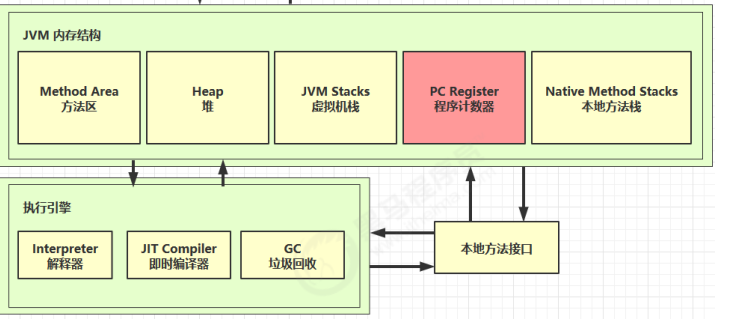
Java内存模型

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

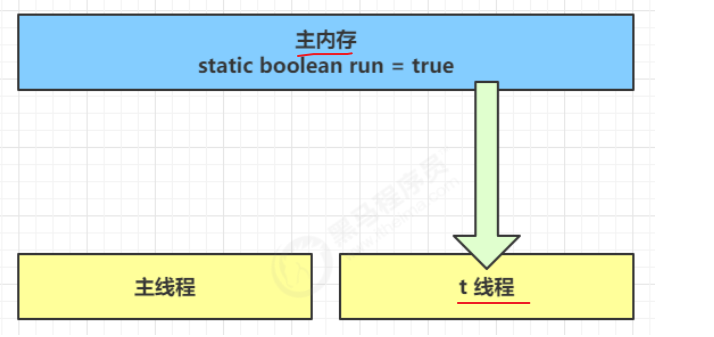
1、可见性问题

*//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();  
 }  
}

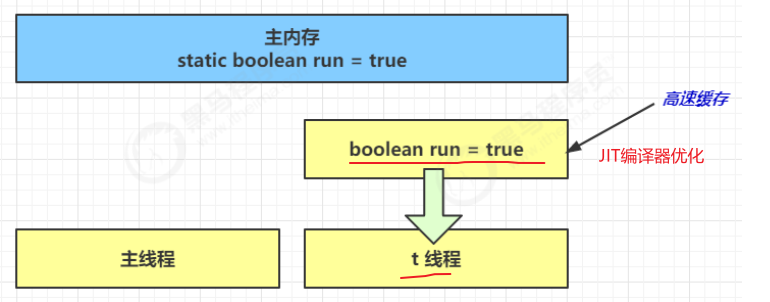
在主线程中修改run=false后，子线程t1不会停止。

2、原因分析

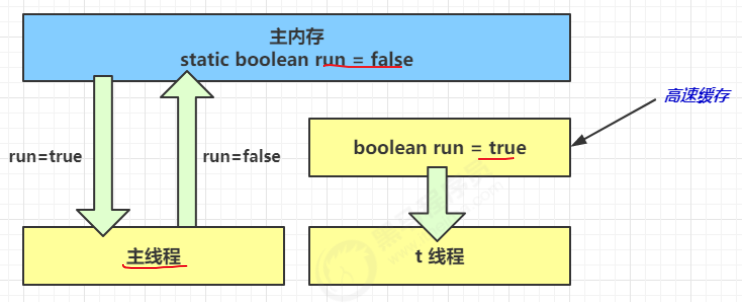
【1】初始状态， t 线程刚开始从主内存读取了 run 的值到工作内存。



【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。

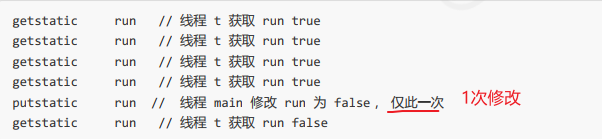
*//添加volatile后，线程会一直在主存中读取值，不再使用缓存。***volatile static boolean** *run* = **true**;  
  
@Test  
**public void** test1() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*run*) {  
 *//...* }  
 }, **"t1"**);  
 t1.start();  
  
 **try** {  
 Thread.*sleep*(1000);  
 *run* = **false**; *//t1线程不会停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

也可以使用synchronized保证共享变量可见性。

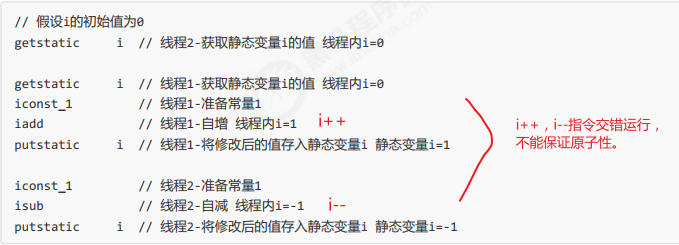
*//使用synchronized解决可见性问题***static boolean** *flag* = **true**;  
**static final** Object ***lock*** = **new** Object();  
  
