第十七章

线程和锁

虽然前面章节的大部分讨论只关注代码在一次执行单个语句或表达式时的行为，也就是说，通过单个线程，Java 虚拟机可以支持多个线程同时执行。这些线程独立地执行对驻留在共享主内存中的值和对象进行操作的代码。支持线程的方式可以是拥有多个硬件处理器，对单个硬件处理器进行时间切片，或者对多个硬件处理器进行时间切片。

线程由 Thread 类表示。用户创建线程的唯一方法是创建该类的对象;每个线程都与这样的对象相关联。当在相应的 Thread 对象上调用 start()方法时，线程将启动。

线程的行为，特别是在没有正确同步的情况下，可能会令人困惑和违反直觉。本章描述多线程程序的语义;它包括一些规则，这些规则的值可以通过多个线程更新的共享内存的读取看到。由于该规范类似于不同硬件架构的内存模型，这些语义被称为 Java 编程语言内存模型。当不会产生混淆时，我们将这些规则简单地称为“内存模型”。

这些语义没有规定多线程程序应该如何执行。相反，它们描述了多线程程序允许展示的行为。任何只生成允许行为的执行策略都是可接受的执行策略。

17.1 同步

Java 编程语言为线程之间的通信提供了多种机制。这些方法中最基本的是同步，它是使用监视器实现的。Java 中的每个对象都与一个监视器相关联，线程可以锁定或解锁监视器。一次只能有一个线程持有监视器上的锁。试图锁定该监视器的任何其他线程都将被阻塞，直到它们能够获得该监视器上的锁。一个线程 t 可以多次锁定一个特定的监视器;每一次解锁都逆转一次锁操作的效果。

synchronized 语句(§14.19)计算一个对象的引用;然后，它尝试对该对象的监视器执行锁操作，直到锁操作成功完成才继续执行。锁操作执行之后，同步语句体就会被执行。如果主体执行完成，无论是正常地还是突然地，都会在同一个监视器上自动执行解锁操作。

一个同步方法(§8.4.3.6)在被调用时自动执行一个锁动作;直到锁操作成功完成，它的主体才会执行。如果该方法是一个实例方法，它将锁定与调用它的实例相关联的监视器(也就是说，在方法体执行期间将被称为 this 的对象)。如果方法是静态的，它将锁定与表示定义方法的类的 Class 对象相关联的监视器。如果方法体的执行完成了，不管是正常地还是突然地，都会在同一个监视器上自动执行一个解锁操作。

Java 编程语言既不阻止也不要求检测死锁条件。线程在多个对象上持有(直接或间接)锁的程序应该使用避免死锁的传统技术，在必要时创建不会死锁的高级锁原语。

其他机制，比如对 volatile 变量的读写以及对 java.util.concurrent 包中的类的使用，提供了另一种同步方式。

17.2 等待设置和通知

每个对象除了具有关联的监视器外，还具有关联的等待集。等待集是一组线程。

当一个对象第一次创建时，它的等待集是空的。向等待集添加线程和从等待集删除线程的基本操作是原子操作。等待集仅通过方法 Object.wait,Object.notify 和 Object.notifyAll 进行操作。

等待集操作也会受到线程的中断状态以及 Thread 类处理中断方法的影响。此外，Thread 类用于睡眠和连接其他线程的方法具有来自于等待和通知动作的属性。

17.2.1 等待

等待动作在调用 wait()时发生，或者计时形式 wait(long millisecs)和 wait(long millisecs, int nanosecs)。

> 用一个为 0 的参数调用 wait(long millisecs), 或用两个未 0 的参数调用 wait(long millisecs, int nanosecs), 等价于调用 wait()。

如果线程返回时不抛出 InterruptedException 异常，则该线程正常地从 wait 返回。

设线程 t 是在对象 m 上执行 wait 方法的线程，设 n 是 t 在对象 m 上未被解锁操作匹配的锁定操作的数量。发生以下操作之一:

- 如果 n 为零(即线程 t 还没有拥有目标 m 的锁)，则抛出一个 IllegalMonitorStateException。
- 如果这是计时等待，并且 nanosecs 参数不在 0-999999 的范围内，或者 millisecs 参数为负，则抛出一个 IllegalArgumentException。
- 如果线程 t 被中断，则抛出 InterruptedException 异常，并将 t 的中断状态设置为 false。
- 否则，将出现以下序列:
  1. 线程 t 被添加到对象 m 的等待集合中，并对 m 执行 n 个解锁操作。
  2. 线程 t 不会执行任何进一步的指令，直到它被从 m 的等待集中移除。该线程可能会被从等待集合中移除，因为下面的任何一个动作，并且会在之后的某个时间恢复:
     - 对 m 执行的 notify 操作，其中 t 被选择从等待集中移除。
     - 对 m 执行的 notifyAll 操作。
     - 在 t 上执行的 interrupt 操作。

- 如果这是一个定时等待，则从 m 的等待集中删除 t 的内部操作发生在从这个等待操作的开始经过至少 millisecs 毫秒加 nanosecs 纳秒之后。
- 实现的内部动作。虽然不鼓励，但允许实现执行“伪唤醒”，即从等待集中删除线程，从而允许在没有明确指令的情况下恢复。

  请注意，这一规定使得 Java 编码实践必须只在循环中使用 wait，这些循环只有在线程持有正在等待的某些逻辑条件时才终止。

每个线程必须确定可能导致其从等待集中移除的事件的顺序。该顺序不必与其他顺序一致，但线程的行为必须如同这些事件按该顺序发生一样。

例如，如果线程 t 在 m 的等待集中，然后 t 的中断和 m 的通知都发生了，则这些事件必须有顺序。如果中断被认为是首先发生的，那么 t 最终将通过抛出 InterruptedException 从 wait 中返回，并且 m 的等待集中的某个其他线程(如果在通知时存在任何线程)必须接收通知。如果该通知被认为是首先发生的，那么 t 最终将从 wait 中正常返回，同时中断仍然挂起。

