A BRIEF IMMERSION FROM

OBJECTIVE-C TO SWIFT

WHO IS THIS GUY?

- 傅若愚 (Ruoyu Fu)
- A Mobile Developer from ThoughtWorks
- Author of SwiftyJSON
- A Geek with Drinking Problems

WHEN SWIFT?

WHY SWIFT?

让我们从一段

TRICKY的CODE

开始今天的旅程

SOME TRICKY CODE

```
someArray = [1,2,3]
var result = someArray.map({"No. \($0)"})
/// 现在 `result` 这个变量的值应该是什么?
```

- 首先将一个数组`[1,2,3]`赋值给someArray
- 然后将someArray.map方法调用的结果赋值给result
- 问：现在result的值是什么?

SOME TRICKY CODE

```
someArray = [1,2,3]
var result = someArray.map({"No. \($0)"})
result
```

```
[0] "No. 1"
[1] "No. 2"
[2] "No. 3"
```

```
/// 当然，一个String的数组
```

IS THAT TRUE?

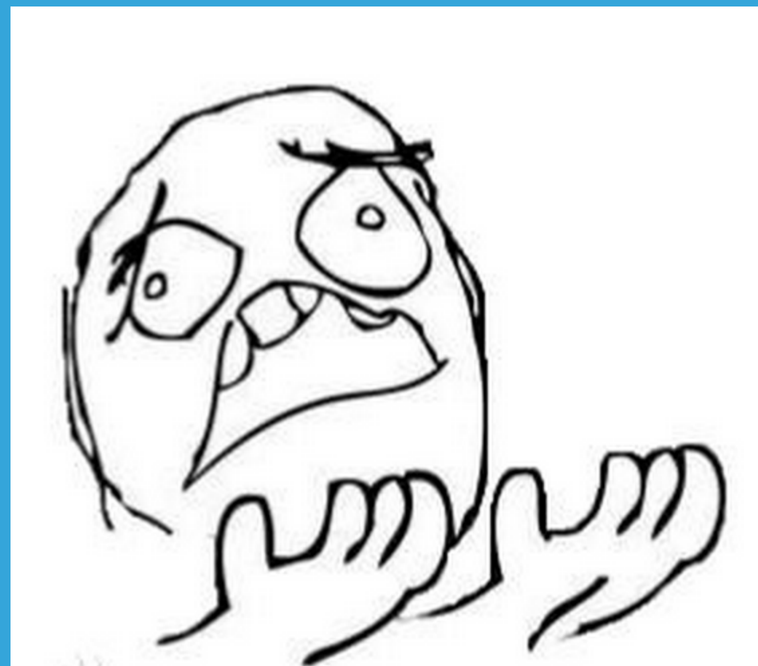
SOME TRICKY CODE

```
someArray = [1,2,3]
var result = someArray.map({"No. \($0)"})
result
```

No. [1, 2, 3]

```
/// 它也可能会是这样的!
```

- 同样一部分代码，可能产生完全不同的结果
- 即便是发生在同一个·对象·上!



WTF?

Really Serious Programmer

NOTE THE TYPES, NOT THE INSTANCES

```
var someArray:[Int]?

someArray = [1,2,3]
var result = someArray.map({"No. \($0)"})
result
```

```
No. [1, 2, 3]
```

ANOTHER EXAMPLE

```
2  var dictionary:[String:String?] = [:]        [:]
3  dictionary = ["key":"value"]                 ["key": {Some "value"}]
4  func justReturnNil()->String?{
5      return nil                               nil
6  }
7
8
9  dictionary["key"] = justReturnNil()          nil
10 dictionary                                   ["key": nil]
11
12 dictionary["key"] = nil                      nil
13 dictionary                                   [:]
14
15 Optional<String>.None                        nil
16 Optional<Optional<String>>.None              nil
```

TYPE SYSTEMS

- 弱类型 -> 强类型
- 欢迎来到类型安全的世界
- [⌘ + Click] -> [⌥ + Click]

```
var result = someArray
```

Declaration `var result: NSArray`

Declared In DemoPlayground.playground

TYPE SYSTEMS

- 动态多态 -> 静态多态
- 以前经常使用继承来实现的多态，现在不到万不得已，不会用到继承，而会优先考虑Protocol + Generics
- NSProxy is gone -_-!
- No more method swizzling
  (And other Runtime tricks)
- 让编译器做它该做的事！

TYPE SYSTEMS

- 强大的Runtime -> 弱小的Runtime
    - 一个悲伤的故事：

      曾经我有一个朋友想用Swift写一个ORM Mapper，后来我多了一个疯掉的朋友。

DESIGN PATTERNS

Snuck in a design
pattern on me, eh?

