ETW提高

刘振东

2015/6/2

本PPT涉及到的技术名词

Event trace For Windows

非托管代码

不安全代码

PerfView.exe

用户模式

内核模式

内容提纲

- 托管代码与非托管代码介绍
- 不安全代码介绍
- 用户模式与内核模式
- ETW执行流程分析
- 日志分析工具介绍：PerfView.exe

ETW与非托管代码

- ETW依赖的SourceEvent和TraceEvent的类库中有很多非托管代码。
- 而SourceEvent和TraceEvent类库又依赖最底层的非托管的advapi32.dll来完成实际工作。
- advapi32.dll 全称是：Advanced Windows 32 Base API DLL，它是一个高级API应用程序接口服务库的一部分，包含的函数与对象的安全性，注册表的操控以及事件日志有关。此文件大小是659KB，一般位于C:\WINDOWS\system32\目录下。
- 因此，为了更好的理解ETW，就需要了解非托管代码。

托管代码&非托管代码

- **托管代码 (managed code)**

由公共语言运行库环境（而不是直接由操作系统）执行的代码。托管代码应用程序可以获得公共语言运行库服务，例如自动垃圾回收、运行库类型检查和安全支持等。这些服务帮助提供独立于平台和语言的、统一的托管代码应用程序行为。

托管代码是可以使用20多种支持Microsoft .NET Framework的高级语言编写的代码，它们包括：C#, J#, Microsoft Visual Basic .NET, Microsoft JScript .NET, 以及C++。所有的语言共享统一的类库集合，并能被编码成为中间语言(IL)。运行库编译器（runtime-aware ompiler）在托管执行环境下编译中间语言（IL）使之成为本地可执行的代码，并使用数组边界和索引检查，异常处理，垃圾回收等手段确保类型的安全。

在托管执行环境中使用托管代码及其编译，可以避免许多典型的导致安全黑洞和不稳定程序的编程错误。同样，许多不可靠的设计也自动的被增强了安全性，例如 类型安全检查，内存管理和释放无效对象。程序员可以花更多的精力关注程序的应用逻辑设计并可以减少代码的编写量。这就意味着更短的开发时间和更健壮的程序。

- **非托管代码 (unmanaged code)**

在公共语言运行库环境的外部，由操作系统直接执行的代码。非托管代码必须提供自己的垃圾回收、类型检查、安全支持等服务；它与托管代码不同，后者从公共语言运行库中获得这些服务。

托管代码与非托管代码的性能比较

- 众所周知，所有.Net语言都将被编译成为一个叫做IL汇编的中间语言。但是计算机是如何执行这个中间代码的，却是很多人不知道，甚至理解错误了。JIT是.NET程序运行的重要部件之一，全称是即时编译器。很多人都以为JIT其实就是跟Java VM差不多的东西，是一个Interpreter，在运行时读取IL汇编代码，然后模拟成x86代码（也就是俗称的虚拟机）。但是事实上，.NET使用的是更为高级的技术。 .Net程序被加载入内存以后，当某段IL代码被第一次运行的时候，JIT编译器就会将这段IL代码，全部编译成本地代码，然后再执行。这也就是为什么.NET程序第一次运行都启动很慢的原因！

- 随.NET库，微软还附带了一个工具，可以事先将.NET程序所有的IL代码都编译成本地代码并保存在缓存区中，这样一来，这个程序就跟c++编译的一模一样了，没有任何区别，运行时也可以脱离JIT了（这里不要混淆了，这里不是说可以脱离.NET库，而是说不需要在进行即时编译这个过程了）。所以，请不要将.NET和Java混为一谈，两个的运行效率根本不是一个等级的！

- JIT的优化指的是可以针对本地CPU，在编译时进行优化。传统程序在编译时，为了保证兼容性，通常使用最通用的指令集（比如古老的386指令集）来编译。而JIT知道CPU的具体类型，可以充分利用这些附加指令集进行编译，这样的性能提升是很可观的。

P/Invoke是什么？有何作用？

- 问题：C#有没有方法可以直接都用这些原本已经存在的功能(比如Windows中的一些功能，C++中已经编写好的一些方法)？答案是肯定的，可以通过P/Invoke的方式直接调用这些功能。

- P/Invoke = Platform Invoke，平台调用服务。

- 平台调用是CLR(公共语言运行时)提供的一种服务。平台调用服务 (PInvoke) 允许托管代码调用在 DLL 中实现的非托管函数。

- 例如：可以通过P/Invoke使得C#程序来调用非托管的win32 API 。

C# 代码两种调用非托管代码的方法

C# 代码有以下两种可以直接调用非托管代码的方法。

- 方法一：直接调用从 DLL 导出的函数。
- 方法二：调用 COM 对象上的接口方法（更多信息，请参见 COM Interop教程）。

对于这两种技术，都必须向 C# 编译器提供非托管函数的声明，并且还可能需要向 C# 编译器提供如何封送与非托管代码之间传递的参数和返回值的说明。

方法一：P/Invoke调用win32 API Demo

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace ConsoleAppUnsafe
{
    class Program
    {
        [DllImport("kernel32.dll", EntryPoint = "MoveFile",
          ExactSpelling = false,
          CharSet = CharSet.Unicode,
          SetLastError = true)]
        static extern bool MoveFile(string source, string dest);

        static void Main(string[] args)
        {
            string source = @"d:\Temp\20150528_041232_659.etl";
            string dest = @"d:\20150528_041232_659.etl";
            if (Program.MoveFile(source, dest) == true)
            {
                Console.WriteLine("Move File Ok!");
            }
            else
            {
                Console.WriteLine("Move File Error!");
            }

