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

当泛型遇上协议

Generic Programming with Protocol in Swift

蓝晨钰 猿题库 iOS 团队负责人

聚 焦 前 沿 技 术　　传 递 实 践 经 验

主办方 Geekbang>. 极客邦科技 InfoQ

当泛型遇上协议

Generic Programming with Protocol in Swift

蓝晨钰 猿题库 iOS 团队负责人

什么是泛型编程？

不用泛型

```
func swapInt(inout a: Int, inout _ b: Int) {
    let tmp = a
    a = b
    b = tmp
}
var someInt = 1
var anotherInt = 5
swapInt(&someInt, &anotherInt)


func swapString(inout a: String, inout _ b: String) {
    let tmp = a
    a = b
    b = tmp
}
```

用泛型

```
func swapValue<T>(inout a: T, inout _ b: T) {
    let tmp = a
    a = b
    b = tmp
}


var someInt = 1
var anotherInt = 5
var someString = "some"
var anotherString = "another"
swapValue(&someInt, &anotherInt)
swapValue(&someString, &anotherString)
```

泛型编程是一种编程范式，它允许程序员在编写代码时不需要立刻指定类型，而是在实例化后再指明这些类型

我们可以用泛型做什么

泛型枚举

```
// ? 是 Optional 的语法糖，这两种写法的声明的类型是相同的
var optInt: Int?
var optInt2: Optional<Int>
```

```
public enum Optional<Wrapped> : … {
    case None
    case Some(Wrapped)
    …
}
```

```swift
enum FailableData<T> {
    case Error(Error)
    case Data(T)
}
```

猿题库里的一个实例，completion 返回的对象类型是 FailableData<Exercise>

```swift
dataController.requestOrCreateExerciseWithTopicTask
(task, completion: { data in
   switch data {
   case .Data(let exercise):
        // 这里的 exercise 的类型为 Exercise，并不会丢失
   case .Error(let error):
        // 处理 Error
   }
})
```

泛型参数可以添加约束

JSONSerializable

```
public protocol JSONSerializable {
    func toJSONObject() -> JSONObject
}


class Man: JSONSerializable {
    var name: String?
    var son: Man?

    func toJSONObject() -> JSONObject {
        var json = JSONDictionary()
        name --> json["name"]
        son --> json["son"]
        return json
    }
}

man.toJSONObject()
```