DESIGN PATTERNS

- 对设计模式进行任何具体的解析，从来都不是一件妥当的事情，他们应当自动地浮现，而不应当执着地实现！
- 但我们在从Objective-C转到Swift的过程中还是可以遵循一定的原则

DESIGN PATTERNS

- Rule No.1：除非不得已，不要使用继承
- Rule No.2：除非不得已，不要使用继承
- Why?

WHY NOT INHERITANCE ?

- 组合往往优于继承 (参见网上大家的讨论)
- Implicit sharing (参见WWDC)
- Mutable......
  Thread hells, locks etc
- “fatal error, should implemented by subclass”
- 子类与父类是两种不同的类型，强静态类型系统下会产生大量的Type Casting，它会导致：
  a，失去编译器对类型安全的保护。
  b，与OC不同Type Casting在Runtime消耗大量性能
  c，大量的冗余代码

当然——技术本身并不可耻

这只是一个适用范围的问题

DESIGN PATTERNS

- 建议1：需要使用继承的地方，优先选择Protocol
- 建议2：更多通过Swift提供的 enum, struct, extension等元素达成目的
- 建议3：与其他编程范式相结合

EXAMPLE

```
2
3  protocol RenderContext{
4
5      func renderText(texts:String...)
6      func renderImage(images:UIImage...)
7
8  }
9
10 protocol ViewModelType{
11
12     func renderInContext(context:RenderContext)
13
14 }
15
```

EXAMPLE

```
29 extension UITableViewCell:RenderContext{
30
31     func renderText(texts: String...) {
32         self.textLabel?.text = texts.first
33         self.detailTextLabel?.text = texts.last
34     }
35
36     func renderImage(images: UIImage...) {
37         self.imageView?.image = images.first
38     }
39 }
```

EXAMPLE

```
16 struct People:ViewModelType {
17
18
19     let name:String
20     let age:Int
21     let avatar:UIImage
22
23     func renderInContext(context: RenderContext) {
24         context.renderText(name, "\(age)")
25         context.renderImage(avatar)
26     }
27 }
```

EXAMPLE

在OC中通过弱类型+多态表示JSON非常容易：对于一个id来说，我们可以发任意方法的消息给它。（虽然会Crash）

而在Swift中呢?

```
enum JSON{
    case JArray([JSON])
    case JObject([String: JSON])
    case JString(String)
    case JNumber(Double)
    case JBool(Bool)
    case JNull
}
```

EXAMPLE

通过一个enum + 自定义的subscript方法我们可以同样简单，同时更加安全地表示JSON

```
subscript(index:Int) -> JSON? {
    guard case .JArray(let array) = self else{
        return nil
    }
    return array[index]
}

subscript(key:String) -> JSON? {
    guard case .JObject(let dic) = self else{
        return nil
    }
    return dic[key]
}
```

PARADIGMS

Swift中可供选择的编程范式非常的广泛，并且都可以有很好地实践，ReactiveCocoa，RxSwift都是经受过大量实际项目检验的不同范式的框架，我们应该将自己眼界放得更开。

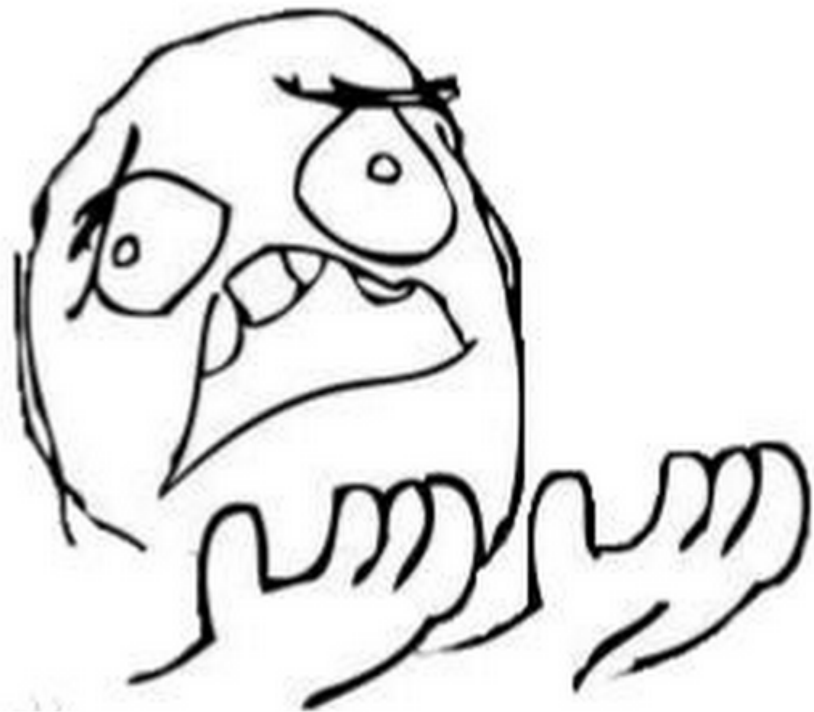
- Protocol Oriented Programming
- Object Oriented Programming
- Reactive Programming
- Functional Programming
- ……

