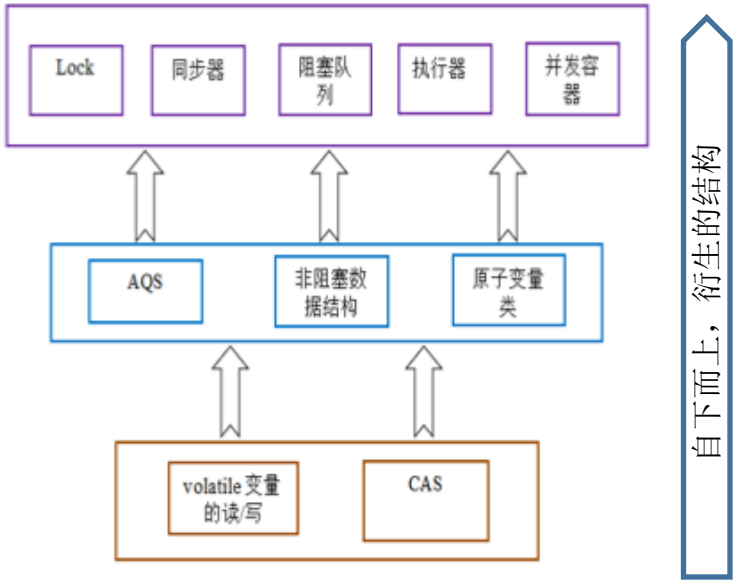
于丙林

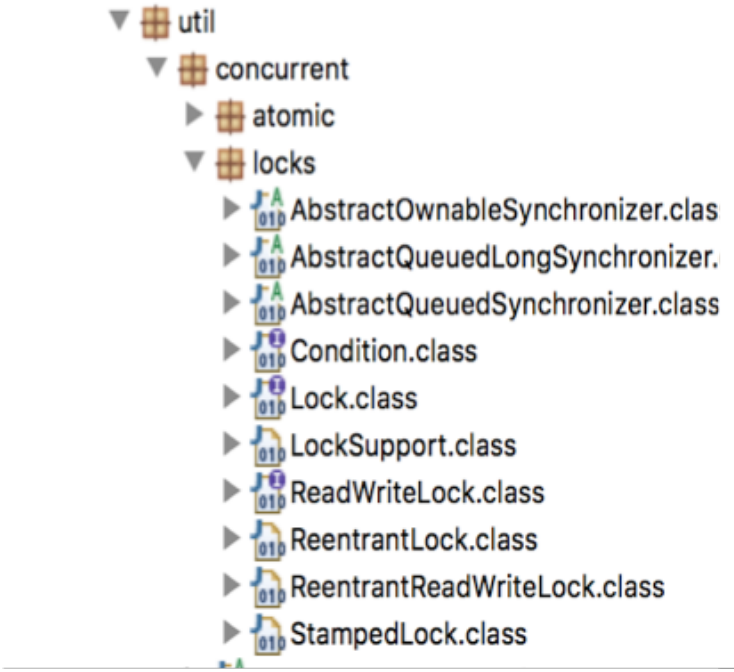
2018.11.19

目录

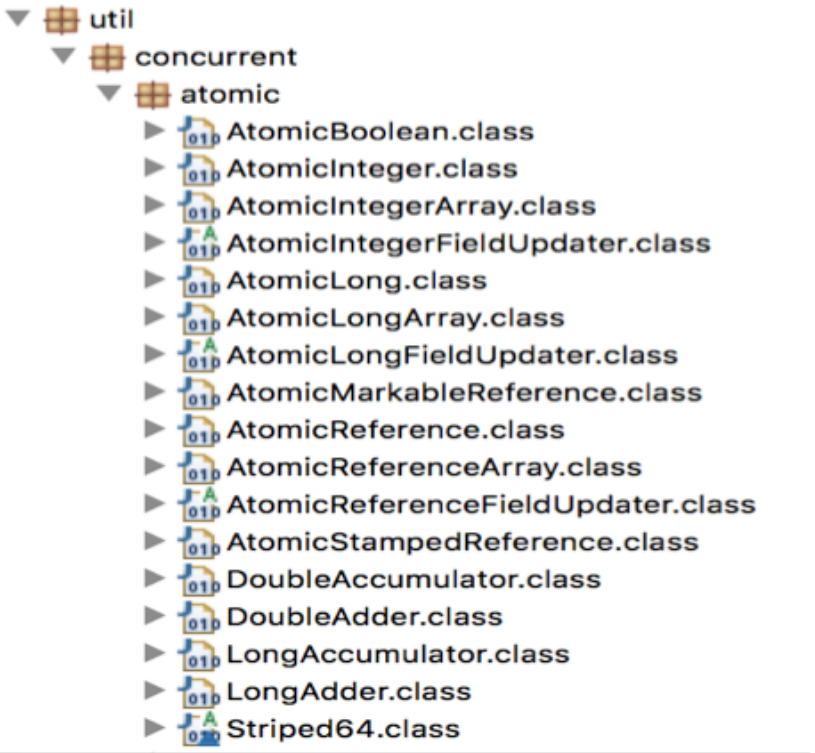
多线程切入方式



1）、Concurrent 包的实现示意图

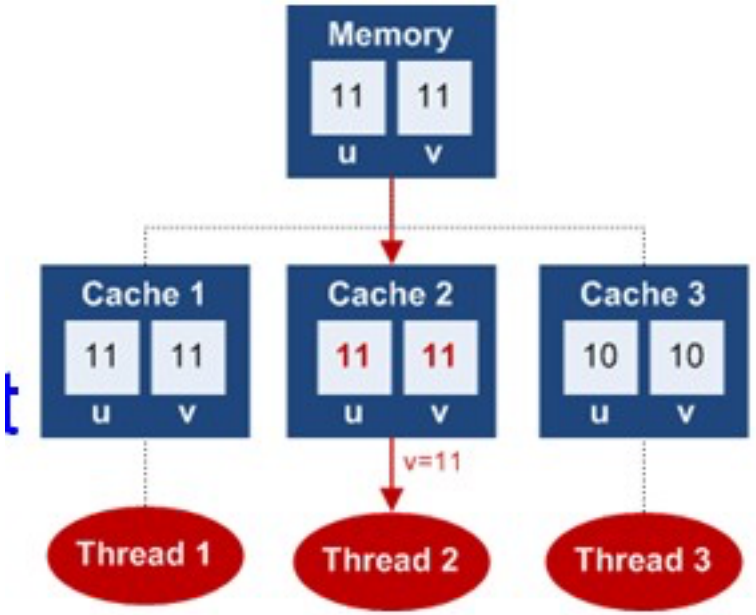


2）、Concurrent 包的类图

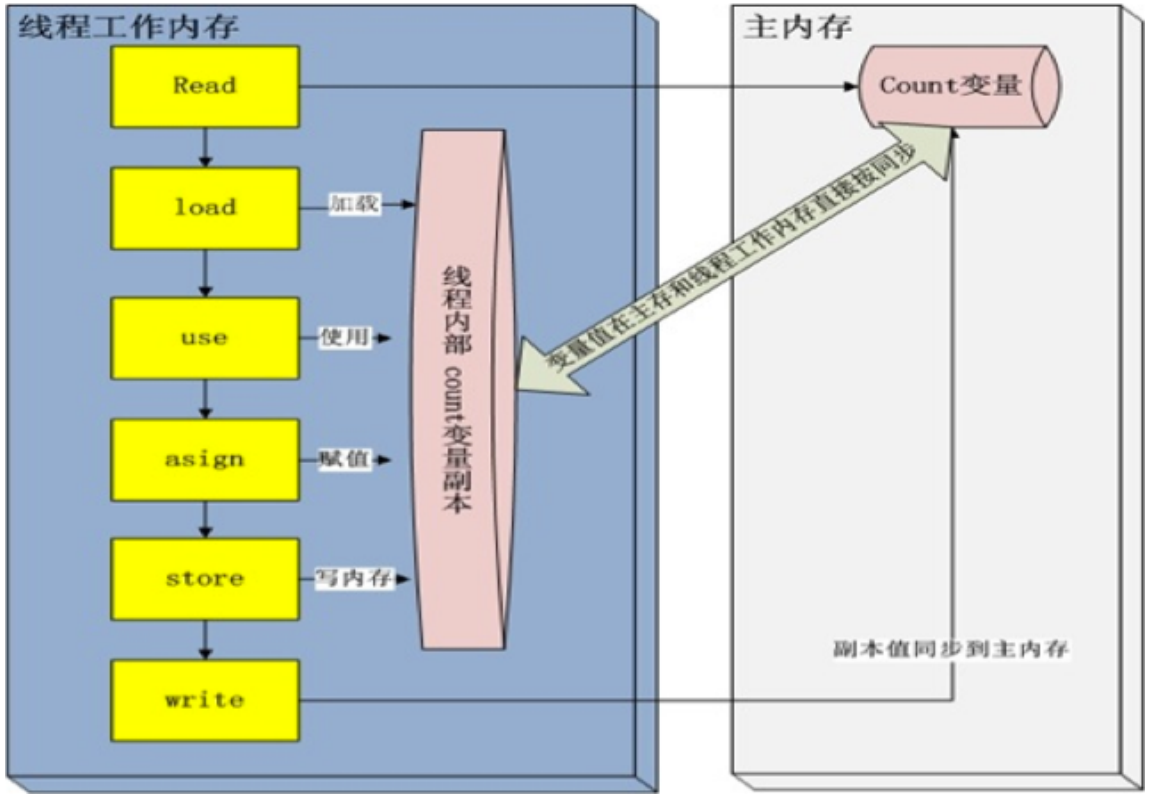


3）、原子操作类图

二、原子操作Vlatile可见性



线程、工作内存、主内存三者之间的交互关系图



所有线程共享主内存
每个线程有自己的工作内存
refreshing local memory to/from main memory must comply to JMM rules

Java代码：　instance = new Singleton();//instance是volatile变量
汇编代码：　0x01a3de1d: movb $0×0,0×1104800(%esi);0x01a3de24: **lock** addl $0×0,(%esp);

实例（原子操作Vlatile 可见性、高频特性）

```
//1、测试比较明显，没有volatile 会出现无限循环之中
private static volatile boolean bChanged;
public static void main(String[] args) throws InterruptedException {
/**
*此段线程证明，
*1、如果本地缓存存在一直从本地缓存获取，失效之后从远程缓存获取
*2、线程强制刷新本地缓存
*/
      new Thread() {
              @Override
              public void run() {
             for (;;) {
                 if (bChanged == !bChanged) {
                       System.out.println("!=");
                       System.exit(0);
                       }
              }
               }
      }.start();
      Thread.sleep(100);
      /**
       * 下面这段线程，不断修改变量值
      */
      new Thread() {
              @Override
              public void run() {
            for (;;) {
                bChanged = !bChanged;
            }
             }
      }.start();
}
```

