2.1

Swift 中的协议编程

Protocol Oriented Programming @Swift

@李洁信

Word lookup

Different word definitions

Kingsoft Def

Youdao Def

Baidu Def

render

View

Swift
[swɪft]
n. 褐雨燕
n. 苹果公司推出的编程语言

Protocols as Types

```
protocol WordType {
    var name: String  { get }
    var pronunciation: String { get }
    var definition: String { get }
}

struct KingsoftWord: WordType {
    //实现WordType的所有属性和方法
}

struct YoudaoWord: WordType {
    //实现WordType的所有属性和方法
}

struct BaiduWord: WordType {
    //实现WordType的所有属性和方法
}
```

Render

```
func renderWord(word: protocol<WordType>) {
    print(word.name)
    print(word.pronunciation)
    print(word.defination)
}

let kingsoftWord = KingsoftWord()
renderWord(kingsoftWord)

let youdaoWord = YoudaoWord()
renderWord(youdaoWord)

let baiduWord = BaiduWord()
renderWord(baiduWord)
```

Protocols in ObjC

- Delegation

  UIApplicationDelegate, UITableViewDataSource,

  UITableViewDelegate

- Share Similarities

  NSCoding, NSCopying

- Low-level syntax features

  NSFastEnumeration

- Hide Implementations

Protocols in Swift

- Support Classes & Structs & Enums
- Requirements
  - Property, Initializer, Class & Instance methods
- Protocol Inheritance
- Protocol Composition
- Protocol Extension

Classes & Structs & Enums

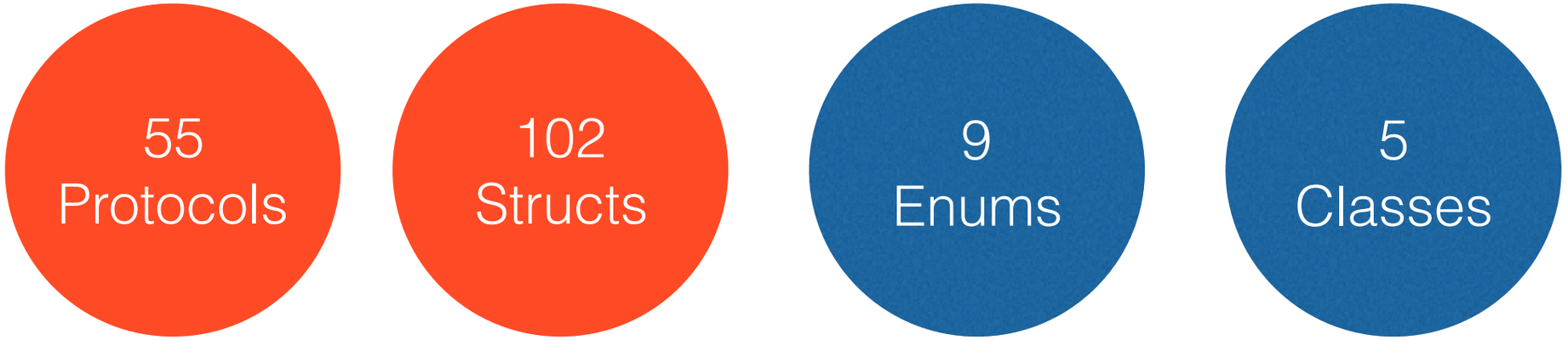
Class

- Properties
- Methods
- Reference Type
- Inheritance
- Protocol

Struct & Enum

- Properties
- Methods
- Value Type
- No Inheritance
- Protocol

The Swift Standard Library



55 protocols

-Type

BooleanType
CollectionType
ErrorType
GeneratorType
IntegerType
OptionSetType
SequenceType
…

