



→ Digitalize society



Formation F# 5.0

Programmation asynchrone



Décembre 2021

SOAT.FR

About me



Romain DENEAU

- SOAT depuis 2009
- Senior Developer C# F# TypeScript
- Passionné de Craft
- Auteur sur le blog de SOAT



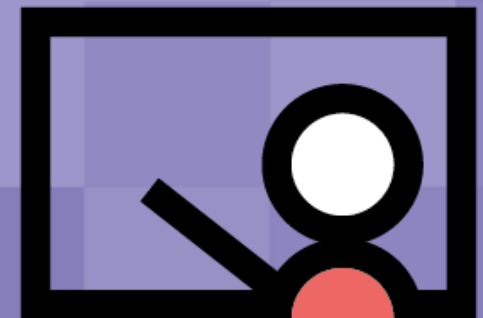
DeneauRomain



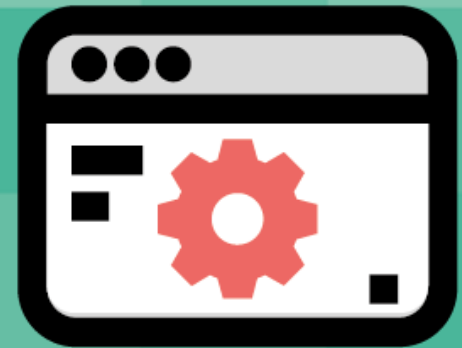
rdeneau

Sommaire

- *Workflow* asynchrone
- Interop avec la TPL .NET




1. Workflow asynchrone



Workflow asynchrone : Besoins

1. Ne pas bloquer le thread courant en attendant un calcul long
2. Permettre calculs en parallèle
3. Indiquer qu'un calcul peut prendre du temps

Type `Async<'T>`

- Représente un calcul asynchrone
- Similaire au pattern `async/await` avant l'heure 
 - 2007 : `Async<'T>` F#
 - 2012 : `Task<T>` .NET et pattern `async` / `await`
 - 2017 : `Promise` JavaScript et pattern `async` / `await`

Méthodes renvoyant un objet Async

`Async.AwaitTask(task: Task or Task<'T>) : Async<'T>`

→ Conversion d'une `Task` (.NET) en `Async` (F#)

`Async.Sleep(milliseconds or TimeSpan) : Async<unit>`

≈ `await Task.Delay()` ≠ `Thread.Sleep` → ne bloque pas le thread courant

[FSharp.Control.CommonExtensions](#) : étend le type `System.IO.Stream`

→ `AsyncRead(buffer: byte[], ?offset: int, ?count: int) : Async<int>`

→ `AsyncWrite(buffer: byte[], ?offset: int, ?count: int) : Async<unit>`

[FSharp.Control.WebExtensions](#) : étend le type `System.Net.WebClient`

→ `AsyncDownloadData(address: Uri) : Async<byte[]>`

→ `AsyncDownloadString(address: Uri) : Async<string>`

Lancement d'un calcul async

```
Async.RunSynchronously(calc: Async<'T>, ?timeoutMs: int, ?cancellationToken) : 'T
```

→ Attend la fin du calcul mais bloque le thread appelant ! (≠ `await` C#) ⚠

```
Async.Start(operation: Async<unit>, ?cancellationToken) : unit
```

→ Exécute l'opération en background (*sans bloqué le thread appelant*)

⚠ Si une exception survient, elle est "avalée" !

```
Async.StartImmediate(calc: Async<'T>, ?cancellationToken) : unit
```

→ Exécute le calcul dans le thread appelant !

💡 Pratique dans une GUI pour la mettre à jour : barre de progression...

```
Async.StartWithContinuations(calc, continuations ..., ?cancellationToken)
```

→ Idem `Async.RunSynchronously` ⚠ ... avec 3 *callbacks* de continuation :

→ en cas de succès ✅, d'exception 💣 et d'annulation 🛑

Bloc `async { expression }`

A.k.a. Async workflow

Syntaxe pour écrire de manière séquentielle un calcul asynchrone

→ Le résultat du calcul est wrappé dans un objet `Async`

Mots clés

- `return` → valeur finale du calcul - `unit` si omis
- `let!` (*prononcer « let bang »*)
→ accès au résultat d'un sous-calcul async (\simeq `await` en C#)
- `use!` → idem `use` (*gestion d'un `IDisposable`*) + `let!`
- `do!` → idem `let!` pour calcul async sans retour (`Async<unit>`)

