JSR-133：Java™ 内存模型与线程规范

翻译：ticmy

本文是 JSR-133 规范，即 *Java™ 内存模型与线程规范*，由 JSR-133 专家组开发。本规范是 JSR-176（定义了 Java™ 平台 Tiger（5.0）发布版的主要特性）的一部分。本规范的标准内容将合并到 *Java™ 语言规范*、*Java™ 虚拟机规范*以及 java.lang 包的类说明中。本 JSR-133 规范将不再通过 JCP 维护和修改。未来所有对这些标准化内容的更新、修正以及说明都会出现在上述这些文档中。

本规范的标准化内容包含在第 5, 7, 9.2, 9.3, 11, 12, 14, 15 以及 16 节。其它章节，以及上述提到的章节的部分内容，属非标准化内容，用于解释和说明标准化内容。如果标准化内容和非标准化内容有冲突，以标准化内容为准。

本规范的讨论与开发异常复杂且专业性强，需要对一些学术论题有深刻的见解并了解它们的发展过程。这些讨论在 JMM web 站点上都有存档。该站点提供了额外的信息，可以帮助理解本规范形成的过程；站点是：

http://www.cs.umd.edu/~pugh/java/memoryModel

上述 web 站点以及邮件列表将持续更新和维护，非标准化内容，有助于人们理解 JSR-133 规范，未来若有更新和扩展，将可以从该站点上找到。

在对 JLS 原始规范的改变中，有两处最有可能要求 JVM 实现也做出相应的变动：

- 加强了 volatile 变量的语义，需要有 acquire 和 release 语义。在原始的规范中，volatile 变量的访问和非 volatile 变量的访问之间可以自由地重排序。
- 加强了 final 字段的语义，无需显式地同步，不可变对象也是线程安全的。这可能需要在给 final 字段赋值的那些构造器的末尾加上 store-store 屏障。

1 介绍

Java™ 虚拟机支持多线程执行。线程是用 Thread 类来表示的。用户创建一个线程的唯一方式是创建一个该类的对象；每个线程都与这样一个对象相关联。在对应的 Thread 对象上调用 start()方法将启动线程。

线程的行为，尤其是未正确同步时的行为，可能会让人困惑和违背直觉。本规范描述了用 Java™ 语言编写的多线程程序的语义；包括多线程更新共享内存时，读操作能看到什么值的规则。因为本规范与不同的硬件架构的内存模型相似，所以，这里的语义都指的是 Java™ 内存模型。

这些语义不会去描述多线程程序该如何执行。而是描述多线程程序允许表现出的行为。任何执行策略，只要产生的是允许的行为，那它就是一个可接受的执行策略。

1.1 锁

有多种机制可以用于线程间通信。其中最基础的是*同步*，同步是用管程来实现的。每个对象都关联着一个管程，线程可以通过它来执行*锁定（lock）*或*解锁（unlock）*操作。每次仅有一个线程可以持有管程上的锁。其它试图锁定该管程的线程会一直阻塞，直到能从该管程上获得锁为止。

线程 *t* 可以锁定一个管程多次，每个 unlock 操作都会将一次 lock 操作撤销。

synchronized 语句需要一个对象的引用；随后会尝试在该对象的管程上执行 lock 动作，如果 lock 动作未能成功完成，将一直等待。当 lock 动作执行成功，就会运行

synchronized 语句块中的代码。一旦语句块中的代码执行结束，不管是正常还是异常结束，都会在之前执行 lock 动作的那个管程上自动执行一个 unlock 动作。

synchronized 方法在调用时会自动执行一个 lock 动作。在 lock 动作成功完成之前，都不会执行方法体。如果是实例方法，锁的是调用该方法的实例（即，方法体执行期间的 this）相关联的管程。如果是静态方法，锁的是定义该方法的类所对应的 Class 对象。一旦方法体执行结束，不管是正常还是异常结束，都会在之前执行 lock 动作的那个管程上自动执行一个 unlock 动作。

语义既不阻止也不要求对死锁条件进行检测。程序中若线程会（直接地或间接地）锁定多个对象，应当采取一些手段来避免死锁，若有必要，创建一些更高级别的不会死锁的锁原语。

其它诸如读写 volatile 变量，以及 java.util.concurrent 包中的类，都为同步提供了可选的机制。

1.2 示例中的表示法

Java 内存模型并没有完全地基于 Java 语言的面向对象特性。为了保持示例简洁明了，通常会展示一些没有 类或方法定义或显式解引用的代码片段。大部分例子都包含两或多个线程访问局部变量、共享全局变量或对象的实例字段。通常使用变量名 r1 或 r2 来表示方法或线程的局部变量，这些变量不会被其它线程访问。

2 未正确同步的程序会表现出出人意料的行为

Java 的语义允许编译器和微处理器进行优化，这会影响到未正确同步的代码，可能会使它们的行为看起来自相矛盾。

Original code
Initially, A == B == 0

Thread 1	Thread 2
1: r2 = A;	3: r1 = B
2: B = 1;	4: A = 2

May observe r2 == 2, r1 == 1

Valid compiler transformation
Initially, A == B == 0

Thread 1	Thread 2
B = 1;	r1 = B
r2 = A;	A = 2

May observe r2 == 2, r1 == 1

图 1：语句重排序导致的出人意料的结果

考虑图 1 中的例子。程序中用到了局部变量 r1 和 r2，以及共享变量 A 和 B。可能会出现 r2 == 2、r1 == 1 这样的结果。直觉上，应当要么指令 1 先执行要么指令 3 先执行。如果指令 1 先执行，它不应该能看到指令 4 中写入的值。如果指令 3 先执行，它不应该能看到指令 2 写的值。

如果某次执行表现出了这样的行为，那么我们可能得出这样的结论，指令 4 要在指令 1 之前执行，指令 1 要在指令 2 之前执行，指令 2 要在指令 3 之前执行，指令 3 要在指令 4 之前执行。如此，从表面看来，有悖常理。

然而，从单个线程的角度看，只要重排序不会影响到该线程的执行结果，编译器就可以对该线程中的指令进行重排序。如果指令 1 与指令 2 重排序，那就很容易看出为什么会出现 r2 == 2 和 r1 == 1 这样的结果了。

一些编程人员可能认为程序表现出这种行为是不对的。但是，需要注意的是，这段代码没有被充分同步：

- 一个线程里有个写操作，
- 另一个线程读取了这个写入的变量值，
- 且读写操作没有被同步排序。

当上述情况发生时，称之为存在*数据争用（data race）*。当代码中存在数据争用时，常有可能出现有违直觉的结果。

有几种机制都可以产生图 1 中的重排序。JIT 编译器和处理器可以对代码进行重新整理。此外，运行 JVM 的机器的分级存储系统可以使代码看起来像被重排序过。为简单起见，任何能够对代码进行重排序的东西，我们称之为编译器。源代码到字节码的转换过程中可以重排序和改变程序，但必须是本规范允许的那些方式。

图 2 是另一个会产生出人意料的结果的例子。这个程序也没有正确同步；它里面对共享内存访问的操作之间没有强加任何顺序。

Original code

Initially: `p == q, p.x == 0`

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

May observe `r2 == r5 == 0, r4 == 3`?

Valid compiler transformation

Initially: `p == q, p.x == 0`

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

May observe `r2 == r5 == 0, r4 == 3`

图 2：前向替换导致的出人意料的结果

一种常规的编译器优化会在使用 r5 的时候重用 r2：它们读取的都是 r1.x 且它们之间没有写 r1.x 的操作。

现在考虑这样一种情况，在线程 1 第一次读取 r1.x 与读取 r3.x 之间，线程 2 对 r6.x 进行了赋值。如果编译器决定在 r5 处重用 r2 的值，那么 r2 和 r5 的值都是 0，r4 的值是 3.从编程人员的角度来看，p.x 的值从 0 变为 3 后又变回了 0.

尽管这样的行为让人颇感意外，但多数 JVM 实现都允许这种行为。然而，JLS 和 JVMS 中最初的 Java 内存模型却不允许这么做：这也是首批表明最初的 JMM 需要重新修订的迹象之一。

3 非正式语义

程序必须正确同步，以避免代码被重排序后能看到那些有违直觉的行为。使用了正确的同步并不能确保程序的所有行为都是正确的。但是，正确的同步确实可以让编码人员用简单的方式推断程序可能的行为；正确同步的程序的行为对可能的重排序依赖性更低。缺乏正确同步，就可能出现十分诡异、让人费解和有违直觉的行为。

理解一个程序是否被正确的同步了，有两个关键概念：

冲突访问（Conflicting Accesses） 对同一个共享字段或数组元素存在两个访问（读或写），且至少有一个访问是写操作，就称作有冲突。

Happens-Before 关系 两个动作（action）可以被 happens-before 关系排序。如果一个动作 happens-before 另一个动作，则第一个对第二个可见，且第一个排在第二个之前。必须强调的是，两个动作之间存在 happens-before 关系并不意味着这些动作在 Java 中必须以这种顺序发生。happens-before 关系主要用于强调两个有冲突的动

作之间的顺序，以及定义数据争用的发生时机。可以通过多种方式包含一个 happens-before 顺序，包括：

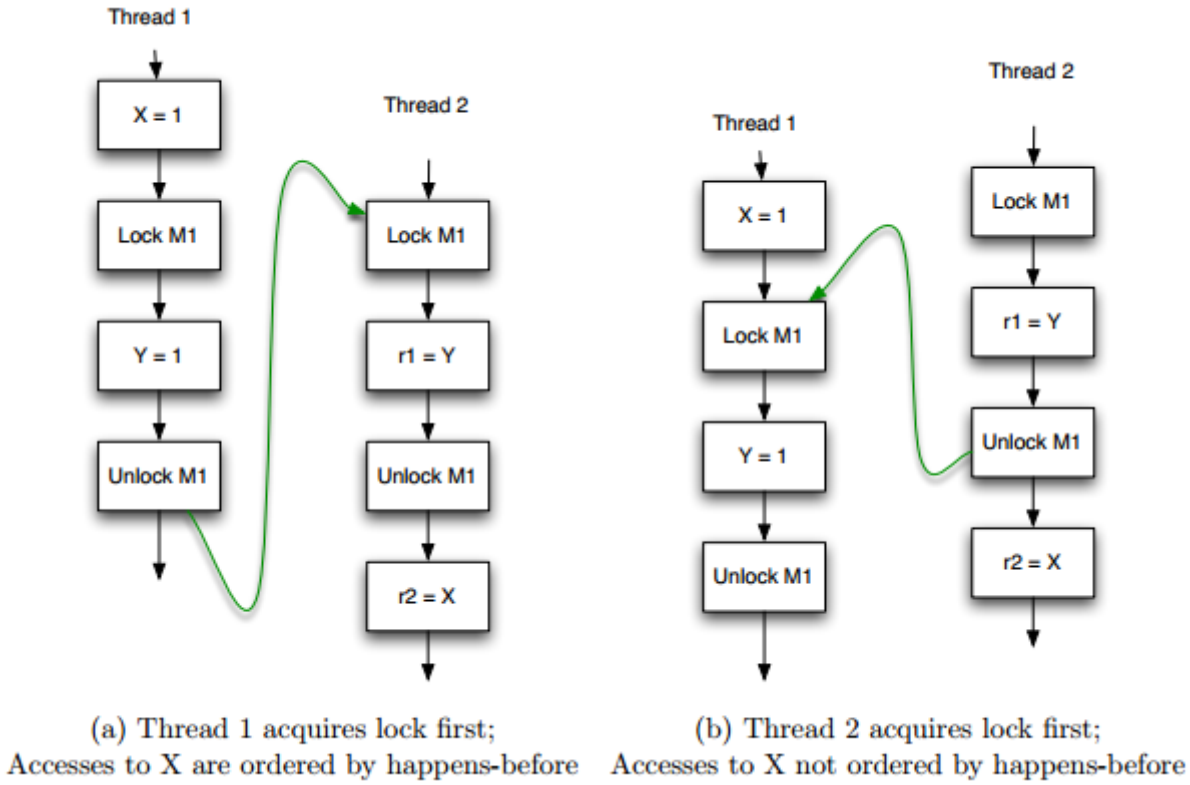
- 某个线程中的每个动作都 happens-before 该线程中该动作后面的动作。
- 某个管程上的 unlock 动作 happens-before 同一个管程上后续的 lock 动作。
- 对某个 volatile 字段的写操作 happens-before 每个后续对该 volatile 字段的读操作。
- 在某个线程对象上调用 start()方法 happens-before 该启动了的线程中的任意动作。
- 某个线程中的所有动作 happens-before 任意其它线程成功从该线程对象上的 join()中返回。
- 如果某个动作 a happens-before 动作 b，且 b happens-before 动作 c，则有 a happens-before c.

