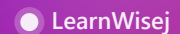


Memoria, asignaciones y fugas de sesión

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Presión de asignación frente a memoria retenida, el flujo de comparación de snapshots, los patrones de retención de Wisej.NET (eventos estáticos, temporizadores, tareas en segundo plano, cachés globales) y la lista de comprobación de liberación que demuestra que una fuga ha desaparecido. 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

Presión de asignación frente a memoria retenida, el flujo de comparación de snapshots, los patrones de retención de Wisej.NET (eventos estáticos, temporizadores, tareas en segundo plano, cachés globales) y la lista de comprobación de liberación que demuestra que una fuga ha desaparecido.

1. El problema en producción

El dashboard de WisejPerfLab ya cumple su presupuesto de CPU, y aun así la aplicación se viene abajo tras cuatro horas en producción. Los problemas de memoria se comportan de forma distinta a los de CPU: son invisibles en una demo de dos minutos, nunca aparecen como un clic lento, y son la razón de que un servidor que aguanta sin esfuerzo ochenta sesiones por la mañana rechace conexiones a media tarde. En el laboratorio hay dos, y no son el mismo problema. La búsqueda de tickets asigna mucha más memoria de la que necesita, lo que cuesta tiempo de GC y CPU mientras se ejecuta. El formulario de detalle del ticket nunca se libera de verdad cuando el usuario lo cierra, lo que cuesta memoria para siempre. Uno se encuentra con una traza de asignaciones; el otro, solo con dos snapshots y una ruta a la raíz.

2. Conceptos clave

Concepto	Por qué importa
presión de asignación	Cuánta memoria crea un escenario mientras se ejecuta; una presión alta aumenta la frecuencia del GC y roba CPU aunque no haya ninguna fuga.
memoria retenida	Lo que sigue siendo alcanzable cuando el escenario debería haber terminado; es lo que se multiplica por el número de sesiones y decide la capacidad.
.NET Object Allocation	La herramienta que registra cada asignación administrada durante un escenario y la agrupa por tipo y por la ruta de llamadas que asigna.
snapshots de Memory Usage	Capturas del heap tomadas en momentos elegidos; comparar el snapshot A y el snapshot B muestra qué tipos ganaron instancias que nunca se liberaron.
ruta a la raíz	La cadena de referencias que mantiene vivo un objeto; es la evidencia real de una fuga y la única forma de demostrar que ha desaparecido.
raíz de retención	Lo que está en lo alto de esa cadena: un campo estático, un evento estático, un temporizador en marcha, una tarea en segundo plano o una caché de larga vida.
presupuesto de memoria por sesión	Una cifra, en megabytes retenidos por sesión inactiva, que convierte «la aplicación usa mucha memoria» en un aprobado o un suspenso y, más adelante, en un umbral del health check.

3. Desarrollo de la lección

Por qué la memoria es una cuestión de capacidad en Wisej.NET

Wisej.NET mantiene vivos los objetos de interfaz del servidor durante toda la sesión. Eso es lo que hace agradable el modelo de programación, y también lo que encarece una fuga. Un formulario modal que nunca se libera, un `BindingSource` que sigue reteniendo cincuenta mil filas, un evento estático que fija una página, un temporizador que sigue haciendo tick después de ocultar el panel: cada uno son unos pocos megabytes, invisibles en el portátil de un desarrollador y decisivos con doscientas sesiones.

El modelo mental que hay que llevar al profiler es que la memoria tiene dos preguntas distintas. *¿Cuánta rotación generó este escenario?* tiene que ver con el coste del GC y la CPU. *¿Qué sigue vivo ahora que el usuario lo ha cerrado todo?* tiene que ver con la capacidad. Un escenario puede ser pésimo en una y perfecto en la otra, así que un informe que responde solo a una de las dos no ha dado el visto bueno a la aplicación.

Encontrar la presión de asignación

Para las asignaciones, ejecuta el escenario con *.NET Object Allocation* en Release, exactamente una vez, y abre las vistas de asignación. La vista de tipos responde a «qué se está creando» y el árbol de llamadas de asignación responde a «quién lo crea». En la búsqueda de tickets la respuesta no es halagadora: los objetos del modelo de fila y sus cadenas formateadas se reconstruyen cada vez que el grid se redibuja, así que una sola búsqueda asigna varias veces los datos que muestra.