            Console.Read();
        }
    }
}
```

C#中extern 是什么意思？

- extern 修饰符用于声明在外部实现的方法。
- extern 修饰符的常见用法是在使用 Interop 服务调入非托管代码时与 DllImport 属性一起使用。
- 在这种情况下，还必须将方法声明为 static。
- 例如：
  DllImport("avifil32.dll")]
  private static extern void AVIFileInit();
  也就是说这个方法是放在申明的类之外的类中实现 的.

C#中DllImport 属性是什么意思？

- MSDN中对DllImportAttribute的解释是这样的可将该属性应用于方法。DllImportAttribute 属性提供对从非托管 DLL 导出的函数进行调用所必需的信息。作为最低要求，必须提供包含入口点的 DLL 的名称。
- DllImport是System.Runtime.InteropServices命名空间下的一个属性类，其功能是提供从非托管DLL导出的函数的必要调用信息。
- DllImport属性应用于方法，要求最少要提供包含入口点的dll的名称。

内容提纲

- 托管代码与非托管代码介绍
- 不安全代码介绍
- 用户模式与内核模式
- ETW执行流程分析
- 日志分析工具介绍：PerfView.exe

什么是不安全代码unsafe？

- 在公共语言运行库 (CLR) 中，不安全代码是指无法验证的代码。C# 中的不安全代码不一定是危险的，只是其安全性无法由 CLR 进行验证的代码。因此，CLR 只对在完全受信任的程序集中的不安全代码执行操作。如果使用不安全代码，由您负责确保您的代码不会引起安全风险或指针错误。

- MSDN：**unsafe** 关键字表示不安全上下文，该上下文是任何涉及指针的操作所必需的。

- 在 C# 中使用不安全代码unsafe，指的就是使用指针的代码。

- 指针可以看做是对内存的直接操作，一旦没用好就会出现问题，导致其他的软件部能运行，甚至系统崩溃，c#将指针定义为不安全代码，是为了防止这些问题出现，你可以理解为使用不安全代码是告诉系统要“小心”。

为何要有unsafe？

- 为了实现CLR类型安全的目标，默认情况下，C#没有提供指针的使用算法。
- 但是有时候程序员非常清楚程序的运行状况，需要使用指针直接访问内存以便于提高性能或者调试、监控程序运行的内存的使用状况，以便于采取相应的措施。
- 在 C# 中很少需要使用指针，但仍有一些需要使用的情况。例如，在下列情况中使用允许采用指针的不安全上下文是正确的：
  1.处理磁盘上的现有结构。
  2.涉及内部包含指针结构的高级 COM 或平台调用方案。
  3.性能关键代码。
- 使用不安全代码的情况有：
  1.使用指针的不安全代码。
  2.方法、类型和可被定义为不安全的代码块。
  3.在某些情况下，通过移除数组界限检查，不安全代码可提高应用程序的性能。

c#里面指针为什么是不安全代码？

- 指针可以看做是对内存的直接操作，一旦没用好就会出现问题，导致其他的软件部能运行，甚至系统崩溃，c#将指针定义为不安全代码，是为了防止这些问题出现，你可以理解为使用不安全代码是告诉系统要“小心”。

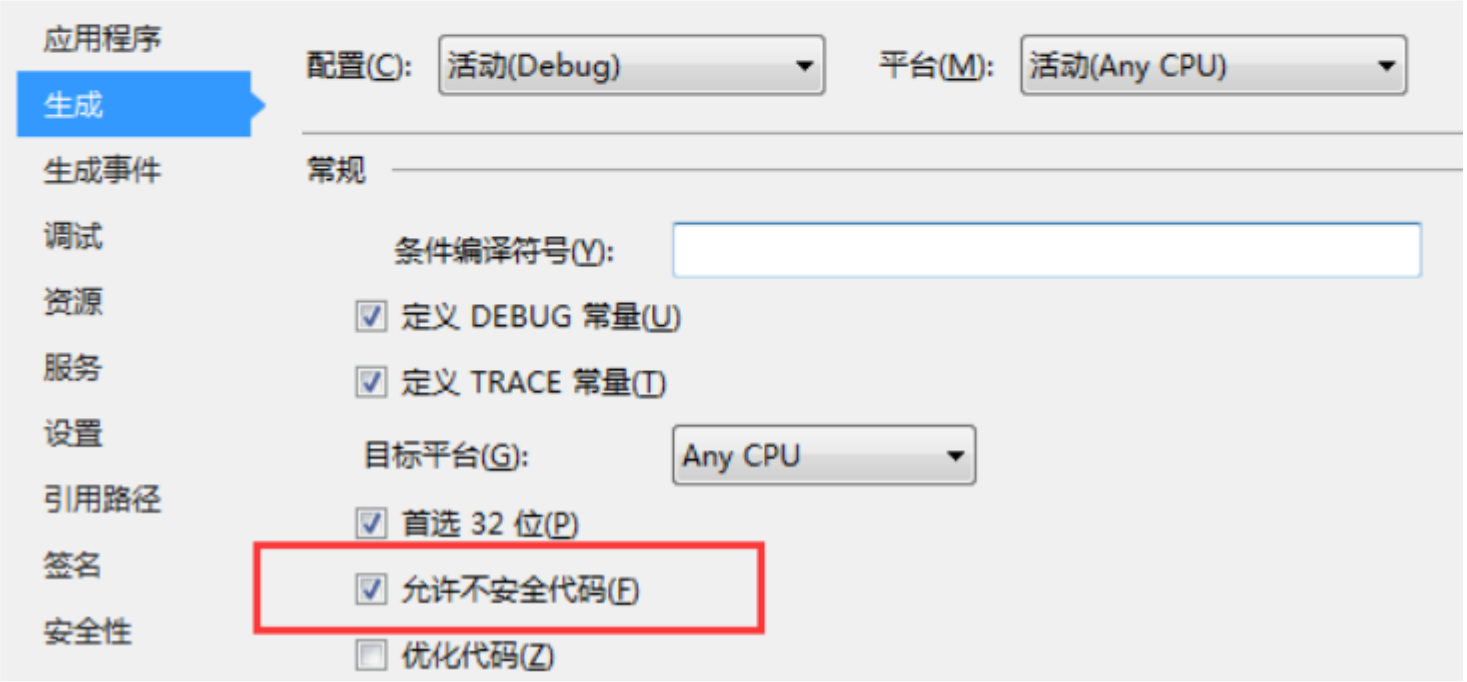
- 指针的使用十分不安全所以Java摒弃了。但也十分方便，所以c#保存下来了，为了解决安全问题就定义为unsafe。

在 VS 中设置打开unsafe编译器选项

在 C# 中，为了编译不安全代码，必须用 unsafe 编译应用程序。

在 Visual Studio 开发环境中设置此编译器选项

1. 打开项目的"属性"页。
2. 单击"生成"属性页。
3. 选中"允许不安全代码"复选框。



C#使用指针的Unsafe代码示例

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleAppUnsafe2
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int i = 99, y = 200;
            Console.WriteLine("i is now {0},y is now {1}", i, y);

            unsafe
            {
                swap(&i, &y);
            }
            Console.WriteLine("i is now {0},y is now {1}", i, y);

            Console.Read();
        }

        1 reference
        public static unsafe void swap(int* a, int* b)
        {
            int temp;
            temp = *a;
            *a = *b;
            *b = temp;
        }

    }
}
```

