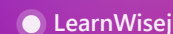


Memoria, allocazioni e perdite di sessione

Guida della lezione · Guida al lab / esame



Guida della lezione

Pressione sulle allocazioni contro memoria trattenuta, il flusso di confronto tra snapshot, i pattern di ritenzione di Wisej.NET (eventi statici, timer, task in background, cache globali) e la checklist di rilascio che dimostra che una perdita è sparita. 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

Pressione sulle allocazioni contro memoria trattenuta, il flusso di confronto tra snapshot, i pattern di ritenzione di Wisej.NET (eventi statici, timer, task in background, cache globali) e la checklist di rilascio che dimostra che una perdita è sparita.

1. Il problema in produzione

La dashboard di WisejPerfLab ora rispetta il suo budget di CPU, eppure l'app cede dopo quattro ore in produzione. I problemi di memoria si comportano diversamente da quelli di CPU: sono invisibili in una demo di due minuti, non si presentano mai come un singolo clic lento, e sono il motivo per cui un server che la mattina regge comodamente ottanta sessioni a metà pomeriggio rifiuta le connessioni. Nel lab ce ne sono due, e non sono lo stesso problema. La ricerca dei ticket alloca molto più del necessario, il che costa tempo di GC e CPU mentre è in esecuzione. Il form di dettaglio del ticket non viene mai davvero rilasciato dopo che l'utente lo chiude, il che costa memoria per sempre. Il primo si trova con una traccia delle allocazioni, il secondo solo con due snapshot e un percorso fino alla radice.

2. Concetti chiave

Concetto	Perché è importante
pressione sulle allocazioni	Quanta memoria crea uno scenario mentre è in esecuzione; una pressione alta aumenta la frequenza del GC e ruba CPU anche quando non c'è nessuna perdita.
memoria trattenuta	Ciò che è ancora raggiungibile quando lo scenario dovrebbe essere finito; è questa che si moltiplica per il numero di sessioni e decide la capacità.
.NET Object Allocation	Lo strumento che registra ogni allocazione gestita durante uno scenario e la raggruppa per tipo e per percorso di chiamata che alloca.
snapshot di Memory Usage	Catture dell'heap prese in momenti scelti; confrontare lo snapshot A e lo snapshot B mostra quali tipi hanno guadagnato istanze mai rilasciate.
percorso fino alla radice	La catena di riferimenti che tiene vivo un oggetto; è la vera prova di una perdita, e l'unico modo per dimostrare che la perdita è sparita.
radice di ritenzione	Ciò che sta in cima a quella catena — un campo statico, un evento statico, un timer in esecuzione, un task in background o una cache a lunga vita.
budget di memoria per sessione	Un numero, in megabyte trattenuti per sessione inattiva, che trasforma "l'app usa molta memoria" in un superato o non superato e più avanti in una soglia di health check.

3. Il filo della lezione

Perché in Wisej.NET la memoria è una questione di capacità

Wisej.NET mantiene vivi gli oggetti di UI lato server per tutta la durata della sessione. È ciò che rende piacevole il modello di programmazione, ed è ciò che rende costosa una perdita. Un form modale mai rilasciato, un `BindingSource` che trattiene ancora cinquantamila righe, un evento statico che inchioda una pagina, un timer che continua a scattare dopo che il pannello è stato nascosto: ciascuno vale pochi megabyte, invisibili sul portatile di uno sviluppatore e decisivi con duecento sessioni.

Il modello mentale da portare nel profiler è che alla memoria sono legate due domande separate. *Quanto ha macinato questo scenario?* riguarda il costo del GC e la CPU. *Che cosa è ancora vivo ora che l'utente ha chiuso tutto?* riguarda la capacità. Uno scenario può essere pessimo nell'una e perfetto nell'altra, quindi un report che risponde solo a una delle due non ha ancora dato il via libera all'app.

Trovare la pressione sulle allocazioni

Per le allocazioni, esegui lo scenario sotto *.NET Object Allocation* in Release, esattamente una volta, e apri le viste delle allocazioni. La vista per tipo risponde a "che cosa viene creato" e l'albero delle chiamate delle allocazioni risponde a "chi lo sta creando". Nella ricerca dei ticket la risposta è poco lusinghiera: gli oggetti modello di riga e le loro stringhe formattate vengono ricostruiti ogni volta che la griglia si ridisegna, quindi una singola ricerca alloca diverse volte i dati che visualizza.

