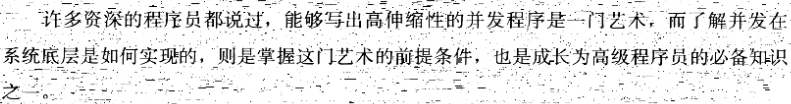
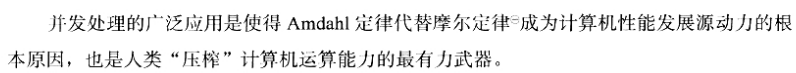
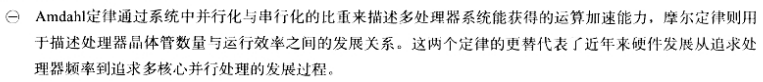
**并发编程**

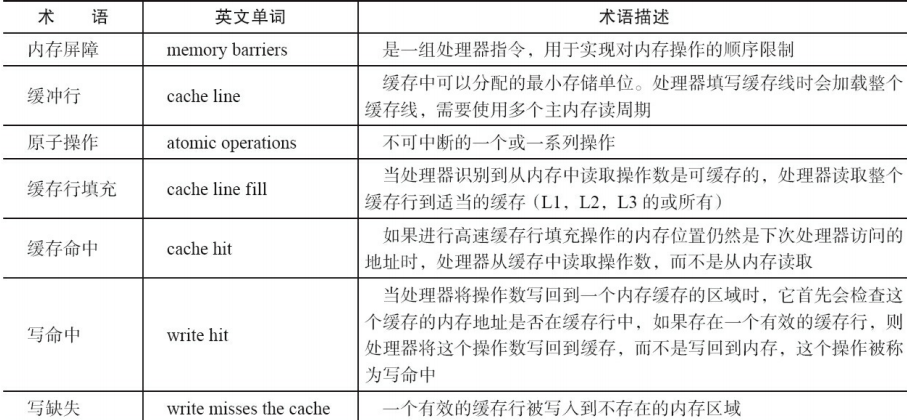
****

# Java内存模型

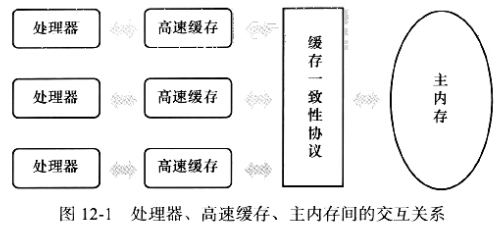


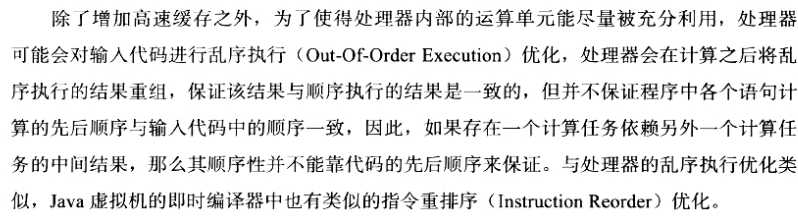


## Cpu术语



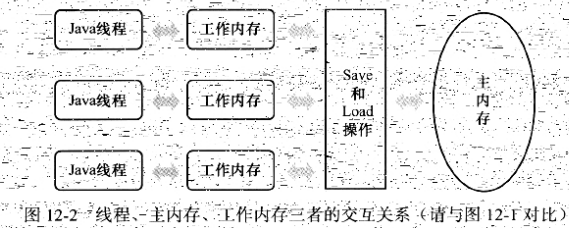
## 硬件架构



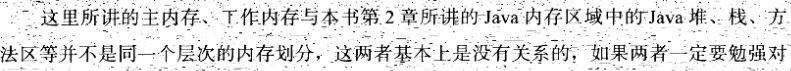


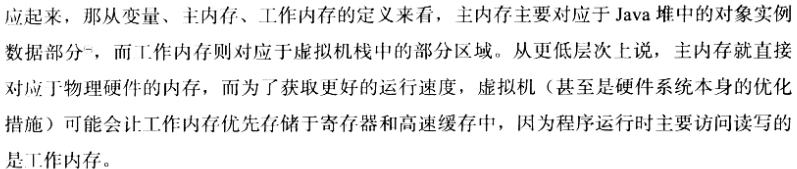


## Jvm架构

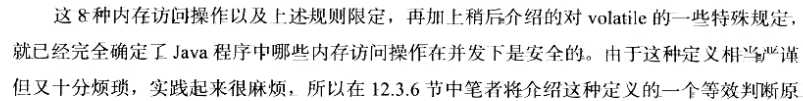






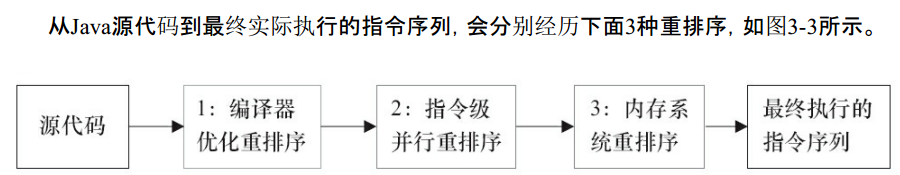


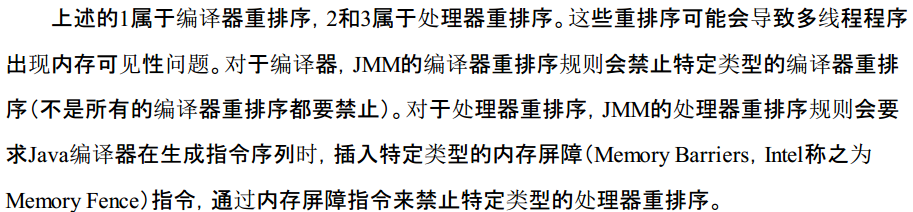
## 内存交互协议





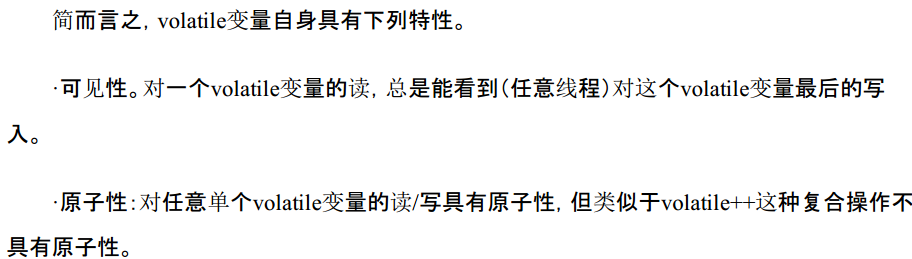
## 重排序

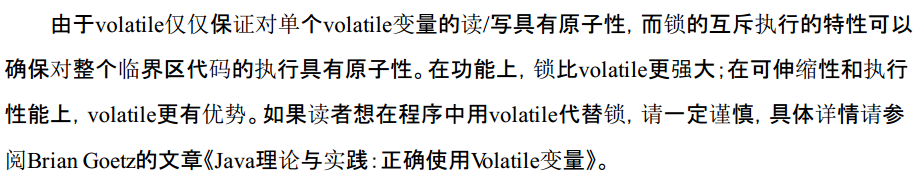


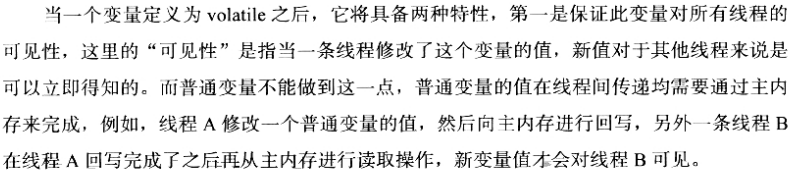




## Volitile语义

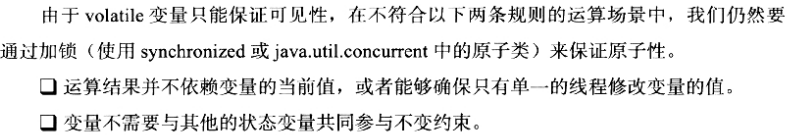




