**Volatile**

Volatile有两个作用:

1 保证内存可见性

内存可见性指的是: 一个线程修改了这个变量的值，另一个线程能立刻看到。

2 禁止指令重排(有序性)

由于cpu在运行时，可能会根据上下文信息对指令做一些重排序，导致执行的顺序和我们期望的不一样，加了volatile之后，cpu将会取消对该变量的重排优化，保证运行顺序和我们代码期望的一样。

volatile最常用在DCL单例中，volatile修饰的变量读操作跟普通的变量几乎没有区别，但是写操作效率会低一些(因为需要加内存屏障)，但是仍然比锁要高效。volatile不能保证原子性，如果volatile可以满足需求，采用volatile是最好的选择，如果不能满足，就应该使用锁。

volatile能保证"可见性"和"有序性"，但不能保证"原子性"，volatile可以用来修饰静态变量和成员变量。

### synchronized

volatile可以保证"可见性"和"有序性"，但不能保证"原子性"，synchronized就能同时保证"可见性"、"有序性"和"原子性"。

synchronized可以修饰: "成员方法"、"静态方法" 和 "代码块"，修饰成员方法时候，锁对象是当前对象this；修饰静态方法时，锁对象是当前class对象；修饰代码块时，锁对象是指定的对象。

synchronized是可重入的，也就是说，如果一个线程拥有了一个synchronized方法需要的锁，那么它调用另一个需要同样锁的方法，可以直接进入。但是一定要区分this锁(成员方法)和.class锁(静态方法)

synchronized使用不注意的话，可能会引起死锁，看下面demo:

public void test1(){

//持有lock1

synchronized(lock1) {

doSomething();

//请求lock2

synchronized(lock2) {

}

}

}

public void test2(){

//持有lock2

synchronized(lock2) {

doSomething();

//请求lock1

synchronized(lock1) {

}

}

}

如上，test1()和test2()分别跑在两个线程T1和T2，其中T1持有lock1，T2持有lock2，这样T1在等T2的lock2，但是T2需要lock1才能释放lock2，lock1又被T1占有，互不相让，就陷入了死锁，开发中要避免这种问题，尽量不要同时占有两把锁。

synchronized不响应中断:

Thread thread = new Thread {

run(){

synchronized(lock) {

}

}

}

//启动线程

thread.start();

//中断线程，假设此时线程没获取到lock，处于BLOCKED状态

thread.interrupt();

如上，我们调用了thread.interrupt()后线程仍然处于BLOCKED状态，并没有中断，这是synchronized的缺点。

**锁的分类**

**悲观锁和乐观锁**: 悲观锁指的是，每次都假设会发生竞争，所以每次都请求锁，会陷入阻塞，效率低，synchronized就是悲观锁；乐观锁是基于CAS，线程会自己循环检测条件条件是否满足，不会阻塞，效率高，比如可重入锁，读写锁等，都是乐观锁。

**可重入锁和不可重入锁**: 可重入锁指的是，如果已经持有需要的锁，再次进入需要同样锁的代码不需要申请锁；不可重入锁则每次进入需要相同锁的代码都需要重新申请锁。java中的锁大都是可重入锁。

**共享锁和独占锁**: 共享锁指一把锁可以被多个线程共享，独占锁指的是一把锁只有一个线程能够占用，最常见的共享锁就是读写锁中的读锁，其他锁大都是独占锁。

**自旋锁和非自旋锁**: 自旋锁指的是如果一个线程拿不到锁，那么会循环尝试，这样可以避免陷入阻塞态，但是如果很长时间都拿不到，那么就白自旋，浪费cpu，所以jdk1.6升级了自旋锁为自适应自旋，每次自旋都会延长时间，从而避免浪费cpu；非自旋锁获取不到就直接陷入阻塞态。

**公平锁和非公平锁**: 公平锁指的是谁先申请锁就让谁先获取锁，严格排队执行；非公平锁指的是谁可以执行就让谁执行，不保证顺序，但是效率更高，比如A先申请，B后申请，但是A在wait()状态，我们可以让B先执行，而不是让B还在傻等，所以效率更高。可重入ReentrantLock的构造函数就可以指定是否公平(fair)。

**可中断锁和不可中断锁**: 可中断锁指的是响应中断的，比如可重入锁ReentrantLock；不可重入锁指的是不响应中断的，比如synchronized。

**偏向锁、轻量级锁和重量级锁**

 **偏向锁**: 会偏向第一个获取它的线程，如果接下来的过程中，该锁没有被其他的线程获取过，则持有偏\* 向锁的线程将不再需要同步，当有竞争时会进化为轻量级锁。

 **轻量级锁**: 采用CAS自旋来获取锁，不会阻塞，开销很小，当存在竞争时，会进化为重量级锁。

 **重量级锁**: 就是一般的synchronized，遇到就阻塞等待。

**JVM对synchronized的优化**

 1 进化: 无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁，系统会根据竞争程度对synchronized进行进化，一步一步从轻量级进化为重量级。

