面向协议编程与 Cocoa 的邂逅

MDCC 16

王巍 - 2016 年 9 月 24 日

主题

- 起 · 初识
- 承 · 相知
- 转 · 热恋
- 合 · 陪伴

主题

- 起・初识 - 什么是 Swift 协议
- 承・相知
- 转・热恋
- 合・陪伴

+ 面向对象范式的不足，用协议能解决什么

主题

- 起・初识 - 什么是 Swift 协议
- 承・相知 - 协议扩展和面向协议编程
- 转・热恋
- 合・陪伴

Swift 2 特有的

主题

- 起・初识 - 什么是 Swift 协议
- 承・相知 - 协议扩展和面向协议编程
- 转・热恋 - 在日常开发中使用协议
- 合・陪伴

结合一些我们在实际项目中使用的例子，说明面向协议编程为我们带来的改善

主题

- 起・初识 - 什么是 Swift 协议
- 承・相知 - 协议扩展和面向协议编程
- 转・热恋 - 在日常开发中使用协议
    - Model (Networking)
    - View
    - ViewController
- 合・陪伴

起・初识

什么是 Swift 协议

Protocol

```swift
protocol Greetable {
    var name: String { get }
    func greet()
}
```

Swift 标准库中有 50 多个复杂不一的协议，几乎所有的实际类型都是满足若干协议的。protocol 是 Swift 语言的底座，语言的其他部分正是在这个底座上组织和建立起来的。这和我们熟知的面向对象的构建方式很不一样。

面向对象

Object-oriented

在深入 Swift 协议的概念之前，我想先重新让大家重新回顾一下面向对象。相信大家不论在教科书或者是各种地方都对这个名词十分熟悉了。面向对象的核心思想是?

面向对象

```
class Animal {
    var leg: Int { return 2 }
    func eat() {
        print("eat food.")
    }
    func run() {
        print("run with \(leg) legs")
    }
}

class Tiger: Animal {
    override var leg: Int { return 4 }
    override func eat() {
        print("eat meat.")
    }
}

let tiger = Tiger()
tiger.eat() // "eat meat"
tiger.run() // "run with 4 legs"
```

是封装和继承，它将一系列相关的内容封装到类中，使用父类抽象共通的内容 Animal，接下来通过继承的方式重用，Tiger。我们的前辈们为了能够对真实世界的对象进行建模，发展出了面向对象编程的概念，但是这套理念有一些缺陷（比如 Animal.leg /多一层继承）。对象往往是一系列特质的组合，而不单单是一脉相承并逐渐扩展。所以最近大家越来越发现面向对象很多时候其实不能很好地对事物进行抽象。

ViewController → UIViewController

```
class ViewCotroller:
                    UIViewController
{
    // 继承
    // view, isFirstResponder()...

    // 新加
    func myMethod() {

    }
}
```

AnotherViewController → UITableViewController → UIViewController

```
class AnotherViewController:
                UITableViewController
{
    // 继承
    // tableView, isFirstResponder()...

    // 新加
    func myMethod() {

    }
}
```

困境之一

Cross-Cutting Concerns

横切关注点

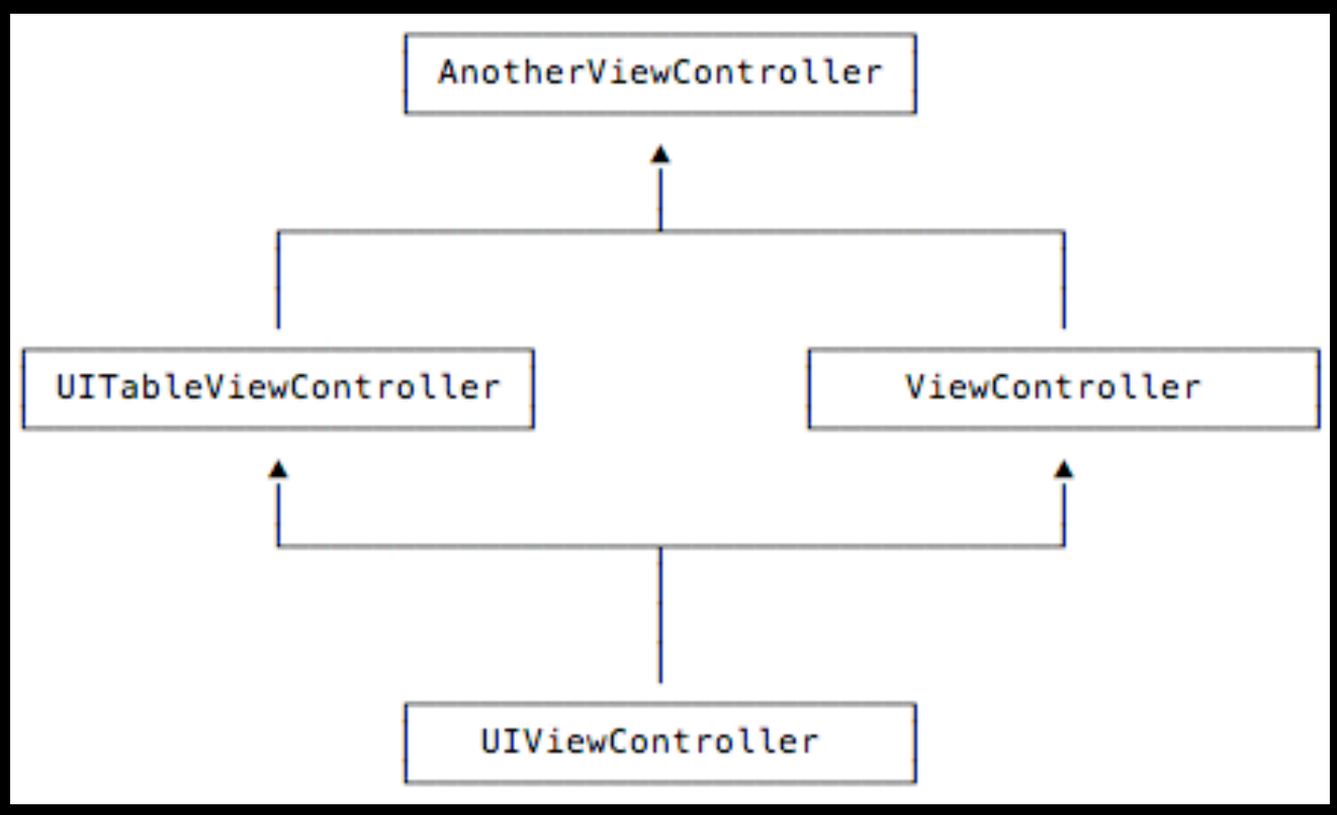
特性的组合要比继承更贴切事物的本质。面向对象是一种不错的抽象方式，但是肯定不是最好的方式。它无法描述两个不同事物具有某个相同特性这一点。