什么是 IntPtr？

- 什么是IntPtr?

先来看看MSDN上说的:用于表示**指针或句柄**的平台特定类型。IntPtr是托管环境中用来描述非托管环境中指针的类型。

这个其实说出了这样两个事实，IntPtr 可以用来表示指针或句柄、它是一个平台特定类型。

- 对它的解释

It's a class that wraps a pointer that is used when calling Windows API functions. The underlying pointer may be 32 bit or 64 bit, depending on the platform.

- 用在什么地方?

（1）C#调用WIN32 API时

（2）C#调用C/C++写的DLL时（其实和1相同，只是这个一般是我们在和他人合作开发时经常用到）

怎样用 IntPtr？

- 例如有一非托管DLL中的函数原型为：

  MCIERROR mciSendString(LPCTSTR lpszCommand, LPTSTR lpszReturnString, UINT cchReturn, **HANDLE hwndCallback**);

- 那么我们在C#中声明时就要这样写：

  [DllImport("winmm.dll")]
  private static extern long mciSendString(string a,string b,uint c, **IntPtr d**);

- 在调用的时候就可以用这样的方法调用，最后一个参数传入某一控件的Handle ：

  mciSendString("set cdaudio door open", null, 0, **this.Handle**);

回顾梳理不安全代码与非托管代码

- 托管代码是指在CLR运行环境的监控下运行的代码。CLR运行环境将负责处理各种“家务”工作，比如：管理对象的内存，执行类型检查，进行内存垃圾回收。总之，用户不用自己处理上面提到的工作。用户不用自己去直接操作内存，因为CLR运行环境将处理这些问题。

- 非托管代码是指在CLR环境以外运行的代码。这个概念最好的例子就是我们传统的WIN 32 DLL例如Kernel32.dll，user32.dll 和 安装在我们系统上的COM组件。如何为其分配内存空间，如何释放内存，怎么（如果需要）进行类型检测这些工作由自己来做。典型的C++编程中内存分配指针指向也是非托管代码的另一例子，因为你作为程序员需要自己来负责处理这些工作：调用内存分配函数, 确保生成的正确性, 确保当任务结束时候内存被释放。

- 不安全代码是托管代码与非托管代码之间的纽带。

- 不安全代码在CLR托管环境的监管下运行, 就像托管代码那样, 但允许你通过使用指针直接访问内存, 就像非托管代码中的做法那样。这样, 你同时获得了两个世界里最好的东西。你也许要写的程序需要使用传统WIN 32 DLL中的函数, 这些函数又需要使用指针。这个时候，不安全代码就派上用场了。

- 总结：C#可以使用Pinvoke平台调用服务来调用非托管世界中的Win32 DLL的函数，如果这些函数没有指针参数，就不必用不安全代码，如果有指针参数，则需要启用不安全代码。

C#中的fixed是什么

- 在讨论fixed之前，我们先回顾一下托管代码和非托管代码，所谓托管代码就是由CLR去执行的代码而不是操作系统去执行的代码，而非托管代码就是绕过CLR，由操作系统直接执行，它有自己的垃圾回收、类型安全检查等服务。

- 而不安全代码就是允许自己使用指针访问内存，但同时又要使用CLR提供的垃圾回收机制、类型安全检查等服务，有的资料认为是介于CLR和非托管代码之间的一种代码运行机制。

- 正因为如此，我们自定义的指针地址就有可能被CLR垃圾回收机制重新调整位置，所以就引入了fixed ，MSDN对fixed的解释是：fixed 语句设置指向托管变量的指针，并在执行该语句期间"固定"此变量。这样就可以防止变量的重定位。

为什么要引入fixed?