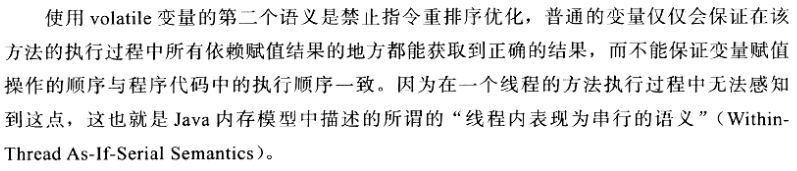


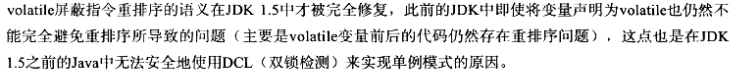
|  |
| --- |
| **package** com.wangguo.jvm;  **public** **class** VolatileTest {  **public** **static** **volatile** **int** *race* = 0;  **public** **static** **void** increase() {  *race*++;  }  **private** **static** **final** **int** ***THREADS\_COUNT*** = 20;  **public** **static** **void** main(String[] args) {  Thread[] threads = **new** Thread[***THREADS\_COUNT***];  **for** (**int** i = 0; i < ***THREADS\_COUNT***; i++) {  threads[i] = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 10000; i++) {  *increase*();  }  }  });  threads[i].start();  }  // 等待所有累加线程都结束  **while** (Thread.*activeCount*() > 1)  Thread.*yield*();  System.***out***.println(*race*);  }  } |

