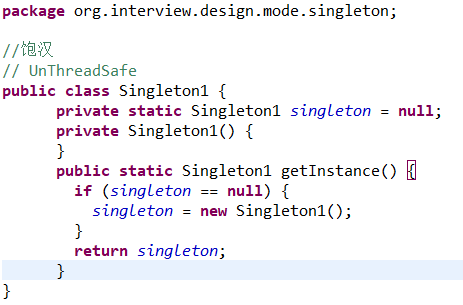
#### 懒汉式

由于懒汉式加载的时候没有加锁，所以在new的时候是线程不安全的，当多个线程同时访问生成单例方法的时候，返回的单例对象可能不相同



#### 懒汉式—线程安全

在生成单例方法上加了一把锁，使得线程是安全的，但是由于加锁是串行化操作，并发性能差



#### 懒汉式—双重加锁检查

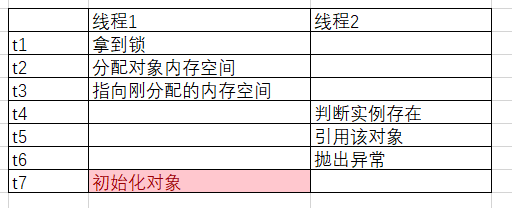
这里有两个问题：

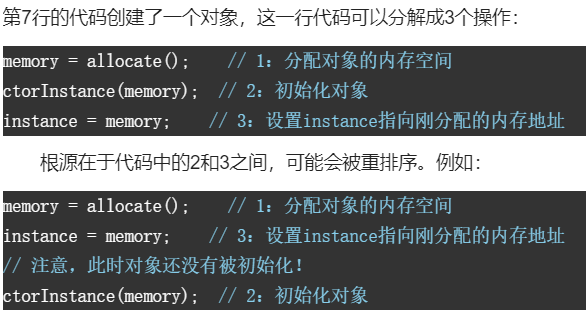
为什么要检查两次？

第一次检查是作为一个优化，仅当单例未生成的时候，才需要加锁检测，才使程序进入串行化，保证程序性能。第二次还需要检查是因为程序在第一次检查后到加锁的这个过程中，有可能已经生成了一个新的单例，为了避免又生成了一个新的实例，所以在加锁里面还需要再次检查

Volatile关键字是否必须？

必须，用于防止指令重排序。中间instance = new Singleton3();这段是可以分解的。没有加volatile的话会可能会出现指令重排序。 如下图所示，假如指令重排序原因，在未初始化前就指向了单例对象，同事线程2又恰好拿到了这个对象并进行调用，就会抛出异常

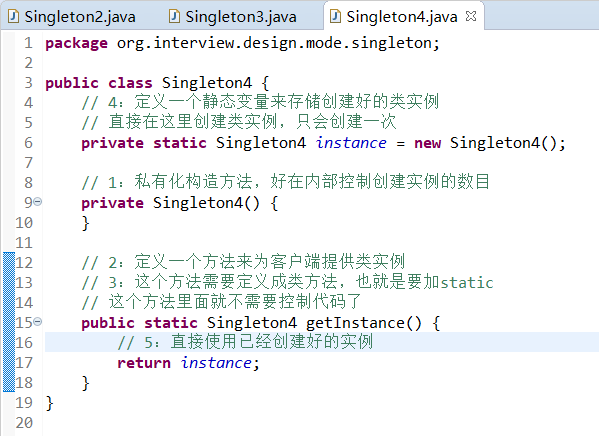






#### 饿汉式

饿汉式会在一开始就生成一个对象，是安全的，但是会导致资源浪费



#### 静态内部类懒加载

加载单例类时，静态内部类不会被同时加载。当访问其静态变量、方法时，静态内部类才被加载并初始化。

类初始化规则

虚拟机规范则严格规定了有且只有5种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要

在此之前开始）：

1、遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

2、使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3、当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

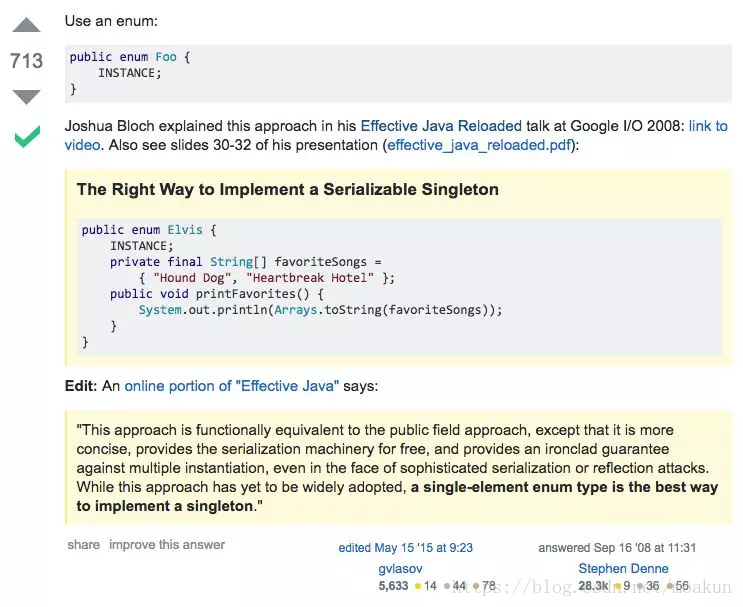
4、当虚拟机启动时，用户需要指定一个要执行的主类（包含main（）方法的那个类），虚拟机会先初始化这个主类。

5、当使用JDK 1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。



#### 枚举类单例

Stack Overflow上单例最好的实现方式，枚举类得票率第一



优点：

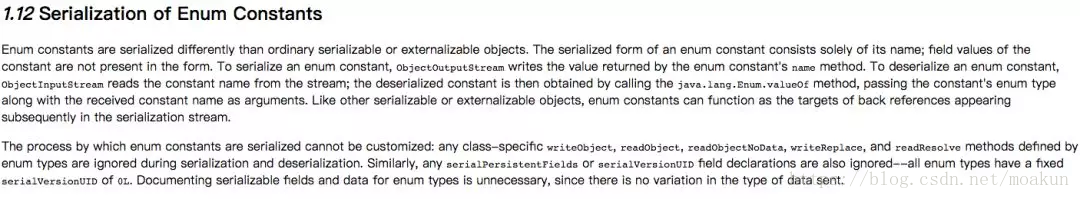
1、安全简单，越简单越不容易出错，而且是线程安全的

通过将定义好的枚举反编译，我们就能发现，其实枚举在经过javac的编译之后，会被转换成形如public final class T extends Enum的定义。而且，枚举中的各个枚举项同事通过static来定义的。如：



static类型的属性会在类被加载之后被初始化，当一个Java类第一次被真正使用到的时候静态资源被初始化、Java类的加载和初始化过程都是线程安全的（因为虚拟机在加载枚举的类的时候，会使用ClassLoader的loadClass方法，而这个方法使用同步代码块保证了线程安全）。所以，创建一个enum类型是线程安全的。

2、枚举类可以避免反序列化破坏单例



在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject等方法。普通的Java类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新new出来的，所以这就破坏了单例。但是，枚举的反序列化并不是通过反射实现的。所以，也就不会发生由于反序列化导致的单例破坏问题。