- 我们知道,不安全代码是托管代码,因此将在CLR托管环境的监管下运行.现在,CLR运行环境可以有权利移动内存中的对象.这是一个可以减少内存碎片的原因.但这样的操作,对程序员来说是不知道的,是对程序员透明的,被指针指向的变量的内存可能被重现安排倒另外的内存位置.(这是由CLR完成的)

- 因此, 如果 *pInt指向的变量的原始地址是1001，CLR执行了一些内存重新安排以便减少内存碎片后,该变量的地址之前为1001，在重新进行内存安排后可能存储在内存中地址为2003的位置.这会是一个大灾难，因为指针指向的1001地址什么都没有了,指针变成了无效的.可能这是在.NET下指针使用被弱化的一个原因.你怎么认为呢?

- 这时就需要fixed救场了. 当在一段代码中使用该关键字时,就告诉了CLR不用去动内存中的那些"问题"对象,它就不会去动它们了.这样,当在C#中使用指针时,使用fixed关键字就很好的避免了在运行时指针无效的问题.

- 现在,因为CLR被告知当"固定"代码段执行过程中不允许CLR移动其位置,当"固定"代码段的语句在执行时,指针所指向的变量在内存中的位置将不会改变,内存中的变量将不会被CLR重新部署内存位置.

- 这就是在C#中指针的使用.确保该函数是用unsafe标注的,确保被指向的对象用fixed标注,你也就已经有了在C#中使用指针的能力!

指针的定义方法

- 指针的定义：当在同一个声明中声明多个指针时，* 仅与基础类型一起使用，而不是作为每个指针名称的前缀。例如：
  1.int* p1, p2, p3; // Ok，p1 是指向整数的指针。
  2.int *p1, *p2, *p3; // Invalid in C#。

- 指针间接寻址运算符 * ：可用于访问位于指针变量所指向的位置的内容。例如，对于下面的声明，
  int* myVariable;
  表达式 *myVariable 表示在 myVariable 中包含的地址处找到的 int 变量。

stackalloc

- Stackalloc：在不安全的代码上下文中使用，可以在堆栈上分配内存块。
- int* fib = stackalloc int[100];
- 上面的示例在堆栈而不是堆上分配了一个内存块，它的大小足以包含 100 个 int 类型的元素；该块的地址存储在 fib 指针中。此内存不受垃圾回收的制约，因此不必将其钉住（通过 fixed）。内存块的生存期受定义它的方法的生存期的限制（没有在方法返回之前释放内存的途径）。
- stackalloc 仅在局部变量的初始值设定项中有效。
- 由于涉及指针类型，stackalloc 要求不安全上下文。
- stackalloc 类似于 C 运行时库中的 _alloca。

指针的运算符

- 下表列出可在不安全的上下文中针对指针执行的运算符和语句：

运算符/语句	用途
*	执行指针间接寻址。
->	通过指针访问结构的成员。
[]	对指针建立索引。
&	获取变量的地址。
++ 和 --	递增或递减指针。
加、减	执行指针算法。
==、!=、<、>、<= 和 >=	比较指针。
stackalloc	在堆栈上分配内存。
fixed 语句	临时固定变量以便可以找到其地址。

指针*与&的实际作用

- *a，*b是指针类型
- & 指针变量的内存地址
- * 指针变量实际的变量值

```
        int i = 99, y = 200;
        Console.WriteLine("i is now {0},y is now {1}", i, y);


        unsafe
        {
            swap(&i, &y);
        }
        Console.WriteLine("i is now {0},y is now {1}", i, y);

        Console.Read();
    }

    1 reference
    public static unsafe void swap(int* a, int* b)
    {
        Console.WriteLine("int* a" + Convert.ToString(*a));
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
    }
```

100 %

Watch 1

Name	Value	Type
&y	0x000000001c85dfa4	int*
&i	0x000000001c85dfa0	int*
*a	99	int
*b	200	int

