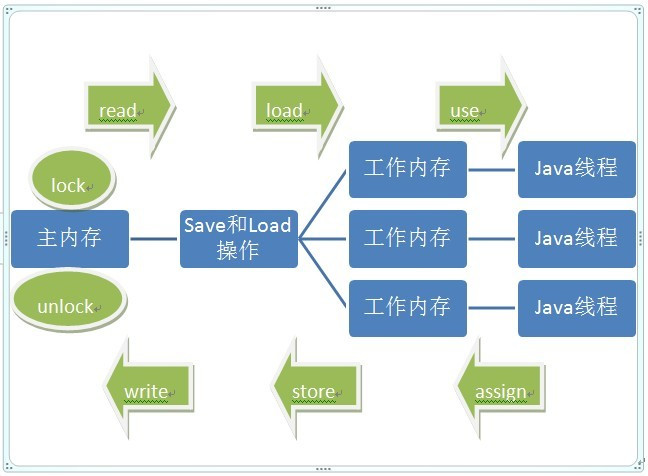
**临界区**：   
  
    临界区指的是一个访问共用资源（例如：共用设备或是共用存储器）的程序片段，而这些共用资源又无法同时被多个线程访问的特性。当有线程进入临界区段时，其他线程或是进程必须等待（例如：bounded waiting 等待法），有一些同步的机制必须在临界区段的进入点与离开点实现，以确保这些共用资源是被互斥获得使用，例如：semaphore。只能被单一线程访问的设备，例如：打印机。   
  
**互斥量**：   
  
    互斥量是一个可以处于两态之一的变量：解锁和加锁。这样，只需要一个二进制位表示它，不过实际上，常常使用一个整型量，0表示解锁，而其他所有的值则表示加锁。互斥量使用两个过程。当一个线程（或进程）需要访问临界区时，它调用mutex\_lock。如果该互斥量当前是解锁的（即临界区可用），此调用成功，调用线程可以自由进入该临界区。   
另一方面，如果该互斥量已经加锁，调用线程被阻塞，直到在临界区中的线程完成并调用mutex\_unlock。如果多个线程被阻塞在该互斥量上，将随机选择一个线程并允许它获得锁。   
  
**管程**：   
  
   管程 (英语：Monitors，也称为监视器) 是一种程序结构，结构内的多个子程序（对象或模块）形成的多个工作线程互斥访问共享资源。这些共享资源一般是硬件设备或一群变数。   
    管程实现了在一个时间点，最多只有一个线程在执行管程的某个子程序。与那些通过修改数据结构实现互斥访问的并发程序设计相比，管程实现很大程度上简化了程序设计   
    系统中的各种硬件资源和软件资源，均可用数据结构抽象地描述其资源特性，即用少量信息和对资源所执行的操作来表征该资源，而忽略了它们的内部结构和实现细节。   
    利用共享数据结构抽象地表示系统中的共享资源，而把对该共享数据结构实施的操作定义为一组过程。   
  
**信号量**：   
  
   信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量VI，然后将Acquire Semaphore VI以及Release Semaphore VI分别放置在每个关键代码段的首末端。确认这些信号量VI引用的是初始创建的信号量。   
  
**CAS操作**（Compare-and-Swap）：   
  
    CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。更详细资料：<http://zl198751.iteye.com/blog/1848575>   
  
**重排序**：   
  
    编译器和处理器”为了提高性能，而在程序执行时会对程序进行的重排序。它的出现是为了提高程序的并发度，从而提高性能！但是对于多线程程序，重排序可能会导致程序执行的结果不是我们需要的结果！重排序分为“编译器”和“处理器”两个方面，而“处理器”重排序又包括“指令级重排序”和“内存的重排序”。   
  
**一、线程与内存交互操作**   
  
   
  
    所有的变量（实例字段，静态字段，构成数组对象的 元素，不包括局部变量和方法参数）都存储在主内存中，每个线程有自己的工作内存，线程的工作内存保存被线程使用到变量的主内存副本拷贝。线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存的变量。不同线程之间也不能直接访问对方工作内存中的变量，线程间变量值的传递通过主内存来完成。   
  
**Java内存模型定义了八种操作**： 

* lock（锁定）：作用于主内存的变量，它把一个变量标识为一个线程独占的状态；
* unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定；
* read（读取）：作用于主内存的变量，它把一个变量的值从主内存传送到线程中的工作内存，以便随后的load动作使用；
* load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中；
* use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎；
* assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存中的变量；
* store（存储）：作用于工作内存的变量，它把工作内存中的一个变量的值传送到主内存中，以便随后的write操作；
* write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值写入主内存的变量中。

