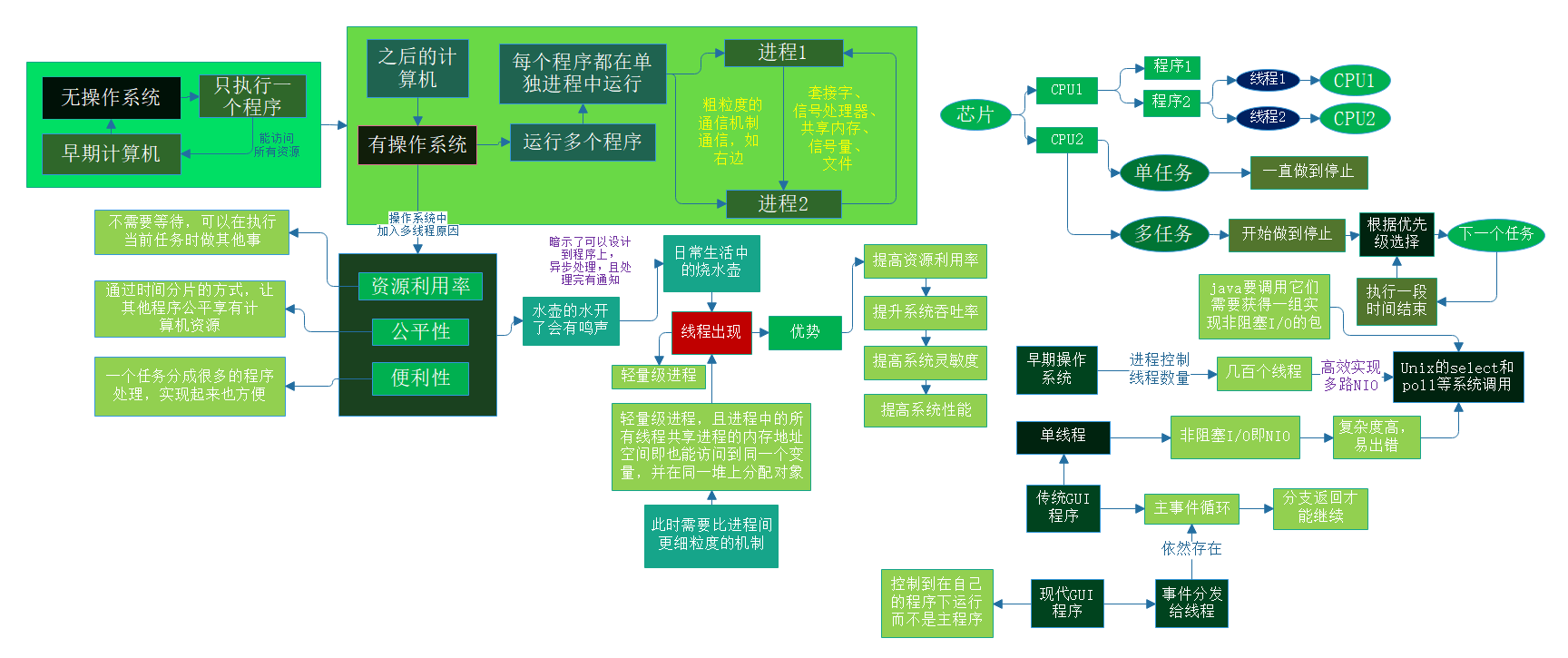
