[秒杀多线程第二篇 多线程第一次亲密接触 CreateThread与\_beginthreadex本质区别](http://blog.csdn.net/morewindows/article/details/7421759)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-05 09:36 117787人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7421759#comments)(103) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7421759#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[thread](http://www.csdn.net/tag/thread)[null](http://www.csdn.net/tag/null)[structure](http://www.csdn.net/tag/structure)[attributes](http://www.csdn.net/tag/attributes)[security](http://www.csdn.net/tag/security)

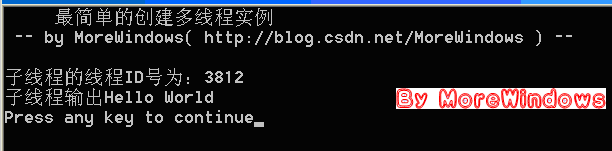
    本文将带领你与多线程作第一次亲密接触，并深入分析CreateThread与\_beginthreadex的本质区别，相信阅读本文后你能轻松的使用多线程并能流畅准确的回答CreateThread与\_beginthreadex到底有什么区别，在实际的编程中到底应该使用CreateThread还是\_beginthreadex？

   使用多线程其实是非常容易的，下面这个程序的主线程会创建了一个子线程并等待其运行完毕，子线程就输出它的线程ID号然后输出一句经典名言——Hello World。整个程序的代码非常简短，只有区区几行。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //最简单的创建多线程实例
2. #include <stdio.h>
3. #include <windows.h>
4. //子线程函数
5. **DWORD** WINAPI ThreadFun(**LPVOID** pM)
6. {
7. printf("子线程的线程ID号为：%d\n子线程输出Hello World\n", GetCurrentThreadId());
8. **return** 0;
9. }
10. //主函数，所谓主函数其实就是主线程执行的函数。
11. **int** main()
12. {
13. printf("     最简单的创建多线程实例\n");
14. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
16. **HANDLE** handle = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);
17. WaitForSingleObject(handle, INFINITE);
18. **return** 0;
19. }

运行结果如下所示：



下面来细讲下代码中的一些函数

第一个 CreateThread

函数功能：创建线程

函数原型：

HANDLEWINAPICreateThread(

  LPSECURITY\_ATTRIBUTESlpThreadAttributes,

  SIZE\_TdwStackSize,

  LPTHREAD\_START\_ROUTINElpStartAddress,

  LPVOIDlpParameter,

  DWORDdwCreationFlags,

  LPDWORDlpThreadId

);

函数说明：

第一个参数表示线程内核对象的安全属性，一般传入NULL表示使用默认设置。

第二个参数表示线程栈空间大小。传入0表示使用默认大小（1MB）。

第三个参数表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。

第四个参数是传给线程函数的参数。

第五个参数指定额外的标志来控制线程的创建，为0表示线程创建之后立即就可以进行调度，如果为CREATE\_SUSPENDED则表示线程创建后暂停运行，这样它就无法调度，直到调用ResumeThread()。

第六个参数将返回线程的ID号，传入NULL表示不需要返回该线程ID号。

函数返回值：

成功返回新线程的句柄，失败返回NULL。

第二个 WaitForSingleObject

函数功能：等待函数 – 使线程进入等待状态，直到指定的内核对象被触发。

函数原形：

DWORDWINAPIWaitForSingleObject(

  HANDLEhHandle,

  DWORDdwMilliseconds

);

函数说明：

第一个参数为要等待的内核对象。

第二个参数为最长等待的时间，以毫秒为单位，如传入5000就表示5秒，传入0就立即返回，传入INFINITE表示无限等待。

因为线程的句柄在线程运行时是未触发的，线程结束运行，句柄处于触发状态。所以可以用WaitForSingleObject()来等待一个线程结束运行。

函数返回值：

在指定的时间内对象被触发，函数返回WAIT\_OBJECT\_0。超过最长等待时间对象仍未被触发返回WAIT\_TIMEOUT。传入参数有错误将返回WAIT\_FAILED

CreateThread()函数是Windows提供的API接口，在C/C++语言另有一个创建线程的函数\_beginthreadex()，在很多书上（包括《Windows核心编程》）提到过尽量使用\_beginthreadex()来代替使用CreateThread()，这是为什么了？下面就来探索与发现它们的区别吧。

