

## Windows 调试工具入门-3

### 基本调试操作

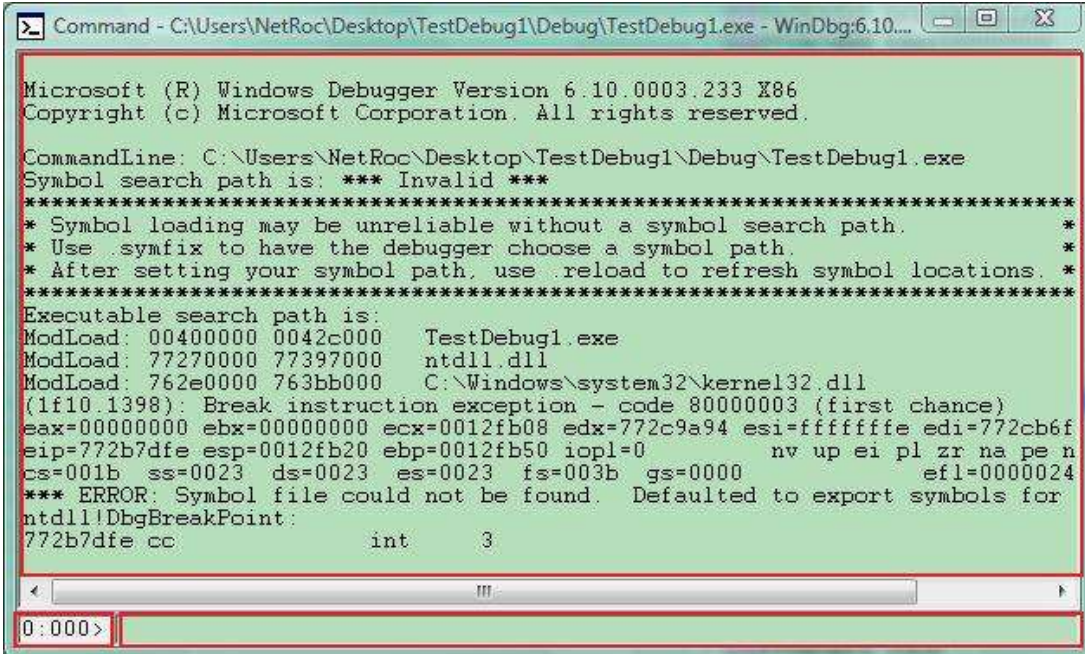
<http://www.DbgTech.net>

#### 一、 调试器命令窗口

##### 1、简介

使用 Windows 调试工具进行调试，大部分和调试器之间的交互都是通过调试器命令窗口来进行的。命令的输入、输出都是在调试器命令窗口中显示出来。对 WinDbg 来说，调试器命令窗口是名为“Command”的窗口；对于 KD、CDB 和 NTSD 来说，整个命令行窗口就是调试器命令窗口。这里主要介绍 WinDbg 中的调试器命令窗口。

一般来说 WinDbg 运行之后都会打开一个标题为 Command 的子窗口，在没有调试目标的时候，这个窗口是不能接受输入输出的，这时 WinDbg 处于静止模式，只有在打开调试目标之后，才能够使用它和调试器交互。



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.10.0003.233
Microsoft (R) Windows Debugger Version 6.10.0003.233 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe
Symbol search path is: *** Invalid ***
*****
* Symbol loading may be unreliable without a symbol search path.
* Use .symfix to have the debugger choose a symbol path.
* After setting your symbol path, use .reload to refresh symbol locations.
*****
Executable search path is:
ModLoad: 00400000 0042c000 TestDebug1.exe
ModLoad: 77270000 77397000 ntdll.dll
ModLoad: 762e0000 763bb000 C:\Windows\system32\kernel32.dll
(1f10.1398): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0012fb08 edx=772c9a94 esi=ffffffe edi=772cb6f
eip=772b7dfe esp=0012fb20 ebp=0012fb50 iopl=0         nv up ei pl zr na pe n
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000024
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
ntdll!DbgBreakPoint:
772b7dfe cc                int     3

0:000>
```

窗口分为三个部分：

- 位于上部的面积最大的是命令输出窗口。所有的命令输出、目标程序的调试信息输出等等都会在里面显示出来。上一篇中介绍的调试器日志中记录的就是显示在这里的内容。
- 下半部分左边是提示符窗口。这里通过提示符能够快速知道调试器目前的状态。上图中 **0:000>**，冒号前的数字表示当前的进程号，同时调试多个进程时，每个进程都会被指派一个进程号；冒号后的 000 表示线程号。进行内核调试时，如果是单处理器系统，提示符是 **kd>** 的形式；如果是多处理器系统，则是 **0: kd>** 的形式，前面的 0 表示处理器号。

提示符还可能是\***BUSY**\*这样的字符串，以表示调试器正忙。也可以通过命令来自定义提示符。

- 下半部分右边是命令输入窗口。需要执行的命令就在这里输入。


调试器命令窗口中输入命令时可以使用一些快捷操作：

- 上下方向键可以查找先前的命令。
- **ESC** 键用于清除当前行的命令。
- **TAB** 键用于自动补完命令。例如一些符号可以只输入一部分，然后通过按下 **TAB** 一次或多次来找到需要的符号。
- 鼠标右键点击命令窗口，可以将剪贴板中的内容粘贴到命令输入框中。
- 直接按下 **ENTER** 键重复上一条命令。这个功能在 **WinDbg** 中可以通过命令来打开或关闭。
- 如果某条命令产生了很长的输出，可以按下 **CTRL+BREAK** 来中断它。

## 二、 控制调试目标的执行

这里的控制目标执行，主要是指如何让运行中的目标中断到调试器中，以及控制中断的目标如何继续执行。

### 1. 中断调试目标

当调试目标处于运行状态时，**WinDbg** 是不能输入命令或者对它进行操作的。可以通过按下 **CTRL+BREAK** 或者 点击工具栏的  按钮来中断它。下面我们继续用上一篇中的 **TestDebug1** 项目来说明。修改 **TestDebug1.cpp** 如下：

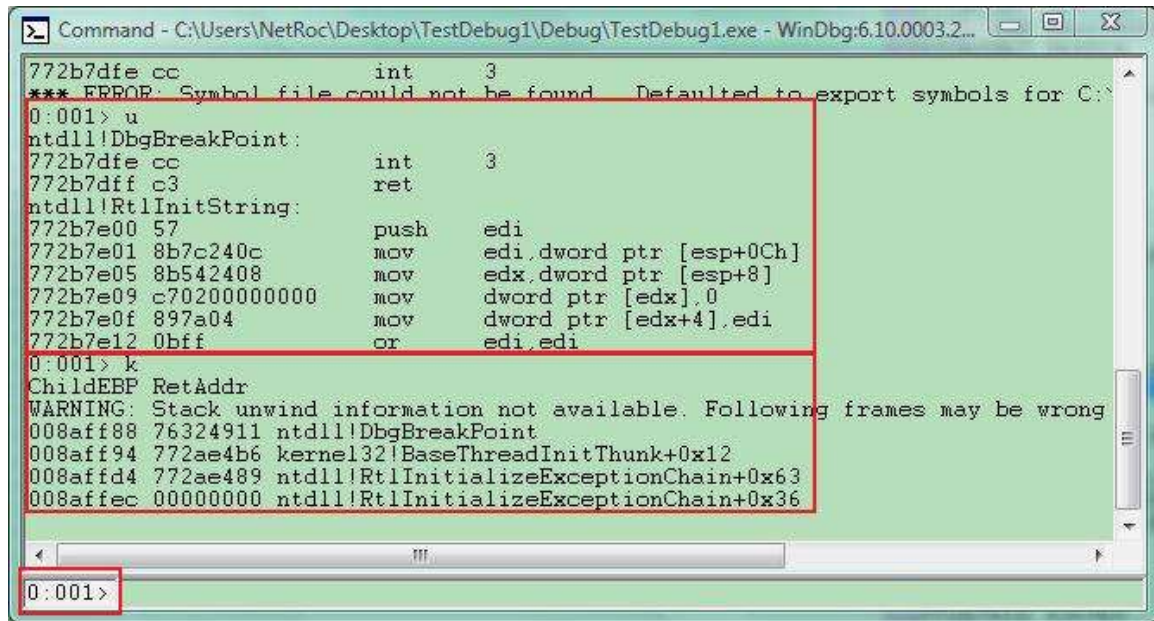
```
#include "stdafx.h"

