Item 40：Use std::atomic for councurrency, volatile for special memory

Item 40: 当需要并发时使用`std::atomic`，特定内存才使用`volatile`

可伶的`volatile`。如此令人迷惑。本不应该出现在本章节，因为它没有关于并发的能力。但是在其他编程语言中（比如，Java和C#），`volatile`是有并发含义的，即使在C++中，有些编译器在实现时也将并发的某种含义加入到了`volatile`关键字中。因此在此值得讨论下关于`volatile`关键字的含义以消除异议。

开发者有时会混淆`volatile`的特性是`std::atomic`（这确实本节的内容）的模板。这种模板的实例化（比如，`std::atomic<int> , std::atomic<bool>, std::atomic<Widget*>`等）给其他线程提供了原子操作的保证。一旦`std::atomic`对象被构建，在其上的操作使用特定的机器指令实现，这比锁的实现更高效。

分析如下使用`std::atmoic`的代码：

```
std::atomic<int> ai(0); // initialize ai to 0
ai = 10; // atomically set ai to 10
std::cout << ai; // atomically read ai's value
++ai; //atomically increment ai to 11
--ai; // atomically decrement ai to 10
```

在这些语句执行过程中，其他线程读取`ai`，只能读取到0，10，11三个值其中一个。在没有其他线程修改`ai`情况下，没有其他可能。

这个例子中有两点值得注意。**首先**，在`std::cout << ai;`中，`std::atomic`只保证了对`ai`的读取时原子的。没有保证语句的整个执行是原子的，这意味着在读取`ai`与将其通过`<<`操作符写入到标准输出之间，另一个线程可能会修改`ai`的值。这对于这个语句没有影响，因为`<<`操作符是按值传递参数的（所以输出就是读取到的`ai`的值），但是重要的是要理解原子性的范围只保证了读取是原子的。

第二点值得注意的是最后两条语句---关于`ai`的加减。他们都是 read-modify-write（RMW）操作，各自原子执行。这是`std::atomic`类型的最优的特性之一：一旦`std::atomic`对象被构建，所有成员函数，包括RMW操作，对于其他线程来说保证原子执行。

相反，使用`volatile`在多线程中不保证任何事情：

```
volatile int vi(0); // initalize vi to 0
vi = 10; // set vi to 10
std::cout << vi;  // read vi's value
++vi; // increment vi to 11
--vi; // decrement vi to 10
```

代码的执行过程中，如果其他线程读取`vi`，可能读到任何值，比如-12，68，4090727。这份代码就是未定义的，因为这里的语句修改`vi`，同时其他线程读取，这就是有没有`std::atomic`或者互斥锁保护的对于内存的同时读写，这就是数据竞争的定义。

为了举一个关于在多线程程序中`std::atomic`和`volatile`表现不同的恰当例子，考虑这样一个加单的计数器，同时初始化为0：

```
std::atomic<int> ac(0);
volatile int vc(0);
```

然后我们在两个同时运行的线程中对两个计数器计数：

```
/*--------- Thread1 ---------*/      /*---------- Thread2 -----------*/
          ++ac;                                  ++ac;
                    ++vc;                                  ++vc;
```

当两个线程执行结束时，`ac`的值肯定是2，以为每个自增操作都是原子的。另一方面，`vc`的值，不一定是2，因为自增不是原子的。每个自增操作包括了读取`vc`的值，增加读取的值，然后将结果写回到`vc`。这三个操作对于`volatile`修饰的整形变量不能保证原子执行，所有可能是下面的执行顺序：

1. Thread1 读取`vc`的值，是0
2. Thread2读取`vc`的值，还是0
3. Thread1 将0加1，然后写回到`vc`
4. Thread2将0加1，然后写回到vc

`vc`的最后结果是1，即使看起来自增了两次。

不仅只有这一种执行顺序的可能，`vc`的最终结果是不可预测的，因为`vc`会发生数据竞争，标准规定数据竞争的造成的未定义行为表示编译器生成的代码可能是任何逻辑，当然，编译器不会利用这种行为来作恶。但是只有在没有数据竞争的程序中编译器的优化才有效，这些优化在存在数据竞争的程序中会造成异常和不可预测的行为。

