实战 Swift 函数式编程

王文槿（@aaaron7，小莲，莲叔）

有

- 函数式编程介绍
- 以及一个(dui)例子：
  - 函数式的解决异步问题（串行，并行）
  - 发明两个人见人爱的运算符 “+>”“<>”
  - 实现一个类 Promise 的异步 API

没有

- RxSwift，ReativeCocoa 使用指南
- Monad 是什么，拿来干嘛的
- 函数式编程的理论范畴：代数系统，范畴论
- 跟 optional 和 flatMap 较劲

什么是函数式编程

- 三大讲究：
- 1. 引用透明（无副作用，Immutability）
- 2. 函数是一等公民
- 3. Lambda Calculus

大白话版

- 没有语句(Statement)，只有表达式(Expression)。
- 没有变量，同时也就没有赋值操作……自然，for循环之类也是没有的。
- 函数可以以函数作为参数，也能够返回函数。

纯函数式编程的缺陷

- 现实世界到处都是副作用(Side Effect)，比如最质朴的 IO 访问；

- 为了处理实际问题，引入了一系列复杂的机制（Monad，Monad Transformer）来封装状态传递，IO 读写等场景；

- 学习曲线陡峭，只适用于部分场景；

Swift 的函数式特性

- Swift 具备的函数式特性：
  - Currying；
  - 函数是一等公民；
  - Closure；
  - 类型推导；

- 缺乏尾递归优化，以及 By Default 的 lazy evaluation 等特性

类型是人类的好朋友

- var x = 5 ==> var x:Int = 5
- var x; ==> 报错
- let d = {x in x + 1} ==> let d:(Int->Int) = {x:Int -> Int in x + 1}
- let d = {x in x} ==> 报错

感受一下

- 首先我们有三个函数：
  let procIndex = {$0 + 3}

let convertName = {(x:Int) -> String in "Image_\(x)" }

let createImage = {UIImage(named: $0)}

传统的方式

- var num = 0

let imageIndex = procIndex(num)

let imageName = convertName(imageIndex)

let imageObject = createImage(imageName)

函数式的方式

- let createImageFromIndex = createImage <-> convertName <-> procIndex

  createImageFromIndex(num)

- func <-><a,b,c>(g : b->c , f : a -> b) -> (a->c)
  {
  return { x in
  let result1 = f(x)
  return g(result1)
  }
  }

异步的问题

- 串行
- 并行
- 交织进行

还是从类型开始

- typealias AsyncFunc = (()->Void) -> Void
- 假设异步操作都是只有一个参数的函数，参数就是该操作执行完毕后的 callback 函数。

- let getUserProfile : AsyncFunc = {complete in
  ….一些很耗时的操作，科科….
  complete()
  }
- getUserProfile{print “Got it!”}

问题来了

- asyncJob1{
  asyncJob2{
  asyncJob3{
  .....
  }
  }
  }

缺点?

- 层次太多，难以维护；
- 靠 callback 来串联，耦合度极高；
- 中间的某一个或多个环节很难独立处理，复用到其他模块；

折叠

- 能否把两个异步操作折叠成一个
- 如果能折叠两个，则就能折叠任意多个
- func fold(left : AsyncFunc , right : AsyncFunc) -> AsyncFunc{

  }

实现折叠

- 既然返回的是函数，那大概，约莫应该是长这样：

- ```
  func fold(left : AsyncFunc , right : AsyncFunc) -> AsyncFunc{
      return { complete in

     }
  }
  ```
  ```

实现折叠（续）