3. 线程 t 对 m 执行 n 个锁定动作。
4. 如果线程 t 在步骤 2 中由于中断而从 m 的等待集中删除，则 t 的中断状态被设置为 false，并且 wait 方法抛出 InterruptedException。

17.2.2 通知

通知操作在调用 notify 和 notifyAll 方法时发生。

设线程 t 是在对象 m 上执行这两个方法之一的线程，n 是 t 在 m 上未与解锁操作匹配的锁定操作的数量。将发生以下操作之一:

- 如果 n 为 0, 则抛出 IllegalMonitorStateException。

  这是线程 t 尚未拥有目标 m 的锁的情况。

- 如果 n 大于零并且这是 notify 动作，则如果 m 的等待集不为空，则选择作为 m 的当前等待集的成员的线程 u 并将其从等待集中移除。

  不能保证选择等待集中的哪个线程。从等待集中移除使 u 能够在等待操作中恢复。然而，请注意，在 t 完全解锁 m 的监视器之后的一段时间内，u 在恢复时的锁定操作不会成功。

- 如果 n 大于零并且这是一个 notifyAll 操作，则从 m 的等待集中删除所有线程，从而恢复。

  但是，请注意，在恢复等待期间，一次只能锁定其中一个监视器。

17.2.3 中断

中断操作在调用 Thread.interrupt 时发生，以及定义为依次调用它的方法，例如 ThreadGroup.interrupt。

假设 t 是调用 u.interrupt 的线程，对于某些线程 u，t 和 u 可能相同。此操作将 u 的中断状态设置为 true。

此外，如果存在某个对象 m 的等待集包含 u，则从 m 的等待集中删除 u。这使 u 能够在等待操作中恢复，在这种情况下，重新锁定 m 的监视器后，此等待将抛出 InterruptedException。

调用 Thread.isInterrupted 可以确定线程的中断状态。线程可以调用静态方法 Thread.interrupted 来观察和清除其自身的中断状态。

17.2.4 等待、通知和中断的交互

上述规范允许我们确定与等待、通知和中断的交互有关的几个属性。

如果线程在等待时既被通知又被中断，则它可以：

- 从 wait 中正常返回，但仍有挂起的中断（换句话说，对 Thread.interrupted 的调用将返回 true）
- 通过抛出 InterruptedException 从 wait 中返回。

线程可能不会重置其中断状态，并从 wait 的调用中正常返回。

同样，通知也不会因中断而丢失。假设一组线程 s 在对象 m 的等待集中，而另一个线程在 m 上执行 notify。然后：

- s 中至少有一个线程必须从 wait 中正常返回，或者
- s 中的所有线程必须通过抛出 InterruptedException 退出 wait。

请注意，如果一个线程通过 notify 被中断和唤醒，并且该线程通过抛出 InterruptedException 从 wait 返回，则必须通知等待集中的其他线程。

17.3 Sleep 和 Yield

Thread.sleep 导致当前执行的线程在指定的持续时间内睡眠（暂时停止执行），这取决于系统定时器和调度程序的精度和准确性。该线程不会失去任何监视器的所有权，恢复执行将取决于执行该线程的调度和处理器的可用性。

需要特别注意的是，Thread.sleep 和 Thread.yield 都没有任何同步语义。具体地说，编译器不必在调用 Thread.sleep 或 Thread.yield 之前将缓存在寄存器中的写入刷新到共享内存，也不必在调用 Thread.sleep 或 Thread.yield 之后重新加载缓存在寄存器中的值。

例如，在以下(损坏的)代码片段中，假定 this.one 是一个非 volatile 布尔字段：

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

编译器可以自由地只读取一次字段 this.done，并在每次执行循环时重复使用缓存值。这意味着，即使另一个线程更改了 this.done 的值，循环也永远不会终止。

17.4 内存模型

内存模型描述给定程序和该程序的执行跟踪，该执行跟踪是否是该程序的合法执行。Java 编程语言内存模型通过检查执行跟踪中的每个读操作，并根据特定规则检查该读操作观察到的写操作是否有效来工作。

内存模型描述程序的可能行为。一个实现可以自由地生成它喜欢的任何代码，只要程序的所有结果执行都能生成一个可以由内存模型预测的结果。

这为实现者提供了大量的自由来执行大量的代码转换，包括操作的重新排序和消除不必要的同步。

例子 17.4-1. 不正确同步的程序可能会表现出令人惊讶的行为

Java 编程语言的语义允许编译器和微处理器执行优化，这些优化可以与不正确同步的代码交互，从而产生看似矛盾的行为。下面是一些同步不正确的程序如何表现出令人惊讶的行为的例子。

例如，考虑表 17.4-A 所示的示例程序跟踪。这个程序使用局部变量 r1 和 r2 以及共享变量 A 和 B。最初，A == B == 0。

表 17.4-A. 语句重新排序导致的惊人结果-原始代码

线程 1	线程 2
1: `r2 = A;`	3: `r1 = B;`
2: `B = 1;`	4: `A = 2;`

这样看来，r2 == 2 和 r1 == 1 的结果是不可能的。直观地说，指令 1 或指令 3 应该在执行中先出现。如果指令 1 先出现，它应该无法看到指令 4 的写入。如果指令 3 先出现，则它应该无法看到指令 2 的写入。

如果某些执行表现出这种行为，那么我们将知道指令 4 在指令 1 之前，指令 1 在指令 2 之前，指令 3 在指令 4 之前。从表面上看，这是荒谬的。

但是，如果这不会单独影响该线程的执行，则允许编译器对任何一个线程中的指令进行重新排序。如果指令 1 与指令 2 重新排序，如表 17.4-B 中的跟踪所示，那么很容易看出结果 r2==2 和 r1==1 可能会出现。