RMW操作不是仅有的`std::atomic`在并发中有效而`volatile`无效的例子。假定一个任务计算第二个任务需要的重要值。当第一个任务完成计算，必须传递给第二个任务。Item 39表明一种使用`std::atomic<bool>`的方法来使第一个任务通知第二个任务计算完成。代码如下：

```
std::atomic<bool> valVailable(false);
auto imptValue = coputeImportantValue(); // compute  value
valAvailable = true; // tell other task it's vailable
```

人类读这份代码，能看到在`valAvailable`赋值true之前对`imptValue`赋值是重要的顺序，但是所有编译器看到的是一对没有依赖关系的赋值操作。通常来说，编译器会被允许重排这对没有关联的操作。这意味着，给定如下顺序的赋值操作：

```
a = b;
x = y;
```

编译器可能重排为如下顺序：

```
x = y;
a = b;
```

即使编译器没有重排顺序，底层硬件也可能重排，因为有时这样代码执行更快。

然而，`std::atomic`会限制这种重排序，并且这样的限制之一是，在源代码中，对`std::atomic`变量写之前不会有任何操作。这意味对我们的代码

```
auto impatValue = computeImportantValue();
valVailable = true;
```

编译器不仅要保证赋值顺序，还要保证生成的硬件代码不会改变这个顺序。结果就是，将`valAvaliable`声明为`std::atomic`确保了必要的顺序---- 其他线程看到`imptValue`值保证`valVailable`设为true之后。

声明为`volatile`不能保证上述顺序：

```
volatile bool valAvaliable(false);
auto imptValue = computeImportantValue();
valAvailable = true;
```

这份代码编译器可能将赋值顺序对调，也可能在生成机器代码时，其他核心看到`valVailable`更改在`imptValue`之前。

“正常”内存应该有这个特性，在写入值之后，这个值会一直保证直到被覆盖。假设有这样一个正常的int

```
int x;
```

编译器看到下列的操作序列：

```
auto y = x; // read x
y = x; // read x again
```

编译器可通过忽略对y的一次赋值来优化代码，因为初始化和赋值是冗余的。

正常内存还有一个特征，就是如果你写入内存没就不会读，再次吸入，第一次写就可以被忽略，因为肯定会被覆盖。给出下面的代码：

```
x = 10; // write x
x = 20; // write x again
```

编译器可以忽略第一次写入。这意味着如果写在一起：

```
auto y = x;
y = x;
x = 10;
x = 20;
```

编译器生成的代码是这样的：

```
auto y = x;
x = 20;
```

可能你会想睡会写这种重复读写的代码（技术上称为redundant loads 和 dead stores），答案是开发者不会直接写，至少我们不希望开发者这样写。但是在编译器执行了模板实例化，内联和一系列重排序优化之后，结果会出现多余的操作和无效存储，所以编译器需要摆脱这样的情况并不少见。

这种有话讲仅仅在内存表现正常时有效。“特殊”的内存不行。最常见的“特殊”内存是用来memory-mapped I/O的内存。这种内存实际上是与外围设备（比如外部传感器或者显示器，打印机，网络端口）通信，而不是读写（比如RAM）。这种情况下，再次考虑多余的代码：

```
auto y = x; // read x
y = x; // read x again
```

如果x的值是一个温度传感器上报的，第二次对于x的读取就不是多余的，因为温度可能在第一次和第二次读取之间变化。

类似的，写也是一样：

```
x = 10;
x = 20;
```

如果x与无线电发射器的控制端口关联，则代码时控制无线电，10和20意味着不同的指令。优化会更改第一条无线电指令。

`volatile`是告诉编译器我们正在处理"特殊"内存。意味着告诉编译器"不要对这块内存执行任何优化"。所以如果x对应于特殊内存，应该声明为`volatile`：

```
volatile int x;
```

带回我们原始代码：

```
auto y = x;
y = x; // can't be optimized away

x = 10; // can't be optimized away
x = 20;
```

如果x是内存映射（或者已经映射到跨进程共享的内存位置等），这正是我们想要的。

那么，在最后一段代码中，y是什么类型：int还是volatile int?

