### WisdomProtocol

1. **简介**

**WisdomProtocol**：一款iOS 面向协议 编程框架，Swift特别版。在开发中，以遵守实现 对应协议，即可得到 相应能力 的理念，来定义一批定制的协议，通过他们绑定实现定制的功能和需求。

**协议功能** 支持 如下列表。

1. **协议功能支持**

1). 定时器功能

2). 模型数据: 编/解码

3). 程序信息跟踪

4). 图片: 加载/缓存

5). 多语言切换

6). 网络连接状态变化

7). 7大路由器

1. **功能详情**
2. **定时器功能**

-> 定时协议

**@objc** **public** **protocol** WisdomTimerable

简介：

继承定时协议 对象，拥有开启计时的能力，且不用管理内部定时器的生命周期。下面具体介绍协议的几个方法API。

-> 协议实现

// **MARK: Class Timerable Protocol**

// \* Support for objective-c/Swift Class

**@objc** **public** **protocol** WisdomTimerable {

    // **MARK: Class Param - UInt, WisdomTimerable**

    // \* Timer task in progress, current time

**@objc** **func** timerable(timerDid currentTime: UInt, timerable: WisdomTimerable)

    // **MARK: Class Param - WisdomTimerable**

    // \* Example End a scheduled task

**@objc** **func** timerable(timerEnd timerable: WisdomTimerable)

}

-> 支持 Swift / OC 类遵守，且协议简单明了

-> 支持 每秒任务状态更新回调 和 定时任务结束状态回调

**extension** WisdomTimerable {

    // **MARK: Class Param - NSInteger. < No need to implement >**

    // \* Start a forward timer task, start the forward time point

**public** **func** startForwardTimer(startTime: UInt)

    // **MARK: Class Param - NSInteger. < No need to implement >**

    // \* Start a countdown timer task, start the total time countdown

**public** **func** startDownTimer(totalTime: UInt)

    // **MARK: Class Timer - destroy. < No need to implement >**

    // Destruction/Release timer task

**public** **func** destroyTimer()

}

-> 支持 从指定时间开始累积计时 和 总时间的倒计时

-> 支持 主动摧毁运行中的定时任务

-> 支持功能：

**public** **func** startForwardTimer(startTime: UInt)

\* 累积计时

**public** **func** startDownTimer(totalTime: UInt)

\* 倒计时

【优势/特点】

-> 开发者无需关心，定时任务的创建和销毁。对于销毁，内部会即时销毁，包括任务结束即时销毁，包括启动任务对象销毁，也会即时销毁定时任务；

-> 定时任务过程中，app前后台状态切换，会对定时数产生的影响，已妥善计算处理，放心使用；

1. **模型数据 编/解码**

-> 编/解码协议

// **MARK: Swift Class/NSObject/Value to coding/decoding Protocol**

**public** **protocol** WisdomCodingable {}

简介：

继承编/解码协议 对象，拥有对模型和数据编/解吗的能力，且支持集合，结构体，枚举类型。下面具体介绍协议的几个方法API。

-> 几种解码场景：

**extension** WisdomCodingable **where** **Self**: Decodable {

    // **MARK: Param - [String: Any], return - Self?**

    // swift dictionary to dictionary model, use 'Decodable' protocol

**public** **static** **func** decodable(value: [String: **Any**])->**Self**?

    // **MARK: Param - [String: Any], return - [Self]**

    // swift dictionary list to dictionary model list, use 'Decodable' protocol

**public** **static** **func** decodable(list: [[String: **Any**]])->[**Self**]

    // **MARK: Param - String, return - Self?**

    // swift json string to model, use 'Decodable' protocol

**public** **static** **func** jsonable(json: String)->**Self**?

    // **MARK: Param - String, return - [Self]**

    // swift jsons string to model list, use 'Decodable' protocol

**public** **static** **func** jsonable(jsons: String)->[**Self**]

}

解码 上到下顺序：1. 字典 转 模型

2. 字典数组 转 模型数组

3. Json 转 模型

4. Json 转 模型数组

-> 几种编码场景：

**extension** WisdomCodingable **where** **Self**: Encodable {

    // **MARK: return - String?**

    // swift model to json string, use 'Encodable' protocol

**public** **func** ableJson()->String?

    // **MARK: return - [String:Any]?**

    // swift model to dictionary, use 'Encodable' protocol

