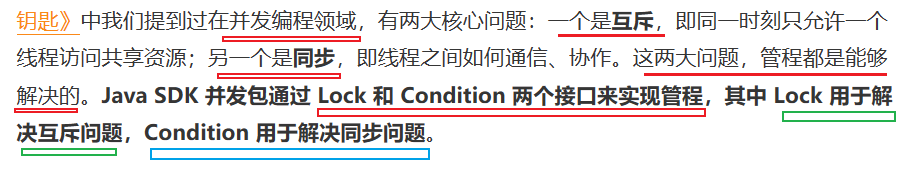
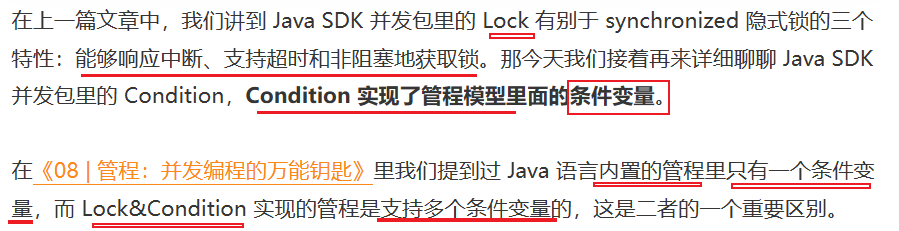
# Lock（上）

