参考：<http://blog.csdn.net/ns_code/article/details/17348313>

**【Java并发编程】之十六：深入Java内存模型——happen-before规则及其对DCL的分析（含代码）**

转载请注明出处：<http://blog.csdn.net/ns_code/article/details/17348313>

happen—before规则介绍

    Java语言中有一个“先行发生”（happen—before）的规则，它是Java内存模型中定义的两项操作之间的偏序关系，如果操作A先行发生于操作B，其意思就是说，在发生操作B之前，操作A产生的影响都能被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等，它与时间上的先后发生基本没有太大关系。这个原则特别重要，它是判断数据是否存在竞争、线程是否安全的主要依据。

    举例来说，假设存在如下三个线程，分别执行对应的操作:

---------------------------------------------------------------------------

线程A中执行如下操作：i=1

线程B中执行如下操作：j=i

线程C中执行如下操作：i=2

---------------------------------------------------------------------------

    假设线程A中的操作”i=1“ happen—before线程B中的操作“j=i”，那么就可以保证在线程B的操作执行后，变量j的值一定为1，即线程B观察到了线程A中操作“i=1”所产生的影响；现在，我们依然保持线程A和线程B之间的happen—before关系，同时线程C出现在了线程A和线程B的操作之间，但是C与B并没有happen—before关系，那么j的值就不确定了，线程C对变量i的影响可能会被线程B观察到，也可能不会，这时线程B就存在读取到不是最新数据的风险，不具备线程安全性。

    下面是Java内存模型中的八条可保证happen—before的规则，它们无需任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随机地重排序。

    1、程序次序规则：在一个单独的线程中，按照程序代码的执行流顺序，（时间上）先执行的操作happen—before（时间上）后执行的操作。

    2、管理锁定规则：一个unlock操作happen—before后面（时间上的先后顺序，下同）对同一个锁的lock操作。

    3、volatile变量规则：对一个volatile变量的写操作happen—before后面对该变量的读操作。

    4、线程启动规则：Thread对象的start（）方法happen—before此线程的每一个动作。

    5、线程终止规则：线程的所有操作都happen—before对此线程的终止检测，可以通过Thread.join（）方法结束、Thread.isAlive（）的返回值等手段检测到线程已经终止执行。

    6、线程中断规则：对线程interrupt（）方法的调用happen—before发生于被中断线程的代码检测到中断时事件的发生。

    7、对象终结规则：一个对象的初始化完成（构造函数执行结束）happen—before它的finalize（）方法的开始。

    8、传递性：如果操作A happen—before操作B，操作B happen—before操作C，那么可以得出A happen—before操作C。

时间上先后顺序和happen—before原则

    ”时间上执行的先后顺序“与”happen—before“之间有何不同呢？

    1、**首先来看操作A在时间上先与操作B发生，是否意味着操作A happen—before操作B？**

    一个常用来分析的例子如下：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/17348313) [copy](http://blog.csdn.net/ns_code/article/details/17348313)

1. **private** **int** value = 0;
3. **public** **int** get(){
4. **return** value;
5. }
6. **public** **void** set(**int** value){
7. **this**.value = value;
8. }

    假设存在线程A和线程B，线程A先（时间上的先）调用了setValue（3）操作，然后（时间上的后）线程B调用了同一对象的getValue（）方法，那么线程B得到的返回值一定是3吗？

    对照以上八条happen—before规则，发现没有一条规则适合于这里的value变量，从而我们可以判定线程A中的setValue（3）操作与线程B中的getValue（）操作不存在happen—before关系。因此，尽管线程A的setValue（3）在操作时间上先于操作B的getvalue（），但无法保证线程B的getValue（）操作一定观察到了线程A的setValue（3）操作所产生的结果，也即是getValue（）的返回值不一定为3（有可能是之前setValue所设置的值）。这里的操作不是线程安全的。

    因此，”一个操作时间上先发生于另一个操作“并不代表”一个操作happen—before另一个操作“。

    解决方法：可以将setValue（int）方法和getValue（）方法均定义为synchronized方法，也可以把value定义为volatile变量（value的修改并不依赖value的原值，符合volatile的使用场景），分别对应happen—before规则的第2和第3条。注意，只将setValue（int）方法和getvalue（）方法中的一个定义为synchronized方法是不行的，必须对同一个变量的所有读写同步，才能保证不读取到陈旧的数据，仅仅同步读或写是不够的 。