表 17.4-B. 语句重新排序导致的意外结果-有效的编译器转换

线程 1	线程 2
`B = 1;`	`r1 = B;`
`r2 = A;`	`A = 2;`

对于一些程序员来说，这种行为可能看起来“破碎”。 但是，应注意，此代码未正确同步：

- 在一个线程中存在写入，
- 由另一线程读取同一变量，
- 并且写入和读取不是通过同步来排序的。

这种情况是数据竞争的一个例子（§17.4.5）。当代码包含数据竞争时，通常可能出现违反直觉的结果。

几种机制可以产生表 17.4-B 中的重新排序。Java 虚拟机实现中的即时编译器可以重新排列代码或处理器。此外，运行 Java 虚拟机实现的体系结构的内存层次结构可能会使代码看起来像是在重新排序。在本章中，

我们将提到任何可以将代码重新排序为编译器的东西。

另一个惊人结果的例子可以在表 17.4-C 中看到。最初，p == q 和 p.x == 0。该程序的同步也不正确；它写入共享内存，而不强制这些写入之间的任何顺序。

表 17.4-C. 前向替换导致的惊人结果

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

一种常见的编译器优化包括在 r5 中重用读取 r2 的值：它们都是读取 r1.x 的值，没有插入写。这种情况如表 17.4- D 所示。

表 17.4-D. 前向替换导致的惊人结果

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

现在考虑线程 2 中对 r6.x 的赋值发生在第一次读取线程 1 中的 r1.x 和读取线程 1 中的 r3.x 之间的情况。如果编译器决定将 r2 的值重新用于 r5，则 r2 和 r5 的值将为 0，而 r4 的值将为 3。从程序员的角度来看，p.x 中存储的值从 0 变成了 3，然后又变回来了。

内存模型决定了在程序中的每个点都可以读取哪些值。每个独立线程的操作必须按照该线程的语义进行管理，但每次读取看到的值是由内存模型确定的除外。当我们提到这一点时，我们说程序遵守线程内语义。线程内语义是单线程程序的语义，它允许基于线程内读操作所看到的值对线程的行为进行完整的预测。为了确定线程 t 在执行中的动作是否合法，我们只需按在单线程上下文中执行的方式评估线程 t 的实现，如本规范的其余部分所定义的那样。

每次线程 t 的计算生成一个线程间操作时，它必须匹配程序中接下来 t 的线程间操作 a 。如果 a 是一个读，那么 t 的进一步计算将使用内存模型确定的 a 所看到的值。

本节提供了 Java 编程语言内存模型的规范，除了§17.5 中描述的处理 final 字段的问题。

这里指定的内存模型并不是基于 Java 编程语言面向对象的本质。在我们的示例中，为了简洁和简单，我们经常展示没有类或方法定义或显式解引用的代码片段。大多数示例由两个或多个线程组成，其中包含访问局部变量、共享全局变量或对象实例字段的语句。我们通常使用诸如 r1 或 r2 这样的变量名来表示方法或线程的局部变量。其他线程不能访问这样的变量。

17.4.1 共享变量

可以在线程之间共享的内存称为共享内存或堆内存。

所有实例字段、静态字段和数组元素都存储在堆内存中。在本章中，我们使用变量一词来指代字段和数组元素。

局部变量(§14.4)、形式方法参数(§8.4.1)和异常处理程序参数(§14.20)从来不在线程之间共享，并且不受内存模型的影响。

如果对同一个变量的两次访问(读或写)中至少有一次是写操作，则被称为冲突。

17.4.2 操作

线程间操作是一个线程执行的操作，它可以被另一个线程检测到或直接影响。程序可能执行的线程间操作有几种:

- 读取(普通或非 volatile)。读一个变量。
- 写入(普通或非 volatile)。写入一个变量。
- 同步操作，包括:
  - volatile 读。对变量的 volatile 读操作。
  - volatile 写。对变量的 volatile 写操作。
  - 锁。锁定一个监视器。
  - 解锁。解锁一个监视器。
  - 线程的（合成）第一个和最后一个操作。
  - 启动线程或检测线程已终止的操作（§17.4.4）。
- 外部操作。外部操作是可以在执行外部观察到的动作，并且具有基于执行外部环境的结果。
- 线程偏离操作(§17.4.9)。线程偏离操作仅由处于无限循环中的线程执行，在无限循环中不执行内存、同步或外部操作。如果线程执行线程偏离操作，则它后面将跟随无限数量的线程偏离操作。

  引入线程偏离操作来模拟一个线程可能如何导致所有其他线程停止并无法取得进展。

此规范仅与线程间操作有关。我们不需要关心线程内的操作(例如，添加两个局部变量并将结果存储在第三个局部变量中)。如前所述，所有线程都需要遵守 Java 程序的正确线程内语义。我们通常将线程间操作更简洁地称为简单操作。

操作 a 由一个元组< t, k, v, u >描述，包括:

- t – 执行操作的线程

- k – 操作的类型
- v – 操作中涉及的变量或监视器。

  对于锁定操作，v 是被锁定的监视器；对于解锁操作，v 表示正在被解锁的监视器。

  如果操作是（volatile 或非 volatile）读取，则 v 是正在读取的变量。

  如果操作是（volatile 或非 volatile）写入，则 v 是正在写入的变量。
- u - 操作的任意唯一标识符

外部操作元组包含一个附加组件，该组件包含执行操作的线程感知到的外部操作的结果。这可能是关于操作成功或失败的信息，以及由操作读取的任何值。

外部操作的参数（例如，哪些字节写入哪个套接字）不是外部操作元组的一部分。这些参数由线程内的其他操作设置，可以通过检查线程内语义来确定。它们在内存模型中没有明确讨论。

在非终止执行中，并非所有外部操作都是可观察的。第 17.4.9 节讨论了非终止执行和可观察操作。

17.4.3 程序和程序顺序

在每个线程 t 执行的所有线程间操作中，t 的程序顺序是反映根据 t 的线程内语义执行这些操作的顺序的总顺序。

如果所有动作以与程序顺序一致的总顺序（执行顺序）发生，则一组动作是顺序一致的，此外，变量 v 的每个读 r 都看到由写 w 写入的值 v，从而：

- 在执行顺序中，w 在 r 之前，并且
- 在执行顺序中，没有其他写入 w'，使得 w 在 w'之前，w'在 r 之前。

顺序一致性是程序执行中可见性和顺序的一个非常有力的保证。在顺序一致的执行中，所有单个操作（如读和写）都有一个总的顺序，该顺序与程序的顺序一致，每个单独的操作都是原子的，每个线程都可以立即看到。