**volatile关键字作用：**   
  
    1）保证了新值能立即存储到主内存，每次使用前立即从主内存中刷新。   
  
    2）禁止指令重排序优化。   
  
    注：volatile关键字不能保证在多线程环境下对共享数据的操作的正确性。可以使用在自己状态改变之后需要立即通知所有线程的情况下。   
  
**二、并发的三个特性**   
  
**原子性**   
  
    原子性是指不可再分的最小操作指令，即单条机器指令，原子性操作任意时刻只能有一个线程，因此是线程安全的。   
Java内存模型中通过read、load、assign、use、store和write这6个操作保证变量的原子性操作。   
  
    long和double这两个64位长度的数据类型java虚拟机并没有强制规定他们的read、load、store和write操作的原子性，即所谓的非原子性协定，但是目前的各种商业java虚拟机都把long和double数据类型的4中非原子性协定操作实现为原子性。所以java中基本数据类型的访问读写是原子性操作。   
  
    对于大范围的原子性保证需要通过lock和unlock操作以及synchronized同步块来保证。   
  
**可见性**   
  
    可见性是指当一个线程修改了共享变量的值，其他线程可以立即得知这个修改。   
Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值来实现可见性的。   
  
Java中通过volatile、final和synchronized这三个关键字保证可见性： 

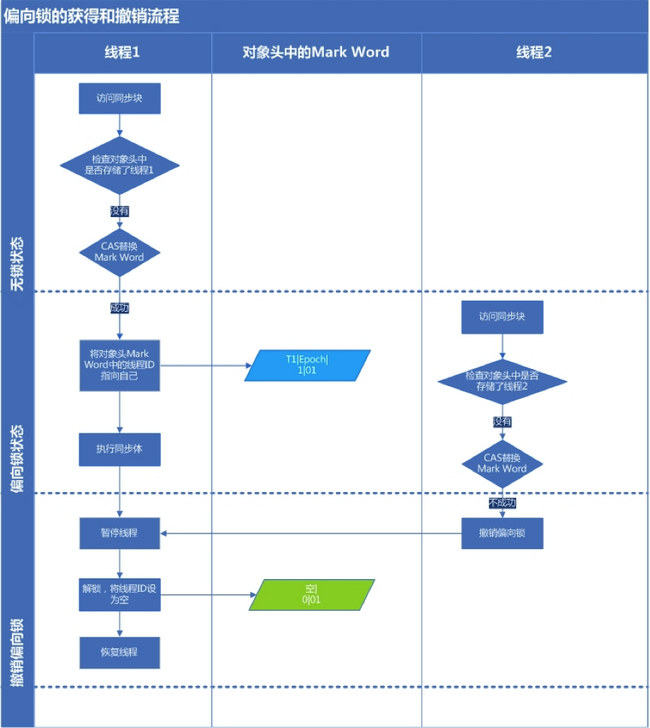
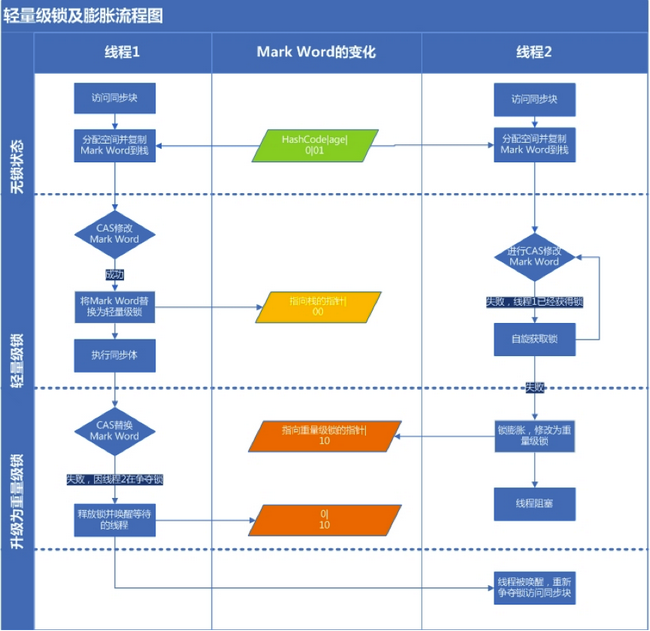
* volatile：通过刷新变量值确保可见性。
* synchronized：同步块通过变量lock锁定前必须清空工作内存中变量值，重新从主内存中读取变量值，unlock解锁前必须把变量值同步回主内存来确保可见性。
* final：被final修饰的字段在构造器中一旦被初始化完成，并且构造器没有把this引用传递进去，那么在其他线程中就能看见final字段的值，无需同步就可以被其他线程正确访问。

