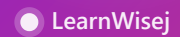


Flujo de profiling con Visual Studio para Wisej.NET

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Ejecutar el Performance Profiler contra una aplicación Wisej.NET de la forma correcta: compilaciones Release sin el depurador, la matriz de selección de herramientas, conectarse a dotnet o w3wp, y leer la línea de tiempo del resumen, el árbol de llamadas, la ruta crítica y las vistas de llamadores/llamados. Qué significa esto para un desarrollador de Wisej.NET que diagnostica cuellos de botella en una aplicación en producción, y cómo leer este módulo.



Enfoque en producción

Ejecutar el Performance Profiler contra una aplicación Wisej.NET de la forma correcta: compilaciones Release sin el depurador, la matriz de selección de herramientas, conectarse a dotnet o w3wp, y leer la línea de tiempo del resumen, el árbol de llamadas, la ruta crítica y las vistas de llamadores/llamados.

1. El problema en producción

El Módulo 1 dejó WisejPerfLab con una cifra y sin explicación: la actualización del dashboard tarda 1840 ms. El siguiente paso obvio es perfilarla, y ese paso obvio es donde se tuercen la mayoría de las investigaciones. Una traza recogida con el depurador mide el depurador. Una traza iniciada antes de que la aplicación esté caliente mide el JIT. Una traza que cubre dos minutos de clics lo mide todo y no demuestra nada. Una traza recogida con cuatro herramientas activas a la vez mide la propia sobrecarga del profiler. Este módulo es la disciplina mecánica que convierte una traza en evidencia admisible: la compilación correcta, la herramienta adecuada para el grupo de coste que sospechas, el proceso correcto cuando la aplicación no la inicia Visual Studio, un escenario por recogida y un informe que sabes leer.

2. Conceptos clave

Concepto	Por qué importa
Performance Profiler (Alt+F2)	La vía de recogida a posteriori: puede iniciar el proyecto sin el depurador, recoger un escenario completo y producir un informe guardado, y de ahí sale la evidencia real.
ventana Diagnostic Tools	La vista en vivo conectada al depurador; útil para echar un vistazo rápido mientras recorres el código paso a paso, engañosa como fuente de tiempos porque el depurador cambia lo que hace el runtime.
matriz de selección de herramientas	CPU Usage para el cálculo, Instrumentation para número de llamadas y tiempo de reloj, Object Allocation para la rotación de objetos, Memory Usage para la retención, Database para las consultas, File I/O para los archivos, .NET Async para las cadenas de await y .NET Counters para la salud en vivo.
.diagsession	El archivo de traza guardado; nombrarlo con módulo, escenario, compilación, conjunto de datos y marca de tiempo es lo que permite una comparación antes/después semanas más tarde.
línea de tiempo del resumen	El gráfico de CPU o de memoria de toda la recogida; acotar la selección a la acción exacta del usuario es el paso que elimina el arranque y el calentamiento del árbol de llamadas.
ruta crítica (hot path)	La cadena de llamadas más costosa del rango seleccionado; Show Hot Path la expande para que encuentres tu propio código bajo los frames del framework.
CPU propia frente a total	La total incluye las funciones hijas y la propia es solo el cuerpo de la función: una propia alta significa que el método en sí es costoso; una total alta con una propia baja significa que el coste está en lo que llama.

3. Desarrollo de la lección

Release, sin depurador, un escenario

El flujo predeterminado tiene cuatro pasos fijos antes de que ocurra nada interesante. Cambia la configuración a Release. Inicia la aplicación y ejecuta el escenario una vez para que el arranque y la compilación JIT ya estén pagados. Abre *Debug > Performance Profiler* con Alt+F2 y marca exactamente una herramienta. Después inicia la recogida, realiza solo el escenario que estás probando y deténla enseguida.

Cada regla tiene su razón de ser. Las compilaciones Debug desactivan las optimizaciones, no insertan nada en línea y pueden inventar pilas de llamadas que hacen que un helper trivial parezca una ruta crítica. El depurador añade su propia sobrecarga y cambia los tiempos. El calentamiento importa porque la primera ejecución de cualquier método incluye el tiempo de JIT, que es real pero no es el coste que nota el usuario en la centésima actualización. Y una herramienta cada vez importa porque la recogida no es gratuita: ejecutar Object Allocation e Instrumentation juntas añade la sobrecarga suficiente para desplazar la ruta crítica.

La última regla es la que más se incumple: *un escenario por traza*. Una recogida que contiene un inicio de sesión, tres actualizaciones, una búsqueda y algo de desplazamiento produce un árbol de llamadas en el que cada cifra interesante es una media de trabajo no relacionado. Cierra las pestañas del navegador que sobran, haz clic una vez y detén la recogida.