如果一个程序没有数据竞争，那么该程序的所有执行看起来都是顺序一致的。

顺序一致性和/或免于数据竞争仍然允许由需要原子感知而不是原子感知的操作组产生的错误。

> 如果我们使用顺序一致性作为我们的内存模型，我们所讨论的许多编译器和处理器优化都将是非法的。例如，在表 17.4-C 中的跟踪中，只要将 3 写入 p.x，就需要后续读取该位置才能看到该值。

17.4.4 同步顺序

每次执行都有一个同步顺序。同步顺序是执行的所有同步操作的总顺序。对于每个线程 t，t 中同步操作的同步顺序(§17.4.2)与 t 的程序顺序(§17.4.3)一致。

同步操作产生操作上的同步-与关系，定义如下：

- 监视器 m 上的解锁操作同步-与 m 上的所有后续锁定操作 (其中“后续”是根据同步顺序定义的)。
- 对 volatile 变量 v(§8.3.1.4)的写操作同步-与通过任何线程后续对 v 的所有读 (其中“后续”是根据同步顺序定义的)。
- 启动线程的操作同步-与它启动的线程中的第一个操作。
- 将默认值(0、false 或 null)写入每个变量的操作是同步-与每个线程中的第一个操作。

  虽然在分配包含变量的对象之前给变量写一个默认值看起来有点奇怪，但从概念上讲，每个对象都是在程序开始时用其默认初始值创建的。

- 线程 T1 中的最后一个操作同步-与检测到 T1 已经终止的另一个线程 T2 中任何操作。

  T2 可以通过调用 T1.isAlive()或 T1.join()来实现。

- 如果线程 T1 中断线程 T2，则 T1 的中断同步-与任何其他线程(包括 T2)确定 T2 已被中断的任何点（通过抛出 InterruptedException 或调用 Thread.interrupted 或 Thread.isInterrupted)。

同步与边缘的源称为释放，目的称为获取。

17.4.5 Happens-before 顺序

两个操作可以通过 happens-before 关系进行排序。如果一个操作 happens-before 另一个操作，那么第一个操作将在第二个操作之前可见并排序。

如果我们有两个操作 x 和 y，我们写 hb(x, y)来表示 x happens-before y。

- 如果 x 和 y 是同一个线程的操作，并且在程序顺序上 x 在 y 之前，那么 hb(x, y)。
- 从对象的构造函数的末尾到终结器的开头(§12.6)，存在一个 happens-before 边。
- 如果操作 x 同步-与后续操作 y，那么我们也有 hb（x，y）。
- 如果 hb(x, y) 并且 hb(y, z), 那么 hb(x, z)。

类 Object 的 wait 方法（§17.2.1）具有与之相关的锁定和解锁操作；它们的 happens-before 关系由这些相关的操作定义。

应该注意的是，两个操作之间 happens-before 关系的存在并不一定意味着它们必须在实现中按该顺序发生。如果重新排序产生的结果与合法执行一致，则不违法。

例如，将默认值写入由线程构造的对象的每个字段不需要在该线程开始之前发生，只要没有读取观察到该事实。

更具体地说，如果两个操作共享“happens-before”关系，则它们不必在与它们不共享“happens-before”关系的任何代码中以该顺序出现。例如，一个线程中的写入与另一个线程的读取处于数据竞争中，可能会出现与这些读取顺序不一致的情况。

happens-before 关系定义数据竞争发生的时间。

一组同步边 S，如果它是最小的集合，那么 S 的传递闭包按照程序顺序决定了所有的执行中的 happens-before 的边，那么它就是充分的。这个集合是唯一的。

由上述定义可得出:

- 对监视器进行解锁 happens-before 对该监视器的每个后续锁。
- 对一个 volatile 字段的写 (§8.3.1.4) happens-before 对那个字段的每个后续读。
- 在线程上调用 start() happens-before 启动线程中的任何操作。
- 线程中的所有操作 happen-before 任何其他线程从该线程上的 join()成功返回。
- 任何对象的默认初始化 happens-before 程序的任何其他操作 (除了默认写入) 。

当一个程序包含两个冲突的访问(§17.4.1)，而这两个访问不是按 happens-before 关系排序的，那么它就被称为包含数据竞争。

线程间操作以外的操作的语义，如读取数组长度(§10.7)、执行检查的强制转换(§5.5、§15.16)和调用虚方法(§15.12)，不会直接受到数据竞争的影响。

> 因此，数据竞争不会导致不正确的行为，例如为数组返回错误的长度。

当且仅当所有顺序一致的执行都没有数据竞争时，程序才能正确同步。

如果程序被正确同步，则该程序的所有执行将看起来是顺序一致的(§17.4.3)。

> 这对程序员来说是一个极其有力的保证。程序员不需要对重新排序进行推理来确定他们的代码是否包含数据竞争。因此，在确定其代码是否正确同步时，它们不需要对重新排序进行推理。一旦确定代码被正确同步，程序员就不需要担心重新排序会影响他或她的代码。
>
> 程序必须正确同步，以避免在重新排序代码时可以观察到的各种违反直觉的行为。使用正确的同步并不能确保程序的整体行为是正确的。然而，它的使用确实允许程序员以一种简单的方式来推理程序的可能行为；正确同步的程序的行为对可能的重新排序的依赖要小得多。如果没有正确的同步，可能会出现非常奇怪、令人困惑和违反直觉的行为。

我们说一个变量 v 的读 r 被允许观察到一个写 w 为 v，如果，在执行轨迹的 happens-before 偏序:

- r 不在 w 之前排序（即，hb（r，w）不成立），并且
- 不存在将 w'插入写到 v 的操作(即不存在将 w'写入 v，这样 hb(w, w')和 hb(w'， r))。

非正式地，如果没有 happens- before 排序来阻止该读取，允许读 r 看到写 w 的结果。

一组操作 A 是 happens-before 一致如果对于所有 A 中的读 r,其中 W(r) 是被 r 看到的写操作, 不是要么 hb(r, W(r)) 或 A 中存在一个写 w 使得 w.v = r.v 并且 hb(W(r), w) 并且 hb(w, r) 的情况。

在 happens-before 一致性操作集中，每一个读看到一个写，它是被 happens-before 排序允许看到的。

例子 17.4.5-1. Happens-before 一致性

对于表 17.4.5-A 中的轨迹，最初为 A==B==0。跟踪可以观察到 r2==0 和 r1==0，并且仍然是 happens-before 一致的，因为存在允许每次读取看到适当写入的执行顺序。

表 17.4.5-A. happens-before 一致性允许的行为，但不是顺序一致性。

线程 1	**线程 2**
`B = 1;`	`A = 2;`
`r2 = A;`	`r1 = B;`

由于没有同步，所以每次读取都可以看到初始值的写入或另一个线程的写入。显示此行为的执行顺序为：

```
1:  B = 1;
3: A = 2;
2:  r2 =A;  // sees  initial write of0
4:  r1 =B;  // sees  initial write of0
```

另一个是 happens-before 一致的执行顺序是：

```
1:  r2 =A;  // sees  write of A = 2
3:  r1 =B;  // sees  write of B = 1
2:  B = 1;
4:  A = 2;
```

在这个执行中，读操作看到的写操作发生在执行顺序的后面。这似乎是违反直觉的，但被 happens-before 一致性所允许的。允许读看到后面的写有时会产生不可接受的行为。

17.4.6 执行

执行 E 由元祖 < P, A, po, so, W, V, sw, hb >描述, 包含:

- P – 一个程序
- A – 一个操作集
- po - 程序顺序，对于每个线程 t 来说，是 A 中 t 执行的所有操作的总顺序
- so - 同步顺序，这是 A 中所有同步操作的总顺序
- W - 一个看见写函数，对于 A 中的每一个读 r，给出 W(r)， 即 E 中的 r 看到的写操作。
- V - 一个写值函数，对 A 中每个写 w，给出 V(w)，即 E 中 w 写的值。
- sw – 同步-与, 同步操作的偏序
- hb - happens-before, 对操作的偏序

请注意，同步-与和 happens-before 元素是由执行的其他组件和格式良好的执行规则所唯一决定的(§17.4.7)。

一个执行是 happens-before 一致，如果它的操作集是 happens-before 一致(§17.4.5)。

17.4.7 格式良好的执行

我们只考虑格式良好的执行。执行 E = < P, A, po, so, W, V, sw, hb > 是格式良好的如果以下都为 true：

1. 每次读都会在执行过程中看到对同一个变量的写操作。

   所有对 volatile 变量的读写都是 volatile 操作。对于 A 中的所有读 r，我们有 A 中的 W(r) 以及 W(r).v = r.v。变量 r.v 是 volatile 当且仅当 r 是一个 volatile 读，并且变量 w.v 是 volatile 当且仅当 w 是一个 volatile 写。

2. happens-before 顺序是偏序。

   happens-before 顺序由同步-与边缘的传递闭包和程序顺序给出。它必须是有效的偏序：自反、传递和反对称。

3. 执行遵循线程内一致性。

   对于每个线程 t，在 A 中由 t 执行的操作与将由该线程以隔离的程序顺序生成的操作相同，其中假定每个读取 r 看到值 V(W(r))，则每个写入 w 写入值 V(w)。每次读取看到的值是由内存模型决定的。给定的程序顺序必须反映根据 P 的线程内语义执行操作的程序顺序。

4. 执行时 happens-before 一致的(§17.4.6)。

5. 执行遵循同步顺序一致性。

   对于 A 中的每个 volatile 读 r，既不是 so(r, W(r))的情况，也不是存在 A 中的写 w 以致 w.v = r.v 和 so(W(r), w) 和 so(w, r)的情况。

17.4.8 执行和因果关系要求

我们用 f|d 来表示通过将 f 的定义域限制为 d 而给出的函数。对于 d 中的所有 x，f|d(x) = f(x)，对于不在 d 中的所有 x， f|d(x) 未定义。

我们用 p|d 来表示偏序 p 对 d 中元素的限制。对于 d 中所有 x 和 y，p(x,y) 当且仅当 p|d(x,y)。如果 x 或 y 不在 d 中，那么 p|d(x,y)不成立。

通过从 A 提交操作来验证形式良好的执行 E=< P, A, po, so, W, V, sw, hb >。如果 A 中的所有操作都可以提交，则执行满足 Java 编程语言内存模型的因果关系要求。

从空集 C_0 开始，我们执行一系列步骤，其中我们从操作集合 A 中获取操作，并将它们添加到提交操作集合 C_i，以获得新的提交操作集合 C_{i+1}。为了证明这是合理的，对于每个 C_i，我们需要演示一个包含 C_i 的执行 E，该执行满足某些条件。

形式上，当且仅当存在以下条件时，执行 E 满足 Java 编程语言内存模型的因果关系要求：

- 操作集 $C_0, C_1, \ldots$ 以使:
  - C_0 是空集

- C_i 是 C_{i+1} 的合适的子集
- $A = U(C_0, C_1, \cdots)$

如果 A 是有限的，那么序列 $C_0, C_1, \ldots$ 将是有限的，以集合 $C_n = A$ 结束。

如果 A 是无限大的，那么序列 $C_0, C_1, \ldots$ 可能是无限的，并且这个无限序列的所有元素的并集等于 A。

- 格式良好的执行 $E1,\cdots$，其中 $E_i = <P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i>$。