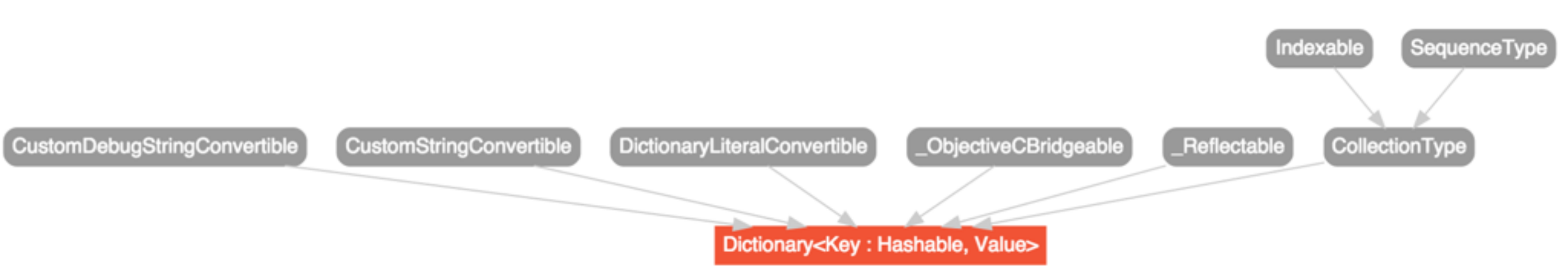
-able

Comparable
Equatable
Hashable
Indexable
RawRepresentable
Streamable
Strippable
CustomPlayground-
Quicklookable
…

-Convertible

ArrayLiteralConvertible
BooleanLiteralConvertible
CustomDebugStringConv
ertible
CustomStringConvertible
NilLiteralConvertible
StringInterpolationConvert
ible
StringLiteralConvertible
FloatLiteralConvertible
IntegerLiteralConvertible
…

Dictionary



Dictionary

```
public protocol CustomDebugStringConvertible {
    /// debugPrint
    public var debugDescription: String { get }
}

public protocol CustomStringConvertible {
    /// print
    public var description: String { get }
}

public protocol DictionaryLiteralConvertible {
    typealias Key
    typealias Value
    /// Create an instance initialized with `elements`.
    public init(dictionaryLiteral elements: (Self.Key, Self.Value)...)
}
```

CollectionType

Instance Variables

```
var count: Self.Index.Distance
var first: Self.Generator.Element?
var isEmpty: Bool
var endIndex: Self.Index REQUIRED
var startIndex: Self.Index REQUIRED
```

CollectionType

Subscripts

```
subscript(_: Range<Self.Index>) REQUIRED
subscript(_: Self.Index) REQUIRED
```

Instance Methods

```
func dropFirst(_:)
func dropLast(_:)
func filter(_:)
func forEach(_:)
func generate() REQUIRED
func map(_:)
func prefix(_:)
func prefixThrough(_:)
func prefixUpTo(_:)
func split(_:allowEmptySlices:isSeparator:)
func suffix(_:)
func suffixFrom(_:)
func underestimateCount()
```

Default Implementations

```
var count: Self.Index.Distance
var first: Self.Generator.Element?
var indices: Range<Self.Index>
var isEmpty: Bool
var lazy: LazyCollection<Self>
func contains(_:)
func dropFirst()
func dropFirst(_:)
func dropLast()
func dropLast(_:)
func elementsEqual(_:isEquivalent:)
func enumerate()
func filter(_:)
func flatMap<T>(_: (Self.Generator.Element) throws
func flatMap<S : SequenceType>(_: (Self.Generator.E
func forEach(_:)
func indexOf(_:)
func lexicographicalCompare(_:isOrderedBefore:)
func map(_:)
func maxElement(_:)
```

Challenge

通过数组赋值的方式初始化Struct

```
let p1: Person = ["Bruce", "10"]
```

ArrayLiteralConvertible

```
public protocol ArrayLiteralConvertible {
    typealias Element
    /// Create an instance initialized with `elements`.
    public init(arrayLiteral elements: Self.Element...)
}
```

Person

```
struct Person: ArrayLiteralConvertible {
    var name: String = ""
    var id: String = ""

    typealias Element = String
    init(arrayLiteral elements: Element...) {
        if elements.count == 2 {
            name = elements[0]
            id = elements[1]
        }
    }
}
```

