

# Table of Contents

Introduction	0
接入指南	1
安装SDK	1.1
配置Xcode工程	1.2
实现mPaaSAppInterface	1.3
使用系统设置服务	1.4
搭建一个完整Demo	1.5
附录	1.6
客户端框架	2
概述	2.1
基础术语	2.2
快速开始	2.3
进阶指南	2.4
常见问题	2.5
发布记录	2.6
附录	2.7
客户端框架	3
概述	3.1
基础术语	3.2
快速开始	3.3
进阶指南	3.4
常见问题	3.5
发布记录	3.6
附录	3.7

基础通信	4
RPC	4.1
概述	4.1.1
基础术语	4.1.2
快速开始	4.1.3
进阶指南	4.1.4
常见问题	4.1.5
发布记录	4.1.6
附录	4.1.7
APNS	4.2
通用服务	5
Hotpatch	5.1
配置服务	5.2
红点提醒	5.3
用户反馈	5.4
文件上传下载	5.5
检查更新	5.6
工具组件集	6
统一存储	6.1
概述	6.1.1
功能模块	6.1.2
APDataCenter	6.1.2.1
Key-Value存储	6.1.2.2
DAO	6.1.2.3
概述	6.1.2.3.1
关键字	6.1.2.3.2
引用方式	6.1.2.3.3

DAO代理	6.1.2.3.4
事务	6.1.2.3.5
函数重载	6.1.2.3.6
并行select	6.1.2.3.7
高级语法	6.1.2.3.8
音频录制播放	6.2
分享组件	6.3
汉字拼音处理	6.4
iOS安全检测组件	6.5
H5容器	6.6
通用控件	6.7
控件	6.7.1
UI样式描述文件	6.7.2
描述文件	6.7.2.1
自定义主题	6.7.2.2
APNavigationBar	6.7.2.3
IconFont	6.7.3
声波通讯组件	6.8
多语言组件	6.9
扫码组件	6.10
开发者工具	7
概述	7.1
Xcode工程模板	7.2
Xcode文件模板	7.3
Xcode插件	7.4
命令行工具	7.5
发布记录	7.6



## Table of Contents |

---

@cnName 1 安装SDK @priority 1

# 1 安装SDK

## 1.1 安装开发者工具

请在终端中运行下列命令安装开发者工具，详情请参考[开发者工具-概述](#)。安装前请完全退出 Xcode。

```
sh
```

## 1.2 安装SDK包

在终端中运行下列命令下载安装移动 SDK。

```
mpaas sdk update
```

---

@cnName 2 配置 Xcode 工程 @priority 2

## 2 配置 Xcode 工程

[TOC]

你可以选择手工配置一个基于移动框架的工程，通常并不需要这么做。推荐使用开发者工具生成模板工程和管理移动模块。

请使用开发者工具管理模块。下面流程会帮助你更好的了解基于移动的 **Xcode** 工程。

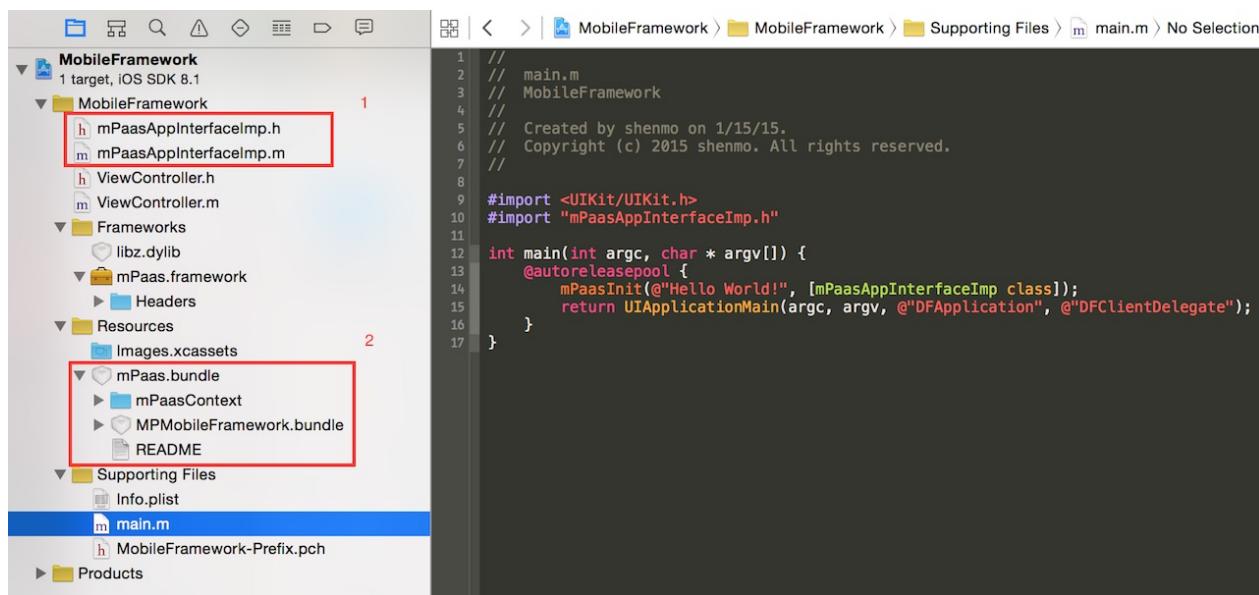
### 2.1 将 mPaas.framework 与资源加入工程

拿到 mPaas.framework 与资源 bundles 后，都加入 Xcode 工程。你只需要引用 mPaas.h 这一个头文件即可。建议创建一个公共的 PCH 文件，这样工程中所有文件都不再需要单独引用头文件了。

```
#ifdef __OBJC__
#import <UIKit/UIKit.h>
#import <mPaas/mPaas.h>
#endif
```

一个工程的目录组织结构如下：

## Table of Contents |



图中 1 为应用需要实现的接口，2 为 mPaas.framework 下的 bundle。

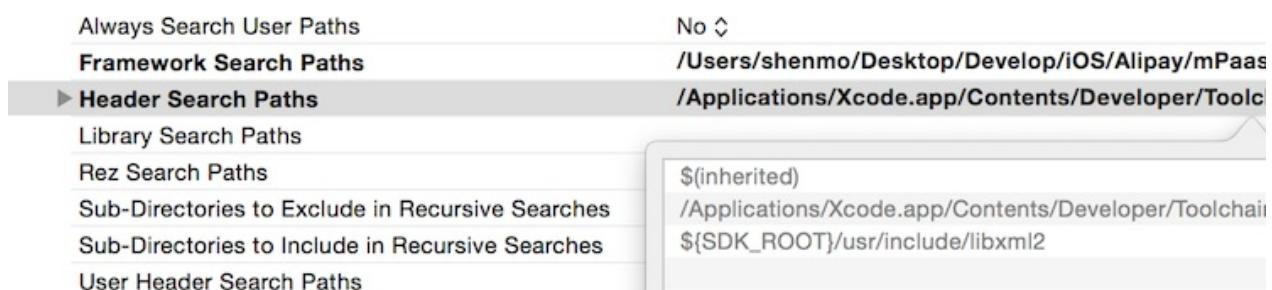
## 2.2 添加系统库

添加 mPaas.framework 后，工程可能编译不通过，需要酌情添加以下系统库：



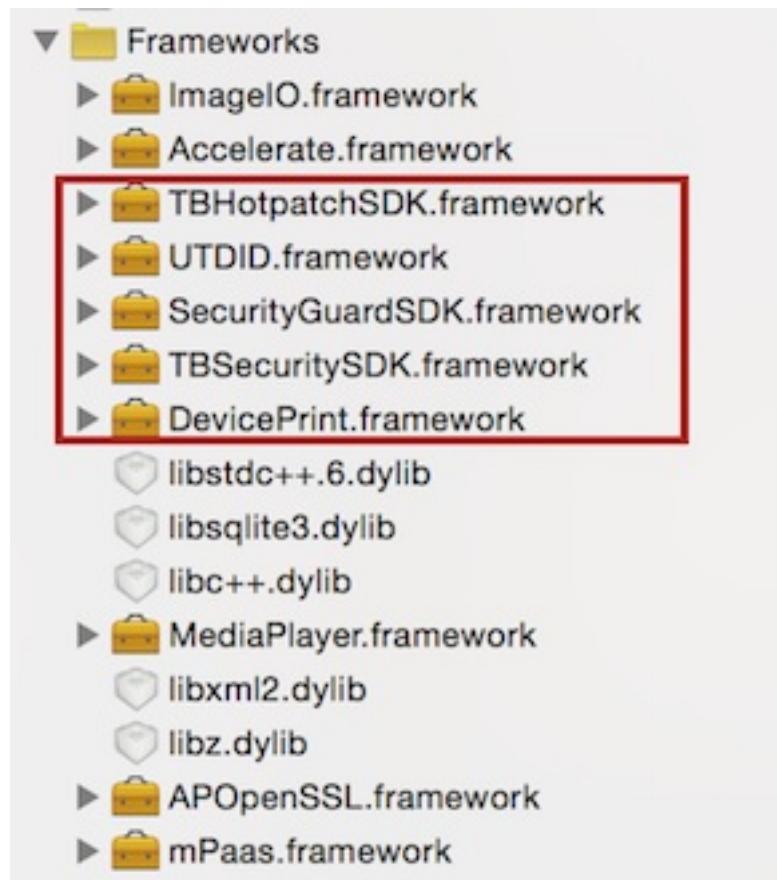
可能需要添加的头文件搜索路径：

`$(SDK_ROOT)/usr/include/libxml2`



## 2.3 添加第三方依赖

根据 mPaas.framework 包含的组件不同，可能需要依赖一些第三方 framework，也需要添加到工程中。



向 app 的 link 选项中添加-ObjC

Other Librarian Flags	
<b>► Other Linker Flags</b>	<b>-ObjC</b>
<b>▼ Path to Link Map File</b>	<Multiple values>
Debug	<input type="text" value="-ObjC"/>
Release	<input type="text" value="-ObjC"/>

## 2.4 接入移动应用框架

请把应用初始化方法修改为下面这样，并在应用的 plist 里把 main storyboard 删除掉。如果不使用移动应用框架，仅使用移动公共组件服务，可以不修改。

```
UIApplicationMain(argc, argv, @"DFApplication", @"DFClientDelegate");
```

App 可以自定义类继承 DFClientDelegate，在 UIApplicationMain 函数中最后一个参数使用自己的类，但不建议这么做。mPaasAppInterface 类的 appDelegateEvent 已经足够。请参考[环境变量](#)

```
- (id)appDelegateEvent:(mPaasEventType)event arguments:(NSDictionary*)arguments;
```

## 2.5 调用 mPaasInit 方法初始化

这个方法是初始化方法，建议在 main 函数 UIApplicationMain 方法前面调用。如果不使用移动应用框架，也可以放在 AppDelegate 的 application:didFinishLaunching 方法里。

```
/**  
 *  <#Description#>  
 *  
 *  @param appKey          应用的 appKey  
 *  @param appInterfaceClass 应用环境变量代理接口类  
 *  
 *  @return 是否成功  
 */  
BOOL mPaasInit(NSString* appKey, Class appInterfaceClass);
```

@cnName 3 实现 mPaasAppInterface @priority 3

## 3 实现 mPaasAppInterface

[TOC]

### 3.1 概述

@protocol mPaasAppInterface 这个接口是由接入方开发者实现的接口，并在 mPaasInit 方法里传入接口类。如下的 mPaasAppInterfaceImp 类。

```
#import <UIKit/UIKit.h>
#import "mPaasAppInterfaceImp.h"

int main(int argc, char * argv[]) {
    @autoreleasepool {
        mPaasInit(@"productId", [mPaasAppInterfaceImp class]);
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFClient");
    }
}
```

移动框架会使用这个类的接口来获取接入应用提供的数据，通常为一些配置。

### 3.2 mPaasAppInterface 接口说明

```
#import <UIKit/UIKit.h>

// 当使用客户端框架时，框架内部类DFClientDelegate将负责处理UIApplicationDidBecomeActive
// DFClientDelegate会在各回调方法中，通过mPaasAppInterface将事件通知给接
// 下面为已经定义的事件名与各自触发方法，如果有需要添加事件，可以重写DFClient
typedef NS_ENUM (NSInteger, mPaasEventType)
```

```
{  
    mPaasAppEventBeforeDidFinishLaunching, // @selector(application:didFinishLaunchingWithOptions:)  
    mPaasAppEventAfterDidFinishLaunching, // @selector(application:didFinishLaunchingWithOptions:)  
    mPaasAppEventBeforeStartLoader, // 启动加载器前调用，加载器加载launc  
    mPaasAppEventDidReceiveRemoteNotification, // @selector(application:didReceiveRemoteNotification:userInfo:completionHandler:)  
    mPaasAppEventDidReceiveRemoteNotificationFetchCompletion, // @selc  
    mPaasAppEventDidFailToRegisterForRemoteNotifications, // @selector(application:didFailToRegisterForRemoteNotificationsWithError:)  
    mPaasAppEventDidReceiveLocalNotification, // @selector(application:didReceiveLocalNotification:userInfo:completionHandler:)  
    mPaasAppEventQueryOpenURL, // @selector(application:openURL:sourceApplication:annotation:)  
    mPaasAppEventWillResignActive, // @selector(applicationWillResignActive:)  
    mPaasAppEventDidEnterBackground, // @selector(applicationDidEnterBackground:)  
    mPaasAppEventWillEnterForeground, // @selector(applicationWillEnterForeground:)  
    mPaasAppEventDidBecomeActive, // @selector(applicationDidBecomeActive:)  
    mPaasAppEventWillTerminate, // @selector(applicationWillTerminate:)  
    mPaasAppEventHandleWatch, // @selector(application:handleWatchKitEvent:withHandler:)  
};  
  
/**  
 * 这个需要接入 app 来实现  
 */  
@protocol mPaasAppInterface <NSObject>  
  
@optional  
  
/**  
 * 应用的简短名称，比如钱包叫“alipay”。暂时无用，可以不实现。  
 */  
- (NSString*)appBriefName;  
  
/**  
 * 应用的BundleIdentifier。暂时无用，可以不实现。  
 */  
- (NSString*)appBundleIdentifier;  
  
/**  
 * 应用RC版的BundleIdentifier，必须以rc结尾。  
 * 支付宝钱包有RC版本作为开发阶段的稳定输出版，RC与线上版使用不同的BundleI  
 * 如果接入的应用也有RC版，需要用这个方法返回RC版的BundleId。参考下面的appli  
 */
```

```
/*
- (NSString*)appRCBundleIdentifier;

#pragma mark 设置服务

/**
 * 是否要使用设置服务。
 * 如果使用设置服务，需要将配置写在GatewayConfig.plist文件中。
 * 在非发布版本，还可以在系统-设置里为应用添加一个选择环境地址的开关。
 * 返回是否要使用设置服务，如果是YES，会优先使用设置服务获取的地址，否则会从
 *
 * 接入方可以自己定义Settings.bundle来修改配置，不过选择的环境名称写入到N
 * 如果使用了设置服务，初始化时读取不到kMPSelectedEnvironment的值，会默认
 * 默认的GatewayConfig.plist结构为：
 * Root
 *   |- Debug
 *   |- Pre-release
 *   |- Release
 *     |- mPaasPushAppId           接入APNS使用的应用Id，通常为不带
 *     |- mPaasLogServerGateway    日志服务器地址（类似“http://10.
 *     |- mPaasLogProductId        日志应用Id，通常为带平台的APPKEY
 *     |- mPaasRpcGateway          RPC网关地址（类似“http://42.126.
 *     |- mPaasRpcETagURL         RPC ETag地址（类似“https://mob
 */
- (BOOL)appUseSettingService;

#pragma mark Hotpatch

/**
 * 配置在无线保镖中的，用于Hotpatch脚本加密的key的名字。Hotpatch模块拿到这
 *
 * @return 配置在无线保镖中的key的名字
*/
- (NSString*)appHotpatchScriptEncryptionKey;

#pragma mark Log

/**
```

```
* 远程日志服务器的地址，当不使用设置服务时，会回调该方法。  
*/  
- (NSString*)appRemoteLogServerURL;  
  
/**  
 * 日志服务的应用名，可能需要加一些参数，与product id可能不同。当不使用设置  
 */  
- (NSString*)appRemoteLogProductId;  
  
/**  
 * 应用定义的默认进行上传的日志种类的集合。  
 * 如果不定义，当前默认上传APLogTypeBehavior/APLogTypeCrash/APLogTypeAu  
 *  
 * @return @{@"APLogTypeBehavior"}, @{@"APLogTypeCrash} ]  
*/  
- (NSArray*)appDefaultUploadLogTypes;  
  
/**  
 * 日志服务支持客户端报活功能，报活后可以在网站上看到客户端报活记录。报活功  
 * 从后台切回前台时，距离上次报活时间少于多少秒时，不再报活。如果传0，每次启  
 * 这个不影响冷启动，如果冷启动，每次都会报活。  
 *  
 * @return 返回秒数  
*/  
- (NSInteger)appReportActiveMinIntervalSeconds;  
  
#pragma mark 框架的回调  
  
/**  
 * 使用客户端框架时，DFClientDelegate接管UIApplicationDelegate事件。接口  
 * 通常只需要实现回调函数即可，不需要重载DFClientDelegate。  
 */  
- (id)appDelegateEvent:(mPaasAppEventType)event arguments:(NSDictionary*)  
  
/**  
 * 每个继承自DTViewController的子VC，都会被默认设置为一个背景色，如果实现  
 */  
- (UIColor*)appBaseViewControllerBackgroundColor;
```

```
#pragma mark 登录态
```

```
/**  
 *  返回当前登录用户userId, 如果不是登录态, 需要返回nil。目前无用, 可以不实现  
 */  
- (NSString*)currentUserId;  
  
/**  
 *  当前的sessionId, 如果不是登录态, 需要返回nil。目前无用, 可以不实现。  
 */  
- (NSString*)currentSessionId;  
  
/**  
 *  登录成功与退出登录的通知名, 长链接、Mtop等服务会通过这个回调获得账户登录  
 */  
- (NSString*)loginSuccessNotificationName;  
- (NSString*)logoutNotificationName;
```

```
#pragma mark Scheme
```

```
/**  
 *  如果想处理RC与正式版本不同的第三方跳转scheme, 需要指定一个pattern, 当scl  
 *  配置文件里如果没有这个字段, 框架则不会处理RC版本的跳转。  
 */  
- (NSString*)appSchemePatternName;  
  
/**  
 *  返回需要添加的scheme处理器类名, 需要为DT.SchemeHandler的子类。返回类名的  
 */  
- (NSArray*)appSchemeHandlerClasses;
```

```
#pragma mark ShareKit
```

```
/**  
 *  分享渠道的配置参数, 主要是key、secret参数, 引入了分享组件时一定要实现此  
 *  
 *  @return 所要分享渠道的配置词典
```

```
* 词典格式为: @{@"laiwang" : @{@"key" : @"your_key", @"secret" : @"yo
    @"weixin" : {}, @"weibo" : {}, @"qq" : {}};
    qq的key值传入十进制APPID即可;
*/
- (NSDictionary*)appShareKitConfig;

#pragma mark PushService

/**
 * Push服务器默认地址, 当勾选了Push服务时使用。暂时无用, 可以忽略。
*/
- (NSString*)appPushProviderServerURL;

/**
 * Push的AppId。为接入APNS时使用的应用Id, 通常为不带平台的APPKEY。不使用
*/
- (NSString*)appPushAppId;

/**
 * 应用接收到远程push。暂时无用, 可以忽略。
*/
- (void)appDidReceiveRemoteNotification:(NSDictionary*)info;

#pragma mark 网络配置

/**
 * RPC的服务器地址, 不使用配置服务, 会回调此方法。
*/
- (NSString*)appRPCGatewayURL;

/**
 * RPC的ETag服务器地址, 不使用配置服务, 会回调此方法。
*/
- (NSString*)appRPCETagURL;

/**
 * 实现此方法, RPC会使用应用定义的超时时间
*
```

```

    * @return NSTimeInterval, 单位秒
    */
- (NSTimeInterval)appRPCTimeoutInterval;

/**
 * RPC请求需要加签时, 使用这个方法返回签名使用的密钥。
 * 一般不需要实现这个方法, RPC模块会默认使用 mPaaSInit 方法初始化传入的 Al
 *
 * @return 使用在无线保镖里保存的那个密钥来签名请求。
*/
- (NSString*)appRPCSignKey;

/**
 * 默认的RPC拦截器容器类名, 如果有这个方法, RPC初始化时, 会使用这个类创建R
 * 应用可能有多个RPC拦截器, 需要将这些拦截器添加到一个容器拦截器中。所有拦截
*/
- (NSString*)appRPCCommonInterceptorClassName;

/**
 * 应用是否要使用Sync服务, 当MPNetworkCtlService存在时才有效。暂时无用, 1
*/
- (BOOL)appSyncServiceActive;

/**
 * Sync服务的应用AppName, 链接地址与端口, 当MPNetworkCtlService存在时才存
*/
- (NSString*)appSyncServiceAppName;
- (NSString*)appSyncServiceConnectionURL;
- (NSInteger)appSyncServiceConnectionPort;

/**
 * 需要转为ip直连的域名, 当MPNetworkCtlService存在时有效。目前对Spdy和Sy
*/
- (NSArray*)appHttpDNSHosts;

/**
 * Mtop配置, 如果不实现这个方法并使用了Mtop功能, 那么Mtop初始化时会默认为线
 * Mtop服务仅用在接入支付宝账密登录, 并希望同步淘宝Mtop网关登录态的场景。

```

```
*  
* @return 返回Mtop环境，0：线上，1：预发，2：日常。  
*/  
- (NSInteger)appMtopEnvironment;  
  
#pragma mark 统一存储  
  
/**  
 * 统一存储功能支持数据加密，加密方法为对称AES加密。加密密钥为应用可配项。  
 *  
 * 实现这个方法，并返回32字节的加密key。统一存储会使用应用配置的加密key。  
 * 如果不实现这个方法，并且使用了统一存储功能。统一存储会使用mPaaSInit方法  
 *  
 * @return 32字节的key，放在NSData里返回。  
*/  
- (NSData*)appDataCenterDefaultCryptKey;  
  
@end
```

## @cnName 4 使用系统设置服务 @priority 4

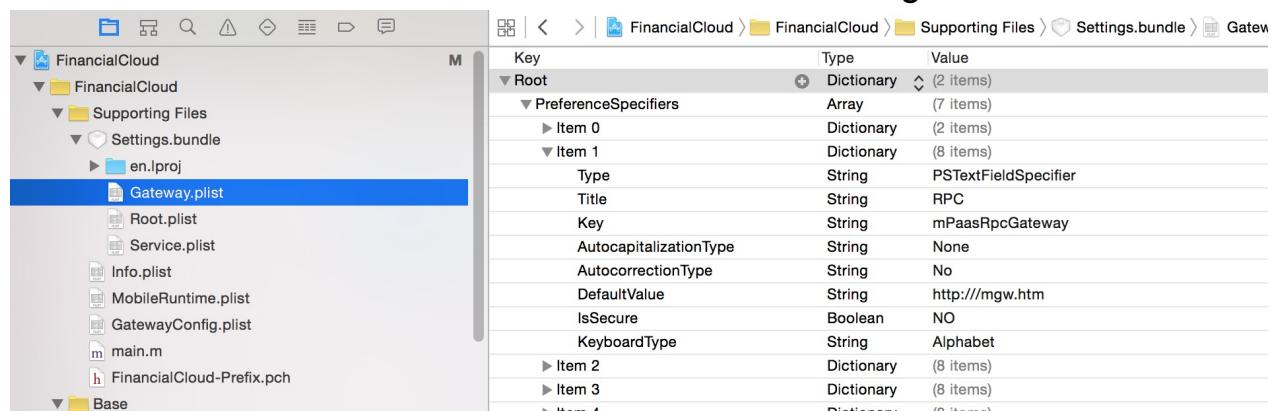
## 4 使用系统设置服务

[TOC]

