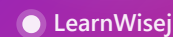


Scalabilità, health check, bilanciamento del carico e ottimizzazione finale

Guida della lezione · Guida al lab / esame



Guida della lezione

Trasformare le misure in un modello di capacità, le soglie di HealthCheck.json, l'affinità di sessione e il bilanciamento del carico compatibile con WebSocket, i contatori di produzione e il report finale sulle prestazioni che giustifica ogni modifica. 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

Trasformare le misure in un modello di capacità, le soglie di HealthCheck.json, l'affinità di sessione e il bilanciamento del carico compatibile con WebSocket, i contatori di produzione e il report finale sulle prestazioni che giustifica ogni modifica.

1. Il problema in produzione

Sei moduli di profiling hanno prodotto un WisejPerfLab più veloce e una cartella di tracce. Niente di tutto questo risponde alla domanda che il team operativo farà il giorno del rilascio: *quanti utenti stanno su un server, e che cosa succede quando uno è pieno?* Il profiling sulla macchina di uno sviluppatore ti dice il costo di uno scenario; la produzione ha bisogno che quel costo venga convertito in un modello di capacità, in soglie che rifiutano nuove sessioni prima che il processo sia in difficoltà, in una forma di deployment che rispetti lo stato di sessione lato server e in un monitoraggio che si accorga se una qualsiasi delle sei correzioni regredisce. Questa traduzione è il contenuto di questo modulo, e il report del capstone è ciò che rende il lavoro verificabile da qualcuno che non ha mai aperto Visual Studio.

2. Concetti chiave

Concetto	Perché è importante
modello di capacità	Memoria trattenuta per sessione inattiva e CPU per scenario comune, moltiplicate per la concorrenza di picco prevista e rapportate alle risorse dell'istanza, con un margine tenuto deliberatamente.
HealthCheck.json	La configurazione con cui un server Wisej.NET può dichiararsi non disponibile in base al numero di sessioni, alla memoria e alla CPU, così che un load balancer smetta di inviargli nuovi utenti.
503 con retry-after	La risposta che un'istanza non in salute restituisce a una <i>prima</i> richiesta; le sessioni esistenti continuano a girare sul loro nodo mentre i nuovi utenti vengono indirizzati altrove.
affinità di sessione	L'instradamento coerente di un client verso l'istanza che contiene il suo stato di sessione, di solito implementato con cookie sticky; senza di essa una sessione Wisej.NET non sopravvive a una seconda richiesta.
bilanciamento compatibile con WebSocket	Una configurazione del load balancer che lascia passare la connessione aggiornata a WebSocket; quando WebSocket non è disponibile l'app ripiega sul polling e il push in tempo reale si perde.
contatori di produzione	Metriche continue e leggere — CPU, heap del GC, frequenza di allocazione, frequenza delle eccezioni, stato del thread pool — che mostrano una regressione senza un profiler agganciato.
report finale sulle prestazioni	Scenari, ambiente, prove di baseline, causa radice, modifica, prove successive, rischi residui, soglie e piano di monitoraggio, in un unico documento verificabile.

3. Il filo della lezione

Da una misura a un numero di sessioni

Il modello di capacità è aritmetica su numeri che hai già. Il Modulo 4 ha prodotto i megabyte trattenuti per sessione inattiva; i Moduli 3 e 6 hanno prodotto il costo di CPU e di attesa per scenario comune. Moltiplica il dato della memoria per la concorrenza che ti aspetti al picco e confrontalo con la memoria che l'istanza può davvero usare, tenendo un margine per picchi, garbage collection e sistema operativo invece di pianificare fino all'ultimo megabyte. Fai lo stesso per la CPU: costo per scenario per la frequenza con cui gli scenari vengono eseguiti al picco.

Qui servono due controlli di onestà. Primo, una sessione inattiva e una attiva non sono la stessa cosa, quindi indica quale hai misurato e come hai pilotato le sessioni. Secondo, un numero ricavato dal portatile di uno sviluppatore ha bisogno di una nota sull'hardware su cui girerà davvero. Un modello di capacità che dichiara le proprie ipotesi può essere corretto; uno che presenta un singolo numero come un dato di fatto no.