#include <stdio.h>

int main(int argc, char* argv[])
{
    int i = 0;
    while( 1)
    {
        printf( "TestDebug1.cpp:%d\r\n", i);
    }
    return 0;
}
```

为了方便，这次使用 **Debug** 选项来重新编译它，这样就不用再设置编译选项和 **WinDbg** 选项来查看符号了。使用 **WinDbg** 菜单的 **File->Open Executable...** 打开 **TestDebug1.exe**，中断下来之后 **F5** 继续运行。由于是个死循环，所以目标不会自己停止下来，可以看到 **WinDbg**

的调试器命令窗口一直处于禁用状态。在 WinDbg 窗口按下 CTRL+BREAK，TestDebug1.exe 就中断到调试器中了，使用 u 命令查看当前正在执行的代码，k 命令查看当前调用堆栈：



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.10.0003.2...
772b7dfe cc          int      3
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\
0:001> u
ntdll!DbgBreakPoint:
772b7dfe cc          int      3
772b7dff c3          ret
ntdll!RtlInitString:
772b7e00 57          push    edi
772b7e01 8b7c240c     mov     edi_dword ptr [esp+0Ch]
772b7e05 8b542408     mov     edx_dword ptr [esp+8]
772b7e09 c70200000000 mov     dword ptr [edx],0
772b7e0f 897a04      mov     dword ptr [edx+4],edi
772b7e12 0bff        or      edi,edi
0:001> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong
008aff88 76324911 ntdll!DbgBreakPoint
008aff94 772ae4b6 kernel32!BaseThreadInitThunk+0x12
008affd4 772ae489 ntdll!RtlInitializeExceptionChain+0x63
008affec 00000000 ntdll!RtlInitializeExceptionChain+0x36
0:001>
```

看调用堆栈和反汇编出来的代码，似乎和 TestDebug1.cpp 中的代码没有任何关系，这是为什么呢？

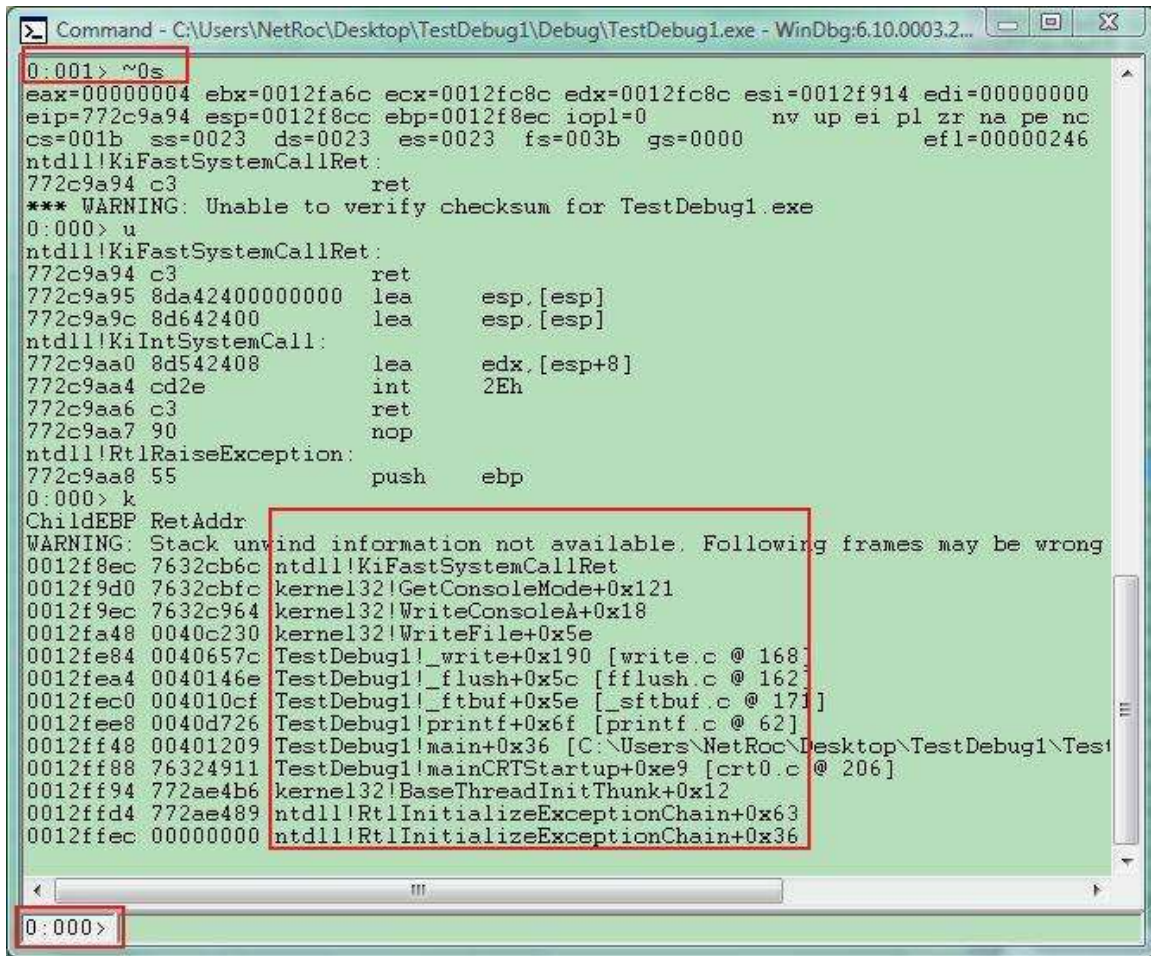
注意到底部提示符位置显示的是 0:001>，说明这是 1 号线程，而正常情况下线程编号都是从 0 开始的。我们继续用 ~ 命令来查看被调试进程中的线程信息，出现的是类似这样的输出：

```
0:001> ~
0 Id: 1998.1358 Suspend: 1 Teb: 7ffde000 Unfrozen
. 1 Id: 1998.17f8 Suspend: 1 Teb: 7ffdd000 Unfrozen
```

每一行是一个线程的信息。第一行中，0 表示这个进程的编号；1998.1358 是 16 进制数字，前者是当前进程的进程 ID，后者是线程 ID；后面的信息是线程状态和 Teb 地址。第二行的线程编号前有一个点号“.”，表示这是当前线程，也就是刚才使用 u 和 k 命令查看到的线程。