**public** **func** ableEncod()->[String:**Any**]?

}

-> 集合编码场景：

**public** **extension** Array **where** Element: WisdomCodingable&Encodable {

    // **MARK: return - [[String:Any]]**

    // swift model list to dictionary list, use 'Encodable' protocol

**func** ableEncod()->[[String:**Any**]]

    // **MARK: return - String?**

    // swift model list to jsons string, use 'Encodable' protocol

**func** ableJsons()->String?

}

编码 上到下顺序：1. 模型 转 Json

2. 模型 转 字典

3. 模型数组 转 字典数组

4. 模型数组 转Json

【优势/特点】

-> 只支持 Swift 类，枚举，struct **的** 编/解码；

-> 内部 编/解码 实现，使用的 Swift原生Coding协议，所以不需要担心稳定性和兼容性；

-> 解析流程中添加了断言处理：

assert(able != **nil**, "decodable failure: \(value)")

调试环境，断言 便于即时发现不合格数据，即时检测；

如果不需要，可以注释此处断言代码；

1. **程序信息跟踪**

**目前支持跟踪功能：**

**-> 1.** 崩溃信息的跟踪**；**

**-> 2.** 控制器的显示/掩藏状态跟踪，和显示时间统计**；**

**-> 3. 协议限制** 条件对象：**UIApplicationDelegate**

**where** **Self**: *UIApplicationDelegate*

**1).** 崩溃跟踪协议

**@objc** **public** **protocol** WisdomCrashingable **where** **Self**: UIApplicationDelegate {

    // **MARK: Catch Crashing Param - String**

    // Swift object type, this parameter is valid in the relase environment but invalid in the debug environment

    // objective-c object type, both debug and relase environments are supported

**@objc** **func** catchCrashing(crash: String)

}

说明：崩溃跟踪，同时支持OC和Swift 语言崩溃场景抓取。

**2).** 控制器展示跟踪协议

**@objc** **public** **protocol** WisdomTrackingable **where** **Self**: UIApplicationDelegate {

    // **MARK: Catch Controller Tracking Param - String, String**

    // UIViewController Catch Tracking 'viewDidAppear'

    // - controller: UIViewController.Type

    // - title: String

**@objc** **func** catchTracking(viewDidAppear controller: UIViewController.**Type**, title: String)

    // **MARK: Catch Controller Tracking Param - String, String**

    // UIViewController Catch Tracking 'viewDidDisappear'

    // - controller: UIViewController.Type

    // - appearTime: NSInteger

    // - title: String

**@objc** **optional** **func** catchTracking(viewDidDisappear controller: UIViewController.**Type**, appearTime: NSInteger, title: String)

}

说明：控制器将要隐藏协议回调中，会带过来时间参数，为当前控制器所展示的时长。

**3). 协议**使用案例

**extension** AppDelegate: WisdomCrashingable {

**func** catchCrashing(crash: String) {

        RCLog.error(text: "[crash]: \(crash)")

    }

}

**extension** AppDelegate: WisdomTrackingable {

    //页面跟踪

**func** catchTracking(viewDidAppear controller: UIViewController.**Type**, title: String) {

        RCLog.info(text: "vc: \(controller) title: "+title)

    }

}

**上面案例为：崩溃消息监听；**

**下面案例为：控制器展示状态监听，这里只实现了展示协议；**

1. **图片 缓存/加载**

**1).** UIImage 本地缓存 和 本地加载 扩展：

// \* Save/Load Image in Memory/Disk Cache \*

**extension** UIImage {

    // **MARK: Save Image in Memory Cache / Disk Cache**

    // imageName: Image Name

**@objc** **public** **func** saveingable(imageName: String)

    // **MARK: Load Image in Memory Cache / Disk Cache**

    // imageName:    Image Name

    // imageClosure: (UIImage,String)->() (has UIImage)

    // emptyClosure: ()->()               (no UIImage)

**@objc** **public** **static** **func** loadingable(imageName: String, imageClosure: **@escaping** (UIImage,String)->(), emptyClosure: **@escaping** ()->())