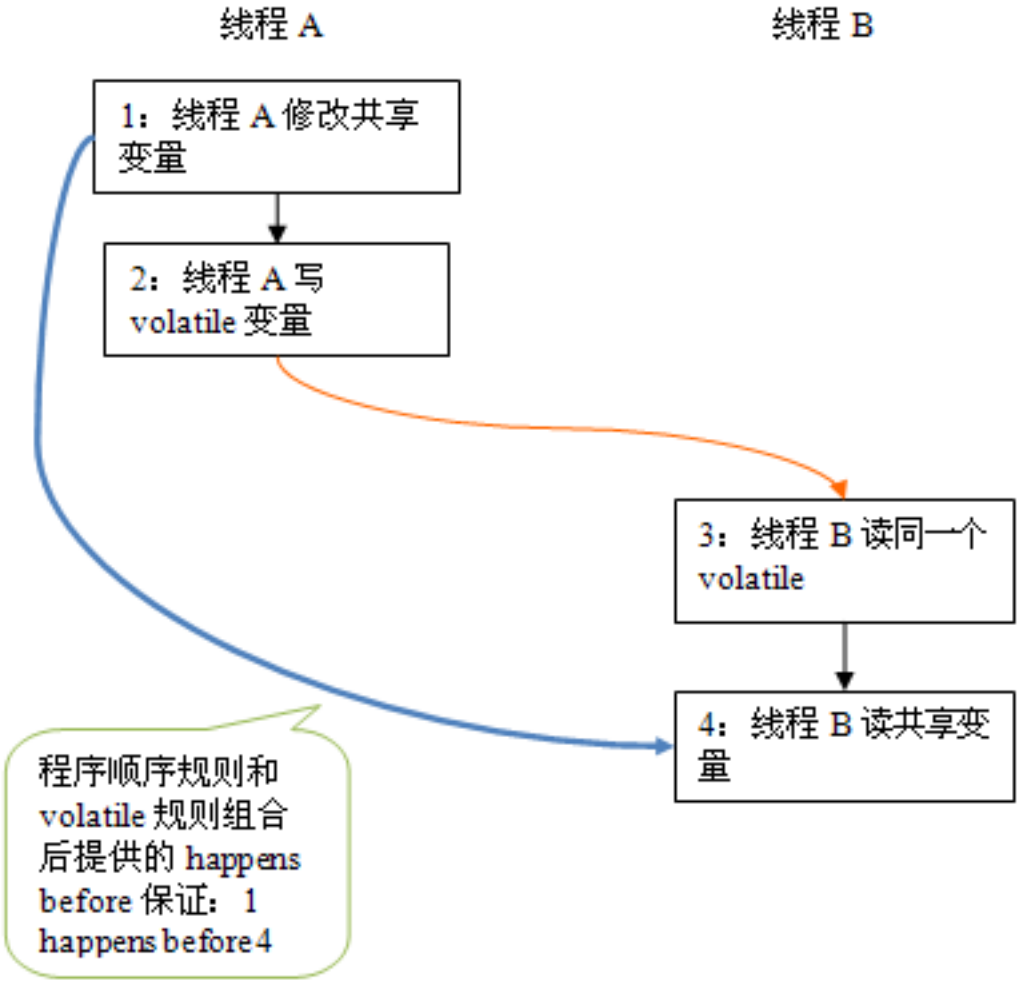
```
public class VolatileExample extends Thread {
            // 设置类静态变量,各线程访问这同一共享变量
            private  boolean flag = false;

            // 无限循环,等待flag变为true时才跳出循环
            public void run() {
                        while (!flag) {
                                    System.out.println(1);//关键点
                        }
            }

            public static void main(String[] args) throws Exception {
                        VolatileExample vol=new VolatileExample();
                        vol.start();
                        Thread.sleep(100);
                        vol.flag = true;
            }
}
```

原子操作Vlatile（happens before原则）

```
class VolatileExample {
    int a = 0;
    volatile boolean flag = false;
    public void writer() {
        a = 1;                  //1
        flag = true;            //2
    }
    public void reader() {
        if (flag) {             //3
            int i =  a;         //4
            ……
        }
    }
}
```



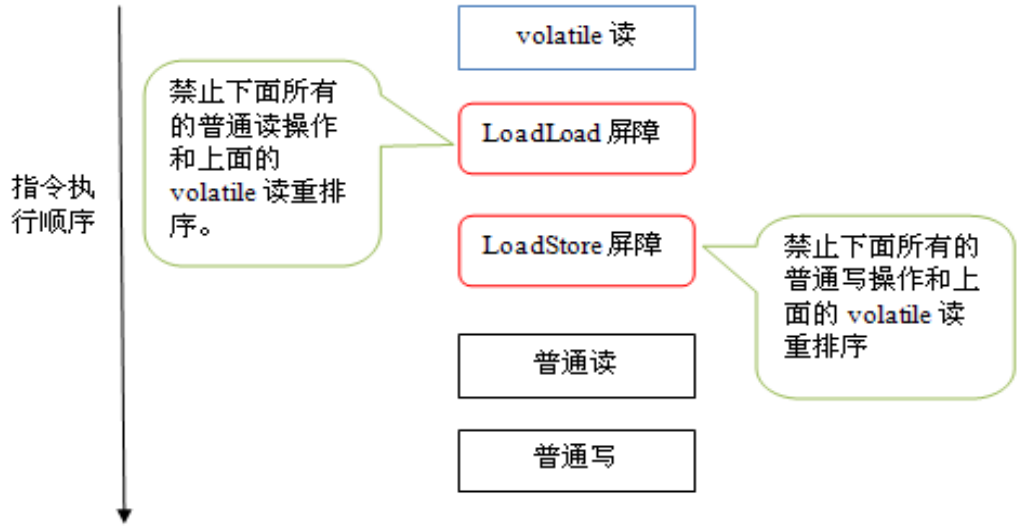
1根据程序次序规则，1 happens before 2; 3 happens before 4。
2根据volatile规则，2 happens before 3。
3根据happens before 的传递性规则，1 happens before 4。

