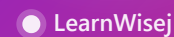


Base de données, E/S fichier, asynchrone et attentes externes

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

La latence que CPU Usage ne voit pas : l'outil Database pour la durée des requêtes et les schémas N+1, File I/O pour les imports et exports, et .NET Async pour les chaînes bloquées ou involontairement séquentielles. Ce que cela signifie pour un développeur Wisej.NET qui diagnostique les goulets d'étranglement d'une application en production — et comment aborder ce module.



Enjeux en production

La latence que CPU Usage ne voit pas : l'outil Database pour la durée des requêtes et les schémas N+1, File I/O pour les imports et exports, et .NET Async pour les chaînes bloquées ou involontairement séquentielles.

1. Le problème en production

Le Module 2 a mesuré 1 840 ms de temps écoulé et n'a trouvé que 300 ms de CPU. Le Module 3 a supprimé l'essentiel de ce CPU et l'écran n'est toujours pas rapide, car les 600 ms manquantes n'ont jamais été du calcul — c'était de l'attente. C'est la catégorie la plus sous-diagnostiquée d'une application métier : le serveur est au repos, la courbe CPU est plate, et l'utilisateur regarde un indicateur de chargement. Quelque part, une requête s'exécute une fois par ligne, un formulaire de détail ramène une colonne que personne n'affiche, ou un gestionnaire de clic bloque sur `.Result` pendant qu'un thread du pool ne fait rien. Les outils qui voient cela sont Database, File I/O, .NET Async et Instrumentation, et c'est dans ce module que la latence restante de WisejPerfLab reçoit enfin un nom.

2. Concepts clés

Concept	Pourquoi c'est important
outil Database	Enregistre les requêtes ADO.NET et EF Core émises par un scénario, avec leur durée et leur ordre, pour que chaque requête puisse être associée à l'action d'interface qui l'a provoquée.
requête N+1	Une requête pour lister, puis une de plus par élément renvoyé ; la liste paraît correcte avec dix lignes et s'effondre à mille, et c'est invisible pour le profilage CPU.
surcharge de données (over-fetching)	Sélectionner des colonnes ou des lignes que l'écran n'affiche jamais — les grandes colonnes de texte et binaires sont les coupables habituels, et elles coûtent du temps de requête, du transfert et de la mémoire.
requête sans suivi	Une requête en lecture seule qui ne met pas en place le suivi des modifications, adaptée aux grilles et aux listes qui affichent des données plutôt qu'elles ne les modifient.
outil .NET Async	Montre les activités asynchrones et l'endroit où elles attendent : c'est ainsi qu'une chaîne involontairement séquentielle ou bloquée devient visible.
blocage sur de l'asynchrone	Appeler <code>.Result</code> ou <code>.Wait()</code> depuis un gestionnaire ; cela monopolise un thread pendant l'attente et, sous charge, produit des symptômes qui ressemblent à une famine du pool de threads.
export en arrière-plan	Une génération longue de fichier ou de rapport sortie du chemin immédiat du clic avec <code>Application.StartTask</code> , qui signale sa progression à des intervalles significatifs plutôt qu'à chaque étape.

3. Déroulé de la leçon

Associer les requêtes aux clics

Exécutez un scénario sous l'outil Database et lisez la liste des requêtes comme un récit. Le chargement d'une grille ne doit pas émettre une requête par ligne affichée. Un éditeur de référence ne doit pas recharger des données de référence inchangées à chaque interaction. Un formulaire de détail ne doit pas sélectionner une grande colonne de description ou de pièce jointe quand l'écran affiche un résumé. Chacun de ces cas saute aux yeux une fois les requêtes listées à côté de l'action qui les a produites, et est presque impossible à repérer lors d'une revue de code.

WisejPerfLab en présente deux sur trois. La grille des tickets émet une requête client par ligne, si bien que la recherche qui renvoie deux cents lignes émet deux cent une requêtes ; et la requête de détail sélectionne toutes les colonnes du ticket, y compris le texte complet de la conversation. La correction combinée est une seule projection sans suivi qui sélectionne exactement les colonnes de la grille, joint le nom du client au lieu de le charger paresseusement, trie et pagine le résultat. Les index comptent pour le filtre et la jointure, et les requêtes fractionnées (split queries) sont un outil contre l'explosion cartésienne — pas un réglage par défaut à saupoudrer sur chaque requête.

L'asynchrone concerne les threads, pas la vitesse

Le second scénario est l'export CSV, et il échoue de deux manières à la fois. Le gestionnaire de clic appelle un service asynchrone et bloque sur `.Result`, ce qui monopolise un thread pendant toute la génération ; et le générateur émet de nombreuses petites écritures au lieu d'écrire en flux. Avec un seul utilisateur, ni l'un ni l'autre n'est fatal. Avec cinquante sessions simultanées, les threads bloqués s'accumulent et les symptômes ressemblent à tout autre chose — toute l'application ralentit, y compris des écrans qui n'ont rien à voir avec les exports.