Equatable

```
extension Person: Equatable {}
func == (p1: Person, p2: Person) -> Bool {
    return p1.id == p2.id
}
```

Test

```
let p1: Person = ["Bruce", "10"]
print(p1)
```

```
let p2: Person = ["布鲁斯", "10"]
print(p2)

print(p1 == p2)
```

Test

```
let p1: Person = ["Bruce", "10"]
print(p1)
```

```
Person(name: "Bruce", id: "10")
```

```
let p2: Person = ["布鲁斯", "10"]
print(p2)

print(p1 == p2)
```

Test

```
let p1: Person = ["Bruce", "10"]
print(p1)
```

```
Person(name: "Bruce", id: "10")
```

```
let p2: Person = ["布鲁斯", "10"]
print(p2)
```

```
Person(name: "布鲁斯", id: "10")
```

```
print(p1 == p2)
```

Test

```
let p1: Person = ["Bruce", "10"]
print(p1)
```

```
Person(name: "Bruce", id: "10")
```

```
let p2: Person = ["布鲁斯", "10"]
print(p2)
```

```
Person(name: "布鲁斯", id: "10")
```

```
print(p1 == p2)
```

```
true
```

Inheritance

```
protocol WordType: CustomStringConvertible {
    var name: String  { get }
    var pronunciation: String { get }
    var definition: String { get }
}
```

```
struct BaiduWord: WordType {
    var name: String
    var pronunciation: String
    var definition: String

    var description: String {
        return "\(name), \(pronunciation), \(definition)"
    }
}
```

Composition

```
protocol WordType {
    var name: String  { get }
    var pronunciation: String { get }
    var definition: String { get }
}

protocol Speakable {
    func speak()
}

typealias SpeakableWordType = protocol<Speakable, WordType>
```

Swift 2.0

Protocol Extension

```
protocol Hello {
    func sayHello()
}
```

```
extension Hello {
    func sayHello() {
        print("Hello!")
    }
}
```

Swift 2.0

Protocol Extension

```
struct Person: Hello {
}

let p = Person()
p.sayHello()
```

“Hello!”

Protocol Extensions

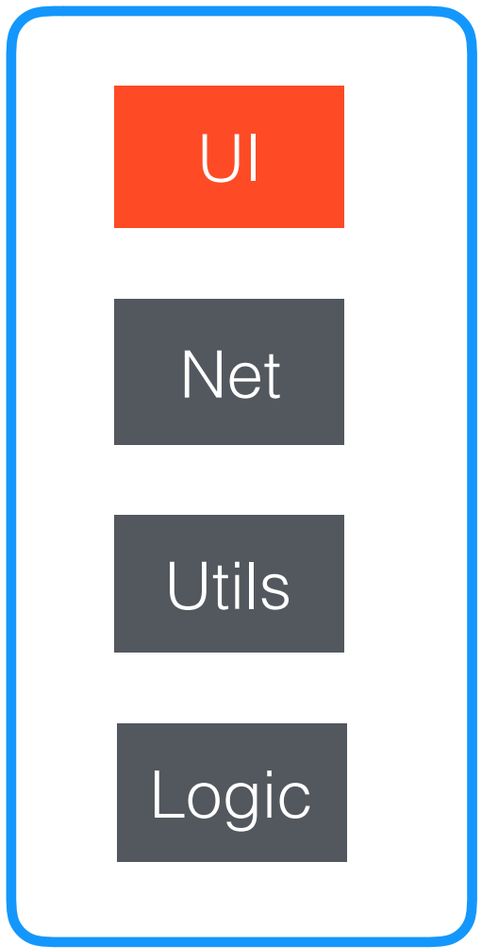
- Adopt protocols on existed Types
- Declaring Protocol Adoption
- Default Implementations
- Adding Constraints
- Generics (Associated Types)