       首先要从标准C运行库与多线程的矛盾说起，标准C运行库在1970年被实现了，由于当时没任何一个操作系统提供对多线程的支持。因此编写标准C运行库的程序员根本没考虑多线程程序使用标准C运行库的情况。比如标准C运行库的全局变量errno。很多运行库中的函数在出错时会将错误代号赋值给这个全局变量，这样可以方便调试。但如果有这样的一个代码片段：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. **if** (system("notepad.exe readme.txt") == -1)
2. {
3. **switch**(errno)
4. {
5. ...//错误处理代码
6. }
7. }

假设某个线程A在执行上面的代码，该线程在调用system()之后且尚未调用switch()语句时另外一个线程B启动了，这个线程B也调用了标准C运行库的函数，不幸的是这个函数执行出错了并将错误代号写入全局变量errno中。这样线程A一旦开始执行switch()语句时，它将访问一个被B线程改动了的errno。这种情况必须要加以避免！因为不单单是这一个变量会出问题，其它像strerror()、strtok()、tmpnam()、gmtime()、asctime()等函数也会遇到这种由多个线程访问修改导致的数据覆盖问题。

为了解决这个问题，Windows操作系统提供了这样的一种解决方案——每个线程都将拥有自己专用的一块内存区域来供标准C运行库中所有有需要的函数使用。而且这块内存区域的创建就是由C/C++运行库函数\_beginthreadex()来负责的。下面列出\_beginthreadex()函数的源代码（我在这份代码中增加了一些注释）以便读者更好的理解\_beginthreadex()函数与CreateThread()函数的区别。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //\_beginthreadex源码整理By MoreWindows( http://blog.csdn.net/MoreWindows )
2. \_MCRTIMP **uintptr\_t** \_\_cdecl \_beginthreadex(
3. **void** \*security,  //安全属性
4. unsigned stacksize,  //栈空间大小
5. unsigned (\_\_CLR\_OR\_STD\_CALL \* initialcode) (**void** \*),  //新线程所执行的 线程函数的地址
6. **void** \* argument,  //线程函数的参数
7. unsigned createflag,
8. unsigned \*thrdaddr  //线程的ID号
9. )
10. {
11. \_ptiddata ptd;          //pointer to per-thread data 见注1
12. **uintptr\_t** thdl;         //thread handle 线程句柄
13. unsigned **long** err = 0L; //Return from GetLastError()
14. unsigned dummyid;    //dummy returned thread ID 线程ID号
16. // validation section 检查initialcode是否为NULL
17. \_VALIDATE\_RETURN(initialcode != NULL, EINVAL, 0);
19. //Initialize FlsGetValue function pointer
20. \_\_set\_flsgetvalue();
22. //Allocate and initialize a per-thread data structure for the to-be-created thread.
23. //相当于new一个\_tiddata结构，并赋给\_ptiddata指针。
24. **if** ( (ptd = (\_ptiddata)\_calloc\_crt(1, **sizeof**(**struct** \_tiddata))) == NULL )
25. **goto** error\_return;
27. // Initialize the per-thread data
28. //初始化线程的\_tiddata块即CRT数据区域 见注2
29. \_initptd(ptd, \_getptd()->ptlocinfo);
31. //设置\_tiddata结构中的其它数据，这样这块\_tiddata块就与线程联系在一起了。
32. ptd->\_initaddr = (**void** \*) initialcode; //线程函数地址
33. ptd->\_initarg = argument;              //传入的线程参数
34. ptd->\_thandle = (**uintptr\_t**)(-1);
36. #if defined (\_M\_CEE) || defined (MRTDLL)
37. **if**(!\_getdomain(&(ptd->\_\_initDomain))) //见注3
38. {
39. **goto** error\_return;
40. }
41. #endif  // defined (\_M\_CEE) || defined (MRTDLL)
43. // Make sure non-NULL thrdaddr is passed to CreateThread
44. **if** ( thrdaddr == NULL )//判断是否需要返回线程ID号
45. thrdaddr = &dummyid;
47. // Create the new thread using the parameters supplied by the caller.
48. //\_beginthreadex()最终还是会调用CreateThread()来向系统申请创建线程
49. **if** ( (thdl = (**uintptr\_t**)CreateThread(
50. (LPSECURITY\_ATTRIBUTES)security,
51. stacksize,
52. \_threadstartex,
53. (**LPVOID**)ptd,
54. createflag,
55. (**LPDWORD**)thrdaddr))
56. == (**uintptr\_t**)0 )
57. {
58. err = GetLastError();
59. **goto** error\_return;
60. }
62. //Good return
63. **return**(thdl); //线程创建成功,返回新线程的句柄.
65. //Error return
66. error\_return:
67. //Either ptd is NULL, or it points to the no-longer-necessary block
68. //calloc-ed for the \_tiddata struct which should now be freed up.
69. //回收由\_calloc\_crt()申请的\_tiddata块
70. \_free\_crt(ptd);
71. // Map the error, if necessary.
72. // Note: this routine returns 0 for failure, just like the Win32
73. // API CreateThread, but \_beginthread() returns -1 for failure.
74. //校正错误代号(可以调用GetLastError()得到错误代号)
75. **if** ( err != 0L )
76. \_dosmaperr(err);
77. **return**( (**uintptr\_t**)0 ); //返回值为NULL的效句柄
78. }