```
func fold(left : AsyncFunc , right : AsyncFunc) ->
AsyncFunc{
        return { complete in
                left{
                    right{
                        complete()
                    }
                }
        }
    }
```

改造一下最初的例子

- let temp = fold(asyncJob1, right: asyncJob2)

let finalJob = fold(temp, right: asyncJob3)

finalJob({ print “all task finished”; })

如果折叠是一种运算

- 回头看看 fold 的签名：
  (AsyncFunc, AsyncFunc)->AsyncFunc
- 满足结合律: fold (fold(a,b), c) == fold(a, fold(b,c))
- 不满足交换律: fold(a,b) != fold(b,a)

中置运算符的魅力

- infix operator +> {associativity left precedence 150}

- let finalJob = asyncJob1 +> asyncJob2 +> asyncJob3
  findJob {print “all job finished”}

如果我们有一组异步操作

- let arr:[AsyncFunc] = [asyncJob1, …,asyncJobN]
- 那么问题来了，现在已经有了神奇的 +> 运算符，那我们能对上面的 list 做 reduce 吗?
- let reducedFunction = arr.reduce(【初始值】, combine: +>)

这是一个哲学问题

- 我们需要一个参与运算，但又不影响运算结果的“Identity Value”， 比如0之于加法，1之于乘法

- let identityFunc:AsyncFunc = {f in f()}

- let reducedFunction = arr.reduce(identityFunc, combine: +>)

  reducedFuncton{print “all item in list finished”}

现实一点

- 参数传递 — 后序操作可能依赖于前序操作
- 错误处理，折叠成一个之后，如果中间某个环出错了怎么办?

let’s do it

- typealias AsyncFunc = (info : AnyObject,complete:(AnyObject?,NSError?)->Void) -> Void

```
func +>(left : AsyncFunc , right : AsyncFunc) -> AsyncFunc{
    return { info , complete in
        left(info: info){ result,error in
            guard error == nil else{
                complete(nil,error)
                return
            }
            right(info: result){result,error in
                complete(result,error)
            }
        }
    }
}
```

说明

- 不管串了多少个，一旦中间有哪个出错了，都会停止余下的执行，不断通过回调 complete 冒泡到最终的complete，并携带有错误信息；
- 第 N 个操作可以访问所有小于 N 的操作的返回值（由 info 逐级往下带）
- AnyObject 只是一个示例的载体，实际情况可以是根据业务场景设计的 protocol

串行是有了，那并行呢?

- 比如有一组任务，需要并发执行，当所有任务都成功执行完毕后，调用 callback
- 我们是否能像解决串行那样的思路简单的解决并行呢?

并没有太多不同

```
func <>(left : AsyncFunc , right : AsyncFunc) -> AsyncFunc{
    return { complete in
        var leftComplete = false
        var rightComplete = false

        let checkComplete = {
            if leftComplete && rightComplete{
                complete()
            }
        }

        left{
            leftComplete = true
            checkComplete()
        }

        right{
            rightComplete = true
            checkComplete()
        }
    }
}
```

一个并行的小例子

```
let delay = dispatch_time(DISPATCH_TIME_NOW, Int64(NSEC_PER_SEC))

let test1:AsyncFunc = { complete in
    print("test1")

    dispatch_after(delay, dispatch_get_main_queue(), {
        complete();
    })
}

let test2:AsyncFunc = { complete in
    print("test2")

    dispatch_after(delay, dispatch_get_main_queue(), {
        complete();
    })
}

let test = test1 <> test2;

test{print("all finished")};
```

What’s next?

有没有更加友好的封装

- 站在巨人的肩膀上，或许参(shan)考(zhai)一下 Promise 是个不错的选择
- PromiseKit 在 GitHub ReadMe 的例子：

```
firstly {
    when(NSURLSession.GET(url).asImage(), CLLocationManager.promise())
}.then { image, location -> Void in
    self.imageView.image = image
    self.label.text = "\(location)"
}.always {
    UIApplication.sharedApplication().networkActivityIndicatorVisible = false
}.error { error in
    UIAlertView(/*…*/).show()
}
```

抽取一些关键的特性

- 通过 firstly注册第一个任务，并返回一个 Promise对象。用于后面的链式代码书写。
- then 可以有任意多个，顺序执行。then 块中直接用同步的方式写代码。
- 不管执行过程中是否出错，都会执行 always 块
- 如果执行过程中出错，则执行 error 块。
- PS：Promise 的实现机制在此不做讨论，仅参考 API 的设计思路。

从 firstly 开始

- 首先 firstly 是一个函数，接受一个同步的 closure，并返回一个 Promise 对象（这样才有后面的 then）

```
func firstly(body : Void->Void)->Promise{

    let starter: AsyncFunc = { _,complete in
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0)) {
            body();
            complete(0,nil);
        }
    }

    return Promise(starter: starter)
}
```

把同步 closure 打包并注册给 Promise 对象

Promise的设计

- 首先要提供一个 designate initializer来接收第一个任务

- 要保存 always 块和 error 块，需要时调用。

- 要实现 chainable 的 then block，需要记录一个每次 then 中传递进来的任务，所以内部需要维护一个异步任务的 accumulator