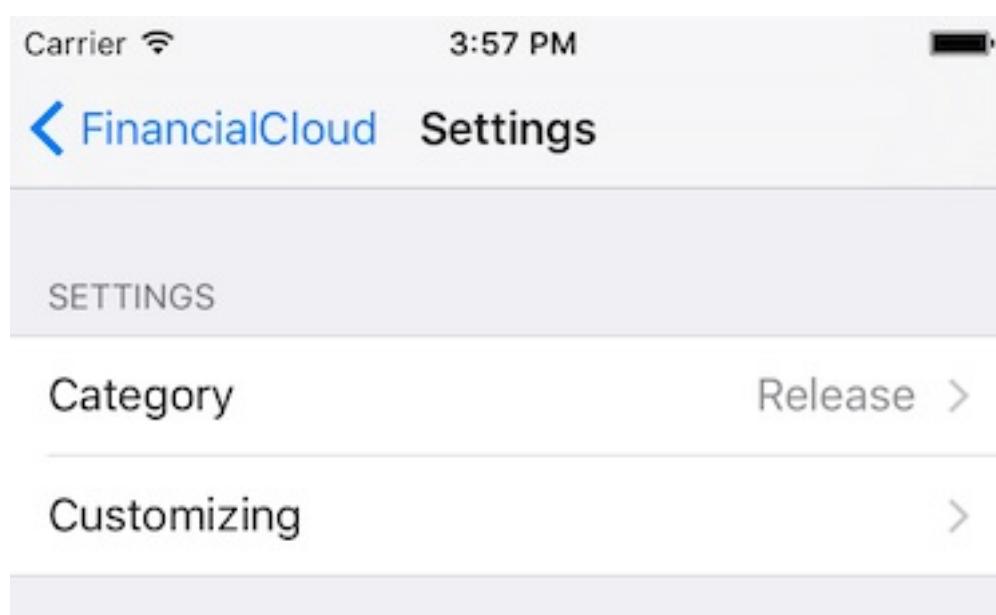
### 4.1 概述

开发者可以在系统设置里为自己的应用添加设置功能。这个功能通常可以用来在开发阶段配置服务器地址，实现在不换包的情况下切换环境。

使用开发者工具生成模板工程后，工程里会附带 `Settings.bundle`。



对应在系统设置里的选项如下：



Category 里有 Customizing , Release , Pre-release , Debug 几个选项。选择 Customizing 后，可以进入 Customizing 里设置具体的环境地址。重启应用即可生效。

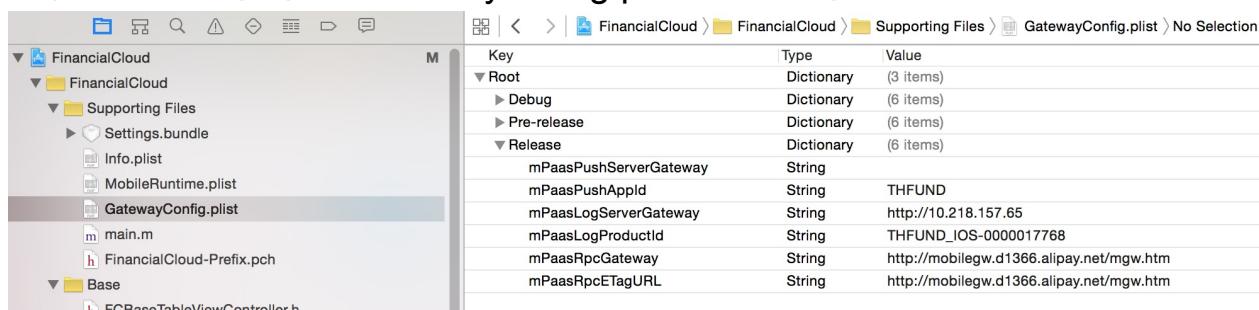
## 4.2 配置设置服务

### 4.2.1 打开设置服务开关

mPaaSAppInterface 的实现类里，appUseSettingService 方法返回 YES。

```
- (BOOL)appUseSettingService
{
    return YES;
}
```

这样会使用工程中的 GatewayConfig.plist 配置的环境变量。



开发者只需要在 GatewayConfig.plist 相应环境下配置需要的地址即可。字典里的 Key 值不可修改。

### 4.2.2 预置的设置项

名字	格式样例	含
mPaasPushServerGateway	无	暂时无可忽略除
mPaasPushAppId	THFUND	Push用 ID, Andro同, 才台信息Appke同。
mPaasLogServerGateway	<a href="http://mdap.alipay.com">http://mdap.alipay.com</a>	日志服地址, 地址是点, C上报, 诊断系服务器址。
mPaasLogProductId	THFUND_IOS-0000017768	日志服用 ID, 为带平Appke Works拼成。
mPaasRpcGateway	<a href="http://mobilegw.d1366.alipay.net/mgw.htm">http://mobilegw.d1366.alipay.net/mgw.htm</a>	RPC 介关地址上应该 https, 调试可用 http
mPaasRpcETagURL	同上	同上

### 4.2.3 不使用设置服务

当不使用设置服务时, 也就是 `appUseSettingService` 方法返回 NO。会使用 `mPaasAppInterface` 的回调方法向接入应用请求环境地址。具体涉及的方法如下:

```
/**  
 *  RPC 的服务器地址  
 */  
- (NSString*)appRPCGatewayURL;  
  
/**  
 *  RPC 的 ETag 服务器地址  
 */  
- (NSString*)appRPCETagURL;  
  
/**  
 *  Push 的 AppId  
 */  
- (NSString*)appPushAppId;  
  
/**  
 *  远程日志服务器的地址  
 */  
- (NSString*)appRemoteLogServerURL;  
  
/**  
 *  日志服务的应用名，可能需要加一些参数，与 product id 可能不同  
 */  
- (NSString*)appRemoteLogProductId;
```

## @cnName 5 搭建一个完整 Demo @priority 5

# 5 搭建一个完整 Demo

[TOC]

本指南指导如何创建基于移动的应用，并配置客户端框架、RPC、日志等基础功能。第三部分会从零开始创建一个 Demo，包括调用 RPC 接口，使用苹果 Push，进行 Crash 上报等。更多模块的配置使用方法，请参考各模块的详细文档。

## 5.1 使用移动主站

### 5.1.1 创建第一个应用

使用开发者账号登录主站后，开发者可以看到自己账号下的所有应用。



点击“添加应用”新建一款应用，输入应用的名称：

使用开发者账号登录主站后，开发者可以看到自己账号下的所有应用。



点击“添加应用”新建一款应用，输入应用的名称：

This is a screenshot of a three-step wizard for creating a new application. Step 1: 基本信息 (Basic Information) is currently selected. It asks for the mobile application name and English name, both of which are set to 'Test'. Step 2: 创建版本 (非必填) (Create Version - optional) is shown but not selected. Step 3: 完成 (Finish) is the final step. A 'Next Step' button is at the bottom.

1 基本信息      2 创建版本 (非必填)      3 完成

\* 移动应用名称: Test

\* 英文名称: Test

[下一步](#)

然后选择应用依赖的移动模块，这一步可以先跳过：

## 5.1.2 生成 mPaaS.framework

开发者可以在主站选择移动模块，生成自己的 mPaaS.framework。点击 iOS 详情，在展开的下拉页面中选择“重新打包”，可以看到可选的模块，勾选后点击确认。

组件名	组件描述
<input type="checkbox"/> MPAlipayCashierKit	支付宝快捷收银台，适用于国内场景。
<input type="checkbox"/> MPInternationalAlipayCashierKit	支付宝快捷收银台，适用于国际场景。
<input type="checkbox"/> MPAlipaySSOKit	支付宝信任登录，适用于第三方App接入支付宝信任登录。请不要与支付宝账号密码登录同时选择。
<input type="checkbox"/> MPAliUniversalAccount	支付宝账号密码登录SDK，包含UI、登录、注销、账号注册、核实身份等核心功能。
<input type="checkbox"/> MPAnimationDirector	iOS动画脚本引擎，使用脚本配置CoreAnimation，支持时间线、关键帧、函数调用等功能。
<input type="checkbox"/> MPAudioKit	AMR格式录音、播放功能。

## Table of Contents |

The screenshot shows a 'Repackage' dialog box on the mPaaS platform. The dialog lists several components with their descriptions:

组件名	组件描述
MPAlipayCashierKit	支付宝快捷收银台，适用于国内场景。
MPInternationalAlipayCashierKit	支付宝快捷收银台，适用于国际场景。
MPAlipaySSOKit	支付宝信任登录，适用于第三方App接入支付宝信任登录。请不要与支付宝账号密码同时选择。
MPAliUniversalAccount	支付宝账号密码登录SDK，包含UI、登录、注销、账号注册、核实身份等核心功能。
MPAnimationDirector	iOS动画脚本引擎，使用脚本配置CoreAnimation，支持时间线、关键帧、函数调用等功能。
MPAudioKit	AMR格式录音、播放功能。

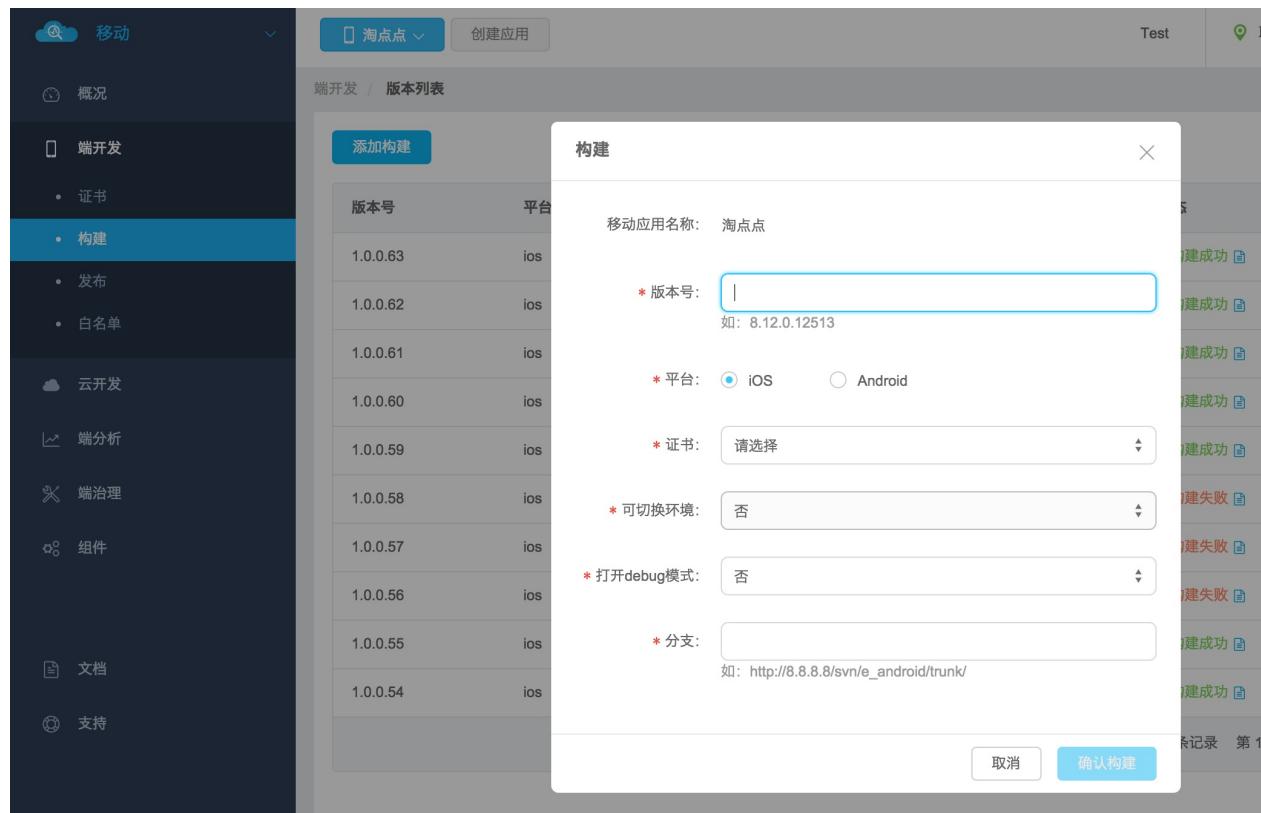
Buttons at the bottom: 取消 (Cancel) and 确认 (Confirm).

打包完成后，开发者可以自行下载压缩包，解压出 `mPaas.framework` 替换到工程中：

The screenshot shows the application details page for 'test'. It displays basic information such as AppKey: TEST\_IOS and AppSecret: 4d791a5dc9a391b10fb411e282098e87. The components section lists MPH5Service and MPMobileFoundation.

### 5.1.3 构建应用

点击构建->添加构建，填写需要的打包选项和使用的证书，就可以开始打包应用了。



构建完成可以下载应用的 ipa 包，或者在线扫描二维码安装。

#### 5.1.4 查看应用线上状态

应用接入移动后，开发者可以在主站上在线查看应用的状态。比如：报活量、版本分布、Crash 率，各项监控指标等。这部分功能会不断丰富，为开发者提供更多有价值的信息。

构建完成可以下载应用的 ipa 包，或者在线扫描二维码安装。

## 5.1.4 查看应用线上状态

应用接入移动后，开发者可以在主站上在线查看应用的状态。比如：报活量、版本分布、Crash 率，各项监控指标等。这部分功能会不断丰富，为开发者提供更多有价值的信息。

时间	闪退	报活	闪退率
13:55	0	0	-
13:54	0	0	-
13:53	0	0	-
13:52	0	0	-
13:51	0	0	-
13:50	0	0	-
13:49	0	0	-
13:48	0	0	-
13:47	0	0	-
13:46	0	0	-

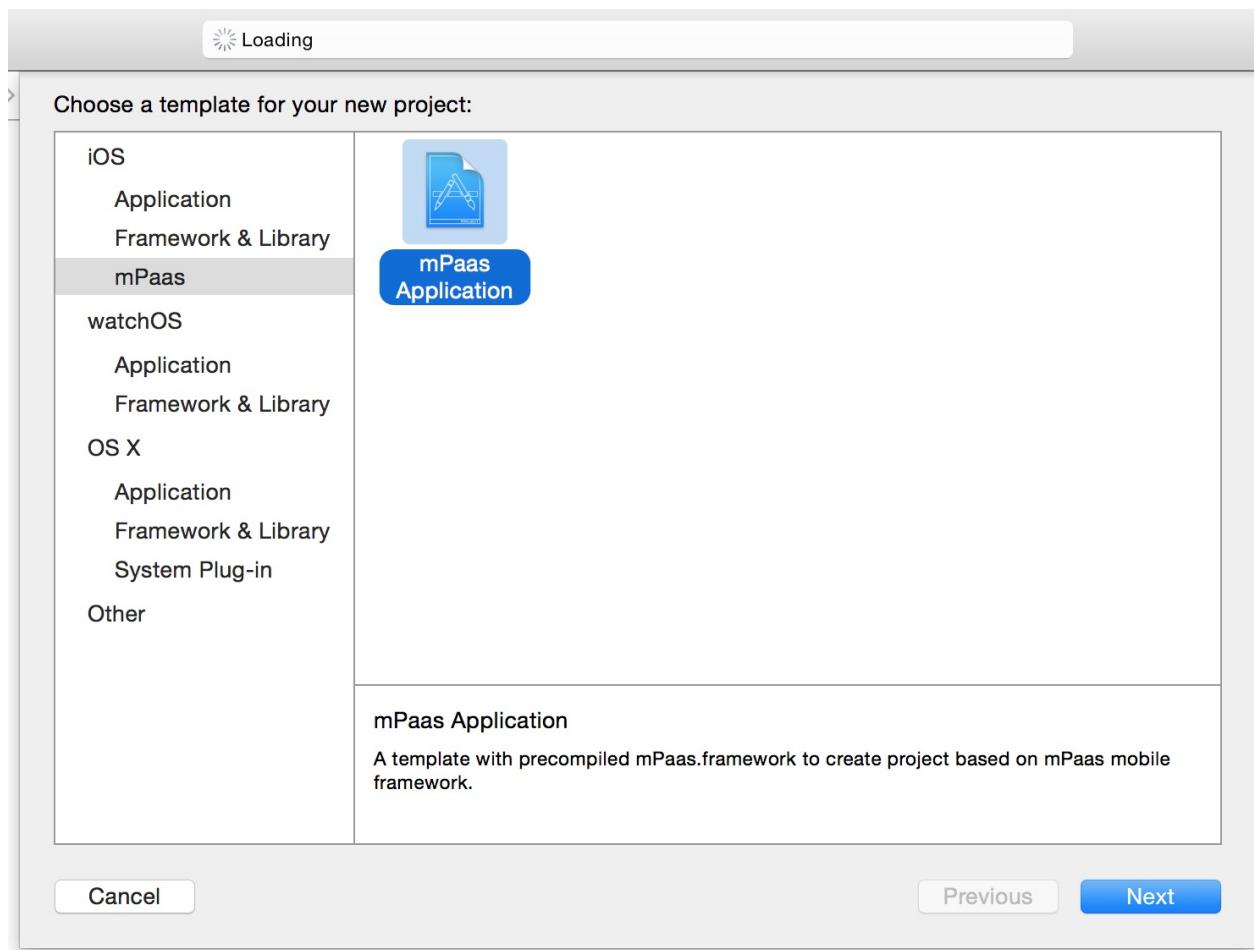
## 5.2 配置基于移动的工程

### 5.2.1 使用命令行工具或 Xcode 模板创建应用

安装命令行工具或 Xcode 模板可以快速创建基于移动的工程，省去复杂的工程配置工作。（具体可以参考[接入指南->配置 Xcode 工程章节](#)）

```
sh <(curl -s http://code.taobao.org/svn/mpaaskit/trunk/install.sh)
```

安装完成后使用命令行 `mpaas createapp xxxx` 创建新应用；或者使用 Xcode 模板



## 5.2.2 添加或删除模块

使用命令行工具或模板创建的工程为最简工程，默认只携带了最基本的移动模块。开发者在日后的开发工作中需要增减模块时，可以参考 5.1.2 生成 `mPaas.framework` 章节，去主站上重新生成 `mPaas.framework`，替换到自己的工程中。

**注意** 增加模块后，可能有新增的系统库，或第三方库依赖，或新的 `mPaas.framework` 下面可能有新增的`*.bundle` 资源目录，也需要添加到工程中。

未来，我们会提供更简单易用的 Xcode 插件，自动完成这些工作。

## 5.2.3 初始化

移动框架的初始化需要调用 `mPaasInit` 方法（用模板生成的工程已经添加该方法）：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        mPaasInit(@"TEST_IOS", [mPaasAppInterfaceImp class]);
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFCL");
    }
}
```

其中第一个参数为应用的 `mPaas appKey`，可以在主站上选择 iOS 应用详情里看到：

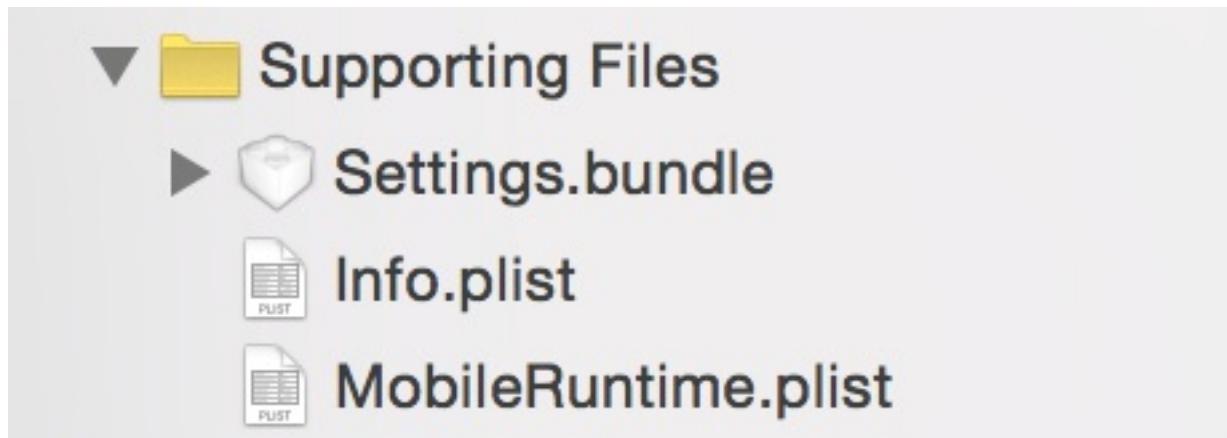
The screenshot shows a user interface for managing mobile applications. At the top, there's a blue button labeled "iOS 详情". Below it, the application name "test" is displayed along with its creation date, "2015-08-05 11:46:27". The interface is divided into sections: "基础信息" (Basic Information) and "组件" (Components). Under "基础信息", the "AppKey" field is set to "TEST\_IOS".

这个 `appKey` 与其对应的密钥，需要配置在无线保镖中，用于 RPC 请求的加签。（如果接入方不使用 RPC 功能，这个 `appKey` 也可以不传）

第二个参数是应用实现的 `@protocol mPaasAppInterface` 接口的类名。有一些参数需要应用提供，移动模块会回调这个接口的方法。详情请参考 [实现 `mPaasAppInterface`](#) 章节。

## 5.2.4 配置框架应用与服务

移动客户端框架管理的 `微应用` 与 `服务`，配置在 `MobileRuntime.plist` 文件中，如图：



具体配置方式请参考 [客户端框架->快速开始](#) 章节。

### 5.2.5 配置日志模块与 Crash 上报功能

日志模块需要两个参数：

`Log Server Gateway`：应用的日志服务器地址

`Log Product Id`：应用的日志服务产品 ID

这两个参数可以使用 `@protocol mPaaSAppInterface` 接口的下面两个回调方法返回给移动日志模块：

- `(NSString*)appRemoteLogServerURL;`
- `(NSString*)appRemoteLogProductId;`

开启 Crash 上报功能只需要在 `main` 函数中调

用 `enable_crash_reporter_service()` 方法即可。当应用使用 Xcode 在模拟器或真机调试时，不会上报 Crash 日志。

如果你的应用使用了 `Setting Service` 请参阅 [接入指南->使用系统设置服务](#) 章节来配置日志模块。

## 5.2.6 配置 RPC

RPC 网关地址使用 `@protocol mPaasAppInterface` 接口的下面这个回调方法返回给移动 RPC 模块：

```
- (NSString*)appRPCGatewayURL;
```

RPC 拦截器是实现客户端控制 RPC 行为的方式。应用应该使用一个 `Common Interceptor` 来作为所有拦截器的容器类，这个 `Common Interceptor` 本身也实现 `@protocol DTRpcInterceptor`。所有自定义的拦截器，请添加到这个 `Common Interceptor` 中。[Common Interceptor 模板代码](#)

将 `Common Interceptor` 的类名在 `@protocol mPaasAppInterface` 接口的下面这个回调方法中返回给移动 RPC 模块：

