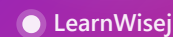


Chemins critiques CPU, gestionnaires d'événements et travail d'interface côté serveur

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



Guide de la leçon

Le code d'interface coûteux côté serveur dans les gestionnaires d'événements, les timers, les actualisations de mise en page et la logique de formatage, ainsi que les modèles de cache, de traitement groupé et de view model qui rendent un clic de nouveau peu coûteux. 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

Le code d'interface coûteux côté serveur dans les gestionnaires d'événements, les timers, les actualisations de mise en page et la logique de formatage, ainsi que les modèles de cache, de traitement groupé et de view model qui rendent un clic de nouveau peu coûteux.

1. Le problème en production

Le Module 2 s'est terminé sur un nom : `TicketFormatter.FormatRow`, 68,9 % de CPU propre, appelée cinquante mille fois en un seul clic. C'est la forme de presque tous les problèmes de CPU dans une interface côté serveur. Personne n'a écrit une méthode lente ; quelqu'un a écrit une méthode raisonnable, puis s'est arrangé pour qu'elle s'exécute une fois par cellule, à chaque actualisation, indéfiniment. Le second coupable du tableau de bord de WisejPerfLab en est le reflet inversé : `RebuildKpiPanel` n'est appelée qu'une fois, mais elle libère et recrée tout un panneau de contrôles qui n'avaient pas besoin de changer. L'un est un problème de nombre d'appels, l'autre un problème de travail par appel, et ce module vous apprend à reconnaître lequel vous avez et à le corriger sans deviner.

2. Concepts clés

Concept	Pourquoi c'est important
coût d'un gestionnaire d'événements	Un clic, un changement de sélection ou un tick de timer s'exécute sur le serveur et bloque l'événement suivant de cette session : la quantité de logique déclenchée par le clic correspond au temps pendant lequel l'utilisateur est figé.
flame graph	Les mêmes données d'arbre des appels dessinées sous forme de barres de largeur proportionnelle, ce qui rend immédiatement visible une frame large isolée — une fonction qui dévore le scénario.
CPU propre ou nombre d'appels	Un CPU propre élevé avec peu d'appels signifie que le corps est coûteux ; un coût par appel faible avec un nombre d'appels énorme signifie qu'il faut l'appeler moins souvent, pas le micro-optimiser.
view model précalculé	Un petit objet contenant des chaînes et des valeurs prêtes à afficher, construit une seule fois par un service, pour que la couche d'interface affecte des valeurs au lieu de les formater.
modification groupée de l'interface	Définir en une seule passe, avec des valeurs préparées, les propriétés que l'utilisateur doit voir, au lieu de modifier des contrôles dans une boucle qui effectue aussi du travail.
reconstruction de contrôles	Libérer et recréer des contrôles pour les actualiser ; cela relance la construction, la mise en page et l'application du thème, et c'est presque toujours plus coûteux que de mettre à jour les contrôles existants.
cache de données de référence immuables	Des données de référence partagées par toute l'application et jamais modifiées ; on peut les mettre en cache globalement sans risque, contrairement à l'état propre à chaque utilisateur, qui doit rester dans la session.

3. Déroulé de la leçon

Trouver le code qui compte dans le flame graph

Partez de là où le Module 2 vous a laissé : une trace CPU Usage dont la chronologie est restreinte au clic. Passez au flame graph et cherchez la largeur. Une frame large isolée est une fonction qui domine le scénario ; une frame large composée de centaines de fines tranches identiques est la même fonction appelée encore et encore. Utilisez ensuite *Show Hot Path* dans l'arbre des appels pour confirmer, puis la vue appelants/appelés pour découvrir qui provoque les appels.

Les règles d'interprétation sont brèves. Un CPU propre élevé signifie que le corps de la fonction lui-même est coûteux — cherchez-y la construction de chaînes, la réflexion, la sérialisation, la conversion d'images ou un tri. Un total élevé avec un CPU propre faible signifie que le coût se trouve dans les enfants — continuez à descendre. Et si Instrumentation montre un nombre d'appels énorme avec un temps par appel microscopique, cessez d'optimiser le corps : la modification rentable se situe en amont, là où se trouve la boucle.

Dans WisejPerfLab, la trace montre les deux schémas dans le même gestionnaire, c'est pourquoi le module corrige les deux avant de mesurer à nouveau. N'en corriger qu'un et crier victoire, c'est ainsi qu'une

optimisation devient impossible à prouver.