讲解下部分代码：

注1．\_ptiddata ptd;中的\_ptiddata是个结构体指针。在mtdll.h文件被定义：

      Typedef struct \_tiddata  \* \_ptiddata

微软对它的注释为Structure for each thread's data。这是一个非常大的结构体，有很多成员。本文由于篇幅所限就不列出来了。

注2．\_initptd(ptd, \_getptd()->ptlocinfo);微软对这一句代码中的getptd()的说明为：

      /\* return address of per-thread CRT data \*/

      \_ptiddata \_\_cdecl\_getptd(void);

对\_initptd()说明如下：

      /\* initialize a per-thread CRT data block \*/

      void\_\_cdecl\_initptd(\_Inout\_ \_ptiddata \_Ptd,\_In\_opt\_ pthreadlocinfo \_Locale);

注释中的CRT （C Runtime Library）即标准C运行库。

注3．if(!\_getdomain(&(ptd->\_\_initDomain)))中的\_getdomain()函数代码可以在thread.c文件中找到，其主要功能是初始化COM环境。

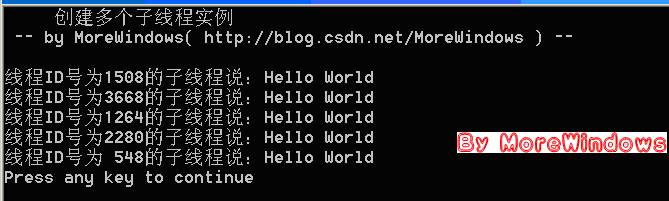
由上面的源代码可知，\_beginthreadex()函数在创建新线程时会分配并初始化一个\_tiddata块。这个\_tiddata块自然是用来存放一些需要线程独享的数据。事实上新线程运行时会首先将\_tiddata块与自己进一步关联起来。然后新线程调用标准C运行库函数如strtok()时就会先取得\_tiddata块的地址再将需要保护的数据存入\_tiddata块中。这样每个线程就只会访问和修改自己的数据而不会去篡改其它线程的数据了。因此，**如果在代码中有使用标准C运行库中的函数时，尽量使用\_beginthreadex()来代替CreateThread()。**相信阅读到这里时，你会对这句简短的话有个非常深刻的印象，如果有面试官问起，你也可以流畅准确的回答了^\_^。

接下来，类似于上面的程序用CreateThread()创建输出“Hello World”的子线程，下面使用\_beginthreadex()来创建多个子线程：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //创建多子个线程实例
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. //子线程函数
6. unsigned **int** \_\_stdcall ThreadFun(**PVOID** pM)
7. {
8. printf("线程ID号为%4d的子线程说：Hello World\n", GetCurrentThreadId());
9. **return** 0;
10. }
11. //主函数，所谓主函数其实就是主线程执行的函数。
12. **int** main()
13. {
14. printf("     创建多个子线程实例 \n");
15. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
17. **const** **int** THREAD\_NUM = 5;
18. **HANDLE** handle[THREAD\_NUM];
19. **for** (**int** i = 0; i < THREAD\_NUM; i++)
20. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
21. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
22. **return** 0;
23. }

