

第6章 线程的基础知识

理解线程是非常关键的，因为每个进程至少需要一个线程。本章将更加详细地介绍线程的知识。尤其是要讲述进程与线程之间存在多大的差别，它们各自具有什么作用。还要介绍系统如何使用线程内核对象来管理线程。与进程内核对象一样，线程内核对象也拥有属性，我们要观察许多用于查询和修改这些属性的函数。此外还要介绍可以在进程中创建和生成更多的线程时所用的函数。

第4章介绍了进程是由两个部分构成的，一个是进程内核对象，另一个是地址空间。同样，线程也是由两个部分组成的：

- 一个是线程的内核对象，操作系统用它来对线程实施管理。内核对象也是系统用来存放线程统计信息的地方。
- 另一个是线程堆栈，它用于维护线程在执行代码时需要的所有函数参数和局部变量（第16章将进一步介绍系统如何管理线程堆栈）。

第4章中讲过，进程是不活泼的。进程从来不执行任何东西，它只是线程的容器。线程总是在某个进程环境中创建的，而且它的整个寿命期都在该进程中。这意味着线程在它的进程地址空间中执行代码，并且在进程的地址空间中对数据进行操作。因此，如果在单进程环境中，你有两个或多个线程正在运行，那么这两个线程将共享单个地址空间。这些线程能够执行相同的代码，对相同的数据进行操作。这些线程还能共享内核对象句柄，因为句柄表依赖于每个进程而不是每个线程存在。

如你所见，进程使用的系统资源比线程多得多，原因是它需要更多的地址空间。为进程创建一个虚拟地址空间需要许多系统资源。系统中要保留大量的记录，这要占用大量的内存。另外，由于.exe和.dll文件要加载到一个地址空间，因此也需要文件资源。而线程使用的系统资源要少得多。实际上，线程只有一个内核对象和一个堆栈，保留的记录很少，因此需要很少的内存。

由于线程需要的开销比进程少，因此始终都应该设法用增加线程来解决编程问题，而要避免创建新的进程。但是，这个建议并不是一成不变的。许多程序设计用多个进程来实现会更好些。应该懂得权衡利弊，经验会指导你的编程实践。

在详细介绍线程之前，首先花一点时间讲一讲如何正确地在应用程序结构中使用线程。

6.1 何时创建线程

线程用于描述进程中的运行路径。每当进程被初始化时，系统就要创建一个主线程。该线程与C/C++运行期库的启动代码一道开始运行，启动代码则调用进入点函数（main、wmain、WinMain或wWinMain），并且继续运行直到进入点函数返回并且C/C++运行期库的启动代码调用ExitProcess为止。对于许多应用程序来说，这个主线程是应用程序需要的唯一线程。不过，进程能够创建更多的线程来帮助执行它们的操作。

每个计算机都拥有一个功能非常强大的资源，即CPU。让CPU闲置起来是绝对没有道理的（如果忽略节省电能问题的话）。为了使CPU处于繁忙状态之中，可以让它执行各种不同的工作。下面是一些例子：

- 可以打开Microsoft Windows 2000配备的内容索引服务程序。它能够创建一个低优先级的线程，以便定期打开你的磁盘驱动器上的文件内容并给内容做索引。若要找到一个文件，可以打开Search Result（搜索结果）窗口（方法是单击Start按钮，从Search菜单中选定For Files Or Folders），再将你的搜索条件输入Containing Text域。这时就可以搜索到索引，相关的文件就会立即显示出来。内容索引服务程序大大改进了性能，因为每次搜索不必打开、扫描和关闭磁盘驱动器上的每个文件。
- 可以使用Windows 2000配备的磁盘碎片整理软件。通常情况下，这种类型的实用程序拥有许多管理选项，一般用户可能不懂，比如该实用程序应该相隔多长时间运行一次，何时运行。使用低优先级线程，可以在后台运行该实用程序，并且在系统空闲时对驱动器进行碎片整理。
- 可以很容易地设想将来版本的编译器，每当暂停键入时，它就可以自动编译你的源代码文件。输出窗口可以向你（几乎）实时显示警告和出错信息。当键入变量和函数名时出现错误时，就能立即发现。在某种程度上讲，Microsoft Visual Studio已经实现了这个功能，使用Workspace的ClassView窗格，就能够看到这些信息。
- 电子表格应用程序能够在后台执行各种计算。
- 字处理程序能够执行重新分页、拼写和语法检查及在后台进行打印。
- 文件可以在后台拷贝到其他介质中。
- Web浏览器在后台与它们的服务器进行通信。因此，在来自当前Web站点的结果输入之前，用户可以缩放浏览器的窗口或者转到另一个Web站点。

这些例子中，有一个重要问题应该注意，那就是多线程能够简化应用程序的用户界面。如果每当停止键入时，编译器建立了你的应用程序，那么就没有必要提供Build菜单选项。文字处理应用程序不需要Check Spelling(拼写检查)和Check Grammar（语法检查）菜单选项。

在Web浏览器的例子中，注意，将不同的线程用于I/O（网络、文件或其他），应用程序的用户界面就能够始终保持工作状态。比如有一个应用程序负责给数据库记录进行排序、打印文档或拷贝文件。如果将独立的线程用于处理这个与I/O相关的任务，用户就可以在进程中继续使用应用程序界面来取消操作。

设计一个拥有多线程的应用程序，就会扩大该应用程序的功能。我们在下一章中可以看到，每个线程被分配了一个CPU。因此，如果你的计算机拥有两个CPU，你的应用程序中有两个线程，那么两个CPU都将处于繁忙状态。实际上，你是让两个任务在执行一个任务的时间内完成操作。

每个进程至少拥有一个线程。因此，如果你在应用程序中不执行任何特殊的操作，在多进程操作系统上运行，就能够得到许多好处。例如，可以建立一个应用程序，并同时使用文字处理程序（我常常这样做）。如果计算机拥有两个CPU，那么该应用程序就可以在一个处理器上执行，而另一个处理器则负责处理文档。另外，如果编译器出现一个错误，导致它的线程进入一个无限循环，仍然可以使用其他的进程（16位Windows和MS-DOS应用程序则不行）。

6.2 何时不能创建线程

至今为止，一直在讨论多线程应用程序的优点。虽然多线程应用程序的优点很多，但是它也存在某些不足之处。有些开发人员认为，解决问题的方法是将它分割成多个线程。这种想法是完全错误的。

线程确实是非常有用的，但是，当使用线程时，在解决原有的问题时可能产生新的问题。

例如，你开发了一个文字处理应用程序，并且想要让打印函数作为它自己的线程来运行。这听起来是个很好的主意，因为用户可以在打印文档时立即回头着手编辑文档。但是，这意味着文档中的数据可能在文档打印时变更。也许最好是不要让打印操作在它自己的线程中发生，不过这种“方案”看起来有点极端。如果你让用户编辑另一个文档，但是锁定正在打印的文档，使得打印结束前该文档不能修改，那将会怎样呢？这里还有第三种思路，将文档拷贝到一个临时文件，然后打印该临时文件的内容，并让用户修改原始文档。当包含该文档的临时文件结束打印时，删除临时文件。

如你所见，线程能够解决某些问题，但是却又会产生新的问题。在开发应用程序的用户界面时，很可能出现对线程的另一种误用。几乎在所有的应用程序中，所有用户界面的组件（窗口）应该共享同一个线程。单个线程应该创建窗口的所有子窗口。有时在不同的线程上创建不同的窗口是有用的，不过这种情况确实非常少见。

