.NET Conf China 2025

改变世界 改变自己

2025 年 11 月 30 日 | 中国 上海

.NET Conf China 2025

改变世界 改变自己

.NET 程序故障的动态跟踪方法

— 演讲人：一线码农 —

Part.1

1. 动态跟踪前置基础

Part.2

2. 注入点分层解读

Part.3

3. 动态追踪三大战役

Part.1

1. 动态跟踪前置基础

1. 为什么要 动态追踪

01 Dump 分析的局限性

Dump分析就像是 **法医** 去命案现场。你可以看到那一刻的如下细节。

1）谁在场？ （提取指纹、DNA、衣物碎片）
2）他们手里拿着什么？ （尸体手握的物品、身旁散落的物件）
3）他们当时在做什么？ （尸体的姿势、创伤的位置、血迹的形态）

通过精湛的法医技术可以推断出大体的死因，但遗憾的是：

1）凶手是如何进入房间的？
2）事件发生前他们有什么异常行为？

这类问题，只看命案现场是没有用的，需要利用 摄像头 开启动态追踪。

02 什么是 动态追踪

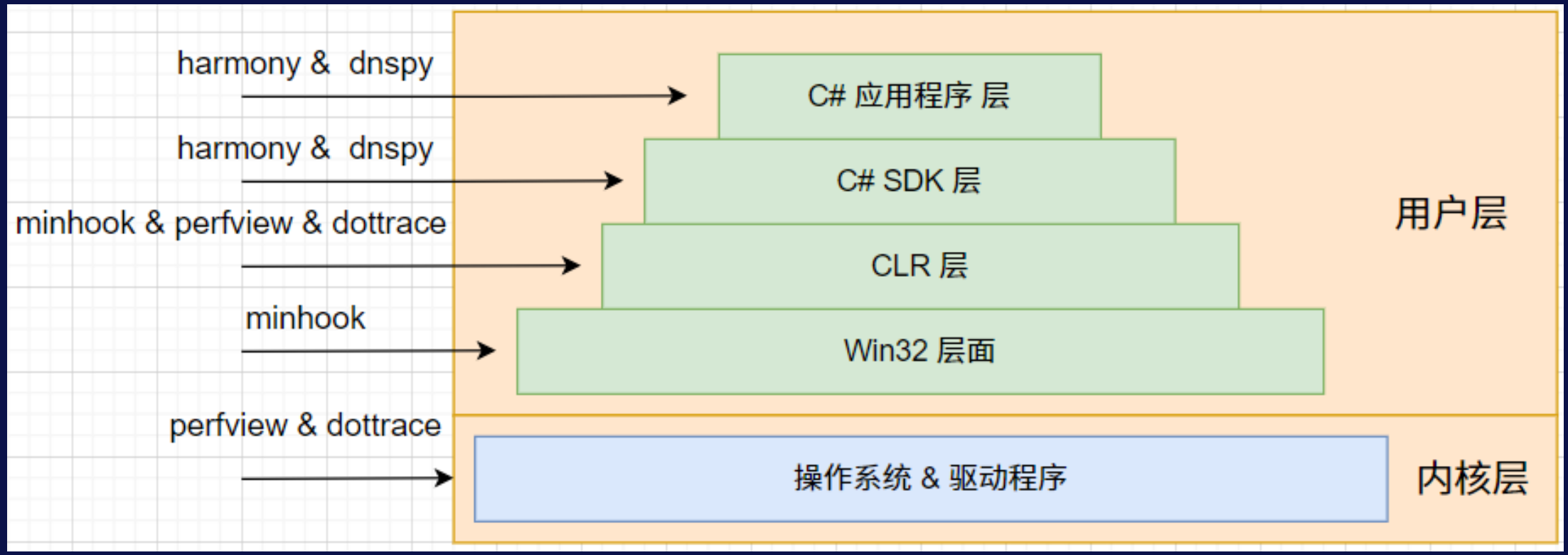
动态追踪则会在系统中布下 **“天眼”** 监控网络，它能完整回放“案发”前后所有的一举一动，让你不仅能锁定“嫌疑人”，更能清晰还原其行动路线、与其他第三方的互动关系，以及每一步的快慢节奏，从而建立起问题与根源之间的因果关系。

