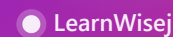


Controlli dati, payload del browser e grandi superfici di UI

Guida della lezione · Guida al lab / esame



Guida della lezione

Griglie, liste, alberi e repeater su larga scala: virtual mode, paginazione, proiezione e caricamento lazy dei nodi, e l'uso delle prove di rete del browser accanto a quelle di Visual Studio per dimensionare il payload degli aggiornamenti. 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

Griglie, liste, alberi e repeater su larga scala: virtual mode, paginazione, proiezione e caricamento lazy dei nodi, e l'uso delle prove di rete del browser accanto a quelle di Visual Studio per dimensionare il payload degli aggiornamenti.

1. Il problema in produzione

Le grandi superfici di UI sono la fonte più comune di problemi di prestazioni in Wisej.NET, e sono il contenitore che il profiler non riesce a vedere del tutto. La griglia dei ticket di WisejPerfLab collega cinquantamila entità con le loro proprietà di navigazione; l'albero dei clienti costruisce tutti i nodi all'avvio. Su un desktop quel pattern era accettabile, perché il controllo e i dati vivevano nello stesso processo della schermata. Qui lo stesso codice paga quattro volte: memoria del server per trattenere gli oggetti, CPU del server per costruire e serializzare lo stato della UI, rete per trasportare l'aggiornamento e tempo del browser per renderlo. Questo modulo è quello in cui la migliore ottimizzazione di solito non è un algoritmo più veloce ma *meno UI che esiste contemporaneamente*.

2. Concetti chiave

Concetto	Perché è importante
riga di proiezione	Un piccolo tipo che contiene solo le colonne visualizzate da una schermata, collegato al posto del grafo delle entità, così né il server né il browser trasportano dati che nessuno guarda.
virtual mode	La griglia mantiene un numero di righe invece di oggetti riga e chiede i valori delle celle quando servono, il che elimina gli oggetti per riga e il rendering delle righe fino a cui nessuno ha scorso.
CellValueNeeded	L'evento che fornisce un valore su richiesta in virtual mode; gira molto spesso, quindi è un percorso critico per costruzione e deve restituire rapidamente.
paginazione e filtri lato server	Chiedere al database la pagina che l'utente sta guardando, invece di caricare tutto e filtrare nel codice di UI.
caricamento lazy dei nodi	Costruire i figli dell'albero quando un ramo viene espanso, con un segnaposto con il conteggio finché è chiuso, invece di materializzare l'intera gerarchia all'avvio.
payload degli aggiornamenti	La differenza serializzata che il server invia dopo un gestore; payload grandi o frequenti costano tempo di rete e del browser che nessuno strumento di Visual Studio riporta.
la compressione non è lavoro gratis	Wisej.NET comprime il traffico HTTP e WebSocket, il che riduce i byte ma non rende economico produrre, trasmettere o rendere un aggiornamento inutile.

3. Il filo della lezione

Griglie: collega meno, poi trattieni meno

La correzione della griglia ha due metà indipendenti, e farne solo una è l'errore abituale. La prima metà è la forma dei dati. Collegare entità trascina proprietà di navigazione, colonne inutilizzate e grandi campi di testo nella memoria del server e in ogni riga che la UI costruisce. Sostituirle con una proiezione `TicketGridRow` — numero, nome del cliente, stato, testo dell'età e nient'altro — riduce il grafo degli oggetti, le allocazioni che hai misurato nel Modulo 4 e l'aggiornamento che riceve il browser.

La seconda metà è quante righe esistono in assoluto. In virtual mode la griglia mantiene un `RowCount` e genera `CellValueNeeded` per le celle che sta effettivamente mostrando, così cinquantamila righe smettono di essere cinquantamila oggetti riga. Il compromesso è esplicito: quell'evento ora gira di continuo, quindi il suo gestore deve essere veloce e thread-safe, e deve appoggiarsi a un servizio di query paginate invece che a una ricerca che interroga il database una volta per cella. Precalcola le stringhe di visualizzazione nella proiezione così che il gestore si limiti a selezionare un campo, e fai il profiling del gestore stesso — il virtual mode sposta il costo, non lo elimina.

