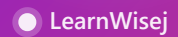


CPU-Hotpaths, Event-Handler und serverseitige UI-Arbeit

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Teurer serverseitiger UI-Code in Event-Handleern, Timern, Layout-Aktualisierungen und Formatierungslogik – und die Caching-, Batching- und View-Model-Muster, die einen Klick wieder günstig machen. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Teurer serverseitiger UI-Code in Event-Handleern, Timern, Layout-Aktualisierungen und Formatierungslogik – und die Caching-, Batching- und View-Model-Muster, die einen Klick wieder günstig machen.

1. Das Problem im Produktivbetrieb

Modul 2 endete mit einem Namen: `TicketFormatter.FormatRow`, 68,9 % eigene CPU-Zeit, fünfzigtausendmal pro Klick aufgerufen. Das ist die Form fast jedes CPU-Problems in einer serverseitigen UI. Niemand hat eine langsame Methode geschrieben; jemand hat eine vernünftige Methode geschrieben und dann dafür gesorgt, dass sie einmal pro Zelle, pro Aktualisierung, für immer läuft. Der zweite Verursacher im WisejPerfLab-Dashboard ist das Spiegelbild: `RebuildKpiPanel` wird nur einmal aufgerufen, verwirft aber ein Panel voller Controls, die sich gar nicht ändern mussten, und erzeugt es neu. Das eine ist ein Problem der Aufrufanzahl, das andere ein Problem der Arbeit pro Aufruf – und in diesem Modul geht es darum, zu erkennen, welches Sie vor sich haben, und es zu beheben, ohne zu raten.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Kosten eines Event-Handlers	Ein Klick, ein Auswahlwechsel oder ein Timer-Tick läuft auf dem Server und blockiert das nächste Event dieser Session. Die Menge an Logik, die der Klick auslöst, ist also die Zeit, die der Benutzer eingefroren ist.
Flammendiagramm	Dieselben Daten wie die Aufrufstruktur, gezeichnet als Balken mit proportionaler Breite. Ein einzelner breiter Frame – eine Funktion, die das Szenario auffrisst – fällt so auf den ersten Blick auf.
eigene CPU-Zeit vs. Aufrufanzahl	Hohe eigene CPU-Zeit bei wenigen Aufrufen heißt, dass der Rumpf teuer ist; geringe Kosten pro Aufruf bei enormer Aufrufanzahl heißen, dass der Fix darin besteht, seltener aufzurufen, nicht mikrozuoptimieren.
vorberechnetes View-Model	Ein kleines Objekt mit anzeigefertigen Strings und Werten, einmal von einem Service erstellt, sodass die UI-Schicht Werte zuweist, statt sie zu formatieren.
gebündelte UI-Änderung	Die Eigenschaften, die der Benutzer sehen muss, in einem Durchgang mit vorbereiteten Werten setzen, statt Controls in einer Schleife zu ändern, die nebenbei auch noch arbeitet.
Control-Neuaufbau	Controls verwerfen und neu erzeugen, um sie zu aktualisieren; dabei laufen Konstruktion, Layout und Theming erneut, und das ist fast immer teurer, als die vorhandenen Controls zu aktualisieren.
unveränderlicher Lookup-Cache	Referenzdaten, die anwendungsweit geteilt und nie verändert werden; sie lassen sich gefahrlos global cachen – anders als benutzerbezogener Zustand, der in der Session bleiben muss.

3. Der rote Faden der Lektion

Im Flammendiagramm den Code finden, auf den es ankommt

Beginnen Sie dort, wo Modul 2 Sie zurückgelassen hat: bei einem CPU Usage-Trace mit auf den Klick eingegrenzter Timeline. Wechseln Sie zum Flammendiagramm und achten Sie auf Breite. Ein einzelner breiter Frame ist eine Funktion, die das Szenario dominiert; ein breiter Frame aus Hunderten dünner, identischer Scheiben ist dieselbe Funktion, immer wieder aufgerufen. Bestätigen Sie das dann mit *Show Hot Path* in der Aufrufstruktur und finden Sie mit Caller/Callee heraus, wer die Aufrufe auslöst.

