

Async и многопоточное программирование

Юрий Литвинов

08.04.2016г

Многопоточное программирование

Зачем это нужно:

- ▶ Не “вешать” GUI
- ▶ Использовать ресурсы “железа”
- ▶ Использовать асинхронные операции ввода-вывода
- ▶ Показывать прогресс

Основные понятия

Процесс — экземпляр выполняемой программы и все связанные с ней данные и ресурсы

Поток — поток исполняемых команд (стек, регистры)

Параллельная программа — программа с несколькими исполняемыми потоками или процессами

Асинхронная программа выполняет запросы, которые выполняются не мгновенно, а через некоторое время

Реактивная программа — программа, нормальное состояние которой — ожидать наступление какого-нибудь события

Простой способ: Async workflow

F#

```
open System.Net
open System.IO
let sites = ["http://se.math.spbu.ru";
             "http://spisok.math.spbu.ru"]
let fetchAsync url =
    async {
        do printfn "Creating request for %s..." url
        let request = WebRequest.Create(url)
        use! response = request.AsyncGetResponse()
        do printfn "Getting response stream for %s..." url
        use stream = response.GetResponseStream()
        do printfn "Reading response for %s..." url
        use reader = new StreamReader(stream)
        let html = reader.ReadToEnd()
        do printfn "Read %d characters for %s..." html.Length url
    }

sites |> List.map (fun site -> site |> fetchAsync |> Async.Start)
|> ignore
```

Что получится

F# Interactive

```
Creating request for http://se.math.spbu.ru ...
Creating request for http://spisok.math.spbu.ru ...
val sites : string list =
    ["http://se.math.spbu.ru"; "http://spisok.math.spbu.ru"]
val fetchAsync : url:string -> Async<unit>
val it : unit = ()

> Getting response stream for http://spisok.math.spbu.ru ...
Reading response for http://spisok.math.spbu.ru ...
Read 4475 characters for http://spisok.math.spbu.ru ...
Getting response stream for http://se.math.spbu.ru ...
Reading response for http://se.math.spbu.ru ...
Read 217 characters for http://se.math.spbu.ru ...
```

Переключение между потоками

Распечатаем Id потоков, в которых вызываются методы printfn:

F#

```
open System.Threading

let tprintfn fmt =
    printf "[.NET Thread %d]"
        Thread.CurrentThread.ManagedThreadId;
    printfn fmt
```

Что получилось теперь

F# Interactive

```
[.NET Thread 47][.NET Thread 49]Creating request
    for http://se.math.spbu.ru ...
Creating request for http://spisok.math.spbu.ru ...
val sites : string list =
    ["http://se.math.spbu.ru"; "http://spisok.math.spbu.ru"]
val tprintfn : fmt:Printf.TextWriterFormat<'a> -> 'a
val fetchAsync : url:string -> Async<unit>
val it : unit = ()

> [.NET Thread 49]Getting response stream for
    http://spisok.math.spbu.ru ...
[.NET Thread 49]Reading response for http://spisok.math.spbu.ru ...
[.NET Thread 50]Getting response stream for http://se.math.spbu.ru ...
[.NET Thread 50]Reading response for http://se.math.spbu.ru ...
[.NET Thread 50][.NET Thread 49]Read 217 characters
    for http://se.math.spbu.ru ...
Read 4475 characters for http://spisok.math.spbu.ru ...
```

Пул потоков

Почему такие результаты

Пул потоков — набор заранее созданных потоков, управляемых рантаймом .NET автоматически.

F#

```
open System.Threading
```

```
fun _ -> printfn "Hello from %d!"  
         <| Thread.CurrentThread.ManagedThreadId  
|> List.replicate 10  
|> List.map ThreadPool.QueueUserWorkItem  
|> ignore
```


Что получится

F# Interactive

```
valHello from 73!  
Hello from 73!  
Hello from 73!  
Hello from 73!  
  it Hello from 77: Hello from Hello from 74!  
!  
unit = 65!  
Hello from Hello from 76!  
78!  
(  
  
> Hello from 75!
```

Подробнее про Async

Async — это Workflow

F#

```
type Async<'a> = Async of ('a -> unit) * (exn -> unit)  
    -> unit
```

```
type AsyncBuilder with  
    member Return : 'a -> Async<'a>  
    member Delay : (unit -> Async<'a>) -> Async<'a>  
    member Using : 'a * ('a -> Async<'b>) ->  
        Async<'b> when 'a :> System.IDisposable  
    member Let : 'a * ('a -> Async<'b>) -> Async<'b>  
    member Bind : Async<'a> * ('a -> Async<'b>) ->  
        Async<'b>
```