**有序性**   
  
    线程的有序性是指：在线程内部，所有的操作都是有序执行的，而在线程之间，因为工作内存和主内存同步的延迟，操作是乱序执行的。   
  
    Java通过volatile和synchronized关键字确保线程之间操作的有序性。 

* volatile禁止指令重排序优化实现有序性。
* synchronized通过一个变量在同一时刻只允许一个线程对其进行lock锁定操作来确保有序性。

**三、java线程的实现方式**   
  
**线程实现的三种方式**：   
  
**内核线程**（Kernal thread）   
      
    内核线程（Kernel Thread， KLT）就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process，LWP），轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核线程之间1：1的关系称为一对一的线程模型。轻量级进程要消耗一定的内核资源（如内核线程的栈空间），而且系统调用的代价相对较高，因此一个系统支持轻量级进程的数量是有限的。   
  
**轻量级用户进程**（Light weight process）   
  
    广义上来讲，一个线程只要不是内核线程，那就可以认为是用户线程（User Thread，UT），而狭义的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知到线程存在的实现，用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。这种进程与用户线程之间1：N的关系称为一对多的线程模型。（Windows和Linux使用的是这种方式）   
  
    使用用户线程的优势在于不需要系统内核的支援，劣势在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理，因而使用用户线程实现的程序一般都比较复杂，现在使用用户线程的程序越来越少了。   
  
**用户线程/混合线程**（User thread）   
  
    既存在用户线程，又存在轻量级进程。用户线程还是完全建立在用户空间中，而操作系统所支持的轻量级进程则作为用户线程和内核线程之间的桥梁。这种混合模式下，用户线程与轻量级进程的数量比是不定的，是M：N的关系。许多Unix系列的系统，都提供了M：N的线程模型实现。   
  
**java线程调度**   
  
    Java线程在JDK1.2之前，是基于名为“绿色线程”的用户线程实现的，而在JDK1.2中，线程模型被替换为基于操作系统原生线程模型来实现。因此，在目前的JDK版本中，操作系统支持怎样的线程模型，在很大程度上就决定了Java虚拟机的线程是怎样映射的，这点在不同的平台上没有办法达成一致，虚拟机规范中也未限定Java线程需要使用哪种线程模型来实现。   
  
**线程调度有两种方式**   
  
**协同式**：线程的执行时间由线程本身来控制，线程任务执行完成之后主动通知系统切换到另一个线程去执行。（不推荐）   
    优点：实现简单，线程切换操作对线程本身是可知的，不存在线程同步问题。   
    缺点：线程执行时间不可控制，如果线程长时间执行不让出CPU执行时间可能导致系统崩溃。   
  
**抢占式**：每个线程的执行时间有操作系统来分配，操作系统给每个线程分配执行的时间片，抢到时间片的线程执行，时间片用完之后重新抢占执行时间，线程的切换不由线程本身来决定（Java使用的线程调度方式就是抢占式调度）。   
    优点：线程执行时间可控制，不会因为一个线程阻塞问题导致系统崩溃。   
  
**四、Java中线程状态的调度关系**   
  
   
  
**五、java中的线程安全等级**   
  
**不可变**：   
  
    可以是基本类型的final；可以是final对象，但对象的行为不会对其状态产生任何影响，比如String的subString就是new一个String对象各种Number类型如BigInteger和BigDecimal等大数据类型都是不可变的，但是同为Number子类型的AtomicInteger和AtomicLong则并非不可变。原因与它里面状态对象是unsafe对象有关，所做的操作都是CAS操作，可以保证原子性。   
  
**绝对线程安全**：   
  
    不管运行时环境如何，调用者都不需要任何额外的同步措施。   
  
**相对线程安全**：   
  
    这是我们通常意义上的线程安全。需要保证对象单独的操作是线程安全的。比如Vector，HashTable，synchronizedCollection包装集合等。   
  
**线程兼容**：   
  
    对象本身不是线程安全的，但可以通过同步手段实现。一般我们说的不是线程安全的，绝大多数是指这个。比如ArrayList，HashMap等。   
  
**线程对立**：   
  
    不管调用端是否采用了同步的措施，都无法在并发中使用的代码。   
  
**六、线程安全的实现方式**   
  
**互斥同步**   
  
    在多线程访问的时候，保证同一时间只有一条线程使用。   
    临界区(Critical Section)，互斥量(Mutex)，信号量(Semaphore)都是同步的一种手段   
     java里最基本的互斥同步手段是synchronized，编译之后会形成monitorenter和monitorexit这两个字节码指令，这两个字节码都需要一个reference类型的参数来指明要锁定和解锁的对象，还有一个锁的计数器，来记录加锁的次数，加锁几次就要同样解锁几次才能恢复到无锁状态。   
  