通常情况下，一个应用程序拥有一个用户界面线程，用于创建所有窗口，并且有一个 GetMessage 循环。进程中的所有其他线程都是工作线程，它们与计算机或 I/O 相关联，但是这些线程从不创建窗口。另外，一个用户界面线程通常拥有比工作线程更高的优先级，因此用户界面负责向用户作出响应。

虽然单个进程拥有多个用户界面线程的情况并不多见，但是这种情况有着某种有效的用途。Windows Explorer 为每个文件夹窗口创建了一个独立的线程。它使你能够将文件从一个文件夹拷贝到另一个文件夹，并且仍然可以查看你的系统上的其他文件夹。另外，如果 Explorer 中存在一个错误，那么负责处理文件夹的线程可能崩溃，但是仍然能够对其他文件夹进行操作，至少在执行的操作导致其他文件夹也崩溃之前，仍然可以对它们进行操作（关于线程和用户界面的详细说明，参见第26和27章）。

上述内容的实质是应该慎重地使用多线程。不要想用就用。仅仅使用赋予进程的主线程，就能够编写出许多非常有用的和功能强大的应用程序。

6.3 编写第一个线程函数

每个线程必须拥有一个进入点函数，线程从这个进入点开始运行。前面已经介绍了主线程的进入点函数：即 main、wmain、WinMain 或 wWinMain。如果要在你的进程中创建一个辅助线程，它必定也是个进入点函数，类似下面的样子：

```
DWORD WINAPI ThreadFunc(PVOID pvParam){
    DWORD dwResult = 0;.
    :
    return(dwResult);
}
```

你的线程函数可以执行你想要它做的任何任务。最终，线程函数到达它的结尾处并且返回。这时，线程终止运行，该堆栈的内存被释放，同时，线程的内核对象的使用计数被递减。如果使用计数降为0，线程的内核对象就被撤消。与进程内核对象的情况相同，线程内核对象的寿命至少可以达到它们相关联的线程那样长，不过，该对象的寿命可以远远超过线程本身的寿命。

下面对线程函数的几个问题作一说明：

- 主线程的进入点函数的名字必须是 main、wmain、WinMain 或 wWinMain，与这些函数不同的是，线程函数可以使用任何名字。实际上，如果在应用程序中拥有多个线程函数，必须为它们赋予不同的名字，否则编译器/链接程序会认为你为单个函数创建了多个实现函数。

- 由于给你的主线程的进入点函数传递了字符串参数，因此可以使用 ANSI/Unicode版本的进入点函数：main/wmain和WinMain/wWinMain。可以给线程函数传递单个参数，参数的含义由你而不是由操作系统来定义。因此，不必担心 ANSI/Unicode问题。
- 线程函数必须返回一个值，它将成为该线程的退出代码。这与 C/C++运行期库关于让主线程的退出代码作为进程的退出代码的原则是相似的。
- 线程函数（实际上是你的所有函数）应该尽可能使用函数参数和局部变量。当使用静态变量和全局变量时，多个线程可以同时访问这些变量，这可能破坏变量的内容。然而，参数和局部变量是在线程堆栈中创建的，因此它们不太可能被另一个线程破坏。

既然懂得了实现线程函数的方法，下面讲述如何让操作系统来创建能够执行线程函数的线程。

6.4 CreateThread函数

前面已经讲述了调用 CreateProcess函数时如何创建进程的主线程。如果想要创建一个或多个辅助函数，只需要让一个已经在运行的线程来调用 CreateThread：

```
HANDLE CreateThread(  
    PSECURITY_ATTRIBUTES psa,  
    DWORD cbStack,  
    PTHREAD_START_ROUTINE pfnStartAddr,  
    PVOID pvParam,  
    DWORD fdwCreate,  
    PDWORD pdwThreadId);
```

当CreateThread被调用时，系统创建一个线程内核对象。该线程内核对象不是线程本身，而是操作系统用来管理线程的较小的数据结构。可以将线程内核对象视为由关于线程的统计信息组成的一个小型数据结构。这与进程和进程内核对象之间的关系是相同的。

系统从进程的地址空间中分配内存，供线程的堆栈使用。新线程运行的进程环境与创建线程的环境相同。因此，新线程可以访问进程的内核对象的所有句柄、进程中的所有内存和在这个相同的进程中的所有其他线程的堆栈。这使得单个进程中的多个线程确实能够非常容易地互相通信。

注意 CreateThread函数是用来创建线程的 Windows函数。不过，如果你正在编写 C/C++代码，决不应该调用 CreateThread。相反，应该使用 Visual C++运行期库函数 _beginthreadex。如果不使用 Microsoft的 Visual C++编译器，你的编译器供应商有它自己的 CreateThred替代函数。不管这个替代函数是什么，你都必须使用。本章后面将要介绍 _beginthreadex能够做什么，它的重要性何在。

这就是 Create Thread函数的概述，下面各节将要具体介绍 CreateThread的每个参数。

6.4.1 psa

psa参数是指向 SECURITY_ATTRIBUTES结构的指针。如果想要该线程内核对象的默认安全属性，可以（并且通常能够）传递 NULL。如果希望所有的子进程能够继承该线程对象的句柄，必须设定一个 SECURITY_ATTRIBUTES结构，它的 bInheritHandle成员被初始化为 TRUE。详细信息参见第3章。

6.4.2 cbStack

cbStack参数用于设定线程可以将多少地址空间用于它自己的堆栈。每个线程拥有它自己的堆栈。当 CreateProcess启动一个进程时，它就在内部调用 CreateThread来对进程的主线程进

行初始化。对于cbStack参数来说，CreateProcess使用存放在可执行文件中的一个值。可以使用链接程序的/STACK开关来控制这个值：

```
/STACK:[reserve] [ ,commit]
```

reserve参数用于设定系统应该为线程堆栈保留的地址空间量。默认值是 1 MB。Commit参数用于设定开始时应该承诺用于堆栈保留区的物理存储器的容量。默认值是 1 页。当线程中的代码执行时，可能需要多个页面的存储器。当线程溢出它的堆栈时，就生成一个异常条件（关于线程堆栈和堆栈溢出的异常条件的详细说明，参见第 16 章，关于一般异常条件的处理的详细说明，参见第 23 章）。系统抓取该异常条件，并且将另一页（或者你为 commit 参数设定的任何值）用于保留空间，这使得线程的堆栈能够根据需要动态地扩大。

当调用CreateThread时，如果传递的值不是 0，就能使该函数将所有的存储器保留并分配给线程的堆栈。由于所有的存储器预先作了分配，因此可以确保线程拥有指定容量的可用堆栈存储器。保留空间的容量既可以是/STACK链接程序设定的容量，也可以是CbStack的值，谁大就用谁。分配的存储器容量应该与传递的 cbStack 值相一致。如果将 0 传递给 CbStack 参数，CreateThread就保留一个区域，并且将链接程序嵌入 .exe 文件的/STACK链接程序开关信息指明的存储器容量分配给线程堆栈。

