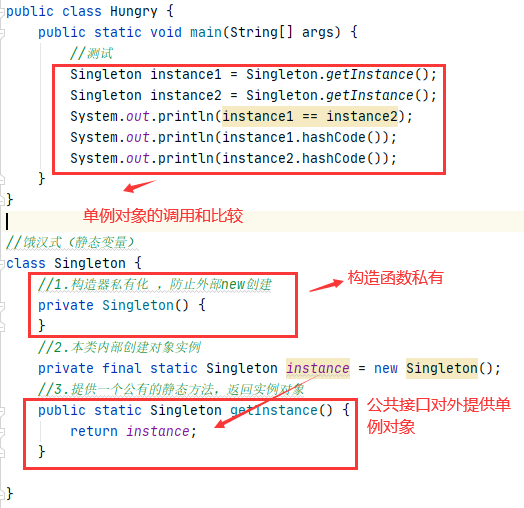
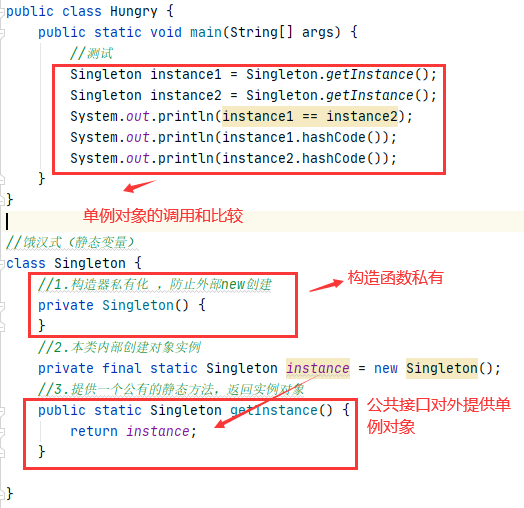
设计模式

1. 设计模式一般分为三大类：
2. 创建型模式：单例模式、工厂模式等
3. 结构型模式：适配器模式等
4. 行为型模式：观察者模式等
5. 单例模式：保证某个对象在多次调用中获取到的都是同一个实例，通常是构造函数私有的方式防止通过new方式创建，然后再提供公共接口提供该单例对象。
6. 单例模式仅仅是对于创建时的线程安全，但是不包括里面属性的安全，里面的属性依然是可以被线程共享的，是线程不安全的，需要增加其他操作
7. 使用场景：线程池、数据库连接池等的创建。



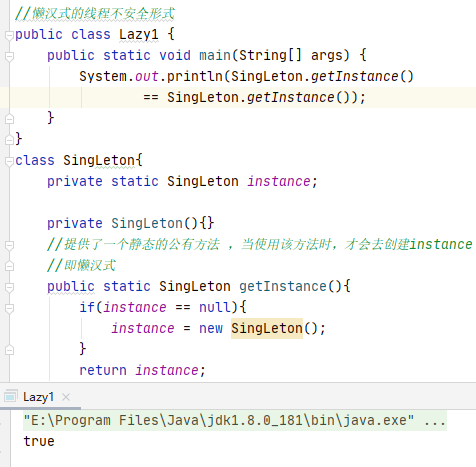
1. 单例模式有八种：
2. 饿汉式（静态常量/静态代码块、线程安全）
3. 懒汉式（线程不安全）
4. 懒汉式（线程安全、同步代码块）
5. 双重检查
6. 静态内部类
7. 枚举
8. 饿汉式
9. 使用原理：
10. 类的加载经过三个阶段：加载、连接、初始化，其中连接又分为验证、准备、解析。
11. 加载阶段：将class文件加载到内存中，并在方法区形成常量池
12. 连接->验证：验证加载的二进制文件是否符合规范
13. 连接->准备：为静态变量设置初始值并在方法区中（其实是在堆中）分配内存，如果是final修饰的，会直接进行赋值操作
14. 连接->解析：将常量池中的符号引用变为直接引用
15. 初始化：执行类构造器方法，类构造器会自动收集静态变量和静态块，为静态变量赋值，并且类构造器在并发操作下是同步执行的，只能有一个线程执行，其他线程是阻塞的，一个线程执行完毕后，其他线程将不再执行
16. 饿汉式就是利用了这个原理，利用静态块或者静态变量在类加载时只加载一次，来实现单例模式的创建。
17. 缺点是：容易造成内存浪费，因为即使不用，常量池中依然有这个静态实例
18. 静态常量方式
19. 特点是：final的存在，导致没有任何一个线程可以对其进行修改，线程安全；但是也是因为final的存在，使其在类加载的准备阶段就已经进行了实例化，即使不用，实例对象也会在内存中占用空间。



