

函数式的设计模式

aaaron7

我是 sei?

- 王文槿, ID: aaron7;
- SwiftGG 翻译组CTO (Chief Talking Officer), 承担主要的灌水工作;
- 目前供职于 UCWeb 浏览器研发组;
- 酷爱 Swift, 尤其是 Functional part。曾写作《Swift 脑洞系列-轻松无痛实现异步操作》、《Swift 脑洞系列-并行异步运算与100行的 ·PromiseKit·》

大纲

- 1. 为什么要聊 FP
- 2. FP 的设计模式：OO 视角
- 3. 真·FP 风格的设计模式

为什么要聊 FP

I ❤️ Objective-C

但是.....

用 OO 来写软件

shooting bird with a missile

刚才不够 OO

shooting `id<IFlyableAnimal>` with
an `AbstractMissileProxyFactory`

解放生产力的根源 — 抽象

抽象的层次



FP 时代

程序员思考如何描述问题

OO 时代

程序员开始思考如何把问题抽象成对象和关系

GC/ARC 时代

程序员专注思考实现逻辑的步骤

C 时代

程序员需要思考如何管理内存

汇编时代

程序员需要思考如何利用有限的寄存器

FP 的设计模式

OO 视角

“FP 世界中，传统 OO 的模式还好使吗？”

OO 的设计模式

FP 中的实现

Factory

Functions(Curry/HOF)

Adapter,Decorator

Functions

Iterator Pattern

Functions(HOF)

Strategy

Functions(HOF)

Facade

Functions

Command

Functions

感觉有点不太对？

- 看上去，OO 的大多数设计模式，都能在 FP 中实现；
- 但 FP 是否应该照搬 OO 的模式来进行设计呢？

设计模式源于设计原则

OO 的 S.O.L.I.D 设计原则：

- S – Single-responsibility principle
- O – Open-closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – Dependency Inversion Principle

设计模式源于设计原则

FP 的设计原则:

- 引用透明, 无副作用
(referential transparent)
- 保持不可变性
(immutability)
- 函数之间使用显式组合
(Explicit Composition)
- 使用静态类型来问题建模以及文档
(Use static type modelling problem and documentation)

虽然不知道你在说啥

但感觉两边差得还挺大

真·FP 风格的设计模式



FP Design Pattern

- 使用函数的正确姿势
 - Functions for decoupling
- 异常处理/异步
 - Monad
- GUI / 事件驱动的开发
 - Functional Reactive Programming
- Embedded DSL/Interpreter 模式
 - Parser Combinator

Functions for decoupling

Functions for decoupling

```
func outputTree(root : Node<String>)  
{  
    print(root.value)  
  
    if let left = root.left{  
        outputTree(root: left)  
    }  
  
    if let right = root.right{  
        outputTree(root: right)  
    }  
}
```

```
func outputTree(root : Node<String>)  
{  
    saveToDB(root.value)  
  
    if let left = root.left{  
        outputTree(root: left)  
    }  
  
    if let right = root.right{  
        outputTree(root: right)  
    }  
}
```

耦合到了 DB 模块

Functions for decoupling

```
func outputTreeEx(root : Node<String>, output : ((String) -> Void))
{
    output(root.value)

    if let left = root.left{
        outputTree(root: left)
    }

    if let right = root.right{
        outputTree(root: right)
    }
}
```


outputTreeEx 函数所在的模块没有必要知道 DB 的存在

Functions for decoupling

现在问题来了，实际上我们写 DB 的函数，类型为：
(DBConnection, String)->Void

```
func saveStringToDB(db : DBConnection, _ value : String)-> Void  
{  
    db.saveString(value: value)  
}
```

类型不符，如何注入？



outputEx : (Node, (String->Void)) -> Void

Functions for decoupling

用闭包bridge 一下?

```
outputTreeEx(root: root) { (str) in
    saveStringToDB(db: db, str)
}
```

如果还有其他 handler function 需要 String->Void?



bang!

“让我来试试”，Swift 中最寂寞的英雄：Curry 起身。

```
func saveStringToDB(db : DBConnection)->(_ value:String) -> Void
{
    return { value in
        return db.saveString(value)
    }
}
```

$(DBConnection, String) \rightarrow Void \implies DBConnection \rightarrow String \rightarrow Void$

Functions for decoupling

```
outputTreeEx(root: root, output: saveStringToDB(db: db))
```

成功注入，一颗赛艇

<code>saveStringToDB:</code>	<code>DBConnection -> String -> Void</code>
<code>saveStringToDB(db):</code>	<code>String -> Void</code>
<code>saveStringToDB(db)("save me"):</code>	<code>Void</code>

