共享模型之内存

共享模型之内存
一. JMM（Java Memory Model）- Java内存模型
1.1 概述
1.2 可见性
1.2.1 无法退出循环示例
1.2.2 上述情况出现原因
1.2.3 解决方案
1.2.4 可见性 vs 原子性
1.2.5终止模式之两阶段终止模式
问题
错误思路
过往思路
使用volatile
1.2.6 同步模式之Balking模式
1.3 有序性
1.3.1 概述
1.3.2 指令重排序
1.3.3 诡异的结果
1.3.4 volatile原理
如何保证可见性
如何保证有序性
volatile无法解决的问题
double-checked locking 问题
double-checked locking问题解决方法
1.3.5 happens-before

一. JMM（Java Memory Model）- Java内存模型

1.1 概述

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

JMM体现在下面这些方面：

- 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

1.2 可见性

1.2.1 无法退出循环示例

```
@Slf4j
public class VolatileDemo {
    static boolean run = true;
    public static void main(String[] args) throws
InterruptedException {
        Thread t1 = new Thread(() -> {
            while (run) {
            }
            log.debug("{}结束执行",
Thread.currentThread().getName());
        });

        t1.start();
        Thread.sleep(1000);

        log.debug("设置run的值为false");
        run = false;
    }
}
```

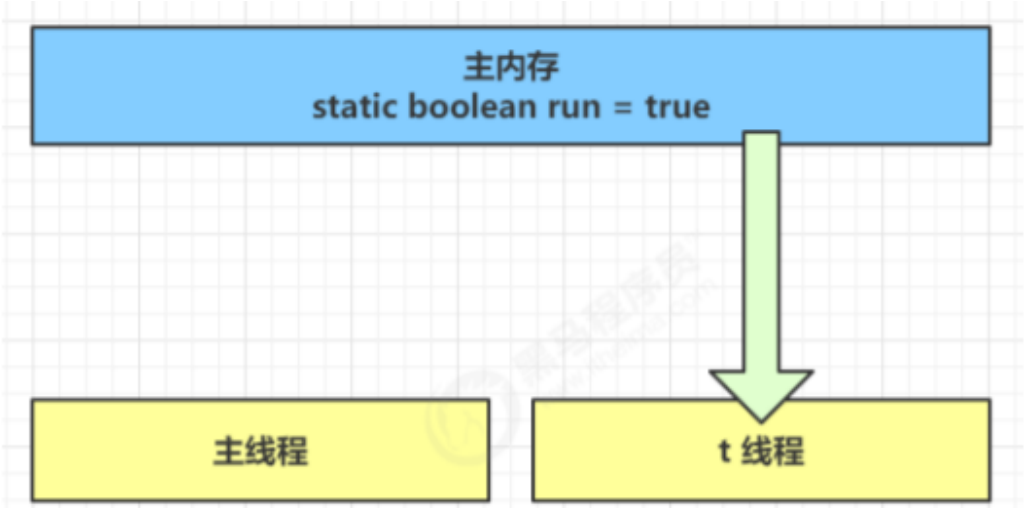
输出结果

```
[main] DEBUG com.ecifics.concurrent.part5.VolatileDemo - 设置run的
值为false
```

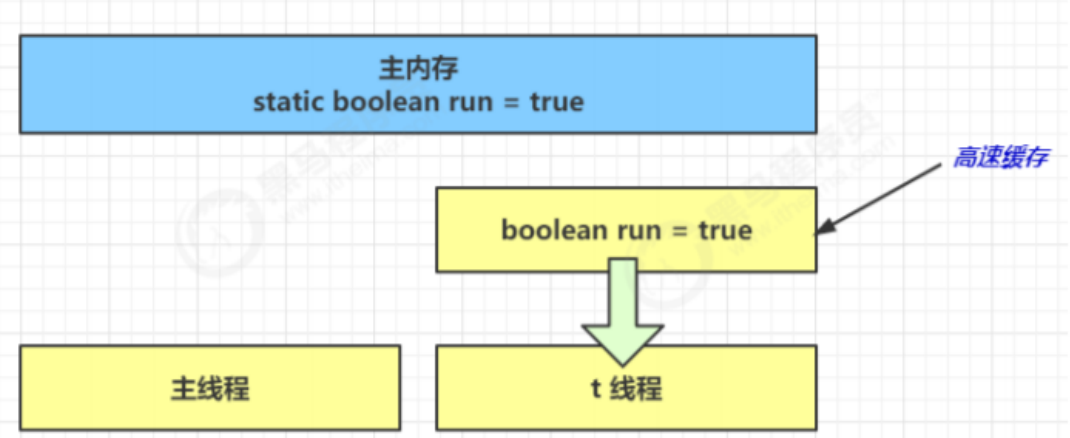
在运行这个程序时，即使后面将run的值修改成false之后，依然无法让线程停止下来，线程中的while一直在循环进行，因此始终没有输出线程结束的日志