内容提纲

- 托管代码与非托管代码介绍
- 不安全代码介绍
- 用户模式与内核模式
- ETW执行流程分析
- 日志分析工具介绍：PerfView.exe

用户模式和内核模式

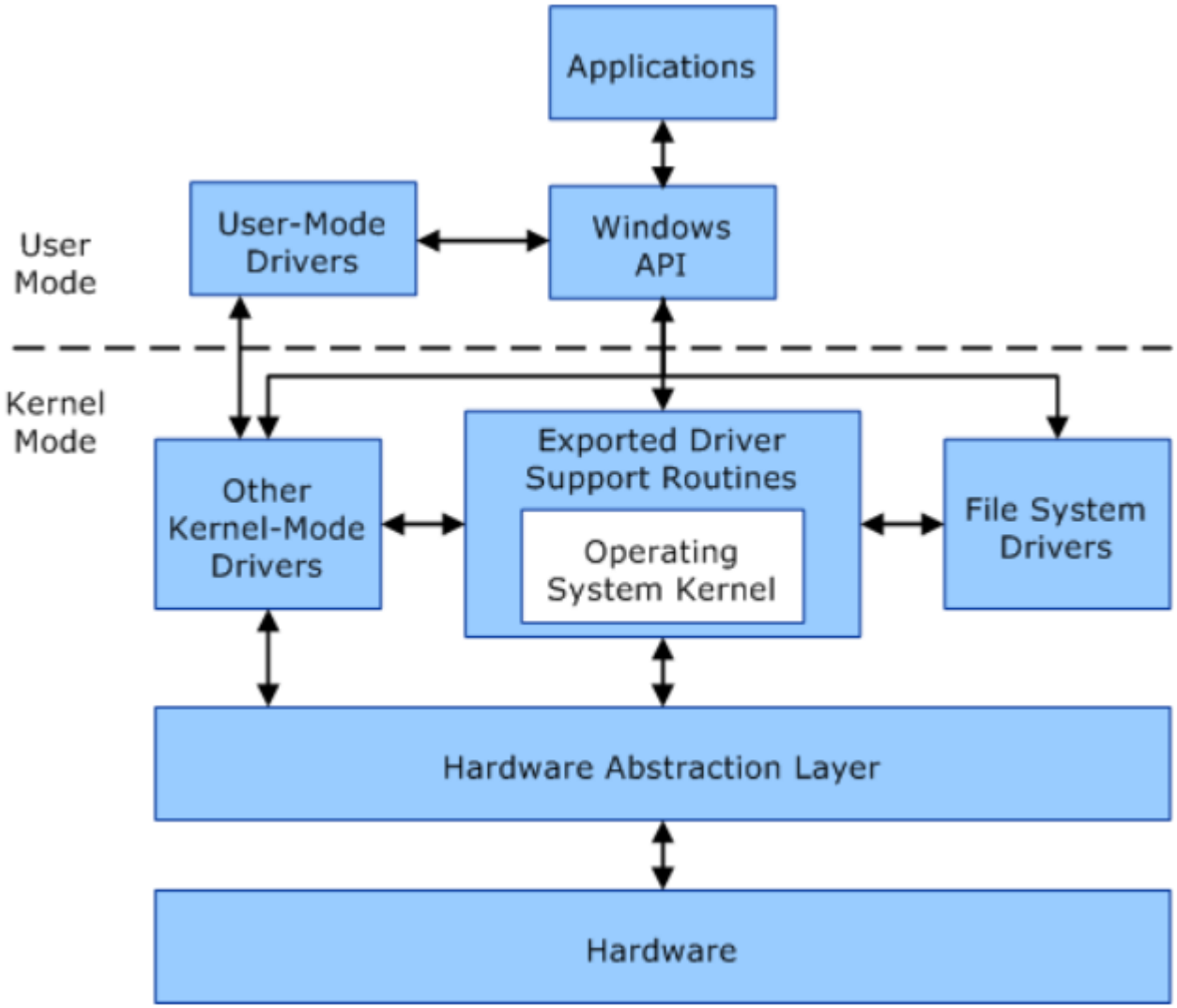
- 运行 Windows 的计算机中的处理器有两个不同模式：“用户模式”和“内核模式”。根据处理器上运行的代码的类型，处理器在两个模式之间切换。应用程序在用户模式下运行，核心操作系统组件在内核模式下运行。多个驱动程序在内核模式下运行，但某些驱动程序在用户模式下运行。

- 当启动用户模式的应用程序时，Windows 会为该应用程序创建“进程”。进程为应用程序提供专用的“虚拟地址空间”和专用的“句柄表格”。由于应用程序的虚拟地址空间为专用空间，一个应用程序无法更改属于其他应用程序的数据。每个应用程序都孤立运行，如果一个应用程序损坏，则损坏会限制到该应用程序。其他应用程序和操作系统不会受该损坏的影响。

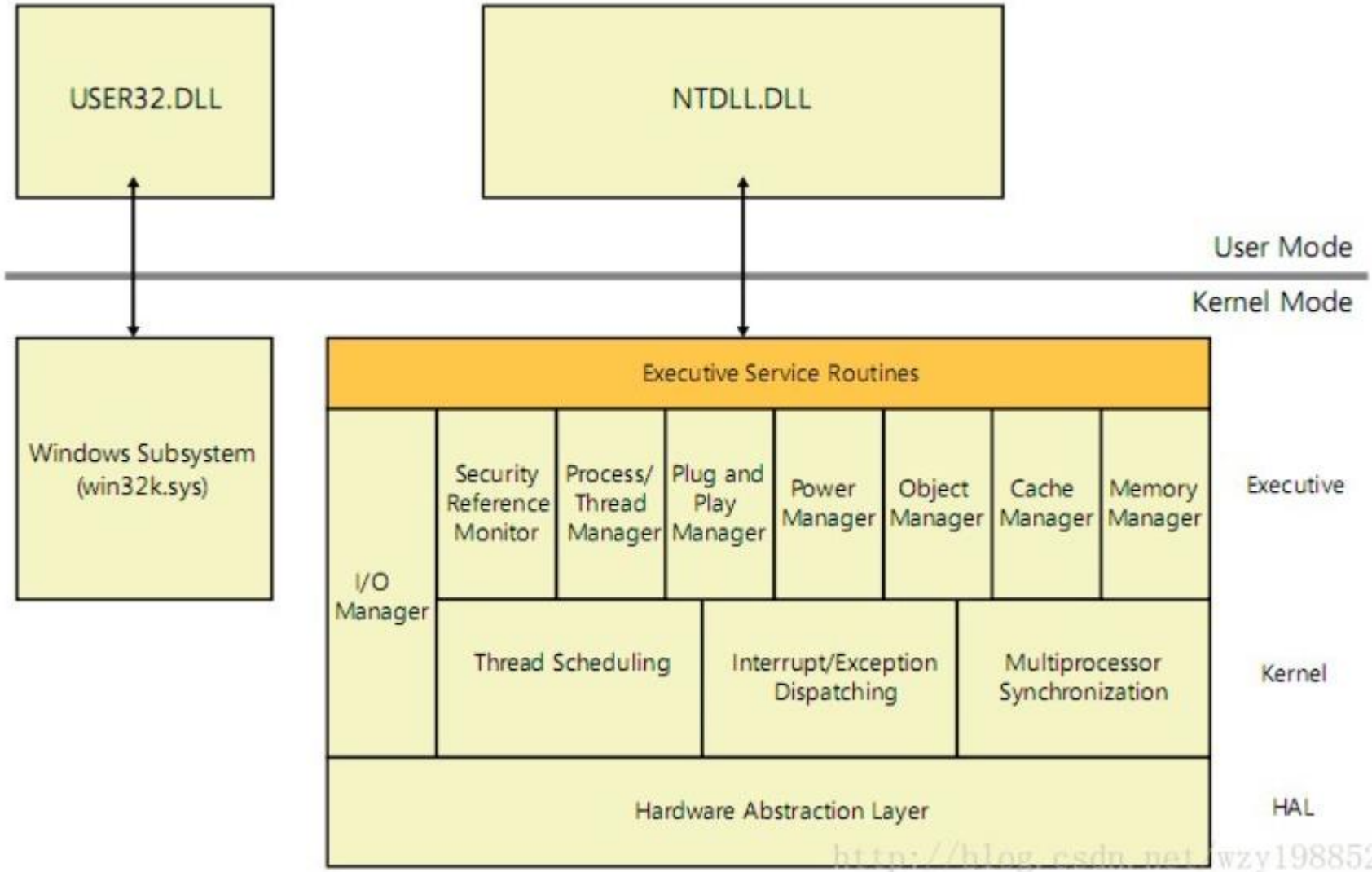
- 用户模式应用程序的虚拟地址空间除了为专用空间以外，还会受到限制。在用户模式下运行的处理器无法访问为该操作系统保留的虚拟地址。限制用户模式应用程序的虚拟地址空间可防止应用程序更改并且可能损坏关键的操作系统数据。

