读 Facebook App 头文件的一些收获

retweet

最近在看一些 App 架构相关的文章，也看了 Facebook 分享的两个不同时期的架构（2013 和 2014），于是就想一窥 Facebook App 的头文件，看看会不会有更多的收获，确实有，还不少。由于在选择 ipa 上的失误，下了个 7.0 版的 Facebook（最新的是 18.1），会稍有过时，不过后来又下了个 18.1 的看了下，发现变动其实不大。以下是我从头文件中获取到的一些信息（20多万行，浏览起来还是挺累的）

让视图组件可以方便地配置

这个在 Facebook 的演讲中也提到过，自定义的 UI 组件在初始化时可以传一些数值来表示想要呈现的效果，就像 HTML 和 CSS 一样，Dom 结构表示这是什么，CSS 对该结构进行个性化定制。 Facebook 是通过 Struct 来做这件事的，比如

```
struct FBActionSheetButtonMetrics {
    CDUnknownFunctionPointerType *_vptr$FBMetrics;
    _Bool _initialized;
    float leftMargin;
    float textLeftMargin;
    float bottomSeperatorSideMargin;
    float bottomSeperatorHeight;
    int detailMaxNumLines;
    UIColor *titleColor;
    //...
};
```

好处是减少了代码量，而且直观，方便复用。

尽量使用组合，适度使用继承

如果过度使用继承，尤其是继承层次过深，往往会带来更大的维护成本。有新需求或需求变更时，会花很多时间在「是否需要在基类/子类增加一个方法」，「是否需要新建一个子类」等设计相关的问题上。而组合则没有这个问题，大不了换一个组件。

不过 Objective-C 对于组合并没有特别的支持，所以实现起来会略麻烦

```
@interface People {}
@property id <Veachle> veachle;
- (void)move;
@end

@implementation People
- (id)initWithVeachle: (id <Veachle>)veachle {
    if (self = [super init]) {
        self.veachle = veachle;
    }
    return self;
}

- (void)move {
    [self.veachle move];
}
@end
```

如果有很多类似 move 这样需要交给外部的 object 来做的方法，就会显得冗余，尽管如此，比起继承来还是更方便维护的。

使用组合的话，一般会使用「依赖注入」，比如这里的 Veachle，并不需要特别指出是 Bike 还是 Car，只要有 move 方法就可以，这样就可以很方便地替换，对于 People 来说不需要做任何改动。在 Objective-C 里是通过 protocol 来实现的。

所以 Facebook 定义了一大堆的接口，包括 Delegate, DataSource 和 Protocol，ViewController 有 Protocol，也有 Delegate(如 FBMediaGalleryViewControllerDelegate)，View / Cell 也有 Delegate(如

FBMediaGalleryViewDelegate)，还有各种零零碎碎的 Protocol，如 FBDiscoveryCardProtocol, FBEventProtocol 等。

定义接口的过程也是梳理架构的过程，如果对架构理解不够深刻，是很难将接口恰当地抽象出来的。很多人放弃使用组合，有一部分原因也是架构上的不合理。

组件的粒度也是个问题，过细会导致组件过多，组合的过程就会花去很多时间；过粗又导致组件臃肿，难以复用。

当组件的接口定义完之后，使用起来大概会是这样：

```
@interface FBResponseHandler : NSObject <FBTestable, FBReceivedDataBufferDelegate,
FBResponseHandlerProtocol>

@interface FBPhotoViewController : UIViewController <FBPagingViewDelegate, FBPagingViewDataSource,
FBPresentableViewController>
```

这样一眼就大概能看出来这个 Class 大概会有哪些功能，如果某个组件要作调整，只需修改一处，就可以全局通用。

适度使用继承，可以在易维护和便利上达到平衡，比如 FBTableViewController, FBDialog 等，自定义的组件可以在它们的基础上进行开发。继承的层次一般不超过2层，比如 UITableViewController <- FBTableViewController <- FBFriendsNearbyTableViewController

依赖注入

前面讲过，组合往往和依赖注入搭配使用，Facebook 主要是通过 FBProvider, FBProviderMapData, FBProviderMap 来实现依赖注入的。

