58 Java 中的原子操作有哪些注意事项?

本课时我们主要讲解 Java 中的原子性和原子操作。

什么是原子性和原子操作

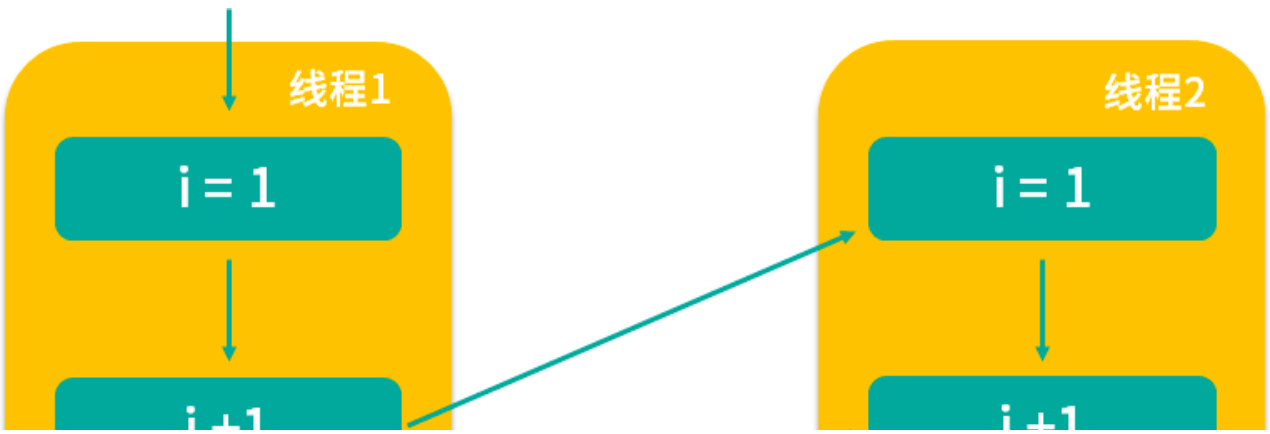
在编程中，具备原子性的操作被称为原子操作。原子操作是指一系列的操作，要么全部发生，要么全部不发生，不会出现执行一半就终止的情况。

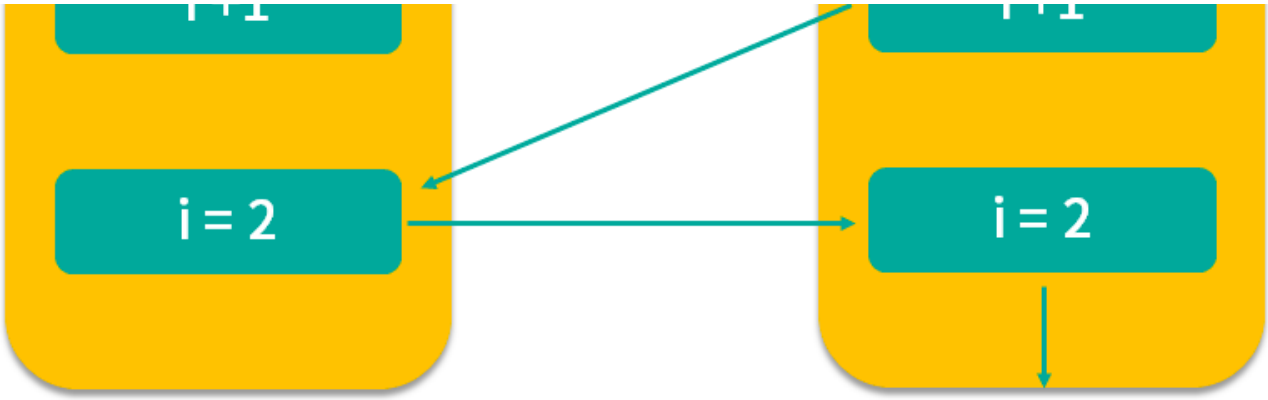
比如转账行为就是一个原子操作，该过程包含扣除余额、银行系统生成转账记录、对方余额增加等一系列操作。虽然整个过程包含多个操作，但由于这一系列操作被合并成一个原子操作，所以它们要么全部执行成功，要么全部不执行，不会出现执行一半的情况。比如我的余额已经扣除，但是对方的余额却不增加，这种情况是不会出现的，所以说转账行为是具备原子性的。而具有原子性的原子操作，天然具备线程安全的特性。

下面我们举一个不具备原子性的例子，比如 i++ 这一行代码在 CPU 中执行时，可能会从一行代码变为以下的 3 个指令：

- 第一个步骤是读取；
- 第二个步骤是增加；
- 第三个步骤是保存。

这就说明 i++ 是不具备原子性的，同时也证明了 i++ 不是线程安全的，正如第 06 课时中所介绍的那样。下面我们简单的复习一下，如何发生的线程不安全问题，如下所示：