Health check con soglie motivate

Gli health check di Wisej.NET permettono a un server di dichiararsi non disponibile in base al numero di sessioni, alla memoria o alla CPU. Il meccanismo è semplice; il valore di questo modulo sta nel fatto che le tue soglie vengono dalle misure invece che da un post di un blog. Se una sessione trattiene 22 MB e l'istanza può dare all'app 4 GB con margine, allora un `maxSessions` di 150 è un numero che puoi difendere — e puoi mostrare il calcolo.

```
{
  "enabled": true,
  "maxMemory": 70,      // % della memoria disponibile
  "maxSessions": 150,   // da 22 MB trattenuti/sessione + margine
  "maxCPU": 85,         // % sostenuta
  "returnCode": 503,
  "retryAfter": 10
}
```

Vale la pena dire con precisione che cosa fa il controllo in pratica, perché viene spesso frainteso: un'istanza oltre i propri limiti restituisce 503 a una richiesta che *avvierebbe* una sessione, così il load balancer manda quell'utente su un altro nodo, mentre le sessioni già presenti sull'istanza continuano normalmente. È un meccanismo di degrado controllato, non un interruttore di emergenza — e non sostituisce la correzione della perdita che sta riempiendo il processo.

Distribuire qualcosa che possiede uno stato

Una sessione Wisej.NET possiede uno stato di UI lato server, e questo decide la forma del deployment. Le richieste di un client devono tornare all'istanza che contiene la sua sessione, quindi il load balancer ha bisogno dell'affinità di sessione — cookie sticky o un altro meccanismo di instradamento coerente. Scalare orizzontalmente senza affinità non fa degradare con eleganza un'app Wisej.NET; la rompe.

Il supporto a WebSocket è il secondo requisito. Il balancer e ogni reverse proxy davanti a esso devono consentire l'upgrade della connessione; dove WebSocket non è disponibile l'app può ancora funzionare tramite polling, ma il push in tempo reale si perde e il pattern degli aggiornamenti cambia. Scrivilo nelle note di deployment invece di scoprirlo durante un incidente. E fai i benchmark con le impostazioni di produzione — opzioni di debug, minificazione e compressione sono tutte diverse, e una misura presa con quelle sbagliate non è confrontabile con le tracce su cui è costruito il report.

Il monitoraggio, e il report che ti sopravvive

Visual Studio serve per le indagini, non per sorvegliare la produzione. Lo stato di salute continuo viene dai contatori .NET e dalle metriche di runtime — CPU, dimensione dell'heap del GC, frequenza di allocazione, frequenza delle eccezioni, stato del thread pool — più i log strutturati a livello applicativo attorno agli scenari che hai strumentato nel Modulo 1. Quella sonda è ancora nel codice, e i suoi record `PERF end` sono il rilevatore di regressioni più economico possibile: se l'aggiornamento della dashboard ricomincia a registrare 1.800 ms, lo saprai prima che un utente lo segnali.

Il report del capstone lega tutto insieme: definizioni degli scenari, ambiente, prove di baseline, spiegazioni delle cause radice, le modifiche, le prove successive, i rischi residui, le soglie degli health check e la checklist di deployment. La sua prova è una domanda che dovresti farti prima di consegnarlo — un collega

che non ha mai aperto il tuo profiler potrebbe riprodurre una misura, capire perché è stata fatta ogni modifica e sapere quale metrica di produzione gli direbbe che il problema è tornato? Se sì, il lavoro di ottimizzazione è finito. Se no, l'app sarà anche più veloce, ma la conoscenza è ancora solo nella tua testa.



Errori tipici

Scalare orizzontalmente senza affinità di sessione. Scegliere `maxSessions` senza misurare la memoria per sessione. Fare benchmark con impostazioni di debug, minificazione o compressione diverse da quelle di produzione. Trattare il bilanciamento del carico come un sostituto della correzione di perdite e scenari lenti. Lasciare le tracce del profiler in esecuzione come se fossero monitoraggio.



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:

- ✓ Stimare dalle misure la memoria per sessione e il costo di CPU per scenario.
- ✓ Configurare gli health check di Wisej.NET con soglie di sessioni massime, memoria massima e CPU massima.
- ✓ Spiegare perché i deployment Wisej.NET richiedono affinità di sessione e un bilanciamento del carico compatibile con WebSocket.
- ✓ Costruire un report finale sulle prestazioni con baseline, correzioni, risultati dei nuovi test, rischi e piano di monitoraggio.