第 5 节有更全面的 happens-before 定义。

当程序包含两个没有被 happens-before 关系排序的冲突访问时，就称存在*数据争用*。正确同步的程序是没有数据争用的程序（3.1 节中有细微但重要的说明）。

未正确同步的代码的一个更微妙的例子见图 3，同一个程序的两次不同执行，对共享的 X 和 Y 的访问都有冲突。程序中的两个线程在管程 M1 上执行 lock 和 unlock 动作。在图 3a 的执行中，所有的两两冲突访问间都有 happens-before 关系。但

是，在图 3b 的执行中，对 X 的冲突访问之间没有 happens-before 顺序。正因如此，这个程序是未正确同步的。



(a) Thread 1 acquires lock first; Accesses to X are ordered by happens-before

(b) Thread 2 acquires lock first; Accesses to X not ordered by happens-before

图 3：happens-before 顺序

若一个程序正确同步了，该程序的所有执行轨迹看起来都像是顺序一致的。这对编程人员来说是一种强有力的保证。编程人员能够确定代码中是否包含数据争用，而无需担心重排序的影响。

3.1 顺序一致性（Sequential Consistency）

*顺序一致性*是程序执行过程中可见性和顺序的强有力保证。在顺序一致的执行过程中，所有动作（如读和写）间存在一个全序关系，与程序的顺序一致。

每个动作都是原子的且立即对所有线程可见。如果一个程序没有数据争用，那么该程序的执行看起来将是顺序一致的。如前面所提到的，在一组操作要保持原子性而未得到保证时，即使有顺序一致性 和/或 未遭遇数据争用，仍然可能会出现错误。

如果将顺序一致性作为内存模型，之前讨论的一些编译器和处理器优化将不再合法。例如，在图 2 中，只要将 3 赋值给 p.x，后续读取该位置就需要看到 3.

讨论完顺序一致性，可以用它为数据争用和未正确同步的程序提供一个重要说明。程序的某次执行中存在冲突的动作，且没有被同步排序，就会发生数据争用。一个程序是正确同步的当且仅当所有顺序一致的执行过程中都不存在数据争用。因此，编程人员只需推断顺序一致的执行过程来确定程序是否正确同步了。

4-6 节更加详细地讨论了非 final 字段的内存模型问题。

3.2 final 字段

声明为 final 的字段初始化一次后，在正常情况下它的值不会再改变。final 字段的详细语义与普通字段稍有不同。尤其是，编译器有很大的自由，能将对 final 字段的读操作移到同步屏障之外，然后调用任意或未知的方法。同样，也允许编译器将 final 字段的值保存到寄存器，在非 final 字段需要重新加载的那些地方，final 字段无需重新加载。

final 字段也允许编程人员在不需要同步的情况下实现线程安全的不可变对象。一个线程安全的不可变对象被所有线程都视为不可变的，即使不可变对象的引用在线程

间传递时存在数据争用。这提供了安全保证，可以防止不正确或恶意代码误用了不可变类。

final 字段必须正确使用才能保证不可变。当对象的构造器执行结束，就认为该对象是*完全初始化*了的。一个线程只有在看到某个对象引用之前，该对象就已经完全初始化了，才能保证这个线程能看到该对象正确初始化的 final 字段值。

final 字段的使用方式很简单。在对象的构造器里为该对象的 final 字段赋值。不要将正在创建过程中的对象引用写到一个其它线程可以看到的地方，这会让其它线程在构造器尚未执行结束时就能访问该对象。如果遵循这个规则，当该对象被其它线程看到时，这些线程总是能看到该对象 final 字段的正确值。且这些 final 字段所引用的任意对象或数组的中的内容，至少是和 final 字段一样的新（*译者注：final 字段所引用的对象里的字段或数组元素可能在后续还会变化，若没有正确同步，其它线程也许不能看到最新改变的值，但一定可以看到完全初始化的对象或数组被 final 字段引用的那个时刻的对象字段值或数组元素。*）。

图 4 中的例子展示了 final 字段和普通字段的差别。FinalFieldExample 类有一个 final int 字段 x 和一个非 final int 字段 y。一个线程可能会执行 writer()方法，另一个线程可能会执行 reader()方法。因为 writer()方法在对象构造器执行结束*之后*才给 f 赋值，所以可以保证 reader()方法能看到 f.x 正确初始化的值：即会读到 3.然而，f.y 不是 final 的；因此不能保证 reader()方法能看到 4.

```
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; // guaranteed to see 3
                        int j = f.y; // could see 0
                }
        }
}
```

图 4：final 字段语义示例

Thread 1	Thread 2

`Global.s = "/tmp/usr".substring(4);`	`String myS = Global.s;`
	`if (myS.equals("/tmp"))`
	`  System.out.println(myS);`

图 5：没有 final 字段或同步，这段代码可能会打印/usr

final 字段设计了是用来提供必要的安全保证的。考虑图 5 中的代码。String 对象是不可变的且字符串操作不需要使用同步。虽然 String 的实现中没有数据争用，但是，其它代码在使用 String 的时候可能会引发数据争用，且对于包含数据争用的程序，内存模型只提供了很弱的保证。尤其是，如果 String 类的字段不为 final，线程 2 最初看到该 String 对象 offset 字段的值可能为默认的 0，并且允许该 String 对象与"/tmp"的比较是相等的。在该 String 对象上的某个后续操作也许能看到正确的 offset 值 4，所以看到了该 String 对象的值是"/usr"。Java 的一些安全特性依赖于 String 对象真正的不可变，即便是恶意代码利用了数据争用在线程间传递 String 的引用。

这里仅是对 final 字段语义的一个概述。更详细的讨论，包括几种这里没有提到的情况，参阅第 9 节。

4 什么是内存模型

给定一个程序和该程序的一串执行轨迹，*内存模型*描述了该执行轨迹是否是该程序的一次合法执行。对于 Java，内存模型检查执行轨迹中的每次读操作，然后根据特定规则，检验该读操作观察到的写是否合法。

内存模型描述了某个程序的可能行为。JVM 实现可以自由地生成想要的代码，只要该程序所有最终执行产生的结果能通过内存模型进行预测。这为大量的代码转换提供了充分的自由，包括动作（action）的重排序以及非必要的同步移除。

内存模型的一个高级、非正式的概述显示其是一组规则，规定了一个线程的写操作何时会对另一个线程可见。通俗地说，读操作 r 通常能看到任何写操作 w 写入的值，意味着 w 不是在 r 之后发生，且 w 看起来没有被另一个写操作 w'覆盖掉（从 r 的角度看）。

在本内存模型规范中使用“读取（read）”这个词时，仅是指读取字段或数组元素的动作（action）。其它操作的语义，如读取数组的长度，执行受检转换以及虚方法调用，都不会被数据争用直接影响到。JVM 实现有责任确保数据争用不会导致诸如返回错误的数组长度或调用虚方法导致段错误这类不正确的行为。

内存语义决定着程序中每个时刻能读到的值。每个单个线程中的动作（action）必须表现为被该线程的语义所控制，不包括读操作看到的值由内存模型决定的情况。当指的是这种情景时，我们说该程序遵守*线程内（intra-thread）语义*。

5 定义

本节将为前面提到的一些非正式概念给出更详细的定义。

共享变量/堆内存（Shared variables/Heap memory） 能够在线程间共享的内存称作*共享*内存或*堆*内存。所有的实例字段，静态字段以及数组元素都存储在堆内存中。我们使用*变量*这个词来表示字段和数组元素。方法中的局部变量永远不会在线程间共享且不会被内存模型影响。

线程间的动作（Inter-thread Actions） 线程间的动作是由某一线程执行，能被另一线程探测或直接影响的动作（action）。线程间的动作包括共享变量的读写以及同步动作（synchronization action），如 lock 或 unlock 某个管程，读写某个 volatile 变量或启动一个线程。也包括与外部世界交互的动作（*外部*动作，*external* action），以及导致某个线程进入无限循环的动作（*线程分散*动作，*thread divergence* action）。有关这些动作的更多信息，参阅 7.1 节。

我们无需关心线程内的（intra-thread）动作（如，将两个局部变量相加并将结果存储到第三个局部变量中）。正如前文所言，每个单个线程需要遵守正确的线程内语义。

每个线程间的动作都与该动作的执行信息有关。所有的动作都与该动作发生所在的线程，以及在该线程中动作发生的程序顺序相关联。与一个动作相关的其它信息包括：

write	要写的变量以及要写的值。
read	要读的变量以及可见的写入值（由此，我们可以确定可见的值）。
lock	要锁定的管程。
unlock	要解锁的管程。

为方便起见，通常将线程间的动作简称为*动作（action）*。

程序顺序（Program Order） 在所有由线程 t 执行的的线程间动作中，t 的程序顺序是一个全序，反映出的是根据 t 的线程内语义，这些动作的执行顺序。

线程内语义（Intra-thread semantics） *线程内语义*是单线程程序的标准语义，基于某个线程内读动作能看到的值，可以完整的预测这个线程的行为。为了确定线程 t 中的动作在某次执行中是否合法，只需简单的看线程 t 的实现是否像是在单线程上下文中执行，如同 JLS 其它部分定义的那样。

每次线程 t 的执行都生成一个线程间的动作，该动作必须与 t 中在程序顺序上紧跟着该动作的线程间动作 a 相配。如果 a 是一个读操作，那么线程 t 后续就会使用到 a 看到的值，该值由内存模型确定。

简而言之，线程内语义决定着某个线程孤立的执行过程；当从堆中读取值时，值是由内存模型决定的。

同步动作（Synchronization Actions） 同步动作包括锁、解锁、读写 volatile 变量，用于启动线程的动作以及用于探测线程是否结束的动作。任何动作，只要是 synchronizes-with 边缘（edge）的起始或结束点，都是同步动作。这些动作会在后面讲到 *happens-before* 边缘的地方详讲。

同步顺序（Synchronization Order） 每个执行过程都有一个*同步顺序*。同步顺序是一次执行过程中的所有同步动作上的全序关系。

Happens-Before 与 Synchronizes-With 边缘（Happens-Before and Synchronizes-With Edges） 如果有两个动作 *x* 和 *y*，$x \xrightarrow{hb} y$ 表示 *x happens-before y*。如果 x 和 y 是同一个线程中的动作，且在程序顺序上 *x* 在 *y* 之前，那么有 $x \xrightarrow{hb} y$。

同步动作也包括 happens-before 边缘。我们称结果导向的边缘（resulting directed edges）为 *synchronized-with* 边缘。它们的定义如下：

- 某个管程 *m* 上的解锁动作 synchronizes-with 所有后续在 *m* 上的锁定动作（这里的*后续*是根据同步顺序定义的）。
- 对 volatile 变量 *v* 的写操作 synchronizes-with 所有后续任意线程对 *v* 的读操作（这里的*后续*是根据同步顺序定义的）。
- 用于启动一个线程的动作 synchronizes-with 该新启动线程中的第一个动作。
- 线程 T1 的最后一个动作 synchronizes-with 线程 T2 中任一用于探测 T1 是否终止的动作。T2 可能通过调用 T1.isAlive()或者在 T1 上执行一个 join 动作来达到这个目的。

- 如果线程 T1 中断了线程 T2，T1 的中断操作 synchronizes-with 任意时刻任何其它线程（包括 T2）用于确定 T2 是否被中断的操作。这可以通过抛出一个 InterruptedException 或调用 Thread.interrupted 与 Thread.isInterrupted 来实现。

- 为每个变量写默认值（0，false 或 null）的动作 synchronizes-with 每个线程中的第一个动作。

  虽然在对象分配之前就为该对象中的变量写入默认值看起来有些奇怪，从概念上看，程序启动创建对象时都带有默认的初始值。因此，任何对象的默认初始化操作 happens-before 程序中的任意其它动作（除了写默认值的操作）。

- 调用对象的终结方法时，会隐式的读取该对象的引用。从一个对象的构造器末尾到该引用的读取之间存在一个 happens-before 边缘。注意，该对象的所有冻结操作（见 9.2 节）happen-before 前面那个 happens-before 边缘的起始点。

如果动作 x synchronizes-with 后面的动作 y，我们也能得到 $x \xrightarrow{hb} y$。此外，happens-before 是传递闭包的。换而言之，如果 $x \xrightarrow{hb} y$，且 $y \xrightarrow{hb} z$，那么可以得到 $x \xrightarrow{hb} z$。

应当注意的是，两个动作之间存在 happens-before 关系并不一定意味着在实现中它们必须以这种顺序发生。如果重排序产生的结果与合法执行的结果一致，那么，重排序就不是非法的。例如，某个线程中写默认值到一个对象的每个字段，不需要发生在该线程开始之前，只要没有被读操作察觉到即可。