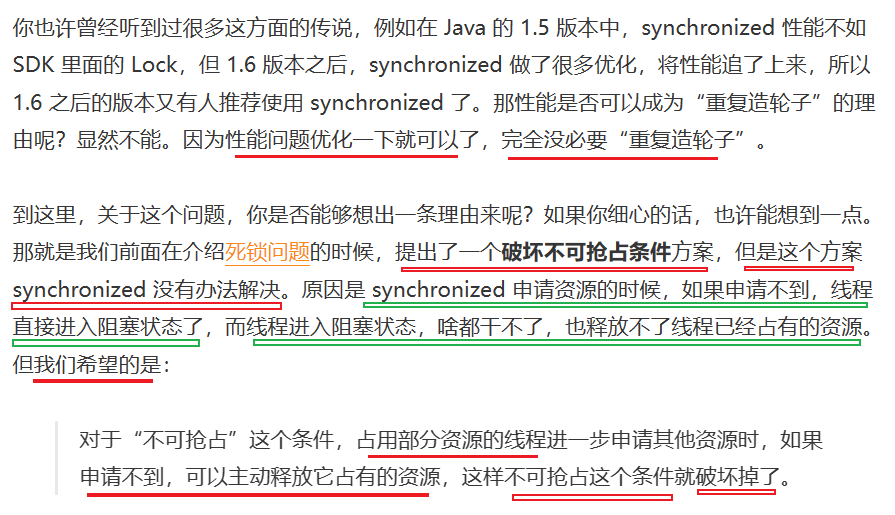
## Lock&Condition与synchronized的对比

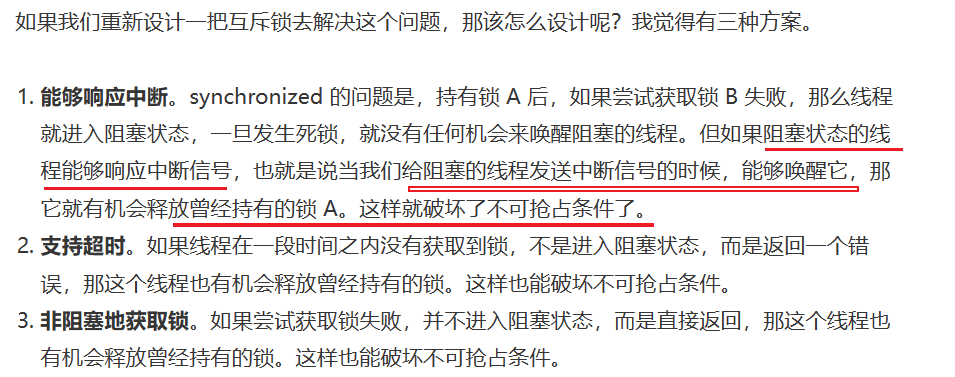
并发包其实就是管程的另一种实现方式





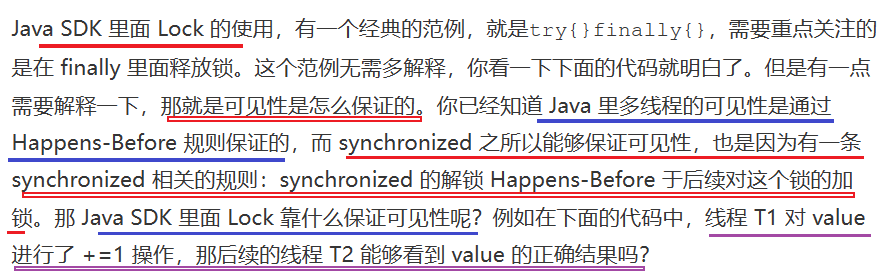
## 再造管程的理由

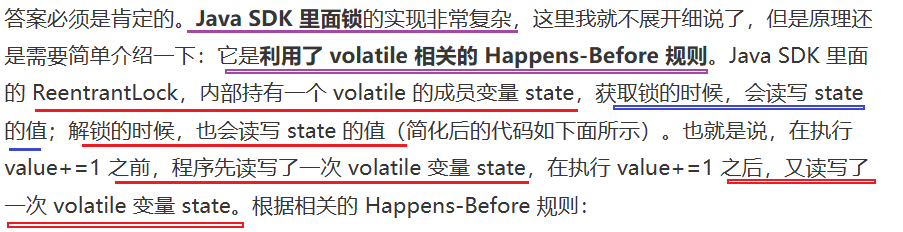


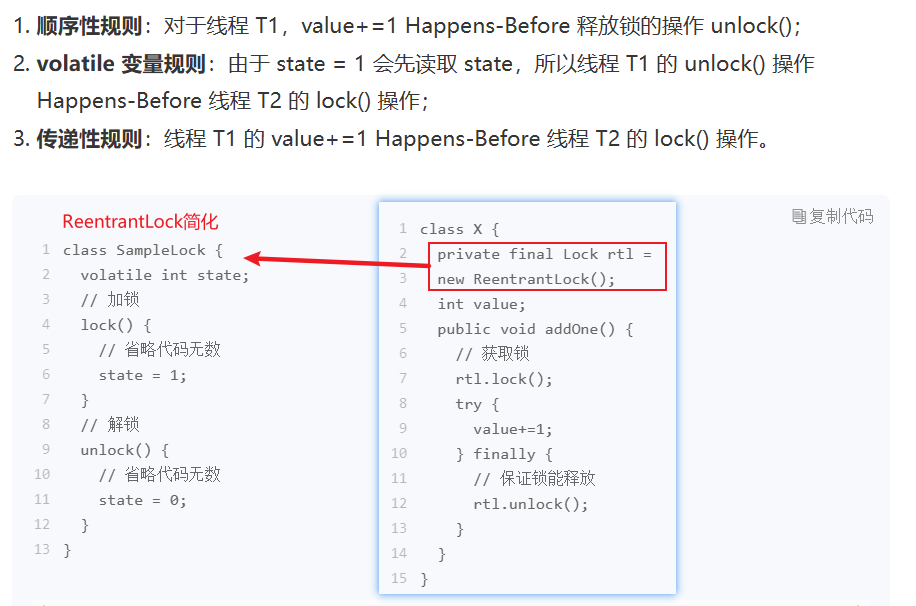




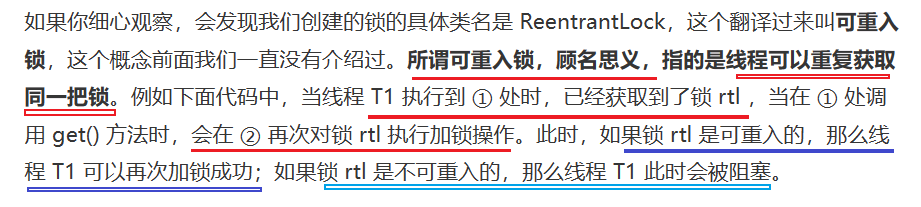
## 如何保证可见性

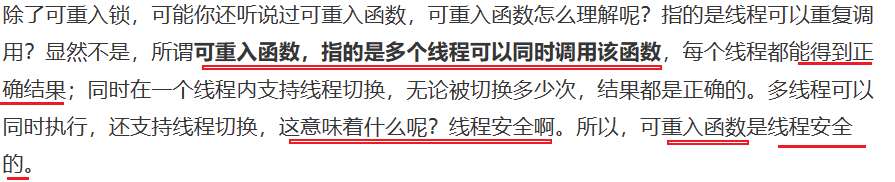
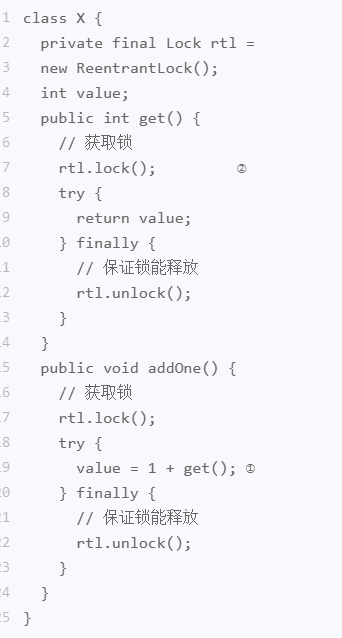




