QCon 全球软件开发大会
INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

BEIJING 2018

从 Observer 到 Observable

使用 Functional Swift 提升复杂 iOS 项目的可维护性

演讲者 / 王文槿

主办方 Geekbang. 极客邦科技 InfoQ

自我介绍

- 王文槿 / aaaron7 / 莲叔
- 2012 - 2016：创业，iOS + Py 后端
- 2016 - 至今： UC，iOS + Weex + 音视频
- 函数式编程爱好者

摘要

- 为什么需要 Functional thinking?
- Functional 经典应用：实现简单的 Observable 系统
- 解决一些实际问题
- 背后隐藏的设计模式
- Q & A

为什么需要 Functional thinking?

异常处理的完备性问题

OC时代的异常处理

```
- (void)statVideoPlay:(NSString *)videoUrl{
    VideoInfo *info = self.videos[videoUrl];
    [StatUtility uploadStatisInfo:@{@"video_title":info.videoTitle}];
}
```

OC时代的异常处理

```
- (void)statVideoPlay:(NSString *)videoUrl{
    assert(videoUrl != nil);
    VideoInfo *info = self.videos[videoUrl];
    [StatUtility uploadStatisInfo:@{@"video_title":info.videoTitle}];
}
```

OC时代的异常处理

```
- (void)statVideoPlay:(NSString *)videoUrl{
    assert(videoUrl != nil);
    VideoInfo *info = self.videos[videoUrl];
    if (info) {
        [StatUtility uploadStatisInfo:@{@"video_title":info.videoTitle}];
    }
}
```

OC时代的异常处理

```
- (void)statVideoPlay:(NSString *)videoUrl{
    assert(videoUrl != nil);
    VideoInfo *info = self.videos[videoUrl];
    if (info && info.videoTitle) {
        [StatUtility uploadStatisInfo:@{@"video_title":info.videoTitle}];
    }
}
```

OC时代的异常处理

```
- (void)statVideoPlay:(NSString *)videoUrl{
    assert(videoUrl != nil);
    VideoInfo *info = self.videos[videoUrl];
    if (info && info.video
        [StatUtility uploa                    _title":info.videoTitle}];
        //...
        //...
        //...
        [StatUtility
uploadStatisInfo:@{@"video                    Category}];
        //...
        //...
    }
}
```

心累，想哭

Swift 时代

```
    func statVideoPlay(videoUrl : String?){
        if let url = videoUrl{
            let videoInfo = self.videos[url]
            if let info = videoInfo{
                if let title = info.title{
                    StatUtility.uploadStatisInfo(statInfo:
["videoTitle":title])
                }
            }
        }
    }
```

