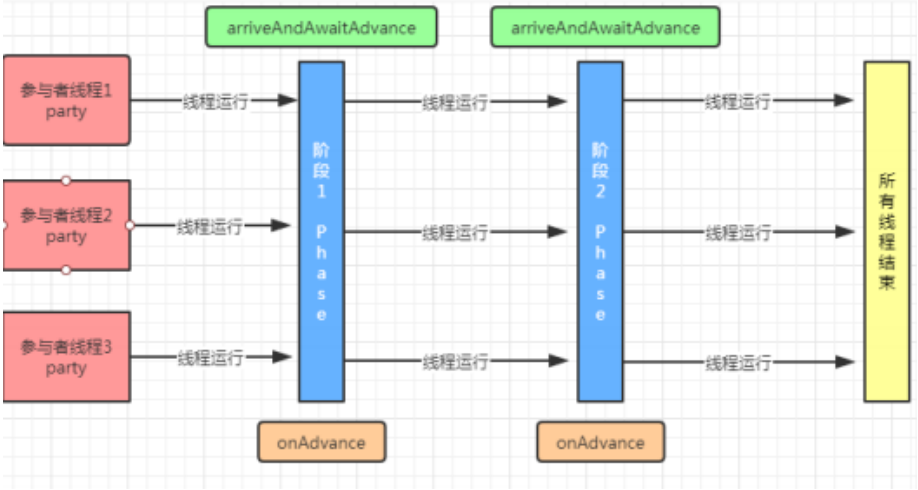
可以在这个方法中定义advance过程中需要执行何种操作。

如果需要进入下一阶段(phase)执行,返回false.如果返回true,会导致phaser结束因此该方法也是终止phaser的关键所在

20.2 简单图示



所有的参赛者必须奔跑第一个起跑线；
等所有人到达后，发令枪会立刻给出信号，所有参赛者一同出发奔向第二个起跑线；
所有参赛者到达第二个起跑线之后，发令枪立刻给出信号，所有参赛者以同出发奔向终点；
最后结束。



