**[线程本地存储TLS(Thread Local Storage)的原理和实现——分类和原理](http://www.cppblog.com/Tim/archive/2012/07/04/181018.html)**

本文为线程本地存储TLS系列之分类和原理。

一、TLS简述和分类

我们知道在一个进程中，所有线程是共享同一个地址空间的。所以，如果一个变量是全局的或者是静态的，那么所有线程访问的是同一份，如果某一个线程对其进行了修改，也就会影响到其他所有的线程。不过我们可能并不希望这样，所以更多的推荐用基于堆栈的自动变量或函数参数来访问数据，因为基于堆栈的变量总是和特定的线程相联系的。

不过如果某些时候（比如可能是特定设计的dll），我们就是需要依赖全局变量或者静态变量，那有没有办法保证在多线程程序中能访问而不互相影响呢？答案是有的。操作系统帮我们提供了这个功能——TLS线程本地存储。TLS的作用是能将数据和执行的特定的线程联系起来。

实现TLS有两种方法：静态TLS和动态TLS。以下我们将分别说明这两类TLS。

二、静态TLS

1、使用静态TLS

之所以先讲静态TLS，是因为他在代码中使用时非常简单，我们只需写类似如下这一句：

\_\_declspec(thread) DWORD myTLSData=0;

我们就为本程序中的每一个线程创建了一个独立的DWORD数据。

\_\_declspec(thread)的前缀是Microsoft添加给Visual C++编译器的一个修改符。它告诉编译器，对应的变量应该放入可执行文件或DLL文件中它的自己的节中。\_\_declspec(thread)后面的变量必须声明为函数中（或函数外）的一个全局变量或静态变量。不能声明一个类型为\_\_declspec(thread)的局部变量，你想，因为局部变量总是与特定的线程相联系的，如果再加上这个声明是代表什么意思？

2、静态TLS原理

静态TLS的使用是如此简单，那么当我们写了如上代码以后，操作系统和编译器是怎么处理的呢？

首先，在编译器对程序进行编译时，它会将所有声明的TLS变量放入它们自己的节，这个节的名字是.tls。而后链接程序将来自所有对象模块的所有.tls节组合起来，形成结果的可执行文件或DLL文件中的一个大的完整的.tls节。  
然后，为了使含有静态TLS的程序能够运行，操作系统必须参与其操作。当TLS应用程序加载到内存中时，系统要寻找可执行文件中的.tls节，并且动态地分配一个足够大的内存块，以便存放所有的静态TLS变量。应用程序中的代码每次引用其中的一个变量时，就要转换为已分配内存块中包含的一个内存位置。因此，编译器必须生成一些辅助代码来引用该静态TLS变量，这将使你的应用程序变得比较大而且运行的速度比较慢。在x86 CPU上，将为每次引用的静态TLS变量生成3个辅助机器指令。如果在进程中创建了另一个线程，那么系统就要将它捕获并且自动分配另一个内存块，以便存放新线程的静态TLS变量。新线程只拥有对它自己的静态TLS变量的访问权，不能访问属于其他线程的TLS变量。

以上是包含静态TLS变量的可执行文件如何运行的情况。我们再来看看DLL的情况：

a、隐式链接包含静态TLS变量的DLL

如果应用程序使用了静态TLS变量，并且隐式链接包含静态TLS变量的DLL时，当系统加载该应用程序时，它首先要确定应用程序的.tls节的大小，并将这个值与应用程序链接的DLL中的所有.tls节的大小相加。当在你的进程中创建线程时，系统自动分配足够大的内存块来存放所有应用程序声明的和所有隐含链接的DLL包含的TLS变量。

b、显式链接包含静态TLS变量的DLL

考虑一下，当我们的应用程序通过调用LoadLibrary，以便显式链接到包含静态TLS变量的DLL时，会发生什么情况呢？系统必须查看该进程中已经存在的所有线程，并扩大它们的TLS内存块，以便适应新DLL对内存的需求。另外，如果调用FreeLibrary来释放包含静态TLS变量的DLL，那么与进程中的每个线程相关的的TLS内存块又都应该被压缩。  
对于操作系统来说，这样的管理任务太重了。所以，虽然系统允许包含静态TLS变量的库在运行期进行显式加载，但是其包含TLS数据却没有进行相应的初始化。如果试图访问这些数据，就可能导致访问违规！

所以，请记住：如果某个DLL包含静态TLS数据，请不要对这个DLL采用显式链接的方式，否则可能会出错！

三、动态TLS

1、使用动态TLS

动态TLS在程序实现中比静态TLS要稍微麻烦一些，需要通过一组函数来实现：

DWORD TlsAlloc();//返回TLS数组可用位置的索引

BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue); //将调用线程的TLS数组索引dwTlsIndex处设为值lpTlsValue

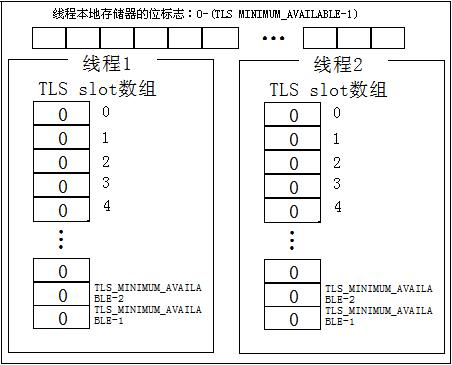
LPVOID TlsGetValue(DWORD dwTlsIndex); //返回调用线程的TLS数组dwTlsIndex索引处的值

BOOL TlsFree(DWORD dwTlsIndex); //释放所有线程的TLS数组位置索引dwTlsIndex，将该位置标记为未使用。

有了以上四个函数，我们可以发现使用动态TLS其实还是很容易很方便的。

2、动态TLS原理

让我们看看windows用来管理TLS的内部数据结构：

  
线程本地存储器的位标志显示了该进程中所有运行的线程正在使用的一组标志。每个标志均可设置为FREE或者INUSE，表示TLS插槽(slot)是否正在使用。Microsoft保证至少TLS\_MINIMUM\_AVAILABLE位标志是可供使用的。另外，TLS\_MINIMUM\_AVAILABLE在WinNT.h中被定义为64。Windows2000将这个标志数组扩展为允许有1000个以上的TLS插槽。

而每一个线程拥有一个自己独立的TLS slot数组，用于存储TLS数据。  
为了使用动态TLS，我们首先调用TlsAlloc()来命令系统对进程的位标志进行扫描，找到一个可用的位置，并返回该索引；如果找不到，就返回TLS\_OUT\_OF\_INDEXES。事实上，除此之外，TlsAlloc函数还会自动清空所有线程的TLS数组的对应索引的值。这避免以前遗留的值可能引起的问题。  
然后，我们就可以调用TlsSetValue函数将对应的索引位保存一个特定的值，可以调用TlsGetValue()来返回该索引位的值。注意，这两个函数并不执行任何测试和错误检查，我们必须要保证索引是通过TlsAlloc正确分配的。  
当所有线程都不需要保留TLS数组某个索引位的时候，应该调用TlsFree。该函数告知系统将进程的位标志数组的index位置为FREE状态。如果运行成功，函数返回TRUE。注意，如果试图释放一个没有分配的索引位，将产生一个错误。  
动态TLS的使用相对静态TLS稍微麻烦一点，但是无论是将其用在可执行文件中还是DLL中，都还是很简单的。而且当用在DLL中时，没有由于DLL链接方式而可能产生的问题，所以，如果要在DLL中用TLS，又不能保证客户始终采用隐式链接方式，那么请采用动态TLS的实现。