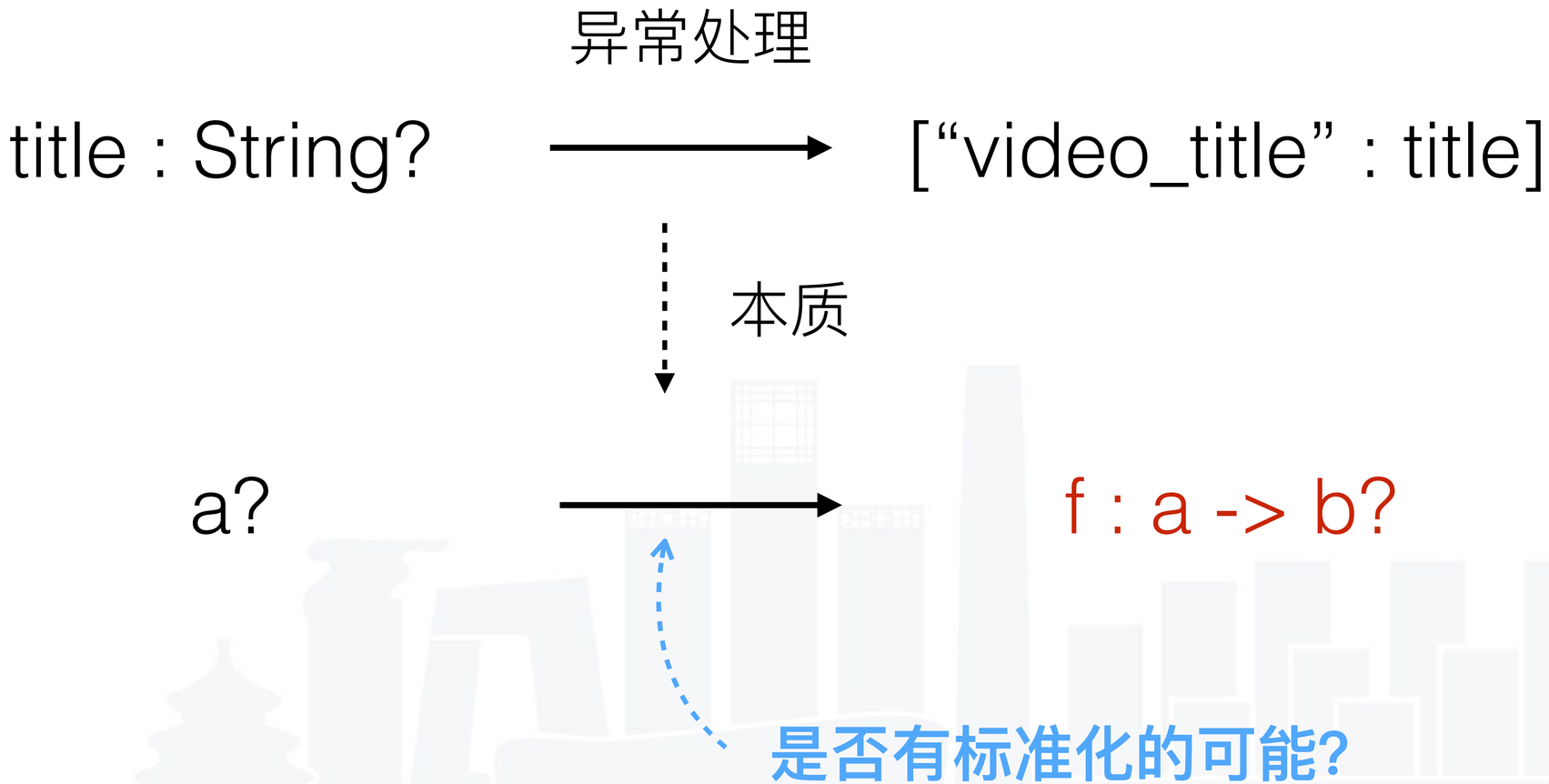
还是心累，还是想哭

异常处理的完备性问题

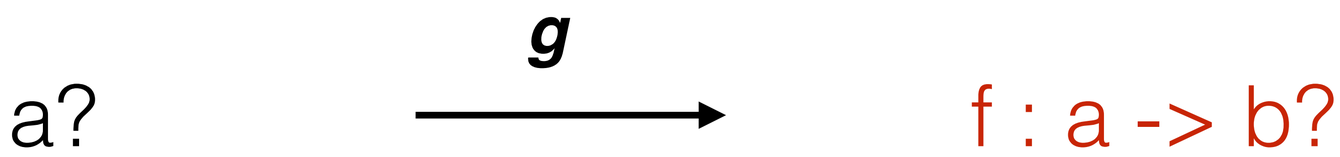
异常处理的完备性问题

f 也可能失败

异常处理的完备性问题



Functional Thinking



g: (input : a?, f : (a) -> b?) -> b?

if *input {return f(input)}* ***else*** *{return nil}*

如果有神奇的 g

```
func statVideoPlay(videoUrl : String?){
    let info:VideoInfo? = g(input: videoUrl) { self.videos[$0] }
    let title:String? = g(input: info) { $0.title }

    _ = g(input: title, f: { (x) -> String? in
        StatUtility.uploadStatisInfo(statInfo: ["videoTitle":x])
        return nil
    })
}
```

通过 g，自始至终没有任何判空，但代码是安全的。

hmmm，改进一下

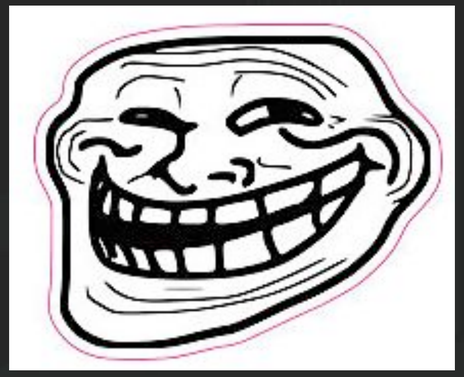
把 g 扔到 optional 的 extension 里

```
func statVideoPlay(videoUrl : String?){
    _ = videoUrl
        .g { self.videos[$0] }
        .g { $0.title }
        .g { (x) -> String? in
        StatUtility.uploadStatisInfo(statInfo: ["videoTitle"
: x])
        return nil
    }
}
```

进一步消除了中间变量

这个 g，有点眼熟?

```
extension Optional{
    func g<b>(f : (Wrapped) -> b?) -> b?{
        return self.flatMap {f($0)}
    }
}
```



重新认识 flatMap

Optional 类型的 flatMap， 本质就是提供了一个**标准化**的方式，来实现 **Optional Value** 到 只接收**非 Optional Value 逻辑**的绑定

Functional Thinking

- 通过抽象来更好的描述问题
  - **本质：非确定性 State -> 确定性 State的转换**
- 对于同构的场景提取模型
  - **普适的模型： (input : a?, f : (a) -> b?) -> b?**
- 最终通过符合直觉的方式解决问题
  - **链式调用，顺序执行，无需维护一堆中间变量和分支逻辑**

Functional 经典应用

一个简单的 Observable 系统

Observable

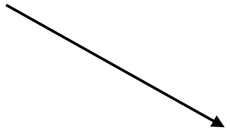
An Observable is an entity that wraps a value and allows to observe the value for changes.