解决方案

- Copy & Paste
- 引入 BaseViewController
- 依赖注入
- 多继承

有一些语言选择了多继承...悲剧..菱形缺陷

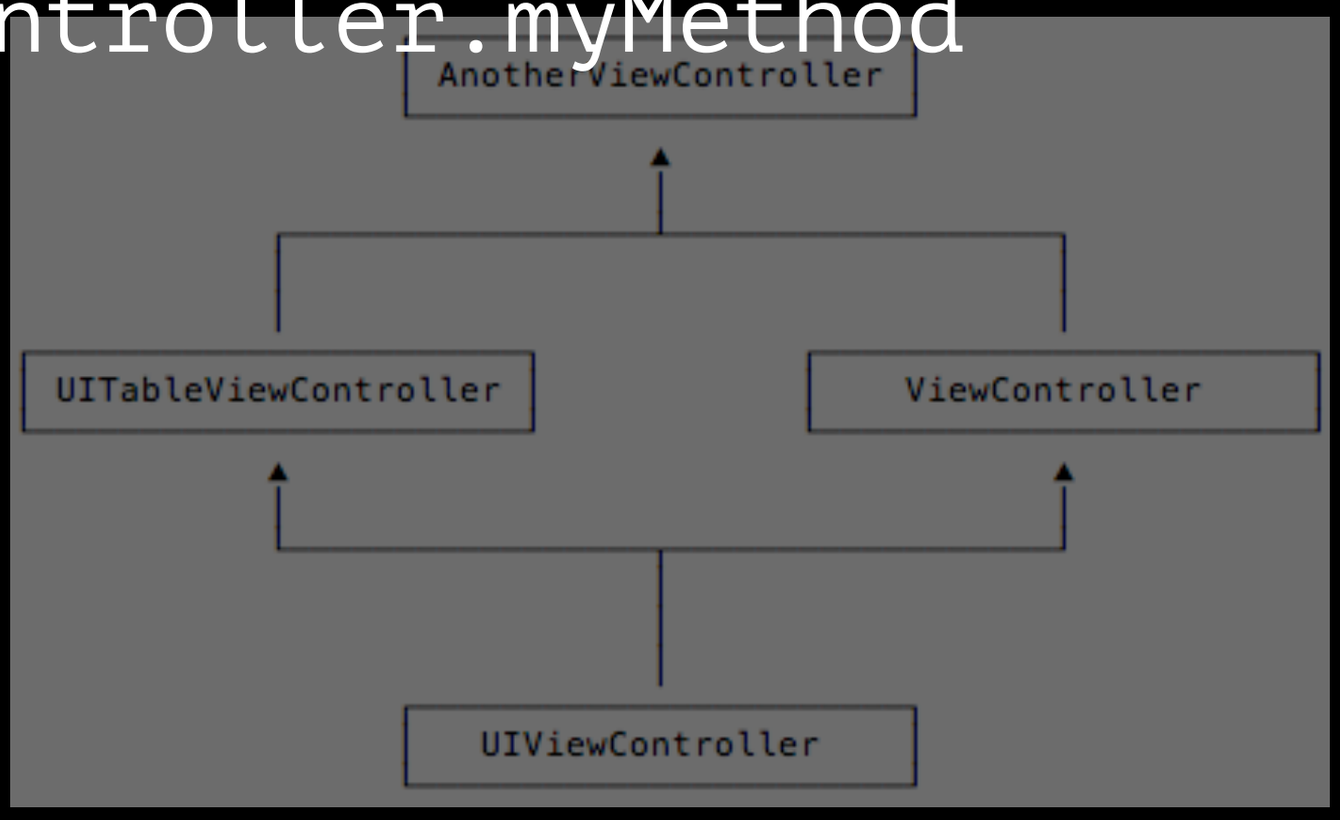
如果我们有多继承...


AnotherViewController
UITableViewController
ViewController
UIViewController

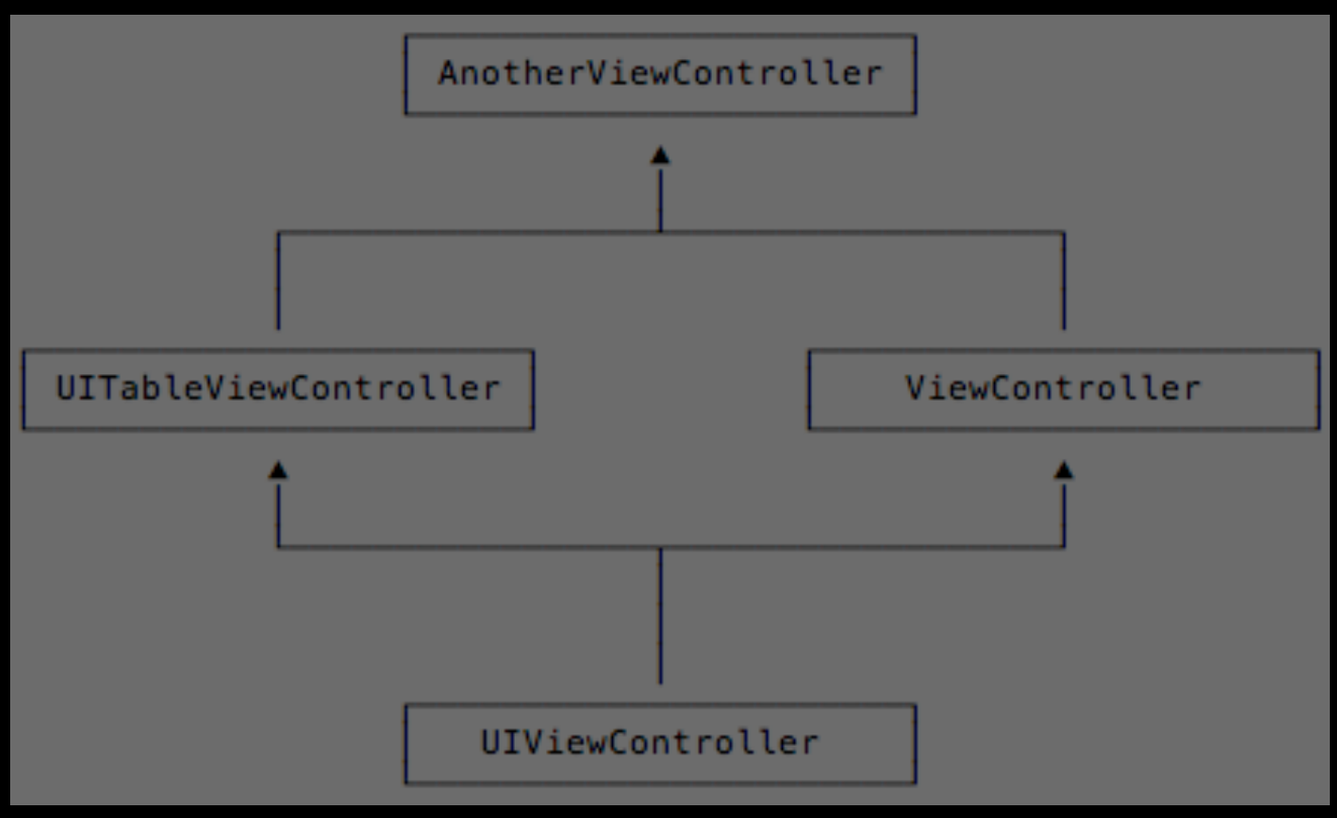
如果我们有多继承...