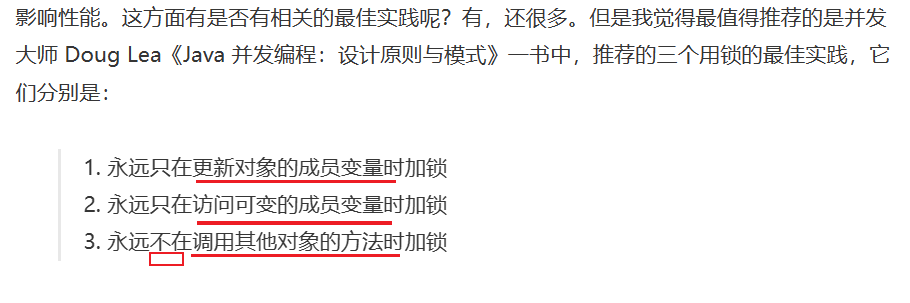


## 什么是可重入锁

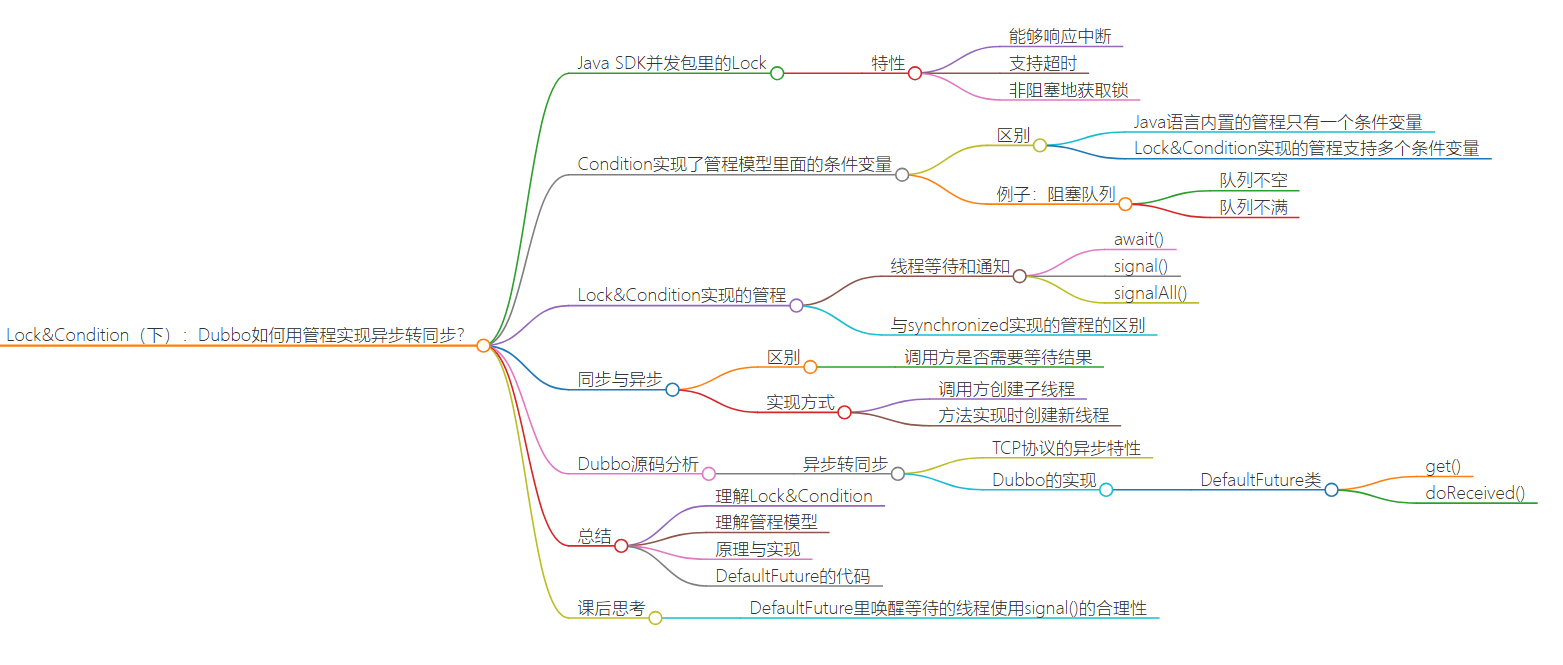




## 实践准则

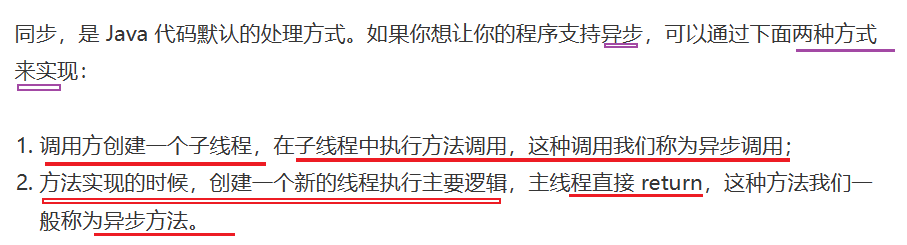


# Condition（下）

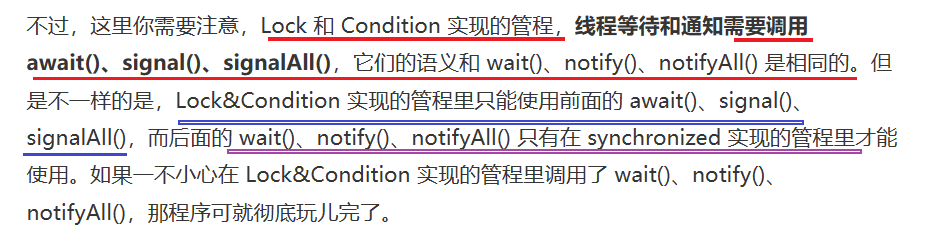


## 同步与异步





## Lock&Condition总结

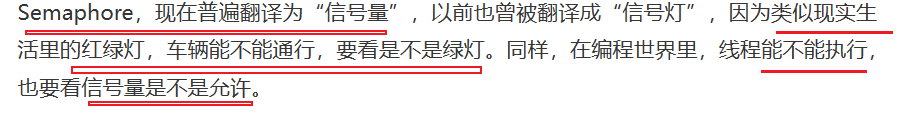
 **所谓可重入锁，顾名思义，指的是线程可以重复获取同一把锁**。

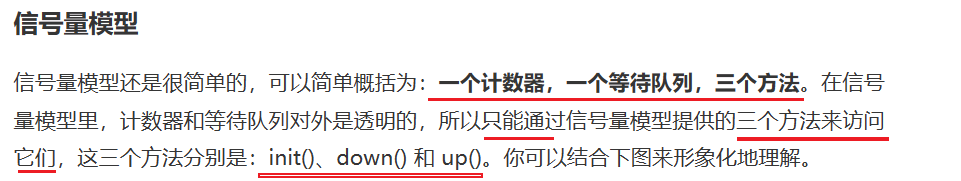
Lock&Condition 是管程的一种实现，所以能否用好 Lock 和 Condition 要看你对管程模型理解得怎么样。

