[翻译]JSR 133 (Java Memory Model) FAQ

2012-12-11 11:12:49 ticmy

JSR 133 (Java Memory Model) FAQ

Jeremy Manson and Brian Goetz, February 2004

内容列表

究竟什么是内存模型？

在多处理器系统中，处理器通常都有一到多级存储缓存，通过加速数据访问（因为数据距处理器更近）以及减少存储总线的流量（因为本地缓存可以满足某些存储操作）以提升性能。存储缓存能极大地提升性能，但也面临着大量新的难题。例如，当两个处理器同时检查相同的内存地址时会怎么样？在什么条件下它们将看到相同的值？

在处理器层面，针对其它处理器写入内存的值何时对当前处理器可见，以及当前处理器写入的值何时对其它处理器可见，内存模型为它们定义了充要条件。有些处理器表现出了强内存模型，对于任一给定的内存地址所有处理器看到的总是相同的值。另一些处理器则表现出了弱内存模型，它们需要特殊的指令，谓之内存屏障，来刷新本地处理器缓存或者使本地处理器缓存失效，以使当前处理器能看见其它处理器写的值或让其它处理器看见当前处理器写的值。这些内存屏障通常在执行lock或unlock时起作用；在高级编程语言中它们对

程序员不可见。

在强内存模型下编写程序常常比较容易，因为不需要内存屏障。然而即使在一些拥有最强内存模型的平台上，常常也还需要内存屏障，虽然这往往有违直觉。近来CPU设计倾向于弱内存模型，因为这种模型宽松的缓存一致性带来了跨多处理器时更大的可伸缩性以及更大量的内存。

编译器对代码的重排序使写操作何时对其它线程可见这个问题更加麻烦。比如，只要代码移动不改变程序的语义，当编译器觉得将程序中的一个写操作移到后面去更高效时，它就可以这么做。如果编译器延缓了某个操作，在这个操作被执行之前其它线程都不会看到操作的结果。这反映出了缓存的影响。

此外，对内存的写可以移到程序中当前位置的前面；这样的话，其它线程也许能看到程序“尚未执行”（实际已经执行了，但从代码的角度看还没到那一句代码）的写操作的值。所有这些弹性都是有意的——通过给编译器、运行时或硬件以灵活性，在内存模型允许下，让操作以最佳顺序执行，来达到更高的性能。

下面是个简单的例子：

```
Class Reordering {
 int x = 0, y = 0;
 public void writer() {
  x = 1;
  y = 2;
 }
 public void reader() {
  int r1 = y;
  int r2 = x;
 }
}
```

假设这段代码在两个线程中并发执行，且读取y得到的值是2。因为写y的操作在写x的后面，程序员可能以为x读到的值一定是1.但是，写操作可能被重排序了。如果发生了重排序，可能先执行了写y的操作，然后是读取x和y，最后是写入x。结果将是r1的值为2，但r2的值是0.

Java内存模型描述了在多线程代码中何种行为是合法的，以及线程通过内存是如何交互的。它描述了程序中的变量与实际计算机系统中的变量在内存或寄存器中存取的底层细节关系。Java内存模型要能在各种各样的硬件上正确实现，且各种编译器优化也能正确实现。

Java有几个关键字，包括volatile，final以及synchronized，来帮助程序员告诉编译器程序的并发需求。Java内存模型定义了volatile和synchronized的行为，更重要地是，该模型会确保一个正确同步的Java程序能在所有的处理器架构上正确运行。

其它语言，像C++，有内存模型吗？

大部分其它编程语言，像C和C++，并没有设计成直接支持多线程。这些语言防止编译器和体系结构中重排序的发生严重依赖于使用的线程库（如pthreads）提供的保障和代码运行的平台。

JSR133是什么？

1997年以来，Java语言规范第17章定义的Java内存模型爆出了一些严重的缺陷。这些缺陷能使行为混乱（如看到final字段的值后该final的值还能改变），还会削弱常见的编译器优化能力。

Java内存模型曾经目标远大；这是首次编程语言规范尝试整合出一种内存模型，以使各种体系结构都能提供一致的并发语义。不幸的是，定义一种一致且直观的内存模型远比期望的要难。JSR133为Java语言定义了一种新的内存模型，它修复了早期内存模型的缺陷。为达到这个目的，final和volatile的语义需要改变。

完整的语义见http://www.cs.umd.edu/users/pugh/java/memoryModel，但正式的语义不适合胆小的人（译者注：应该指的是正式的语义晦涩难懂）。你会很惊讶、毫不夸张地发现像同步这样看似简单的概念实际上是何等的复杂。幸运的是，你无需懂得正式语义的细节——JSR133的目标是创建一个正式语义集，为volatile，synchronized以及final工作机制提供一种直观的架构。

