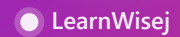


Datenbank, Datei-I/O, Async und externe Wartezeiten

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Die Latenz, die CPU Usage nicht sieht: das Database-Tool für Abfragezeiten und N+1-Muster, File I/O für Import- und Exportarbeit und .NET Async für Ketten, die blockiert oder versehentlich sequenziell sind. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Die Latenz, die CPU Usage nicht sieht: das Database-Tool für Abfragezeiten und N+1-Muster, File I/O für Import- und Exportarbeit und .NET Async für Ketten, die blockiert oder versehentlich sequenziell sind.

1. Das Problem im Produktivbetrieb

Modul 2 hat 1.840 ms verstrichene Zeit gemessen und nur 300 ms CPU-Zeit gefunden. Modul 3 hat den Großteil dieser CPU-Zeit beseitigt, und die Maske ist immer noch nicht schnell, denn die fehlenden 600 ms waren nie Berechnung – sie waren Warten. Das ist die am seltensten richtig diagnostizierte Kategorie in einer Geschäftsanwendung: Der Server ist untätig, der CPU-Graph ist flach, und der Benutzer schaut auf einen Ladeindikator. Irgendwo läuft eine Abfrage einmal pro Zeile, ein Detailformular holt eine Spalte, die niemand anzeigt, oder ein Klick-Handler blockiert auf `.Result`, während ein Thread-Pool-Thread nichts tut. Die Tools, die das sehen, sind Database, File I/O, .NET Async und Instrumentation, und in diesem Modul bekommt die verbleibende Latenz von WisejPerfLab endlich einen Namen.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Database-Tool	Zeichnet die ADO.NET- und EF Core-Abfragen eines Szenarios mit Dauer und Reihenfolge auf, sodass sich jede Abfrage der UI-Aktion zuordnen lässt, die sie ausgelöst hat.
N+1-Abfrage	Eine Abfrage für die Liste, dann eine weitere pro zurückgegebenem Element; mit zehn Zeilen fühlt sich die Liste gut an, bei tausend bricht sie ein – und für CPU-Profiling ist das unsichtbar.
Over-Fetching	Spalten oder Zeilen abfragen, die die Maske nie anzeigt – große Text- und Binärspalten sind die üblichen Verursacher und kosten Abfragezeit, Übertragung und Speicher.
No-Tracking-Abfrage	Eine schreibgeschützte Abfrage ohne Change Tracking, passend für Grids und Listen, die Daten anzeigen statt bearbeiten.
Tool .NET Async	Zeigt Async-Aktivitäten und wo sie warten – so wird eine versehentlich sequenzielle oder blockierte Kette sichtbar.
Blockieren auf Async	<code>.Result</code> oder <code>.Wait()</code> aus einem Handler aufrufen; das hält während des Wartens einen Thread fest und erzeugt unter Last Symptome, die wie Thread-Pool-Starvation aussehen.
Export im Hintergrund	Lange Datei- oder Berichtserzeugung mit <code>Application.StartTask</code> aus dem unmittelbaren Klickpfad herausgelöst, mit Fortschrittsmeldungen in sinnvollen Abständen statt bei jedem Schritt.

3. Der rote Faden der Lektion

Abfragen den Klicks zuordnen

Führen Sie ein Szenario unter dem Database-Tool aus und lesen Sie die Abfrageliste wie eine Geschichte. Das Laden eines Grids sollte nicht eine Abfrage pro angezeigter Zeile absetzen. Ein Lookup-Editor sollte unveränderte Referenzdaten nicht bei jeder Interaktion neu laden. Ein Detailformular sollte keine große Beschreibungs- oder Anhangsspalte abfragen, wenn die Maske eine Zusammenfassung zeigt. Jedes dieser Probleme ist offensichtlich, sobald die Abfragen neben der Aktion stehen, die sie erzeugt hat – und im Code-Review nahezu unmöglich zu entdecken.

WisejPerfLab hat zwei der drei. Das Ticket-Grid setzt pro Zeile eine Kundenabfrage ab, sodass die Suche mit zweihundert Ergebniszeilen zweihundertundeine Abfrage erzeugt; und die Detailabfrage holt jede Spalte des Tickets einschließlich des vollständigen Gesprächsverlaufs. Der kombinierte Fix ist eine einzige No-Tracking-Projektion, die genau die Spalten des Grids auswählt, den Kundennamen per Join holt, statt ihn per Lazy Loading nachzuladen, und das Ergebnis sortiert und seitenweise liefert. Indizes sind für den Filter und den Join wichtig, und Split Queries sind ein Werkzeug gegen kartesische Explosion – keine Voreinstellung, die man über jede Abfrage streut.