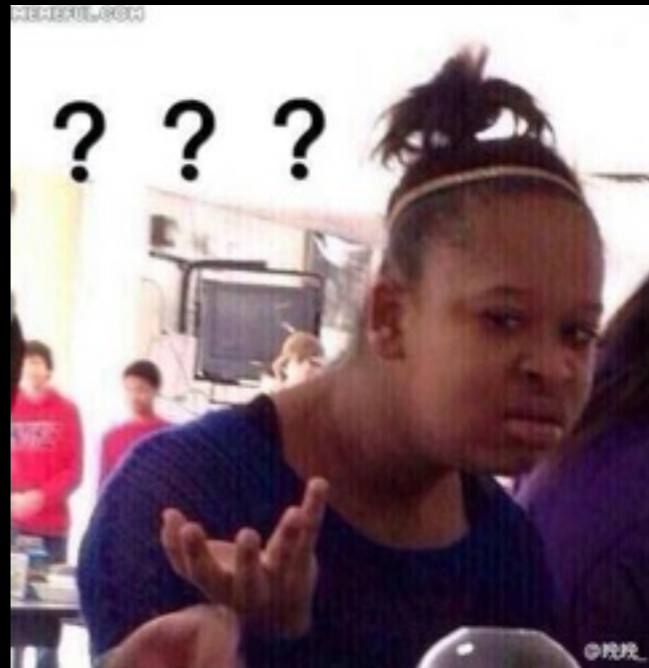
Functions for decoupling

- 对外部的依赖，尽量使用函数参数的形式注入。
- 通过合理使用 **Currying** 来解除函数的依赖。
- 不管是 **OO** 还是 **FP**，根据职责来分层都是必要的，所以才需要解决依赖的问题

使用 `Monad` 来异常处理

使用 Monad 实现异常处理

Monad?



从一个实际问题开始

```
func loadStringFromDB(db : DBConnection) -> String
```

如果失败了，该返回什么？

传统做法：

1. Win32时代，设全局 error code
2. OC 时代，传入 &error
3. Swift 时代，Optional Value

我们先暂时忘掉 Optional

enum as ADT

归根结底，我们需要一个类型来表示“可失败”的值，当成功时，该类型返回具体的值，失败时，该类型可以返回失败的标记

```
enum Maybe<a>
{
    case None
    case Some(a)
}

func loadStringFromDB(db : DBConnection) -> Maybe<String>

let result = loadStringFromDB(db: db)
if case .Some(let x) = result
{
    print("Oh yeah, I got \(x)")
}else{
    print("Shit, im failed")
}
```

问题似乎解决了，可是.....

错误处理

```
func loadStringFromDB(db : DBConnection) -> Maybe<String>
```

```
func getUserByName(db : DBConnection)->(name:String) -> Maybe<User>
```

```
func getUserScores(db : DBConnection)->(user:User) -> Maybe<Float>
```

```
let result = loadStringFromDB(db: db)
if case .Some(let x) = result
{
    let user = getUserByName(db: db)(x)
    if case .Some(let y) = user
    {
        let score = getUserScores(db: db)(y)
        if case .Some(let z) = score
        {
            print ("Finally got here");
        }
    }
}
```

这代码安全吗? 安全

看着蛋疼吗? 相当蛋疼

错误处理

把刚才流程符号化：

$() \rightarrow \text{Maybe}\langle \text{String} \rangle \implies$
 $\text{String} \rightarrow \text{Maybe}\langle \text{User} \rangle \implies$
 $\text{User} \rightarrow \text{Maybe}\langle \text{Float} \rangle$

能不能我们写代码也像这样按顺序写下来就好了，别整那么多 `if`。然后错误自动被处理呢？

那 `\implies` 的类型是什么呢？

第一个 `\implies`：

$(\text{Maybe}\langle \text{String} \rangle, (\text{String} \rightarrow \text{Maybe}\langle \text{User} \rangle)) \rightarrow \text{Maybe}\langle \text{User} \rangle$

第二个 `\implies`：

$(\text{Maybe}\langle \text{User} \rangle, (\text{User} \rightarrow \text{Maybe}\langle \text{Float} \rangle)) \rightarrow \text{Maybe}\langle \text{Float} \rangle$

$(\text{Maybe}\langle a \rangle, a \rightarrow \text{Maybe}\langle b \rangle) \rightarrow \text{Maybe}\langle b \rangle$

神奇的`==>`

```
func bind<a,b>(left : Maybe<a>, f : ((a)->Maybe<b>)) -> Maybe<b>
{
  if case .Some(let x) = left
  {
    return f(x)
  }
  else
  {
    return .None
  }
}
```

```
let r1 = loadStringFromDB(db: db)
let r2 = bind(left: r1, f: getUserByName(db: db))
let r3 = bind(left: r2, f: getUserScores(db: db))
```

不知道你们发现没，`curry`又乱入了.....

