何谓乐观锁和悲观锁

悲观锁

总社假想最坏的情况，每次拿数据都认为别人会修改，所以每次都先上锁，这样别人想拿这个数据就会阻塞到他拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁

总是假设最好情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是更新的时候会判断一下在此期间有没有别人去更新这个数据，使用版本号机制和CAS算法来实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，数据库中类似于write_condition机制，在JAVA中原子变量类使用了乐观锁的实现方式CAS实现的。

使用场景

乐观锁适用于写比较少的情况下（多读场景） 悲观锁使用于多写场景。

乐观锁常见两种实现方式

乐观锁一般会使用版本号机制或者CAS算法实现

版本号机制

一般在数据表中加一个数据版本号version字段，表示数据被修改的次数，当属被修改时，version+1,。当线程A要更显数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version为当前数据version值，相同时才更新，否则充实更新操作，知道成功为止。

举一个简单的例子： 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ； 而当前帐户余额字段（ balance ）为 $100 。

操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50 （ $100-$50 ）。 在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。 操作员 A 完成了修改工作，将数据版本号加一（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。 操作员 B 完成了操作，也将版本号加一（ version=2 ）试图向数据库提交数据（ balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须大于记录当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。 这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

CAS算法

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数：

> 需要读写的内存值V
> 进行比较的值A
> 拟写入的新值B

当且仅当V的值等于A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作，一般情况下是一个自旋操作，即不断的重试。

乐观锁的问题

ABA问题

如果一个变量V除此读取的时候是A值，最后准备赋值的时候还是A值，那我们不能说明其没有被其他线程修改过，可能经过有限次操作只会又变回A值，但CAS可能误以为他从来没有被修改过，这就是ABA问题

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环开销大

自旋CAS，也就是不成功就一直循环执行到成功，如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升。

只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）