支持各种类型
支持嵌套

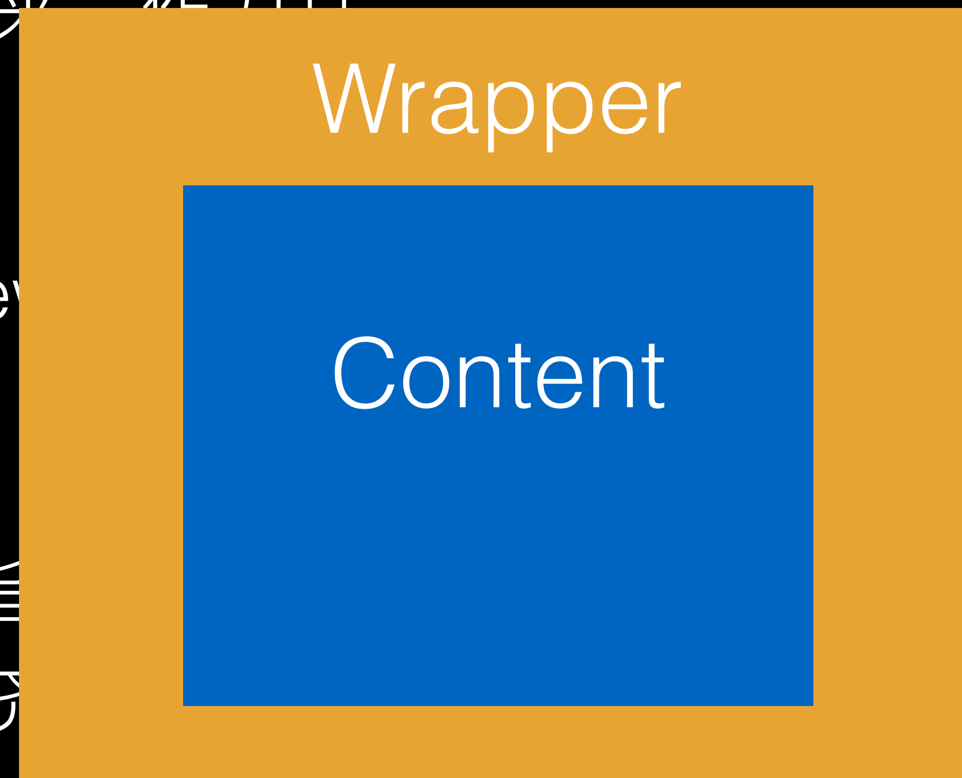
```swift
struct Serialization {

    static func convertAndAssign<T: JSONSerializable>(property: T?, 
inout toJSONObject jsonObject: JSONObject?) -> JSONObject? {
        if let property = property {
            jsonObject = property.toJSONObject()
        }
        return jsonObject
    }

}

infix operator --> { associativity right precedence 150 }

public func --> <T: JSONSerializable>(property: T?, inout jsonObject: 
JSONObject?) -> JSONObject? {
    return Serialization.convertAndAssign(property, toJSONObject: 
&jsonObject)
}


extension String: JSONSerializable {
    public func toJSONObject() -> JSONObject {
        return self
    }
}
```

CaesarParser 的一部分，已开源：https://github.com/lancy/CaesarParser

泛型可以不丢失类型信息

GenericViewWrapper

- ViewWrapper 能把传入的 contentView 包起来 subview
- 传入的 contentVie 为 UIView
- ViewWrapper 能指 contentView 的对
- contentView 的类型不能丢失



```swift
class GenericViewWrapper<ContentView: UIView>: UIView {

    /// The configurator type
    typealias Configurator = (config: GenericViewWrapper<ContentView>)
-> ()

    /// Vertical alignment
    var verticalAlignment: AlignmentMode = .Fill

    /// Horizontal alignment
    var horizontalAlignment: AlignmentMode = .Fill

    /// The content view
    var contentView: ContentView

    /// Init
    init(contentView: ContentView, configurator: Configurator? = nil) {
        self.contentView = contentView
        super.init(frame: CGRectZero)
        configurator?(config: self)
        setupUserInterface()
    }

    // implementation …
}
```

声明一个 labelWrapper：水平左对齐并且有15pt的padding，垂直居中对齐

```
lazy var labelWrapper: GenericViewWrapper<UILabel> = {
    let label = UILabel()
    label.text = "Label"
    label.font = UIFont.systemFontOfSize(12)
    let wrapper = GenericViewWrapper(contentView: label)
    { config in
      config.horizontalAlignment = .LeadingPadding(15)
      config.verticalAlignment = .Center
    }
    return wrapper
}()
```

之后直接使用 labelWrapper 就像直接使用 UILabel 一样，并不会丢失类型信息

```
labelWrapper.contentView.text = "Text"
```

泛型类也可以继承

假如我们需要一个 ViewWrapper 可以响应点击事件

```swift
class GenericTouchableViewWrapper<ContentView: UIView>: 
GenericViewWrapper<ContentView> {

    private let target: AnyObject

    private let action: Selector

    ///  Create and return a touchable view wrapper that wrap 
`contentView` and send `action` to `target` while user touch up 
inside the wrapper view. Can be config with `configurator`.
    init(contentView: ContentView, target: AnyObject, action: 
Selector, configurator: Configurator? = nil) {
        self.target = target
        self.action = action
        super.init(contentView: contentView, configurator: 
configurator)
    }

    //  Handle touch event and send action to target…
}
```

