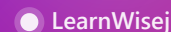


Database, I/O su file, async e attese esterne

Guida della lezione · Guida al lab / esame



Guida della lezione

La latenza che CPU Usage non vede: lo strumento Database per i tempi delle query e i pattern N+1, File I/O per il lavoro di importazione ed esportazione e .NET Async per le catene bloccate o involontariamente sequenziali. 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

La latenza che CPU Usage non vede: lo strumento Database per i tempi delle query e i pattern N+1, File I/O per il lavoro di importazione ed esportazione e .NET Async per le catene bloccate o involontariamente sequenziali.

1. Il problema in produzione

Il Modulo 2 ha misurato 1.840 ms di tempo trascorso e ha trovato solo 300 ms di CPU. Il Modulo 3 ha eliminato la maggior parte di quella CPU e la schermata non è ancora veloce, perché i 600 ms mancanti non sono mai stati calcolo — erano attesa. È la categoria meno diagnosticata in un'applicazione gestionale: il server è inattivo, il grafico della CPU è piatto e l'utente sta guardando un indicatore di caricamento. Da qualche parte una query gira una volta per riga, un form di dettaglio preleva una colonna che nessuno visualizza, oppure un gestore del clic è bloccato su `.Result` mentre un thread del thread pool non fa nulla. Gli strumenti che vedono tutto questo sono Database, File I/O, .NET Async e Instrumentation, e questo modulo è quello in cui la latenza residua di WisejPerfLab riceve finalmente un nome.

2. Concetti chiave

Concetto	Perché è importante
strumento Database	Registra le query ADO.NET ed EF Core lanciate da uno scenario, con le loro durate e il loro ordine, così ogni query può essere abbinata all'azione di UI che l'ha causata.
query N+1	Una query per elencare, poi una in più per ogni elemento restituito; l'elenco sembra a posto con dieci righe e crolla con mille, ed è invisibile al profiling della CPU.
prelievo eccessivo	Selezionare colonne o righe che la schermata non mostra mai — le grandi colonne di testo e binarie sono le colpevoli abituali, e costano tempo di query, trasferimento e memoria.
query no-tracking	Una query in sola lettura che non attiva il change tracking, adatta a griglie ed elenchi che visualizzano dati invece di modificarli.
strumento .NET Async	Mostra le attività async e dove restano in attesa, ed è così che una catena involontariamente sequenziale o bloccata diventa visibile.
bloccare su async	Chiamare <code>.Result</code> o <code>.Wait()</code> da un gestore; trattiene un thread durante l'attesa e, sotto carico, produce sintomi che sembrano esaurimento del thread pool.
esportazione in background	La generazione lunga di file o report spostata fuori dal percorso immediato del clic con <code>Application.StartTask</code> , segnalando l'avanzamento a intervalli significativi invece che a ogni passo.

3. Il filo della lezione

Abbinare le query ai clic

Esegui uno scenario sotto lo strumento Database e leggi l'elenco delle query come un racconto. Il caricamento di una griglia non dovrebbe lanciare una query per ogni riga visualizzata. Un editor di lookup non dovrebbe ricaricare dati di riferimento invariati a ogni interazione. Un form di dettaglio non dovrebbe selezionare una grande colonna di descrizione o di allegato quando la schermata mostra un riepilogo. Ognuno di questi casi è ovvio una volta che le query sono elencate accanto all'azione che le ha prodotte, e quasi impossibile da individuare in una code review.

WisejPerfLab ne ha due su tre. La griglia dei ticket lancia una query sul cliente per ogni riga, quindi la ricerca che restituisce duecento righe lancia duecento query; e la query dei dettagli seleziona tutte le colonne del ticket compreso il testo completo della conversazione. La correzione combinata è un'unica proiezione no-tracking che seleziona esattamente le colonne della griglia, unisce il nome del cliente con un join invece di caricarlo in modo lazy, ordina e pagina il risultato. Gli indici contano per il filtro e per il join, e le split query sono uno strumento contro l'esplosione cartesiana — non un'impostazione predefinita da spargere su ogni query.

