Uther 小记 - 一个简单地蠢萌机器人

发表于 2015-08-09 | 分类于 开发笔记 | 5条评论

前阵子做了一个简单地蠢萌机器人 Uther ，主要实现以下功能：

- 收到消息后解析情感并返回对应情感的颜文字
- 使用动画切换颜文字
- 显示并可编辑历史消息

花了一周时间做了 1.0 然后上线，现在 1.2 版本刚刚审核通过。简单分享一下 Uther 从诞生到上线的过程。

代码开源在 Github: Uther。比较粗糙，没有仔细整理，各位见笑啦。

AppStore 下载地址： Uther。

Project

对于一个全新的 iOS 的项目，我的流程主要分为以下几步。

Xcode

这就不啰嗦了，直接创建一个 Swift 项目即可。不过我一般不会勾选 Xcode 自带的 git 选项，而会在后面手动配置。

Git

通过 `git init` 初始化仓库，然后在 gitignore.io 找上对应的配置，填入 `.gitignore` 文件。至此，git 初始化完毕。可以提交一次 `commit` 保存。注意，个人习惯把 Pods 文件夹目录也放到 `.gitignore` 中，如果使用自动生成的配置文件，取消那一行注释即可。

Cococapods

创建 Podfile 文件，然后通过 `def` 按照模块定义，然后再配置各个 `target` ：

```
platform :ios, '8.0'
use_frameworks!

def import_networking
    pod 'Alamofire'
    pod 'Moya'
end

target '__placeholder__' do
    import_networking
end
```

Font

字体我以前一直用的是 Consolas ，后来在微博看到了汤哥的推荐从此成了 Monoid 的忠实粉丝。

Develop

Folder

开发过程中，我的文件夹目录一般如下：

```
General
    |- Macro
    |- Models
    |- Tools
        |- LOG.swift
        |- GCD.swift
    |- Models

Sections
    |- Main
```

```
        |- Preview
        |- Setting
    |- User

Resources
```

Tool

接下来聊一聊项目中的通用工具类。

GCD

项目中使用 GCD 这个库，封装了一些常见操作。然后定义了一个简单的 struct 封装一下常用的两个方法：

```
struct GCD {
    static func async_in_worker(closure: GCDClosure) {
        gcd.async(.Default, closure: closure)
    }
    static func async_in_main(closure: GCDClosure) {
        gcd.async(.Main, closure: closure)
    }
}
```

LOG

项目中使用 `XCGLogger` 作为 Log 工具，配置起来十分方便，声明一个全局常量即可：

```
let log: XCGLogger = {
    let log = XCGLogger.defaultInstance()
    let logPath : NSURL = cacheDirectory.URLByAppendingPathComponent("XCGLogger.Log")
    log.setup(logLevel: .Debug, showThreadName: true, showLogLevel: true, showFileNames: tru
    log.xcodeColorsEnabled = true
    log.xcodeColors = [
        .Verbose: .lightGrey,
        .Debug: .darkGrey,
        .Info: .darkGreen,
        .Warning: .orange,
        .Error: .red,
        .Severe: .whiteOnRed
    ]

    return log
}()
```

DB

数据库方面，我使用的是 SQLite.swift 。可以用泛型方便的拼接各种 SQL 语句：

```
typealias Pid = Int64

struct DB {
    private static let db = SQLite.Database(documentsDirectory.URLByAppendingPathComponent('

    struct MessageDB {
        static let table = db["message"]
        // 唯一索引，主键
        static let pid = Expression<Pid>("pid")
        // 消息创建的时间
        static let createdTime = Expression<NSTimeInterval>("created_time")
        // 消息的内容
        static let content = Expression<String>("content")
    }

    static func setupDatabase() {
        db.create(table: MessageDB.table, ifNotExists: true) { t in
            t.column(MessageDB.pid, primaryKey: true)
            t.column(MessageDB.createdTime)
            t.column(MessageDB.content)
        }
    }
}
```

Flurry

使用 Flurry 做一些简单的统计，通过 extension 添加了一些代码。

比如一个全局配置的静态方法：

```
extension Flurry {
    static func start() {
        Flurry.setUserID(Keychain.userId);
        Flurry.startSession("YOUR_SESSION_ID");
    }

}
```