2. 动态追踪 的体系结构

01 体系层面介绍

以 Windows 为例，在其上部署的 .NET程序 往往会有 5 个层级，从上往下 依次为：

1）应用程序层　　2）SDK层　　3）CLR层　　4）Win32层　　5）操作系统&驱动程序层



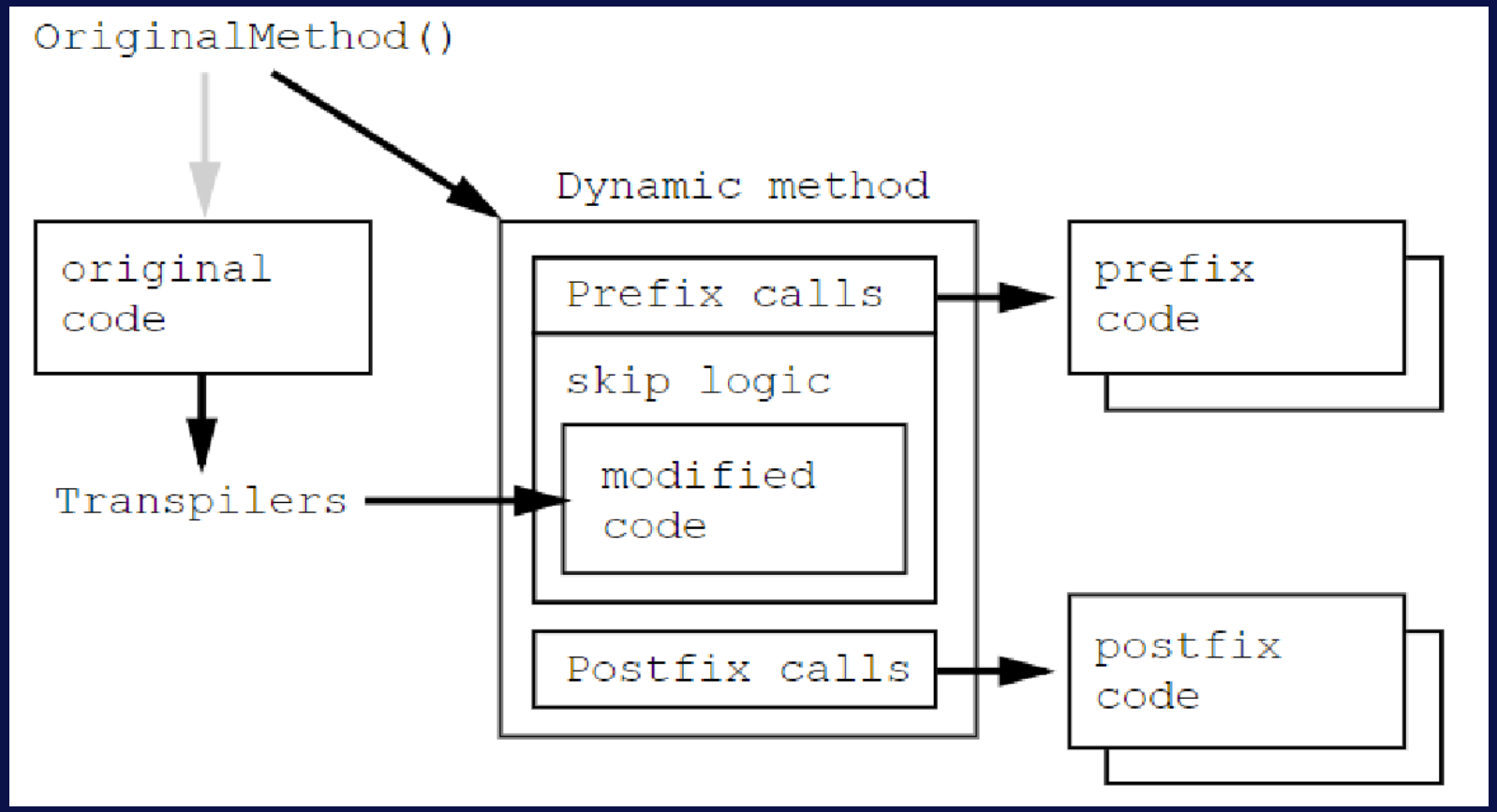
Part.2

2. 注入点分层解读

1. C# 应用层 & SDK 层注入

01 Harmony 原理介绍

它是一个强大的 .NET 运行时代码注入库，可在运行时修改、替换，扩展已编织的 .NET 应用程序 的方法，简单的说 Harmony 是一个 IL代码修改器，但在某些情况下，它会利用汇编代码级别的技巧来实现其目标。



tips：

1）Dynamic method：动态生成的方法。
2）Prefix calls：AOP 中的 前置方法。
3）modified：可被修改的 主方法体。
4）Postfix calls：AOP的后置方法。

1.1 C# 应用层 & SDK 层注入——追踪篇（上）

01 Harmony 案例之对 new Thread() 追踪

如果你要调查是谁在不断的 new Thread().Start()，可以用 harmony 给 Start() 的底层 Thread.StartCore() 方法加日志拦截。

