Swift面向协议编程技术细节与工程演练

北京世纪好未来教育科技有限公司
研发工程师 陈刚

InfoQ ueue

促进软件开发领域知识与创新的传播

关注InfoQ官方信息

及时获取移动大会演讲视频信息

ArchSummit

全球架构师峰会2016

[深圳站] 2016年07月15-16日

咨询热线：010-89880682

QCon

全球软件开发大会

[上海站] 2016年10月20-22日

咨询热线：010-64738142

面向协议编程之前的协议

- OC&Swift2.0版本之前，协议是没有实现的，协议的用法大致就是：delegate、datasource。
- 比如最常用的“点赞”功能，在cell上放置按钮，覆盖cell的用户响应、保留按钮的用户响应。
- 用户点击按钮后需要修改数据源的状态。

- 按钮的定义在cell的子类中，IBAction也在定义在cell的子类中，而数据源却在Controller的子类中。
- 需要解决的问题是：如何把按钮的点击事件传递给Controller的子类。
- 写一个delegate协议，请看工程演示。

面向协议编程之前的协议

- 让我们来分解一下上面的操作：
- 1.声明一个Deleagte协议，这个协议可能只用到一次
- 2.在cell中定义可选型类型的属性，把1中定义的协议作为类型使用。
- 3.Controller子类遵循自定义的Delegate协议，并定义具体实现。
- 4.在获得每个Cell的实例的时候，指定实例的delegate属性为Controller，实现绑定。

- 更加Swift化的方案：使用闭包替代协议。
  - 请看工程演示。

GMTC 全球移动技术大会 GLOBAL MOBILE TECH CONFERENCE

- 使用闭包的步骤分解：
- 1.向定义其他类型的属性一样，定义一个闭包并定义初始值（一般都是空操作）。
- 2.在controller子类中为闭包属性重新赋值。

主办方 Geekbang. 极客邦科技 InfoQ

- 在Swift中闭包（当然还有函数和方法）是“一级公民”，补充一些闭包的知识。
- 使用闭包的好处：

1.步骤更简单，可读性强

2.代码耦合度更高，避免跨越代码，增加无意义的理解成本

- 使用闭包的注意事项：
- 闭包和类一样，也是引用类型的，会持有内部的对象，所以有“循环引用”的风险。在示例中使用了“捕获列表”来避免“循环引用”。不过不需要太过担心，苹果的官方文档中有说明，闭包只在特定情况下才有出现“循环引用”的风险。
- 方法中的闭包参数不会出现“循环引用”的风险。需要注意的是，如果方法体中有循环调用某个闭包参数的代码，经常在参数列表中为闭包参数加上@noescape关键字。@noescape的主要目的是避免在循环中不断引用相同的闭包，提升内存利用率。标准库中CollectionType协议中的map、filter等常用的方法都是基于@noescape的。

- 捕获列表是API的使用者在使用时添加的，使用者需要明确“循环引用”的触发条件，避免添加无意义的关键字。
- @noescape是API的开发者在定义时加入的，在使用闭包时{}中不会要求加入self关键字，API的使用者可以不用关心@noescape。

- 来明确一下“循环引用”风险的触发条件。
- 方法和函数都是以“名称”为第一标识符的，然后是参数类型和返回值类型，作为一门强类型语言，参数类型和返回值类型同时适用于方法重载。而闭包是匿名的，如示例中所示，闭包单独使用时，需要作为一个常量或者变量的类型使用，因此闭包自然而然会成为某个数据类型的成员。如果持有闭包的是类，那么会有“循环引用”的风险。这种情况需要特别注意。

- 闭包作为方法参数时不会出现循环引用。Swift中的方法的实现方式很有趣，无论是类型方法还是实例方法。闭包作为参数不会发生循环引用是因为方法本身并不属于某个数据结构，实例方法、构造器的实现机制是基于“柯里化”的。

