02 | Java内存模型：看Java如何解决可见性和有序性问题

2019-03-02 王宝令

上一期我们讲到在并发场景中，因可见性、原子性、有序性导致的问题常常会违背我们的直觉，从而成为并发编程的Bug之源。这三者在编程领域属于共性问题，所有的编程语言都会遇到，Java在诞生之初就支持多线程，自然也有针对这三者的技术方案，而且在编程语言领域处于领先地位。理解Java解决并发问题的解决方案，对于理解其他语言的解决方案有触类旁通的效果。

那我们就先来聊聊如何解决其中的可见性和有序性导致的问题，这也就引出来了今天的主角——**Java内存模型**。

Java内存模型这个概念，在职场的很多面试中都会考核到，是一个热门的考点，也是一个人并发水平的具体体现。原因是当并发程序出问题时，需要一行一行地检查代码，这个时候，只有掌握Java内存模型，才能慧眼如炬地发现问题。

什么是Java内存模型？

你已经知道，导致可见性的原因是缓存，导致有序性的原因是编译优化，那解决可见性、有序性最直接的办法就是**禁用缓存和编译优化**，但是这样问题虽然解决了，我们程序的性能可就堪忧了。

合理的方案应该是**按需禁用缓存以及编译优化**。那么，如何做到“按需禁用”呢？对于并发程序，何时禁用缓存以及编译优化只有程序员知道，那所谓“按需禁用”其实就是指按照程序员的要

求来禁用。所以，为了解决可见性和有序性问题，只需要提供给程序员按需禁用缓存和编译优化的方法即可。

Java内存模型是个很复杂的规范，可以从不同的视角来解读，站在我们这些程序员的视角，本质上可以理解为，Java内存模型规范了JVM如何提供按需禁用缓存和编译优化的方法。具体来说，这些方法包括 **volatile**、**synchronized** 和 **final** 三个关键字，以及六项 **Happens-Before 规则**，这也正是本期的重点内容。

使用volatile的困惑

volatile关键字并不是Java语言的特产，古老的C语言里也有，它最原始的意义就是禁用CPU缓存。

例如，我们声明一个volatile变量 volatile int x = 0，它表达的是：告诉编译器，对这个变量的读写，不能使用CPU缓存，必须从内存中读取或者写入。这个语义看上去相当明确，但是在实际使用的时候却会带来困惑。

例如下面的示例代码，假设线程A执行writer()方法，按照 volatile 语义，会把变量 “v=true” 写入内存；假设线程B执行reader()方法，同样按照 volatile 语义，线程B会从内存中读取变量v，如果线程B看到 “v == true” 时，那么线程B看到的变量x是多少呢？

直觉上看，应该是42，那实际应该是多少呢？这个要看Java的版本，如果在低于1.5版本上运行，x可能是42，也有可能是0；如果在1.5以上的版本上运行，x就是等于42。

```
// 以下代码来源于【参考1】
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }
  public void reader() {
    if (v == true) {
      // 这里x会是多少呢？
    }
  }
}
```

分析一下，为什么1.5以前的版本会出现x = 0的情况呢？我相信你一定想到了，变量x可能被CPU

缓存而导致可见性问题。这个问题在1.5版本已经被圆满解决了。Java内存模型在1.5版本对volatile语义进行了增强。怎么增强的呢？答案是一项 Happens-Before 规则。

Happens-Before 规则

如何理解 Happens-Before 呢？如果望文生义（很多网文也都爱按字面意思翻译成“先行发生”），那就南辕北辙了，Happens-Before 并不是说前面一个操作发生在后续操作的前面，它真正要表达的是：**前面一个操作的结果对后续操作是可见的**。就像有心灵感应的两个人，虽然远隔千里，一个人心之所想，另一个人都看得到。Happens-Before 规则就是要保证线程之间的这种“心灵感应”。所以比较正式的说法是：Happens-Before 约束了编译器的优化行为，虽允许编译器优化，但是要求编译器优化后一定遵守 Happens-Before 规则。

