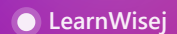


Speicher, Allokationen und Session-Lecks

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Allokationsdruck im Vergleich zu gehaltenem Speicher, der Workflow zum Vergleichen von Snapshots, die Retention-Muster in Wisej.NET (statische Events, Timer, Hintergrund-Tasks, globale Caches) und die Dispose-Checkliste, die belegt, dass ein Leak beseitigt ist. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Allokationsdruck im Vergleich zu gehaltenem Speicher, der Workflow zum Vergleichen von Snapshots, die Retention-Muster in Wisej.NET (statische Events, Timer, Hintergrund-Tasks, globale Caches) und die Dispose-Checkliste, die belegt, dass ein Leak beseitigt ist.

1. Das Problem im Produktivbetrieb

Das WisejPerfLab-Dashboard hält sein CPU-Budget inzwischen ein, und trotzdem bricht die App nach vier Stunden im Produktivbetrieb zusammen. Speicherprobleme verhalten sich anders als CPU-Probleme: In einer zweiminütigen Demo sind sie unsichtbar, sie zeigen sich nie als einzelner langsamer Klick, und sie sind der Grund, warum ein Server, der morgens bequem achtzig Sessions trägt, am Nachmittag Verbindungen ablehnt. Im Lab gibt es zwei davon, und es ist nicht dasselbe Problem. Die Ticketsuche allokiert weit mehr als nötig, was während der Ausführung GC-Zeit und CPU kostet. Das Ticket-Detailformular wird nach dem Schließen nie wirklich freigegeben, was dauerhaft Speicher kostet. Das eine finden Sie mit einem Allokations-Trace, das andere nur mit zwei Snapshots und einem Path to Root.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Allokationsdruck	Wie viel Speicher ein Szenario während der Ausführung erzeugt; hoher Druck erhöht die GC-Häufigkeit und kostet CPU, auch wenn nichts leckt.
gehaltener Speicher	Was noch erreichbar ist, nachdem das Szenario hätte beendet sein sollen; genau das multipliziert sich mit der Anzahl der Sessions und entscheidet über die Kapazität.
.NET Object Allocation	Das Tool, das jede verwaltete Allokation während eines Szenarios aufzeichnet und nach Typ und allozierendem Aufrufpfad gruppiert.
Memory Usage-Snapshots	Heap-Aufnahmen zu gewählten Zeitpunkten; der Vergleich von Snapshot A und Snapshot B zeigt, welche Typen Instanzen hinzugewonnen haben, die nie freigegeben wurden.
Path to Root	Die Kette von Referenzen, die ein Objekt am Leben hält; sie ist der eigentliche Beleg für ein Leck und die einzige Möglichkeit zu beweisen, dass es beseitigt ist.
Retention-Root	Was ganz oben in dieser Kette steht – ein statisches Feld, ein statisches Event, ein laufender Timer, ein Hintergrund-Task oder ein langlebiger Cache.
Speicherbudget pro Session	Eine Zahl in Megabyte gehaltenen Speichers pro inaktiver Session, die aus „die App braucht viel Speicher“ ein Bestanden oder Nicht bestanden und später einen Health-Check-Schwellenwert macht.

3. Der rote Faden der Lektion

Warum Speicher in Wisej.NET eine Kapazitätsfrage ist

Wisej.NET hält serverseitige UI-Objekte für die Dauer der Session am Leben. Das macht das Programmiermodell angenehm – und ein Leck teuer. Ein modales Formular, das nie freigegeben wird, eine **BindingSource**, die noch fünfzigtausend Zeilen hält, ein statisches Event, das eine Page festhält, ein Timer, der weitertickt, nachdem das Panel ausgeblendet wurde: Jedes davon sind ein paar Megabyte, unsichtbar auf einem Entwickler-Laptop und entscheidend bei zweihundert Sessions.

Das mentale Modell, das Sie in den Profiler mitnehmen sollten: Zu Speicher gehören zwei getrennte Fragen. *Wie viel Speicher hat dieses Szenario umgeschlagen?* betrifft GC-Kosten und CPU. *Was lebt noch, nachdem der Benutzer alles geschlossen hat?* betrifft die Kapazität. Ein Szenario kann bei der einen Frage katastrophal und bei der anderen tadellos sein, und ein Bericht, der nur eine davon beantwortet, hat die App nicht freigegeben.

Den Allokationsdruck finden

Für Allokationen führen Sie das Szenario unter *.NET Object Allocation* in Release genau einmal aus und öffnen die Allokationsansichten. Die Typansicht beantwortet „Was wird erzeugt?“, die Allokations-Aufrufstruktur „Wer erzeugt es?“. Bei der Ticketsuche ist die Antwort wenig schmeichelhaft: Zeilenmodell-Objekte und ihre formatierten Strings werden bei jedem Neuzeichnen des Grids neu aufgebaut, sodass eine einzige Suche ein Vielfaches der angezeigten Daten allokiert.

