二

13 Java内存模型

我们先来看一个反常识的例子。

```
int a=0, b=0;

public void method1() {
  int r2 = a;
  b = 1;
}

public void method2() {
  int r1 = b;
  a = 2;
}
```

这里我定义了两个共享变量 a 和 b，以及两个方法。第一个方法将局部变量 r2 赋值为 a，然后将共享变量 b 赋值为 1。第二个方法将局部变量 r1 赋值为 b，然后将共享变量 a 赋值为 2。请问（r1，r2）的可能值都有哪些?

在单线程环境下，我们可以先调用第一个方法，最终（r1，r2）为（1，0）；也可以先调用第二个方法，最终为（0，2）。

在多线程环境下，假设这两个方法分别跑在两个不同的线程之上，如果 Java 虚拟机在执行了任一方法的第一条赋值语句之后便切换线程，那么最终结果将可能出现（0，0）的情况。

除上述三种情况之外，Java 语言规范第 17.4 小节 [1] 还介绍了一种看似不可能的情况（1，2）。

造成这一情况的原因有三个，分别为即时编译器的重排序，处理器的乱序执行，以及内存系统的重排序。由于后两种原因涉及具体的体系架构，我们暂且放到一边。下面我先来讲一下编译器优化的重排序是怎么一回事。

首先需要说明一点，即时编译器（和处理器）需要保证程序能够遵守 as-if-serial 属性。通俗地说，就是在单线程情况下，要给程序一个顺序执行的假象。即经过重排序的执行结果要与顺序执行的结果保持一致。

另外，如果两个操作之间存在数据依赖，那么即时编译器（和处理器）不能调整它们的顺序，否则将会造成程序语义的改变。

```
int a=0, b=0;

public void method1() {
  int r2 = a;
  b = 1;
  .. // Code uses b
  if (r2 == 2) {
    ..
  }
}
```

在上面这段代码中，我扩展了先前例子中的第一个方法。新增的代码会先使用共享变量 b 的值，然后再使用局部变量 r2 的值。

此时，即时编译器有两种选择。

第一，在一开始便将 a 加载至某一寄存器中，并且在接下来 b 的赋值操作以及使用 b 的代码中避免使用该寄存器。第二，在真正使用 r2 时才将 a 加载至寄存器中。这么一来，在执行使用 b 的代码时，我们不再霸占一个通用寄存器，从而减少需要借助栈空间的情况。

```
int a=0, b=0;

public void method1() {
  for (..) {
    int r2 = a;
    b = 1;
    .. // Code uses r2 and rewrites a
  }
}
```

另一个例子则是将第一个方法的代码放入一个循环中。除了原本的两条赋值语句之外，我只在循环中添加了使用 r2，并且更新 a 的代码。由于对 b 的赋值是循环无关的，即时编译器很有可能将其移出循环之前，而对 r2 的赋值语句还停留在循环之中。

如果想要复现这两个场景，你可能需要添加大量有意义的局部变量，来给寄存器分配算法施加压力。

可以看到，即时编译器的优化可能将原本字段访问的执行顺序打乱。在单线程环境下，由于 as-if-serial 的保证，我们无须担心顺序执行不可能发生的情况，如（r1，r2）=（1，2）。

然而，在多线程情况下，这种数据竞争（data race）的情况是有可能发生的。而且，Java 语言规范将其归咎于应用程序没有作出恰当的同步操作。

Java 内存模型与 happens-before 关系

为了让应用程序能够免于数据竞争的干扰，Java 5 引入了明确定义的 Java 内存模型。其中最为重要的一个概念便是 happens-before 关系。happens-before 关系是用来描述两个操作的内存可见性的。如果操作 X happens-before 操作 Y，那么 X 的结果对于 Y 可见。

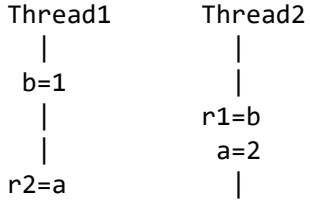
在同一个线程中，字节码的先后顺序（program order）也暗含了 happens-before 关系：在程序控制流路径中靠前的字节码 happens-before 靠后的字节码。然而，这并不意味着前者一定在后者之前执行。实际上，如果后者没有观测前者的运行结果，即后者没有数据依赖于前者，那么它们可能会被重排序。