Pourquoi les événements d'interface deviennent coûteux

Le catalogue des chemins critiques CPU d'une application Wisej.NET est court et répétitif : formater chaque cellule de la grille à chaque actualisation ; reconstruire des arborescences entières de contrôles au lieu de mettre à jour des valeurs ; convertir ou redimensionner des images à répétition ; analyser du JSON ou du XML dans la couche d'interface ; utiliser la réflexion dans des boucles ; trier et filtrer de grandes collections en mémoire à chaque événement ; recalculer des valeurs de thème ou de mise en page ; et préparer un export ou un rapport de façon synchrone sur le chemin du clic.

Ils ont une cause commune. Chacun est un code parfaitement acceptable s'il s'exécute une fois, placé à un endroit qui s'exécute par ligne, par cellule ou à chaque tick. Un formateur qui coûte 21 microsecondes ne pose aucun problème ; le même formateur invoqué pour cinquante mille lignes, c'est une seconde de session figée. Et comme une session Wisej.NET traite ses événements un par un, cette seconde n'est pas seulement le problème du tableau de bord — elle retarde tout ce que l'utilisateur fait ensuite et consomme un cœur que se disputent les autres sessions de l'instance.

View models en cache et mises à jour groupées

La correction du lab a deux volets. D'abord, le formatage sort de l'interface.

`DashboardService.GetSnapshot()` renvoie un `DashboardSnapshot` dont les propriétés sont déjà des chaînes : `OpenTicketsText` , `OverdueTicketsText` , `AverageAgeText` , plus une petite liste de lignes pour le graphique. Tout ce qui est coûteux — l'agrégation, le formatage selon la culture, le calcul des âges — a lieu une seule fois, dans un service qui peut être testé unitairement et profilé isolément.

Ensuite, le gestionnaire devient une suite d'affectations de valeurs déjà préparées. Il désactive le bouton, modifie une seule fois les labels et la source de données du graphique, puis réactive le bouton dans un `finally` . Rien n'est formaté, rien n'est recréé, et la portée de sonde du Module 1 encadre toujours l'ensemble, si bien que la modification apparaît dans la même ligne de log que vous comparez depuis la référence.

```
private void btnRefresh_Click(object sender, EventArgs e)
{
    using (_probe.Measure("Dashboard", "Refresh"))
    {
        btnRefresh.Enabled = false;
        try
        {
            var snapshot = _dashboardService.GetSnapshot(); // formatted once
            lblOpen.Text = snapshot.OpenTicketsText; // assign, don't format
            lblOverdue.Text = snapshot.OverdueTicketsText;
            lblAvgAge.Text = snapshot.AverageAgeText;
            chartLoad.DataSource = snapshot.ChartRows;
        }
        finally { btnRefresh.Enabled = true; }
    }
}
```

Deux mises en garde accompagnent ce modèle. Mettez en cache au niveau de l'application les données de référence immuables — noms de statut, libellés de priorité, tables de SLA — mais ne placez jamais d'état mutable propre à un utilisateur dans un champ statique : sur un serveur multi-sessions, c'est un bug de correction, pas une simple question de style. Et ne « corrigez » pas un travail CPU en marquant la méthode `async` : `await` ne réduit pas le calcul, il déplace simplement le même CPU sur un autre thread du serveur.

Prouver la modification par une seconde trace

Le livrable n'est pas une application plus rapide, c'est une optimisation défendable. Réexécutez le même scénario — même jeu de données, même build, même mise en température — et collectez une seconde trace CPU Usage. Ce que vous cherchez, c'est un chemin critique modifié, pas seulement un chiffre plus petit : `FormatRow` doit avoir disparu du haut de l'arbre, le nombre d'appels doit s'être effondré, et ce qui se trouve désormais en tête devient le prochain candidat, ou le point où vous vous arrêtez parce que le budget est respecté.

Consignez les deux traces dans un tableau avant/après avec les trois colonnes qui comptent : le CPU total du scénario, la fonction principale et son nombre d'appels. Si les chiffres ont bougé mais que le chemin critique est inchangé, vous avez probablement mesuré de la variance. Et notez honnêtement le risque restant — un snapshot en cache introduit une fenêtre d'obsolescence, et le relecteur doit voir que vous le savez.



Pièges courants

Formater les contrôles ligne par ligne à chaque actualisation alors qu'une valeur en cache suffirait. Appeler `Application.Update` dans une boucle CPU serrée. Traiter un CPU total élevé dans le code du framework comme un bug du framework sans examiner les appels enfants. Masquer un problème de CPU derrière des mots-clés `async` qui effectuent toujours le même travail sur les threads du serveur. Mettre en cache un état utilisateur mutable dans un champ statique.



