可变状态变量在并发中解决方案:

1. 不在线程之间共享该状态变量
2. 将状态变量修改为不可变
3. 在访问状态变量时使用同步

JAVA的内置锁是互斥锁，最多只有一个线程持有该锁

当执行时间较长的计算或是可能无法完成的操作如IO，一定不要加锁。

非volatile的64位变量(double/long)的读写操作分为两个32位操作，所以也不是原子操作，即非线程安全

加锁的含义不仅仅局限于互斥行为，还包括内存可见性，为了确保所有线程都能看到共享变量的最新值，所以执行读操作或者写操作的线程都必须在同一个锁上同步。

volatile是削弱版的同步机制，被其修饰的变量会在所有线程可见，并总是读到最新值。

但是volatile无法保证类似count++操作的原子性，所以syncronise加锁机制即可以保证原子性和可见性，但volatile只能保证可见性，要保证count++操作原子性，可以使用AtomicInteger count = new AtomicInteger(0)

线程封闭，即使用单线程的形式访问数据。常见的JDBC Connection和ThreadLocal就是线程封闭的。

在孔浩的项目里，Contoller如果需要向Service传递大量参数时，会造成性能上的影响。所以使用了ThreadLocal来存储这些通用参数，从这个线程中直接读取参数就会效率更高。

ThreadLocal为每个线程提供了一份变量副本，而不会和其他线程的副本冲突。JDBC connection可以被多线程共享，所以在不加锁的情况下是非线程安全的，如A线程事务没结束，B线程获得了该connection然后commit了，数据就会发生混乱。通过ThreadLocal<connection>每个线程都获得自己的connection副本，所以之间不会冲突，也就线程安全了。

final能确保初始化过程的安全性，从而可以不受限制地访问不可变对象，并在共享这些对象时无须同步（可见性）。

不可变对象的所有需求必须满足

1. 状态不可修改
2. 所有域都是final类型
3. 正确的构造器初始化final值

要安全的发布一个对象，对象的引用以及对象的状态必须同时对其他线程可见，一个正确构造的对象可以通过以下方式来安全地发布：

1. 在静态初始化函数中初始化一个对象

Public static Holder holder = new Holder(42);

静态初始化器由JVM在类的初始化阶段执行，由于在jvm内部存在着同步机制，因此这种方式可以安全发布对象。

1. 使用final + volatile 的搭配，保证初始化过程安全性和对象的可见性
2. 保存到由锁保护的域中如使用 vector/synchronizedMap/synchronizedList 等

-----架构师视屏-----

synchronized修饰的方法是对象锁

Static synchronised修饰的方式对类级别的锁

锁重入：得到对象锁的方法可以不等待再次得到该对象锁

//建立对象锁

Synchronized(this){

}

//建立类级别锁

Sychronnized(Object.Class){

}

wait方法释放锁, notify方法不释放锁。他俩必须配合synchronized(object){}使用

Object object = new Object

Thread1-- run(){

synchronized(object){

//等待释放锁

object.wait();

}

Thread2-- run(){

synchronized(object){

//唤醒其他进程,不释放锁,被唤醒的进程只有等待其释放锁才能执行

object.notify();

}

}

在多线程访问容器(List/Map)的情况下，为了避免同步加锁机制带了的串行化执行(牺牲效率),jdk新出了 CopyOnWriteList(先拷贝一份复制list,进行写操作，有并发读的时候访问原有容器，然后将指针指向这个新创建的list), ConcurrentHashMap(则是将map最多分16段，每段进行加锁，不同段可以异步操作)

CountDownLatch - 主线程await(), 其他线程countdown减1，当为0时唤醒主线程继续执行

CyclicBarrier- 同步所有线程同时执行

使用ReentrantLock重入锁的lock()和unlock()方法可以替代synchronized(obj)的wait()和notify从而更加灵活