Elegir la herramienta según el grupo de coste

Los cinco grupos de coste del Módulo 1 se corresponden casi uno a uno con la lista de herramientas del Performance Profiler, y esa correspondencia es la clave de este módulo. Usa primero *CPU Usage* siempre que el síntoma sea CPU alta o cálculo lento: muestrea las pilas de forma barata y muestra adónde fue el procesador. Recurre a *Instrumentation* cuando necesites números de llamadas exactos o tiempo de reloj, o cuando sospeches que el tiempo se va en esperar y no en calcular: mide la entrada y la salida de cada función instrumentada, así que ve el tiempo bloqueado que el muestreo no ve.

Usa *.NET Object Allocation* cuando un escenario parezca generar mucha rotación de memoria, y los snapshots de *Memory Usage* cuando sospeches que algo queda retenido después de que el usuario lo cerrara. Usa *Database* cuando las consultas de EF Core o ADO.NET puedan dominar, *File I/O* para importaciones, exportaciones y generación de informes, y *.NET Async* cuando una cadena async sea más lenta de lo que sugiere la suma de sus await. *.NET Counters* es la excepción: es una vista en vivo de la CPU, el heap del GC, la tasa de asignación, la tasa de excepciones y el estado del thread pool, lo que la convierte en la herramienta adecuada para observar una aplicación bajo carga y no para diseccionar un clic.

En WisejPerfLab el laboratorio ejecuta las dos primeras. CPU Usage responde a «adónde fue el procesador durante la actualización»; Instrumentation responde a «cuántas veces se llamó a eso y qué parte de los 1840 ms se pasó realmente calculando». Dos trazas, un escenario y una conclusión que resiste una revisión.

Perfilar un proceso que Visual Studio no inició

Cuando el proyecto de inicio es la aplicación Wisej.NET, el profiler la lanza por ti y no hay nada que elegir. Con un alojamiento similar al de producción es distinto. Con Kestrel la aplicación es un proceso `dotnet` ; con IIS o IIS Express es `w3wp` o `iisexpress` , uno por grupo de aplicaciones. El Performance Profiler puede conectarse a un proceso en ejecución, y la única trampa real es conectarse al equivocado: en una máquina con varios sitios, confirma el id del proceso y el grupo de aplicaciones, envía una petición y comprueba que la recogida muestra actividad antes de ejecutar el escenario de verdad.

Los escenarios con WebSocket requieren una consideración más. Si el navegador se conecta mientras la traza todavía está arrancando, el handshake de la conexión y el renderizado inicial caen dentro de tu medición. O bien abres el navegador cuando la recogida ya está en marcha, o bien grabas una traza de calentamiento aparte y comparas lo mismo con lo mismo.

Leer el informe sin dejarse engañar

Todos los informes se abren en la línea de tiempo del resumen. Antes de mirar un solo nombre de función, arrastra una selección alrededor del intervalo exacto de la acción del usuario y deja que las vistas se recalculen; eso es lo que elimina de las cifras el arranque, la carga de la página y el tiempo inactivo. Después trabaja de arriba abajo: el árbol de llamadas con *Show Hot Path* para la cadena dominante, el flame graph para la misma información en forma gráfica, la vista de funciones ordenada por CPU propia

para los cuerpos costosos, y la vista de llamadores/llamados para saber quién llama a una función costosa y con qué frecuencia.

En una traza de Wisej.NET tu propio código está bastante abajo en la pila, por debajo del despacho de eventos de Wisej.NET y de los frames de alojamiento de ASP.NET Core. Eso es normal y no es un bug del framework. La regla es desplazarse más allá de la fontanería hasta el primer frame de tu propio namespace y empezar a razonar ahí. Si el frame del framework que está por encima de tu código es costoso, pregúntate qué le está pidiendo tu código: casi siempre resulta ser una modificación repetida miles de veces y no un framework lento.

```
// Trace note saved beside the .diagsession
// Module: 02
// Scenario: DashboardRefresh_5000Tickets
// Build: Release, x64, no debugger, warm
// Hosting: Kestrel, local SQL Server Browser: Edge
// Tool: CPU Usage, then Instrumentation follow-up
// Budget: < 800 ms server handler time
```

Por último, respeta el umbral de ruido. Un cambio de treinta milisegundos entre dos ejecuciones de un escenario de dos segundos no es una mejora, es variabilidad. Repite el escenario, toma la mediana y solo reivindica una mejora que puedas reproducir.



Errores habituales

Perfilar el arranque cuando la queja es un clic lento en un botón. Activar todas las herramientas a la vez y ahogar el escenario en sobrecarga de recogida. Comparar una traza Debug con una traza Release. Dejar el profiler en marcha durante una sesión larga hasta que nadie encuentra la interacción. Conectarse al `w3wp` equivocado y perfilar un grupo de aplicaciones inactivo.



Regla de evidencia

Define el escenario antes de abrir un profiler, recoge desde una compilación Release sin el depurador, cambia una sola cosa y vuelve a ejecutar el mismo escenario para comparar. Clasifica el retraso en uno de los cinco grupos —CPU en tu código, bloqueos y esperas, asignación y retención, acceso a datos y E/S, o carga útil de actualización—, porque cada grupo tiene su propia herramienta de Visual Studio, y una optimización que no puedes defender con una traza de antes y otra de después es una conjetura.