1.2.2 上述情况出现原因

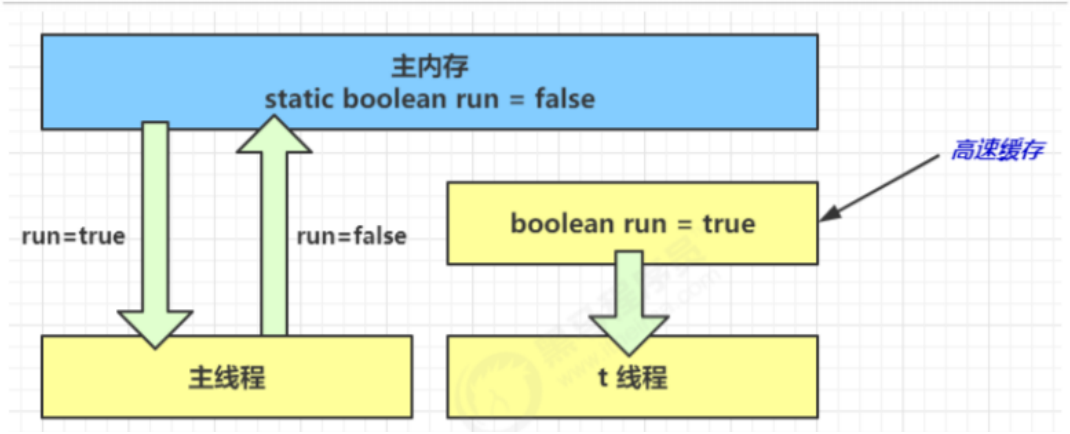
- 初始状态， t线程刚开始从主内存(成员变量), 因为主线程sleep(1)秒, 这时候t1线程循环了好多次run的值, 超过了一定的阈值, JIT就会将主存中的run值读取到工作内存 (相当于缓存了一份, 不会去主存中读run的值了)。



- 因为t1线程频繁地从主存中读取run的值，JIT即时编译器会将run的值缓存至自己工作内存中的高速缓存中，减少对主主存中run的访问以提高效率



- 1 秒之后，main线程修改了run的值, 并同步至主存。而 t线程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**



1.2.3 解决方案

volatile（关键字）

它可以用来修饰成员变量和静态成员变量，它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

修改过后的代码如下

```
@Slf4j
public class VolatileDemo {
    volatile static boolean run = true;
    public static void main(String[] args) throws
InterruptedException {
        Thread t1 = new Thread(() -> {
            while (run) {
            }
            log.debug("{}结束执行",
Thread.currentThread().getName());
        });

        t1.start();
        Thread.sleep(1000);

        log.debug("设置run的值为false");
        run = false;
    }
}
```

或者可以使用synchronized关键字进行解决

```
@Slf4j
public class VolatileDemo {
    static boolean run = true;

    final static Object lock = new Object();

    public static void main(String[] args) throws
InterruptedException {
        Thread t1 = new Thread(() -> {
            while (true) {
                synchronized (lock) {
                    if (!run) {
                        break;
```

```
                    }
                }
            }
            log.debug("{}结束执行",
Thread.currentThread().getName());
        });

        t1.start();
        Thread.sleep(1000);

        log.debug("设置run的值为false");
        synchronized (lock) {
            run = false;
        }
    }
}
```