```
class Promise {
    var chain : AsyncFunc
    var alwaysClosure : (Void->Void)?
    var errorClosure : (NSError?->Void)?

    init(starter : AsyncFunc){
        chain = starter
    }

    func then(body : AnyObject throws->Void )->Promise{
        let async: AsyncFunc = { info, complete in
            dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0)) {
                body(info)
                complete(0,nil)
            }
        }
        chain = chain +> async
        return self
    }

    func always(closure : Void->Void)->Promise{
        alwaysClosure = closure
        return self
    }

    func error(closure : NSError?->Void)->Promise{
        errorClosure = closure
        fire()
        return self
    }
}
```

所有 job 的 accumulator

类似 firstly 的形式，把同步代码打包成异步任务

通过 +> 把新的任务叠加到 accmulator 上

注册完 error 后自动触发任务

如何进行错误的处理?

- 重写 then 的实现

```
func then(body : AnyObject throws->Void )->Promise{
    let async: AsyncFunc = { info, complete in
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0)) {
            var error : NSError?
            do{
                try body(info)
            }catch let err as NSError{
                error = err
            }
            complete(0,error)
        }
    }
    chain = chain +> async
    return self
}
```

执行！

```
func fire(){
    chain(info: 0) { (info, error) in
        if let always = self.alwaysClosure{
            always()
        }

        if error == nil{
            print("all task finished")
        }else{
            if let errorC = self.errorClosure{
                errorC(error)
            }
        }
    }
}
```

激动人心的时刻来临了

```
firstly {
    print("firstly")
    sleep(5)
}.then { info in
    print("second job")
}.then { info in
    print("third job")
}.always {
    print("always block")
}.error { error in
    print("error occurred")
}
```

输出：

firstly

（间隔5秒）

second job

third job

always block

激动人心的时刻还在继续

```
firstly {  in
    print("firstly")
    sleep(5)
}.then { info in
    print("second job”)
    throw NSError(domain: "error",
                  code: 0, userInfo: [:])
}.then { info in
    print("third job")
 }.always {  in
    print("always block")
 }.error { error in
    print("error occurred")
 }
```

输出：

firstly

（间隔5秒）

second job

always block

error occurred

基本功能都有了吗

- 回顾一下 PromiseKit 的例子：

```
firstly {
    when(NSURLSession.GET(url).asImage(),
            CLLocationManager.promise())
}.then { image, location -> Void in ..
```

- when 处理并行的异步请求

让脑洞再开一会儿

- when 函数应该有两个参数，接受两个同步的闭包。然后内部封装成异步
- when 函数内部需要处理等待请求返回后才返回的逻辑，因为 when 本身会在 promise 链中执行（全异步）。所以简单阻塞即可
- when 本身不需要和 promise 打交道
- 这次该轮到 <> 出场了。

```
func when(fstBody : (Void->Void), sndBody : (Void->Void)){
    let async1 : AsyncFunc = { _ , complete in
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0)) {
            fstBody();
            complete(0,nil);
        }
    }

    let async2 : AsyncFunc = { _ , complete in
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0)) {
            sndBody();
            complete(0,nil);
        }
    }

    let async = async1 <> async2

    var finished = false

    async(info: 0) { (_, _) in
        finished = true
    }

    while finished == false {

    }
}
```

老规矩，分别把同步打包成异步

合并两个异步

调用并阻塞等待

真正激动人心的时刻来了

```
firstly { () in
    when({ () in
        print(“begin fst job")
        sleep(1)
        print("fst job in when finished")
        }, sndBody: { () in
            print(“begin snd job")
            sleep(5)
            print("snd job in when finished")
    })
}.then { (info) in
    print("second job")
    throw NSError(domain: "error", code: 0, userInfo: [:])
}.then { (info) in
    print("third job")
}.always { () in
    print("always block")
}.error { (error) in
    print("error occurred")
}
```

输出：

begin fst job
begin snd job
(间隔1秒)
fst job in when finished
(间隔4秒)
snd job in when finished
second job
always block
error occurred

回顾一下

- 函数式编程的一些基本概念
- 折叠串行异步操作的原理，并实现了 +>
- 折叠并行异步操作的原理，并实现了 <>
- 参数的传递与错误的处理
- 基于 +> 和 <>, 实现了一个类 promise 的接口

- 谢谢！