更具体地说，如果两个动作间有 happens-before 关系，在不存在 happens-before 关系的代码中，这两个动作的发生就无需表现出这样的顺序。如，一个线程中的写操作，当与另一个线程中的读操作存在数据争用时，另一个线程中的读操作表现出的顺序可能就是错乱的。

Object 类的 wait 方法关联了锁和解锁动作；它们的 happens-before 关系是由这些关联的动作所定义的。这些方法在第 14 节会有进一步的说明。

6 Java 内存模型的近似模型

3.1 节中，我们描述了顺序一致性。它太严格了，不适合做 Java 内存模型，因为它禁止了标准的编译器和处理器优化。这一节将回顾下顺序一致性，然后展示另一个模型，称作 happens-before 内存模型。这个模型已经非常接近 Java 内存模型的需求，但是，它太弱了；其允许违反因果关系这种不可接受的事情发生。6.3 节描述了因果关系相关的问题。

第 7 节中展示了 Java 内存模型，它是一个正式的模型，对 happens-before 内存模型作了加强，以提供对因果关系的充分保证。

6.1 顺序一致的内存模型

正式地，在顺序一致性里，所有动作以全序（执行顺序）的顺序发生，与程序顺序一致；而且，每个对变量 v 的读操作 *r* 都将看到写操作 *w* 写入 v 的值，只要：

- 执行顺序上 *w* 在 *r* 之前，且
- 执行顺序上不存在这样一个 *w′*，*w* 在 *w′*之前且 *w′*在 *r* 之前。

6.2 Happens-Before 内存模型

在展示完整的 Java 内存模型之前，先来展示一个更简单的内存模型，称作 *happens-before 内存模型*。

这个模型包含几个属性/需求：

- 在所有的同步动作上都有一个全序关系，即同步顺序。该顺序与程序顺序以及锁的互斥一致。
- 同步动作包括相配对的动作间的 synchronizes-with 边缘，如第 5 节所描述。
- synchronizes-with 边缘的传递闭包与程序顺序产生了 happens-before 顺序，如第 5 节所描述。
- 某个非 volatile 读操作能看到的值由 happens-before 一致性规则决定。
- 某个 volatile 读操作能看到的值由同步顺序一致性规则决定。

Happens-before 一致性说的是，在 happens-before 偏序执行轨迹中，若满足下列条件，*允许*对变量 v 的读操作 r 看到写操作 w 写入 v 的值：

- r 没有排在 w 前面（亦即，不是 $r \xrightarrow{hb} w$ 这种情况），且
- 没有一个介入的对 v 的写操作 w'（亦即，不存在这样一个对 v 的写操作 w'：$w \xrightarrow{hb} w' \xrightarrow{hb} r$）。

同步顺序一致性说的是，每个对 volatile 变量 v 的读操作 r 都返回的是，同步顺序上读操作之前最后写入 v 的值。

例如，图 1 中表现的行为在 happens-before 内存模型中是允许的。线程间没有 synchronizes-with 或 happens-before 边缘，允许每个读操作看到其它线程写入的值。

6.3 因果关系

happens-before 内存模型描绘了一个必要而非充分的约束集。所有 Java 内存模型允许的行为，happens-before 内存模型也允许，但是，happens-before 内存模型允许*不可接受的*行为 —— 这些行为违反了我们建立的需求。可以将它们都看作是以一种不可接受的方式违背了因果关系。本节中，将谈到几个 happens-before 模型不符合要求的地方，以及如何调整它们以使其能提供必要的保证。

6.3.1 Happens-Before 太弱了

Happens-before 模型最致命的弱点是其允许值“凭空出现（out of thin air）”。详细内容可以在图 6 中看到。

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 1;	x = 1;

正确同步的，所以 r1 == r2 == 0 是唯一合法的行为

图 6：happens-before 允许的违规行为

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

未正确同步的，但决不允许 r1 == r2 == 42

图 7：不可接受的违反因果关系的行为

图 6 中的代码是正确同步的。这看起来有点奇怪，因为里面没有执行任何同步动作。但是，记住，如果程序以顺序一致的方式执行时，没有数据争用，程序就是正确同步的。如果这个代码以顺序一致的方式执行，每个动作都将按程序顺序发生，然后两个写操作就都不会发生了。既然没有写操作，就没有数据争用：该程序就是正确同步的。

因为该程序是正确同步的，所以唯一允许的行为就是顺序一致的行为（所谓“顺序一致的行为”，意思是，任意行为，从外部看来，都与一次顺序一致地执行的结果相同）。然而，在 happens-before 内存模型下，存在执行结果是 r1 == r2 == 1 的情况，因为它里面没有 synchronizes-with 或 happens-before 边缘，允许每个读操作看到其它线程写的值。

当一个写操作发生在了一个其依赖的读操作之前，我们将这样的问题称为因果关系，因为它涉及写操作是否会触发自身发生的问题。图 6 中，读操作促使写操作发生，然后写操作使得读操作能看到它们都看到的值。这里的动作没有“第一原因（first cause）”。因此，我们的内存模型需要一个可接受以及一致的方式来确定哪些明显的因果循环是允许的。

即使程序未正确同步，也有某些违反因果关系的不可接受的行为。图 7 中有个这样的例子；happens-before 内存模型允许 r1 == r2 == 42 这样的行为，但是，是不可接受的。

6.3.2 难以捉摸的因果关系

因果关系的概念并不简单，因为不容易确定哪个动作引起了其它动作的发生。Java 内存模型允许图 8 中的行为，即使该例子看似也存在循环因果关系。必须允许这样的行为，因为编译器能够：

- 消除多余的读取 a 的操作，将 r2 = a 替换为 r2 = r1，然后，
- 确定了表达式 r1 == r2 总是为 true，消除条件分支 3，然后最终
- 将 4: b = 2 移到前面。

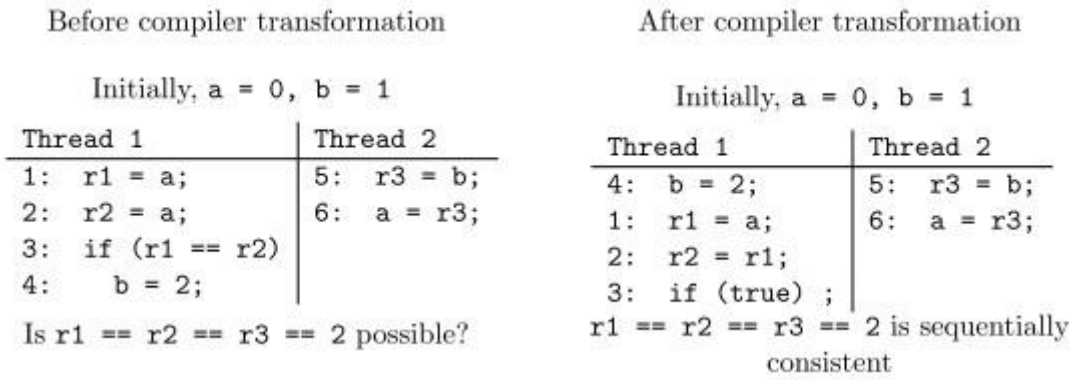


图 8：消除多余读操作的结果

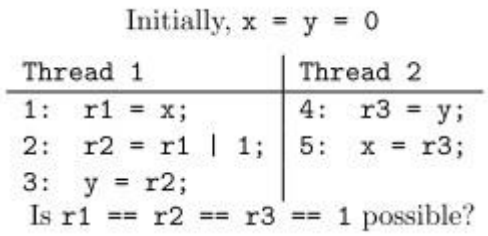


图 9：使用全局分析（Global Analysis）

编译器消除多余的读操作后，4: b = 2 中的赋值就一定会发生；第二次读取 a 总会返回跟第一次一样的结果。这样，简单的编译器优化就能导致明显的因果循环。因此，读取 a 与写入 b 之间必须没有因果关系，尽管粗略检查该代码可能显示存在一

个因果关系。注意，线程内语义和最低安全保障（out-of-thin-air safety）会保证如果 r1 ≠ r2，线程 1 不会对 b 执行写操作，然后就有 r3 == 1。

图 9 展示了另一个例子，这里应该允许编译器执行全局分析（global analysis），消除乍看之下的那个因果关系。要使 r1 == r2 == r3 == 1，线程 1 看似需要在读取 x 之前将 1 写入 y。然而，编译器可以使用跨线程分析（interthread analysis）来确定 x 和 y 的值仅可能是 0 和 1.知道了这个，编译器就能确定 r2 永远是 1.这样，将 1 写入 y 就不会受读取 x 的影响，且写操作可以提前执行。

6.3.3 分析因果关系的途径

Java 内存模型将一个特定的执行过程和一个程序作为输入，然后确定该执行过程是否是该程序的一次合法执行。它是通过逐步地建立一组“提交的”动作来实现的，这些动作反映出了我们知道的哪些动作能够被程序执行而不需要一个“因果循环”。通常，下一个将要提交的动作表示的是能被顺序一致的执行过程执行的下一个动作。然而，为了表明读操作能看到程序顺序里后面其它线程写的值，我们允许一些动作比更早发生的其它动作先提交。

显然，有些动作可以提前提交，有些则不能。例如，如果图 6 中的写操作之一在读取被写变量之前提交，读操作将看到写入的值，然后就会出现凭空产生的结果。通俗地讲，如果我们知道某个动作的发生不会产生数据争用，就允许该动作提前提交。图 6 中，两个写操作都不可以提前执行，因为，除非是让读操作看到数据争用的结果，否则那个写操作就不能发生。另一方面，在图 8 中，不管是否有数据争用发生，对 b 的写操作都会发生。

7 Java 内存模型的正式规范

本节规定了 Java 内存模型的正式规范（不包括 final 字段，final 字段会在第 9 节中讲）。

7.1 动作与执行过程（Actions and Executions）

动作 *a* 是通过元组<*t, k, v, u*>来描述的，其中：

t - 执行该动作的线程

k - 动作的类型：volatile read，volatile write，（非 volatile）read，（非 volatile）写，lock，unlock，特殊的同步动作，外部动作以及线程分散（thread divergence）动作。volatile read，volatile write，lock 和 unlock 都是同步动作，特殊的同步动作有诸如启动一个线程，线程中由编译器生成的第一个或最后一个动作，以及探测某个线程是否终止的动作，如第 5 节描述的那样。

v - 动作中涉及到的变量或管程

u - 该动作的任意一个唯一标识符

执行过程 *E* 通过下面元组描述

$$<P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow}>$$

其中：

P - 一个程序

A - 一组动作

$\xrightarrow{po}$ - 程序顺序，对每个线程 *t* 来说，程序顺序是 *A* 中由 *t* 执行的所有动作上的全序关系。

$\xrightarrow{so}$ - 同步顺序，是 *A* 中所有同步动作上的全序关系。

W - 一个写可见（write-seen）函数，对于 *A* 中的每个读操作 *r*，*W(r)*表示 *E* 中对 *r* 可见的写操作。

V - 一个值写入（value-written）函数，对于 *A* 中的每个写操作 *w*，*V(w)*表示 *E* 中 *w* 写入的值。

$\xrightarrow{sw}$ - synchronizes-with，同步动作上的偏序关系。

$\xrightarrow{hb}$ - happens-before，动作上的偏序关系。

注意，synchronizes-with 和 happens-before 是由执行过程中的其它部分以及良构的执行过程的规则唯一确定的。

有两个动作类型需要特别说明，更多细节见 7.5 节。引入这两个动作是为了解释为什么这样一个线程会导致所有其它线程停止不前。

外部动作 - 外部动作是一个能在执行过程外部观察得到的动作，且可能有一个基于执行过程外部环境的结果。外部动作元组包含一个额外的组成部分，即，执行该动作的线程能够看到的外部动作的结果。这可能是关于该动作成功或失败的信息，以及该动作读取的任何值。

外部动作的参数（如，哪个字节写到哪个套接字）不是外部动作元组的一部分。这些参数由该线程中的其它动作构建，是可以通过检查线程内语义来确定的。内存模型中不会对它们进行明确讨论。

在不能终止的执行过程中，并不是所有的外部动作都能看得见。不能终止的执行过程以及可观察的动作会在 7.5 节中讨论。

线程分散（thread divergence）动作 - 线程分散动作只会被这样的线程执行，该线程处在一个无限循环中，循环里没有内存、同步或外部动作的执行。如果某个线程执行一个线程分散动作，程序顺序上，该动作紧接着的是其它线程分散动作的无限序列。

7.2 定义

1. **synchronizes-with 的定义。**第 5 节中定义了 synchronizes-with 边缘。synchronizes-with 边缘的起点称作 *release*，终点称为 *acquire*。

2. **happens-before 的定义。**happens-before 顺序是通过 synchronizes-with 和程序顺序这两个顺序的传递闭包给出的。在第 5 节中有详细讨论。