PUSH-DRIVEN EVENT MODEL

Observable 的方案选择

复杂框架所带来的问题

陡峭的Learning Curve



无法低成本在团队推广

最终因编码风格的一致性等原则导致难产

Observable 的核心

- 一个具备 observe closure 管理功能的泛型容器
- 几个操作该容器的核心方法（subscribleNext/bind/map/filter）
- 一般用 Signal 表示，因为 value emit 的行为很像信号
- 具备自研条件，易于落地

动动手

```
public class Signal<a> : NSObject
{
    typealias SignalToken = Int
    typealias Subscriber = (a) -> Void
    var subscribers = [SignalToken:Subscriber]()
    public private(set) var value : a?

    let queue = DispatchQueue(label: "com.swift.let.token")

    init(value : a)
    {
        self.value = value
    }

    public func subscribeNext(hasInitialValue:Bool = false, subscriber :
@escaping (a) -> Void) -> SignalToken
    public func bind(signal : Signal<a>) -> SignalToken
    public func update(_ value : a)
}
```

closure 容器

订阅更新的接口

语法糖，直接绑定两个 Signal 的值（type一致）

更新 value，并逐个调用所有 closure

Playground

```
let xSignal = Signal(value: 0)
let ySignal = Signal(value: 1)
_ = xSignal.bind(signal: ySignal)

_ = xSignal.subscribeNext { (x) in
    print("got \(x) in xsignal")
}

_ = ySignal.subscribeNext(subscriber: { (x) in
    print("got \(x) in ysignal”)
})

xSignal.update(33)
```

```
got 33 in xsignal
got 33 in ysignal
```

来点儿好玩的

```
extension Signal{
    public func map<b>(f : @escaping (a) -> b) -> Signal<b>{
        let mappedValue = Signal<b>(value: nil)
        _ = self.subscribeNext { (x) in
            mappedValue.update(f(x))
        }
        return mappedValue
    }

    public func filter(f : @escaping (a) -> Bool) -> Signal<a>{
        let filterValue = Signal(value: self.value)
        _ = self.subscribeNext { (x) in
            if f(x){
                filterValue.update(x)
            }
        }
        return filterValue
    }
}
```

把一个 Signal map 成另一个 Signal

基于现有 Signal 生成一个新的 Signal
新的 Signal 只有在旧信号满足一定条件才触发

Playground

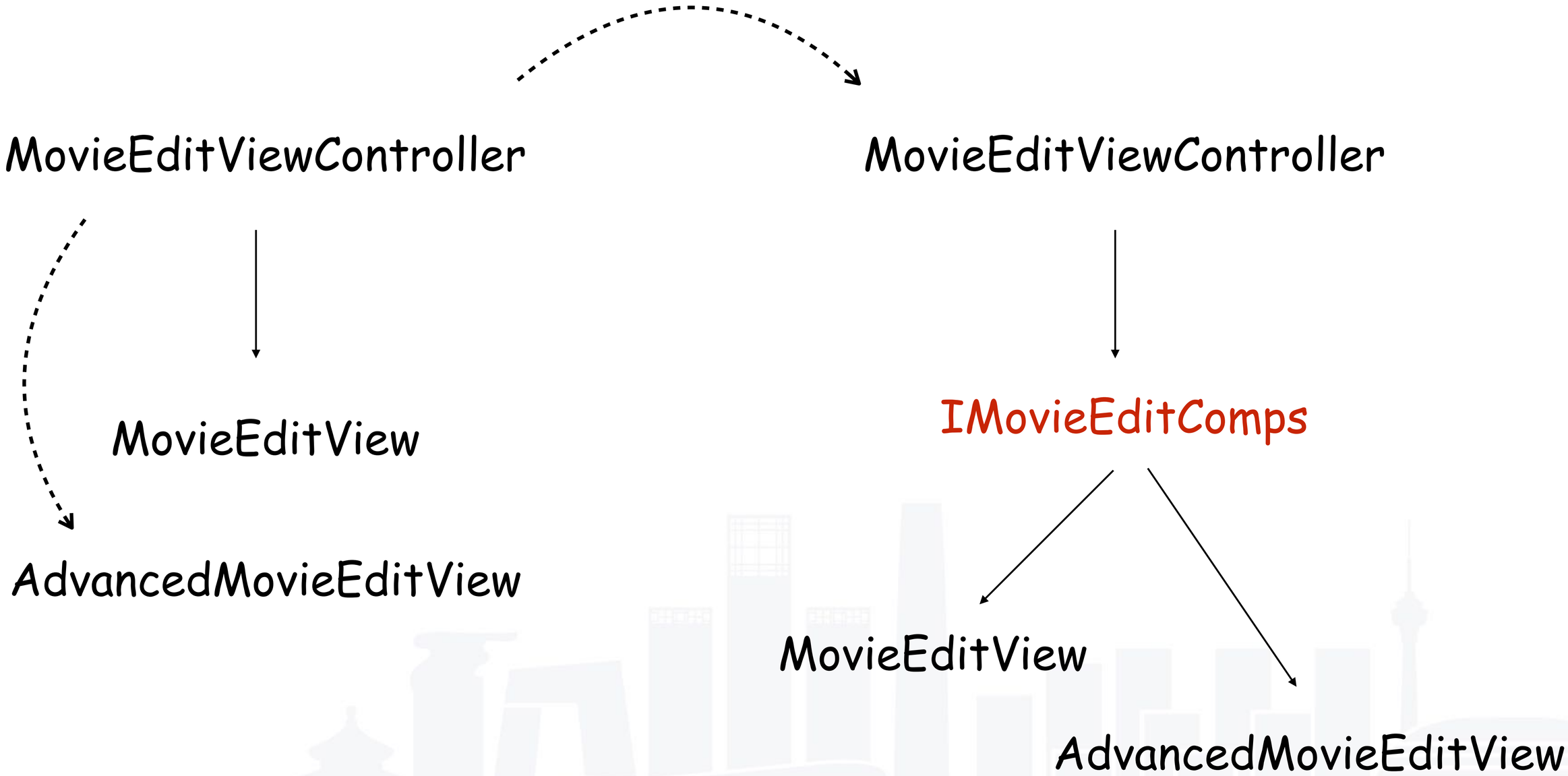
```
    let greaterThan33StringSignal = xSignal.filter(f: {$0 > 39}).map { (x)
-> String in
        return "hahah, im a string converted from \(x)"
    }
    _ = greaterThan33StringSignal.subscribeNext(subscriber: { (str) in
        print(str)
    })


    xSignal.update(33)
    xSignal.update(44)
```

```
hahah, im a string converted from 44
```