La correzione è la stessa disciplina del Modulo 3, applicata agli oggetti invece che alla CPU: proietta una sola volta in un tipo di riga leggero e passalo alla griglia, invece di costruire nuovi oggetti a ogni ridisegno.

Quello che cerchi nella traccia successiva è un calo sia dei byte allocati in totale sia del numero di istanze del tipo incriminato — e, idealmente, una linea del GC visibilmente più tranquilla.

Trovare la ritenzione

La ritenzione richiede un flusso diverso, e l'ordine dei passi è l'intera tecnica. Avvia l'app in Release e riscaldala. Cattura lo snapshot A. Esegui lo scenario che dovrebbe essere neutro — apri il form di dettaglio del ticket e chiudilo, cinquanta volte, così un'istanza persa diventa cinquanta e non può essere scambiata per rumore. Riporta l'app a un vero punto di inattività. Cattura lo snapshot B e confrontali.

Nel confronto cerchi i tipi il cui numero di istanze è cresciuto e non è più tornato indietro:

`TicketDetailForm` a cinquanta istanze non lascia dubbi. Seleziona il tipo, apri le istanze ed esamina il percorso fino alla radice. In WisejPerfLab compaiono due radici. Una è un evento statico — il form si è sottoscritto a `GlobalTicketBus.TicketChanged` nel costruttore e non ha mai annullato la sottoscrizione, quindi la lista statica dei delegate trattiene ogni form mai aperto. L'altra è un `System.Timers.Timer` che il form avvia per aggiornare la propria intestazione; la sua callback cattura il form, quindi finché il timer è vivo lo è anche il form.