给定这些操作集 $C_0, \ldots$ 和执行 $E_1, \ldots$，C_i 中的每一个操作都是 E_i 中的一个操作。C_i 中的所有操作在 E_i 和 E 中必须具有相同的 happens-before 顺序和同步顺序。理论上：

1. C_i 是 A_i 的子集
2. $hbi|_{C_i} = hb|_{C_i}$
3. $soi|_{C_i} = so|_{C_i}$

在 C_i 中被写者所写的值在 E_i 和 E 中必须相同。只有 C_{i-1} 中的读操作需要看到 E_i 中与 E 中相同的写操作。理论上：

4. $Vi|_{C_i} = V|_{C_i}$
5. $Wi|_{C_{i-1}} = W|_{C_{i-1}}$

所有不在 C_{i-1} 中的 E_i 中的读必须看到 happen-before 它们的写。每个 C_i 到 C_{i-1} 中的读 r 必须看到 E_i 和 E 中的 C_{i-1} 写操作，但是在 E_i 和 E 中看到的写操作可能不同。理论上：

6. 对于任何 A_i 到 C_{i-1} 中的读 r，我们有 $hb_i(W_i(r), r)$
7. 对于任何 C_i 到 C_{i-1} 中的读 r，我们有 $W_i(r)$ 在 C_{i-1} 以及 W(r) 在 C_{i-1}

为 E_i 给出一组充分的同步-与边缘，如果有一个释放-获取对 happens-before (§17.4.5)你正在提交的操作，则该对必须出现在所有 E_j 中，其中 $j \geq i$。理论上：

8. 设 ssw_i 是也在 hb_i 的传递归约中但不在 po 中的 sw_i 边。我们称 ssw_i 为 E_i 的充分的同步-与边缘。如果 $ssw_i(x, y)$ 并且 $hb_i(y, z)$ 并且 z 在 C_i 中，那么对所有 $j \geq i$，$sw_j(x, y)$。

   如果提交了操作 y，那么在 y 之前发生的所有外部操作也会提交。

9. 如果 y 在 C_i 中，x 是一个外部操作并且 $hb_i(x, y)$，那么 x 在 C_i 中。

例子 17.4.8-1. Happens-before 一致性是不够的

Happens-before 是一组必要但不充分的约束。仅仅强制执行 happens-before 将允许不可接受的行为-那些违反我们为程序建立的要求的行为。例如，happens-before 一致性允许值"凭空"出现。这可以通过表 17.4.8-A 中跟踪的详细检查看到。

表 17.4.8-A. Happens-before 一致性是不够的

线程 1	线程 2
`r1 = x;`	`r2 = y;`

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

表 17.4.8-A 所示的代码已正确同步。这看起来很奇怪，因为它不执行任何同步操作。但是，请记住，如果以顺序一致的方式执行程序时，没有数据竞争，则程序是正确同步的。如果以顺序一致的方式执行此代码，则每个操作将按程序顺序执行，并且两次写入都不会发生。由于不发生写入，因此不存在数据竞争：程序已正确同步。

由于此程序已正确同步，因此我们只能允许顺序一致的行为。然而，这个程序的执行是 happens-before 一致，但不是顺序一致：

```
r1 = x; // sees write of x = 1
y = 1;
r2 = y; // sees write of y = 1
x = 1;
```

结果是 happens-before 一致：没有防止它发生的 happens-before 关系。然而，这显然是不可接受的：没有顺序一致的执行会导致这种行为。我们允许读看到执行顺序后面的写，这一事实有时会导致不可接受的行为。

虽然有时不希望允许读取看到执行顺序后面的写入，但有时也是必要的。正如我们在上面看到的，表 17.4.5-A 中的跟踪需要一些读取来查看执行顺序中稍后发生的写入。由于读取在每个线程中首先出现，所以执行顺序中的第一个操作必须是读取。如果该读取无法看到稍后发生的写入，则它无法看到其读取的变量的初始值以外的任何值。这显然不是所有行为的反映。

我们提到读取何时可以将未来的写入视为因果关系的问题，因为在如表 17.4.8-A 所示的情况下会出现问题。在这种情况下，读操作导致写操作发生，写操作导致读操作发生。这些操作没有“首要原因”。因此，我们的内存模型需要一种一致的方式来确定哪些读操作可以尽早看到写操作。

如表 17.4.8-A 所示的例子表明，当说明一个读是否可以看到执行过程中稍后发生的写操作时，规范必须非常小心(请记住，如果一个读看到执行过程中稍后发生的写操作，这就表示写操作实际上是在较早的时候执行的)。

内存模型将给定的执行和程序作为输入，并确定该执行是否合法。它通过逐步构建一组“已提交”的操作来实现这一点，这些操作反映了程序执行了哪些操作。通常，要提交的下一个操作将反映可由顺序一致执行执行的下一个操作。但是，为了反映需要查看后续写入的读取，我们允许一些操作被提交早于happen-before 它们的其他操作。

显然，有些操作可能会提前提交，而有些则不会。例如，如果表 17.4.8-A 中的一次写操作在读取该变量之前提交，则读取操作可以看到写入操作，并且可能出现“凭空”的结果。非正式地说，如果我们知道可以在不假设发生某些数据竞争的情况下发生某个操作，我们就允许提前提交该操作。在表 17.4.8-A 中，我们不能提前执行写入，因为除非读取看到数据竞争的结果，否则写入不会发生。

17.4.9 可观察行为和非终止执行

对于总是在某个有界有限时间段内终止的程序，其行为可以简单地（非正式地）根据其允许的执行来理解。对于无法在有限时间内终止的程序，会出现更微妙的问题。

程序的可观察行为由程序可能执行的有限组外部操作定义。例如，简单地永远打印“Hello”的程序由一组行为描述，对于任何非负整数 i，包括打印“Hello”i 次的行为。