☑ UITableViewController.tableView

☑ ViewController.myMethod



但是两个父类都实现了的方法怎么办?



困境之二

Diamond Problem

菱形缺陷

像是 C++ 这样的语言选择粗暴地将菱形缺陷的问题交给程序员处理，复杂，人为错误的可能性。

Objective-C

Message Sending

另一个问题，虽然并不是面向对象的模式所特有的问题，但是却是 Objective-C 里常见的。Small Talk 带来的消息发送思想十分灵活，是 OC 的基础思想，但是有时候相对危险。

```
ViewController *v1 = ...
[v1 myMethod];

AnotherViewController *v2 = ...
[v2 myMethod];

NSArray *array = @[v1, v2];
for (id obj in array) {
    [obj myMethod];
}
```

```objc
NSObject *v3 = [NSObject new]
// v3 没有实现 `myMethod`

NSArray *array = @[v1, v2, v3];
for (id obj in array) {
    [obj myMethod];
}

// Runtime error:
// unrecognized selector sent to instance blabla
```

困境之三

Dynamic Dispatch Safety

动态派发安全性

OOP 困境

- 动态派发安全性
- 横切关注点
- 菱形缺陷

首先，在 OC 中动态派发让我们承担了在运行时才发现错误的风险，这很有可能是发生在上线产品中的错误。其次，横切关注点让我们难以对对象进行完美的建模，代码的重用也会更加糟糕。

协议

Protocol

Java, C#

Interface

Swift

protocol

Swift protocol

```
protocol Greetable {
    var name: String { get }
    func greet()
}
```

Swift protocol

```
protocol Greetable {
    var name: String { get }
    func greet()
}
struct Person: Greetable {
    let name: String
    func greet() {
        print("你好 \(name)")
    }
}
Person(name: "Wei Wang").greet()
```

我们接下来将看到协议的使用方式以及它是如何解决 OOP 所面临的几个困境的。

起・初识

- Objective-C 适合使用 OOP，但存在困境
- 协议定义了一组需要被实现的属性和方法

总结一下上面的内容。。。我们接下来将看到协议的使用方式以及它是如何解决 OOP 所面临的几个困境的。

承・相知

协议扩展和面向协议编程

OOP 困境

- 动态派发安全性
- 横切关注点
- 菱形缺陷

回顾 OOP 困境，协议是如何解决这些困境的。

```swift
protocol Greetable {
    var name: String { get }
    func greet()
}

struct Person: Greetable {
    let name: String
    func greet() {
        print("你好 \(name)")
    }
}
```

Person 实现了 Greetable 协议。其他类型也可以实现 Greetable

```swift
protocol Greetable {
    var name: String { get }
    func greet()
}

struct Cat: Greetable {
    let name: String
    func greet() {
        print("meow~ \(name)")
    }
}
```

```swift
let array: [Greetable] = [
        Person(name: "Wei Wang"),
        Cat(name: "onevcat")]
for obj in array {
    obj.greet()
}
// 你好 Wei Wang
// meow~ onevcat
```

```
struct Bug: Greetable {
    let name: String
}

// Compiler Error:
// 'Bug' does not conform to protocol 'Greetable'
// protocol requires function 'greet()'
```

对于没有实现 Greetbale 的类型，编译器错误。不存在消息误发送。

OOP 困境

- ✅ 动态派发安全性
- 横切关注点
- 菱形缺陷

如果你保持在 Swift 的世界里，那这个你的所有代码都是安全的

使用协议共享代码

```
protocol P {
    func myMethod()
}
```

```swift
// class ViewController: UIViewController
extension ViewController: P {
    func myMethod() {
        doWork()
    }
}

// class AnotherViewController: UITableViewController
extension AnotherViewController: P {
    func myMethod() {
        doWork()
    }
}
```

和 Copy Paste 没有区别?

Swift 2 - 协议扩展

协议本身并不是很强大，只是静态类型语言的编译器保证，在很多静态语言中也有类似的概念。那到底是什么让 Swift 成为了一门协议优先的语言？真正使协议发生质变，并让大家如此关注的原因，其实是在去年 WWDC 和 Swift 2 发布时，Apple 为协议引入了一个新特性，协议扩展，为 Swift 语言带来了一次革命性的变化。

```
protocol P {
    func myMethod()
}

extension P {
    func myMethod() {
        doWork()
    }
}
```

为 P 定义的方法提供默认实现

```
extension ViewController: P { }
extension AnotherViewController: P { }

viewController.myMethod()
anotherViewController.myMethod()
```

实现未在协议中声明的内容

```
protocol P {
    func myMethod()
}

extension P {
    func myMethod() {
        doWork()
    }

    func anotherMethod() {
        myMethod()
        someOtherWork()
    }
}

viewController.anotherMethod()
```

除了已经定义过的方法，我们甚至可以在扩展中添加没有定义过的方法。在这些额外的方法中，我们可以依赖协议定义的方法进行操作。我们之后会看到更多的例子。

- 协议定义
  - 提供实现的入口
  - 遵循协议的类型需要对其进行实现
- 协议扩展
  - 为入口提供默认实现
  - 根据入口提供额外实现

OOP 困境

- ✅ 动态派发安全性
- ✅ 横切关注点
- 菱形缺陷

简单，安全

多个协议中出现同名元素

```
protocol Nameable {
    var name: String { get }
}

protocol Identifiable {
    var name: String { get }
    var id: Int { get }
}
```

实现多个协议

```
struct Person: Nameable, Identifiable {
    let name: String
    let id: Int
}
    // `name` 属性同时满足 Nameable 和 Identifiable 的 name
```

实现多个协议

```
extension Nameable {
    var name: String { return "default name" }
}

struct Person: Nameable, Identifiable {
    // let name: String
    let id: Int
}

// Identifiable 也将使用 Nameable extension 中的 name
```

比较有意思，又有点让人困惑的是...

实现多个协议

```
extension Nameable {
    var name: String { return "default name" }
}

extension Identifiable {
    var name: String { return "another default name" }
}

struct Person: Nameable, Identifiable {
    // let name: String
    let id: Int
}

// 无法编译，name 属性冲突
```

实现多个协议

```
extension Nameable {
    var name: String { return "default name" }
}

extension Identifiable {
    var name: String { return "another default name" }
}

struct Person: Nameable, Identifiable {
    let name: String
    let id: Int
}

Person(name: "onevcat", id: 123).name // onevcat
```

OOP 困境

- ✅ 动态派发安全性
- ✅ 横切关注点
- ❓ 菱形缺陷

菱形缺陷没有被完全解决，当 protocol extension 中同时包含了签名一样的属性或者方法时，你需要重新进行实现，至少可以确定唯一性。不过不能很好地处理多个协议的冲突，确实也是 Swift 现在的不足。

转・热恋

在日常开发中使用协议

WWDC 15 #408

Protocol-Oriented Programming in Swift

比较理论化，通用思想。主要举了两个例子，画图表，排序。但我们每天做的事情是...如何把 POP 的思想运用到日常的开发?