原子操作Vlatile（重排序）

是否能重排序	第二个操作		
第一个操作	普通读/写	volatile读	volatile写
普通读/写			NO
volatile读	NO	NO	NO
volatile写		NO	NO

- 在每个volatile写操作的前面插入一个StoreStore屏障。
- 在每个volatile写操作的后面插入一个StoreLoad屏障。
- 在每个volatile读操作的后面插入一个LoadLoad屏障。
- 在每个volatile读操作的后面插入一个LoadStore屏障。

1）、排序规则

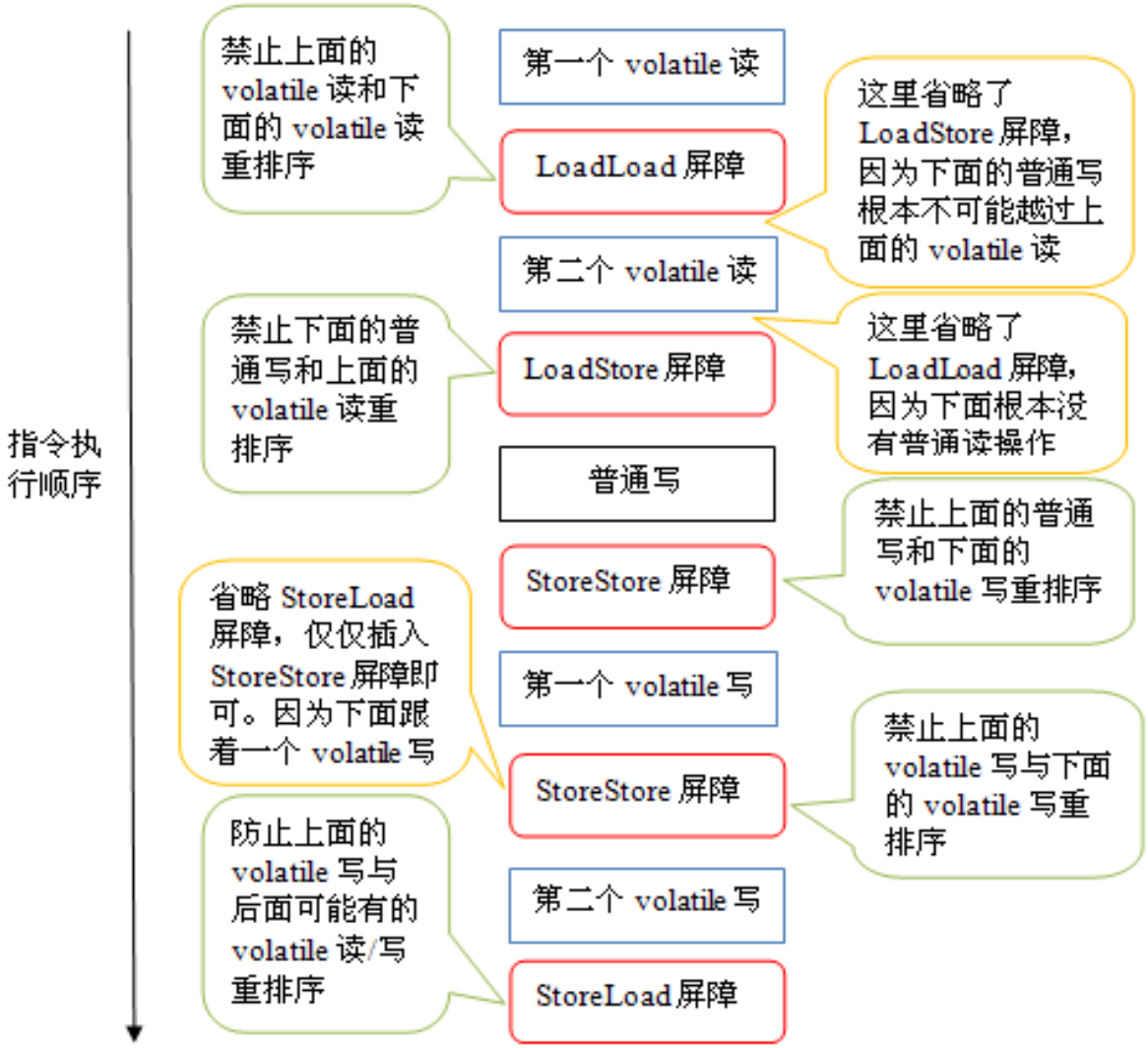


实例

```
class VolatileBarrierExample {
    int a;
    volatile int v1 = 1;
    volatile int v2 = 2;

    void readAndWrite() {
        int i = v1;                 //第一个
volatile读
        int j = v2;                 // 第二个
volatile读
        a = i + j;                  //普通写
        v1 = i + 1;                 // 第一个
volatile写
        v2 = j * 2;                 //第二个
volatile写
    }

    …                               //其他方法
}
```