Во что Async раскрывает компилятор

Если кто не помнит про Workflow-ы

F#

```
async {  
    let request = WebRequest.Create(url)  
    let! response = request.AsyncGetResponse()  
    let stream = response.GetResponseStream()  
    let reader = new StreamReader(stream)  
    let html = reader.ReadToEnd()  
    html  
}
```

```
async.Delay(fun () ->  
    WebRequest.Create(url) |> (fun request ->  
        async.Bind(request.AsyncGetResponse(), (fun response ->  
            response.GetResponseStream() |> fun stream ->  
                new StreamReader(stream) |> fun reader ->  
                    reader.ReadToEnd() |> fun html ->  
                        async.Return(html))))))
```

Какие конструкции поддерживает Async

Конструкция	Описание
<code>let! pat = expr</code>	Выполняет асинхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> , когда оно заканчивается
<code>let pat = expr</code>	Выполняет синхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> немедленно
<code>use! pat = expr</code>	Выполняет асинхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> , когда оно заканчивается. Вызовет <code>Dispose</code> для каждого имени из <code>pat</code> , когда <code>Async</code> закончится.
<code>use pat = expr</code>	Выполняет синхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> немедленно. Вызовет <code>Dispose</code> для каждого имени из <code>pat</code> , когда <code>Async</code> закончится.
<code>do! expr</code>	Выполняет асинхронную операцию <code>expr</code> , эквивалентно <code>let! () = expr</code>
<code>do expr</code>	Выполняет синхронную операцию <code>expr</code> , эквивалентно <code>let () = expr</code>
<code>return expr</code>	Оборачивает <code>expr</code> в <code>Async<'T></code> и возвращает его как результат <code>Workflow</code>
<code>return! expr</code>	Возвращает <code>expr</code> типа <code>Async<'T></code> как результат <code>Workflow</code>

Control.Async

Что можно делать со значением Async<'T>, сконструированным билдером

Метод	Тип	Описание
RunSynchronously	Async<'T> * ?int * ?CancellationTokens -> 'T	Выполняет вычисление синхронно, возвращает результат
Start	Async<unit> * ?CancellationTokens -> unit	Запускает вычисление асинхронно, тут же возвращает управление
Parallel	seq<Async<'T> > -> Async<'T []>	По последовательности Async-ов делает новый Async, исполняющий все Async-и параллельно и возвращающий массив результатов
Catch	Async<'T> -> Async<Choice<'T,exn> >	По Async-у делает новый Async, исполняющий Async и возвращающий либо результат, либо исключение

Пример

F#

```
let writeFile fileName bufferData =  
    async {  
        use outputFile = System.IO.File.Create(fileName)  
        do! outputFile.AsyncWrite(bufferData)  
    }  
  
Seq.init 1000 (fun num -> createSomeData num)  
|> Seq.map1 (fun num value ->  
    writeFile ("file" + num.ToString() + ".dat") value)  
|> Async.Parallel  
|> Async.RunSynchronously  
|> ignore
```

Подробнее про Async.Catch

F#

```
asyncTaskX
|> Async.Catch
|> Async.RunSynchronously
|> fun x ->
    match x with
    | Choice1Of2 result ->
        printfn "Async operation completed: %A" result
    | Choice2Of2 (ex : exn) ->
        printfn "Exception thrown: %s" ex.Message
```

Обработка исключений прямо внутри Async

F#

```
async {  
    try  
        // ...  
    with  
    | :? IOException as ioe ->  
        printfn "IOException: %s" ioe.Message  
    | :? ArgumentException as ae ->  
        printfn "ArgumentException: %s" ae.Message  
}
```


Отмена операции

Задача, которую можно отменить

F#

```
open System
open System.Threading

let cancelableTask =
    async {
        printfn "Waiting 10 seconds ..."
        for i = 1 to 10 do
            printfn "%d..." i
            do! Async.Sleep(1000)
        printfn "Finished!"
    }
```

Отмена операции

Код, который её отменяет

F#

```
let cancelHandler (ex : OperationCanceledException) =  
    printfn "The task has been canceled."  
  
Async.TryCancelled(cancelableTask, cancelHandler)  
|> Async.Start  
  
// ...  
  
Async.CancelDefaultToken()
```

CancellationToken

F#

```
open System.Threading

let computation = Async.TryCancelled(cancelableTask ,
                                     cancelHandler)
let cancellationSource = new CancellationTokenSource()

Async.Start(computation , cancellationSource.Token)

// ...

cancellationSource.Cancel()
```

Async.StartWithContinuations

F#

```
Async.StartWithContinuations(  
    someAsyncTask,  
    (fun result -> printfn "Task completed with result %A"  
    (fun exn ->  
        printfn "Task threw an exception with Message:  
                %s" exn.Message),  
    (fun oca -> printfn "Task was cancelled.  
                Message: %s" oca.Message)  
)
```

