

Многопоточное программирование в F#

Юрий Литвинов
y.litvinov@spbu.ru

14.04.2023

Async workflow

```
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:

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...
```

Подробнее про Async

Async — это Workflow

```
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>
```

Какие конструкции поддерживает Async

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

Control.Async

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

Метод	Тип	Описание
<code>RunSynchronously</code>	<code>Async<'T> * ?int * ?CancellationToken -> 'T</code>	Выполняет вычисление синхронно, возвращает результат
<code>Start</code>	<code>Async<unit> * ?CancellationToken -> unit</code>	Запускает вычисление асинхронно, тут же возвращает управление
<code>Parallel</code>	<code>seq<Async<'T> > -> Async<'T []></code>	По последовательности Async-ов делает новый Async, исполняющий все Async-и параллельно и возвращающий массив результатов
<code>Catch</code>	<code>Async<'T> -> Async<Choice<'T,exn> ></code>	По Async-у делает новый Async, исполняющий Async и возвращающий либо результат, либо исключение
<code>StartImmediate</code>	<code>Async<unit> * ?CancellationToken -> unit</code>	Выполняет вычисление асинхронно в вызвавшем потоке
<code>StartAsTask</code>	<code>Async<'T> * ?TaskCreationOptions * ?CancellationToken -> Task<'T></code>	Запускает вычисление и оборачивает его в Task
<code>AwaitTask</code>	<code>Task<'T> -> Async<'T></code>	Оборачивает Task в Async

Пример

```
let writeFile fileName bufferData =  
    async {  
        use outputFile = System.IO.File.Create(fileName)  
        do! outputFile.AsyncWrite(bufferData)  
    }
```

```
Seq.init 1000 (fun num -> createSomeData num)  
|> Seq.mapi (fun num value ->  
    writeFile ("file" + num.ToString() + ".dat") value)  
|> Async.Parallel  
|> Async.RunSynchronously  
|> ignore
```

Подробнее про Async.Catch

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

```
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

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

```
let cancelHandler (ex : OperationCanceledException) =  
    printfn "The task has been canceled."
```

```
Async.TryCancelled(cancelableTask, cancelHandler)  
|> Async.Start
```

```
// ...
```

```
Async.CancelDefaultToken()
```

CancellationToken

open System.Threading

let computation = **Async**.TryCancelled(cancelableTask,
cancelHandler)

let cancellationSource = **new** CancellationTokenSource()

Async.Start(computation, cancellationSource.Token)

// ...

cancellationSource.Cancel()

Async.StartWithContinuations

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

Async.AwaitEvent

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
```


Взаимодействие с .NET

- ▶ Потребление Task из F#:

```
let getValueFromLibrary param =  
    async {  
        let! value =  
            DotNetLibrary.GetValueAsync param  
        |> Async.AwaitTask  
        return value  
    }
```

- ▶ Преобразование async в Task:

```
let computationForCaller param =  
    async {  
        let! result = getAsyncResult param  
        return result  
    } |> Async.StartAsTask
```

Прямое написание Task

```
let printTotalFileBytesUsingTasks (path: string) =
    task {
        let! bytes = File.ReadAllBytesAsync(path)
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }
```

```
[<EntryPoint>]
```

```
let main argv =
    let task = printTotalFileBytesUsingTasks "path-to-file.txt"
    task.Wait()
```

```
Console.Read() |> ignore
```

```
0
```

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

Агентно-ориентированный подход

- ▶ Давайте рассматривать параллельную программу как набор независимых последовательных агентов, общающихся сообщениями
- ▶ Сообщения постятся в очередь
- ▶ Агент достаёт сообщения из очереди и последовательно обрабатывает
- ▶ Преимущества:
 - ▶ Никаких (явных) блокировок
 - ▶ Разделение кода на производителей и потребителей
 - ▶ Слабая связность

