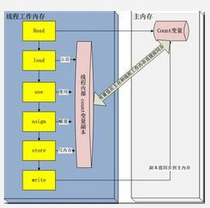
推荐一个定义为volatile的[变量](http://baike.so.com/doc/2977483-3140702.html" \t "_blank)是说这变量可能会被意想不到地改变，这样，编译器就不会[volatile](http://p9.qhmsg.com/t016f3253d9a3e66fbb.png)去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在[寄存器](http://baike.so.com/doc/1248822-1320752.html" \t "_blank)里的备份。下面是volatile变量的几个例子：

1）. 并行设备的硬件寄存器（如：状态寄存器）

2）. 一个中断服务子程序中会访问到的非自动变量（Non-automatic variables)

3）. 多线程应用中被几个任务共享的变量

我认为这是区分C程序员和[嵌入式系统](http://baike.so.com/doc/1201766-1271169.html" \t "_blank)程序员的最基本的问题。嵌入式系统程序员经常同硬件、中断、RTOS等等打交道，所用这些都要求volatile变量。不懂得volatile内容将会带来灾难。

假设被面试者正确地回答了这是问题（嗯，怀疑是否会是这样），我将稍微深究一下，看一下这家伙是不是真正懂得volatile完全的重要性。

1）. 一个参数既可以是const还可以是volatile吗？解释为什么。

2）. 一个指针可以是volatile 吗？解释为什么。

3）. 下面的函数有什么错误：

int square(volatile int \*ptr)

{

return \*ptr \* \*ptr;

}

下面是答案：

1）. 是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。

2）. 是的。尽管这并不很常见。一个例子是当一个中断服务子程序修改一个指向一个buffer的[指针](http://baike.so.com/doc/1043844-1104112.html" \t "_blank)时。

3）. 这段代码是个恶作剧。这段代码的目的是用来返指针\*ptr指向值的平方，但是，由于\*ptr指向一个volatile型参数，编译器将产生类似下面的代码：

int square(volatile int \*ptr)

{

int a,b;

a = \*ptr;

b = \*ptr;

return a \* b;

}

由于\*ptr的值可能被意想不到地改变，因此a和b可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

long square(volatile int \*ptr)

{

int a;

a = \*ptr;

return a \* a;

}

讲讲我的理解：（欢迎打板子...~~！）

关键在于两个地方：

⒈ 编译器的优化（请高手帮我看看下面的理解）

在本次线程内，当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后，再取变量值时，就直接从寄存器中取值；

当变量值在本线程里改变时，会同时把变量的新值copy到该寄存器中，以便保持一致

当变量在因别的线程等而改变了值，该寄存器的值不会相应改变，从而造成[应用程序](http://baike.so.com/doc/3417785-3597266.html)读取的值和实际的变量值不一致

当该寄存器在因别的线程等而改变了值，原变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致

举一个不太准确的例子：

发薪资时，会计每次都把员工叫来登记他们的银行卡号；一次会计为了省事，没有即时登记，用了以前登记的银行卡号；刚好一个员工的银行卡丢了，已挂失该银行卡号；从而造成该员工领不到工资

员工 －－ 原始变量地址

银行卡号 －－ 原始变量在寄存器的备份

⒉ 在什么情况下会出现（如1楼所说）

1）. 并行设备的硬件寄存器（如：状态寄存器）

2）. 一个中断服务子程序中会访问到的非自动变量（Non-automatic variables)

3）. 多线程应用中被几个任务共享的变量

补充：volatile应该解释为“直接存取原始内存地址”比较合适，“易变的”这种解释简直有点误导人；

“易变”是因为外在因素引起的，象多线程，中断等，并不是因为用volatile修饰了的变量就是“易变”了，假如没有外因，即使用volatile定义，它也不会变化；

而用volatile定义之后，其实这个变量就不会因外因而变化了，可以放心使用了； 大家看看前面那种解释（易变的）是不是在误导人

－－－－－－－－－－－－简明示例如下：－－－－－－－－－－－－－－－－－－

volatile[关键字](http://baike.so.com/doc/4227659-4429378.html)是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

使用该关键字的例子如下：

int volatile nVint;

>>>>；当要求使用volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

例如：

volatile int i=10;

int a = i;

...