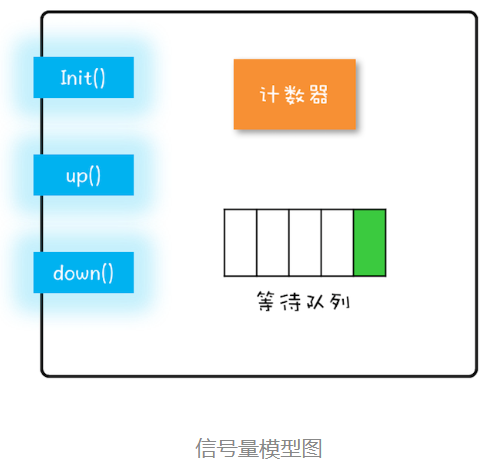
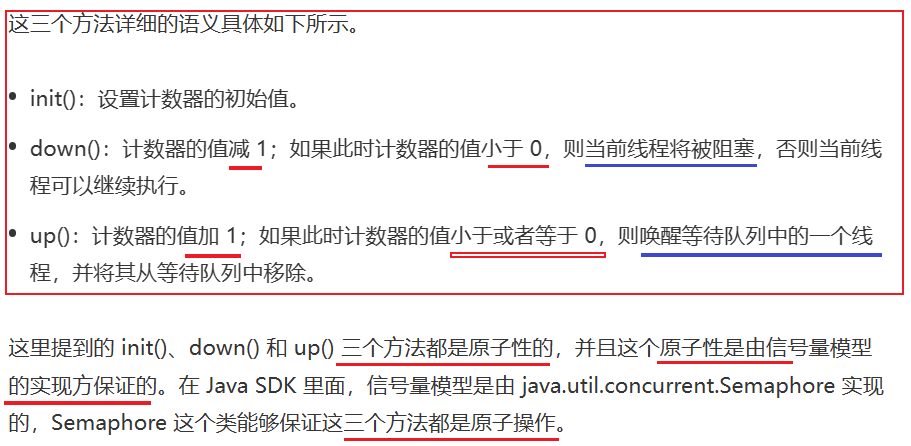
[《Java 并发编程的艺术》](time://mall?url=https%3A%2F%2Fh5.youzan.com%2Fv2%2Fgoods%2F35z7jjvd4r4oo)一书的第 5 章《Java 中的锁》，里面详细介绍了实现原理，我觉得写得非常好。

Lock&Condition 实现的管程相对于 synchronized 实现的管程来说更加灵活、功能也更丰富。

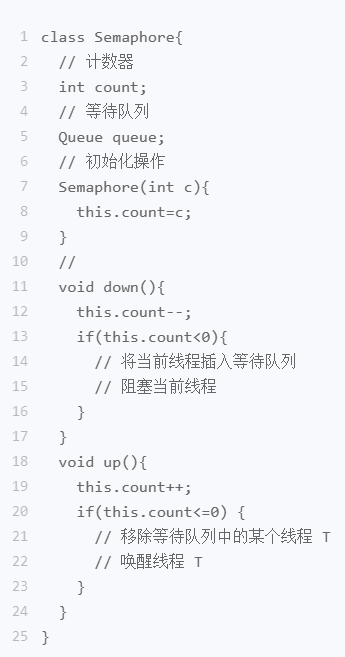
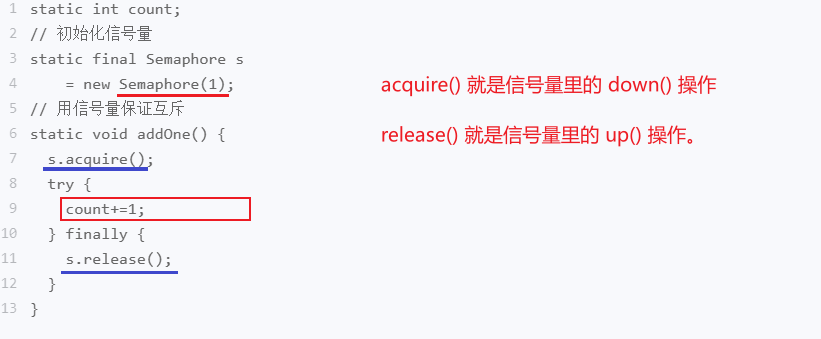
# 16 | Semaphore：如何快速实现一个限流器？





## 如何使用信号量



下面我们再来分析一下，信号量是如何保证互斥的。

假设两个线程 T1 和 T2 同时访问 addOne() 方法，当它们同时调用 acquire() 的时候，由于 acquire() 是一个原子操作，所以只能有一个线程（假设 T1）把信号量里的计数器减为 0，另外一个线程（T2）则是将计数器减为 -1。

对于线程 T1， 信号量里面的计数器的值是 0，大于等于 0，所以线程 T1 会继续执行； 对于线程 T2，信号量里面的计数器的值是 -1，小于 0，按照信号量模型里对 down()