Happens-Before 规则应该是Java内存模型里面最晦涩的内容了，和程序员相关的规则一共有如下六项，都是关于可见性的。

恰好前面示例代码涉及到这六项规则中的前三项，为便于你理解，我也会分析上面的示例代码，来看看规则1、2和3到底该如何理解。至于其他三项，我也会结合其他例子作以说明。

1. 程序的顺序性规则

这条规则是指在一个线程中，按照程序顺序，前面的操作 Happens-Before 于后续的任意操作。这还是比较容易理解的，比如刚才那段示例代码，按照程序的顺序，第6行代码 “x = 42;” Happens-Before 于第7行代码 “v = true;”，这就是规则1的内容，也比较符合单线程里面的思维：程序前面对某个变量的修改一定是对后续操作可见的。

（为方便你查看，我将那段示例代码在这儿再呈现一遍）

```
// 以下代码来源于【参考1】
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }
  public void reader() {
    if (v == true) {
      // 这里x会是多少呢？
    }
  }
}
```

2. volatile变量规则

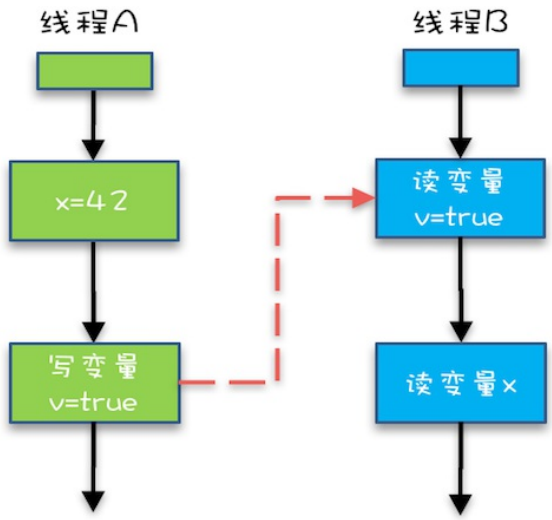
这条规则是指对一个volatile变量的写操作， Happens-Before 于后续对这个volatile变量的读操作。

这个就有点费解了，对一个volatile变量的写操作相对于后续对这个volatile变量的读操作可见，这怎么看都是禁用缓存的意思啊，貌似和1.5版本以前的语义没有变化啊？如果单看这个规则，的确是这样，但是如果我们关联一下规则3，就有点不一样的感觉了。

3. 传递性

这条规则是指如果A Happens-Before B，且B Happens-Before C，那么A Happens-Before C。

我们将规则3的传递性应用到我们的例子中，会发生什么呢？可以看下面这幅图：



示例代码中的传递性规则

从图中，我们可以看到：

1. “x=42” Happens-Before 写变量 “v=true” ，这是规则1的内容；
2. 写变量“v=true” Happens-Before 读变量 “v=true”，这是规则2的内容 。

再根据这个传递性规则，我们得到结果：“x=42” Happens-Before 读变量“v=true”。这意味着什么呢？

如果线程B读到了“v=true”，那么线程A设置的“x=42”对线程B是可见的。也就是说，线程B能看到 “x == 42” ，有没有一种恍然大悟的感觉？这就是1.5版本对volatile语义的增强，这个增强意义

重大，1.5版本的并发工具包（java.util.concurrent）就是靠volatile语义来搞定可见性的，这个在后面的内容中会详细介绍。

4. 管程中锁的规则

这条规则是指对一个锁的解锁 Happens-Before 于后续对这个锁的加锁。

要理解这个规则，就首先要了解“管程指的是什么”。**管程**是一种通用的同步原语，在Java中指的就是synchronized，synchronized是Java里对管程的实现。

