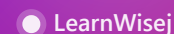


Flusso di profiling in Visual Studio per Wisej.NET

Guida della lezione · Guida al lab / esame



Guida della lezione

Eseguire nel modo giusto il Performance Profiler su un'app Wisej.NET: build Release senza debugger, la matrice di scelta degli strumenti, l'aggancio a dotnet o w3wp e la lettura della timeline di riepilogo, dell'albero delle chiamate, del percorso critico e delle viste caller/callee. 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

Eseguire nel modo giusto il Performance Profiler su un'app Wisej.NET: build Release senza debugger, la matrice di scelta degli strumenti, l'aggancio a dotnet o w3wp e la lettura della timeline di riepilogo, dell'albero delle chiamate, del percorso critico e delle viste caller/callee.

1. Il problema in produzione

Il Modulo 1 ha lasciato WisejPerfLab con un numero e nessuna spiegazione: l'aggiornamento della dashboard impiega 1.840 ms. Il passo successivo ovvio è farne il profiling, ed è proprio qui che la maggior parte delle indagini va storta. Una traccia raccolta sotto il debugger misura il debugger. Una traccia avviata prima che l'app sia riscaldata misura il JIT. Una traccia che copre due minuti di clic misura tutto e non dimostra nulla. Una traccia raccolta con quattro strumenti attivi contemporaneamente misura l'overhead del profiler stesso. Questo modulo è la disciplina meccanica che rende una traccia una prova ammissibile: la build giusta, lo strumento giusto per il contenitore che sospetti, il processo giusto quando l'app non viene avviata da Visual Studio, uno scenario per raccolta e un report che sai leggere.

2. Concetti chiave

Concetto	Perché è importante
Performance Profiler (Alt+F2)	Il percorso di raccolta post-mortem: può avviare il progetto senza debugger, raccogliere un intero scenario e produrre un report salvato, ed è da lì che vengono le prove vere.
finestra Diagnostic Tools	La vista dal vivo, con il debugger agganciato; utile per un'occhiata veloce mentre esegui il codice passo passo, fuorviante come fonte di tempi perché il debugger cambia ciò che fa il runtime.
matrice di scelta degli strumenti	CPU Usage per il calcolo, Instrumentation per numero di chiamate e tempo reale, Object Allocation per il ricambio di oggetti, Memory Usage per la ritenzione, Database per le query, File I/O per i file, .NET Async per le catene di await, .NET Counters per lo stato di salute dal vivo.
.diagsession	Il file di traccia salvato; dargli un nome con modulo, scenario, build, dataset e timestamp è ciò che rende possibile un confronto prima/dopo settimane più tardi.
timeline di riepilogo	Il grafico di CPU o memoria sull'intera raccolta; restringere la selezione all'esatta azione dell'utente è il passo che elimina avvio e riscaldamento dall'albero delle chiamate.
percorso critico	La singola catena di chiamate più costosa nell'intervallo selezionato; Show Hot Path la espande così puoi trovare il tuo codice sotto i frame del framework.
CPU propria e CPU totale	La totale include i figli, la propria è il solo corpo della funzione: una CPU propria alta significa che il metodo in sé è costoso, una totale alta con una propria bassa significa che il costo sta in ciò che chiama.

3. Il filo della lezione

Release, nessun debugger, un solo scenario

Il flusso predefinito ha quattro passi fissi prima che succeda qualcosa di interessante. Passa la configurazione a Release. Avvia l'app ed esegui lo scenario una volta, così avvio e compilazione JIT sono già pagati. Apri *Debug > Performance Profiler* con Alt+F2 e seleziona esattamente uno strumento. Poi avvia la raccolta, esegui solo lo scenario sotto test e fermati subito.

Ogni regola ha il suo motivo. Le build Debug disabilitano le ottimizzazioni, non fanno inlining e possono inventare stack di chiamate che fanno sembrare un banale helper un percorso critico. Il debugger aggiunge il proprio overhead e cambia i tempi. Il riscaldamento conta perché la prima esecuzione di qualsiasi metodo include il tempo del JIT, che è reale ma non è il costo che l'utente percepisce al centesimo aggiornamento. E uno strumento alla volta conta perché la raccolta non è gratuita: eseguire insieme Object Allocation e Instrumentation aggiunge abbastanza overhead da spostare il percorso critico.

L'ultima regola è quella infranta più spesso: *uno scenario per traccia*. Una raccolta che contiene un login, tre aggiornamenti, una ricerca e un po' di scorrimento produce un albero delle chiamate in cui ogni numero interessante è una media di lavoro non correlato. Chiudi le schede del browser in più, fai clic una volta, fermati.