Model (Networking)

案例

基于 Protocol 的网络请求

Demo

- 基于协议
- 类型安全
- 解耦合
- 可单独测试
- 扩展性

我们会构建一个网络模型层。发送请求，解析结果 ...

- Networking:
  - AFNetworking
  - Alamofire
  - (ASIHTTPRequest) 😇
- Model Parser
  - SwiftyJSON
  - Argo
  - Himotoki

优先考虑使用协议

高度协议化有助于解耦，测试以及扩展

如果你只能记住一句话

结合泛型发挥协议特性

避免动态调用和类型转换，保证安全性

APIKit[1] + Himotoki[2]

---

[1] https://github.com/ishkawa/APIKit

[2] https://github.com/ikesyo/Himotoki

View

协议和 UIKit，特别是 UI 如何协作?

案例

从 xib 创建 view

从 xib 创建 view

- View Class
  - `MyCell`
- 同名 view 的 xib 文件
  - `MyCell.xib`
- View 实例
  - `let myCell = MyCell.createFromNib()`

```swift
protocol NibCreatable {
    static func createFromNib(owner: Any?) -> Self?
}
```

```swift
extension NibCreatable {
    static func createFromNib(owner: Any?) -> Self? {
        guard let nibName = nibName else {
            return nil
        }
        let bundleContents = Bundle.main
                        .loadNibNamed(nibName, owner: owner, options: nil)
        guard let result = bundleContents?.last as? Self else {
            return nil
        }
        return result
    }

    static var nibName: String? {
        guard let n = NSStringFromClass(Self.self)
                        .components(separatedBy: ".").last else { return nil }
        return n
    }
}

// Cannot convert value of type 'Self.Type' to
// expected argument type 'AnyClass' (aka 'AnyObject.Type')
```

```swift
extension NibCreatable where Self: Any {
    static func createFromNib(owner: Any?) -> Self? {
        guard let nibName = nibName else {
            return nil
        }
        let bundleContents = Bundle.main
                        .loadNibNamed(nibName, owner: owner, options: nil)
        guard let result = bundleContents?.last as? Self else {
            return nil
        }
        return result
    }

    static var nibName: String? {
        guard let n = NSStringFromClass(Self.self)
                        .components(separatedBy: ".").last else { return nil }
        return n
    }
}
```

```swift
extension NibCreatable {
    static func createAndAdd(to superView: UIView) {



    }
}
```

```swift
extension NibCreatable where Self: UIView {
    static func createAndAdd(to superView: UIView) {
        if let view = createFromNib(owner: nil) {
            superView.addSubview(view)
        }
    }
}
```

```swift
extension NibCreatable where Self: UIView {
    static func createAndAdd(to superView: UIView) {
        if let view = createFromNib(owner: nil) {
            superView.addSubview(view)
            // view.backgroundColor = .clear
            // ...
        }
    }
}
```

```swift
final class MyCell: UIView {
    @IBOutlet weak var label: UILabel!
    //...
}

extension MyCell: NibCreatable {
    // 留空即可
}

let cell = MyCell.createFromNib()
view.addSubView(cell)

MyCell.createAndAdd(to: view)
```

对比 UIView extension

```
extension UIView {
    static func createFromNib(owner: Any?) -> UIView? {
        //...
    }

    static func createAndAdd(to superView: UIView) {
        //...
    }
}
```

UIView extension 需要类型转换；被加到了所有 UIView 中，并不是所有的都可以通过 nib 生成；只能针对 view 子类，nib 远不止于此。

使用 where 语句控制扩展范围

只对必要的类型进行协议扩展

Controller

案例：Overlay 页面

错误页面

空 tableview

方案一：使用代码进行 UI 构建

```
class OverlayView: UIView {
    //...
}

let overlay = OverlayView(frame: f)
view.addSubview(overlay)
```

方案一：使用代码进行 UI 构建

- 纯代码开发速度慢
- 任何 ViewController 都能不加限制地调用

方案二：通过 xib 进行加载

```
let overlay = NSBundle.mainBundle()
    .loadNibNamed("my_overlay_view", owner: nil, options: nil)
    .last as? UIView
view.addSubview(overlay)
```

方案二：通过 xib 进行加载

- nib 名字和类型安全
- ViewController 中额外的状态管理

方案三：协议

```
protocol OverlayPresentable {
    associatedtype Overlay: UIView
    var overlay: Overlay { get }
}

extension OverlayPresentable
                    where Self: UIViewController {
    func showOverlay() {
        view.addSubview(overlay)
    }

    func removeOverlay() {
        overlay.removeFromSuperview()
    }
}
```

```swift
extension ViewController: OverlayPresentable {}
class ViewController: UIViewController {
    let overlay = MyView.createFromNib(owner: nil)!

    func someMethod() {
        //...
        showOverlay()
        //...
        removeOverlay()
    }
}
```

方案三：协议

- 类型安全
- 统一管理状态
- 作用域限定

案例

分页加载

分页加载的网络请求

```
struct Pagination<T> {
    let items: [T]
    let hasNext: Bool
}
```

```
struct ChannelsResquest: Request {
    typealias Response = Pagination<Channel>
    let lastId: Int?
}
```

```
class ChannelsTableViewController: UITableViewController {
    private var lastId: Int? = nil
    private var hasNext = true

    override func viewDidLoad() {
        super.viewDidLoad()
        load()
    }
    func load() {
        if hasNext {
            client.send(ChannelsResquest(lastId: lastId)) {
                result in
            }
        }
    }
}
```

```swift
class ChannelsTableViewController: UITableViewController {
    private var lastId: Int? = nil
    private var hasNext = true

    private var data: [Channel] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        load()
    }

    func load() {
        if hasNext {
            client.send(ChannelsResquest(lastId: lastId)) {
                result in
                self.lastId = result!.items.last?.id
                self.hasNext = result!.hasNext
                self.data = result.items
                self.tableView.reloadData()
            }
        }
    }
}
```

```
extension ChannelsTableViewController: UITableViewDelegate {
    override func tableView(tableView: UITableView,
                   willDisplayCell cell: UITableViewCell,
             forRowAtIndexPath indexPath: NSIndexPath)
{
    if indexPath.row == data.count - 1 {
        load()
    }
}
```