保留空间的容量用于为堆栈设置一个上限，这样就可以抓住代码中的循环递归错误。例如，你编写一个递归自调用函数，该函数也包含导致循环递归的一个错误。每次函数调用自己的时候，堆栈上就创建一个新的堆栈框。如果系统不设定堆栈的最大值，该递归函数就永远不会停止对自己的调用。进程的所有地址空间将被分配，大量的物理存储器将被分配给该堆栈。通过设置一个堆栈限制值，就可以防止应用程序用完大量的物理存储器，同时，也可以更快地知道何时程序中出现了错误（第16章中的Summation示例应用程序显示了如何跟踪和处理应用程序中的堆栈溢出）。

6.4.3 pfnStartAddr和pvParam

pfnStartAddr参数用于指明想要新线程执行的线程函数的地址。线程函数的 pvParam 参数与原先传递给 CreateThread 的 pvParam 参数是相同的。CreateThread 使用该参数不做别的事情，只是在线程启动执行时将该参数传递给线程函数。该参数提供了一个将初始化值传递给线程函数的手段。该初始化数据既可以是数字值，也可以是指向包含其他信息的一个数据结构的指针。

创建多个线程，使这些线程拥有与起始点相同的函数地址，这是完全合乎逻辑的并且是非常有用的。例如，可以实现一个Web服务器，以便创建一个新线程来处理每个客户机的请求。每个线程都知道它正在处理哪个客户机的请求，因为当创建线程时，你传递了一个不同的pvParam值。

记住，Windows是个抢占式多线程系统，这意味着新线程和调用 CreateThread 的线程可以同时执行。由于线程可以同时运行，就会出现一些问题。请看下面的代码：

```
DWORD WINAPI FirstThread(PVOID pvParam) {
    // Initialize a stack-based variable
    int x = 0;
    DWORD dwThreadId;

    // Create a new thread.
    HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID) &x,
        0, &dwThreadId);

    // We don't reference the new thread anymore,
    // so close our handle to it.
    CloseHandle(hThread);
}
```

```
// Our thread is done.
// BUG: our stack will be destroyed, but
//      SecondThread might try to access it.
return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    // Do some lengthy processing here.

    :

    // Attempt to access the variable on FirstThread's stack.
    // NOTE: This may cause an access violation - it depends on timing!
    * ((int *) pvParam) = 5;

    :

    return(0);
}
```

在上面这个代码中，FirstThread可以在SecondThread将5分配给FirstThread的x之前结束它的操作。如果出现这种情况，SecondThread将不知道FirstThread已经不再存在，并且仍然试图修改现在已经是无效地址的内容。这会导致SecondThread产生一次访问违规，因为FirstThread的堆栈已经在FirstThread终止运行时被撤消。解决这个问题方法之一是将x声明为一个静态变量，这样，编译器就为应用程序的数据部分中的x创建一个存储区，而不是在堆栈上创建存储区。

但是这使得函数成为不可重新进入的函数。换句话说，无法创建两个执行相同函数的线程，因为两个线程将共享该静态变量。解决这个问题（和它的更复杂的变形）的另一种方法是使用正确的线程同步技术（第8、9章和10章介绍）。

6.4.4 fdwCreate

fdwCreate参数可以设定用于控制创建线程的其他标志。它可以是两个值中的一个。如果该值是0，那么线程创建后可以立即进行调度。如果该值是CREATE_SUSPENDED，系统可以完整地创建线程并对它进行初始化，但是要暂停该线程的运行，这样它就无法进行调度。

CREATE_SUSPENDED标志使得应用程序能够在它有机会执行任何代码之前修改线程的某些属性。由于这种必要性很少，因此该标志并不常用。第5章介绍的JobLab应用程序说明了该标志的正确方法。

6.4.5 pdwThreadId

CreateThread的最后一个参数是pdwThreadId，它必须是DWORD的一个有效地址，CreateThread使用这个地址来存放系统分配给新线程的ID（进程和线程的ID已经在第4章中作了介绍）。

注意 在Windows 2000（和Windows NT 4）下，可以（并且通常是这样做的）为该参数传递NULL。它告诉函数，你对线程的ID不感兴趣，但是线程已经创建了。在Windows 95和Windows 98下，为该参数传递NULL会导致函数运行失败，因为函数试图将ID写入地址NULL（这是不合法的）。因此线程不能创建。

当然，操作系统之间的不一致现象会给编程人员带来一些问题。例如，在Windows 2000下（即使为pdwThreadId参数传递了NULL，它也创建了该线程）编写和测试了一

个应用程序，当后来在Windows 98上运行该应用程序时，CreateThread将不创建新的线程。必须始终在你声称支持的所有操作系统（和所有版本）上充分测试应用程序。

6.5 终止线程的运行

若要终止线程的运行，可以使用下面的方法：

- 线程函数返回（最好使用这种方法）。
- 通过调用ExitThread函数，线程将自行撤消（最好不要使用这种方法）。
- 同一个进程或另一个进程中的线程调用 TerminateThread函数（应该避免使用这种方法）。
- 包含线程的进程终止运行（应该避免使用这种方法）。

下面将介绍终止线程运行的方法，并且说明线程终止运行时会出现什么情况。

6.5.1 线程函数返回

始终都应该将线程设计成这样的形式，即当想要线程终止运行时，它们就能够返回。这是确保所有线程资源被正确地清除的唯一办法。

如果线程能够返回，就可以确保下列事项的实现：

- 在线程函数中创建的所有C++对象均将通过它们的撤消函数正确地撤消。
- 操作系统将正确地释放线程堆栈使用的内存。
- 系统将线程的退出代码（在线程的内核对象中维护）设置为线程函数的返回值。
- 系统将递减线程内核对象的使用计数。

6.5.2 ExitThread函数

可以让线程调用ExitThread函数，以便强制线程终止运行：

```
VOID ExitThread(DWORD dwExitCode);
```

该函数将终止线程的运行，并导致操作系统清除该线程使用的所有操作系统资源。但是，C++资源（如C++类对象）将不被撤消。由于这个原因，最好从线程函数返回，而不是通过调用ExitThread来返回（详细说明参见第4章）。

当然，可以使用ExitThread的dwExitThread参数告诉系统将线程的退出代码设置为什么。ExitThread函数并不返回任何值，因为线程已经终止运行，不能执行更多的代码。

注意 终止线程运行的最佳方法是让它的线程函数返回。但是，如果使用本节介绍的方法，应该知道ExitThread函数是Windows用来撤消线程的函数。如果编写C/C++代码，那么决不应该调用ExitThread。应该使用Visual C++运行库函数_endthreadex。如果不使用Microsoft的Visual C++编译器，你的编译器供应商有它自己的ExitThread的替代函数。不管这个替代函数是什么，都必须使用。本章后面将说明_endthreadex的作用和它的重要性。

6.5.3 TerminateThread函数

调用TerminateThread函数也能够终止线程的运行：

```
BOOL TerminateThread(  
    HANDLE hThread,  
    DWORD dwExitCode);
```

与ExitThread不同,ExitThread总是撤消调用的线程,而TerminateThread能够撤消任何线程。hThread参数用于标识被终止运行的线程的句柄。当线程终止运行时,它的退出代码成为你作为dwExitCode参数传递的值。同时,线程的内核对象的使用计数也被递减。

注意 TerminateThread函数是异步运行的函数,也就是说,它告诉系统你想要线程终止运行,但是,当函数返回时,不能保证线程被撤消。如果需要确切地知道该线程已经终止运行,必须调用 WaitForSingleObject(第9章介绍)或者类似的函数,传递线程的句柄。

