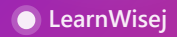


Percorsi critici della CPU, gestori di eventi e lavoro di UI lato server

Guida della lezione · Guida al lab / esame



Guida della lezione

Codice di UI lato server costoso in gestori di eventi, timer, aggiornamenti di layout e logica di formattazione, e i pattern di caching, batching e view model che rendono di nuovo economico un clic. Che cosa significa per uno sviluppatore Wisej.NET che diagnostica i colli di bottiglia in un'app in produzione — e come affrontare questo modulo.



Focus di produzione

Codice di UI lato server costoso in gestori di eventi, timer, aggiornamenti di layout e logica di formattazione, e i pattern di caching, batching e view model che rendono di nuovo economico un clic.

1. Il problema in produzione

Il Modulo 2 si è chiuso con un nome: `TicketFormatter.FormatRow`, 68,9% di CPU propria, chiamato cinquantamila volte in un solo clic. È la forma di quasi ogni problema di CPU in una UI lato server. Nessuno ha scritto un metodo lento; qualcuno ha scritto un metodo ragionevole e poi ha fatto in modo che girasse una volta per cella, a ogni aggiornamento, per sempre. Il secondo colpevole nella dashboard di WisejPerfLab è l'immagine speculare: `RebuildKpiPanel` viene chiamato una volta sola, ma rilascia e ricrea un pannello pieno di controlli che non avevano bisogno di cambiare. Uno è un problema di numero di chiamate e l'altro un problema di lavoro per chiamata, e questo modulo insegna a riconoscere quale dei due hai davanti e a correggerlo senza andare a tentativi.

2. Concetti chiave

Concetto	Perché è importante
costo del gestore di eventi	Un clic, un cambio di selezione o un tick di timer gira sul server e blocca l'evento successivo di quella sessione, quindi la quantità di logica innescata dal clic è la quantità di tempo in cui l'utente resta bloccato.
flame graph	Gli stessi dati dell'albero delle chiamate disegnati come barre di larghezza proporzionale, che rendono evidente a colpo d'occhio un singolo frame largo — una funzione che si mangia lo scenario.
CPU propria e numero di chiamate	Una CPU propria alta con poche chiamate significa che il corpo è costoso; un costo per chiamata basso con un numero di chiamate enorme significa che la correzione è chiamarla meno, non micro-ottimizzarla.
view model precalcolato	Un piccolo oggetto che porta stringhe e valori pronti per la visualizzazione, costruito una sola volta da un servizio, così che il livello di UI assegni valori invece di formattarli.
modifica della UI a batch	Impostare le proprietà che l'utente deve vedere in un'unica passata, con valori già preparati, invece di modificare i controlli dentro un ciclo che fa anche del lavoro.
ricostruzione dei controlli	Rilasciare e ricreare i controlli per aggiornarli; riesegue costruzione, layout e applicazione del tema, ed è quasi sempre più costoso che aggiornare i controlli esistenti.
cache di lookup immutabile	Dati di riferimento condivisi da tutta l'applicazione e mai modificati; sicuri da mettere in una cache globale, a differenza dello stato per utente che deve restare nella sessione.

3. Il filo della lezione

Trovare nel flame graph il codice che conta

Parti da dove ti ha lasciato il Modulo 2: una traccia CPU Usage con la timeline ristretta al clic. Passa al flame graph e cerca la larghezza. Un singolo frame largo è una funzione che domina lo scenario; un frame largo fatto di centinaia di sottili fette identiche è la stessa funzione chiamata più e più volte. Poi usa *Show Hot Path* nell'albero delle chiamate per confermarlo, e caller/callee per scoprire chi guida le chiamate.

Le regole di interpretazione sono brevi. Una CPU propria alta significa che il corpo della funzione in sé è costoso — cercaci dentro costruzione di stringhe, reflection, serializzazione, conversione di immagini o un ordinamento. Una totale alta con una propria bassa significa che il costo sta nei figli — continua a scendere. E se Instrumentation mostra un numero di chiamate enorme con un tempo per chiamata microscopico, smetti di ottimizzare il corpo: la modifica che paga sta più a monte, dove si trova il ciclo.

In WisejPerfLab la traccia mostra entrambi i pattern nello stesso gestore, ed è per questo che il modulo li corregge entrambi prima di rimisurare. Correggerne solo uno e cantare vittoria è il modo in cui un'ottimizzazione finisce per essere indimostrabile.

Perché gli eventi di UI diventano costosi

Il catalogo dei percorsi critici della CPU in un'app Wisej.NET è breve e ripetitivo: formattare ogni cella della griglia a ogni aggiornamento; ricostruire interi alberi di controlli invece di aggiornare i valori; convertire o ridimensionare immagini ripetutamente; fare il parsing di JSON o XML nel livello di UI; reflection dentro i cicli; ordinare e filtrare grandi collezioni in memoria a ogni evento; ricalcolare valori di tema o di layout; e preparare un'esportazione o un report in modo sincrono sul percorso del clic.