1) HarmonyPatch：用特性方式对 StartCore 进行拦截。
2) Prefix：这是 Thread.StartCore() 执行之前要执行的方法。
3) Osid： 记录这个 Thread.Start() 所对应的 操作系统线程id。
4) StackTrace： 记录调用线程的 函数调用栈。

```
[HarmonyPatch(typeof(Thread))]
[HarmonyPatch("StartCore")]
0 references | hxc, 31 days ago | 1 author, 1 change
public class ThreadHook
{
    0 references | hxc, 31 days ago | 1 author, 1 change
    static void Prefix(Thread __instance, ref int ____managedThreadId)
    {
        var osid = AccessTools.Property(__instance.GetType(), "CurrentOSThreadId").GetValue(null);

        Console.WriteLine("----------------------------");
        Console.WriteLine($"当前上下文 tid={Thread.CurrentThread.ManagedThreadId}, " +
                          $"创建线程的 mid={____managedThreadId}" +
                          $"创建线程的 osid=0x{((ulong)osid).ToString("X")}");
        Console.WriteLine(Environment.StackTrace);
        Console.WriteLine("----------------------------");
    }
}
```

```
D:\travels\src\Example\Examp
Task线程开始，正在创建子线程...
----------------------------
当前上下文 tid=11，创建线程的 mid=12创建线程的 osid=0x71F0
   at System.Environment.get_StackTrace()
   at Example_0_6.ThreadHook.Prefix(Thread __instance, Int32
\HarmonyHook.cs:line 35
   at System.Threading.Thread.StartCore_Patch1(Thread this)
   at Example_0_6.Program.ProcessRequest() in D:\travels\src
   at Example_0_6.Program.<>c.<Main>b__0_0() in D:\travels\s
   at System.Threading.ExecutionContext.RunFromThreadPoolDis
nContext, ContextCallback callback, Object state)
   at System.Threading.Tasks.Task.ExecuteWithThreadLocal(Tas
   at System.Threading.ThreadPoolWorkQueue.Dispatch()
   at System.Threading.PortableThreadPool.WorkerThread.Worke
----------------------------
Task线程继续执行其他工作...
子线程开始执行...
2025/11/19 8:35:09: tid=12 子线程正在运行...
2025/11/19 8:35:10: tid=12 子线程正在运行...
2025/11/19 8:35:11: tid=12 子线程正在运行...
tid= 12 子线程已完成3秒工作，即将退出...
Task线程检测到子线程已退出。
```

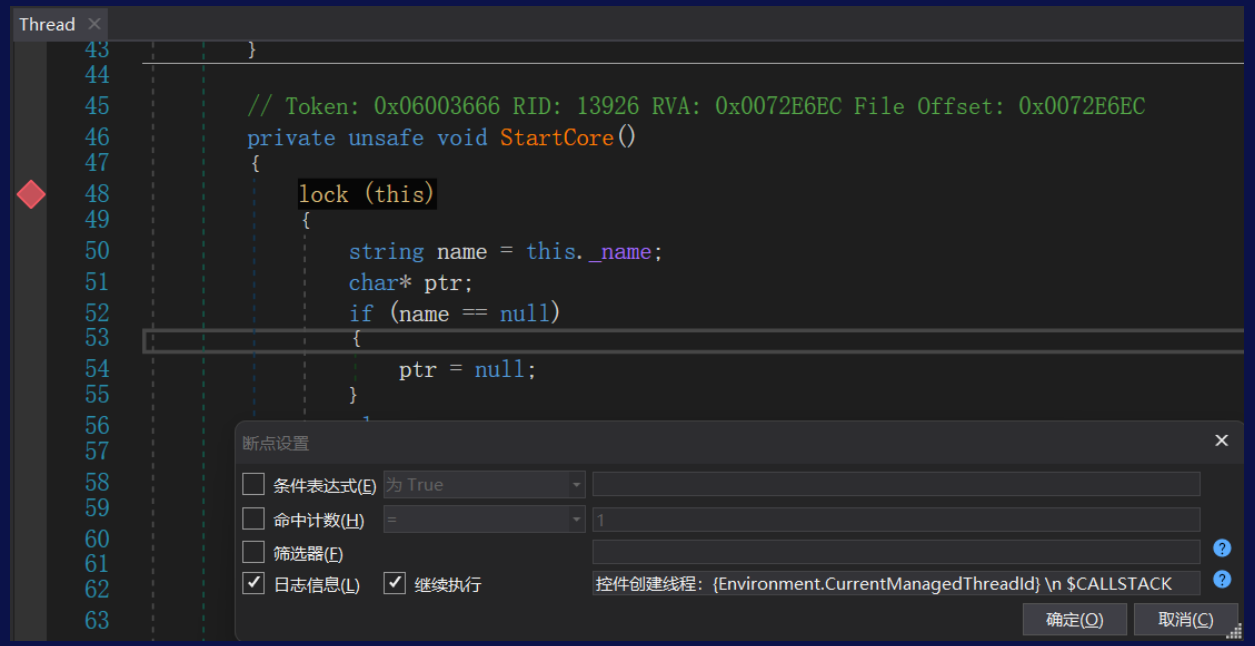
1.2 C# 应用层 & SDK 层注入——追踪篇(下)