}

--> UIImage 本地缓存 案例：

UIImage().saveingable(imageName: “本地缓存图片名称”)

--> UIImage 本地加载 案例：

UIImage.loadingable(imageName: “本地缓存图片名称” ) { image, imageName **in**

  // 获取到本地缓存到图片

} emptyClosure: {

  // 本地没有缓存的图片

}

说明：

1. 图片本地缓存为沙盒缓存，如果app内存吃紧，缓存图片会被清除。
2. 图片获取缓存过程：内存缓存 -> 磁盘缓存。
3. **.** UIImageView本地缓存 和 网络加载 扩展：

**extension** UIImageView {

    // **MARK: Load Image in Memory Cache / Disk Cache**

    // imageName:        Image Name (historical save Memory/Disk Cache)

    // placeholderImage: Placeholder picture (Memory/Disk Cache no Image)

**@objc** **public** **func** loadingImageable(imageName: String, placeholderImage: UIImage?=**nil**)

    // **MARK: Load Image in Network / Memory Cache / Disk Cache**

    // imageUrl:         Image Url (historical save Memory/Disk Cache, if not, network download)

    // placeholderImage: Placeholder picture (Memory/Disk Cache no Image)

**@objc** **public** **func** loadingImageable(imageUrl: String, placeholderImage: UIImage?=**nil**)

}

说明：

--> 第一种，根据 imageName 只在本地缓存中加载图片；

--> 第二种，先本地加载缓存图片，在没有的情况，继续根据 imageUrl 进行网络下载，下载完成做本地缓存；

1. **.** UIImageView图片动态跟踪 扩展：

**extension** UIImageView {

    // **MARK: Tracking save Image in Memory Cache / Disk Cache, Paired use**

    // Tracking UIImage ‘@objc public func saveingable(imageName: String)’ method, update icon

    // 当有图片本地缓存, 调用 UIImage ‘@objc public func saveingable(imageName: String)’ 方法时，对跟踪图片控件，刷新图片

**@objc** **public** **func** trackingImageable()

    // **MARK: Missing save Image in Memory Cache / Disk Cache, Paired use**

    // Make UIImage ‘@objc public func trackingImageable()’ method Missing

    // 使对跟踪图片控件刷新失效，失效设置 UIImage ‘@objc public func saveingable(imageName: String)’ 方法

**@objc** **public** **func** missingImageable()

}

说明：

--> ‘func trackingImageable()’: 对本地图片缓存任务，进行监听，并重新刷新之前UIImageView加载失败的图片；

--> ‘func missingImageable()’: 使 ‘func trackingImageable()’监听任务失效，不再刷新之前UIImageView加载失败的图片；

1. **多语言切换**

**1). 多语言支持种类 枚举**：

**@objc** **public** **enum** WisdomLanguageStatus: Int

-> 具体类型：

    /// 跟随系统语言

**case** system=1

    /// 英语

**case** en

    /// 简体中文

**case** zh\_Hans

    /// 繁体中文

**case** zh\_Hant

    /// 繁体中文(香港特别行政区)

**case** zh\_Hant\_HK

    /// 繁体中文(中国台湾省)

**case** zh\_Hant\_TW

    /// 法语

**case** fr

    /// 德语

**case** de

    /// 意大利语

**case** it

    /// 日语

**case** ja

    /// 韩语

**case** ko

    /// 葡萄牙语

**case** pt\_PT

    /// 俄语

**case** ru

    /// 西班牙语

**case** es

    /// 荷兰语

**case** nl

    /// 泰语

**case** th

    /// 阿拉伯语

**case** ar

    /// 乌克兰

**case** uk

1. **. 多语言功能注册 协议**：

// **MARK: Language Registerable**

**@objc** **public** **protocol** WisdomLanguageRegisterable **where** **Self**: UIApplicationDelegate

说明：

--> 在使用多语言主协议功能前，需要实现 **多语言功能注册 协议，如果未实现多语言注册协议，**多语言主协议功能 不可用；

-->WisdomLanguageRegisterabl限制 UIApplicationDelegate 实现；

-> 具体功能注册：

// **MARK: Language Registerable**

**@objc** **public** **protocol** WisdomLanguageRegisterable **where** **Self**: UIApplicationDelegate {

    // **MARK: return - String?**

    // Get the 'String' local save language key

**@objc** **func** registerLanguageKey()->String?

    // **MARK: Param - WisdomLanguageStatus, return - Bundle**

    // Get the ‘Bundle’ based on the type

**@objc** **func** registerLanguage(language: WisdomLanguageStatus)->Bundle

    // **MARK: Param - WisdomLanguageStatus**

    // Current Language Update

**@objc** **func** registerLanguageUpdate(language: WisdomLanguageStatus)

}

说明：

--> registerLanguageKey():本地保存语言设置类型的key，每次保存/获取本地设置，会调用。设置nil 不做本地缓存;