操作的描述，线程 T2 将被阻塞。

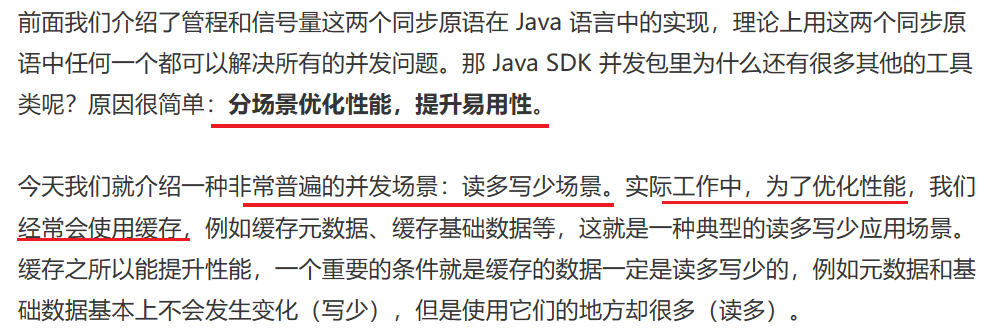
所以此时只有线程 T1 会进入临界区执行count+=1；。

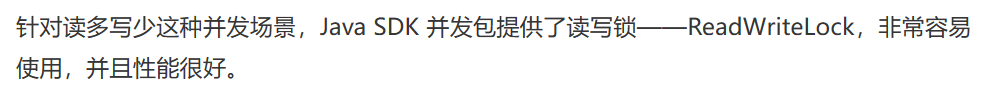
当线程 T1 执行 release() 操作，也就是 up() 操作的时候，信号量里计数器的值是 -1，加 1 之后的值是 0，小于等于 0，按照信号量模型里对 up() 操作的描述，此时等待队列中的 T2 将会被唤醒。

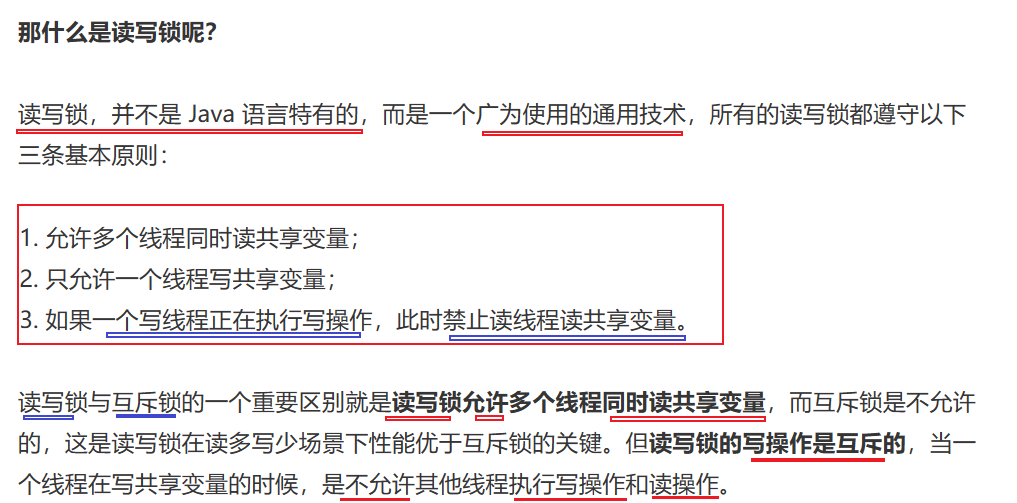
于是 T2 在 T1 执行完临界区代码之后才获得了进入临界区执行的机会，从而保证了互斥性。