La corrección es la misma disciplina del Módulo 3, aplicada a objetos en lugar de a CPU: proyectar una sola vez en un tipo de fila ligero y entregárselo al grid, en lugar de construir objetos nuevos en cada redibujado. Lo que buscas en la traza «después» es una bajada tanto del total de bytes asignados como del número de instancias del tipo culpable y, idealmente, una línea del GC visiblemente más tranquila.

Encontrar la retención

La retención necesita otro flujo de trabajo, y el orden de los pasos es toda la técnica. Inicia la aplicación en Release y caliéntala. Toma el snapshot A. Ejecuta el escenario que debería ser neutro —abrir y cerrar el formulario de detalle del ticket cincuenta veces, para que una instancia fugada se convierta en cincuenta y no pueda confundirse con ruido—. Devuelve la aplicación a un punto realmente inactivo. Toma el snapshot B y compáralos.

En la comparación buscas tipos cuyo número de instancias creció y nunca volvió a bajar:

`TicketDetailForm` con cincuenta instancias no deja lugar a dudas. Selecciona el tipo, abre las instancias e inspecciona la ruta a la raíz. En WisejPerfLab aparecen dos raíces. Una es un evento estático: el formulario se suscribió a `GlobalTicketBus.TicketChanged` en su constructor y nunca canceló la suscripción, así que la lista estática de delegados retiene todos los formularios abiertos alguna vez. La otra es un `System.Timers.Timer` que el formulario inicia para actualizar su cabecera; su callback captura el formulario, así que mientras el temporizador esté vivo, el formulario también lo estará.