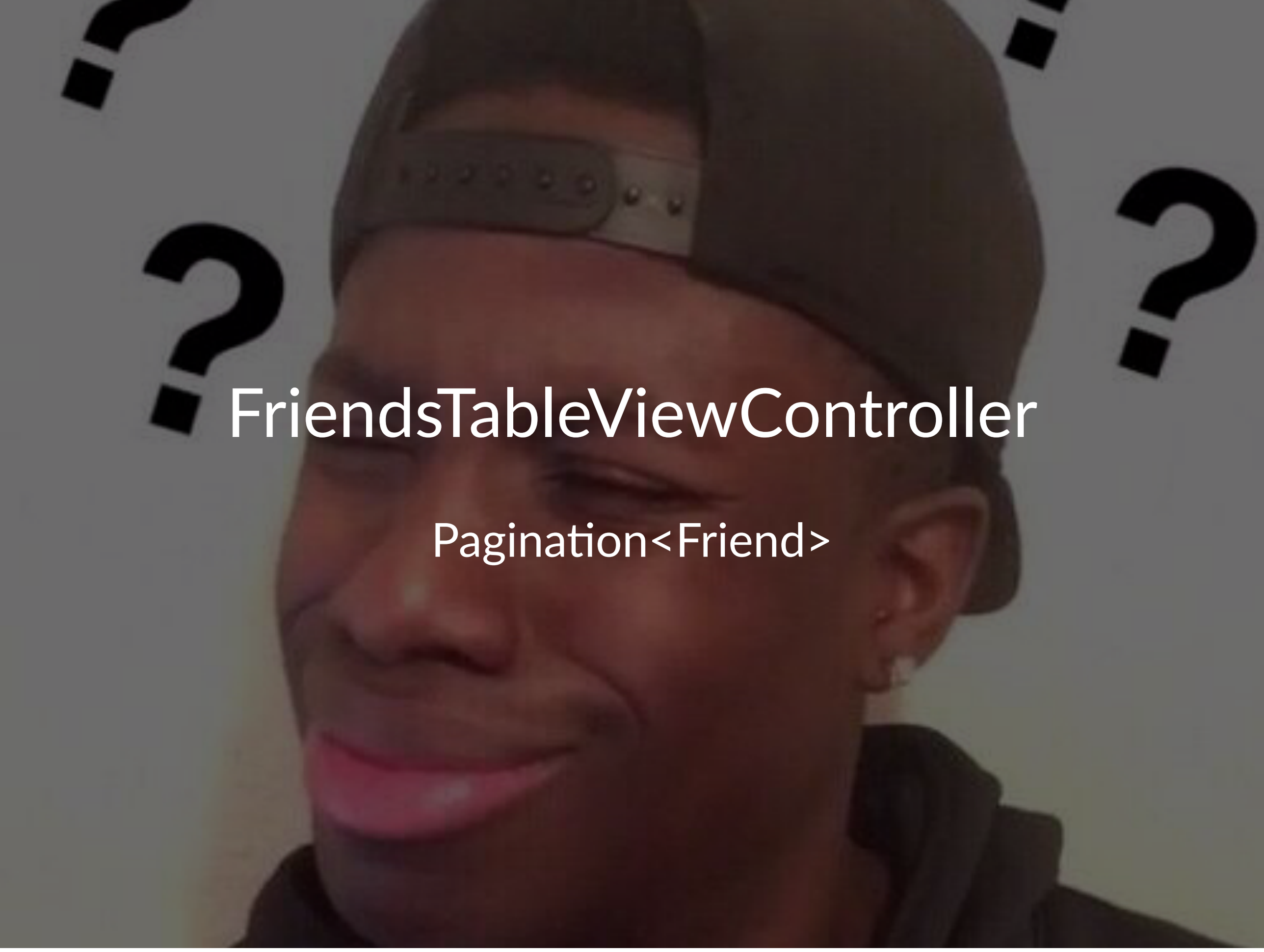
故事还没有结束...

故事还没有结束...

```
private var isLoading = false

func load() {
      if isLoading { return }
      if hasNext {
          isLoading = true
          client.send(ChannelsResquest(lastId: lastId)) {
              result in
              //...
              self.isLoading = false
          }
      }
  }
```

ChannelsTableViewController

Pagination<Channel>


FriendsTableViewController
Pagination<Friend>

方案一：复制粘贴

珍藏本

汉译世界学术名著丛书

自杀论

〔法〕埃米尔·迪尔凯姆 著

方案一：复制粘贴

商務印書館
The Commercial Press

方案二：父类

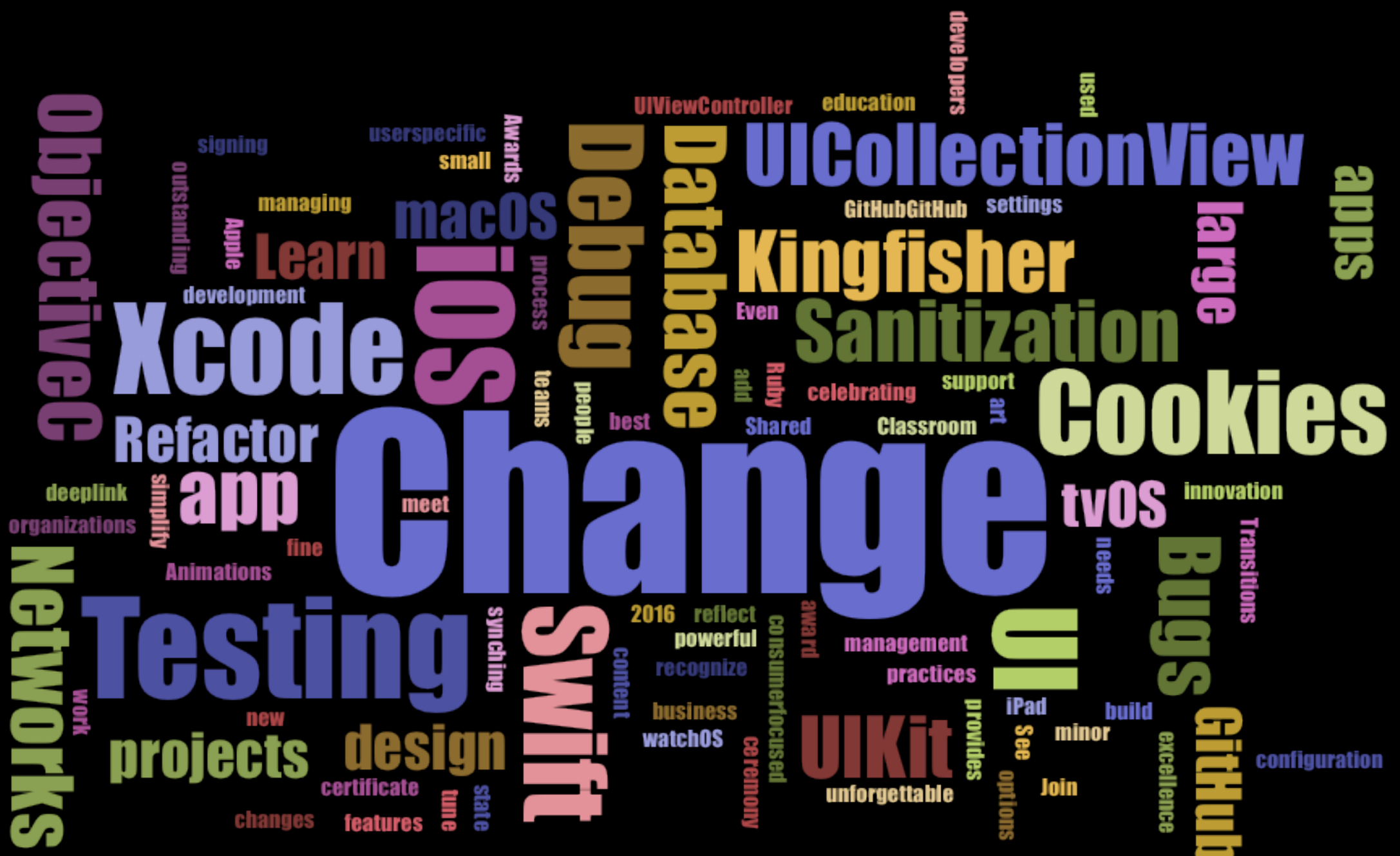
BaseTableViewController

```
class BaseTableViewController: UITableViewController {
    var lastId: Int? = nil
    var hasNext = true
    var isLoading = false

    func loadNext() {
        if isLoading { return }
        if hasNext {
            isLoading = true
            doLoad {result in
                self.lastId = //...
                self.hasNext = //...
            }
        }
    }

    func doLoad(handler: (Any?)->Void) {
        // ??
    }
}
```