Il vaut la peine d'être précis sur ce que fait l'asynchrone. Attendre une opération d'E/S avec `await` libère le thread pendant l'attente, ce qui augmente le nombre de requêtes simultanées qu'un serveur peut absorber. Cela ne fait pas finir l'opération plus tôt et ne réduit pas le travail CPU. `async` est donc la bonne réponse à « mes threads sont tous bloqués » et la mauvaise réponse à « cette méthode est lente ». Deux autres mises en garde : n'exécutez jamais d'opérations parallèles sur des objets qui ne sont pas thread-safe, et gardez les mises à jour de l'interface dans le contexte de la session plutôt que de les déclencher depuis des threads quelconques.

```
public async Task<IReadOnlyList<TicketGridRow>> SearchTicketsAsync(TicketFilter f,
    CancellationToken ct)
    => await _db.Tickets
        .AsNoTracking() // read-only list: no change tracking
        .Where(t => t.Status == f.Status)
        .OrderByDescending(t => t.UpdatedAt)
        .Select(t => new TicketGridRow // projection, not the entity
        {
            Id = t.Id, Number = t.Number,
            Customer = t.Customer.Name, // joined, not lazy-loaded per row
            Status = t.Status, UpdatedAt = t.UpdatedAt
        })
        .Take(f.PageSize) // the page, not the table
        .ToListAsync(ct);
```

Sortir l'export du chemin du clic

Le profilage File I/O montre l'autre moitié : des petites lectures et écritures répétées là où un flux suffirait, et un modèle relu à chaque requête. Le schéma qui corrige cela est celui vers lequel le cours tend depuis le début. Désactivez le bouton, lancez le travail avec `Application.StartTask`, écrivez la sortie en flux au lieu de l'assembler en mémoire, et poussez la progression aux étapes significatives — tous les dix pour cent, pas à chaque ligne. Réactivez le bouton et livrez le fichier dans un `finally`, et vérifiez `IsDisposed` avant de toucher aux contrôles, car l'utilisateur a peut-être quitté l'écran.

La mise en cache mérite ici une ligne de discernement. Les modèles d'export sont partagés et immuables : les mettre en cache, c'est de la capacité gratuite. Les documents générés sont propres à chaque utilisateur et souvent confidentiels : les mettre en cache globalement est à la fois un problème de mémoire et un risque de divulgation. La distinction est la même que celle posée aux Modules 3 et 4, et elle ne change pas pour les fichiers.

Prouver qu'une attente a été supprimée

La preuve de ce module n'est pas un chiffre de CPU. Pour la recherche, indiquez le nombre de requêtes avant et après, la durée de la requête la plus lente, et le temps écoulé du scénario d'après la sonde en place depuis le Module 1. Pour l'export, indiquez le temps réel écoulé, si le gestionnaire bloque encore un thread, et le schéma d'E/S issu de la trace File I/O. Puis, si possible, réexécutez avec un nombre réaliste de sessions simultanées — les problèmes d'attente sont la catégorie qui se comporte le plus mal sous simultanéité, et une correction qui ne tient que pour un utilisateur n'en est pas une.



Pièges courants

Considérer un CPU faible comme la preuve que l'application est rapide. Utiliser `.Result` dans un gestionnaire de clic et provoquer des symptômes de famine du pool de threads. Laisser le chargement d'une grille produire une requête par ligne affichée. Recharger des données de référence statiques à chaque action sans mesurer. Mettre en cache globalement des documents générés privés à un utilisateur.



Réflexe de preuve

Définissez le scénario avant d'ouvrir un profileur, collectez à partir d'un build Release sans débogueur, changez une seule chose, puis réexécutez le même scénario et comparez. Classez le délai dans l'une des cinq catégories — CPU dans votre code, blocage et attente, allocation et rétention, accès aux données et E/S, ou charge utile des mises à jour — car chaque catégorie a son propre outil Visual Studio, et une optimisation que vous ne pouvez pas défendre avec une trace avant et après n'est qu'une supposition.



Objectifs d'apprentissage

À la fin de ce module, vous serez capable de :

- ✓ Collecter une trace Database de Visual Studio pour les requêtes ADO.NET ou EF Core déclenchées par des écrans Wisej.NET.
- ✓ Trouver les requêtes lentes, les requêtes répétées, les schémas N+1 et la surcharge de données.
- ✓ Utiliser .NET Async et Instrumentation pour identifier les flux asynchrones bloqués ou séquentiels.
- ✓ Profiler des scénarios d'import/export de fichiers et sortir les E/S coûteuses des événements d'interface immédiats.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.



