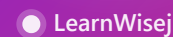


Démarche de profilage Visual Studio pour Wisej.NET

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



Guide de la leçon

Exécuter correctement le Performance Profiler sur une application Wisej.NET : builds Release sans débogueur, matrice de choix des outils, attachement à dotnet ou w3wp, et lecture de la chronologie récapitulative, de l'arbre des appels, du chemin critique et des vues appelants/appelés. 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

Exécuter correctement le Performance Profiler sur une application Wisej.NET : builds Release sans débogueur, matrice de choix des outils, attachement à dotnet ou w3wp, et lecture de la chronologie récapitulative, de l'arbre des appels, du chemin critique et des vues appelants/appelés.

1. Le problème en production

Le Module 1 a laissé WisejPerfLab avec un chiffre et aucune explication : l'actualisation du tableau de bord prend 1 840 ms. L'étape suivante évidente consiste à la profiler, et c'est précisément là que la plupart des investigations dérapent. Une trace collectée sous le débogueur mesure le débogueur. Une trace démarrée avant la mise en température de l'application mesure le JIT. Une trace qui couvre deux minutes de clics mesure tout et ne prouve rien. Une trace collectée avec quatre outils activés à la fois mesure la surcharge du profileur lui-même. Ce module est la discipline mécanique qui fait d'une trace une preuve recevable : le bon build, le bon outil pour la catégorie soupçonnée, le bon processus quand l'application n'est pas lancée par Visual Studio, un seul scénario par collecte, et un rapport que vous savez lire.

2. Concepts clés

Concept	Pourquoi c'est important
Performance Profiler (Alt+F2)	La voie de collecte a posteriori : il peut lancer le projet sans débogueur, collecter un scénario complet et produire un rapport enregistré, et c'est de là que viennent les vraies preuves.
fenêtre Diagnostic Tools	La vue en direct, attachée au débogueur ; utile pour un coup d'œil rapide pendant l'exécution pas à pas, trompeuse comme source de mesures de temps, car le débogueur modifie le comportement du runtime.
matrice de choix des outils	CPU Usage pour le calcul, Instrumentation pour les nombres d'appels et le temps réel, Object Allocation pour le renouvellement de la mémoire, Memory Usage pour la rétention, Database pour les requêtes, File I/O pour les fichiers, .NET Async pour les chaînes d'await, .NET Counters pour la santé en direct.
.diagsession	Le fichier de trace enregistré ; le nommer avec le module, le scénario, le build, le jeu de données et l'horodatage est ce qui rend possible une comparaison avant/après des semaines plus tard.
chronologie récapitulative	Le graphique CPU ou mémoire sur toute la collecte ; restreindre la sélection à l'action exacte de l'utilisateur est l'étape qui retire le démarrage et la mise en température de l'arbre des appels.
chemin critique (hot path)	La chaîne d'appels la plus coûteuse de la plage sélectionnée ; Show Hot Path la déploie pour que vous puissiez trouver votre propre code sous les frames du framework.
CPU propre ou total	Le total inclut les appels enfants, le CPU propre ne concerne que le corps de la fonction : un CPU propre élevé signifie que la méthode elle-même est coûteuse, un total élevé avec un CPU propre faible signifie que le coût se trouve dans ce qu'elle appelle.

3. Déroulé de la leçon

Release, sans débogueur, un seul scénario

La démarche par défaut comporte quatre étapes fixes avant que quoi que ce soit d'intéressant ne se produise. Passez la configuration en Release. Lancez l'application et exécutez le scénario une fois pour que le démarrage et la compilation JIT soient déjà payés. Ouvrez *Debug > Performance Profiler* avec Alt+F2 et cochez exactement un outil. Démarrez ensuite la collecte, effectuez uniquement le scénario testé, et arrêtez sans tarder.

Chaque règle a sa raison d'être. Les builds Debug désactivent les optimisations, n'inlinent rien et peuvent produire des piles d'appels qui font passer un utilitaire trivial pour un chemin critique. Le débogueur ajoute sa propre surcharge et modifie les temps. La mise en température compte parce que la première exécution de toute méthode inclut le temps de JIT, qui est réel mais n'est pas le coût que ressent l'utilisateur à la centième actualisation. Et un seul outil à la fois compte parce que la collecte n'est pas gratuite : exécuter Object Allocation et Instrumentation ensemble ajoute assez de surcharge pour déplacer le chemin critique.

La dernière règle est celle que l'on enfreint le plus souvent : *un seul scénario par trace*. Une collecte qui contient une connexion, trois actualisations, une recherche et un peu de défilement produit un arbre des appels dans lequel chaque chiffre intéressant est une moyenne de travaux sans rapport. Fermez les onglets superflus du navigateur, cliquez une fois, arrêtez.