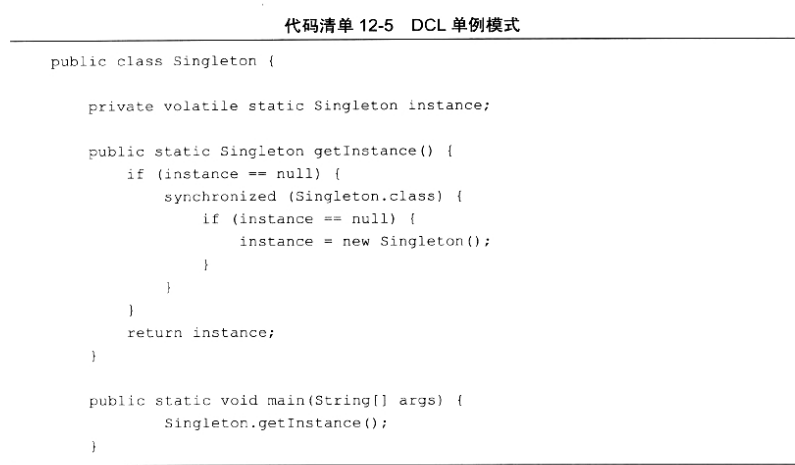
**多次运行值小于200000！！！**



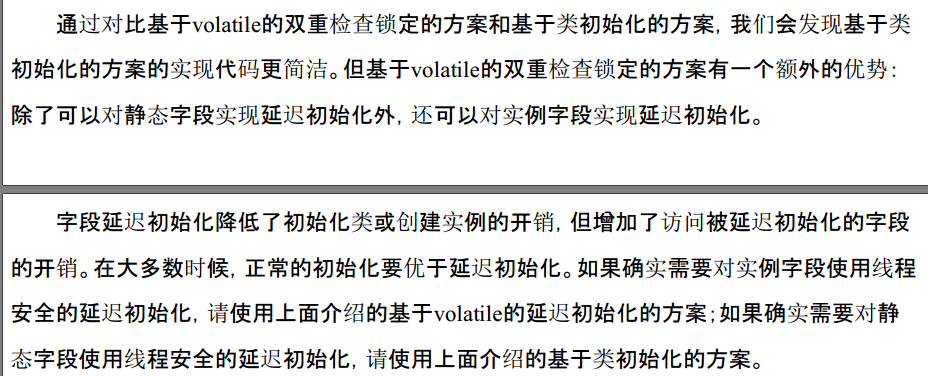
**对于“可见性”的理解，请不要太执着于书本的意思，这里只要简单理解为：用volatile修饰的变量，每次进行读取操作时都必须去主存进行读取，而普通变量则不一定读取的是主存数据，有可能是缓存中的过期数据！！同理写操作时也一样，在本地缓存修改完后，会立即同步主存中的内容，其他线程就能读取最新的修改值！！**



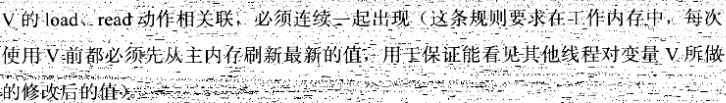


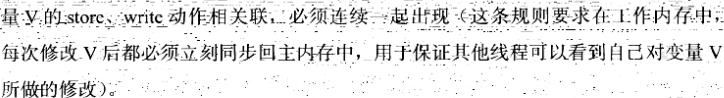


|  |
| --- |
| public class Singleton {  private volatile static Singleton instance;  public static Singleton getInstance() {  if (instance == null) {  synchronized (Singleton.class) {  if (instance == null) {  instance = new Singleton();  }  }  }  return instance;  }  public static void main(String[] args) {  Singleton.getInstance();  }  } |