解决一些实际问题

中间层的消息传递问题

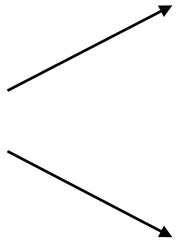
中间层是怎么来的


MovieEditViewController
MovieEditViewController
MovieEditView
IMovieEditComps
AdvancedMovieEditView
MovieEditView
AdvancedMovieEditView

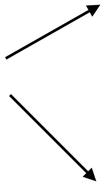
MovieEditViewController

IMovieEditComps

MovieEditView

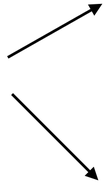


IMoviePlayer



IPlayerView

IPlayerControl



IPlayerBizInfoView

IPlayerProgressView

IOperationPanel

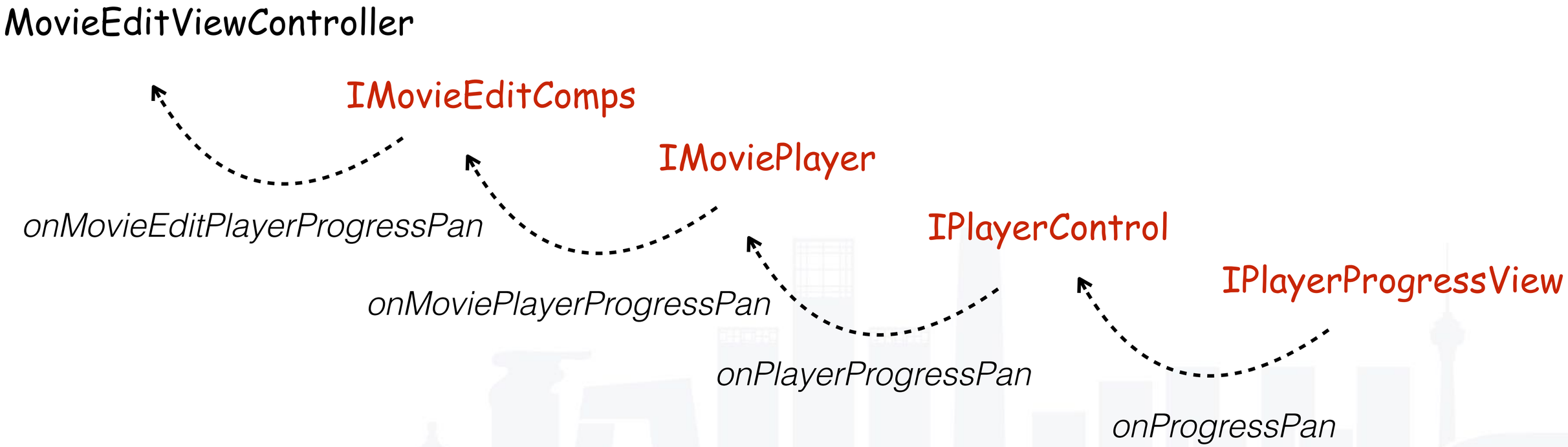
多变的业务需求往往都会催生出复杂的中间层

中间层过多带来的问题