运行结果如下：

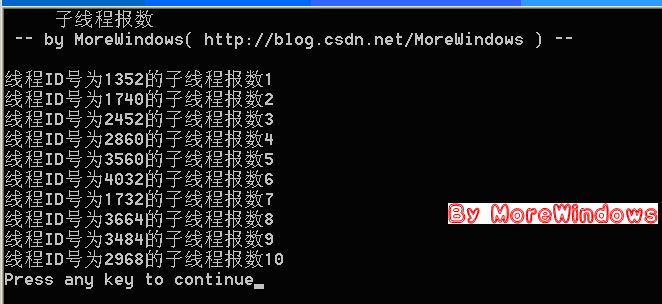


图中每个子线程说的都是同一句话，不太好看。能不能来一个**线程报数**功能，即第一个子线程输出1，第二个子线程输出2，第三个子线程输出3，……。要实现这个功能似乎非常简单——每个子线程对一个全局变量进行递增并输出就可以了。代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //子线程报数
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. **int** g\_nCount;
6. //子线程函数
7. unsigned **int** \_\_stdcall ThreadFun(**PVOID** pM)
8. {
9. g\_nCount++;
10. printf("线程ID号为%4d的子线程报数%d\n", GetCurrentThreadId(), g\_nCount);
11. **return** 0;
12. }
13. //主函数，所谓主函数其实就是主线程执行的函数。
14. **int** main()
15. {
16. printf("     子线程报数 \n");
17. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
19. **const** **int** THREAD\_NUM = 10;
20. **HANDLE** handle[THREAD\_NUM];
22. g\_nCount = 0;
23. **for** (**int** i = 0; i < THREAD\_NUM; i++)
24. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
25. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
26. **return** 0;
27. }

对一次运行结果截图如下：



显示结果从1数到10，看起来好象没有问题。

       答案是不对的，虽然这种做法在逻辑上是正确的，但在多线程环境下这样做是会产生严重的问题，下一篇《[秒杀多线程第三篇 原子操作 Interlocked系列函数](http://blog.csdn.net/morewindows/article/details/7429155)》将为你演示错误的结果（可能非常出人意料）并解释产生这个结果的详细原因。

转载请标明出处，原文地址：<http://blog.csdn.net/morewindows/article/details/7421759>

57楼 [programeppp](http://blog.csdn.net/programeppp) 2015-07-02 11:32发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7421759#reply)

[](http://blog.csdn.net/programeppp)

引用“MoreWindows”的评论：回复ithzhang：本方主要是介绍为什么推荐使用\_beginthread...  
很遗憾，你还是没说清楚.CreateThread发生了什么,只说尽量用\_beginthreadex  
ithzhang 倒是说明白了

楼主，能不能推荐下相关的书籍？多谢了

Re: [Templar101](http://blog.csdn.net/Templar101) 2014-09-02 16:47发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7421759#reply)

[](http://blog.csdn.net/Templar101)

回复domyself：windows核心编程 你值得拥有

你好 请问下 \_beginthreadex() 中 “线程都将拥有自己专用的一块内存区域来供标准C运行库中所有有需要的函数使用”这句话 我不是很懂 能否举个例子了

[](http://blog.csdn.net/yzm365487848)

微软的c库是自己写的么， 比如在strtok()时就会先取得\_tiddata块的地址再将需要保护的数据存入\_tiddata块中， 那么微软的每个c库函数是不是都要加上这些特殊代码啊？ 各操作系统对c标准库的实现是有差异的？？

Re: [MoreWindows](http://blog.csdn.net/MoreWindows) 2012-12-11 15:32发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7421759#reply)

[http://avatar.csdn.net/C/F/7/3_morewindows.jpg](http://blog.csdn.net/MoreWindows)

回复yzm365487848：是的，strtok函数已经作了修改：  
char \* \_\_cdecl strtok (  
char \* string,  
const char \* control  
)  
#endif /\* \_SECURE\_VERSION \*/  
{  
unsigned char \*str;  
const unsigned char \*ctrl = control;  
  
unsigned char map[32];  
int count;  
  
#ifdef \_SECURE\_VERSION  
  
/\* validation section \*/  
\_VALIDATE\_RETURN(context != NULL, EINVAL, NULL);  
\_VALIDATE\_RETURN(string != NULL || \*context != NULL, EINVAL, NULL);  
\_VALIDATE\_RETURN(control != NULL, EINVAL, NULL);  
  
/\* no static storage is needed for the secure version \*/  
  
#else /\* \_SECURE\_VERSION \*/  
  
\_ptiddata ptd = \_getptd();  
。。。。。。  
}

[http://avatar.csdn.net/8/1/1/3_ithzhang.jpg](http://blog.csdn.net/ithzhang)