JSR133的目标包括：

- 保持现有的安全保证，像类型安全，并加强其它方面。例如，变量的值不能凭空创建：线程观察到变量的每个值必须是被某一线程合理赋值的。
- 正确同步的程序的语义应尽可能简单、直观。
- 应该定义未完全或未正确同步的程序的语义，以最小化潜在的安全风险。
- 程序员应当能够自信的推断出多线程程序是如何与内存交互的。
- 应该能在大范围流行的硬件架构上设计正确、高性能的JVM实现。
- 应当提供新的初始化安全保证。如果一个对象被正确地构造（意思是该对象的引用在构造期间没有逸出），所有能看到该对象引用的线程将会看到其在构造器（constructor，译者注：这里或许换做<*init*>或*对象初始化方法*更好，后文提到构造器都是这个意思）中设置的final字段的值，而无需同步。
- 对现有代码影响最小。

重排序意味着什么？

在有些场景下访问程序变量（对象实例字段，类静态字段以及数组元素）会表现出与程序指定的顺序不一样。编译器可以以优化的名义来改变指令的顺序。处理器在特定情况下会不按顺序执行指令。数据可能以不

同于程序指定的顺序在寄存器、处理器缓存以及主存之间移动。

比如，一个线程写入字段a，然后写入字段b，且b的值不依赖于a，编译器就可以自由地对这两个操作进行重排序，且缓存可以在a之前将b刷回主存。潜在的重排序来源有多种，如编译器、运行时以及缓存。

编译器、运行时和硬件应协力创造出“似乎是串行（as-if-serial）”的语义的假象，意思是在单线程程序中，程序不应该观察到重排序的影响。然而，在未正确同步的多线程程序中重排序的影响就显现出来了，一个线程也许能看到其它线程的结果，也许会察觉到变量对其它线程可见的顺序与程序中执行或指定的顺序不一致。

大部分时候，线程不用知道其它线程正在做什么。但是当它想知道的时候，就是需要使用同步的时候。

老的内存模型有什么问题？

老的内存模型有几个严重的问题。由于它难于理解，很多地方都违背了它。例如，在有些场景下，老的内存模型不允许重排序的发生。实现老模型时的混乱促成了JSR133的形成。

人们广泛认为，如果使用了final字段，就不需要线程间的同步来确保其它线程会看到该字段正确的值。虽然这是个合理的假设且是个明显的行为，也确实是我们想要的行为，但在老的内存模型下，却不是这样。老的内存模型对待final字段与其它字段的方式无异——也就是说同步是确保所有线程看到final字段值确是构造器中写入的值的唯一方式。结果就是一个线程可能会看到final字段的默认值，一段时间后看到构造器中设置的值。这意味着，诸如String这样的不可变对象的值可能会变化——着实令人担忧。

老的内存模型允许volatile写与非volatile读、写间重排序，这跟开发者对volatile的直观认识不一致，因此会引起混淆。

正如我们所看到的，对于未正确同步的程序会发生什么，程序员常常有着错误的直觉。JSR133的目的之一就是要引起这方面的注意。

未正确同步是什么意思？

未正确同步的代码对不同的人意味不同。在Java内存模型上下文中提到未正确同步的代码时，意思是这样的代码：

1、有一个线程写一个变量，

2、有另一个线程读取同一个变量，且

3、写操作和读操作间没有被同步排序。

在违背了这些规则时，就说那个变量存在数据争用。存在数据争用的程序就是未正确同步的程序。

同步做了什么？

同步包含几个方面。最容易理解的是互斥——同一时刻只能有一个线程持有监视器，所以，在某个监视器上同步意味着一旦一个线程进入了被该监视器保护的同步块，其它线程就无法进入由该监视器保护的任意块，直到持有监视器的线程退出同步块。

但是同步除了互斥还有其它含义。同步确保一个线程在同步块之内或之前的内存写以可预见的方式对在同一个监视器上同步的其它线程可见。在退出同步块后，释放了监视器，这有刷新缓存到主存的作用，所以该线程的写操作能对其它线程可见。在进入一个同步块之前，需要先获取监视器，这有使本地处理器缓存失效的作用，所以变量会从主存重新装载。前个线程释放监视器使得之前所写的全部内容可见，此时后续线程就可以看到这些内容。

讨论这些一直使用了缓存这个词，听起来像是这些问题只会影响多处理器机器。但重排序的影响很容易在单处理器机器上看到。处理器不可能将代码移到获取动作之前或释放动作之后。当说到缓存上的获取（acquire）或释放动作（release），代表该动作带来的一系列可能的效果。