@Test  
**public void** test2() {  
 Thread t1 = **new** Thread(() -> {  
 **while** (*run*) {  
 **synchronized** (***lock***) {  
 *//...* **if** (!*run*) {  
 **break**;  
 }  
 }  
 }  
 }, **"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();  
 }  
}

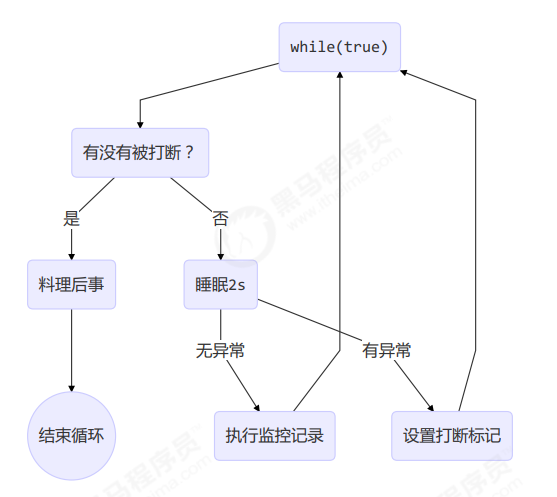
println()方法使用了synchronized。

终止模式-两阶段终止模式

1. interrupt实现

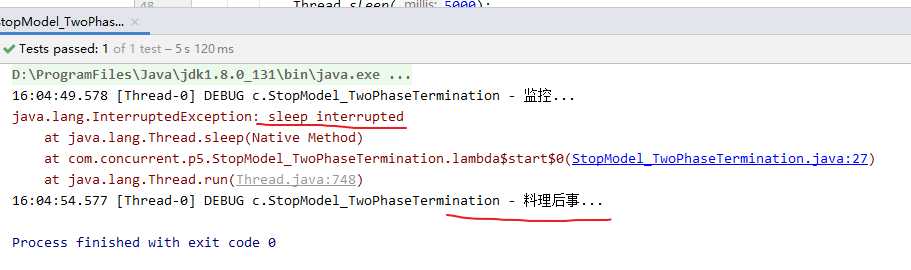
2、volatile实现

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：

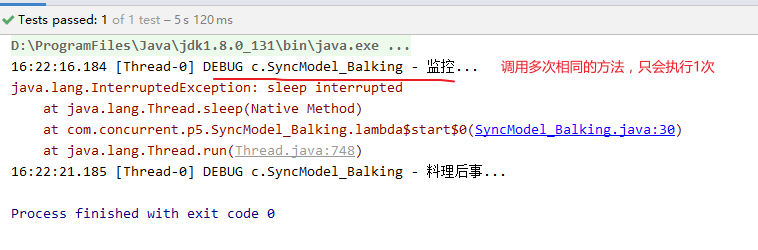


同步模式-Balking

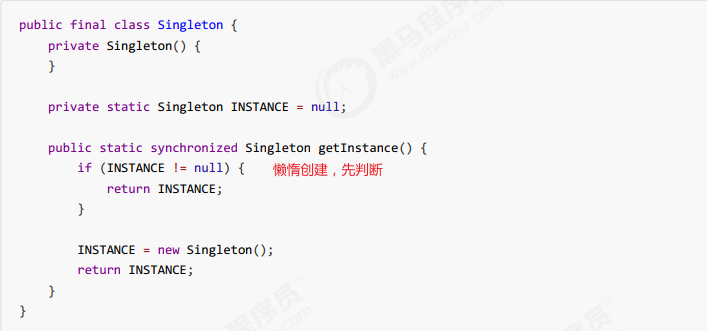
Balking （犹豫）模式用在一个线程发现另一个线程或本线程已经做了某一件相同的事，那么本线程就无需再做了，直接结束返回。

**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。

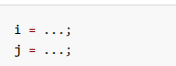
有序性

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：

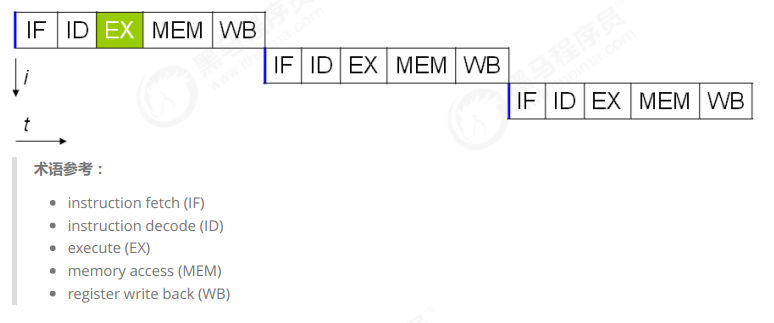
**程序 CPU 执行时间 = 指令数 \* CPI \* Clock Cycle Time**