我们的代码中并没有任何创建线程的操作，为什么会多出一个线程来呢？这是由于 WinDbg 中断运行中的调试目标的方式造成的。按下 CTRL+BREAK 之后，WinDbg 会在调试目标的进程中创建一个远线程，并在这个远线程中执行 ntdll!DbgBreakPoint 函数，即上面 u 命令所显示出来的内容。它会在目标进程中产生一次 int3 异常，这个异常被 WinDbg 捕获，所以 TestDebug1.exe 就中断到调试器中了。因此，当采用 CTRL+BREAK 这种方式中断目标之后，看到的代码是在这个远线程中的，如果要查看调试目标正在执行的代码就需要切换当前线程。可以使用 ~Thread s 命令。如下：







```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.10.0003.2...
0:001> ~0s
eax=00000004 ebx=0012fa6c ecx=0012fc8c edx=0012fc8c esi=0012f914 edi=00000000
eip=772c9a94 esp=0012f8cc ebp=0012f8ec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
772c9a94 c3          ret
*** WARNING: Unable to verify checksum for TestDebug1.exe
0:000> u
ntdll!KiFastSystemCallRet:
772c9a94 c3          ret
772c9a95 8da4240000000000 lea     esp,[esp]
772c9a9c 8d642400          lea     esp,[esp]
ntdll!KiIntSystemCall:
772c9aa0 8d542408          lea     edx,[esp+8]
772c9aa4 cd2e          int     2Eh
772c9aa6 c3          ret
772c9aa7 90          nop
ntdll!RtlRaiseException:
772c9aa8 55          push    ebp
0:000> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong
0012f8ec 7632cb6c ntdll!KiFastSystemCallRet
0012f9d0 7632cbfc kernel32!GetConsoleMode+0x121
0012f9ec 7632c964 kernel32!WriteConsoleA+0x18
0012fa48 0040c230 kernel32!WriteFile+0x5e
0012fe84 0040657c TestDebug1!_write+0x190 [write.c @ 168]
0012fea4 0040146e TestDebug1!_flush+0x5c [fflush.c @ 162]
0012fec0 004010cf TestDebug1!_ftbuf+0x5e [_sftbuf.c @ 171]
0012fee8 0040d726 TestDebug1!printf+0x6f [printf.c @ 62]
0012ff48 00401209 TestDebug1!main+0x36 [C:\Users\NetRoc\Desktop\TestDebug1\Test
0012ff88 76324911 TestDebug1!mainCRTStartup+0xe9 [crt0.c @ 206]
0012ff94 772ae4b6 kernel32!BaseThreadInitThunk+0x12
0012ffd4 772ae489 ntdll!RtlInitializeExceptionChain+0x63
0012ffec 00000000 ntdll!RtlInitializeExceptionChain+0x36
0:000>
```

这里就可以清楚看到在 main 函数中的 print 调用产生的调用堆栈了。

除了采用 CTRL+BREAK 这样直接中断运行中目标的方式之外，当调试目标发生异常、退出或者遭遇断点等事件时，也会自动中断到调试器中。这时就不会出现额外的线程了。内核调试时中断目标机的操作和用户模式下一样。

## 2. 控制目标的执行

调试目标中断之后，就可以通过单步或者跟踪指令来控制它执行了。

WinDbg 中的单步操作快捷键和 Visual Studio 调试器中相同。也是 F5 运行、F10 逐过程单步、F11 逐语句单步。需要注意的是，单步的定义在汇编模式调试和源码模式调试时是不一样的。汇编模式调试时，每次单步执行一条指令；源码模式调试时，每次单步执行一行源码。点击工具栏上的  按钮或使用 lt 命令来启用汇编模式；点击工具栏上的  或使用 ltt 命令来启用源码模式。

控制目标执行的命令分为三大类。g\*类的命令用于直接运行目标、p\*类的命令用于单步执行、t\*类的命令类似 p\*命令，但是当遇到 call 指令时会跟踪进去。下面是这些命令的列表，摘自 WinDbg 帮助文档：

命令	WinDbg 按钮	WinDbg 命令	WinDbg 快捷键	作用
		<a href="#">Debug   Run to Cursor</a>	F7 CTRL + F10	(仅 WinDbg) 运行到光标位置。
		<a href="#">Debug   Stop Debugging</a>	SHIFT + F5	停止所有的调试并关闭目标。
(仅 CDB/KD) <a href="#">CTRL+C</a>		<a href="#">Debug   Break</a>	CTRL + BREAK	停止执行，调试器中断目标。
<a href="#">.restart (Restart Target Application)</a>		<a href="#">Debug   Restart</a>	CTRL + SHIFT + F5	(仅 User mode) 重起目标程序
<a href="#">g (Go)</a>		<a href="#">Debug   Go</a>	F5	目标自由执行。
<a href="#">gc (Go from Conditional Breakpoint)</a>				在一次 <a href="#">条件断点</a> 之后恢复执行。
<a href="#">gh (Go with Exception Handled)</a>		<a href="#">Debug   Go Handled Exception</a>		和 <b>g (Go)</b> 相同，但是当前异常被当作已处理。
<a href="#">gn (Go with Exception Not Handled)</a>		<a href="#">Debug   Go Unhandled Exception</a>		和 <b>g (Go)</b> 相同，但是当前异常被当作未处理。
<a href="#">gu (Go Up)</a>		<a href="#">Debug   Step Out</a>		目标运行到当前函数执行完成。
<a href="#">p (Step)</a>		<a href="#">Debug   Step Over</a>		目标执行一条指令。如果该指令是函数调用，则这个调用被当作一步执行。
<a href="#">pa (Step to Address)</a>				目标运行直到到达指定的地址。该函数中执行的每一步都会显示出来(但是不显示被调用的函数中的内容。)
<a href="#">pc (Step to Next Call)</a>				目标运行直到遇到下一个 <b>call</b> 指令。如果当前指令是 <b>call</b> ，则这个 <b>call</b> 会被完成并执行到

				下一个 <b>call</b> 。
<a href="#">pct (Step to Next Call or Return)</a>				目标继续执行，直到遇到一个 <b>call</b> 指令或者 <b>return</b> 指令。
<a href="#">ph (Step to Next Branching Instruction)</a>				目标执行，直到到达任何一种分支指令，包括条件和非条件分支、 <b>call</b> 调用、函数返回和系统调用。
<a href="#">pt (Step to Next Return)</a>				目标执行，直到遇到 <b>return</b> 指令。
<a href="#">t (Trace)</a>		<a href="#">Debug   Step Into</a>	F11 F8	目标执行一条指令。如果该指令是一条 <b>call</b> ，调试器跟踪到这个 <b>call</b> 中。
<a href="#">ta (Trace to Address)</a>				目标执行直到指定地址。本函数和被调用函数中的每一步都会显示出来。
<a href="#">tb (Trace to Next Branch)</a>				<i>(除内核模式之外的所有模式，仅在基于 x86 的系统上)</i> 目标运行到下一条分支指令。
<a href="#">tc (Trace to Next Call)</a>				目标运行到下一条 <b>call</b> 指令。如果当前指令是 <b>call</b> ，该命令会跟踪进去直到遇到另一条 <b>call</b> 。
<a href="#">tct (Trace to Next Call or Return)</a>				目标运行到下一条 <b>call</b> 指令或 <b>return</b> 指令。如果当前指令是 <b>call</b> 或 <b>return</b> ，命令会跟踪进去知道遇到另一个 <b>call</b> 或 <b>return</b> 。
<a href="#">th (Trace to Next Branching Instruction)</a>				目标执行直到遇到任意类型的分支指令，包括条件和非条件跳转、 <b>call</b> 、 <b>return</b> 和系统调用。如果当前指令是分

