

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



讲述：王宝令

时长 14:27 大小 13.24M



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、有序性最直接的解决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。

 复制代码

```
1 // 以下代码来源于【参考 1】
2 class VolatileExample {
3     int x = 0;
4     volatile boolean v = false;
5     public void writer() {
```

```
6      x = 42;
7      v = true;
8  }
9  public void reader() {
10     if (v == true) {
11         // 这里 x 会是多少呢?
12     }
13 }
14 }
```

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` 于后续的任何操作。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)

 复制代码

```
1 // 以下代码来源于【参考 1】
2 class VolatileExample {
3     int x = 0;
4     volatile boolean v = false;
5     public void writer() {
6         x = 42;
7         v = true;
8     }
9     public void reader() {
10         if (v == true) {
11             // 这里 x 会是多少呢？
12         }
13     }
14 }
```

2. volatile 变量规则

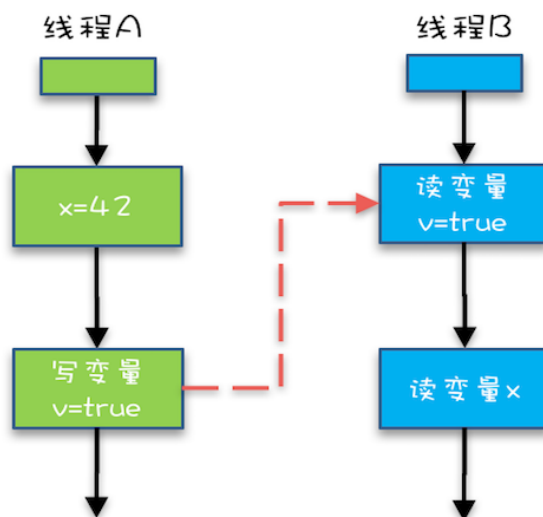
这条规则是指对一个 volatile 变量的写操作，Happens-Before 于后续对这个 volatile 变量的读操作。

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。

 复制代码


```
1 synchronized (this) { // 此处自动加锁
2     // x 是共享变量，初始值 =10
3     if (this.x < 12) {
4         this.x = 12;
5     }
6 } // 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。

 复制代码


```
1 Thread B = new Thread()->{
2     // 主线程调用 B.start() 之前
3     // 所有对共享变量的修改，此处皆可见
4     // 此例中，var==77
5 };
6 // 此处对共享变量 var 修改
7 var = 77;
8 // 主线程启动子线程
9 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。

 复制代码

```
1 Thread B = new Thread()->{
2     // 此处对共享变量 var 修改
3     var = 66;
4 };
5 // 例如此处对共享变量修改，
6 // 则这个修改结果对线程 B 可见
7 // 主线程启动子线程
8 B.start();
9 B.join()
10 // 子线程所有对共享变量的修改
11 // 在主线程调用 B.join() 之后皆可见
12 // 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”。

 复制代码

```
1 // 以下代码来源于【参考 1】
2 final int x;
3 // 错误的构造函数
4 public FinalFieldExample() {
5     x = 3;
6     y = 4;
7     // 此处就是讲 this 逸出，
8     global.obj = this;
9 }
```

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](#)



Java 并发编程实战

全面系统提升你的并发编程能力

王宝令
资深架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 01 | 可见性、原子性和有序性问题：并发编程Bug的源头

精选留言 (47)

 写留言



Handongya...

2019-03-02

 31

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。...

展开 ∨

作者回复: 厉害厉害!!!



senekis

2019-03-02

👍 29

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。...

展开 ∨

作者回复: 这三种方式都正确, 理解的不错!



狂战俄洛伊

2019-03-02

👍 7

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...
```

展开 ∨

作者回复: 感谢回复!



Jerry银银

2019-03-02

👍 5

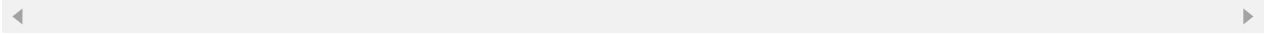
思考题的通用性表述为: 如何保证一个共享变量的可见性?

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...

展开 ∨

作者回复: 很全面了!



发条橙子 ...

2019-03-02

👍 5

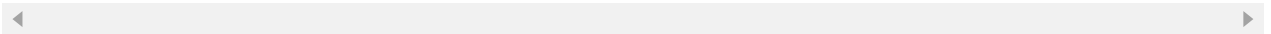
感悟：

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...