典型的面向对象的思想

```
class BaseTableViewController: UITableViewController {
    var lastId: Int? = nil
    var hasNext = true
    var isLoading = false

    func loadNext() {
        if isLoading { return }
        if hasNext {
            isLoading = true
            doLoad {result in
                self.lastId = //...
                self.hasNext = //...
            }
        }
    }

    func doLoad(handler: (Any?)->Void) {
        fatalError("You should implement it in subclass!")
    }
}
```

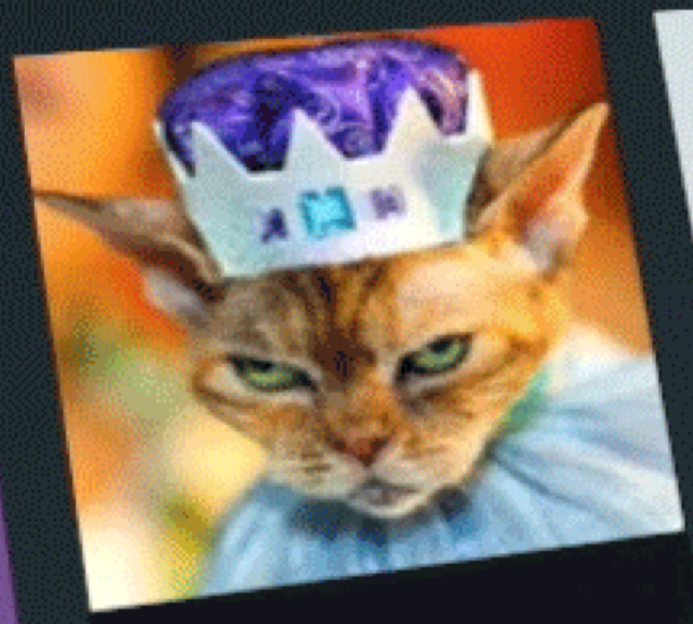
```
class FriendsTableViewController: BaseTableViewController {
    private var data: [Friend] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        loadNext()
    }

    override func doLoad(handler: (Any?)->Void) {
        client.send(FriendsRequest(lastId: lastId)) {
            result in
            handler(result)
            // ...
            self.data = result!.items
            self.tableView.reloadData()
        }
    }
}
```


Objective-C
Xcode
iOS
Debug
Database
UICollectionView
developers
used
education
UIViewController
userspecific
signing
Awards
small
outstanding
managing
Apple
macOS
Learn
development
process
Kingfisher
GitHubGitHub
settings
large
apps
Sanitization
Even
Ruby
add
celebrating
support
art
Cookies
teams
people
best
Shared
Classroom
Refactor
app
Change
deeplink
organizations
simplify
meet
fine
Animations
tvOS
innovation
Transitions
needs
Bugs
Networks
Testing
Swift
2016
reflect
powerful
award
management
practices
UI
syncing
content
recognize
consumerfocused
business
watchOS
ceremony
UIKit
iPad
See
provides
options
unforgettable
Join
minor
build
excellence
GitHub
configuration
work
new
projects
design
certificate
changes
features
tune
state