我们根据箭头指向依次看，线程 1 首先拿到 i=1 的结果，然后进行 i+1 操作，但假设此时 i+1 的结果还没有来得及被保存下来，线程 1 就被切换走了，于是 CPU 开始执行线程 2，它所做的事情和线程 1 是一样的 i++ 操作，但此时我们想一下，它拿到的 i 是多少？实际上和线程 1 拿到的 i 结果一样，同样是 1，为什么呢？因为线程 1 虽然对 i 进行了 +1 操作，但结果没有保存，所以线程 2 看不到修改后的结果。

然后假设等线程 2 对 i 进行 +1 操作后，又切换到线程 1，让线程 1 完成未完成的操作，即将 i+1 的结果 2 保存下来，然后又切换到线程 2 完成 i=2 的保存操作，虽然两个线程都执行了对 i 进行 +1 的操作，但结果却最终保存了 i=2，而不是我们期望的 i=3，这样就发生了线程安全问题，导致数据结果错误，这也是最典型的线程安全问题。

Java 中的原子操作有哪些

在了解了原子操作的特性之后，让我们来看一下 Java 中有哪些操作是具备原子性的。Java 中的以下几种操作是具备原子性的，属于原子操作：

- 除了 long 和 double 之外的基本类型（int、byte、boolean、short、char、float）的读/写操作，都天然的具备原子性；
- 所有引用 reference 的读/写操作；
- 加了 volatile 后，所有变量的读/写操作（包含 long 和 double）。这也就意味着 long 和 double 加了 volatile 关键字之后，对它们的读写操作同样具备原子性；
- 在 java.concurrent.Atomic 包中的一部分类的一部分方法是具备原子性的，比如 AtomicInteger 的 incrementAndGet 方法。

long 和 double 的原子性

在前面，我们讲述了 long 和 double 和其他的基本类型不太一样，好像不具备原子性，这是什么原因造成的呢？ 官方文档对于上述问题的描述，如下所示：

Non-Atomic Treatment of double and long

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid possible complications.

从刚才的 JVM 规范中我们可以知道，long 和 double 的值需要占用 64 位的内存空间，而对于 64 位值的写入，可以分为两个 32 位的操作来进行。

这样一来，本来是一个整体的赋值操作，就可能被拆分为低 32 位和高 32 位的两个操作。如果在这两个操作之间发生了其他线程对这个值的读操作，就可能会读到一个错误、不完整的值。

JVM 的开发者可以自由选择是否把 64 位的 long 和 double 的读写操作作为原子操作去实现，并且规范推荐 JVM 将其实现为原子操作。当然，JVM 的开发者也有权利不这么做，这同样是符合规范的。

规范同样规定，如果使用 volatile 修饰了 long 和 double，那么其读写操作就必须具备原子性了。同时，规范鼓励程序员使用 volatile 关键字对这个问题加以控制，由于规范规定了对于 volatile long 和 volatile double 而言，JVM 必须保证其读写操作的原子性，所以加了 volatile 之后，对于程序员而言，就可以确保程序正确。

实际开发中

此时，你可能会有疑问，比如，如果之前对于上述问题不是很了解，在开发过程中没有给 long 和 double 加 volatile，好像也没有出现过问题？而且，在以后的开发过程中，是不是

必须给 long 和 double 加 volatile 才是安全的?

其实在实际开发中，读取到“半个变量”的情况非常罕见，这个情况在目前主流的 Java 虚拟机中不会出现。因为 JVM 规范虽然不强制虚拟机把 long 和 double 的变量写操作实现为原子操作，但它其实是“强烈建议”虚拟机去把该操作作为原子操作来实现的。

而在目前各种平台下的主流虚拟机的实现中，几乎都会把 64 位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要为了避免读到“半个变量”而把 long 和 double 声明为 volatile 的。

原子操作 + 原子操作 != 原子操作

值得注意的是，简单地把原子操作组合在一起，并不能保证整体依然具备原子性。比如连续转账两次的操作行为，显然不能合并当做一个原子操作，虽然每一次转账操作都是具备原子性的，但是将两次转账合为一次的操作，这个组合就不具备原子性了，因为在两次转账之间可能会插入一些其他的操作，例如系统自动扣费等，导致第二次转账失败，而且第二次转账失败并不会影响第一次转账成功。

以上就是本课时的内容，我们介绍了什么是原子性，Java 中的原子操作有哪些，并且还对 long 和 double 这一具有特殊性的情况进行了详细说明，最后我们还介绍了简单地把原子操作组合在一起，并不能保证整体依然具备原子性。

上一页 下一页