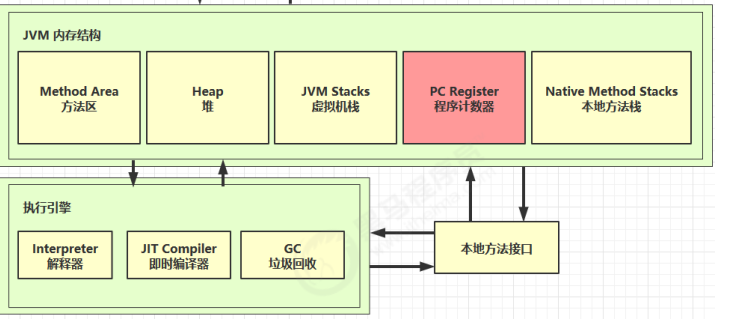
Java内存模型

JMM 即 Java Memory Model，它定义了主存、工作内存抽象概念，底层对应着 CPU 寄存器、缓存、硬件内存、CPU 指令优化等。

JMM 体现在以下几个方面：

* 原子性 - 保证指令不会受到**线程上下文切换**的影响
* 可见性 - 保证指令不会受 **cpu 缓存**的影响
* 有序性 - 保证指令不会受 **cpu 指令并行优化**的影响



可见性

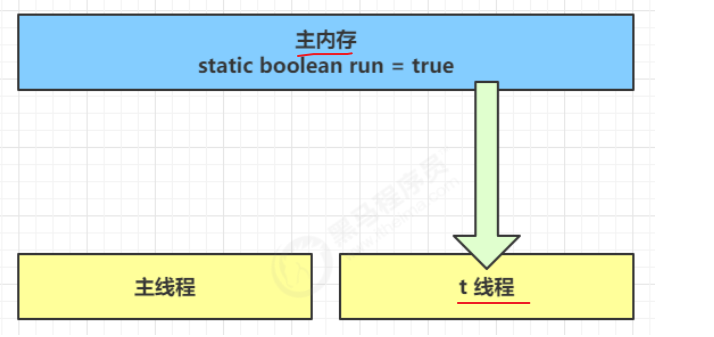
1、可见性问题

*//共享变量可见性问题***static boolean** *run* = **true**;  
  
@Test  
**public void** test1() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*run*) {  
 *//...* }  
 }, **"t1"**);  
 t1.start();  
  
 **try** {  
 Thread.*sleep*(1000);  
 *run* = **false**; *//t1线程不会停止* t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

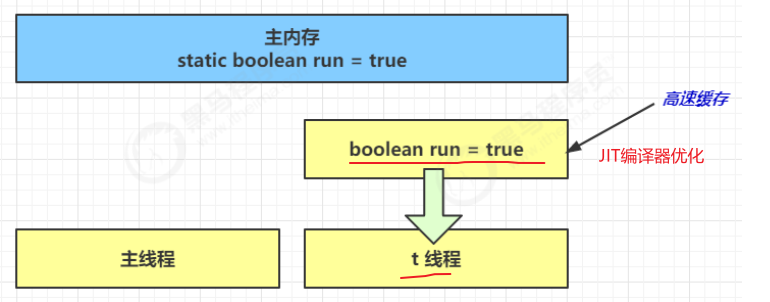
在主线程中修改run=false后，子线程t1不会停止。

2、原因分析

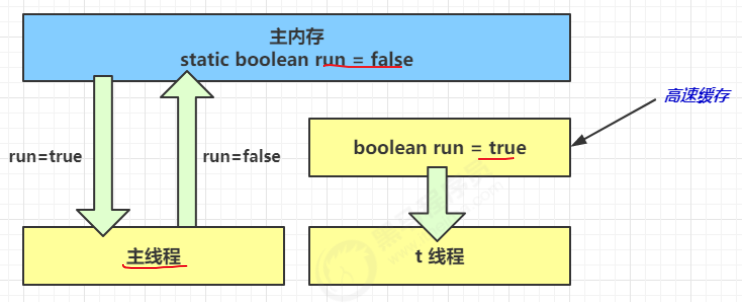
【1】初始状态， t 线程刚开始从主内存读取了 run 的值到工作内存。



【2】因为 t 线程要频繁从主内存中读取 run 的值，**JIT 编译器**会将 run 的值缓存至自己工作内存中的高速缓存中，减少对主存中 run 的访问，提高效率。



【3】1 秒之后，main 线程修改了 run 的值，并同步至主存，而 t 是从自己工作内存中的高速缓存中读取这个变量的值，结果永远是旧值。



3、解决方法

volatile（易变关键字）

它可以用来修饰成员变量和静态成员变量，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。

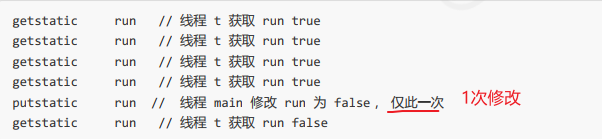
*//添加volatile后，线程会一直在主存中读取值，不再使用缓存。***volatile static boolean** *run* = **true**;  
  
@Test  
**public void** test1() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*run*) {  
 *//...* }  
 }, **"t1"**);  
 t1.start();  
  
 **try** {  
 Thread.*sleep*(1000);  
 *run* = **false**; *//t1线程不会停止* t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