运行结果

```
[main] DEBUG com.ecifics.concurrent.part5.VolatileDemo - 设置run的
值为false
[Thread-0] DEBUG com.ecifics.concurrent.part5.VolatileDemo -
Thread-0结束执行

Process finished with exit code 0
```

1.2.4 可见性 vs 原子性

上述例子体现了可见性，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**

上述例子从字节码理解是这样的：

```
getstatic run // 线程 t 获取 run true
getstatic run // 线程 t 获取 run true
getstatic run // 线程 t 获取 run true
getstatic run // 线程 t 获取 run true
putstatic run // 线程 main 修改 run 为 false,  仅此一次
getstatic run // 线程 t 获取 run false
```

比较一下之前我们讲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(可见性)，不能解决指令交错(原子性)**

```
// 假设i的初始值为0
getstatic i // 线程2-获取静态变量i的值 线程内i=0
getstatic i // 线程1-获取静态变量i的值 线程内i=0
iconst_1 // 线程1-准备常量1
iadd // 线程1-自增 线程内i=1
putstatic i // 线程1-将修改后的值存入静态变量i 静态变量i=1
iconst_1 // 线程2-准备常量1
isub // 线程2-自减 线程内i=-1
putstatic i // 线程2-将修改后的值存入静态变量i 静态变量i=-1
```

> 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？因为 println方法里面有 `synchronized修饰` 。还有那个等烟的示例, 为啥没有出现可见性问题?和synchrozized是一个道理。

1.2.5终止模式之两阶段终止模式

问题

在一个线程T1中如何**优雅**的终止线程T2呢？（这里的优雅是指如何给T2一个料理后事的机会）

错误思路

- 使用线程对象的stop()方法
  - stop方法会真正的杀死线程，如果这个线程锁住了共享资源，那么当它被杀死后就**没有机会释放锁了，其他线程将永远无法获得锁**
- 使用System.exit(int)方法停止线程
  - 目的仅仅是停止一个线程，这个做法会让整个程序都停止

过往思路

之前是采用interrupt方法进行优雅打断

```
@Slf4j
public class TerminateThreadPattern {

    public static void main(String[] args) throws
InterruptedException {
        TerminateThreadPattern terminateThreadPattern = new
TerminateThreadPattern();
        terminateThreadPattern.start();

        Thread.sleep(3500);
        log.debug("停止监控");
        terminateThreadPattern.stop();
    }

    private Thread monitorThread;

    public void start() {
        monitorThread = new Thread(() -> {
            while (true) {
                Thread currentThread = Thread.currentThread();

                if (currentThread.isInterrupted()) {
                    log.debug("料理后事");
                    break;
                }

                try {
                    Thread.sleep(1000);
                    log.debug(("执行监控记录"));
                } catch (InterruptedException e) {
                    //sleep出现异常后，会清除打断标记，需要重新设置标记
                    currentThread.interrupt();
                }
            }
        }, "monitor");

        monitorThread.start();
    }

    public void stop() {
        monitorThread.interrupt();
```

```
    }
}
```

使用volatile

```
@Slf4j
public class TerminateThreadPattern {

    public static void main(String[] args) throws
InterruptedException {
        TerminateThreadPattern terminateThreadPattern = new
TerminateThreadPattern();
        terminateThreadPattern.start();

        Thread.sleep(3500);
        log.debug("停止监控");
        terminateThreadPattern.stop();
    }

    private Thread monitorThread;

    private volatile boolean stop;

    public void start() {
        monitorThread = new Thread(() -> {
            while (true) {
                Thread currentThread = Thread.currentThread();

                if (stop) {
                    log.debug("料理后事");
                    break;
                }

                try {
                    Thread.sleep(1000);
                    log.debug(("执行监控记录"));
                } catch (InterruptedException e) {
                    //sleep出现异常后，会清除打断标记，需要重新设置标记
                    currentThread.interrupt();
                }
            }
        }, "monitor");
```

```
        monitorThread.start();
    }

    public void stop() {
        stop = true;
        monitorThread.interrupt();
    }
}
```