面向协议编程之前的协议

- “柯里化”是什么？
- “柯里化”的方法都有多组参数列表，每传入一组参数，都会返回剩下的参数列表与返回值所组成的新函数。
- 请看playground演示。

- 结论：Swift的方法、构造器不是保存在数据结构中的，数据结构为定义在它内部的每个方法（无论是类型方法还是实例方法）定义了命名空间。调用方法的实例其实是这个命名空间中的方法的第一组参数，闭包是第二组参数列表中的参数，所以二者是平级的，不存在持有关系，因此即便在尾随闭包中使用了self中的内容，也不会发生“循环引用”。

面向协议编程之前的协议

- Swift中的闭包合理地接管了delegate的职责，此时协议的定义有点尴尬了。直到Swift2.0中协议扩展的引入。
- 首先来看个简单的例子，感受一下Swift2.0之后标准库中的协议扩展。
- 请看playground中的演示。

- Swift标准库中的Equatable协议要求用户重载==操作符（操作符是全局的函数，不用太在意函数和方法，二者本质是一样的），由于!=和==在逻辑上是互斥的，所以在Equatable的API设计中，一旦重载了==操作符，Equatable在扩展中定义的!=的默认实现就起作用了。

协议扩展初探

- 协议是什么？
- 一个人拥有名字、一辆车可以行使、一个数据集合可以搜索等等都可以抽象成协议。
- 协议是功能的最小化描述，粒度要比继承小得多。并且协议本身不能保存任何数据（协议扩展中不能定义存储属性），因此遵循一个协议不会带来额外的数据负担。
- 协议的核心思想：组合优于继承

协议扩展详解

- 协议扩展的使用步骤：

第一步 协议的声明，很像其他数据类型的声明，只不过没有实现而已。

第二步 协议的扩展，可以指定扩展的适用对象，在扩展中定义默认的实现。

第三步 有类、结构体或者枚举表示遵循这个协议。

第四步 遵循协议的数据类型来实现协议中声明的属性和方法，改写免费获得的默认实现

协议扩展详解

- 协议的声明：

```
protocol 协议:继承的协议1,继承的协议2 {
    var 某个属性:类型{set,get}
    func 某个方法(参数列表) -> 返回值类型
    init 构造器(参数列表)
}
```

属性需要明确读写的最低权限，协议遵守者需要符合属性的名称和最低权限，但是不限制属性是存储属性还是计算属性。

- 遵循协议

```
class 某个类:父类,协议1,协议2...{}
```

不同于OC，遵循协议格式上与继承是相同的。如果多个协议总是一起出现，可以使用typealias关键字给多个协议起一个别名，typealias并不会生成新的协议，现在上面的代码可以使用如下的格式来遵守协议：

```
typealias 协议组合别名 = protocol<协议1,协议2,...>
class 某个类:父类,协议组合别名{}
struct 某个结构体:协议组合别名{}
```

- 协议扩展

因为Swift是单类继承，而且结构体和枚举还不能被继承，这就为很多有用信息的传递造成了一定的麻烦。首先从数据冗余的角度来举例：

```
protocol Coder {
    var haveFun:Bool{ get set }
    var ownMoney:Bool{ get set }
}
protocol Swifter {
    var codingLevel:Int{ get set }
}
```

协议扩展详解

```
struct CoderFromA:Coder {
    var name:String
    var haveFun:Bool
    var ownMoney:Bool
}

struct CoderFromB:Coder,Swifter {
    var name:String
    var haveFun = true
    var ownMoney = true
    var codingLevel = 3
}

struct CoderFromC:Coder,Swifter {
    var name:String
    var haveFun = true
    var ownMoney = true
    var codingLevel = 5
}
```

所有的程序员都关心自己是否快乐、是否有钱，所以每个结构体都遵循协议Coder。A公司的程序员不是Swift程序员，而B公司和C公司的程序员都是Swift程序员，每个公司的Swift程序员的编程能力等级不同。观察上述代码可以发现，Swift程序员都是快乐且富有的，因此结构体CoderFromB和CoderFromC中会有冗余的部分，这是由不同协议Swifer与Coder间的因果关系所引起的。虽然我们知道这个事实，但是由于规则的关系我们不得不重复地去赋值haveFun和ownMoney属性。