Hanno una causa comune. Ognuno è codice che andrebbe benissimo eseguito una volta, messo in un punto che gira per riga, per cella o per tick. Un formatter che costa 21 microsecondi va bene; lo stesso formatter invocato per cinquantamila righe è un secondo di sessione bloccata. E poiché una sessione Wisej.NET elabora i suoi eventi uno alla volta, quel secondo non è solo un problema della dashboard — ritarda qualunque cosa l'utente faccia dopo e consuma un core per cui competono le altre sessioni dell'istanza.

View model in cache e aggiornamenti a batch

La correzione nel lab ha due metà. Primo, la formattazione esce dalla UI.

`DashboardService.GetSnapshot()` restituisce un `DashboardSnapshot` le cui proprietà sono già stringhe: `OpenTicketsText`, `OverdueTicketsText`, `AverageAgeText`, più un piccolo elenco di righe per il grafico. Tutto ciò che è costoso — l'aggregazione, la formattazione sensibile alla cultura, l'aritmetica sulle età — avviene una sola volta, in un servizio che può essere testato e sottoposto a profiling per conto suo.

Secondo, il gestore diventa una sequenza di assegnazioni di valori già preparati. Disabilita il pulsante, modifica le label e la sorgente dati del grafico una sola volta, e lo riabilita in un `finally`. Nulla viene formattato, nulla viene ricreato, e lo scope della sonda del Modulo 1 continua ad avvolgere il tutto, così la modifica compare nella stessa riga di log che confronti fin dalla baseline.

```
private void btnRefresh_Click(object sender, EventArgs e)
{
    using (_probe.Measure("Dashboard", "Refresh"))
    {
        btnRefresh.Enabled = false;
        try
        {
            var snapshot = _dashboardService.GetSnapshot(); // formattato una volta sola
            lblOpen.Text = snapshot.OpenTicketsText; // assegna, non formattare
            lblOverdue.Text = snapshot.OverdueTicketsText;
            lblAvgAge.Text = snapshot.AverageAgeText;
            chartLoad.DataSource = snapshot.ChartRows;
        }
        finally { btnRefresh.Enabled = true; }
    }
}
```

Il pattern porta con sé due avvertenze. Metti in cache a livello di applicazione i dati di lookup immutabili — nomi degli stati, etichette delle priorità, tabelle degli SLA — ma non mettere mai stato per utente modificabile in un campo statico: in un server multi-sessione è un bug di correttezza, non solo una

preferenza di stile. E non "correggere" il lavoro di CPU marcando il metodo come `async`: l'`await` non riduce il calcolo, sposta solo la stessa CPU su un altro thread del server.

Dimostrare la modifica con una seconda traccia

Il deliverable non è un'app più veloce, è un'ottimizzazione difendibile. Riesegui lo scenario identico — stesso dataset, stessa build, stesso riscaldamento — e raccogli una seconda traccia CPU Usage. Quello che cerchi è un percorso critico cambiato, non semplicemente un numero più piccolo: `FormatRow` dovrebbe essere sparito dalla cima dell'albero, il numero di chiamate dovrebbe essere crollato, e qualunque cosa stia ora in cima diventa il candidato successivo oppure il punto in cui ti fermi perché il budget è rispettato.

Registra entrambe le tracce in una tabella prima/dopo con le tre colonne che contano: CPU totale dello scenario, la funzione principale e il suo numero di chiamate. Se i numeri si sono mossi ma il percorso critico è invariato, probabilmente hai misurato varianza. E metti per iscritto con onestà il rischio residuo — uno snapshot in cache introduce una finestra in cui i dati possono essere vecchi, e chi fa la revisione deve vedere che lo sai.



Errori tipici

Formattare i controlli riga per riga a ogni aggiornamento quando basterebbe un valore in cache. Chiamare `Application.Update` dentro un ciclo stretto di CPU. Trattare una CPU totale alta nel codice del framework come un bug del framework senza esaminare le chiamate figlie. Nascondere un problema di CPU dietro parole chiave `async` che fanno comunque lo stesso lavoro sui thread del server. Mettere in cache stato utente modificabile in un campo statico.



La regola delle prove

Definisci lo scenario prima di aprire un profiler, raccogli da una build Release senza debugger, cambia una sola cosa, poi riesegui lo stesso scenario e confronta. Classifica il ritardo in uno dei cinque contenitori — CPU nel tuo codice, blocchi e attese, allocazione e ritenzione, accesso ai dati e I/O, oppure payload degli aggiornamenti — perché ogni contenitore ha il suo strumento di Visual Studio, e un'ottimizzazione che non puoi difendere con una traccia prima e dopo è solo un'ipotesi.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Usare CPU Usage, flame graph e albero delle chiamate per individuare i percorsi lenti di gestione degli eventi.
- ✓ Riconoscere formattazione ripetuta, reflection, serializzazione, lavoro sulle immagini e cicli sincroni dentro gli eventi di UI.
- ✓ Ridurre il lavoro dei gestori di eventi con caching, batching, caricamento lazy e view model precalcolati.
- ✓ Distinguere il lavoro legato alla CPU dal tempo di blocco e dal tempo di attesa del database.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.



