大多数现代多线程编程语言都可以避免顺序一致性与性能之间的冲突，因为它们知道：

* 顺序一致性的问题是由于某些程序转换引起的，例如我们的例子中交换了无关变量的访问顺序，这不会改变单线程程序的意图，但是会改变多线程程序的意图（例如例子中允许r1和r2都为0）。
* 只有当代码允许两个线程同时访问相同的共享数据，并且是以某种冲突的方式访问时（例如当一个线程读取数据的同时另一个线程写入该数据），才有可能察觉到这种程序转换。如果程序强制以特定顺序来访问共享变量，那么我们就无法判断对独立变量的访问是否被重排序，就如同在单线程程序中也无法判断。
* 无限制地同时访问普通共享变量会让程序变得难以处理，一般需要避免这种情况。坚持完全的顺序一致性对我们没有好处。我们将在下文用单独的一节来讨论这个问题。

因此编程语言通常会提供方便的机制来限制通过不同的线程同时访问变量，并且仅当不存在不受控的并发访问时（例如我们的示例程序）才保证顺序一致性。更准确地说：

如果两个普通内存操作访问相同的内存位置（例如变量、数组元素），并且至少一个内存操作写入该存储单元，则称这两个内存操作是冲突的。

如果程序存在顺序一致的执行，则称其允许在特定输入集合上有数据竞争（data race），即在线程操作的交错执行中两个冲突的操作可以“同时”执行。为了我们的目的，如果两个操作在交错执行中相邻，并对应不同的线程，则这两个操作可以被“同时”执行。如果两个对应不同线程的常规内存操作在交错执行中相邻，我们知道如果他们以相反顺序执行也会得到相同结果；如果一些操作强制了顺序，则他们就必须要在交错执行中间出现。因此模型中的相邻实际上意味着他们本应该在真正的并发模型中同时发生。

仅当程序避免了数据竞争时我们才保证顺序一致性。

这种保证是Java和下一代C++标准的线程编程模型的核心。

注意本文简介中的例子确实有数据竞争，至少当变量x和y是普通整型变量时是这样。

大多数现代编程语言都提供了简单的方法来指定同步变量（synchronization variables）用于在线程间通信，同步变量是特意用来进行并发访问的。这种形式的并发访问不被认为是数据竞争。换言之，只要当冲突的并发访问仅发生在同步变量上时，就能确保顺序一致性。

所以如果把我们例子中的x和y都设为同步变量，那么程序就能保证程序按预期执行。在Java中，可以将它们声明为volatile int。

将变量声明为同步变量不仅保证了变量能被不可分割地访问，还能阻止编译器和硬件以对程序可见的方式对内存访问进行重排序。这就能防止上文例子中出现r1 = r2 = 0这种结果。另一个更常见的例子如下，该例中只有标志位x\_init（初始值为false）是同步变量：

|  |  |
| --- | --- |
| Thread 1 | Thread 2 |
| x = 42; x\_init = true; | while(!x\_init) {} r1 = x; |
|

这里的思路是线程1初始化x，在实际程序中，可能比仅仅赋值为42更为复杂。然后设置x\_init标志位表明它已经完成了赋值。另一个线程2将等待x\_init标志位被设置，然后就能知道x已被初始化。

虽然x是一个普通变量，但这个程序中没有数据竞争。线程2能保证直到线程1完成并设置了x\_init之后才会执行第二条语句。所以交错执行中不可能出现x = 42和r1 = x相邻的情况。

这意味着我们确保了顺序一致的执行，即保证r1 = 42。为了保证这一点，实现时必须让线程1对x和x\_init的赋值操作对其他线程按照顺序可见，并且只有当设置了x\_init之后线程2才能开始r1 = x操作。在许多机器架构中，这两点都要求编译器遵循额外的约束，生成特殊的代码来防止潜在的硬件优化，例如防止线程1因为先访问到x\_init的内存就在对x赋值之前先对x\_init赋值。

**锁**

实际上，大多数情况下不会通过将普通变量变为同步变量来避免数据竞争，而是通过防止对普通变量的并发访问来避免数据竞争。这通常通过引入锁（有时称为互斥锁）来实现，锁是编程语言或支持库提供的特殊同步变量。例如，如果我们想维护一个共享计数器变量c，它的值可以被多个线程递增，并在线程结束时读取其值。我们可以引入一个对应的锁l，编写如下代码：

|  |  |  |
| --- | --- | --- |
| 1 | void increment\_c() | |
| 2 | { |

|  |  |  |
| --- | --- | --- |
| 3 | l.lock(); | |
| 4 | ++c; |

|  |  |  |
| --- | --- | --- |
| 5 | l.unlock(); | |
| 6 | } |

lock()和unlock()的实现确保了在这两个调用之间同时至多只能有一个线程，所以同时只有一个线程能访问c。我们可以将这看做对交错执行进行了限制：对于给定的锁l，l.lock()和l.unlock()交替调用，并且初始调用是l.lock()。（一般情况下还有一个额外的需求：锁必须由获取它的那个线程释放，但不总是这样。）因此即便increment\_c()被多个线程并发调用，在c也上是没有数据竞争的。交错执行中任何两个对c的访问都会至少被第一个线程中的unlock()和第二个线程中的lock()分开。

举例说明，假设一个程序有两个线程，每个线程都执行increment\_c()。如下是一个可接受的交错执行：

l.lock();

+c;

l.unlock();

l.lock();

++c;

l.unlock();

这个交错执行没有数据竞争。唯一的相邻的对应不同线程的操作就是中间两步。而这两步操作都是对同步变量锁l进行的，所以不会参与数据竞争。

如果我们想要重排操作来产生数据竞争，可能会是如下这种交错存取：

l.lock();

l.lock();

++c;

++c;

l.unlock();

l.unlock();

但是这种交错执行无疑是被禁止的，因为规则要求对锁l的获取和释放必须交替进行。在任何可接受的交错执行中，第二个l.lock()只有当第一个unlock()调用完成后才有可能在交错执行中出现。因此，作为唯一潜在的数据竞争，两次++c调用一定会被这两个调用分开。

这也阐释了数据竞争的另一个等价描述：数据竞争要求不同线程对相同普通共享位置的任意两次冲突访问必须被这两次访问中的同步（synchronization ）所隔离，同步确保了访问的顺序。通常这可以通过在一个线程中释放锁并在另一个线程中获取该锁来实现。但是还可以这样实现：在一个线程中设置一个整型同步变量（例如Java中的volatile），然后再第二个线程中等待该变量被设置（可能用一个简单的循环）。编程语言标准通常用这种方式来定义数据竞争。