Demo #1

God Class

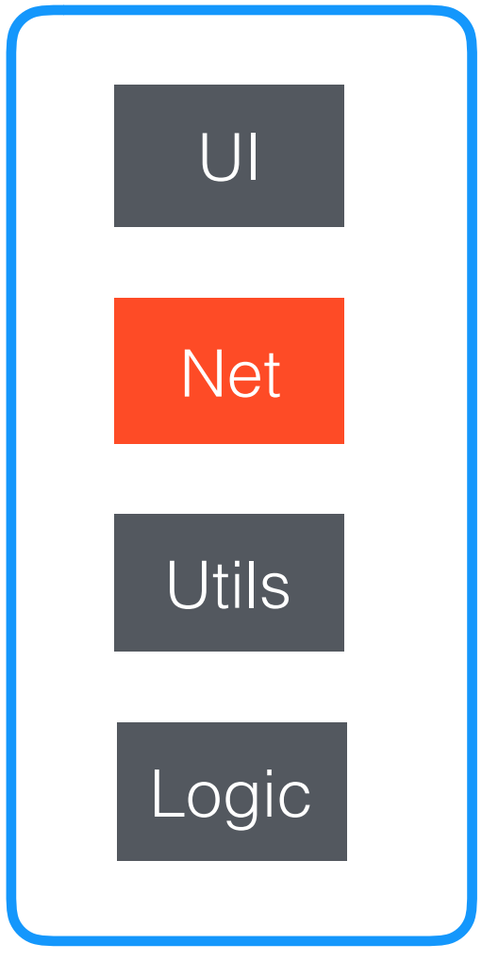

Base
BaseViewController.swift
BaseTableViewController.swift
BaseCollectionViewController.swift

UI helpers



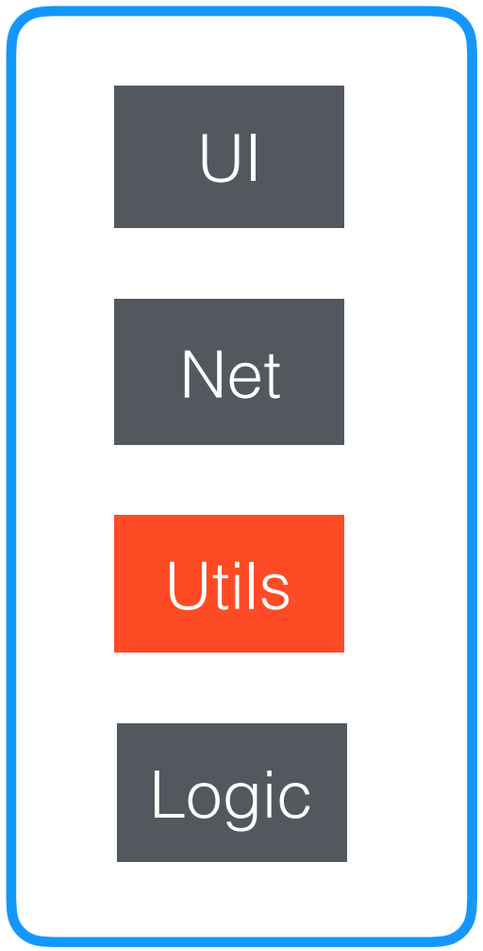

```
// Loading View
func setupLoadingView() {}
func setLoadingViewHidden(hidden: Bool) {}
func onLoadingViewTapped() {}

// Custom Back Button
func setBackButtonWithTitle(title: String) {}
func onBack() {}

// Custom Sidebar Button
func setupSidebarButton() {}
func setSidebarButtonHidden(hidden: Bool) {}
func onSidebarButtonTapped() {}
```