				支指令，该命令跟踪进入直到遇到下一个分支指令。
<a href="#">tt (Trace to Next Return)</a>				目标运行直到遇到 <b>return</b> 指令。如果当前指令是一条 <b>return</b> ，则跟踪进入直到另外一条 <b>return</b> 。
<a href="#">wt (Trace and Watch Data)</a>				目标执行，直到指定的函数执行完成。这时会显示统计信息。

### 三、 使用断点

合理、巧妙的设置断点是软件调试中的一门艺术，好的断点能使调试工作事半功倍。WinDbg 中提供了丰富的断点命令，下面通过示例对这些命令进行简单的介绍。

在上面的项目中，添加了一个 dll 项目，名为 TestDebugDll1。修改一下上面的 TestDebug1.cpp 如下(整个项目可以下载附件)：

```
#include "stdafx.h"

#include <stdio.h>

#include <windows.h>

class CTestClass
{
public:
    CTestClass(){};
    ~CTestClass(){};
    void SetChar( unsigned char ucChar)
    {
        m_ucTestChar = ucChar;
    }
protected:
    unsigned char m_ucTestChar;
```

```
};

int main(int argc, char* argv[])
{
    typedef int (*pfnTestDllAdd)( int a, int b);

    int i;

    HMODULE hMod = LoadLibraryA( "TestDebugDll1.dll");

    pfnTestDllAdd TestDllAdd = (pfnTestDllAdd)::GetProcAddress( hMod, "TestDllAdd");

    if ( TestDllAdd)
    {
        i = TestDllAdd( 1, 2);
    }

    CTestClass objTestClass;

    objTestClass.SetChar( 123);

    return 0;
}
```

还是使用 **Debug** 选项，重新编译。用 **WinDbg** 打开 **TestDebug1.exe** 后会自动中断到初始断点。由于是 **Debug** 选项编译的，所以这里可以省去符号路径的设置就能识别符号。

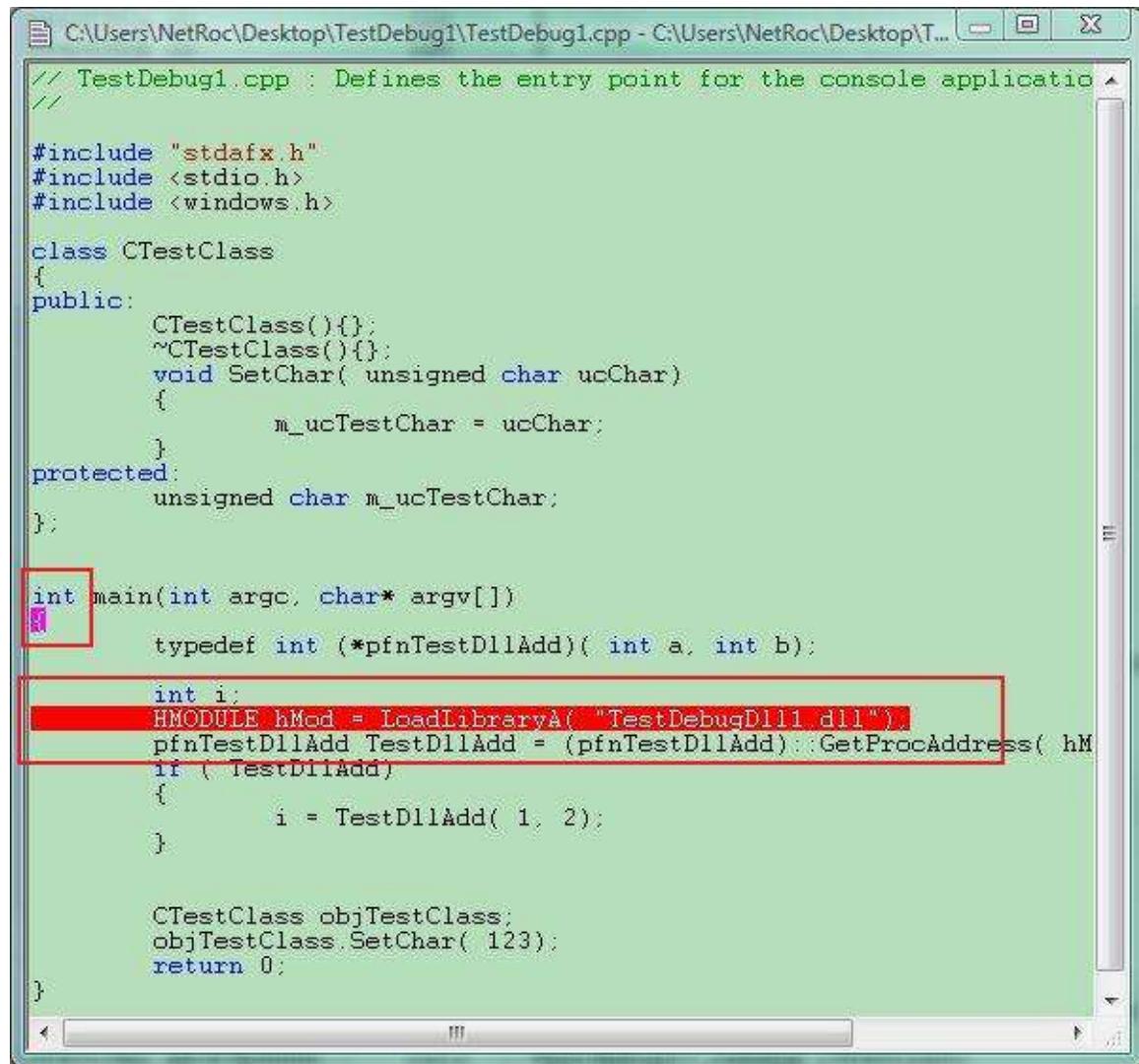
**bp** 命令是最常用的断点命令之一，它可以直接对某个代码地址设置断点。例如我们想中断到 **main** 函数，可以这样：

```
0:000> bp TestDebug1!main
```

前面的 **TestDebug1** 明确指定 **main** 符号所在的模块，这样通常可以减少搜索符号的时间，也避免了相同名字的符号可能造成的冲突。**F5** 运行，就发现已经中断到 **main** 函数了，并且源码窗口会自动弹出来。

在源码窗口或者反汇编窗口中，可以将光标移动到要设置断点的行并用 **F9** 快捷键来设置断点。这和 **Visual Studio** 中一样。现在我们在 **HMODULE hMod = LoadLibraryA( "TestDebugDll1.dll");** 这一行处按下 **F9** 设置一个断点。可以看到源码窗口中将当前正中断到的断点和未触发的断点用不同的颜色标识出来：





```
// TestDebug1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <windows.h>

class CTestClass
{
public:
    CTestClass(){};
    ~CTestClass(){};
    void SetChar( unsigned char ucChar)
    {
        m_ucTestChar = ucChar;
    }
protected:
    unsigned char m_ucTestChar;
};

int main(int argc, char* argv[])
{
    typedef int (*pfnTestDllAdd)( int a, int b);

    int i;
    HMODULE hMod = LoadLibraryA( "TestDebugDll1.dll");
    pfnTestDllAdd TestDllAdd = (pfnTestDllAdd) GetProcAddress( hMod, "TestDllAdd");
    if ( TestDllAdd)
    {
        i = TestDllAdd( 1, 2);
    }

    CTestClass objTestClass;
    objTestClass.SetChar( 123);
    return 0;
}
```