# 17 | ReadWriteLock：如何快速实现一个完备的缓存？

## 读写锁原理



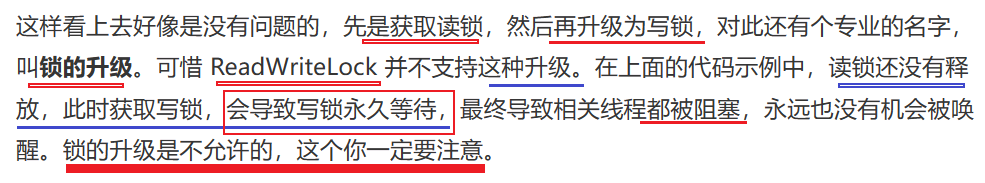




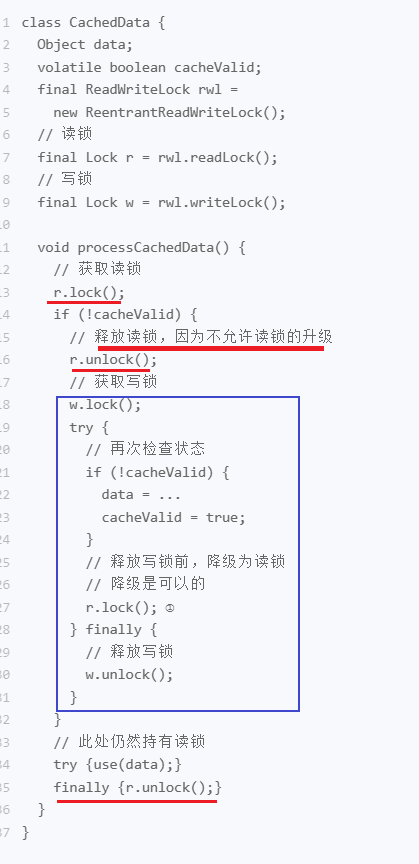
也就是说 **读锁与写锁是互斥的**

## 锁的升级

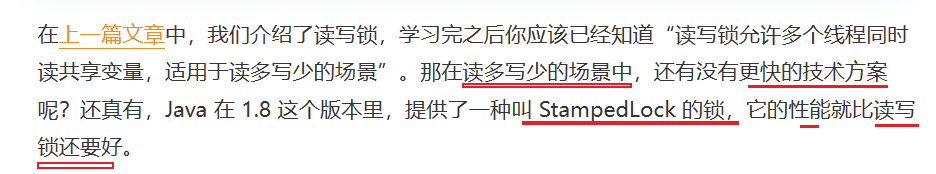




## 锁降级



# 18 | StampedLock：有没有比读写锁更快的锁？



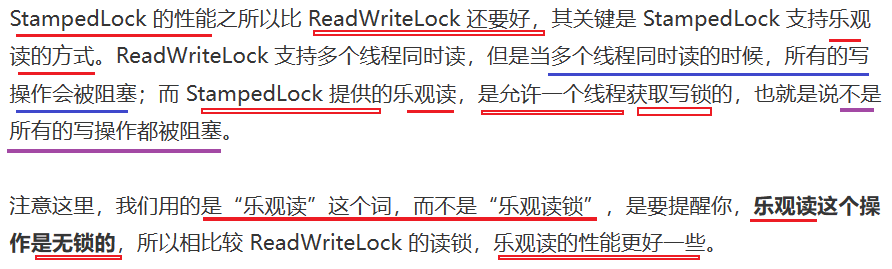
## 与读写锁的区别

使用上 StampedLock 和ReadWriteLock 有哪些区别。

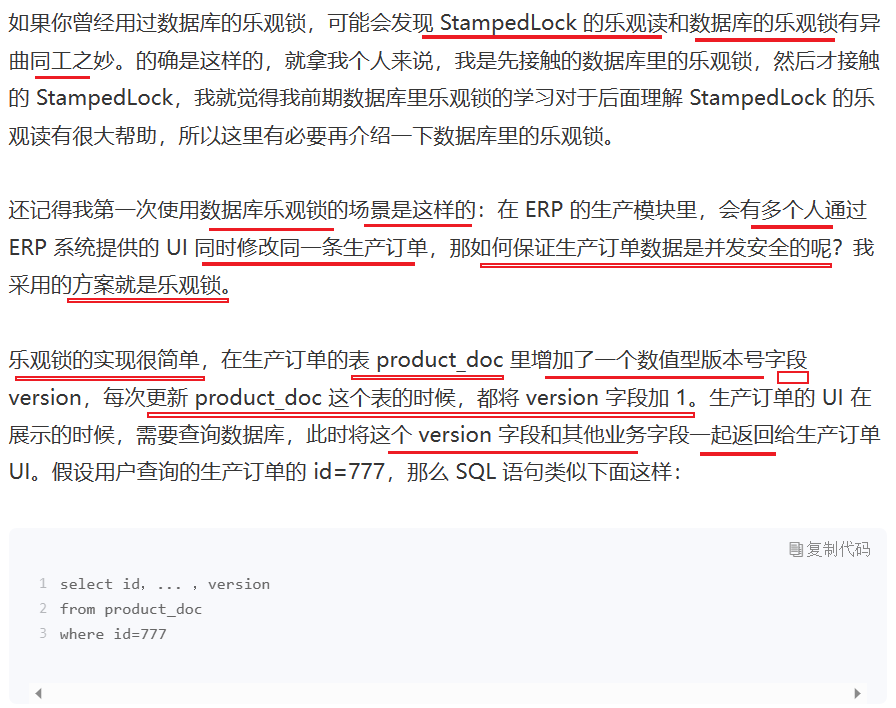
ReadWriteLock 支持两种模式：一种是读锁，一种是写锁。而 StampedLock 支持三种模式，分别是：**写锁、悲观读锁和乐观读**。