****

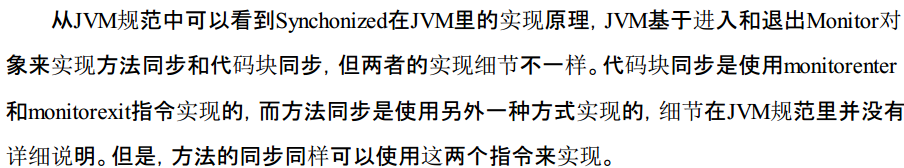
**总结三条规则：**

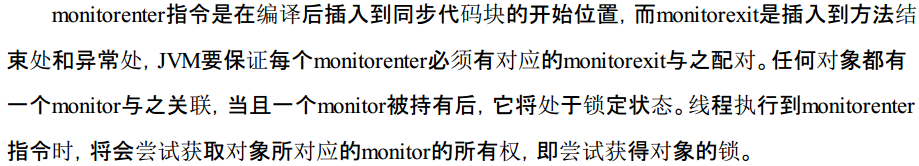


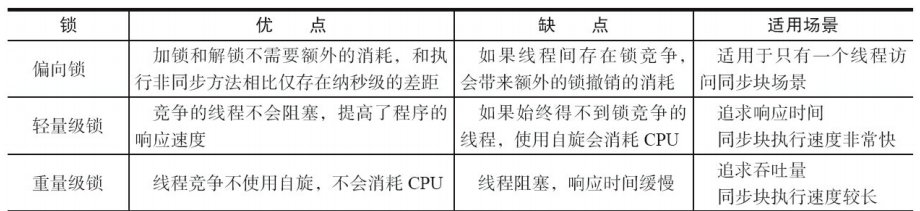




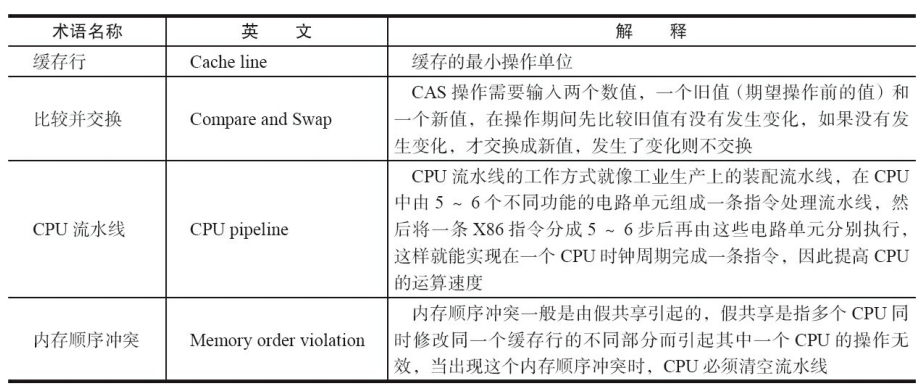
## Synchronized

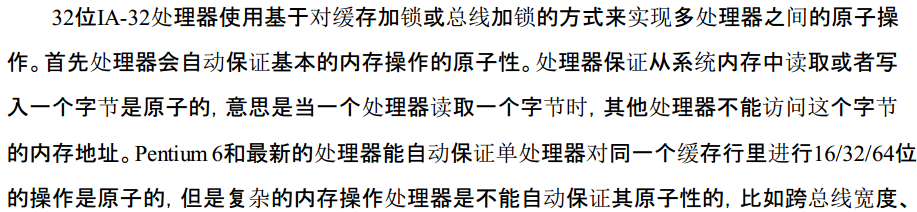


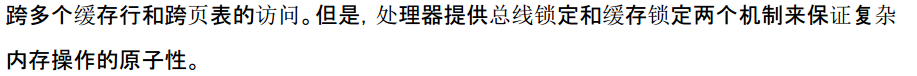




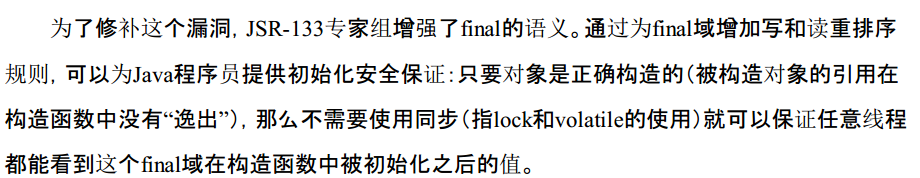
## Atomic



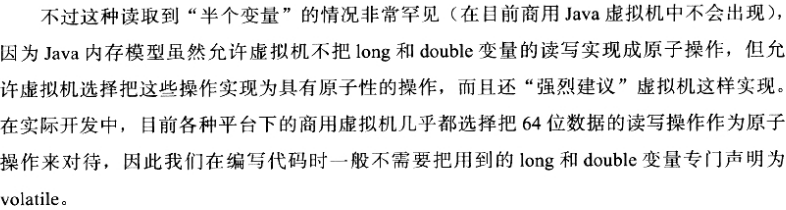




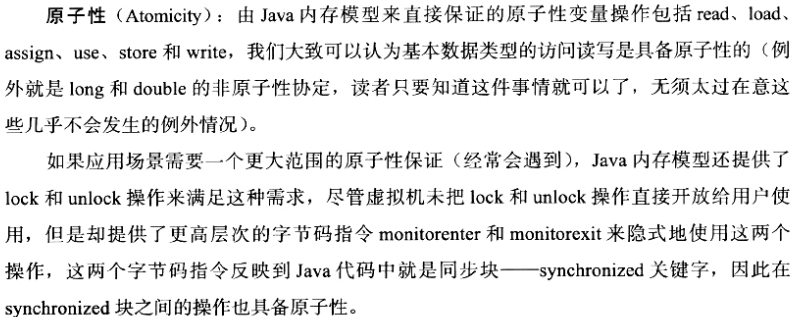
## Final

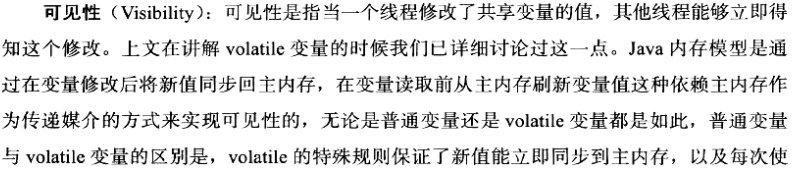


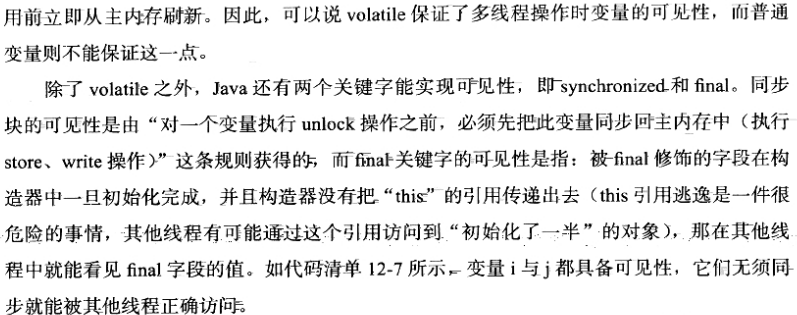
## Long与double类型

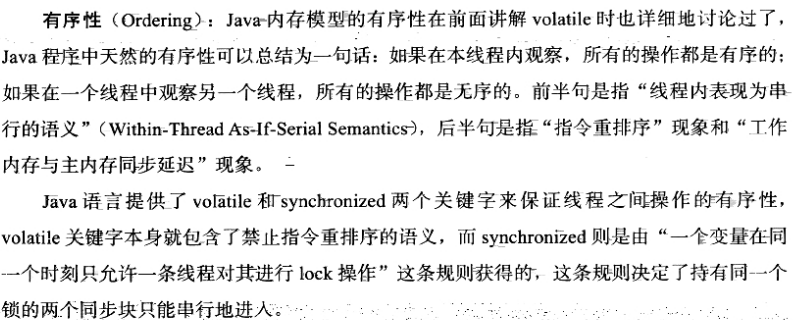


## 原子性、可见性、有序性



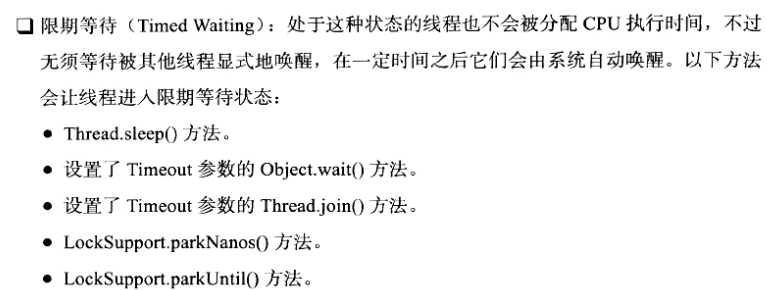