也可以使用synchronized保证共享变量可见性。

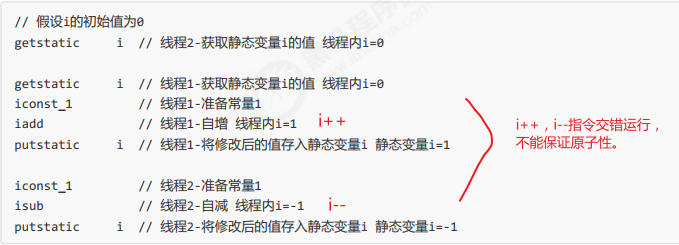
*//使用synchronized解决可见性问题***static boolean** *flag* = **true**;  
**static final** Object ***lock*** = **new** Object();  
  
@Test  
**public void** test2() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*run*) {  
 **synchronized** (***lock***) {  
 *//...* **if** (!*run*) {  
 **break**;  
 }  
 }  
 }  
 }, **"t1"**);  
 t1.start();  
 **try** {  
 Thread.*sleep*(1000);  
 *run* = **false**; *//t1线程不会停止* t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

4、可见性和原子性区别

前面例子体现的实际就是可见性，它保证的是在多个线程之间，一个线程对 volatile 变量的修改对另一个线程可见， 不能保证原子性，仅用在一个写线程，多个读线程的情况： 上例从字节码理解是这样的：



比较一下之前我们将线程安全时举的例子：两个线程一个 i++ 一个 i-- ，只能保证看到最新值，不能解决指令交错。



注意 ：

synchronized 语句块既可以保证代码块的原子性，也同时保证代码块内变量的可见性。但缺点是synchronized 是属于重量级操作，性能相对更低。

如果在前面示例的死循环中加入 System.out.println() 会发现即使不加 volatile 修饰符，线程 t 也能正确看到对 run 变量的修改了，想一想为什么？

*//在while块中添加 System.out.println();***static boolean** *flag3* = **true**;  
  
@Test  
**public void** test3() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*flag3*) {  
 *//...* System.***out***.println(**"sout"**);  
 }  
 }, **"t1"**);  
 t1.start();  
  
 **try** {  
 Thread.*sleep*(1000);  
 *flag3* = **false**; *//t1线程不会停止* t1.join();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

原因分析：

*/\*\*  
 \* Prints a String and then terminate the line. This method behaves as  
 \* though it invokes <code>{****@link*** *#print(String)}</code> and then  
 \* <code>{****@link*** *#println()}</code>.  
 \*  
 \** ***@param x*** *The <code>String</code> to be printed.  
 \*/***public void** println(String x) {  
 **synchronized** (**this**) {  
 print(x);  
 newLine();  
 }  
}

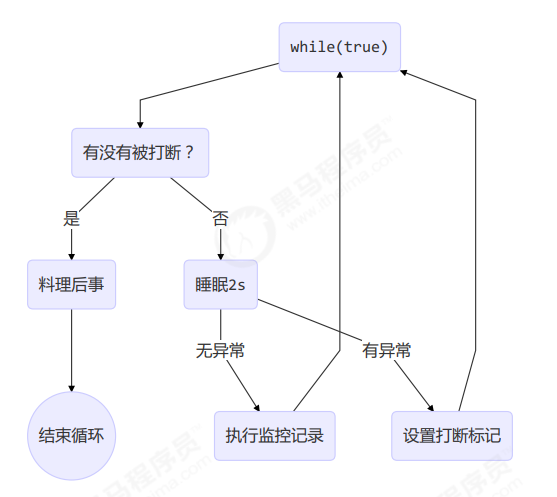
println()方法使用了synchronized。

终止模式-两阶段终止模式

1. interrupt实现

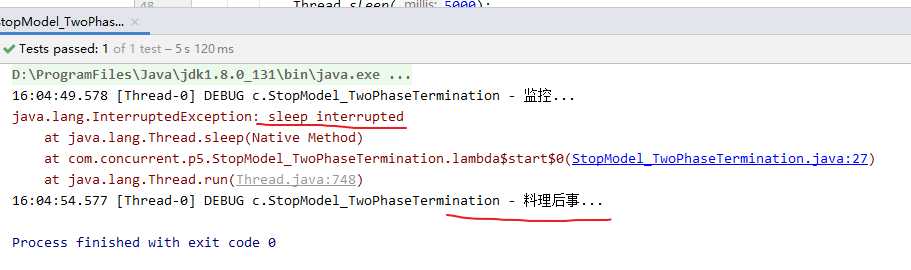
2、volatile实现

在一个线程 T1 中如何“优雅”终止线程 T2？这里的【优雅】指的是给 T2 一个料理后事的机会。



**package** com.concurrent.p5;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
*/\*\*  
 \* 两阶段终止模式  
 \*/*@Slf4j(topic = **"c.StopModel\_TwoPhaseTermination"**)  
**public class** StopModel\_TwoPhaseTermination {  
 *//监控线程* **private** Thread **monitorThread**;  
 *//控制线程停止* **volatile private boolean stop** = **false**;  
  
 *//启动监控线程* **public void** start() {  
 **monitorThread** = **new** Thread(() -> {  
 **while** (**true**) {  
 **if** (**stop**) {  
 ***log***.debug(**"料理后事..."**);  
 **break**;  
 }  
 **try** {  
 ***log***.debug(**"监控..."**);  
 Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 });  
 **monitorThread**.start();  
 }  
  
 *//停止方法* **public void** stop() {  
 **stop** = **true**;  
 *//如果虽然修改了stop，但是线程正在sleep，依然可以使用interrupt()打断sleep* **monitorThread**.interrupt();  
 }  
  