展开 ▾

作者回复: 你分析的比我还要好！



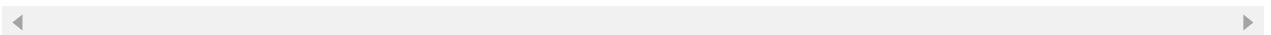
tracer

2019-03-02

👍 4

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

作者回复: 终于理解了！



WL

2019-03-02

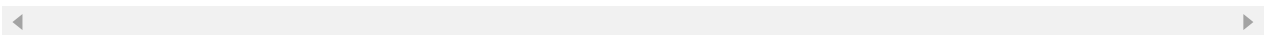
👍 4

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

展开 ▾

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。





Nevermore

2019-03-02

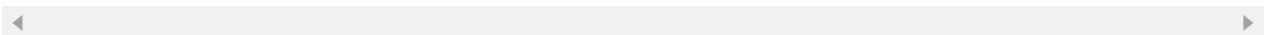
👍 4

// 以下代码来源于【参考 1】

```
class VolatileExample {  
    int x = 0;  
    volatile boolean v = false;  
    public void writer() {...
```

展开 ▾

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。



李

2019-03-03

👍 3

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?

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;
```

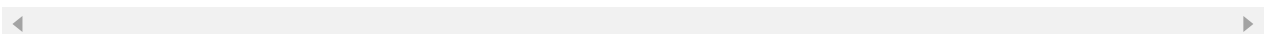
上面的代码重排成这样:

```
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。



飞翔的花狸...

👍 3

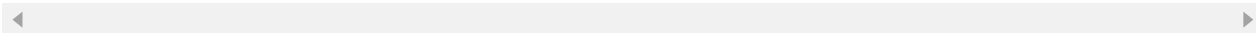


2019-03-02

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

展开 ▾

作者回复: 那小说得写的有多烂！



峰

2019-03-02

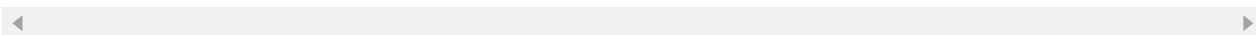
👍 3

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...

展开 ▾

作者回复: 多谢鼓励啊！



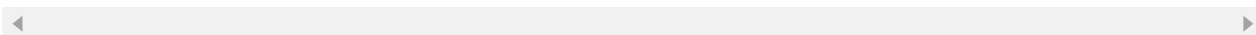
小和尚笨南...

2019-03-02

👍 2

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

作者回复: 这个我称为炫技！



wang

2019-03-02

👍 2

老师。对呀 发条橙子 提到如果不加 volatile，当读到v的时候，x就一定能读到42，使用的是使用的是规则1。

我认为不对呀，规则一不是只适用于单线程吗？而读取v是在另一个线程，所以不能使用规则一判断吧。

希望老师可以解释一下，谢谢

展开 ▾

作者回复: 不加volatile，多线程会有问题



柳絮飞

2019-03-04

👍 1

王老师，请教一下，为什么这些编译优化规则叫内存模型？jvm的堆和栈应该叫什么？多谢！这样叫是不是容易让人误解



magict4

2019-03-04

👍 1

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。

...

展开 ▾



Junzi

2019-03-04

👍 1

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？

...

展开 ▾



倚梦流

2019-03-02

👍 1

这里实例里的共享变量为什么没有用static修饰？是因为这里的线程操作的都是同一个实例，所以共享变量不需要用static修饰吗？如果用了static，结果应该也是一样的吧

作者回复: 是这样的，多个线程操作一个实例。

共享变量不一定用static修饰

结果都一样



tracer

2019-03-02

👍 1

提一个可能比较幼稚的问题，参考1中的代码，volatile能保证不被重排序，但是volatile只修饰了v，意思是代码块中只要有一个volatile，代码块中都不会被重排序？volatile变量规则说写操作对读操作可见，v是有初始值的，线程b先读v不可以吗？那如果线程a中多次修改了v的值呢？

作者回复: 代码块中有volatile，也挡不住被重排。但是重排后的结果一定符合happens-before规则。

b先读v可以啊，只是这个时候v==false



拨云见天

2019-03-02

👍 1

@何方妖孽

你提的这个问题是java8的新特性之lambda表达式，可以学习下这个新特性，简化代码必备良药哦！

展开 ▾



相民

2019-03-02

👍 1

@何方妖孽 JAVA8引入的Lambda语法

展开 ▾