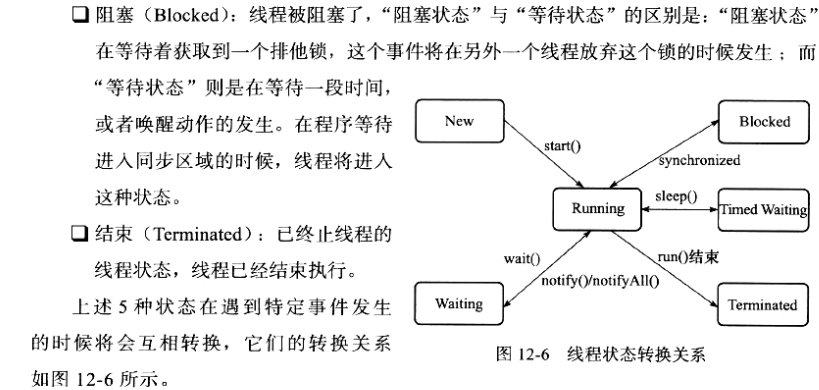




# Java线程

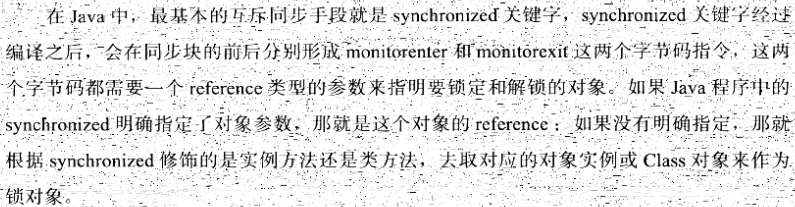


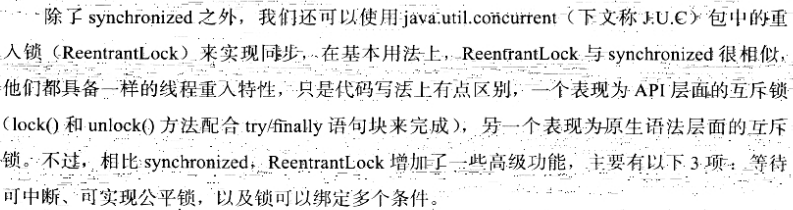




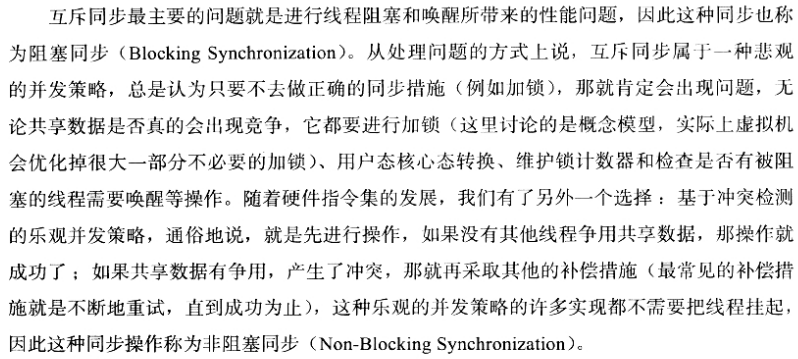
# 线程安全

## 互斥同步

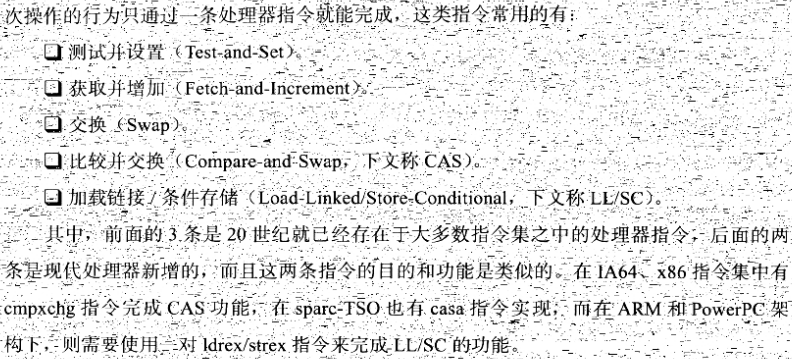




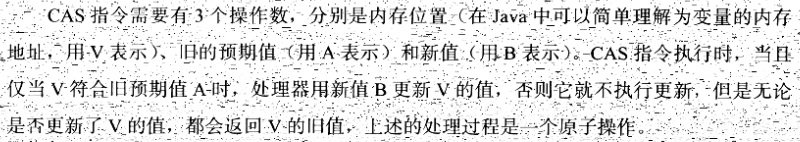
## 非阻塞同步

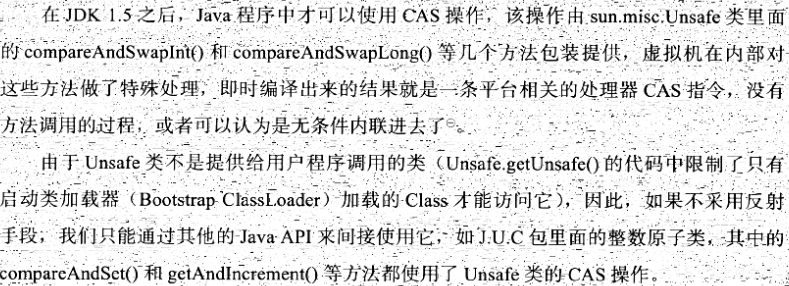


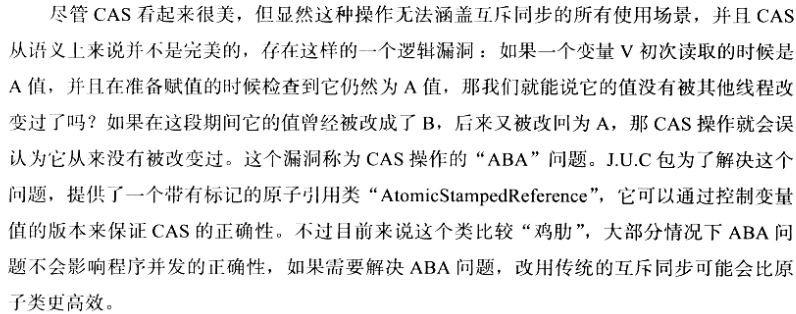
**前提是硬件处理器指令的支持！！-------CAS指令**



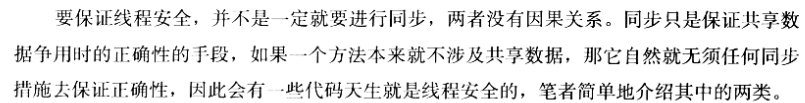
**CAS指令**

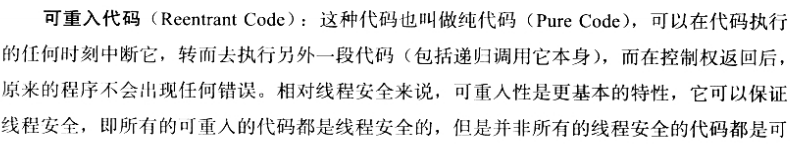
****

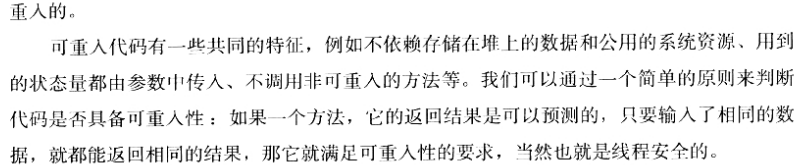
****

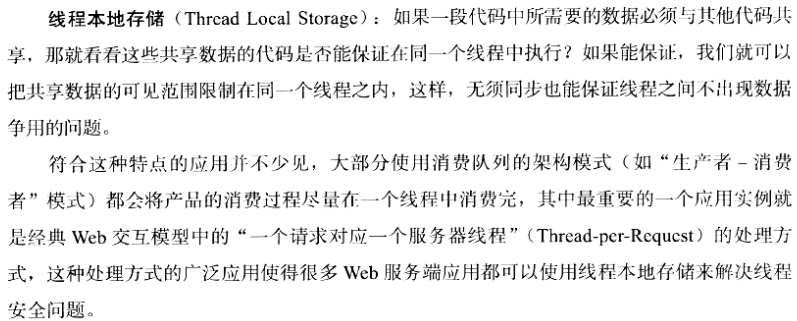
****

## 无同步方案

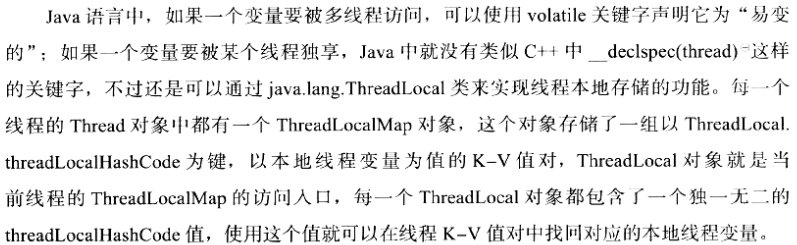