终止没有显式地建模为一种行为，但是可以很容易地扩展程序来生成一个额外的外部操作 executionterminate，该操作在所有线程都终止时发生。

我们还定义了一个特殊的挂起操作。如果行为是由一组外部操作(包括一个挂起操作)描述的，那么它表明，在观察到外部操作之后，程序可以无限制地运行一段时间，而不执行任何额外的外部操作或终止。如果所有线程都被阻塞，或者如果程序可以执行无限数量的操作而不执行任何外部操作，程序就会挂起。

线程可以在各种情况下阻塞，例如当它试图获取锁或执行依赖于外部数据的外部操作(如读取)时。

执行可能会导致线程被无限阻塞，并且执行不会终止。在这种情况下，被阻塞线程生成的操作必须包含该线程生成的所有操作，直到并包括导致该线程被阻塞的操作，并且在该操作之后线程将不会生成任何操作。

为了推理可观察到的行为，我们需要讨论一系列可观察到的操作。

如果 O 是执行 E 的可观察操作的集合，那么集合 O 必须是 E 的操作的子集，并且必须只包含有限数量的操作，即使 A 包含无限数量的操作。此外，如果操作 y 在 O 中，并且 hb(x, y)或 so(x, y)，那么 x 就在 O 中。

注意，一组可观察的操作并不局限于外部操作。相反，只有在一组可观察操作中的外部操作才被认为是可观察的外部操作。

行为 B 是程序 P 允许的行为，当且仅当 B 是一个有限的外部操作集，并且:

- P 有一个执行 E, E 有一个可观察操作的集合 O, B 是 O 中的外部操作集合(如果 E 中的任何线程以阻塞状态结束，O 包含 E 中的所有操作，那么 B 也可能包含一个挂起动作);或者
- 存在一组操作 O，其中 B 由一个挂起操作加上 O 中的所有外部操作组成，对于所有 k≥|O |，存在带操作 A 的 P 的执行 E，存在一组操作 O'，使得:
  - O 和 O' 都是 A 的子集，它们满足可观察操作集的需求。
  - O ⊆ O'⊆A
  - |O'| ⩾ k
  - O' - O 不包含外部操作

  注意，行为 B 并不描述观察 B 中的外部操作的顺序，但是关于外部操作如何生成和执行的其他(内部)约束可能会施加这样的限制。

17.5 final 字段语义

声明为 final 的字段只初始化一次，但在正常情况下从不更改。final 字段的详细语义与普通字段有所不同。特别是，编译器有很大的自由，可以跨同步障碍移动 final 字段的读取和对任意或未知方法的调用。相应地，允许编译器将 final 字段的值缓存在寄存器中，在需要重新加载非 final 字段的情况下，不从内存中重新加载它。

final 字段还允许程序员实现线程安全的不可变对象，而无需同步。线程安全的不可变对象

被所有线程视为不可变的，即使使用数据竞争在线程之间传递对不可变对象的引用。这可以提供安全保证，防止不可变类被错误或恶意代码误用。必须正确使用 final 字段，以保证不变性。

当一个对象的构造函数完成时，该对象被认为是完全初始化的。一个线程只能在对象被完全初始化之后才能看到对象的引用，这保证可以看到该对象的 final 字段的正确初始值。

final 字段的使用模型很简单: 在对象的构造函数中设置对象的 final 字段; 并且在对象的构造函数完成之前，不要在另一个线程可以看到的地方写入对正在构建的对象的引用。如果遵循此操作，则当另一个线程看到该对象时，该线程将始终看到该对象 final 字段的正确构造版本。

它还将看到 final 字段引用的任何对象或数组的版本，这些版本至少与 final 字段一样最新。

例子 17.5-1. Java 内存模型中的 final 字段

下面的程序说明了 final 字段与正常字段的比较。

```
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
       }
    }
}
```


类 FinalFieldExample 有一个 final int 字段 x 和一个非 final int 字段 y。一个线程可能执行方法 writer，另一个可能执行方法 reader。

因为 writer 方法在对象的构造函数完成之后写 f, reader 方法保证能看到 f.x 的合适的初始化值：它将读到值 3。然而 f.y 不是 final; 因此，不能保证 reader 方法看到它的值 4。

例子 17.5-2. 安全的 final 字段

final 字段被设计为允许必要的安全保障。考虑以下程序。一个线程（我们将其称为线程 1）执行:

```
Global.s = "/tmp/usr".substring(4);
```

而另一个线程（线程 2）执行

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

String 对象是不可变的，字符串操作不执行同步。虽然 String 实现不存在任何数据竞争，但其他代码可能存在涉及使用 String 对象的数据竞争，并且内存模型对具有数据竞争的程序做出了弱保证。特别是，如果 String 类的字段不是 final，那么线程 2 可能（尽管不太可能）最初会看到字符串对象偏移量的默认值 0，从而允许它与“/tmp”进行比较。稍后对 String 对象的操作可能会看到正确的偏移量为 4，因此 String 对象被视为“/usr”。 Java 编程语言的许多安全特性依赖于 String 对象被视为真正不可变的，即使恶意代码正在使用数据竞争在线程之间传递 String 引用。

17.5.1 final 字段的语义

设 o 为对象，c 为 o 的构造函数，其中写入了 final 字段 f。当 c 正常或突然退出时，在 o 的 final 字段 f 上发生冻结操作。

请注意，如果一个构造函数调用另一个构造函数，并且被调用的构造函数设置了一个 final 字段，那么 final 字段的冻结将发生在被调用构造函数的末尾。

对于每个执行，读取行为受两个附加的偏序影响，即解引用链 dereferences()和内存链 mc(), 它们被视为执行的一部分（因此，对于任何特定的执行都是固定的）。这些偏序必须满足以下约束（无需具有唯一解）：

- 解引用链: 如果操作 a 是未初始化 o 的线程 t 对对象 o 的字段或元素的读或写，则一定存在线程 t 看到 o 的地址的某个读 r，使得 r dereferences(r, a)。
- 内存链: 内存链排序有几个约束:
  - 如果 r 是看到写 w 的读，那么 mc(w, r)成立。
  - 如果 r 和 a 是操作使得 dereferences(r, a), 那么 mc(r, a)成立。
  - 如果 w 是未初始化 o 的线程 t 对对象 o 的地址的写入，则一定存在线程 t 看到 o 的地址的某个读 r，使得 mc(r, w)。