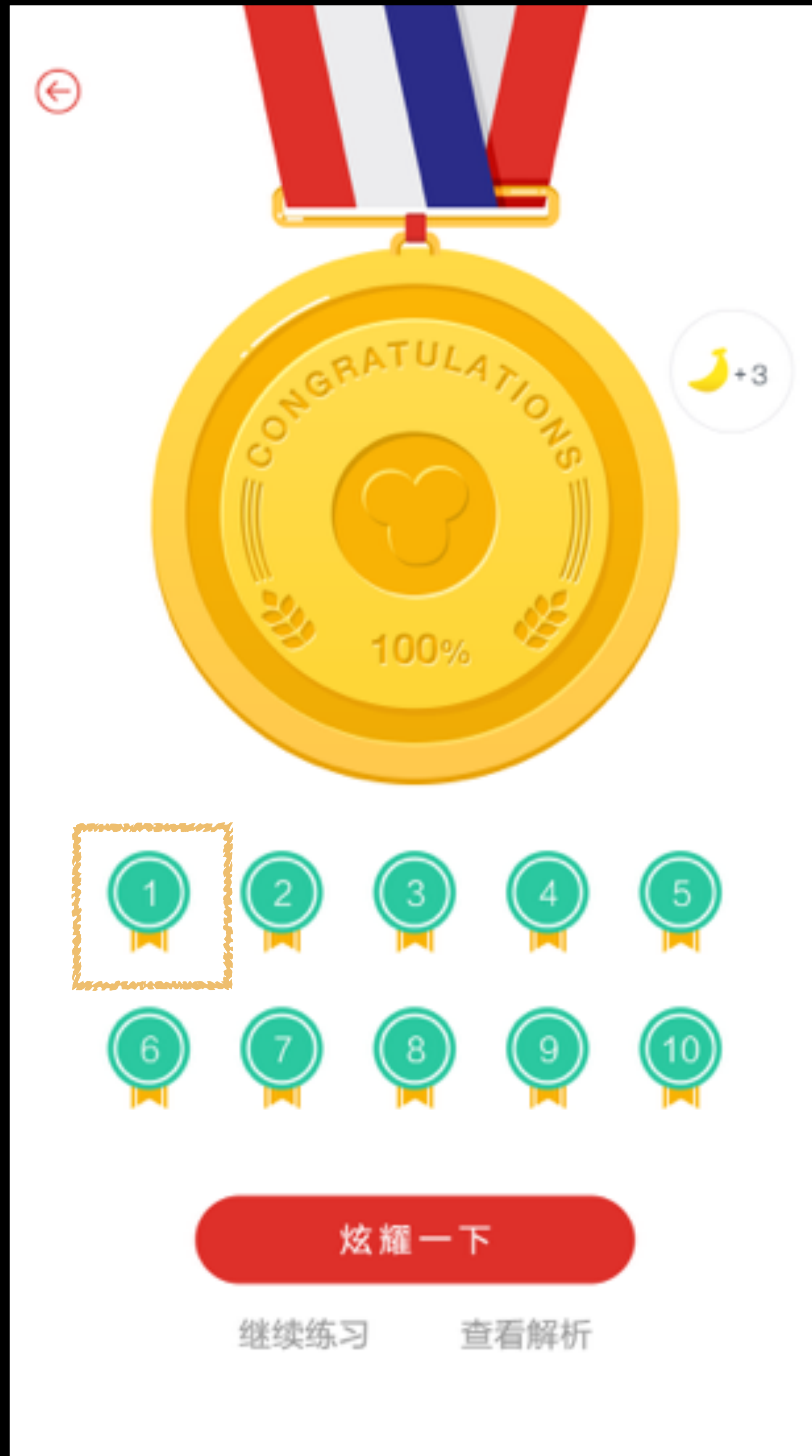
更多的例子

```
// 一个 ScrollViewWrapper，使被包住的 ContentView 如同获得了滚动的能力
class GenericScrollableViewWrapper<ContentView: UIView>: UIScrollView


// 一个可以保留item类型的 StackLayoutView
class GenericStackLayoutView<ArrangedView: UIView>: UIView
```