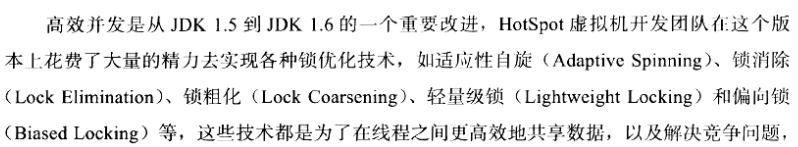




**Threadlocal**

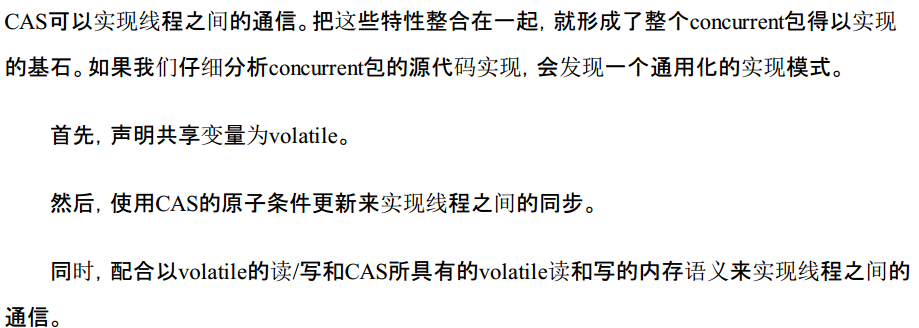


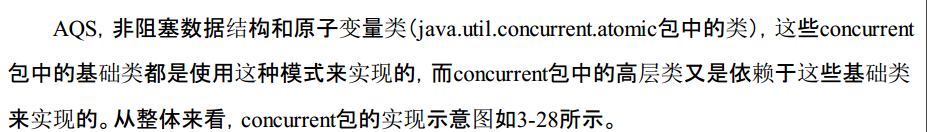
# 锁优化

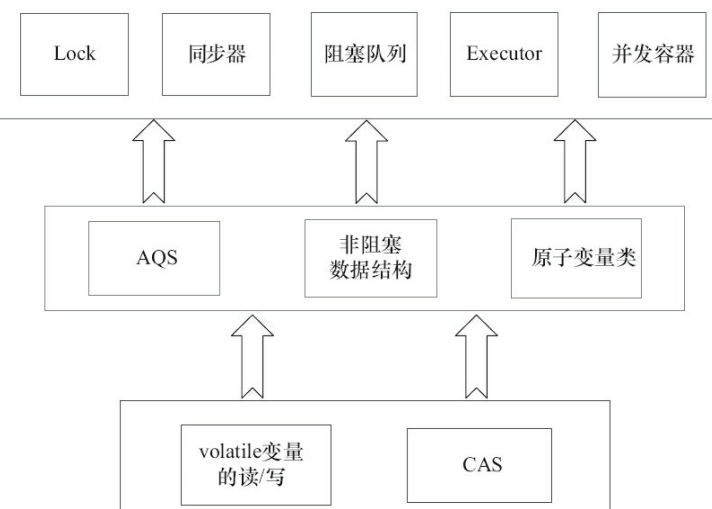


# 并发编程实战

## Concurrent的由来







## Synchronized

## Volatile

## Threadlocal

## Atomic\*原子类

## 线程池（Executorservice）

## Callable、Future

## Reenterlock

## **ReadWriteLock**

## Condition

多路阻塞：阻塞队列实现

|  |
| --- |
| * class BoundedBuffer { * **final Lock lock = new ReentrantLock();** * final Condition notFull = **lock.newCondition();** * final Condition notEmpty = **lock.newCondition();** * final Object[] items = new Object[100]; * int putptr, takeptr, count; * public void put(Object x) throws InterruptedException { * **lock.lock();** * **try {** * while (count == items.length) * **notFull.await();** * items[putptr] = x; * if (++putptr == items.length) putptr = 0; * ++count; * **notEmpty.signal();** * **} finally {** * **lock.unlock();** * **}** * } * public Object take() throws InterruptedException { * **lock.lock();** * **try {** * while (count == 0) * **notEmpty.await();** * Object x = items[takeptr]; * if (++takeptr == items.length) takeptr = 0; * --count; * **notFull.signal();** * return x; * **} finally {** * **lock.unlock();** * **}** * } * } |