Scegliere lo strumento dal contenitore

I cinque contenitori del Modulo 1 corrispondono quasi uno a uno all'elenco degli strumenti del Performance Profiler, e quella corrispondenza è il senso stesso di questo modulo. Usa per primo *CPU Usage* ogni volta che il sintomo è CPU alta o calcolo lento — campiona gli stack a basso costo e mostra dove è andato il processore. Passa a *Instrumentation* quando ti servono numeri di chiamate esatti o il tempo reale, oppure quando sospetti che il tempo venga speso in attesa invece che in calcolo: misura l'ingresso e l'uscita di ogni funzione strumentata, quindi vede il tempo di blocco che il campionamento perde.

Usa *.NET Object Allocation* quando uno scenario sembra macinare memoria, e gli snapshot di *Memory Usage* quando sospetti che qualcosa resti trattenuto dopo che l'utente l'ha chiuso. Usa *Database* quando le query EF Core o ADO.NET potrebbero dominare, *File I/O* per importazioni, esportazioni e generazione di report, e *.NET Async* quando una catena async è più lenta di quanto suggerirebbe la somma dei suoi await. *.NET Counters* è il caso a parte: è una vista dal vivo di CPU, heap del GC, frequenza di allocazione, frequenza delle eccezioni e stato del thread pool, il che lo rende lo strumento giusto per osservare un'app sotto carico invece che per sezionare un singolo clic.

In WisejPerfLab il lab esegue i primi due. CPU Usage risponde a "dove è andato il processore durante l'aggiornamento"; Instrumentation risponde a "quante volte è stato chiamato, e quanto dei 1.840 ms è stato davvero speso a calcolare". Due tracce, uno scenario e una conclusione che regge a una revisione.

Fare il profiling di un processo che Visual Studio non ha avviato

Quando il progetto di avvio è l'app Wisej.NET, il profiler la lancia per te e non c'è nulla da scegliere. Un hosting simile alla produzione è diverso. Sotto Kestrel l'app è un processo `dotnet`; sotto IIS o IIS Express è `w3wp` o `iisexpress`, uno per application pool. Il Performance Profiler può agganciarsi a un processo in esecuzione, e l'unica vera trappola è agganciarsi a quello sbagliato: su una macchina con diversi siti, verifica l'id del processo e l'application pool, poi invia una richiesta e controlla che la raccolta mostri davvero attività prima di eseguire lo scenario sul serio.

Gli scenari con WebSocket richiedono un'attenzione in più. Se il browser si connette mentre la traccia sta ancora partendo, l'handshake della connessione e il rendering iniziale finiscono nella tua misura. Avvia il browser quando la raccolta è già in corso, oppure registra una traccia di riscaldamento separata e confronta situazioni omogenee.

Leggere il report senza farsi ingannare

Ogni report si apre sulla timeline di riepilogo. Prima di guardare anche un solo nome di funzione, trascina una selezione attorno all'esatto intervallo dell'azione dell'utente e lascia che le viste si ricalcolino; è questo che elimina avvio, caricamento della pagina e tempo di inattività dai numeri. Poi procedi dall'alto verso il basso: l'albero delle chiamate con *Show Hot Path* per la catena dominante, il flame graph per la stessa informazione sotto forma di sagoma, la vista delle funzioni ordinata per CPU propria per i corpi costosi, e caller/callee per scoprire chi chiama una funzione costosa e quanto spesso.

In una traccia Wisej.NET il tuo codice si trova piuttosto in basso nello stack, sotto il dispatch degli eventi di Wisej.NET e i frame di hosting di ASP.NET Core. È normale e non è un bug del framework. La regola è scorrere oltre l'infrastruttura fino al primo frame nel tuo namespace e iniziare a ragionare da lì. Se il frame

del framework sopra il tuo codice è costoso, chiediti che cosa il tuo codice gli sta chiedendo di fare — di solito si scopre che è una modifica ripetuta migliaia di volte, non un framework lento.

```
// Nota della traccia salvata accanto al .diagsession
// Modulo: 02
// Scenario: DashboardRefresh_5000Tickets
// Build: Release, x64, senza debugger, riscaldato
// Hosting: Kestrel, SQL Server locale Browser: Edge
// Strumento: CPU Usage, poi approfondimento con Instrumentation
// Budget: < 800 ms di tempo del gestore sul server
```

Infine, rispetta la soglia di rumore. Una differenza di trenta millisecondi tra due esecuzioni di uno scenario da due secondi non è un miglioramento, è varianza. Ripeti lo scenario, prendi la mediana e rivendica solo un risultato che riesci a riprodurre.