协议扩展详解

- 使用协议扩展解决数据冗余：

```
extension Coder where Self:Swifter {
    var haveFun:Bool{ return true }
    var ownMoney:Bool{ return true }
}
```

老话说过了：协议扩展中的属性不能是存储属性。协议扩展的语法是非常“语言化”的，上面的代码很好理解：如果一个程序员是Swift程序员，那么他有钱又开心。

协议扩展影响的是协议的遵循。现在你可以删掉CoderFromB以及CoderFromC中的haveFun以及ownMoney的定义了。

很多时候计算属性和方法是很相似的，让我们来看看协议扩展中的方法，请看playground演示。

协议扩展详解

- 协议扩展的动态特性和静态特性：声明在协议声明列表中的方法会具有动态特性，会被完全重写。声明在协议扩展中的方法会具有静态特性，在切换上下文时可以获得协议的两个版本。在协议扩展中定义的方法如果调用了另一个在协议扩展中定义的方法，那么它得到的永远是最原始的版本。

- 简单来说：在设计一个协议时，把希望被重写的那些方法定义在协议的声明中，把希望保留原始逻辑的协议定义在协议的扩展中，以备其他基于该方法默认实现的方法使用该方法的原始版本，这样即便在该方法被遵守者重写的时候，其他方法的逻辑不会受到影响。

协议扩展中的泛型

- 面向协议编程中泛型是非常重要的。

首先依旧从用法来切入，Swift标准库中数组的判等定义如下：

```
public func ==<Element : Equatable>(lhs: [Element], rhs: [Element]) -> Bool
```

尖括号中是泛型定义，Element是一个占位符，代表了所有遵循Equatable协议的数据结构。

协议扩展中的泛型

- 声明一个泛型协议：

```
protocol SomeProtocol{
    associatedtype PrivateElement
    func elementMethod1(element:PrivateElement)
    func elementMethod2(element:PrivateElement)
}
```

这里虽然没有出现节点语法，但是上面的协议却是个不折不扣的泛型协议，PrivateElement起到了占位符的作用，指示了某种类型。根据协议的规则，协议SomeProtocol的遵守者必须实现上面两个方法，此时PrivateElement除了显式地体现了泛型的优势，还隐性地约束了两个方法的参数必须是相同类型的。你不用刻意去指定PrivateElement的具体类型，编译器会根据你实现方法的时候传入的参数类型确定PrivateElement的具体类型

GMTC 全球移动技术大会 GLOBAL MOBILE TECH CONFERENCE

- 由于关联对象本身也是一种泛型，所以可以使用协议对关联对象做限制：

```
protocol SomeProtocol{
    associatedtype PrivateElement:Comparable
    func elementMethod1(element:PrivateElement)
    func elementMethod2(element:PrivateElement)
}
```


主办方 Geekbang. 极客邦科技 InfoQ

协议扩展中的泛型

- 在协议的遵守者的声明中泛型转变成具体的类型：

```
struct TestStruct:SomeProtocol{
    func elementMethod1(element:String){
        print("elementFromMethod1:\(element)")

    }

    func elementMethod2(element:String){
        print("elementFromMethod2:\(element)")
    }
}
```

注意在实现的时候不能直接用PrivateElement了， PrivateElement只存在于具体实现之前，如果你尝试使用PrivateElement，编译器会报错。

协议扩展中的泛型

类似于的associatedtype的还有Self关键字，代表协议的遵守者本身的类型，适用于像比较这类的方法，你必须传入另一个相同类型的参数才有意义:

```
protocol CanCompare{
    func isBigger(other:Self) -> Bool
}
```