Async riguarda i thread, non la velocità

Il secondo scenario è l'esportazione CSV, e fallisce in due modi diversi contemporaneamente. Il gestore del clic chiama un servizio async e si blocca su `.Result`, il che trattiene un thread per tutta la generazione; e il writer emette molte piccole scritture invece di scrivere in streaming. Con un solo utente nessuno dei due è

fatale. Con cinquanta sessioni concorrenti, i thread bloccati si accumulano e i sintomi sembrano tutt'altro — l'intera app rallenta, comprese schermate che con le esportazioni non hanno nulla a che fare.

Vale la pena essere precisi su che cosa fa `async`. Fare `await` di un'operazione di I/O libera il thread mentre l'attesa avviene, il che migliora quante richieste concorrenti un server riesce a sostenere. Non fa terminare prima l'operazione, e non riduce il lavoro di CPU. Quindi `async` è la risposta giusta a "i miei thread sono tutti bloccati" e la risposta sbagliata a "questo metodo è lento". Altre due avvertenze: non eseguire mai operazioni parallele su oggetti che non sono thread-safe, e mantieni gli aggiornamenti della UI nel contesto della sessione invece di lancialli da thread qualsiasi.

```
public async Task<IReadOnlyList<TicketGridRow>> SearchTicketsAsync(TicketFilter f,
    CancellationToken ct)
    => await _db.Tickets
        .AsNoTracking() // elenco in sola lettura: niente
change tracking
        .Where(t => t.Status == f.Status)
        .OrderByDescending(t => t.UpdatedAt)
        .Select(t => new TicketGridRow // proiezione, non l'entità
        {
            Id = t.Id, Number = t.Number,
            Customer = t.Customer.Name, // unito con un join, non caricato lazy
per riga
            Status = t.Status, UpdatedAt = t.UpdatedAt
        })
        .Take(f.PageSize) // la pagina, non la tabella
        .ToListAsync(ct);
```

Spostare l'esportazione fuori dal percorso del clic

Il profiling di File I/O mostra l'altra metà: letture e scritture piccole e ripetute dove basterebbe uno stream, e un template riletto a ogni richiesta. Il pattern che lo corregge è quello verso cui il corso sta costruendo da tempo. Disabilita il pulsante, avvia il lavoro con `Application.StartTask`, scrivi l'output in streaming invece di assemblarlo in memoria e invia l'avanzamento nei passi significativi — ogni dieci per cento, non a ogni riga. Riabilita il pulsante e consegna il file in un `finally`, e controlla `IsDisposed` prima di toccare i controlli, perché l'utente potrebbe essere passato a un'altra schermata.

Qui il caching merita una riga di giudizio. I template di esportazione sono condivisi e immutabili, quindi metterli in cache è capacità gratuita. I documenti generati sono per utente e spesso riservati, quindi metterli in una cache globale è sia un problema di memoria sia un rischio di divulgazione. La distinzione è la stessa tracciata dai Moduli 3 e 4, e non cambia per i file.

Dimostrare che un'attesa è stata eliminata

La prova per questo modulo non è un numero di CPU. Per la ricerca, riporta il numero di query prima e dopo, la durata della query più lenta e il tempo trascorso dello scenario dalla sonda in uso dal Modulo 1. Per l'esportazione, riporta il tempo reale, se il gestore blocca ancora un thread e il pattern di I/O dalla traccia File I/O. Poi, se puoi, riesegui con un numero realistico di sessioni concorrenti — i problemi di attesa sono la categoria che si comporta peggio sotto concorrenza, e una correzione che regge solo per un utente non è una correzione.



Errori tipici

Trattare una CPU bassa come prova che l'app è veloce. Usare `.Result` in un gestore del clic e creare sintomi di esaurimento del thread pool. Lasciare che il caricamento di una griglia produca una query per ogni riga visualizzata. Ricaricare dati di lookup statici a ogni azione senza misurare. Mettere in una cache globale documenti generati privati dell'utente.



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:

- ✓ Raccogliere una traccia Database di Visual Studio per le query ADO.NET o EF Core innescate dalle schermate Wisej.NET.
- ✓ Trovare query lente, query ripetute, pattern N+1 e prelievi eccessivi.
- ✓ Usare .NET Async e Instrumentation per individuare flussi async bloccati o sequenziali.
- ✓ Fare il profiling di scenari di importazione/esportazione di file e spostare l'I/O costoso fuori dagli eventi immediati della UI.