Bei Async geht es um Threads, nicht um Geschwindigkeit

Das zweite Szenario ist der CSV-Export, und er scheitert auf zwei Arten zugleich. Der Klick-Handler ruft einen Async-Service auf und blockiert auf `.Result`, was für die gesamte Erzeugung einen Thread festhält; und der Writer erzeugt viele kleine Schreibvorgänge, statt zu streamen. Bei einem einzelnen Benutzer ist

keines davon fatal. Bei fünfzig gleichzeitigen Sessions sammeln sich blockierte Threads an, und die Symptome sehen nach etwas völlig anderem aus – die ganze App wird langsam, auch Masken, die mit Exporten nichts zu tun haben.

Es lohnt sich, genau zu sein, was Async leistet. Das Warten per await auf eine I/O-Operation gibt den Thread frei, solange gewartet wird, und erhöht damit, wie viele gleichzeitige Requests ein Server tragen kann. Es lässt die Operation nicht schneller enden und verringert keine CPU-Arbeit. `async` ist also die richtige Antwort auf „Alle meine Threads sind blockiert“ und die falsche auf „Diese Methode ist langsam“. Zwei weitere Warnungen: Führen Sie niemals parallele Operationen auf Objekten aus, die nicht threadsicher sind, und halten Sie UI-Updates im Session-Kontext, statt sie von beliebigen Threads aus auszulösen.

```
public async Task<IReadOnlyList<TicketGridRow>> SearchTicketsAsync(TicketFilter f,
    CancellationToken ct)
    => await _db.Tickets
        .AsNoTracking() // schreibgeschützte Liste: kein Change
        Tracking
        .Where(t => t.Status == f.Status)
        .OrderByDescending(t => t.UpdatedAt)
        .Select(t => new TicketGridRow // Projektion, nicht die Entität
        {
            Id = t.Id, Number = t.Number,
            Customer = t.Customer.Name, // per Join, nicht pro Zeile
            Status = t.Status, UpdatedAt = t.UpdatedAt
        })
        .Take(f.PageSize) // die Seite, nicht die Tabelle
        .ToListAsync(ct);
```

Den Export aus dem Klickpfad herauslösen

File I/O-Profiling zeigt die andere Hälfte: wiederholte kleine Lese- und Schreibvorgänge, wo ein Stream genügen würde, und eine Vorlage, die bei jedem Request neu eingelesen wird. Das Muster, das dies behebt, ist das, auf das der Kurs die ganze Zeit hingearbeitet hat. Deaktivieren Sie den Button, starten Sie die Arbeit mit `Application.StartTask`, streamen Sie die Ausgabe, statt sie im Speicher zusammenzusetzen, und pushen Sie Fortschritt bei sinnvollen Schritten – alle zehn Prozent, nicht bei jeder Zeile. Aktivieren Sie den Button wieder und liefern Sie die Datei in einem `finally` aus, und prüfen Sie `IsDisposed`, bevor Sie Controls berühren, denn der Benutzer könnte die Maske bereits verlassen haben.

Caching verdient hier eine Zeile Urteilsvermögen. Exportvorlagen sind geteilt und unveränderlich, sie zu cachen ist also geschenkte Kapazität. Erzeugte Dokumente sind benutzerbezogen und oft vertraulich, sie global zu cachen ist daher sowohl ein Speicherproblem als auch ein Offenlegungsrisiko. Die Unterscheidung ist dieselbe, die Modul 3 und Modul 4 getroffen haben, und für Dateien ändert sie sich nicht.

Belegen, dass eine Wartezeit beseitigt wurde

Der Beleg für dieses Modul ist keine CPU-Zahl. Geben Sie für die Suche die Abfrageanzahl vorher und nachher an, die Dauer der langsamsten Abfrage und die verstrichene Zeit des Szenarios aus der Probe, die

seit Modul 1 im Einsatz ist. Geben Sie für den Export die Wall-Clock-Zeit an, ob der Handler weiterhin einen Thread blockiert, und das I/O-Muster aus dem File I/O-Trace. Führen Sie das Ganze dann, wenn möglich, mit einer realistischen Zahl gleichzeitiger Sessions erneut aus – Warteprobleme sind die Kategorie, die sich unter Parallelität am schlechtesten verhält, und ein Fix, der nur für einen Benutzer trägt, ist kein Fix.



Typische Fehler

Niedrige CPU als Beweis dafür werten, dass die App schnell ist. `.Result` in einem Klick-Handler verwenden und damit Symptome von Thread-Pool-Starvation erzeugen. Beim Laden eines Grids eine Abfrage pro angezeigter Zeile zulassen. Statische Lookup-Daten bei jeder Aktion ungemessen neu laden. Private, benutzerbezogene erzeugte Dokumente global cachen.