Errori tipici

Fare il profiling dell'avvio quando la lamentela riguarda il clic lento su un pulsante. Attivare tutti gli strumenti insieme e affogare lo scenario nell'overhead della raccolta. Confrontare una traccia Debug con una traccia Release. Lasciare il profiler in esecuzione per una lunga sessione finché nessuno riesce più a trovare l'interazione. Agganciarsi al [w3wp](#) sbagliato e fare il profiling di un application pool inattivo.



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 build Release con Debug > Performance Profiler / Alt+F2.
- ✓ Raccogliere tracce CPU Usage, Instrumentation, .NET Object Allocation, Memory Usage, .NET Counters, Database, File I/O e .NET Async.
- ✓ Agganciarti quando serve a un processo Wisej.NET in esecuzione e individuare il processo corretto.
- ✓ Interpretare albero delle chiamate, percorso critico, caller/callee, elenco delle funzioni, flame graph, albero delle allocazioni e confronti tra snapshot.

Guida al lab / esame

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



Lab pratico

Esegui correttamente il profiling del lento aggiornamento della dashboard di WisejPerfLab. Passa a Release, riscalda l'app una volta, poi raccogli con Alt+F2 una traccia CPU Usage di quell'unico scenario, restringi la timeline di riepilogo al solo clic e usa Show Hot Path per trovare il percorso principale sotto i frame di dispatch di Wisej.NET. Raccogli una seconda traccia Instrumentation dello stesso scenario per ottenere numero di chiamate e tempo reale, stabilisci dalle due tracce se il costo è CPU dentro il tuo codice o tempo passato in attesa, e salva entrambi i file .diagsession con un nome che riporta modulo, scenario, build, dataset e timestamp.

Punto di partenza

Apri WisejPerfLab dal Modulo 1 con i suoi tre scenari strumentati e il suo `Baseline.md`. L'aggiornamento della dashboard registra circa 1.840 ms per 5.000 righe e nessuno sa ancora perché. In questo lab non cambia nulla nell'applicazione: si tratta interamente di raccogliere prove su cui il modulo successivo possa agire.

Approccio suggerito

- ✓ Passa la soluzione a Release, chiudi tutte le schede del browser tranne l'app, riporta i dati di seed al dataset di baseline ed esegui l'aggiornamento una volta senza registrare, così JIT e avvio sono già pagati.
- ✓ Apri *Debug* > *Performance Profiler* con Alt+F2, seleziona *CPU Usage* e nient'altro, avvia la raccolta, fai clic su Refresh esattamente una volta e fermati appena la schermata si stabilizza.
- ✓ Nel report, trascina una selezione sulla timeline di riepilogo attorno al clic stesso, poi apri l'albero delle chiamate, attiva *Show Hot Path* e scorri oltre i frame di dispatch di ASP.NET Core e di Wisej.NET fino alla prima funzione nel tuo namespace. Annota la sua CPU totale e propria, e usa caller/callee per vedere chi la chiama e quanto spesso.
- ✓ Raccogli una seconda traccia dello scenario identico con *Instrumentation*. Confronta numero di chiamate e tempo reale con il quadro della CPU campionata: una funzione con un gran numero di chiamate e un costo minimo per chiamata è un problema diverso da una funzione con un corpo lungo.
- ✓ Stabilisci quale contenitore è supportato dalle prove, scrivi la nota della traccia accanto a ogni `.diagsession` e dai un nome a entrambi i file in modo che modulo, scenario, build, dataset e timestamp siano leggibili senza aprirli.

Deliverable richiesti

- ✓ Traccia CPU Usage in Release dello scenario di aggiornamento della dashboard, con la timeline ristretta al clic
- ✓ Screenshot del percorso critico che indica la principale funzione applicativa e la sua CPU propria rispetto a quella totale

- ✓ Traccia Instrumentation dello stesso scenario con numero di chiamate e tempo reale
- ✓ Nota della traccia che indica corso, modulo, scenario, build, hosting, strumento, browser e budget atteso
- ✓ Un paragrafo sulla causa radice sospetta che dice verso quale dei cinque contenitori puntano le prove

Domande di revisione per lo studente

- ✓ La tua traccia CPU Usage spiega solo 300 ms dei 1.840 ms misurati dalla sonda. Che cosa significa quella differenza, e quale strumento useresti per la raccolta successiva?
- ✓ Un metodo del framework sta in cima al percorso critico con una CPU totale molto alta e quasi nessuna CPU propria. Spiega che cosa significa e che cosa guarderesti invece di aprire un bug sul framework.
- ✓ Perché una seconda persona ha bisogno della nota della traccia per riprodurre il tuo risultato, quando il `.diagsession` contiene già l'intera traccia?



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.