Choisir l'outil à partir de la catégorie

Les cinq catégories du Module 1 correspondent presque une à une à la liste d'outils du Performance Profiler, et cette correspondance est tout l'objet de ce module. Utilisez d'abord *CPU Usage* chaque fois que le symptôme est un CPU élevé ou un calcul lent — il échantillonne les piles à faible coût et montre où est passé le processeur. Tournez-vous vers *Instrumentation* quand il vous faut des nombres d'appels exacts ou le temps réel écoulé, ou quand vous soupçonnez que le temps est passé à attendre plutôt qu'à calculer : il mesure l'entrée et la sortie de chaque fonction instrumentée, et voit donc le temps bloqué que l'échantillonnage ne voit pas.

Utilisez *.NET Object Allocation* quand un scénario semble faire tourner la mémoire, et les instantanés *Memory Usage* quand vous soupçonnez que quelque chose reste retenu après que l'utilisateur l'a fermé. Utilisez *Database* quand les requêtes EF Core ou ADO.NET risquent de dominer, *File I/O* pour les imports, les exports et la génération de rapports, et *.NET Async* quand une chaîne asynchrone est plus lente que ne le laisserait penser la somme de ses await. *.NET Counters* fait figure d'exception : c'est une vue en direct du CPU, du tas GC, du taux d'allocation, du taux d'exceptions et de l'état du pool de threads, ce qui en fait le bon outil pour observer une application sous charge plutôt que pour disséquer un clic.

Dans WisejPerfLab, le lab utilise les deux premiers. CPU Usage répond à « où est passé le processeur pendant l'actualisation » ; Instrumentation répond à « combien de fois cela a-t-il été appelé, et quelle part des 1 840 ms a réellement servi à calculer ». Deux traces, un scénario, et une conclusion qui résiste à la revue.

Profiler un processus que Visual Studio n'a pas lancé

Quand le projet de démarrage est l'application Wisej.NET, le profileur la lance pour vous et il n'y a rien à choisir. Un hébergement proche de la production est différent. Sous Kestrel, l'application est un processus `dotnet` ; sous IIS ou IIS Express, c'est `w3wp` ou `iisexpress`, un par pool d'applications. Le Performance Profiler peut s'attacher à un processus en cours d'exécution, et le seul vrai piège est de s'attacher au mauvais : sur une machine hébergeant plusieurs sites, vérifiez l'identifiant du processus et le pool d'applications, puis envoyez une requête et contrôlez que la collecte montre bien de l'activité avant d'exécuter le scénario pour de bon.

Les scénarios WebSocket demandent une réflexion supplémentaire. Si le navigateur se connecte pendant que la trace démarre encore, la négociation de connexion et le rendu initial tombent dans votre mesure. Lancez donc le navigateur une fois la collecte en cours, ou enregistrez une trace de mise en température distincte et comparez ce qui est comparable.

Lire le rapport sans se laisser tromper

Chaque rapport s'ouvre sur la chronologie récapitulative. Avant de regarder le moindre nom de fonction, tracez une sélection autour de l'intervalle exact de l'action de l'utilisateur et laissez les vues se recalculer ; c'est ce qui retire des chiffres le démarrage, le chargement de la page et l'inactivité. Travaillez ensuite du haut vers le bas : l'arbre des appels avec *Show Hot Path* pour la chaîne dominante, le flame graph pour la

même information sous forme visuelle, la vue des fonctions triée par CPU propre pour les corps coûteux, et la vue appelants/appelés pour savoir qui appelle une fonction coûteuse et à quelle fréquence.

Dans une trace Wisej.NET, votre propre code se trouve assez bas dans la pile, sous le dispatch d'événements de Wisej.NET et les frames d'hébergement d'ASP.NET Core. C'est normal et ce n'est pas un bug du framework. La règle consiste à dépasser la plomberie jusqu'à la première frame de votre propre espace de noms et à commencer à raisonner à partir de là. Si la frame du framework située au-dessus de votre code est coûteuse, demandez-vous ce que votre code lui demande de faire — le plus souvent, il s'agit d'une modification répétée des milliers de fois plutôt que d'un framework lent.

```
// Trace note saved beside the .diagsession
// Module: 02
// Scenario: DashboardRefresh_5000Tickets
// Build: Release, x64, no debugger, warm
// Hosting: Kestrel, local SQL Server Browser: Edge
// Tool: CPU Usage, then Instrumentation follow-up
// Budget: < 800 ms server handler time
```

Enfin, respectez le bruit de mesure. Un écart de trente millisecondes entre deux exécutions d'un scénario de deux secondes n'est pas une amélioration, c'est de la variance. Répétez le scénario, prenez la médiane et ne revendiquez qu'un gain que vous pouvez reproduire.