这么多泛型特性加上协议，我们可以做出什么？

泛型编程的目标?
以最小的假定来实现算法

答题卡



把相似的部分抽象为算法

- 实现一个答题卡的布局
- 由多个SectionView组成
- 每个SectionView有一个HeaderView和一组ItemView

不同的部分抽象为最小假定

- 不关心 ItemView 是什么，只需要假定他是固定大小的一组 View
- 不关心 SectionHeaderView 是什么，只需要假定他是固定高度的一个 View
- ItemView 和 SectionHeaderView 能通过 ViewModel 进行装配

实现一个可以用 ViewModel 装配的答题卡

```swift
func makeViewModel(paper: Paper) -> ViewModel {
    let sectionViewModels = paper.chapters.map { chapter in
        let headerViewModel = HeaderViewModel(chapter)
        let itemViewModels = chapter.questions.map { question in
            return ItemViewModel(question)
        }
        return SectionViewModel(headerViewModel, itemViewModels)
    }
    return ViewModel(sectionViewModels)
}

let cardView = CardView(viewModel: makeViewModel())
```

假定：能够使用ViewModel装配

```swift
///  Able to be created with default initializer
protocol Creatable {

    ///  Default initializer
    ///
    ///  - returns: Created instance
    init()

}

/// Support view model
protocol ViewModeled: class {

    /// View model type
    associatedtype ViewModel

    func bindDataWithViewModel(viewModel: ViewModel)
}


///  Both Creatable and ViewModeled
typealias CreatableViewModeled = protocol<Creatable, ViewModeled>
```

假定：StaticSize和StaticHeight

```swift
///  Size is constant
protocol StaticSize: StaticHeight, StaticWidth {

    /// Static height
    static var staticSize: CGSize { get }

}
```

进入复杂的部分

Protocol	ViewModelClass	ViewClass
GenericCard ViewModelType	GenericCard ViewModel	GenericCard View
GenericCard SectionViewModelType	GenericCard SectionViewModel	GenericCard SectionView
	SectionHeader ViewModel	SectionHeader View
		GenericCard SectionContentView
	ItemViewModel	ItemView

有 Generic 前缀的部分是抽象出来的可复用逻辑
非 Generic 前缀的部分是使用时实现的业务逻辑

GenericCard
ViewModelType

```
protocol GenericCardViewModelType: class {

    associatedtype SectionViewModelType: GenericCardSectionViewModelType

    var sectionViewModels: [SectionViewModelType] { get set }

}
```

GenericCard
SectionViewModelType

```
protocol GenericCardSectionViewModelType: class {
    associatedtype ItemViewType: UIView, CreatableViewModeled, StaticSize

    associatedtype SectionHeaderViewType: UIView, CreatableViewModeled,
StaticHeight

    var itemViewModels: [ItemViewType.ViewModel] { get set }

    var sectionHeaderViewModel: SectionHeaderViewType.ViewModel? { get
set }
}
```

泛型协议作为约束的
泛型ViewModel

GenericCard
ViewModel

```
class GenericCardViewModel<SectionViewModelType:
GenericCardSectionViewModelType>: GenericCardViewModelType {
      var sectionViewModels: [SectionViewModelType]

      init(sectionViewModels: [SectionViewModelType]) {
          self.sectionViewModels = sectionViewModels
      }


}
```