设计良好的应用程序从来不使用这个函数,因为被终止运行的线程收不到它被撤消的通知。线程不能正确地清除,并且不能防止自己被撤消。

注意 当使用返回或调用 ExitThread的方法撤消线程时,该线程的内存堆栈也被撤消。但是,如果使用 TerminateThread,那么在拥有线程的进程终止运行之前,系统不撤消该线程的堆栈。Microsoft故意用这种方法来实现 TerminateThread。如果其他仍然正在执行的线程要引用强制撤消的线程堆栈上的值,那么其他的线程就会出现访问违规的问题。如果将已经撤消的线程的堆栈留在内存中,那么其他线程就可以继续很好地运行。

此外,当线程终止运行时,DLL通常接收通知。如果使用 Terminate Thread 强迫线程终止,DLL就不接收通知,这能阻止适当的清除(详细信息参见第20章)。

6.5.4 在进程终止运行时撤消线程

第4章介绍的ExitProcess和TerminateProcess函数也可以用来终止线程的运行。差别在于这些线程将会使终止运行的进程中的所有线程全部终止运行。另外,由于整个进程已经被关闭,进程使用的所有资源肯定已被清除。这当然包括所有线程的堆栈。这两个函数会导致进程中的剩余线程被强制撤消,就像从每个剩余的线程调用 TerminateThread一样。显然,这意味着正确的应用程序清除没有发生,即C++对象撤消函数没有被调用,数据没有转至磁盘等等。

6.5.5 线程终止运行时发生的操作

当线程终止运行时,会发生下列操作:

- 线程拥有的所有用户对象均被释放。在 Windows中,大多数对象是由包含创建这些对象的线程的进程拥有的。但是一个线程拥有两个用户对象,即窗口和挂钩。当线程终止运行时,系统会自动撤消任何窗口,并且卸载线程创建的或安装的任何挂钩。其他对象只有在拥有线程的进程终止运行时才被撤消。
- 线程的退出代码从 STILL_ACTIVE改为传递给ExitThread或TerminateThread的代码。
- 线程内核对象的状态变为已通知。
- 如果线程是进程中最后一个活动线程,系统也将进程视为已经终止运行。
- 线程内核对象的使用计数递减1。

当一个线程终止运行时,在与它相关联的线程内核对象的所有未结束的引用关闭之前,该内核对象不会自动被释放。

一旦线程不再运行,系统中就没有别的线程能够处理该线程的句柄。然而别的线程可以调用GetExitcodeThread来检查由hThread标识的线程是否已经终止运行。如果它已经终止运行,则确定它的退出代码:


```
BOOL GetExitCodeThread(  
    HANDLE hThread,  
    PDWORD pdwExitCode);
```

退出代码的值在pdwExitCode指向的DWORD中返回。如果调用GetExitCodeThread时线程尚未终止运行，该函数就用 STILL_ACTIVE 标识符（定义为 0x103）填入DWORD。如果该函数运行成功，便返回 TRUE（第9章将详细地介绍如何使用线程的句柄来确定何时线程终止运行）。

6.6 线程的一些性质

到现在为止，讲述了如何实现线程函数和如何让系统创建线程以便执行该函数。本节将要介绍系统如何使这些操作获得成功。

图6-1显示了系统在创建线程和对线程进行初始化时必须做些什么工作。让我们仔细看一看这个图，以便确切地了解发生的具体情况。调用 CreateThread可使系统创建一个线程内核对象。该对象的初始使用计数是2（在线程停止运行和从CreateThread返回的句柄关闭之前，线程内核对象不会被撤消）。线程的内核对象的其他属性也被初始化，暂停计数被设置为1，退出代码始终为STILL_ACTIVE（0x103），该对象设置为未通知状态。

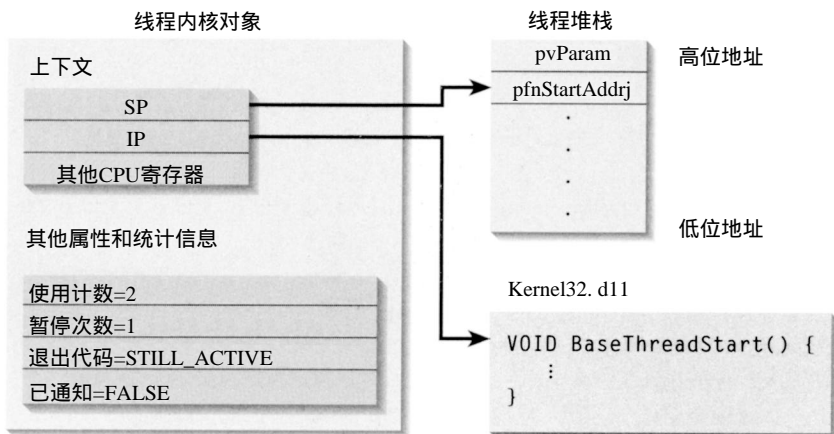


图6-1 线程的创建和初始化的示意图

一旦内核对象创建完成，系统就分配用于线程的堆栈的内存。该内存是从进程的地址空间分配而来的，因为线程并不拥有它自己的地址空间。然后系统将两个值写入新线程的堆栈的上端（线程堆栈总是从内存的高地址向低地址建立）。写入堆栈的第一个值是传递给CreateThread的pvParam参数的值。紧靠它的下面是传递给CreateThread的pfnStartAddr参数的值。

每个线程都有它自己的一组CPU寄存器，称为线程的上下文。该上下文反映了线程上次运行时该线程的CPU寄存器的状态。线程的这组CPU寄存器保存在一个CONTEXT结构（在WinNT.h头文件中作了定义）中。CONTEXT结构本身则包含在线程的内核对象中。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器。记住，线程总是在进程的上下文中运行的。因此，这些地址都用于标识拥有线程的进程地址空间中的内存。当线程的内核对象被初始化时，CONTEXT结构的堆栈指针寄存器被设置为线程堆栈上用来放置pfnStartAddr的地址。指令指针寄存器置为称为BaseThreadStart的未文档化（和未输出）的函数的地址中。该函数包含在Kernel32.dll模块中（这也是实现CreateThread函数的地方）。图6-1显示了它