Provider 会产生一个 Object，比如 CameraControllerProvider 调用 get 方法后，会生成一个 MNCameraController 的实例。同时 Provider 还有两个子类 SingletonProvider 和 BlockProvider，前者用来生成一个单例，后者用在需要初始化参数的情景。

ProviderMap 跟 ProviderMapData 有些重复，它们之间的关系我也没有捋清，感觉 ProviderMap 像是一个 Manager，注册了一堆 Provider，然后可以通过 Provider 的 ID 来找到之前注册的 Provider。

模块化

不光是在 Cocoa 开发领域，其他的编程领域也一样，模块化是一个理想的状态，高内聚，低耦合。像 shell 命令一样，接受参数或标准输入，生成格式化的标准输出，通过管道传递给其他支持标准输入的命令行工具。

但现实场景要复杂的多，模块化的实现也更加困难。Facebook 有一个 FBAppModule 协议

```
@protocol FBAppModule <NSObject>
+ (id <FBAppModule>)instanceForSession:(FBSession *)arg1 providerMap:(FBProviderMap *)arg2;
@property(readonly, nonatomic) NSArray *supportedURLSchemes;
@property(readonly, nonatomic) NSArray *supportedKeys;
@property(retain, nonatomic) id <FBMenuItem> activeMenuItem;
@property(readonly, nonatomic) NSString *defaultIcon;
@property(readonly, nonatomic) NSString *ID;
- (UIViewController *)viewControllerForMenuItem:(id <FBMenuItem>)arg1;
```

初始化时传入一个 FBSession (后面会讲到) 和 ProviderMap，然后设置支持的 url schemes，keys(具体作用未知)，对应的 menuItem，icon(用于在 menuItem 显示) 和 ID

有了 Module ，自然还有 ModuleManager，它的作用是注册 Module，当一个 url 过来时，可以遍历 Module，看看是不是有模块可以处理这个 url，有的话，就调用该 Module 的 openURL: 方法。当然也可以根据 ModuleID 来获取 Module。

FBAppModule 是一个 Protocol，FBNativeAppModule 是对该协议的实现，所以具体的模块都继承该类。

导航管理

一般来说系统的 UINavigationController 已经够使用了，如果需要更大的自由度和更高的可定制性，可以自定义一

个导航管理器，Facebook 使用了 FBUINavigationController (Protocol) 来实现自定义导航的管理，属性和方法跟系统的差不多。 它有多个实现：FBTariffedNavigationController, FBSwipeNavigationController, FBCustomNavigationController, FBNavigationController。前面讲过继承一般不超过2层，这里是一般之外的情况，有3层。

MVVM

MVVM 是解决 Massive View Controller 的一个有效方法，独立出一个 ViewModel 作为 View 的数据源，以及处理 View 的一些交互操作，而 VC 只需要将 ViewModel 和 View 关联起来即可。一般会搭配某种绑定的实现，KVO 或 ReactiveCocoa 都可以，这样 ViewModel 的数据有变化就可以自动映射到 View 上。

Facebook 也采用了这种方式，有一个 FBViewModel 基类

```
@interface FBViewModel : NSObject

// 省略了一些相关性不大的属性和方法
@property __weak FBViewModelManager *viewModelManager; // @synthesize
viewModelManager=_viewModelManager;
@property(nonatomic) unsigned int viewModelSource; // @synthesize viewModelSource=_viewModelSource;
@property(retain, nonatomic) FBViewModelConfiguration *viewModelConfiguration; // @synthesize
viewModelConfiguration=_viewModelConfiguration;
@property(readonly, nonatomic) unsigned int viewModelVersion; // @synthesize
viewModelVersion=_viewModelVersion;
@property(readonly, nonatomic) NSString *viewModelUUID; // @synthesize viewModelUUID=_viewModelUUID;
@property(retain) FBMemModelObject *memModel; // @synthesize memModel=_memModel;
- (void)setNilValueForKey:(id)arg1;
- (id)initWithViewModelUUID:(id)arg1 viewModelVersion:(unsigned int)arg2;
- (void)setViewModelVersion:(unsigned int)arg1;
- (id)humanDescription;
- (void)loadPermanentDataModelObjectIDFromDataModelObjectID:(id)arg1 block:(CDUnknownBlockType)arg2;
- (void)didUpdateWithChangedProperties:(id)arg1;
@property __weak FBViewModelController *modelController;
@property(nonatomic) int loadState;

@end
```