新的内存模型语义在内存操作（读字段，写字段，lock，unlock）和其它线程操作（start和join）上创建了一个偏序，就说某些action happen before其它操作。当一个action happen before另外一个，第一个就保证排在第二个的前面且对第二个可见。这种偏序的规则如下：

- 某个线程中的每个action happens before该线程在程序顺序上的后续action。
- unlock某个监视器 happens before 每个后续在相同监视器上的lock操作。
- 写入volatile字段 happens before 每个后续读取相同字段。
- 某个线程对象上的start()方法happens before该已启动线程中的任何action。
- 某个线程中的所有action happen before 任何其他线程成功从该线程的join()调用中返回。

这意味着任一退出同步块之前对某线程可见的内存操作，都对所有其它进入由同一个监视器保护的同步块的线程可见，因为所有的内存操作happen before释放动作（release），且释放动作happen before获取动作（acquire）。

下面的模式被一些人用来强制执行一个内存屏障，但不会起作用：

```
synchronized (new Object()) {}
```

这是一个空操作，编译器可以将其完全移除，因为编译器知道不会有其它线程会在同一个监视器上同步。需要建立一个happens-before关系让一个线程看到其它线程的结果。

重要提示：注意两个线程要在同一个监视器上同步以正确建立happens-before关系。并不是所有对在对象X上同步的线程A可见的内容都对在对象Y上同步后的线程B可见。释放和获取动作需要“匹配”（即，在同一个监视器上操作）才能实现正确的语义。否则，代码存在数据争用。

final字段的值是如何看起来会变的？

final字段的值看起来会变的最好例子之一涉及到String类的一个特殊实现。

String可以实现成包含三个字段——一个字符数组，位于数组中的偏移量，和长度。以这种方式实现String而不是只要一个字符数组是因为这样可以让多个String和StringBuffer对象共享同一个字符数组，避免了额外的对象分配与拷贝。所以，如方法String.substring()可以实现成创建一个共用原String字符数组的String对象，仅是长度和偏移量字段的值不一样。String中这些字段都是final的。

String s1 = “/usr/tmp”;
String s2 = s1.substring(4);

s2字符串的偏移量将是4，长度为4.但是，在老的模型下，其它线程有可能先看到它的偏移量是默认值0，随后再看到正确的值4，看起来像是”/usr”变成了”/tmp”。

最初的Java内存模型允许这种行为；有几个JVM已经表现出了这种行为。新的Java内存模型则不允许这样。

在新的JMM下final字段是怎么工作的？

一个对象的final字段的值是在其构造器里设置的。假设对象是正确构造的，一旦一个对象被构建，构造器中赋给final字段的值无需同步就将对其它线程可见。此外，这些final字段引用的对象或数组的可见值起码是final字段中最新的。

对象的正确构造是什么意思？很简单，在构造期间这个正在构建的对象的引用没有“逸出(secape)”（见Safe Construction Techniques）。换而言之，不要将正在构建的对象的引用放到任何其它线程能看到的地方；不要将其赋值给静态字段，不要将其注册为其它对象的监听器等等。这些工作应该在构造器完成之后进行，而不是在构造器中。

```
class FinalFieldExample {
 final int x;
 int y;
 static FinalFieldExample f;
 public FinalFieldExample() {
  x = 3;
  y = 4;
 }
 static void writer() {
  f = new FinalFieldExample();
 }
```

```
 static void reader() {
  if (f != null) {
   int i = f.x;
   int j = f.y;
  }
 }
}
```

上面的类展示了final字段该如何使用。执行reader的线程保证能看到f.x的值为3，因为它是final的。不能保证看到y的值是4，因为它不是final的。如果FinalFieldExample的构造器是像这样的：

```
public FinalFieldExample() { // bad!
 x = 3;
 y = 4;
 // bad construction - allowing this to escape
 global.obj = this;
}
```

那么从global.obj读取到this的引用的线程将不保证看到x的值是3.

能看到字段正确构建的值当然好，但是如果字段本身是一个引用，你仍然需要让你的代码也能看到所指对象（或数组）的最新值。如果是final字段，这也有保证。所以，可以使用final引用指向一个数组，就不需要担心其它线程看到数组引用的值是正确的，而数组内容的值不对。再说一次，这里的“正确”意思是“构造器结束时的最新值”，而不是“最后改变的值”。

说了这么多，如果在某个线程构建了一个不可变对象（即，只包含final字段的对象）之后，想要确保对所有线程正确可见，仍需要同步。没有其它办法可以保证指向不可变对象的引用会对第二个线程可见。要深入、细心地理解代码中并发管理方式，小心使用程序从final字段得到的保障。

