Java 语言中的 volatile 变量可以被看作是一种 “程度较轻的 synchronized”；与 synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是 synchronized 的一部分。本文介绍了几种有效使用 volatile 变量的模式，并强调了几种不适合使用 volatile 变量的情形。

锁提供了两种主要特性：*互斥（mutual exclusion）* 和*可见性（visibility）*。互斥即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。可见性要更加复杂一些，它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的 —— 如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题。

**Volatile 变量**

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子特性。这就是说线程能够自动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。因此，单独使用 volatile 还不足以实现计数器、互斥锁或任何具有与多个变量相关的不变式（Invariants）的类（例如 “start <=end”）。

出于简易性或可伸缩性的考虑，您可能倾向于使用 volatile 变量而不是锁。当使用 volatile 变量而非锁时，某些习惯用法（idiom）更加易于编码和阅读。此外，volatile 变量不会像锁那样造成线程阻塞，因此也很少造成可伸缩性问题。在某些情况下，如果读操作远远大于写操作，volatile 变量还可以提供优于锁的性能优势。

**正确使用 volatile 变量的条件**

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

* 对变量的写操作不依赖于当前值。
* 该变量没有包含在具有其他变量的不变式中。

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

第一个条件的限制使 volatile 变量不能用作线程安全计数器。虽然增量操作（x++）看上去类似一个单独操作，实际上它是一个由读取－修改－写入操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使 x 的值在操作期间保持不变，而 volatile 变量无法实现这点。（然而，如果将值调整为只从单个线程写入，那么可以忽略第一个条件。）

大多数编程情形都会与这两个条件的其中之一冲突，使得 volatile 变量不能像 synchronized 那样普遍适用于实现线程安全。清单 1 显示了一个非线程安全的数值范围类。它包含了一个不变式 —— 下界总是小于或等于上界。

**清单 1. 非线程安全的数值范围类**

@NotThreadSafe

public class NumberRange {

private int lower, upper;

public int getLower() { return lower; }

public int getUpper() { return upper; }

public void setLower(int value) {

if (value > upper)

throw new IllegalArgumentException(...);

lower = value;

}

public void setUpper(int value) {

if (value < lower)

throw new IllegalArgumentException(...);

upper = value;

}

}

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全；从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态。例如，如果初始状态是(0, 5)，同一时间内，线程 A 调用 setLower(4) 并且线程 B 调用 setUpper(3)，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是 (4, 3) —— 一个无效值。至于针对范围的其他操作，我们需要使 setLower() 和setUpper() 操作原子化 —— 而将字段定义为 volatile 类型是无法实现这一目的的。

**性能考虑**

使用 volatile 变量的主要原因是其简易性：在某些情形下，使用 volatile 变量要比使用相应的锁简单得多。使用 volatile 变量次要原因是其性能：某些情况下，volatile 变量同步机制的性能要优于锁。

很难做出准确、全面的评价，例如 “X 总是比 Y 快”，尤其是对 JVM 内在的操作而言。（例如，某些情况下 VM 也许能够完全删除锁机制，这使得我们难以抽象地比较 volatile 和 synchronized 的开销。）就是说，在目前大多数的处理器架构上，volatile 读操作开销非常低 —— 几乎和非 volatile 读操作一样。而 volatile 写操作的开销要比非 volatile 写操作多很多，因为要保证可见性需要实现内存界定（Memory Fence），即便如此，volatile 的总开销仍然要比锁获取低。

volatile 操作不会像锁一样造成阻塞，因此，在能够安全使用 volatile 的情况下，volatile 可以提供一些优于锁的可伸缩特性。如果读操作的次数要远远超过写操作，与锁相比，volatile 变量通常能够减少同步的性能开销。

**正确使用 volatile 的模式**

很多并发性专家事实上往往引导用户远离 volatile 变量，因为使用它们要比使用锁更加容易出错。然而，如果谨慎地遵循一些良好定义的模式，就能够在很多场合内安全地使用 volatile 变量。要始终牢记使用 volatile 的限制 —— 只有在状态真正独立于程序内其他内容时才能使用 volatile —— 这条规则能够避免将这些模式扩展到不安全的用例。