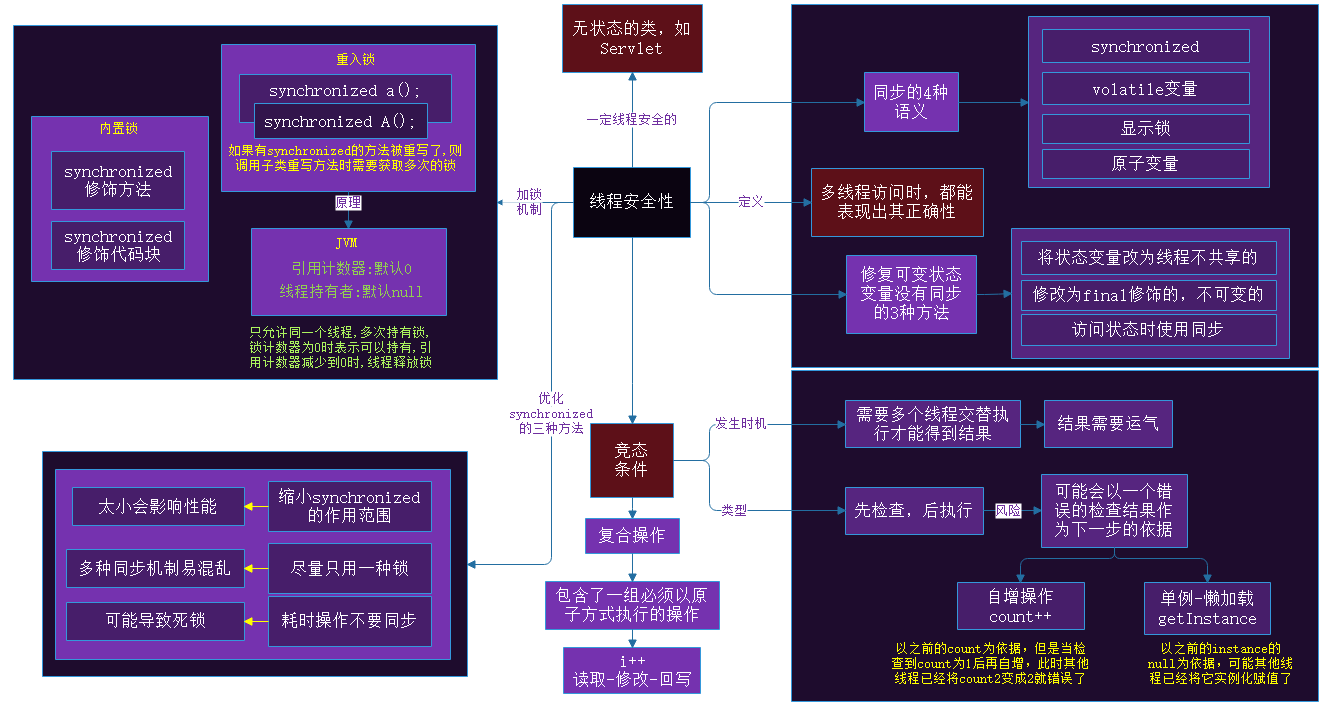
# 简介