实例

```
public class LazySingleton {
        private int someField;

        private static LazySingleton instance;

        private LazySingleton() {
                this.someField = new Random().nextInt(200) + 1; // (1)
        }

        public static LazySingleton getInstance() {
                if (instance == null) { // (2)
                        synchronized (LazySingleton.class) { // (3)
                                if (instance == null) { // (4)
                                        instance = new LazySingleton(); // (5)
                                }
                        }
                }
                return instance; // (6)
        }

        public int getSomeField() {
                return this.someField; // (7)
        }
}
```

A=1;

B=2;

C=3;

1. 给 singleton 分配内存
2. 调用 Singleton 的构造函数来初始化成员变量，形成实例
3. 将singleton对象指向分配的内存空间（执行完这步 singleton才是非 null了）

原子操作Vlatile的适用场景

并发专家建议我们远离volatile是有道理的，这里再总结一下：

volatile是在synchronized性能低下的时候提出的。如今synchronized的效率已经大幅提升，所以volatile存在的意义不大。

如今非volatile的共享变量，在访问不是超级频繁的情况下，已经和volatile修饰的变量有同样的效果了。

volatile不能保证原子性，这点是大家没太搞清楚的，所以很容易出错。

volatile可以禁止重排序。

所以如果我们确定能正确使用volatile，那么在禁止重排序时是一个较好的使用场景，否则我们不需要再使用它。这里只列举出一种volatile的使用场景，即作为标识位的时候(比如本文例子中boolean类型的flag)。用专业点更广泛的说法就是“对变量的写操作不依赖于当前值且该变量没有包含在其他具体变量的不变式中”，具体参见《Java 理论与实践: 正确使用 Volatile 变量》。

三、CAS 原子操作

CAS 重点三个数：

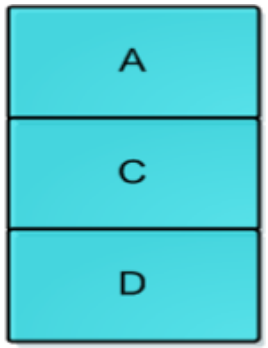
内存位置（V）、预期原值（A）和新值(B)

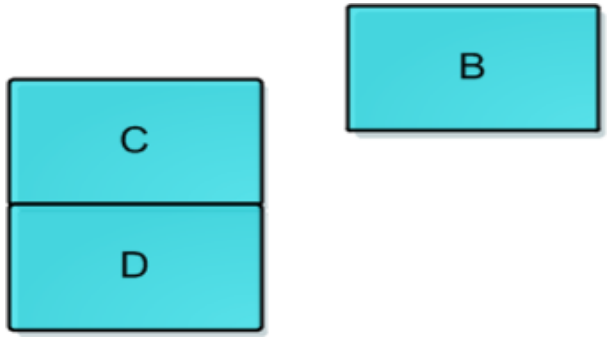
```
//如果当前值 ==预期值，则以原子方式将该值设置为给定的更新值。如果成功就返回，否则返回，并且不修改原值。
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