3. **充分的同步边缘（suffcient synchronization edges）的定义。**如果某一组同步边缘是这样一个最小集，利用该最小集中的边缘和程序顺序边缘上的传递闭包就可以确定执行过程中的所有 happens-before 边缘，则这一组同步边缘是*充分的*。这样的同步边缘组是唯一的。

4.**偏序和函数的限制（ Restrictions of partial orders and functions）。**我们用 $f|_d$ 表示将 f 的定义域限制为 d 而给出的函数：对所有的 $x \in d$，有 $f(x) = f|_d(x)$ 且对所有的 $x \notin d$，$f|_d(x)$ 是未定义的。同样地，我们使用 $\xrightarrow{e}|_d$ 表示 d 中元素的偏序关系 $\xrightarrow{e}$ 的限制：对所有的 $x, y \in d$，当且仅当 $x \xrightarrow{e}|_d y$ 时有 $x \xrightarrow{e} y$。如果 $x \notin d$ 或 $y \notin d$，将不存在 $x \xrightarrow{e}|_d y$。

7.3 良构的（ Well-Formed）执行过程

我们只考虑良构的执行过程。当下列条件为 true 时，执行过程 $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ 就是良构的：

1.**每个对变量 *x* 的读都能看到一个对 *x* 的写。所有对 volatile 变量的读写都是 volatile 动作。**对于所有的读操作 $r \in \mathcal{A}$，有 $\mathcal{W}(r) \in \mathcal{A}$ 且 $\mathcal{W}(r).v = r.v$。当且仅当 r 是 volatile 读时，变量 $r.v$ 才是 volatile 的，当且仅当 w 是 volatile 写时变量 $w.v$ 才是 volatile 的。

2.**同步顺序与程序顺序以及互斥是一致的。**同步顺序与程序顺序是一致的意味着 happens-before 顺序（通过 synchronizes-with 边缘与程序顺序的传递闭包得出）是一种合法的偏序关系：自反的、传递的以及反对称的。同步顺序与互斥是一致的意味着在每个管程上，lock 和 unlock 动作是正确嵌套的。

3.**线程的运行遵守线程内（intra-thread）一致性。**对于每个线程 t，在 $\mathcal{A}$ 中执行的动作与该线程单独以程序顺序生成的动作是一样的，对于每个写入值 $\mathcal{V}(w)$ 的写操作 w，每个读操作 r 就会看到返回的值 $\mathcal{V}(\mathcal{W}(r))$。每个

线程所看到的值由内存模型确定。程序顺序必须反映出根据 P 的线程内语义，动作将会执行的程序顺序。

4.**线程的运行遵守同步顺序一致性。**对于所有的 volatile 读操作 $r \in \mathcal{A}$。并不存在 $r \overset{so}{\rightarrow} W(r)$。此外，必须不存在满足 $w.v = r.v$ 且 $W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r$ 的写操作 w。

5.**线程的运行遵守 happens-before 一致性。**对于所有的读操作 $r \in \mathcal{A}$。并不存在 $r \overset{hb}{\rightarrow} W(r)$。此外，必须不存在满足 $w.v = r.v$ 且 $W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r$ 的写操作 w。

7.4 执行过程的因果（Causality）要求

良构的执行过程 $E = <P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow}>$ 是通过从 $\mathcal{A}$ 中提交（committing）动作来验证的。如果 $\mathcal{A}$ 中所有的动作都能提交，那么该执行过程就满足了 Java 内存模型的因果要求。从一个空的集合 C_o 开始，我们执行一系列的步骤，从 $\mathcal{A}$ 中拿出动作，然后加到一个已提交的动作集 C_i 来获得一个新的已提交动作集 C_{i+1}。为了证明这是合理的，对于每个 C_i，我们需要证明一个包含 C_i 的执行过程 $\mathcal{E}_i$ 能满足一定条件。

正式地，当且仅当满足下列条件时，执行过程 $\mathcal{E}$ 就满足了 Java 内存模型的因果要求：

- 一组动作 $C_o, C_1, ...$ 使得
  - $C_o = \varnothing$

– $C_i \subset C_{i+1}$

– $\mathcal{A} = \cup(C_0, C_1, C_2, ...)$

使得 $\mathcal{E}$ 和$(C_0, C_1, C_2, ...)$满足下列约束。

序列 $C_0, C_1, ...$可能是有穷的，以 $C_n = \mathcal{A}$ 结束。然而，如果 $\mathcal{A}$ 是无穷的，那么，序列 $C_0, C_1, ...$可能也是无穷的，且该无穷序列中的所有元素的并集必须等于 $\mathcal{A}$。

- 良构的执行过程 $E_1, ...,$有 $E_i = <P_i, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow}>$.

给定这些动作集 $C_0, ...$以及执行过程 $\mathcal{E}_1, ...$，C_i 中的每个动作必须是 $\mathcal{E}_i$ 中的动作之一。C_i 中的所有动作必须共享 $\mathcal{E}_i$ 和 $\mathcal{E}$ 中相同的相关 happens-before 顺序以及同步顺序。正式地，

1. $C_i \subseteq \mathcal{A}_i$

2. $\overset{hb_i}{\rightarrow}|C_i = \overset{hb}{\rightarrow}|C_i$

3. $\overset{so_i}{\rightarrow}|C_i = \overset{so}{\rightarrow}|C_i$

C_i中写操作写入的值在 $\mathcal{E}_i$ 和 $\mathcal{E}$ 中必须是相同的。C_{i-1} 中的读操作需要看到 $\mathcal{E}_i$ 和 $\mathcal{E}$ 中相同的写操作写入的值（但不是 C_i - C_{i-1}中的读操作），正式地，

4. $\mathcal{V}_i|c_i = \mathcal{V}|c_i$

5. $\mathcal{W}_i|c_{i-1} = \mathcal{W}|c_{i-1}$

在 $\mathcal{E}_i$ 中但不在 C_{i-1} 中的所有读操作必须要看到在该读操作之前的写。C_i - C_{i-1} 中的每个读操作 r 必须要看到 $\mathcal{E}_i$ 和 $\mathcal{E}$ 中 C_{i-1} 里的写，但所看到的 $\mathcal{E}_i$ 中的写可能与 $\mathcal{E}$ 中的不同。正式地，

6.对于任一读操作 $r \in \mathcal{A}_i$ - C_{i-1}，有 $\mathcal{W}_i(r) \overset{hb_i}{\rightarrow} r$

7.对于任一读操作 $r \in C_i$ - C_{i-1}，有 $\mathcal{W}_i(r) \in C_{i-1}$ 且 $\mathcal{W}(r) \in C_{i-1}$

给 $\mathcal{E}_i$ 一组充分 synchronizes-with 边缘，如果有 release-acquire 对 happens-before C_i - C_{i-1} 中的一个动作，那么，该 release-acquire 对必须出现在所有的 $\mathcal{E}_j$ 中，这里 $j \geq i$。正式地，

8.设 $\overset{ssw_i}{\rightarrow}$ 为位于 $\overset{hb_i}{\rightarrow}$ 而非 $\overset{po_i}{\rightarrow}$ 传递规约（transitive reduction）中的 $\overset{sw_i}{\rightarrow}$ 边缘。我们称 $\overset{ssw_i}{\rightarrow}$ 为 $\mathcal{E}_i$ 的充分 synchronizes-with 边缘。如果 $x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z$ 且 $z \in C_i$ - C_{i-1}，那么对于所有的 $j \geq i$ 有 $x \overset{sw_i}{\rightarrow} y$。

如果动作 y 被提交，所有在 y 之前发生的外部动作也被提交了。

9.如果 $y \in C_i$，x 是一个外部动作，且 $x \overset{hb_i}{\rightarrow} y$，那么 $x \in C_i$。

7.5 可观察的行为与不会终止的执行过程

对于在某一有限时间内总是能执行结束的程序，依照他们容许的执行过程，其行为简单易懂。对于不能在有限时间内终止的程序，会有很多微妙的问题。

一个程序可观察的行为是通过该程序可能执行的外部动作的有限集合来定义的，例如，某个程序简单地一直打印“Hello”，可以由一组行为来描述，这组行为中对于任一非负整数 *i*，包含了打印“Hello” *i* 次的行为。

终止，并不显式地建模为一种行为，但是一个程序可以很容易地扩展以生成一个额外的外部动作“executionTermination”，这个动作在所有线程终止的时候发生。

我们还定义了一个特殊的 *hang* 动作。如果某个行为由一组包含了一个 *hang* 动作的外部动作描述，这表明，在（非 hang）外部动作之后的某个行为被观察到了，这个程序在没有执行任意额外的外部动作或终止，就可以运行无限的时间。下列情形会导致程序挂起（hang）：

- 如果所有未终止的线程都被阻塞，且至少存在这样一个阻塞的线程，或
- 如果该程序在没有执行任意外部动作时可以执行无限次动作。

一个线程可以在很多情况下被阻塞，如试图获取一个锁，或执行一个依赖于外部数据的外部动作（如读操作）。如果某个线程处于这样的状态，Thread.getState 将返回 BLOCKED 或 WAITING.一个执行过程可能会导致线程一直阻塞，然后该执行过程得不到终止。这种情况下，被阻塞线程生成的动作必须是由该线程生成的，到引起该线程一直阻塞的动作（包括引起阻塞的动作）为止的所有动作组成，但不包含该线程生成的在这个动作之后的动作。

为推断可观察的行为，我们需要说一说可观察的动作集。如果 O 是 $\mathcal{E}$ 中一个可观察的动作集，那么，即使 $\mathcal{A}$ 包含了无限数量的动作，O 也必须是 $\mathcal{A}$ 的一个子集，

且必须只包含有限数量的动作。而且，如果动作 $y \in O$，且有 $x \xrightarrow{hb} y$ 或 $x \xrightarrow{so} y$，那么 $x \in O$。

注意，一个可观察的动作集不局限于只包含外部动作。而是，仅存在于可观察的动作集中的外部动作才被认为是可观察的外部动作。

当且仅当行为 $\mathcal{B}$ 是一个外部动作的有限集合且满足下列条件之一时，$\mathcal{B}$ 就是程序 $\mathcal{P}$ 中一个允许的行为：

- $\mathcal{P}$ 中存在一个执行过程 $\mathcal{E}$，以及 $\mathcal{E}$ 中有个可观察的动作集 O，

  且 $\mathcal{B}$ 是 O 中的外部动作集（如果 $\mathcal{E}$ 中的任一线程以阻塞状态告终，

  且 O 包含了 $\mathcal{E}$ 中的所有动作，那么 $\mathcal{B}$ 可能也包含了一个 *hang* 动作），或者

- 存在一个动作集 O，使得

  - $\mathcal{B}$ 由 *hang* 动作加 O 中所有的外部动作组成，且

  - 对于所有的 $\mathcal{K} \geq |O|$，$\mathcal{P}$ 存在一个执行过程 $\mathcal{E}$ 和 一组动作 O'，使得：

    - O 和 O' 都是 $\mathcal{A}$ 的子集，且满足可观察动作集的条件
    - $O \subseteq O' \subseteq \mathcal{A}$
    - O' - O 中不包含外部动作

注意，行为 $\mathcal{B}$ 并不描述 $\mathcal{B}$ 中外部动作被观察到的顺序，但是，外部动作的生成以及执行过程上的其他（隐式或未说明的）限制可能会施加这样的约束。

8 经典测试用例与行为

本节中，罗列了几个例子，他们的行为或是 Java 内存模型允许的，或是禁止的。大部分例子要么是违背了因果性被禁止，要么是看起来违背了因果性，但是是由于符合标准的编译器优化造成的，而实际上是允许的。

图 10 中的例子展示了内存模型的工作方式。注意，开始时对 x 和 y 写入了默认值。我们希望得到的结果是 r1 == r2 == 1，如果编译器重排序了线程 1 中的语句，那么就能得到希望的结果。

Initially, x = y = 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = 1;	x = r2;

r1 == r2 == 1 是合法的

图 10：符合要求的重排序

Action	Final Value	First Committed In	First Sees Final Value In
x = 0	0	C_1	E_1
y = 0	0	C_1	E_1
y = 1	1	C_1	E_1
r2 = y	1	C_2	E_3
x = r2	1	C_3	E_3
r1 = x	1	C_4	E

图 11：图 10 的提交集表

动作集 C_o 是个空集，且不存在执行过程 $\mathcal{E}_o$。

因此，按照规则 6，执行过程 $\mathcal{E}_1$ 中的所有读操作都能看到 happens-before 这些读操作的写操作。对于此例，在 $\mathcal{E}_1$ 中两个读操作都必须看到值 0。我们首先

给 x 和 y 提交了写入 0 的操作 ，以及在线程 1 中给 y 提交了写入 1 的操作；这些写操作都在 C_1 中。

我们希望添加 r2 = y。C_1 不可能包含这个动作，不管它能看到什么写操作：两个对 y 的写操作都还没提交。C_2 可以包含该动作；但是，因为规则 6，在 $\mathcal{E}_2$ 中对 y 的读操作必须返回 0.因此，执行过程 $\mathcal{E}_2$ 等同于 $\mathcal{E}_1$。

在 $\mathcal{E}_3$ 中，happens-before 一致性使得 r2 = y 能够看到 C_2 中任一冲突写（通过规则 7，以及 happens-before 一致性规则）。这个读操作能看到的写是线程 1 中将 1 写入 y 的写操作，该写操作是在 C_1 里提交。我们在 C_3 里提交了一个额外的动作：通过 x = r2 这句将 1 写入 x.