 @Test  
 **public void** test\_StopModel\_TwoPhaseTermination() {  
 StopModel\_TwoPhaseTermination monitor = **new** StopModel\_TwoPhaseTermination();  
 monitor.start();  
 **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 monitor.stop();  
 }  
}

运行结果：



同步模式-Balking

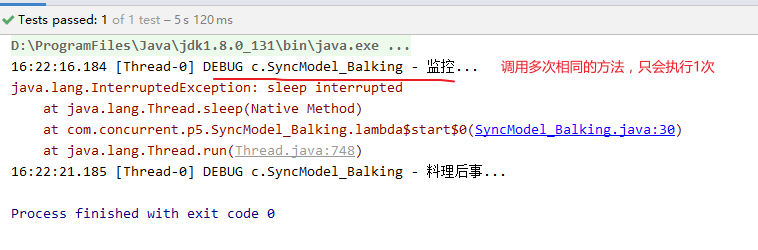
1、概念

Balking （犹豫）模式用在一个线程发现另一个线程或本线程已经做了某一件相同的事，那么本线程就无需再做了，直接结束返回。

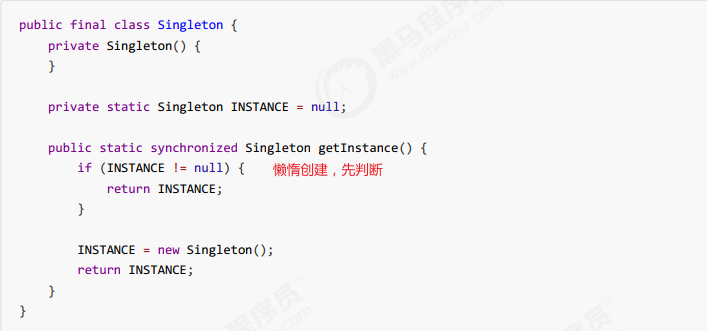
2、代码

**package** com.concurrent.p5;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
@Slf4j(topic = **"c.SyncModel\_Balking"**)  
**public class** SyncModel\_Balking {  
 *//监控线程* **private** Thread **monitorThread**;  
 *//控制线程停止* **volatile private boolean stop** = **false**;  
 *//判断同一个方法是否被执行过* **private boolean start** = **false**;  
  
 *//启动监控线程* **public void** start() {  
 **synchronized** (**this**) { *//要加同步，因为volatile只能保证可见性，不能保证原子性* **if** (**start**) { *//被执行过，直接结束* **return**;  
 }  
 **start** = **true**; *//将标识设置为true* }  
 **monitorThread** = **new** Thread(() -> {  
 **while** (**true**) {  
 **if** (**stop**) {  
 ***log***.debug(**"料理后事..."**);  
 **break**;  
 }  
 **try** {  
 ***log***.debug(**"监控..."**);  
 Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 **monitorThread**.start();  
 }  
  
 *//停止方法* **public void** stop() {  
 **stop** = **true**;  
 *//如果虽然修改了stop，但是线程正在sleep，依然可以使用interrupt()打断sleep* **monitorThread**.interrupt();  
 }  
  
 @Test  
 **public void** test\_SyncModel\_Balking() {  
 SyncModel\_Balking monitor = **new** SyncModel\_Balking();  
 monitor.start();  
 monitor.start();  
 monitor.start();  
 **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 monitor.stop();  
 }  
}

运行结果：



它还经常用来实现线程安全的单例，



对比一下保护性暂停模式：

保护性暂停模式用在一个线程等待另一个线程的执行结果，当条件不满足时线程等待。

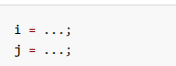
有序性

1、概念

JVM 会在不影响正确性的前提下，可以调整语句的执行顺序，思考下面一段代码。



可以看到，至于是先执行 i 还是 先执行 j ，对最终的结果不会产生影响。所以，上面代码真正执行时，既可以是



或者



这种特性称之为『指令重排』。

多线程下『指令重排』会影响正确性。

为什么要有重排指令这项优化呢？从 CPU执行指令的原理来理解：

2、指令重排原理\*\*\*

Clock Cycle Time

主频的概念大家接触的比较多，而 CPU 的 Clock Cycle Time（时钟周期时间），等于主频的倒数，意思是 **CPU 能够识别的最小时间单位**，比如说 4G 主频的 CPU 的 Clock Cycle Time 就是 0.25 ns，作为对比，我们墙上挂钟的Cycle Time 是 1s。

例如，运行一条加法指令一般需要一个时钟周期时间

CPI

有的指令需要更多的时钟周期时间，所以引出了 CPI （Cycles Per Instruction）指令**平均时钟周期数**

IPC

IPC（Instruction Per Clock Cycle） 即 CPI 的倒数，表示**每个时钟周期能够运行的指令数**

CPU 执行时间

程序的 CPU 执行时间，即我们前面提到的 user + system 时间，可以用下面的公式来表示：

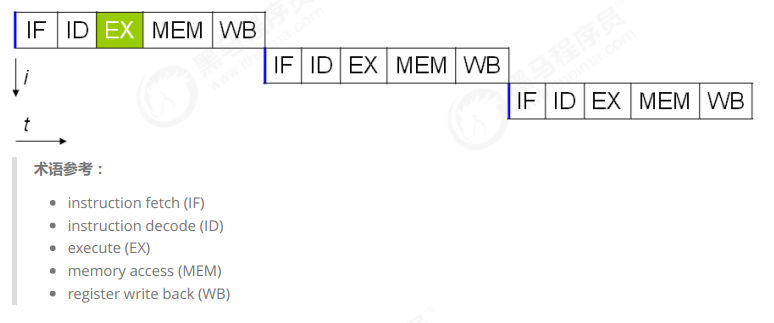
**程序 CPU 执行时间 = 指令数 \* CPI \* Clock Cycle Time**

