Низкоуровневые потоки, события

Юрий Литвинов

06.05.2016г

Управление потоками “вручную”

- Сложнее и опаснее, чем async
- Позволяет управлять приоритетом потока, делать поток foreground или background, точнее управлять временем жизни и поведением потока, использовать низкоуровневые механизмы синхронизации
- Абстракция отдельного ядра, а не асинхронного вычисления

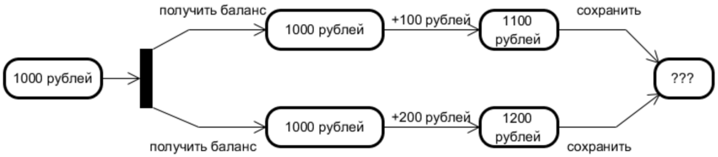
Класс System.Threading.Thread

F#

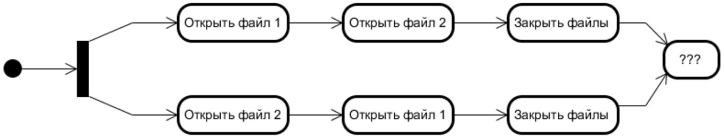
```
open System.Threading

let t = new Thread(ThreadStart(fun _ ->
    printfn "Thread %d: Hello"
            Thread.CurrentThread.ManagedThreadId))

t.Start()
printfn "Thread %d: Waiting!"
            Thread.CurrentThread.ManagedThreadId
t.Join()
printfn "Done!"
```

Потенциальные проблемы с потоками

- Гонки (Race condition)



- Тупики (Deadlock)



Пример гонки

F#

```
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 () })
```

Пример примитива синхронизации: монитор

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

BackgroundWorker

Более высокоуровневый способ работы с потоками

F#

```
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

F#

```
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

F#

```
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 → unit) → IEvent<'Del,'T> → unit
filter	('T → bool) → IEvent<'Del,'T> → IEvent<'T>
choose	('T → 'U option) → IEvent<'Del,'T> → IEvent<'U>
map	('T → 'U) → IEvent<'Del, 'T> → IEvent<'U>
merge	IEvent<'Del1,'T> → IEvent<'Del2,'T> → IEvent<'T>
pairwise	IEvent<'Del,'T> → IEvent<'T * 'T>
partition	('T → bool) → IEvent<'Del,'T> → IEvent<'T> * IEvent<'T>
scan	('U → 'T → 'U) → 'U → IEvent<'Del,'T> → IEvent<'U>
split	('T → Choice<'U1,'U2>) → IEvent<'Del,'T> → IEvent<'U1> * IEvent<'U2>

Как описывать свои события

F#

```
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)

F#

```
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

F#

```
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)

F#

```
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)

F#

```
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

F#

```
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

F#

```
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

F#

```
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
```

Большой пример (1)

F#

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

let form = new Form(Visible = false, TopMost = true)

let panel = new FlowLayoutPanel(Visible = true,
    Height = 20,
    Dock = DockStyle.Bottom,
    BorderStyle = BorderStyle.FixedSingle)

let progress = new ProgressBar(Visible = false,
    Anchor=(AnchorStyles.Bottom ||| AnchorStyles.Top),
    Value = 0)

let text = new Label(Text = "Paused",
    Anchor = AnchorStyles.Left,
    Height = 20,
    TextAlign = ContentAlignment.MiddleLeft)
```

Большой пример (2)

F#

```
panel.Controls.Add(progress)
panel.Controls.Add(text)
form.Controls.Add(panel)

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

// Run the iterative algorithm 500 times before reporting intermediate res
// Burn some additional cycles to make sure it runs slowly enough
let rec RepeatN n f s = if n <= 0 then s else RepeatN (n - 1) f (f s)
let rec BurnN n f s = if n <= 0 then f s else ignore (f s); BurnN (n - 1)
let step = (RepeatN 500 (BurnN 1000 fibOneStep))

// Create the iterative worker.
let worker = new IterativeBackgroundWorker<_>(step,(1I,1I),100)
```

Большой пример (3)

F#

```
worker.ProgressChanged.Add(fun (progressPercentage,state)->
        progress.Value <- progressPercentage)

worker.WorkerCompleted.Add(fun (_,result) ->
        progress.Visible <- false;
        text.Text <- "Paused";
        MessageBox.Show(sprintf "Result = %A" result) |> ignore)

worker.WorkerCancelled.Add(fun () ->
        progress.Visible <- false;
        text.Text <- "Paused";
        MessageBox.Show(sprintf "Cancelled OK!") |> ignore)

worker.WorkerError.Add(fun exn ->
        text.Text <- "Paused";
        MessageBox.Show(sprintf "Error: %A" exn) |> ignore)
```

Большой пример (4)

F#

```
form.Menu <- new MainMenu()
let workerMenu = form.Menu.MenuItems.Add("&Worker")

workerMenu.MenuItems.Add(new MenuItem("Run",onClick=(fun _ args ->
        text.Text <- "Running";
        progress.Visible <- true;
        worker.RunWorkerAsync()))) |> ignore

workerMenu.MenuItems.Add(new MenuItem("Cancel",onClick=(fun _ args ->
        text.Text <- "Cancelling";
        worker.CancelAsync()))) |> ignore

form.Closed.Add(fun _ -> worker.CancelAsync())

form.ShowDialog () |> ignore
```