

# Многопоточное программирование в F#

Юрий Литвинов

19.04.2019г

# 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
<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&lt;'T&gt;</code> и возвращает его как результат <code>Workflow</code>
<code>return! expr</code>	Возвращает <code>expr</code> типа <code>Async&lt;'T&gt;</code> как результат <code>Workflow</code>

# Control.Async

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

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



# 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()

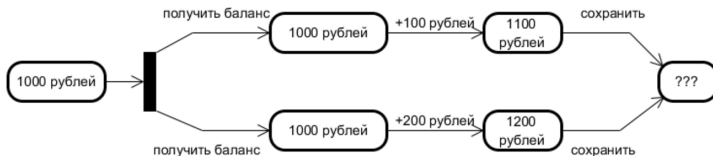
printfn "Doing something useful while waiting for event"
Async.RunSynchronously timerEvent

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

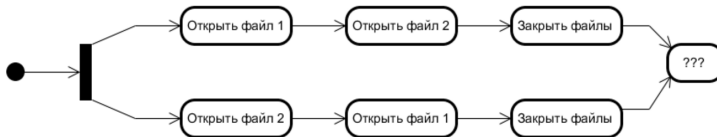


# Потенциальные проблемы с потоками

## ► Гонки (Race condition)



## ► Тупики (Deadlock)



## Пример гонки

**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 })
```

# Примитивы синхронизации

- ▶ Лучше необходимости синхронизации вообще избегать
- ▶ Бывают:
  - ▶ User-mode — атомарные операции, реализующиеся на процессоре и не требующие участия планировщика
  - ▶ Kernel-mode — примитивы, управляющие тем, как поток обрабатывается планировщиком
    - ▶ Более тяжеловесные и медленные (до 1000 раз по сравнению с “без синхронизации вообще”)
    - ▶ Позволяют синхронизировать даже разные процессы

# 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)) })
```

# Примитивы синхронизации

Пространство имён System.Threading

Примитив	Описание
AutoResetEvent	Точка синхронизации. WaitOne блокирует поток, пока кто-нибудь другой не вызовет Set.
ManualResetEvent	То же, что AutoResetEvent, но сбрасывается вручную, вызовом Reset
Monitor	Ограничивает доступ к критической секции
Mutex	Ограничивает доступ к критической секции, работает между процессами
Semaphore	Позволяет находиться в критической секции не более N потоков
Interlocked	Атомарные арифметические операции

# Управление планировщиком

- ▶ **Thread.Sleep(0)** — ничего не делает, если остальные готовые потоки меньше приоритетом
- ▶ **Thread.Sleep(1)** — отдаёт управление потоку, даже если его приоритет меньше
- ▶ **Thread.Yield()** — нечто среднее (не вызовет переключения потоков, если желающих нет, в отличие от **Thread.Sleep(1)**, но отдаст ядро потоку с меньшим приоритетом)
- ▶ **Thread.SpinWait()** — подождать в цикле, не переключая контексты
- ▶ Очередной способ прострелить себе ногу — инверсия приоритетов
  - ▶ Поток с низким приоритетом захватил ресурс, нужный потоку с высоким приоритетом
  - ▶ Поток с высоким приоритетом крутится в ожидании, никогда не отдавая управление потоку, который мог бы отдать ресурс (livelock)

# BackgroundWorker

Более высокоуровневый способ работы с потоками

```
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	$(T \rightarrow \text{unit}) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{unit}$
filter	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle T \rangle$
choose	$(T \rightarrow U \text{ option}) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
map	$(T \rightarrow U) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
merge	$\text{IEvent}\langle \text{'Del1'}, T \rangle \rightarrow \text{IEvent}\langle \text{'Del2'}, T \rangle \rightarrow \text{IEvent}\langle T \rangle$
pairwise	$\text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle T * T \rangle$
partition	$(T \rightarrow \text{bool}) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle T \rangle * \text{IEvent}\langle T \rangle$
scan	$(U \rightarrow T \rightarrow U) \rightarrow U \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle U \rangle$
split	$(T \rightarrow \text{Choice}\langle U1, U2 \rangle) \rightarrow \text{IEvent}\langle \text{'Del'}, T \rangle \rightarrow \text{IEvent}\langle U1 \rangle * \text{IEvent}\langle U2 \rangle$

# Как описывать свои события