的全部情况。

下面是BaseThreadStart函数执行的基本操作：

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // NOTE: We never get here.
}
```

当线程完全初始化后，系统就要查看 CREATE_SUSPENDED标志是否已经传递给CreateThread。如果该标志没有传递，系统便将线程的暂停计数递减为0，该线程可以调度到一个进程中。然后系统用上次保存在线程上下文中的值加载到实际的CPU寄存器中。这时线程就可以执行代码，并对它的进程的地址空间中的数据进行操作。

由于新线程的指令指针被置为BaseThreadStart，因此该函数实际上是线程开始执行的地方。BaseThreadStart的原型会使你认为该函数接收了两个参数，但是这表示该函数是由另一个函数来调用的，而实际情况并非如此。新线程只是在此处产生并且开始执行。BaseThreadStart认为它是由另一个函数调用的，因为它可以访问两个函数。但是，之所以可以访问这些参数，是因为操作系统将值显式写入了线程的堆栈（这就是参数通常传递给函数的方法）。注意，有些CPU结构使用CPU寄存器而不是堆栈来传递参数。对于这些结构来说，系统将在允许线程执行BaseThreadStart函数之前对相应的寄存器正确地进行初始化。

当新线程执行BaseThreadStart函数时，将会出现下列情况：

- 在线程函数中建立一个结构化异常处理（SEH）帧，这样，在线程执行时产生的任何异常情况都会得到系统的某种默认处理（关于结构化异常处理的详细说明参见第3、24和25章）。
- 系统调用线程函数，并将你传递给CreateThread函数的pvParam参数传递给它。
- 当线程函数返回时，BaseThreadStart调用ExitThread，并将线程函数的返回值传递给它。该线程内核对象的使用计数被递减，线程停止执行。
- 如果线程产生一个没有处理的异常条件，由BaseThreadStart函数建立的SEH帧将负责处理该异常条件。通常情况下，这意味着向用户显示一个消息框，并且在用户撤消该消息框时，BaseThreadStart调用ExitThread，以终止整个进程的运行，而不只是终止线程的运行。

注意，在BaseThreadStart函数中，线程要么调用ExitThread，要么调用ExitProcess。这意味着线程不能退出该函数，它总是在函数中被撤消。这就是BaseThreadStart的原型规定返回VOID，而它从来不返回的原因。

另外，由于使用BaseThreadStart，线程函数可以在它完成处理后返回。当BaseThreadStart调用线程函数时，它会把返回地址推进堆栈，这样，线程函数就能知道在何处返回。但是，BaseThreadStart不允许返回。如果它不强制撤消线程，而只是试图返回，那么几乎可以肯定会引发访问违规，因为线程堆栈上不存在返回地址，并且BaseThreadStart将试图返回到某个随机内存位置。

当进程的主线程被初始化时，它的指令指针被设置为另一个未文档化的函数，称为BaseProcessStart。该函数几乎与BaseThreadStart相同，形式类似下面的样子：

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
    __try {
```

```
ExitThread((pfnStartAddr)());  
}  
__except(UnhandledExceptionFilter(GetExceptionInformation())) {  
    ExitProcess(GetExceptionCode());  
}  
// NOTE: We never get here.  
}
```

这两个函数之间的唯一差别是，BaseProcessStart没有引用pvParam参数。当BaseProcessStart开始执行时，它调用C/C++运行期库的启动代码，该启动代码先要初始化main、wmain、WinMain或wWinMain函数，然后调用这些函数。当进入点函数返回时，C/C++运行期库的启动代码就调用ExitProcess。因此，对于C/C++应用程序来说，主线程从不返回BaseProcessStart函数。

6.7 C/C++运行期库的考虑

Visual C++ 配有6个C/C++运行期库。表6-1对它们进行了描述。

表6-1 C/C++运行期库

库 名	描 述
LibC.lib	用于单线程应用程序的静态链接库（当创建新应用程序时，它是默认库）
LibCD.lib	用于单线程应用程序的静态链接库的调试版
LibCMT.lib	用于多线程应用程序的静态链接库的发行版
LibCMTD.lib	用于多线程应用程序的静态链接库的调试版
MSVCRt.lib	用于动态链接MSVCRt.dll库的发行版的输入库
MSVCRtD.lib	用于动态链接MSVCRtD.dll的调试版的输入库。该库同时支持单线程应用程序和多线程应用程序

当实现任何类型的编程项目时，必须知道将哪个库与你的项目相链接。可以使用图6-2所示的Project Settings对话框来选定一个库。在C/C++选项卡上，在Code Generation（生成的代码）类别中，从Use run-time library（使用运行期库）组合框中选定6个选项中的一个。

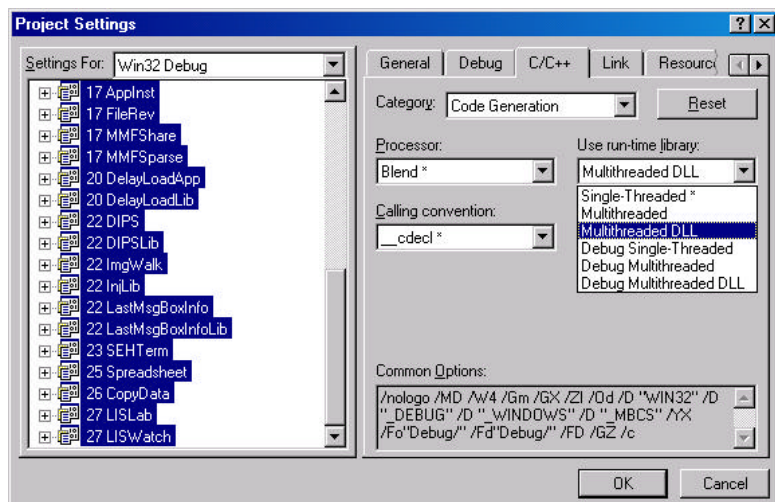


图6-2 Project Settings 对话框

应该考虑的第一件事情是，“为什么必须将一个库用于单线程应用程序，而将另一个库用于多线程应用程序？”，原因是，标准C运行期库是1970年问世的，它远远早于线程在任何应用程序上的应用。运行期库的发明者没有考虑到将C运行期库用于多线程应用程序的问题。

考虑一下标准C运行期的全局变量errno。有些函数在发生错误时设置该变量。假设拥有下面这个代码段：

```
BOOL fFailure = (system("NOTEPAD.EXE README.TXT") == -1);

if (fFailure) {
    switch (errno) {
        case E2BIG:    // Argument list or environment too big
            break;
        case ENOENT:   // Command interpreter cannot be found
            break;
        case ENOEXEC:  // Command interpreter has bad format
            break;
        case ENOMEM:   // Insufficient memory to run command
            break;
    }
}
```

现在，假设在调用system函数之后和调用if语句之前，执行上面代码的线程中断运行，同时假设，该线程中断运行是为了让同一进程中的第二个线程开始执行，而这个新线程将执行另一个负责设置全局变量errno的C运行期函数。当CPU在晚些时候重新分配给第一个线程时，errno的值将不再能够反映调用上面代码中的system函数时的错误代码。为了解决这个问题，每个线程都需要它自己的errno变量。此外，必须有一种机制，使得线程能够引用它自己的errno变量，但是又不触及另一个线程的errno变量。

这是标准C/C++运行期库原先并不是设计用于多线程应用程序的唯一一个例子。在多线程环境中存在问题的C/C++运行期库变量和函数包括errno、_doserrno、strtok、_wcstok、strerror、_strerror、tmpnam、tmpfile、asctime、_wasctime、gmtime、_ecvt和_fcvt等。

若要使多线程C和C++程序能够正确地运行，必须创建一个数据结构，并将它与使用C/C++运行期库函数的每个线程关联起来。当你调用C/C++运行期库时，这些函数必须知道查看调用线程的数据块，这样就不会对别的线程产生不良影响。

那么系统是否知道在创建新线程时分配该数据块呢？回答是它不知道。系统根本不知道你得到的应用程序是用C/C++编写的，也不知道你调用函数的线程本身是不安全的。问题在于你必须正确地进行所有的操作。若要创建一个新线程，绝对不要调用操作系统的CreateThread函数，必须调用C/C++运行期库函数_beginthreadex：

```
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned (*start_address)(void *),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr);
```