Network helpers



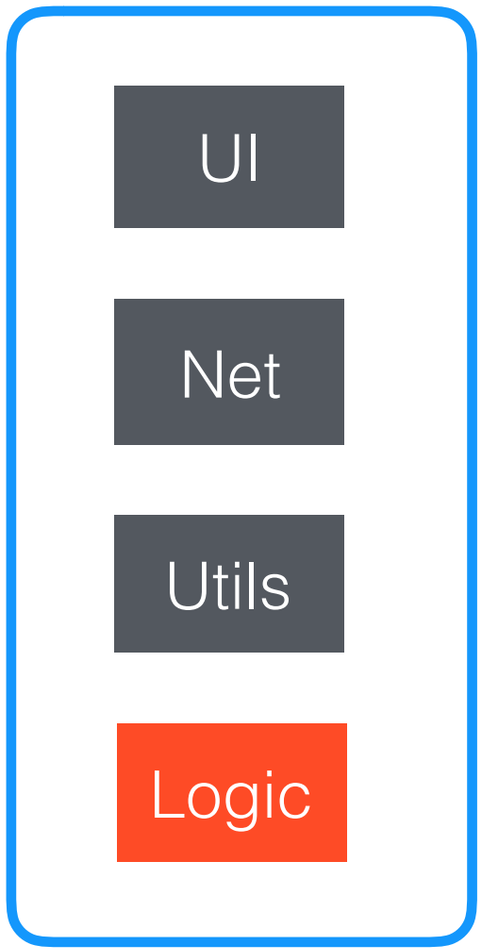

```
// Network
var requestQueue = NSOperationQueue()
func requestWithUrl(
    url: String,
    type: String,
    params: [String:AnyObject],
    handler: (Bool, AnyObject) -> Void
    ) -> NSOperation? {
        …
}
```

Utils helpers




```swift
// Utils
func showAlertWithTitle() { }
func showActionSheetWithTitles() { }

func validateUsername(name: String) -> Bool {}
func validateEmail(name: String) -> Bool {}
```

Logic




```
// Logics
func presentLoginController() { }
func showSharingPage() { }
func pushWebViewController(url: String) { }
```

The Problem

100+个子类继承了
BaseViewController

```
class ViewController: BaseViewController {
}
```

- 代码 > 1000 行, 难以维护
- 高耦合, 父类改动牵连所有子类
- 子类里继承了无用的实例变量和方法
- BaseTableViewController呢？复制代码?

Solution A

- Singleton
- Helpers & Static Methods

- 拆分了代码，但没有降低耦合
- 难以测试

Solution B

"Composition Over Inheritance."

LoadingView

(original version)

```
class BaseViewController: UIViewController {
    var loadingView: CustomLoadingView!

    func setupLoadingView() {
        //Create loadingView and add Tap ...
    }

    func setLoadingViewHidden(hidden: Bool) {
        loadingView?.hidden = hidden
    }

    func onLoadingViewTapped() {
        print("Reload page.")
    }
}
```

LoadingView

(composition version)

```
typealias ReloadHandler = (Void -> Void)
class LoadingPresenter {
    var loadingView: CustomLoadingView!
    var containerView: UIView!
    var reloadHandler: ReloadHandler?

    func setupLoadingView(container: UIView!, handler: ReloadHandler?) {
        containerView = container
        reloadHandler = handler
        //Create loadingView and add Tap ...
    }

    func setLoadingViewHidden(hidden: Bool) {
        loadingView?.hidden = hidden
    }
    func onLoadingViewTapped() {
        reloadHandler?()
    }
}
```

LoadingView

(composition version)

```
class ViewController: UIViewController {
    var presenter = LoadingPresenter()

    override func viewDidLoad() {
        super.viewDidLoad()
        presenter.setupLoadingView(self.view) {
            print("reload page.")
        }

        presenter.setLoadingViewHidden(false)
    }
}
```

But...