[bl](#) 命令用于查看已存在的断点:

```
0:000> bl
```

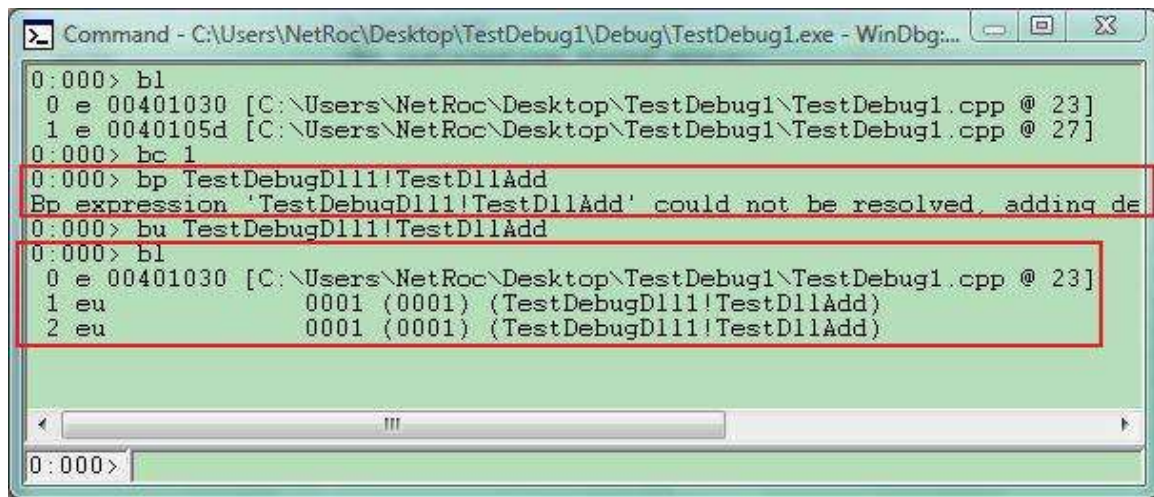
```
0 e 00401030 [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 23] 0001
(0001) 0:**** TestDebug1!main
```

```
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27]
```

如上面命令输出中的第二行, 1 表示断点 ID。当使用 [bd](#) 命令禁用断点、[be](#) 命令重新启用断点或者其他命令来操作这个断点时, 都需要用到这个 ID; 第二个“e”表示断点是启用的, 如果是“d”则表示当前被禁用, 如果带“u”则说明是后面将要介绍的未定断点; 第三列的 **0040105d** 是该断点的地址; 后面的内容是断点所在的源文件和行号。

有时候我们想要设置断点的模块还没有被加载到内存中, 如这个例子中的 TestDebugDll1.dll, 只有在调用了 LoadLibrary 之后才会加载进来。如果使用 [bp](#) 来对这个模块中的函数设置断点, 会找不到符号, 这时就会被调试器自动转变成用 [bu](#) 命令来设置的未定断点。[bu](#) 可以对还不能识别的符号设置断点, 当系统中有新模块加载进来时, 调试器会

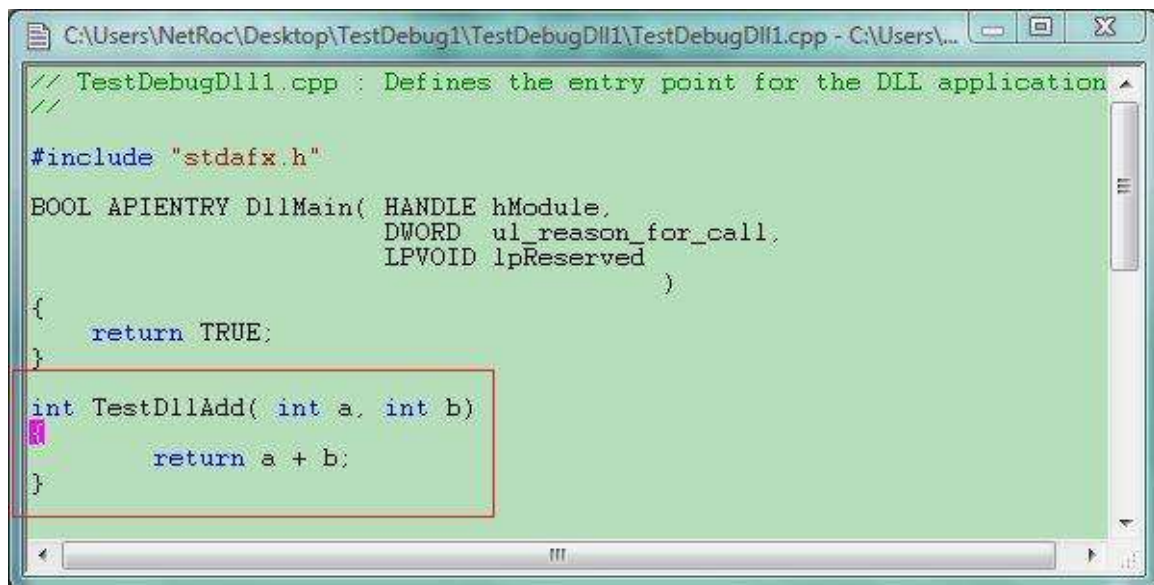
对未定断点再次进行识别，如果找到了匹配的符号则会设置它。现在我们首先用 `bc` 命令删除上面的 1 号断点，然后用 `bu TestDebugDll1!TestDllAdd` 命令对 TestDebugDll1.exe 中的 TestDllAdd 函数设置未定断点，结果如下：



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg...
0:000> bl
0 e 00401030 [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 23]
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27]
0:000> bc 1
0:000> bp TestDebugDll1!TestDllAdd
Bp expression 'TestDebugDll1!TestDllAdd' could not be resolved, adding de
0:000> bu TestDebugDll1!TestDllAdd
0:000> bl
0 e 00401030 [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 23]
1 eu          0001 (0001) (TestDebugDll1!TestDllAdd)
2 eu          0001 (0001) (TestDebugDll1!TestDllAdd)
```

第一个 `bl` 命令可以看到我们之前设置的两个断点，然后 `bc` 命令将 1 号断点删除。接下来使用了一次 `bp` 命令，系统提示找不到 TestDebugDll1!TestDllAdd，将断点自动转换成未定断点。第三次，使用 `bu` 命令对 TestDebugDll1!TestDllAdd 成功设置了未定断点。最后查看存在的断点有三个。0 号是最开始的断点，1 号是 `bp` 命令失败后 WinDbg 自动转换的断点，2 号是 `bu` 命令设置的。

接下来的程序会加载 TestDebugDll1.dll 并调用 TestDllAdd 函数，我们 F5 继续：



```
C:\Users\NetRoc\Desktop\TestDebug1\TestDebugDll1\TestDebugDll1.cpp - C:\Users\...
// TestDebugDll1.cpp : Defines the entry point for the DLL application
//