_beginthreadex函数的参数列表与CreateThread函数的参数列表是相同的，但是参数名和类型并不完全相同。这是因为Microsoft的C/C++运行期库的开发小组认为，C/C++运行期函数不应该对Windows数据类型有任何依赖。_beginthreadex函数也像CreateThread那样，返回新创建

的线程的句柄。因此，如果调用源代码中的 `CreateThread`，就很容易用对 `_beginthreadex` 的调用全局取代所有这些调用。不过，由于数据类型并不完全相同，所以必须进行某种转换，使编译器运行得顺利些。为了使操作更加容易，我在源代码中创建了一个宏 `chBEGINTHREADEX`：

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
    pvParam, fdwCreate, pdwThreadID) \
    ((HANDLE) _beginthreadex( \
        (void *) (psa), \
        (unsigned) (cbStack), \
        (PTHREAD_START) (pfnStartAddr), \
        (void *) (pvParam), \
        (unsigned) (fdwCreate), \
        (unsigned *) (pdwThreadID)))
```

注意，`_beginthreadex` 函数只存在于 C/C++ 运行期库的多线程版本中。如果链接到单线程运行期库，就会得到一个链接程序报告的“未转换的外部符号”错误消息。当然，从设计上讲，这个错误的原因是单线程库在多线程应用程序中不能正确地运行。另外需要注意，当创建一个新项目时，Visual Studio 默认选定单线程库。这并不是最安全的默认设置，对于多线程应用程序来说，必须显式转换到多线程的 C/C++ 运行期库。

由于 Microsoft 为 C/C++ 运行期库提供了源代码，因此很容易准确地确定 `CreateThread` 究竟无法执行哪些 `_beginthreadex` 能执行的操作。实际上，我搜索了 Visual Studio 的光盘，发现 `_beginthreadex` 的源代码在 `Threadex.c` 中。代换重新打印它的源代码，这里提供了它的伪代码版本，并且列出它的一些令人感兴趣的要点：

```
unsigned long __cdecl _beginthreadex (
    void *psa,
    unsigned cbStack,
    unsigned (__stdcall * pfnStartAddr) (void *),
    void * pvParam,
    unsigned fdwCreate,
    unsigned *pdwThreadID) {

    _ptiddata ptd;           // Pointer to thread's data block
    unsigned long thd1;      // Thread's handle

    // Allocate data block for the new thread.
    if ((ptd = _calloc_crt(1, sizeof(struct tiddata))) == NULL)
        goto error_return;

    // Initialize the data block.
    initptd(ptd);

    // Save the desired thread function and the parameter
    // we want it to get in the data block.
    ptd->_initaddr = (void *) pfnStartAddr;
    ptd->_initarg = pvParam;

    // Create the new thread.
    thd1 = (unsigned long) CreateThread(psa, cbStack,
        _threadstartex, (PVOID) ptd, fdwCreate, pdwThreadID);
    if (thd1 == NULL) {
        // Thread couldn't be created, cleanup and return failure.
```



```

    goto error_return;
}

// Create created OK, return the handle.
return(thd1);

error_return:
// Error: data block or thread couldn't be created.

_free_crt(ptd);
return((unsigned long)0L);
}

```

下面是关于_beginthreadex的一些要点：

- 每个线程均获得由C/C++运行期库的堆栈分配的自己的tiddata内存结构。(tiddata结构位于Mtdll.h文件中的Visual C++源代码中)。我在清单6-1中重建了它的结构。
- 传递给_beginthreadex的线程函数的地址保存在tiddata内存块中。传递给该函数的参数也保存在该数据块中。
- _beginthreadex确实从内部调用CreateThread，因为这是操作系统了解如何创建新线程的唯一方法。
- 当调用CreateThread时，它被告知通过调用_threadstartex而不是pfnStartAddr来启动执行新线程。还有，传递给线程函数的参数是tiddata结构而不是pvParam的地址。
- 如果一切顺利，就会像CreateThread那样返回线程句柄。如果任何操作失败了，便返回NULL。

清单6-1 C/C++运行期库的线程局部tiddata结构

```

struct _tiddata {
    unsigned long    _tid;        /* thread ID */

    unsigned long    _thandle;    /* thread handle */

    int              _terrno;      /* errno value */
    unsigned long    _tdoserrno;  /* _doserrno value */
    unsigned int      _fpds;      /* Floating Point data segment */
    unsigned long    _holdrand;   /* rand() seed value */
    char *           _token;      /* ptr to strtok() token */
#ifdef _WIN32
    wchar_t *        _wtoken;     /* ptr to wcstok() token */
#endif /* _WIN32 */
    unsigned char *   _mtoken;    /* ptr to _mbstok() token */

    /* following pointers get malloc'd at runtime */
    char *            _errmsg;     /* ptr to strerror()/_strerror() buff */
    char *            _namebuf0;   /* ptr to tmpnam() buffer */
#ifdef _WIN32
    wchar_t *         _wnamebuf0;  /* ptr to _wtmpnam() buffer */
#endif /* _WIN32 */
    char *            _namebuf1;   /* ptr to tmpfile() buffer */
#ifdef _WIN32
    wchar_t *         _wnamebuf1;  /* ptr to _wtmpfile() buffer */
#endif /* _WIN32 */
    char *            _asctimebuf; /* ptr to asctime() buffer */
}

```

```

#ifdef _WIN32
    wchar_t * _wasctimebuf; /* ptr to _wasctime() buffer */
#endif /* _WIN32 */
void * _gmtimebuf; /* ptr to gmtime() structure */
char * _cvtbuf; /* ptr to ecvt()/fcvt buffer */

/* following fields are needed by _beginthread code */
void * _initaddr; /* initial user thread address */
void * _initarg; /* initial user thread argument */

/* following three fields are needed to support signal handling and
 * runtime errors */
void * _pxcptacttab; /* ptr to exception-action table */
void * _tpxcptinfopters; /* ptr to exception info pointers */
int _tfpecode; /* float point exception code */

/* following field is needed by NLG routines */
unsigned long _NLG_dwCode;

/*
 * Per-Thread data needed by C++ Exception Handling
 */
void * _terminate; /* terminate() routine */
void * _unexpected; /* unexpected() routine */
void * _translator; /* S.E. translator */
void * _curexception; /* current exception */
void * _curcontext; /* current exception context */
#ifdef (_M_MRX000)
    void * _pFrameInfoChain;
    void * _pUnwindContext;
    void * _pExitContext;
    int _MipsPtdDelta;
    int _MipsPtdEpsilon;
#elif defined (_M_PPC)
    void * _pExitContext;
    void * _pUnwindContext;
    void * _pFrameInfoChain;
    int _FrameInfo[6];
#endif /* defined (_M_PPC) */
};

typedef struct _tiddata * _ptiddata;

```

既然为新线程指定了tiddata结构，并且对该结构进行了初始化，那么必须了解该结构与线程之间是如何关联起来的。让我们观察一下_threadstartex函数（它也位于C/C++运行期库的Threadex.c文件中）。这里是该函数的伪代码版本：

```

static unsigned long WINAPI threadstartex (void* ptd) {
    // Note: ptd is the address of this thread's tiddata block.

    // Associate the tiddata block with this thread.
    TlsSetValue(__tlsindex, ptd);

    // Save this thread ID in the tiddata block.
    ((_ptiddata) ptd)->_tid = GetCurrentThreadId();
}

```

```
// Initialize floating-point support (code not shown).