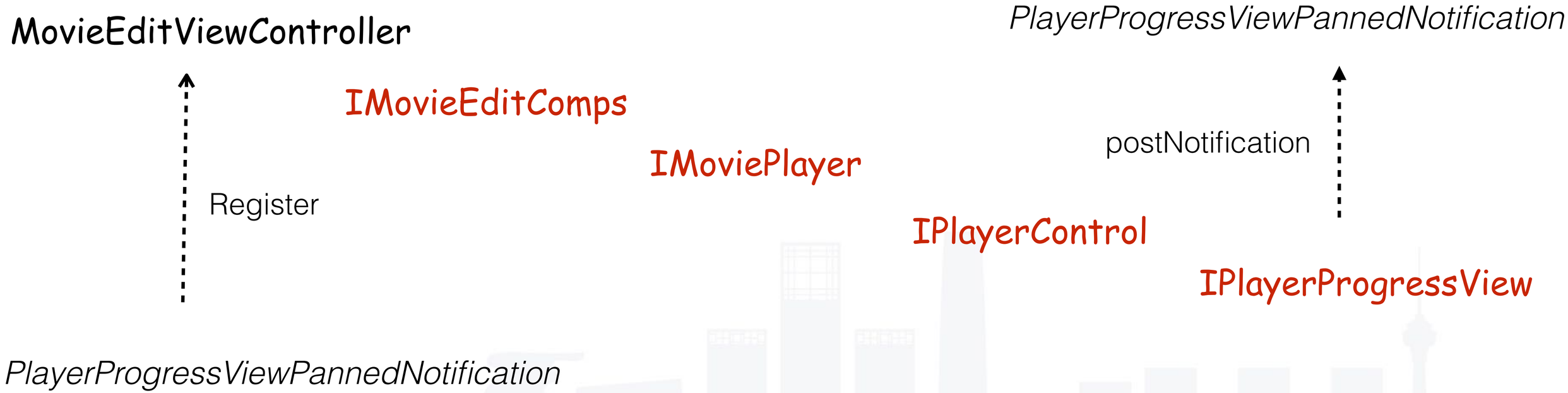
一个简单的需求：

新增一个 ProgressView 的 Pan 回调，由 Controller 处理

方式1 Delegate逐级上传



方式2 NotificationCenter



弊端

Delegate 方式

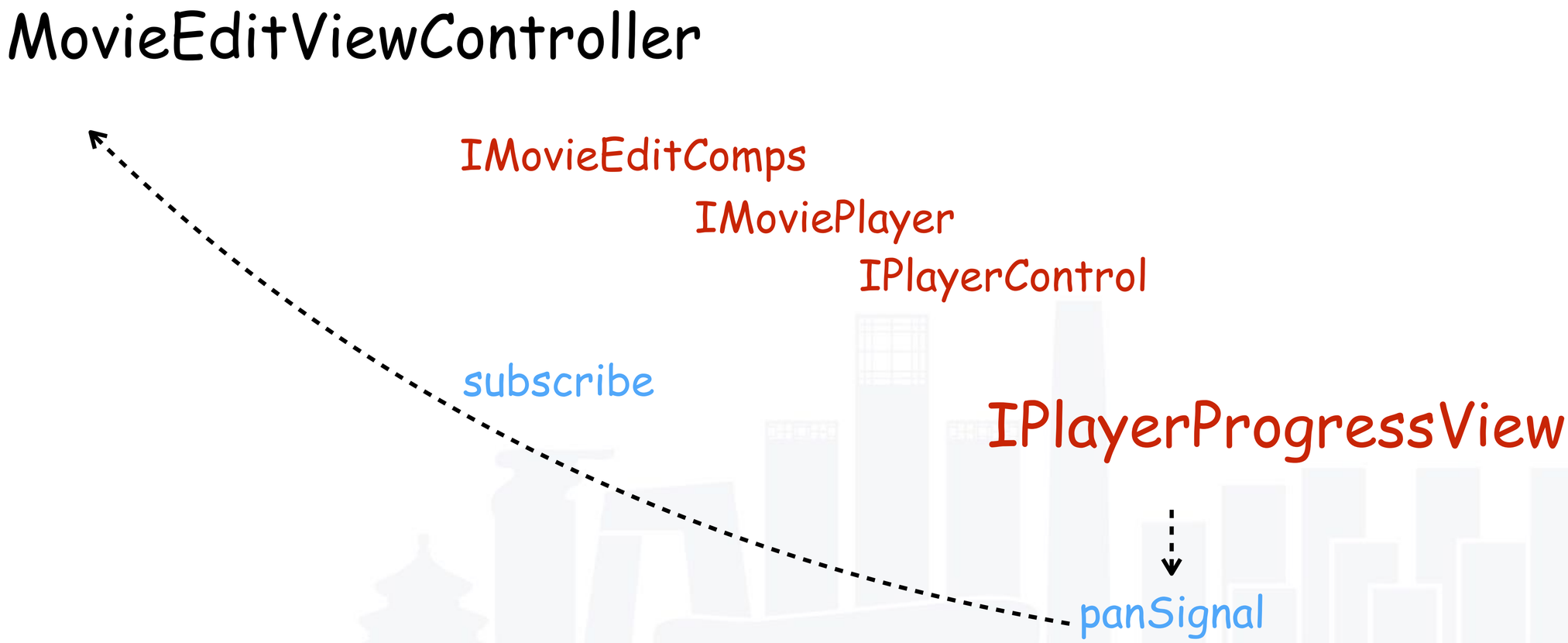
- boilerplate code
- 一个简单改动需改 N 个地方

Notification 方式

- 全局 NotificationName
- 无法反映出逻辑的依赖关系，不易维护

潜在的效率杀手

如果我们有 Signal



PlayerProgressView

```
lazy var panSignal : Signal<UIView> = {
    let signal = Signal(value: playerProgressView)
    return signal
}()

func onPan(sender : UIControl){
    panSignal.update(sender)
}
```