## Semaphore

|  |
| --- |
| import java.util.concurrent.ExecutorService;  4 import java.util.concurrent.Executors;  5 import java.util.concurrent.Semaphore;  6  7 public class SemaphoreTest {  8 public static void main(String[] args) {  9 ExecutorService service = Executors.newCachedThreadPool();  10 final Semaphore sp = new Semaphore(3);//创建Semaphore信号量，初始化许可大小为3  11 for(int i=0;i<10;i++){  12 try {  13 Thread.sleep(100);  14 } catch (InterruptedException e2) {  15 e2.printStackTrace();  16 }  17 Runnable runnable = new Runnable(){  18 public void run(){  19 try {  20 sp.acquire();//请求获得许可，如果有可获得的许可则继续往下执行，许可数减1。否则进入阻塞状态  21 } catch (InterruptedException e1) {  22 e1.printStackTrace();  23 }  24 System.out.println("线程" + Thread.currentThread().getName() +  25 "进入，当前已有" + (3-sp.availablePermits()) + "个并发");  26 try {  27 Thread.sleep((long)(Math.random()\*10000));  28 } catch (InterruptedException e) {  29 e.printStackTrace();  30 }  31 System.out.println("线程" + Thread.currentThread().getName() +  32 "即将离开");  33 sp.release();//释放许可，许可数加1  34 //下面代码有时候执行不准确，因为其没有和上面的代码合成原子单元  35 System.out.println("线程" + Thread.currentThread().getName() +  36 "已离开，当前已有" + (3-sp.availablePermits()) + "个并发");  37 }  38 };  39 service.execute(runnable);  40 }  41 }  42  43 } |

## CyclicBarrier

## **CountDownLatch**

|  |
| --- |
| * class Driver { // ... * void main() throws InterruptedException { * CountDownLatch startSignal = new CountDownLatch(1); * CountDownLatch doneSignal = new CountDownLatch(N); * for (int i = 0; i < N; ++i) // create and start threads * new Thread(new Worker(startSignal, doneSignal)).start(); * doSomethingElse(); // don't let run yet * startSignal.countDown(); // let all threads proceed * doSomethingElse(); * doneSignal.await(); // wait for all to finish * } * } * class Worker implements Runnable { * private final CountDownLatch startSignal; * private final CountDownLatch doneSignal; * Worker(CountDownLatch startSignal, CountDownLatch doneSignal) { * this.startSignal = startSignal; * this.doneSignal = doneSignal; * } * public void run() { * try { * startSignal.await(); * doWork(); * doneSignal.countDown(); * } catch (InterruptedException ex) {} // return; * } * void doWork() { ... } * } |

## Exchanger

|  |
| --- |
| * class FillAndEmpty { * Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>(); * DataBuffer initialEmptyBuffer = ... a made-up type * DataBuffer initialFullBuffer = ... * class FillingLoop implements Runnable { * public void run() { * DataBuffer currentBuffer = initialEmptyBuffer; * try { * while (currentBuffer != null) { * addToBuffer(currentBuffer); * if (currentBuffer.isFull()) * currentBuffer = exchanger.exchange(currentBuffer); * } * } catch (InterruptedException ex) { ... handle ... } * } * } * class EmptyingLoop implements Runnable { * public void run() { * DataBuffer currentBuffer = initialFullBuffer; * try { * while (currentBuffer != null) { * takeFromBuffer(currentBuffer); * if (currentBuffer.isEmpty()) * currentBuffer = exchanger.exchange(currentBuffer); * } * } catch (InterruptedException ex) { ... handle ...} * } * } * void start() { * new Thread(new FillingLoop()).start(); * new Thread(new EmptyingLoop()).start(); * } * } |

## BlockingQueue

|  |
| --- |
| * class Producer implements Runnable { * private final BlockingQueue queue; * Producer(BlockingQueue q) { queue = q; } * public void run() { * try { * while (true) { queue.put(produce()); } * } catch (InterruptedException ex) { ... handle ...} * } * Object produce() { ... } * } * class Consumer implements Runnable { * private final BlockingQueue queue; * Consumer(BlockingQueue q) { queue = q; } * public void run() { * try { * while (true) { consume(queue.take()); } * } catch (InterruptedException ex) { ... handle ...} * } * void consume(Object x) { ... } * } * class Setup { * void main() { * BlockingQueue q = new SomeQueueImplementation(); * Producer p = new Producer(q); * Consumer c1 = new Consumer(q); * Consumer c2 = new Consumer(q); * new Thread(p).start(); * new Thread(c1).start(); * new Thread(c2).start(); * } * } |

## ConcurrentHashMap

## CopyOnWriteArrayList

## CopyOnWriteArraySet