**02 | Java内存模型：看Java如何解决可见性和有序性问题**

**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。

// 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？

}

}

}

1. **volatile 变量规则**

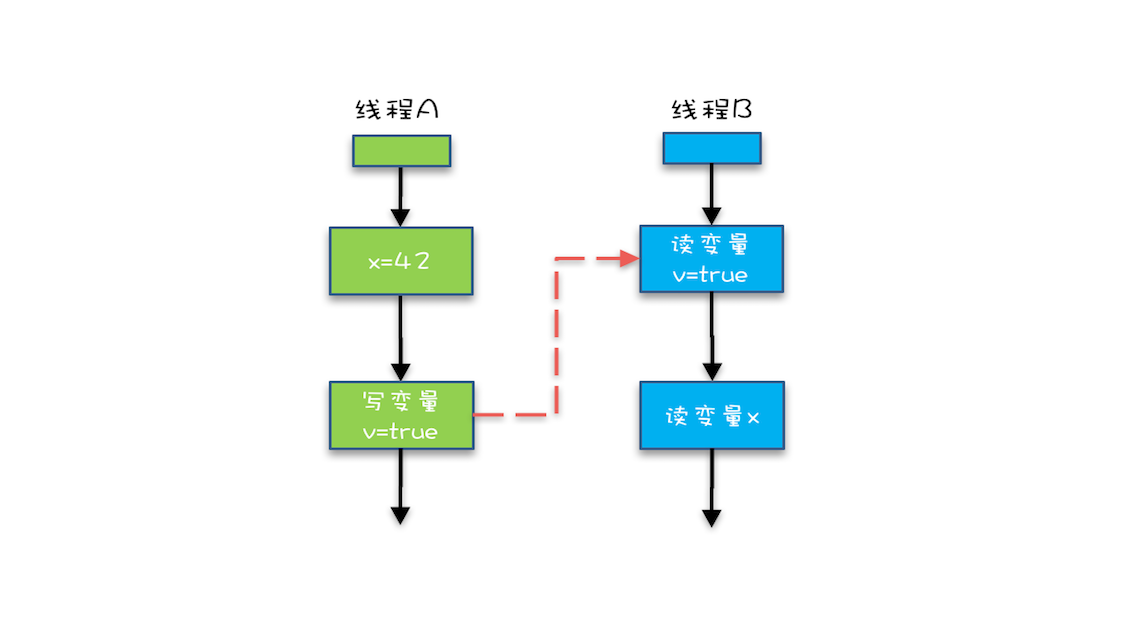
这条规则是指对一个 volatile 变量的写操作， Happens-Before 于后续对这个 volatile 变量的读操作。

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。

1. **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。

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。

**优质总结**

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。

1. 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。
2. (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。

**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？

1. 使用volatile修饰abc -禁止cpu缓存直接从内存获取和volatile写 happens before volatile读。  
2. synchronized 代码块中操作abc 解锁happens before 加锁。  
3. 线程A操作共享变量abc然后start方法启动B线程 B线程中可见abc操作。  
4. 线程A操作共享变量abc，B join A 对于B线程可见。

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