Die Regeln zur Interpretation sind kurz. Hohe eigene CPU-Zeit bedeutet, dass der Funktionsrumpf selbst teuer ist – suchen Sie darin nach String-Aufbau, Reflection, Serialisierung, Bildkonvertierung oder einer Sortierung. Hohe Gesamtzeit bei niedriger eigener Zeit bedeutet, dass die Kosten in den aufgerufenen Funktionen liegen – steigen Sie weiter ab. Und wenn Instrumentation eine enorme Aufrufanzahl bei winziger Zeit pro Aufruf zeigt, hören Sie auf, den Rumpf zu optimieren: Die lohnende Änderung liegt weiter oben, dort, wo die Schleife ist.

In WisejPerfLab zeigt der Trace beide Muster im selben Handler, weshalb das Modul beide behebt, bevor erneut gemessen wird. Nur eines zu beheben und den Sieg zu erklären ist der sichere Weg zu einer

Optimierung, die sich nicht belegen lässt.

Warum UI-Events teuer werden

Der Katalog der CPU-Hotpaths in einer Wisej.NET-App ist kurz und wiederholt sich: jede Grid-Zelle bei jeder Aktualisierung formatieren; ganze Control-Bäume neu aufbauen, statt Werte zu aktualisieren; Bilder wiederholt konvertieren oder skalieren; JSON oder XML in der UI-Schicht parsen; Reflection in Schleifen; große In-Memory-Collections bei jedem Event sortieren und filtern; Theme- oder Layout-Werte neu berechnen; und einen Export oder Bericht synchron im Klickpfad vorbereiten.

Sie haben eine gemeinsame Ursache. Jedes Mal handelt es sich um Code, der einmal ausgeführt völlig in Ordnung wäre, aber an einer Stelle sitzt, die pro Zeile, pro Zelle oder pro Tick läuft. Ein Formatter, der 21 Mikrosekunden kostet, ist unproblematisch; derselbe Formatter für fünfzigtausend Zeilen ist eine Sekunde eingefrorene Session. Und weil eine Wisej.NET-Session ihre Events nacheinander verarbeitet, ist diese Sekunde nicht nur das Problem des Dashboards – sie verzögert alles, was der Benutzer als Nächstes tut, und belegt einen Kern, um den die anderen Sessions auf der Instanz konkurrieren.

Gecachte View-Models und gebündelte Updates

Der Fix im Lab hat zwei Hälften. Erstens wandert die Formatierung aus der UI heraus.

`DashboardService.GetSnapshot()` gibt einen `DashboardSnapshot` zurück, dessen Eigenschaften bereits Strings sind: `OpenTicketsText`, `OverdueTicketsText`, `AverageAgeText` sowie eine kleine Liste von Diagrammzeilen. Alles Teure – die Aggregation, die kulturabhängige Formatierung, die Altersberechnung – passiert einmal, in einem Service, der sich eigenständig testen und profilieren lässt.

Zweitens wird der Handler zu einer Folge von Zuweisungen bereits vorbereiteter Werte. Er deaktiviert den Button, ändert die Labels und die Datenquelle des Diagramms einmal und aktiviert den Button in einem `finally` wieder. Nichts wird formatiert, nichts neu erzeugt, und der Probe-Scope aus Modul 1 umschließt weiterhin das Ganze, sodass die Änderung in derselben Logzeile sichtbar wird, die Sie seit der Baseline vergleichen.

```
private void btnRefresh_Click(object sender, EventArgs e)
{
    using (_probe.Measure("Dashboard", "Refresh"))
    {
        btnRefresh.Enabled = false;
        try
        {
            var snapshot = _dashboardService.GetSnapshot(); // einmal formatiert
            lblOpen.Text = snapshot.OpenTicketsText; // zuweisen, nicht formatieren
            lblOverdue.Text = snapshot.OverdueTicketsText;
            lblAvgAge.Text = snapshot.AverageAgeText;
            chartLoad.DataSource = snapshot.ChartRows;
        }
        finally { btnRefresh.Enabled = true; }
    }
}
```