--> registerLanguage(language: WisdomLanguageStatus)->Bundle:根据WisdomLanguageStatus**获取多语言资源**Bundle;

--> registerLanguageUpdate(language: WisdomLanguageStatus):**更新当前设置的语言类型** WisdomLanguageStatus 时调用;

**3). 多语言功能主 协议**：

**@objc** **public** **protocol** WisdomLanguageable

-> 具体功能：

**extension** WisdomLanguageable {

    // **MARK: return - WisdomLanguageStatus?**

    // Gets the language type of the setting

**public** **static** **func** getCurrentLanguage()->WisdomLanguageStatus?

    // **MARK: return - String**

    // Gets the language type of the System

**public** **static** **func** getSystemLanguage()->String

    // **MARK: Param - WisdomLanguageStatus, return - Bool**

    // Update Language

**@discardableResult**

**public** **static** **func** updateLanguage(language: WisdomLanguageStatus)->Bool

    // **MARK: Reset Language**

**public** **static** **func** resetLanguage()

}

说明：

--> getCurrentLanguage():**获取当前设置的语言类型** WisdomLanguageStatus，未设置 为nil;

--> getSystemLanguage():**获取当前系统的语言类型** WisdomLanguageStatus;

--> updateLanguage(language: WisdomLanguageStatus):**更新当前系统的语言类型** WisdomLanguageStatus;

--> resetLanguage():重置当**前系统的语言类型** WisdomLanguageStatus，设置为nil;

**4). 多语言协议注册案例**：

**extension** AppDelegate: WisdomLanguageRegisterable {

**func** registerLanguageKey()->String? {

**return** "Language"

    }

**func** registerLanguage(language: WisdomLanguageStatus)->Bundle {

**let** bundlePath = (Bundle.main.path(forResource: "RainbowStone", ofType: "bundle") ?? "")

**var** path = bundlePath+"/Lan/"+language.file\_lproj

**var** bundle: Bundle?

**switch** language {

**case** .zh\_Hans, .zh\_Hant, .zh\_Hant\_HK, .zh\_Hant\_TW:

            path = bundlePath+"/Lan/"+WisdomLanguageStatus.zh\_Hans.file\_lproj

            bundle = Bundle.init(path: path)

            MJRefreshConfig.**default**.languageCode = WisdomLanguageStatus.zh\_Hans.fileName

**default**:

            path = bundlePath+"/Lan/"+WisdomLanguageStatus.en.file\_lproj

            bundle = Bundle.init(path: path)

            MJRefreshConfig.**default**.languageCode = WisdomLanguageStatus.en.fileName

        }

**return** bundle ?? Bundle.main

    }

**func** registerLanguageUpdate(language: WisdomLanguageStatus) {

        NotificationCenter.default.post(name: NSNotification.Name(rawValue: "LanguageChangeNC") , object: **nil**)

    }

}

说明：

--> 案例只设置了 中/英 文切换；

--> 根据参数 WisdomLanguageStatus 类型，返回对应的Bundle类型；

**5). 多语言协议使用案例**：

**class** RCUpdate: WisdomLanguageable

说明：

--> 继承 WisdomLanguageable 协议的对象，即拥有了获取当前语言设置信息的功能，和设置切换当前语言的功能；

1. 网络连接状态监听

**1).** 网络状态 枚举：

**@objc** **public** **enum** WisdomNetworkReachabilityStatus: NSInteger

-> 具体状态：

/// It is unknown whether the network is reachable.

**case** unknown=0

    /// The network is not reachable.

