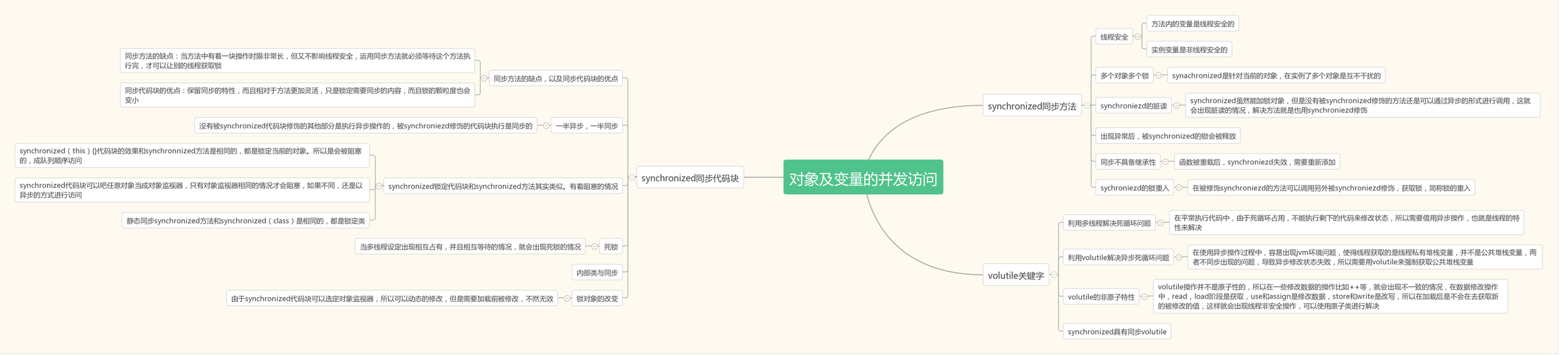
对象及变量的并发访问
synchronized同步代码块
同步方法的缺点，以及同步代码块的优点
同步方法的缺点：当方法中有着一块操作时限非常长，但又不影响线程安全，运用同步方法就必须等待这个方法执行完，才可以让别的线程获取锁
同步代码块的优点：保留同步的特性，而且相对于方法更加灵活，只是锁定需要同步的内容，而且锁的颗粒度也会变小
一半异步，一半同步
没有被synchronized代码块修饰的其他部分是执行异步操作的，被synchroniezd修饰的代码块执行是同步的
synchronized锁定代码块和synchronized方法其实类似。有着阻塞的情况
synchronized（this）{}代码块的效果和synchronnized方法是相同的，都是锁定当前的对象。所以是会被阻塞的，成队列顺序访问
synchronized代码块可以吧任意对象当成对象监视器，只有对象监视器相同的情况才会阻塞，如果不同，还是以异步的方式进行访问
静态同步synchronized方法和synchronized（class）是相同的，都是锁定类
死锁
当多线程设定出现相互占有，并且相互等待的情况，就会出现死锁的情况
内部类与同步
锁对象的改变
由于synchronized代码块可以选定对象监视器，所以可以动态的修改，但是需要加载前被修改，不然无效
synchronized同步方法
线程安全
方法内的变量是线程安全的
实例变量是非线程安全的
多个对象多个锁
synachronized是针对当前的对象，在实例了多个对象是互不干扰的
synchroniezd的脏读
synchronized虽然能加锁对象，但是没有被synchronized修饰的方法还是可以通过异步的形式进行调用，这就会出现脏读的情况，解决方法就是也用synchroniezd修饰
出现异常后，被synchronized的锁会被释放
同步不具备继承性
函数被重载后，synchroniezd失效，需要重新添加
sychroniezd的锁重入
在被修饰synchroniezd的方法可以调用另外被synchroniezd修饰，获取锁，简称锁的重入
volutile关键字
利用多线程解决死循环问题
在平常执行代码中，由于死循环占用，不能执行剩下的代码来修改状态，所以需要借用异步操作，也就是线程的特性来解决
利用volutile解决异步死循环问题
在使用异步操作过程中，容易出现jvm环境问题，使得线程获取的是线程私有堆栈变量，并不是公共堆栈变量，两者不同步出现的问题，导致异步修改状态失效，所以需要用volutile来强制获取公共堆栈变量
volutile的非原子特性
volutile操作并不是原子性的，所以在一些修改数据的操作比如++等，就会出现不一致的情况，在数据修改操作中，read，load阶段是获取，use和assign是修改数据，store和write是改写，所以在加载后是不会在去获取新的被修改的值，这样就会出现线程非安全操作，可以使用原子类进行解决
synchronized具有同步volutile