Zwei Warnungen gehören zu diesem Muster. Cachen Sie unveränderliche Lookup-Daten – Statusnamen, Prioritätsbezeichnungen, SLA-Tabellen – auf Anwendungsebene, aber legen Sie niemals veränderlichen

benutzerbezogenen Zustand in ein statisches Feld: Auf einem Server mit vielen Sessions ist das ein Korrektheitsfehler, keine Stilfrage. Und „beheben“ Sie CPU-Arbeit nicht, indem Sie die Methode als `async` markieren: `await` verringert keine Berechnung, es verschiebt dieselbe CPU-Arbeit nur auf einen anderen Server-Thread.

Die Änderung mit einem zweiten Trace belegen

Das Ergebnis ist nicht eine schnellere App, sondern eine belastbare Optimierung. Führen Sie das identische Szenario erneut aus – gleiche Datenmenge, gleicher Build, gleiches Aufwärmen – und erfassen Sie einen zweiten CPU Usage-Trace. Gesucht ist ein veränderter Hot Path, nicht bloß eine kleinere Zahl: `FormatRow` sollte von der Spitze der Aufrufstruktur verschwunden sein, die Aufrufanzahl sollte eingebrochen sein, und was nun oben steht, wird zum nächsten Kandidaten – oder zu dem Punkt, an dem Sie aufhören, weil das Budget erreicht ist.

Halten Sie beide Traces in einer Vorher-nachher-Tabelle mit den drei Spalten fest, auf die es ankommt: gesamte CPU-Zeit des Szenarios, oberste Funktion und deren Aufrufanzahl. Haben sich die Zahlen bewegt, der Hot Path aber nicht, haben Sie wahrscheinlich Streuung gemessen. Und notieren Sie das verbleibende Risiko ehrlich – ein gecachter Snapshot bringt ein Zeitfenster veralteter Daten mit sich, und wer Ihre Arbeit prüft, sollte sehen, dass Sie das wissen.



Typische Fehler

Controls bei jeder Aktualisierung Zeile für Zeile formatieren, obwohl ein gecachter Wert genügen würde. `Application.Update` in einer engen CPU-Schleife aufrufen. Hohe Gesamt-CPU-Zeit in Framework-Code für einen Framework-Bug halten, ohne die untergeordneten Aufrufe zu prüfen. Ein CPU-Problem hinter `async`-Schlüsselwörtern verstecken, die dieselbe Arbeit weiterhin auf Server-Threads erledigen. Veränderlichen Benutzerzustand in einem statischen Feld 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:

- ✓ mit CPU Usage, Flammendiagramm und Aufrufstruktur langsame Pfade der Event-Verarbeitung lokalisieren.
- ✓ wiederholte Formatierung, Reflection, Serialisierung, Bildverarbeitung und synchrone Schleifen in UI-Events erkennen.

- ✓ die Arbeit in Event-Handlern mit Caching, Batching, verzögertem Laden und vorberechneten View-Models reduzieren.
- ✓ CPU-gebundene Arbeit von Blockierzeit und Wartezeit auf die Datenbank unterscheiden.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Reparieren Sie den Aktualisieren-Button in WisejPerfLab. Profilieren Sie das Aktualisierungsszenario mit CPU Usage, öffnen Sie das Flammendiagramm und Show Hot Path und identifizieren Sie die beiden Verursacher: einen Formatter, der bei jeder Aktualisierung pro Zeile läuft, und einen Control-Neuaufbau, der das KPI-Panel jedes Mal neu erzeugt. Ersetzen Sie beide durch ein DashboardSnapshot-View-Model, dessen Anzeigetexte einmal im Service berechnet werden, ändern Sie die Labels und die Datenquelle des Diagramms einmal innerhalb eines Probe-Scopes statt Zeile für Zeile, und halten Sie den Button während der Arbeit in einem try/finally deaktiviert. Führen Sie das identische Szenario erneut aus und erfassen Sie den neuen Hot Path.