Der Fix folgt derselben Disziplin wie in Modul 3, nur auf Objekte statt auf CPU angewendet: einmal in einen schlanken Zeilentyp projizieren und diesen an das Grid übergeben, statt bei jedem Neuzeichnen neue Objekte zu konstruieren. Im Trace nachher suchen Sie einen Rückgang sowohl der insgesamt allokierten Bytes als auch der Instanzanzahl des auffälligen Typs – und im Idealfall eine sichtbar ruhigere GC-Linie.

Die Retention finden

Retention erfordert einen anderen Workflow, und die Reihenfolge der Schritte ist die ganze Technik. Starten Sie die App in Release und wärmen Sie sie auf. Erstellen Sie Snapshot A. Führen Sie das Szenario aus, das neutral sein sollte – öffnen und schließen Sie das Ticket-Detailformular fünfzigmal, damit aus einer geleakten Instanz fünfzig werden, die sich nicht mit Rauschen verwechseln lassen. Bringen Sie die App in einen echten Leerlaufzustand zurück. Erstellen Sie Snapshot B und vergleichen Sie.

Im Vergleich suchen Sie Typen, deren Instanzanzahl gewachsen und nie wieder gesunken ist:

`TicketDetailForm` mit fünfzig Instanzen ist eindeutig. Wählen Sie den Typ aus, öffnen Sie die Instanzen und untersuchen Sie den Path to Root. In WisejPerfLab tauchen zwei Roots auf. Das eine ist ein statisches Event – das Formular hat `GlobalTicketBus.TicketChanged` in seinem Konstruktor abonniert und nie abgemeldet, sodass die statische Delegate-Liste jedes jemals geöffnete Formular hält. Das andere ist ein `System.Timers.Timer`, den das Formular startet, um seinen Kopfbereich zu aktualisieren; sein Callback erfasst das Formular, und solange der Timer lebt, lebt auch das Formular.