```
- (NSString*)appRPCCommonInterceptorClassName
```

参考 [基础通信->RPC->快速开始](#) 章节获取更多信息。

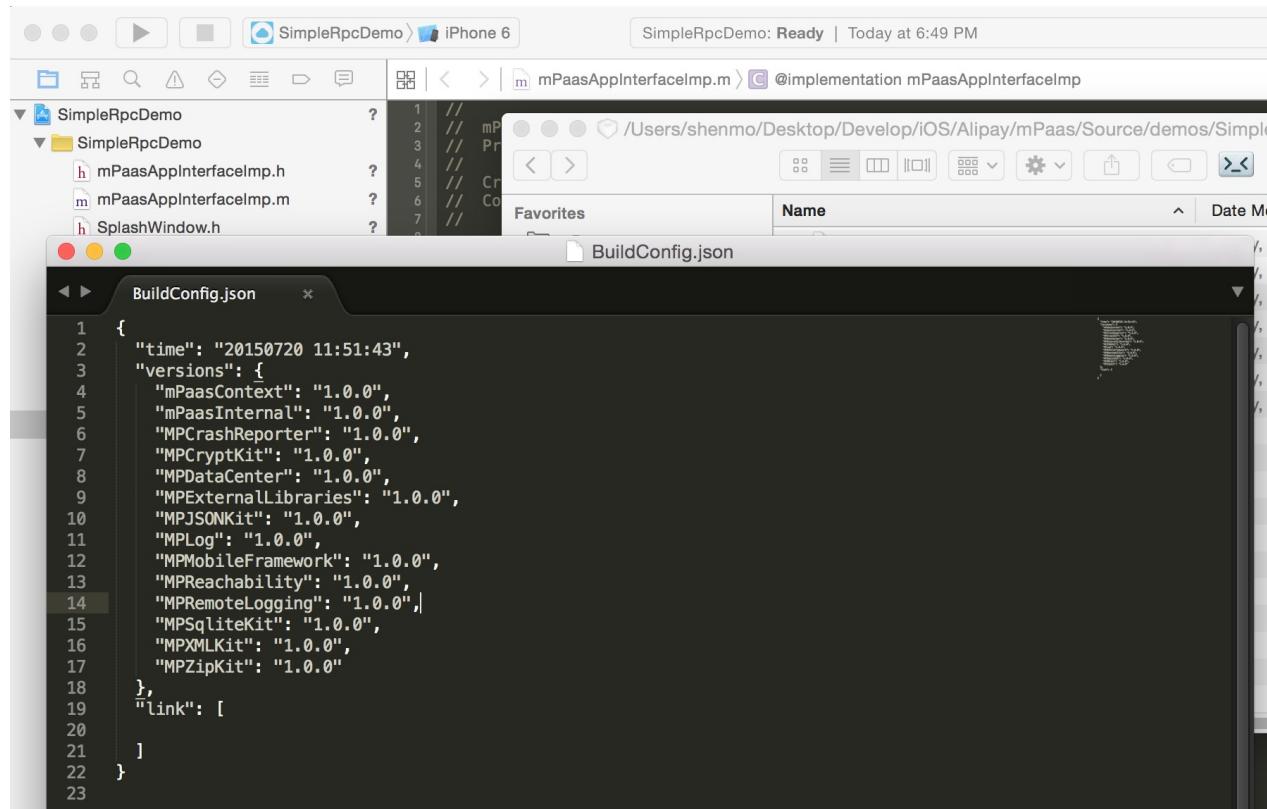
如果你的应用使用了 `Setting Service` 请参阅 [接入指南->使用系统设置服务](#) 章节来配置 RPC 模块。

## 5.3 动手搭建一个 Demo

### 5.3.1 创建工程

有了上面的介绍，我们现在开始搭建一个拥有登录，发RPC请求，接收APNS，上报Crash功能的Demo。

首先我们用命令行工具创建一个工程，叫 SimpleDemoRpc 好了。同时到主站上创建我们的应用。



我们的 Demo 要使用 RPC 以及 CommonUI 里的登录窗组件，但是打开 Demo 工程的 mPaas.framework 里面的 BuildConfig.json 文件，发现没有 RPC 和 CommonUI 模块。所以去主站上勾选MPHttpClient、MPCommonUI 重新打一个 mPaas.framework 替换进来。

注意，替换 framework 后，也要把 APCommonUI.bundle 加到工程里。

运行一下我们的工程，好多符号找不到。这是因为引入 RPC 模块后，需要添加一些第三方依赖。

Link /Users/shenmo/Library/Developer/Xcode/DerivedData/SimpleRpcDemo-dxchv  
 ① "\_OBJC\_CLASS\_\$\_OpenSecurityGuardManager", referenced from:  
 Objc-class-ref in mPaas  
 ① "\_OBJC\_CLASS\_\$\_OpenSecurityGuardParamContext", referenced from:  
 Objc-class-ref in mPaas  
 ① "\_OPEN\_ENUM\_SIGN\_COMMON\_MD5", referenced from:  
 -[DTRpcOperation signRequestBody] in mPaas  
 ① "\_OPEN\_KEY\_SIGN\_INPUT", referenced from:

把 无线保镖（SecurityGuardSDK） 添加到工程里。编译通过！ [三方库下载](#)

把代码提交到 svn 上， svn 地址可以在主站应用详情里看到：

## SimpleDemoRpc 创建于: 2015-08-05 19:54:22

---

### 基础信息

AppKey: SIMPLEDEMORPC\_IOS

Appsecret: 2ec1c750fab610f50fd818dc99c00e26

仓库类型: svn

仓库地址: [http://10.209.7.228/svn/SIMPLEDEMORPC\\_ios](http://10.209.7.228/svn/SIMPLEDEMORPC_ios)

### 5.3.2 配置

我们的 Demo 使用 Setting Service 来管理环境，所以  
让 `appUseSettingService` 这个方法返回 YES：

```
- (BOOL)appUseSettingService
{
    return YES;
}
```

测试网关地址： <http://10.218.157.194/mgw.htm>

测试日志服务器地址： <http://10.218.157.65>

日志应用 ID: SIMPLEDEMORPC\_IOS-0000017768

Push 应用 ID: SIMPLEDEMORPC

都配置到 `GatewayConfig.plist` 里（这些值为 Demo 使用，接入方 App 会使用自己的配置），如图：

SimpleRpcDemo > SimpleRpcDemo > Supporting Files > GatewayConfig.plist > No Selection		
Key	Type	Value
Root	Dictionary	(3 items)
Debug	Dictionary	(5 items)
Pre-release	Dictionary	(5 items)
Release	Dictionary	(5 items)
mPaasPushAppId	+ String	SIMPLEDEMORPC
mPaasLogServerGateway	String	http://10.218.157.65
mPaasLogProductId	String	SIMPLEDEMORPC_IOS-0000017768
mPaasRpcGateway	String	http://10.218.157.194/mgw.htm
mPaasRpcETagURL	String	http://10.218.157.194/mgw.htm

### 5.3.3 添加登录与获取APNS Token 功能

登录功能抽象成登录的微应用 `LoginAppDelegate` 与登录服务 `LoginService`。添加到 `MobileRuntime.plist` 中：

SimpleRpcDemo > SimpleRpcDemo > Supporting Files > MobileRuntime.plist > No Selection		
Key	Type	Value
Root	Dictionary	(3 items)
Launcher	String	20000001
Services	Array	(2 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
class	String	LoginServiceImpl
lazyLoading	Boolean	NO
name	String	LoginService
Applications	Array	(3 items)
Item 0	Dictionary	(3 items)
delegate	String	LoginAppDelegate
description	String	Login
name	String	20000003

登录服务负责获取应用的 APNS Token，公开账密登录接口（该接口仅为 Demo 使用），并且在登录成功后，记录该次登录的 sessionId：

```
@interface LoginServiceImpl : NSObject
{
    NSString* _sessionId;
    NSString* _pushToken;
}

@end
```

```
@implementation LoginServiceImpl

@synthesize sessionId = _sessionId;
@synthesize pushToken = _pushToken;

- (void)start
{
    NSLog(@"LoginServiceImpl started");

    // 注册推送
#ifdef __IPHONE_8_0      //ios8 注册推送
    if ([[UIDevice currentDevice] systemVersion] floatValue] >= 8.0f)
        UIUserNotificationSettings *settings = [UIUserNotificationSettings
            [[UIApplication sharedApplication] registerUserNotificationSettings:
        }];
    else
#endif
{
    [[UIApplication sharedApplication] registerForRemoteNotifications];
}

[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(applicationDidRegisterForRemoteNotifications:)
                                         name:UIApplicationDidRegisterForRemoteNotificationsNotification
                                         object:nil];

- (void)applicationDidRegisterForRemoteNotifications:(NSNotification *)notification
{
    NSString* token = [[notification.object description] stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceAndNewlineCharacterSet]];
    _pushToken = [token stringByReplacingOccurrencesOfString:@" " withString:@""];
    NSLog(@"%@", token);
}

- (NSString*)loginWithUserName:(NSString*)userName password:(NSString*)password
{
    id<MPSettingService> settings = [mPaaS() serviceWithName:@"Setting"];
    NSString* appCode = settings.settings[@"mPaaSPushAppId"];
    appCode = appCode ?: @"";
}
```

```
MPAASGWLoginDemoServiceFacade* facade = [[MPAASGWLoginDemoServiceFacade alloc] init];
_sessionId = [facade login:@{@"userId":userName, @"password":password} completion:^(NSDictionary *result, NSError *error) {
    NSLog(@"sessionId:%@", _sessionId);
    return _sessionId;
}];

@end
```

登录服务启动时，向系统请求获取APNS Token，并监听框架的通知 `UIApplicationDidRegisterForRemoteNotifications`，在获取到 APNS Token 后记录到`_pushToken` 中。

`loginWithUserName:password:`: 这个接口接收账号、密码，调用登录 RPC 请求，`appCode` 是 Push 应用 ID，可以从 `SettingService` 里获取。`osType` 传 2，表示这是 iOS 客户端，同时上传`_pushToken` 让后台将这个 Token 与用户名绑定。

在登录窗添加代码调用 `LoginService` 的接口，并在登录成功后，向日志系统和统一存储同步用户登录态，具体可参考 [用户态处理](#)

```
__block BOOL succeed = YES;
__weak DMLoginBoxViewController* weakSelf = self;
NSString* userName = _loginBox.usernameField.text;
NSString* password = _loginBox.passwordField.text;
_loginCaller = [self callAsyncBlock:^{
    @try
    {
        id<LoginService> service = [DTContextGet() findServiceByName:([
            service loginWithUserName:userName password:password];
        ]
        @catch (DTRpcException *exception)
        {
            succeed = NO;
        }
    } completion:^{
        DMLoginBoxViewController* strongSelf = weakSelf;
        strongSelf->_loginCaller = nil;
        if (succeed)
        {
            [[[APDataCenter defaultCenter] setCurrentUserId:userName];
             [[APMonitorPointManager sharedInstance] setAuthenticated:YES];
             [[APMonitorPointManager sharedInstance] setUserId:userName];
             [APRemoteLogger writeLogWithActionId:KActionID_Event extParam:]];
            [[[NSNotificationCenter defaultCenter] postNotificationName:LOG_IN_SUCCESS
                object:[DTContextGet() findApplicationByName:@"20000003"] exitAnimation:nil];
        }
        else
        {
            [strongSelf showAlert:[NSString stringWithFormat:@"登录失败"]];
        }
    }];
}
```

### 5.3.4 添加 RPC 拦截器，模拟发送需要登录态的 RPC 请求

登录完成后，我们可以发送需要登录态的 RPC 请求。为了模拟这种情况，我们约定将登录完成后获取的 sessionId 放在 RPC 请求的头里。（通常客户端软件是使用 Http Cookie 来实现，这里我们简化一下）

添加一个自定义的 RPC 拦截器，FCHeaderSessionRpcInterceptor。这个拦截器截获所有 RPC 请求，如果不是登录请求，那么就在 Header 里填写 sessionId 字段。

```
@interface FCHeaderSessionRpcInterceptor : NSObject <DTRpcInterceptor>
@end

#import "LoginService.h"
@implementation FCHeaderSessionRpcInterceptor

- (DTRpcOperation *)beforeRpcOperation:(DTRpcOperation *)operation
{
    if (![operation.rpcOperationType isEqualToString:@"alipay.mpaasgw"])
    {
        // 把 session 放到 header 里
        id<LoginService> service = [DTContextGet() findServiceByName:@"LoginService"];
        if (service.sessionId)
        {
            NSMutableURLRequest* request = (NSMutableURLRequest*)operation.request;
            [request setValue:[service.sessionId copy] forHTTPHeaderField:@"Session-Id"];
        }
    }
    return operation;
}
@end
```

在我们的 RPC 拦截器容器类里添加 FCHeaderSessionRpcInterceptor

```
@implementation DTRpcCommonInterceptor

- (id)init
{
    self = [super init];
    if (self)
    {
        NSMutableArray *interceptorList = [[NSMutableArray alloc] init];
        FCHeaderSessionRpcInterceptor *hsInterceptor = [[FCHeaderSessionRpcInterceptor alloc] init];
        [interceptorList addObject:hsInterceptor];
        self.interceptorArray = interceptorList;
    }
    return self;
}

@end
```

在 `mPaasAppInterfaceImp` 的回调中告诉 RPC 我们的拦截器容器类名

```
- (NSString*)appRPCCommonInterceptorClassName
{
    return @"DTRpcCommonInterceptor";
}
```

因为测试网关没有开启 gzip 压缩，所以我们全局设置一下 RPC，不使用 gzip 压缩。这段代码可以放在 `mPaasAppInterfaceImp` 的 `appDelegateEvent:arguments:` 方法下，实现 在 `mPaasAppEventAfterDidFinishLaunching` 事件中。

```
// 测试网关不支持 gzip
DTRpcConfig* demoRpcConfig = [[[DTRpcClient defaultClient] configForScope:kDTRpcClientScopeDefault] mutableCopy];
demoRpcConfig.requestGZip = NO;
[[[DTRpcClient defaultClient] setConfig:demoRpcConfig forScope:kDTRpcClientScopeDefault] autorelease];
```

下面我们添加代码模拟发送一个 RPC 请求

```
__block NSString* result = nil;
[self callAsyncBlock:^{
    @try
    {
        DTRpcMethod *method = [[DTRpcMethod alloc] init];
        method.operationType = @"alipay.mpaasgw.needlogintest";
        method.checkLogin = NO;
        method.returnType = @":\"NSString\"";
        method.signCheck = YES;

        result = [[DTRpcClient defaultClient] executeMethod:method pair];
        NSLog(@"%@", operation);
    }];
}
@catch(DTRpcException *exception)
{
    result = exception.reason;
}
} completion:^{
    [self showAlert:result];
}];
```

### 5.3.5 开发界面并添加模拟 Crash 的代码

在 main 函数中开启 Crash 上报功能

```
int main(int argc, char * argv[]) {  
    enable_crash_reporter_service();  
    @autoreleasepool {  
        mPaasInit(@"SIMPLEDEMORPC_IOS", [mPaasAppInterfaceImp class]);  
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFC1:  
    }  
}
```

添加一段代码模拟 Crash

```
- (void)raiseOutOfRangeException  
{  
    NSArray * array = @[@"Hello", @"World"];  
    NSString * str = [array objectAtIndex:2];  
    NSLog(@"String value:%@", str);  
}
```

这就是我们的界面，比较简陋。



登录

发送Rpc请求

制造一个Crash

OK，我们的 Demo 已经完成了，上传代码到 svn。

### 5.3.6 测试我们的 Demo

首先，上传打包使用的证书（为包含p12 与 mobileprovision 的 zip 压缩包）

平台	名称	创建时间
ios	企业发布	2015-08-05 20:30:04

同时上传苹果 Push 使用的证书（为 p12 文件）

属性	值
alias	mpaasdemo_distribution
证书环境	生产环境
主机	gateway.push.apple.com
端口	2195
生效时间	2015-05-25 15:15:43
失效时间	2016-05-24 15:15:43
issuerDN	CN=Apple Worldwide Developer Relations Certification Authority, OU=Apple Worldwide Developer Relations, O=Apple Inc., C=US
subjectDN	C=US, OU=LQ38NAVXP6, CN=Apple Production IOS Push Services: com.alipay.mpaasdemo, UID=com.alipay.mpaasdemo

**重新上传**

构建完成后，安装到手机上，使用 `alipayAdmin` 账号登录（密码为 `ali123`）。登录成功，点击发送 RPC 请求，可以看到 RPC 返回提示语“欢迎进入 mpaas!”



在 云开发->Push->推送消息 界面向我们的登录用户 alipayAdmin 推送一条消息



成功接收！



点击“制造一个 Crash”，让我们的 Demo 模拟一次 Crash，然后重新启动 Demo，让 Crash 上传到服务器。过两分钟，我们就可以在主站上看到这次 Crash 啦！

## Table of Contents |

The screenshot shows the Alibaba Cloud mobile monitoring interface. On the left, there's a sidebar with icons for '移动' (Mobile), '概况' (Overview), '端开发' (Client Development), '云开发' (Cloud Development), '端分析' (Client Analysis), '端治理' (Client Governance), '闪退监控' (Crash Monitoring) which is selected and highlighted in blue, '闪退统计' (Crash Statistics) which is also highlighted in blue, '闪退趋势' (Crash Trends), and '闪退分析' (Crash Analysis). The main area has tabs for 'SimpleDemoRpc' and '创建应用' (Create Application). It shows a breadcrumb path: 端治理 / 闪退监控 / 闪退统计. There are filters for time (15:50), platform (SIMPLEDEMORPC\_IOS), version (ALL), and detailed version (ALL). A search bar is also present. The main content is a table with columns: 时间 (Time), 闪退 (Crash), 报活 (Alive Report), and 闪退率 (Crash Rate). The data in the table is as follows:

时间	闪退	报活	闪退率
15:50	2	4	50.00%
15:49	0	0	-
15:48	0	0	-
15:47	0	0	-
15:46	0	0	-

Demo 的完整代码点此下载

@cnName 6 附录 @priority 6

# 6 附录

[TOC]

## 6.1 用户态处理

第三方应用会有自己的账号登录系统，而部分移动组件需要获取接入应用的 userId 才能正常工作。接入应用在自己的登录态发生改变时，请使用下面方法通知移动组件。（包括用户登出事件，请传 nil）

### 6.1.1 通知统一存储

当使用了统一存储时，请第一时间通知统一存储，让统一存储进行用户数据库的切换，再通知其它业务层

```
[[APDataCenter defaultCenter] setCurrentUserId:userId];
```

当用户登出时，可以不调用 setCurrentUserId 方法，统一存储会继续打开上一个用户的数据库，但这没有什么问题。

### 6.1.2 通知监控埋点

```
[[APMonitorPointManager sharedInstance] setAuthenticated:YES];
[[APMonitorPointManager sharedInstance] setUserId:userId];
```

### 6.1.3 添加 mPaasAppInterface 回调

mPaasAppInterface 这个 Protocol 里有 currentUserId 方法，这是一些移动模块用来向接入应用请求当前用户 userId 的方法。

```
- (NSString*)currentUser
{
    return userId;
}
```

## 6.2 屏蔽 NSLog 输出

开发阶段我们会允许自己的应用输出控制台日志，但是发版后希望有快捷方法屏蔽所有日志输出。请在你的应用中添加如下代码，并用宏控制，在编译 App Store 版本时重定向日志方法为空方法。

```
#include "stdio.h"

// 非 DEBUG 模式将日志关闭
#ifndef DEBUG
#define HIDE_ON_RELEASE 1
#endif

// 是否需要把 Log 中的转义字符转成中文
#ifdef DEBUG
#define CONVERT_UNICODE_ESCAPES
#endif

#if HIDE_ON_RELEASE
void NSLogv (NSString *format, va_list args) { }
int printf (const char *format, ...) { return 0; }
#endif

#ifdef CONVERT_UNICODE_ESCAPES
// 使用 NSLog 打印NSArray 或 NSDictionary 时，里面的非 ASCII 字符会显示为
static NSString* convertUnicodeCharEscapes(NSString* source)
{
    const NSInteger MAX_UNICHAR_LEN = 100;
    NSInteger lenSource = [source length];
    unichar unicharBuf[MAX_UNICHAR_LEN]; // 一次最多转 100 个
```

```

NSMutableString* dest = [[NSMutableString alloc] initWithCapacity
NSInteger i = 0, copyStart = 0, slashPos = -2, escapeIndex = -1, e

while (i < lenSource)
{
    unichar c = [source characterAtIndex:i];
    if (escapeIndex >= 0)
    {
        if (escapeIndex == 3)
        {
            escapeCharValue = 0;
            escapePow = 16 * 16 * 16;
        }

        // 16 进制转换
        if (c >= '0' && c <= '9')
            escapeCharValue += escapePow * (c - '0');
        else if (c >= 'a' && c <= 'f')
            escapeCharValue += escapePow * (c - 'a' + 10);
        else
        {
            // someting wrong
            if (escapeCharIndex > 0)
                [dest appendString:[[NSString alloc] initWithChar:unicharBuf[escapeCharIndex]]];
            copyStart = i - (5 - escapeIndex);
            escapeIndex = -1;
            i++;
            continue;
        }
    }

    if (escapeIndex == 0)
    {
        unicharBuf[escapeCharIndex] = (unichar)escapeCharValue;
        copyStart = i + 1;
        // 判断接下来还是不是\U
        if (escapeCharIndex < MAX_UNICHAR_LEN - 1 && i < lenSource)
        {
            escapeCharIndex++;
        }
    }
}

```

```
        escapeIndex = 3;
        i += 2;
    }
    else
    {
        escapeIndex = -1;
        [dest appendString:[[NSString alloc] initWithChar:c]];
    }
}
else
{
    escapeIndex--;
    escapePow /= 16;
}
}
else if (c == '\\\\')
{
    slashPos = i;
}
else if (c == 'U')
{
    if (slashPos == i - 1)
    {
        escapeIndex = 3;
        escapeCharIndex = 0;
        if (i - 2 - copyStart + 1 > 0)
        {
            [dest appendString:[source substringWithRange:NSMakeRange(i - 2 - copyStart + 1, 1]]];
            copyStart = i - 1;
        }
    }
}
i++;
}

if (copyStart == 0)
    return source;
else if (lenSource - copyStart > 0)
```

```
[dest appendString:[source substringWithRange:NSMakeRange(cop
    return dest;
}

#endif

static void _NSLog(id NIL, ...)
{
    va_list args;
    va_start(args, NIL);
    NSLogv(@"%@", args);
    va_end(args);
}