1.2.6 同步模式之Balking模式

Balking模式用在一个线程发现另一个线程或者本线程已经作了某一件相同的事，那么本线程就无需再做了，直接结束返回，上面示例的代码经过改变可以称为Balking模式

```
@Slf4j
public class TerminateThreadPattern {

    public static void main(String[] args) throws
InterruptedException {
        TerminateThreadPattern terminateThreadPattern = new
TerminateThreadPattern();
        terminateThreadPattern.start();
        terminateThreadPattern.start();
        terminateThreadPattern.start();

        Thread.sleep(3500);
        log.debug("停止监控");
        terminateThreadPattern.stop();
    }

    private Thread monitorThread;

    private volatile boolean stop;

    /**
     * 判断是否执行过start方法
     */
    private boolean starting = false;

    public void start() {
        //加锁是为了防止第二个进程调用start()方法时，第一个进程还未修改
starting的值
```

```
        synchronized (this) {
            if (starting) {
                return ;
            }

            starting = true;
        }


        monitorThread = new Thread(() -> {
            while (true) {
                Thread currentThread = Thread.currentThread();

                if (stop) {
                    log.debug("料理后事");
                    break;
                }

                try {
                    Thread.sleep(1000);
                    log.debug(("执行监控记录"));
                } catch (InterruptedException e) {
                    //sleep出现异常后，会清除打断标记，需要重新设置标记
                    currentThread.interrupt();
                }
            }
        }, "monitor");

        monitorThread.start();

    }

    public void stop() {
        stop = true;
        monitorThread.interrupt();
    }
}
```

程序中 `terminateThreadPattern.start();` 一共出现了三次，但是线程只执行了一次

1.3 有序性

1.3.1 概述

JVM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

```
static int i;
static int j;

//在某个线程内执行如下赋值操作
i = ...;
j = ...;
```

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，上面的代码真正执行时，既可以是

```
i = ...;
j = ...;
```

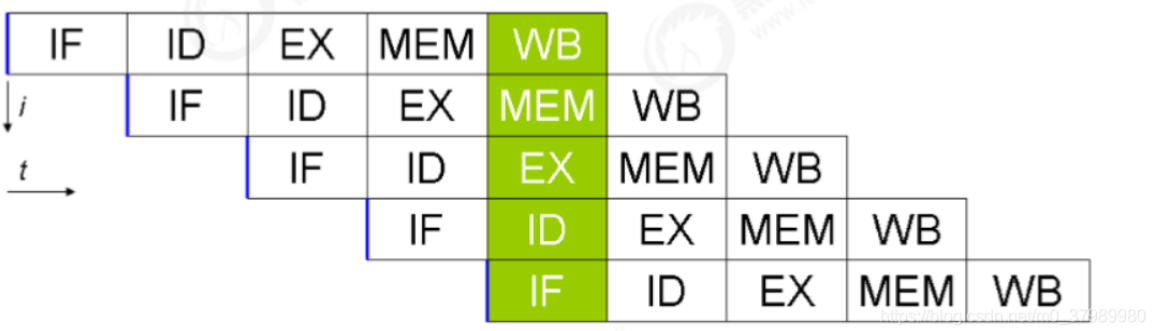
也可以是

```
j = ...;
i = ...;
```

这种特性称为**指令重排**，多线程下指令重排会影响正确性。为什么会有指令重排这种优化呢？从CPU执行指令的原理理解一下吧

1.3.2 指令重排序

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。



在多线程环境下，指令重排序可能导致出现意料之外的结果

1.3.3 诡异的结果

```
int num = 0;

// volatile 修饰的变量，可以禁用指令重排 volatile boolean ready =
false; 可以防止变量之前的代码被重排序
boolean ready = false;

// 线程1 执行此方法
public void actor1(I_Result r) {
     if(ready) {
         r.r1 = num + num;
     }
     else {
         r.r1 = 1;
     }
}
// 线程2 执行此方法
public void actor2(I_Result r) {
     num = 2;
     ready = true;
}
```