除了线程内的 happens-before 关系之外，Java 内存模型还定义了下述线程间的 happens-before 关系。

1. 解锁操作 happens-before 之后（这里指时钟顺序先后）对同一把锁的加锁操作。
2. volatile 字段的写操作 happens-before 之后（这里指时钟顺序先后）对同一字段的读操作。
3. 线程的启动操作（即 Thread.starts()） happens-before 该线程的第一个操作。
4. 线程的最后一个操作 happens-before 它的终止事件（即其他线程通过 Thread.isAlive() 或 Thread.join() 判断该线程是否中止）。
5. 线程对其他线程的中断操作 happens-before 被中断线程所收到的中断事件（即被中断线程的 InterruptedException 异常，或者第三个线程针对被中断线程的 Thread.interrupted 或者 Thread.isInterrupted 调用）。
6. 构造器中的最后一个操作 happens-before 析构器的第一个操作。

happens-before 关系还具备传递性。如果操作 X happens-before 操作 Y，而操作 Y happens-before 操作 Z，那么操作 X happens-before 操作 Z。

在文章开头的例子中，程序没有定义任何 happens-before 关系，仅拥有默认的线程内 happens-before 关系。也就是 r2 的赋值操作 happens-before b 的赋值操作，r1 的赋值操作 happens-before a 的赋值操作。



拥有 happens-before 关系的两对赋值操作之间没有数据依赖，因此即时编译器、处理器都

可能对其进行重排序。举例来说，只要将 b 的赋值操作排在 r2 的赋值操作之前，那么便可以按照赋值 b，赋值 r1，赋值 a，赋值 r2 的顺序得到（1，2）的结果。

那么如何解决这个问题呢？答案是，将 a 或者 b 设置为 volatile 字段。

比如说将 b 设置为 volatile 字段。假设 r1 能够观测到 b 的赋值结果 1。显然，这需要 b 的赋值操作在时钟顺序上先于 r1 的赋值操作。根据 volatile 字段的 happens-before 关系，我们知道 b 的赋值操作 happens-before r1 的赋值操作。

```
int a=0;
volatile int b=0;

public void method1() {
  int r2 = a;
  b = 1;
}

public void method2() {
  int r1 = b;
  a = 2;
}
```

根据同一个线程中，字节码顺序所暗含的 happens-before 关系，以及 happens-before 关系的传递性，我们可以轻易得出 r2 的赋值操作 happens-before a 的赋值操作。

这也就意味着，当对 a 进行赋值时，对 r2 的赋值操作已经完成了。因此，在 b 为 volatile 字段的情况下，程序不可能出现（r1，r2）为（1，2）的情况。

由此可以看出，解决这种数据竞争问题的关键在于构造一个跨线程的 happens-before 关系：操作 X happens-before 操作 Y，使得操作 X 之前的字节码的结果对操作 Y 之后的字节码可见。

Java 内存模型的底层实现

在理解了 Java 内存模型的概念之后，我们现在来看看它的底层实现。Java 内存模型是通过内存屏障（memory barrier）来禁止重排序的。

对于即时编译器来说，它会针对前面提到的每一个 happens-before 关系，向正在编译的目标方法中插入相应的读读、读写、写读以及写写内存屏障。

这些内存屏障会限制即时编译器的重排序操作。以 volatile 字段访问为例，所插入的内存屏障将不允许 volatile 字段写操作之前的内存访问被重排序至其之后；也将不允许 volatile 字段读操作之后的内存访问被重排序至其之前。

然后，即时编译器将根据具体的底层体系架构，将这些内存屏障替换成具体的 CPU 指令。以我们日常接触的 X86_64 架构来说，读读、读写以及写写内存屏障是空操作（no-op），只有写读内存屏障会被替换成具体指令 [2]。

在文章开头的例子中，method1 和 method2 之中的代码均属于先读后写（假设 r1 和 r2 被存储在寄存器之中）。X86_64 架构的处理器并不能将读操作重排序至写操作之后，具体可参考 Intel Software Developer Manual Volumn 3，8.2.3.3 小节。因此，我认为例子中的重排序必然是即时编译器造成的。

举例来说，对于 volatile 字段，即时编译器将在 volatile 字段的读写操作前后各插入一些内存屏障。

然而，在 X86_64 架构上，只有 volatile 字段写操作之后的写读内存屏障需要用具体指令来替代。（HotSpot 所选取的具体指令是 lock add DWORD PTR [rsp],0x0，而非 mfence[3]。）

该具体指令的效果，可以简单理解为强制刷新处理器的写缓存。写缓存是处理器用来加速内存存储效率的一项技术。

在碰到内存写操作时，处理器并不会等待该指令结束，而是直接开始下一指令，并且依赖于写缓存将更改的数据同步至主内存（main memory）之中。

强制刷新写缓存，将使得当前线程写入 volatile 字段的值（以及写缓存中已有的其他内存修改），同步至主内存之中。

由于内存写操作同时会无效化其他处理器所持有的、指向同一内存地址的缓存行，因此可以认为其他处理器能够立即见到该 volatile 字段的最新值。

锁，volatile 字段，final 字段与安全发布

下面我来讲讲 Java 内存模型涉及的几个关键词。

前面提到，锁操作同样具备 happens-before 关系。具体来说，解锁操作 happens-before 之后对同一把锁的加锁操作。实际上，在解锁时，Java 虚拟机同样需要强制刷新缓存，使得当前线程所修改的内存对其他线程可见。