指令重排序优化

事实上，现代处理器会设计为一个时钟周期完成一条执行时间最长的 CPU 指令。为什么这么做呢？可以想到指令还可以再划分成一个个更小的阶段，例如，每条指令都可以分为：

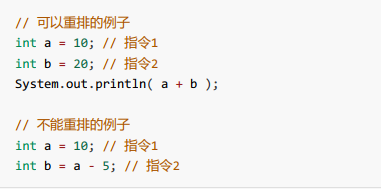
**【取指令 - 指令译码 - 执行指令 - 内存访问 - 数据写回】**

这 5 个阶段



在不改变程序结果的前提下，这些指令的各个阶段可以通过重排序和组合来实现指令级并行。

指令重排的前提是，重排指令不能影响结果，例如

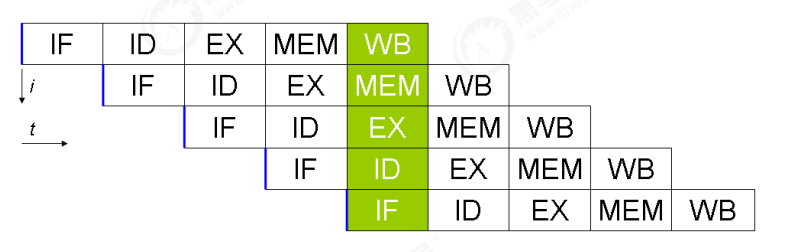


支持流水线的处理器

现代 CPU 支持多级指令流水线，例如支持同时执行 取指令 - 指令译码 - 执行指令 - 内存访问 - 数据写回 的处理器，就可以称之为五级指令流水线。

这时 CPU 可以在一个时钟周期内，同时运行五条指令的不同阶段（相当于一条执行时间最长的复杂指令），IPC = 1，

本质上，流水线技术并不能缩短单条指令的执行时间，但它**变相地提高了指令地吞吐率**。



volatile原理\*\*\*

3、指令重排导致的问题



【情况1】先执行线程1，再执行线程2，r1=1

【情况2】先执行线程2，再执行线程1，r1=4

【情况3】先执行2，但发生指令重排，先执行ready=true，r1=0

情况3可以使用并发压测工具复现：

借助 java 并发压测工具 jcstress <https://wiki.openjdk.java.net/display/CodeTools/jcstress>

mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=org.openjdk.jcstress -  
DarchetypeArtifactId=jcstress-java-test-archetype -DarchetypeVersion=0.5 -DgroupId=cn.itcast -  
DartifactId=ordering -Dversion=1.0

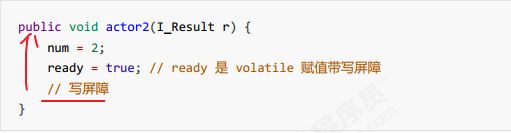
1. volatile原理

volatile 的底层实现原理是内存屏障，Memory Barrier（Memory Fence）

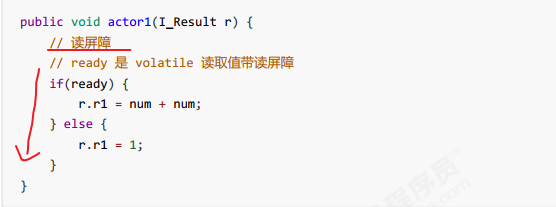
* 对 volatile 变量的写指令后会加入写屏障
* 对 volatile 变量的读指令前会加入读屏障