Bloc `async` - Exemples

```
let repeat (computeAsync: int → Async<string>) times = async {  
    for i in [ 1..times ] do  
        printf $"Start operation #{i} ... "  
        let! result = computeAsync i  
        printfn $"Result: {result}"  
}
```

```
let basicOp (num: int) = async {  
    do! Async.Sleep 150  
    return $"#{num} * ({num} - 1) = {num * (num - 1)}"  
}
```

```
repeat basicOp 5 ▷ Async.RunSynchronously
```

```
// Start operation #1 ... Result: 1 * (1 - 1) = 0  
// Start operation #2 ... Result: 2 * (2 - 1) = 2  
// Start operation #3 ... Result: 3 * (3 - 1) = 6  
// Start operation #4 ... Result: 4 * (4 - 1) = 12  
// Start operation #5 ... Result: 5 * (5 - 1) = 20
```

F#

Usage inapproprié de `Async.RunSynchronously`

`Async.RunSynchronously` lance le calcul et renvoie son résultat MAIS en bloquant le thread appelant ! Ne l'utiliser qu'en « bout de chaîne » et pas pour *unwrap* des calculs asynchrones intermédiaires ! Utiliser plutôt un bloc `async`.

```
// ❌ À éviter
let a = calcA ▷ Async.RunSynchronously
let b = calcB a ▷ Async.RunSynchronously
calcC b

// ✅ À préférer
async {
    let! a = calcA
    let! b = calcB a
    return calcC b
} ▷ Async.RunSynchronously
```

F#

Calculs en parallèle

1. `Async.Parallel(computations: seq<Async<'T>>, ?maxBranches) : Async<'T[]>`

≈ `Task.WhenAll` : modèle [Fork-Join](#)

- *Fork* : calculs lancés en parallèle
- Attente de la terminaison de tous les calculs
- *Join* : agrégation des résultats (*qui sont du même type*)
 - dans le même ordre que les calculs

Async.Parallel - Exemple

```
let downloadSite (site: string) = async {  
    do! Async.Sleep (100 * site.Length)  
    printfn $"{site} ✓"  
    return site.Length  
}  
  
[ "google"; "msn"; "yahoo" ]  
▷ List.map downloadSite // string list  
▷ Async.Parallel        // Async<string[]>  
▷ Async.RunSynchronously // string[]  
▷ printfn "%A"  
  
// msn ✓  
// yahoo ✓  
// google ✓  
// [6; 3; 5]
```

F#

Calculs en parallèle (2)

2. `Async.StartChild(calc: Async<'T>, ?timeoutMs: int) : Async<Async<'T>>`

Permet de lancer en parallèle plusieurs calculs

→ ... dont les résultats sont de types différents (\neq `Async.Parallel`)

S'utilise dans bloc `async` avec 2 `let!` par calcul enfant (cf. `Async<Async<'T>>`)

Annulation conjointe 

→ Calcul enfant partage jeton d'annulation du calcul parent

Async.StartChild - Exemple partie 1

Soit le fonction `delay`

→ qui renvoie la valeur spécifiée `x`

→ au bout de `ms` millisecondes

```
let delay (ms: int) x = async {  
    do! Async.Sleep ms  
    return x  
}
```

```
// 💡 Minutage avec la directive FSI `#time` • https://kutt.it/Zbp6ot  
#time "on" // → Minutage activé  
"a" ▷ delay 100 ▷ Async.RunSynchronously // Réel : 00:00:00.111, Proc ...  
#time "off" // → Minutage désactivé
```

F#

Async.StartChild - Exemple partie 2

```
let inSeries = async {  
    let! result1 = "a" ▷ delay 100  
    let! result2 = 123 ▷ delay 200  
    return (result1, result2)  
}  
  
let inParallel = async {  
    let! child1 = "a" ▷ delay 100 ▷ Async.StartChild  
    let! child2 = 123 ▷ delay 200 ▷ Async.StartChild  
    let! result1 = child1  
    let! result2 = child2  
    return (result1, result2)  
}  
  
#time "on"  
inSeries ▷ Async.RunSynchronously // Réel : 00:00:00.317, ...  
#time "off"  
#time "on"  
inParallel ▷ Async.RunSynchronously // Réel : 00:00:00.205, ...  
#time "off"
```