void NSLog (NSString *format, ...)
{
#ifndef HIDE_ON_RELEASE
    // EMPTY REWRITE
#else
    va_list args;
    va_start(args, format);
    NSString *logString = [[NSString alloc] initWithFormat:format argu
    va_end(args);
#endif
    #ifdef CONVERT_UNICODE_ESCAPES
        logString = convertUnicodeCharEscapes(logString);
   #endif
    _NSLog(nil, logString); // 这么弄一下，要不然不知道怎么传给 NSLogv
#endif
}
```

## 6.3 第三方库列表

SDK	英文名	功能	版本	下载地址
无线保镖	SecurityGuardSDK	黑盒保存加密数据、密钥	1.5.32	<a href="#">下载</a>
设备唯一标识	UTDID	生成设备唯一标识	1.0.2	<a href="#">下载</a>
OpenSSL	APOpenSSL	OpenSSL 轻量级封装	1.0.0	<a href="#">下载</a>

@cnName 1 概述 @priority 1

# 1 概述

[TOC]

## 1.1 客户端框架设计思想

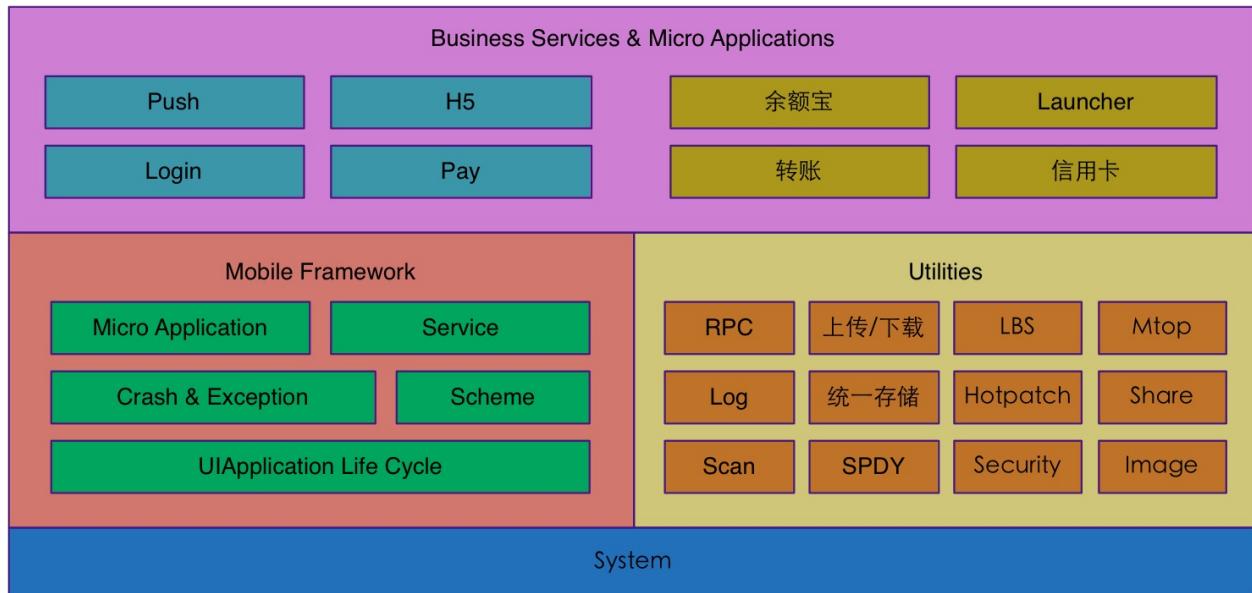
- Framework 的核心设计思想是将业务隔离成相对独立的模块，并着力追求模块与模块之间高内聚，低耦合。
- 以是否有 UI 界面 作为标准，Framework 将不同的 模块 划分为 微应用/Micro Application 和 服务/Service， 并定义了它们的基本类 DTMicroApplication 和 DTService， 进行生命周期的管理。
- Application 是独立的业务流程； Service 则是提供通用服务；
- Service 有状态，一旦启动后，其在整个客户端的生命周期中一直存在；
- Service 在后台执行，没有 UI 元素；

## 1.2 客户端框架职责

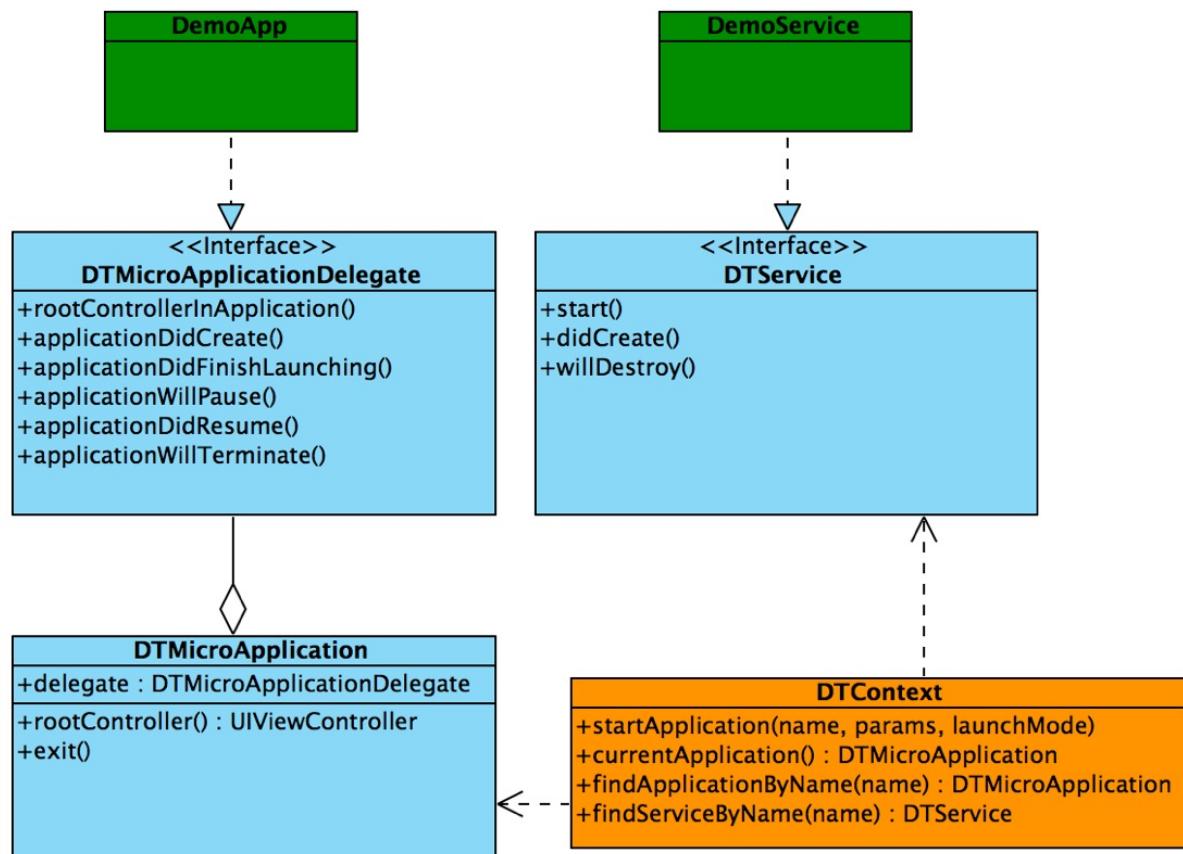
- 管理应用生命周期，UIApplication 代理事件处理与分发；
- 管理 Micro Application 与 Service，处理应用的跳转与生命周期；
- 处理客户端 Crash 捕获与上报；
- 提供 DTViewController、DT.SchemeHandler 等基类；

## 1.3 钱包客户端架构图

其中 Mobile Framework 为客户端框架



## 1.4 Micro Application 与 Service 管理 UML 图



@cnName 2 基础术语 @priority 2

## 2 基础术语

中文	英文	解释
微组件上下文	Micro Application Context	客户端微组件运行期上下文
微应用	Micro Application	客户端运行期带有用户界面的微应用
微服务	Micro Service	客户端运行期提供的轻量级服务抽象
多 Tab 应用	Multi-Tab Application	支付宝、微信这种多 Tab 切换的应用
客户端框架	Mobile Framework	源自支付宝的客户端微应用管理、服务管理、应用生命周期管理的框架

@cnName 3 快速开始 @priority 3

## 3 快速开始

[TOC]

### 3.1 客户端框架模块

客户端框架模块为 MPMobileFramework.framework，使用开发者创建的模板应用默认会依赖客户端框架。客户端框架同时依赖 Crash 日志、埋点日志等模块。

### 3.2 使用客户端框架

接入移动客户端框架的核心是将应用的生命周期交给框架来管理，只需要在 main.m 文件中指定使用 DFAApplication 与 DFClientDelegate 即可。

```
int main(int argc, char * argv[]) {
    enable_crash_reporter_service();
    @autoreleasepool {
        mPaasInit(@"BASKETBALL_IOS", [mPaasAppInterfaceImp class]);
        return UIApplicationMain(argc, argv, @"DFAApplication", @"DFClientDelegate");
    }
}
```

### 3.3 框架上下文（DTContext）

- 框架的控制中心，各个微应用都通过 context 与框架打交道；
- 提供接口以启动微应用，根据名字查找微应用，关闭微应用，管理微应用跳转等；
- 服务的注册、发现和反注册；

### 3.3.1 获得上下文接口

```
DTContext * DTContextGet();
```

### 3.3.2 微应用管理相关接口

```
/**  
 * 根据指定的名称启动一个应用。  
 *  
 * @param name 要启动的应用名。  
 * @param params 启动应用时，需要传递给另一个应用的参数。  
 * @param animated 指定启动应用时，是否显示动画。  
 *  
 * @return 应用启动成功返回 YES，否则返回 NO。  
 */  
- (BOOL)startApplication:(NSString *)name params:(NSDictionary *)params  
  
/**  
 * 根据指定的名称启动一个应用。  
 *  
 * @param name 要启动的应用名。  
 * @param params 启动应用时，需要传递给另一个应用的参数。  
 * @param launchMode 指定 app 启动的方式。  
 *  
 * @return 应用启动成功返回 YES，否则返回 NO。  
 */  
- (BOOL)startApplication:(NSString *)name params:(NSDictionary *)params  
  
/**  
 * 查找指定的应用。  
 *  
 * @param name 要查找的应用名。  
 *  
 * @return 如果指定的应用已在应用栈中，则返回对应的应用对象；否则返回 nil。  
 */  
- (DTMicroApplication *)findApplicationByName:(NSString *)name;
```

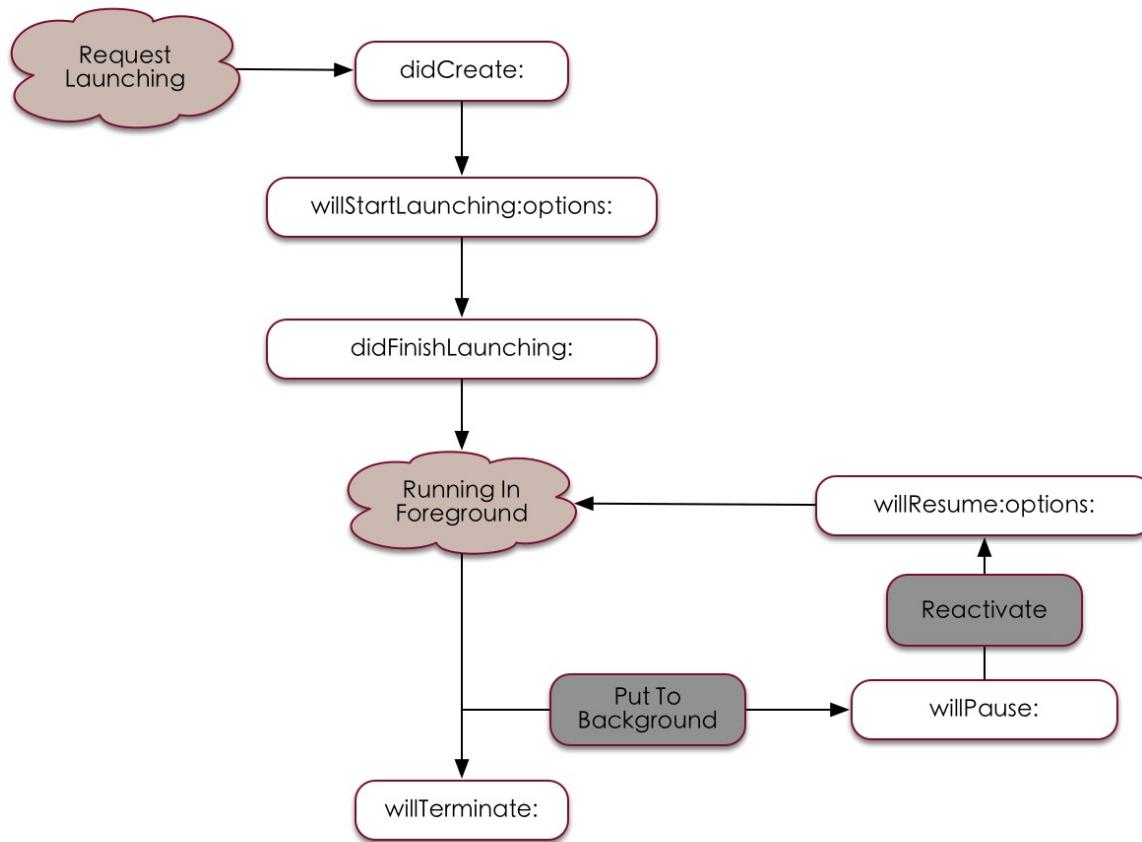
```
/**  
 * 返回当前在栈顶的应用，即对用户可见的应用。  
 *  
 * @return 当前可见的应用。  
 */  
- (DTMicroApplication *)currentApplication;
```

### 3.3.3 服务管理相关接口

```
/**  
 * 根据指定的名称查找服务。  
 *  
 * @param name 服务名  
 *  
 * @return 如果找到指定名称的服务，则返回一个服务对象，否则返回空。  
 */  
- (id)findServiceByName:(NSString *)name;  
  
/**  
 * 注册一个服务。  
 *  
 * @param name 服务名  
 */  
- (BOOL)registerService:(id)service forName:(NSString *)name;  
  
/**  
 * 反注册一个已存在的服务。  
 *  
 * @param name 服务名。  
 */  
- (void)unregisterServiceForName:(NSString *)name;
```

## 3.4 微应用（DTMicroApplication）

### 3.4.1 微应用的生命周期



### 3.4.2 DTMicroApplication

```

/**
 * 应用的描述信息。
 */
@property(nonatomic, strong) DTMicroApplicationDescriptor *descriptor

/**
 * 当前 app 的代理。
 */
@property(nonatomic, strong) id <DTMicroApplicationDelegate> delegate

/**
 * 这个 app 的运行模式。
 */
  
```

```
@property(nonatomic, assign) DTMicroApplicationLaunchMode launchMode;

/** 
 * 获取当前应用的根控制器。
 *
 * @return 当前应用的根控制器对象，这个控制器必须是DTViewController
 */
- (UIViewController *)rootController;

/** 
 * 退出本应用。
 *
 * @param animated 指定退出应用时，是否需要动画。
 */
- (void)exitAnimated:(BOOL)animated;

/** 
 * 处理 push 消息。
 *
 * @param params push 数据。
 *
 * @return 处理成功返回 YES，否则 NO。
 */
- (BOOL)handleRemoteNotifications:(NSDictionary *)params;
```

### 3.4.3 DTMicroApplicationDelegate

```
@required
/** 
 * 请求应用对象的代理返回根视图控制器。
 *
 * @param application 应用对象。
 *
 * @return 应用的根视图控制器。
 */
- (UIViewController *)rootControllerInApplication:(DTMicroApplication
```

```
@optional

/**
 * 通知应用代理，应用对象已经对经被实例化。
 *
 * @param application 应用对象。
 */
- (void)applicationDidCreate:(DTMicroApplication *)application;

/**
 * 通知应用代理，应用将要启动。
 *
 * @param application 启动的应用对象。
 * @param options 应用运行参数。
 */
- (void)application:(DTMicroApplication *)application willStartLaunch:

/**
 * 通知应用代理，应用已启动。
 *
 * @param application 启动的应用对象。
 */
- (void)applicationDidFinishLaunching:(DTMicroApplication *)application;

/**
 * 通知应用代理，应用即将暂停进入后台运行。
 *
 * @param application 启动的应用对象。
 */
- (void)applicationWillPause:(DTMicroApplication *)application;

/**
 * 通知应用代理，应用将被重新激活。
 *
 * @param application 要激活的应用对象。
 */
- (void)application:(DTMicroApplication *)application willResumeWithO
```

```
/**  
 * 通知应用代理，应用已经被激活。  
 *  
 * @param application 要激活的应用对象。  
 */  
- (void)applicationDidResume:(DTMicroApplication *)application;  
  
/**  
 * 通知应用代理，应用已经被激活。  
 *  
 * @param application 要激活的应用对象，带上参数的版本。  
 */  
- (void)application:(DTMicroApplication *)application didResumeWithOptions:(  
    DTMicroApplicationResumeOptions)options;  
  
/**  
 * 通知应用的代理，应用将要退出。  
 *  
 * @param application 应用对象。  
 */  
- (void)applicationWillTerminate:(DTMicroApplication *)application;  
  
/**  
 * 通知应用的代理，应用将要退出。  
 *  
 * @param application 应用对象。  
 * @param animated 是否以动画方式退出。  
 */  
- (void)applicationWillTerminate:(DTMicroApplication *)application animated:(BOOL)animated;  
  
/**  
 * 询问应用的代理，应用是否可以退出。  
 * 注意：只用特殊情况返回： NO，要保证默认是 YES 可以退出的。  
 *  
 * @param application 应用对象。  
 *  
 * @return 是否可以退出  
 */  
- (BOOL)applicationCanExit:(DTMicroApplication *)application;
```

```
- (BOOL)applicationShouldTerminate:(DTMicroApplication *)application;
```

## 3.5 服务（DTService）

- 服务与微应用不同，服务是在后台运行的；
- 服务启动后，通常在整个客户端的生命周期中一直存在，任何时候都可以被获取；
- 服务之间，或服务与微应用之间可以互相调用方法（如执行某个功能或获取数据等）；

### 3.5.1 DTService

```
@required
```

```
/**  
 * 启动一个服务。  
 * 注意：  
 * 框架在完成初始化操作后，会调用该方法。  
 * 如果一个服务要启动一个应用，必须在该方法被调用之后，才能启动其它的应用。  
 */  
- (void)start;
```

```
@optional
```

```
/**  
 * 创建服务完成。  
 */  
- (void)didCreate;  
  
/**  
 * 服务将要销毁。  
 */  
- (void)willDestroy;
```

### 3.5.2 创建服务的模板

定义服务的协议（Protocol）

```
@protocol MyService <DTService>

// 定义接口
- (void)doTask;

@end
```

实现服务

```
@interface MyServiceImpl : MyService
@end

@implementation MyServiceImpl

- (void)doTask
{
    // TODO: Add your code here...
}

@end
```

### 3.6 修改微应用与服务配置

使用客户端框架的应用需要添加 MobileRuntime.plist 配置文件（使用模板创建的工程已经有这个文件），结构如下：

File   < >   FinancialCloud > FinancialCloud > Supporting Files > MobileRuntime.plist > No Selection		
Key	Type	Value
Root	Dictionary	(3 items)
Launcher	String	20000001
Services	Array	(2 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
class	String	LoginServiceImpl
lazyLoading	Boolean	NO
name	String	LoginService
Applications	Array	(4 items)
Item 0	Dictionary	(3 items)
delegate	String	AboutAppDelegate
description	String	About
name	String	20000004
Item 1	Dictionary	(3 items)
Item 2	Dictionary	(3 items)
Item 3	Dictionary	(3 items)

### 3.6.1 服务配置项

字段	说明
name	服务的唯一标识
class	服务的实现类，框架在创建该服务时，会利用运行时的反射机制，创建服务实现类的实例
lazyLoading	是否延迟加载。如果是延迟加载，在框架启动时，该服务不会被实例化，只有用到该服务时才会实例化并启动。如果是非延迟加载，在框架启动时会实例化并启动该服务。默认为 NO

### 3.6.2 微应用配置项

字段	说明
delegate	应用实现 DTMicroApplicationDelegate 的类名
description	应用的描述
name	应用的名字，通常使用数字序号，startApplication 时使用 name 来启动应用

@cnName 4 进阶指南 @priority 4

## 4 进阶指南

[TOC]

### 4.1 处理 Scheme

#### 4.1.1 DTSSchemeService & DTSSchemeHandler

框架公开了 DTSSchemeService 服务和 DTSSchemeHandler 基类。三方应用接入框架后，只需要在自己工程的 MobileRuntime.plist 里将 DTSSchemeService 服务配置进去即可。

▼ Services	Array	(3 items)
▼ Item 0	Dictionary	(3 items)
class	String	DTSSchemeServiceImpl
lazyLoading	Boolean	NO
name	String	DTSSchemeService

处理不同scheme 的方法，可以实现在不同的 DTSSchemeHandler 的子类中。

```
/*
 * \code DFSchemeHandler 类用于处理操作系统对 [UIApplicationDelegate ap
 * 调用机制。
 */
@interface DTSSchemeHandler : NSObject

/**
 * 试图注册一个 <code>DFSchemeHandler</code> 的子类，使其可以处理打开链接
 *
 * @param handlerClass 要注册的 <code>DFSchemeHandler</code> 的子类。
 *
 * @return 注册成功返回 YES， 否则返回 NO。如果出现注册失败的唯一情况就是，
 */
+ (BOOL)registerClass:(Class)handlerClass;

/**
 * 反注册 open URL 的处理类。
 *
 * @param handlerClass open URL 的处理类。
 */
+ (void)unregisterClass:(Class)handlerClass;

+ (BOOL)canHandleOpenURL:(NSURL *)aURL;

- (BOOL)handleOpenURL:(NSURL *)aURL userInfo:(NSDictionary *)userInfo;

@end
```

应用重新实现 mPaasAppInterface 协议中的如下方法：

```
/*
 * 返回需要添加的 scheme 处理器类名，需要为 DTSSchemeHandler 的子类。返回
 */
- (NSArray*)appSchemeHandlerClasses;
```

将 Scheme handler 子类的类名装在数组里返回。 (数组元素为 NSString)

## 4.1.2 不使用 SchemeHandler

如果不想使用 SchemeHandler 的处理方式，还可以在 mPaasAppInterface 协议的 `appDelegateEvent` 方法里处理 `mPaasAppEventQueryOpenURL` 这个事件。

```
- (id)appDelegateEvent:(mPaasAppEventType)event arguments:(NSDictionary*)arguments
{
    if (event == mPaasAppEventQueryOpenURL) {

    }
    return @(YES);
}
```

## 4.2 处理 APNS

### 4.2.1 获得Device Token

推荐应用创建一个 PushService 来处理 APNS 的 DeviceToken。App 可以注册监听通知 `UIApplicationDidRegisterForRemoteNotifications`，`DFClientDelegate` 类在获取到 Device Token 时，会将 token 用该通知返回。代码如下：

```
@protocol PushService <DTService>

@property (nonatomic, strong, readonly) NSString* pushToken;

@end

@interface PushServiceImpl : NSObject <PushService>
{
```

```
    NSString* _pushToken;
}

@end

@implementation PushServiceImpl

@synthesize pushToken = _pushToken;

- (void)start
{
    NSLog(@"PushServiceImpl started");

    // 注册推送
#ifndef __IPHONE_8_0      //ios8 注册推送
    if ([[UIDevice currentDevice] systemVersion] floatValue] >= 8.0f)
        UIUserNotificationSettings *settings = [UIUserNotificationSettings
            [[UIApplication sharedApplication] registerUserNotificationSettings:settings
        }
    else
#endif
    {
        [[UIApplication sharedApplication] registerForRemoteNotifications
    }

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(applicationDidRegisterForRemoteNotifications:)
                                             name:UIApplicationDidRegisterForRemoteNotificationsNotification
    }

    - (void)applicationDidRegisterForRemoteNotifications:(NSNotification *)
    {
        NSString* token = [[notification.object description] stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceAndNewlineCharacterSet]];
        _pushToken = [token stringByReplacingOccurrencesOfString:@" " withString:@""];
        NSLog(@"%@", token);
    }

@end
```

## 4.2.2 上报 Device Token

应用后台开发同学需要为客户端提供 RPC 接口，负责上报 Device Token。如果需要使用登录名（userId）推通知，还需要在上报 Device Token 时绑定应用的登录账号。具体请联系应用的后台同学。

## 4.2.3 接收 Push 通知

DFClientDelegate 类在接收到 Remote | Local Notifications 时会抛下面两个通知，并将 userInfo 放在 NSNotification 的 userInfo 里返回。

```
NSString *const UIApplicationDidReceiveRemoteNotification = @"UIApplicat
NSString *const UIApplicationDidReceiveLocalNotification = @"UIApplicat
```

也可以使用 mPaasAppInterface 定义的下面几个 Event 获得，userInfo 会放在字典中返回。

```
- (id)appDelegateEvent:(mPaasAppEventType)event arguments:(NSDictionary*
```

```
mPaasAppEventDidReceiveRemoteNotification, // @selector(application:did
mPaasAppEventDidReceiveRemoteNotificationFetchCompletion, // @selector(application:did
mPaasAppEventDidReceiveLocalNotification, // @selector(application:did
```

## 4.3 Crash 捕获与上报

移动客户端框架集成了 Crash 上报功能，其中 Crash 监控封装了开源 PLCrashReporter。具有以下特点。

- 全自动化捕获 Crash 日志，记录日志，上报日志。使用者无须关心。
- Crash 日志内容更详细，可以获取窗口栈、手势等信息。

- 后台查看反解日志、日志过滤查询。

### 4.3.1 开启 Crash 上报功能

```
#import <UIKit/UIKit.h>
#import "mPaasAppInterfaceImp.h"

int main(int argc, char * argv[]) {
    enable_crash_reporter_service();
    @autoreleasepool {
        mPaasInit(@"THFUND_IOS", [mPaasAppInterfaceImp class]);
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFClient");
    }
}
```

在 main 函数中调用 enable\_crash\_reporter\_service() 方法开启日志上报功能。 (日志上报需要依赖监控系统，配置监控网关地址)

### 4.4 Apple Watch

框架 DFClientDelegate 类会处理 UIApplication 的代理方法：

```
- (void)application:(UIApplication *)application handleWatchKitExtens:
```

监听通知，原始参数 userInfo 与 reply 会放在字典里，通过通知的 userInfo 返回。

```
NSString *const UIApplicationWatchKitExtensionRequestNotifications = (
```

或使用 mPaasAppInterface 定义的 appDelegateEvent 方法处理 mPaasAppEventHandleWatch 事件。

```
- (id)appDelegateEvent:(mPaasEventType)event arguments:(NSDictionary*)arguments;
```

@cnName 5 常见问题 @priority 5

## 5 常见问题

[TOC]

### 5.1 如何使用自己的 **UIApplication** 代理类？

你可以声明自己的类继承 DFClientDelegate，覆盖 DFClientDelegate 的方法，并在 main.m 里使用自己的类启动应用。

### 5.2 如何退出所有微应用，回到 **Launcher**？

```
[DTContextGet() startApplication:@"Launcher" 的 appid" params:nil anima
```

### 5.3 A 应用上层有 B 应用，B 应用如何重新启动 A 应用并传递参数？

假设 A 应用已经启动，上层又启动了 B 应用，那么重新启动 A 应用会退出 B 应用（及所有 A 上层应用）

```
[DTContextGet() startApplication:@"A" 的 appid" params:@{@"arg": @"some
```

同时 A 应用的 DTMicroApplicationDelegate 会接收到下面事件，options 里会携带参数