MovieEditViewController

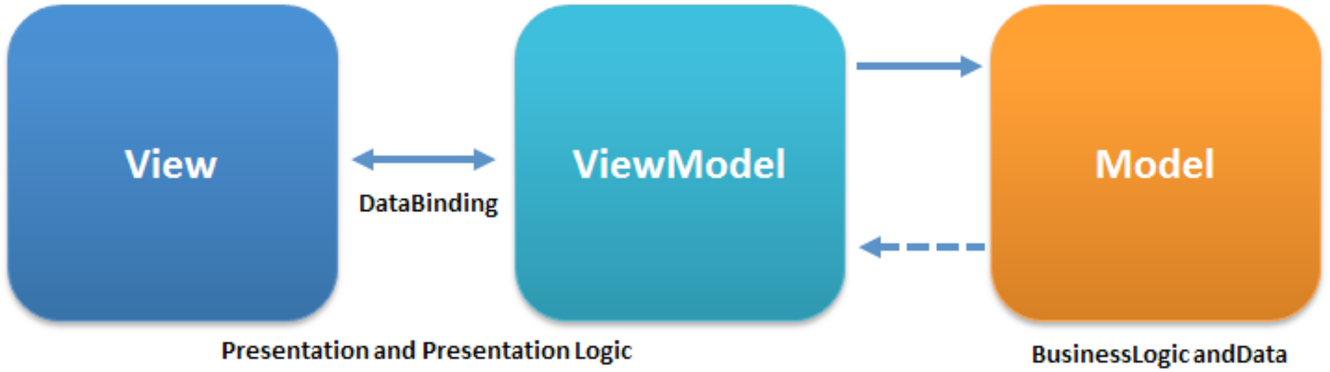
```
_ = editComps.moviePlayer
    .playerControl
    .playerProgressView
    .panSignal
    .subscribeNext(subscriber: { (sender) in
        //do something
})
```

优点

- 与 delegate 的一对一不同，panSignal 可以被任意多个对象订阅
- 新增事件，只需要新增一个 signal，中间层无需修改
- 子view内部的控件无需暴露，只需暴露 signal，保证了封闭性
- 依赖关系清晰，无需维护全局通知列表

MVVM 中的绑定问题

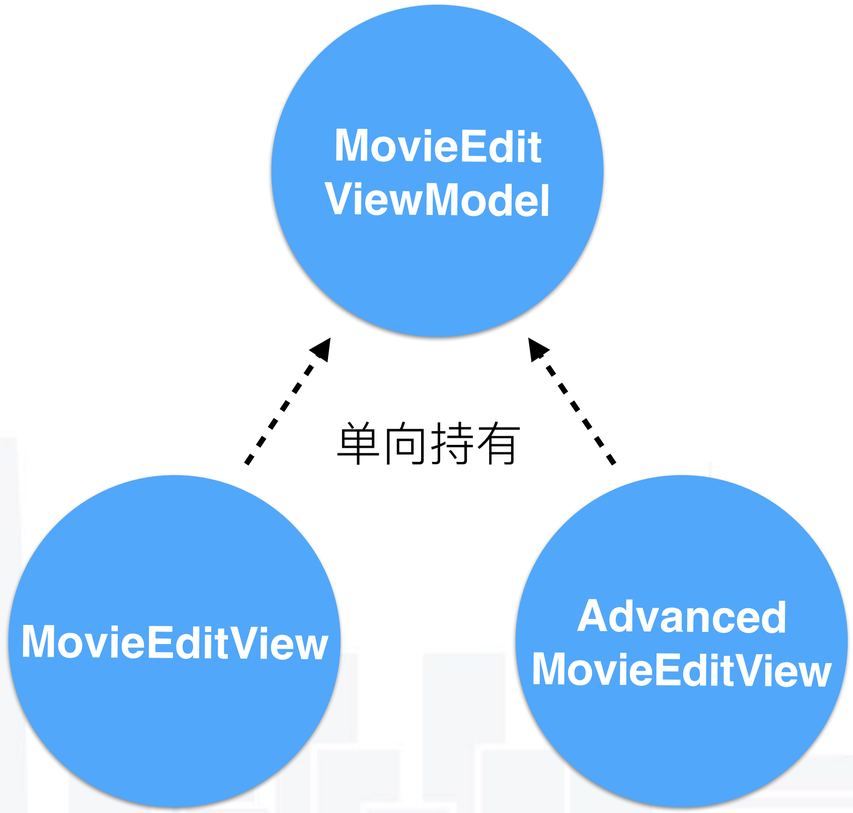
Massive VC 的救星：MVVM



- 比 Controller 更轻量级的逻辑处理单元 => ViewModel
- View 组件都有对应的 ViewModel 模块为其服务，不再强依赖 Controller
- 核心问题：ViewModel 和 View 如何交互?