    2、**其次来看，**操作A happen—before操作B，是否意味着操作A在时间上先与操作B发生？

    看有如下代码：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/17348313) [copy](http://blog.csdn.net/ns_code/article/details/17348313)

1. x = 1；
2. y = 2;

    假设同一个线程执行上面两个操作：操作A：x=1和操作B：y=2。根据happen—before规则的第1条，操作A happen—before 操作B，但是由于编译器的指令重排序（Java语言规范规定了JVM线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。这个过程通过叫做指令的重排序。指令重排序存在的意义在于：JVM能够根据处理器的特性（CPU的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能。在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整）等原因，操作A在时间上有可能后于操作B被处理器执行，但这并不影响happen—before原则的正确性。

    因此，”一个操作happen—before另一个操作“并不代表”一个操作时间上先发生于另一个操作“。

    最后，一个操作和另一个操作必定存在某个顺序，要么一个操作或者是先于或者是后于另一个操作，或者与两个操作同时发生。同时发生是完全可能存在的，特别是在多CPU的情况下。而两个操作之间却可能没有happen-before关系，也就是说有可能发生这样的情况，操作A不happen-before操作B，操作B也不happen-before操作A，用数学上的术语happen-before关系是个偏序关系。两个存在happen-before关系的操作不可能同时发生，一个操作A happen-before操作B，它们必定在时间上是完全错开的，这实际上也是同步的语义之一（独占访问）。

利用happen—before规则分析DCL

   DCL即双重检查加锁，关于单例模式的DCL机制，可以参看：[**http://blog.csdn.net/ns\_code/article/details/17359719**](http://blog.csdn.net/ns_code/article/details/17359719)一文，这里不再详细介绍。下面是一个典型的在单例模式中使用DCL的例子：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/17348313) [copy](http://blog.csdn.net/ns_code/article/details/17348313)

1. **public** **class** LazySingleton {
2. **private** **int** someField;
4. **private** **static** LazySingleton instance;
6. **private** LazySingleton() {
7. **this**.someField = **new** Random().nextInt(200)+1;         // (1)
8. }
10. **public** **static** LazySingleton getInstance() {
11. **if** (instance == **null**) {                               // (2)
12. **synchronized**(LazySingleton.**class**) {               // (3)
13. **if** (instance == **null**) {                       // (4)
14. instance = **new** LazySingleton();           // (5)
15. }
16. }
17. }
18. **return** instance;                                      // (6)
19. }
21. **public** **int** getSomeField() {
22. **return** **this**.someField;                                // (7)
23. }
24. }

    这里得到单一的instance实例是没有问题的，问题的关键在于尽管得到了Singleton的正确引用，但是却有可能访问到其成员变量的不正确值。具体来说Singleton.getInstance().getSomeField()有可能返回someField的默认值0。如果程序行为正确的话，这应当是不可能发生的事，因为在构造函数里设置的someField的值不可能为0。为也说明这种情况理论上有可能发生，我们只需要说明语句(1)和语句(7)并不存在happen-before关系。

   假设线程Ⅰ是初次调用getInstance()方法，紧接着线程Ⅱ也调用了getInstance()方法和getSomeField()方法，我们要说明的是线程Ⅰ的语句(1)并不happen-before线程Ⅱ的语句(7)。线程Ⅱ在执行getInstance()方法的语句(2)时，由于对instance的访问并没有处于同步块中，因此线程Ⅱ可能观察到也可能观察不到线程Ⅰ在语句(5)时对instance的写入，也就是说instance的值可能为空也可能为非空。我们先假设instance的值非空，也就观察到了线程Ⅰ对instance的写入，这时线程Ⅱ就会执行语句(6)直接返回这个instance的值，然后对这个instance调用getSomeField()方法，该方法也是在没有任何同步情况被调用，因此整个线程Ⅱ的操作都是在没有同步的情况下调用 ，这时我们便无法利用上述8条happen-before规则得到线程Ⅰ的操作和线程Ⅱ的操作之间的任何有效的happen-before关系（主要考虑规则的第2条，但由于线程Ⅱ没有在进入synchronized块，因此不存在lock与unlock锁的问题），这说明线程Ⅰ的语句(1)和线程Ⅱ的语句(7)之间并不存在happen-before关系，这就意味着线程Ⅱ在执行语句(7)完全有**可能**观测不到线程Ⅰ在语句(1)处对someFiled写入的值，这就是DCL的问题所在。很荒谬，是吧？DCL原本是为了逃避同步，它达到了这个目的，也正是因为如此，它最终受到惩罚，这样的程序存在严重的bug，虽然这种bug被发现的概率绝对比中彩票的概率还要低得多，而且是转瞬即逝，更可怕的是，即使发生了你也不会想到是DCL所引起的。