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        GlobalTicketBus.TicketChanged -= OnTicketChanged; // la radice statica
        _refreshTimer?.Stop(); // la radice del timer
        _refreshTimer?.Dispose();
        bindingSource1.DataSource = null; // libera le righe
        _largeImage?.Dispose();
    }
    base.Dispose(disposing);
}
```

Entrambe le correzioni vanno nello stesso punto: un override di `Dispose` che annulla le sottoscrizioni che ha fatto, ferma e rilascia ciò che ha avviato e azzerà i binding che trattengono grandi quantità di dati. Tutto ciò che tocca la UI da un timer o da un task in background dovrebbe inoltre controllare `IsDisposed` prima di aggiornare i controlli, perché la callback può sopravvivere al form di qualche millisecondo anche quando il rilascio è corretto.

Dimostrarlo, e scrivere il budget

Una correzione della memoria non è finita quando il codice sembra giusto. Ripeti lo scenario identico di apertura e chiusura, cattura di nuovo lo snapshot B e mostra due cose: il numero di istanze di `TicketDetailForm` torna al valore di baseline, e il vecchio percorso fino alla radice non esiste più. La seconda è quella importante — un conteggio che scende perché è capitato che partisse una garbage collection non è una prova, e forzare le collection finché i numeri sembrano accettabili è il modo in cui una perdita sopravvive a una revisione.

Concludi con i due documenti di cui ha bisogno il resto del corso. Il budget di memoria per sessione indica quanti megabyte può trattenere una sessione inattiva, misurati sulla tua app invece che stimati a occhio; il Modulo 7 trasforma quel numero nella soglia `maxSessions` dell'health check. La checklist di rilascio è la

versione verificabile della stessa conoscenza: annulla le sottoscrizioni agli eventi, ferma e rilascia i timer, annulla l'elaborazione in background, azzera i binding, rilascia immagini e stream, libera le liste grandi e non memorizzare mai stato utente modificabile in un campo statico.



Errori tipici

Misurare le allocazioni e dichiarare l'app priva di perdite senza mai controllare la ritenzione. Forzare le garbage collection finché i numeri sembrano buoni invece di leggere i percorsi fino alla radice. Tenere per sempre nella sessione un `DbContext`, una `DataTable` o una cache di immagini. Sottoscrivere a un evento statico da una pagina senza mai annullare la sottoscrizione. Aggiornare i controlli dalla callback di un timer senza controllare `IsDisposed`.



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 .NET Object Allocation per trovare i percorsi di codice che allocano molto durante uno scenario.
- ✓ Usare gli snapshot di Memory Usage per confrontare lo stato dell'heap prima e dopo la chiusura di oggetti di UI Wisej.NET.
- ✓ Individuare le radici di ritenzione causate da campi statici, eventi, timer, cache e servizi a lunga vita.
- ✓ Creare un budget di memoria per sessione e una checklist di rilascio.

Guida al lab / esame

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



Lab pratico

Affronta entrambe le metà del problema di memoria in WisejPerfLab. Esegui lo scenario di ricerca dei ticket sotto `.NET Object Allocation` e trova i modelli di riga e le stringhe allocati a ogni ricerca, poi riducili proiettando una volta sola invece di ricostruire gli oggetti a ogni ridisegno. Poi cattura uno snapshot di `Memory Usage` dopo il riscaldamento, apri e chiudi il form di dettaglio del ticket cinquanta volte, cattura un secondo snapshot e confrontali: i form trattenuti hanno come radice una sottoscrizione statica a `GlobalTicketBus.TicketChanged` e un timer di aggiornamento mai fermato. Annulla la sottoscrizione e ferma entrambi in un override di `Dispose`, azzerla la sorgente dati del `BindingSource`, poi riesegui lo scenario di apertura/chiusura e mostra che l'heap trattenuto torna vicino allo snapshot A.

Punto di partenza

WisejPerfLab dopo il Modulo 3: l'aggiornamento della dashboard rientra nel suo budget di CPU, la griglia dei ticket collega ancora entità complete, e il form di dettaglio del ticket si sottoscrive a

`GlobalTicketBus.TicketChanged` nel costruttore e avvia un `System.Timers.Timer` per aggiornare la propria intestazione. Nessuna delle due cose viene annullata da qualche parte. Il budget del Modulo 1 contiene già un obiettivo di memoria trattenuta che l'app non rispetta.

Approccio suggerito

- ✓ Esegui la ricerca dei ticket una volta sotto `.NET Object Allocation` in Release. Usa la vista per tipo per trovare che cosa domina e l'albero delle chiamate delle allocazioni per trovare chi lo crea, e annota i byte allocati in totale più il numero di istanze del tipo principale.
- ✓ Proietta una sola volta in un tipo di riga leggero e collega quello, così gli oggetti riga e le loro stringhe di visualizzazione vengono creati una sola volta per ricerca invece di essere ricostruiti a ogni ridisegno. Riesegui la traccia delle allocazioni sullo scenario identico.
- ✓ Passa a `Memory Usage`. Riscalda l'app, cattura lo snapshot A, poi apri e chiudi il form di dettaglio del ticket cinquanta volte così un'unica istanza persa diventa cinquanta, riporta l'app a un vero punto di inattività e cattura lo snapshot B.
- ✓ Confronta gli snapshot, seleziona il tipo di form il cui numero di istanze è cresciuto e leggi il percorso fino alla radice di due istanze. Cattura entrambe le radici: la sottoscrizione all'evento statico e il timer ancora attivo che trattiene il form catturato dalla callback.
- ✓ Esegui l'override di `Dispose(bool)` sul form per annullare la sottoscrizione all'evento statico, fermare e rilasciare il timer, azzerare la sorgente dati del `BindingSource` e rilasciare le eventuali immagini; aggiungi un controllo `IsDisposed` prima che la callback del timer tocchi i controlli. Riesegui lo scenario identico di apertura/chiusura e dimostra che sono spariti sia il conteggio sia il percorso fino alla radice.

Deliverable richiesti

- ✓ Traccia delle allocazioni che indica il tipo che alloca di più e il percorso di chiamata per lo scenario di ricerca, prima e dopo
- ✓ Confronto tra gli snapshot A/B che mostra il numero di istanze trattenute del form di dettaglio prima della correzione
- ✓ Screenshot del percorso fino alla radice che identifica la sottoscrizione all'evento statico e il timer ancora attivo
- ✓ Override di Dispose che annulla la sottoscrizione all'evento statico, ferma e rilascia il timer e azzerà il binding
- ✓ Budget di memoria per sessione e una checklist di rilascio, con lo snapshot finale che dimostra che il percorso di ritenzione è sparito

Domande di revisione per lo studente

- ✓ La tua traccia delle allocazioni è migliorata in modo drastico ma il confronto tra snapshot è invariato. Che tipo di problema hai corretto, quale no, e come si accorgerebbe della differenza un utente?
- ✓ Perché il lab apre e chiude il form cinquanta volte invece di una, e che cosa concluderesti da una singola istanza che compare nello snapshot B?
- ✓ La callback del timer è lunga una riga e imposta solo una label. Spiega con precisione come tiene in vita un intero form, e che cosa risolve e che cosa non risolve un controllo `IsDisposed`.



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.