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        GlobalTicketBus.TicketChanged -= OnTicketChanged; // the static root
        _refreshTimer?.Stop(); // the timer root
        _refreshTimer?.Dispose();
        bindingSource1.DataSource = null; // release the rows
        _largeImage?.Dispose();
    }
    base.Dispose(disposing);
}
```

Las dos correcciones van en el mismo sitio: un `Dispose` sobrescrito que cancela las suscripciones que hizo, detiene y libera lo que inició, y limpia los enlaces de datos que retienen datos grandes. Todo lo que toque la interfaz desde un temporizador o una tarea en segundo plano debería además comprobar `IsDisposed` antes de actualizar controles, porque el callback puede sobrevivir al formulario unos milisegundos aunque la liberación sea correcta.

Demostrarlo y escribir el presupuesto

Una corrección de memoria no está terminada cuando el código parece correcto. Repite el mismo escenario de apertura y cierre, vuelve a tomar el snapshot B y muestra dos cosas: el número de instancias de `TicketDetailForm` vuelve a su línea base, y la antigua ruta a la raíz ya no existe. La segunda es la importante: un recuento que baja porque casualmente se ejecutó una recolección de basura no es una prueba, y forzar recolecciones hasta que las cifras parezcan aceptables es la forma en que una fuga sobrevive a una revisión.

Termina con los dos documentos que necesita el resto del curso. El presupuesto de memoria por sesión indica cuántos megabytes puede retener una sesión inactiva, medidos en tu propia aplicación y no supuestos; el Módulo 7 convierte esa cifra en el umbral `maxSessions` del health check. La lista de comprobación de liberación es la versión revisable del mismo conocimiento: cancelar suscripciones a eventos, detener y liberar temporizadores, cancelar el trabajo en segundo plano, limpiar enlaces de datos, liberar imágenes y streams, soltar listas grandes y no guardar nunca estado mutable del usuario en un campo estático.



Errores habituales

Medir las asignaciones y declarar la aplicación libre de fugas sin comprobar nunca la retención. Forzar recolecciones de basura hasta que las cifras queden bien en lugar de leer las rutas a la raíz. Mantener un `DbContext`, un `DataTable` o una caché de imágenes en la sesión para siempre. Suscribirse a un evento estático desde una página y no cancelar nunca la suscripción. Actualizar controles desde el callback de un temporizador sin comprobar `IsDisposed`.



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:

- ✓ Usar .NET Object Allocation para encontrar rutas de código con muchas asignaciones durante un escenario.
- ✓ Usar snapshots de Memory Usage para comparar el estado del heap antes y después de cerrar objetos de interfaz de Wisej.NET.
- ✓ Identificar raíces de retención causadas por campos estáticos, eventos, temporizadores, cachés y servicios de larga vida.
- ✓ Crear un presupuesto de memoria por sesión y una lista de comprobación de liberación.

Guía del laboratorio / examen

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



Laboratorio práctico

Ataca las dos mitades del problema de memoria en WisejPerfLab. Ejecuta el escenario de búsqueda de tickets con `.NET Object Allocation`, encuentra los modelos de fila y las cadenas asignadas en cada búsqueda y redúcelos proyectando una sola vez en lugar de reconstruir objetos en cada redibujado. Después toma un snapshot de `Memory Usage` tras el calentamiento, abre y cierra el formulario de detalle del ticket cincuenta veces, toma un segundo snapshot y compáralos: los formularios retenidos están enraizados por una suscripción estática a `GlobalTicketBus.TicketChanged` y por un temporizador de actualización que nunca se detuvo. Cancela la suscripción y detén ambos en un `Dispose` sobrescrito, limpia el origen de datos del `BindingSource` y vuelve a ejecutar el escenario de apertura/cierre para mostrar que el heap retenido vuelve a acercarse al snapshot A.

Punto de partida

WisejPerfLab tras el Módulo 3: la actualización del dashboard está dentro de su presupuesto de CPU, el grid de tickets sigue enlazando entidades completas, y el formulario de detalle del ticket se suscribe a `GlobalTicketBus.TicketChanged` en su constructor e inicia un `System.Timers.Timer` para actualizar su cabecera. Ninguna de las dos cosas se deshace en ninguna parte. El presupuesto del Módulo 1 ya incluye un objetivo de memoria retenida que la aplicación no cumple.

Enfoque sugerido

- ✓ Ejecuta la búsqueda de tickets una vez con `.NET Object Allocation` en Release. Usa la vista de tipos para ver qué domina y el árbol de llamadas de asignación para ver quién lo crea, y anota el total de bytes asignados más el número de instancias del tipo principal.
- ✓ Proyecta una sola vez en un tipo de fila ligero y enlaza ese tipo, para que los objetos de fila y sus cadenas de presentación se creen una única vez por búsqueda en lugar de reconstruirse en cada redibujado. Vuelve a ejecutar la traza de asignaciones sobre el mismo escenario.
- ✓ Cambia a `Memory Usage`. Calienta la aplicación, toma el snapshot A, abre y cierra el formulario de detalle del ticket cincuenta veces para que una única instancia fugada se convierta en cincuenta, devuelve la aplicación a un punto realmente inactivo y toma el snapshot B.
- ✓ Compara los snapshots, selecciona el tipo de formulario cuyo número de instancias creció y lee la ruta a la raíz de dos instancias. Captura ambas raíces: la suscripción al evento estático y el temporizador activo que retiene el formulario capturado por el callback.
- ✓ Sobrescribe `Dispose(bool)` en el formulario para cancelar la suscripción al evento estático, detener y liberar el temporizador, limpiar el origen de datos del `BindingSource` y liberar las imágenes; añade una comprobación de `IsDisposed` antes de que el callback del temporizador toque los controles. Vuelve a ejecutar el mismo escenario de apertura/cierre y demuestra que han desaparecido tanto el recuento como la ruta a la raíz.

Entregables requeridos

- ✓ Traza de asignaciones que nombra el tipo con más asignaciones y la ruta de llamadas del escenario de búsqueda, antes y después
- ✓ Comparación de los snapshots A/B que muestra el número de instancias retenidas del formulario de detalle antes de la corrección
- ✓ Captura de la ruta a la raíz que identifica la suscripción al evento estático y el temporizador activo
- ✓ Sobrescritura de Dispose que cancela la suscripción al evento estático, detiene y libera el temporizador y limpia el enlace de datos
- ✓ Presupuesto de memoria por sesión y lista de comprobación de liberación, con el snapshot «después» que demuestra que la ruta de retención ha desaparecido

Preguntas de repaso

- ✓ Tu traza de asignaciones mejoró muchísimo, pero la comparación de snapshots no ha cambiado. ¿Qué tipo de problema corregiste, cuál no, y cómo notaría la diferencia un usuario?
- ✓ ¿Por qué el laboratorio abre y cierra el formulario cincuenta veces y no una, y qué concluirías si en el snapshot B apareciera una sola instancia?
- ✓ El callback del temporizador ocupa una línea y solo asigna una etiqueta. Explica con precisión cómo mantiene vivo un formulario entero, y qué resuelve y qué no una comprobación de `IsDisposed`.



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.