 2 锁消除: 如果有不必要的锁，那么会进行消除，比如对不存在竞争的局部变量加锁，那么会在编译期去除，从而提升效率。

 3 锁粗化: 如果有在短期内进行大量加锁的碎片代码，那么会直接进行合并，从而避免不必要的加锁操作。

public void test(){

Object lock = new Object();

//这个不需要加锁，因为是局部变量，且不存在逃逸，所以jvm在编译期会进行"锁消除"

synchronized(lock) {

}

}

public synchronized void test(){

//这里不需要锁，因为方法上已经有了synchronized，意味着进入此处一定获取了this锁，所以jvm在编译期会进行"锁粗化"

synchronized(this) {

}

}

**常用的锁**

synchronized是隐式锁，不响应中断，并且会引起死锁，这是它的缺点，显式锁就能解决这个问题。

#### 1 理解CAS

synchronized的原理是遇到锁就陷入阻塞态，这会引起线程上下文切换，效率很低，而CAS是操作系统直接支持的原子操作，线程会通过自旋来循环检测条件是否满足，避免了线程上下文环境的切换。

public final boolean compareAndSet(int expect, int update);

CAS的实现原理很简单，如果当前值是expect，则更新为update，如果更新成功则返回true，否则返回false。整个步骤是原子操作。我们看下AtomicInteget的实现:

public final int incrementAndGet(){

for(;;){

int current = get();

int next = current + 1;

if(compareAndSet(current,next)) return next;

}

}

先获取当前的值current，计算期望值next，然后使用CAS更新，如果更新失败，则表示value被其他线程改了，重新尝试，直到更新成功才返回next。

#### 2 显式锁

可重入锁 ReentrantLock

lock()/unlock(); 一般的锁方法，lock()会阻塞直到成功获取锁  
lockInterruptibly(); 可以响应中断，如果被中断(隔壁synchronized懵逼了)，则抛出InterruptedException  
tryLock(); 尝试获取锁，立刻返回，不阻塞，如果获取成功，则返回true，否则返回false，这个API可以有效避免死锁(synchronized再次懵逼)  
newCondition(); 创建一个条件，一个Lock可以关联多个条件。

值得一提的是，ReentrantLock的构造函数可以接收一个boolean值，表示是否公平，传true则会为公平锁。

public class Test {

private final Lock lock = new ReentrantLock();

private volatile int number;

public void add(){

//加锁

lock.lock();

//自增

try {

count++;

}finally {

//解锁，解锁操作建议放在finally，从而保证一定能执行

lock.unlock();

}

}

public int getNumber(){

return number;

}

}

简易用法如上，解锁操作建议放在finally里面，保证一定能解锁

因为系统对synchronized的优化，使得synchronized的效率大大提升，所以一般来说，如果synchronized能满足条件，直接使用synchronized就行，不满足再来考虑ReentrantLock。

读写锁 ReadWriteLock

readLock(); 获取读锁，读锁是共享锁，可以多个线程共享，也就是说，可以多个线程同时读。  
writeLock(); 获取写锁，写锁是独占锁，也就是说，同一时刻只有一个线程可以写。

#### 3 显式条件

显式锁适用于替换synchronized的，那么显式条件就是用来替换wait()/notify()的。

显式条件Condition的api如下:

public interface Condition {

void await() throws InterruptedException; //等价于Object的await()

void awaitUninterruptibly(); //不可中断的await()

//....其他await()变体

void signal(); //等价于Object.notify()

void signalAll(); //等价于Object.notifyAll()

}

await在进入等待队列后，会释放锁，释放cpu，当其他线程唤醒它 或 等待超时 或 发生中断，都需要重新获取锁，从await()方法中退出。

public class Test {

private final Lock lock = new ReentrantLock();

private Condition condition = lock.newCondition();

private volatile boolean stop = false;

private void test(){

try {

//加锁

lock.lock();

try {

//检测终止条件

while(!stop) {

//等待

condition.await();

}

}finally {

//释放锁

lock.unlock();

}

}catch(InterruptedException e) {

//处理中断

Thread.interrupted();

}

}

//停止

private void stop(){

//加锁

lock.lock();

try {

//更新条件

stop = true;

//唤醒

condition.signal();

}finally {

//解锁

lock.unlock();

}

}

}

如上述代码，condition要和lock配合使用，await()和signal()需要在lock()和unlock()之间，正如wait()/notify()和synchronized一样。