Guida al lab / esame

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



Lab pratico

Esegui il profiling dei due scenari dominati dalle attese in WisejPerfLab. Esegui la ricerca dei ticket sotto lo strumento Database e abbina ogni query all'azione di UI: il caricamento della griglia lancia una query sul cliente per ogni riga visualizzata e la query dei dettagli preleva grandi colonne di testo che nessuno mostra. Sostituisci entrambe con un'unica query di proiezione no-tracking che seleziona solo le colonne della griglia e pagina il risultato. Poi esegui l'esportazione CSV sotto File I/O e .NET Async, trova la chiamata bloccante `.Result` nel gestore del clic e le piccole scritture ripetute, e sposta la generazione in un flusso in background con `Application.StartTask` che scrive il file in streaming e segnala l'avanzamento in modo limitato invece di inviare ogni singolo passo.

Punto di partenza

WisejPerfLab dopo il Modulo 5: la griglia è virtuale e proiettata per la visualizzazione, ma la query che la alimenta carica ancora il cliente in modo lazy per ogni riga, e il form di dettaglio seleziona ancora tutte le colonne compreso il testo della conversazione. Il pulsante di esportazione CSV chiama

`_exportService.BuildAsync( ... ).Result` e scrive il file una riga alla volta. La sonda del Modulo 1 misura ancora tutti e tre gli scenari.

Approccio suggerito

- ✓ Raccogli in Release una traccia Database della ricerca dei ticket ed elenca le query in ordine accanto alle azioni di UI che le hanno prodotte. Contale, e annota la durata della più lenta e le colonne selezionate dalla query dei dettagli.
- ✓ Sostituisci la ricerca del cliente per riga e la query dei dettagli che preleva troppo con un'unica proiezione `AsNoTracking` che seleziona esattamente le colonne della griglia, unisce il nome del cliente con un join, ordina e pagina. Verifica che le colonne del filtro e del join siano indicizzate prima di concludere che la query in sé è lenta.
- ✓ Raccogli una traccia `.NET Async` dell'esportazione, e una traccia Instrumentation se l'attesa non è ancora chiara, per dimostrare dove si blocca il gestore del clic. Verifica che la chiamata `.Result` stia trattenendo un thread e non sia solo codice poco ordinato.
- ✓ Sposta la generazione dell'esportazione in `Application.StartTask`: disabilita il pulsante, scrivi l'output in streaming invece di assemblarlo in memoria, invia l'avanzamento a intervalli significativi invece che per riga, e riabilita il pulsante e consegna il file in un `finally` dopo aver controllato `IsDisposed`.
- ✓ Riesegui entrambi gli scenari e, se puoi, riesegui l'esportazione con diverse sessioni concorrenti — i problemi di attesa si comportano peggio sotto concorrenza, e un miglioramento con un solo utente potrebbe non sopravvivere.

Deliverable richiesti

- ✓ Traccia Database che abbina le query alle azioni di UI, con il numero di query N+1 prima della correzione
- ✓ Un'unica query di proiezione no-tracking con paginazione al posto delle query per riga e di quelle che prelevano troppo
- ✓ Prove File I/O e .NET Async per l'esportazione, che indicano la chiamata bloccante e il pattern di scrittura
- ✓ Esportazione spostata in un task in background che scrive in streaming e invia l'avanzamento solo nei passi significativi
- ✓ Numero di query, durata delle query e tempo reale dell'esportazione prima/dopo, con i rischi residui

Domande di revisione per lo studente

- ✓ La ricerca ora lancia una query invece di duecentouno, ma il tempo trascorso si è solo dimezzato. Dove si trova il resto del tempo, e quale strumento useresti per scoprirlo?
- ✓ Un collega dice che l'esportazione va bene perché `.Result` "blocca solo un thread". Spiega che cosa succede con cinquanta esportazioni concorrenti e quali schermate l'utente noterebbe rallentare.
- ✓ Tra le cose che questo lab mette in cache, quale è sicura da condividere tra sessioni e quale no, e quale regola hai usato per decidere?



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.