**case** notReachable

    /// 蜂窝网络

**case** cellular

    /// 以太网/WiFi

**case** ethernetOrWiFi

**2).** 获取**当前**网络状态信息：

**extension** WisdomNetworkReachabilityStatus {

    // **MARK: get 'currentNetworkReachabilityState: WisdomNetworkReachabilityStatus'**

**public** **static** **var** currentNetworkReachabilityState: WisdomNetworkReachabilityStatus

    // **MARK: Whether the network is available network: 'cellular / ethernetOrWiFi'**

**public** **static** **var** isCurrentReachable: Bool

    // **MARK: Whether the current network is a cellular network: 'cellular'**

**public** **static** **var** isCurrentReachableOnCellular: Bool

    // **MARK: Whether the current network is Ethernet / WiFi network: 'ethernetOrWiFi'**

**public** **static** **var** isCurrentReachableOnEthernetOrWiFi: Bool

}

说明：

--> currentNetworkReachabilityState:获取**当前**网络状态；

--> isCurrentReachable: **当前网络状态是否有网（**蜂窝网络

**或者以太网或者Wi-Fi）**;

--> isCurrentReachableOnCellular:**当前网络状态是否是** 蜂窝网络;

--> isCurrentReachableOnEthernetOrWiFi:**当前网络状态是否是 以太网或者Wi-Fi**;

**3).** 网络状态主协议：

**@objc** **public** **protocol** WisdomNetworkReachabilityable {

    // **MARK: networkReachability 'WisdomNetworkReachabilityStatus' - didChange**

    // \* network reachability did change

**@objc** **func** networkReachability(didChange currentState: WisdomNetworkReachabilityStatus)

}

说明：

-> 需要监听网络状态变化，对象必须继承此协议，并实现协议方法;

-> 对象继承协议完成好，就可以去 开启/关闭 监听；

**4).** 网络状态监听 开启/关闭：

**extension** WisdomNetworkReachabilityable {

    // **MARK: self start 'WisdomNetworkReachabilityStatus' Listening. < No need to implement >**

**public** **func** startReachabilityListening()

    // **MARK: self stop 'WisdomNetworkReachabilityStatus' Listening. < No need to implement >**

**public** **func** stopReachabilityListening()

}

**5).** 网络状态监听 案例：

**class** RCMsgUser: WisdomNetworkReachabilityable

{

**init**(userId: String) {

**self**.userId = userId

// 开启网络监听

        startReachabilityListening()

}

**deinit** {

// 释放，停止网络监听

        stopReachabilityListening()

    }

// 网络状态变化回调

**func** networkReachability(didChange currentState: WisdomNetworkReachabilityStatus) {

    }

}

说明：

--> 网络状态监听，使用比较简单；

--> 内部实现，网络状态对象不会频繁创建，会复用；

## Routing Protocol/路由协议篇

## Routing protocol is the core function of WisdomProtocol. The following describes how to use it/路由协议是 WisdomProtocol 核心功能，以下为您介绍如何去使用

【\*\*协议支持\*\*】：跨工程/模块/动态库/静态库 中 类/UIView/UIViewController/参数 之间的交互/传递/调用功能。

【\*\*protocol support\*\*】: across the project/module/dynamic library/class/UIView UIViewController/parameters in static library interaction/transfer/call function.

【\*\*注册绑定\*\*】：路由协议在使用之前，需要实现 WisdomRegisterable 注册协议，将唯一的协议与类绑定，注册到 WisdomProtocol sdk 中，像这样：

【\*\*registering binding\*\*】：Before the routing protocol is used, it is necessary to implement the WisdomRegisterable registration protocol, bind the unique protocol with the class, and register it in the WisdomProtocol sdk as follows:

@objc protocol WisdomProtocolLeftVCProtocol {}

class WisdomProtocolLeftVC: UIViewController, WisdomRegisterable, WisdomProtocolLeftVCProtocol {

// Here the protocol is registered and bound to the class/这里协议与类进行注册绑定

static func registerable() -> WisdomClassable {

return WisdomClassable(register: WisdomProtocolLeftVCProtocol.self, conform: Self.self)

}

}

【\*\*路由功能\*\*】

【\*\*Routing Function\*\*】

【1】. Class Routing protocol/路由协议：

// MARK: - Router Class Protocol

@objc public protocol \*\*WisdomRouterClassable\*\* {

// MARK: Param - Any?