线程1执行actor1方法, 线程2执行actor2方法

- I_Result 是一个对象，有一个属性 r1 用来保存结果，问可能的结果有几种?
  - 情况1：线程1 先执行，这时 ready = false，所以进入 else 分支结果为 `1`
  - 情况2：线程2 先执行 num = 2，但没来得及执行 ready = true，线程1 执行，还是进入 else 分支，结果为 `1`
  - 情况3：线程2 执行到 ready = true，线程1 执行，这回进入 if 分支，结果为 `4` （因为 num 已经执行过了）
- 但是结果还有可能是 0 ，这种情况下是： **`线程2 执行 ready = true，切换到线程1，进入 if 分支，相加为 0，再切回线程2 执行 num = 2。`**

1.3.4 volatile原理

volatile的底层实现原理是内存屏障，Memory Barrier

- 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

写屏障保证在该屏障之前的（代码），对共享变量的改动，动同步到主存中

```
public void actor1(I_Result r) {
    num = 2;
    //ready 是volatile类型变量，带有写屏障
    ready = true;
    //写屏障
}
```

在 `ready=true` 改动之后，写屏障之前的代码会同步到主存当中（本例中是将num的改动和ready的改动同步到主存）

读屏障保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据

```
public void actor2(I_Result r) {
    //读屏障
    //ready 是volatile类型变量，带有读屏障（在对其操作之前）
    if (ready) {
        r.r1 = num + num;
    } else {
        r.r1 = 1;
    }
}
```

在进如if判断条件之前，就会将之前的对所有变量的修改同步到主存当中

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，也就下面这段代码

```
public void actor1(I_Result r) {
    num = 2;
    //ready 是volatile类型变量，带有写屏障
    ready = true;
    //写屏障
}
```

不会出现下面这种将写屏障之前的代码重排到写屏障之后

```
public void actor1(I_Result r) {
    //ready 是volatile类型变量，带有写屏障
    ready = true;
    //写屏障
    num = 2;
}
```

读屏障会确保指令重新排序时，不会将读屏障之后的代码排在读屏障之前

volatile无法解决的问题

volatile无法解决**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会被重排序

double-checked locking 问题

double-checked locking单例模式

```
public final class Singleton {
    private Singleton() { }
    private static Singleton INSTANCE = null;
    public static synchronized Singleton getInstance() {
        if (INSTANCE == null) { // t1
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

以上代码等同于下面代码

```
public final class Singleton {
    private Singleton() { }
    private static Singleton INSTANCE = null;
    public static Singleton getInstance() {
        synchronized(Singleton.class) {
            if (INSTANCE == null) { // t1
                INSTANCE = new Singleton();
            }
        }
        return INSTANCE;
    }
}
```

上述代码因为同步代码块内（也就是synchronized中的代码）代码太多，会影响性能，上面的例子每一次获取单例对象都需要等待锁进入同步代码块，性能太差，故上述代码可以修改为

```
public final class Singleton {
    private Singleton() { }
    private static Singleton INSTANCE = null;
    public static Singleton getInstance() {
        if (INSTANCE == null) {
            synchronized(Singleton.class) {
                if (INSTANCE == null) { // t1
                    INSTANCE = new Singleton();
                }
            }
        }

        return INSTANCE;
    }
}
```

注意: 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;
3: ifnonnull 37 // 判断是否为空
// ldc是获得类对象
6: ldc #3 // class cn/itcast/n5/Singleton
// 复制操作数栈栈顶的值放入栈顶，将类对象的引用地址复制了一份
```