顺序书写成就达成

```
let r1 = loadStringFromDB(db: db)
let r2 = bind(left: r1, f: getUserByName(db: db))
let r3 = bind(left: r2, f: getUserScores(db: db))
```

- 1. `getUserByName/getUserScores`，都不关心自己的参数是否合法，因为 `bind` 保证了只有参数合法才会调用；
- 2. 中间任何环节出错，程序照常运行，`r3` 也会如期为 `.None`；
- 3. 本质：我们把一个不确定的值“塞”到了一个只接受确定值的函数中；
- 4. 优点：我们书写确定、简单的函数 (**bug free**)，通过 `bind` 机制来用这些函数去处理真实环境中不确定的值 (安全)

Maybe 和 Optional

故事讲到这里，大家应该都明白了。

Maybe 就是 Optional
而 bind 就是 flatMap

Maybe<a>, Maybe

a/b 为泛型变量，代表某一个具体的类型。比如 Int， 或 String。

Maybe 代表一个“上下文”。其他常见的上下文还有 Array<a>。

同样的 a，在不同的上下文中有不同的语义。

如果上下文也用变量来表示呢？

Monad

- 给定一个上下文 M ，如果我们能为其定义两个方法：
- $\text{pure} : a \rightarrow M\langle a \rangle$
- $\text{bind} : (M\langle a \rangle, a \rightarrow M\langle b \rangle) \rightarrow M\langle b \rangle$
- 则这个上下文 M 就是一个 Monad

在 Swift 中， M 的落地可以是 `struct/class/enum/function`
也就是任意支持泛型参数的结构

我们的 Maybe 是 Monad 吗?

```
func pure<a>(x : a) -> Maybe<a>
{
  return Maybe.Some(x)
}
```

```
func bind<a,b>(left : Maybe<a>, f : ((a)->Maybe<b>)) -> Maybe<b>
{
  if case .Some(let x) = left
  {
    return f(x)
  }
  else
  {
    return .None
  }
}
```

- **bind** 在刚才我们改造包含异常的顺序结构代码时起到了关键的作用;
- **bind** 同样是 **Monad** 的核心组成部分;
- 莫非.....这一切都是 **Monad** 的功劳?

Monad 的作用

- 从之前的 Monad Laws 可以得知，构造一个 Monad 其实很容易；
- 关键是要构造有用的 Monad，bind 提供了 monad 的核心语义；
- 从结构上看，Monad 一般是用来“串联”逻辑块的。
- 我们的程序本身就是顺序执行的，为何还需要 Monad 来串联？
- 现实世界中，程序在异常处理、异步调用、IO 操作等场景往往无法用顺序结构的代码来表达；
- Monad 提供一个更高层次的抽象，对“同构”场景进行封装。实现顺序结构来“表达”真实的问题；
- Monad 组合简单函数解决问题，通过形式约束了代码出 BUG 的概率；

GUI/事件驱动的开发

— FRP

Why FRP

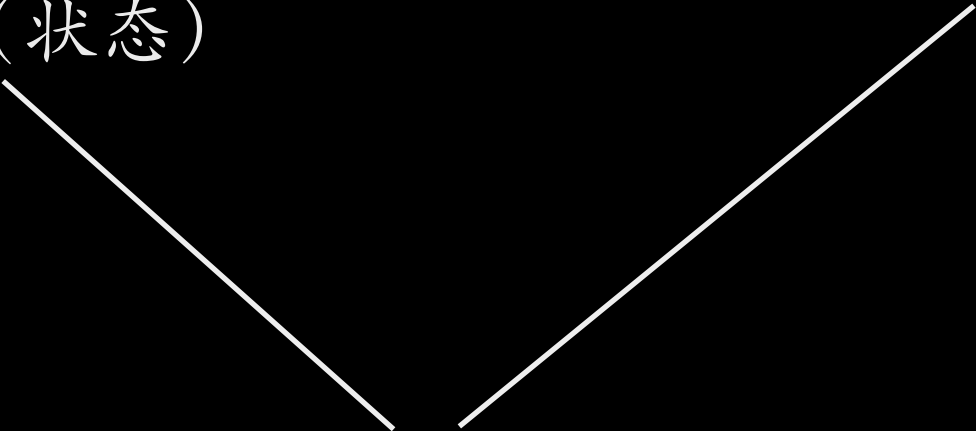
GUI の日常:

1. 点击按钮, 刷新文字 (状态)
2. 滑动屏幕, 刷新视图的 `offset` (状态)
3. 输入文字, 发送请求 (状态)
4. 如果请求成功, 刷新界面 (状态)

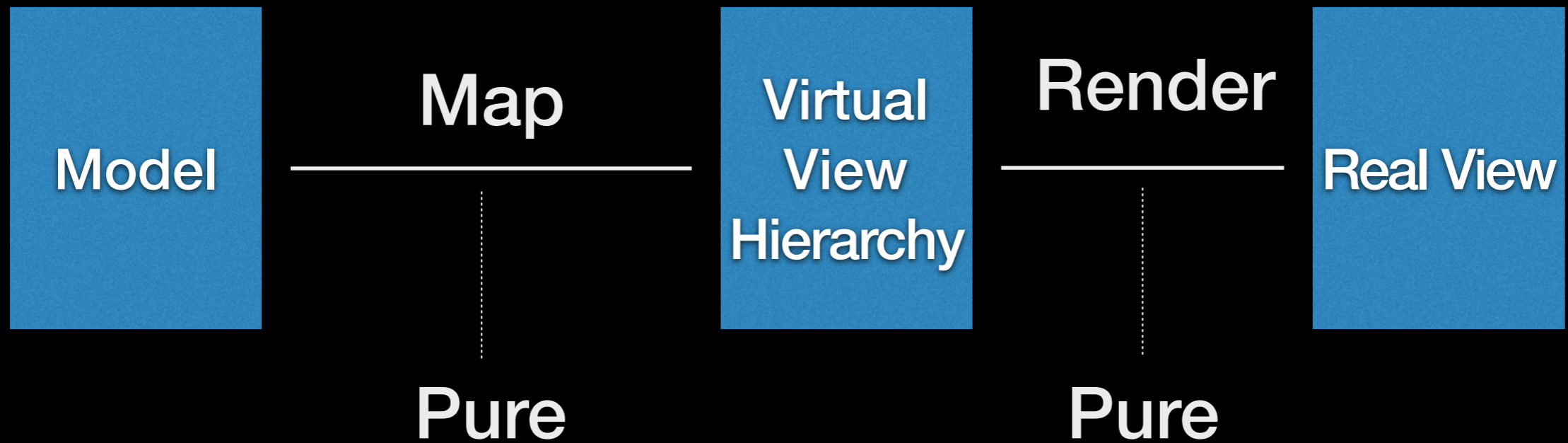
FP 的特征:

1. 不可变性
2. 无状态
3. 引用透明

怎么办?