然后使用我们之前定义的盒子类型来试验一下:

```
struct BoxInt:CanCompare{
    var intValue:Int
    func isBigger(other: BoxInt) -> Bool {
        return self.intValue > other.intValue
    }
}
```

编译通过，现在新建两个实例测试一下：

```
BoxInt(intValue: 3).isBigger(BoxInt(intValue: 2))//结果为true
```

协议扩展中的泛型

关联对象是协议层面的泛型，类似于数据结构层面的类型，上面的方法如果使用数据结构层面的泛型的话，格式如下：

```
struct TestStruct< T:SignedNumberType >{
    func elementMethod1(element:T){
        print("elementFromMethod1:\(element)")

    }

    func elementMethod2(element:T){
        print("elementFromMethod2:\(element)")
    }
}

let test = TestStruct<Int>()
test.elementMethod1(1)
```

协议扩展中的泛型

- 泛型特化：无论是数据结构层面的泛型还是协议层面的泛型，泛型特化都发生在编译期，泛型特化之后需要是一个类型，不能再是泛型，否则在类型推断时会出现问题：

```
struct TestStruct< T:Comparable >{

    var array:[T] = [1,2,""]//报错

}
```

方法中的泛型

- 数据结构和协议层面的泛型在特化后都是某一个具体的类型，如果你想要定义泛型的方法，使用如下方式：

```
struct TestStruct{

    func elementMethod1< T:Comparable >(element:T){
        print("elementFromMethod1:\(element)")

    }

    func elementMethod2< T:Comparable >(element:T){
        print("elementFromMethod2:\(element)")
    }
}
let test = TestStruct()
test.elementMethod1(1)
test.elementMethod1("abc")
```

协议扩展中的Where关键字

我们可以在协议扩展中继续“开发”SomeProtocol，定义在协议扩展中的方法可以对某些遵守者“隐身”，而对另一些遵守者“可见”，依赖where关键字所进行的筛选。

继续深入上面的例子，需要注意的一点是，协议的遵守者如果遵守了多个协议，那么这些协议的关联对象名称不能重复，否则编译器会报错，所以关联对象的命名最好能有足够的描述力，并且避开标准库中的关联对象命名：比如Element、Generator等。

协议扩展中的Where关键字

where关键字可以与Self关键字配合（注意首字母大写），在协议扩展中新定义一个方法，指定当协议的遵守者必须是集合类型时，打印出遵守者的元素个数。

```
extension SomeProtocol where Self:CollectionType{
    func showCount(){
        print(self.count)
    }
}
```

我们所熟悉的count实际是定义在协议CollectionType中的，也就是说遵循CollectionType的对象必须要返回count属性的值，where的限制是强制的，它是一个编译器强制的开关。

协议扩展中的Where关键字

下面让我们来尝试一下，Array类型已经遵循了CollectionType协议，所以如果你让Array同时遵循SomeProtocol，那么它将免费获得shwoCount方法：

```
extension Array:SomeProtocol{

    func elementMethod1(element:String){
        print("elementFromMethod1:\(element)")

    }

    func elementMethod2(element:String){
        print("elementFromMethod2:\(element)")
    }
}

  [1,2,3].showCount()//打印3
```

协议扩展中的Where关键字

协议扩展中的where关键字为我们提供了一个安全的数据世界。除了使用Self关键字指定遵守者本身，where所限定的粒度还可以继续缩小。比如协议SomeProtocol，你可以使用where限定协议中的关联对象：

```
    extension SomeProtocol where
PrivateElement:SignedNumberType{
        func showElement(){
            print(OwnElement)
        }
    }
```

协议扩展中的Where关键字

- 如果关联对象中在定义时指定了必须遵循的协议，那么where可以对关联对象所遵循的协议中的成员做限制：

```
extension SequenceType where Self.Generator.Element :
Comparable {
    @warn_unused_result
     public func sort() -> [Self.Generator.Element]
}
```