- 有点笨重，实现代码多
- 使用时要管理实例的创建和释放
- 使用时用通过间接变量, 多了一层结构

Protocol Version

```
protocol LoadingPresenter {
    var loadingView: CustomLoadingView! { get set }
    func setupLoadingView()
    func setLoadingViewHidden(hidden: Bool)
    func onLoadingViewTapped()
}

extension LoadingPresenter where Self: UIViewController {
    mutating func setupLoadingView() {
        //Create loadingView and add Tap ...
    }

    func setLoadingViewHidden(hidden: Bool) {
        loadingView?.hidden = hidden
    }
}
```

Protocol Version

```
class ViewController: UIViewController, LoadingPresenter {
    var loadingView: CustomLoadingView!

    override func viewDidLoad() {
        super.viewDidLoad()

        setupLoadingView()
        setLoadingViewHidden(false)
    }
}
```

Merits

- 代码没有增多，却更复用
- 面向协议(接口)，而不是实现，充分解耦
- 静态类型检查帮助在编译时发现问题
- 写代码像搭积木，先设计接口，再逐一实现
- 依赖少，更容易调试

More Protocols

```
class LoginController: UIViewController,
    Requestable, LoadingPresenter, AlertPresenter,
    UserNameValidator, EmailValidator {
    var loadingView: CustomLoadingView!

}
```

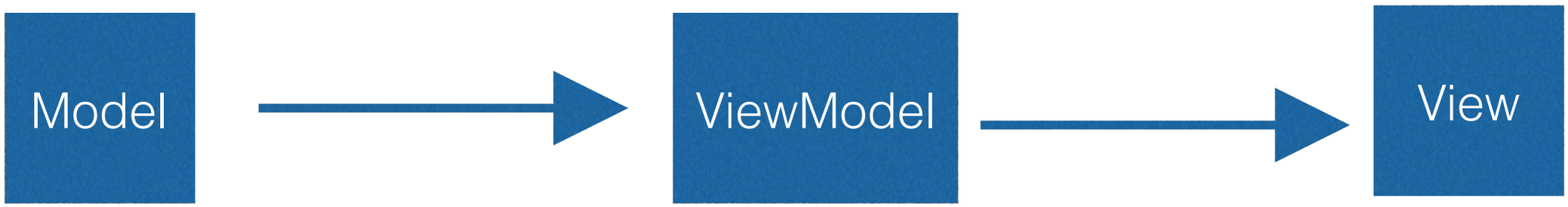
Demo #2

POP in MVVM

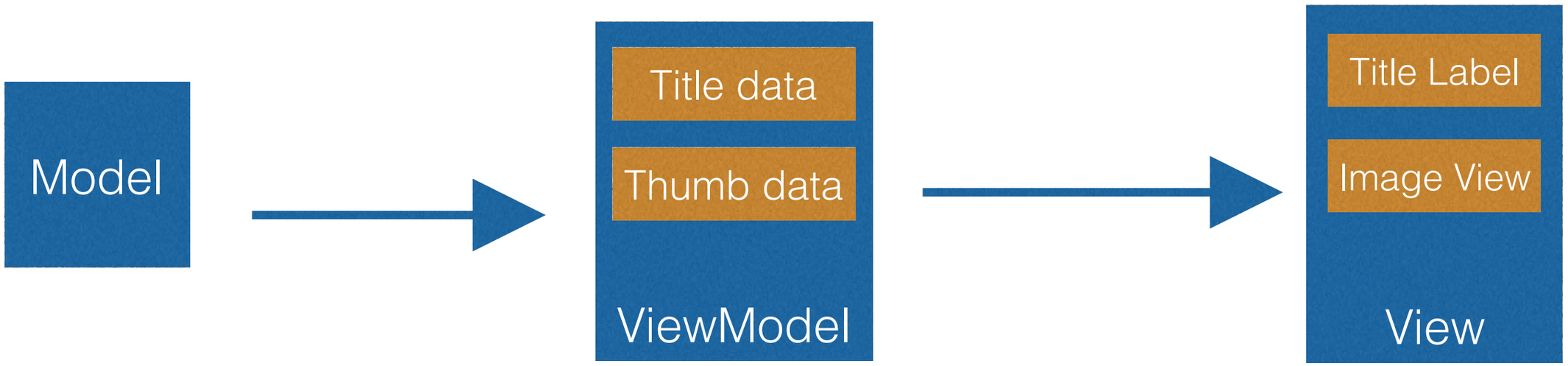
数据的加工以及格式化放哪里?
(e.g. NSDate -> “xxx分钟前”)


```
let cell = tableView.dequeueReusableCellWithIdentifier() …
let news = newsList[indexPath.row]
cell.updateWithNews(news)
```