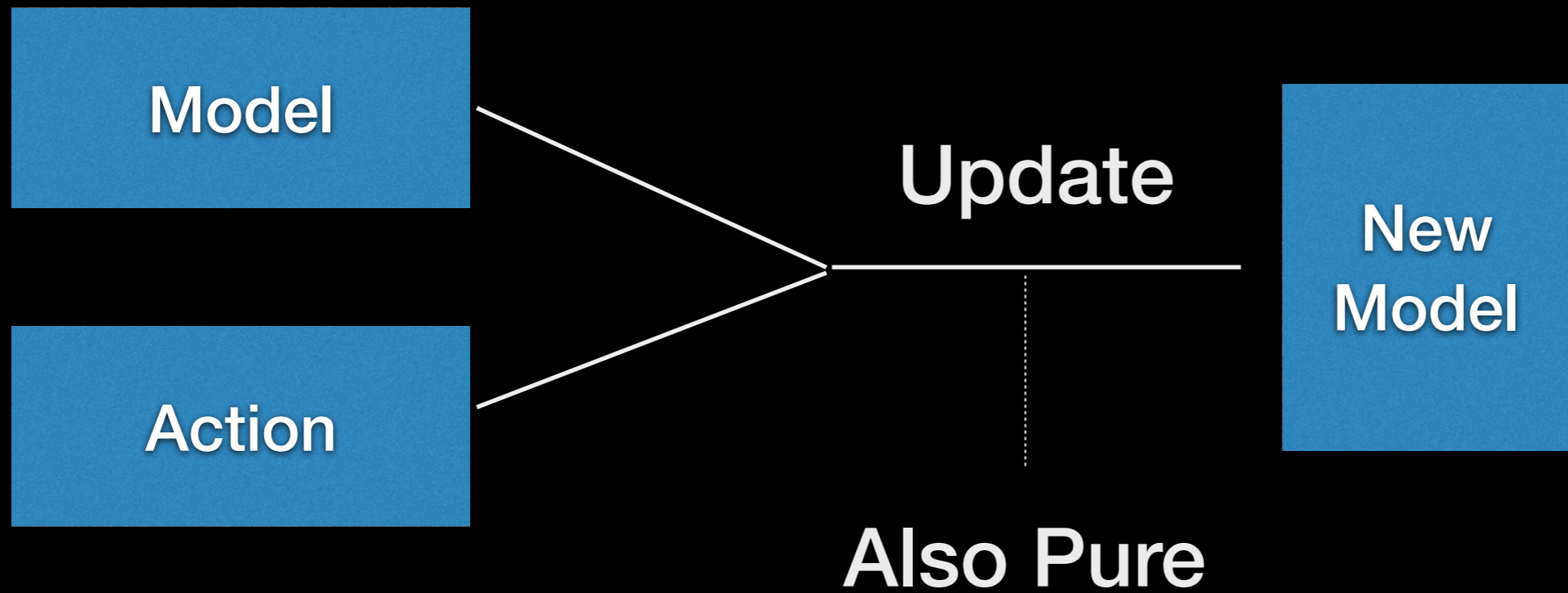
如果我们把状态都扔到一起



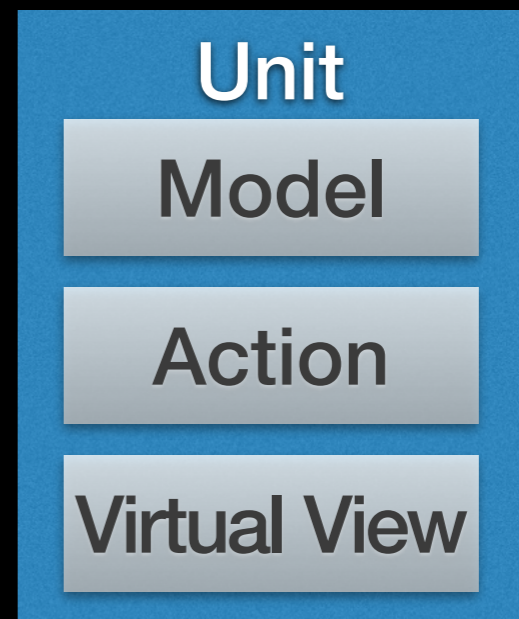
```
func Map(m : Model)->VirtualView  
{  
  
}
```

```
func Render(v : VirtualView) -> UIView  
{  
  
}
```

状态的更新



```
func Update(m : Model, a : Action) -> Model  
{  
  
}  
}
```



看着不错，但好使吗？

来了一个小需求：“用鼠标拖动 `ImageView`”

- 假设 `ImageView` 已经由我们之前的模型实现，作为一个 `Unit`
- 每次拖动，都需要：
- 用 `Mouse Move` 的 `Action`，`Update` 一次 `Model`（生成新的 `Model`）
- 用新的 `Model` 生成新的 `Virtual View Hierarchy`（不能刷新，因为 `VVH` 无状态）
- 用新的 `Virtual View Hierarchy`，生成新的 `UIView hierarchy`.
- 本质：暴力干掉 `state` 之后，没办法清晰的描述逻辑的依赖关系。（如：当鼠标选中时，`View` 的位置等于鼠标移动的位置）

想来想去，State 和 Immutability 都想要

怎么破？

Stream!



Stream: way to modelling changes

可变的 x :



```
x = 3;  
x = 4;  
x = 5;
```

x 不可变，但每次对其赋值，
`Stream<x>` 都会产生一个新的 x

`Stream<x>`

这货不是数组，可以用 Generator 来理解



用 Stream 来建模 GUI 中的 State

```
class Stream<a>{
    var closures : [(a)->Void] = []

    func subscribeNext(f : @escaping (_ value : a)->Void){
        closures.append(f)
    }

    func setVal(newValue : a)
    {
        for f in closures{
            f(newValue)
        }
    }
}
```

```
let mousePos : Stream<Float> = Stream()

mousePos.subscribeNext { (newPos) in
    print(newPos)
}
```

```
mousePos.setVal(newValue: 1.0)
mousePos.setVal(newValue: 2.0)
```

用一段最简单的 **Observer** 模式来实现 **Stream**

Stream 类中没有任何的状态，**Stream** 对象本身也是 **Immutable** 的。
但这并不妨碍我们拿到该对象不同时刻的值。

Stream = Lazy List != List

用 Stream 来解决鼠标移动问题

```
struct Model{
    var pos : Stream<Float>
}

struct VirtualView{
    var pos : Stream<Float>
}

enum Action{
    case MouseMove(Float)
}

func Map(m : Model)->VirtualView{
    return VirtualView(pos : m.pos)
}

func Update(m : Model, a : Action)
    -> Model{
    switch a {
    case .MouseMove(let newPos):
        m.pos.setVal(newValue: newPos)
        break
    }
    return m
}

func Render(v : VirtualView) -> UIView{
    let view = UIView()

    v.pos.subscribeNext { (pos) in
        view.frame = CGRect(x: 0, y: Int(pos), width:
100, height: 100)
        print ("move view to \(pos)")
    }

    return view;
}

let m = Model(pos: Stream())
let vv = Map(m: m)
let view = Render(v: vv)
Update(m: m, a: .MouseMove(130))
```