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 :

- ✓ Utiliser CPU Usage, le flame graph et l'arbre des appels pour localiser les chemins de traitement d'événements lents.
- ✓ Reconnaître le formatage répété, la réflexion, la sérialisation, le traitement d'images et les boucles synchrones dans les événements d'interface.

- ✓ Réduire le travail des gestionnaires d'événements grâce au cache, au traitement groupé, au chargement différé et aux view models précalculés.
- ✓ Distinguer le travail lié au CPU du temps de blocage et du temps d'attente de la base de données.

Guide du lab / de l'examen

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



Lab pratique

Corrigez le bouton Refresh de WisejPerfLab. Profilez le scénario d'actualisation avec CPU Usage, ouvrez le flame graph et Show Hot Path, et identifiez les deux coupables : un formateur qui s'exécute pour chaque cellule à chaque actualisation et une reconstruction de contrôles qui recrée le panneau des KPI à chaque fois. Remplacez-les par un view model DashboardSnapshot dont les chaînes d'affichage sont calculées une seule fois dans le service, modifiez les labels et la source de données du graphique une seule fois dans une portée de sonde plutôt que ligne par ligne, et gardez le bouton désactivé dans un try/finally pendant le traitement. Réexécutez le même scénario et capturez le nouveau chemin critique.

Point de départ

Vous ouvrez WisejPerfLab avec les traces du Module 2 sous la main : l'actualisation du tableau de bord coûte environ 1 840 ms, `TicketFormatter.FormatRow` concentre 68,9 % de CPU propre sur environ 50 000 appels, et `RebuildKpiPanel` libère et recrée le panneau des KPI à chaque clic. La couche de services renvoie pour l'instant des entités brutes et la page effectue tout le formatage.

Approche suggérée

- ✓ Reproduisez d'abord le constat : collectez vous-même la trace CPU Usage, ouvrez le flame graph et confirmez avec la vue appelants/appelés quelle boucle provoque les 50 000 appels au formateur. Ne modifiez pas un code que vous n'avez pas vu dans une trace.
- ✓ Introduisez un view model `DashboardSnapshot` avec des propriétés prêtes à afficher — texte du nombre de tickets ouverts, texte du nombre de tickets en retard, texte de l'âge moyen et une petite liste de lignes pour le graphique — et construisez-le une seule fois dans `DashboardService.GetSnapshot()`.
- ✓ Déplacez tous les appels de formatage de la page vers le service, y compris le formatage des nombres et des durées selon la culture. Ne mettez en cache que les données de référence immuables comme les libellés de statut et de priorité ; gardez tout ce qui est propre à l'utilisateur dans la session.
- ✓ Réécrivez le gestionnaire sous forme d'affectations de valeurs préparées dans la portée de sonde existante : désactivez le bouton, définissez une seule fois les labels et la source de données du graphique, réactivez dans un `finally`. Supprimez la reconstruction du panneau des KPI et mettez plutôt à jour les contrôles existants.
- ✓ Réexécutez le même scénario — même jeu de données, même build, même mise en température —, collectez une seconde trace CPU Usage et construisez le tableau avant/après à partir du CPU total, de la fonction principale et du nombre d'appels.

Livrables attendus

- ✓ Trace CPU Usage « avant » identifiant le formateur répété et la reconstruction de contrôles avec leur CPU total et propre

- ✓ View model DashboardSnapshot avec des chaînes d’affichage précalculées dans le service
- ✓ Gestionnaire d’actualisation groupé qui modifie les contrôles une seule fois dans une portée de sonde, avec réactivation dans un try/finally
- ✓ Trace CPU Usage « après » du même scénario avec le nouveau chemin critique
- ✓ Tableau avant/après du CPU total, de la fonction principale et du nombre d’appels, avec le risque restant

Questions de révision

- ✓ Les deux coupables étaient lents, mais pour des raisons opposées. Expliquez lequel relevait d’un problème de nombre d’appels et lequel d’un problème de travail par appel, et pourquoi les preuves du profileur diffèrent pour chacun.
- ✓ Un collègue propose de marquer le gestionnaire d’actualisation `async` pour le rendre plus rapide. Qu’est-ce qui changerait réellement, qu’est-ce qui ne changerait pas, et comment la trace le montrerait-elle ?
- ✓ Votre snapshot est construit une fois par actualisation. Quel risque d’obsolescence ou de correction cela introduit-il, et que faudrait-il changer si le même snapshot était mis en cache entre les sessions ?



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.