    前面我们说了，线程Ⅱ在执行语句(2)时也有可能观察空值，如果是种情况，那么它需要进入同步块，并执行语句(4)。在语句(4)处线程Ⅱ还能够读到instance的空值吗？不可能。这里因为这时对instance的写和读都是发生在同一个锁确定的同步块中，这时读到的数据是最新的数据。为也加深印象，我再用happen-before规则分析一遍。线程Ⅱ在语句(3)处会执行一个lock操作，而线程Ⅰ在语句(5)后会执行一个unlock操作，这两个操作都是针对同一个锁--Singleton.class，因此根据第2条happen-before规则，线程Ⅰ的unlock操作happen-before线程Ⅱ的lock操作，再利用单线程规则，线程Ⅰ的语句(5) -> 线程Ⅰ的unlock操作，线程Ⅱ的lock操作 -> 线程Ⅱ的语句(4)，再根据传递规则，就有线程Ⅰ的语句(5) -> 线程Ⅱ的语句(4)，也就是说线程Ⅱ在执行语句(4)时能够观测到线程Ⅰ在语句(5)时对Singleton的写入值。接着对返回的instance调用getSomeField()方法时，我们也能得到线程Ⅰ的语句(1) -> 线程Ⅱ的语句(7)（由于线程Ⅱ有进入synchronized块，根据规则2可得），这表明这时getSomeField能够得到正确的值。但是仅仅是这种情况的正确性并不妨碍DCL的不正确性，一个程序的正确性必须在所有的情况下的行为都是正确的，而不能有时正确，有时不正确。

    对DCL的分析也告诉我们一条经验原则：对引用（包括对象引用和数组引用）的非同步访问，即使得到该引用的最新值，却并不能保证也能得到其成员变量（对数组而言就是每个数组元素）的最新值。

**解决方案：**

    1、最简单而且安全的解决方法是使用static内部类的思想，它利用的思想是：一个类直到被使用时才被初始化，而类初始化的过程是非并行的，这些都有JLS保证。

如下述代码：

**[java]** [view plain](http://blog.csdn.net/ns_code/article/details/17348313) [copy](http://blog.csdn.net/ns_code/article/details/17348313)

1. **public** **class** Singleton {
3. **private** Singleton() {}
5. // Lazy initialization holder class idiom for static fields
6. **private** **static** **class** InstanceHolder {
7. **private** **static** **final** Singleton instance = **new** Singleton();
8. }
10. **public** **static** Singleton getSingleton() {
11. **return** InstanceHolder.instance;
12. }
13. }

    2、另外，可以将instance声明为volatile，即

**private** **volatile** **static** LazySingleton instance; 

    这样我们便可以得到，线程Ⅰ的语句(5) -> 语线程Ⅱ的句(2)，根据单线程规则，线程Ⅰ的语句(1) -> 线程Ⅰ的语句(5)和语线程Ⅱ的句(2) -> 语线程Ⅱ的句(7)，再根据传递规则就有线程Ⅰ的语句(1) -> 语线程Ⅱ的句(7)，这表示线程Ⅱ能够观察到线程Ⅰ在语句(1)时对someFiled的写入值，程序能够得到正确的行为。

**注：**

**1、volatile屏蔽指令重排序的语义在JDK1.5中才被完全修复，此前的JDK中及时将变量声明为volatile，也仍然不能完全避免重排序所导致的问题（主要是volatile变量前后的代码仍然存在重排序问题），这点也是在JDK1.5之前的Java中无法安全使用DCL来实现单例模式的原因。**

    2、把volatile写和volatile读这两个操作综合起来看，在读线程B读一个volatile变量后，写线程A在写这个volatile变量之前，所有可见的共享变量的值都将立即变得对读线程B可见。

   3、 在java5之前对final字段的同步语义和其它变量没有什么区别，在java5中，final变量一旦在构造函数中设置完成（前提是在构造函数中没有泄露this引用)，其它线程必定会看到在构造函数中设置的值。而DCL的问题正好在于看到对象的成员变量的默认值，因此我们可以将LazySingleton的someField变量设置成final，这样在java5中就能够正确运行了。

参考资料：<http://www.iteye.com/topic/260515/>

                 《深入理解Java虚拟机——JVM高级特性与最佳实践》第12章