无论鼠标怎么移动，**Model**、**VirtualView** 和 **View** 都不用重新创建，**View** 位置的更新直接通过 **stream** 注入。
当然，整个过程依然是 **Pure** 的。

Stream 的大家族

Signal(Elm, ReactiveCocoa)

SequenceType(Native Swift)

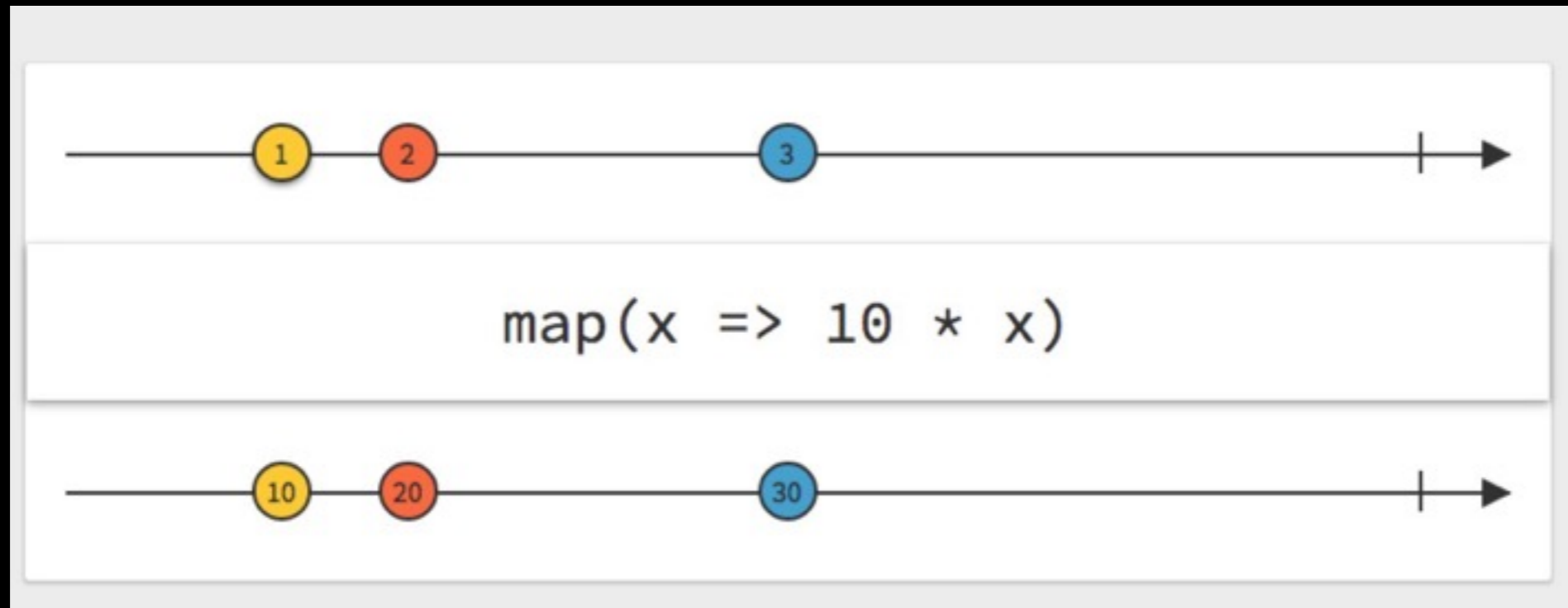
Observable(Rx*)



Promise(PromiseKit)

Stream(we defined before)

Stream 的哲学：Composable Transformation



- 使用 **Stream** 来建模 **State** 的另一大优势是，**Stream** 具备 **List** 一样的结构（逻辑上）。
- 这样我们就可以对其进行 **Map/Filter/Combine** 等操作。最终使得通过描述 **Stream** 的变换就能实现很多程序的逻辑。
- **Stream** 的这个特性是声明式编程的核心（**Declarative**），**Declarative** 使得我们可以通过“描述问题”来“解决问题”。代码紧凑，逻辑清晰。

Stream 变换的栗子

```
class Stream<a>{  
  ...  
  
  func map<b>(f : @escaping (a) -> b) -> Stream<b>  
  {  
    let newStream = Stream<b>()  
  
    self.subscribeNext { (x) in  
      newStream.setVal(newValue: f(x))  
    }  
  
    return newStream  
  }  
}
```

```
let mousePos : Stream<Float> = Stream()  
let mousePosReporter : Stream<String> = mousePos.map { (pos) -> String in  
  return "current pos is \ (pos)"  
}
```

```
mousePosReporter.subscribeNext { (str) in  
  print(str)  
}
```

通过 mousePos stream 生成一个新的 stream: mousePosReporter

```
mousePos.setVal(newValue: 1.0)  
mousePos.setVal(newValue: 2.0)
```


回到最初，什么是 FRP?