管程中的锁在Java里是隐式实现的，例如下面的代码，在进入同步块之前，会自动加锁，而在代码块执行完会自动释放锁，加锁以及释放锁都是编译器帮我们实现的。

```
synchronized (this) { //此处自动加锁
  // x是共享变量,初始值=10
  if (this.x < 12) {
    this.x = 12;
  }
} //此处自动解锁
```

所以结合规则4——管程中锁的规则，可以这样理解：假设x的初始值是10，线程A执行完代码块后x的值会变成12（执行完自动释放锁），线程B进入代码块时，能够看到线程A对x的写操作，也就是线程B能够看到x==12。这个也是符合我们直觉的，应该不难理解。

5. 线程 start() 规则

这条是关于线程启动的。它是指主线程A启动子线程B后，子线程B能够看到主线程在启动子线程B前的操作。

换句话说就是，如果线程A调用线程B的 start() 方法（即在线程A中启动线程B），那么该start() 操作 Happens-Before 于线程B中的任意操作。具体可参考下面示例代码。

```
Thread B = new Thread(()->{
  // 主线程调用B.start()之前
  // 所有对共享变量的修改，此处皆可见
  // 此例中，var==77
});
// 此处对共享变量var修改
var = 77;
// 主线程启动子线程
B.start();
```

6. 线程 join() 规则

这条是关于线程等待的。它是指主线程A等待子线程B完成（主线程A通过调用子线程B的join()方法实现），当子线程B完成后（主线程A中join()方法返回），主线程能够看到子线程的操作。当然所谓的“看到”，指的是对**共享变量**的操作。

换句话说就是，如果在线程A中，调用线程B的 join() 并成功返回，那么线程B中的任意操作 Happens-Before 于该 join() 操作的返回。具体可参考下面示例代码。

```
Thread B = new Thread(()->{
  // 此处对共享变量var修改
  var = 66;
});
// 例如此处对共享变量修改，
// 则这个修改结果对线程B可见
// 主线程启动子线程
B.start();
B.join()
// 子线程所有对共享变量的修改
// 在主线程调用B.join()之后皆可见
// 此例中，var==66
```

被我们忽视的final

前面我们讲volatile为的是禁用缓存以及编译优化，我们再从另外一个方面来看，有没有办法告诉编译器优化得更好一点呢？这个可以有，就是**final关键字**。

final修饰变量时，初衷是告诉编译器：这个变量生而不变，可以可劲儿优化。Java编译器

在1.5以前的版本的确优化得很努力，以至于都优化错了。

问题类似于上一期提到的利用双重检查方法创建单例，构造函数的错误重排导致线程可能看到final变量的值会变化。详细的案例可以参考这个文档。

当然了，在1.5以后Java内存模型对final类型变量的重排进行了约束。现在只要我们提供正确构造函数没有“逸出”，就不会出问题了。

“逸出”有点抽象，我们还是举个例子吧，在下面例子中，在构造函数里面将this赋值给了全局变量global.obj，这就是“逸出”，线程通过global.obj读取x是有可能读到0的。因此我们一定要避免“逸出”。

```
// 以下代码来源于【参考1】
final int x;
// 错误的构造函数
public FinalFieldExample() {
  x = 3;
  y = 4;
  // 此处就是讲this逸出，
  global.obj = this;
}
```

总结

Java的内存模型是并发编程领域的一次重要创新，之后C++、C#、Golang等高级语言都开始支持内存模型。Java内存模型里面，最晦涩的部分就是Happens-Before规则了，Happens-Before规则最初是在一篇叫做**Time, Clocks, and the Ordering of Events in a Distributed System**的论文中提出来的，在这篇论文中，Happens-Before的语义是一种因果关系。在现实世界里，如果A事件是导致B事件的起因，那么A事件一定是先于（Happens-Before）B事件发生的，这个就是Happens-Before语义的现实理解。

在Java语言里面，Happens-Before的语义本质上是一种可见性，A Happens-Before B 意味着A事件对B事件来说是可见的，无论A事件和B事件是否发生在同一个线程里。例如A事件发生在线程1上，B事件发生在线程2上，Happens-Before规则保证线程2上也能看到A事件的发生。