```
- (void)application:(DTMicroApplication *)application willResumeWithOptions:(
```

## 5.4 如果自定义导航条样式？

如果使用了MPCommonUI控件库，可以通过[UI样式描述文件](#)定义APNavigationBar的样式。如果未使用控件库，可以自行设置UINavigationBar的样式。

@cnName 6 发布记录 @priority 6

## 6 发布记录

@cnName 7 附录 @priority 7

## 7 附录

@cnName 1 概述 @priority 1

# 1 概述

[TOC]

## 1.1 客户端监控简介

客户端的监控与日志上报是优秀移动应用必不可少的组成部分。客户端监控日志往往是分析问题，解决问题的关键所在。

客户端监控系统能提供丰富的监控手段与日志种类，涵盖 Crash，网络，存储，线程，内存，用户行为等多种类型。搭配的后台有能力处理海量用户的数据上报，并从海量数据中为开发者分析出有价值的数据。开发者可以使用主站直观地看到这些数据。

## 1.2 客户端监控组成部分



## 1.2.1 数据统计与趋势分析

提供应用安装量，UV，PV，版本更新情况的数据统计，分析地域、时间上的应用使用习惯，为开发者提供有力数据支撑。

## 1.2.2 舆情监控

监控媒体、微博等渠道上的应用反馈，第一时间发现问题。（该功能还在开发中）

## 1.2.3 埋点与自动化埋点

通过开发者手工埋点或自动化埋点，捕获应用的运行状态，出现问题时最大程度复原问题场景。除了最基本的埋点，还会自动化的进行内存、线程、流量、应用启动时间等性能监控埋点。

@cnName 2 基础术语 @priority 2

## 2 基础术语

中文	英文	解释
埋点	Monitor Point	程序运行到某处触发的事件，会记录日志。
自动化埋点	Automatic Monitor Point	通过 Hook UI 组件，自动监控用户的手势操作；同时可以自动监控应用的界面切换等事件。

@cnName 3 快速开始 @priority 3

## 3 快速开始

[TOC]

### 3.1 监控模块

完整的监控模块包括以下几个组成部分：

库	功能
MPRemoteLogging	日志埋点处理模块
MPCrashReporter	日志捕获模块
MPMonitor	自动化埋点与自动化性能监控模块
MPLog	行为日志模块

你不需要单独引用这些库的头文件，只需要引用 `mPaas.h` 即可。如果使用我们的工具生成模板工程，这也会为你做好。

### 3.2 配置监控的网关地址与应用 ID

当使用系统设置服务时，请将监控的网关地址与应用 ID 配置到 `GatewayConfig.plist` 文件中。

Key	Type	Value
▼ Root	Dictionary	(3 items)
▶ Debug	Dictionary	(6 items)
▶ Pre-release	Dictionary	(6 items)
▼ Release	Dictionary	(6 items)
mPaasPushServerGateway	String	
mPaasPushAppId	String	THFUND
mPaasLogServerGateway	String	http://10.218.157.65
mPaasLogProductId	String	THFUND_IOS-0000017768
mPaasRpcGateway	String	http://mobilegw.d1366.alipay.net/mgw.htm
mPaasRpcETagURL	String	http://mobilegw.d1366.alipay.net/mgw.htm

如果不使用系统设置服务，请在 `mPaasApplInterface` 接口的实现类里，使用下面两个方法将日志配置信息返回。

```
/**  
 * 远程日志服务器的地址  
 */  
- (NSString*)appRemoteLogServerURL;  
  
/**  
 * 日志服务的应用名，可能需要加一些参数，与 product id 可能不同  
 */  
- (NSString*)appRemoteLogProductId;
```

更多信息请参考[使用系统设置服务](#)

@cnName 4 进阶指南 @priority 4

## 4 进阶指南

[TOC]

### 4.1 用户诊断日志

#### 4.1.1 功能介绍

用户诊断日志是非自动上传的日志，会按照日期与小时分别记录文件。开发者可以在主站通过远程命令向目标手机发送推送，目标用户的手机会将特定时间段的日志文件上传。

诊断日志所在模块为 `MPLog`，引用的头文件为 `MPLog/APLog.h`。通常只需要引用 `mPaas.h` 即可。

诊断日志工作流程如下：

- 按照时间段写入本地文件，形成独立的日志文件。
- 开发者在主站配置日志上传参数，添加采集诊断日志的任务。
- 客户端收到任务后，根据网络类型和日志时间段，打包日志文件上传。  
    开发者在主站下载日志查看。

#### 4.1.2 日志接口

这几个接口都可以，目前还不支持分级别上传的功能。

```
#define APLogError(tag,fmt, ...) \
APLogToFile(tag, kAPLogLevelError, fmt, ##__VA_ARGS__)

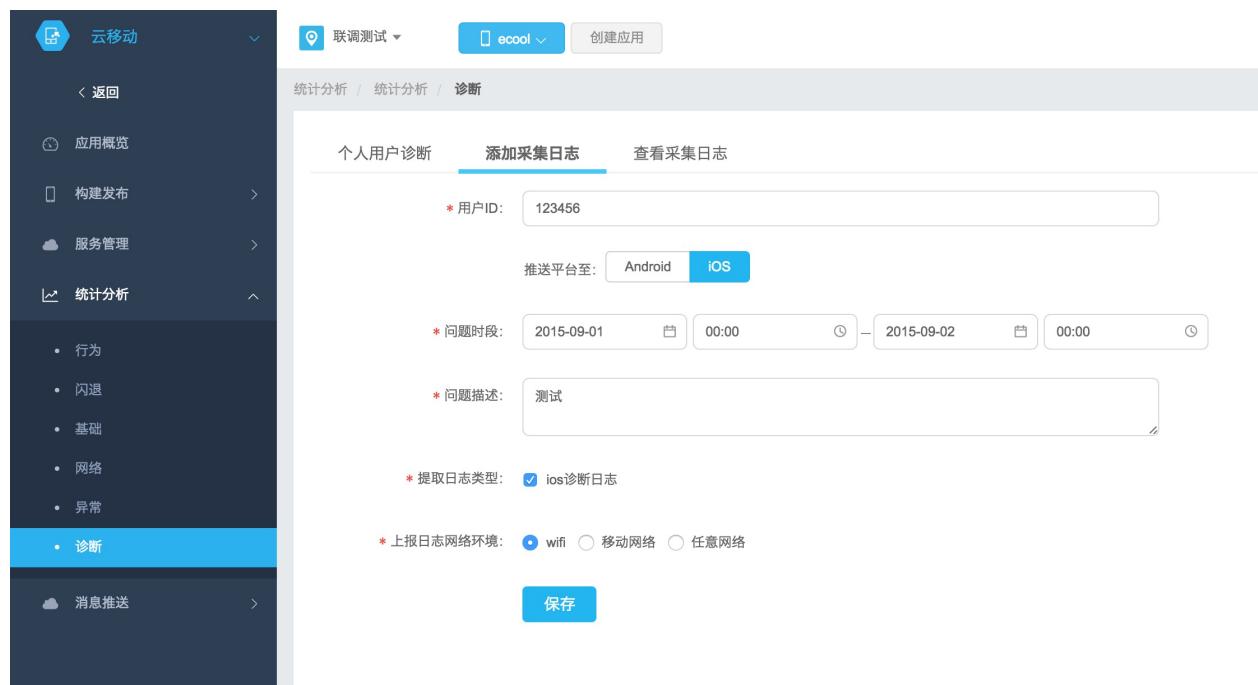
#define APLogWarn(tag,fmt, ...) \
APLogToFile(tag, kAPLogLevelWarn, fmt, ##__VA_ARGS__)

#define APLogInfo(tag,fmt, ...) \
APLogToFile(tag, kAPLogLevelInfo, fmt, ##__VA_ARGS__)

#define APLogDebug(tag,fmt, ...) \
APLogToFile(tag, kAPLogLevelDebug, fmt, ##__VA_ARGS__)
```

### 4.1.3 在主站创建采集任务

进入主站，选择应用，在 纵向分析 - 诊断 中选择 添加采集日志，并选择 iOS 平台。输入需要采集的用户 ID 与时间段，点击保存。



可以在 `查看采集日志` 中看到自己创建的任务。

The screenshot shows the 'Cloud Mobile' dashboard with the 'Statistics Analysis' module selected. In the center, under the 'Diagnosis' tab, there is a table titled 'My Collection' showing one log entry:

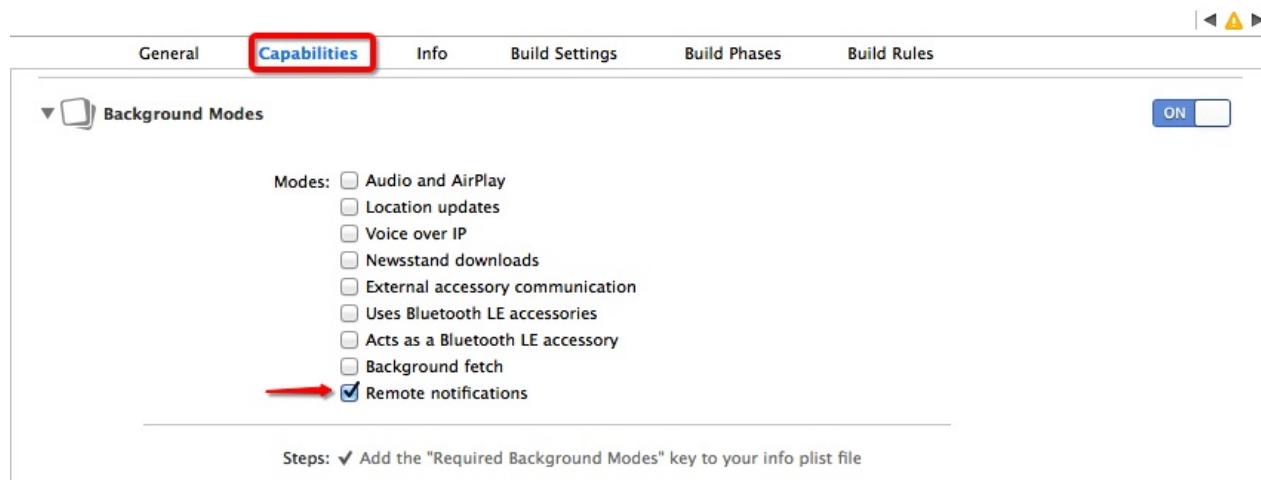
Creation Time	User ID	Description	Executor	Status	Operations
2015-09-09 19:47:57	123456	Test	alipayAdmin	Task issued successfully	<a href="#">View</a>   <a href="#">Reschedule Task</a>   <a href="#">Delete</a>

点击 `查看`，可以看到任务执行情况，如果执行并上传成功，可以在这里下载日志。

This screenshot displays detailed information about a specific task. It includes the following details:

- Task Basic Information:** Task ID: 219, User ID: 123456, Executor: alipayAdmin, Description: Test.
- Configuration Parameters:** Configuration parameters: {}.
- Task Status:** Task issued successfully, with a link to [Reschedule Task](#).
- iOS Diagnosis Log:** Problem time period: 2015-09-01 00:00:00 - 2015-09-02 00:00:00, Network type: WIFI, Sub-task status: Unhandled.

采集日志需要客户端支持静默Push，如图



## 4.2 客户端报活

客户端报活逻辑已经预埋在框架中，如果不使用框架，可以使用下面的方法报活。

```
[APRemoteLogger writeLogWithActionId:KActionID_Event extParams:nil appContext:nil]
```

移动客户端框架默认的报活逻辑为：

- 无应用进程时，启动时上报一次；
- 有应用进程时，从后台切回前台，上报一次。

如果想降低报活频率，可以重写 `mPaasAppInterface` 里下面的方法，并返回一个时间间隔：

```
/**  
 * 从后台切回前台时，距离上次报活时间少于多少时，不再报活。如果传 0，每次后台启动都报活。  
 * 这个不影响无进程启动，如果无进程启动，每次都会报活。  
 * @return 返回秒数  
 */  
- (NSInteger)appReportActiveMinIntervalSeconds;
```

## 4.3 登录上报

应用可以选择性的上传用户登录事件，并在上报参数里上传 userId：

```
[APRemoteLogger writeLogWithActionId:KActionID_Event extParams:@[{"use
```

## 4.4 日志模块与上传行为

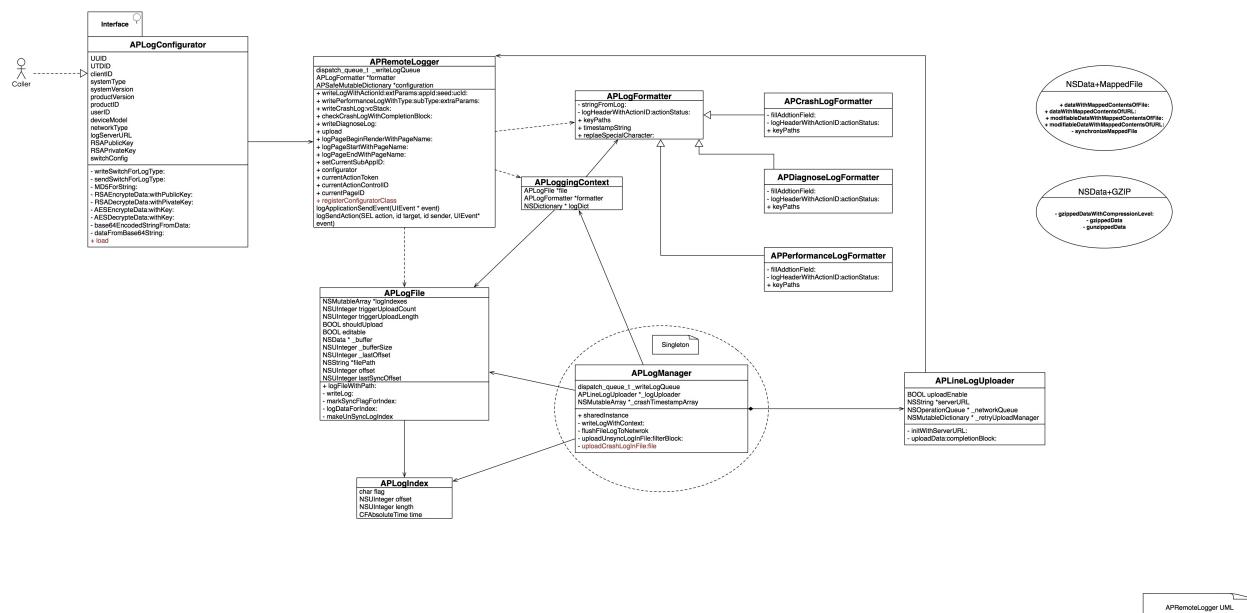
[TOC]

### 4.4.1 埋点工作流程

- 根据所调用的埋点接口和所传入的参数将日志写入本地文件，文件的组织格式是一行表示一条 log，每条 log 开头有标记信息。其中第一个数字代表日志的上传状态，0 表示日志还没有上传，1 代表日志已经上传。
- 在一个日志文件的范围内，如果满足 100 条或者写入的日志已经够 10KB（参数可能有微调），则触发一次上传操作。
- 上传成功则标记日志已经上传过，否则等待下次触发继续上传。
- 日志上传到服务器后，服务器根据日志的文件头或者其他参数将日志归入不同的日志文件。
- Crash 日志由于需要及时保证上传，每次发生 Crash 的时候会在下次启动时触发上传。

### 4.4.2 日志模块类图

# Table of Contents |



APRemoteLogger UML

@cnName 5 常见问题 @priority 5

## 5 常见问题

@cnName 6 发布记录 @priority 6

## 6 发布记录

@cnName 7 附录 @priority 7

## 7 附录

@cnName 1 概述 @priority 1

# 1 概述

RPC 为远程调用，对业务代码公开稳定的接口，但在下层可以支持使用 HTTP、HTTPs、SPDY、Socket 等多重通道发送请求。具有以下特点：

- 客户端调用代码由工具自动生成，客户端不需要关心网络通信、协议以及使用的数据格式；
- 服务端的数据返回自动反解生成 Objective-C 对象，无需额外编码；
- 提供数据压缩、缓存、批量调用等优化；并支持国际化；
- RPC Exception 统一的异常处理，弹对话框、toast 等；
- 支持 RPC 拦截器，实现定制化的请求与处理；
- 请求签名验证机制；

@cnName 2 基础术语 @priority 2

## 2 基础术语

中文	英文	解释
远程过程调用	RPC	无
RPC拦截器	RPC Interceptor	在请求发送前与发送后由 RPC 模块调用上层注册的回调接口，可以定制化 RPC 请求与个性化错误处理。
SPDY	SPDY	Google 开发的基于 TCP 的应用层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。
RPC 网关	RPC Gateway	RPC 请求到业务服务器前都需要经过网关的分发，网关还会进行与业务逻辑无关的错误处理。

@cnName 3 快速开始 @priority 3

## 3 快速开始

[TOC]

### 3.1 RPC 模块

RPC 模块为 MPHttpClient，里面的关键类功能如下：

类	功能
DTRpcClient	RPC 客户端管理类，为单例。负责发送 RPC 请求，处理 RPC 请求配置。
DTRpcConfig	RPC 配置，指定 RPC 的网关、缓存策略、是否使用 GZip 等。可以控制每个配置的影响范围。
DTRpcMethod	RPC 请求方法描述，记录 RPC 请求的方法名、参数、返回类型等信息。
DTRpcOperation	RPC 请求操作实体，为 NSOperation 子类。
DTRpcAsyncCaller	RPC 异步调用器，提供异步调用 RPC 的快捷接口。

### 3.2 网关配置

应用在 mPaasAppInterface 接口的实现中，通过下面的接口返回网关地址。如果使用系统设置服务，可以将 RPC 网关地址配置在 GatewayConfig.plist 中。

```
#pragma mark 网络配置

/**
 *  RPC 的服务器地址
 */
- (NSString*)appRPCGatewayURL;

/**
 *  RPC 的 ETag 服务器地址
 */
- (NSString*)appRPCETagURL;
```

更多信息请参考[使用系统设置服务](#)

### 3.3 发 RPC 请求

RPC 请求必须在子线程调用，同时回调方法默认为主线程。

- 使用 DTRpcAsyncCaller

```
__block ALPRSAPKeyResult* result = nil;
[DTRpcAsyncCaller callAsyncBlock:^{
    @try
    {
        DTRpcMethod *method = [[DTRpcMethod alloc] init];
        method.operationType = @"alipay.client.getRSAKey";
        method.checkLogin = NO ;
        method.returnType = @"@\"ALPRSAPKeyResult\"";
        method.signCheck = YES;

        result = [ [DTRpcClient defaultClient] executeMethod:method p;
    }
    @catch (NSEException *exception) {
        NSLog(@"%@", exception);
    }
} completion:^{
    NSLog(@"%@", result);
}];
```

- 使用框架内 DTViewController

下面例子为在 DTViewController 的子类 ViewController 的方法里发送 RPC 请求。

```
__block ALPRSAPKeyResult* result = nil;
[self callAsyncBlock:^{
    @try
    {
        DTRpcMethod *method = [[DTRpcMethod alloc] init];
        method.operationType = @"alipay.client.getRSAKey";
        method.checkLogin = NO ;
        method.returnType = @"@\"ALPRSAPKeyResult\"";
        method.signCheck = YES;

        result = [ [DTRpcClient defaultClient] executeMethod:method p;
    }
    @catch (NSEException *exception) {
        NSLog(@"%@", exception);
    }
} completion:^{
    NSLog(@"%@", result);
}];
```

@cnName 4 进阶指南 @priority 4

## 4 进阶指南

[TOC]

### 4.1 使用 RPC 拦截器

#### 4.1.1 RPC 拦截器工作原理

下面是简化的 DTRpcClient 执行 RPC 操作的代码。在请求发送前与发送后，会将这个 DTRpcOperation 对象传给每个 interceptor 进行处理。拦截器不需要处理时直接放过即可。

```
- (id)executeOperation:(DTRpcOperation *)operation responseHeaderField:(NSString *)fieldValue:(NSString *)value
{
    if (self.interceptor) {
        [self.interceptor beforeRpcOperation:operation];
    }

    [self.requestQueue addOperation:operation];
    [operation waitUntilFinished];

    if (self.interceptor) {
        operation = [self.interceptor afterRpcOperation:operation];
    }

    if (operation.error) {
        [DTRpcException raise:(DTRpcErrorCode)operation.error.code message:(operation.error.message)];
    }
}

return operation.resultObject;
}
```

## 4.1.2 RPC 拦截器接口协议

```

/**
 * 对 RPC 请求的拦截器。所有的 RPC 请求，在发送到服务端之前或从服务端接收到
 * 都会经过拦截器，拦截器可以做相应的处理。
 */
@protocol DTRpcInterceptor <NSObject>

@optional

/**
 * RPC 请求的前置拦截器。在 RPC 请求开始之前，会调用该方法。
 *
 * @param operation 当前的 PRC 请求。
 */
- (DTRpcOperation *)beforeRpcOperation:(DTRpcOperation *)operation;

/**
 * RPC 请求后置拦截器。在 PRC 请求结束之后，会调用该方法。
 *
 * @param operation 当前的 RPC 请求。
 */
- (DTRpcOperation *)afterRpcOperation:(DTRpcOperation *)operation;

- (void)handleException:(NSError *)exception;

@end

```

- 注册 RPC 拦截器容器

从上面代码看到，`DTRpcClient` 只会记录一个拦截器对象，但是应用可能有多个拦截器一起处理 RPC 请求。这里我们可以创建一个拦截器容器，让这个容器实现 `beforeRpcOperation` 与 `afterRpcOperation` 接口，并将事件分发给自己的所有子拦截器。

这个拦截器容器需要开发者添加到自己的工程中，并在 `mPaasApplInterface` 的下面方法里将类名返回，RPC 模块会自动创建这个容器对象。

```
- (NSString*)appRPCCommonInterceptorClassName
{
    return @"DTRpcCommonInterceptor";
}
```

RPC 拦截器是实现客户端控制 RPC 行为的方式。应用应该使用一个 `Common Interceptor` 来作为所有拦截器的容器类，这个 `Common Interceptor` 本身也实现 `@protocol DTRpcInterceptor`。所有自定义的拦截器，请添加到这个 `Common Interceptor` 中。

#### [Common Interceptor 模板代码](#)

- 基本错误处理 RPC 拦截器

容器拦截器并不能处理任何错误，开发者需要接入自己的拦截器处理网络错误。下面提供的这个 `Base Interceptor` 模板，可以处理网络错误，与 `Session` 失效的错误。开发者根据应用需要进行修改，并集成到自己的工程代码里。将这个 `DTRpcBaseInterceptor` 添加到拦截器容器中，即可使用。

#### [Base Interceptor 模板代码](#)

## 4.2 RPC 请求签名

- 签名逻辑

为保证客户端请求不被篡改和伪造，RPC 请求有签名机制，加签会自动完成。基本签名方法为对 `requestBody` 中的内容生成串，并使用保存在无线保镖中的加密密钥进行加密。将签名放在请求中发给网关，网关使用相同方式签名，校验两个签名是否相等。

- 控制请求加签

`DTRpcMethod` 类中的 `signCheck` 属性指定该请求是否需要加签：

```
/** 是否签名 */
@property(nonatomic, assign) BOOL signCheck;
```

但通常，应用中的所有 RPC 请求都需要签名，这可以通过 RPC 拦截器实现。示例代码：

```
@implementation RPCDefaultSignCheckInterceptor

- (DTRpcOperation *)beforeRpcOperation:(DTRpcOperation *)operation
{
    operation.method.signCheck = YES;
}

@end
```

- 使用无线保镖保存签名密钥

客户端使用的签名密钥保存在无线保镖中，如图：

### Appkey配置 i

Appkey是MTOP平台给应用颁发的唯一标识，MTOP平台通过Appkey来鉴别应用身份；Appsecret与Appkey一一对应，是每次接口调用生成请求签名的密钥串，能够保证每次接口调用安全可靠。

注意：有些sdk（比如mtop）需要根据索引来访问数据，请按照实际需求填写，第一行为index0，第二行为index1，以此类推

mtop默认情况下index0为线上环境、index1为预发环境、index2日常环境

Appkey	Appsecret	
THFUND_IOS	*****	删除
<a href="#">增加一行</a>		

应用在生成无线保镖密钥文件时，在 Appkey 栏目里创建项目，Key 值为自己的应用 Appkey。这个 Appkey 与 mPaasInit 方法里初始化的 Key 必须保持一致。RPC 模块会使用这个 Appkey 对应的密钥来对请求进行加签。