```
//以原子方式将当前值加 1相当于线程安全版本的++i操作。
 public final int incrementAndGet() {
     for (;;) {
         int current = get();
         int next = current + 1;
         if (compareAndSet(current, next))
             return next;
     }
 }
```

CAS ABA问题

图一 

图二 

图三 

注意：非单一化数据

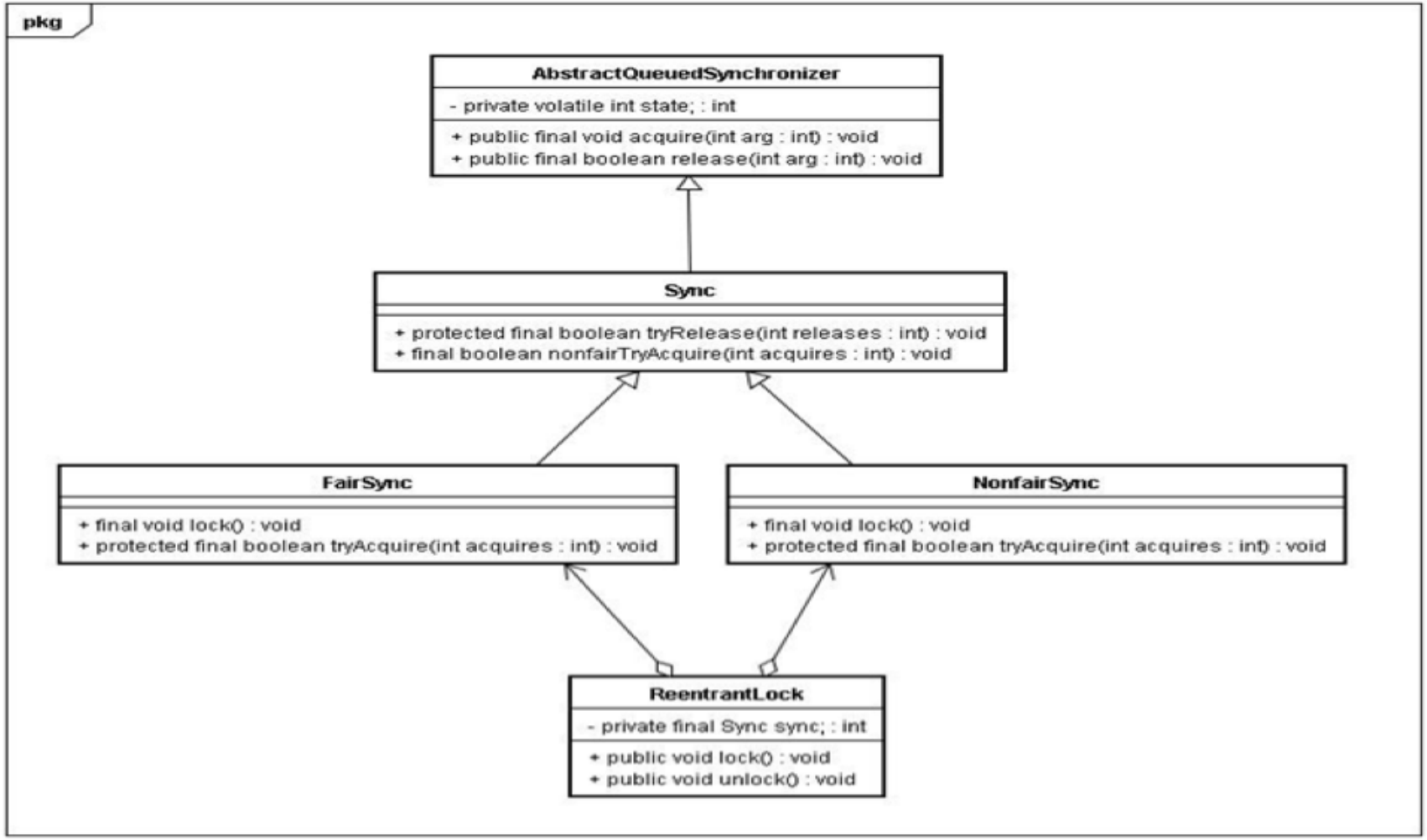
一、图一现有一个用单向链表实现的堆栈，栈顶为A，这时线程T1已经知道A.next为B，然后希望用CAS将栈顶替换为B：head.compareAndSet(A,B);
二、在T1执行上面这条指令之前，线程T2介入，将A、B出栈，再pushD、C、A，此时堆栈结构如图2，而对象B此时处于游离状态
三、此时轮到线程T1执行CAS操作，检测发现栈顶仍为A，所以CAS成功，栈顶变为B，但实际上B.next为null，所以此时的情况变为图3
四、其中堆栈中只有B一个元素，C和D组成的链表不再存在于堆栈中，平白无故就把C、D丢掉了

```
public class AtomicMarkableReference<V> {

    private static class Pair<T> {
        final T reference;
        final boolean mark;
```

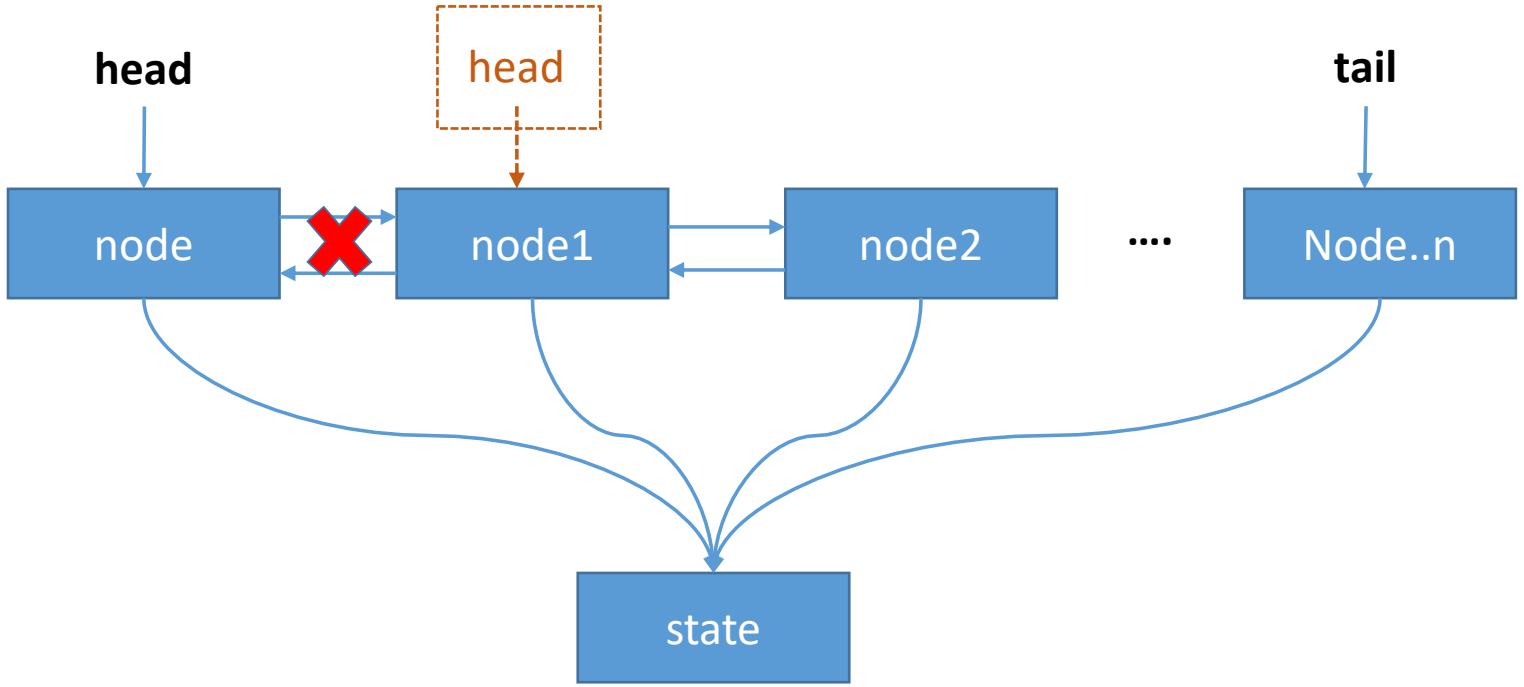
四、AQS

1、队列结构
2、线程模型
3、通知机制



AQS-CLH队列（FIFO）&Condition

1、volatile int state（代表共享资源）
2、FIFO线程等待队列
3、Condition 线程通信（wait、notify）



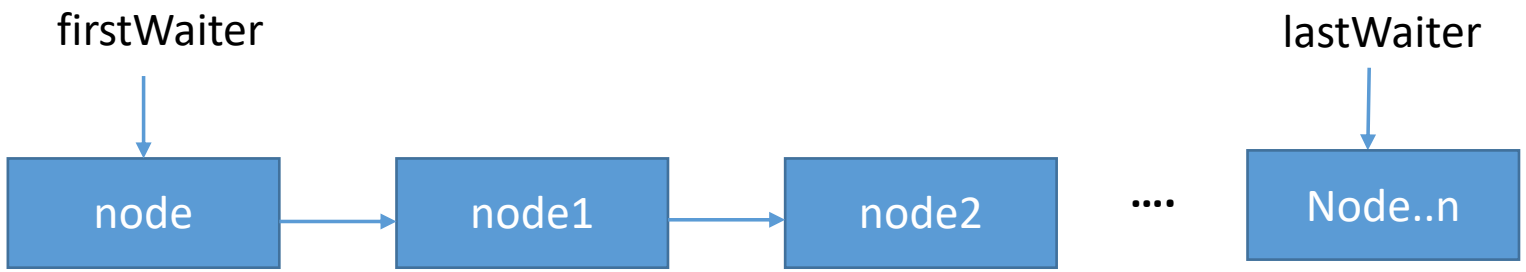
CLH整体队列结构

AQS-thread mode锁

- 一、***EXCLUSIVE***
  - *应用于锁，保证线程之间的排他性（**state**）*
- 二、***SHARED***
- ***1、申请锁时区别***
- **int r = tryAcquireShared(arg);**
- **boolean tryAcquire(arg)；**
  - 多个线程可以同时执行

AQS –通知机制（wait/notify）

- 1　 wait 状态添加到wait队列
- 2、notify 移除等待队列，加入aqs队列



Condition 队列结构(FIFO)

共享模式实例