MailboxProcessor

```
let printerAgent = MailboxProcessor.Start(fun inbox->
    let rec messageLoop() = async {
        // read a message
        let! msg = inbox.Receive()
        // process a message
        printfn "message is: %s" msg
        // loop to top
        return! messageLoop()
    }
    // start the loop
    messageLoop()
)
...
printerAgent.Post "hello"
```

© <https://fsharpforfunandprofit.com/posts/concurrency-actor-model/>

Более продвинутый пример

```

type MessageBasedCounter () =
    static let updateState (count,sum) msg =
        let newSum, newCount = sum + msg, count + 1
        printfn "Count is: %i. Sum is: %i" newCount newSum
        (newCount, newSum)

    static let agent = MailboxProcessor.Start(fun inbox ->
        let rec messageLoop oldState = async {
            let! msg = inbox.Receive()
            let newState = updateState oldState msg
            return! messageLoop newState
        }
        messageLoop (0, 0)
    )

    static member Add i = agent.Post i
  
```

Особенности

- ▶ Очень легковесны (можно иметь десятки тысяч агентов)
- ▶ Похожий подход применяется в Erlang
 - ▶ Но там агенты могут быть в разных процессах
- ▶ Сообщения не персистентны
 - ▶ Используйте RabbitMQ, ZeroMQ и т.д., если надо
- ▶ Есть PostAndReply, для удобного двустороннего обмена

BackgroundWorker

```
let worker = new BackgroundWorker()
```

```
let numIterations = 1000
```

```
worker.DoWork.Add(fun args ->
```

```
    let rec computeFibonacci resPrevPrev resPrev i =
```

```
        let res = resPrevPrev + resPrev
```

```
        if i = numIterations then
```

```
            args.Result <- box res
```

```
        else
```

```
            computeFibonacci resPrev res (i + 1)
```

```
computeFibonacci 1 1 2)
```

BackgroundWorker, как запустить

```
worker.RunWorkerCompleted.Add(fun args ->  
    MessageBox.Show (sprintf "Result = %A"  
        args.Result) |> ignore)
```

```
worker.RunWorkerAsync()
```


События

F# Interactive

```
> open System.Windows.Forms;;  
> let form = new Form(Text="Click Form",  
    Visible=true, TopMost=true);;  
val form : Form  
  
> form.Click.Add(fun evArgs -> printfn "Clicked!");;  
val it : unit = ()  
  
> form.MouseMove.Add(fun args -> printfn "Mouse,  
    (X,Y) = (%A,%A)" args.X args.Y);;  
val it : unit = ()
```

Microsoft.FSharp.Control.Event

Form.MouseMove

```
|> Event.filter (fun args -> args.X > 100)
|> Event.add (fun args -> printfn "Mouse,
(X,Y) = (%A,%A)" args.X args.Y)
```

Что ещё с ними можно делать

Примитив	Описание
add	$(T \rightarrow \text{unit}) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{unit}$
filter	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<T>$
choose	$(T \rightarrow U \text{ option}) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<U>$
map	$(T \rightarrow U) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<U>$
merge	$\text{IEvent}<\text{'Del1}, T> \rightarrow \text{IEvent}<\text{'Del2}, T> \rightarrow \text{IEvent}<T>$
pairwise	$\text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<T * T>$
partition	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<T> * \text{IEvent}<T>$
scan	$(U \rightarrow T \rightarrow U) \rightarrow U \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<U>$
split	$(T \rightarrow \text{Choice}<U1, U2>) \rightarrow \text{IEvent}<\text{'Del}, T> \rightarrow \text{IEvent}<U1> * \text{IEvent}<U2>$

Как описывать свои события

```
type RandomTicker(approxInterval) =  
    let timer, rnd = new Timer(), new System.Random 99  
    let tickEvent = new Event<_>()  
  
    let chooseInterval() :float =  
        approxInterval + approxInterval / 4 - rnd.Next(approxInterval / 2) |> float  
  
    do timer.Interval <- chooseInterval()  
  
    do timer.Elapsed.Add(fun args ->  
        let interval = chooseInterval()  
        tickEvent.Trigger(interval)  
        timer.Interval <- interval)  
  
    member x.RandomTick = tickEvent.Publish  
    member x.Start() = timer.Start()  
    member x.Stop() = timer.Stop()
```