C_4 包含了 r1 = x，但在 $\mathcal{E}_4$ 中，因为规则 6，它看到的仍然是 0.然而，在最后的执行过程 $\mathcal{E}$ 中，规则 7 允许 r1 = x 看到将 1 写入 x 的写操作，该写操作是在 C_3 里提交的。

图 11 中的表格展示了给定的动作的提交时间。

8.1 内存模型允许的怪异行为

图 12 展示了一个小而有趣的例子。出现 r1 == 2 且 r2 == 1 是合法的行为，尽管不太容易看出这是怎么发生的。某个编译器可能不会对各个线程中的语句重排序；该代码必然不会致使 r1 == 1 或 r2 == 2.然而，r1 == 2 且 r2 == 1 是处理器架构允许的，处理器提前执行写操作，但写操作不会对在程序顺序上先于该写操作的本地读

操作可见。这种行为，虽然怪异，但却是 Java 内存模型所允许的。为了在内存模型下获得这样的结果，就要先将两个写操作提交，然后提交两个读操作。

Initially, x = 0

Thread 1	Thread 2
r1 = x;	r2 = x;
x = 1;	x = 2;

r1 == 2 和 r2 == 1 是合法的

图 12：意外的重排序

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

会发生 r1 == r2 == r3 == 2 吗

图 13：消除冗余读的结果

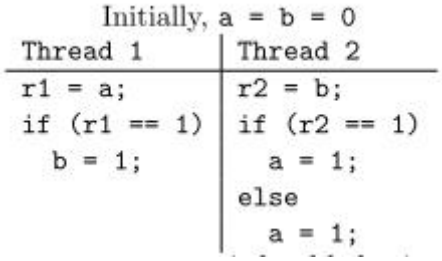
如 6.3.2 中所讨论的，图 13 中简单的编译器优化所表现的行为是允许的。为了在内存模型下取得这样的结果，提交的动作如下：

C_1 中，默认写以及 b == 2 的写操作

C_2 中，r3 == b 的读操作

C_3 中，a == r3 的写操作，写入了值 2

C_4 中，两个读 a 的操作



r1 == r2 == 1 是合法的

图 14：写操作可以提前执行

图 14 是又一个特别的例子。这种行为看起来不太可能，因为线程 2 在读取 b 之前无法确定它将执行哪条语句。但是，r2 的值为 1 这个结果又表明写操作 a = 1 是在读操作 r1 = a 之前执行的。

这种行为是由编译器检测到每个执行过程都会执行到 a = 1 导致的。如此一来，这个动作就可以提前执行，即便提前并不知道这个动作的发生是由哪个分支的语句引起的。这就使得线程 1 中 r1 看到了 1，b 被执行了写操作，然后线程 2 看到 b == 1，但是对 a 的写操作已经不是发生在原来的地方了。内存模型通过先提交对 a 的写操作取得了我们正在讨论的这个结果。

图 15 也在 6.3.2 节中讨论过。为了取得 r1 == r2 == r3 == 1 这样的结果，某编译器使用跨线程分析，确定了 x 和 y 的值只可能是 0 或 1.然后它就提前执行对 y 的写操作。内存模型验证该执行过程的方式与验证图 10 的执行过程完全相同。因为不管该程序读得 x 的值是 0 还是 1，y 的结果都是一样，所以，允许在读取 x 之前提交这个写操作。

Initially, x == y == 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

r1 == r2 == r3 == 1 是合法的

图 15：编译器可以仔细分析出何时动作会一定发生

再看图 16 中的代码。某个编译器可以确定赋给 x 的值只能是 0 或 42.于是，该编译器可以推断，在执行 r1 = x 的点上，要么刚执行过将 42 写入 x 的操作，要么刚读取了 x 且看到值为 42.任何一种情况，读取 x 看到值 42 都是合法的。随后就将 r1 = x 修改为 r1 = 42；这将允许 y = r1 转换为 y = 42，然后提前执行，就导致了我们正在讨论的这种行为。这种情况下，对 y 的写操作先被提交。

Initially, x == y == z == 0

Thread 1	Thread 2
r3 = x;	r2 = y;
if (r3 == 0)	x = r2;
x = 42;	
r1 = x;	
y = r1;	

r1 == r2 == r3 == 42 是合法的

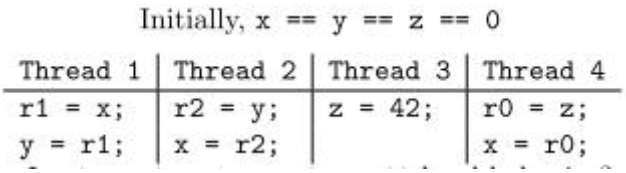
图 16：一个复杂的推断

8.2 内存模型禁止的行为

图 17 和 18 中的例子类似于图 7 中的例子，只有一个重要区别。图 7 中，在任何顺序一致的执行过程中，42 永远也不会被写到 x.图 17 和 18 的例子中，42 只是有时候会被写到 x。即使线程 4 没有写入 42，线程 1 和 2 看到值 42 是合法的吗？

注意，这两个例子的主要区别在于，在图 17 中，线程 1 和 2 中的写操作依赖于读操作的数据流，图 18 中，写操作依赖于读操作的控制流。为此，Java 内存模型没有在控制和数据依赖间加以区分。

这是一个潜在的安全问题；如果 42 代表由线程 4 控制的某个对象的引用，但在线程 4 还没有看到 z 的值为 1 时，不想线程 1 和 2 看到 42，那么，线程 1 和线程 2 可以说是凭空造出了该引用。

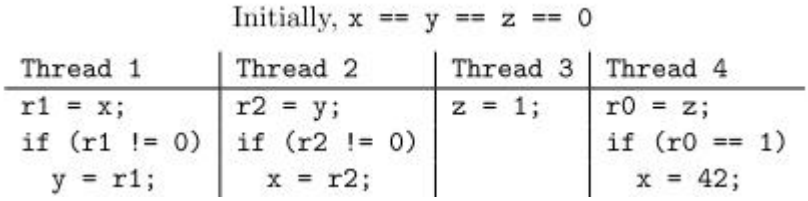


Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 42;	r0 = z;
y = r1;	x = r2;		x = r0;

r0 == 0，r1 == r2 == 42 是合法的吗？

图 17：如果线程 4 没有写入 42，线程 1 和 2 能看到 42 吗？



Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 1;	r0 = z;
if (r1 != 0)	if (r2 != 0)		if (r0 == 1)
y = r1;	x = r2;		x = 42;

r0 == 0，r1 == r2 == 42 是合法的吗？

图 18：如果线程 4 没有写入 x，线程 1 和 2 能看到 42 吗？

虽然允许这些行为所导致的安全影响是不确定的，但 JLS 本着尽可能安全、简单以及不奇怪的语义规则，Java 内存模型禁止图 17 和 18 中的行为。

9 final 字段的语义

3.2 节中有对 final 字段的简要讨论。这样的字段只初始化一次且不再改变。这种语义既允许编译器在读取这类字段时进行激进优化，还可以用来保证在不需要同步时不可变对象的线程安全。

9.1 final 字段语义的目标与要求

final 字段语义是基于几个互有矛盾的目标的：

- final 字段的值不会变化。编译器不应该因为获得了一个锁，读取了一个 volatile 变量或调用了一个未知方法，而重新加载一个 final 字段。事实上，编译器可以将线程 *t* 中对对象 *X*final 字段 *f* 的读取提前到紧跟在读取对象 *X* 的引用之后；该线程永远也不需要重新加载那个字段。

- 一个对象，仅包含 final 字段且在构建期间没有对其他线程可见，应当视作不可变的，即使这类对象的引用在线程间传递时存在数据争用。

  - 在构建对象 *X* 时将其引用存储到堆上并不违背这个要求。例如，同步可以确保没有其他线程能够在 *X* 构建期间加载到 *X* 的引用。或者，在 *X* 构造期间，*X* 的引用可能会存进另一个对象 *Y* 中；如果在 *X* 构建完成之前，*Y* 的引用不对其他线程可见，那么，final 字段的保证仍然成立。

- 将字段 *f* 设为 final，在读取 *f* 时应当利用最小的编译器/架构代价。

- 该语义必须支持诸如反序列化的场景，在这种情况下，一个对象的 final 字段会在该对象构建完成后改变。

final 字段语义为确定哪些写操作 happens-before 一个读操作提供了补充和备选规则。对于一个 final 实例字段，如果读取该字段的线程并非创建该字段的线程，仅由这些规则确定的顺序适用。对于非 final 字段，这些规则补充了常规规则。

如果一个对象在初始构建时，该对象的引用与其他线程共享了，那么该对象的 final 字段的大部分保证都会失效；这包括程序的其他部分继续使用该字段的原始值的场景。

9.1.1 final 字段构建后再改变

在有些时候（如反序列化），系统需要在对象创建完之后修改对象的 final 字段值。final 字段可以通过反射和其他依赖于实现的方式来修改。这种情况唯一存在合理语义的场景是，对象被创建，然后其 final 字段被更新。在该对象的 final 字段的所有更新完成之前，该对象不应该对其他线程可见，且 final 字段也不应该被读取。在设置 final 的构造器结束时，以及通过反射或其他机制一修改完 final 字段，final 字段就被冻结了。

即使如此，还是会有一些并发问题。如果 final 字段在字段声明中被初始化成一个编译时常量，对该 final 字段的修改将不可见，因为使用该 final 字段的地方都在编译时被替换成了编译时常量。

静态 final 字段仅能在类初始化时赋值，不能通过反射修改。

另一个问题是，该语义是被设计来允许 final 字段的激进优化的。在一个线程内，允许将对 final 字段的读取与可能会通过反射改变 final 字段的方法调用进行重排序。

例如，考虑图 19 中的代码片段。在方法 d()中，编译器可以自由地对读 *x* 的操作与 g()调用进行重排序。这样，A().f()可能会返回-1,0 或 1.

```
class A {
        final int x;
        A() {
                x = 1;
        }
        int f() {
                return d(this,this);
        }
        int d(A a1, A a2) {
                int i = a1.x;
                g(a1);
                int j = a2.x;
                return j - i;
        }
        static void g(A a) {
                // uses reflection to change a.x to 2
        }
}
```

图 19：读取 final 字段的重排序与反射修改 final 字段的例子

为了避免这种问题，有些实现可能会提供一种方式，在 *final 字段安全上下文（final field safe context）*中执行一块代码。如果对象是在 final 字段安全上下文里创建的，final 字段的读取将不会与发生在 final 字段安全上下文中的 final 字段修改进行重排序。例如，clone 方法或 ObjectInputStream.readObject 方法可能有必要在 final 字段安全上下文里执行。这意味着，对于通过这种方法创建完成的对象的 final 字段，如果用反射来修改，依然会有正确的保证。

final 字段安全上下文还有其他保护作用。如果一个线程看到了某个未正确发布的对象引用（允许该线程看到 final 字段的默认值），那么，在 final 字段安全上下文

里，在读取该对象的某个正确发布的引用后，那些读操作（译者注：使用未正确发布的对象引用的读操作）也将确保能看到 final 字段正确的值。

适合使用 final 字段安全上下文的一个地方是在 executor 或线程池中。通过在单独的 final 字段安全上下文里执行各个 Runnable，executor 可以保证当某个 Runnable 未正确访问某个对象 o 时，不会导致该 executor 处理的其他 Runnable 失去 final 字段保证。

实现中，一般来讲，编译器不应该将对 final 字段的访问移入或移出 final 字段安全上下文。但可以在这样的上下文执行的周围移动，只要对象不是在该上下文里创建的。

9.2 final 字段的正式语义

final 实例字段的正式语义如下。当对象 *o* 的构造器中有写 final 字段 *f* 的操作，在退出构造器后，不管是异常还是正常退出，对象 *o* 的 *f* 上都会发生冻结动作。

反射和其他特殊机制（如反序列化）可以在对象构造器执行完成后修改 final 字段。java.lang.reflect 中 Field 类里的 setX(...)方法可以实现这种效果。如果底层字段是 final 的，除非在该字段上成功执行了 setAccessible(true)且该字段是非静态的，否则将抛出 IllegalAccessException。如果通过这种特殊机制修改了某个 final 字段，在修改后该字段应该立即发生冻结动作。