- 在内核模式下运行的所有代码都共享单个虚拟地址空间。这表示内核模式驱动程序未从其他驱动程序和操作系统自身独立开来。如果内核模式驱动程序意外写入错误的虚拟地址，则属于操作系统或其他驱动程序的数据可能会受到损坏。如果内核模式驱动程序损坏，则整个操作系统会损坏。

用户模式组件与内核模式组件之间通信



内核层次架构


USER32.DLL
NTDLL.DLL
User Mode
Kernel Mode
Windows Subsystem
(win32k.sys)
Executive Service Routines
I/O Manager
Security Reference Monitor
Process/ Thread Manager
Plug and Play Manager
Power Manager
Object Manager
Cache Manager
Memory Manager
Executive
Thread Scheduling
Interrupt/Exception Dispatching
Multiprocessor Synchronization
Kernel
Hardware Abstraction Layer
HAL
http://blog.csdn.net/wzy198852

内核的层次划分

- 硬件抽象层(HardwareAbstraction Layer) (HAL) (hal.dll)

最底层隔离硬件的，底层的第三方驱动程序就运行在这层。

- 内核(Kernel)

实现操作系统的一些底层服务，比如线程调度，多处理器的同步，中断/异常处理等。

- 执行体(Executive)（ntoskrnl.exe）

实现基本的操作系统服务，比如基本的线程进程管理，内存管理， IO及进程间通讯等。

- 窗口图形子系统(Windows Graphics Subsystem)

由win32K.sys在内核层实现,用户界面相关都依赖该层,User32.dll的大部分功能都由该层实现。

Windows系统用户层关键系统进程

- Smss.exe(session manager Subsystem)

关于Session的概念可以参考我的这篇Sessions, Window Stationsand Desktops，在操作系统启动时会创建一个不与任何Session关联的Smss.exe管理者实例，然后当有用户登录时它会为每个Sessin拷贝一份与之关联的Smss.exe实例，然后由该关联的Smss.exe实例启动winlogon.exe和csrss.exe.

- WinLogon.exe

该进程管理用户的登录和注销，我们按Ctrl+Alt+Del出现的界面和登录后出现的桌面窗口都是由它启动的。

- Csrss.exe(Client/Server Runtime Subsystem)

我们可以看到我们的桌面窗口(GetDesktopWindow)是由该进程创建的，该进程主要负责Win32子系统的用户模式部分(内核模式部分由win32k.sys实现)。

- Lsass.exe(Local Security Authority Subsystem)

WinLogon.exe通过该进程验证用户登录，登录后产生安全访问令牌对象，通过该令牌创建Explorer.exe,我们其他用户进程都由Explorer.exe启动，并且继承了该令牌权限。

- Services.exe

该进程简称为SCM(NT Service Control Manager),该进程负责启动用户态一些特殊进程，也就是我们通常所说的服务程序。

ETW与用户模式和内核模式

- 我们写的应用程序，例如: WFSite 就是运行在用户模式下。
- ETW 使用内核中实现的缓冲和日志记录机制，提供对用户模式应用程序和内核模式设备驱动程序引发的事件的跟踪机制。
- 小结：ETW用户模式和内核模式通吃，无论哪种模式下的程序，都能用ETW。

内容提纲

- 托管代码与非托管代码介绍
- 不安全代码介绍
- 用户模式与内核模式
- ETW执行流程分析
- 日志分析工具介绍：PerfView.exe

ETW执行流程分析

- 有了前面的知识储备（平台调用Pinvoke，非托管代码，不安全代码，指针等），现在再来分析ETW的底层类库TraceEvent和EventSource就不困难了。

启动ETW Session的流程

- 启动ETW Session 时，会执行TraceEvent\TraceEventSession.cs的InsureStarted()方法，它又调用TraceEventNativeMethods类的StartTraceW()方法。

- TraceEvent\TraceEventNativeMethods.cs的代码如下，它实际上是通过平台调用服务PInvoke调用了advapi32.dll的来吗来完成实际工作的。

```
        [DllImport("advapi32.dll", CharSet = CharSet.Unicode),
SuppressUnmanagedCodeSecurityAttribute]
        internal extern static int StartTraceW(
            [Out] out UInt64 sessionHandle,
            [In] string sessionName,
            EVENT_TRACE_PROPERTIES* properties);
```