```
int main(int argc, char * argv[]) {  
    enable_crash_reporter_service();  
    @autoreleasepool {  
        mPaasInit(@"THFUND_IOS", [mPaasAppInterfaceImp class]);  
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFC1:  
    }  
}
```

@cnName 5 常见问题 @priority 5

## 5 常见问题

[TOC]

### 5.1 在 iOS9 上无法发送 RPC 请求

因为 iOS9 上，苹果默认限制应用必须使用 https 协议，所以当调试网关为 http 时无法发送请求。解决方法为在应用的 plist 文件中添加如下字段，允许应用发送 http 请求。

▼ NSAppTransportSecurity	Dictionary	(1 item)
NSAllowsArbitraryLoads	Boolean	YES

### 5.2 为 RPC 请求添加特殊的 Header 字段

通常，RPC 请求的 Http Header 是由 RPC 模块默认添加的。如果某条 RPC 请求有特殊的 Header 字段，可以在调用 RPC 时传入 requestHeaderField 字典。

```
/**
 * 根据指定的 \code DTRpcMethod 执行一个 RPC 请求。
 *
 * @param method 一个 \code DTRpcCode 类型的实例，描述了 RPC 请求的相关信息。
 * @param params RPC 请求需要的参数。
 * @param field 添加到 request 里面的 head
 * @param responseBlock 回调方法
 *
 * @return 如果请求成功，返回指定类型的对象，否则返回 nil。
 */
- (id)executeMethod:(DTRpcMethod *)method params:(NSArray *)params re
```

使用 RPC 拦截器也可以实现，下面示例演示在调用 `rpc.operationA` 这个 RPC 请求时，向 Header 中添加字段。

```
@implementation HeaderFillRpcInterceptor

- (DTRpcOperation *)beforeRpcOperation:(DTRpcOperation *)operation
{
    if (![operation.rpcOperationType isEqualToString:@"rpc.operationA"])
    {
        NSMutableURLRequest* request = (NSMutableURLRequest*)operation;
        [request setValue:@"field" forHTTPHeaderField:@"key"];
    }
    return operation;
}

@end
```

## 5.3 如何控制某个 RPC 请求走特殊的网关？

应用设置全局网关后，某个 RPC 请求需要走特殊网关时，可以通过 DTRpcConfig 来实现。

下面代码控制 `alipay.client.saveUserProposalInfo` 这个 RPC 请求走固定网关地址。

```
DTRpcConfig* config = [[[DTRpcClient defaultClient] configForScope:kDT
feedbackConfig.gatewayURL = [NSURL URLWithString:@"https://mobilegw.alip
feedbackConfig.operationType = @"alipay.client.saveUserProposalInfo";
[[DTRpcClient defaultClient] setConfig:config forScope:kDT
configScope];
```

## 5.4 如何监控RPC耗时与错误

主站提供RPC耗时与流量监控功能，如图。



这个功能由一个RPC拦截器负责，请下载下面代码，将RPC拦截器代码加入自己的工程中。并将这个拦截器添加到公共拦截器中使其生效。[代码下载](#)

@cnName 6 发布记录 @priority 6

## 6 发布记录

@cnName 7 附录 @priority 7

## 7 附录

@cnName APNS @priority 2

## APNS

### 1 客户端

请参考[客户端框架-进阶指南-处理 APNS](#)

### 2 服务端与主站管理

请参考[无线通知服务](#)

@cnName 1 Hotpatch @priority 1

# 1 Hotpatch

[TOC]

## 1.1 简介

### 1.1.1 Hotpatch 组成部分

苹果商店应用发布有审核过程，无法像安卓平台那样灵活。Hotpatch 提供一种不需要发布新版本，快速修复线上严重 Bug 的方法。

Hotpatch 框架为 MPCommandCenter 模块，包含指令推送与动态修复两个模块。

指令推送：负责与服务器同步脚本，控制脚本的运行版本和运行时机。

动态修复：执行 Lua 脚本，替换 Objective-C 方法完成 Hotpatch。

### 1.1.2 Hotpatch 工作过程

- 程序启动，执行已经下载并存储到本地的脚本。
- 程序启动，与服务器通信并下载脚本，下载成功后会立刻执行新的脚本。
- 程序从后台切回前台，与服务器通信并同步脚本，下载成功后会在下次启动时执行。

## 1.2 发布脚本

开发者本地验证通过后，在服务端配置需要发布的脚本。可以设置推送百分比以及推送的用户 ID。（该功能正在开发中）

## 1.2.1 脚本验签

为了保障安全性，Hotpatch 有自定义的验签流程，保证脚本来源的正确性。

验签流程如下：

- 读取Lua 脚本内容，对内容字符串做 md5 加密。
- 以二进制格式读取Lua 脚本内容，对二进制内容做 AES128 加密。
- 创建 zip 文件，将上述两项内容添加到 zip 文件中。
- 以二进制格式读取上述 zip 文件，再进行一次 AES128 加密，加密后数据保存为 zip 文件。
- 以二进制格式读取上述 zip 文件，使用 SHA1 算法，做一次签名。
- 以二进制格式读取上述 zip 文件，加上数据分隔符，加上签名数据，组成 js 脚本。

客户端获取 js 脚本，也会使用以上反顺序得到可以真正运行的脚本。

## 1.2.2 Lua2Js 工具使用方法

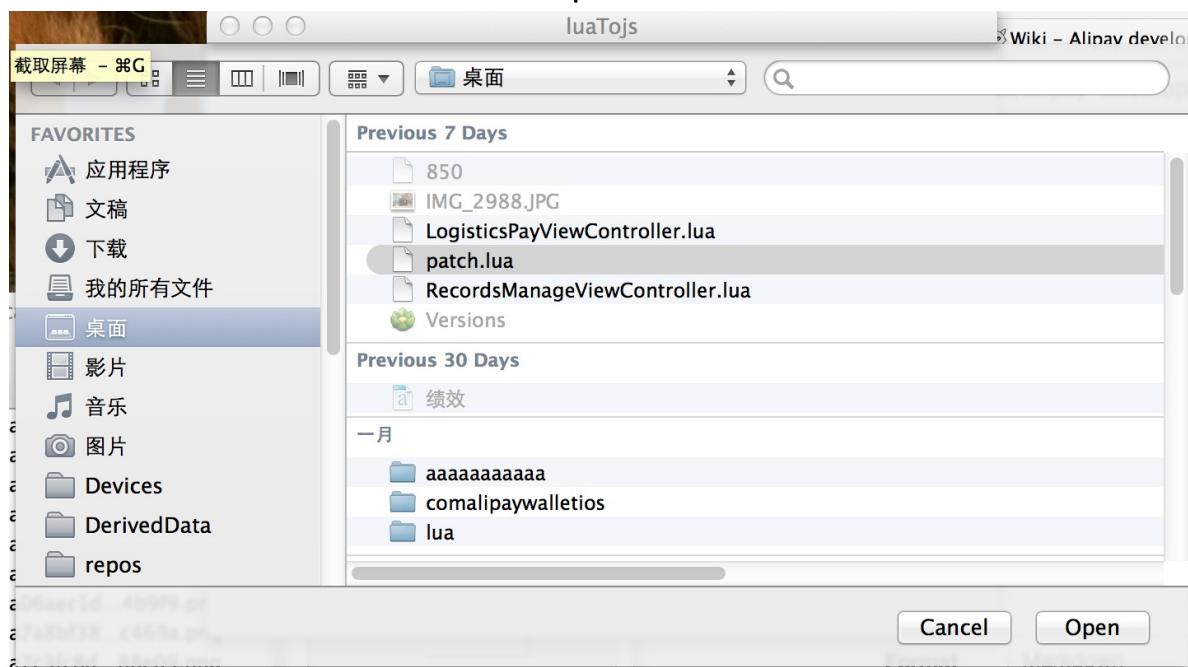
一、生成自己的 Lua2Js 工具

二、使用方法

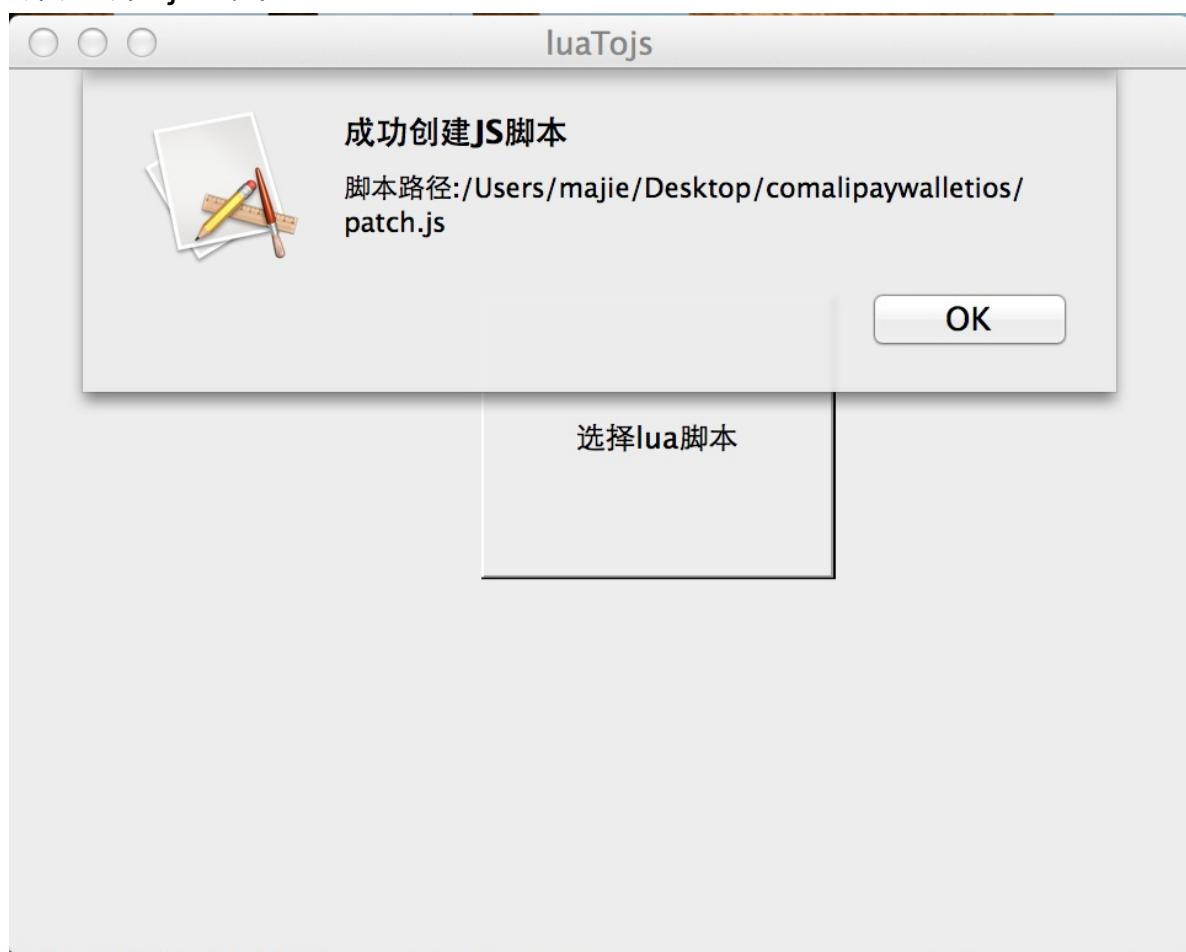
- 点击“选择 lua 脚本”按钮



- 选择需要转换的 lua 脚本，点击“open”按钮



- 成功转化 js 脚本



## 1.3 客户端接入 Hotpatch

### 1.3.1 管理对称加密 Key

实现 mPaaSAppInterface 的下面接口，使 Hotpatch 模块可以回调接入应用，获知使用配置在无线保镖中的哪个 key 进行加密处理。（需要应用将 key 与 secret 配置到无线保镖中管理）

```
#pragma mark Hotpatch

/**
 * 配置在无线保镖中的，用于 Hotpatch 脚本加密的 key 的名字。Hotpatch 模块
 *
 * @return 配置在无线保镖中的 key 的名字
 */
- (NSString*)appHotpatchScriptEncryptionKey;
```

### 1.3.2 管理非对称加密公钥文件

调用 APMobileSecurity 库的方法初始化自己的公钥文件，请在 main 函数中调用。

```
APSecBufferRef APSecGetPKS()
{
    char sig[] = {};

    size_t len = sizeof(sig);
    APSecBufferRef buf = (APSecBufferRef)malloc(sizeof(APSecBuffer) +
buf->length = len;
    memcpy(buf->data, sig, len);
    return buf;
}

int main(int argc, char *argv[])
{
    NSString *path;
    APSecBufferRef buf;
    int ret;

    path = [[NSBundle mainBundle] pathForResource:@"pubkey" ofType:@"";
APSecInitPublicKey([path UTF8String]);

    buf = APSecGetPKS();
    ret = APSecVerifyFile([path UTF8String], buf->data, buf->length);
    free(buf);
    if (ret != 0) {
        NSLog(@"The public key is modified.");
    }

    @autoreleasepool {
        mPaasInit(@"12345", [mPaasAppInterfaceImp class]);
        return UIApplicationMain(argc, argv, @"DFApplication", @"DFC1:");
    }
}
```

@cnName 2 配置服务 @priority 2

## 2 配置服务

[TOC]

### 2.1 简介

配置服务使客户端有能力拉取动态配置的参数，修改客户端行为。可用来进行活动运营、入口控制等功能。

配置服务需要应用后台提供相应的 RPC 接口，客户端在启动时会调用该 RPC 拉取开关配置字典。开发者可以在主站上进行开关的配置。（主站管理配置服务功能还在开发中，后期会对外开放）

### 2.2 使用方法

#### 2.2.1 获取配置服务实例

```
id<MPConfigService> configService = [mPaas() serviceWithName:@"ConfigS
```

建议 app 封装一个宏或方法，避免每次都强制类型转换。如果业务未勾选配置服务，会取到 nil。

#### 2.2.2 获取某个配置

```
id<MPConfigService> configService = [mPaas() serviceWithName:@"ConfigS
NSString* aConfig = [configService stringValueForKey:@"AConfigName"];
```

## 2.2.3 监听配置更新拉取成功的通知

```
extern NSString* const MPConfigServiceDidFetchedFromServerNotification
```

@cnName 3 红点提醒 @priority 3

## 3 红点提醒

[TOC]

### 3.1 简介

当客户端需要提醒用户关注某 UI 元素时，往往会在上面显示红点，红点会按照 UI 控件的级别进行累加。

移动红点提醒组件具有以下特点：

- 客户端在界面中使用的红点控件 `BadgeView` 具有唯一的红点标识，即 `widgetId` 属性；
- 红点控件支持丰富的显示样式：红点，数字，更新等，开发者也可以自定义红点控件；
- 客户端用红点管理器 `BadgeManager` 统一管理红点控件的显示，刷新，删除；
- 客户端在界面初始化的时候，向红点管理器注册红点控件；在销毁时，向红点管理器注销红点控件；
- 当红点管理器收到红点显示数据时，会自动刷新相应注册过的红点控件，不需要开发者额外代码控制 UI 元素；
- 红点组件之间有父子关系，父节点红点数是所有子节点红点数的和；当所有子节点红点消失时，父节点红点也不显示；

### 3.2 接口类定义

#### 3.2.1 MPBadgeManager

红点管理器，负责红点控件的注册，注销，刷新以及红点数据的管理。

一、 红点管理接口

```
/**  
 * 设置红点管理配置对象  
 *  
 * @param config 配置对象  
 *  
 * @return 无  
 */  
- (void)setBadgeServiceConfig:(id<MPBadgeServiceConfig>)config;  
  
/**  
 * 用户登录后，调用该接口处理红点的显示  
 *  
 * @param userId 当前登录用户 ID  
 *  
 * @return 无  
 */  
- (void)refreshAfterLogin:(NSString *)userId;  
  
/**  
 * 清除内存缓存的红点数据和界面上所有红点控件的显示  
 *  
 * @param 无  
 *  
 * @return 无  
 */  
- (void)clearAllBadges;
```

## 二、 红点控件使用接口

```
/**  
 * 注册红点控件到通用红点管理模块  
 *  
 * @param badgeView 红点控件  
 *  
 * @return 无  
 */  
- (void)registerBadgeView:(MPAbsBadgeView *)badgeView;  
  
/**  
 * 在通用红点管理模块中，注销掉红点控件  
 *  
 * @param badgeView 红点控件  
 *  
 * @return 无  
 */  
- (void)unregisterBadgeView:(MPAbsBadgeView *)badgeView;  
  
/**  
 * 如果是叶子节点的红点就消除红点，否则不处理。 (不用关心红点的类型)  
 *  
 * @param badgeView 红点控件  
 *  
 * @return 无  
 */  
- (void)tapBadgeView:(MPAbsBadgeView *)badgeView;  
  
/**  
 * 点击红点触发和该红点有关的显示处理 (不用关心红点的类型)  
 *  
 * @param widgetId 红点控件 Id  
 *  
 * @return 无  
 */  
- (void)tapBadgeViewWithWidgetId:(NSString *)widgetId;
```

### 三、 红点信息获取接口

```
/**  
 * 根据指定的红点路径 ID 获取红点信息  
 *  
 * @param badgeId 红点路径 ID, 如:"#M_PUBLIC_RD#201311230002181,#M_TRA  
 *  
 * @return 成功返回红点信息, 否则返回 nil  
 */  
- (MPBadgeInfo *)badgeInfoWithBadgeId:(NSString *)badgeId;  
  
/**  
 * 根据指定的业务 ID 获取相关的红点总数  
 *  
 * @param bizId 业务 ID  
 *  
 * @return 指定业务相关的所有红点数  
 */  
- (NSUInteger)badgeCountWithBizId:(NSString *)bizId;  
  
/**  
 * 根据指定的红点控件 ID 获取相关的红点总数  
 *  
 * @param widgetId 红点控件 ID  
 *  
 * @return 指定控件 ID 相关的所有红点数  
 */  
- (NSUInteger)badgeCountWithWidgetId:(NSString *)widgetId;
```

## 四、 红点数据注入接口

```
/**  
 * 全量方式：覆盖服务端返回的红点信息，并缓存  
 *  
 * @param badgeList MPBadgeInfo 数组  
 *  
 * @return 无  
 */  
- (void)updateRemoteBadgeInfo:(NSArray *)badgeList;  
  
/**  
 * 增量方式：插入（红点）服务端返回的红点信息，并缓存  
 *  
 * @param badgeList MPBadgeInfo 数组  
 *  
 * @return 无  
 */  
- (void)insertRemoteBadgeInfo:(NSArray *)badgeList;  
  
/**  
 * 增量方式：插入本地红点信息  
 * 注意： 1、不做本地缓存； 2、点击不上报服务器；  
 *  
 * @param badgeList 红点数据，元素是 MPBadgeInfo  
 *  
 * @return 无  
 */  
- (void)insertLocalBadgeInfo:(NSArray *)badgeList;
```

### 3.2.2 MPBadgeInfo

红点数据，定义红点样式，父子关系。

```

@property(nonatomic, readonly) NSString *badgeId; // 红点路径 ID
@property(nonatomic, copy) NSString *style; // 控件显示样式; 三
@property(nonatomic, assign) NSUInteger temporaryBadgeNumber; // 
@property(nonatomic, assign) NSUInteger persistentBadgeNumber; // 

/**
 * 创建红点数据
 *
 * @param badgeId 红点路径 ID
 * 注意: 若存在父子关系时, 用逗号分割, 如:"#M_PUBLIC_RD#2013112300
 *
 * @return 红点数据
 */
+ (MPBadgeInfo *)badgeInfoWithBadgeId:(NSString *)badgeId;

/**
 * 获取红点路径上的所有红点控件 ID
 *
 * @param 无
 *
 * @return 返回红点控件 ID
 */
- (NSArray *)allWidgetIds;

```

### 3.2.3 MPAbsBadgeView

#### 红点控件抽象基类

```

@property(nonatomic, copy) NSString *widgetId; // 红点控件 ID, 在 UITab
@property(nonatomic, copy) void (^ callback)(); // 提供业务监控红点控件
@property(nonatomic, strong) MPWidgetInfo *widgetInfo; // 红点控件信息
@property(nonatomic, weak) id<MPBadgeViewDelegate> delegate; // 刷新代

/**
 * 数字的计算方式

```

## Table of Contents |

```
* 注意： 1、叶子节点不要设置该属性；  
*       2、在 UITableViewCell 中使用时，要注意复用的问题，每次都要设置该属  
*  
* @"point" 只计算红点个数  
* @"new"    只计算 new 个数  
* @"num"    只计算数字个数  
* nil      默认计算方式，所有的个数  
*/  
@property(nonatomic, copy) NSString *numberCalculateMode;  
  
/**  
* 更新显示“红点”样式  
*  
* @param badgeValue: @"." 显示红点  
*                      @"new" 显示 new  
*                      @"数字" 显示数字，大于 99 都显示图片more (...)  
*                      @"惠" 显示“惠”字  
*                      nil 清除当前显示  
*  
* @return 无  
*/  
- (void)updateBadgeValue:(NSString *)badgeValue;  
  
/**  
* 绘制“红点”样式显示  
*  
* @param style:      @"." 显示红点  
*                      @"new" 显示 new  
*                      @"数字" 显示数字，大于 99 都显示图片more (...)  
*                      @"惠" 显示“惠”字  
*                      nil 清除当前显示  
*  
* @return 无  
*/  
- (void)drawBadgeStyle:(NSString *)style;
```

### 3.2.4 MPBadgeView

红点控件，MPAbsBadgeView 子类。

### 3.2.5 MPBadgeServiceConfig

红点管理服务配置接口

```
/**  
 * 点击上报  
 *  
 * @param widgetIds 上报数据  
 * @param completion 上报完成回调  
 *  
 * @return 无  
 */  
- (void)ack:(NSArray *)widgetIds completion:(void (^)(BOOL success))com  
  
/**  
 * 字符串加密  
 *  
 * @param string 待加密的字符串  
 *  
 * @return 加密后的字符串  
 */  
- (NSString *)encryptString:(NSString *)string;
```

@cnName 4 用户反馈 @priority 4

## 4 用户反馈

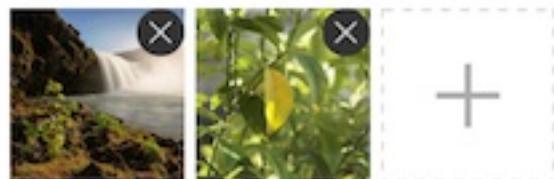
[TOC]

### 4.1 简介

移动框架里集成了一个简单实用的用户反馈界面，并集成了上传图片的功能。同时公开了用户反馈的接口，接入方可以重写自己的 UI 界面。



Test



236



## 4.2 使用方法

用户反馈组件为 MPFeedbackKit。当使用移动框架提供的反馈界面时，可以在接入应用的 MobileRuntime.plist 文件中将用户反馈配置为一个微应用，如图：

MobileFramework > MobileFramework > Supporting Files > MobileRuntime.plist		
Key	Type	Value
Root	Dictionary	(3 items)
Applications	Array	(18 items)
Item 0	Dictionary	(3 items)
delegate	String	MPFeedbackAppDelegate
description	String	用户反馈
name	String	20000018
Item 1	Dictionary	(3 items)
Item 2	Dictionary	(3 items)

应用的名字可以由接入方自己定，只要是唯一的子应用 ID 即可。然后在代码中使用下面代码就可以唤起用户反馈界面：

```
[DTContextGet() startApplication:@"20000018" params:@{@"userId":@"111":}
```

其中 `userId` 和 `mobileNo` 是可选参数，为第三方应用当前登录用户的信息，方便收到反馈后联系用户，可以不设置。

@cnName 5 文件上传下载 @priority 5

# 5 文件上传下载

[TOC]

## 5.1 简介

移动框架提供文件的上传下载功能，实现静态资源的上传和下载，同时在服务器端的配合下做到断点续传、分片上传功能。（目前分片与断点功能暂未开放）

## 5.2 上传

### 5.2.1 启动一个上传任务

```
NSString* fullPathName = @"...";
NSString* uploadUrl = @"...";
APFileTransferCenter* tcenter = [APFileTransferCenter sharedInstance];
NSInteger taskTag = [tcenter addUpTask:uploadUrl file:fullPathName delegate:self];
if(taskTag > 0)
{
    NSLog(@"UPLOAD[%d] start success", (int)taskTag);
}
else
{
    NSLog(@"UPLOAD[%d] start fail", (int)taskTag);
}
```

- 可在任意线程创建任务
- 参数 `fullPathName` 必须是带有完整路径的文件名
- 参数 `uploadUrl` 是带有 `http` 或 `https` 格式的完整 url 地址

- 参数 `delegate` 接收上传任务进度，成功或失败的通知（由主线程通知）
- 返回值用来唯一标识一个上传任务的状态，大于 0 为上传成功，否则表示上传失败

## 5.2.2 APFileUpTaskDelegate