协议的继承

协议是可以被继承的，不过因为协议的粒度足够小，秉承组合优于继承的理念，大部分情况下协议都不需要继承。不过在数据结构比较复杂的时候依然需要用到协议的继承。

依旧参考Swift标准库中用到的协议继承：

```
public protocol CollectionType : Indexable, SequenceType{…}
```

标准库中有非常多的SequenceType协议的遵守者是使用Int作为下标类型的，比如Array、Range，所以这些数据结构遵循了CollectionType，CollectionType对下标等操作的定义做了封装,以减少定义，所以协议定义中的继承其实也是一种“组合”。这种组合不会丧失底层协议的一般性，String.characters不是使用Int作为下标的，所以它遵循了SequenceType协议。

协议的继承

- 通常我们的数据结构不会像标准库那样复杂，所以你可以避免使用协议的继承，专注于协议的组合。

协议总结

- Swift协议是“绝对开关”，所有遵循协议的数据结构，必须满足协议参数列表中的所有方法，如果有不能满足的情况，那只能去单独定义该方法，用泛型去限制方法的参数。比如Array可以使用==，但是它没有遵循Equatable协议。

协议扩展的影响

- 协议扩展对iOS开发的影响：为什么WWDC2015中408号414号视频主讲人在互相推荐？
- 408_protocoloriented_programming_in_swift
- 414_building_better_apps_with_value_types_in_swift

协议扩展的影响

- 协议扩展解放了Swift中的值类型。
- 值类型是定长的，运行在栈上，速度更快，不会被共享，线程安全，不会产生循环引用，不会递归...
- 为什么一直以来值类型都得不到广泛的使用？
- 因为我们太过依赖类的继承特性。

协议扩展的影响

- 跨出面向协议编程的第一步，从使用值类型开始。
- 暂歇一下，稍后将带来：
- 1.面向协议编程“伪架构”，终结MVC与MVVM之争。
- 2.一些编程建议。
- 3.自由交流时间

- 从一个简单的Demo开始：

这个应用会使用一个TableView混合展示一个时间段的所有待办事宜和这个时间段的节日提醒，由于待办事件和节日的数据构成是不同的，所以需要创建两个模型，它们在TableView上展示的样式也应该有所不同，很常规的我们还需要一个TableViewCell的子类。

- 需要两个数据模型（结构体而不是类）

```
struct Event {
    var date = ""
    var eventTitle = ""
    init(date:String,title:String){
        self.date = date
        self.eventTitle = title
    }
}
struct Festival {
    var date = ""
    var festivalName = ""
    init(date:String,name:String){
        self.date = date
        self.festivalName = name
    }
}
```

- cell子类的定义：

```
class ShowedTableViewCell: UITableViewCell {
    //用来展示事件主题或节日名称的Label
    @IBOutlet weak var mixLabel: UILabel!
    //用来展示日期的Label
    @IBOutlet weak var dateLabel: UILabel!

}
```

- 由于我们的table需要混排，所以需要数据源以后，不要使用子类，在面向协议编程中使用协议的手段制造异构：

```
protocol HasDate{
    var date:String {get}
}
```

控制器中的异构数据源

- 异构数据源：

```
var dataList = [HasDate](){
    didSet{
        tableView.reloadData()
    }
}
var loadeddataList:[HasDate] = [Event(date: "2月14",
eventTitle: "送礼物"),Festival(date: "1月1日", festivalName:
"元旦"),Festival(date: "2月14", festivalName: "情人节")]
```

模拟数据源的加载

- 模拟的延迟加载：

```
override func viewDidLoad() {
    super.viewDidLoad()
    let delayInSeconds = 2.0
    let popTime = dispatch_time(DISPATCH_TIME_NOW,
                                Int64(delayInSeconds *
Double(NSEC_PER_SEC)))
    dispatch_after(popTime, dispatch_get_main_queue())
{ () -> Void in
        self.dataList =
self.loadeddataList.sort{$0.date < $1.date}
    }
}
```