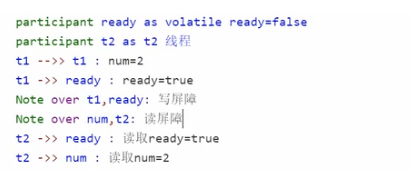
如何保证可见性？

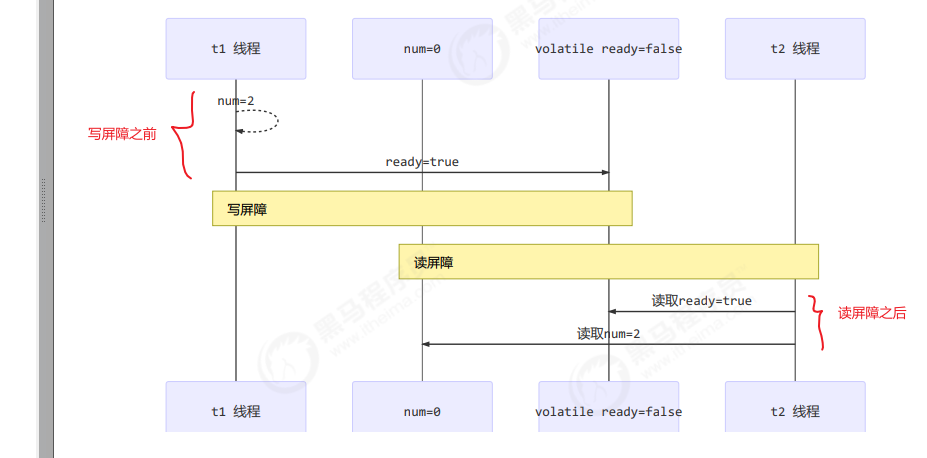
写屏障（sfence）保证在该屏障之前的，对共享变量的改动，都同步到主存当中。



而读屏障（lfence）保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据。

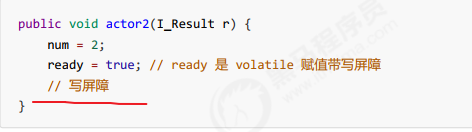




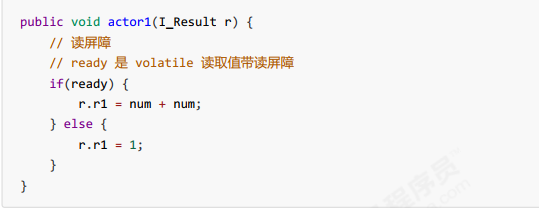


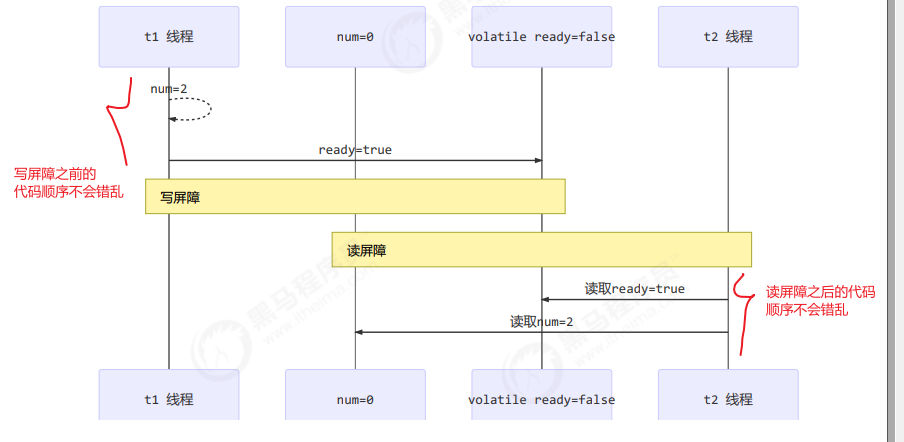
如何保证有序性？

写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后。



读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前。

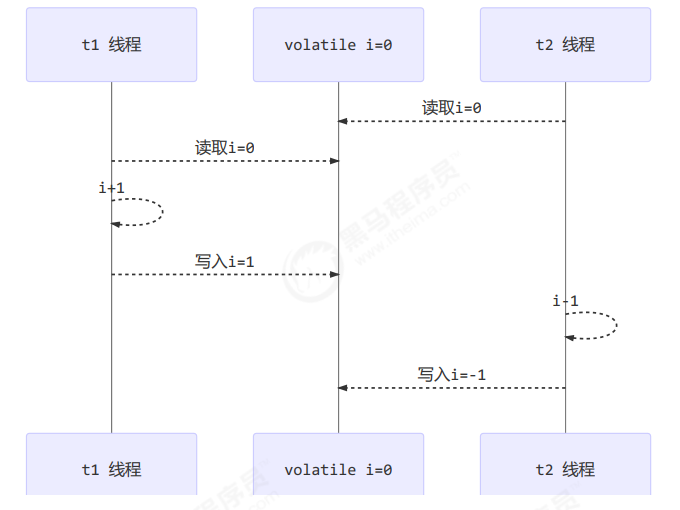




还是那句话，volatile不能解决指令交错：

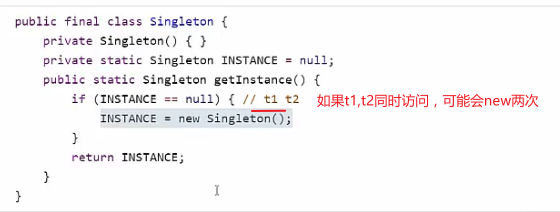
写屏障仅仅是保证之后的读能够读到最新的结果，但不能保证读跑到它前面去

而有序性的保证也只是保证了本线程内相关代码不被重排序。



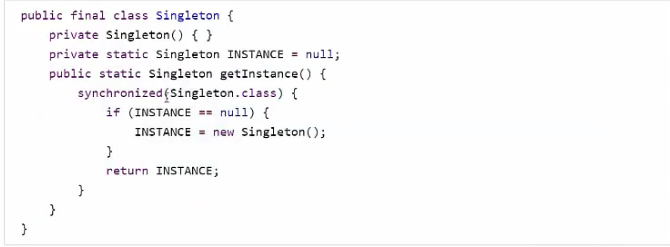
double-checked locking 问题

不加synchronized，



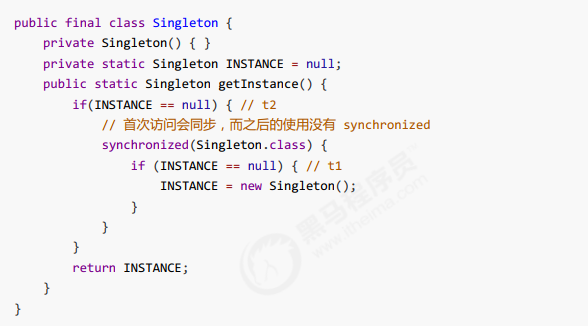
单次判断：

每调用1次，就进入1次synchronized，效率低。



改进双判断：

首次可以创建，进入synchronized块；其他线程在调用的时候，会在第一个判断处结束，不会进入synchronized。

  
以上的实现特点是：

* 懒惰实例化
* 首次使用 getInstance() 才使用 synchronized 加锁，后续使用时无需加锁
* 有隐含的，但很关键的一点：第一个 if 使用了 INSTANCE 变量，是在同步块之外