Functional Part:

High Order Function, Controlled Side Effects

+

React Part:

Composable Stream

=

Functional Reactive Programming

DSL/Interpreter

在 OO 世界里被无视得最彻底的模式



使用Parser-Combinator快速构建 DSL Parser

DSL, 领域特定语言, 专门为处理某个细分领域而生。比如为了加速我们写代码时创建 UI 的速度, 我们可以创造一门对应的 DSL, 如下:

```
let View = Gen("UIView:0,0,200,200:hidden=NO:color=red")
```

控件类名

初始 frame

属性初始化, 任意多个

建立抽象语法树

```
indirect enum ViewAst
{
    case ViewType(String)
    case Frame(Int,Int,Int,Int)
    case PropertyAssign(String,String)
    case Connector(ViewAst,ViewAst)
}
```

因为属性赋值部分可以有任意多个，所以我们定义了一个 **Connector** 类型来串起整棵树

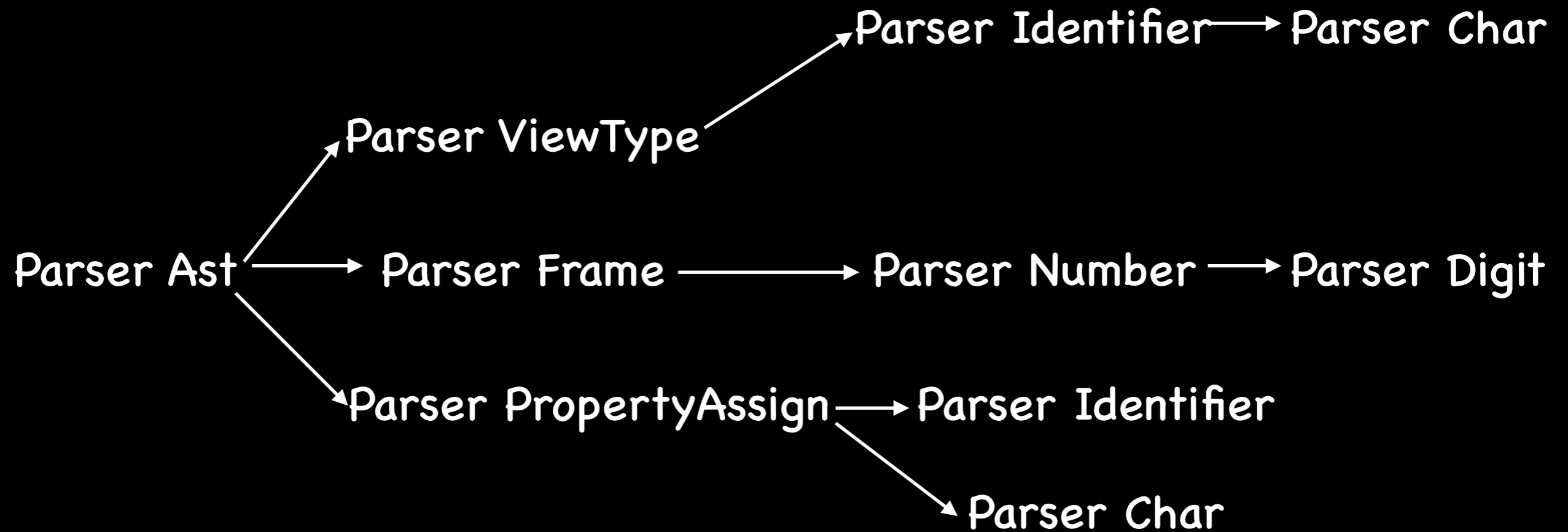
“UIView:0,0,200,200:hidden=NO:color=red”



```
ViewAst.Connector(ViewAst.ViewType("UIView"),
    ViewAst.Connector(ViewAst.Frame(0, 0, 200, 200),
        ViewAst.Connector(ViewAst.PropertyAssign("hidden", "NO"),
            ViewAst.PropertyAssign("color", "red"))))
```