传统的MVC

- 传统MVC中，数据与视图绑定的过程在控制器中：

```
    override func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell =
tableView.dequeueReusableCellWithIdentifier(cellReusedID, forIndexPath:
indexPath) as! ShowedTableViewCell
        //注意这里，通过可选绑定进行异构数据的类型控制
        if let event = dataList[indexPath.row] as? Event{
            cell.mixLabel.text = event.eventTitle
            cell.dateLabel.text = event.date
            cell.backgroundColor = UIColor.redColor()
            return cell
        } else if let festival = dataList[indexPath.row] as? Festival{
            cell.mixLabel.text = festival.festivalName
            cell.dateLabel.text = festival.date
            cell.backgroundColor = UIColor.whiteColor()
            return cell
        } else {
            return cell
        }
    }
```

运行效果

- 运行的效果如下：

分析MVC

- 我不喜欢谈架构，架构有种莫名的压力。
- 我喜欢MVC，因为一切都看起来那么自然，创建了一个Cell的实例，从数据源中取出一个数据的实例，将二者结合。
- 可惜控制器里面的代码太多了，而数据和视图中除了定义什么都没有，这不公平！

GMTC 全球移动技术大会 GLOBAL MOBILE TECH CONFERENCE

- 来看看MVVM吧！
- 不修改数据源和视图中的代码，定义一个协议，这里的属性并不是真实的数据源中的属性，只能通过名字来关联记忆。

```
protocol CellPresentable{
    var mixLabelData:String {get set}
    var dateLabelData:String {get set}
    var color: UIColor {get set}
    func updateCell(cell:ShowedTableViewCell)
}
```


主办方 Geekbang. 极客邦科技 InfoQ

中间层的协议扩展

- 声明一个cell样式所需要的全部数据，然后对号入座。

```
extension CellPresentable{
    func updateCell(cell:ShowedTableViewCell){
        cell.mixLabel.text = mixLabelData
        cell.dateLabel.text = dateLabelData
        cell.backgroundColor = color
    }
}
```

视图中新增方法

- 你需要在视图中新增一个方法：

```
class ShowedTableViewCell: UITableViewCell {
    //用来展示事件主题或节日名称的Label
    @IBOutlet weak var MixLabel: UILabel!
    //用来展示日期的Label
    @IBOutlet weak var dateLabel: UILabel!

    func updateWithPresenter(presenter:
CellPresentable) {
        presenter.updateCell(self)
    }
}
```

定义一个ViewModel

- 定义ViewModel，吸纳异构数据源的每一个类型：

```
struct ViewModel:CellPresentable{
    var dateLabelData = ""
    var mixLabelData = ""
    var color = UIColor.whiteColor()
    init(modal:Event){
        self.dateLabelData = modal.date
        self.mixLabelData = modal.eventTitle
        self.color = UIColor.redColor()
    }
    init(modal:Festival){
        self.dateLabelData = modal.date
        self.mixLabelData = modal.festivalName
        self.color = UIColor.whiteColor()
    }
}
```

修改控制器

- 终于，我们可以回到控制器中来了：

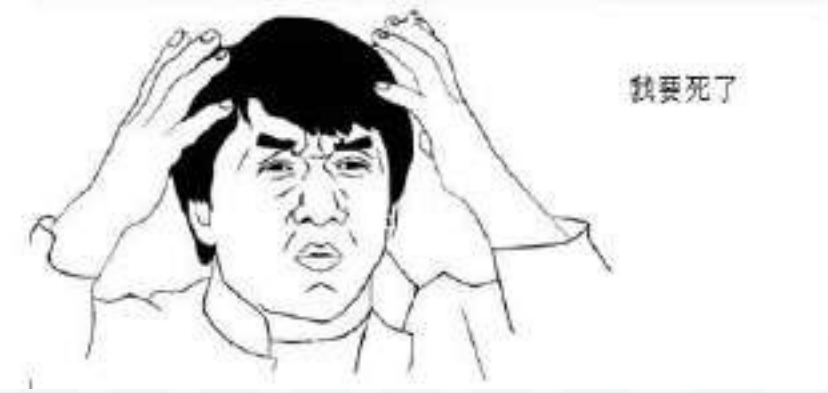
```
    override func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell =
tableView.dequeueReusableCellWithIdentifier(cellReusedID,
forIndexPath: indexPath) as! ShowedTableViewCell
        if let event = dataList[indexPath.row] as? Event{
            let viewModel = ViewModel(modal: event)
            cell.updateWithPresenter(viewModel)
            return cell
        } else if let festival = dataList[indexPath.row] as? Festival{
            let viewModel = ViewModel(modal: festival)
            cell.updateWithPresenter(viewModel)
            return cell
        } else {
            return cell

        }
    }
```