@discardableResult

@objc optional static func routerClassable(param: Any?)->Self

// MARK: Param - Any?, ((Any)->Void)?

@discardableResult

@objc optional static func routerClassable(param: Any?, closure: ((Any)->Void)?)->Self

// MARK: Param - Any?, ((Any)->(Any))?

@discardableResult

@objc optional static func routerClassable(param: Any?, returnClosure: ((Any)->(Any))?)->Self

}

---> \*\*Class Route Application Case/路由使用案例\*\*：

Registering a binding/注册绑定：

@objc protocol WisdomProtocolLeftModelable {}

class WisdomProtocolLeftModel: WisdomRegisterable, WisdomProtocolLeftModelable {

// Register the binding protocol/class/注册绑定协议/类

static func registerable() -> WisdomClassable {

return WisdomClassable(register: WisdomProtocolLeftModelable.self, conform: Self.self)

}

var bgColor: String?

var textColor: String?

var codeColor: String?

}

Implementation of Protocol/协议实现：

extension WisdomProtocolLeftModel: WisdomRouterClassable{

// Implementing Routing Protocols/实现路由协议

static func routerClassable(param: Any?) -> Self {

var dict: [String:Any] = [:]

if let value = param as? [String:Any] {

dict = value

}

let cla = WisdomProtocolLeftModel.decodable(value: dict)

return cla as! Self

}

}

---> \*\*External start routing/外部开始路由\*\*：

// Get the category by registering with the protocol/通过注册到协议，获取到类别

let cla = WisdomProtocol.getRouterClassable(from: WisdomProtocolLeftModelable.self)

let param = ["bgColor":"708069","textColor":"FFFFFF","codeColor":"33A1C9"]

// Creating an Object/创建对象

let objc = cla?.routerClassable?(param: param)

print(objc)

【2】. UIViewController Routing protocol/路由协议：

// MARK: - Router UIViewController Protocol

@objc public protocol WisdomRouterControlable where Self: UIViewController {

// MARK: Param - UIViewController?, Any?

@discardableResult

@objc optional static func routerControlable(rootVC: UIViewController?, param: Any?)->Self

// MARK: Param - UIViewController?, Any?, ((Any)->Void)?

@discardableResult

@objc optional static func routerControlable(rootVC: UIViewController?, param: Any?, closure: ((Any)->Void)?)->Self

// MARK: Param - UIViewController?, Any?, ((Any)->(Any))?

@discardableResult

@objc optional static func routerControlable(rootVC: UIViewController?, param: Any?, returnClosure: ((Any)->(Any))?)->Self

}

--> \*\*UIViewController Route Application Case/路由使用案例\*\*：

Registering a binding/注册绑定：

@objc protocol WisdomProtocolLeftVCProtocol {}

class WisdomProtocolLeftVC: UIViewController, WisdomRegisterable, WisdomProtocolLeftVCProtocol {

// Registering a binding protocol/注册绑定协议/UIViewController

static func registerable() -> WisdomClassable {

return WisdomClassable(register: WisdomProtocolLeftVCProtocol.self, conform: Self.self)

}

}

Implementation of Protocol/协议实现：

extension WisdomProtocolLeftVC: WisdomRouterControlable {

// 实现路由协议

@discardableResult

static func routerControlable(rootVC: UIViewController?, param: Any?) -> Self {

let vc = Self.init()

vc.modalPresentationStyle = .fullScreen

rootVC?.navigationController?.pushViewController(vc, animated: true)

return vc

}

}

--> External start routing/外部开始路由:

// Get the UIViewController by registering with the protocol/通过注册到协议，获取到UIViewController

// MARK: WisdomRouterControlable Route controller - None/WisdomRouterControlable 路由控制器-无参数

let vcClass = WisdomProtocol.getRouterControlable(from: LeftVCProtocol.self)

// Creating an Object/创建对象

\_=vcClass?.routerControlable?(rootVC: self, param: nil)

【3】. UIView Routing protocol/路由协议：

// MARK: - Router UIView Protocol

@objc public protocol WisdomRouterViewable where Self: UIView {

// MARK: Param - UIView?, Any?