**相同点时**：写锁、悲观读锁的语义和 ReadWriteLock 的写锁、读锁的语义非常类似，允许多个线程同时获取悲观读锁，但是只允许一个线程获取写锁，写锁和悲观读锁是互斥的。

**不同的是**：StampedLock 里的写锁和悲观读锁加锁成功之后，都会返回一个 stamp；然后解锁的时候，需要传入这个 stamp。相关的示例代码如下。



## 如何理解乐观读



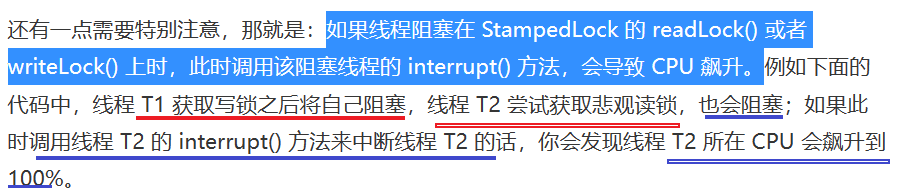


## 注意

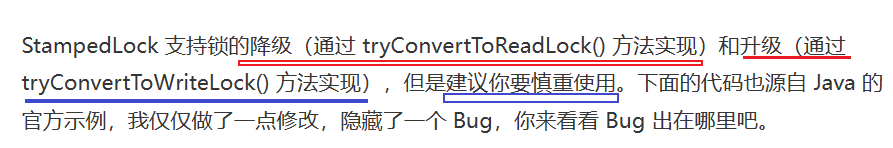
1.StampedLock 的功能仅仅是 ReadWriteLock 的子集

2.StampedLock 不支持重入

3.StampedLock 的悲观读锁、写锁都不支持条件变量。







**读锁与写锁是互斥的**

# 19 | CountDownLatch和CyclicBarrier：如何让多线程步调一致？

## 特点

**1.CountDownLatch 主要用来解决一个线程等待多个线程的场景**，可以类比旅游团团长要等待所有的游客到齐才能去下一个景点；

**2.CyclicBarrier 是一组线程之间互相等待**，更像是几个驴友之间不离不弃。

## 区别

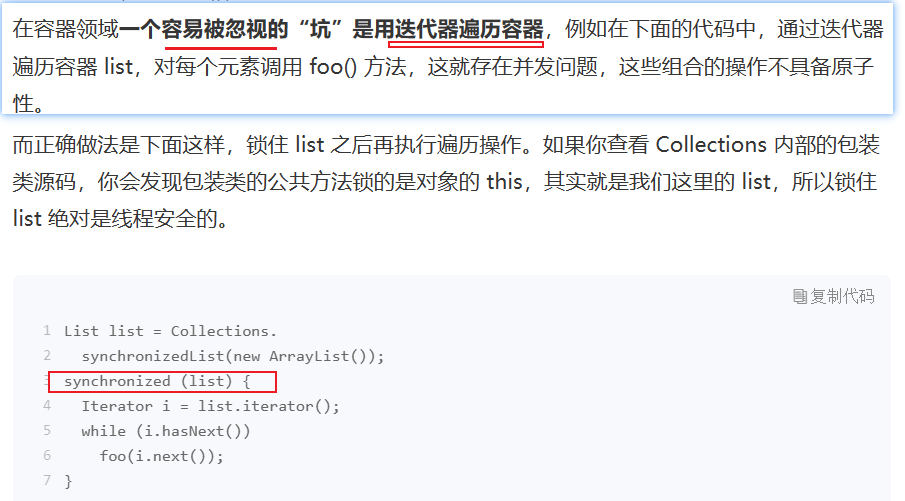
1. CountDownLatch的计数器是**不能循环利用**，也就是说一旦计数器减到 0，再有线程调用 await()，该线程会直接通过。

2. **CyclicBarrier 的计数器是可以循环利用的**，而且具备自动重置的功能，一旦计数器减到 0 会自动重置到你设置的初始值。

3. CyclicBarrier 还可以设置回调函数，可以说是功能丰富。

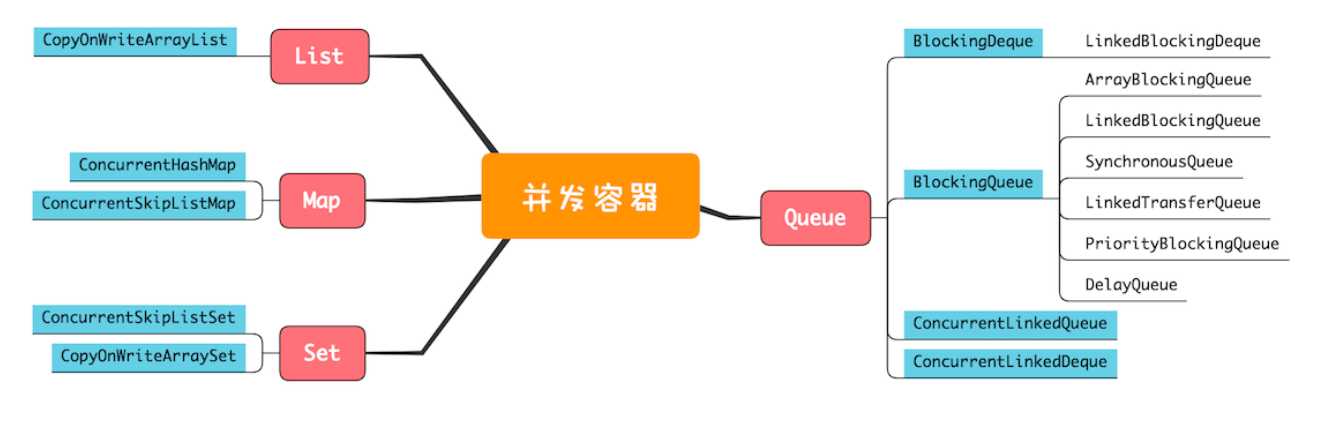
# 20 | 并发容器：都有哪些“坑”需要我们填？

## 注意事项



注意，遍历是线程安全的，单foo有可能对结合操作，导致非线程安全问题，所以要上锁，用于控制对list修改。

## 并发容器





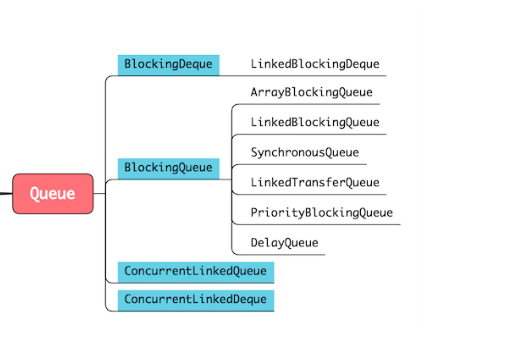
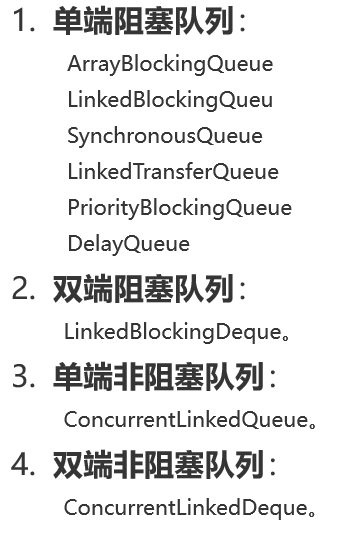
## 队列

Java 并发包里面 Queue 这类并发容器是最复杂的，可以从以下两个维度来分类。

1. 一个维度是**阻塞与非阻塞**，所谓阻塞指的是当队列已满时，入队操作阻塞；当队列已空时，出队操作阻塞。
2. 另一个维度是**单端与双端**，单端指的是只能队尾入队，队首出队；而双端指的是队首队尾皆可入队出队。