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：

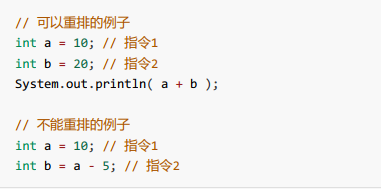
**【取指令 - 指令译码 - 执行指令 - 内存访问 - 数据写回】**

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

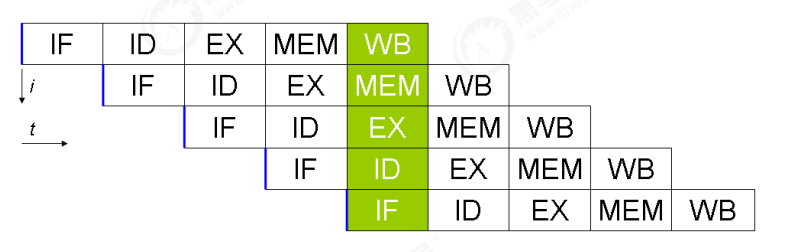


支持流水线的处理器

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

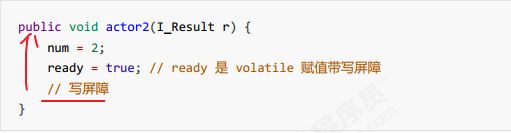
1. volatile原理

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

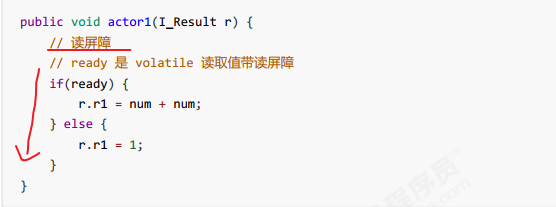
* 对 volatile 变量的写指令后会加入写屏障
* 对 volatile 变量的读指令前会加入读屏障

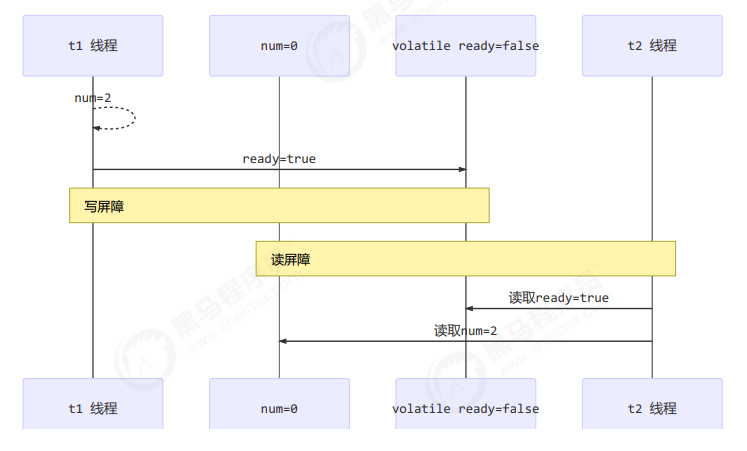
如何保证可见性？

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。



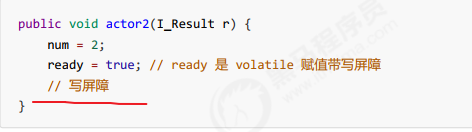
而读屏障（lfence）保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据。



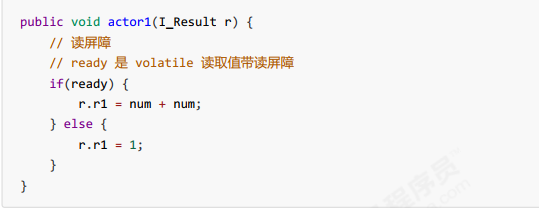


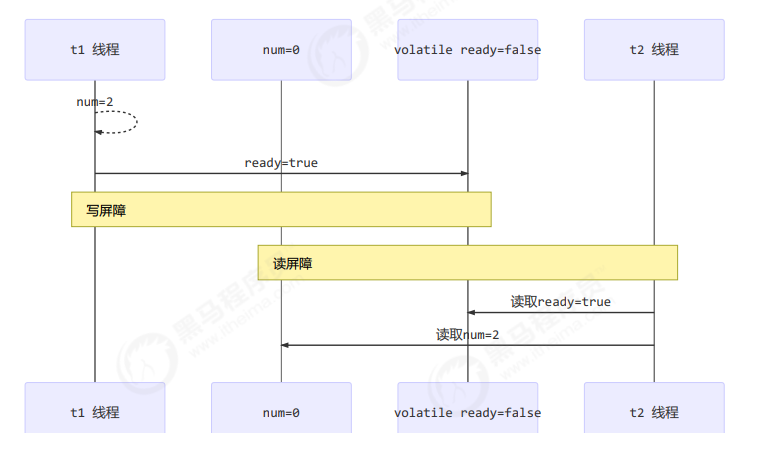
如何保证有序性？

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。