Facebook 自己实现了一套 ViewModel 的更新通知机制，因为 ViewModel 都是 Immutable 的，所以无法改变，那么就需要有一个地方去集中管理这些 ViewModel，有更新时可以及时通知到， FBViewModelController 应该就是干这事的，里面有一个方法 `- (void)_notifyViewModel:(id)arg1 didUpdateWithChanges:(id)arg2;` 。但 FBViewModelManager 看起来更合适，二者的功能没有太理清楚。

FBViewModelController 还有一个 Delegate，主要有3个方法 `didUpdate[Delegate][Insert]ViewModel:` ，可以做一些事后的操作。

Builder Pattern

在定义一个 ViewController 时，往往需要接收很多个参数，以 `initWith:` 这种形式出现不太合适，除非你能容忍一个10行的方法声明。通常的做法是把这些参数声明为 property，然后在初始化 VC 后，对这些 property 赋值，然后在 ViewDidLoad 里使用这些 property。这样做有几个问题：1) 不知道哪些是需要在 ViewDidLoad 前设置的，会出现忘了设置的现象。2) 这些属性可以在外部被改动。 3) 代码不够优雅。

Builder Pattern 就是用来解决这个问题的，它跟工厂模式有点像。Facebook 也用到了这个模式，比如有一个 FBMUserFetchStatus 类，该类初始化时需要一些参数，于是就有了 FBMUserFetchStatusBuilder 类

```
@interface FBMUserFetchStatusBuilder : NSObject

+ (id)aMUserFetchStatusFromExistingMUserFetchStatus:(id)arg1;
+ (id)aMUserFetchStatus;
- (id)withIdentifiers:(BOOL)arg1;
- (id)withImageUrls:(BOOL)arg1;
- (id)withHasVerifiedPhone:(BOOL)arg1;
- (id)withCanInstallMessenger:(BOOL)arg1;
- (id)withHasMessenger:(BOOL)arg1;
- (id)withIsFriend:(BOOL)arg1;
- (id)withNickname:(BOOL)arg1;
- (id)withPhoneticName:(BOOL)arg1;
- (id)withName:(BOOL)arg1;
- (id)withUserId:(BOOL)arg1;
- (id)build;
```

```
@end
```

最后的 build 方法会生成一个 FBMUserFetchStatus 实例，有了这个 Builder 就知道有哪些参数是可以在初始化时进行设置的。

Data Manager

这是重头戏，所以看起来略累，东西很多，很可能推断错误。

先来看看实体类，首先是 FBEntityRequest

```
@protocol FBEntityRequestParse
@optional
+ (BOOL)canParse:(id)arg1 error:(id *)arg2;
@property(retain, nonatomic) NSError *syncError;
@property(nonatomic, getter=isSyncing) BOOL syncing;
- (unsigned int)parse:(id)arg1 request:(id <FBRequest>)arg2 error:(id *)arg3;
- (id <FBRequest>)request;
@end
```

所以实体都是可以被解析和同步的，还自带了一个 Request。

再来看看 FBEntity

```
@protocol FBEntity <FBEntityRequestParse, NSObject>
+ (NSURL *)entityURLForFBID:(NSString *)arg1;
@property(readonly, nonatomic) NSURL *entityURL;
@property(readonly, nonatomic, getter=isDataStale) BOOL dataStale;
@property(retain, nonatomic) NSDate *lastSyncTime;
@property(retain, nonatomic) NSString *fbid;

@optional
+ (unsigned int)collection:(FBEntityCollection *)arg1 parse:(id)arg2 request:(id <FBRequest>)arg3
error:(id *)arg4;
+ (id <FBRequest>)collectionRequest:(FBEntityCollection *)arg1;
@property(readonly, nonatomic) FBEntityDownloader *entityDownloader;
- (NSSet *)parentEdges;
- (NSSet *)parentCollections;
- (void)entityInitializeWithFBID:(NSString *)arg1;
@end
```

每个 Entity 都有一个 entityURL，或许可以用来同步？ dataStale 应该是用来表示数据是否 dirty，如果是的话，可能需要同步。 还可以请求 Collection。

FBEntityCollection 跟 FBEntity 类似，不过多了 syncAll / memberClass / allObjects 这些属性/方法。