楼主说尽量使用\_beginthreadex，但是并没有提\_endthreadex。以及使用CreateThread和ExitThread导致的后果。如果使用CreateThread后，线程调用一个需要\_tiddata结构的C/C++运行库函数时会发生下面的情况：首先C/C++运行库尝试取得线程数据块的地址，如果NULL被当做\_tiddata数据块的地址返回，表明主调线程没有与之关联的\_tiddata数据块。此时，C/C++运行库函数会为主调线程分配并初始化一个\_tiddata块，然后将此块与线程关联。而且只要线程还在这个块就一直与线程进行关联。此后的任何C/C++运行库函数也可以使用。也就是说即使你使用CreateThread，C/C++运行库检测到NULL时，也会分配并初始化一个\_tiddata块，但是在调用ExitThread时此内存块不会被释放，导致内存泄露。另一个问题是结构化异常没有就绪，当使用C/C++运行库的signal函数时将会导致进程终止。  
不知楼主如何认为！！鲁莽指出，如有错误请多包涵！！  
http://blog.csdn.net/ithzhang/article/details/8046721

回复ithzhang：本方主要是介绍为什么推荐使用\_beginthreadex来代替CreateThread。不过“\_tiddata数据块如果为NULL，系统会自己为其分配空间的”这个也是正确的。系统肯定要考虑到兼容问题，因此对CreateThread也要采取个些措施，不然旧程序没法运行了。至于 \_endthreadex 如果不显式调用，它会在线程结束时自动调用。请见MSDN的解释。  
You can call \_endthread or \_endthreadex explicitly to terminate a thread; however, \_endthread or \_endthreadex is called automatically when the thread returns from the routine passed as a parameter.   
当然显式调用会使代码更加规范一些。

请教一下：DWORD WINAPI ThreadFun(LPVOID pM)中的参数pM在整个函数定义内没有调用，于是我就将其申明为DWORD WINAPI ThreadFun()，但这样编译通不过，网上查不到相关资料，报错不能将参数 3 从“DWORD (\_\_stdcall \*)(void)”转换为“LPTHREAD\_START\_ROUTINE”

回复beyondliyang：CreateThread()对第三个参数有要求的呀，类型不匹配当然会提示出错，如果你确实不想写LPOVID pM。那你就强制转换下：  
//子线程函数  
DWORD WINAPI ThreadFun()  
{  
子线程代码  
}  
然后创建子线程时强制转换下：  
HANDLE handle = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)ThreadFun, NULL, 0, NULL);  
这样才可以编译通过。

13楼 [packel](http://blog.csdn.net/packel) 2012-04-12 13:14发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7421759#reply)

[http://avatar.csdn.net/8/9/8/3_packel.jpg](http://blog.csdn.net/packel)

\_beginthreadex要用\_endthreadex释放线程

回复packel：根据MSDN上说，你可以显式的调用\_endthread()或者\_endthreadex()。如果没有，线程返回时系统会自动调用。  
在《Windows核心编程》一书中推荐通过线程函数返回来使线程终止运行。

[http://avatar.csdn.net/E/3/2/3_qweewqpkn.jpg](http://blog.csdn.net/qweewqpkn)

我写代码测试过，其实就算你不用\_beginthreadex，而是用CreateThread，每个线程的c变量，或者是对象，如errno对象都是独立的，各个线程相互不关联。如gmtime这些函数返回的缓冲区地址，每个线程都是不同的，如：http://www.cnblogs.com/0x07/archive/2011/12/31/2307808.html，我用都是用cl /MD选项编译的。这让我觉得\_beginthreadex是不是有些多余。

回复qs\_coding：当调用CreateThread()的时候，你使用C运行库，这时操作系统会主动为你分配一个\_tiddate，这时你使用时是没有问题的！但是，这个\_tiddate时得不到释放的，因为CreateThread()调用的不是endthread（）这个函数！ 所以资源会泄露，切记！

回复monk\_CD：看了核心编程，如果在线程中要使用C库，那么\_beginthreadex更安全些。谢谢提醒。

回复qs\_coding：老兄呀。。。你那测试方法明显有误呀。就像我这子线程报数程序现在不也是显示了一个正确的结果了吗？

[](http://blog.csdn.net/xinlongli)

好像最后没有释放句柄资源哦。

Re: [MoreWindows](http://blog.csdn.net/MoreWindows) 2012-04-06 15:13发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7421759#reply)

[http://avatar.csdn.net/C/F/7/3_morewindows.jpg](http://blog.csdn.net/MoreWindows)

回复xinlongli：呵呵，对，你比较细心，程序最后应该要调用CloseHandle()。后面文章中程序会加上这个。