1. 静态代码块



1. 懒汉式
2. 特点是不浪费内存，只有使用的时候才会进行加载，因为类加载的时候，只会执行静态块和类变量。
3. 线程不安全形式，容易在getInstance处发生并发操作。



1. 线程安全形式，getInstance添加同步锁，拒绝多线程同时访问，但是有个缺点是每次需要这个实例的时候都要去实例化创建，而不是用创建好的直接return，造成效率低下。



1. 双重检查，推荐使用，面试重点。
2. 既解决了饿汉式的浪费缓存的问题，又解决了懒汉式效率低的问题。
3. 相关原理解释：volatile的作用：在new一个对象，JVM内会发生以下变化
4. 首先在堆中开辟内存，赋予初始值并传一个引用到栈中；
5. 对空间进行初始化，数据的赋值；
6. 将栈中的引用赋值给字面量，即Student S =new Student();中的S，当字面量被赋值以后，JVM就会认为这个不是空值。

按照这个顺序创建new值不会出错的，但是编译器、CPU等会对第二步和第三步进行重排序，当对字面量进行引用赋值而没有初始化空间时，JVM就会认为这是一个已经被创建好的变量，但是其实它是一个没有初始化的空值。

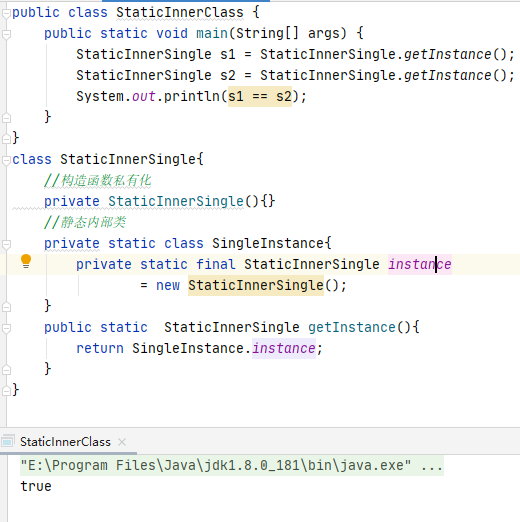
当下图中位置2执行到给字面量赋值而没有进行对空间初始化的时候，这个时候有一个线程到了位置1，它就会认为这不是空值了，直接返回了一个堆空间还没有被初始化的变量，造成了空指针异常，

Volatile就可以防止被重排序，是被修饰的变量按照标准的流程进行。

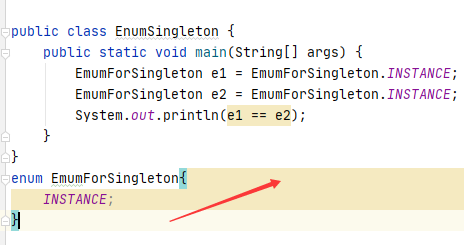




1. 静态内部类，（推荐使用，和双重检验锁同级）
2. 和双重检验一样，既解决了饿汉式的内存浪费问题又解决了懒汉式的加载效率问题。
3. 原理：类加载的时候，内部类是不会随着类被加载而被加载的，只有当使用的时候才会被加载。以静态内部类的方式做单例，内部用静态变量或静态块的方式对单例进行实例化，因为静态块或静态变量在初始化阶段是同步执行的，保证了类加载的线程安全问题。



1. 枚举 java作者推荐 可以防止反射的破坏，之前的单例会被反射破坏但是这个不会。



1. 桥接模式（详情见CSDN）
2. 适配器模式（详情见CSDN）