比如通过 enum 打一些 error log ：

```
// ERROR
extension Flurry {
    enum Error: String {
        case Setup   = "SetupError"
        case Wenzhi  = "WenzhiError"
        func logError(message: String) {
            let error = NSError(domain: "com.callmewhy.uther", code: 1001, userInfo: ["Messa
            Flurry.logError(self.rawValue, message: message, error: error)
```

```
            log.error(message)
        }
    }
}
```

比如内嵌个 struct 来统计 Message 相关的行为数据：

```
// MESSAGE
extension Flurry {
    struct Message {
        private static let send     = "Send Message"
        private static let receive  = "Receive Positive"

        static func sendMessage(l: Int) {
            Flurry.logEvent(send, withParameters: ["MessageLength": l])
        }

        static func receivePositive(p: Double) {
            Flurry.logEvent(receive, withParameters: ["MessagePositive": p])
        }
    }

}
```

Localization

我们可以通过 extension 给 string 加上 Localization 的属性，返回本地化之后的字符串：

```
extension String {
    var localized: String {
        let s = NSLocalizedString(self, tableName: nil, bundle: NSBundle.mainBundle(), value
        return s
    }
}
```

使用的时候直接调用 `"LOCALIZATION_KEY".localized` 即可。

Network

网络方面使用 Moya 作为业务和 Alamofire 的中间层。以前是自己做了个 `WhyEngine` 封装了 `Task` 和 `Request` ，后来有了 `Moya` 就简单多了：

```
// MARK: - MoyaProvider
let endpointResolver = { (endpoint: Endpoint<Sentiment>) -> (NSURLRequest) in
    let request: NSMutableURLRequest = endpoint.urlRequest.mutableCopy() as! NSMutableURLRe
    request.timeoutInterval = 2.0
    return request
}
```

```
let SentimentProvider = MoyaProvider(endpointResolver: endpointResolver)

// TODO: extension MoyaTarget to handle respose
extension MoyaProvider {
    typealias positiveHandler = PositiveValue? -> Void
    func requestPositive(endpoint: T, completion: positiveHandler) -> Cancellable {
        ...
    }
}

// MARK: - MoyaTarget
extension Sentiment: MoyaTarget {
    public var baseURL: NSURL {
        return NSURL(string: "https://wenzhi.api.qcloud.com")!
    }

    public var path: String {
        return "api/sentiment/"
    }
}
```

后面等上了 Swift2.0 可以扩展协议，就可以直接用 `MoyaTarget` 直接处理 `Response` 了。想直接把返回结果封装成 JSON 也很简单：

```
extension MoyaProvider {
    func requestJSON(endpoint: T, completion: (JSON?) -> Void) -> Cancellable {
        return self.request(endpoint) { (data, status, response, error) in
            if let d = data {
                let json = JSON(data: d)
                log.debug("\(json)")
                completion(json)
            } else {
                log.error("\(error)")
                completion(nil)
            }
        }
    }
}
```

小结

前面做过几个 Swift 项目，不过都是练练手的 Demo 级别。 Uther 算是第一个完全的 Swift 项目，没有任何 objc 的代码。这感觉真是爽，干净利落。

Swift 的最佳实践还在探索中，Uther 项目有很多可以继续改进的地方。接下来的项目准备融入一些函数式编程的元素，进一步感受各种有趣的编程范型。

欢迎讨论，多多指教~

#Uther

iOS 中的 NSTimer ›

请叫我汪二

回复 王振宇C++: 嗯玩过，但是当时不支持 Swift 。。。后来想想这种第三方还是先算了

19小时前 回复 顶 转发

王振宇C++

数据存储听说 Reaml 挺好啊，不造有没有玩过

9月2日 回复 顶 转发

余书懿

不错哦

8月25日 回复 顶 转发

__weak_Point

吊~

8月25日 回复 顶 转发

土土哥

赞~

8月25日 回复 顶 转发

社交帐号登录: 微博 QQ 人人 豆瓣 更多»

说点什么吧...

发布

汪海的实验室正在使用多说