Alberi, liste e repeater

Lo stesso principio ridisegna gli altri controlli di grandi dimensioni. Un `TreeView` costruito per intero all'avvio paga per ogni ramo che nessuno apre; caricare i figli all'espansione, con un segnaposto con il conteggio sui rami chiusi, rende il costo proporzionale a ciò che l'utente guarda davvero. Anche liste e

repeater si prestano al trattamento virtuale, e i repeater hanno una trappola in più: un template di scheda con una dozzina di controlli annidati si moltiplica per il numero di elementi, quindi preferisci un riepilogo con un dettaglio su richiesta a un ricco albero di controlli ripetuto mille volte.

Il prefetch è il contrappeso della pigrizia. Uno scorrimento virtuale che recupera esattamente ciò che è a schermo può sembrare fiacco durante uno scorrimento rapido, quindi recuperare un po' oltre l'area visibile di solito vale la pena. È una decisione di ottimizzazione da prendere in base alle misure, non un'impostazione predefinita da applicare ovunque.

Le prove che il profiler non può darti

Visual Studio ti dice che cosa ha fatto il server: CPU per costruire le righe, allocazioni per trattenerle, query per recuperarle. Non dice nulla di ciò che ha attraversato la rete. Per quello lo strumento è il pannello di rete del browser — la dimensione del payload dell'aggiornamento dopo il caricamento di una griglia, i frame WebSocket dopo l'espansione di un albero e il tempo che il browser ha impiegato a renderli.

Due avvertenze. La compressione rende rassicuranti i conteggi dei byte; un aggiornamento che non avrebbe mai dovuto essere inviato resta lavoro sprecato da entrambe le parti, quindi confronta prima e dopo il *numero e la dimensione degli aggiornamenti*, non solo il totale compresso. E anche se disabilitare la compressione WebSocket può aiutarti a leggere i frame durante il debug, non fare benchmark con impostazioni di debug che la produzione non userà.

```
dataGridView1.VirtualMode = true;
dataGridView1.RowCount = _tickets.Count(CurrentFilter); // righe, non oggetti
dataGridView1.CellValueNeeded += (s, e) =>
{
    var row = _tickets.GetProjectedRow(CurrentFilter, e.RowIndex); // paginato, in cache
    e.Value = e.ColumnIndex switch
    {
        0 => row.Number, 1 => row.Customer, 2 => row.Status, 3 => row.AgeText, _ => null
    };
};
```

Regole di ottimizzazione da ricordare

Rendi meno righe, meno nodi e meno controlli. Invia meno aggiornamenti. Collega oggetti più piccoli. Rimanda i dettagli finché l'utente non li chiede. Metti in cache i lookup immutabili; non mettere mai in una cache globale dati per utente modificabili. Preferisci paginazione e filtri lato server al caricare tutto e filtrare nel codice di UI. E non aggiornare un'intera griglia quando è cambiata una sola riga o una sola label — quest'unica abitudine spiega una quota sorprendente del payload nelle dashboard guidate da timer.

Sotto tutte queste regole c'è una domanda di design più che tecnica: a quale domanda risponde questa schermata per l'utente? Una schermata progettata attorno a ogni record possibile sarà sempre costosa da rendere veloce. Una schermata progettata attorno alla decisione che l'utente deve prendere di solito è economica per costruzione, e i numeri prima/dopo del lab sono l'argomento da portare in quella discussione.



Errori tipici

Collegare a ogni riga entità ORM complete con grandi grafi di navigazione. Costruire l'intero albero all'avvio invece di caricare i nodi su richiesta. Aggiornare intere dashboard a ogni tick del timer. Dare per scontato che la compressione risolva i payload sovradimensionati. Scrivere un gestore `CellValueNeeded` che interroga il database una volta per cella.



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:

- ✓ Fare il profiling di scenari con DataGridView, ListView, TreeView e DataRepeater per CPU, allocazioni e costo di rendering.
- ✓ Applicare in modo appropriato virtual mode, paginazione, filtri, proiezione e strategie di prefetch.
- ✓ Ridurre il ricambio del payload WebSocket/HTTP limitando le modifiche alla UI e la dimensione della superficie dei dati.
- ✓ Usare gli strumenti per sviluppatori del browser insieme a Visual Studio per diagnosticare problemi di rete e di dimensione degli aggiornamenti.