F#

Annulation d'une tâche

Se base sur un `CancellationToken/Source` par défaut ou explicite :

- `Async.RunSynchronously(computation, ?timeout, ?cancellationToken)`
- `Async.Start(computation, ?cancellationToken)`

Déclencher l'annulation

- Token explicite + `cancellationTokenSource.Cancel()`
- Token explicite avec timeout `new CancellationTokenSource(timeout)`
- Token par défaut : `Async.CancelDefaultToken()` → `OperationCanceledException` 

Vérifier l'annulation

- Implicite : à chaque mot clé dans bloc async : `let`, `let!`, `for` ...
- Explicite local : `let! ct = Async.CancellationToken` puis `ct.IsCancellationRequested`
- Explicite global : `Async.OnCancel(callback)`

Annulation d'une tâche - Exemple - Partie 1

```
let sleepLoop = async {  
    let stopwatch = System.Diagnostics.Stopwatch()  
    stopwatch.Start()  
    let log message = printfn $"[{stopwatch.Elapsed.ToString("s\\.fff")}] {message}"  
  
    use! __ = Async.OnCancel (fun () →  
        log $"  Cancelled ✖")  
  
    for i in [ 1..5 ] do  
        log $"Step #{i} ... "  
        do! Async.Sleep 500  
        log $"  Completed ✔"  
}
```

F#

Annulation d'une tâche - Exemple - Partie 2

```
open System.Threading

printfn "1. RunSynchronously:"
Async.RunSynchronously(sleepLoop)

printfn "2. Start with CancellationTokenSource + Sleep + Cancel"
use manualCancellationSource = new CancellationTokSource()
Async.Start(sleepLoop, manualCancellationSource.Token)
Thread.Sleep(1200)
manualCancellationSource.Cancel()

printfn "3. Start with CancellationTokenSource with timeout"
use cancellationByTimeoutSource = new CancellationTokSource(1200)
Async.Start(sleepLoop, cancellationByTimeoutSource.Token)
```

F#

Annulation d'une tâche - Exemple - Outputs

```
1. RunSynchronously:
  [0.009] Step #1 ...
  [0.532]   Completed ✓
  [0.535] Step #2 ...
  [1.037]   Completed ✓
  [1.039] Step #3 ...
  [1.543]   Completed ✓
  [1.545] Step #4 ...
  [2.063]   Completed ✓
  [2.064] Step #5 ...
  [2.570]   Completed ✓

2. Start with CancellationTokenSource + Sleep + Cancel
  [0.000] Step #1 ...
  [0.505]   Completed ✓
  [0.505] Step #2 ...
  [1.011]   Completed ✓
  [1.013] Step #3 ...
  [1.234]   Cancelled ✗

3. Start with CancellationTokenSource with timeout
... idem 2.
```

2. Interop avec TPL.NET



TPL : Task Parallel Library

Interaction avec librairie .NET

Librairies asynchrones en .NET et pattern `async` / `await` C# :

→ Basés sur **TPL** et le type `Task`

Passerelles avec worflow asynchrone F# :

- Fonctions `Async.AwaitTask` et `Async.StartAsTask`
- Bloc `task {}`

Fonctions passerelles

`Async.AwaitTask: Task<'T> → Async<'T>`

→ Consommer une librairie .NET asynchrone dans bloc `async`

`Async.StartAsTask: Async<'T> → Task<'T>`

→ Lancer un calcul async sous forme de `Task`

```
let getValueFromLibrary param = async {  
    let! value = DotNetLibrary.GetValueAsync param ▷ Async.AwaitTask  
    return value  
}  
  
let computationForCaller param =  
    async {  
        let! result = getAsyncResult param  
        return result  
    } ▷ Async.StartAsTask
```

F#

Bloc `task {}`

“ Permet de consommer directement une librairie .NET asynchrone en ne faisant qu'un seul `Async.AwaitTask` plutôt que 1 à chaque méthode appelée ”

💡 Disponible en F# 6 ou via package nuget [Ply](#)

```
#r "nuget: Ply"
open FSharp.Control.Tasks

task {
    use client = new System.Net.Http.HttpClient()
    let! response = client.GetStringAsync("https://www.google.fr/")
    response.Substring(0, 300) ▷ printfn "%s"
} // Task<unit>
▷ Async.AwaitTask // Async<unit>
▷ Async.RunSynchronously
```

F#

Async VS Task

1. Mode de démarrage du calcul

Task = *hot tasks* → calculs démarrés immédiatement !