GenericCard SectionViewModel

```
class GenericCardSectionViewModel
  <ItemViewType: UIView,
   SectionHeaderViewType: UIView
   where
   ItemViewType: protocol<CreatableViewModeled, StaticSize>,
   SectionHeaderViewType: protocol<CreatableViewModeled, StaticHeight>
  > : GenericCardSectionViewModelType {

    var itemViewModels: [ItemViewType.ViewModel]
    var sectionHeaderViewModel: SectionHeaderViewType.ViewModel?

    init(itemViewModels: [ItemViewType.ViewModel], sectionHeaderViewModel:
SectionHeaderViewType.ViewModel? = nil) {
        self.itemViewModels = itemViewModels
        self.sectionHeaderViewModel = sectionHeaderViewModel
    }

}
```

GenericCard View

```
final class GenericCardView<ViewModelType: GenericCardViewModelType> :
UIView {

    weak var delegate: GenericCardViewDelegate?

    init(viewModel: ViewModelType) {
        super.init(frame: CGRect.zero)
        var subviews = [UIView]()
        for (index, svm) in viewModel.sectionViewModels.enumerate() {
            let sectionView =
GenericCardSectionView<ViewModelType.SectionViewModelType>(viewModel: svm,
delegate: self, index: index)
            subviews.append(sectionView)
        }

        let list = StackLayoutView(arrangedSubviews: subviews)
        addSubview(list)
        LayoutUtils.setEdgesFor(list)

    }

}
```

```swift
final class GenericCardSectionView<ViewModelType:
GenericCardSectionViewModelType>: UIView {

    typealias ItemViewType = ViewModelType.ItemViewType
    typealias SectionHeaderViewType = ViewModelType.SectionHeaderViewType

    init(viewModel: ViewModelType, delegate:
GenericCardSectionContentViewDelegate, index: Int) {
        super.init(frame: CGRectZero)
        var subviews = [UIView]()
        if let headerViewModel = viewModel.sectionHeaderViewModel {
            let headerView = SectionHeaderViewType()
            headerView.bindDataWithViewModel(headerViewModel)
            LayoutUtils.setHeightFor(headerView, height:
SectionHeaderViewType.staticHeight)
            subviews.append(headerView)
        }

        let contentView =
GenericCardSectionContentView<ItemViewType>(itemViewModels:
viewModel.itemViewModels)
        contentView.delegate = delegate
        contentView.tag = index
        subviews.append(contentView)

        let list = StackLayoutView(arrangedSubviews: subviews)
        addSubview(list)
        LayoutUtils.setEdgesFor(list)
    }

}
```

GenericCard SectionView

SectionHeader View

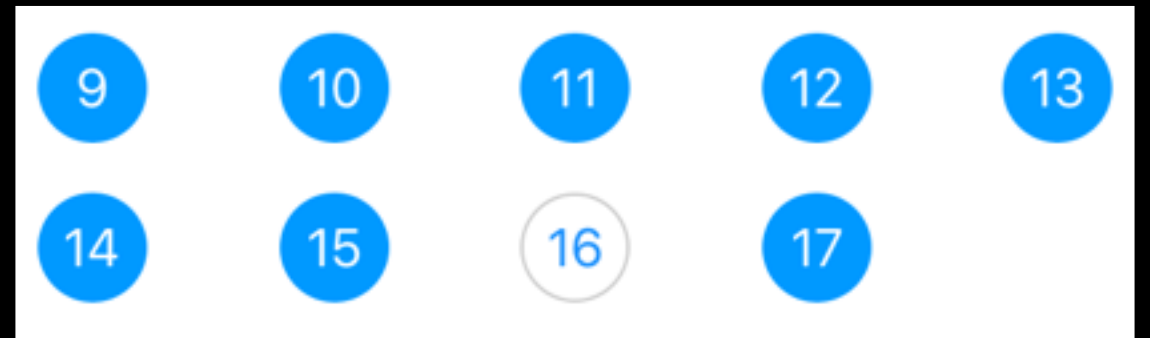
GenericCard SectionContentView

GenericCard
SectionContentView

```
final class GenericCardSectionContentView<ItemViewType: UIView where
ItemViewType: protocol<CreatableViewModeled, StaticSize>>: UIView {
    weak var delegate: GenericCardSectionContentViewDelegate?

    typealias ItemViewModel = ItemViewType.ViewModel

    private let itemViewModels: [ItemViewModel]

    init(itemViewModels: [ItemViewModel]) {
        self.itemViewModels = itemViewModels
        super.init(frame: CGRect.zero)
        setupUserInterface()
    }
}
```