ViewModel 和 View 交互

- View -> ViewModel, 比较直接
- 但是，View 如何嗅探到 ViewModel 中的更新
- 前提是，ViewModel 只应该关心内部如何实现逻辑，不应该知道 View 的存在



又到了 Signal 闪亮登场的时刻

```
class MovieListViewModel {
    var movies : Signal<[Movie]> = Signal(value: [])
    var selectedMovie : Signal<Movie?> = Signal(value: nil)

    func updateData(){
        getMovies { (results) in
            movies.update(results)
        }
    }
}
```

MovieListView

```
class MovieListView: UIView{

    var allMovies : [Movie] = []

    var viewModel : MovieListViewModel = MovieListViewModel()

    func setupViews() -> Void {
        _ = viewModel.movies.subscribeNext { (newValue) in
                self.allMovies = newValue
                self.contentView.reloadData()
        }

        viewModel.updateData()
    }
}
```

实现了什么?

- ViewModel 内的数据不管以何种方式更新，View 都能得到通知并刷新
- ViewModel “两耳不闻窗外事”，只关注自己内部的逻辑如何实现，无需关注使用自己的 View
- 纯逻辑的 ViewModel， 非常容易编写 unit test
- View 在初始化阶段就做完了生命周期的所有事情（声明式）

背后隐藏的设计模式

张总：我这需求就三句话，你按照写就完了，为啥要两天?

小明：话是这么说没错，但张总这块之前写的有点复杂，这个回调我不知道在哪个文件处理，得找找……

写代码有没有办法可以按照**符合直觉**的**逻辑顺序**来怼?

回顾一下flatMap

```
func statVideoPlay(videoUrl : String?){
    _ = videoUrl
        .g { self.videos[$0] }
        .g { $0.title }
        .g { (x) -> String? in
        StatUtility.uploadStatisInfo(statInfo: ["videoTitle"
: x])
        return nil
    }
}
```

异常处理本来有许多分支控制

但 flatMap 实现了逻辑 “**顺序化**”

Monad - 关于“顺序结构”的设计模式

- 名字来源于 Category theory（犯愁论）
- 面向对象的视角：当一个类以及其方法满足是一个 Monad 时，代表其可以被用来将一些 **“非顺序”** 结构的代码，改造为**“顺序”**结构
- 经典的非顺序结构代码：event handler 和 async operation
- 什么样的类才是 Monad ?

Monad Laws：

g: (input : a?, f : (a) -> b?) -> b?

pure<a> : (x : **a**) -> **M<a>**
bind<a,b> : (X : **M<a>**, f: **(a -> M<b>)**) -> **M<b>**

经典 Monad

Optional		**Promise**
实现异常处理顺序化		实现异步回调顺序化
let s:String? = “Hello"	pure	Promise { fetch... }
flatMap	bind	then

事件处理顺序化?