THINK FUNCTIONALLY

A Sad Story About API disasters

```
func doSomeAsyncOperation(params: String, success: String -> Void, failure: NSError -> Void) {
    SomeAPI.fetchSomeAPI(params,
        success: { result in
            SomeOtherAPI.fetchSomeOtherAPI(result,
                success: { someOtherResult in
                    /*
                        Do Some Thing with someOtherResult
                    */
                    let finallyTheParams = self.transformResult(someOtherResult)
                    SomeThirdAPI.fetchSomeThirdAPI(finallyTheParams,
                        success: { finalResult in
                            let finishedResult = self.transformFinalResult(finalResult)
                            success(finishedResult)
                        },
                        failure: { someThirdError in
                            //handles error
                            failure(someThirdError)
                        }
                    )
                },
                failure: { someOtherError in
                    //handles error
                    failure(someOtherError)
                }
            )
        },
        failure: { error in
            //handles error
            failure(error)
        }
    )
}
```

THINK FUNCTIONALLY



是可忍孰不可忍?

THINK FUNCTIONALLY

自己写个Promise?

THINK FUNCTIONALLY

或许，还有更简单的办法！

THINK FUNCTIONALLY

完工之后的样子
看起来棒棒哒！

```
func someAsyncOperation(params: String) -> Async<String> {
    return fetchSomeAPI(params)
        .then(fetchSomeOtherAPI)
        .map(transformResult)
        .then(fetchSomeThirdAPI)
        .map(transformFinalResult)
}
```

THINK FUNCTIONALLY

第一步，定义一个叫做Async的东西，将异步过程封装起来

```
struct Async<T> {
    let function:(Result<T> -> Void) -> Void
}
```

其中用到的Result如下：

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

THINK FUNCTIONALLY

注意到它是一个Functor，所以
第二步，定义一个map函数

```
// Functor
extension Async {
    func map<U>(f: T throws -> U) -> Async<U>{
        return Async<U>{ cont in
            self.function{cont($0.map(f))}
        }
    }
}
```

THINK FUNCTIONALLY

注意它也是一个Monad，所以
第三步，定义一个flatMap函数

```
//Monad
extension Result {
    func flatMap<U>(@noescape f: T throws-> Result<U>) -> Result<U>{
        switch self{
        case .Success(let v):
            do{
                return try f(v)
            }catch let e{
                return .Failure(e)
            }
        case .Failure(let e):
            return .Failure(e)
        }
    }

}
```

THINK FUNCTIONALLY

第四步，给flatMap改个名字，叫做then

```
//Readable
extension Result {
    func then<U>(@noescape f: T throws-> Result<U>) -> Result<U>{
        switch self{
        case .Success(let v):
            do{
                return try f(v)
            }catch let e{
                return .Failure(e)
            }
        case .Failure(let e):
            return .Failure(e)
        }
    }

}
```

THINK FUNCTIONALLY

OK，Done！

并且

- 你还可以再加入apply让它成为一个Applicative
- 可以再加入foldl让它支持递归执行
- 甚至定义组合他们的运算符，以及更多……

THINK FUNCTIONALLY

可以看到，在Swift中，适当引入函数式编程的思想和方法，常常会有奇效

然而，凡事总有个然而

FUNCTIONAL TIPS

- 请根据自己项目的实际情况来考量是否引入FP，注意它的学习曲线并不平滑。
- 如果引入，尽可能与Reactive Programming结合着一起引入，很可能事半功倍。(RxSwift, Reactive Cocoa)

THANKS