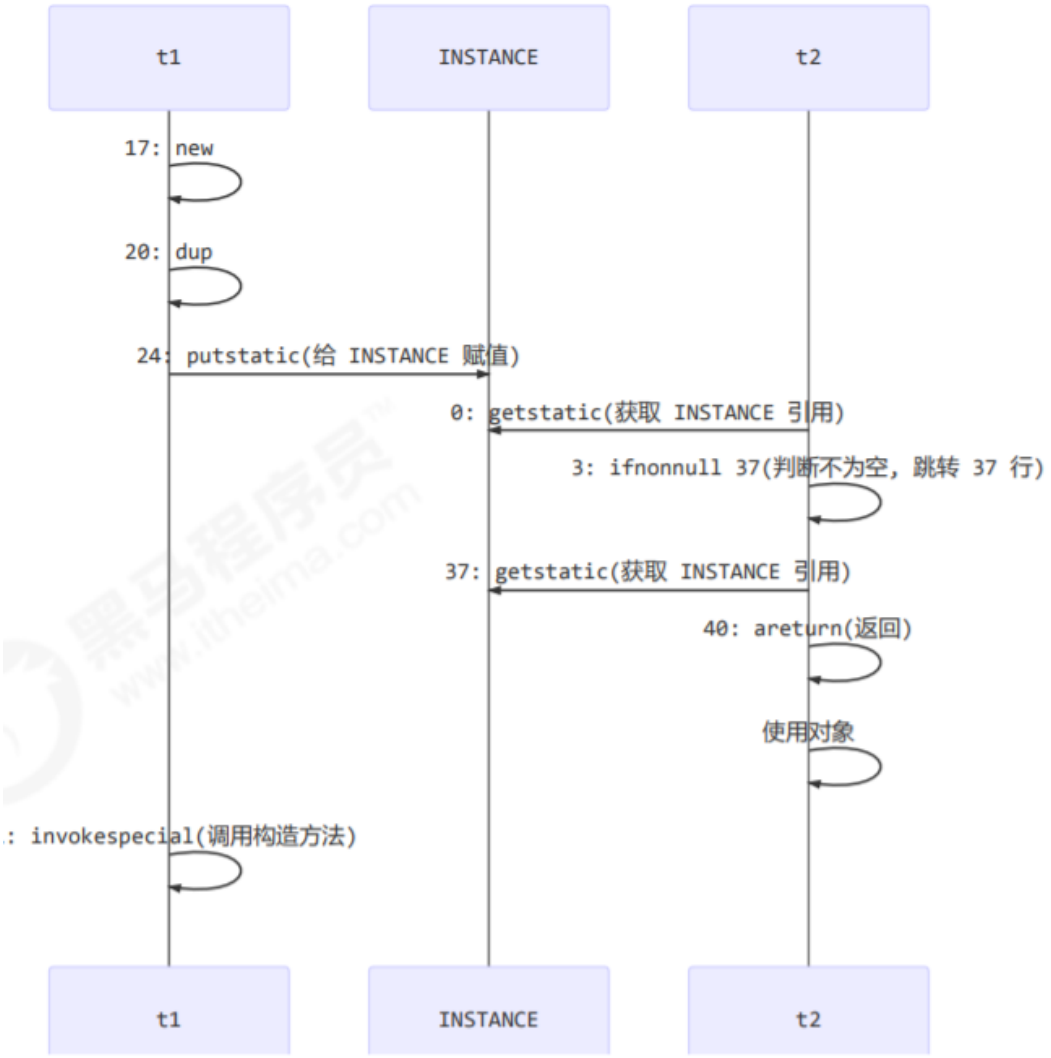
```
8: dup
// 操作数栈栈顶的值弹出，即将对象的引用地址存到局部变量表中
// 将类对象的引用地址存储了一份，是为了将来解锁用
9: astore_0
10: monitorenter
11: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;
14: ifnonnull 27
// 新建一个实例
17: new #3 // class cn/itcast/n5/Singleton
// 复制了一个实例的引用
20: dup
// 通过这个复制的引用调用它的构造方法
21: invokespecial #4 // Method "<init>":()V
// 最开始的这个引用用来进行赋值操作
24: putstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;
27: aload_0
28: monitorexit
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
12345678910111213141516171819202122324252627282930
```

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 // 复制了引用地址, 解锁使用
- 21 表示利用一个对象引用，调用构造方法 // 根据复制的引用地址调用构造方法
- 24 表示利用一个对象引用，赋值给 static INSTANCE

可能jvm 会优化为： `先执行 24(赋值)，再执行 21(构造方法)` 。如果两个线程 t1，t2 按如下时间序列执行：

- 通过上面的字节码发现, 这一步 `INSTANCE = new Singleton();` 操作不是一个 `原子操作` , 它分为 `21，24两个指令` , 此时可能就会发生 `指令重排` 的问题