Guida al lab / esame

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



Lab pratico

Consegna il capstone di WisejPerfLab. Ricava un modello di capacità dalle tue misure: memoria trattenuta per sessione inattiva, CPU per scenario comune, concorrenza di picco prevista e il margine che tieni. Scrivi un `HealthCheck.json` i cui `maxSessions`, `maxMemory` e `maxCPU` derivano da quei numeri e spiega ogni soglia, restituendo 503 con un `retry-after` così che un load balancer indirizzi altrove i nuovi utenti mentre le sessioni esistenti continuano a funzionare. Scrivi le note di deployment per due istanze dietro un load balancer con affinità di sessione e supporto WebSocket, e componi il report finale: scenari, ambiente, prove di baseline, cause radice, modifiche, prove successive, rischi residui e i contatori di produzione che mostreranno se una regressione ritorna.

Punto di partenza

WisejPerfLab dopo sei moduli: l'aggiornamento della dashboard, la ricerca dei ticket, l'espansione dell'albero e l'esportazione sono tutti misurabilmente più veloci, il form di dettaglio non perde più memoria e hai prove prima/dopo per ciascuno. Ciò che non esiste ancora è qualcosa su cui un team operativo possa agire — nessun numero di capacità, nessun health check, nessuna nota di deployment e nessun piano di monitoraggio.

Approccio suggerito

- ✓ Raccogli i dati di partenza dai tuoi report: megabyte trattenuti per sessione inattiva dal Modulo 4, costo di CPU e di attesa per scenario dai Moduli 3 e 6, e la concorrenza di picco prevista per l'app. Indica quali misure sono su sessioni inattive e quali su sessioni attive.
- ✓ Costruisci il modello di capacità come aritmetica visibile — memoria per sessione per concorrenza rispetto alla memoria utilizzabile dell'istanza, CPU per scenario per frequenza degli scenari al picco — e tieni un margine esplicito per picchi, garbage collection e sistema operativo invece di pianificare fino all'ultimo megabyte.
- ✓ Scrivi `HealthCheck.json` con `maxSessions`, `maxMemory`, `maxCPU`, il codice di ritorno 503 e un `retry-after`, e scrivi una frase per soglia che spiega la misura da cui deriva. Verifica che un'istanza oltre i propri limiti rifiuti nuove sessioni mentre quelle esistenti continuano.
- ✓ Scrivi le note di deployment per due istanze dietro un load balancer: affinità di sessione e come è implementata, supporto WebSocket attraverso ogni proxy del percorso, che cosa cambia se WebSocket non è disponibile e le impostazioni di produzione che le misure devono usare.
- ✓ Componi il report finale dalle prove dei sei moduli, poi concludi con il piano di monitoraggio: quali contatori, quali soglie, quali alert e quali record di log `PERF` mostrerebbero il ritorno di ciascuna specifica regressione.

Deliverable richiesti

- ✓ Modello di capacità che ricava le sessioni per server dalla memoria per sessione e dalla CPU per scenario misurate

- ✓ HealthCheck.json con maxSessions, maxMemory, maxCPU, codice di ritorno e retry-after, ogni soglia motivata
- ✓ Note di deployment che coprono affinità di sessione, supporto WebSocket e fallback con polling
- ✓ Report finale sulle prestazioni con baseline, causa radice, modifica e prove successive per ogni correzione dei moduli
- ✓ Piano di monitoraggio che indica i contatori, le soglie e gli alert che intercetterebbero ogni regressione

Domande di revisione per lo studente

- ✓ Il tuo health check rifiuta nuove sessioni a 150. Un collega chiede perché non 300, visto che nei test il processo non è andato in crash a 300. Rispondi con i tuoi numeri.
- ✓ Un deployment va in produzione dietro un load balancer senza affinità di sessione. Descrivi esattamente che cosa sperimenta un utente, e perché è un errore di correttezza e non un rallentamento.
- ✓ Fra sei mesi l'aggiornamento della dashboard è di nuovo lento. Quale singolo segnale di produzione te lo direbbe per primo, e che cosa raccoglieresti dopo?



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.