```
@protocol APFileUpTaskDelegate <NSObject>
- (void)APFileUpTaskDidSuccessed:(APFileUpTaskInfo*)info;
- (void)APFileUpTaskDidFailed:(APFileUpTaskInfo*)info;
- (void)APFileUpTaskDidProgressUpdated:(APFileUpTaskInfo*)info;
@end
```

## 5.3 下载

### 5.3.1 启动一个下载任务

```
NSString* fullPathName = @"...";
NSString* downloadUrl = @"...";
APFileTransferCenter* tcenter = [APFileTransferCenter sharedInstance];
NSInteger taskTag = [tcenter addDownTask:downloadUrl file:fullPathName];
if(taskTag > 0)
{
    NSLog(@"DOWNLOAD[%d] start success", (int)taskTag);
}
else
{
    NSLog(@"DOWNLOAD[%d] start fail", (int)taskTag);
}
```

- 创建任务可任意线程调用
- 参数 `fullPathName` 必须是带有完整路径的文件名
- 参数 `downloadUrl` 是带有 `http` 或 `https` 格式的完整 `url` 地址

- 参数 `delegate` 接收下载任务进度，成功或失败的通知（由主线程通知）
- 返回值用来唯一标识一个下载任务，大于 0 为下载成功，否则表示下载失败

## 5.3.2 APFileDownTaskDelegate

```
@protocol APFileDownTaskDelegate <NSObject>
- (void)APFileDownTaskDidSuccesssed:(APFileDownTaskInfo*)info;
- (void)APFileDownTaskDidFailed:(APFileDownTaskInfo*)info;
- (void)APFileDownTaskDidProgressUpdated:(APFileDownTaskInfo*)info;
@end
```

## @cnName 6 检查更新 @priority 6

# 6 检查更新

## 6.1 简介

移动主站提供在线打包功能，打包完成后可以选择发布该版本的包。此时线上便可以通过更新接口检测到该版本，提醒用户下载更新。同时发布的包可以设置更新提示、是否强制更新等选项。



注：App Store 条例里已经不允许上线应用内置升级检测功能。所以检测更新功能可以用来开发阶段内测使用。

## 6.2 使用方法

提供检查更新的 RPC 接口文件，应用集成到自己的代码中自行调用。调用样例：

```
- (void)checkUpdate:(void (^)(MPAAS_CheckUpgradeResp* resp))callback
{
    __block MPAAS_CheckUpgradeResp *result = nil;
    [DTRpcAsyncCaller callAsyncBlock:^{
        MPAAS_CheckUpgradeServiceFacade* facade = [[MPAAS_CheckUpgradeServiceFacade alloc] init];
        MPAAS_CheckUpgradeReq* request = [[MPAAS_CheckUpgradeReq alloc] init];
        request.appkey = [mPaas() appKey];
        request.version = [[NSBundle mainBundle] objectForInfoDictionaryKey:@"CFBundleVersion"];
        request.systemPlatform = @"ios";
        @try
        {
            result = [facade checkUpdate:request];
        }
        @catch(DTRpcException *exception)
        {
            NSLog(@"%@", exception);
        }
    } completion:^{
        if (result)
        {
            callback(result);
        }
    }];
}
```

检测更新协议文件与调用代码

@cnName 1 概述 @priority 1

# 1 概述

## 1.1 使用统一存储的目的

- 减少 `NSUserDefaults` 的使用，不将较大数据和有隐私性数据存储在 `NSUserDefaults` 里，存取效率相对使用 `NSUserDefaults` 有大幅提升。
- 减少业务自动维护文件的情况，减少 `Documents`、`Library` 目录下的杂乱文件。
- 统一存储分按存储空间划分为：与用户无关的空间，当前用户的存储空间。业务层无需关注用户切换，并且不需要使用 `userId` 来获取当前用户数据。
- 基于 `sqlite`，提供 DAO 支持，相比 `CoreData` 更加灵活。通过配置文件将数据库操作封装起来并与业务隔离，业务层使用接口存取数据，操作数据库表。
- 底层提供数据加密支持。
- 提供多样化的存储方式，满足不同需求，并提供高效的内存缓存。

## 1.2 统一存储公开类说明

类名	功能
APDataCenter	单例类，统一存储入口类
APSharedPreferences	对应一个数据库文件，提供 Key-Value 存储接口，同时容纳 DAO 建表。
APDataCrypt	对称加密结构体
APLRUDiskCache	支持 LRU 淘汰规则的磁盘缓存
APLRUMemoryCache	支持 LRU 淘汰规则的内存缓存，线程安全。
APObjectArrayService	基于 DAO，可以分业务对支持 NSCoding 的对象提供持久化，支持加密、容量限制与内存缓存。
APAsyncFileArrayService	基于 DAO，对二进制数据提供持久化，支持加密、容量限制与内存缓存。
APCustomStorage	自定义存储空间，同时在这个空间内提供完整的用户管理，Key-Value、DAO 存储功能。
APDAOProtocol	接口描述，为 DAO 对象支持的接口。

@cnName 2.1 APDataCenter @priority 1

## 2.1 APDataCenter

[TOC]

### 2.1.1 简介

APDataCenter 为统一存储的入口类，为一个单例，可在代码任意地方调用

```
[APDataCenter defaultDataCenter]
```

也可以使用宏

```
#define APDefaultDataCenter [APDataCenter defaultDataCenter]
```

即会初始化 APDataCenter。

### 2.1.2 接口介绍

#### 宏定义

```
#define APDefaultDataCenter [APDataCenter defaultDataCenter]
#define APCommonPreferences [APDefaultDataCenter commonPreferences]
#define APUserPreferences [APDefaultDataCenter userPreferences]
#define APCurrentVersionStorage [APDefaultDataCenter currentVersionSto
```

#### 常量定义

这几个事件通知业务代码通常无须关注，但是统一存储会抛出这些通知。

```

/**
 * 前一个用户的数据库文件将要关闭的事件通知
 */
extern NSString* const kAPDataCenterWillLastUserResign;

/**
 * 用户状态已经发生切换的通知。有可能是 user 变为 nil 了，具体 userId 可以
 * 这个通知附加的 object 是个字典，如果不为 nil，里面@"switched"这个键值
 */
extern NSString* const kAPDataCenterDidUserUpdated;

/**
 * 用户并没切换，APDataCenter 重新收到登入事件。会抛这个通知。
 */
extern NSString* const kAPDataCenterDidUserRenew;

```

### 2.1.3 接口与属性

**void APDataCenterLogSwitch(BOOL on);**

打开或关闭统一存储的控制台 log，默认为打开。

**@property (atomic, strong, readonly) NSString\* currentUserId;**

当前登录用户的 userId

**+ (NSString\*)preferencesRootPath;**

得到存储commonPreferences 和 userPreferences 数据库文件夹的路径。

**- (void)setCurrentUserId:(NSString\*)currentUserId;**

设置当前登录的用户 Id，业务代码请不要调用，需要由登录模块调用。设置用户 ID )

**- (void)reset;**

完全重置整个统一存储目录，请谨慎。

**- (APSharedPreferences\*)commonPreferences;**

与用户无关的全局存储数据库

**- (APSharedPreferences\*)userPreferences;**

当前登录用户的存储数据库。不是登录态时，取到的是 nil。

**- (APSharedPreferences)preferencesForUser:(NSString)userId;**

返回指定用户 id 的存储对象，业务层通常使用 userPreferences 方法即可。当有异

**- (APPREFERENCESaccessor)accessorForBusiness:(NSString)business;**

根据 business 名生成一个存取器，业务层需要自行持有这个对象。使用这个存取器后

```
APPREFERENCESACCESSOR* accessor = [[APDATACENTER DEFAULTDATACENTER] ACCESSOR];  
[[accessor commonPreferences] doubleForKey:@"aKey"];
```

// 等价于

```
[[[[APDATACENTER DEFAULTDATACENTER] COMMONPREFERENCES] DOUBLEFORKEY:@"aKey"]]
```

## - (APCustomStorage\*)currentVersionStorage;

统一存储会维护一个当前版本的数据库，当版本发生升级时，这个数据库会重置。

## - (id<APDAOProtocol>)daoWithPath:(NSString\*)filePath userDependent:(BOOL)userDependent;

从一个配置文件生成 DAO 访问对象

@param filePath DAO 配置文件的文件路径，在 main bundle 里的文件使用下面方

法获得： NSString\* filePath = [[NSBundle mainBundle] pathForResource:@"file" ofType:@"plist"];

@param userDependent 指定这个 DAO 对象操作哪个数据库。如果 userDependent 为

@return 返回 DAO 对象，业务不用关心它的类名，只需要使用业务自己定义的 id<APDAOProtocol> 接口即可。

## - (id<APDAOProtocol>)daoWithPath:(NSString)filePath databasePath:(NSString)databasePath;

创建一个维护自己独立数据库文件的 DAO 访问对象，而不使用 APSharedPreference 使用 `daoWithPath:userDependent:` 接口创建的 DAO 对象，操作的是 `commonPreference` 这个接口会创建一个 DAO 对象，并且操作的是 `databasePath` 指定的特定数据库文件。可以创建多个 DAO 对象，指定相同的 `databasePath`。

`@param filePath` 同 `daoWithPath:userDependent:` 接口  
`@param databasePath` DAO 数据库文件的位置，可以传绝对路径，也可以传 'Document'。  
`@return` DAO 对象

@cnName 2.2 Key-Value 存储 @priority 2

## 2.2 Key-Value 存储

[TOC]

### 2.2.1 简介

客户端许多场景使用 Key-Value 存储就能很好的满足需求，通常我们会使用 NSUserDefaults，但 NSUserDefaults 不支持加密，持久化速度慢。

统一存储Key-Value 存储提供接口存储： NSInteger, long long ( 64 位上与 NSInteger 相同) , BOOL, double, NSString 等 PList 对象，支持 NSCoding 的对象，可通过反射转换成 JSON 表达的 Objective-C 对象，极大简化客户端持久化对象的复杂度。

关于Key-Value 存储中大部分接口说明，请开发者参考 APSharedPreferences.h 头文件方法描述。

### 2.2.2 存储基本类型

统一存储提供下列接口存储基本类型。

```
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business;
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business defaultValue:(NSInteger)defaultValue;
- (void)setInteger:(NSInteger)value forKey:(NSString*)key business:(NSString*)business;

- (long long)longLongForKey:(NSString*)key business:(NSString*)business;
- (long long)longLongForKey:(NSString*)key business:(NSString*)business defaultValue:(long long)defaultValue;
- (void)setLongLong:(long long)value forKey:(NSString*)key business:(NSString*)business;

- (BOOL)boolForKey:(NSString*)key business:(NSString*)business;
- (BOOL)boolForKey:(NSString*)key business:(NSString*)business defaultValue:(BOOL)defaultValue;
- (void)setBool:(BOOL)value forKey:(NSString*)key business:(NSString*)business;

- (double)doubleForKey:(NSString*)key business:(NSString*)business;
- (double)doubleForKey:(NSString*)key business:(NSString*)business defaultValue:(double)defaultValue;
- (void)setDouble:(double)value forKey:(NSString*)key business:(NSString*)business;
```

其中 `defaultValue` 参数为数据不存在时返回的默认值。

## 2.2.3 存储Objective-C 对象

### 接口说明

统一存储提供下列接口存储Objective-C 对象。

```
- (NSString*)stringForKey:(NSString*)key business:(NSString*)business;
- (NSString*)stringForKey:(NSString*)key business:(NSString*)business extens:
- (void)setString:(NSString*)string forKey:(NSString*)key business:(NS
- (void)setString:(NSString*)string forKey:(NSString*)key business:(NS

- (id)objectForKey:(NSString*)key business:(NSString*)business;
- (id)objectForKey:(NSString*)key business:(NSString*)business extens:

- (void)setObject:(id)object forKey:(NSString*)key business:(NSString
- (void)setObject:(id)object forKey:(NSString*)key business:(NSString

- (void)archiveObject:(id)object forKey:(NSString*)key business:(NSSti
- (void)archiveObject:(id)object forKey:(NSString*)key business:(NSSti

- (void)saveJsonObject:(id)object forKey:(NSString*)key business:(NSS
- (void)saveJsonObject:(id)object forKey:(NSString*)key business:(NSS
```

## **setString & stringForKey**

保存 NSString 时，推荐使用 setString, stringForKey 接口，名称上更有解释性。

如果数据未加密，使用这个接口存储的字符串，可以通过 sqlite db 查看器看到，更直观。使用 setObject 保存的字符串会首先通过 Property List 转成 NSData 再保存到数据库里。

## **setObject**

保存 Property List 对象时，建议使用 setObject，这样效率最高。

Property List 对象：NSNumber、NSString、NSData、NSDate、  
NSArray、NSDictionary，并且 NSArray 和 NSDictionary 里的子对象也只能是 PList 对象。

使用 setObject 保存 Property List 后，使用 objectForKey 取到的对象是 Mutable 的。下面代码里拿到的 savedArray 是 NSMutableArray。

```
NSArray* array = [[NSArray alloc] initWithObjects:@"str", nil];
[APCommonPreferences setObject:array forKey:@"array" business:@"biz"].

NSArray* savedArray = [APCommonPreferences objectForKey:@"array" busi
```

## **archiveObject**

对于支持 NSCoding 协议的 Objective-C 对象，统一存储调用系统的 NSKeyedArchiver 将对象转成 NSData 并进行持久化。

Property List 对象也可以使用这个接口，但效率比较低，不推荐。

## **saveJsonObject**

当一个 Objective-C 对象既不是 Property List 对象，也不支持 NSCoding 协议时，可以使用这个方法进行持久化。

这个方法通过运行时动态反射，将 Objective-C 对象映射成 JSON 字符串。但不是所有 Objective-C 对象都可以使用这个方法保存，比如含有 C 结构体指针的属性，互相引用的 Objective-C 对象，对象属性里有字典或数组。

## **objectForKey**

统一存储保存 Objective-C 对象数据时，会同时记录它的归档方式，取对象统一使用 objectForKey。注意：使用 `setString` 保存的字符串需要使用 `stringForKey` 获取。

## **数据加密**

### **使用默认加密**

带 extension 的接口支持加密，传入 APDataCrypt 结构体。

`APDefaultEncrypt` 为默认加密方法，为 AES 对称加密。

`APDefaultDecrypt` 为默认解密方法，与 `APDefaultEncrypt` 使用相同密钥。

通常情况下使用统一存储提供的默认加密即可，如下：

```
[APUserPreferences setObject:aObject forKey:@"key" business:@"biz" exten:  
id obj = [APUserPreferences objectForKey:@"key" business:@"biz" exten:  
// or  
id obj = [APUserPreferences objectForKey:@"key" business:@"biz"];
```

因为使用的是默认加密，所以取数据的接口可以省略 extension 参数。

## 使用自定义加密方法

如果业务有更高的加密安全要求，可以自己实现 APDataCrypt 结构体，并指定加密、解密的函数指针。但请保证加密与解密方法对应，这样才能正确保存、还原数据。

### 基础类型加密

如果想加密存储BOOL, NSInteger, double, long long, 可以把它们转成字符串，或放到 NSNumber 里，再调用 `setString`、`setObject` 接口即可。

@cnName 2.3.1 概述 @priority 1

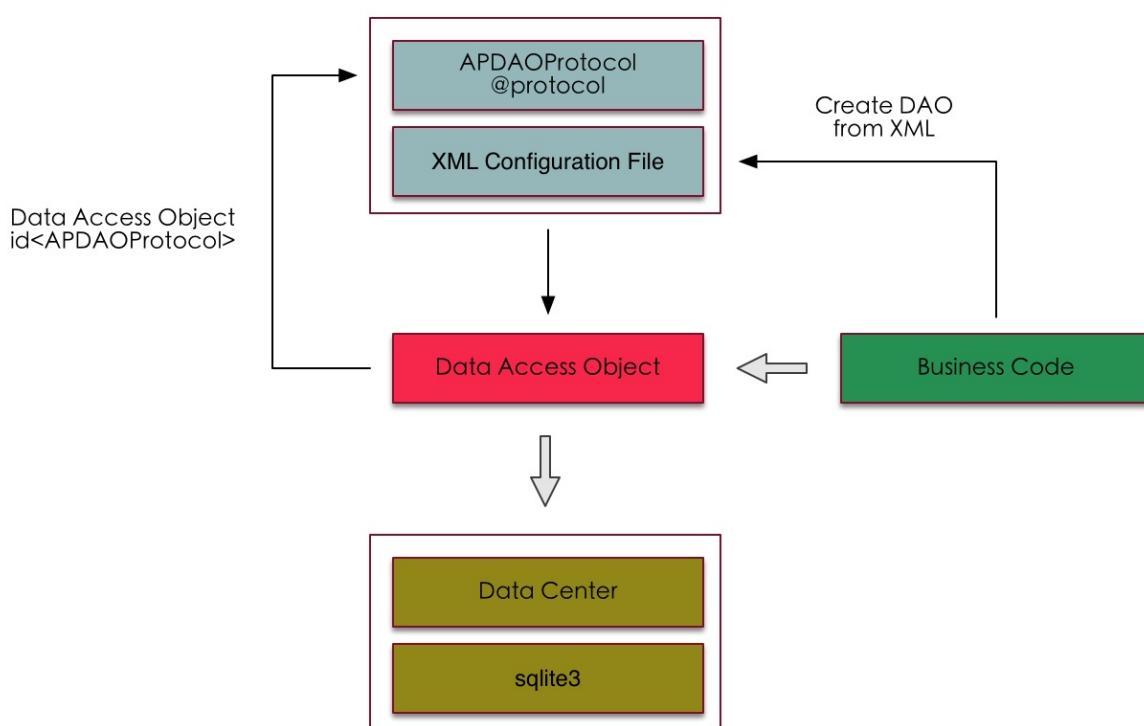
## 2.3.1 概述

[TOC]

### DAO 使用场景

普通 KV 存储只能存储简单数据类型，或封装好的 OC 对象，也不支持搜索。当业务有 sqlite 访问需要时，可由统一存储的 DAO 功能进行简化和封装。

### DAO 基本工作原理



- 定义一个 xml 格式的配置文件，描述各 sqlite 操作的函数，返回的数据类型，需要加密的字段等。
- 定义一个 DAO 对象的接口 DAOInterface（@protocol），接口方法名、参数与配置文件里的描述一致。
- 业务将 xml 配置文件传给 APDataCenter 的 daoWithPath 方法，生成 DAO 访问对象。该对象直接强转为 id

- 接下来业务就可以直接调用 DAO 对象的方法，统一存储会将该方法转换为配置文件里描述的数据库操作。

## 一个最简单的例子

- 配置文件第一行定义了默认表名和数据库版本，以及初始化方法。
- insertItem 和 getItem 是插入和读取数据的两个方法，接收参数并格式化到 sql 表达式里。
- createTable 方法会在底层被默认调用一次。

```

<module name="Demo" initializer="createTable" tableName="demoTable">
  <update id="createTable">
    create table if not exists ${T} (index integer primary key, content string)
  </update>

  <insert id="insertItem" arguments="content">
    insert into ${T} (content) values(${content})
  </insert>

  <select id="getItem" arguments="index" result="string">
    select * from ${T} where index = ${index}
  </select>
</module>

```

- DAO 接口定义

```

@protocol DemoProtocol <APDAOProtocol>
- (APDAOResult*)insertItem:(NSString*)content;
- (NSString*)getItem:(NSNumber*)index;
@end

```

- 创建 DAO 代理对象，假设配置文件叫 demo\_config.xml，在 Main Bundle 里。
- 用 insertItem 方法写入一个数据，获取它的索引，再用该索引把写入的数据读出来。

```
NSString* filePath = [[NSBundle mainBundle] pathForResource:@"demo"
id<DemoProtocol> proxy = [[APDataCenter defaultCenter] daoWithProtocol];
[proxy insertItem:@"something"];
long long lastIndex = [proxy lastInsertRowId];
NSString* content = [proxy getItem:[NSNumber numberWithInt:lastIndex]];
 NSLog(@"%@", content);
```

- 其中 lastInsertRowId 是 APDAOProtocol 的一个方法，用来取最后插入的行的 rowId。想让 DAO 对象支持该方法，只要在声明 DemoProtocol 时继承自 APDAOProtocol 即可。

## @cnName 2.3.2 关键字 @priority 2

## 2.3.2 关键字

[TOC]

### module

```
<module name="demo" initializer="createTable" tableName="tableDemo" v>
```

- **initializer** 参数可选，对于 initializer 指定的 update 方法，DAO 认为是数据库建表方法，会在第一次 DAO 请求时默认执行一次；
- **tableName** 指定下面方法里默认操作的表名，在 sql 语句里可以用\${T} 或 \${t} 代替，不用每次都写表名了。建议每个配置文件只操作一张表。**tableName** 可空，应对同一配置文件操作相同格式的多张表的情况。比如聊天消息分表处理，可以调用 DAO 对象的 **setTableName** 方法设置需要操作的表名。
- **version** 是配置文件的版本号，请使用'x.x'的格式，创建 table 后，会以 **tableName** 作为 key，把表的版本存到数据库文件的 **TableVersions** 表里，配合 **upgrade** 块进行表的更新。
- **resetOnUpgrade**，如果为 true 或 YES，当 **version** 更新后，会删除原表，而不是调用 **ungrade** 块。无此参数为默认为 false。
- **upgradeMinVersion**，如果不为空，对于小于这个版本的数据库文件，直接重置，否则执行升级操作。

### const

```
<const table_columns="(id, time, content, uin, read)" />
```

- 定义一个字符串类型的常量，**table\_columns** 是常量的名字，等号后面的是常量值，在配置文件里可以使用\${常量名}来引用。

## select

```
<select id="find" arguments="id, time" result=":messageModelMap">
    select * from ${T} where id = #{id} and time > #{@{time}}
</select>
```

### @arguments

- 参数名的列表，用','分隔。调用者传进来的参数依赖 `arguments` 里的描述依次命名。DAO 对象的 `selector` 调用时不会携带参数名，所以必须在这里按顺序命名。
- 参数名前面有\$符号时，表示这个参数不接受 nil 值。业务的调用 DAO 接口，是允许传 nil 参数的，如果某个参数前面有\$符号，当调用者不小心传入了 nil 值，DAO 调用会自动失败。防止发生不可预知的问题。
- 关于如何引用参数，参考[DAO 引用方式](#)。

比如上面这句，对应的 `selector` 为：

```
- (MessageModel*)find:(NSNumber*)id time:(NSNumber*)time;
```

如果 DAO 对象调用[`daoProxy find:@1234 time:@2014`]，那么 sql 语句拼好后是：

```
select * from tableDemo where id = ? and time > 2014
```

并且@1234 这个 NSNumber 会交给 sqlite 绑定。

### @result

- `result` 里可以填写 DAO 方法的返回值，用[]括起来表示返回数组类型，会对 `select` 执行的返回一直进行迭代，直到 sqlite 无结果返回。如果不用[]括起来，表示只返回一个结果，对 `select` 执行的返回只迭代一次，类似 FMDB 库里 `FMResultSet` 只调用一次 `next` 方法。

### 返回类型

- int: 只有一个结果，返回[NSNumber numberWithInt]类型
- long long: 只有一个结果，返回[NSNumber numberWithLongLong]类型
- bool: 只有一个结果，返回[NSNumber numberWithBool]类型
- double: 只有一个结果，返回[NSNumber numberWithDouble]类型
- string: 只有一个结果，返回 NSString\*类型
- binary: 只有一个结果，返回 NSData\*类型
- [int]: 返回数组，数组里为[NSNumber numberWithInt]
- [long long]: 返回数组，数组里为[NSNumber numberWithLongLong]
- [bool]: 返回数组，数组里为[NSNumber numberWithBool]
- [double]: 返回数组，数组里为[NSNumber numberWithDouble]
- [string]: 返回数组，数组里为 NSString\*
- [binary]: 返回数组，数组里为 NSData\*
- [{}]: 返回数组，数组里是列名->列值的 map
- [AType]: 返回数组，数组里是填好的自定义类
- {}: 只有一个结果，返回列名->列值的 map
- AType: 只有一个结果，返回填好的自定义类
- [:AMap]: 返回数组，数组里是使用 xml 里定义的 AMap 映射出来的对象
- :AMap: 只有一个结果，使用配置文件里定义的 AMap 来描述对象

比如上面的例子，返回类型为":messageModelMap"。具体返回的 Objective-C 类型，以及需要特殊映射的列都会在 messageModelMap 里定义。参考 map 关键字。

@foreach select 也支持 foreach 字段，用法下文介绍的 insert、update、delete 里的相似。不同的是，select 方法如果指定了 foreach 参数，那么会执行 N 次 SQL 的 select 操作，并把结果放到数组里返回。所以如果 DAO 的 select 方法是 foreach 的，它的返回值在 protocol 里一定要定义成 NSArray\*

## insert, update, delete