- 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**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。
- 注意在 JDK 5 以上的版本的 volatile 才会真正有效

double-checked locking问题解决方法

在Singleton对象上加上volatile关键字

```
public final class Singleton {
    private Singleton() { }
    private static volatile Singleton INSTANCE = null;
    public static Singleton getInstance() {
        if (INSTANCE == null) {
            synchronized(Singleton.class) {
                if (INSTANCE == null) { // t1
```

```
                    INSTANCE = new Singleton();
                }
            }
        }

        return INSTANCE;
    }
}
```

字节码文件

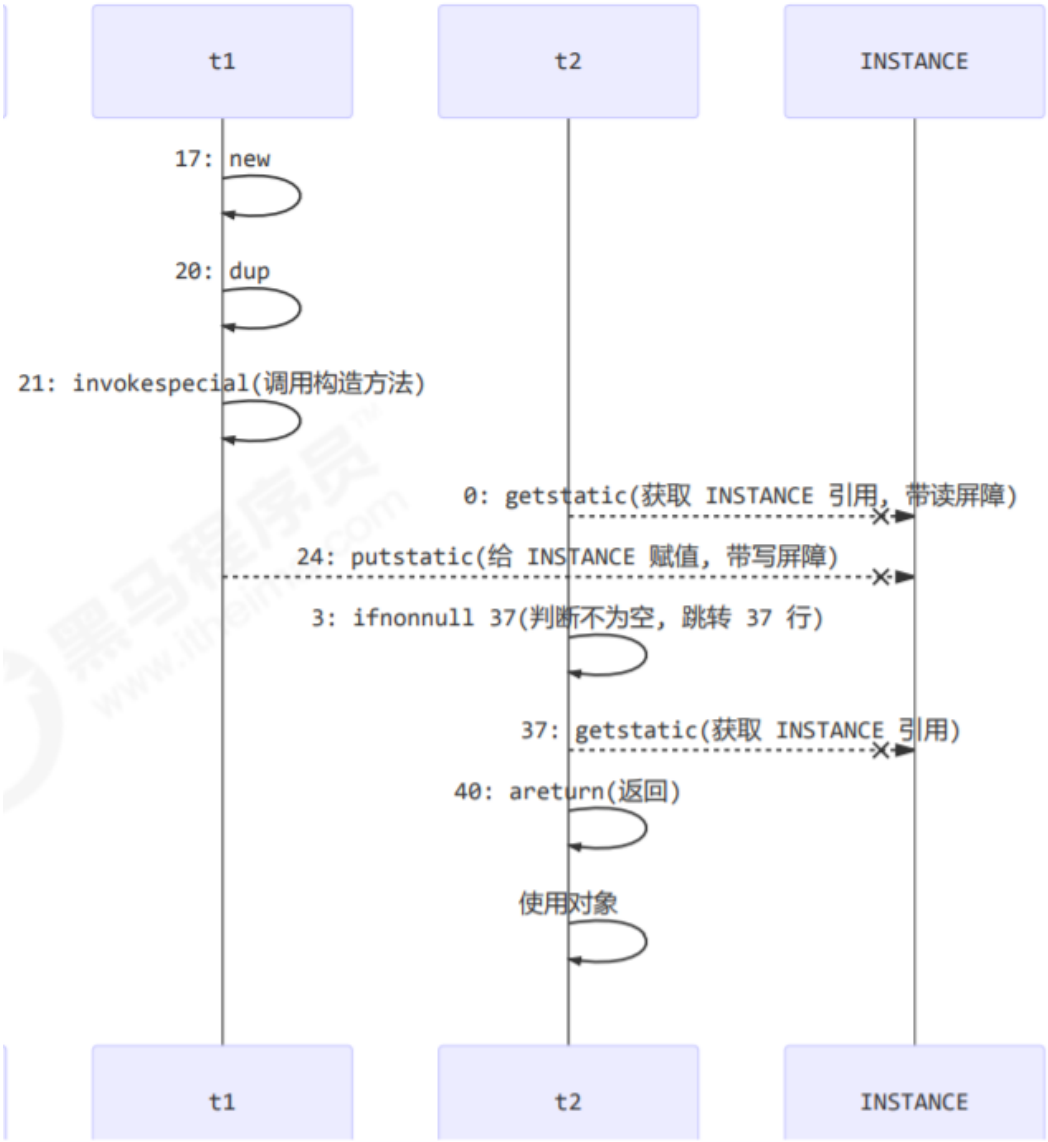
```
// ------------------------------------> 加入对 INSTANCE 变量的读屏障
0: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;
3: ifnonnull 37
6: ldc #3 // class cn/itcast/n5/Singleton
8: dup
9: astore_0
10: monitorenter ----------------------> 保证原子性、可见性
11: getstatic #2 // Field INSTANCE:Lcn/itcast/n5/Singleton;
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
```

