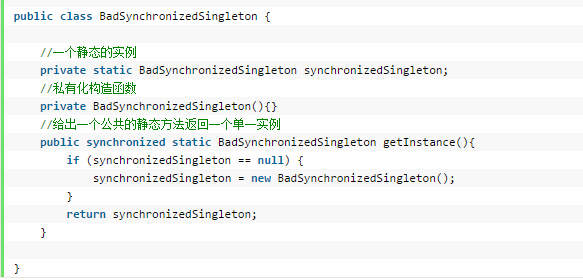
# 代理模式

# 单例模式



上面的做法很简单，就是将整个获取实例的方法同步，这样在一个线程访问这个方法时，其它所有的线程都要处于挂起等待状态，倒是避免了刚才同步访问创造出多个实例的危险，但是我只想说，这样的设计实在是糟糕透了，这样会造成很多无谓的等待，所以为了表示我的愤怒，我在类名上加入Bad。其实我们同步的地方只是需要发生在单例的实例还未创建的时候，在实例创建以后，获取实例的方法就没必要再进行同步控制了，所以我们将上面的示例改为很多教科书中标准的单例模式版本，也称为**双重加锁**。

双重检锁：



这种做法与上面那种最无脑的同步做法相比就要好很多了，因为我们只是在当前实例为null，也就是实例还未创建时才进行同步，否则就直接返回，这样就节省了很多无谓的线程等待时间，值得注意的是在同步块中，我们再次判断了synchronizedSingleton是否为null，解释下为什么要这样做。

     假设我们去掉同步块中的是否为null的判断，有这样一种情况，假设A线程和B线程都在同步块外面判断了synchronizedSingleton为null，结果A线程首先获得了线程锁，进入了同步块，然后A线程会创造一个实例，此时synchronizedSingleton已经被赋予了实例，A线程退出同步块，直接返回了第一个创造的实例，此时B线程获得线程锁，也进入同步块，此时A线程其实已经创造好了实例，B线程正常情况应该直接返回的，但是因为同步块里没有判断是否为null，直接就是一条创建实例的语句，所以B线程也会创造一个实例返回，此时就造成创造了多个实例的情况。

     经过刚才的分析，貌似上述双重加锁的示例看起来是没有问题了，但如果再进一步深入考虑的话，其实仍然是有问题的。

     如果我们深入到JVM中去探索上面这段代码，它就有可能（注意，只是有可能）是有问题的。

              因为虚拟机在执行创建实例的这一步操作的时候，其实是分了好几步去进行的，也就是说创建一个新的对象并非是原子性操作。在有些JVM中上述做法是没有问题的，但是有些情况下是会造成莫名的错误。

              首先要明白在JVM创建新的对象时，主要要经过三步。

**1.分配内存**

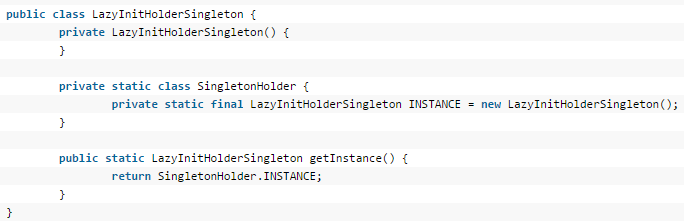
**2.初始化构造器**

**3.将对象指向分配的内存的地址**

              这种顺序在上述双重加锁的方式是没有问题的，因为这种情况下JVM是完成了整个对象的构造才将内存的地址交给了对象。但是如果2和3步骤是相反的（2和3可能是相反的是因为JVM会针对字节码进行调优，而其中的一项调优便是调整指令的执行顺序），就会出现问题了。

              因为这时将会先将内存地址赋给对象，针对上述的双重加锁，就是说先将分配好的内存地址指给synchronizedSingleton，然后再进行初始化构造器，这时候后面的线程去请求getInstance方法时，会认为synchronizedSingleton对象已经实例化了，直接返回一个引用。如果在初始化构造器之前，这个线程使用了synchronizedSingleton，就会产生莫名的错误。

             所以我们在语言级别无法完全避免错误的发生，我们只有将该任务交给JVM，所以有一种比较标准的单例模式。如下所示。



根据jvm规范，当某对象第一次调用LazyInitHolderSingleton.getInstance()时，LazyInitHolderSingleton类被首次主动使用，jvm对其进行初始化（此时并不会调用LazyInitHolderSingleton()构造方法），然后LazyInitHolderSingleton调用getInstance()方法，该方法中，又首次主动使用了SingletonHolder类，所以要对SingletonHolder类进行初始化，初始化中，INSTANCE常量被赋值时才调用了 LazyInitHolderSingleton的构造方法LazyInitHolderSingleton()，完成了实例化并返回该实例。

当再有对象（也许是在别的线程中）再次调用LazyInitHolderSingleton.getInstance()时，因为已经初始化过了，不会再进行初始化步骤，所以直接返回INSTANCE常量即同一个LazyInitHolderSingleton实例。