Ausgangspunkt

Sie öffnen WisejPerfLab mit den Traces aus Modul 2: Die Dashboard-Aktualisierung kostet rund 1.840 ms, `TicketFormatter.FormatRow` hält 68,9 % eigene CPU-Zeit über etwa 50.000 Aufrufe, und `RebuildKpiPanel` verwirft das KPI-Panel bei jedem Klick und erzeugt es neu. Die Service-Schicht gibt derzeit rohe Entitäten zurück, und die Page übernimmt die gesamte Formatierung.

Vorgeschlagenes Vorgehen

- ✓ Reproduzieren Sie zuerst den Befund: Erfassen Sie den CPU Usage-Trace selbst, öffnen Sie das Flammendiagramm und bestätigen Sie über Caller/Callee, welche Schleife die 50.000 Formatter-Aufrufe auslöst. Ändern Sie keinen Code, den Sie nicht in einem Trace gesehen haben.
- ✓ Führen Sie ein `DashboardSnapshot` -View-Model mit angelegten Eigenschaften ein – Text für offene Tickets, Text für überfällige Tickets, Text für das Durchschnittsalter und eine kleine Liste von Diagrammzeilen – und erstellen Sie es einmal in `DashboardService.GetSnapshot()`.
- ✓ Verlagern Sie jeden Formatierungsaufwurf aus der Page in den Service, einschließlich der kulturabhängigen Formatierung von Zahlen und Zeitspannen. Cachen Sie nur unveränderliche Lookup-Daten wie Status- und Prioritätsbezeichnungen; alles Benutzerbezogene bleibt in der Session.
- ✓ Schreiben Sie den Handler als Zuweisungen vorbereiteter Werte innerhalb des vorhandenen Probe-Scopes um: Button deaktivieren, Labels und Datenquelle des Diagramms einmal setzen, in einem `finally` wieder aktivieren. Entfernen Sie den Neuaufbau des KPI-Panels und aktualisieren Sie stattdessen die vorhandenen Controls.
- ✓ Führen Sie das identische Szenario erneut aus – gleiche Datenmenge, gleicher Build, gleiches Aufwärmen –, erfassen Sie einen zweiten CPU Usage-Trace und erstellen Sie die Vorher-nachher-Tabelle aus gesamter CPU-Zeit, oberster Funktion und Aufrufanzahl.

Geforderte Ergebnisse

- ✓ CPU Usage-Trace vorher, der den wiederholten Formatter und den Control-Neuaufbau mit ihrer gesamten und eigenen CPU-Zeit nennt

- ✓ DashboardSnapshot-View-Model mit im Service vorberechneten Anzeigetexten
- ✓ Gebündelter Aktualisierungs-Handler, der die Controls einmal innerhalb eines Probe-Scopes ändert und den Button per try/finally wieder aktiviert
- ✓ CPU Usage-Trace nachher vom identischen Szenario mit dem neuen Hot Path
- ✓ Vorher-nachher-Tabelle mit gesamter CPU-Zeit, oberster Funktion und Aufrufanzahl sowie dem verbleibenden Risiko

Wiederholungsfragen

- ✓ Beide Verursacher waren langsam, aber aus entgegengesetzten Gründen. Erklären Sie, welcher ein Problem der Aufrufanzahl und welcher ein Problem der Arbeit pro Aufruf war und warum die Profiler-Belege jeweils anders aussehen.
- ✓ Ein Kollege schlägt vor, den Aktualisierungs-Handler als `async` zu markieren, um ihn schneller zu machen. Was würde sich tatsächlich ändern, was nicht, und wie würde der Trace das zeigen?
- ✓ Ihr Snapshot wird einmal pro Aktualisierung erstellt. Welches Risiko veralteter Daten oder welches Korrektheitsrisiko entsteht dadurch, und was müssten Sie ändern, wenn derselbe Snapshot über Sessions hinweg gecacht würde?



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.