MVVM运行结果

- 运行结果是正确的：

MVVM总结

- 就我们的刚才的例子来说，收益甚低，因为控制器中的代码没有省掉几行，却创建了一堆中间代码。或许MVVM更适合复杂的视图
- 什么时候用MVC，什么时候用MVVM？



终结MVC与MVVM之争

- 终结MVC与MVVM之争！

伪架构

- 四月份的时候我研究出了“新套路”，在博文中起了个炫酷的名字叫“幽灵架构”。因为相比于MVVM，效果真的有点鬼魅。从编程舒适度的角度来看，你不需要把它当做架构，因为真的很简单，暂且称之为“伪架构”

伪架构核心协议


```
//视图使用的协议
protocol ViewType{
    func getData<M:ModelType>(model:M)
}
extension ViewType{
    //定义默认实现，因为可能出现方法重载
    func getData<M:ModelType>(model:M){

    }
}
//数据使用的协议
protocol ModelType{
}
//定义默认方法giveData
extension ModelType{
    func giveData<V:ViewType>(view:V){
        view.getData(self)
    }
}
```

方法参数说明

- 请注意参数类型是泛型的：
```
func getData<M:ModelType>(model:M)
```
- 在方法中不要把协议当做类型：
```
func getData(model:ModelType)
```
- 2015WWDC408号视频中有提到，但是没具体讲，因为协议类型的参数会产生Box类型，浪费性能。
- 请看playground演示

伪架构的模型

- 需要展示的数据模型都要遵循ModelType
- 别忘了组合一下协议：
- `typealias DateViewModel = protocol<hasDate,ModelType>`
- 现在的模型：

```
struct Event:DateViewModel{…}
struct Festival:DateViewModel{…}
```

- 除此之外没有别的工作了

伪架构的视图


```
extension ShowedTableViewCell:ViewType{
    func getData<M : ModelType>(model: M) {
        //xcode7.3之前这里不能写成guard let dateModel = model as?
DateViewModel else{}
        guard let dateModel = model as? DateViewModel else{
            return
        }
        //处理相同属性
        dateLabel.text = dateModel.date
        //处理数据源异构
        if let event = dateModel as? Event{
            mixLabel.text = event.eventTitle
            backgroundColor = UIColor.redColor()
        } else if let festival = dateModel as? Festival{
            mixLabel.text = festival.festivalName
            backgroundColor = UIColor.whiteColor()
        }
    }
}
```

伪架构的控制器

```
        override func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
            let cell =
tableView.dequeueReusableCellWithIdentifier(cellReusedID,
forIndexPath: indexPath) as! ShowedTableViewCell

            dataList[indexPath.row].giveData(cell)

            return cell
        }
```

伪架构工程演示

- 运行效果请直接看工程演示。
- 如果数据源不是异构的，比如[Event]或者[Festival]，直接调用getData方法即可，giveData方法是专门拆封异构数据源的,比如工程中的CollectionView展示的是Festival，直接用了getData。