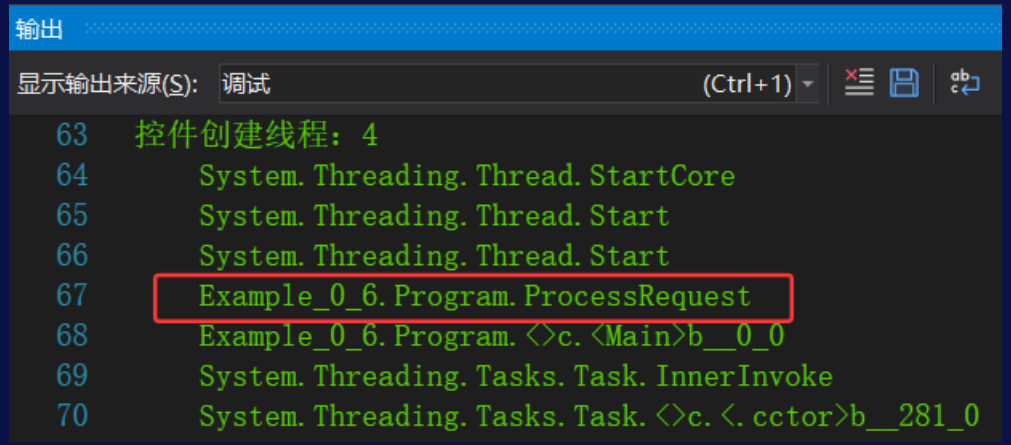
01 DnSpy 案例之 new Thread() 追踪

dnSpy 是一款免费开源轻量级的 .NET 程序反编译和调试神器，让你能查看、修改甚至调试没有源代码的 .NET 程序。

如果你想要寻找一款 **无侵入** 的工具，那就是 dnspy 了，可以把它当成 **应用程序** 直接部署在用户机器上，并采用 **断点日志** 功能实现 harmony 同样的效果。

tips： 1）$CALLSTACK：输出调用栈



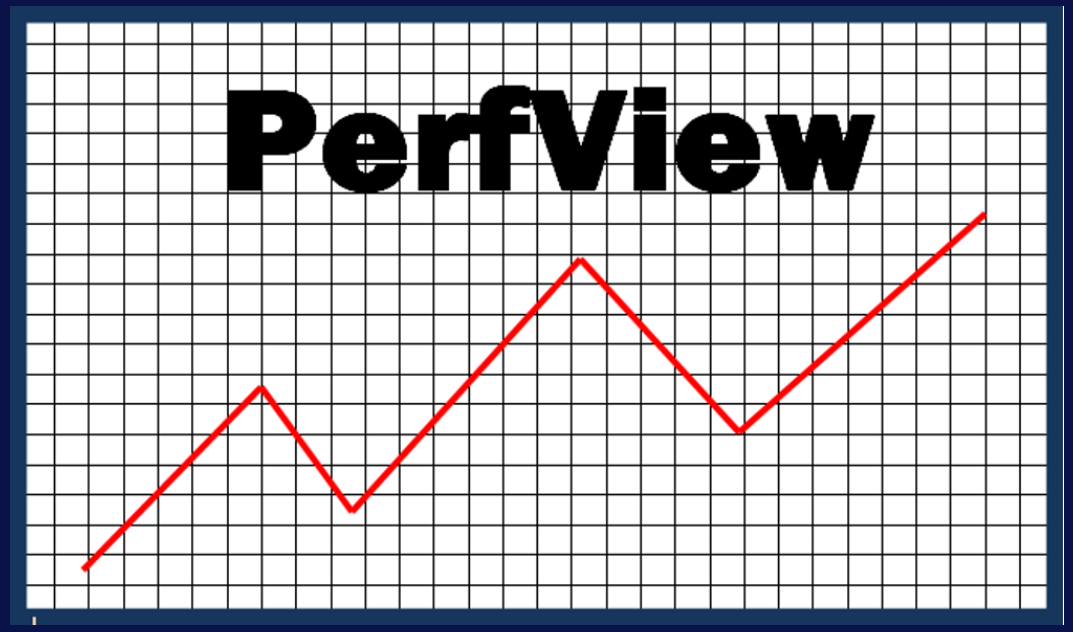


2. CLR 基础承载层 注入

01 perfview 原理介绍

perfView 是一款由微软开发的、免费且功能强大的性能分析工具，主要用于帮助开发者诊断 .NET 应用程序以及 Windows 操作系统自身的性能问题，其底层使用 Windows 事件跟踪（Event Tracing for Windows，ETW）技术。ETW 是 Windows 内核级的高性能事件记录系统，可以提供非常详细且低开销的诊断信息。

tips：在 CLR 中有大量的日志操作，默认是关闭的，由 informational_event_enabled_p 全局变量控制。一旦 perfview 开启，这个开关就会打开，随后大量的日志写入，比如 gc 处理日志。




```
inline
void gc_heap::fire_mark_event (int root_type, size_t& current_promoted_bytes,
{
#ifdef FEATURE_EVENT_TRACE
    if (informational_event_enabled_p)
    {
        current_promoted_bytes = get_promoted_bytes();
        size_t root_promoted = current_promoted_bytes - last_promoted_bytes;
        dprintf (3, ("h%d marked root %s: %zd (%zd - %zd)",
            heap_number, str_root_kinds[root_type], root_promoted,
            current_promoted_bytes, last_promoted_bytes));
        FIRE_EVENT(GCMarkWithType, heap_number, root_type, root_promoted);
        last_promoted_bytes = current_promoted_bytes;
    }
#endif // FEATURE_EVENT_TRACE
}
```

```
0:008> x coreclr!*informational_event_enabled_p*
00007ffa`34891fec coreclr!SVR::gc_heap::informational_event_enabled_p = false
00007ffa`34893d9c coreclr!WKS::gc_heap::informational_event_enabled_p = true
```

2.1 CLR 基础承载层——追踪篇(上)

01 perfview 案例之 观察 GC 详情

你发现程序运行了一段时间之后出现系统级变慢，想知道是不是因为高频GC触发所致，这时候就可以通过 perfview 的 stats 视图观察 GC 的所有详情。

eg：下面是向程序灌 1.5G 数据的 GC 触发详情。

GC Rollup By Generation

All times are in msec.