再来看看数据请求，首先是 FBRequest，不太明白这个 Class 的具体功能，因为没有 URL，一个没有 URL 的 Request 能做什么？ 然后看到了 FBRequester，这个看起来是一个数据请求类，有 URL, responseHandler, connection状态, delegate等。但这只是单个的请求，如何对多个请求进行管理呢，这时看到了 FBNetworker，它有 +sharedNetworker, requestQueue, cancelRequests:, addRequest: 所以就是它了。等等，为什么下面还有一个 FBNetworkerRequest？ 看起来像是 FBNetworker 的 Delegate，但不确定。

为了避免 URI 散落在各处，Facebook 还专门为 NSURL 写了个 Category 来统一管理 URI。

```
@interface NSURL (FBFoundation)
+ (id)friendsNearbyURL;
+ (id)codeGeneratorURL;
+ (id)tagApprovalURLWithTagId:(id)arg1;
+ (id)tagApprovalURL;
+ (id)pokesURL;
+ (id)personExpandedAboutURLWithFBID:(id)arg1;
// ...
```

还有一个 URL 生成类，FBURLRequestGenerator，该类保存了 appSecret 和 appVersion，生成的 URL 会自动带上这些属性。

其实还有很多，实在看不下来了…

Smarter Views

我们都知道 ViewController 自带了一个 view，可以直接在这个 view 上 addSubview，正是由于这个便利性，很多创建 View 的代码也挤在了 VC 里，实在是不雅观。

更好的方法是替换 VC 的 view 为自定义的 View，然后把这个自定义 View 独立出去。比如在 `-loadView` 时覆盖 view

```
@implementation MyProfileViewController

- (void)loadView {
    self.view = [MyProfileView new];
}
```

可以同时重定义 view 的类型，如 `@property (nonatomic) MyProfileView *view`，让编译器明白 view 的类型已经变了。

因为看到了不少 VC 中都有 `-loadView` 方法，所以推断可能使用了这项技术。

FBSession

在 Web 开发领域，Session 是用来保存用户相关的信息的，FBSession 自然也不例外，不过它保存的内容还真是多呢。

```
@interface FBSession : NSObject <FBInvalidating>

+ (void)setCurrentSession:(id)arg1;
+ (id)_globalSessionForDebugging;
+ (id)DO_NOT_USE_OR_YOU_WILL_BE_FIREDcurrentSession;

@property(readonly) FBAPISessionStore *apiSessionStore; // @synthesize
apiSessionStore=_apiSessionStore;
@property(readonly) FBSessionDiskStore *sessionDiskStore; // @synthesize
sessionDiskStore=_sessionDiskStore;
@property(readonly) FBStore *store; // @synthesize store=_store;
@property(readonly) NSString *appSecret; // @synthesize appSecret=_appSecret;
@property(readonly, nonatomic, getter=isValid) BOOL valid;
@property(readonly) BOOL hasUser;
@property(readonly) NSString *userFBID;
@property(retain) FBViewerContext *viewerContext;
@property(retain) FBUserPreferences *userPreferences;
@property(retain) FBPreferences *sessionPreferences;

- (void)updateAccessToken:(id)arg1;
- (id)updateActingViewer:(id)arg1;
- (void)clearPreferences;
- (void)invalidate;
- (id)DO_NOT_USE_OR_YOU_WILL_BE_FIREDvalueForKeyRequiresUser:(id)arg1 withInitializer:
(CDUnknownBlockType)arg2;
- (id)valueForKey:(id)arg1 withInitializer:(CDUnknownBlockType)arg2;
- (id)valueForKey:(id)arg1;
- (id)initWithAppSecret:(id)arg1 store:(id)arg2 apiSessionStore:(id)arg3;

@property(readonly, nonatomic) FBReactionController *reactionController;
@property(readonly, nonatomic) FBLocationPingback *locationPingback;
@property(readonly, nonatomic) FBAppSectionManager *appSectionManager;
@property(readonly, nonatomic) FBBookmarkManager *bookmarkManager;

// and many more...
```

Session 是可以保存到本地的，有一个状态变量用来标识是否有效(valid)，是否已登录(hasUser)，用户的一些设置(这些设置会保存到本地)，可以更新 AccessToken，还带了各种 Controller 和 Manager，所以东西还是挺多的。