Async = *task generators* = spécification de calculs, indépendante du démarrage
→ Approche fonctionnelle : sans effet de bord ni mutation, composabilité
→ Contrôle du mode de démarrage : quand et comment 👍

2. Support de l'annulation

Task : en ajoutant un paramètre **CancellationToken** aux méthodes async
→ Oblige à tester manuellement si token est annulé = fastidieux + *error prone* !

Async : support automatique dans les calculs - token à fournir au démarrage 👍

Pièges du pattern `async` / `await` en C#

Piège 1 - Vraiment asynchrone ?

En C# : méthode `async` reste sur le thread appelant jusqu'au 1er `await`
→ Sentiment trompeur d'être asynchrone dans toute la méthode

```
async Task WorkThenWait() {  
    Thread.Sleep(1000);           // ⚠ Bloque thread appelant !  
    await Task.Delay(1000);       // Vraiment async à partir d'ici 🤔  
}
```

C#

En F# : `async` ne définit pas une fonction mais un **bloc**

```
let workThenWait () =  
    Thread.Sleep(1000)  
    async { do! Async.Sleep(1000) } // Async que dans ce bloc 🤔
```

F#

Préconisation pour fonction asynchrone en F#

Fonction asynchrone = renvoyant un `Async<_>`

→ On s'attend à ce qu'elle soit **totale**ment asynchrone

→ Fonction précédente `workThenWait` ne respecte pas cette attente

👉 Préconisation :

» Mettre tout le corps de la fonction asynchrone dans un bloc `async`

```
let workThenWait () = async {  
    Thread.Sleep(1000)  
    printfn "work"  
    do! Async.Sleep(1000)  
}
```

F#

Piège 2 - Omettre le `await` en C#

```
async Task PrintAfterOneSecond(string message) {  
    await Task.Delay(1000);  
    Console.WriteLine($"[{DateTime.Now:T}] {message}");  
}  
  
async Task Main() {  
    PrintAfterOneSecond("Before"); // ⚠ Manque `await` → warning CS4014  
    Console.WriteLine($"[{DateTime.Now:T}] After");  
    await Task.CompletedTask;  
}
```

Cela compile  et produit un résultat inattendu *After* avant *Before* !

```
[11:45:27] After  
[11:45:28] Before
```

Piège 2 - Équivalent en F#

```
let printAfterOneSecond message = async {  
    do! Async.Sleep 1000  
    printfn $"[{DateTime.Now:T}] {message}"  
}  
  
async {  
    printAfterOneSecond "Before" // ⚠ Manque `do!` → warning FS0020  
    printfn $"[{DateTime.Now:T}] After"  
} ▷ Async.RunSynchronously
```

F#

Cela compile aussi 📌 et produit un autre résultat inattendu : pas de *Before* !

```
[11:45:27] After
```

Piège 2 - Étude des warnings

Les exemples précédemment compilent mais avec des gros *warnings* !

En C#, le [warning CS4014](#) indique :

“ *Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the `await` operator...* ”

En F#, le *warning FS0020* est accompagné du message :

“ *The result of this expression has type `Async<unit>` and is implicitly ignored. Consider using `ignore` to discard this value explicitly...* ”

👉 **Préconisation** : veuillez à **toujours** traiter ce type de *warning* !
C'est encore + crucial en F# où la compilation est + délicate.

3 ■ Le Récap'



Programmation asynchrone en F#

Via bloc `async {}` en F# pur

- Similaire mais antérieur au pattern `async / await`
- Permet d'éviter quelques pièges du `async / await`
- Oblige à démarrer manuellement calcul
- Mais compilation empêche d'oublier

Via bloc `task {}`

- Facilite interactions avec librairie .NET asynchrone



Ressources complémentaires

 <https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/async>

Merci 🙏

SOAT

→ Digitalize society



SOAT.FR