Пример использования

F# Interactive

```
> let rt = new RandomTicker(1000);;  
val rt : RandomTicker  
> rt.RandomTick.Add(fun nextInterval -> printfn "Tick,  
    next = %A" nextInterval);;  
val it : unit = ()  
  
> rt.Start();;  
Tick, next = 1072  
Tick, next = 927  
Tick, next = 765  
...  
val it : unit = ()  
> rt.Stop();;  
val it : unit = ()
```

Особенности

- ▶ События не требуют языковой поддержки
 - ▶ Publish — относительно элегантный способ инкапсулировать источник события
- ▶ События рассматриваются как IEnumerable
 - ▶ Обычная ленивая последовательность, которую можно лениво преобразовывать, что гораздо гибче, чем в C# принято
 - ▶ Такой же подход используется в Rx.NET
 - ▶ И его тоже можно использовать из F#!
 - ▶ <https://github.com/fsprojects/FSharp.Control.Reactive>

Пример гонки на async-ах

open System.Threading

```
type MutablePair<'a,'b>(x:'a, y:'b) =  
    let mutable currentX = x  
    let mutable currentY = y  
    member p.Value = (currentX, currentY)  
    member p.Update(x, y) =  
        currentX <- x  
        currentY <- y
```

```
let p = MutablePair (0, 0)
```

```
Async.Start (async { while true do p.Update(10, 10) })
```

```
Async.Start (async { while true do p.Update(20, 20) })
```

```
Async.RunSynchronously (async { while true do printfn "%A" p.Value })
```

Монитор в F#

```
let lock (lockobj : obj) f =  
    Monitor.Enter lockobj  
    try  
        f()  
    finally  
        Monitor.Exit lockobj
```

```
Async.Start (async {  
    while true do lock p (fun () -> p.Update(10, 10)) })
```

```
Async.Start (async {  
    while true do lock p (fun () -> p.Update(20, 20)) })
```


Атомарные операции

- ▶ Нет синхронизации — нет deadlock-ов!
- ▶ Чтения и записи следующих типов всегда атомарны: Boolean, Char, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single, ссылочные типы
- ▶ Volatile
 - ▶ Volatile.Write
 - ▶ Volatile.Read
 - ▶ Связано с понятием Memory Fence, требует синхронизации ядер
 - ▶ Есть атрибут VolatileField
 - ▶ Volatile.Write должен быть последней операцией записи, Volatile.Read — первой операцией чтения

Пример

```
let mutable flag = 0
let mutable value = 0

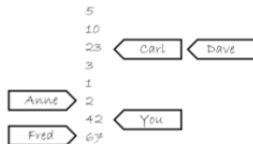
let thread1 () =
    value <- 5
    Volatile.Write(ref flag, 1)

let thread2 () =
    if Volatile.Read(ref flag) = 1
    then
        printfn "%d" value;
```

Синхронизация ядер, метафора

Relaxed ordering

- ▶ Каждую атомарную переменную можно понимать как список значений
- ▶ Каждый поток может спросить текущее значение, переменная вернёт ЛЮБОЕ значение из списка (текущее или одно из предыдущих)
- ▶ Переменная “запомнит”, какое значение она вернула этому потоку
- ▶ Когда поток спросит значение в следующий раз, она вернёт ЛЮБОЕ значение между текущим и последним, которое она вернула ЭТОМУ потоку