**模式 #1：状态标志**

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

很多应用程序包含了一种控制结构，形式为 “在还没有准备好停止程序时再执行一些工作”，如清单 2 所示：

**清单 2. 将 volatile 变量作为状态标志使用**

volatile boolean shutdownRequested;

...

public void shutdown() { shutdownRequested = true; }

public void doWork() {

while (!shutdownRequested) {

// do stuff

}

}

很可能会从循环外部调用 shutdown() 方法 —— 即在另一个线程中 —— 因此，需要执行某种同步来确保正确实现 shutdownRequested 变量的可见性。（可能会从 JMX 侦听程序、GUI 事件线程中的操作侦听程序、通过 RMI 、通过一个 Web 服务等调用）。然而，使用 synchronized块编写循环要比使用清单 2 所示的 volatile 状态标志编写麻烦很多。由于 volatile 简化了编码，并且状态标志并不依赖于程序内任何其他状态，因此此处非常适合使用 volatile。

这种类型的状态标记的一个公共特性是：通常只有一种状态转换；shutdownRequested 标志从 false 转换为 true，然后程序停止。这种模式可以扩展到来回转换的状态标志，但是只有在转换周期不被察觉的情况下才能扩展（从 false 到 true，再转换到 false）。此外，还需要某些原子状态转换机制，例如原子变量。

**模式 #2：一次性安全发布（one-time safe publication）**

缺乏同步会导致无法实现可见性，这使得确定何时写入对象引用而不是原语值变得更加困难。在缺乏同步的情况下，可能会遇到某个对象引用的更新值（由另一个线程写入）和该对象状态的旧值同时存在。（这就是造成著名的双重检查锁定（double-checked-locking）问题的根源，其中对象引用在没有同步的情况下进行读操作，产生的问题是您可能会看到一个更新的引用，但是仍然会通过该引用看到不完全构造的对象）。

实现安全发布对象的一种技术就是将对象引用定义为 volatile 类型。清单 3 展示了一个示例，其中后台线程在启动阶段从数据库加载一些数据。其他代码在能够利用这些数据时，在使用之前将检查这些数据是否曾经发布过。

**清单 3. 将 volatile 变量用于一次性安全发布**

public class BackgroundFloobleLoader {

public volatile Flooble theFlooble;

public void initInBackground() {

// do lots of stuff

theFlooble = new Flooble(); // this is the only write to theFlooble

}

}

public class SomeOtherClass {

public void doWork() {

while (true) {

// do some stuff...

// use the Flooble, but only if it is ready

if (floobleLoader.theFlooble != null)

doSomething(floobleLoader.theFlooble);

}

}

}

如果 theFlooble 引用不是 volatile 类型，doWork() 中的代码在解除对 theFlooble 的引用时，将会得到一个不完全构造的 Flooble。

该模式的一个必要条件是：被发布的对象必须是线程安全的，或者是有效的不可变对象（有效不可变意味着对象的状态在发布之后永远不会被修改）。volatile 类型的引用可以确保对象的发布形式的可见性，但是如果对象的状态在发布后将发生更改，那么就需要额外的同步。

**模式 #3：独立观察（independent observation）**

安全使用 volatile 的另一种简单模式是：定期 “发布” 观察结果供程序内部使用。例如，假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值。

使用该模式的另一种应用程序就是收集程序的统计信息。清单 4 展示了身份验证机制如何记忆最近一次登录的用户的名字。将反复使用lastUser 引用来发布值，以供程序的其他部分使用。

**清单 4. 将 volatile 变量用于多个独立观察结果的发布**

public class UserManager {

public volatile String lastUser;

public boolean authenticate(String user, String password) {

boolean valid = passwordIsValid(user, password);

if (valid) {

User u = new User();

activeUsers.add(u);

lastUser = user;

}

return valid;

}

}

该模式是前面模式的扩展；将某个值发布以在程序内的其他地方使用，但是与一次性事件的发布不同，这是一系列独立事件。这个模式要求被发布的值是有效不可变的 —— 即值的状态在发布后不会更改。使用该值的代码需要清楚该值可能随时发生变化。

**模式 #4：“volatile bean” 模式**

volatile bean 模式适用于将 JavaBeans 作为“荣誉结构”使用的框架。在 volatile bean 模式中，JavaBean 被用作一组具有 getter 和/或 setter 方法 的独立属性的容器。volatile bean 模式的基本原理是：很多框架为易变数据的持有者（例如 HttpSession）提供了容器，但是放入这些容器中的对象必须是线程安全的。

