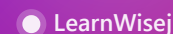


Mémoire, allocations et fuites de session

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



Guide de la leçon

La pression d'allocation par opposition à la mémoire retenue, la démarche de comparaison d'instantanés, les schémas de rétention propres à Wisej.NET (événements statiques, timers, tâches en arrière-plan, caches globaux) et la checklist de libération qui prouve qu'une fuite a disparu. 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 pression d'allocation par opposition à la mémoire retenue, la démarche de comparaison d'instantanés, les schémas de rétention propres à Wisej.NET (événements statiques, timers, tâches en arrière-plan, caches globaux) et la checklist de libération qui prouve qu'une fuite a disparu.

1. Le problème en production

Le tableau de bord de WisejPerfLab respecte désormais son budget CPU, et pourtant l'application s'effondre au bout de quatre heures en production. Les problèmes de mémoire ne se comportent pas comme les problèmes de CPU : ils sont invisibles dans une démo de deux minutes, ils ne se manifestent jamais sous la forme d'un clic lent, et ce sont eux qui expliquent qu'un serveur qui tient confortablement quatre-vingts sessions le matin refuse des connexions en milieu d'après-midi. Le lab en contient deux, et ce n'est pas le même problème. La recherche de tickets alloue bien plus que nécessaire, ce qui coûte du temps de GC et du CPU pendant son exécution. Le formulaire de détail d'un ticket n'est jamais réellement libéré après sa fermeture par l'utilisateur, ce qui coûte de la mémoire pour toujours. Le premier se trouve avec une trace d'allocation, le second uniquement avec deux instantanés et un chemin vers la racine.

2. Concepts clés

Concept	Pourquoi c'est important
pression d'allocation	La quantité de mémoire qu'un scénario crée pendant son exécution ; une forte pression augmente la fréquence du GC et vole du CPU, même quand rien ne fuit.
mémoire retenue	Ce qui reste accessible alors que le scénario devrait être terminé ; c'est ce qui se multiplie par le nombre de sessions et décide de la capacité.
.NET Object Allocation	L'outil qui enregistre chaque allocation managée pendant un scénario et les regroupe par type et par chemin d'appel qui alloue.
instantanés Memory Usage	Des captures du tas prises à des moments choisis ; comparer l'instantané A et l'instantané B montre quels types ont gagné des instances jamais libérées.
chemin vers la racine	La chaîne de références qui maintient un objet en vie ; c'est la preuve concrète d'une fuite, et le seul moyen de prouver que la fuite a disparu.
racine de rétention	Ce qui se trouve en haut de cette chaîne — un champ statique, un événement statique, un timer actif, une tâche en arrière-plan ou un cache à longue durée de vie.
budget mémoire par session	Un chiffre, en mégaoctets retenus par session inactive, qui transforme « l'application consomme beaucoup de mémoire » en verdict de réussite ou d'échec, puis en seuil de health check.

3. Déroulé de la leçon

Pourquoi la mémoire est une question de capacité dans Wisej.NET

Wisej.NET maintient les objets d'interface côté serveur en vie pendant toute la durée de la session. C'est ce qui rend le modèle de programmation agréable, et c'est ce qui rend une fuite coûteuse. Un formulaire modal jamais libéré, un `BindingSource` qui retient encore cinquante mille lignes, un événement statique qui épingle une page, un timer qui continue de battre après que le panneau a été masqué : chacun représente quelques mégaoctets, invisibles sur l'ordinateur portable d'un développeur et décisifs à deux cents sessions.

Le modèle mental à emporter dans le profileur est que la mémoire soulève deux questions distinctes. *Combien ce scénario a-t-il fait tourner de mémoire ?* concerne le coût du GC et le CPU. *Qu'est-ce qui est encore en vie maintenant que l'utilisateur a tout fermé ?* concerne la capacité. Un scénario peut être catastrophique sur l'une et parfait sur l'autre ; un rapport qui ne répond qu'à l'une des deux n'a donc pas blanchi l'application.

Trouver la pression d'allocation

Pour l'allocation, exécutez le scénario sous *.NET Object Allocation* en Release, une seule fois, et ouvrez les vues d'allocation. La vue des types répond à « qu'est-ce qui est créé » et l'arbre des appels d'allocation à « qui le crée ». Pour la recherche de tickets, la réponse n'est pas flatteuse : les objets modèles de ligne et leurs chaînes formatées sont reconstruits à chaque rafraîchissement de la grille, si bien qu'une seule recherche alloue plusieurs fois le volume des données qu'elle affiche.