// Wrap desired thread function in SEH frame to
// handle run-time errors and signal support.
__try {
    // Call desired thread function, passing it the desired parameter.
    // Pass thread's exit code value to _endthreadex.
    _endthreadex(
        ( (unsigned (WINAPI *) (void *)) (( _ptiddata) ptd) -> _initaddr ) )
        ( ( _ptiddata) ptd) -> _initarg );
}
__except( _XcptFilter( GetExceptionCode(), GetExceptionInformation() ) ) {
    // The C run-time's exception handler deals with run-time errors
    // and signal support; we should never get it here.
    _exit( GetExceptionCode() );
}

// We never get here; the thread dies in this function.
return( 0L );
}
```

下面是关于 `_threadstartex` 的一些重点：

- 新线程开始从 `BaseThreadStart` 函数（在 `kernel32.dll` 文件中）执行，然后转移到 `_threadstartex`。
- 到达该新线程的 `tiddata` 块的地址作为其唯一参数被传递给 `_threadstartex`。
- `TlsSetValue` 是个操作系统函数，负责将一个值与调用线程联系起来。这称为线程本地存储器（TLS），将在第21章介绍。`_threadstartex` 函数将 `tiddata` 块与线程联系起来。
- 一个SEH帧被放置在需要的线程函数周围。这个帧负责处理与运行期库相关的许多事情——例如，运行期错误（比如放过了没有抓住的 C++ 异常条件）和 C/C++ 运行期库的 `signal` 函数。这是特别重要的。如果用 `CreateThread` 函数来创建线程，然后调用 C/C++ 运行期库的 `signal` 函数，那么该函数就不能正确地运行。
- 调用必要的线程函数，传递必要的参数。记住，函数和参数的地址由 `_beginthreadex` 保存在 `tiddata` 块中。
- 必要的线程函数返回值被认为是线程的退出代码。注意，`_threadstartex` 并不只是返回到 `BaseThreadStart`。如果它准备这样做，那么线程就终止运行，它的退出代码将被正确地设置，但是线程的 `tiddata` 内存块不会被撤消。这将导致应用程序中出现一个漏洞。若要防止这个漏洞，可以调用另一个 C/C++ 运行期库函数 `_endthreadex`，并传递退出代码。

需要介绍的最后一个函数是 `_endthreadex`（位于 C 运行期库的 `Threadex.c` 文件中）。下面是该函数的伪代码版本：

```
void __cdecl _endthreadex (unsigned retcode) {
    _ptiddata ptd;          // Pointer to thread's data block

    // Clean up floating-point support (code not shown).

    // Get the address of this thread's tiddata block.
    ptd = _getptd();

    // Free the tiddata block.
    _freeptd(ptd);
}
```

```
// Terminate the thread.
ExitThread(retcode);
}
```

下面是关于_endthreadex的一些要点：

- C运行期库的_getptd函数内部调用操作系统的 TlsGetValue函数，该函数负责检索调用线程的tiddata内存块的地址。
- 然后该数据块被释放，而操作系统的ExitThread函数被调用，以便真正撤消该线程。当然，退出代码要正确地设置和传递。

本章前面说过，始终都应该设法避免使用 ExitThread函数。这一点完全正确，我并不要收回我已经说过的话。ExitThread 函数将撤消调用函数，并且不允许它从当前执行的函数返回。由于该函数不能返回，所以创建的任何 C++对象都不会被撤消。避免调用 ExitThread的另一个原因是，它会使得线程的tiddata内存块无法释放，这样，应用程序将会始终占用内存（直到整个进程终止运行为止）。

Microsoft的Visual C++开发小组认识到编程人员喜欢调用 ExitThread，因此他们实现了他们的愿望，并且不会让应用程序始终占用内存。如果真的想要强制撤消线程，可以让它调用 _endthreadex（而不是调用ExitThread）以便释放线程的tiddata块，然后退出。不过建议不要调用_endthreadex函数。

现在应该懂得为什么 C/C++运行期库的函数需要为它创建的每个线程设置单独的数据块，同时，也应该了解如何通过调用 _beginthreadex来分配数据块，再对它进行初始化，将该数据块与你创建的线程联系起来。你还应该懂得 _endthreadex函数是如何在线程终止运行时释放数据块的。

一旦数据块被初始化并且与线程联系起来，线程调用的任何需要单线程实例数据的 C/C++运行期库函数都能很容易地（通过 TlsGetValue）检索调用线程的数据块地址，并对线程的数据进行操作。这对于函数来说很好，但是你可能想知道它对 errno之类的全局变量效果如何。Errno定义在标准的C头文件中，类似下面的形式：

```
#if defined(_MT) || defined(_DLL)
extern int * __cdecl _errno(void);
#define errno (*_errno())
#else /* ndef _MT && ndef _DLL */
extern int errno;
#endif /* _MT || _DLL */
```

如果创建一个多线程应用程序，必须在编译器的命令行上设定 /MT（指多线程应用程序）或/MD（指多线程DLL）开关。这将使编译器能够定义 _MT标识符。然后，每当引用errno时，实际上是调用内部的C/C++运行期库函数_errno。该函数返回调用线程的相关数据块中的 errno数据成员的地址。你将会发现，errno宏被定义为获取该地址的内容的宏。这个定义是必要的，因为可以编写类似下面形式的代码：

```
int *p = &errno;
if (*p == ENOMEM) {
    :
    :
}
```

如果内部函数_errno只返回errno的值，那么上面的代码将不进行编译。

多线程版本的C/C++运行期库还给某些函数设置了同步的基本要素。例如，如果两个线程同时调用 malloc，那么内存堆栈就可能遭到破坏。多线程版本的 C/C++运行期库能够防止两个线程同时从堆栈中分配内存。为此，它要让第二个线程等待，直到第一个线程从 malloc 返回。然后第二个线程才被允许进入（关于线程同步的问题将在第 8、9 章和 10 章详细介绍）。

显然，所有这些附加操作都会影响多线程版本的 C/C++运行期库的性能。这就是为什么 Microsoft 公司除了多线程版本外，还提供单线程版本的静态链接的 C/C++运行期库的原因。

C/C++运行期库的动态连接版本编写成为一种通用版本。这样它就可以被使用 C/C++运行期库函数的所有正在运行的应用程序和 DLL 共享。由于这个原因，运行期库只存在于多线程版本中。由于 DLL 中提供了 C/C++运行期库，因此应用程序（.exe 文件）和 DLL 不需要包含 C/C++运行期库函数的代码，结果它们的规模就比较小。另外，如果 Microsoft 排除了 C/C++运行期库 DLL 中的错误，应用程序中的错误也会自动得到解决。

正如希望的那样，C/C++运行期库的启动代码为应用程序的主线程分配了数据块，并且对数据块进行了初始化，这样，主线程就能安全地调用 C/C++运行期函数中的任何函数。当主线程从它的进入点函数返回时，C/C++运行期库就会释放相关的数据块。此外，启动代码设置了相应的结构化异常处理代码，以便主线程能够成功地调用 C/C++运行期库的 signal 函数。

6.7.1 Oops——错误地调用了 CreateThread

也许你想知道，如果调用 CreateThread，而不是调用 C/C++运行期库的 _beginthreadex 来创建新线程，将会发生什么情况。当一个线程调用要求 tiddata 结构的 C/C++运行期库函数时，将会发生下面的一些情况（大多数 C/C++运行期库函数都是线程安全函数，不需要该结构）。首先，C/C++运行期库函数试图（通过调用 TlsGetValue）获取线程的数据块的地址。如果返回 NULL 作为 tiddata 块的地址，调用线程就不拥有与该地址相关的 tiddata 块。这时，C/C++运行期库函数就在现场为调用线程分配一个 tiddata 块，并对它进行初始化。然后该 tiddata 块（通过 TlsSetValue）与线程相关联。此时，只要线程在运行，该 tiddata 将与线程待在一起。这时，C/C++运行期库函数就可以使用线程的 tiddata 块，而且将来被调用的所有 C/C++运行期函数也能使用 tiddata 块。

当然，这看来有些奇怪，因为线程运行时几乎没有任何障碍。不过，实际上还是存在一些問題。首先，如果线程使用 C/C++运行期库的 signal 函数，那么整个进程就会终止运行，因为结构化异常处理帧尚未准备好。第二，如果不是调用 _endthreadex 来终止线程的运行，那么数据块就不会被撤消，内存泄漏就会出现（那么谁还为使用 CreateThread 函数创建的线程来调用 _endthreadex 呢？）。

注意 如果程序模块链接到多线程 DLL 版本的 C/C++运行期库，那么当线程终止运行并释放 tiddata 块（如果已经分配了 tiddata 块的话）时，该运行期库会收到一个 DLL_THREAD_DETACH 通知。尽管这可以防止 tiddata 块的泄漏，但是强烈建议使用 _bdginthreadex 而不是使用 Createthread 来创建线程。

6.7.2 不应该调用的 C/C++运行期库函数

C/C++运行期库也包含另外两个函数：