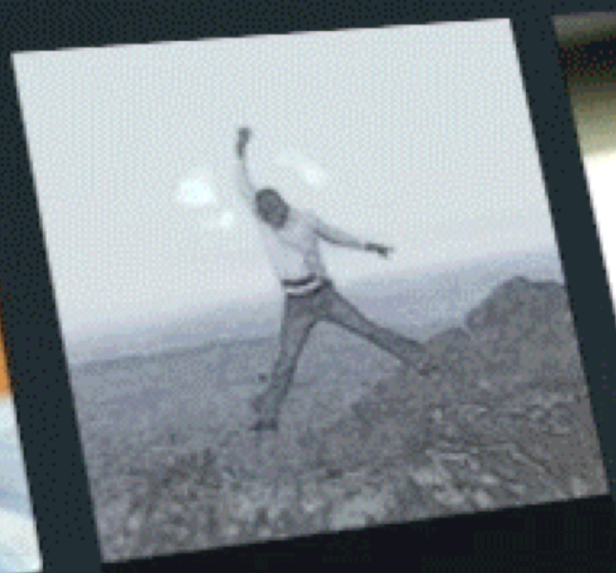
Yalantis 9:41 AM 100%

Friends

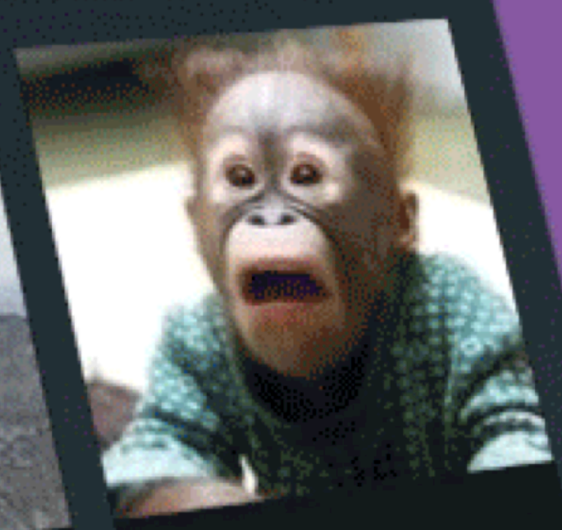
Search



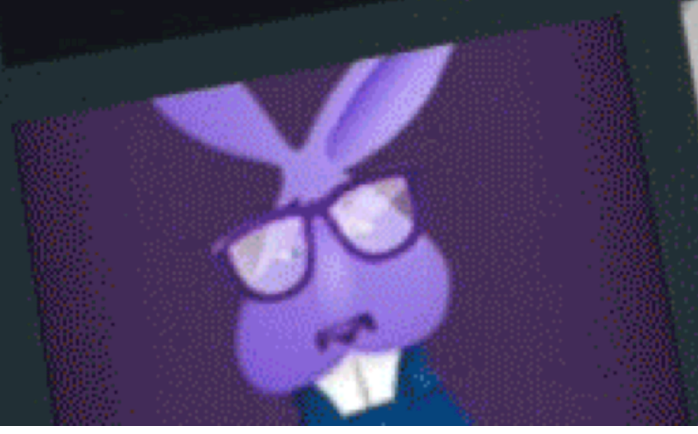
Grace Simpson

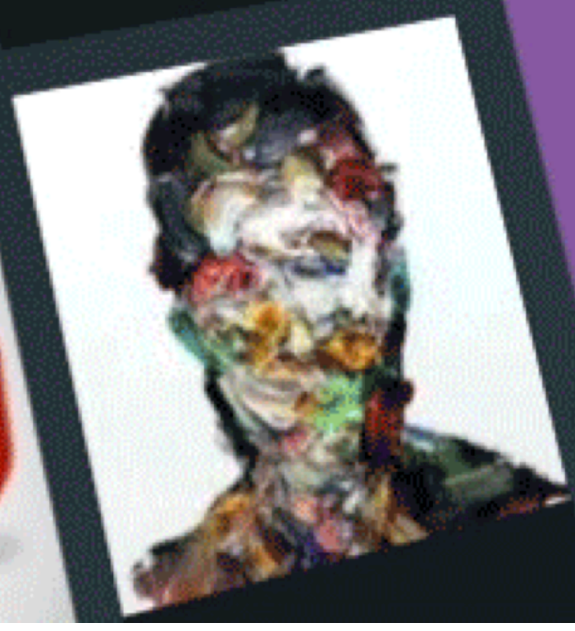


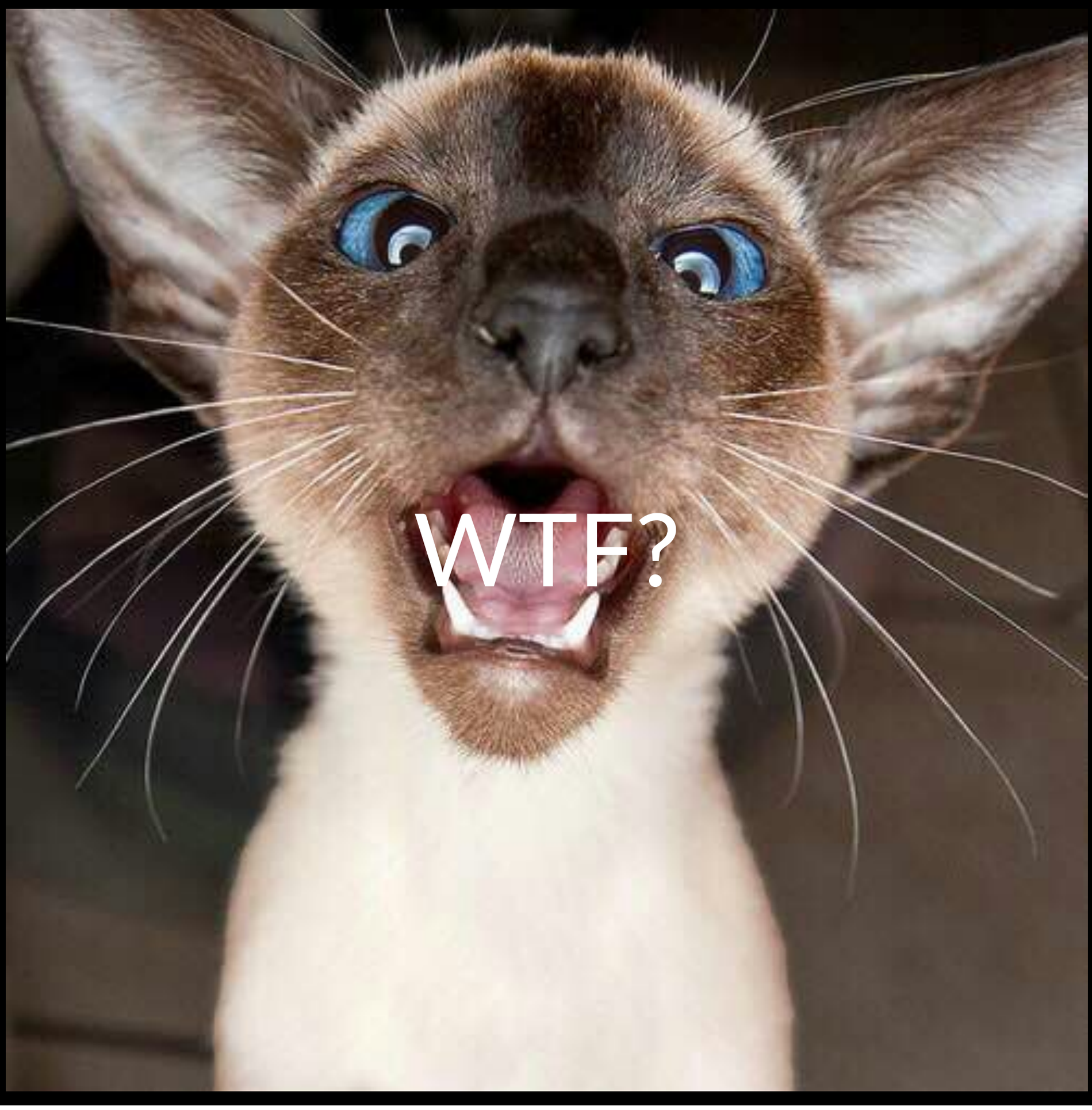
George Barrett



Joan Gonzales






WTF?

- FriendsTableViewController → FriendsCollectionViewController

- FriendsTableViewController → FriendsCollectionViewController
- BaseTableViewController → BaseCollectionViewController

- FriendsTableViewController → FriendsCollectionViewController
- BaseTableViewController → BaseCollectionViewController

复制粘贴?