Java 并发包里

1. **阻塞队列都用 Blocking 关键字标识，**
2. **单端队列使用 Queue 标识，**
3. **双端队列使用 Deque 标识**。

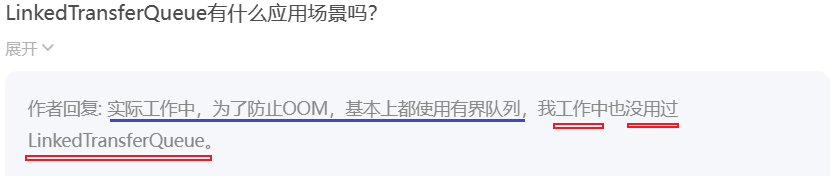
 

### 单端阻塞队列

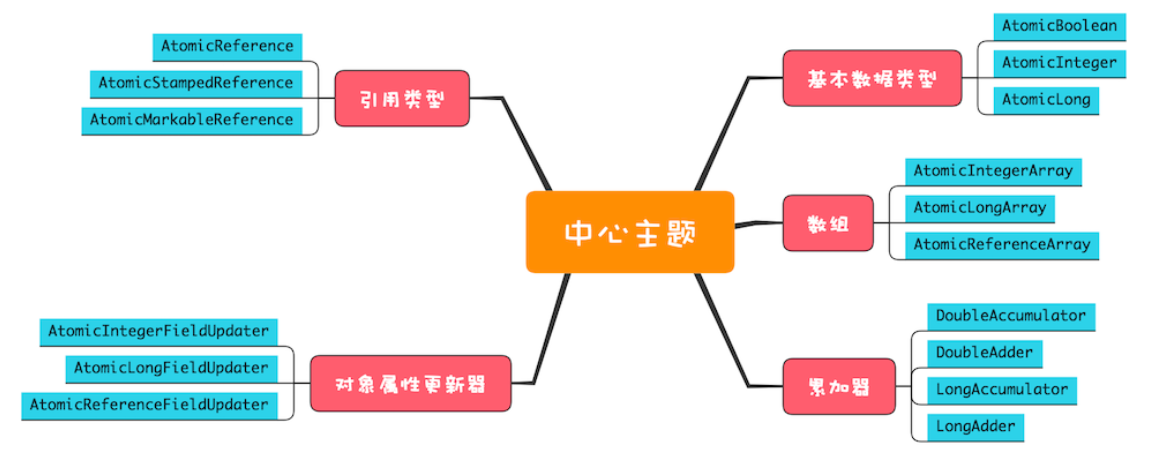
1. 内部一般会持有一个队列，这个队列可以是数组（其实现是 ArrayBlockingQueue）也可以是链表（其实现是 LinkedBlockingQueue）；
2. 甚至还可以不持有队列（其实现是 SynchronousQueue），此时生产者线程的入队操作必须等待消费者线程的出队操作。
3. 而 LinkedTransferQueue 融合 LinkedBlockingQueue 和 SynchronousQueue 的功能，性能比 LinkedBlockingQueue 更好；
4. PriorityBlockingQueue 支持按照优先级出队；

5.DelayQueue 支持延时出队。

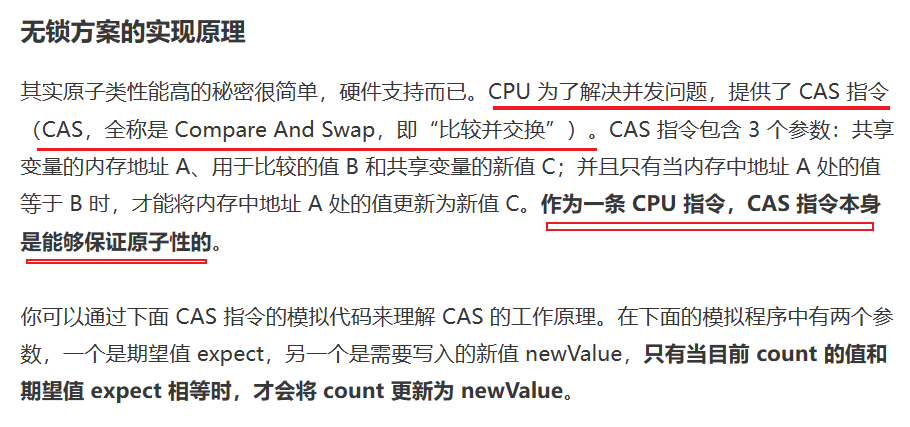
 Queue 中，只有 ArrayBlockingQueue 和 LinkedBlockingQueue 是支持有界的，所以**在使用其他无界队列时，一定要充分考虑是否存在导致 OOM 的隐患**。



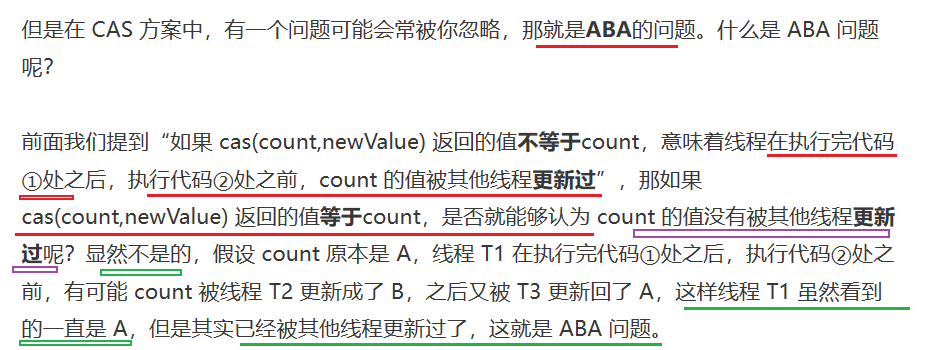
# 21 | 原子类：无锁工具类的典范



## CAS



## ABA问题



## ABA解决方案