#include "stdafx.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    return TRUE;
}

int TestDllAdd( int a, int b)
{
    return a + b;
}
```

调试器自动打开了 TestDebugDll1.dll 的源文件，并且发现中断在 TestDllAdd 函数开头。

下面我们再试验一下对类成员函数下断和内存访问断点。继续上面的调试会话，源码中有一个类成员函数 CTestClass::SetChar()，可以直接使用符号对它设置断点。下面几条命令等效：

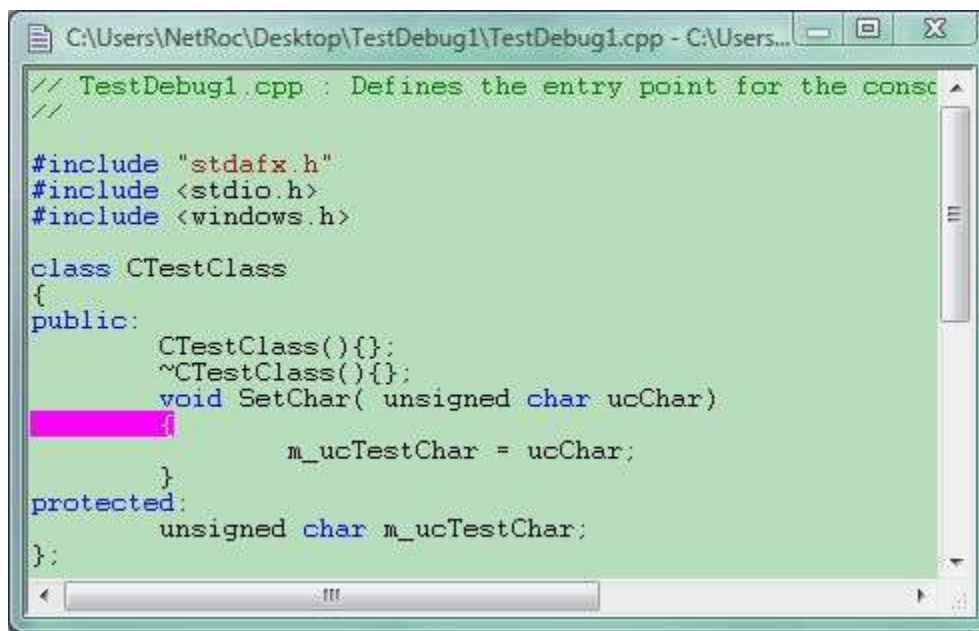
```
bp TestDebug1.exe!CTestClass::SetChar
```

```
bp TestDebug1.exe!CTestClass__SetChar
```

```
bp @@C++(TestDebug1.exe!CTestClass::SetChar)
```

Windows 调试工具支持两种语法的表达式：**MASM** 语法和 **C++**语法。如果没有特别指明的话，默认是使用 **MASM** 表达式语法。一般来说，**MASM** 语法的表达式用来表示地址比较方便，而 **C++**表达式用来表示结构或者类成员比较方便。可以通过**@@C++(...)**或者**@@masm(...)**来包含表达式以明确指明所使用的语法。当使用 **MASM** 语法时，可以用双冒号(:)或者双下划线(\_\_)来表示类成员；但是使用 **C++**语法时则只能使用双冒号。

用上面的命令之一对 **CTestClass::SetChar** 设置断点并 **F5** 运行，可以看到成功中断到了 **CTestClass::SetChar** 函数处。



```
C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp - C:\Users...
// TestDebug1.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>

class CTestClass
{
public:
    CTestClass(){};
    ~CTestClass(){};
    void SetChar( unsigned char ucChar)
    {
        m_ucTestChar = ucChar;
    }
protected:
    unsigned char m_ucTestChar;
};
```

**ba** 命令用于设置访问断点。访问断点可以在某个内存地址处的数据被读取、写入或者执行的时候中断下来。首先用 **.restart** 命令重新启动调试目标，并且用前面的方法之一中断到源代码中 **HMODULE hMod = LoadLibraryA( "TestDebugDll1.dll")**;这一行处。我们看到后面的代码对局部变量 **i** 有赋值操作。我们继续试着使用 **C++**语法来使用命令，输入 **ba w4 @@C++(&i)** 命令。“**&i**”在 **C++**语法中表示变量 **i** 的地址，“**w**”表示写入操作，“**4**”表示只处理 **&i** 地址处 4 字节的写入操作。**F5** 运行，程序被成功中断下来：



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.10...
0:000> bl
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27] 00
0:000> ba w4 @@C++(&i)
0:000> g
ModLoad: 10000000 10038000 C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDe
Breakpoint 0 hit
eax=00000003 ebx=7ffda000 ecx=00000000 edx=00000000 esi=0012fedc edi=0012ff3
eip=004010ab esp=0012fedc ebp=0012ff48 iopl=0         nv up ei pl zr na pe n
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0000024
TestDebug1!main+0x7b:
004010ab 8d4de4          lea     ecx,[ebp-1Ch]
0:000> bl
0 e 0012ff38 w 4 0001 (0001) 0:****
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27] 00
```

输出中有几处值得注意的地方。第一个 [bl](#) 可以看到，之前已经存在了一个 ID 为 1 的断点，然后我们又使用 [ba](#) 设置了一个断点。在第二次 [bl](#) 输出中可以看见新加的断点 ID 为 0、“w”表示是一个写断点、“4”表示写入的数据长度、要监控的内存地址为“0012ff38”。G 命令之后，0 号断点被触发，也就是刚才设置的数据断点。但是下面显示的当前指令却没有访问到我们设置断点的 0x0012ff38。这里又涉及到 WinDbg 数据断点实现的原理。来通过 VC 的窗口看一看相关代码和对应的汇编代码：

```
0040108C 8B 45 E8        mov     dword ptr [ebp-18h],eax
29:             if ( TestDllAdd)
0040108F 83 7D E8 00     cmp     dword ptr [ebp-18h],0
00401093 74 16          je      main+7Bh (004010ab)
30:             {
31:             i = TestDllAdd( 1, 2);
00401095 8B F4          mov     esi,esp
00401097 6A 02          push    2
00401099 6A 01          push    1          调用TestDllAdd
0040109B FF 55 E8       call    dword ptr [ebp-18h]
0040109E 83 C4 08       add     esp,8
004010A1 3B F4          cmp     esi,esp
004010A3 E8 08 06 00 00 call    chkesp (004016b0) 对i赋值
004010A8 8B 45 F0       mov     dword ptr [ebp-10h],eax
32:             }
33:
34:
35:             CTestClass objTestClass;
004010AB 8D 4D E4       lea     ecx,[ebp-1Ch] 中断在这里
004010AE E8 57 FF FF FF call    @ILT+5(CTestClass::CTestClass)
```