Objetivos de aprendizaje

Al terminar este módulo serás capaz de:

- ✓ Perfilar compilaciones Release con Debug > Performance Profiler / Alt+F2.
- ✓ Recoger trazas de CPU Usage, Instrumentation, .NET Object Allocation, Memory Usage, .NET Counters, Database, File I/O y .NET Async.
- ✓ Conectarte a un proceso Wisej.NET en ejecución cuando sea necesario e identificar el proceso correcto.

- ✓ Interpretar el árbol de llamadas, la ruta crítica, la vista de llamadores/llamados, la lista de funciones, el flame graph, el árbol de asignaciones y las comparaciones de snapshots.

Guía del laboratorio / examen

Qué construirás en el laboratorio práctico, el enfoque sugerido y los entregables requeridos.



Laboratorio práctico

Perfila como es debido la lenta actualización del dashboard de WisejPerfLab. Cambia a Release, calienta la aplicación una vez y recoge una traza de CPU Usage de ese único escenario con Alt+F2; acota la línea de tiempo del resumen al propio clic y usa Show Hot Path para encontrar la ruta principal por debajo de los frames de despacho de Wisej.NET. Recoge una segunda traza de Instrumentation del mismo escenario para obtener el número de llamadas y el tiempo de reloj, decide a partir de las dos trazas si el coste es CPU dentro de tu código o tiempo de espera, y guarda ambos archivos .diagsession con un nombre que incluya el módulo, el escenario, la compilación, el conjunto de datos y la marca de tiempo.

Punto de partida

Abres WisejPerfLab del Módulo 1 con sus tres escenarios instrumentados y su `Baseline.md`. La actualización del dashboard registra unos 1840 ms para 5000 filas y nadie sabe todavía por qué. En este laboratorio no cambia nada de la aplicación: se trata solo de recoger la evidencia sobre la que podrá actuar el siguiente módulo.

Enfoque sugerido

- ✓ Cambia la solución a Release, cierra todas las pestañas del navegador salvo la de la aplicación, restablece los datos de prueba al conjunto de la línea base y ejecuta la actualización una vez sin grabar para que el JIT y el arranque ya estén pagados.
- ✓ Abre *Debug > Performance Profiler* con Alt+F2, marca *CPU Usage* y nada más, inicia la recogida, haz clic en Refresh exactamente una vez y detén la recogida en cuanto la pantalla se estabilice.
- ✓ En el informe, arrastra una selección en la línea de tiempo del resumen alrededor del propio clic; después abre el árbol de llamadas, activa *Show Hot Path* y desplázate más allá de los frames de despacho de ASP.NET Core y Wisej.NET hasta la primera función de tu propio namespace. Anota su CPU total y propia, y usa la vista de Llamadores/llamados para ver quién la llama y con qué frecuencia.
- ✓ Recoge una segunda traza del mismo escenario con *Instrumentation*. Compara los números de llamadas y el tiempo de reloj con la imagen de CPU muestreada: una función con muchas llamadas y un coste por llamada diminuto es un problema distinto de una función con un cuerpo largo.
- ✓ Decide qué grupo de coste respalda la evidencia, escribe la nota de traza junto a cada `.diagsession` y nombra ambos archivos de modo que el módulo, el escenario, la compilación, el conjunto de datos y la marca de tiempo se lean sin abrirlos.

Entregables requeridos

- ✓ Traza de CPU Usage en Release del escenario de actualización del dashboard, con la línea de tiempo acotada al clic

- ✓ Captura de la ruta crítica que nombra la función principal de la aplicación y su CPU propia frente a la total
- ✓ Traza de Instrumentation del mismo escenario con número de llamadas y tiempo de reloj
- ✓ Nota de traza que indica curso, módulo, escenario, compilación, alojamiento, herramienta, navegador y presupuesto esperado
- ✓ Causa raíz sospechada en un párrafo que diga a cuál de los cinco grupos de coste apunta la evidencia

Preguntas de repaso

- ✓ Tu traza de CPU Usage solo explica 300 ms de los 1840 ms que midió la sonda. ¿Qué significa esa diferencia y qué herramienta usarías a continuación?
- ✓ Un método del framework está en lo alto de la ruta crítica con una CPU total muy alta y casi nada de CPU propia. Explica qué significa y qué mirarías en lugar de abrir un bug contra el framework.
- ✓ ¿Por qué otra persona necesita la nota de traza para reproducir tu resultado, si el `.diagsession` ya contiene toda la traza?



Regla de evidencia

Define el escenario antes de abrir un profiler, recoge desde una compilación Release sin el depurador, cambia una sola cosa y vuelve a ejecutar el mismo escenario para comparar. Clasifica el retraso en uno de los cinco grupos —CPU en tu código, bloqueos y esperas, asignación y retención, acceso a datos y E/S, o carga útil de actualización—, porque cada grupo tiene su propia herramienta de Visual Studio, y una optimización que no puedes defender con una traza de antes y otra de después es una conjetura.