Java内存模型主要分为两部分，一部分面向你我这种编写并发程序的应用开发人员，另一部分是面向JVM的实现人员的，我们可以重点关注前者，也就是和编写并发程序相关的部分，这部分内容的核心就是Happens-Before规则。相信经过本章的介绍，你应该对这部分内容已经有了深入的认识。

课后思考

有一个共享变量 abc，在一个线程里设置了abc的值 abc=3，你思考一下，有哪些办法可以让其他线程能够看到abc==3？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

参考

1. JSR 133 (Java Memory Model) FAQ
2. Java内存模型FAQ
3. JSR-133: Java™ Memory Model and Thread Specification

精选留言

Handongyang 320

老师，还差两个规则，分别是：
线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。
对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始。

所以，个人对于Java内存模型总结起来就是：

1. 为什么定义Java内存模型？现代计算机体系大部是采用的对称多处理器的体系架构。每个处理器均有独立的寄存器组和缓存，多个处理器可同时执行同一进程中的不同线程，这里称为处理器的乱序执行。在Java中，不同的线程可能访问同一个共享或共享变量。如果任由编译器或处理器对这些访问进行优化的话，很有可能出现无法想象的问题，这里称为编译器的重排序。除了处理器的乱序执行、编译器的重排序，还有内存系统的重排序。因此Java语言规范引入了Java内存模型，通过定义多项规则对编译器和处理器进行限制，主要是针对可见性和有序性。
2. 三个基本原则：原子性、可见性、有序性。
3. Java内存模型涉及的几个关键词：锁、volatile字段、final修饰符与对象的安全发布。其中：第一是锁，锁操作是具备happens-before关系的，解锁操作happens-before之后对同一把锁的加锁操作。实际上，在解锁的时候，JVM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。第二是volatile字段，volatile字段可以看成是一种不保证原子性的同步但保证可见性的特性，其性能往往是优于锁操作的。但是，频繁地访问 volatile字段也会出现因为不断地强制刷新缓存而影响程序的性能的问题。第三是final修饰符，final修饰的实例字段则是涉及到新建对象的发布问题。当一个对象包含final修饰的实例字段时，其他线程能够看到已经初始化的final实例字段，这是安全的。
4. Happens-Before的7个规则：
(1).程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构
。
(2).管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须强调的是同一个锁，而"后面"是指时间上的先后顺序。
(3).volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作，这里的"后面"同样是指时间上的先后顺序。
(4).线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
(5).线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过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()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。
(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)先行发生于它的finalize()方法的开始。
5. Happens-Before的1个特性：传递性。
6. Java内存模型底层怎么实现的？主要是通过内存屏障(memory barrier)禁止重排序的，即时编译器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。对于编译器而言，内存屏障将限制它所能做的重排序优化。而对于处理器而言，内存屏障将会导致缓存的刷新操作。比如，对于volatile，编译器将在volatile字段的读写操作前后各插入一些内存屏障。

2019-03-02

作者回复

厉害厉害！！！

2019-03-02

senekis

68

我思考下认为有三种方式可以实现:

1.声明共享变量abc，并使用volatile关键字修饰abc
2.声明共享变量abc，在synchronized关键字对abc的赋值代码块加锁，由于Happen-before管程锁的规则，可以使得后续的线程可以看到abc的值。
3.A线程启动后，使用A.JOIN()方法来完成运行，后续线程再启动，则一定可以看到abc==3

如有错误，请给指出错误所在！谢谢大家！谢谢老师！

听课后感觉对我帮助好大，以前零碎的知识被重新系统的整理。错误的理解也得到修正，感谢老师！

2019-03-02

作者回复

这三种方式都正确，理解的不错！

2019-03-02

Junzi

49

参考1中write()方法代码：
x=45; // 1
v=true; // 2
这两行会不会导致指令重排？

因为volatile关键字应该只保证了变量v的可见性，happen-before第一条原则在单线程中，1与2重排并不影响结果，那应该有可能出现重排的情况，这样线程B读取到read()的时候也有可能出现x=0。还请老师解答。