伪架构工程演示

- 如果数据和视图的绑定逻辑需要发生变化，那么重载giveData和getData，同理同构数据源的时候只重载getData即可：

```
//针对数据源的扩展
extension ModelType where Self:HasDate{
    func giveData<V:ViewType>(view:V,tag:Bool){
        if let tableCell = view as? ShowedTableViewCell{
            tableCell.getData(self,tag:tag)
        }
    }
}
//针对视图的扩展
extension ViewType where Self:ShowedTableViewCell {
    func getData<M:ModelType>(model:M,tag:Bool){
        //留空，依旧在视图代码中重新实现
    }
}
```

伪架构工程演示

- 视图中只重写一个getData方法（默认的或定制的）：

```
extension ShowedTableViewCell:ViewType{
    func getData<M : ModelType>(model: M,tag:Bool) {
        guard let dateModel = model as? HasDate else{
            return
        }
        //处理相同属性
        dateLabel.text = dateModel.date
        //处理数据源异构
        if let event = dateModel as? Event{
            mixLabel.text = event.eventTitle
            backgroundColor = tag ? UIColor.redColor() :
UIColor.whiteColor()//加入tag的逻辑
        } else if let festival = dateModel as? Festival{
            mixLabel.text = festival.festivalName
            backgroundColor = tag ? UIColor.whiteColor(): UIColor.redColor()
        }
    }
}
```

伪架构工程演示

- 在如上的工程中，存在两种视图：UICollectionViewCell、UITableViewCell。同时也存在两种数据模型：Event和Festival。数据源和视图自由绑定的话有非常多种情况，这是一种多对多的关系，所以非常适合使用408号视频中的异构处理，也就是getData方法中的：

```
let 别名 = model as? 类型{…}
```

伪架构总结

- 是不是很清爽？没有引入中间层，一切都依靠协议完成，转移了数据和视图的绑定。

伪架构总结

- 想想在下次开发的时候你能做点什么？
- 在后台人员返回可用的数据接口的之前，在你思考控制器中复杂的数据处理之前，你的数据模型和视图的绑定已经提早完成了。
- getData中的代码是不依赖控制器的，即便你删掉整个控制器代码也不会有影响。

- 上面展示了处理数据和视图的绑定的伪架构，接着展示如何使用面向协议编程思想处理纯数据。
- 为上面的Demo增加搜索功能，由于数据保存在tableView和CollectionView，所以我们需要一个通用的数据处理模型

通用搜索模型

- 日期的模糊匹配搜索协议：

```
protocol SearchDate{
}

extension SearchDate{
    func search<D:HasDate>(source:[D],key:String) ->
[D]{
        return source.filter{
            $0.date.containsString(key)
        }
    }
}
```

同构数据

- 上面的通用搜索协议只能搜同构的数据源，并且返回的也是相同类型的数据源，异构的数据不能定义泛型的方法，返回的类型也是异构的，但是处理异构也是非常简单的。

异构数据

- 定义一个针对异构数据源的方法：

```
extension SearchDate{
    func search<D:HasDate>(source:[D],key:String) -> [D]{
        return source.filter{
            $0.date.containsString(key)
        }
    }
    func search(source:[DateViewModel],key:String) ->
[DateViewModel]{
        return source.filter{
//            if let festival = $0 as? Festival{
//                return festival.date.containsString(key)
//            } else if let event = $0 as? Event{
//                return event.date.containsString(key)
//            }
//            return false
            //实际上不需要对异构数据源特化，只需要写下面这行代码
            return $0.date.containsString(key)
        }
    }
}
```

运行结果

- 请看示例工程中的控制台

总结

- 面向协议编程有无限的可能性等待发掘，下面是我个人的一些学习建议，仅做参考：
- 1.使用值类型，注意定义Copy-on-Write
- 2.不要用Any、AnyObject
- 3.使用泛型提升性能
- 4.合理使用可选型，需要过程信息的API使用错误处理。

GMTC 全球移动技术大会
GLOBAL MOBILE TECH CONFERENCE

THANKS

聚焦前沿技术　传递实践经验

主办方 Geekbang. 极客邦科技　InfoQ