这里有两个特殊方法，使用后会被Fire…

Services

Service 顾名思义，提供某种服务，往往跟界面无关。从目录层级上看，Service并不在Module里面，也就是说这二者是独立的，比如 FBTimelineModule 并不包含 FBTimelineService。

Service 之间可以有依赖，这里是通过 `startAppServiceWithDependencies:` 来实现的，不过不清楚 Service 自身如何声明依赖哪些其他的 Services。

Style

App 的 Style 是一个容易被忽视的地方，开发往往看着设计图就开始写了，这样很容易造成样式不统一，且将来调整起来也不方便。

Facebook 是通过 Category 来自定义样式的，举个简单的例子：

```
@interface UIButton (FBMediaKit)
+ (id)fb_buttonTypeSystemWithTitle:(id)arg1;
+ (id)fb_buttonWithNormalImage:(id)arg1 highlightedImage:(id)arg2 selectedImage:(id)arg3;
+ (id)fb_buttonWithTemplateImage:(id)arg1;
+ (id)fb_buttonWithStyle:(int)arg1 title:(id)arg2;
@end

@interface UIButton (FBUIKit)
+ (id)fb_moreOptionsNavBarButton;
+ (id)fb_backArrowButtonWithText;
+ (id)fb_backArrowButtonWithRightPadding:(float)arg1;
+ (id)fb_backArrowButton;
@end

@interface UIButton (MNLoginFormAppearanceHelpers)
+ (id)phoneFormHeaderButton;
+ (id)singleSignOnButton;
+ (id)skipButton;
+ (id)formFieldButtonInvertedColors;
@end
```

这样也不用关心fontColor，margin，backgroundColor等，直接拿来用即可。

其他

从目录结构上来看，Facebook 有 FBUIKit, FBFoundation, FBAppKit, Module。其中 FBUIKit 和 FBFoundation 是业务无关的，可以用在其他 App 上，FBAppKit 和 Module 是业务相关的。

Module 自带资源，可以看成是一个 mini app。

使用了 EGODatabase, SDWebImage, SSZipArchive, CocoaLumberjack 这几个开源类库（可能还有更多）。

时间和能力有限，只能挖掘出这些信息，希望能带来些帮助。

--EOF--

8 Comments lzyy's blog Login

Sort by Best Share Favorite

Join the discussion…

lesvio • a day ago
有参考意义
• Reply • Share ›

swang • a day ago
hmm 自己尝试了一下，发现facebook v19.0不行（别的都可以，不知道是不是v19里加密了），每次class-dump的时候都报错。。。不知道为什么。能不能介绍一下你是怎么看头文件的？非常感谢。
• Reply • Share ›

lzyy Mod ➚ swang • a day ago
从 App Store 直接下下来的 ipa 是不能用的（因为被加密了），可以在网上找破解过的 ipa，或者在越狱的机器上破解。
• Reply • Share ›

swang ➚ lzyy • 16 hours ago
谢谢回复。恩，是的，直接class dump倒是可以的（不报错，只不过dump出来的东西是加密过的所以没什么实质内容），所以我用clutch在越狱的iPhone4上解密（过程中没有报错什么的），然后用class dump z去dump解密后的二进制文件的时候报错Seguement Fault。所以想看看你是怎么成功的。方便的话，能不能分享一下你的ipa文件的来源？谢谢！
• Reply • Share ›

PeterPan • 2 days ago
文章中里面梳理的一些模式很好的和我平时遇到的问题对应起来了 这对于我google问题关键词 更好的拓展很有帮助 谢谢楼主了
• Reply • Share ›

三泽 蔡 • 3 days ago
撸主~，能分享一下“Facebook 分享的两个不同时期的架构（2013 和 2014）”这个的地址么？
• Reply • Share ›

lzyy Mod ➚ 三泽 蔡 • a day ago
在优酷搜一下 Facebook's iOS Infrastructure， 前两个就是
• Reply • Share ›

Wilson Yuan • 6 days ago
楼主,辛苦了!赞一个!
• Reply • Share ›

Subscribe Add Disqus to your site Privacy

make the world a little better and easier

github 300+ twitter 1000+ rss 1000+