停止ETW Session的流程

- 停止ETW Session时，会执行TraceEvent\TraceEventSession.cs的Stop()方法，它又调用TraceEventNativeMethods类的ControlTrace()方法。

- TraceEvent\TraceEventNativeMethods.cs的代码如下，它实际上也是通过平台调用服务PInvoke调用了

```
        [DllImport("advapi32.dll", CharSet = CharSet.Unicode),
SuppressUnmanagedCodeSecurityAttribute]
        internal static extern int ControlTrace(
            ulong sessionHandle,
            string sessionName,
            EVENT_TRACE_PROPERTIES* properties,
            uint controlCode);
```

启用Provider的流程

- 在启动ETWSession之后，还要启用Provider，并与ETWSession关联。
- 启用Provider 时，会执行TraceEvent\TraceEventSession.cs的EnableProvider()方法，该方法又调用了TraceEventNativeMethods.cs的EnableTraceEx2()方法。
- TraceEvent\TraceEventNativeMethods.cs的EnableTraceEx2方法代码如下，它实际上也是通过平台调用服务PInvoke调用了advapi32.dll的来吗来完成实际工作的。

```
[DllImport("advapi32.dll", CharSet = CharSet.Unicode), SuppressUnmanagedCodeSecurityAttribute]
5 references
internal static extern int EnableTraceEx2(
    [In] ulong TraceHandle,
    [In] ref Guid ProviderId,
    [In] uint ControlCode,          // See EVENT_CONTROL_CODE_*
    [In] byte Level,
    [In] ulong MatchAnyKeyword,
    [In] ulong MatchAllKeyword,
    [In] uint Timeout,
    [In] ref ENABLE_TRACE_PARAMETERS EnableParameters);
```

向ETW Session中写入的流程

- 数据提供程序Provider向ETW Session中写入的流程：在自己的程序里创建一个继承自EventSource类的WFEventProvider类，作为数据提供程序，然后就可以调用EventSource类的WriteEvent方法记日志了。

- 流程如下：->Microsoft.Diagnostics.Tracing.EventSource.WriteEvent()，执行WriteEvent方法前，首先首先执行EventSource()构造函数，用来向ETW注册event source provider。

- ->EventProvider.cs类的Register(eventSourceGuid)函数，里面注册了回调函数，然后执行EventProvider.cs类的 EventRegister(ref this.m_providerId, this.m_etwCallback)，加了callback函数。

- ->接着执行UnsafeNativeMethods.ManifestEtw.EventRegister()，最终调用advapi32.dll的EventRegister()函数，如果此时, ETW Session已经启动了，Provider也已启用，则立刻执行回调函数，将this.m_eventSourceEnabled设置为true，如果ETW Session没有启动，则回调函数不执行， 此时this.m_eventSourceEnabled仍为默认值false；

- ->接着返回头执行EventSource.WriteEvent()后面的语句EventSource.WriteEventVarargs()，首先要判断event source provider是否被启用了。如果未启用，就会发现if (this.m_eventSourceEnabled) {} 为false，就不执行了，不记就直接走了；如果为true，就写入日志。

内容提纲

- 托管代码与非托管代码介绍
- 不安全代码介绍
- 用户模式与内核模式
- ETW执行流程分析
- 日志分析工具介绍：PerfView.exe

日志分析工具介绍：PerfView.exe

- **系统性能实时查看(PerfView)**可以看到一些其他的系统硬件检测工具无法查看的数据与信息，实际上是内存和处理器的性能评估和确定RAM和CPU的问题。
- PerfView能够收集Windows事件跟踪（ETW）数据来追踪程序的调用流向，这些程序通过调用哪个函数识别频率。除了配置程序性能数据（Perfmon、PAL和Xperf等工具不能轻松完成），PerfView还能分析程序内存堆来帮助确定内存的运用是否高效。它还有一个Diff功能，可以让你确定跟踪间的任意差别来帮助你认出所有逆行。最后，该工具还有一个Dump功能可以生成一个程序内存转储。