所有线程首先要运行到阶段1（phase 1）；
等所有线程达到后再前进（advice）到下一个阶段（phase2）；

先到达的要等待（wait），全部到达后再前进到终点。

代码演示

> *com.mkevin.demo10.PhaserDemo1 *（展示这个类的用法）

```
public class PhaserDemo1 {
 public static void main(String[] args) throws
InterruptedException {
     //Phaser(5)代表注册的party数量, 不传入默认为0
     Phaser phaser = new Phaser(5);
     new Runner(phaser).start();

Thread.sleep(ThreadLocalRandom.current().nextInt(10
00));
     new Runner(phaser).start();

Thread.sleep(ThreadLocalRandom.current().nextInt(10
00));
     new Runner(phaser).start();

Thread.sleep(ThreadLocalRandom.current().nextInt(10
00));
     new Runner(phaser).start();

Thread.sleep(ThreadLocalRandom.current().nextInt(10
00));
     new Runner(phaser).start();
 }
 static  class Runner extends  Thread{
     private Phaser phaser;
     // 多个线程必须持有同一个phaser
     public Runner(Phaser phaser){
         this.phaser=phaser;
     }
```

```
        @Override
        public void run() {
            try {
                System.out.println(this.getName()+" is
ready1");
                //当执行到这一步的线程数量不够5的时候就要阻塞
等待，一旦线程数量等于5时，阻塞的线程就会同时向下继续执行run
方法
                phaser.arriveAndAwaitAdvance();

                System.out.println(this.getName()+"
running...");
                //模拟不同的线程需要不同的时间，以便体现出第二
阶段

Thread.sleep(ThreadLocalRandom.current().nextInt(30
00));
                System.out.println(this.getName()+" is
ready2");
                phaser.arriveAndAwaitAdvance();

                System.out.println(this.getName()+"
running...");
                //同上

Thread.sleep(ThreadLocalRandom.current().nextInt(30
00));
                System.out.println(this.getName()+" is
ready3");
                phaser.arriveAndAwaitAdvance();

                //同上

Thread.sleep(ThreadLocalRandom.current().nextInt(30
00));
                System.out.println(this.getName()+"
over");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

com.mkevin.demo10.PhaserDemo2（展示onAdvice方法）
com.mkevin.demo10.PhaserDemo3（展示坑，想表达的是：一个线程在某个阶段中突然注册或者注销，没有问题，但是如果一个线程没有注销，而是直接退出/报错，那么其他线程就会在下一个阶段夯住，等待这个退出，但是没有注销的线程）

21. StampedLock

分为：写锁、悲观读锁、乐观读锁；其中读锁是共享锁，写锁是排他锁（独占锁）。写写互斥、(悲观读)写互斥、(乐观读) 写不互斥、读读共享。

21.1 StampedLock简介&写锁writeLock

StampedLock类，在JDK8中加入全路径为java.util.concurrent.locks.StampedLock。

功能与RRW（ReentrantReadWriteLock）功能类似，只是多了一种乐观读锁。

StampedLock中引入了一个stamp（邮戳）的概念。它代表线程获取到锁的版本（是一个long类型的数字），每一把锁都有一个唯一的stamp（相当于一把钥匙，加锁获得，解锁使用）。

21.2 写锁

写锁writeLock，是排它锁、也叫独占锁，相同时间只能有一个线程获取锁，其他线程请求读锁和写锁都会被阻塞。

功能类似于ReetrantReadWriteLock.writeLock。

区别是**StampedLock的写锁是不可重入锁**。当前没有线程持有读锁或写锁的时候才可以获得获取到该锁。

示例：
Demo：com.mkevin.demo9.StampedLockDemo1

writeLock与unlockWrite必须成对儿使用，解锁时必须需要传入相对应的stamp才可以释放锁。每次获得锁之后都会得到一个新stamp值。

同一个线程获取锁后，再次尝试获取锁而无法获取，则证明其为非重入锁。

对于ReentrantLock，同一个线程获取锁后，再次尝试获取锁可以获取，则证明其为重入锁。

对于ReentrantReadWriteLock.WriteLock，同一个线程获取写锁后，再次尝试获取锁依然可获取锁，则证明其为重入锁。

StampedLock.tryWriteLock:尝试获取，如果能获取则获取锁，获取不到不阻塞，而是得到一个stamp为0的邮戳。

21.3 悲观读readLock

Demo: com.mkevin.demo9.StampedLockDemo2

```
//StampedLock.readLock正常使用：注意每次获得的写锁stamp不同,必须所有的读锁都释放才可以获得写锁
StampedLock sl = new StampedLock();
long stamp1 = sl.readLock();
System.out.println("get read lock1,stamp="+stamp1);
long stamp2 = sl.readLock();
System.out.println("get read lock2,stamp="+stamp2);
sl.unlockRead(stamp1);
sl.unlockRead(stamp2);
//在使用写锁之前套确保读锁已经全部释放，否则会被阻塞
long stamp3 = sl.writeLock();
System.out.println("get write lock,stamp="+stamp3);
```

悲观读锁是一个共享锁，没有线程占用写锁的情况下，多个线程可以同时获取读锁。如果其他线程已经获得了写锁，则阻塞当前线程；在获取到了悲观读锁并且没有释放的情况下，不可以使用写锁（这是和乐观读锁最大的区别）。

读锁可以多次获取（没有写锁占用的情况下），写锁必须在读锁全部释放之后才能获取写锁。

只要还有任意的锁没有释放（无论是写锁还是读锁），这时候来尝试获取写锁都会失败，因为读写互斥，写写互斥。写锁本身就是排它锁。

在多个线程之间依然存在写写互斥、读写互斥、读读共享的关系

为什么叫悲观读锁？悲观锁认为数据是极有可能被修改的，所以在使用数据之前都需要先加锁，锁未释放之前如果有其他线程想要修改数据（加写锁）就必须阻塞它。

悲观读锁并算不上绝对的悲观，排他锁才是真正的悲观锁，由于读锁具有读读共享的特性，所以对于读多写少的场景十分适用，可以大大提高并发性能。

21.4乐观读readLock

乐观读锁是一个共享锁，没有线程占用写锁的情况下，多个线程可以同时获取读锁。如果其他线程已经获得了写锁，也不阻塞当前线程，而是得到一个为0的邮戳；可以在获取到了乐观读锁并且没有释放的情况下，使用写锁（这是和悲观读锁最大的区别）。

tryOptimisticRead通过名字来记忆很简单，try代表尝试，说明**它是无阻塞的**。**Optimistic乐观的**， Read代表读锁。

乐观锁认为有极大可能数据在加上乐观读锁之后不会轻易的被修改，因此在操作数据前并没有加锁（使用cas方式更新锁的状态），而是采用试探的方式，只要当前没有写锁就可以获得一个非0的stamp，如果已经存在写锁则返回一个为0的stamp。

由于使用cas方法，没有真正的加锁，所以并发性能要比readLock还要高。

但是由于没有使用真正的锁，如果数据中途被修改，就会造成数据不一致问题。（从下面的代码中可以看出来：在使用了乐观读锁并且没有释放该锁的情况下，就可以使用写锁，这也是会造成数据不一致问题的原因，这也是叫做乐观读锁的原因，这也是和悲观读锁最大的区别）

但是**StampLock使用快照的方式**，需要复制一份要操作的变量到方法栈，操作的数据只是一个快照，从而保证了数据的最终一致性。但是读线程可能使用的数据不是最新的。特别适用于读多写少的高并发场景

Demo：
com.mkevin.demo9.StampedLockDemo3