2019-03-04

狂战俄洛伊

33

回复tracer的问题@tracer，你说的这个问题其实就是一个happens-before原则。例如有以下代码：
int a = 1;//代码1
int b = 2;//代码2
volatile int c = 3;//代码3
int d = 4;//代码4
int e = 5;//代码5
编译器解释这5行代码的时候，会保证代码1和代码2会在代码3之前执行，而代码1和代码2的执行顺序则不一定（这就是重排序，在不影响执行结果的情况下，虚拟机可能会对命令重排。当然所谓的不影响执行结果，java只保证在单线程中不影响执行结果）。代码4和代码5也一定会在代码3之后执行，同理代码4和代码5的执行顺序也是不一定的。
所以这篇文章中你说的那段代码，由于v是volatile修饰的，对v的赋值永远在对x的赋值之后。所以在reader中输出的x一定是42

2019-03-02

作者回复

感谢回复！

2019-03-02

Jerry银银 20

思考题的通用性表述为：如何保证一个共享变量的可见性？

有以下方法：

1. 保证共享变量的可见性，使用volatile关键字修饰即可
2. 保证共享变量是private，访问变量使用set/get方法，使用synchronized对方法加锁，此种方法不仅保证了可见性，也保证了线程安全
3. 使用原子变量，例如：AtomicInteger等
4. 最后一种不是办法的办法：保证多个线程是「串行执行」^_^

2019-03-02

作者回复

很全面了!

2019-03-02

小和尚笨南北 19

补充一个： 在abc赋值后对一个volatile变量A进行赋值操作，然后在其他线程读取abc之前读取A的值，通过volatile的可见性和happen-before的传递性实现abc修改后对其他线程立即可见

2019-03-02

作者回复

这个我称为炫技!

2019-03-03

发条橙子 。 18

感悟：

老师用第一篇介绍了造成并发问题的由来引出了此文如果解决其中的 可见性、排序性问题 。有了第一篇做铺垫让此篇看起来更加的流畅 。

尤其以前看书中讲解 happens-before原则只是单单把六个规则点列了出来，很难吃透。此篇文章给出详细的事例逐点分析，使得更好的去理解每个点。

例如 我之前看到的文章都说 在单线程中不会出现有序性问题，在多线程中会出现有序性问题 。 之前很难理解单线程中没有有序性的问题是什么原因， 原来是happens-before第一条规则限制住了编译器的优化

问题：

第一个例子中添加了 volatile 关键字， 如果例子中， v变量没有使用volatile，那么x会是什么呢？？

答案： 42

我的思考是， 没有了volatile那么规则二就不满足， 但是规则一和规则三还是满足，虽然 writer()方法修改 v不能让其他立即可见，但是如果是循环调用reader()方法，等到可见到 v == true，根据第一条原则， x happens-before v，所以能读到 x=42

老师请问我的判断正确么？

思考题：

一个共享变量在一个线程中修改让另其他线程可见， 那就是解决可见性（缓存）的问题，happens-before的规则就是用于对可见性进行约束的

按照老师课中所讲：
思考如下：

1. 第一条规则同线程中编译会保证顺序性， 和问题不符合

2. 第二条规则， 使用volatile关键字， 这个关键字可以让其他线程写之前先读最新的值，所以保证读到的是最新的值，可行

3. 第三条规则，传递性， 和问题不符

4. 第四条规则， 使用管程，由于是访问共享变量，如果是在syn中修改值只能保证当前线程下一次进入syn可以看见最新的值，其他线程直接访问还可能不是最新值， 不行