double-checked locking 在多线程下存在指令重排的问题，



但在多线程环境下，上面的代码是有问题的，getInstance 方法对应的字节码为：

0: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton; 获取静态变量INSTANCE  
3: ifnonnull 37 判断静态对象是否为空，如果不为空跳转到37行  
6: ldc #3 // class cn/itcast/n5/Singleton 开始加锁，获取类对象  
8: dup 复制类对象的引用  
9: astore\_0 把类对象的指针临时存储一份，将来解锁用  
10: monitorenter 创建monitor，加锁  
11: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton; 获取静态变量   
14: ifnonnull 27 判断不为空  
**17: new #3 // class cn/itcast/n5/Singleton 创建Singleton实例  
20: dup 复制一份实例  
21: invokespecial #4 // Method "<init>":()V 调用构造方法  
24: putstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton; 赋值操作，将对象赋值给静态变量**27: aload\_0 获取Singleton.class类对象引用  
28: monitorexit Monitor解锁  
29: goto 37 进到37行  
32: astore\_1   
33: aload\_0  
34: monitorexit  
35: aload\_1  
36: athrow  
37: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton; 获取静态变量  
40: areturn 返回静态对象

其中

17 表示创建对象，将对象引用入栈 // new Singleton

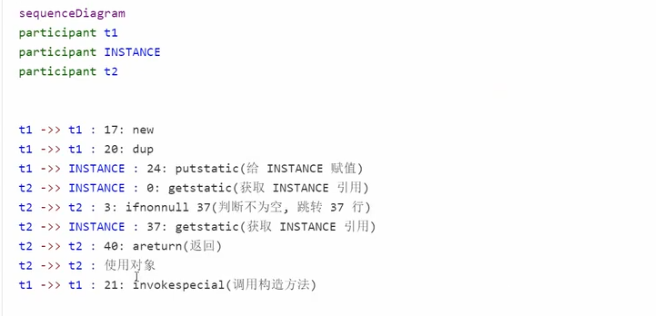
20 表示复制一份对象引用 // 引用地址

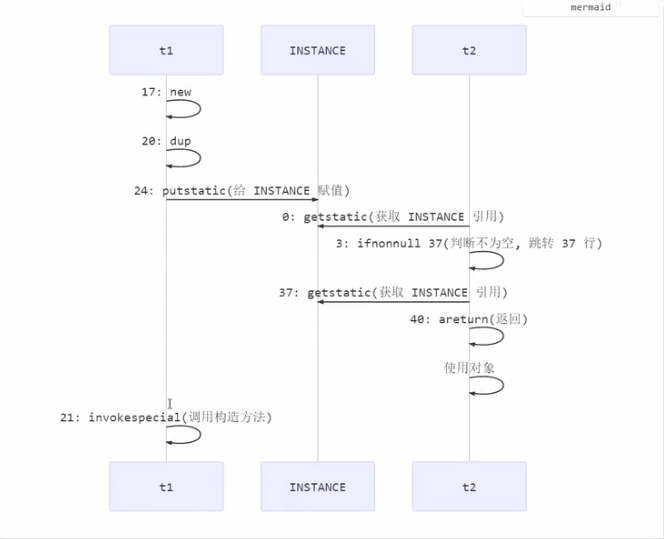
21 表示利用一个对象引用，调用构造方法

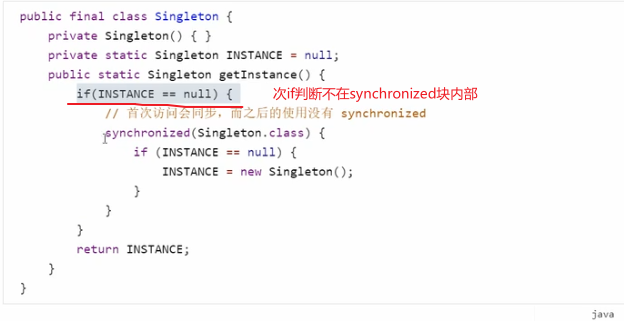
24 表示利用一个对象引用，赋值给 static INSTANCE

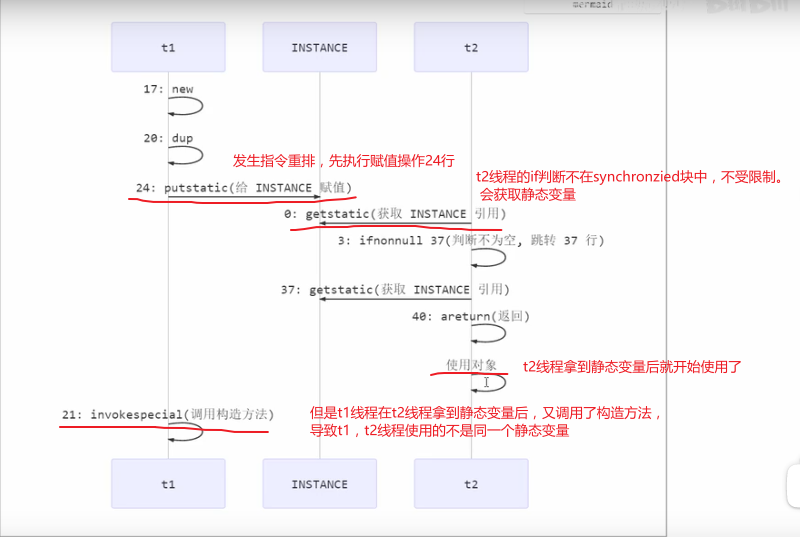
INSTANCE = new Singleton(); 正常是先调用无参构造再进行赋值；但是这不是一个原子操作，可能会先赋值，再调用无参构造。导致原来的值被new 出来的新对象覆盖。