La correction applique la même discipline qu’au Module 3, aux objets plutôt qu’au CPU : projeter une seule fois dans un type de ligne léger et le transmettre à la grille, au lieu de construire de nouveaux objets à chaque rafraîchissement. Ce que vous cherchez dans la trace « après », c’est une baisse à la fois du total d’octets alloués et du nombre d’instances du type fautif — et, idéalement, une courbe de GC visiblement plus calme.

Trouver la rétention

La rétention exige une autre démarche, et l’ordre des étapes constitue toute la technique. Lancez l’application en Release et faites-la chauffer. Prenez l’instantané A. Exécutez le scénario censé être neutre — ouvrez puis fermez le formulaire de détail d’un ticket, cinquante fois, pour qu’une instance qui fuit en devienne cinquante et ne puisse pas passer pour du bruit. Ramenez l’application à un véritable état d’inactivité. Prenez l’instantané B, et comparez.

Dans la comparaison, vous cherchez des types dont le nombre d’instances a augmenté sans jamais redescendre : `TicketDetailForm` à cinquante instances ne laisse aucune ambiguïté. Sélectionnez le type, ouvrez les instances et examinez le chemin vers la racine. Dans WisejPerfLab, deux racines apparaissent. La première est un événement statique — le formulaire s’est abonné à `GlobalTicketBus.TicketChanged` dans son constructeur sans jamais se désabonner, si bien que la liste statique de délégués retient tous les formulaires jamais ouverts. La seconde est un `System.Timers.Timer` que le formulaire démarre pour actualiser son en-tête ; son callback capture le formulaire, donc tant que le timer est en vie, le formulaire l’est aussi.