Parser 的架构

假设 Parser 是可组合的.....我们就能够分而治之的解决这个问题



看上去是可组合的，如果是可组合的，那这些 Parser 应该都是同构的

定义 Parser 的架构

```
struct Parser<a>{  
    let p : (String) -> [(a,String)]  
}
```

- 不难看出，Parser 本质是一个 $\text{String} \rightarrow a$ 的函数；
- a 是泛型变量，代表解析的产物，可以是 `char,number,string` 或者是我们的 `AST`；
- Parser 是一个渐进的过程，所以更加准确的定义是 $\text{String} \rightarrow (a, \text{String})$ 后面的 `String` 代表还没有被处理的字符串；
- 因为 Parser 可能会失败，也可能会返回多种结果，所以我们将 `tuple` 用数组括起来；失败则返回 `[]`，成功则返回 `[(a, String)]`；
- 用结构体包一层仅仅是为了方便书写；

Parser 如何组合?

不论解析的结果是什么, 所有 Parser 都是同构的: `Parser<a>`

同构的结构组合?

Monad!



定义 Monad Laws 的两个方法

```
func pure<a>( _ item : a) -> Parser<a>{  
    return Parser { cs in [(item,cs)] }  
}
```

```
func >>= <a,b>(p : Parser<a>, f : @escaping (a)->Parser<b>) -> Parser<b>{  
    return Parser { cs in  
        let p1 = parse(p, input: cs)  
        guard p1.count>0 else{  
            return []  
        }  
        let p = p1[0]  
  
        let p2 = parse(f(p.0), input: p.1)  
        guard p2.count > 0 else{  
            return []  
        }  
  
        return p2  
    }  
}
```

- **bind** 在这里换成了中置运算符 `>>=`, 仅是为了方便书写
- **bind** 的本质: 利用现有的 `Parser(Parser<a>)`, 构造新的 `Parser(Parser)`, 并且后者可以利用前者的结果。
- 比如:

```
parserViewType = parserIdentifier >>= {x in pure(.ViewType(x))}
```

Building Block

```
func satisfy(_ condition : @escaping (Character) -> Bool) -> Parser<Character>
func digit() -> Parser<Int>
func number() -> Parser<Int>
func many<a>(_ p: Parser<a>) -> Parser<[a]>
func many1<a>(_ p : Parser<a>) -> Parser<[a]>
```

具体实现可以看我在 [Github](#) 上的 [Demo](#)

Composable parser

```
func identifier()->Parser<String>{
    return many1(satisfy(isNotSpecChar)) >>= { cs in
        pure(String(cs))
    }
}

func viewType()->Parser<ViewAst>{
    return identifier() >>= { str in
        pure(.ViewType(str))
    }
}

func numberItem()->Parser<Int>{
    return number() >>= { x in
        many(parserChar(",")) >>= { _ in
            pure(x)
        }
    }
}

}
```

核心：通过 `bind(>>=)` 基本款 `Parser` 和以
闭包形式提供的逻辑，生成新的 `Parser`

```
func frame()->Parser<ViewAst>{
    return colon() >>= { _ in
        many1(numberItem()) >>= { xs in
            pure(.Frame(xs[0],xs[1],xs[2],xs[3]))
        }
    }
}

func propertyAssign()->Parser<ViewAst>{
    return colon() >>= { _ in
        identifier() >>= { left in
            parserChar("=") >>= { _ in
                identifier() >>= { right in
                    pure(.PropertyAssign(left,right))
                }
            }
        }
    }
}

}
```

消除左递归

```
func viewDef()->Parser<ViewAst>{
    return viewType() +++ frame() +++ propertyAssign()
}

func viewDefFull()->Parser<ViewAst>{
    return connector() +++ viewDef()
}

func connector()->Parser<ViewAst>{
    return viewDef() >>= { left in
        viewDefFull() >>= { right in
            pure(.Connector(left, right))
        }
    }
}

let r = parse(viewDefFull(), input: "UIView:0,0,200,200:hidden=N0:color=red")

print (r)
```

最后输出:

```
[(ViewAst.Connector(ViewAst.ViewType("UIView"),
ViewAst.Connector(ViewAst.Frame(0, 0, 200, 200),
ViewAst.Connector(ViewAst.PropertyAssign("hidden", "N0"),
ViewAst.PropertyAssign("color", "red")))), "")]
```

总结

- 正确的使用高阶函数，来实现解耦；
- 通过 **Monad** 封装异常处理，实现清晰的顺序式代码结构；
- 通过 **FRP** 来实现 **GUI** 编程；
- 通过 **Parser Combinator** 快速开发 **DSL**；

“谢谢。”