9.2.1 final 字段安全上下文

某个实现可能会提供一种方式来在 final 字段安全上下文里执行一块代码。例如，指定一个方法在 final 字段安全上下文里执行。final 字段安全上下文的作用域等同于所执行方法的作用域，传入方法的参数将是 final 字段安全上下文的参数。

final 字段安全上下文里执行的那些动作后面会按程序顺序紧跟着一个合成的用于标记 final 字段安全上下文结束的动作。

9.2.2 节中讲述的解引用链和内存链必须满足其章节里描述的约束，而不管 final 字段安全上下文里执行的动作是否发生在独立的线程中。至于判断普通的 happens-before 顺序是否有用于决定在读取 final 字段时能看到哪个写操作，正如 9.2.2 中描述的，final 字段安全上下文总是被视作一个独立的线程（As far as determining whether normal happens-before orderings are used in determining which writes can be seen by a read of a final field, as described in Section 9.2.2, a final field safe context is always considered to be a separate thread）。

有一个例外，final 字段安全上下文不会影响除了 9.2.1 - 9.2.2 节中描述的任何语义。例如，final 字段安全上下文里的动作与 final 字段安全上下文外的动作仍是按程序顺序发生的。例外的是每个终结器会在一个独立的 final 字段安全上下文里发生。

9.2.2 替换 和/或 补充顺序约束（Replacement and/or Supplemental Ordering Constraints）

对于每个执行过程，有两个额外的偏序关系会影响到读操作的行为，解引用链（$\xrightarrow{dc}$）和内存链（$\xrightarrow{mc}$），他们被当作执行过程的一部分。这些偏序关系必须满足下列约束（不需要唯一解）：

- **解引用链（Dereference Chain）** 如果某个动作 *a* 是线程 *t*（非创建对象 *o* 的线程）对一个字段或对象 *o* 中的元素的读或写操作，且 *o* 的地址没有作为参数传递给包含 *a* 的 final 字段安全上下文，那么，必须存在线程 *t* 中的某个读操作 *r*，能看到 *o* 的地址，使得 $r \overset{dc}{\rightarrow} a$。

- **内存链（Memory Chain）** 内存链顺序上有几个约束：

  a)如果 r 是读操作，且能看到写操作 w，那么，必须有 $w \overset{mc}{\rightarrow} r$。

  b)如果 r 和 a 是使得 $r \overset{dc}{\rightarrow} a$ 成立的动作，那么，必须有 $r \overset{mc}{\rightarrow} a$。

  c)如果 w 是线程 t（非创建对象 *o* 的线程）写对象 o 的操作，且 o 的地址没有作为参数传递给包含 *a* 的 final 字段安全上下文，那么，必须存在线程 *t* 中的某个读操作 *r*，能看到 *o* 的地址，使得 $r \overset{mc}{\rightarrow} w$.

  d)如果读操作 r 读取的是在 final 字段安全上下文（以生成的动作 a 结尾，使得 $a \overset{po}{\rightarrow} r$）里创建的对象的 final 实例字段，那么，必须有 $a \overset{mc}{\rightarrow} r$。

因为 final 字段外加的语义，我们使用一个不同的顺序约束集来确定哪个写会发生在读之前，用以确定哪个写能够被读操作看到。

我们从普通的 happens-before 开始，除去读操作是读取 final 实例字段以及读写发生在不同的线程中或写操作通过一种诸如反射的特殊机制发生的场景。

此外，我们利用来自 final 实例字段用到的顺序。给定一个写操作 w，一个冻结操作 f，一个动作 a（不是对 final 字段的读操作），一个对由 f 冻结的 final 字段的读操作 r1 以及一个读操作 r2，使得 $w \overset{hb}{\rightarrow} f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r1 \overset{dc}{\rightarrow} r2$，那么，当确定哪个值可以被 r2

看到时，我们认为 w $\xrightarrow{hb}$ r2（但这些顺序不与其他的 $\xrightarrow{hb}$ 顺序有传递闭包关系）。注意 $\xrightarrow{dc}$ 顺序是自反的，且 r1 不能与 r2 一样。注意，不管 r2 读的是 final 或非 final 字段，这些约束都有效。

我们以常规的方式使用这些顺序来确定哪个写能够被读看到：如果读操作 r 排在写操作 w 前面（a read r can see a write w if r is ordered before w and there is no intervening write w' ordered after w but before r），且中间没有插进一个 w'排在 w 后面但在 r 前面，那么，r 就可以看到写操作 w。

9.2.3 静态 final 字段

类初始化规则会确保任意线程在读取静态字段时会与该类的静态初始化进行同步，这是唯一一处能为静态 final 字段赋值的地方。如此一来，静态 final 字段在 JMM 中就不需要特殊规则。

静态 final 字段只能在定义字段的类的初始化器中修改，java.lang.System.in，java.lang.System.out 以及 java.lang.System.err 是例外，他们可以分别在 java.lang.System.setIn，java.lang.System.setOut 以及 java.lang.System.setErr 方法中修改。

9.3 用于 final 字段的 JVM 规则

最初的 JVM 规范中有关修改 final 字段的规则不像 JLS 中定义的那样清晰，有些模棱两可。JVM 规范中对于可接受的类文件会由本规范澄清/修改以更接近 JLS 中的描述。

putfield 指令在更新一个 final 字段时必须是在声明该 final 字段的类的<init>方法中。否则，将抛出 IllegalAccessError。

putstatic 指令在更新一个静态 final 字段时必须是在声明该静态 final 字段的类的<clinit>方法中。否则，将抛出 IllegalAccessError。

10 典型测试用例与 final 字段的行为

为了确定某个对 final 字段的读操作是否能够保证看到该字段的初始值，就必须确定，不会存在这样一种方式，能够构建出偏序 $\overset{mc}{\rightarrow}$，使得从对该字段的冻结操作 f 到对该字段的读操作 r1 之间，有 f $\overset{hb}{\rightarrow}$ a $\overset{mc}{\rightarrow}$ r1 的关系。

图 20 是个会出问题的例子。对象 o 是在线程 1 中创建的，会被线程 2 和 3 读取。线程 2 中对 r4.f 的读操作的解引用和内存链能够穿过线程 2 中任一对 o 的引用的读操作。在穿过全局变量 p 的链上，没有动作排在冻结动作的后面。如果用了这个链，对 r4.f 的读操作将不会与有关的冻结操作正确地排序。因此，不能保证 r5 能看到 final 字段正确构建的值。

f 是 final 字段，默认值为 0

Thread 1	Thread 2	Thread 3

r1.f = 42;
p = r1;
freeze r1.f;
q = r1;
	r2 = p;
	r3 = r2.f;
	r4 = q;
	if (r2 == r4)
	  r5 = r4.f;
		r6 = q;
		r7 = r6.f;

我们假设 r2，r4，r6 看不到 null 值。r3 和 r5 可以为 0 或 42，r7 必须为 42.

图 20：引用会被读两次的 final 字段示例

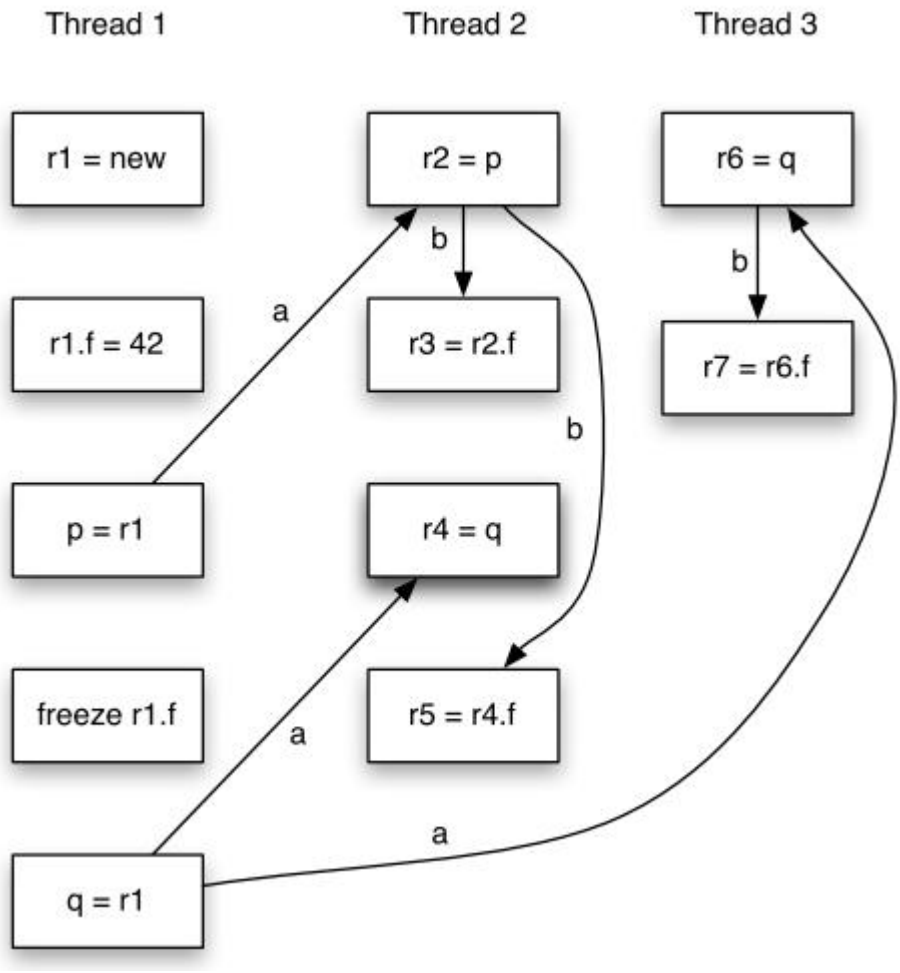


图 21：图 20 某个执行过程的内存链

r5 获得不了这样的保证反映出了编译器的合法转换。编译器可以分析这段代码，然后发现 r2.f 和 r4.f 读的是同一个 final 字段。因为 final 字段旨在不变，编译器可能会将线程 2 中的 r5 = r4.f 替换成 r5 = r3。

正式地，这是通过解引用链顺序(r2 = p)$\overset{dc}{\rightarrow}$(r5 = r4.f)得来的，但不是通过(r4 = q)$\overset{dc}{\rightarrow}$(r5 = r4.f)。一个替换偏序，其解引用链顺序(r4 = q)$\overset{dc}{\rightarrow}$(r5 = r4.f)也是合法的。然而，为了保证 final 字段读操作能够看到正确的值，必须确保所有可能的解引用和内存链的正确顺序。

不像线程 2，在线程 3 中，所有可能的对 r6.f 的读操作的链都包含线程 1 中对 q 的写操作。因此，该读操作与冻结操作是正确排序的，就能保证看到正确的值。

一般而言，如果线程 t_2 中对 final 字段 x 的读操作 $\mathcal{R}$ 通过内存链以及 happens-before 与线程 t_1 中的冻结操作 $\mathcal{F}$ 正确排序了，那么，就能保证该读操作能看到在冻结操作 $\mathcal{F}$ 之前设置的 x 的值。而且，如果一个对象只存在于线程 2 范围内，对其元素的读操作，放到从 x 中加载一个引用的后面，就能保证这些读操作会发生在所有写操作 w 之后，使得 $w \overset{hb}{\rightarrow} \mathcal{F}$.