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        GlobalTicketBus.TicketChanged -= OnTicketChanged; // der statische Root
        _refreshTimer?.Stop(); // der Timer-Root
        _refreshTimer?.Dispose();
        bindingSource1.DataSource = null; // die Zeilen freigeben
        _largeImage?.Dispose();
    }
    base.Dispose(disposing);
}
```

Beide Fixes gehören an dieselbe Stelle: ein überschriebenes `Dispose`, das abmeldet, was abonniert wurde, stoppt und verwirft, was gestartet wurde, und die Bindings leert, die große Datenmengen halten. Alles, was aus einem Timer oder einem Hintergrund-Task heraus die UI berührt, sollte außerdem `IsDisposed` prüfen, bevor es Controls aktualisiert, denn der Callback kann das Formular um einige Millisekunden überleben, selbst wenn das Dispose korrekt ist.

Den Fix belegen und das Budget schreiben

Ein Speicher-Fix ist nicht fertig, wenn der Code richtig aussieht. Wiederholen Sie das identische Öffnen- und-Schließen-Szenario, erstellen Sie erneut Snapshot B und zeigen Sie zwei Dinge: Die Instanzanzahl von `TicketDetailForm` kehrt auf ihre Baseline zurück, und der alte Path to Root existiert nicht mehr. Das Zweite ist das Entscheidende – eine Anzahl, die sinkt, weil zufällig eine Garbage Collection lief, ist kein Beleg, und Collections so lange zu erzwingen, bis die Zahlen akzeptabel aussehen, ist genau der Weg, auf dem ein Leck eine Prüfung übersteht.

Schließen Sie mit den beiden Dokumenten ab, die der Rest des Kurses braucht. Das Speicherbudget pro Session legt fest, wie viele Megabyte eine inaktive Session halten darf – gemessen an Ihrer eigenen App, nicht geschätzt; Modul 7 macht aus dieser Zahl den Schwellenwert `maxSessions` des Health Checks. Die Dispose-Checkliste ist die prüfbare Fassung desselben Wissens: Events abmelden, Timer stoppen und verwerfen, Hintergrundverarbeitung abbrechen, Bindings leeren, Bilder und Streams verwerfen, große Listen freigeben und niemals veränderlichen Benutzerzustand in einem statischen Feld speichern.



Typische Fehler

Allokationen messen und die App für leckfrei erklären, ohne je die Retention geprüft zu haben. Garbage Collections erzwingen, bis die Zahlen gut aussehen, statt die Paths to Root zu lesen. Einen `DbContext`, eine `DataTable` oder einen Bild-Cache für immer in der Session halten. Aus einer Page ein statisches Event abonnieren und nie abmelden. Controls aus einem Timer-Callback aktualisieren, ohne `IsDisposed` zu prüfen.



Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ mit .NET Object Allocation allokationsintensive Codepfade während eines Szenarios finden.
- ✓ mit Memory Usage-Snapshots den Heap-Zustand vor und nach dem Schließen von Wisej.NET-UI-Objekten vergleichen.
- ✓ Retention-Roots identifizieren, die durch statische Felder, Events, Timer, Caches und langlebige Services entstehen.
- ✓ ein Speicherbudget pro Session und eine Dispose-Checkliste erstellen.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



Praktisches Lab

Gehen Sie in WisejPerfLab beide Hälften des Speicherproblems an. Führen Sie das Szenario Tickets suchen unter .NET Object Allocation aus, finden Sie die pro Suche allokierten Zeilenmodelle und Strings und reduzieren Sie sie, indem Sie einmal projizieren, statt Objekte bei jedem Neuzeichnen neu aufzubauen. Erstellen Sie danach nach dem Aufwärmen einen Memory Usage-Snapshot, öffnen und schließen Sie das Ticket-Detailformular fünfzigmal, erstellen Sie einen zweiten Snapshot und vergleichen Sie: Die gehaltenen Formulare werden von einem statischen GlobalTicketBus.TicketChanged-Abonnement und einem nie gestoppten Aktualisierungs-Timer verankert. Melden Sie das Abonnement ab und stoppen Sie den Timer in einem überschriebenen Dispose, leeren Sie die Datenquelle der BindingSource, führen Sie dann das Öffnen/Schließen-Szenario erneut aus und zeigen Sie, dass der gehaltene Heap wieder nahe an Snapshot A liegt.

Ausgangspunkt

WisejPerfLab nach Modul 3: Die Dashboard-Aktualisierung liegt innerhalb ihres CPU-Budgets, das Ticket-Grid bindet noch vollständige Entitäten, und das Ticket-Detailformular abonniert in seinem Konstruktor `GlobalTicketBus.TicketChanged` und startet einen `System.Timers.Timer`, um seinen Kopfbereich zu aktualisieren. Keines von beiden wird irgendwo rückgängig gemacht. Das Budget aus Modul 1 enthält bereits ein Ziel für den gehaltenen Speicher, das die App nicht erreicht.

Vorgeschlagenes Vorgehen

- ✓ Führen Sie die Ticketsuche einmal unter *.NET Object Allocation* in Release aus. Finden Sie mit der Typansicht heraus, was dominiert, und mit der Allokations-Aufrufstruktur, wer es erzeugt, und notieren Sie die insgesamt allokierten Bytes sowie die Instanzanzahl des obersten Typs.
- ✓ Projizieren Sie einmal in einen schlanken Zeilentyp und binden Sie diesen, sodass die Zeilenobjekte und ihre Anzeigetexte nur einmal pro Suche erzeugt und nicht bei jedem Neuzeichnen neu aufgebaut werden. Führen Sie den Allokations-Trace für das identische Szenario erneut aus.
- ✓ Wechseln Sie zu *Memory Usage*. Wärmen Sie die App auf, erstellen Sie Snapshot A, öffnen und schließen Sie dann das Ticket-Detailformular fünfzigmal, damit aus einer einzelnen geleakten Instanz fünfzig werden, bringen Sie die App in einen echten Leerlaufzustand zurück und erstellen Sie Snapshot B.
- ✓ Vergleichen Sie die Snapshots, wählen Sie den Formulartyp aus, dessen Instanzanzahl gewachsen ist, und lesen Sie den Path to Root für zwei Instanzen. Halten Sie beide Roots fest: das statische Event-Abonnement und den laufenden Timer, dessen Callback das Formular erfasst hat.
- ✓ Überschreiben Sie `Dispose(bool)` im Formular, um das statische Event abzumelden, den Timer zu stoppen und zu verwerfen, die Datenquelle der `BindingSource` zu leeren und Bilder zu verwerfen; ergänzen Sie eine `IsDisposed`-Prüfung, bevor der Timer-Callback Controls berührt.

Führen Sie das identische Öffnen/Schließen-Szenario erneut aus und belegen Sie, dass sowohl die Anzahl als auch der Path to Root verschwunden sind.

Geforderte Ergebnisse

- ✓ Allokations-Trace, der den am stärksten allozierenden Typ und Aufrufpfad des Suchszenarios nennt, vorher und nachher
- ✓ Vergleich von Snapshot A/B, der die Anzahl gehaltener Detailformular-Instanzen vor dem Fix zeigt
- ✓ Path-to-Root-Screenshot, der das statische Event-Abonnement und den laufenden Timer identifiziert
- ✓ Dispose-Override, das das statische Event abmeldet, den Timer stoppt und verwirft und das Binding löst
- ✓ Speicherbudget pro Session und eine Dispose-Checkliste, wobei der Snapshot nachher belegt, dass der Retention-Pfad beseitigt ist

Wiederholungsfragen

- ✓ Ihr Allokations-Trace hat sich drastisch verbessert, aber der Snapshot-Vergleich ist unverändert. Welche Art von Problem haben Sie behoben, welche nicht, und wie würde ein Benutzer den Unterschied bemerken?
- ✓ Warum öffnet und schließt das Lab das Formular fünfzigmal statt einmal, und was würden Sie aus einer einzelnen Instanz in Snapshot B schließen?
- ✓ Der Timer-Callback ist eine Zeile lang und setzt nur ein Label. Erklären Sie genau, wie er ein ganzes Formular am Leben hält und was eine `IsDisposed`-Prüfung löst und was nicht.



Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.