5. 第五条规则， 如果前提是其他线程都在 主线程修改abc变量后 start()，则可见

6. 第六条规则，如果前提是其他线程等 修改abc变量线程 join()执行，则可见

7. Final关键字， 由于final关键字表示已经定义了常量，任意线程都不可以修改， 不可用

综上总结：

使用2 添加volatile可行 。在符合某些场景下时，56可让其他线程可见

2019-03-02

作者回复

你分析的比我还要好！

2019-03-02

tracer 14

我明白了，写先于读指的是不会因为cpu缓存，导致a线程已经写了，但是b线程没读到的情况。我错误理解成了b要读，一定要等a写完才行

2019-03-02

作者回复

终于理解了！

2019-03-02

WL

11

想问一下老师最后关于逸出的例子，是因为有可能通过global.obj 可能访问到还没有初始化的this对象吗，但是将this赋值给global.obj不也是初始化时才赋值的吗，这部分不太理解，请老师指点一下

2019-03-02

作者回复

有可能通过global.obj 可能访问到还没有初始化的this对象

将this赋值给global.obj时，this还没有初始化完，this还没有初始化完，this还没有初始化完。

2019-03-02

小麦

9

@发条橙子 ...

有问题吧，我是这样理解的，第一条规则是串行语义，在单线程的场景下，优化后的结果会与顺序执行一致，但是不代表对x的写操作会比对v的写先执行，所以多线程下会出现问题。加volatile关键字后，volatile变量在写操作之后会插入一个store屏障（Store屏障，是x86的”sfence“指令，强制所有在store屏障指令之前的store指令，都在该store屏障指令执行之前被执行，并把store缓冲区的数据都刷到CPU缓存。这会使得程序状态对其它CPU可见，这样其它CPU可以根据需要介入。--并发编程网），所以禁止了重排序，这才保证了对x的写操作会比对v的写先执行，然后再根据volatile变量规则跟传递性原则，才保证了x=42对线程B可见。

2019-03-04

Nevermore

9

```
// 以下代码来源于【参考 1】
class VolatileExample {
int x = 0;
volatile boolean v = false;
public void writer() {
x = 42;
v = true;
}
public void reader() {
if (v == true) {
// 这里 x 会是多少呢？
}
}
}
```

感觉老师对这个volatile变量规则这块讲的有点草率，volatile变量的写对于读是可见的，对于程序来说，也就是线程A执行write中的v=true对于reader中的v==true是可见的 ，但是这对于x有什么关系？x并没有被volatile修饰。

根据我的理解，volatile强制所修饰的变量及它前边的变量刷新至内存，并且volatile禁止了指令

的重排序。

望指正

2019-03-02

作者回复

你的理解是对的，volatile的实现就是这样的。指导JVM这么实现的规范就是内存模型。这个专栏的侧重点是让大家学会写并发程序，至于底层是怎么实现的，有精力和兴趣的同学，可以自己来把握。

2019-03-02

强哥 8

关于java内存模型、jvm内存结构及java对象模型分别深入讲解一下，这样效果更好一些。

2019-03-02

作者回复

咱们这个专栏还是专注于并发相关的部分，我怕有人说挂羊头卖狗肉

2019-03-02

magict4 6

老师你好，

我对『3. 传递性』中您的解释，还是有点疑惑。感觉许多留言的小伙伴们也都有类似的疑惑，还请老师再耐心回答一次。

您提到：
> “x=42” Happens-Before 写变量 “v=true” ，这是规则 1 的内容；
我的疑惑：变量 x 和 v 没有任何依赖关系，为什么对 x 的赋值 Happens-Before 对 v 的赋值呢？

这个 Happens-Before 关系，根据我的理解，不是由规则 1 决定的，而是有 volatile 决定的。如果 v 没有被 volatile 修饰，编译器是可以对 x、v 的赋值语句进行重排的。 不知道我的理解是否有问题？

