单例模式有如下实现方式：

[java] view plain copy

package com.zzj.pattern.singleton;

public class Singleton {

private static Singleton instance;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种方式称为延迟初始化，但是在多线程的情况下会失效，于是使用同步锁，给getInstance() 方法加锁：

[java] view plain copy

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

同步是需要开销的，我们只需要在初始化的时候同步，而正常的代码执行路径不需要同步，于是有了双重检查加锁（DCL）：

[java] view plain copy

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

这样一种设计可以保证只产生一个实例，并且只会在初始化的时候加同步锁，看似精妙绝伦，但却会引发另一个问题，这个问题由指令重排序引起。

指令重排序是为了优化指令，提高程序运行效率。指令重排序包括编译器重排序和运行时重排序。JVM规范规定，指令重排序可以在不影响单线程程序执行结果前提下进行。例如 instance = new Singleton() 可分解为如下伪代码：

[java] view plain copy

memory = allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance = memory; //3：设置instance指向刚分配的内存地址

但是经过重排序后如下：

[java] view plain copy

memory = allocate(); //1：分配对象的内存空间

instance = memory; //3：设置instance指向刚分配的内存地址

//注意，此时对象还没有被初始化！

ctorInstance(memory); //2：初始化对象

将第2步和第3步调换顺序，在单线程情况下不会影响程序执行的结果，但是在多线程情况下就不一样了。线程A执行了instance = memory（这对另一个线程B来说是可见的），此时线程B执行外层 if (instance == null)，发现instance不为空，随即返回，但是得到的却是未被完全初始化的实例，在使用的时候必定会有风险，这正是双重检查锁定的问题所在！

鉴于DCL的缺陷，便有了修订版：

[java] view plain copy

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

Singleton temp = instance;

if (temp == null) {

synchronized (Singleton.class) {

temp = new Singleton();

}

instance = temp;

}

}

}

return instance;

}

修订版试图引进局部变量和第二个synchronized来解决指令重排序的问题。但是，Java语言规范虽然规定了同步代码块内的代码必须在对象锁释放之前执行完毕，却没有规定同步代码块之外的代码不能在对象锁释放之前执行，也就是说 instance = temp 可能会在编译期或者运行期移到里层的synchronized内，于是又会引发跟DCL一样的问题。

在JDK1.5之后，可以使用volatile变量禁止指令重排序，让DCL生效：

[java] view plain copy

package com.zzj.pattern.singleton;

public class Singleton {

private static volatile Singleton instance;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

volatile的另一个语义是保证变量修改的可见性。

单例模式还有如下实现方式：

[java] view plain copy

package com.zzj.pattern.singleton;

public class Singleton {

private static class InstanceHolder {

public static Singleton instance = new Singleton();

}

private Singleton() {

}

public static Singleton getInstance() {

return InstanceHolder.instance;

}

}

这种方式称为延迟初始化占位（Holder）类模式。该模式引进了一个静态内部类（占位类），在内部类中提前初始化实例，既保证了Singleton实例的延迟初始化，又保证了同步。这是一种提前初始化（恶汉式）和延迟初始化（懒汉式）的综合模式。

# 至此，正确的单例模式有三种实现方式：

## 1.提前初始化。

[java] view plain copy

package com.zzj.pattern.singleton;

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton() {

}

public static Singleton getInstance() {

return instance;

}

}

## 2.双重检查锁定 + volatile。

## 3.延迟初始化占位类模式。