额外的内容：函数式和链式调用

```
private func setupUserInterface() {

    let list = itemViewModels.split(step:5).map { vmsSlice in

    Array(vmsSlice).map { vm in
            let itemView = ItemViewType()
            itemView.bindDataWithViewModel(vm)
            itemView.tag == index
            index += 1
            return itemView.touchableWrapped(self, action:
#selector(itemViewPressed))
    }
    .stackLayoutWrapped { config in
            config.axis = .Horizontal
            config.alongAlignment = .Fill
            config.distribution = .EqualSpacing
        }
    }
    .stackLayoutWrapped()

    addSubview(list)
    LayoutUtils.setEdgesFor(list)

}
```



泛型代理回调

```
@objc private func itemViewPressed(view: UIView) {
    delegate?.cardSectionContentView(self, didTapItemAtIndex: view.tag)
}
```

```
protocol GenericCardSectionContentViewDelegate: class {

    func cardSectionContentView<ContentViewType: UIView>(contentView: ContentViewType, didTapItemAtIndex index: Int)
}
```

泛型协议与 objc 不兼容，所以这里只能使用 UIView

```
protocol GenericCardViewDelegate: class {

    func cardView<CardViewType: UIView>(contentView: CardViewType,
didTapItemAtIndexPath indexPath: NSIndexPath)

}
```

```
extension GenericCardView: GenericCardSectionContentViewDelegate {

    func cardSectionContentView<T: UIView>(contentView: T,
didTapItemAtIndex index: Int) {
        let indexPath = NSIndexPath(forRow: index, inSection:
contentView.tag)
        delegate?.cardView(self, didTapItemAtIndexPath: indexPath)
    }

}
```

搞定
使用时是什么体验？

实现 ItemView 和 HeaderView

```
final class DemoCardItemView: UIView, CreatableViewModeled,
StaticSize {
    init() {……}

    static var staticSize: CGSize = CGSize(width: 66, height: 66)

    struct ViewModel {
        let text: String
        let textColor: UIColor
        let image: UIImage
    }

    func bindDataWithViewModel(viewModel: ViewModel) {
        label.text = viewModel.text
        label.textColor = viewModel.textColor
        imageView.image = viewModel.image
    }

  ……
}

final class DemoCardChapterView: LabelWrapper, CreatableViewModeled,
StaticHeight {
  ……
}
```

定义别名和构建ViewModel

```swift
typealias DemoCardSectionViewModel =
GenericCardSectionViewModel<DemoCardItemView,
DemoCardChapterView>

typealias DemoCardViewModel =
GenericCardViewModel<DemoCardSectionViewModel>


typealias DemoCardView = GenericCardView<DemoCardViewModel>



func makeViewModel(data: …) -> DemoCardViewModel {
    // 根据传入的数据构建 viewModel
}
```

使用

```swift
let cardView = DemoCardView(viewModel: makeViewModel())
cardView.delegate = self
```

```swift
let scrollableCardView = cardView.scrollableWrapped()
view.addSubview(scrollableCardView)
```

就是这么简单

发生了什么?