图 22 和 23 中展示了 final 字段提供的传递性保证。对于这个例子，线程 2 中没有解引用链，可以允许读操作通过 a 追溯到 p 的一个不正确发布。由于 final 字段 a 必须正确读取，所以，该程序不仅能保证看到 a 正确的值，也能保证看到数组内容正确的值。

a is a final field of a class A

Thread 1	Thread 2
r1 = new A;	r3 = p;
r2 = new int[1];	r4 = r3.a;
r1.a = r2;	r5 = r4[0]
r2[0] = 42	
freeze r1.a;	
p = r1;	

假设线程 2 读取 p 看到了线程 1 写入的值，那么，线程 2 读取 r3.a 以及 r4[0]就能保证看到线程 1 中的写操作

图 22：final 字段的传递性保证

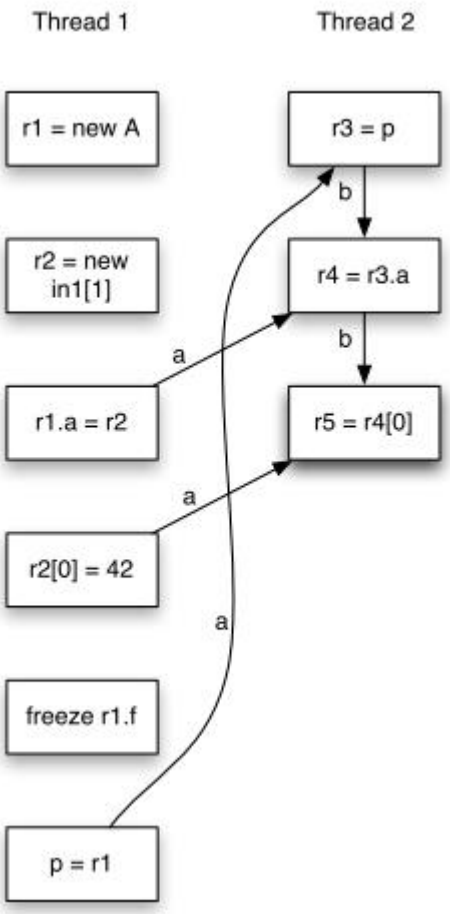


图 23：图 22 中某个执行过程的内存链

图 24 和 25 展示了一个例子两个有趣的特征。首先，拥有一个 final 字段的对象的引用在该 final 字段冻结之前被存到了（通过 r2.x = r1）堆上。因为被 r2 引用的对象在 p = r2 之前都不可达，这发生在冻结操作之后，因此，该对象是正确发布的，其 final 字段保证依然适用。

f is a final field; x is non-final

Thread 1	Thread 2	Thread 3
r1 = new ;	r3 = p;	r5 = q;
r2 = new ;	r4 = r3.x;	r6 = r5.f;
r2.x = r1;	q = r4;	
r1.f = 42;		
freeze r1.f;		
p = r2;		

假设线程 2 看到线程 1 写入的值，且线程 3 读取 q 能看到线程 2 的写操作，那么，可以保证 r6 能看到 42.

图 24：另一个 final 字段示例

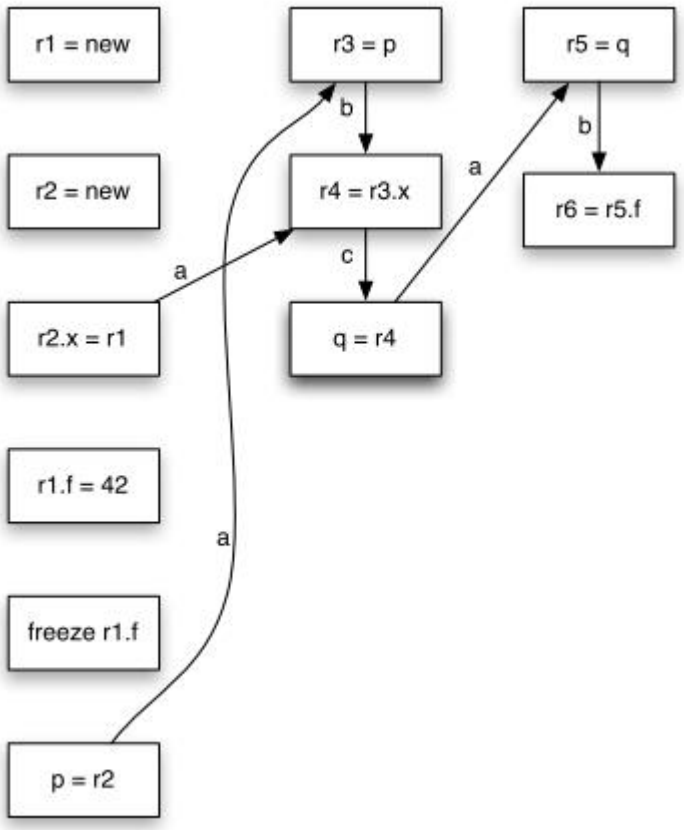


图 25：图 24 中某个执行过程的内存链

这个例子还展示了内存链规则的使用。保证线程 3 看到 f 正确初始值的内存链穿过了线程 2.一般说来，这可以保证一个对象的不可变性，不管是哪个线程写出了该对象的引用。

11 字分裂(Word Tearing)

实现 Java 虚拟机需要考虑的一件事情是，字段之间以及数组元素之间是独立的，更新一个字段或元素不能影响任何其它字段或元素的读取与更新。尤其是，两个线程在分别更新 byte 数组相邻的元素时，不能互相影响与干扰，且不需要同步来保证顺序一致性。

有些处理器（尤其是早期的 Alphas 处理器）没有提供写单个字节的功能。在这样的处理器上更新 byte 数组，若只是简单地读取整个字，更新对应的字节，然后将整个字再写回内存，将是不合法的。这个问题有时候被称为“字分裂(word tearing)”，在单独更新单个字节有难度的处理器上，就需要寻求其它方式了。图 26 展示了一个检测字分裂的测试用例。

```
public class WordTearing extends Thread {
 static final int LENGTH = 8;
 static final int ITERS = 1000000;
 static byte[] counts = new byte[LENGTH];
 static Thread[] threads = new Thread[LENGTH];
 final int id;
 WordTearing(int i) {
          id = i;
 }
 public void run() {
          byte v = 0;
          for (int i = 0; i < ITERS; i++) {
                   byte v2 = counts[id];
                   if (v != v2) {
                            System.err.println("Word-Tearing found: " + "counts[" + id
                                     + "] = " + v2 + ", should be " + v);
                            return;
                   }
                   v++;
                   counts[id] = v;
          }
 }
```

```
    public  static  void  main(String[]  args)  {
            for  (int  i  =  0;  i  <  LENGTH;  ++i)
                    (threads[i]  =  new  WordTearing(i)).start();
    }
}
```

图 26：写字节时禁止覆盖相邻字节

12 double 和 long 的非原子性处理

某些 Java™ 实现可能发现将对 64 位 long 或 double 值的写操作分成两次相邻的 32 位值写操作更方便。为了效率起见，这种行为是实现可以自行决定的。Java™ 虚拟机可以自由地决定是原子性的对待 long 和 double 值的写操作还是一分为二的对待。

鉴于本内存模型的目的，我们将对非 volatile long 或 double 值的单次写操作视作两次分开的写操作：每次 32 位。这可能会导致一种情况，某个线程会看到某次写操作中 64 位的前 32 位，以及另外一次写操作的后 32 位。读写 volatile 的 long 和 double 总是原子的。读写引用也总是原子的，而不管引用的实现采用的是 32 位还是 64 位。

我们鼓励 VM 的实现者尽可能避免将 64 位值的写操作分开。鼓励编码人员将共享的 64 位值声明为 volaitle 的或将其程序正确同步以避免可能的并发问题。

13 公平性

JLS 规范不保证抢占式的多线程，也没有任何公平性保证：没有硬保证说任意线程会让出 CPU 供其他线程调度。缺乏这样的保证部分是因为这对诸如线程优先级和实时线程（实时 Java 虚拟机规范）来说过于复杂。大部分 Java 虚拟机实现多少都

会提供一定程度的公平性保证，但细节都是与实现相关的，这属于服务质量方面的问题了，而不是硬性的要求。

这个问题对程序的影响可以参见图 27。由于缺乏公平性，CPU 一直运行线程 1 且永不让出 CPU 给线程 2 就是一种合法的行为：如此一来，程序永远也不会停止。因为这种行为是合法的，编译器将 synchronized 块移到 while 循环的外面也是合法的了，效果跟不让出 CPU 一样。

Thread 1	Thread 2

while (true)
  synchronized (o) {
  if (done) break;
  think
  }
	synchronized (o) {
	  done = true;
	}

图 27：缺乏公平性容许线程 1 永远不让出 CPU

编译器的这种转换虽然合法但却不是我们想要的。一般说来，编译器可以进行锁粗化（即，如果编译器发现有在同一个对象上的连续两个同步方法的调用，在方法调用期间就无需释放锁）。这里的精确的取舍很微妙，且所提升的性能需要能够平衡获取锁时的更长耗时。

然而，编译器的转换存在一些会降低公平性的限制。例如，图 28，如果我们看到了线程 2 打印的信息，且除了线程 1 和 2 没有其他线程在运行，那么线程 1 必须看到写入 v 的值，打印消息然后终止。这就阻止了编译器将线程 1 中读取 v 的操作提到循环外面。

Initially, v is volatile and v = false

Thread 1	Thread 2
while (!v);	v = true;
System.out.println("Thread 1 done");	System.out.println("Thread 2 done");

图 28：如果我们看到打印出了信息，线程 1 必须要看到写入 v 的值并退出循环

依照 7.5 节中可观察的动作上的规则，如果看到线程 2 打印的消息，线程 1 就必须终止。若线程 2 中的打印是在可观察的动作集 ***O*** 中，那么，对 v 的写操作和读取 v 看到 0 的操作必须也在 ***O*** 中。此外，程序不能执行无限数量的不在 ***O*** 中的额外动作。因此，致使该程序 hang 的（一直运行，没有执行额外的外部动作）唯一可观察的行为里，除了 hang 以外没有其他可观察的外部动作。包括打印动作。

14 wait 集与通知（Notification）

每个对象，除了拥有相关联的锁，还有个关联的 wait 集。wait 集是一组线程。对象首次被创建时，其 wait 集是空的。往 wait 集中添加线程和从 wait 集中移除线程的基础方法都是原子的。wait 集仅能通过 Object.wait, Object.notify 以及 Object.notifyAll 来操作。 线程的中断状态以及 Thread 类里处理中断的方法也都会影响到 wait 集。此外，Thread 类里休眠（sleep）与加入其他线程（join）相关的方法拥有源自这些等待和通知动作的属性。

14.1 等待（Wait）

一调用 wait()或有时限参数的 wait(long millisecs)和 wait(long millisecs, int nanosecs)就会发生等待动作，以 0 为参数调用 wait(long millisecs)，或以两个 0 为参数调用 wait(long millisecs, int nanosecs)，都等价于调用 wait()。

假设 t 是在对象 *m* 上执行 wait 方法的线程，设 *n* 是线程 *t* 在 *m* 上没有执行过相应的 unlock 的 lock 操作的次数。下列动作之一就会发生。

- 如果 *n* 为 0，将抛出 IllegalMonitorStateException(即，线程 t 尚未在 *m* 上执行 lock 操作)。
- 如果是带有时间限制参数的 wait，且毫微秒参数不在 0-999999 范围内或毫秒参数为负数，将抛出 IllegalArgumentException。
- 如果线程 *t* 被中断，将抛出 InterruptedException 且 *t* 的中断状态会被设为 false。
- 否则，将会发生下列事件序列：

    1. 线程 *t* 被添加到对象 *m* 的 wait 集，并在 *m* 上执行 *n* 次 unlock 动作。
    2. 在线程 *t* 从 *m* 的 wait 集中移除之前，线程 *t* 不会执行更多指令。下列任一动作都可能引起线程从 wait 集中移除，并在后续某个时间点继续运行。

        – 在 *m* 上执行了通知（notify）动作，t 被选择从 wait 集中移除。

        – 在 *m* 上执行了 notifyAll 动作。

        – 在 *t* 上执行了中断（interrupt）动作。

– 如果是带时间参数的 wait，从 wait 动作开始过了至少 millisecs 毫秒加 nanosecs 毫微秒之后，内部动作就会将 *t* 从 *m* 的 wait 集中移除。

– JVM 实现相关的内部动作也会导致 *t* 从 *m* 的 wait 集中移除。JVM 实现可以，但是不鼓励，执行“虚假唤醒” —— 即将线程从 wait 集中移除，然后线程可以恢复执行，而没有明确的指令让 JVM 这么做。注意这就必须仅在循环里使用 wait，且仅在该线程等待的逻辑条件满足后才退出循环。

每个线程必须确定一个会导致该线程从 wait 集中移除事件的顺序。这个顺序无需与其他顺序一致，但该线程必须表现的仿佛那些事件就是以那样的顺序发生的。例如，若线程 *t* 在 *m* 的 wait 集中，然后 *t* 上的中断（interrupt）和 *m* 上的通知动作都发生了，在这些事件上就必须有个顺序。

如果将中断作为第一个发生的事件，那么 *t* 最后会从 wait 调用中抛出 InterruptedException 返回，*m* 的 wait 集中的某一其他线程（如果在通知那一刻 wait 集中还有线程的话）就必须接受通知事件。如果通知被当作是第一个发生的事件，那么 *t* 最后会从 wait 中正常返回，而留下中断待处理。

3. 线程 *t* 在 *m* 上执行 *n* 次 lock 动作。

4. 如果线程 *t* 在第 2 步中因中断而被移除出了 *m* 的 wait 集，那么 *t* 的中断状态将被设为 false 且 wait 方法要抛出 InterruptedException。

14.2 通知（Notification）

一调用 notify 和 notifyAll 就会发生通知动作。设线程 *t* 是在对象 *m* 上执行任一上述方法的线程，*n* 为 *t* 在 *m* 上执行的还没有执行过相应 unlock 动作的 lock 动作的次数。会发生下面的动作之一。