2019-03-04

作者回复

“x=42” Happens-Before 写变量 “v=true”
是因为程序顺序就是这么写的：x=42；v=true

这个案例是综合了 程序的顺序规则+传递规则+volatile 规则

这三这个规则组合在一起就是你所谓的：“而是有 volatile 决定的”。编译器优化要遵循所有的HB规则。所有，不是一条。所以只有把他们组合在一起才有意义。

2019-03-04

李 6

老师，第一章里提到程序中x=5；x=6可能被重排。可是今天第一个规则里提到，同一个线程里，是顺序的。这两个不就矛盾了吗？

2019-03-03

作者回复

可以重排，但是要保证符合Happens-Before规则，Happens-Before规则关注的是可见性，

x=5;

y=6;

z=x+y;

上面的代码重排成这样：

y=6;

x=5;

z=x+y;

也是可以的。

所谓顺序，指的是你可以用顺序的方式推演程序的执行，但是程序的执行不一定是完全顺序的。编译器保证结果一定 == 顺序方式推演的结果

这几条规则，都是告诉你，可以按照这个规则推演程序的执行。但是编译怎么优化，那就百花齐放了。

2019-03-03

鸠翱

5

对于@Junzi的问题：

x=45; // 1

v=true; // 2

这两行会不会导致指令重排？ 答：不会

如果这两行重排序了，那么线程B读取到read()的时候也有可能出现x=0，也就是说线程B看到了v=true却没又看到x=45，这不符合第一条规则（请问老师 这么理解对不对）

我课外查询了一下，从实现方法上，volatile的读写前后会插入内存屏障，保证一些操作是无法没重排序的，其中就有对于volatile的写操作之前的动作不会被重排序到之后

2019-03-04

作者回复

是这样。

2019-03-04

峰

5

我觉得课后题其实就是利用happenbefore规则去构建abc的写入happenfore于另外一个线程的读取。而6条规则中传递性规则是纽带，然后采用比如规则4，就是把abc的赋值加入一同步块，并先执行，同时另外一个线程申请同一把锁即可。其他的也类似。

java内存模型对程序员来说提供了按需禁止缓存禁止指令重排的方法。这是我第一次看到这么简单又深刻的解释，老师棒棒哒！！！

2019-03-02

作者回复

多谢鼓励啊！

2019-03-02

飞翔的花狸猫 4

Happen-before 这个知识点终于理解了，追并发专栏比以前看小说还勤快，盼老师速更啊

2019-03-02

作者回复

那小说得写的有多烂！

2019-03-02

王位庆 3

@Junzi，您提的问题刚开始我也很疑惑，但查看了java并发编程的艺术，书上写了jdk1.5之后，增强了volatile的内存含义，限制了编译器和处理器对volatile变量和普通变量的重排序。p47页

2019-04-03

null 2

@发条橙子... 的思考题分析，有些不太准确吧，例如评论里指出的程序顺序性。还有 synchronized 的分析也不太准确吧，synchronized(abc) 可能保证后续操作可见。

老师是否应该在回复评论时指正，否则童鞋们看到“分析得比我好”的回复，很大可能就照着分析来理解了。

2019-04-10

作者回复

synchronized 的分析没有问题，其他线程直接访问还可能不是最新值，我理解直接就是没有使用任何同步手段。

即便用 synchronized，用法不对，也达不到效果。

语言本来就不准确，留言的同学，写的都很随意，实在不太适合以批判的眼光来看待。

2019-04-10

Hello,Hello,Hello Kitty 2

x = 42;

v = true;

老师 我想请假一下 x并没有加volatile修饰 JVM是如何保证x的结果也是被可见呢 是因为x跟v都在同一个寄存器中 voatile修饰的v被刷新回内存的时候 整个寄存器中的值都被刷新到内存中了吗？

2019-03-11

作者回复

这方面，我也很难通透地讲清楚从Java代码到CPU指令的过程，单线程保证x可见很容易，多线程x跟v不能都同时在一个寄存器里。voatile v刷新回内存的时候，所有CPU缓存里的值都会失效。下次访问就能访问到新的了。

2019-03-11