Guida al lab / esame

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



Lab pratico

Riduci le due superfici più grandi di WisejPerfLab. La griglia dei ticket collega 50.000 entità complete con le proprietà di navigazione: sostituisci il binding con una proiezione `TicketGridRow` delle sole colonne visibili, imposta `VirtualMode` e `RowCount` e fornisci le celle da un gestore `CellValueNeeded` appoggiato a un servizio di query paginate. L'albero dei clienti costruisce tutti i nodi all'avvio: carica invece i figli all'espansione, usando un segnaposto con il conteggio per i rami chiusi. Misura ogni schermata prima e dopo con CPU Usage e .NET Object Allocation sul server e con il pannello di rete del browser per la dimensione del payload degli aggiornamenti.

Punto di partenza

WisejPerfLab dopo il Modulo 4: le allocazioni durante la ricerca sono calate e il form di dettaglio non perde più memoria, ma la griglia collega ancora l'elenco completo delle entità e l'albero dei clienti viene ancora costruito in un'unica passata all'avvio. Il budget del Modulo 1 ha un obiettivo per l'espansione dell'albero che l'app manca di molto.

Approccio suggerito

- ✓ Misura entrambe le schermate prima di toccarle: CPU Usage e .NET Object Allocation per il caricamento della griglia e l'espansione dell'albero, più il pannello di rete del browser per il payload degli aggiornamenti prodotto da ciascuno. Annota il numero e la dimensione degli aggiornamenti, non solo il totale compresso.
- ✓ Definisci `TicketGridRow` con le sole colonne visualizzate e le loro stringhe di visualizzazione precalcolate, e fai in modo che il servizio di query restituisca quel tipo. Nulla che abbia proprietà di navigazione o grandi colonne di testo deve arrivare alla UI.
- ✓ Passa la griglia in virtual mode: imposta `VirtualMode`, imposta `RowCount` da una query di conteggio e rispondi a `CellValueNeeded` da una pagina paginata e in cache di righe di proiezione. Tratta quel gestore come un percorso critico — nessuna chiamata al database per cella, nessuna formattazione, nessun lock che blocchi le richieste parallele.
- ✓ Rifai l'albero dei clienti in modo che carichi i figli all'espansione, mostrando un segnaposto con il conteggio sui rami chiusi, e stabilisci in base alle misure se un piccolo prefetch oltre l'area visibile vale i recuperi aggiuntivi.
- ✓ Riesegui entrambi gli scenari sullo stesso dataset e con la stessa build, e componi il confronto: CPU del server, allocazioni e righe trattenute da Visual Studio; numero di aggiornamenti e dimensione del payload dal browser.

Deliverable richiesti

- ✓ Proiezione `TicketGridRow` collegata al posto delle entità complete, con le sole colonne visualizzate

- ✓ Griglia in virtual mode con RowCount e un gestore CellValueNeeded servito da un servizio di query paginate
- ✓ Albero dei clienti che carica i nodi figli all'espansione con un segnaposto con il conteggio per i rami chiusi
- ✓ Numeri di CPU e allocazioni prima/dopo per gli scenari di caricamento della griglia ed espansione dell'albero
- ✓ Prove di rete del browser che confrontano la dimensione del payload degli aggiornamenti prima e dopo, con l'avvertenza sulla compressione

Domande di revisione per lo studente

- ✓ Il virtual mode ha eliminato gli oggetti riga ma la tua traccia di CPU è migliorata appena. Che cosa ti dice questo su dove si trovava davvero il costo, e quale misura guarderesti dopo?
- ✓ Il tuo gestore `CellValueNeeded` è corretto ma lento durante uno scorrimento rapido. Elenca tre cose che potrebbe fare e che un gestore per cella non deve mai fare.
- ✓ Il payload compresso dell'espansione dell'albero è cambiato appena dopo il caricamento lazy, ma gli utenti dicono che la schermata sembra più veloce. Fornisci una spiegazione plausibile che il solo conteggio dei byte nasconderebbe.



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.