```
ExecutorService exec = Executors.newCachedThreadPool();
        // 只能5个线程同时访问
        final Semaphore semp = new Semaphore(5);
        // 模拟20个客户端访问
        for (int index = 0; index < 20; index++) {
            final int NO = index;
            Runnable run = new Runnable() {
                public void run() {
                    try {
                        // 获取许可
                        semp.acquire();
                        System.out.println("Accessing: " + NO);
                        Thread.sleep((long) (Math.random() * 10000));
                        // 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
                        semp.release();
                    } catch (InterruptedException e) {
                    }
                }
            };
            exec.execute(run);
        }
        // 退出线程池
        exec.shutdown();
```

执行结果

```
Accessing: 0
Accessing: 4
Accessing: 2
Accessing: 3
Accessing: 1
Accessing: 5
Accessing: 6
Accessing: 7
Accessing: 8
Accessing: 9
Accessing: 10
Accessing: 11
Accessing: 12
Accessing: 13
Accessing: 14
Accessing: 15
Accessing: 16
Accessing: 17
Accessing: 18
Accessing: 19
```

AQS-ReentrantLock

- 1、可重入排它锁
- 同一个线程操作不需要入队，修改状态位，继续执行
- 2、 NonfairSync & FairSync
- 区别：线程请求获取锁的过程中，是否允许插队。
- 例如：非公平锁：加锁首先看状态位、然后再入队列；如果状态位满足直接执行，否则入队
- 效率较高，但是有可能导致队列线程等待时间较长
- 公平锁：队列为空，状态为满足才执行，否则入队
- 效率相对较低，但是线程运行公平对待

NonfairSync & FairSync

1、竞争之前是否统一入队，公平操作

a、非公平锁

```
final void lock() {
            //CAS 成功返回，不需要入队操作
            if (compareAndSetState(0, 1))
                setExclusiveOwnerThread(Thread.currentThread());
            else
                acquire(1);
}

final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            //当前线程修改状态位置即可
            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;
}
```

b、公平锁