给定写入 w、冻结 f、操作 a（不是对 final 字段的读取）、由 f 冻结的 final 字段的读 r_1 和读 r_2，使得 hb（w，f）、hb（f，a）、mc（a，r_1）和 dereferences(r_1, r_2)，那么在确定 r_2 可以看到哪些值时，我们考虑 hb（w，r_2）。（这 happens-before 排序，不会与其他 happens-before 排序传递关闭。）

请注意，解引用的顺序是自反式的，并且 r_1 可以与 r_2 相同。

对于 final 字段的读取，唯一被认为是在读取 final 字段之前进行的写入是通过 final 字段语义派生的写入。

17.5.2 构造期间读 final 字段

构造该对象的线程内的对象的 final 字段的读取是根据 happens-before 规则对构造函数内该字段的初始化排序的。如果读取发生在构造函数中设置字段之后，它会看到 final 字段被赋值，否则会看到默认值。

17.5.3 final 字段的后续修改

在某些情况下，例如反序列化，系统将需要在构造后更改对象的 final 字段。final 字段可以通过反射和其他依赖于实现的方法进行更改。唯一具有合理语义的模式是，先构造一个对象，然后更新对象的 final 字段。该对象不应该对其他线程可见，也不应该读取 final 字段，直到所有对该对象 final 字段的更新完成。final 字段的冻结既发生在设置 final 字段的构造函数的末尾，也发生在通过反射或其他特殊机制对 final 字段进行每次修改之后。

即便如此，仍有许多复杂的问题。如果在字段声明中，将 final 字段初始化为常量表达式(§15.29)，则不能观察到对 final 字段的更改，因为对 final 字段的使用在编译时被替换为常量表达式的值。

另一个问题是规范允许对 final 字段进行积极的优化。在线程中，允许用那些在构造函数中没有发生的 final 字段修改来重新排序 final 字段的读取操作。

例子 17.5.3-1. final 字段的主动优化

```
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
    }
}
```


在 d 方法中，允许编译器自由地对 x 的读取和对 g 的调用重新排序。因此, new A().f()可以返回 -1, 0, 或 1。

实现可以提供一种方法来在 final 字段安全上下文中执行代码块。如果一个对象是在 final 字段安全上下文中构造的，那么对该对象 final 字段的读取将不会与在 final 字段安全上下文中发生的 final 字段的修改一起重新排序。

一个 final 字段安全上下文具有额外的保护。如果一个线程看到了一个错误发布的对象引用，该对象允许线程看到 final 字段的默认值，然后，在 final 字段安全的上下文中，读取一个正确发布的对象引用，它将保证看到 final 字段的正确值。在形式主义中，在 final 字段安全上下文中执行的代码被视为一个单独的线程(仅针对 final 字段语义)。

在实现中，编译器不应该将对 final 字段的访问移进或移出 final 字段安全的上下文中(尽管它可以在这样的上下文的执行中移动，只要对象不是在该上下文中构造的)。

一个适合使用 final 字段安全上下文的地方是执行器或线程池。通过在单独的 final 字段安全上下文中执行每个 Runnable，执行器可以保证一个 Runnable 对对象 o 的不正确访问不会移除对由同一执行器处理的其他 Runnable 的 final 字段保证。

17.5.4 写保护字段

正常情况下，不能修改 final 和静态的字段。然而, System.in, System.out, 和 System.err 是静态 final 字段, 由于遗留原因, 必须允许由方法 System.setIn, System.setOut,和 System.setErr 更改。我们将这些字段称为写保护字段，以将它们与普通的 final 字段区分开来。

编译器需要将这些字段与其他 final 字段区别对待。例如，对普通 final 字段的读取不受同步的影响：锁或 volatile 读取所涉及的障碍不必影响从 final 字段读取的值。由于写保护字段的值可能会发生更改，因此同步事件应该会对它们产生影响。因此，语义规定这些字段被视为不能被用户代码更改的普通字段，除非该用户代码在 System 类中。

17.6 撕字

Java 虚拟机实现的一个考虑因素是每个字段和数组元素都被认为是不同的;对一个字段或元素的更新不能与任何其他字段或元素的读取或更新交互。特别地，两个单独更新字节数组相邻元素的线程不能相互干扰或交互，也不需要同步来确保顺序的一致性。

有些处理器不提供写入单个字节的能力。在这样的处理器上实现字节数组更新是不合法的，只需要读取整个字，更新相应的字节，然后将整个字写回内存。这个问题有时被称为字撕裂，对于不能轻松地单独更新单个字节的处理器，将需要一些其他方法。

例子 17.6-1. 单词撕裂检测

下面的程序是一个检测撕字的测试用例:

```
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();
    }
}
```

这表明字节一定不能被写入相邻字节的操作覆盖。

17.7 double 和 long 的非原子处理

出于 Java 编程语言内存模型的目的，对非 volatile 的 long 或 double 的一次写操作被视为两次单独的写操作:对每个 32 位的一半进行一次写操作。这可能导致这样一种情况:线程从一次写操作中看到 64 位值的前 32 位，从另一次写操作中看到后 32 位。

对 volatile long 和 double 值的写和读总是原子的。

对引用的写和读始终是原子性的，不管它们是实现为 32 位值还是 64 位值。

> 一些实现可能会发现，将 64 位 long 或 double 上的单个写操作划分为相邻的 32 位值上的两个写操作非常方便。为了提高效率，这种行为是特定于实现的;Java 虚拟机的一个实现可以自由地对 long 和 double 执行写操作，原子地或者分两部分执行。
>
> 鼓励 Java 虚拟机的实现尽可能避免拆分 64 位值。鼓励程序员将共享的 64 位值声明为 volatile，或者正确地同步他们的程序，以避免可能的复杂性。