图中的 `mov dword ptr [ebp-10h],eax` 才是对 `i` 赋值。但是断点触发后却中断到了赋值之后的下一条指令。

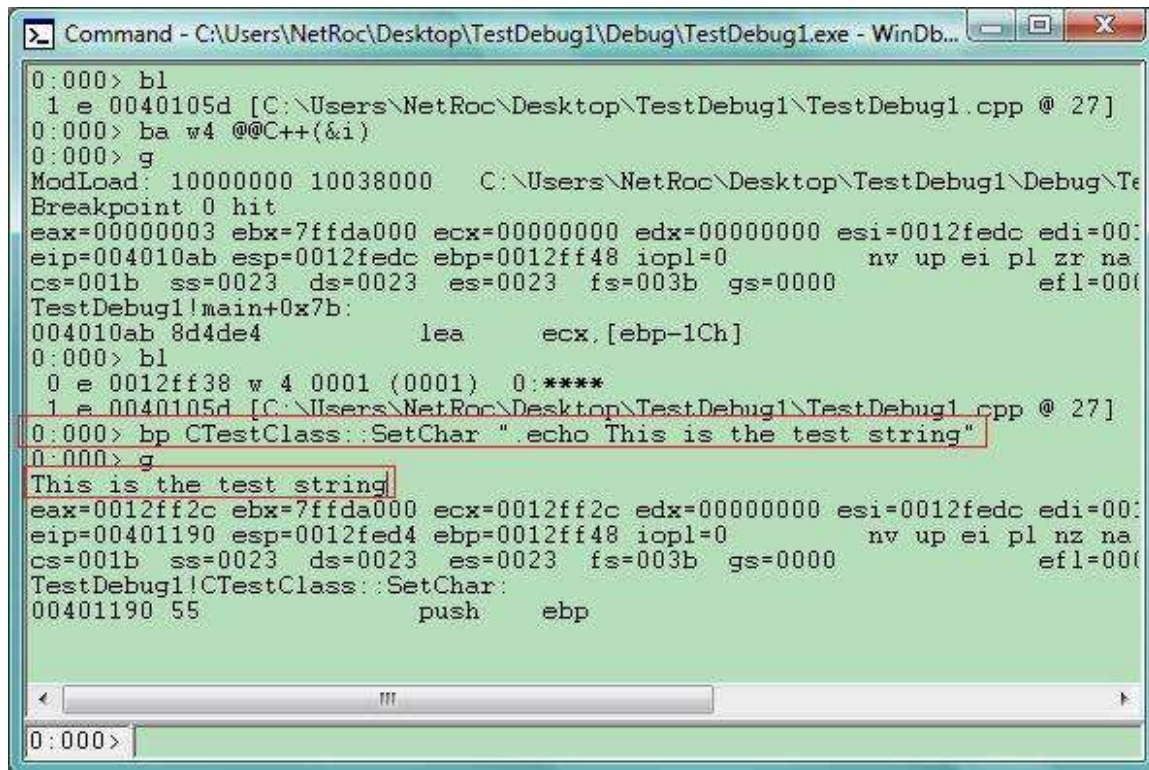
WinDbg 的数据断点是通过 CPU 硬件断点实现的。而 `DRx` 寄存器所设置的内存访问断点属于陷阱(Trap)而不是错误(Fault)，CPU 对陷阱的处理是执行完该条指令后触发异常。因此 WinDbg 只能在之后的一条指令处断下来。

`ba` 命令支持的断点种类有以下几个：

选项	行为
<b>e</b> (执行)	当 CPU 取指定地址的指令时中断到调试器。
<b>r</b> (读/写)	当 CPU 读写指定地址时中断到调试器。
<b>w</b> (写)	当 CPU 写指定地址时中断到调试器。
<b>i</b> (i/o)	(Microsoft Windows XP 和之后版本、仅内核模式、仅 x86 系统)当指定 Address 的 I/O 端口被访问时中断到调试器。

`e` 选项所指定的数据长度必须是 1，即只能指定 `e1`。`r/w` 选项支持 1、2、4 的数据长度，在 X64 机器上可以支持 8。

断点命令中可以设置一条或多条命令，当断点被触发时会自动执行它。接着上面的调试会话，使用下面的命令：



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg
0:000> bl
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27]
0:000> ba w4 @@C++(&i)
0:000> g
ModLoad: 10000000 10038000 C:\Users\NetRoc\Desktop\TestDebug1\Debug\Te
Breakpoint 0 hit
eax=00000003 ebx=7ffda000 ecx=00000000 edx=00000000 esi=0012fedc edi=00:
eip=004010ab esp=0012fedc ebp=0012ff48 iopl=0         nv up ei pl zr na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=000
TestDebug1!main+0x7b:
004010ab 8d4de4          lea     ecx,[ebp-1Ch]
0:000> bl
0 e 0012ff38 w 4 0001 (0001) 0:****
1 e 0040105d [C:\Users\NetRoc\Desktop\TestDebug1\TestDebug1.cpp @ 27]
0:000> bp CTestClass::SetChar ".echo This is the test string"
0:000> g
This is the test string
eax=0012ff2c ebx=7ffda000 ecx=0012ff2c edx=00000000 esi=0012fedc edi=00:
eip=00401190 esp=0012fed4 ebp=0012ff48 iopl=0         nv up ei pl nz na
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=000
TestDebug1!CTestClass::SetChar:
00401190 55             push    ebp
```



这里使用了 `bp CTestClass::SetChar ".echo This is the test string"` 命令。`.echo` 是调试器命令的关键字，用于向调试器命令窗口输出一串字符串。这个命令的结果就是，在 `CTestClass::SetChar` 成员函数设置断点，并且在中断的时候执行 `.echo This is the test string` 命令。可以看到，`g` 命令重新运行程序之后，断点触发时调试器命令窗口中出现了这个字符串。

WinDbg 的条件断点也是采用这种方式的。通过“命令的命令”配合 `.if` 这样的命令关键字，就可以实现灵活多样的条件断点。

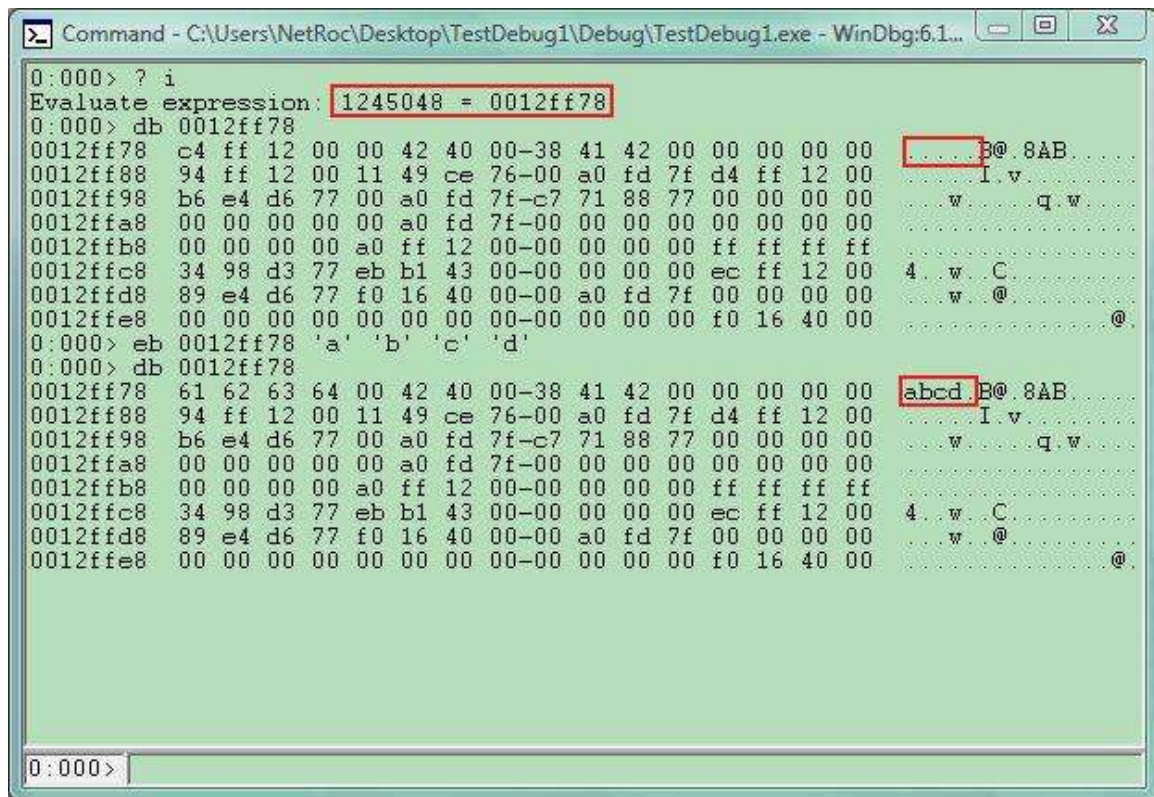
#### 四、访问内存和寄存器

WinDbg 可以通过命令或者 GUI 界面来访问内存和寄存器。常用的几条命令如下：

- 以 `d` 开头的 `d*` 系列命令用于查看内存值。命令的第二个字符用于指定按何种数据类型查看该内存中的数据，如 `db` 是按 `BYTE` 类型查看，`dd` 是按 `DWORD` 类型查看。

重新中断到 `TestDebug1.exe` 的 `main` 函数处。用 `db 400000` 命令查看 PE 文件头的内容，在右边会自动列出对应的 ASCII 字符。直接使用 `d` 命令会按照上一次 `d*` 命令的方式来查看。如果不带地址参数，则从上一次显示结束的地方继续显示。

- `?表达式求值` 命令常常用来查看符号所代表的值。
- `e*` 命令可以将值写入内存。命令第二个字符的定义和 `d*` 一样，用于指定数据类型。可以用一条命令按照顺序向指定地址写入多个值。