```
unsigned long _beginthread(  
    void (__cdecl *start_address)(void *),  
    unsigned stack_size,
```



```
void *arglist);
```

和

```
void _endthread(void);
```

创建这两个函数的目的是用来执行 `_beginthreadex` 和 `_endthreadex` 函数的功能。但是，如你所见，`_beginthread` 函数的参数比较少，因此比特性全面的 `_beginthreadex` 函数受到更大的限制。例如，如果使用 `_beginthread`，就无法创建带有安全属性的新线程，无法创建暂停的线程，也无法获得线程的 ID 值。`_endthread` 函数的情况与之类似。它不带参数，这意味着线程的退出代码必须硬编码为 0。

`endthread` 函数还存在另一个很难注意到的大问题。在 `_endthread` 调用 `ExitThread` 之前，它调用 `CloseHandle`，传递新线程的句柄。若要了解为什么这是个大问题，请看下面的代码：

```
DWORD dwExitCode;  
HANDLE hThread = _beginthread(...);  
GetExitCodeThread(hThread, &dwExitCode);  
CloseHandle(hThread);
```

新创建的线程可以在第一个线程调用 `GetExitCodeThread` 之前运行、返回和终止。如果出现这种情况，`hThread` 中的值将无效，因为 `_endthread` 已经关闭了新线程的句柄。不用说，由于相同的原因，对 `CloseHandle` 的调用也将失败。

新的 `_endthreadex` 函数并不关闭线程的句柄，因此，如果用调用 `beginthreadex` 来取代调用 `_beginthread`，那么上面的代码段将能正确运行。记住，当线程函数返回时，`_beginthreadex` 调用 `_endthreadex`，而 `_beginthread` 则调用 `_endthread`。

6.8 对自己的 ID 概念应该有所了解

当线程运行时，它们常常想要调用 Windows 函数来改变它们的运行环境。例如，线程可能想要改变它的优先级或它的进程的优先级（优先级将在第 7 章中介绍）。由于线程常常要改变它的（或它的进程的）环境，因此 Windows 提供了一些函数，使线程能够很容易引用它的进程内核对象，或者引用它自己的线程内核对象：

```
HANDLE GetCurrentProcess();  
HANDLE GetCurrentThread();
```

上面这两个函数都能返回调用线程的进程的伪句柄或线程内核对象的伪句柄。这些函数并不在创建进程的句柄表中创建新句柄。还有，调用这些函数对进程或线程内核对象的使用计数没有任何影响。如果调用 `CloseHandle`，将伪句柄作为参数来传递，那么 `CloseHandle` 就会忽略该函数的调用并返回 `FALSE`。

当调用一个需要进程句柄或线程句柄的 Windows 函数时，可以传递一个伪句柄，使该函数执行它对调用进程或线程的操作。例如，通过调用下面的 `GetProcessTimes` 函数，线程可以查询它的进程的时间使用情况：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;  
GetProcessTimes(GetCurrentProcess(),  
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

同样，通过调用 `GetThreadTimes` 函数，线程可以查询它自己的线程时间：

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;  
GetThreadTimes(GetCurrentThread(),  
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

少数 Windows 函数允许用进程或线程在系统范围内独一无二的 ID 来标识某个进程或线程。

下面这两个函数使得线程能够查询它的进程的惟一 ID 或它自己的惟一 ID：

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

这两个函数通常不像能够返回伪句柄的函数那样有用，但是有的时候用起来还是很方便的。

将伪句柄转换为实句柄

有时可能需要获得线程的实句柄而不是它的伪句柄。所谓“实句柄”，我是指用来明确标识一个独一无二的线程的句柄。请看下面的代码：

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent = GetCurrentThread();
    CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
    // Function continues...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    // Function continues...
}

```

你能发现这个代码段存在的问题吗？这个代码的目的是让父线程给子线程传递一个线程句柄，以标识父线程。但是，父线程传递了一个伪句柄，而不是一个实句柄。当子线程开始运行时，它将一个伪句柄传递给 `GetThreadTime` 函数，使子线程得到它自己的 CPU 时间，而不是父线程的 CPU 时间。出现这种情况的原因是线程的伪句柄是当前线程的句柄，也就是说，它是调用函数的线程的句柄。

为了修改这个代码，必须将伪句柄变成实句柄。DuplicateHandle函数能够执行这一转换：

```

BOOL DuplicateHandle(
    HANDLE hSourceProcess,
    HANDLE hSource,
    HANDLE hTargetProcess,
    PHANDLE phTarget,
    DWORD fdwAccess,
    BOOL bInheritHandle,
    DWORD fdwOptions);

```

通常可以使用这个函数，用与另一个进程相关的内核对象来创建一个与进程相关的新句柄。然而，可以用一种特殊的方法来使用这个函数，以便修改上面介绍的代码段。正确的代码段应该是下面的样子：

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent;

    DuplicateHandle(
        GetCurrentProcess(),           // Handle of process that thread
                                      // pseudo-handle is relative to
        GetCurrentThread(),            // Parent thread's pseudo-handle
        GetCurrentProcess(),           // Handle of process that the new, real,
                                      // thread handle is relative to
        &hThreadParent,               // Will receive the new, real, handle
    );
}

```

```
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(),    // Handle of process that the process
                           // pseudo-handle is relative to
    GetCurrentProcess(),    // Process's pseudo-handle
    GetCurrentProcess(),    // Handle of process that the new, real,
                           // process handle is relative to
    &hProcess,              // Will receive the new, real
                           // handle identifying the process
    0,                      // Ignored because of DUPLICATE_SAME_ACCESS
    FALSE,                  // New thread handle is not inheritable
    DUPLICATE_SAME_ACCESS); // New process handle has same
                           // access as pseudo-handle
```