@discardableResult

@objc optional static func routerViewable(superview: UIView?, param: Any?)->Self

// MARK: Param - UIView?, Any?, ((Any)->Void)?

@discardableResult

@objc optional static func routerViewable(superview: UIView?, param: Any?, closure: ((Any)->Void)?)->Self

// MARK: Param - UIView?, Any?, ((Any)->(Any))?

@discardableResult

@objc optional static func routerViewable(superview: UIView?, param: Any?, returnClosure: ((Any)->(Any))?)->Self

}

--> \*\*UIView Route Application Case/路由使用案例\*\*：

Registration Agreement/注册协议：

@objc protocol WisdomProtocolLeftVIProtocol {}

class WisdomProtocolLeftVI: UIView, WisdomRegisterable, WisdomProtocolLeftVIProtocol {

// Registering a binding protocol/注册绑定协议/UIView

static func registerable() -> WisdomClassable {

return WisdomClassable(register: WisdomProtocolLeftVIProtocol.self, conform: Self.self)

}

}

Implementation protocol/实现协议：

extension WisdomProtocolLeftVI: WisdomRouterViewable {

// Implementing Routing Protocols/实现路由协议

static func routerViewable(superview: UIView?, param: Any?) -> Self {

let vi = Self.init()

// Add/layout view/添加/布局视图

if let supervi = superview {

supervi.addSubview(vi)

vi.snp.makeConstraints { make in

make.top.equalTo(supervi).offset(10)

make.bottom.equalTo(supervi).offset(-20)

make.left.equalTo(supervi).offset(30)

make.right.equalTo(supervi).offset(-30)

}

}

return vi

}

}

--> External start routing/外部开始路由:

// You get the UIView by registering with the protocol/通过注册到协议，获取到UIView

// MARK: WisdomRouterViewable Route UIView- No parameter/WisdomRouterViewable 路由UIView-无参数

let viClass = WisdomProtocol.getRouterViewable(from: LeftVIProtocol.self)

// Creating an Object/创建对象

let viewable=viClass?.routerViewable?(superview: self.view, param: nil)

【4】. Param Routing protocol/Param 路由协议：

// MARK: - Router Param Protocol

@objc public protocol WisdomRouterParamable {

// MARK: Param - Any?

@objc optional func routerParamable(param: Any?)

// MARK: Param - Any?, ((Any)->Void)?

@objc optional func routerParamable(param: Any?, closure: ((Any)->Void)?)

// MARK: Param - Any?, ((Any)->(Any))?

@objc optional func routerParamable(param: Any?, returnClosure: ((Any)->(Any))?)

}

--> \*\*Param Route Application Case/路由使用案例\*\*：

Implementation protocol/实现协议：

extension WisdomProtocolLeftVI: WisdomRouterParamable{

// Receive the external to route/接收外部到路由

func routerParamable(param: Any?) {

if let colorDic = param as? [String:Any] {

model = WisdomProtocolLeftModel.decodable(value: colorDic)

paramLabel.text = " 6. 参数路由代码示例：\n\n (1). WisdomProtocolLeftVI 需实现协议: \n -- WisdomRouterParamable 参数路由协议\n\n // MARK: 调用 路由参数方法\nlet param = ['bgColor':bgColor,'textColor':textColor,'codeColor':codeColor]\n (self?.viewable as? WisdomRouterParamable)?.routerParamable?(param: param)\n"

}

}

}

---> External start routing/外部开始路由:

// MARK: WisdomRouterViewable Route UIView- No parameter/WisdomRouterViewable 路由UIView-无参数

let viClass = WisdomProtocol.getRouterViewable(from: LeftVIProtocol.self)

viewable=viClass?.routerViewable?(superview: self.view, param: nil)

DispatchQueue.main.asyncAfter(deadline: DispatchTime.now()+2) { [weak self] in

// MARK: WisdomRouterParamable Indicates the routing protocol/WisdomRouterParamable 参数路由协议

let param = ["bgColor":"708069","textColor":"FFFFFF","codeColor":"33A1C9"]

// Optional unpack, route parameters/可选解包，路由参数

(self?.viewable as? WisdomRouterParamable)?.routerParamable?(param: param)

}

### 以上就是SDK路由核心功能介绍，完成