```
 final void lock() {
            acquire(1);
 }
protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                //判断队列是否为null
                if (!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;
}
```

syn与reentrantLock 性能对比

图 1. synchronized 和 Lock 的吞吐率，单 CPU

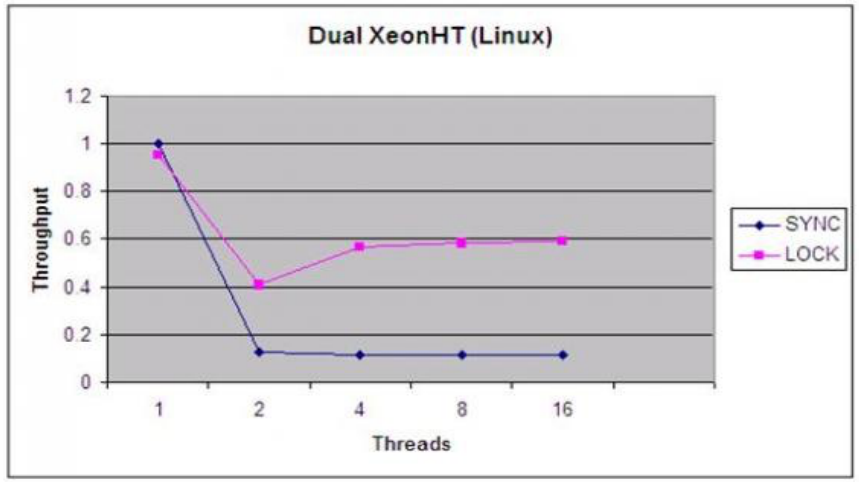


图 2. synchronized 和 Lock 的吞吐率（标准化之后），4 个 CPU

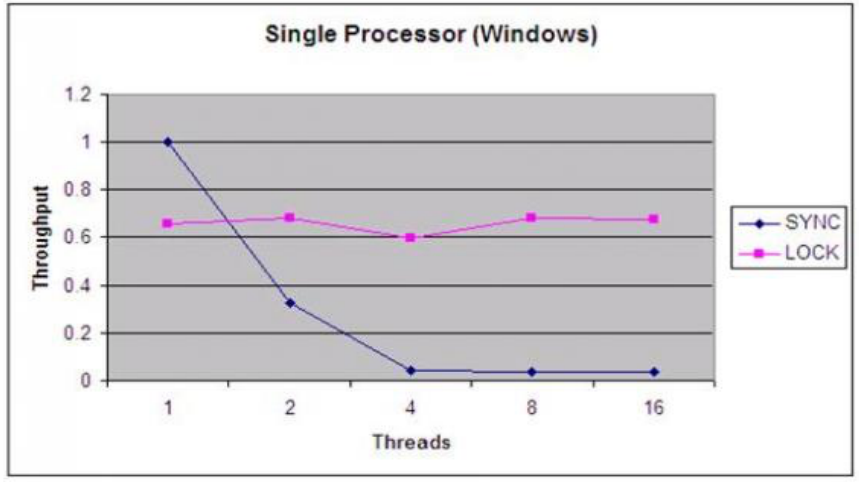


图 3. 使用 4 个 CPU 时的同步、协商锁和公平锁的相对吞吐率

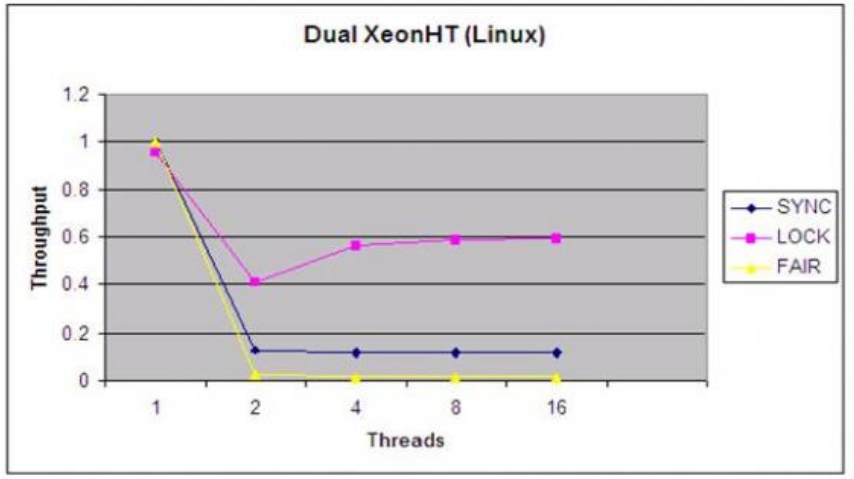
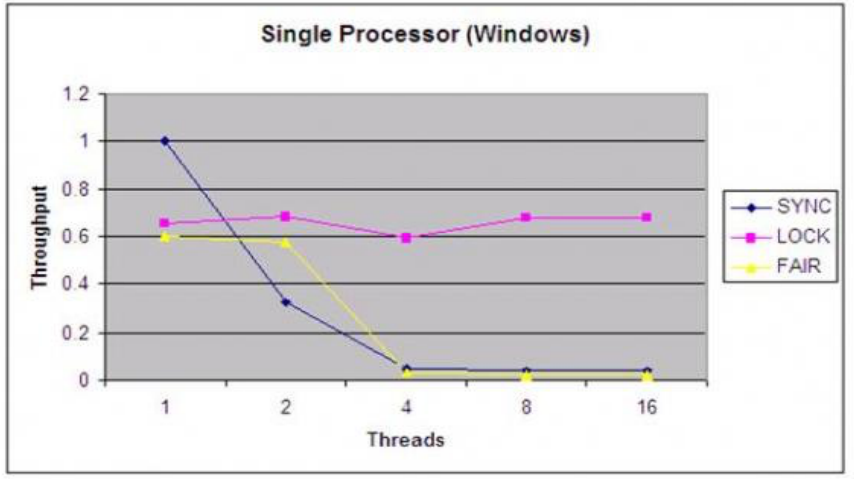


图 4. 使用 1 个 CPU 时的同步、协商和公平锁的相对吞吐率



Q&A


评