在处理特殊内存时，必须保留看似多余的读取或者无效存储的事实，顺便说明了为什么`std::atomic`不适合这种场景。`std::atomic`类型允许编译器消除此类冗余操作。代码的编写方式与使用`volatile`的方式完全不同，但是如果我们暂时忽略它，只关注编译器执行的操作，则可以说，

```
std::atomic<int> x;
auto y = x; // conceptually read x (see below)
y = x; // conceptually read x again(see below)

x = 10; // write x
y = 20; // write x again
```

原则上，编译器可能会优化为：

```
auto y = x; // conceptually read x
x = 20; // write x
```

对于特殊内存，显然这是不可接受的。

现在，就当他没有优化了，但是对于x是`std::atomic<int>`类型来说，下面的两条语句都编译不通过。

```
auto y = x; // error
y = x; // error
```

这是因为`std::atomic`类型的拷贝操作时被删除的（参见Item 11）。想象一下如果y使用x来初始化会发生什么。因为x是`std::atomic`类型，y的类型被推导为`std::atomic`（参见Item 2）。我之前说了`std::atomic`最好的特性之一就是所有成员函数都是原子的，但是为了执行从x到y的拷贝初始化是原子的，编译器不得不生成读取x和写入x为原子的代码。硬件通常无法做到这一点，因此`std::atomic`不支持拷贝构造。处于同样的原因，拷贝赋值也被delete了，这也是为什么从x赋值给y也编译失败。（移动操作在`std::atomic`没有显式声明，因此对于Item 17中描述的规则来看，`std::atomic`既不提移动构造器也不提供移动赋值能力）。

可以将x的值传递给y，但是需要使用`std::atomic`的`load`和`store`成员函数。`load`函数原子读取，`store`原子写入。要使用x初始化y，然后将x的值放入y，代码应该这样写：

```
std::atomic<int> y(x.load());
y.store(x.load());
```

这可以编译，但是可以清楚看到不是整条语句原子，而是读取写入分别原子化执行。

给出的代码，编译器可以通过存储x的值到寄存器代替读取两次来“优化”：

```
register = x.load(); // read x into register
std::atomic<int> y(register); // init y with register value
y.store(register); // store register value into y
```

结果如你所见，仅读取x一次，这是对于特殊内存必须避免的优化（这种优化不允许对`volatile`类型值执行）。

事情越辩越明：

- `std::atomic`用在并发程序中
- `volatile`用于特殊内存场景

因为`std::atomic`和`volatile`用于不同的目的，所以可以结合起来使用：

```
volatile std::atomic<int> vai; // operations on vai are atomic and can't be
optimized away
```

这可以用在比如`vai`变量关联了memory-mapped I/O内存并且用于并发程序的场景。

最后一点，一些开发者尤其喜欢使用`std::atomic`的`load`和`store`函数即使不必要时，因为这在代码中显式表明了这个变量不“正常”。强调这一事实并非没有道理。因为访问`std::atomic`确实会更慢一些，我们也看到了`std::atomic`会阻止编译器对代码执行顺序重排。调用`load`和`store`可以帮助识别潜在的可扩展性瓶颈。从正确性的角度来看，没有看到在一个变量上调用`store`来与其他线程进行通信（比如flag表示数据的可用性）可能意味着该变量在声明时没有使用`std::atomic`。这更多是习惯问题，但是，一定要知道`atomic`和`volatile`的巨大不同。

必须记住的事

- `std::atomic`是用在不使用锁，来使变量被多个线程访问。是用来编写并发程序的
- `volatile`是用在特殊内存的场景中，避免被编译器优化内存。