Item16:让const成员函数线程安全

条款16: 让const成员函数线程安全

如果我们在数学领域中工作，我们就会发现用一个类表示多项式是很方便的。在这个类中，使用一个函数来计算多项式的根是很有用的。也就是多项式的值为零的时候。这样的一个函数它不会更改多项式。所以，它自然被声明为const函数。

```
class Polynomial {
public:
    using RootsType =           // 数据结构保存多项式为零的值
      std::vector<double>;      //  (“using” 的信息查看条款9)

    RootsType roots() const;

};
```

计算多项式的根是很复杂的，因此如果不需要的话，我们就不做。如果必须做，我们肯定不会只做一次。所以，如果必须计算它们，就缓存多项式的根，然后实现`roots`来返回缓存的值。下面是最基本的实现:

```
class Polynomial {
public:
    using RootsType = std::vector<double>;

    RootsType roots() const
    {
        if (!rootsAreVaild) {                // 如果缓存不可用
                                             // 计算根
            rootsAreVaild = true;            // 用`rootVals`存储它们
        }

        return rootVals;
    }

private:
    mutable bool rootsAreVaild{ false };    // initializers 的更多信息
    mutable RootsType rootVals{};           // 请查看条款7
};
```

从概念上讲，`roots`并不改变它所操作的多项式对象。但是作为缓存的一部分，它也许会改变`rootVals`和`rootsAreVaild`的值。这就是`mutable`的经典使用样例，这也是为什么它是数据成员声明的一部分。

假设现在有两个线程同时调用`Polynomial`对象的`roots`方法:

```
Polynomial p;


/*------ Thread 1 ------*/      /*-------- Thread 2 --------*/
auto rootsOfp = p.roots();      auto valsGivingZero = p.roots();
```

这些用户代码是非常合理的。`roots`是const 成员函数，那就表示着它是一个读操作。在没有同步的情况下，让多个线程执行读操作是安全的。它最起码应该做到这点。在本例中却没有做到线程安全。因为在`roots`中，这些线程中的一个或两个可能尝试修改成员变量`rootsAreVaild`和`rootVals`。这就意味着在没有同步的情况下，这些代码会有不同的线程读写相同的内存，这就是`data race`的定义。这段代码的行为是未定义的。

问题就是`roots`被声明为const，但不是线程安全的。const声明在c++11和c++98 中都是正确的（检索多项式的根并不会更改多项式的值），因此需要纠正的是线程安全的缺乏。

解决这个问题最普遍简单的方法就是-------使用互斥锁:

```
class Polynomial {
public:
    using RootsType = std::vector<double>;

    RootsType roots() const
    {
        std::lock_guard<std::mutex> g(m);         // lock mutex

        if (!rootsAreVaild) {                     // 如果缓存无效
                                                  // 计算/存储roots
            rootsAreVaild = true;
        }

        return rootsVals;
    }                                             // unlock mutex

private:
    mutable std::mutex m;
    mutable bool rootsAreVaild { false };
    mutable RootsType rootsVals {};
};
```

`std::mutex m`被声明为`mutable`，因为锁定和解锁它的都是non-const函数。在`roots`（const成员函数）中，`m`将被视为const对象。
值得注意的是，因为`std::mutex`是一种`move-only`的类型（一种可以移动但不能复制的类型），所以将`m`添加进多项式中的副作用是使它失去了被复制的能力。不过，它仍然可以移动。

在某些情况下，互斥量是过度的（？）。例如，你所做的只是计算成员函数被调用了多少次。使用`std::atomic`修饰的counter（保证其他线程视这个操作为不可分割的发生，参见item40）。（然而它是否轻量取决于你使用的硬件和标准库中互斥量的实现。）以下是如何使用`std::atomic`来统计调用次数。

```
class Point {                                     // 2D point
public:
    // noexcept的使用参考Item 14
    double distanceFromOrigin() const noexcept
    {
        ++callCount;                              // 原子的递增

        return std::sqrt((x * x) + (y * y));
    }

private:
    mutable std::atomic<unsigned> callCount{ 0 };
    double x, y;
```

```
};
```

与`std::mutex`一样，`std::atomic`是`move-only`类型，所以在`Point`中调用`Count`的意思就是`Point`也是`move-only`的。

因为对`std::atomic`变量的操作通常比互斥量的获取和释放的消耗更小，所以你可能更倾向与依赖`std::atomic`。例如，在一个类中，缓存一个开销昂贵的`int`，你就会尝试使用一对`std::atomic`变量而不是互斥锁。

```
class Widget {
public:

    int magicValue() const
    {
        if (cacheVaild) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;              // 第一步
            cacheVaild = true;                      // 第二步
            return cachedVaild;
        }
    }

private:
    mutable std::atomic<bool> cacheVaild{ false };
    mutable std::atomic<int> cachedValue;
};
```

这是可行的，但有时运行会比它做到更加困难。考虑：

- 一个线程调用`Widget::magicValue`，将`cacheValid`视为`false`，执行这两个昂贵的计算，并将它们的和分配给`cachedValue`。
- 此时，第二个线程调用`Widget::magicValue`，也将`cacheValid`视为`false`，因此执行刚才完成的第一个线程相同的计算。（这里的“第二个线程”实际上可能是其他几个线程。）

这种行为与使用缓存的目的背道而驰。将`cachedValue`和`CacheValid`的顺序交换可以解决这个问题，但结果会更糟：

```
class Widget {
public:

    int magicValue() const
    {
        if (cacheVaild) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cacheVaild = true;                      // 第一步
            return cachedValue = val1 + val2;       // 第二步
        }
    }

}
```

假设`cacheVaild`是false，那么：

- 一个线程调用`Widget::magicValue`，在`cacheVaild`被设置成true时执行到它。
- 在这时，第二个线程调用`Widget::magicValue`随后检查缓存值。看到它是true，就返回`cacheValue`，即使第一个线程还没有给它赋值。因此返回的值是不正确的。

这里有一个坑。对于需要同步的是单个的变量或者内存位置，使用`std::atomic`就足够了。
不过，一旦你需要对两个以上的变量或内存位置作为一个单元来操作的话，就应该使用互斥锁。对于`Widget::magicValue`是这样的。

```
class Widget {
public:

    int magicValue() const
    {
        std::lock_guard<std::mutex> guard(m);   // lock m
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;
            cacheValid = true;
            return cachedValue;
        }
    }                                           // unlock m

private:
    mutable std::mutex m;
    mutable int cachedValue;                    // no longer atomic
    mutable bool cacheValid{ false };           // no longer atomic
};
```

现在，这个条款是基于，多个线程可以同时在一个对象上执行一个const成员函数这个假设的。如果你不是在这种情况下编写一个const成员函数。也就是你可以保证在对象上永远不会有多个线程执行该成员函数。再换句话说，该函数的线程安全是无关紧要的。比如，为单线程使用而设计类的成员函数的线程安全是不重要的。在这种情况下你可以避免，因使用`mutex`和`std::atomics`所消耗的资源，以及包含它们的类只能使用移动语义带来的副作用。然而，这种单线程的场景越来越少见，而且很可能会越来越少。可以肯定的是，const成员函数应支持并发执行，这就是为什么你应该确保const成员函数是线程安全的。

> 应该注意的事情
>
> - 确保const成员函数线程安全，除非你确定它们永远不会在临界区（concurrent context）中使用。
> - `std::atomic`可能比互斥锁提供更好的性能，但是它只适合操作单个变量或内存位置。