Gen	Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/ MSec GC	Survived MB/ MSec GC	Mean Pause	Induced
ALL	92	58.1	1,622.8	14,430.839	204.4	1,626.8	8.0	53.275	2.2	1
0	1	1.6	18.9	4.831	1.6	18.8	1,011.8	18.248	1.6	0
1	84	6.5	1,611.3	14,430.839	89.5	1,586.5	18.2	37.617	1.1	0
2	7	58.1	1,622.8	3,061.444	113.3	21.5	14.4	76.556	16.2	1

GC Events by Time

All times are in msec. Hover over columns for help.

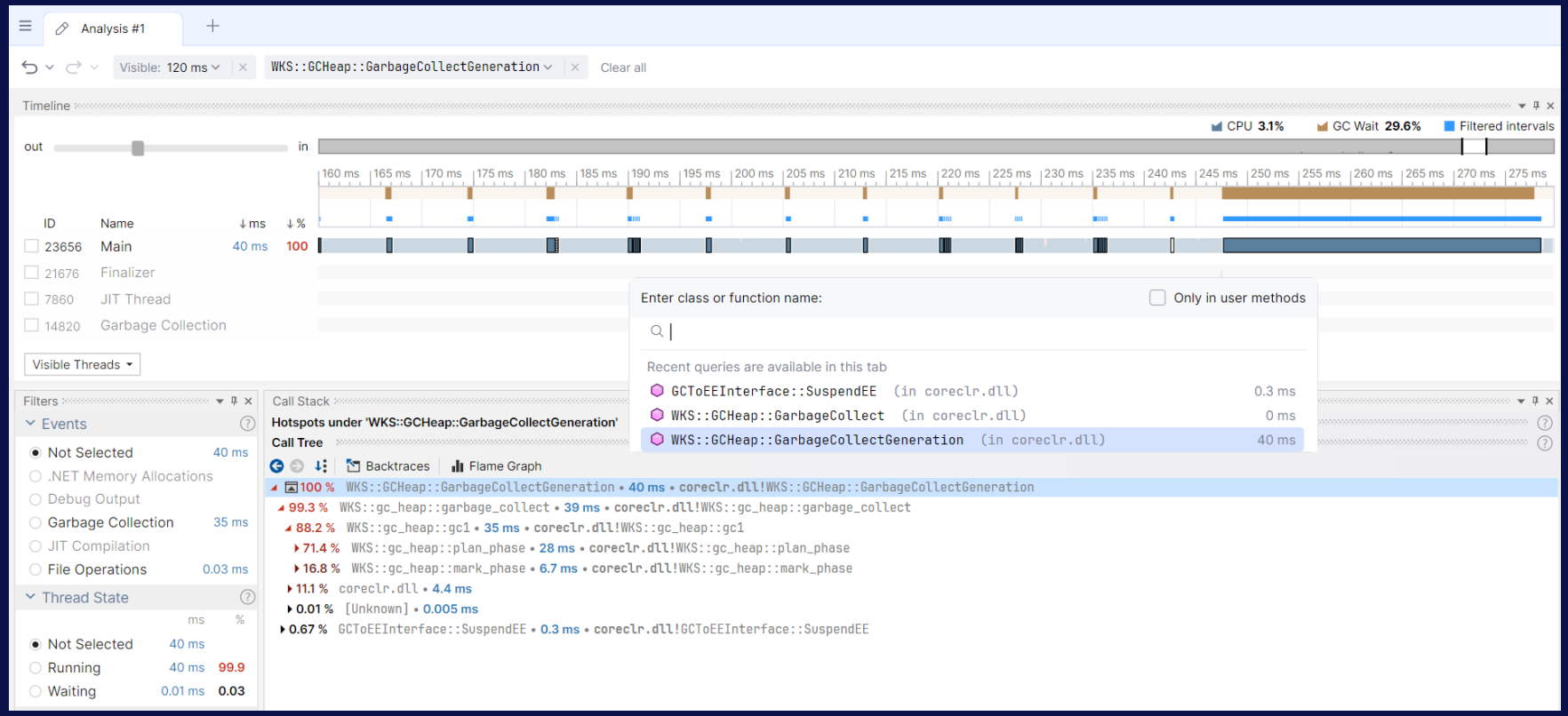
GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB
1	3,891.168	AllocSmall	0N	0.004	1.608	0.0	2.0	18.800	4.83	18.886	18.886	1.00	18.248	0.000	96	NaN	18.870
2	3,899.353	AllocSmall	1N	0.003	0.655	9.1	0.0	18.860	2,865.23	37.884	37.884	1.00	36.965	0.000	99	NaN	18.866
3	3,906.018	AllocSmall	2B	0.008	0.396	4.3	0.0	1.070	897.52	56.755	57.898	0.98	74.716	1.143	0	0.23	18.871
4	3,907.286	AllocSmall	1N	0.000	0.725	9.1	14.3	17.961	2,466.85	56.755	56.755	1.00	37.620	0.000	99	NaN	18.871
5	3,923.197	AllocSmall	1N	0.005	0.859	5.8	14.3	18.865	1,344.47	75.885	75.885	1.00	37.628	0.000	99	NaN	18.868
6	3,930.404	AllocSmall	1N	0.003	0.637	9.1	14.3	18.867	2,970.44	94.753	94.753	1.00	37.624	0.000	99	NaN	18.867
7	3,937.527	AllocSmall	2B	0.007	0.960	9.5	0.0	1.070	363.97	113.620	114.763	0.99	131.292	1.143	0	0.29	18.867
8	3,937.576	AllocSmall	1N	0.000	0.772	10.6	0.0	17.728	2,711.96	113.620	113.620	1.00	37.624	0.000	99	NaN	18.867