如果用JNI去改变final字段，其行为是未定义的。

volatile做了什么？

volatile字段是一种特殊的字段，用来在线程间传递状态。每个对volatile字段的读都将看到任意线程最后写入此volatile的值。实际上，它们是由程序员指定的绝不能看到缓存或重排序导致的过期值的字段。禁止编译器和运行时在寄存器中为它们分配空间。还需要确保一旦值被写入volatile字段，就要立即将值从缓存中刷新到主存，所以可以立即对其它线程可见。同样地，读取一个volatile字段之前，缓存必须失效，以使看到的内容是主存而不是本地处理器缓存中的。重排序访问volatile变量上也有更多的限制。

在新的内存模型下，volatile变量间仍然不能互相重排序。不同的是，它们现在不能随意与周围的普通字段重排序了。写入一个volatile字段有着与释放监视器相同的内存效应，读取一个volatile字段和获取监视器的内存效应相同。实际上，由于新的内存模型在volatile字段访问与其它volatile或非volatile字段访问的重排序上加了更严格的限制，当在线程A写入volatile字段f时所有对线程A可见的东西都在线程B读取f后对线程B可见。

这里有个volatile字段用法的简单例子：

```
class VolatileExample {
 int x = 0;
 volatile boolean v = false;
 public void writer() {
  x = 42;
  v = true;
 }
 public void reader() {
  if (v == true) {
   //uses x - guaranteed to see 42.
  }
 }
}
```

假设一个线程调用writer，另一个调用reader。writer中写入v使得写入到x的值释放（release）到主存，读取使得从内存中获取（acquire）x的值。这样，如果reader看到v的值是true，就能保证看到在其之前写入的42。在老的内存模型下则不是这样。如果v不是volatile的，编译器可能对writer中写操作重排序，然后reader中读取x看到的也许是0.

事实上，volatile的语义已经大大的加强了，几乎达到了同步的级别。为了可见性，每个读或写volatile字段就像半个同步。

重要提示：这点很重要，为了正确地建立happens-before关系，两个线程访问的要是同一个volatile变量。并不是当线程A写入volatile字段f时对线程A可见的内容都在线程B读取volatile字段g后对线程B可见。释放和获取动作需要“匹配”（即，在同一个volatile字段上操作）才能实现正确的语义。

新的内存模型修复了“双重锁定检查”问题吗？

臭名昭著的双重锁定检查模式是设计来支持延迟初始化同时又避免同步开销的一个技巧。在早期的JVM中，同步是很慢的，开发者很想移除它 —— 也许太急了。双重锁定检查像这样：

```
// double-checked-locking - don't do this!
private static Something instance = null;
public Something getInstance() {
 if (instance == null) {
  synchronized (this) {
   if (instance == null)
    instance = new Something();
  }
 }
 return instance;
}
```

看起来是个非常聪明的做法 —— 在常规代码路径上避免了同步。就是有个问题 —— 它不起作用。为什么会不起作用？最显而易见的原因是初始化instance的写操作与写入instance字段的操作能被编译器或缓存重排序，这会导致看到一个部分构建的Something。结果就是我们读到了一个未初始化的对象。还有很多其它原因导致这种模式是错误的以及对它的算法纠正也是错误的。在老的Java内存模型下没有办法修复它。更多深入的信息参见Double-checked locking: Clever, but broken和Double Checked Locking is broken.

有些人认为使用volatile关键字能够消除在双重锁定检查模式中出现的问题。在1.5之前的JVM中，volatile不保证会起作用（因环境而异）。在新的内存模型下，使用volatile的instance将会修复双重锁定检查的这个问题，因为在构造线程初始化Something与读取线程返回instance值之间有happens-before关系。

~~然而，对于喜欢使用双重锁定检查的人（我真心希望没有人用）来说仍然不是好消息。双重锁定检查的要点在于避免同步导致的性能开销。Java1.0时代以来，不仅同步的开销更小，且在新的内存模型下，使用volatile的性能损耗在上升，几乎达到了同步的损耗。所以，没有理由去使用双重锁定检查。~~ 修订——在大部分平台上volatile的开销都比较低。

改为使用线程安全且更易读的Initialization On Demand Holder模式：

```
private static class LazySomethingHolder {
 public static Something something = new Something();
}
public static Something getInstance() {
 return LazySomethingHolder.something;
}
```

静态字段的初始化安全保障保证了这个代码的正确性。如果在静态初始化器中给一个字段赋值，就正确保证

该字段对访问该类的任意线程可见。

如果我要写一个VM呢？

参考http://gee.cs.oswego.edu/dl/jmm/cookbook.html.

为什么我要关心Java内存模型？

为什么我要关心Java内存模型？并发bug难以调试。它们常常不会在测试中暴露，直到程序运行在大负载的情况下，且难以重现和追踪。最好提前花一些时间以确保程序被正确同步了。虽然这不易做到，但相比调试一个未正确同步的应用来说则简单得多。