```
public class StampedLockDemo3 {
    public static void main(String[] args) throws
InterruptedException {
        StampedLock sl = new StampedLock();
        //测试一：先获取乐观读锁，在不释放的情况下获取到写锁
        //乐观读
        long stamp = sl.tryOptimisticRead();

 System.out.println(Thread.currentThread().getName()+"
tryOptimisticRead stamp="+stamp);
        new Thread(new Runnable() {
            @Override
            public void run() {

 System.out.println(Thread.currentThread().getName()+"
run");
                long stamp131 = sl.writeLock();

 System.out.println(Thread.currentThread().getName()+" get
write lock1,stamp="+stamp131);
                sl.unlockWrite(stamp131);

 System.out.println(Thread.currentThread().getName()+"
unlock write lock1,stamp="+stamp131);
            }
        }).start();
        Thread.sleep(3000);
        //验证邮戳stamp指向的乐观读锁的代码~此处的过程中是否使用过写锁，返回false证明，在这个过程中用过写锁，如果在这个过程中使用过另一个读锁，那么没有问题，返回true
        boolean validate = sl.validate(stamp);

 System.out.println(Thread.currentThread().getName()+"
tryOptimisticRead validate="+validate);


        //推荐在使用乐观读锁的时候使用这种方式，使用乐观读锁之后及时验证
        stamp = sl.tryOptimisticRead();
```

```
 System.out.println(Thread.currentThread().getName()+"
tryOptimisticRead stamp="+stamp);
        //验证新获得的的邮戳指向的乐观读锁到此处的过程中是否有用过写
锁
        validate = sl.validate(stamp);

 System.out.println(Thread.currentThread().getName()+"
tryOptimisticRead validate="+validate);
    }



    //测试二：先获取到写锁，在不释放的情况下获取乐观读锁
            StampedLock sl = new StampedLock();
        long stamp1 = sl.writeLock();

 System.out.println(Thread.currentThread().getName()+"
writeLock stamp="+stamp1);
        //如果不加这段代码，对下面获取到的乐观读锁执行validate方法
的时候得到的值是false
        //提前关闭了写锁，那么在下面验证的时候得到的结果是true。
        //sl.unlockWrite(stamp1);

        //乐观读
        long stamp = sl.tryOptimisticRead();