# 线程安全性

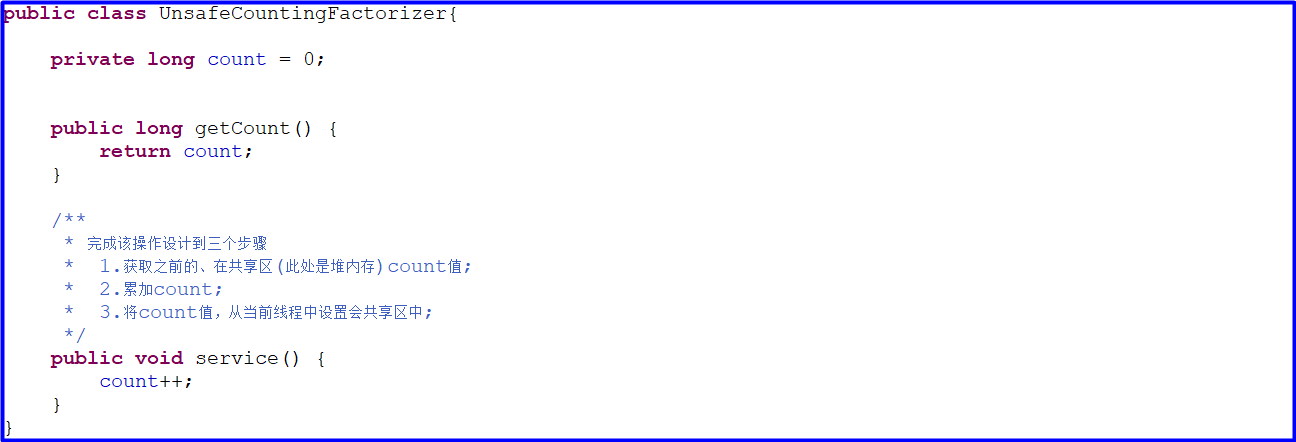


## 两种常见竞态条件问题详解

### count++

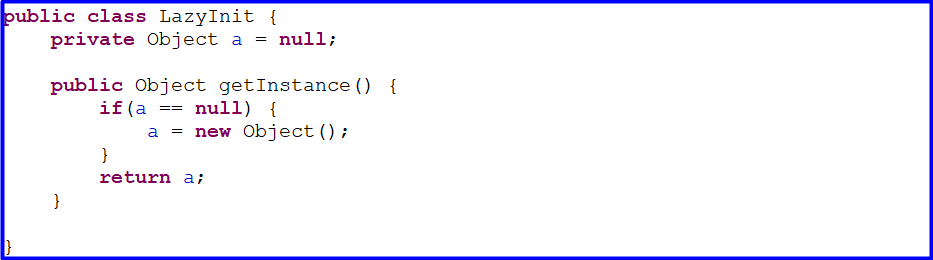
如下图代码所示，由于类中存在可变的状态变量count，它是属于对象的，也就是和当前对象一起再堆内存中分配的，而堆内存是属于线程共享的区域。

而对于count++操作，其实要拆分成三个步骤，获取共享内存中的变量值、对改变量进行自增操作、将自增好的变量值设置回共享区；但是由于这三个操作并不具备原子性，所以可能导致多个线程同时进行count++操作，每个线程刚开始都获取到count值为0，然后第一个线程设置值为1，本来第二个要为2，但是第二个线程也是从0-1的转变，然后后续线程都将count设置为1，没有达到累加的效果了。



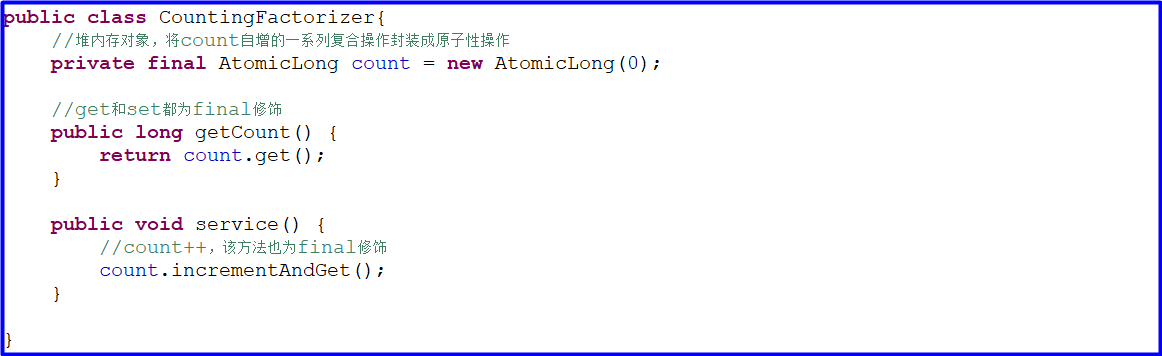
### 延迟初始化

同理，延迟初始化也是如此，当某个操作还没有完全完成时，另外一个线程过来了，导致创建出两个内存地址不一样的线程，这个尤其在框架中的单例创建，可能会导致两个对象含有一个标识。



综上所述，竞态条件的类型一般都是先检查后执行，但是往往都是以失效的检查结果来决定下一个步骤，比如count++，发现为0，然后再加1，但没有发现其他线程可能已经将0变成1，还继续用错误的0自增；延迟初始化也是，利用a=null，然后new Object()但是没有发现其他线程已经让a!=null了。

## AtomicLong解决count++问题



### 源码分析

内置了long类型的变量，且是volatile类型的，方法都为final修饰，保证了线程的安全性。