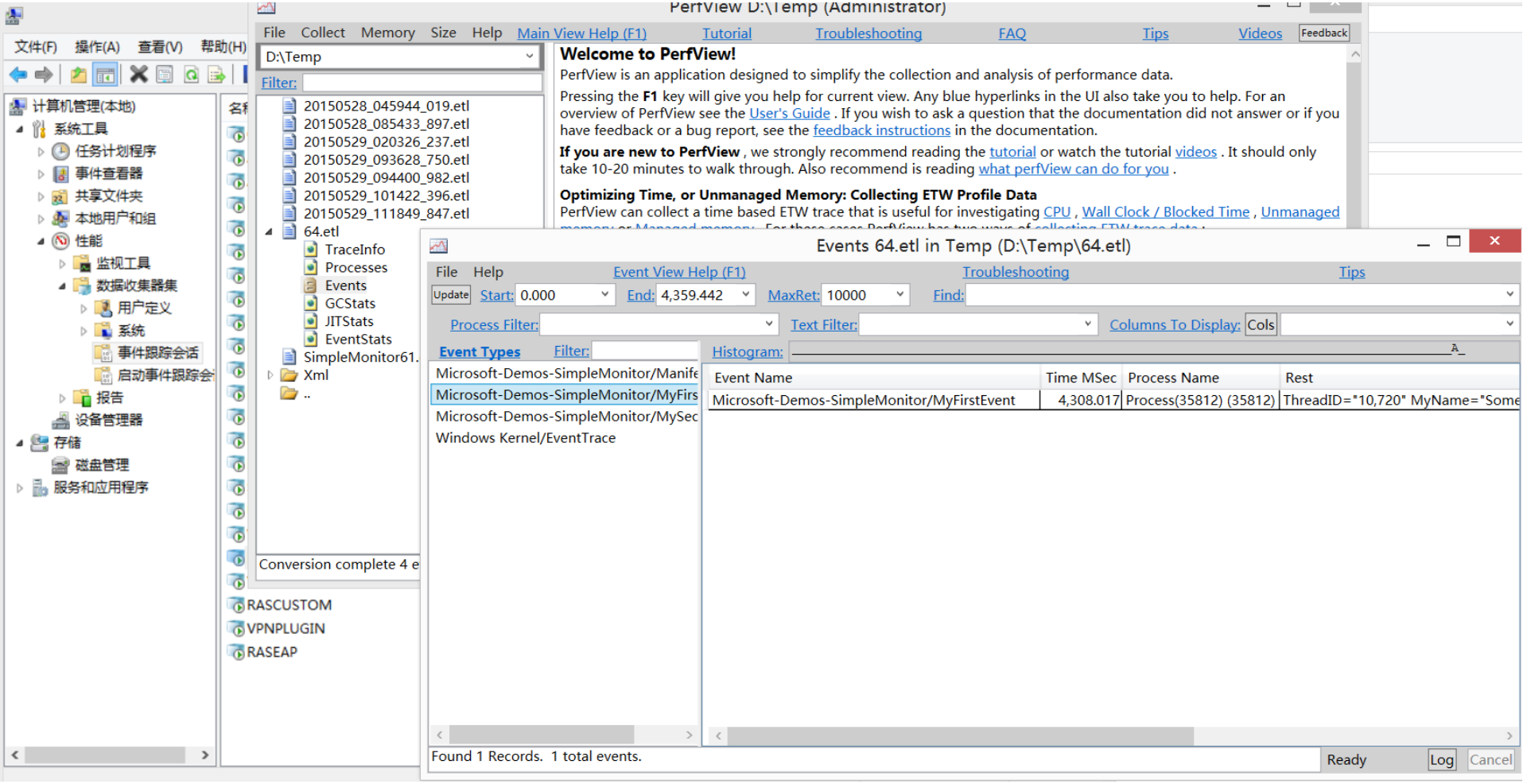
- **安装PerfView：**
- 从微软下载的 PerfView 包括一个zip压缩文件，其中只有一个可执行的文件perfview.exe，这简化了安装。你可以将这个文件复制到多个你想跟踪的服务器上，然后在这些服务器或你本地的工作站中分析数据。PerfView在Windows Vista、Windows 7、Windows Server 2008、Windows Server 2008 R2和Windows Server 2012上都受到支持，要求.NET FX 2.0以上。
- 该软件需要在 .Net 环镜下才能运行，请安装 .net framework V2.0 可再发行组件包: http://www.cr173.com/soft/2572.html

- **收集配置数据：**
- PerfView利用Windows事件追踪，而ETW从Windows 2000 Server以来就一直内置于操作系统中。只是最近才有XPerf和PerfView一类的工具利用ETW数据来解决性能问题。事件数据被收集到一个事件跟踪日志（ETL）中。根据你想要跟踪事件的数量和时间的长度，ETL文件可能会非常大。你可以限制这个日志文件的大小，如果空间受限或者你不知道问题何时发生的话，你还可以让它们循环。默认每毫秒一次的采样间隔在收集时间内产生了大概百分之十的CPU开支。建议大概5000个样本（5秒）用于一次代表性配置采样。
- 开始一次数据收集有两种方式，用运行命令启动一个程序或者用收集命令在计算机范围内收集数据。这些命令可以由收集下拉菜单下的GUI引发，或者从CLI或脚本中执行“PerfView run”或“PerfView collect”命令。下图显示运行命令tutorial.exe时收集数据的过程，tutorial.exe是一个内置的训练练习。

日志分析工具介绍：PerfView.exe

- **查看结果：**
- 一旦你在些之间针对性能问题收集了数据，你可以用PerfView分析ETL文件。该ETL文件会出现在左边的窗口，有收集日志或运行命令期间你提供的名字。通过双击该RTL文件，十来个独立的节点会和指代它们内容的名字一起出现。例如，你会在下图中看到跟踪信息、程序、事件、CPU堆栈。双击各个节点，适当的查看器会打开这些内容。
- 为了针对一个特定程序分析计算密集型性能问题，你将需要学习要调用的堆栈和函数。这可以通过双击左侧窗口中的“CPU堆栈”节点完成。接着你会得到提示来选择你感兴趣的程序。最后，该CPU堆栈查看器会在独立的窗口中打开，如下图QQ进程的信息 ，你可以确定调用了哪个函数以及它们的频率。
- PerfView是一个便于用户的工具，可以用来收集和分析ETW数据用于解决配置程序性能数据的问题。这个工具可以快速地显示为这个程序执行的操作系统函数，了解性能问题可能潜藏的位置。

PerfView.exe读取etl文件的效果图



资源链接

不安全代码：

- https://msdn.microsoft.com/zh-cn/library/aa288474(v=vs.71).aspx
- https://msdn.microsoft.com/zh-cn/library/t2yzs44b(v=VS.80).aspx
- http://www.cnblogs.com/jxnclyk/archive/2010/05/28/1746132.html

指针：

- https://msdn.microsoft.com/zh-cn/library/y31yhkeb(v=vs.80).aspx
- https://msdn.microsoft.com/zh-cn/library/tcy5wf0h(v=vs.80).aspx

用户模式与内核模式：

- https://msdn.microsoft.com/zh-cn/library/windows/hardware/ff554836(v=vs.85).aspx
- http://blog.csdn.net/wzy198852/article/details/32335371

谢谢