 System.out.println(Thread.currentThread().getName()+"
tryOptimisticRead stamp="+stamp);
        System.out.println(sl.validate(stamp));
}
```

tryOptimisticRead与**validate**一定要紧紧挨着使用，否则在获取和验证之间很可能数据被修改（使用了写锁）。如果这期间锁发生变化（使用读锁）则**validate**返回**false**，否则返回**true**。

原理很简单，就是*我先尝试获取，这时候没有写锁我就拿到了一个锁，在我真正要使用的时候我再验证一下是否发生了改变，如果没有发生改变就可以安心使用。*

如果某个线程已经获取了写锁，这时候再尝试获取乐观锁也是可以获取的，只是得到的stamp为0，无法通过validate验证。

乐观读锁本质上并未加锁，而是提供了*获取和检测*的方法，由程序人员来控制该做些什么。

虽然性能大大提升，但是却增加了开发人员的复杂度，如果不是特别高的并发场景，对性能不要求极致，可以不考虑使用。

22. 锁的分类

22.1 五类锁

> *五类锁是并发编程的基础、线程安全的重要保障*
>
> *乐观锁/悲观锁*
>
> *是否在修改之前给记录增加排它锁*
>
> *公平锁/非公平锁*
>
> *请求锁的时间顺序是否与获得锁的时间顺序一致。一致为公平锁，不一致为非公平锁*
>
> *独占锁/共享锁*
>
> *是否可以被多个线程共同持有，可以则为共享锁、不可以则为独占锁*
>
> *可重入锁*
>
> *一个线程再次获取它自己已经获取的锁时是否会被阻塞，同时可以是独占锁、悲观锁*
>
> *自旋锁*
>
> *无法获取锁时是否立刻阻塞，还是继续尝试获取指定次数*

22.2 乐观锁与悲观锁

1. 乐观锁和悲观锁的概念来自于数据库
2. 悲观锁对数据被修改持悲观态度，**认为数据很容易就会被其他线程修改**，所以在**处理数据之前先加锁**，处理完毕释放锁。
3. 乐观锁对数据被修改持乐观态度，**认为数据一般情况下不会被其他线程修改**，所以在**处理数据之前不会加锁**，而是在数据进行更新时进行冲突检测。
4. 对于数据库的悲观锁就是排它锁，在处理数据之前，先尝试给记录加排它锁，如果成功则继续处理，如果失败则挂起或抛出异常，直到数据处理完毕释放锁。
5. 对于数据库的乐观锁锁典型的就是CAS方式更新, 例如：update name='kevin' where id=1 and name='kevin0'， 在更新数据的时候校验这个值是否发生了变化，类似于CAS的操作。

22.3 公平锁与非公平锁

1. 据线程获取锁的抢占机制，锁可以分为公平锁和非公平锁，**最早请求锁的线程将最早获取到锁**。而非公平锁则先请求不一定先得。JUC中的ReentrantLock提供了公平和非公平锁特性。·
2. 公平锁：ReentrantLock pairLock = new ReentrantLock(true)
3. 非公平锁：ReentrantLock pairLock = new ReentrantLock(false)，如果构造函数不传递参数，则默认是非公平锁。
4. 非必要情况下使用非公平锁，**公平锁存在性能开销**

22.4 独占锁与共享锁

1. 只能被单个线程所持有的锁是独占锁，可以被多个线程持有的锁是共享锁。
2. ReentrantLock就是以独占方式实现的，属于悲观锁
3. ReadWriteLock读写锁是以共享锁方式实现的，属于乐观锁
4. StampedLock的写锁，属于悲观锁。
5. StampedLock的乐观读锁，悲观读锁、属于乐观锁。

22.5 可重入锁

1. 当一个线程想要获取本线程已经持有的锁时，不会被阻塞，而是能够再次获得这个锁，这就是重入锁。
2. Synchornized是一种可重入锁，内部维护一个线程标志(谁持有锁），以及一个计数器（重复的次数是有上线的）。
3. ReentrantLock也是一种可重入锁
4. ReadWriteLock、StampedLock的读锁也是可重入锁

22.6 自旋锁

1. 当获取锁的时候如果发现锁已经被其他线程占有，则不阻塞自己，也不释放CPU使用权，而是尝试多次获取，如果尝试了指定次数之后仍然没有获得锁，再阻塞线程，典型的一种方式是使用CAS的方式去操作。
2. 自旋锁认为锁不会被长时间持有，使用CPU时间来换取线程上下文切换的开销，从而提高性能。但是可能会浪费CPU资源。
3. -XX:PreBlockSpin=n可以设置自旋次数（已经成为了历史），在Jdk7u40时被删除了，其实在jkd6的时候就已经无效了，现在HotSpotVM采用的是adaptive spinning（自适应自旋），虚拟机会根据情况来对每个线程使用不同的自旋次数。

关于线程池和并发设计模式章节的内容我移动到了《Java并发编程精通篇-2.md中》