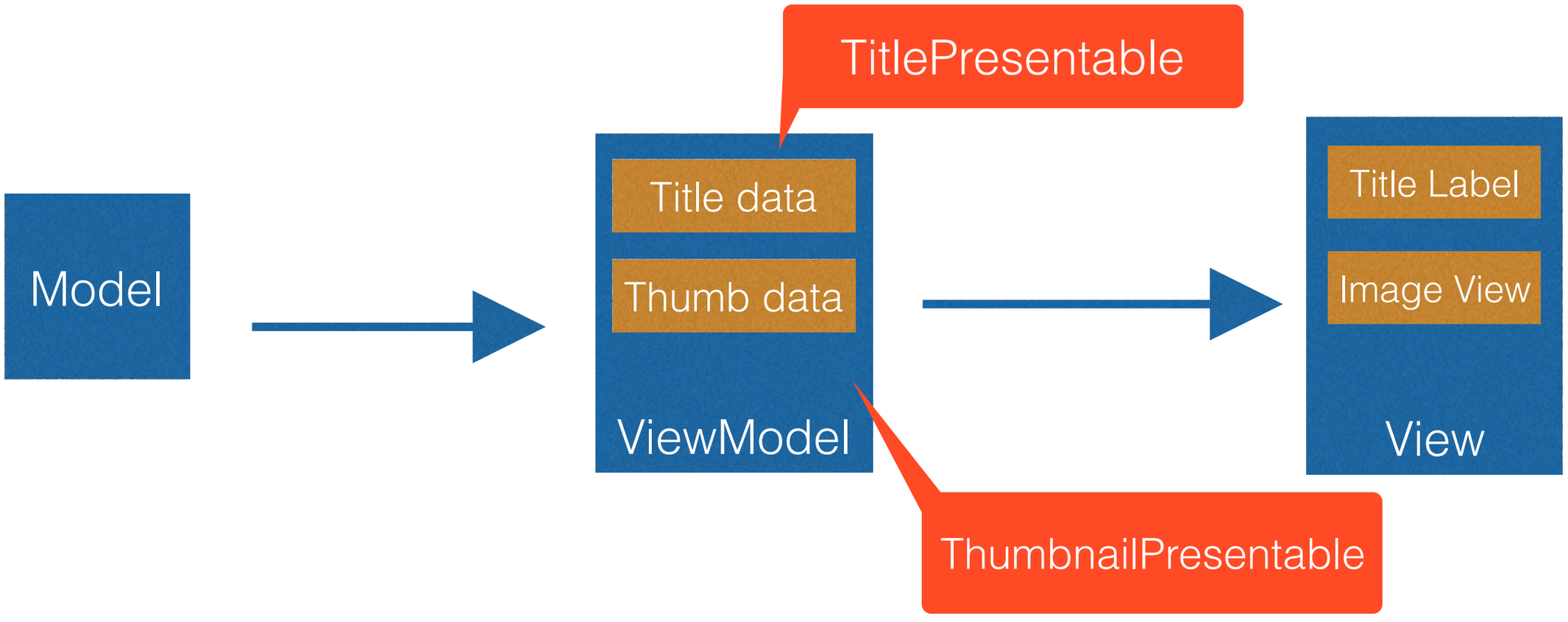
POP in MVVM



POP with MVVM



POP with MVVM



TitlePresentable

```
protocol TitlePresentable {
    var title: String { get }
    var titleColor: UIColor { get }
    var titleFont: UIFont! { get }
    func updateTitleLabel(label: UILabel)
}

extension TitlePresentable {
    var titleColor: UIColor {
        return UIColor.blackColor()
    }
    var titleFont: UIFont! {
        return UIFont(name: "Helvetica", size: 18)!
    }

    func updateTitleLabel(label: UILabel) {
        label.text = title
        label.textColor = titleColor
        label.font = titleFont
    }
}
```