需要注意的是，锁操作的 happens-before 规则的关键字是同一把锁。也就意味着，如果编译器能够（通过逃逸分析）证明某把锁仅被同一线程持有，那么它可以移除相应的加锁解锁操作。

因此也就不再强制刷新缓存。举个例子，即时编译后的 synchronized (new Object()) {}，可能等同于空操作，而不会强制刷新缓存。

volatile 字段可以看成一种轻量级的、不保证原子性的同步，其性能往往优于（至少不亚于）锁操作。然而，频繁地访问 volatile 字段也会因为不断地强制刷新缓存而严重影响程序的性能。

在 X86_64 平台上，只有 volatile 字段的写操作会强制刷新缓存。因此，理想情况下对 volatile 字段的使用应当多读少写，并且应当只有一个线程进行写操作。

volatile 字段的另一个特性是即时编译器无法将其分配到寄存器里。换句话说，volatile 字段的每次访问均需要直接从内存中读写。

final 实例字段则涉及新建对象的发布问题。当一个对象包含 final 实例字段时，我们希望其他线程只能看到已初始化的 final 实例字段。

因此，即时编译器会在 final 字段的写操作后插入一个写写屏障，以防某些优化将新建对象的发布（即将实例对象写入一个共享引用中）重排序至 final 字段的写操作之前。在 X86_64 平台上，写写屏障是空操作。

新建对象的安全发布（safe publication）问题不仅仅包括 final 实例字段的可见性，还包括其他实例字段的可见性。

当发布一个已初始化的对象时，我们希望所有已初始化的实例字段对其他线程可见。否则，其他线程可能见到一个仅部分初始化的新建对象，从而造成程序错误。这里我就不展开了。如果你感兴趣的话，可以参考这篇博客 [4]。

总结与实践

今天我主要介绍了 Java 的内存模型。

Java 内存模型通过定义了一系列的 happens-before 操作，让应用程序开发者能够轻易地表达不同线程的操作之间的内存可见性。

在遵守 Java 内存模型的前提下，即时编译器以及底层体系架构能够调整内存访问操作，以达到性能优化的效果。如果开发者没有正确地利用 happens-before 规则，那么将可能导致数据竞争。

Java 内存模型是通过内存屏障来禁止重排序的。对于即时编译器来说，内存屏障将限制它所能做的重排序优化。对于处理器来说，内存屏障会导致缓存的刷新操作。

今天的实践环节，我们来复现文章初始的例子。由于复现需要大量的线程切换事件，因此我借助了 OpenJDK CodeTools 项目的 jcstress 工具 [5]，来对该例子进行并发情况下的压力测试。具体的命令如下所示：

```
$ mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=org.openjdk.jcs
$ cd test
$ echo 'package org.sample;
import org.openjdk.jcstress.annotations.*;
import org.openjdk.jcstress.infra.results.IntResult2;
@JCStressTest
@Outcome(id = {"0, 0", "0, 2", "1, 0"}, expect = Expect.ACCEPTABLE, desc = "Normal
@Outcome(id = {"1, 2"}, expect = Expect.ACCEPTABLE_INTERESTING, desc = "Abnormal ou
@State
public class ConcurrencyTest {
  int a=0;
  int b=0; // 改成 volatile 试试?
  @Actor
  public void method1(IntResult2 r) {
    r.r2 = a;
    b = 1;
  }
  @Actor
  public void method2(IntResult2 r) {
    r.r1 = b;
    a = 2;
  }
}' > src/main/java/org/sample/ConcurrencyTest.java
$ mvn package
$ java -jar target/jcstress.jar
```

如果你想要复现非安全发布的情形，那么你可以试试这一测试用例 [6]。

[1] https://docs.oracle.com/javase/specs/jls/se10/html/jls-17.html#jls-17.4
[2] http://gee.cs.oswego.edu/dl/jmm/cookbook.html [3] https://blogs.oracle.com
/dave/instruction-selection-for-volatile-fences-:-mfence-vs-lock:add [4] http://vlkan.com
/blog/post/2014/02/14/java-safe-publication/ [5] https://wiki.openjdk.java.net/display
/CodeTools/jcstress [6] http://hg.openjdk.java.net/code-tools/jcstress/file/64f2cf32fa0a
/tests-custom/src/main/java/org/openjdk/jcstress/tests/unsafe/UnsafePublication.java

上一页 下一页