- FriendsTableViewController → FriendsCollectionViewController
- BaseTableViewController → BaseCollectionViewController

方案三：协议

```swift
struct NextPageState<T> {
    private(set) var hasNext: Bool
    private(set) var isLoading: Bool
    private(set) var lastId: T?

    init() {
        hasNext = true
        isLoading = false
        lastId = nil
    }

    mutating func reset() {
        hasNext = true
        isLoading = false
        lastId = nil
    }

    mutating func update(hasNext: Bool, isLoading: Bool, lastId: T?) {
        self.hasNext = hasNext
        self.isLoading = isLoading
        self.lastId = lastId
    }
}
```

```
protocol NextPageLoadable: class {
    associatedtype DataType
    associatedtype LastIdType

    var data: [DataType] { get set }
    var nextPageState: NextPageState<LastIdType> { get set }
    func performLoad(
        successHandler: (_ rows: [DataType],
                         _ hasNext: Bool,
                         _ lastId: LastIdType?) -> (),
        failHandler: () -> ()
    )
}
```

```
protocol NextPageLoadable: class {
    associatedtype DataType
    associatedtype LastIdType

    var data: [DataType] { get set }
    var nextPageState: NextPageState<LastIdType> { get set }
    func performLoad(
        successHandler: (_ rows: [DataType],
                         _ hasNext: Bool,
                         _ lastId: LastIdType?) -> (),
        failHandler: () -> ()
    )
}

extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        guard nextPageState.hasNext else { return }
        if nextPageState.isLoading { return }
```

```
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        guard nextPageState.hasNext else { return }
        if nextPageState.isLoading { return }

        nextPageState.isLoading = true
        performLoad(successHandler: { rows, hasNext, lastId in
            self.data += rows
            self.nextPageState.update(hasNext: hasNext,
                                      isLoading: false,
                                      lastId: lastId)
```

```
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        guard nextPageState.hasNext else { return }
        if nextPageState.isLoading { return }

        nextPageState.isLoading = true
        performLoad(successHandler: { rows, hasNext, lastId in
            self.data += rows
            self.nextPageState.update(hasNext: hasNext,
                                      isLoading: false,
                                      lastId: lastId)

            self.tableView.reloadData()

        }, failHandler: {
            //..
        })
    }
}
```

```
class FriendTableViewController: UITableViewController {
    var nextPageState = NextPageState<Int>()
    var data: [Friend] = []
}

extension FriendTableViewController: NextPageLoadable {
    func performLoad(
                successHandler: ([String], Bool, Int?) -> (),
                failHandler: () -> ())
{
        client.send(FriendsRequest()) {
                result in
                if let result = result {
                    successHandler(result.items,
                                   result.hasNext,
                                   result.items.last.id)
                } else {
                    failHandler()
                }
        }
    }
}
```

```
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() { ... }
}
```

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() { ... }
}

extension FriendTableViewController: NextPageLoadable { ... }
class FriendTableViewController: UITableViewController {
    //...
    override func viewDidLoad() {
        super.viewDidLoad()
        loadNext()
    }

    override func tableView(tableView: UITableView,
                  willDisplayCell cell: UITableViewCell,
            forRowAtIndexPath indexPath: NSIndexPath)
    {
        if indexPath.row == data.count - 1 {
            loadNext()
        }
    }
}
```

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() { ... }
}
```

UICollectionView 怎么办?

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() { ... }
}
```

UICollectionView 怎么办?

复制粘贴?

```swift
extension NextPageLoadable where Self: UICollectionViewController {
    func loadNext() { ... }
}
```

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        guard nextPageState.hasNext else { return }
        if nextPageState.isLoading { return }

        nextPageState.isLoading = true
        performLoad(successHandler: { rows, hasNext, lastId in
            self.data += rows
            self.nextPageState.update(hasNext: hasNext, isLoading: false, lastId: lastId)

            self.tableView.reloadData()

            }, failHandler: {
                // Failed when first loading
                if self.nextPageState.lastId == nil {
                    self.data = []
                    self.nextPageState.reset()
                }
        })
    }
}
```

使用 Self: UITableViewController 的意义，只有 tableView.reloadData

```
tableView.reloadData()
colletionView.reloadData()
```

```
tableView.reloadData()
colletionView.reloadData()

protocol ReloadableType {
    func reloadData()
}

extension UITableView: ReloadableType {}
extension UICollectionView: ReloadableType {}
```

```
extension NextPageLoadable
    where Self: UITableViewController {
    func loadNext() {

        //...

        self.tableView.reloadData()

            //...
    }
}
```

```
extension NextPageLoadable {

    func loadNext(view: ReloadableType) {

        //...

        view.reloadData()

        //...
    }
}
```

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        loadNext(reloadView: tableView)
    }
}
```

```swift
extension NextPageLoadable where Self: UITableViewController {
    func loadNext() {
        loadNext(reloadView: tableView)
    }
}

extension NextPageLoadable where Self: UICollectionViewController {
    func loadNext() {
        loadNext(reloadView: collectionView)
    }
}
```

- FriendsTableViewController → FriendsCollectionViewController

```
class FriendTableViewController: UITableViewController {
    var nextPageState = NextPageState<Int>()
    var data: [Friend] = []
}

extension FriendTableViewController: NextPageLoadable {
    func performLoad(
                successHandler: ([String], Bool, Int?) -> (),
                failHandler: () -> ())
{
        client.send(FriendsRequest()) {
                result in
                if let result = result {
                    successHandler(result.items,
                                   result.hasNext,
                                   result.items.last.id)
                } else {
                    failHandler()
                }
        }
    }
}
```

```
class FriendCollectionViewController: UITableViewController {
    var nextPageState = NextPageState<Int>()
    var data: [Friend] = []
}

extension FriendCollectionViewController: NextPageLoadable {
    func performLoad(
                successHandler: ([String], Bool, Int?) -> (),
                failHandler: () -> ())
{
        client.send(FriendsRequest()) {
                result in
                if let result = result {
                    successHandler(result.items,
                                   result.hasNext,
                                   result.items.last.id)
                } else {
                    failHandler()
                }
        }
    }
}
```

ViewController 测试

NextPageState, performLoad 等方法都进行了抽象，单独测试。结合前面的 Request 的方法，对 performLoad 进行依赖注入测试非常简单

使用协议来组织 ViewController

- 保持简单的 ViewController 继承 (减少继承)

使用协议来组织 ViewController

- 保持简单的 ViewController 继承 (减少继承)
- 使用协议来「拼装」ViewController 所需要的功能

使用协议来组织 ViewController

- 保持简单的 ViewController 继承 (减少继承)
- 使用协议来「拼装」ViewController 所需要的功能
- 将代码和责任分离出 ViewController

使用协议来组织 ViewController

- 保持简单的 ViewController 继承 (减少继承)
- 使用协议来「拼装」ViewController 所需要的功能
- 将代码和责任分离出 ViewController
- 使用重构的方法将协议逐渐通用化

合・陪伴

使用协议帮助改善代码设计

通过面向协议的编程，我们可以从传统的继承上解放出来，用一种更灵活的方式，搭积木一样对程序进行组装。每个协议专注于自己的功能，特别得益于协议扩展，我们可以减少类和继承带来的共享状态的风险，让代码更加清晰。

推荐资料

- Protocol-Oriented Programming in Swift - WWDC 15 #408
- Protocols with Associated Types - @alexisgallagher
- Protocol Oriented Programming in the Real World - @_matthewpalmer
- Practical Protocol-Oriented-Programming - @natashatherobot

谢谢聆听

FAQ

Email: onev@onevcat.com, GitHub: onevcat