2.2 CLR 基础承载层——追踪篇(下)

01 dottrace 案例之 观察 慢GC

你发现某些 **方法变慢** 或发现 **慢GC**，想要观察到底慢在哪里？虽然可以用 perfview 进一步观察，但此类问题还有一个更加直观的方式，那就是利用 dotrtace 的时间轴模式，一目了然。

eg：发现程序有一个慢GC，可以直接搜索入口函数 GarbageCollectGeneration() 作为调查入口点。



3. Win32 层注入

01 Minhook 原理介绍

MinHook 是一个极简主义的 x86/x64 API 钩取（Hook）库。它的主要功能是拦截并重定向 Windows API 函数的调用。

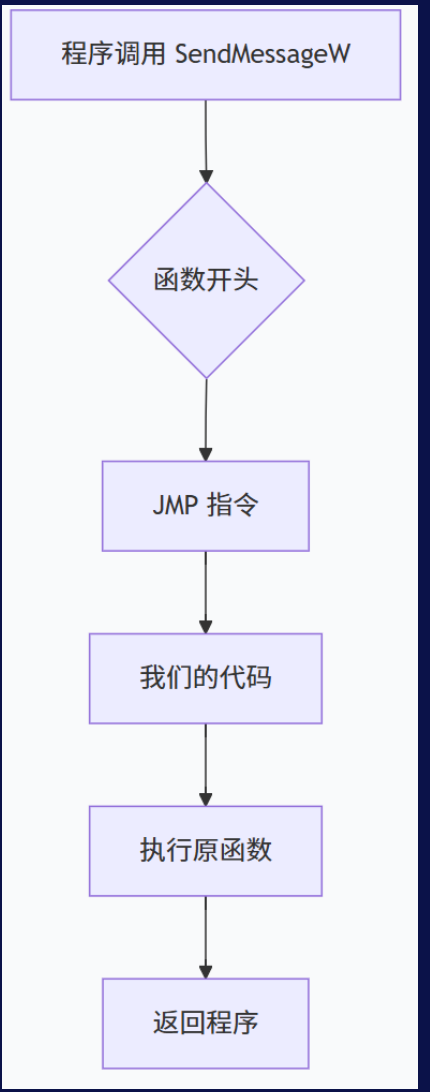
1）**游戏外挂/修改器：** 如钩取 DirectX/OpenGL 函数来绘制叠加层 或 模拟按键 等。
2）**软件调试与逆向工程：** 监视程序对特定 API 的调用，分析其行为。
3）**性能剖析：** 统计某个耗时函数被调用的次数和执行时间。
4）**功能扩展：** 为现有程序增加新的功能，而无需修改其原始代码。

eg：它的原理很简单，参见右图

MinHook

License BSD 2-Clause

The Minimalistic x86/x64 API Hooking Library for Windows



Part.3

3. 动态追踪三大战役

1. 寻找被意外创建的 内核句柄

01 问题成因

这是我的一位学员公司的生产故障，他发现自己的程序变得越来越卡，通过任务管理器发现这个进程有高达 870w 的进程内核句柄，想知道为什么？

02 分析难度在哪里

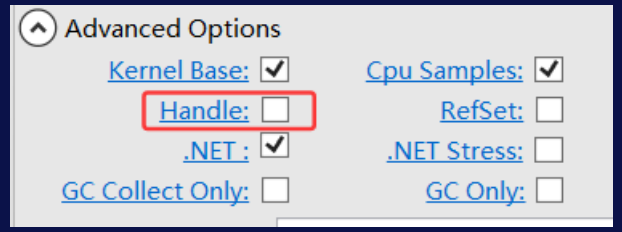
在事后的 windbg 分析中，只能识别出 内核句柄 类型，但无法知道这个句柄是 **哪个代码** 创建的，这时候只能求助 动态跟踪 技术。