Lab pratico

Sistema il pulsante Refresh di WisejPerfLab. Esegui il profiling dello scenario di aggiornamento con CPU Usage, apri il flame graph e Show Hot Path e individua i due colpevoli: un formatter che gira per ogni cella a ogni aggiornamento e una ricostruzione di controlli che ricrea ogni volta il pannello dei KPI. Sostituiscili con un view model DashboardSnapshot le cui stringhe di visualizzazione vengono calcolate una sola volta nel servizio, modifica le label e la sorgente dati del grafico una sola volta dentro uno scope della sonda invece che riga per riga, e tieni il pulsante disabilitato in un try/finally mentre il lavoro è in corso. Riesegui lo scenario identico e cattura il nuovo percorso critico.

Punto di partenza

Apri WisejPerfLab con in mano le tracce del Modulo 2: l'aggiornamento della dashboard costa circa 1.840 ms, `TicketFormatter.FormatRow` detiene il 68,9% di CPU propria su circa 50.000 chiamate, e `RebuildKpiPanel` rilascia e ricrea il pannello dei KPI a ogni clic. Il livello di servizio al momento restituisce entità grezze e la pagina fa tutta la formattazione.

Approccio suggerito

- ✓ Riproduci prima la scoperta: raccogli tu stesso la traccia CPU Usage, apri il flame graph e verifica con caller/callee quale ciclo guida le 50.000 chiamate al formatter. Non modificare codice che non hai visto in una traccia.
- ✓ Introduci un view model `DashboardSnapshot` con proprietà pronte per la visualizzazione — testo del numero di ticket aperti, testo del numero di ticket scaduti, testo dell'età media e un piccolo elenco di righe per il grafico — e costruiscilo una sola volta dentro `DashboardService.GetSnapshot()`.
- ✓ Sposta ogni chiamata di formattazione fuori dalla pagina e dentro il servizio, compresa la formattazione di numeri e durate sensibile alla cultura. Metti in cache solo i dati di lookup immutabili come le etichette di stato e priorità; tutto ciò che è per utente resta nella sessione.
- ✓ Riscrivi il gestore come assegnazioni di valori già preparati dentro lo scope della sonda esistente: disabilita il pulsante, imposta le label e la sorgente dati del grafico una sola volta, riabilita in un `finally`. Elimina la ricostruzione del pannello dei KPI e aggiorna invece i controlli esistenti.
- ✓ Riesegui lo scenario identico — stesso dataset, stessa build, stesso riscaldamento — raccogli una seconda traccia CPU Usage e costruisci la tabella prima/dopo con CPU totale, funzione principale e numero di chiamate.

Deliverable richiesti

- ✓ Traccia CPU Usage prima della modifica che indica il formatter ripetuto e la ricostruzione dei controlli con la loro CPU totale e propria
- ✓ View model DashboardSnapshot con le stringhe di visualizzazione precalcolate nel servizio

- ✓ Gestore di aggiornamento a batch che modifica i controlli una sola volta dentro uno scope della sonda, con riabilitazione in try/finally
- ✓ Traccia CPU Usage dopo la modifica dello scenario identico con il nuovo percorso critico
- ✓ Tabella prima/dopo di CPU totale, funzione principale e numero di chiamate, più il rischio residuo

Domande di revisione per lo studente

- ✓ Entrambi i colpevoli erano lenti, ma per ragioni opposte. Spiega quale era un problema di numero di chiamate e quale un problema di lavoro per chiamata, e perché le prove del profiler per ciascuno appaiono diverse.
- ✓ Un collega propone di marcare il gestore dell'aggiornamento come `async` per renderlo più veloce. Che cosa cambierebbe davvero, che cosa no, e come lo mostrerebbe la traccia?
- ✓ Il tuo snapshot viene costruito una volta per aggiornamento. Quale rischio di dati vecchi o di correttezza introduce, e che cosa dovresti cambiare se lo stesso snapshot fosse messo in cache tra più sessioni?



La regola delle prove

Definisci lo scenario prima di aprire un profiler, raccogli da una build Release senza debugger, cambia una sola cosa, poi riesegui lo stesso scenario e confronta. Classifica il ritardo in uno dei cinque contenitori — CPU nel tuo codice, blocchi e attese, allocazione e ritenzione, accesso ai dati e I/O, oppure payload degli aggiornamenti — perché ogni contenitore ha il suo strumento di Visual Studio, e un'ottimizzazione che non puoi difendere con una traccia prima e dopo è solo un'ipotesi.