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        GlobalTicketBus.TicketChanged -= OnTicketChanged; // the static root
        _refreshTimer?.Stop(); // the timer root
        _refreshTimer?.Dispose();
        bindingSource1.DataSource = null; // release the rows
        _largeImage?.Dispose();
    }
    base.Dispose(disposing);
}
```

Les deux corrections ont leur place au même endroit : une surcharge de `Dispose` qui désabonne ce à quoi l’objet s’est abonné, arrête et libère ce qu’il a démarré, et vide les bindings qui retiennent de gros volumes de données. Tout ce qui touche à l’interface depuis un timer ou une tâche en arrière-plan doit aussi vérifier `IsDisposed` avant de mettre à jour les contrôles, car le callback peut survivre quelques millisecondes au formulaire même quand la libération est correcte.

Le prouver, et rédiger le budget

Une correction mémoire n’est pas terminée quand le code a l’air juste. Répétez le même scénario d’ouverture et de fermeture, reprenez l’instantané B et montrez deux choses : le nombre d’instances de `TicketDetailForm` revient à sa référence, et l’ancien chemin vers la racine n’existe plus. La seconde est la plus importante — un nombre qui baisse parce qu’une garbage collection s’est justement déclenchée n’est

pas une preuve, et forcer des collections jusqu'à ce que les chiffres paraissent acceptables est la manière dont une fuite survit à une revue.

Terminez par les deux documents dont le reste du cours a besoin. Le budget mémoire par session indique combien de mégaoctets une session inactive peut retenir, mesuré sur votre propre application plutôt que deviné ; le Module 7 transforme ce chiffre en seuil `maxSessions` du health check. La checklist de libération est la version vérifiable de ce même savoir : désabonner les événements, arrêter et libérer les timers, annuler le traitement en arrière-plan, vider les bindings, libérer les images et les flux, relâcher les grandes listes, et ne jamais stocker d'état utilisateur mutable dans un champ statique.



Pièges courants

Mesurer les allocations et déclarer l'application sans fuite sans jamais vérifier la rétention. Forcer des garbage collections jusqu'à ce que les chiffres soient bons au lieu de lire les chemins vers la racine.

Conserver un `DbContext`, une `DataTable` ou un cache d'images dans la session indéfiniment. S'abonner à un événement statique depuis une page sans jamais se désabonner. Mettre à jour des contrôles depuis un callback de timer sans vérifier `IsDisposed`.



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 .NET Object Allocation pour trouver les chemins de code qui allouent beaucoup pendant un scénario.
- ✓ Utiliser les instantanés Memory Usage pour comparer l'état du tas avant et après la fermeture d'objets d'interface Wisej.NET.
- ✓ Identifier les racines de rétention dues aux champs statiques, aux événements, aux timers, aux caches et aux services à longue durée de vie.
- ✓ Établir un budget mémoire par session et une checklist de libération.

Guide du lab / de l'examen

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



Lab pratique

Attaquez les deux moitiés du problème de mémoire dans WisejPerfLab. Exécutez le scénario de recherche de tickets sous .NET Object Allocation et repérez les modèles de ligne et les chaînes alloués à chaque recherche, puis réduisez-les en projetant une seule fois au lieu de reconstruire des objets à chaque rafraîchissement de l'affichage. Prenez ensuite un instantané Memory Usage après la mise en température, ouvrez et fermez cinquante fois le formulaire de détail d'un ticket, prenez un second instantané et comparez : les formulaires retenus sont enracinés par un abonnement statique à `GlobalTicketBus.TicketChanged` et par un timer d'actualisation jamais arrêté. Désabonnez l'un et arrêtez l'autre dans une surcharge de `Dispose`, videz la source de données du `BindingSource`, puis réexécutez le scénario d'ouverture/fermeture et montrez que le tas retenu revient à un niveau proche de l'instantané A.

Point de départ

WisejPerfLab après le Module 3 : l'actualisation du tableau de bord respecte son budget CPU, la grille des tickets est toujours liée à des entités complètes, et le formulaire de détail d'un ticket s'abonne à `GlobalTicketBus.TicketChanged` dans son constructeur et démarre un `System.Timers.Timer` pour actualiser son en-tête. Ni l'un ni l'autre n'est défait nulle part. Le budget du Module 1 contient déjà un objectif de mémoire retenue que l'application n'atteint pas.

Approche suggérée

- ✓ Exécutez une fois la recherche de tickets sous *.NET Object Allocation* en Release. Utilisez la vue des types pour trouver ce qui domine et l'arbre des appels d'allocation pour trouver qui le crée, et notez le total d'octets alloués ainsi que le nombre d'instances du type principal.
- ✓ Projetez une seule fois dans un type de ligne léger et liez-le, pour que les objets de ligne et leurs chaînes d'affichage soient créés une seule fois par recherche au lieu d'être reconstruits à chaque rafraîchissement. Réexécutez la trace d'allocation sur le même scénario.
- ✓ Passez à *Memory Usage*. Faites chauffer l'application, prenez l'instantané A, puis ouvrez et fermez cinquante fois le formulaire de détail pour qu'une seule instance qui fuit en devienne cinquante, ramenez l'application à un véritable état d'inactivité et prenez l'instantané B.
- ✓ Comparez les instantanés, sélectionnez le type de formulaire dont le nombre d'instances a augmenté et lisez le chemin vers la racine de deux instances. Capturez les deux racines : l'abonnement à l'événement statique et le timer actif qui retient le formulaire capturé par le callback.
- ✓ Surchargez `Dispose(bool)` sur le formulaire pour désabonner l'événement statique, arrêter et libérer le timer, vider la source de données du `BindingSource` et libérer les éventuelles images ; ajoutez une vérification `IsDisposed` avant que le callback du timer ne touche aux contrôles. Réexécutez le même scénario d'ouverture/fermeture et prouvez que le nombre d'instances et le chemin vers la racine ont tous deux disparu.

Livrables attendus

- ✓ Trace d'allocation identifiant le type qui alloue le plus et le chemin d'appel pour le scénario de recherche, avant et après
- ✓ Comparaison des instantanés A/B montrant le nombre d'instances retenues du formulaire de détail avant la correction
- ✓ Capture d'écran du chemin vers la racine identifiant l'abonnement à l'événement statique et le timer actif
- ✓ Surcharge de Dispose qui désabonne l'événement statique, arrête et libère le timer et vide le binding
- ✓ Budget mémoire par session et checklist de libération, avec l'instantané « après » prouvant que le chemin de rétention a disparu

Questions de révision

- ✓ Votre trace d'allocation s'est nettement améliorée, mais la comparaison d'instantanés est inchangée. Quel type de problème avez-vous corrigé, lequel ne l'avez-vous pas corrigé, et comment un utilisateur percevrait-il la différence ?
- ✓ Pourquoi le lab ouvre-t-il et ferme-t-il le formulaire cinquante fois plutôt qu'une, et que concluriez-vous d'une seule instance présente dans l'instantané B ?
- ✓ Le callback du timer tient en une ligne et ne fait que définir un label. Expliquez précisément comment il maintient en vie tout un formulaire, et ce qu'une vérification `IsDisposed` résout et ne résout pas.



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.