     其实在“Java与线程”里已经提到，java的线程是映射到操作系统的原生线程之上的，不管阻塞还是唤醒都需要操作系统的帮忙完成，都需要从用户态转换到核心态，这是很耗费时间的，是java语言中的一个重量级(Heavyweight)操作，虽然虚拟机本身会做一点优化的操作，比如通知操作系统阻塞之前会加一段自旋等待的过程，避免频繁切换到核心态。   
  
ReentrantLock相比于synchronized的优势： 

* **等待可中断**：在持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待.
* **公平锁**：按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁.    new RenentrantLock(boolean fair)
* **锁绑定多个条件**：通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

**非阻塞同步**   
  
    互斥和同步最主要的问题就是阻塞和唤醒所带来的性能问题，所以这通常叫阻塞同步(悲观的并发策略)。随着硬件指令集的发展，我们有另外的选择：基于冲突检测的乐观并发策略，通俗讲就是先操作，如果没有其他线程争用共享的数据，操作就成功，如果有，则进行其他的补偿(最常见就是不断的重试)，这种乐观的并发策略许多实现都不需要把线程挂起，这种同步操作被称为非阻塞同步。   
  
这类的指令有：   
    1)测试并设置(test-and-set)   
    2)获取并增加   
    3)交换   
    4)比较并交换(CAS)   
    5)加载链接/条件储存(Load-Linked/Store-Conditional  LL/SC)   
  
    后面两条是现代处理器新增的处理器指令，在JDK1.5之后，java中才可以使用CAS操作，就是传说中的sun.misc.Unsafe类里面的compareAndSwapInt()和compareAndSwapLong()等几个方法的包装提供，虚拟机对这些方法做了特殊的处理，及时编译出来的结果就是一条平台相关的处理器CAS指令，没有方法调用的过程，可以认为是无条件的内联进去。   
  
    原来需要对i++进行同步，但现在有了这种CAS操作来保证原子性，比如用AtomicInteger。 但是CAS存在一个ABA的问题。可以通过AtomicStampedReference来解决（鸡肋）。   
  
**无同步**   
  
    有一些代码天生就是线程安全的，不需要同步。其中有如下两类：   
  
    **可重入代码**（Reentrant Code）：纯代码，具有不依赖存储在堆上的数据和公用的系统资源，用到的状态量都由参数中传入，不调用非可重入的方法等特征，它的返回结果是可以预测的。   
  
    **线程本地存储**（Thread Local Storage）：把共享数据的可见范围限制在同一个线程之内，这样就无须同步也能保证线程之间不出现数据争用问题。可以通过java.lang.ThreadLocal类来实现线程本地存储的功能。   
  
**七、java中的锁机制**   
  
**悲观锁**   
  
    假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。悲观锁假定其他线程企图访问或者改变你正在访问、更改的对象的概率是很高的，因此在悲观锁的环境中，在你开始改变此对象之前就将该对象锁住，并且直到你提交了所作的更改之后才释放锁。   
  
**乐观锁**   
  
    假设不会发生并发冲突。轻易不加锁。   
  
**自旋锁与自适应自旋**   
  
    线程挂起和恢复的操作都需要转入内核态中完成，这些操作给系统的并发性能带来了很大的压力，在许多应用中，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得，可以让后请求锁的线程等待一会儿，但不放弃处理器的执行时间，让线程执行一个忙循环（自旋）。   
  
    自旋锁默认的自旋次数值是10次，可以使用参数-XX:PreBlockSpin更改。   
  
    自适应自旋意味着自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。   
  
**锁清除**：   
  
    虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持。   
  
**锁粗化**：   
  
    如果虚拟机探测到有一系列连续操作都对同一个对象反复加锁和解锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部。   
  
**锁升级**   
  
    Java SE1.6为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。   
  
   
  
**偏向锁**   
  
    Hotspot的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要花费CAS操作来加锁和解锁，而只需简单的测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁），如果没有设置，则使用CAS竞争锁，如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程。   
  
    偏向锁的撤销：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态，如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。下图中的线程1演示了偏向锁初始化的流程，线程2演示了偏向锁撤销的流程。   
  
   
  
关闭偏向锁：偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟-XX：BiasedLockingStartupDelay = 0。如果你确定自己应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁-XX:-UseBiasedLocking=false，那么默认会进入轻量级锁状态。   
  
**轻量级锁**：   
  
    轻量级锁加锁：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。   
  
    轻量级锁解锁：轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。下图是两个线程同时争夺锁，导致锁膨胀的流程图。   
  
   
  
    因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。   
  
**重量级锁**：   
  
    重量锁在JVM中又叫对象监视器（Monitor），它至少包含一个竞争锁的队列，和一个信号阻塞队列（wait队列），前者负责做互斥，后一个用于做线程同步。