Interlocked

- ▶ Одновременные чтение и запись в одной “транзакции”
 - ▶ Increment : location:int byref -> int
 - ▶ Decrement : location:int byref -> int
 - ▶ Add : location:int byref * value:int -> int
 - ▶ Exchange : location:int byref * value:int -> int
 - ▶ CompareExchange
 - : location:int byref * value:int * comparand:int -> int
 - ▶ Read : location:**int64** byref -> **int64** — не нужен в x64
 - ▶ MemoryBarrier : **unit** -> **unit**

Interlocked lock-free-максимум

```
let maximum (target: int ref) value =  
    let mutable currentVal = !target  
    let mutable startVal = 0  
    let mutable desiredVal = 0  
    let mutable isDone = false  
    while not isDone do  
        startVal <- currentVal  
        desiredVal <- max startVal value  
        // Тут другой поток мог уже испортить target, так что если она изменилась,  
        // надо начать всё сначала.  
        currentVal <- Interlocked.CompareExchange(target, desiredVal, startVal)  
        if startVal = currentVal then  
            isDone <- true  
    desiredVal
```

Lock-free-список

```

type MutableList<'item when 'item: equality>(init) =
    let mutable items: 'item list = init

member x.Value = items

member x.Update updater =
    let current = items
    let newItems = updater current
    if not <| obj.ReferenceEquals
        (current, Interlocked.CompareExchange(&items, newItems, current))
    then x.Update updater
    else x

member x.Add item = x.Update (fun l -> item :: l)
member x.Remove item = x.Update (fun l -> List.filter (fun i -> i <> item) l)

static member empty = new MutableList<'item>([])

```

Проблема АВА

Не всё так просто

1. Поток 1 читает переменную x и видит A
2. Поток 1 выполняет операцию над A
3. Поток 1 засыпает
4. Поток 2 выставляет значение x в B
5. Поток 2 портит значение, ассоциированное с A
 - ▶ Например, затирает запись с ключом A в хеш-таблице или удаляет файл A
6. Поток 2 меняет x назад в A , но ассоциирует с A новое значение
 - ▶ Например, добавляет новую запись или создаёт новый файл A
7. Поток 1 просыпается и выполняет CompareExchange, видя A
 - ▶ Для него это то самое A , с которого он начал, так что всё падает

Пример

```

type LockFreeStack<'a>() =
    let mutable head: StackNode<'a> = Nil

    member this.Push (data: 'a) =
        let currentHead = head
        let newNode = Node(data, currentHead)
        if obj.ReferenceEquals
            (head, Interlocked.CompareExchange(&head, newNode, currentHead)) |> not
        then this.Push (data)

    member this.Pop () =
        let currentHead = head
        match currentHead with
        | Nil -> failwith "Stack empty"
        | Node (data, next) ->
            if obj.ReferenceEquals
                (head, Interlocked.CompareExchange(&head, next, currentHead)) |> not
            then this.Pop ()
            else data
  
```


Однако

```
let stack = LockFreeStack<int>()
```

```
Async.Start (async {  
    stack.Push 1  
    stack.Push 2 // Тут засыпаем после let currentHead = head  
})
```

// Тут просыпается поток 2

```
Async.Start (async {  
    stack.Pop () |> ignore // ...и скидывает 1  
    stack.Push 3 // ...и кладёт на её место 3  
})
```

// Поток 1 просыпается, ReferenceEquals true, и он затирает тройку

Итого

- ▶ Lock-free — когда несколько потоков могут получить доступ к структуре данных одновременно и гарантированно могут завершить операцию даже если остальные потоки сняты с исполнения
 - ▶ Опасность *голодания* — один поток в цикле делает своё дело, второй в цикле пытается снова и снова, и не успевает
- ▶ Wait-free — это Lock-free плюс гарантия, что все потоки закончат работу за ограниченное число шагов
- ▶ Lock-free и wait-free-алгоритмы могут быть в разы эффективнее алгоритмов с блокировкой
- ▶ Но в сотни раз сложнее и труднее в сопровождении
- ▶ В общем: избегайте lock-free, если нет веских причин поступить иначе!