Lab pratique

Profilez les deux scénarios dominés par l'attente dans WisejPerfLab. Exécutez la recherche de tickets sous l'outil Database et associez chaque requête à l'action d'interface qui l'a produite : le chargement de la grille émet une requête client par ligne affichée et la requête de détail ramène de grandes colonnes de texte que personne n'affiche. Remplacez-les par une seule requête de projection sans suivi qui ne sélectionne que les colonnes de la grille et pagine le résultat. Exécutez ensuite l'export CSV sous File I/O et .NET Async, repérez l'appel bloquant `.Result` dans le gestionnaire de clic et les petites écritures répétées, et déplacez la génération dans un traitement en arrière-plan `Application.StartTask` qui écrit le fichier en flux et signale une progression limitée au lieu de pousser chaque étape.

Point de départ

WisejPerfLab après le Module 5 : la grille est virtuelle et projetée pour l'affichage, mais la requête sous-jacente charge toujours paresseusement le client ligne par ligne, et le formulaire de détail sélectionne toujours toutes les colonnes, y compris le texte de la conversation. Le bouton d'export CSV appelle `_exportService.BuildAsync( ... ).Result` et écrit le fichier ligne par ligne. La sonde du Module 1 mesure toujours les trois scénarios.

Approche suggérée

- ✓ Collectez en Release une trace Database de la recherche de tickets et listez les requêtes dans l'ordre à côté des actions d'interface qui les ont produites. Comptez-les, et notez la durée de la plus lente et les colonnes sélectionnées par la requête de détail.
- ✓ Remplacez la recherche du client ligne par ligne et la requête de détail trop gourmande par une seule projection `AsNoTracking` qui sélectionne exactement les colonnes de la grille, joint le nom du client, trie et pagine. Vérifiez que les colonnes de filtre et de jointure sont indexées avant de conclure que la requête elle-même est lente.
- ✓ Collectez une trace `.NET Async` de l'export, et une trace Instrumentation si l'attente reste floue, pour prouver où bloque le gestionnaire de clic. Confirmez que l'appel `.Result` monopolise réellement un thread, et n'est pas simplement disgracieux.
- ✓ Déplacez la génération de l'export dans `Application.StartTask` : désactivez le bouton, écrivez la sortie en flux au lieu de l'assembler en mémoire, poussez la progression à des intervalles significatifs plutôt qu'à chaque ligne, puis réactivez le bouton et livrez le fichier dans un `finally` après avoir vérifié `IsDisposed`.
- ✓ Réexécutez les deux scénarios et, si possible, réexécutez l'export avec plusieurs sessions simultanées — les problèmes d'attente se comportent au plus mal sous simultanéité, et une amélioration mesurée avec un seul utilisateur peut ne pas y résister.

Livrables attendus

- ✓ Trace Database associant les requêtes aux actions d'interface, avec le nombre de requêtes N+1 avant la correction
- ✓ Requête unique de projection sans suivi avec pagination, remplaçant les requêtes par ligne et celles qui ramènent trop de données
- ✓ Preuves File I/O et .NET Async pour l'export, identifiant l'appel bloquant et le schéma d'écriture
- ✓ Export déplacé dans une tâche en arrière-plan qui écrit en flux et ne pousse la progression qu'aux étapes significatives
- ✓ Nombre de requêtes, durée des requêtes et temps réel de l'export avant/après, avec les risques restants

Questions de révision

- ✓ La recherche émet désormais une requête au lieu de deux cent une, mais le temps écoulé n'a été divisé que par deux. Où se trouve le reste du temps, et quel outil utiliseriez-vous pour le découvrir ?
- ✓ Votre collègue affirme que l'export ne pose pas de problème, car `.Result` « ne bloque qu'un thread ». Expliquez ce qui se passe avec cinquante exports simultanés et quels écrans l'utilisateur verrait ralentir.
- ✓ Parmi les éléments que ce lab met en cache, lesquels peuvent être partagés sans risque entre les sessions et lesquels ne le peuvent pas, et quelle règle avez-vous appliquée pour en décider ?



Réflexe de preuve

Définissez le scénario avant d'ouvrir un profileur, collectez à partir d'un build Release sans débogueur, changez une seule chose, puis réexécutez le même scénario et comparez. Classez le délai dans l'une des cinq catégories — CPU dans votre code, blocage et attente, allocation et rétention, accès aux données et E/S, ou charge utile des mises à jour — car chaque catégorie a son propre outil Visual Studio, et une optimisation que vous ne pouvez pas défendre avec une trace avant et après n'est qu'une supposition.