//其他代码，并未明确告诉编译器，对i进行过操作

int b = i;

>>>>volatile 指出 i是随时可能发生变化的，每次使用它的时候必须从i的地址中读取，因而编译器生成的汇编代码会重新从i的地址读取数据放在b中。而优化做法是，由于编译器发现两次从i读数据的代码之间的代码没有对i进行过操作，它会自动把上次读的数据放在b中。而不是重新从i里面读。这样一来，如果i是一个寄存器变量或者表示一个端口数据就容易出错，所以说volatile可以保证对特殊地址的稳定访问。

>>>>；注意，在vc6中，一般调试模式没有进行代码优化，所以这个关键字的作用看不出来。下面通过插入汇编代码，测试有无volatile关键字，对程序最终代码的影响：

>>>>；首先，用classwizard建一个win32 console工程，插入一个voltest.cpp文件，输入下面的代码：

>>

#i nclude <stdio.h>

void main()

{

int i=10;

int a = i;

printf("i= %d",a);

//下面汇编语句的作用就是改变内存中i的值，但是又不让编译器知道

\_\_asm {

mov dword ptr [ebp-4],20h

}

int b = i;

printf("i= %d",b);

}

然后，在调试版本模式运行程序，输出结果如下：

i = 10

i = 32

然后，在release版本模式运行程序，输出结果如下：

i = 10

i = 10

输出的结果明显表明，release模式下，编译器对代码进行了优化，第二次没有输出正确的i值。下面，我们把 i的声明加上volatile关键字，看看有什么变化：

#i nclude <stdio.h>

void main()

{

volatile int i=10;

int a = i;

printf("i= %d",a);

\_\_asm {

mov dword ptr [ebp-4],20h

}

int b = i;

printf("i= %d",b);

}

分别在调试版本和release版本运行程序，输出都是：

i = 10

i = 32

这说明这个关键字发挥了它的作用！

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

volatile对应的变量可能在你的程序本身不知道的情况下发生改变

比如多线程的程序，共同访问的内存当中，多个程序都可以操纵这个变量

你自己的程序，是无法判定何时这个变量会发生变化

还比如，他和一个外部设备的某个状态对应，当外部设备发生操作的时候，通过[驱动程序](http://baike.so.com/doc/104837-110638.html)和中断事件，系统改变了这个变量的数值，而你的程序并不知道。

对于volatile类型的变量，系统每次用到他的时候都是直接从对应的内存当中提取，而不会利用cache当中的原有数值，以适应它的未知何时会发生的变化，系统对这种变量的处理不会做优化——显然也是因为它的数值随时都可能变化的情况。

--------------------------------------------------------------------------------

典型的例子

for (int i=0; i<100000; i++);

这个语句用来测试空循环的速度的

但是编译器肯定要把它优化掉，根本就不执行

如果你写成

for (volatile int i=0; i<100000; i++);

它就会执行了

volatile的本意是“易变的”

由于访问寄存器的速度要快过RAM，所以编译器一般都会作减少存取外部RAM的优化。比如：

static int i=0;

int main(void)

{

...

while ⑴

{

if (i) dosomething();

}

}

/\* Interrupt service routine. \*/

void ISR\_2(void)

{

i=1;

}

程序的本意是希望ISR\_2中断产生时，在main当中调用dosomething函数，但是，由于编译器判断在main函数里面没有修改过i，因此

可能只执行一次对从i到某寄存器的读操作，然后每次if判断都只使用这个寄存器里面的“i副本”，导致dosomething永远也不会被

调用。如果将变量加上volatile修饰，则编译器保证对此变量的读写操作都不会被优化（肯定执行）。此例中i也应该如此说明。

[折叠](http://baike.so.com/doc/1171999-1239710.html)volatile一般使用的地方

[折叠](http://baike.so.com/doc/1171999-1239710.html)volatile代码说明

[折叠](http://baike.so.com/doc/1171999-1239710.html)[编辑本段](http://baike.so.com/create/edit/?eid=1171999&sid=1239710&secid=4)相关信息

[折叠](http://baike.so.com/doc/1171999-1239710.html)如何在java中正确使用volatile

Java? 语言包含两种内在的同步机制：同步块（或方法）和 volatile 变量。这两种机制的提出都是为了实现代码线程的安全性。其中 Volatile [变量](http://baike.so.com/doc/2977483-3140702.html)的同步性较差（但有时它更简单并且开销更低），而且其使用也更容易出错。在这期的 *Java 理论与实践*中，Brian Goetz 将介绍几种正确使用 volatile 变量的模式，并针对其适用性限制提出一些建议。

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](http://baike.so.com/doc/115404-121766.html" \t "_blank)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。

[折叠](http://baike.so.com/doc/1171999-1239710.html)Volatile变量

[折叠](http://baike.so.com/doc/1171999-1239710.html)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](http://baike.so.com/doc/2977483-3140702.html)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; } } |

收起

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 setLower⑷ 并且线程 B 调用 setUpper⑶，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。

[折叠](http://baike.so.com/doc/1171999-1239710.html)性能考虑

使用 volatile [变量](http://baike.so.com/doc/2977483-3140702.html)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。

[折叠](http://baike.so.com/doc/1171999-1239710.html)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](http://baike.so.com/doc/2977483-3140702.html)，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](http://baike.so.com/doc/3417785-3597266.html)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); } } } |

展开

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](http://baike.so.com/doc/115404-121766.html" \t "_blank)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](http://baike.so.com/doc/1863840-1971314.html" \t "_blank)。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; } } |

展开

**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。