### 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需求。

下面具体介绍下几点协议和功能。

1. 功能
2. **定时功能**

-> 定时协议

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

-> 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 编/解码**

-> 编/解码协议

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

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

-> 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**?{

**return** WisdomProtocolCore.decodable(**Self**.**self**, value: value)

    }

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

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

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

**return** WisdomProtocolCore.decodable(**Self**.**self**, list: list)

    }

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

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

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

**return** WisdomProtocolCore.jsonable(**Self**.**self**, json: json)

    }

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

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

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

**return** WisdomProtocolCore.jsonable(**Self**.**self**, jsons: jsons)

    }

}

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?{

**return** WisdomProtocolCore.ableJson(**self**)

    }

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

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

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

**return** WisdomProtocolCore.ableEncod(**self**)

    }

}

-> 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**]]{

**return** WisdomProtocolCore.ableEncod(ables: **self**)

    }

    // **MARK: return - String?**

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

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

**return** WisdomProtocolCore.ableJsons(ables: **self**)

    }

}

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；

不需要，可以注释此处断言代码；

1. **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：

**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)

}

**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)

}

**3).** 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)

    }

}