- 如果 *n* 为 0，将抛出 IllegalMonitorStateException。这是线程 *t* 尚未在 *m* 上执行 lock 动作的情况。
- 如果 *n* 大于 0 且这是一个 notify 动作，那么，若 *m* 的 wait 集非空，将从当前 *m* 的 wait 集中选择一个线程 *u* 并移除出 wait 集（没有保证说会选择 wait 集中的哪一个线程）。这里从 wait 集中移除 *u* 会使得 *u* 从 wait 动作中恢复执行。但是需要注意，*u* 从 wait 中恢复后，lock 操作并不能立马成功，直到 *t* 在某个时刻对 *m* 的管程执行了 unlock 操作。
- 如果 *n* 大于 0 且这是一个 notifyAll 动作，那么，所有线程都会从 *m* 的 wait 集中移除，然后恢复执行。但是需要注意，从 wait 恢复期间一次仅会有一个线程能锁定需要的管程。

14.3 中断（Interruptions）

一调用 Thread.interrupt 方法以及会轮流调用该方法的其它方法如 ThreadGroup.interrupt，就会发生中断动作。设 *t* 为调用 U.interrupt 的线程，对于某一线程 *u*，*t* 和 *u* 可能是同一线程。这个动作会导致 *u* 的中断状态被设为 true。

此外，如果存在某一对象 *m*，其 wait 集中包含了 *u*，*u* 将会从 m 的 wait 集中移除。这会使得 *u* 从 wait 动作中恢复，在重新锁定 *m* 的管程后，抛出 InterruptedException。

Thread.isInterrupted 调用能够确定线程的中断状态。静态方法 Thread.interrupted 可以被线程调用来获取并清除其自身的中断状态。

14.4 等待（Waits），通知（ Notification）以及中断（Interruption）间的相互影响

上述规范允许我们确定有关等待（Waits），通知（ Notification）以及中断相互作用时的几个属性。如果一个线程在等待的时候既被通知又被中断，可能会发生下列两种情形之一：

- 从 wait 中正常返回，留下中断待处理（换而言之，Thread.interrupted 会返回 true）
- 通过抛出 InterruptedException 从 wait 中返回

线程也许不会重置其中断状态，然后从 wait 中正常返回。

类似地，通知事件不能因为中断而丢失。假如有一组线程 *s* 在对象 *m* 的 wait 集中，然后另外一个线程在 *m* 上执行了通知操作，可能会发生下列两种情形之一

- *s* 中至少有一个线程要从 wait 中正常返回。“正常返回”的意思是返回时不抛出 InterruptedException。
- *s* 中所有线程都必须抛出 InterruptedException 来退出 wait。

注意，如果一个线程既被中断又通过 notify 来唤醒，若该线程通过抛出 InterruptedException 来从 wait 中返回，那么 wait 集中的某一其他线程必须响应通知事件。

15 Sleep 与 Yield

Thread.sleep 会导致当前正在运行的线程休眠（暂停执行）指定的时间，该时间受限于系统时钟和调度器的精度与准确度。该线程不会失去任何管程的所有权，且恢复执行要看调度以及处理器是否空闲。

休眠 0 毫秒以及 yield 操作都不要求有可观察到的效果。

需要重点强调的是，Thread.sleep 和 Thread.yield 都没有任何同步语义。特别是，编译器不需要在 sleep 或 yield 调用之前将寄存器中缓存的写操作回写到共享主存，也不需要在 sleep 或 yield 调用之后重新加载缓存在寄存器里的值。例如，下面（存在问题的）的代码段中，假设 this.done 是一个非 volatile boolean 字段：

```
while (!this.done)
    Thread.sleep(1000);
```

编译器可以只读取 this.done 字段一次，然后在每次循环中重用缓存起来的值。这意味着即使有其他线程改变了 this.done 的值，循环也可能永远不会停止。

16 终结操作（Finalization）

本附录将详细说明 Java™ 语言规范 12.6 节中的变化，是与终结操作处理相关的。相关的部分复制到这里了。

类 Object 有一个 protected 方法 finalize；这个方法可以被其他类重写。这种特别定义的能通过一个对象来调用的 finalize 被称为该对象的终结器（finalizer）。在对象的存储空间被垃圾回收之前，Java 虚拟机将会调用该对象的终结器。

终结器提供了一个回收无法被自动存储管理器释放的资源的机会。在这种情况下，简单地回收对象所使用的内存不能保证对象持有的资源也被回收。

JLS 没有规定终结器会在多久被调用，只是说在重用对象所占存储时会调用。而且，也没有规定对于一个给定的对象，会由哪个线程来调用其终结器。但是，可以保证，调用终结器的线程在终结器被调用时不会持有任何对用户可见的同步锁。如果在终结操作过程中抛出了未捕获的异常，异常会被忽略，然后中止该对象的终结操作。

还应该注意的是，一个对象构造器的完成 happens-beofre finalize 方法的执行（正式意义上的 happens-before）。

需要重点强调的是，多个终结器线程可能同时活动（在大型 SMP 机器上有时候需要这样），如果一个大的连接在一起的数据结构变成了垃圾，该数据结构上的所有对象所有的 finalize 方法可能会在同一时刻调用，每个终结器都运行在不同的线程中。

Object 类里声明的 finalize 不做任何事情。

实际上，Object 类声明一个 finalize 方法旨在任意类总能调用其超类的 finalize。除非编码人员有意废弃超类中终结器的动作，否则，总是应该在子类中调用超类的终结器。（不像构造器，终结器不会自动调用超类的终结器；这种调用必须通过显式地编码来完成。）

为了提高效率，JVM 实现可以追踪那些没有重写 finalize 方法或只是像如下这种简单重写了的对象，

```
protected void finalize() throws Throwable {
        super.finalize();
}
```

我们鼓励 JVM 实现对待这样的对象像没有重写终结器的那样，然后高效的终结他们，如 16.1 中的描述。

终结器可以像任意其他方法一样被显式地调用。

java.lang.ref 包描述了弱引用，他们与垃圾回收和终结操作存在相互影响。对于与 Java 有特殊交互的任意 API，JVM 实现者必须认识到 java.lang.ref API 中的强制要求。本规范不会以任何形式讨论弱引用。细节内容读者可以参考 API 文档。

16.1 终结操作的实现

每个对象都可以通过两类属性来描述：可达的、终结器可达或不可达的，还可以是未终结的，可终结的或终结了的。

可达的对象是指从任一活动的线程中在任何潜在的持续计算中能访问到的任意对象。任何被某个字段或数组元素引用的对象都是可达的。最后，如果某个对象引用被传入了 JNI 方法，那么该对象在 JNI 方法完成前必须视作是可达的。

可以设计程序的优化转换，降低简单判断下认为是可达对象的数目。例如，编译器或代码生成器可以选择将一个不再使用的变量或参数设为 null，以便这样的对象可能被更早地回收。

另一个例子，如果对象字段的值存在寄存器中，也可以进行优化转换。程序可能直接访问寄存器而非对象，且再也不会访问该对象了。这意味着该对象已经成为了垃圾。

注意，仅当引用是在栈上，没有存到堆上的时候才允许这种优化。例如，终结器守卫者（Finalizer Guardian）模式：

```
class Foo {
        private final Object finalizerGuardian = new Object() {
                protected void finalize() throws Throwable {
                        /* finalize outer Foo object */
                }
        }
}
```

如果某个子类重写了 finalize 但没有显式地调用 super.finalize，终结器守卫者会强制调用 super.finalize。

如果这些优化允许针对存储在堆上的引用，那么，编译器可以检测到 finalizerGuardian 字段从不会被读取，然后将其清除，立马回收这个对象，提早调用终结器。这与原本的意图背道而驰：编码人员可能想要在 Foo 实例变得不可达时

调用 Foo 的终结器。因此，这种转换是不合法的：只要外部对象可达，该内部对象应当也可达。

这种转换可能会导致 finalize 方法的调用比预期发生的早，为了让用户可以阻止这种情况，我们提出同步可以保持对象存活的概念。如果一个对象的终结器能导致该对象上的同步，那么，该对象必须是存活的，且只要持有该对象上的锁都认为其是可达的。

注意，这不会阻止同步移除：仅当终结器在对象上同步时，同步才会保持一个对象的存活。由于终结器会在另一个线程中调用，在很多情况下同步会被移除掉。

终结器可达的对象能从某些可终结的对象通过一个引用链可达，但不是从任意存活的线程可达。不可达对象通过以上两种方式都不可达。

未终结的对象其终结器从未被自动调用过；终结的对象其终结器已经被自动调用过了。可终结的对象的终结器从未被自动调用过，但 Java 虚拟机最终会自动调用其终结器。在对象所有的构造器完成之前，不能认为其是可终结的。一个对象要成为可终结的，类 Object 的构造器必须被调用且正常完成了，其他构造器可以抛出异常终止。每个在终结操作之前对对象字段的写操作必须对该对象的终结操作可见。而且，终结操作之前对对象字段的读操作都不可以看到该对象终结操作已经被初始化之后的写操作。

16.2 与内存模型的交互

内存模型必须能够确定何时可以提交发生在终结器中的动作。本节描述了终结操作与内存模型的交互。

终结器以一种特殊方式与 final 字段语义相互作用。每个终结器都发生在一个 final 字段安全上下文（见 9.2.1 节中的描述）里。而且，如第 5 节所言，每个终结器都以逻辑读取被终结对象的一个引用为开头。该 final 字段的内存链（见 9.2 节）穿过该读操作，为该终结器提供了不可变保证。

每个执行过程都有一些*可达性决策点（reachability decision points）*，记为 d_i，一个动作，要么是在 d_i之前（*comes-before* d_i），要么是在 d_i之后（*comes-after* d_i）。除非明确说明，本节中 *comes-before* 与本内存模型中的所有其它顺序都无关。

如果 *r* 是个读操作，能够看见写操作 *w*，且 r 在 d_i之前（r *comes-before* d_i），那么，w 也必须在 d_i之前（w *comes-before* d_i）。如果 *x* 和 *y* 是在同一个变量或管程上同步的两个动作，有 $x \xrightarrow{SO} y$ 且 y 在 d_i之前（y *comes-before* d_i），那么，x 必须是在 d_i之前（x *comes-before* d_i）。

在每个可达性决策点上，有一个对象集被标记为不可达，这些对象的一个子集会被标记为可终结的。这些可达性决策点也是 Reference 根据 *java.lang.ref* 中提供的规则进行检查、入队和清理的点。

如 9.2.1 所描述，每个终结器都发生在 final 字段安全上下文里。

可达性

在 d_i 点明确可达的对象是在下列规则下可达的对象：

- 如果存在一个对某个类 C 静态字段 v 的写操作 w1，使 w1 写入的值是对象 B 的引用，其中类 C 是由可达的加载器加载的，且不存在对 v 的写操作 w2 使 $\neg(w2 \xrightarrow{hb} w1)$，同时，w1 和 w2 都是在 d_i 之前（*comes-before* d_i），那么，在 d_i 点，对象 B 从该静态字段明确可达。
- 如果存在一个对 A 的元素 v 的写操作 w1，使 w1 写入的值是对象 B 的引用且不存在对 v 的写操作 w2 使得 $\neg(w2 \xrightarrow{hb} w1)$，同时，w1 和 w2 都是在 d_i 之前（*comes-before* d_i），那么，在 d_i 点，对象 B 从 A 明确可达。
- 如果对象 C 从对象 B 明确可达，对象 B 从对象 A 明确可达，那么，C 从 A 明确可达。

动作 a 是对 X 的主动使用，当且仅当：

- 对 X 的元素有读写操作。
- 对 X 有 lock 或 unlock 操作，且在 X 上存在一个发生在 X 终结器调用之后的 lock 动作。（it locks or unlocks X and there is a lock action on X that happens-after the invocation of the finalizer for X.）
- 有对 X 引用的写操作。
- 如果有对对象 Y 的主动使用，且 X 从 Y 明确可达。

如果对象 X 在 d_i 点被标记为不可达，那么

- X 在 d_i 点不可以从静态字段可达。
- 所有线程 t 中位于 d_i 点之后（come-after d_i）的对 X 的主动使用，都必须发生在 X 的终结器调用中，或者是由于线程 t 在 d_i 点之后执行了一个对 X 引用的读操作。
- 所有位于 d_i 点之后的（come-after d_i）看到了 X 的引用的读操作，必须要看得到对在 d_i 点不可达的对象元素的写操作，或者看到 d_i 之后的写操作。

如果一个对象 X 在 d_i 点被标记为可终结的，那么

- X 在 d_i 点必须被标记为不可达，
- d_i 点必须是 X 被标记为可终结的唯一位置，
- 发生在终结调用之后的动作必须跟在 d_i 后面（actions that happen-after the finalizer invocation must come-after d_i）。