链滴

多线程中的单件模式

作者：localvar
原文链接：https://ld246.com/article/1522202803325
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

单件模式可能是所有设计模式中最简单的一个了，但在C++中，尤其是还要支持多线程的话，要想写个正确的实现却并不容易，不信请看：

```
class CSingleton
{
public:
    CSingleton()
    {
        _tprintf( _T("CSingleton::Constructor: Before Sleep\n") );
        Sleep( 1000 ); // 不会改变逻辑, 但增大了问题出现的概率
        _tprintf( _T("CSingleton::Constructor: After Sleep\n") );
    }
    void DoSomeThing()
    {
        _tprintf( _T("CSingleton::DoSomeThing\n") );
    }
    static CSingleton* GetInstance()
    {
        static CSingleton* p = NULL;
        if( p == NULL )
        p = new CSingleton();
        return p;
    }
};

unsigned __stdcall thread( void* )
{
    CSingleton* p = CSingleton::GetInstance();
    p->DoSomeThing();
    return 0;
}

int _tmain( int argc, _TCHAR* argv[] )
{
    for( int i = 0; i < 3; ++i )
    {
        uintptr_t t = _beginthreadex( NULL, 0, thread, NULL, 0, NULL );
        CloseHandle( (HANDLE)t );
    }
    _getch();
    return 0;
}
```

上面的单件实现在单线程中肯定是正确的，不过在多线程中的输出却如下：

```
CSingleton::Constructor: Before Sleep
CSingleton::Constructor: Before Sleep
CSingleton::Constructor: Before Sleep
CSingleton::Constructor: After Sleep
CSingleton::DoSomeThing
CSingleton::Constructor: After Sleep
CSingleton::DoSomeThing
CSingleton::Constructor: After Sleep
CSingleton::DoSomeThing
```

很明显，虽然我们想做个单件，但它却出现了多个实例（或一个实例被初始化了多次）。其原因是我的实现根本没有考虑多线程，那下面的代码把创建实例的部分锁住是不是就行了呢？

```
class CCriSec : CRITICAL_SECTION
{
public:
    CCriSec()
    {
        Sleep( 1000 ); // 增大出问题的概率, 但不改变逻辑
        InitializeCriticalSection( this );
    }

    ~CCriSec() { DeleteCriticalSection( this ); }

    void Enter()
    {
        EnterCriticalSection( this );
    }

    void Leave() { LeaveCriticalSection( this ); }
};

static CSingleton* GetInstance()
{
    static CSingleton* p = NULL;
    static CCriSec lock;
    lock.Enter();
    if( p == NULL )
        p = new CSingleton();
    lock.Leave();
    return p;
}
```

运行一下，不管输出是什么，程序崩溃了。分析一下可以发现，这个例子中的我们确实控制好了对CSingleton实例的初始化，但这种控制却依赖于另一个静态变量（CCriSec的实例）的初始化，而这个新静态变量导致了程序的崩溃，也就是说我们在解决问题的同时引入了新的问题。而且，在这种情况下就算再引入多少个新的临界区也无济于事，因为对最外层的临界区的初始化总会有问题。

上面的例子的问题在于CCriSec是一种复杂的数据类型，所以对它的初始化总要到运行时才能完成，果用整数这样简单的、能在编译期完成初始化的数据类型来做是不是可以呢？

```
static CSingleton* GetInstance()
{
    static CSingleton* p = NULL;
    static volatile long lock = 0;
    if( InterlockedCompareExchange( &lock, 1, 0 ) == 0 )
        p = new CSingleton();
    return p;
}
```

看起来好像没有问题，但运行一下却是下面的输出：

```
CSingleton::Constructor: Before Sleep
CSingleton::DoSomeThing
CSingleton::DoSomeThing
```

```
CSingleton::Constructor: After Sleep
CSingleton::DoSomeThing
```

也就是说DoSomeThing在构造函数返回之前已经被调用了，这显然也是错误的。其原因是我们忽略“对象的创建时需要时间的”，把这个问题也修正一下，就是最终的正确实现了：

```
static CSingleton* GetInstance()
{
    static CSingleton* p = NULL;
    static volatile long lock = 0;
    if( InterlockedCompareExchange( &lock, 1, 0 ) != 0 )
    {
        while( lock != 2 ) // 等待对象创建完成
            Sleep( 0 );
        return p;
    }
    p = new CSingleton();
    lock = 2;
    return p;
}
```

本文采用的单件实现是函数内的静态变量，如果你采用其它方式，也会有类似问题。其实在我看来，件模式是一个看起来简单、做对了很难（上面演示的是多线程中的问题，在具体的实践中还会遇到很其他问题）、同时又没有太多实用价值的东西。

另外，从Windows Vista开始，微软提供了一种多线程下对象初始化的方法，有兴趣的可以中搜一下“NITONCE”，个人认为INITONCE有点完美的过头了，真正好玩又有用的是与它同时出现的“条件变（condition variable）”，后面会写一些与它相关的内容。

PS：从VS2015开始，VC编译器开始保证函数内的静态变量会在使用前完成初始化，所以文中的一些子行为会有所不同。