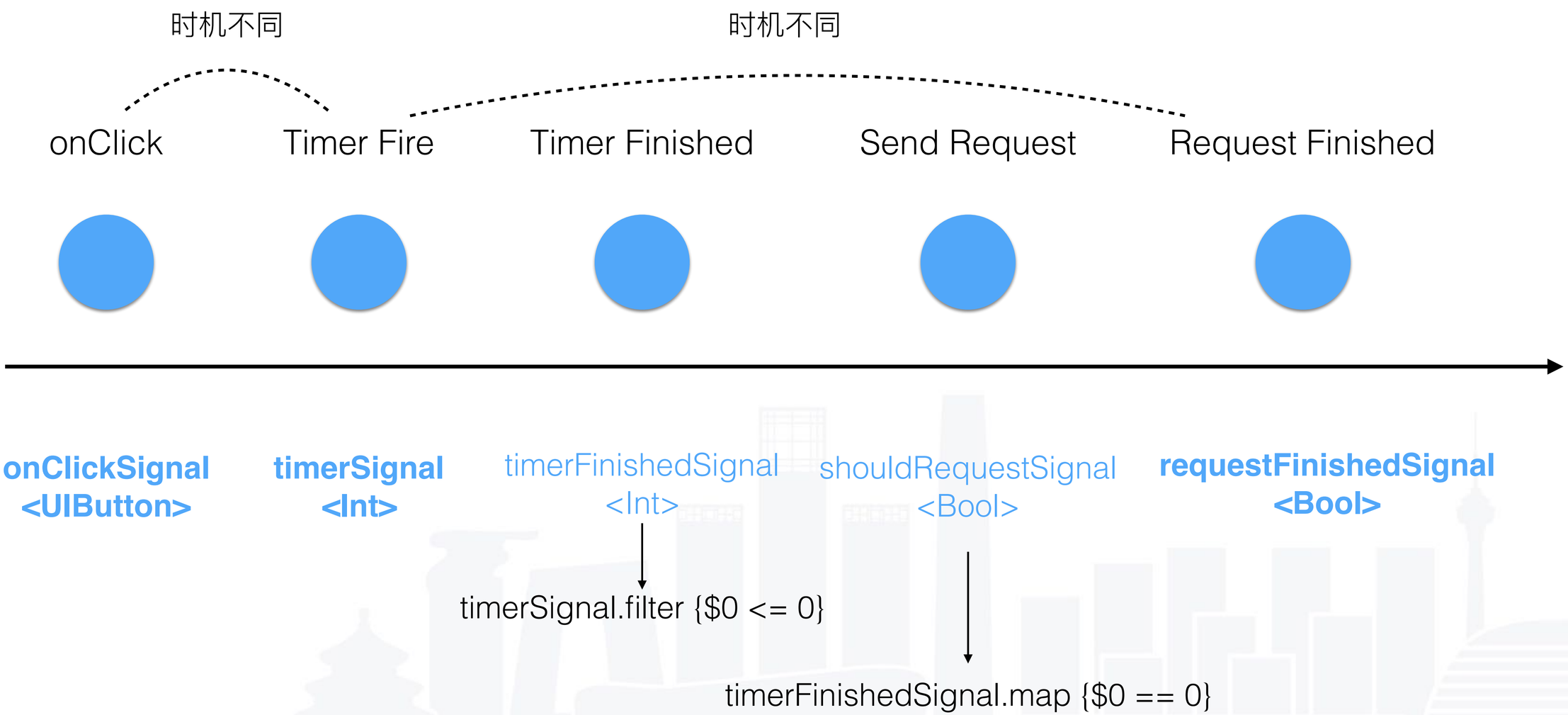
```
public func flatNext<b>(f : @escaping (a) -> Signal<b>) -> Signal<b>{
    let signal:Signal<b> = Signal<b>(value: nil)
    _ = self.subscribeNext(subscriber: { (x) in
        let newS = f(x)
        signal.update(newS.peek())
        _ = newS.bind(signal: signal)
    })
    return signal
}
```

- flatNext 就是 Signal 的 bind
- 思考：map 和 flatNext 都是通过 f 来生成新的 signal；核心区别是什么?

一个简单的需求

- 1. 点击按钮，开始计时, 倒数30秒
- 2. 倒计时完毕后提交记录到服务器
- 3. 请求成功返回后更新文本框提示

timeline 分析


时机不同
时机不同
onClick
Timer Fire
Timer Finished
Send Request
Request Finished
onClickSignal
<UIButton>
timerSignal
<Int>
timerFinishedSignal
<Int>
shouldRequestSignal
<Bool>
requestFinishedSignal
<Bool>
timerSignal.filter {$0 <= 0}
timerFinishedSignal.map {$0 == 0}

Playground

```
    onClickSignal.flatNext { (x) -> Signal<Int> in

        return timerSignal
    }.filter{$0 <= 0}.map{$0 == 0}
    .flatNext(f: { (shouldRequestNetwork) ->
Signal<Bool> in

        return finishedNetworkSignal
    }).flatNext(f: { (isFinished) -> Signal<String>
in

        return Signal(value: "")
    })
```

启动 timer，并返回 timerSignal

请求网络

网络请求成功

Playground

```
onClickSignal.flatNext { (x) -> Signal<Int> in
    let timerSignal = Signal(value: 30)
    print("button clicked")
    Timer.scheduledTimer(withTimeInterval: 1.0,
repeats: true, block: { (timer) in
        let value = timerSignal.peek()
        if value > 0{
            timerSignal.update(value - 1)
        }
    })
    return timerSignal
}.filter{$0 <= 0}.map{$0 == 0}
```

```
    .flatNext(f: { (shouldRequestNetwork) ->
Signal<Bool> in
        let finishedNetworkSignal = Signal(value: false)
        if shouldRequestNetwork{
            print("begin request network")
            uploadRecord(complete: { (x) in
                finishedNetworkSignal.update(true)
            })
        }
        return finishedNetworkSignal
    }).flatNext(f: { (isFinished) -> Signal<String> in
        if isFinished{
            print("upload successful")
            self.tipsLabel.text = "upload successful"
        }
        return Signal(value: "")
    })
```

Signal 是广义、泛化的 Promise，本身包含了 Promise 的语义

强势挤入前三

Optional	Promise	Signal
实现异常处理顺序化	实现异步回调顺序化	同时支持 Event+Async 顺序化
A 可直接强转为 a?	**pure** Promise构造函数	**pure** Signal 构造函数
flatMap	**bind** then	**bind** flatNext

Q & A

关注QCon微信公众号，
获得更多干货！

Thanks!

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

主办方 Geekbang. 极客邦科技 InfoQ