```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.1...
0:000> ? i
Evaluate expression: 1245048 = 0012ff78
0:000> db 0012ff78
0012ff78 c4 ff 12 00 00 42 40 00-38 41 42 00 00 00 00 00 .....B@.8AB.....
0012ff88 94 ff 12 00 11 49 ce 76-00 a0 fd 7f d4 ff 12 00 .....I.v.....
0012ff98 b6 e4 d6 77 00 a0 fd 7f-c7 71 88 77 00 00 00 00 .....w.....q.w....
0012ffa8 00 00 00 00 00 a0 fd 7f-00 00 00 00 00 00 00 00 .....
0012ffb8 00 00 00 00 a0 ff 12 00-00 00 00 00 ff ff ff ff .....
0012ffc8 34 98 d3 77 eb b1 43 00-00 00 00 00 ec ff 12 00 4..w..C.....
0012ffd8 89 e4 d6 77 f0 16 40 00-00 a0 fd 7f 00 00 00 00 .....w..@.....
0012ffe8 00 00 00 00 00 00 00 00-00 00 00 00 f0 16 40 00 .....@.
0:000> eb 0012ff78 'a' 'b' 'c' 'd'
0:000> db 0012ff78
0012ff78 61 62 63 64 00 42 40 00-38 41 42 00 00 00 00 00 .....abcd.B@.8AB.....
0012ff88 94 ff 12 00 11 49 ce 76-00 a0 fd 7f d4 ff 12 00 .....I.v.....
0012ff98 b6 e4 d6 77 00 a0 fd 7f-c7 71 88 77 00 00 00 00 .....w.....q.w....
0012ffa8 00 00 00 00 00 a0 fd 7f-00 00 00 00 00 00 00 00 .....
0012ffb8 00 00 00 00 a0 ff 12 00-00 00 00 00 ff ff ff ff .....
0012ffc8 34 98 d3 77 eb b1 43 00-00 00 00 00 ec ff 12 00 4..w..C.....
0012ffd8 89 e4 d6 77 f0 16 40 00-00 a0 fd 7f 00 00 00 00 .....w..@.....
0012ffe8 00 00 00 00 00 00 00 00-00 00 00 00 f0 16 40 00 .....@.
```

首先使用 `?i` 命令，它可以显示符号 `i` 对应的值，即局部变量 `i` 的地址。命令输出的等号两边分别是 10 进制数字和 16 进制数字。然后使用 `db 0012ff78` 查看变量 `i` 处的内存内容，目前的值是 `0x0012ffc4`。`eb 0012ff78 'a' 'b' 'c' 'd'` 命令会在从 `0012ff78` 开始的地址处依次写入后面的数值，命令执行时 WinDbg 会像 C/C++ 一样自动将单引号中的 ASCII 字符转换为数字。最后，再通过 `db` 命令查看内存，可以看到刚才的“abcd”已经写入了。

- `r` 命令用于查看或者修改寄存器和伪寄存器。Windows 调试工具定义了一些伪寄存器，他们不是机器上实际的寄存器，而是根据调试环境不同自动变化的值。详细可以查看帮助文档中的[伪寄存器语法](#)。
- `dt` 命令用于查看结构。参考下面的命令序列：

```
Command - C:\Users\NetRoc\Desktop\TestDebug1\Debug\TestDebug1.exe - WinDbg:6.1...
0:000> .symfix+ d:\symbols
0:000> .reload
Reloading current modules
*** WARNING: Unable to verify checksum for TestDebug1.exe
0:000> ?$peb
Evaluate expression: 2147328000 = 7ffda000
0:000> r $peb
$peb=7ffda000
0:000> dt nt!_PEB 7ffda000
ntdll!_PEB
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged : 0x1 ''
+0x003 BitField : 0 ''
+0x003 ImageUsesLargePages : 0y0
+0x003 IsProtectedProcess : 0y0
+0x003 IsLegacyProcess : 0y0
+0x003 IsImageDynamicallyRelocated : 0y0
+0x003 SkipPatchingUser32Forwarders : 0y0
+0x003 SpareBits : 0y000
+0x004 Mutant : 0xffffffff
+0x008 ImageBaseAddress : 0x00400000
+0x00c Ldr : 0x77df4cc0 _PEB_LDR_DATA
+0x010 ProcessParameters : 0x002c14f0 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : (null)
+0x018 ProcessHeap : 0x002c0000
+0x01c FastPebLock : 0x77df4460 _RTL_CRITICAL_SECTION
+0x020 AtlThunkSListPtr : (null)
+0x024 IFEOKey : (null)
+0x028 CrossProcessFlags : 1
+0x028 ProcessId : 0
+0x028 ProcessType : 0
```

首先用上一篇中介绍过的 [.symfix](#) 和 [.reload](#) 命令加载 Windows 符号, `$peb` 是一个伪寄存器, 调试器将它定义为当前进程的进程环境块地址。使用 [?](#) 或者 [r](#) 命令都能看到它的内容。进程环境块是一个 `nt!_PEB` 结构, 所以可以用 [dt](#) 来显示出当前进程的 PEB 内容。

- [!address](#) 扩展命令可以显示指定的内存地址的信息。接着上面的调试会话, 对 PE 文件头使用 `!address` 看看:

```
0:000> !address 400000
```

```
ProcessParameters 002c14f0 in range 002c0000 002c4000
```

```
Environment 002c0808 in range 002c0000 002c4000
```

```
00400000 : 00400000 - 00001000
```

```
Type      01000000 MEM_IMAGE
```

```
Protect   00000002 PAGE_READONLY
```

```
State     00001000 MEM_COMMIT
```

```
Usage     RegionUsagelImage
```



FullPath TestDebug1.exe

这里可以看到指定的地址 0x400000 的内存类型、保护属性、拥有该地址的模块等等。

- [dv](#) 命令可以查看当前作用域下局部变量的类型和值：

```
0:000> dv
```

```
argc = 2147328000
```

```
Type information missing error for argv
```

```
Type information missing error for objTestClass
```

```
i = 1684234849
```

```
Type information missing error for TestDllAdd
```

```
Type information missing error for hMod
```

main 函数有些局部变量没有类型信息，这是因为 VC6 中默认的 Debug 选项编译出来之后，.pdb 文件中符号信息并不完全。

## 五、 查看调用堆栈

WinDbg 中查看调用堆栈也可以通过 GUI 界面和命令两种方式。详细信息可以点击[这里](#)查看 WinDbg 帮助文档中的内容。