Async.FromContinuations

F#

```
open System
```

```
let trylet f x = (try Choice1Of2 (f x) with exn -> Choice2Of2(exn))
```

```
let protect cont econt f x =  
    match trylet f x with  
    | Choice1Of2 v -> cont v  
    | Choice2Of2 exn -> econt exn
```

```
type System.IO.Stream with  
    member stream.ReadAsync (buffer, offset, count) =  
        Async.FromContinuations (fun (cont, econt, cancel) ->  
            stream.BeginRead  
                (buffer = buffer,  
                 offset = offset,  
                 count = count,  
                 state = null,  
                 callback =  
                     AsyncCallback(protect cont econt stream.EndRead))  
            |> ignore)
```

Пример

Реализация Async.Parallel

F#

```
open System.Threading
```

```
let Parallel(taskSeq) =  
    Async.FromContinuations (fun (cont, econt, cancel) ->  
        let tasks = Seq.toArray taskSeq  
        let count = ref tasks.Length  
        let results = Array.zeroCreate tasks.Length  
        tasks |> Array.iteri (fun i p ->  
            Async.Start  
                (async { let! res = p  
                        do results.[i] <- res;  
                        let n = Interlocked.Decrement(count)  
                        do if n = 0 then cont results })))
```

Async.AwaitEvent

Для более простых случаев

F#

```
open System
```

```
let timer = new Timers.Timer(2000.0)
```

```
let timerEvent = Async.AwaitEvent (timer.Elapsed)  
    |> Async.Ignore
```

```
printfn "Waiting for timer at %O" DateTime.Now.TimeOfDay  
timer.Start()
```

```
printfn "Doing something useful while waiting for event"  
Async.RunSynchronously timerEvent
```

```
printfn "Timer ticked at %O" DateTime.Now.TimeOfDay
```

Большой пример: обработка “изображений”

Шаг 1: подготовка тестовых данных

F#

```
open System.IO

let numImages = 200
let size = 512
let numPixels = size * size

let MakeImageFiles() =
    printfn "making %d %dx%d images... " numImages size size
    let pixels = Array.init numPixels (fun i -> byte i)
    for i = 1 to numImages do
        File.WriteAllBytes(sprintf "Image%d.tmp" i, pixels)
    printfn "done."
```


Большой пример: обработка “изображений”

Шаг 2: вычислительно сложная функция над “изображением”

F#

```
let processImageRepeats = 20

let TransformImage(pixels , imageNum) =
    printfn "TransformImage %d" imageNum
    // Perform a CPU-intensive operation on the image.
    let mutable newPixels = pixels
    for i in 1..processImageRepeats do
        newPixels <- Array.map (fun b -> b + 1uy)) pixels
    newPixels
```

Большой пример: обработка “изображений”

Шаг 3: синхронная обработка

F#

```
let ProcessImageSync(i) =  
    use inStream = File.OpenRead(sprintf "Image%d.tmp" i)  
    let pixels = Array.zeroCreate numPixels  
    let nPixels = inStream.Read(pixels, 0, numPixels);  
    let pixels' = TransformImage(pixels, i)  
    use outputStream  
        = File.OpenWrite(sprintf "Image%d.done" i)  
    outputStream.Write(pixels', 0, numPixels)  
  
let ProcessImagesSync() =  
    printfn "ProcessImagesSync...";  
    for i in 1 .. numImages do  
        ProcessImageSync(i)
```

Большой пример: обработка “изображений”

Шаг 4: асинхронная обработка

F#

```
open Microsoft.FSharp.Control
open Microsoft.FSharp.Control.CommonExtensions

let ProcessImageAsync i =
    async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
            let! pixels = inStream.AsyncRead(numPixels)
            let pixels' = TransformImage(pixels, i)
            use outStream = File.OpenWrite(sprintf "Image%d.done" i)
            do! outStream.AsyncWrite(pixels') }

let ProcessImagesAsync() =
    printfn "ProcessImagesAsync..."
    let tasks = [ for i in 1 .. numImages -> ProcessImageAsync(i) ]
    Async.Parallel tasks |> Async.RunSynchronously |> ignore
    printfn "ProcessImagesAsync finished!"
```

Доклады

Как обычно, успешный доклад — -1 домашка. 2 недели на подготовку.

- ▶ Юнит-тестирование в F# (NUnit, FsUnit, FsCheck)
- ▶ Единицы измерения в F#
- ▶ Ленивые вычисления в F# (lazy, seq)
- ▶ Active patterns
- ▶ Code Quotations
- ▶ WebSharper