读写 volatile 变量操作（即getstatic操作和putstatic操作）时会加入内存屏障（Memory Barrier（Memory Fence）），保证下面两点：

- 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
- 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**

加上 `volatile` 之后, 保证了指令的有序性, 不会发生指令重排, 21就不会跑到24之后执行了



小结 ：

- `synchronized` 既能保证**原子性、可见性、有序性**， 其中有序性是在该共享变量完全被`synchronized` 所接管（包括共享变量的读写操作） ，上面的例子中synchronized 外面的 if (INSTANCE == null) 中的INSTANCE读操作没有被 synchronized 接管，因此无法保证INSTANCE共享变量的有序性（即不能防止指令重排）。

- 对共享变量加 `volatile` 关键字可以保证 可见性 和 有序性 ，但是**不能保证原子性**（即不能防止指令交错）。

1.3.5 happens-before

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

方式一：

- 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
  - synchronized锁, 保证了 可见性

```
static int x;

static Object m = new Object();

new Thread(()->{
    synchronized(m) {
        x = 10;
    }
},"t1").start();

new Thread(()->{
    synchronized(m) {
        System.out.println(x);
    }
},"t2").start();

// 10
```

方式二：

- 线程对

  volatile 变量的写，对接下来其它线程对该变量的读可见
  - **volatile修饰的变量, 通过 写屏障 , 共享到主存中, 其他线程通过 读屏障 , 读取主存的数据**

```
volatile static int x;

new Thread(()->{
    x = 10;
},"t1").start();

new Thread(()->{
    System.out.println(x);
},"t2").start();
```

方式三:

- 线程 start() 前对变量的写，对该线程开始后对该变量的读可见
    - 线程还没启动时, 修改变量的值, 在启动线程后, 获取的变量值, 肯定是修改过的

```
static int x;
x = 10;

new Thread(()->{
    System.out.println(x);
},"t2").start();
```

方式四 :

- 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）
    - 主线程获取的x值, 是线程执行完对x的写操作之后的值。

```
static int x;

Thread t1 = new Thread(()->{
    x = 10;
},"t1");
t1.start();

t1.join();
System.out.println(x);
```

方式五 :

- 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）

```
static int x;
public static void main(String[] args) {
    Thread t2 = new Thread(()->{
        while(true) {
            if(Thread.currentThread().isInterrupted()) {
                System.out.println(x); // 10，打断了，读取的也是打断
前修改的值
                break;
            }
        }
    },"t2");
    t2.start();

    new Thread(()->{
        sleep(1);
        x = 10;
        t2.interrupt();
    },"t1").start();

    while(!t2.isInterrupted()) {
        Thread.yield();
    }
    System.out.println(x);  // 10
}
```

方式五：

- 对 变量默认值（0，false，null）的写 ，对其它线程对该变量的 读可见 (最基本)
- 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**
- 因为 x加了volatile , 所以在volatile static int x 代码的上面添加了 读屏障 , 保证读到的x和y的变化是可见的(包括y, 只要是读屏障下面都OK); 通过传递性, t2

线程对x,y的写操作, 都是可见的

```
volatile static int x;
static int y;

new Thread(()->{
    y = 10;
    x = 20;
},"t1").start();

new Thread(()->{
    // x=20 对 t2 可见, 同时 y=10 也对 t2 可见
    System.out.println(x);
},"t2").start();
```