也许 jvm 会优化为：先执行 24，再执行 21。如果两个线程 t1，t2 按如下时间序列执行：









* 关键在于 0: getstatic 这行代码在 monitor 控制之外，它就像之前举例中不守规则的人，可以越过 monitor 读取INSTANCE 变量的值。
* 这时 t1 还未完全将构造方法执行完毕，如果在构造方法中要执行很多初始化操作，那么 t2 拿到的是将是一个未初始化完毕的单例。
* 对 INSTANCE 使用 volatile 修饰即可，可以禁用指令重排，但要注意在 JDK 5 以上的版本的 volatile 才会真正有效。

synchronized不能阻止重排序，volatile可以阻止重排序。

但是如果共享变量完全包含在synchronzied中，则是可以保证可见性、原子性、禁止指令重排的。

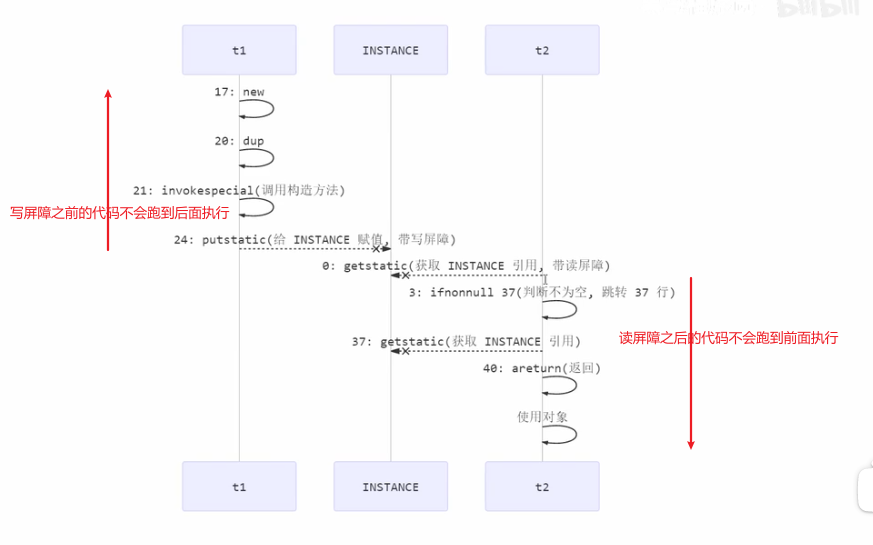
不能一部分在synchronized之外，脱离synchronzied的管理。

double-checked locking 解决方法

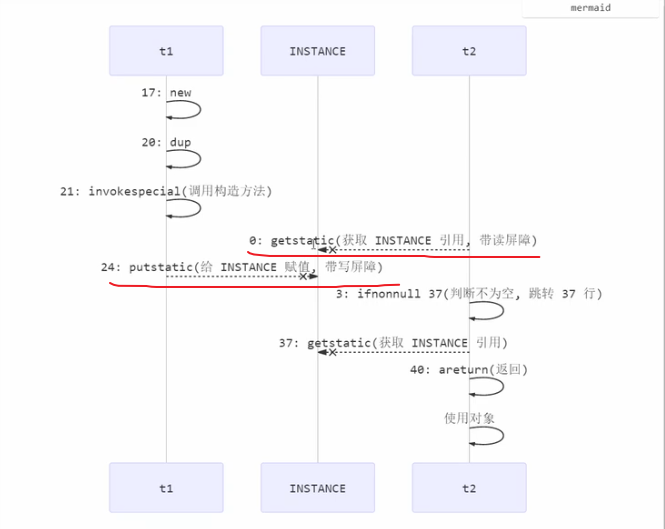


// -------------------------------------> 加入对 INSTANCE 变量的读屏障  
0: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;

3: ifnonnull 37  
6: ldc #3 // class cn/itcast/n5/Singleton  
8: dup  
9: astore\_0  
10: monitorenter -----------------------> 保证原子性、可见性  
11: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;  
14: ifnonnull 27  
**17: new #3 // class cn/itcast/n5/Singleton  
20: dup  
21: invokespecial #4 // Method "<init>":()V  
24: putstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;**// -------------------------------------> 加入对 INSTANCE 变量的写屏障  
27: aload\_0  
28: monitorexit ------------------------> 保证原子性、可见性  
29: goto 37  
32: astore\_1  
33: aload\_0  
34: monitorexit  
35: aload\_1  
36: athrow  
37: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;  
40: areturn



即使t2线程的getstatic发生在t1线程putstatic之前也可以保证正确性。因为此时t2获取到的是空静态变量，还是会进入synchronized同步块等待。



如上面的注释内容所示，读写 volatile 变量时会加入内存屏障（Memory Barrier（Memory Fence）），保证下面两点：

【1】可见性

* 写屏障（sfence）保证在该屏障之前的 t1 对共享变量的改动，都同步到主存当中。
* 读屏障（lfence）保证在该屏障之后 t2 对共享变量的读取，加载的是主存中最新数据。

【2】有序性

* 写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后
* 读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前