ThumbnailPresentable

```
protocol ThumbnailPresentable {
    var thumbnailUrl: String { get }
    var thumbnailHandler: (Void -> Void)? { get }
    func updateImageView(imageView: UIImageView)
}

extension ThumbnailPresentable {
    var thumbnailHandler: (Void -> Void)? {
        return nil
    }

    func updateImageView(imageView: UIImageView) {
        //Set imageView with thumbnailUrl
        //Add tap Handler
    }

    func thumbnailImageViewtapped() {
        thumbnailHandler?()
    }
}
```

View

```
typealias NewsPresentable =
          protocol<ThumbnailPresentable, TitlePresentable>

class NewsCell: UITableViewCell {
    @IBOutlet private weak var titleLabel: UILabel!
    @IBOutlet private weak var headView: UIImageView!

    func updateWithPresenter(presenter: NewsPresentable) {
        presenter.updateTitleLabel(titleLabel)
        presenter.updateImageView(headView)
    }
}
```

ViewModel

```
struct NewsViewModel: NewsPresentable {
    var title: String
    var thumbnailUrl: String
    var thumbnailHandler: (Void -> Void)?

    init(news: News, thumbnailHandler: (Void -> Void)?) {
        self.text = news.title
        self.thumbnailUrl = news.thumbnailUrl
        self.thumbnailHandler = thumbnailHandler
    }
}
```

View Controller

```
class NewsController: UITableViewController {
    var newsList: [News] = []
    override func tableView(tableView: UITableView, …) -> UITableViewCell
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "NewsCell", forIndexPath: indexPath) as! NewsCell
        let news = newsList[indexPath.row]

        let viewModel = NewsViewModel(news: news) {
            print("image tapped.")
        }
        cell.updateWithPresenter(viewModel)


        return cell
    }
}
```

“这里需要显示时间!”

TimePresentable

```
protocol TimePresentable {
    var timeText: String { get }
    var timeColor: UIColor { get }
    var timeFont: UIFont! { get }
    func updateTimeLabel(label: UILabel)
}

extension TimePresentable {
    var timeColor: UIColor {
        return UIColor.blackColor()
    }
    var timeFont: UIFont! {
        return UIFont(name: "Helvetica", size: 18)!
    }

    func updateTimeLabel(label: UILabel) {
        label.text = timeText
        label.textColor = timeColor
        label.font = timeFont
    }
}
```

View

```
typealias NewsPresentable = protocol<
    TitlePresentable, ThumbnailPresentable, TimePresentable
>


class NewsCell: UITableViewCell {
    @IBOutlet private weak var titleLabel: UILabel!
    @IBOutlet private weak var headView: UIImageView!
    @IBOutlet private weak var timeLabel: UILabel!

    func updateWithPresenter(presenter: NewsPresentable) {
        presenter.updateTitleLabel(titleLabel)
        presenter.updateImageView(headView)
        presenter.updateTimeLabel(timeLabel)
    }
}
```

ViewModel

```
struct NewsViewModel: NewsPresentable {
    var title: String
    var thumbnailUrl: String
    var thumbnailHandler: (Void -> Void)?
    var timeText: String

    init(news: News, thumbnailHandler: (Void -> Void)?) {
        self.text = news.title
        self.thumbnailUrl = news.thumbnailUrl
        self.thumbnailHandler = thumbnailHandler
        self.timeText =
        "\(NSDate().timeIntervalSince1970-news.updatedAt)秒前"
    }
}
```

Merits

- 将UI配置和响应操作封装成protocol，代码更复用
- UI改动对代码影响很小
- 代码扁平化

Q&A

参考

- The Swift Programming Language (Swift 2.1) - Protocols
- The Swift Standard Library
- Mixing and Traits in Swift 2.0
- Updated: Protocol-Oriented MVVM in Swift 2.0
- WWDC 2014 Session 404 - Advanced Swift
- WWDC 2015 Session 408 - Protocol-Oriented Programming in Swift