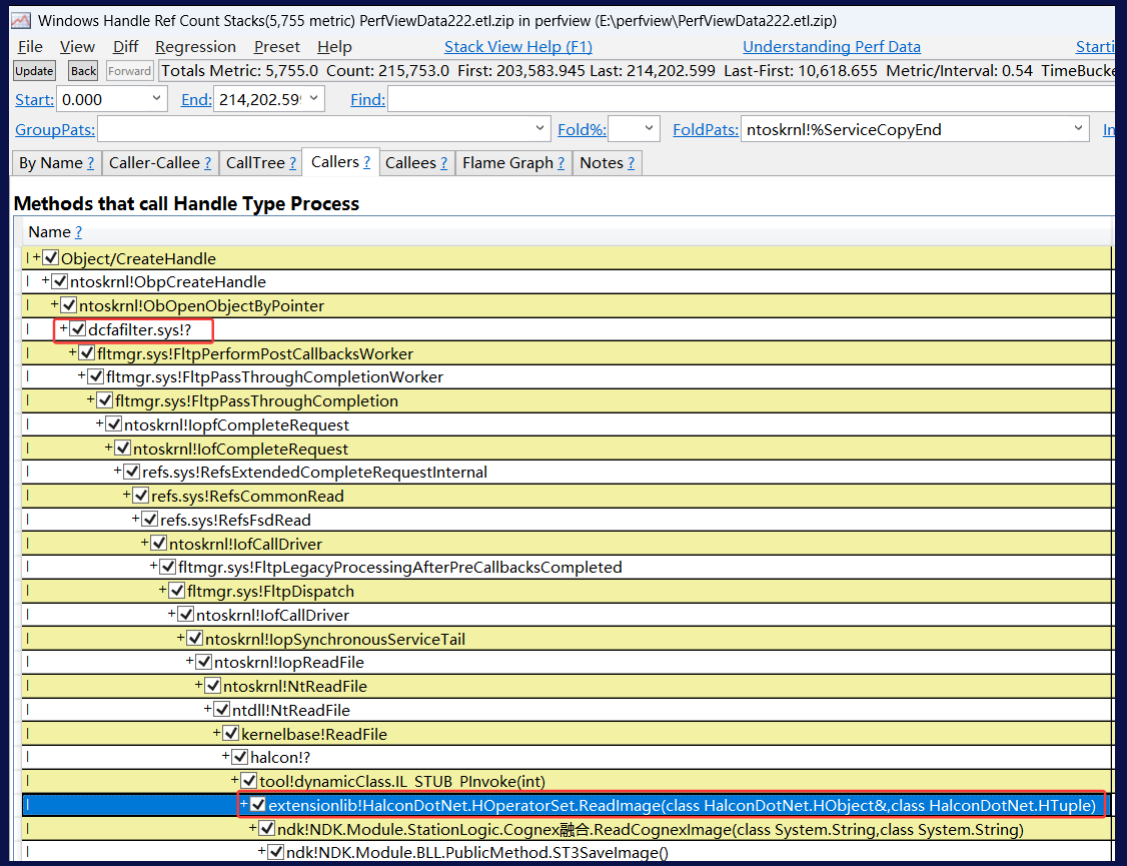
03 解决办法

handle 的创建和销毁是一个 etw 事件，所以开启 etw 追踪即可。

1）勾选 Handle 复选框。
2）在 Events 列表中观察调用栈。

tips：
最终发现是 dcfafilter.sys 所致，
经追查是一款 **安全软件**。

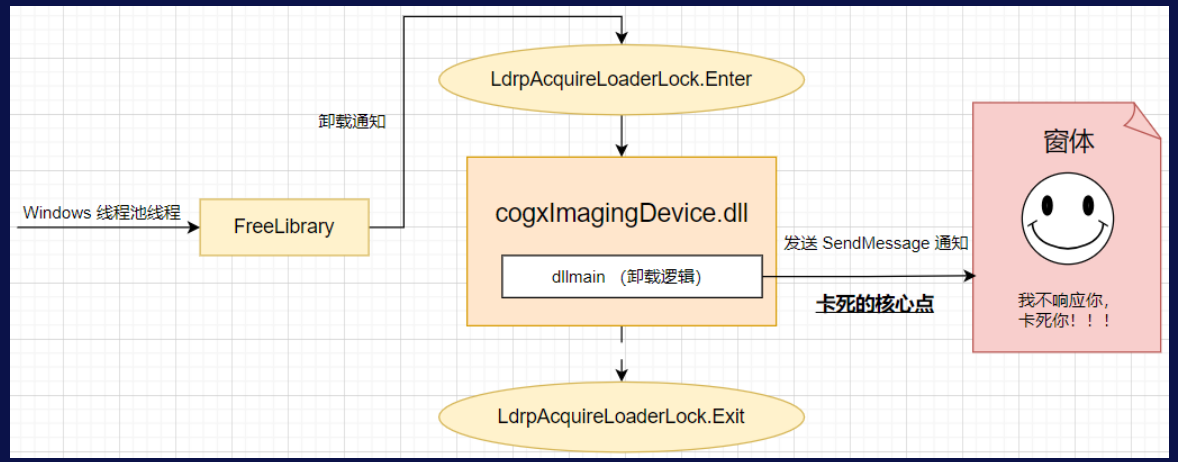




2. 寻找不能返回的 SendMessage

01 问题成因

这是一个工控程序的生产故障，其视觉组件 cogxImageingdevice.dll 实现了 dll 卸载通知，当进程中某一个dll卸载时， cogxImageingdevice.dll 会在获取进程加载锁之后 执行卸载逻辑，在执行这段逻辑中会有一个发送 SendMessage 操作，但可惜对方没有给与响应，导致程序卡死。