Pièges courants

Profiler le démarrage alors que la plainte porte sur un clic de bouton lent. Activer tous les outils à la fois et noyer le scénario sous la surcharge de collecte. Comparer une trace Debug à une trace Release. Laisser le profileur tourner pendant une longue session jusqu'à ce que plus personne ne retrouve l'interaction. S'attacher au mauvais `w3wp` et profiler un pool d'applications inactif.



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 :

- ✓ Profiler des builds Release avec Debug > Performance Profiler / Alt+F2.
- ✓ Collecter des traces CPU Usage, Instrumentation, .NET Object Allocation, Memory Usage, .NET Counters, Database, File I/O et .NET Async.
- ✓ S'attacher à un processus Wisej.NET en cours d'exécution si nécessaire et identifier le bon processus.

- ✓ Interpréter l'arbre des appels, le chemin critique, les vues appelants/appelés, la liste des fonctions, le flame graph, l'arbre d'allocation et les comparaisons d'instantanés.

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 correctement l'actualisation lente du tableau de bord de WisejPerfLab. Passez en Release, faites chauffer l'application une fois, puis collectez avec Alt+F2 une trace CPU Usage de ce seul scénario, restreignez la chronologie récapitulative au clic lui-même et utilisez Show Hot Path pour trouver le chemin principal sous les frames de dispatch de Wisej.NET. Collectez une seconde trace Instrumentation du même scénario pour obtenir le nombre d'appels et le temps réel écoulé, déterminez à partir des deux traces si le coût est du CPU dans votre code ou du temps passé à attendre, et enregistrez les deux fichiers .diagsession sous un nom qui porte le module, le scénario, le build, le jeu de données et l'horodatage.

Point de départ

Vous ouvrez WisejPerfLab issu du Module 1 avec ses trois scénarios instrumentés et son `Baseline.md`. L'actualisation du tableau de bord journalise environ 1 840 ms pour 5 000 lignes et personne ne sait encore pourquoi. Rien ne change dans l'application pendant ce lab : il consiste entièrement à collecter des preuves sur lesquelles le module suivant pourra agir.

Approche suggérée

- ✓ Passez la solution en Release, fermez tous les onglets du navigateur sauf celui de l'application, remettez les données au jeu de données de référence, et exécutez l'actualisation une fois sans enregistrer pour que le JIT et le démarrage soient payés.
- ✓ Ouvrez *Debug > Performance Profiler* avec Alt+F2, cochez *CPU Usage* et rien d'autre, démarrez la collecte, cliquez exactement une fois sur Refresh, et arrêtez dès que l'écran est stabilisé.
- ✓ Dans le rapport, tracez une sélection sur la chronologie récapitulative autour du clic lui-même, puis ouvrez l'arbre des appels, activez *Show Hot Path* et faites défiler au-delà des frames de dispatch d'ASP.NET Core et de Wisej.NET jusqu'à la première fonction de votre propre espace de noms. Notez son CPU total et son CPU propre, et utilisez la vue appelants/appelés pour voir qui l'appelle et à quelle fréquence.
- ✓ Collectez une seconde trace du même scénario avec *Instrumentation*. Comparez les nombres d'appels et le temps réel écoulé avec l'image CPU échantillonnée : une fonction avec un grand nombre d'appels et un coût minuscule par appel est un problème différent d'une fonction au corps long.
- ✓ Décidez quelle catégorie les éléments de preuve soutiennent, rédigez la note de trace à côté de chaque `.diagsession`, et nommez les deux fichiers de sorte que le module, le scénario, le build, le jeu de données et l'horodatage soient lisibles sans les ouvrir.

Livrables attendus

- ✓ Trace CPU Usage en Release du scénario d'actualisation du tableau de bord, chronologie restreinte au clic

- ✓ Capture d'écran du chemin critique indiquant la principale fonction applicative et son CPU propre par rapport à son CPU total
- ✓ Trace Instrumentation du même scénario avec le nombre d'appels et le temps réel écoulé
- ✓ Note de trace indiquant le cours, le module, le scénario, le build, l'hébergement, l'outil, le navigateur et le budget attendu
- ✓ Paragraphe sur la cause racine soupçonnée, indiquant vers laquelle des cinq catégories de coût pointent les éléments de preuve

Questions de révision

- ✓ Votre trace CPU Usage ne représente que 300 ms des 1 840 ms mesurées par la sonde. Que signifie cet écart, et quel outil utiliseriez-vous pour la collecte suivante ?
- ✓ Une méthode du framework se trouve en haut du chemin critique avec un CPU total très élevé et presque aucun CPU propre. Expliquez ce que cela signifie et ce que vous examineriez au lieu de signaler un bug du framework.
- ✓ Pourquoi une autre personne a-t-elle besoin de la note de trace pour reproduire votre résultat, alors que le `.diagsession` contient déjà toute la trace ?



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.