在 volatile bean 模式中，JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter 方法必须非常普通 —— 除了获取或设置相应的属性外，不能包含任何逻辑。此外，对于对象引用的数据成员，引用的对象必须是有效不可变的。（这将禁止具有数组值的属性，因为当数组引用被声明为 volatile 时，只有引用而不是数组本身具有 volatile 语义）。对于任何 volatile 变量，不变式或约束都不能包含 JavaBean 属性。清单 5 中的示例展示了遵守 volatile bean 模式的 JavaBean：

**清单 5. 遵守 volatile bean 模式的 Person 对象**

@ThreadSafe

public class Person {

private volatile String firstName;

private volatile String lastName;

private volatile int age;

public String getFirstName() { return firstName; }

public String getLastName() { return lastName; }

public int getAge() { return age; }

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public void setAge(int age) {

this.age = age;

}

}

**volatile 的高级模式**

前面几节介绍的模式涵盖了大部分的基本用例，在这些模式中使用 volatile 非常有用并且简单。这一节将介绍一种更加高级的模式，在该模式中，volatile 将提供性能或可伸缩性优势。

volatile 应用的的高级模式非常脆弱。因此，必须对假设的条件仔细证明，并且这些模式被严格地封装了起来，因为即使非常小的更改也会损坏您的代码！同样，使用更高级的 volatile 用例的原因是它能够提升性能，确保在开始应用高级模式之前，真正确定需要实现这种性能获益。需要对这些模式进行权衡，放弃可读性或可维护性来换取可能的性能收益 —— 如果您不需要提升性能（或者不能够通过一个严格的测试程序证明您需要它），那么这很可能是一次糟糕的交易，因为您很可能会得不偿失，换来的东西要比放弃的东西价值更低。

**模式 #5：开销较低的读－写锁策略**

目前为止，您应该了解了 volatile 的功能还不足以实现计数器。因为 ++x 实际上是三种操作（读、添加、存储）的简单组合，如果多个线程凑巧试图同时对 volatile 计数器执行增量操作，那么它的更新值有可能会丢失。

然而，如果读操作远远超过写操作，您可以结合使用内部锁和 volatile 变量来减少公共代码路径的开销。清单 6 中显示的线程安全的计数器使用synchronized 确保增量操作是原子的，并使用 volatile 保证当前结果的可见性。如果更新不频繁的话，该方法可实现更好的性能，因为读路径的开销仅仅涉及 volatile 读操作，这通常要优于一个无竞争的锁获取的开销。

**清单 6. 结合使用 volatile 和 synchronized 实现 “开销较低的读－写锁”**

@ThreadSafe

public class CheesyCounter {

// Employs the cheap read-write lock trick

// All mutative operations MUST be done with the 'this' lock held

@GuardedBy("this") private **volatile** int value;

public int getValue() { return value; }

public **synchronized** int increment() {

return value++;

}

}

之所以将这种技术称之为 “开销较低的读－写锁” 是因为您使用了不同的同步机制进行读写操作。因为本例中的写操作违反了使用 volatile 的第一个条件，因此不能使用 volatile 安全地实现计数器 —— 您必须使用锁。然而，您可以在读操作中使用 volatile 确保当前值的*可见性*，因此可以使用锁进行所有变化的操作，使用 volatile 进行只读操作。其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作，因此当使用 volatile 保证读代码路径时，要比使用锁执行全部代码路径获得更高的共享度 —— 就像读－写操作一样。然而，要随时牢记这种模式的弱点：如果超越了该模式的最基本应用，结合这两个竞争的同步机制将变得非常困难。

**结束语**

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件 —— 即变量真正独立于其他变量和自己以前的值 —— 在某些情况下可以使用 volatile 代替 synchronized 来简化代码。然而，使用 volatile 的代码往往比使用锁的代码更加容易出错。本文介绍的模式涵盖了可以使用 volatile 代替 synchronized 的最常见的一些用例。遵循这些模式（注意使用时不要超过各自的限制）可以帮助您安全地实现大多数用例，使用 volatile 变量获得更佳性能。