02 分析难度在哪里

在事后的dump分析中，可以用 windbg 提取 SendMessage 的 hWnd 参数，由于 hWnd 是系统资源，所以并不知道这个 hWnd 所对应的窗口是哪个进程，哪个线程创建的，进而分析进展不能走下去。

03 解决办法

使用 MinHook 对 SendMessage 进行注入，可获取如下重要信息：

1）GetWindowThreadProcessId： 用 hWnd 获取所属的 进程ID 和线程ID。

当程序出现卡死时，根据日志 比对 hWnd 即可。

```
1 reference | 0 changes | 0 authors, 0 changes
private static IntPtr HookedSendMessageW(IntPtr hWnd, uint Msg, IntPtr wParam, IntPtr lParam)
{
    Console.WriteLine($"[HOOK] SendMessageW: hWnd=0x{hWnd.ToInt64():X}, Msg=0x{Msg:X}");

    // 获取窗口所属的线程和进程ID
    uint processId = 0;
    uint threadId = GetWindowThreadProcessId(hWnd, out processId);

    // 使用 System.Diagnostics.Process 获取进程信息
    string processName = "Unknown";
    try
    {
        var targetProcess = System.Diagnostics.Process.GetProcessById((int)processId);
        processName = targetProcess.ProcessName;

        Console.WriteLine($"Window belongs to - ThreadID: {threadId}, ProcessID: {processId},
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }

    // 调用原始函数
    return _originalSendMessageW(hWnd, Msg, wParam, lParam);
}
```

3. 寻找被意外退出的线程

01 问题成因

这也是学员公司的一个dump，程序总是偶发性的崩溃，所有的崩溃点都在 GC 触发的内部函数上，没有底层知识几乎无法分析。

崩溃的诱因是托管线程被非托管代码退出了，比如 TerminalThread 会导致 GC 在 **一无所知** 的情况下进入到这个 **虚无线程** 上寻找引用根，引发访问违例异常。

```
0:000> k a
 # ChildEBP RetAddr
00 0297ee24 5f7d93bd     coreclr!InlinedCallFrame::FrameHasActiveCall+0xb
01 0297ee24 5f7d92c4     coreclr!ScanStackRoots+0x2d [D:\a\_work\1\s\src\c
02 0297ee48 5f871b05     coreclr!GCToEEInterface::GcScanRoots+0x8a [D:\a\_
03 (Inline) --------     coreclr!GCScan::GcScanRoots+0x13 [D:\a\_work\1\s\
04 0297eea0 5f872634     coreclr!WKS::gc_heap::mark_phase+0x153 [D:\a\_wor
05 0297eed8 5f870ee7     coreclr!WKS::gc_heap::gc1+0x6e [D:\a\_work\1\s\sr
06 0297eef0 5f899a25     coreclr!WKS::gc_heap::garbage_collect+0x166 [D:\a
07 0297ef1c 5fa5f84d     coreclr!WKS::GCHeap::GarbageCollectGeneration+0xf
08 0297ef2c 5fa5f7b3     coreclr!WKS::GCHeap::GarbageCollectTry+0x59 [D:\a
09 0297ef5c 5f9d44c6     coreclr!WKS::GCHeap::GarbageCollect+0xc3 [D:\a\_w
```

02 分析难度在哪里

在事后的dump分析中，我们无法找到是谁 **意外退出** 了这个线程，这种情况只能求助于 动态跟踪，追捕调用 TerminalThread() 函数的代码。

DBG	ID	OSID	ThreadOBJ	State	GC Mode	GC Alloc Context	Domain	Lock Count	Apt	Exception
0	1	554c	02B935A8	2a020	Cooperative	00000000:00000000	02b8ea88	-00001	MTA	(GC)
6	2	3a28	02B5D568	2b220	Preemptive	00000000:00000000	02b8ea88	-00001	MTA	(Finalizer)
XXXX	4	31dc	02BCD908	102b220	Preemptive	00000000:00000000	02b8ea88	-00001	Ukn	(Threadpool Worker)
9	5	1988	02BCF9E8	302b220	Preemptive	00000000:00000000	02b8ea88	-00001	MTA	(Threadpool Worker)
10	6	82a0	02BC16A8	102b220	Preemptive	00000000:00000000	02b8ea88	-00001	MTA	(Threadpool Worker)
7	3	5694	02BD6B58	2b220	Preemptive	00000000:00000000	02b8ea88	-00001	MTA	

03 解决办法

dump 现场残留着 Thread 的尸体信息，如图上的 **ID=4，OSID=0x31dc，**这也成为解决问题的突破口，有如下三种解决办法：

1）**process monitor**：启动 ETW 捕获 线程的退出事件，观察调用栈。
2）**minhook**：注入 TerminalThread() 方法，观察调用栈。
3）**harmony**：注入 Thread.StartCore 方法，记录 id=4 的线程调用栈。