```
open System
```

```
open System.Windows.Forms
```

```
type RandomTicker(approxInterval) =
```

```
    let timer = new Timer()
```

```
    let rnd = new System.Random 99
```

```
    let tickEvent = new Event<_>()
```

```
    let chooseInterval() :int =
```

```
        approxInterval + approxInterval / 4  
        - rnd.Next(approxInterval / 2)
```

```
    do timer.Interval <- chooseInterval()
```

## Как описывать свои события (2)

```
do timer.Tick.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()  
  
interface IDisposable with  
    member x.Dispose() = timer.Dispose()
```

## 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 = ()
```

## Свой worker, с событиями

```
open System.ComponentModel
open System.Windows.Forms
```

```
type IterativeBackgroundWorker<'a>(oneStep:('a -> 'a),
    initialState:'a,
    numIterations:int) =
    let worker =
        new BackgroundWorker(WorkerReportsProgress = true,
            WorkerSupportsCancellation = true)

    let completed = new Event<_>()
    let error = new Event<_>()
    let cancelled = new Event<_>()
    let progress = new Event<_>()
```

## Свой worker (2)

```
do worker.DoWork.Add(fun args ->
let rec iterate state i =
    if worker.CancellationPending then
        args.Cancel <- true
    elif i < numIterations then
        let state' = oneStep state
        let percent = int ((float (i + 1)
            / float numIterations) * 100.0)
        do worker.ReportProgress(percent, box state);
        iterate state' (i + 1)
    else
        args.Result <- box state
```

```
iterate initialState 0)
```



## Свой worker (3)

```
do worker.RunWorkerCompleted.Add(fun args ->  
    if args.Cancelled then cancelled.Trigger ()  
    elif args.Error <> null then error.Trigger args.Error  
    else completed.Trigger (args.Result :?> 'a))
```

```
do worker.ProgressChanged.Add(fun args ->  
    progress.Trigger (args.ProgressPercentage, (args.UserState :?> 'a)))
```

```
member x.WorkerCompleted = completed.Publish
```

```
member x.WorkerCancelled = cancelled.Publish
```

```
member x.WorkerError = error.Publish
```

```
member x.ProgressChanged = progress.Publish
```

```
member x.RunWorkerAsync() = worker.RunWorkerAsync()
```

```
member x.CancelAsync() = worker.CancelAsync()
```

## Тип того, что получилось

```

type IterativeBackgroundWorker<'a> =
  class
    new : oneStep:('a -> 'a)
      * initialState:'a
      * numIterations:int
      -> IterativeBackgroundWorker<'a>
    member CancelAsync : unit -> unit
    member RunWorkerAsync : unit -> unit
    member ProgressChanged : Event<int * 'a>
    member WorkerCancelled : Event<unit>
    member WorkerCompleted : Event<'a>
    member WorkerError : Event<exn>
  end

```

## Пример использования

```
let fibOneStep (fibPrevPrev:bigint,fibPrev) =  
    (fibPrev, fibPrevPrev + fibPrev)
```

```
let worker = new IterativeBackgroundWorker<_>(fibOneStep,  
    (1I, 1I), 100)
```

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

```
worker.ProgressChanged.Add(fun (percentage, state) ->  
    printfn "%d%% complete, state = %A" percentage state)
```

```
worker.RunWorkerAsync()
```

# Своё новое событие

open System

open System.Threading

type IterativeBackgroundWorker<'a>(…) =

let worker = ...

let syncContext = SynchronizationContext.Current

do if syncContext = null then failwith  
"no synchronization context found"

let started = new Event<\_>()

do worker.DoWork.Add(fun args ->  
syncContext.Post(SendOrPostCallback(fun \_ ->  
started.Trigger(DateTime.Now)),  
state= null))

...

member x.Started = started.Publish

# 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

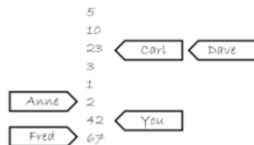
let thread1 () =
    value <- 5
    Volatile.Write(ref flag, 1)

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 : location1:int byref \* value:int -> int
  - ▶ Exchange : location1:int byref \* value:int -> int
  - ▶ CompareExchange  
: location1:int byref \* value:int \* comparand:int -> int



# Interlocked lock-free-максимум

```
let maximum target 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(ref target, desiredVal, startVal)  
        if startVal = currentVal then  
            isDone <- true  
    desiredVal
```