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:

- ✓ einen Database-Trace in Visual Studio für ADO.NET- oder EF Core-Abfragen erfassen, die von Wisej.NET-Masken ausgelöst werden.
- ✓ langsame Abfragen, wiederholte Abfragen, N+1-Muster und Over-Fetching finden.
- ✓ mit .NET Async und Instrumentation blockierte oder sequenzielle Async-Abläufe identifizieren.
- ✓ Datei-Import- und -Exportszenarien profilieren und teure I/O aus unmittelbaren UI-Events herauslösen.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Profilieren Sie die beiden wartelastigen Szenarien in WisejPerfLab. Führen Sie die Ticketsuche unter dem Database-Tool aus und ordnen Sie jede Abfrage der UI-Aktion zu: Das Laden des Grids setzt pro angezeigter Zeile eine Kundenabfrage ab, und die Detailabfrage holt große Textspalten, die niemand anzeigt. Ersetzen Sie beide durch eine einzige No-Tracking-Projektionsabfrage, die nur die Grid-Spalten auswählt und das Ergebnis seitenweise liefert. Führen Sie dann den CSV-Export unter File I/O und .NET Async aus, finden Sie den blockierenden .Result-Aufruf im Klick-Handler und die wiederholten kleinen Schreibvorgänge, und verlagern Sie die Erzeugung in einen Application.StartTask-Hintergrund-Workflow, der die Datei streamt und gedrosselten Fortschritt meldet, statt jeden Schritt zu pushen.

Ausgangspunkt

WisejPerfLab nach Modul 5: Das Grid ist virtuell und für die Anzeige projiziert, aber die Abfrage dahinter lädt den Kunden weiterhin pro Zeile per Lazy Loading nach, und das Detailformular fragt weiterhin jede Spalte einschließlich des Gesprächsverlaufs ab. Der CSV-Export-Button ruft

`_exportService.BuildAsync( ... ).Result` auf und schreibt die Datei zeilenweise. Die Probe aus Modul 1 misst weiterhin alle drei Szenarien.

Vorgeschlagenes Vorgehen

- ✓ Erfassen Sie in Release einen Database-Trace der Ticketsuche und listen Sie die Abfragen der Reihe nach neben den UI-Aktionen auf, die sie erzeugt haben. Zählen Sie sie und notieren Sie die Dauer der langsamsten Abfrage sowie die Spalten, die die Detailabfrage auswählt.
- ✓ Ersetzen Sie den Kunden-Lookup pro Zeile und die zu breite Detailabfrage durch eine einzige `AsNoTracking`-Projektion, die genau die Spalten des Grids auswählt, den Kundennamen per Join holt, sortiert und seitenweise liefert. Prüfen Sie, ob Filter- und Join-Spalten indiziert sind, bevor Sie schließen, dass die Abfrage selbst langsam ist.
- ✓ Erfassen Sie einen .NET Async-Trace des Exports und, falls die Wartezeit dann noch unklar ist, einen Instrumentation-Trace, um zu belegen, wo der Klick-Handler blockiert. Bestätigen Sie, dass der `.Result`-Aufruf tatsächlich einen Thread festhält und nicht bloß unschön ist.
- ✓ Verlagern Sie die Exporterzeugung in `Application.StartTask`: Button deaktivieren, die Ausgabe streamen, statt sie im Speicher zusammenzusetzen, Fortschritt in sinnvollen Abständen statt pro Zeile pushen, und den Button in einem `finally` nach der Prüfung von `IsDisposed` wieder aktivieren und die Datei ausliefern.
- ✓ Führen Sie beide Szenarien erneut aus und, wenn möglich, den Export mit mehreren gleichzeitigen Sessions – Warteprobleme verhalten sich unter Parallelität am schlechtesten, und eine Verbesserung für einen einzelnen Benutzer übersteht das womöglich nicht.

Geforderte Ergebnisse

- ✓ Database-Trace, der Abfragen den UI-Aktionen zuordnet, mit der Anzahl der N+1-Abfragen vor dem Fix
- ✓ Eine einzige No-Tracking-Projektionsabfrage mit Paging anstelle der Abfragen pro Zeile und der zu breiten Abfragen
- ✓ File I/O- und .NET Async-Belege für den Export, die den blockierenden Aufruf und das Schreibmuster nennen
- ✓ Export in einen Hintergrund-Task verlagert, der die Ausgabe streamt und Fortschritt nur bei sinnvollen Schritten pusht
- ✓ Abfrageanzahl, Abfragedauer und Wall-Clock-Zeit des Exports vorher und nachher, mit verbleibenden Risiken

Wiederholungsfragen

- ✓ Die Suche setzt nun eine Abfrage statt zweihundertundeiner ab, aber die verstrichene Zeit hat sich nur halbiert. Wo steckt der Rest der Zeit, und mit welchem Tool würden Sie es herausfinden?
- ✓ Ihr Kollege sagt, der Export sei in Ordnung, weil `.Result` „nur einen Thread blockiert“. Erklären Sie, was bei fünfzig gleichzeitigen Exporten passiert und bei welchen Masken der Benutzer eine Verlangsamung bemerken würde.
- ✓ Welche der Dinge, die dieses Lab cacht, lassen sich gefahrlos zwischen Sessions teilen und welche nicht, und nach welcher Regel haben Sie das entschieden?



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.