- 我们实现了一个 GenericCardView，使用时通过泛型参数指定 ItemView 和 HeaderView 的类型
- 只需要实现 ItemView 和 HeaderView 各自对应的 ViewModel 就能获得一个泛型的 CardViewModel
- 使用这个 CardViewModel 就能正确渲染 CardView

泛型编程的目标
以最小的假定来实现算法

One More Thing

GenericTableViewModeled
和
GenericCollectionViewModeled

通过面向协议编程自动实现 delegate 和 dataSource 回调

```swift
///  Generic collection view modeled.
protocol GenericCollectionViewModeled: ViewModeled {

    /// View model type.
    associatedtype ViewModel: GenericCollectionViewModelType

    /// Collection view model.
    var viewModel: ViewModel? { get set }

    /// Cell identifier, for generating and getting cell.
    var cellIdentifier: String { get }

    /// Collection view.
    var collectionView: UICollectionView { get }

    ///  Return number of sections from view model
    func numberOfSections() -> Int

    ///  Return number of items in a `section` from view model
    func numberOfItemsInSection(section: Int) -> Int

    ///  Return section view model in given `section`
    func sectionViewModelInSection(section: Int) ->
ViewModel.SectionViewModel?

    ///  Return cell view model at `indexPath`
    func cellViewModelAtIndexPath(indexPath: NSIndexPath) ->
ViewModel.SectionViewModel.CellView.ViewModel?

    ///  Delete cell view model at `indexPath`
    func deleteCellViewModelAtIndexPath(indexPath: NSIndexPath) ->
Bool

    ///  Generate/get cell at `indexPath`
    func dequeueReusableCellAndBindDataForIndexPath(indexPath:
NSIndexPath) ->
GenericCollectionCell<ViewModel.SectionViewModel.CellView>
```

```swift
extension GenericCollectionViewModeled {

    /// Default cell identifier
    var cellIdentifier: String {
        return "CellIdentifier"
    }

    ///  Default implementation of binding data with view mo
    ///
    ///  - parameter viewModel: The view model
    func bindDataWithViewModel(viewModel: ViewModel) {
        self.viewModel = viewModel
        collectionView.reloadData()
    }

    ///  Default implementation of getting section number fo
    func numberOfSections() -> Int {
        return viewModel?.sectionViewModels.count ?? 0
    }

    ///  Default implementation of getting items number for
    func numberOfItemsInSection(section: Int) -> Int {
        return viewModel?.sectionViewModels.elementAtIndex(s
    }

    ///  Default implementation of getting section view mode
    func sectionViewModelInSection(section: Int) -> ViewMode
        return viewModel?.sectionViewModels.elementAtIndex(s
    }

    ///  Default implementation of getting cell view model f
    func cellViewModelAtIndexPath(indexPath: NSIndexPath) ->
        return viewModel?.sectionViewModels.elementAtIndex(i
                         .cellViewModels.elementAtIndex(inde
    }
```

通过泛型参数指定 CellViewType, HeaderViewType, FooterViewType

```
final class AnswerCardView: CollectionViewContainer,
GenericCollectionViewModeled {

    typealias CellView = ItemView
    typealias SectionHeaderView = ChapterView
    typealias ViewModel =
GenericSectionGroupViewModel<GenericSectionViewModel<CellView,
SectionHeaderView, VoidView>>

    …
}
```

根据View的类型实现不同逻辑

```
extension GenericCollectionViewModeled where
ViewModel.SectionViewModel.CellView: StaticSize {
    /// Static item size
    var staticItemSize: CGSize {
        return ViewModel.SectionViewModel.CellView.staticSize
    }
    ///  Return size for item at 'indexPath'.
    func sizeForItemAtIndexPath(indexPath: NSIndexPath) -> CGSize {
        return staticItemSize
    }
}

extension GenericCollectionViewModeled where
ViewModel.SectionViewModel.CellView: ViewModeledSize {
    ///  Return size for item at 'indexPath'.
    func sizeForItemAtIndexPath(indexPath: NSIndexPath) -> CGSize {
        guard let cvm = cellViewModelAtIndexPath(indexPath) else { return
CGSize.zero }
        return ViewModel.SectionViewModel.CellView.sizeWithViewModel(cvm)
    }

}
```

只关心你想关心的

Thanks
Q&A

We're Hiring!

lancy@fenbi.com