更底层是读写变量时使用 lock 指令来多核 CPU 之间的可见性与有序性。

1. happens-before规则

happens-before 规定了对共享变量的写操作对其它线程的读操作可见，它是可见性与有序性的一套规则总结，抛开以下 happens-before 规则，JMM 并不能保证一个线程对共享变量的写，对于其它线程对该共享变量的读可见。

【1】线程解锁 m 之前对变量的写，对于接下来对 m 加锁的其它线程对该变量的读可见。



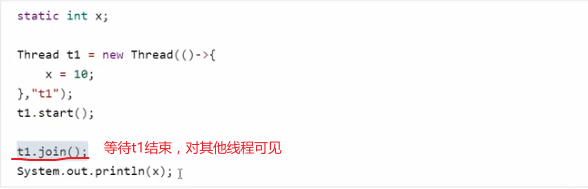
【2】线程对 volatile 变量的写，对接下来其它线程对该变量的读可见。



【3】线程启动（start）前对变量的写，对该线程开始后对该变量的读可见。



【4】线程结束前对变量的写，对其它线程得知它结束后的读可见（比如其它线程调用 t1.isAlive() 或 t1.join()等待它结束）。

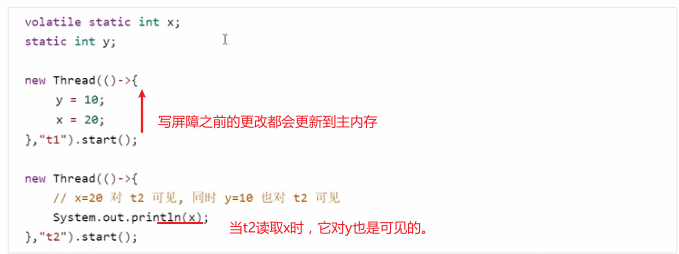


【5】线程 t1 打断 t2（interrupt）前对变量的写，对于其他线程得知 t2 被打断后对变量的读可见（通过t2.interrupted 或 t2.isInterrupted）



【6】对变量默认值（0，false，null）的写，对其它线程对该变量的读可见。

【7】具有传递性，如果 x hb-> y 并且 y hb-> z 那么有 x hb-> z ，配合 volatile 的防指令重排，有下面的例子。



1. 习题

【1】balking模式

希望 doInit() 方法仅被调用一次，下面的实现是否有问题，为什么？



【2】线程安全单例

单例模式有很多实现方法，饿汉、懒汉、静态内部类、枚举类，试分析每种实现下获取单例对象（即调用getInstance）时的线程安全，并思考注释中的问题。

* 饿汉式：类加载就会导致该单实例对象被创建。
* 懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建。

实现1



问题1：阻止子类继承该类，破坏了单例方法。

问题2：创建对象可以不用new实现，采用反序列化创建对象，但反序列化会破环单例。

要添加readResolve()方法，返回类中的实例，而不是字节码实例。

public Object readResolve() {

return INSTANCE;

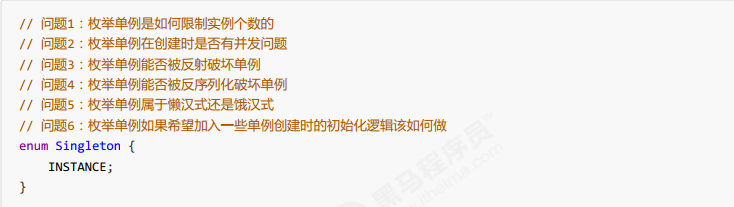
}

问题3：如果设置成public，那么外部可以调用该方法创建对象。不能防止通过暴力反射的方法创建新实例。

问题4：类加载阶段对成员变量的赋值都是安全的。

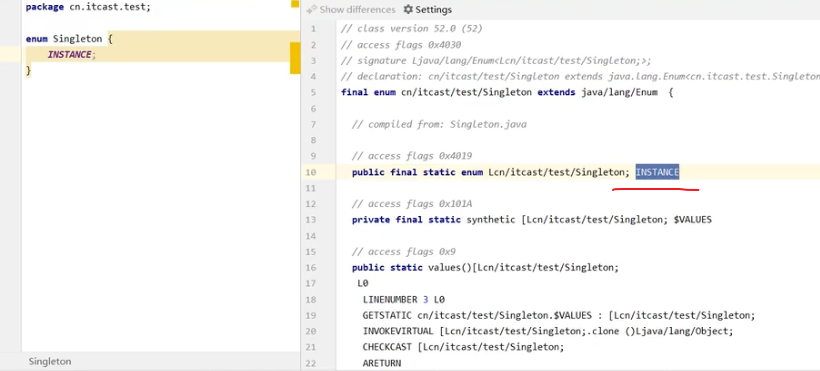
问题5：可以封装，提供懒惰的初始化。

实现2



问题1：

枚举类的静态成员变量，单实例的。



问题2：

静态成员变量在类加载阶段完成，不存在线程安全问题。

问题3：

不能用反射破坏单例。

问题4：

枚举类默认实现序列化接口，可以避免反序列化破坏单例。

问题5：

枚举单例属于饿汉式。

问题6：

加构造方法。

实现3



这里是线程安全，但是每次调用都会加锁，效率低。

实现4



问题1：

加上volatile可以防止指令重排到的空的实例。

问题2：

首次判断没有加synchronized，可以提高代码的效率。

问题3：

防止阻塞中的线程创建新的实例，加上判断之后，阻塞中的线程发现其他已经创建好实例时，会结束。

实现5



问题1：

属于饿汉式，使用时调用方法创建。

问题2：

没有并发问题，创建时jvm保证安全性。