```

<insert id="addMessages" arguments="messages" foreach="messages.model"
        insert or replace into ${T} (id, content, uin, read) values(${model})
</insert>

<update id="createTable">
    <step>
        create table if not exists ${T} (id integer primary key, content +
    </step>
    <step>
        create index if not exists uin_idx on ${T} (uin)
    </step>
</update>

<delete id="remove" arguments="msgId">
    delete from ${T} where msgId = #{msgId}
</delete>

```

- `insert`、`update`、`delete` 方法格式相同，参数的拼接与引用和 `select` 方法相同。
- `insert` 和 `delete` 关键字是为了更好的区分方法用途。你完全可以把一个 `delete from table` 的操作写在 `update` 函数中。
- 在 DAO 接口中，需要执行 `insert`、`update` 或 `delete` 的方法，返回值为 `APDAOResult*`，标识执行是否成功。参考[DAO 最简单例子](#)

### @foreach

- 当 `insert`、`update`、`delete` 方法后面有'foreach'字段时，这个方法被调用时，会依次对参数数组里的每个元素执行一次 sql。
- 要求 `foreach` 字段遵循格式： `collectionName.item`。其中 `collectionName` 必须对应 `arguments` 里的一个参数，指定循环的容器对象，并且调用时必须为一个 `NSArray` 或 `NSSet` 类型； `item` 表示从容器里取出的对象，用作循环变量。不能和 `arguments` 里的任何参数重名，这个 `item` 可以当作一个普通参数用在 sql 语句里。

比如代理方法为

```
- (void)addMessages:(NSArray*)messages;
```

messages 为 MessageModel 数组，那么对于 messages 里的每个 model，都会执行一次 sql 调用，这样就能实现把数组里的元素一次性插入数据库而上层不需要关心循环的调用。底层会把这次操作合并为一个事务而提升效率。

## @step

- 在 insert, update, delete 方法里，可能遇到这种情况：执行一个 DAO 方法，但是需要调用多次 sql update 操作执行多条 sql 语句。比如用户建表后，又要创建一些索引。在函数里包裹的语句，都会当作一次 sql update 被单独执行，底层会把所有操作合并为一次事务。比如上图的 createTable 方法。
- 如果一个函数里面有 step 了，那么在 step 外面不允许有文本了。step 内不能再含其它 step。

## map

```
<map id="messageModelMap" result="MessageModel">
    <result property="msgId" column="id"/>
</map>
```

- 定义一个名为 messageModelMap 的映射，实际生成的 Objective-C 对象是 MessageModel 类。
- Objective-C 对象的 msgId 属性，对应表里名为 id 的列值；未列的属性认为和表的列名相同，可以省略。

## upgrade

```
<upgrade toVersion="1.1">
  <step>
    alter table...
  </step>
  <step>
    alter table...
  </step>
</upgrade>
```

- 随着版本升级，数据库可能有升级需求；处理升级的 sql 语句写在这里。比如最初，配置文件 module 的版本是 1.0，升级后，配置文件的版本为 1.1。那么新版本第一次运行 DAO 方法时，会检测当前表的版本与配置文件的版本，当发现不一致时，会依次执行升级方法。这个方法会由底层自动调用，并在升级完成后再执行 DAO 方法。
- upgrade 按照 sql update 来执行，如果 sql 不止一个，可以用括起来，类似的实现。
- 如果需要使用 upgrade 块定义的操作来升级表，那么 module 里的 resetOnUpgrade 必须设置为 false。

## crypt

```
<crypt class="MessageModel" property="content"/>
```

- 描述 class 这个类的 property 属性进行加密处理，当向数据库写入时从这个属性里取出的值会进行加密；当数据库读出时，生成对象向这个属性里赋值会先解密再赋值。

比如执行 DAO 方法，model 是 MessageModel 类。因为取了 model 的 content 属性，所以会加密后再写入数据库。

```
<insert id="insertMessage" arguments="model">
  insert into ${T} (content) values(${model.content})
</insert>
```

执行这个 `select` 方法时，返回对象是 `MessageModel` 类。底层从数据库里取出数据向 `MessageModel` 的实例写入 `content` 属性时，会将数据先解密再写入。最后返回处理好的 `MessageModel` 对象。

```
<select id="getMessage" arguments="msgId" result="MessageModel">
    select * from ${T} where msgId = #{msgId}
</select>
```

加密方式的设置方法定义在[APDAOProtocol](#)中，如下

```
/**
 * 设置加密器，用于加密表里标记为需要加密的列的数据。向表里写数据时，碰到这
 *
 * @param crypt 加密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，
 */
- (void)setEncryptMethod:(APDataCrypt*)crypt;

/**
 * 设置解密器，用于解密表里标记为需要加密的列的数据。从表里读数据时，碰到这
 *
 * @param crypt 解密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，
 */
- (void)setDecryptMethod:(APDataCrypt*)crypt;
```

如果不进行设置，会使用 `APDataCenter` 的默认加密，见[KV 存储](#)。如果一个 DAO 代理对象是 `id`，并且 `DAOProtocol` 是`@protocol`，那么可以直接用 DAO 代理对象调用 `setEncryptMethod` 和 `setDecryptMethod` 来设置加密、解密方法。

**if**

```

<insert id="addMessages" arguments="messages, onlyRead" foreach="messag
  <if true="model.read or (not onlyRead)">
    insert or replace into ${T} (msgId, content, read) values(${model
  </if>
</insert>

```

- 在 `insert`、`update`、`delete`、`select` 方法里，可以嵌套使用 `if` 条件判断语句。当 `if` 条件满足时，会把 `if` 块内的文本拼接到最终的 `sql` 语句里。
- `if` 后可以接 `true="expr"`，也可以接 `flase="expr"`； `expr` 为表达式，可以使用方法的参数，并且可以使用`".`来链式访问参数对象的属性。
- 表达式支持的运算符如下：

`()`: 括号

`+`: 正号

`-`: 负号

`+`: 加号

`-`: 减号

`*`: 乘号

`/`: 除号

`\`: 整除

`%`: 取模

`>`: 大于

`<`: 小于

`>=`: 大于等于

`<=`: 小于等于

`==`: 等于

**!=**: 不等于

**and**: 逻辑与，不区分大小写

**or**: 逻辑或，不区分大小写

**not**: 逻辑非，不区分大小写

**xor**: 异或，不区分大小写

- 大于号、小于号字符需要使用转义。参考 <http://lidongbest5.com/blog/5/>
- 里面的参数就是外部调用传入的参数名，但是不要像 sql 块里使用#{}或@{}包裹；
- nil 含义同Objective-C 里的 nil；
- 表达式里的字符串使用单引号起止，不支持转义字符，但是支持\'代表一个单引号；
- 当参数为 Objective-C 对象时，支持用'.'来访问它的属性，比如上面例子 model.read，如果参数是数组或字典，可以用'参数名.count'取元素数。

一个较复杂的表达式如下：

```
<if true="(title != nil and year > 2010) or authors.count >= 2">
```

title, year, authors 都是调用者传来的参数，调用时 title 是可以传 nil 的；那么上面含义为"当书名不为空，并且出版年份大于 2010 年，或作者数大于 2"

## choose

```

<choose>
  <when true="title != nil">
    AND title like #{title}
  </when>
  <when true="author != nil and author.name != nil">
    AND author_name like #{author.name}
  </when>
  <otherwise>
    AND featured = 1
  </otherwise>
</choose>

```

- 实现类似 switch 的语法，表达式要求类似 if 语句； when 后面也可以接 true="expr" 或 false="expr"。
- 只会执行第一个符合条件的 when 或者 otherwise；可以没有 otherwise。

## foreach

```

<foreach item="iterator" collection="list" open="(" separator="," close=")">
  @{@{iterator}}
</foreach>

```

- open, separator, close, reverse 可以省略。
- item 表示循环变量， collection 表示循环数组参数名。

比如一个方法从外部接收字符串数组参数， list 内容为@[@"1", @"2", @"3"]，有另一个参数是 prefix=@"abc"，使用'()'包裹，','分隔。那么执行结果为：(abc1,abc2,abc3)

```

<update id="proc" arguments="list, prefix">
    <foreach item="iterator" collection="list" open="(" separator="," close=")">
        #{prefix}{iterator}
    </foreach>
</update>

```

`foreach` 语句通常用于拼接 `select` 语句里的 `in` 块，比如：

```

select * from ${T} where id in
<foreach item="id" collection="idList" open="(" separator="," close=")">
    #{id}
</foreach>

```

## where, set, trim

```

<where onVoid="quit">
    <if true="state != nil">
        state = #{state}
    </if>
    <if true="title != nil">
        AND title like #{title}
    </if>
    <if true="author != nil and author.name != nil">
        AND author_name like #{author.name}
    </if>
</where>

```

`where` 会处理多余的 `AND`、`OR`（大小写无所谓），并在任何条件都不符合时连`where` 都不返回。用于在 `sql` 语句里拼接有大量判断条件的 `where` 子句。比如上例，如果只有最后一个判断成立，该语句会正确返回 `where author_name like XXX`，而不是 `where AND author_name like XXX`。

```
<set>
  <if false="username != nil">username=#{username},</if>
  <if false="password != nil">password=#{password},</if>
  <if true="email != nil">email=#{email},</if>
  <if true="bio != nil">bio=#{bio},</if>
</set>
```

`set` 会处理结尾多余的'，' 并在任何条件都不符合时什么都不返回。与 `where` 语句类似，只是它处理的是后缀的逗号。

```
<trim prefix="WHERE" prefixOverrides="AND | OR | and | or " onVoid="ignoreAfter" />
</trim>
<!--
    等价于<where>
-->

<trim prefix="SET" suffixOverrides="," />
</trim>
<!--
    等价于<set>
-->
```

- `where` 和 `set` 语句可以使用 `trim` 替换。`Trim` 语句定义了语句的整体前缀，以及对每个子句需要处理的多余的前缀与后缀列表（用|划分）。
- `onVoid` 参数可以出现在 `where`、`set`、`trim` 里，有两个取值"ignoreAfter"和"quit"。分别代表当这个 `trim` 语句里任何子句都不成立，导致生成一个空串时，采取什么逻辑。`ignoreAfter` 代码忽略下面的格式化语句，直接返回当前生成的 `sql` 语句执行，`quit` 代表不再执行这条 `sql` 语句，但会返回成功。

## sql

```

<sql id="userColumns"> id,username,password </sql>
<select id="selectUsers" result="{}">
    select ${userColumns}
    from some_table
    where id = #{id}
</select>

```

定义可重用的 sql 代码段，在其它语句中使用\${name}来原文引入进来。  
name 不能为'T'或't'，因为\${T}和\${t}代表默认的表名了。sql 块里面可以再引用别的 sql 块。

## try except

```

<insert id="insertTemplates" arguments="templates" foreach="templates"
<try>
    insert into ${T} (tplId, tplVersion, time, data, tag) values(${templates})
    <except>
        update ${T} set %{'* = #{model.*}'}, data, tag where tplId = #{model.tplId}
    </except>
</try>
</insert>

```

有时，同一个 model 可能多次插入数据库，用 insert or replace 会导致当 model 主键冲突（同主键的 model 已经在数据库存在）时，原先的数据被删除掉，重新 insert。这样会导致同条数据的 rowid 发生变化。用 try except 语句块可以解决这个问题（当然不仅限于解决这种问题）。try except 只能出现在 DAO Method 定义里，前后不能再有其它语句。try 和 except 里面可以包含其它语句块。

当这条 DAO 方法执行时，如果 try 里面的语句执行失败，会自动尝试执行 except 里的语句。如果都失败，这次 DAO 调用才会返回失败。

@cnName 2.3.3 引用方式 @priority 3

## 2.3.3 引用方式

[TOC]

### @引用

@{something}，用于方法参数，参数名为 something，在格式化 sql 语句时会把对象内容拼到 sql 语句中；因为参数都为 id 类型，所以默认使用 [NSString stringWithFormat:@"%@", id] 来格式化；@{something or ""} 这种格式，表示传入的参数如果为 nil，会转成一个空字符串而不是 NULL。

不建议使用@{}的方式来引用参数，效率比较低，有 SQL 注入风险。如果参数对象是个 NSString，拼接进去后，会自动添加引号将字符串括起来，保证 SQL 语句格式的正确性。如果用户在配置文件里自己写了引号，底层不会自动添加引号了。

使用 @{something no ""}，这种格式，可以强制不添加引号。

```
<select id="getMessage" arguments="id" result="[MessageModel]">
    select * from ${T} where id = {@{id}}
</select>
```

比如上例，id 参数传进来是个 NSString，上面的写法是正确的，生成的 SQL 会自动把 id 格式化进去，并且在前后添加引号。

### #引用

#{something}，用于方法参数，参数名为 something，在格式化 sql 语句时会转为一个'?'，然后将对象绑定给 sqlite；建议书写 sql 时尽量使用这种方式，效率更高。#{something or ""} 这种格式，表示传入的参数如果为 nil，会转成一个空字符串而不是 NULL。

### \$引用

`${something}`，用来引用配置文件里的内容，比如引用默认表名`{T}`或`{t}`、引用配置文件里定义的常量和 sql 代码块。

## 链式访问

对于`@`和`#`引用，可以使用`.`来访问参数对象的属性。比如传入的参数名是`model`，并且是一个`MessageModel`类型，它有属性`NSString* content`。那么`@{model.content}`，会取出其`content`属性的值。内部实现为`[NSObject valueForKey:]`，所以如果参数是一个字典（字典的`valueForKey`等价于`dict[@""]`），那么也可以使用`#`引用`adict[@"aaa"]`值。

@cnName 2.3.4 DAO 代理 @priority 4

## 2.3.4 DAO 代理

每个生成的 DAO 对象代理对象都支持 APDAOProtocol。

比如

```
@protocol MyDAOOperations <APDAOProtocol>
- (APDAOResult*)insertMessage:(MyMessageModel*)model;
@end
```

具体方法见代码的函数注释，如果有特殊需求，可以提给我，添加方法。

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>
#import "APDataCrypt.h"
#import "APDAOResult.h"
#import "APDAOTransaction.h"

@protocol APDAOProtocol;

typedef NS_ENUM (NSUInteger, APDAOProxyEventType)
{
    APDAOProxyEventShouldUpgrade = 0,      // 即将升级
    APDAOProxyEventUpgradeFailed,          // 表升级失败
    APDAOProxyEventTableCreated,           // 表被创建
    APDAOProxyEventTableDeleted,           // 表被删除
};

typedef void(^ APDAOProxyEventHandler)(id<APDAOProtocol> proxy, APDAOI

/**
 * 这个 Protocol 定义的方法每个 DAO 代理对象都支持，使用时使用 id<APDAOPI
 */
@protocol APDAOProtocol <NSObject>
```

```
/**  
 * 配置文件里 module 可以设置表名，如果想实现配置文件作为一个模板，操作不同  
 * 比如要对与每个 id 的对话消息进行分表的情况。  
 */  
@property (atomic, strong) NSString* tableName;  
  
/**  
 * 返回这个 proxy 操作的表所在数据库文件的路径  
 */  
@property (atomic, strong, readonly) NSString* databasePath;  
  
/**  
 * 获取这个 proxy 操作的数据库文件的句柄  
 */  
@property (atomic, assign, readonly) sqlite3* sqliteHandle;  
  
/**  
 * 注册全局变量参数，这些参数配置文件里的所有方法都可以使用，在配置文件里使  
 */  
@property (atomic, strong) NSDictionary* globalArguments;  
  
/**  
 * 这个 proxy 的事件回调，业务自行设置。回调线程不确定。  
 注意循环引用的问题，业务对象持有 proxy，这个 handler 方法里不要访问业务  
 */  
@property (atomic, copy) APDAOProxyEventHandler proxyEventHandler;  
  
/**  
 * 设置加密器，用于加密表里标记为需要加密的列的数据。向表里写数据时，碰到这  
 *  
 * @param crypt 加密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，  
 */  
@property (atomic, assign) APDataCrypt* encryptMethod;  
  
/**  
 * 设置解密器，用于解密表里标记为需要加密的列的数据。从表里读数据时，碰到这  
 *  
 * @param crypt 解密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，  
 */
```

```

/*
@property (atomic, assign) APDataCrypt* decryptMethod;

/**
 * 返回 sqlite 的最后一条 rowId
 *
 * @return sqlite3_last_insert_rowid()
 */
- (long long)lastInsertRowId;

/**
 * 获取配置文件定义的所有方法列表
 */
- (NSArray*)allMethodsList;

/**
 * 删除配置文件里定义的表，可以用于特殊情况下的数据还原。删除表后，DAO 对象
 */
- (APDAOResult*)deleteTable;

/**
 * 删除符合某个正则规则的所有表，请确保只删除本 Proxy 操作的表，否则可能发
 *
 * @param pattern 正则表达式
 * @param autovacuum 删除完成是否自动调用 vacuum 清理数据库空间
 * @param progress 进度回调，可传 nil，回调不保证主线程。为百分之后的结
 *
 * @return 操作是否成功
 */
- (APDAOResult*)deleteTablesWithRegex:(NSString*)pattern autovacuum:(I

/**
 * 调用自己的数据库链接执行 vacuum 操作
 */
- (void)vacuumDatabase;

/**
 * DAO 对象可以自己把操作放在事务里提升速度，实际调用的是该 DAO 对象操作的

```

```
 */
- (APDAOResult*)daoTransaction:(APDAOTransaction)transaction;

/**
 * 创建一个数据库副连接，为接下来可能发生的并发select 操作加速使用。可以调
 * 这些创建的链接会自动关闭，业务层无须处理。
 *
 * @param autoclose 在空闲状态多少秒后自动关闭，0 表示使用系统值
 */
- (void)prepareParallelConnection:(NSTimeInterval)autoclose;

@end
```

## @cnName 2.3.5 事务 @priority 5

## 2.3.5 事务

每个 DAO 调用都是自成一个事务的，如果一个 DAO 操作，含有多条 SQL 语句执行，也是自成一个事务。比如

```
<update id="createTable">
    <step>
        create table if not exists ${T} (localID long, clientMsgID char
    </step>
    <step>
        create index if not exists messageState_idx on ${T} (messageStat
    </step>
</update>
```

这个创建表的操作（创建表的 DAO 操作会自动执行），里面用 step 分为两条 SQL 调用，先创建表，再创建索引，这两条语句整体是一个事务。

再比如 insert 操作里有 foreach 参数

```
<insert id="addMessages" arguments="messages" foreach="messages.model">
    insert or replace into ${T} (id, content, uin, read) values(${model})
</insert>
```

业务调用这条 DAO 时，传入消息模型列表 messages，DAO 会对每个 model 执行一次插入操作的 SQL（底层的循环处理），整个操作也是一个事务。

有些时候，可能有很多次 DAO 调用，虽然每条自成一个事务，但是因为 DAO 调用次数过多，导致效率仍然不高。业务可以在所有 DAO 调用前开启事务，在所有 DAO 调用后结束事务。

```

NSString* filePath = [[NSBundle mainBundle] pathForResource:@"demo_common"
id<DemoProtocol> proxy = [[APDataCenter defaultCenter] daoWithPath:filePath];
[proxy daoTransaction:APDAOTransactionBegin];
...// massive DAO invocations
[proxy daoTransaction:APDAOTransactionCommit];

```

这里直接调用了 proxy 的 daoTransaction 方法。这个方法实际是声明在 APDAOProtocol 里的，所以要求 DemoProtocol 继承自 APDAOProtocol。(实际，所有 DAO proxy 对象都支持 APDAOProtocol 里的方法，是否显式声明它的 Protocol 继承自 APDAOProtocol 由业务自行决定。)

调用 proxy 的 daoTransaction 方法是最简单直接的。实际上由于上个例子中的 proxy 创建时，userDependent 为 YES。所以它操作的数据库文件实际为 APDataCenter.userPreferences。所以上面的例子等价于下面的调用：

```

NSString* filePath = [[NSBundle mainBundle] pathForResource:@"demo_common"
id<DemoProtocol> proxy = [[APDataCenter defaultCenter] daoWithPath:filePath];
[APUserPreferences daoTransaction:APDAOTransactionBegin];
...// massive DAO invocations
[APUserPreferences daoTransaction:APDAOTransactionCommit];

```

如果 proxy 创建时 userDependent 为 NO，可以调用 APCommonPreferences 的 daoTransaction 方法。

一个比较合理的把一系列 DAO 操作放到事务里的调用模板

```
- (BOOL)multipleDAOInvocations
{
    APDAOResult* transactionResult = [proxy daoTransaction:APDAOTransac-
    if ([transactionResult failed])
        return NO;

    BOOL daoResult = YES;
    @try
    {
        // massive DAO invocations
        daoResult = [[proxy daoMethod1] succeeded];
        if (!daoResult)
            return NO;

        daoResult = [[proxy daoMethod2] succeeded];
        if (!daoResult)
            return NO;

        return daoResult;
    }
    @finally
    {
        [proxy daoTransaction:daoResult ? APDAOTransactionCommit : API-
    }
}
```

## @cnName 2.3.6 函数重载 @priority 6

## 2.3.6 函数重载

DAO 接口（@protocol）与 XML 配置文件里的方法，使用方法名和参数个数进行匹配，与参数名无关。XML 里的参数名，可以与 DAO 接口里 selector 的参数名不一致，只要保证顺序对应即可。

比如，xml 里定义两个接口，方法名相同，参数数量不一样

```
<select id="getMessage" arguments="id" result="[{}]">
    select * from ${T} where id=#{id}
</select>
<select id="getMessage" arguments="id, time, state" result="{}">
    select * from ${T} where id=#{id} and time=#{time} and state=#{st
</select>
```

DAO 接口里的定义

```
@protocol DemoProtocol <APDAOProtocol>
- (NSArray*)getMessage:(NSString*)msgId;
- (NSDictionary*)getMessage:(NSString*)msgId msgTime:(NSNumber*)msgTi
@end
```

可以看到，DAO 接口中第二个方法里 selector 的参数名与 xml 里的参数名不一致，这是允许的，同样可以成功调用。

## @cnName 2.3.7 并行 select @priority 7

## 2.3.7 并行 select

默认发行版 sqlite 为完全串行，iOS 自带的 sqlite 为多线程模式，可以支持多个数据库连接。DAO 功能新增并行 select 功能，支持指定某 select 方法使用自己的数据库连接。这样在其它密集的写入操作时，select 操作可以不被挂起等待。

```
<select id="getTemplatesById" arguments="id" result="[PPDynamicTemplate]>
    select * from ${T} where id = #{id}
</select>
```

定义方法如上，在 select 方法后添加参数 parallel="true" 即可。

APDAOProtocol 接口新增方法 prepareParallelConnection

```
/**
 * 创建一个数据库副连接，为接下来可能发生的并发select 操作加速使用。可以调
 * 这些创建的链接会自动关闭，业务层无须处理。
 *
 * @param autoclose 在空闲状态多少秒后自动关闭，0 表示使用系统值
 */
- (void)prepareParallelConnection:(NSTimeInterval)autoclose;
```

你可以在创建 DAO 代理对象时，调用一次这个方法。统一存储会为这个数据库额外准备一份数据库连接，用于接下来可能发生的密集操作。如果不调用这个方法预先准备数据库连接，则会在调用 select 时，有需要再创建。这些数据库连接空闲时会自动关闭，业务不需要管理。

## @cnName 2.3.8 高级语法 @priority 8

## 2.3.8 高级语法

[TOC]

### 快捷语法

```
<insert id="addMessages" arguments="messages" foreach="messages.model">
    insert or replace into ${T} (id, time, content, uin, read, date) 
</insert>
```

比如上面这条插入语句，如果 model 对象有很多属性都需要插入到数据库里，那么 values 里面每条属性都要写一次，费时费力。

再比如

```
<update id="updateTemplate" arguments="template">
    update ${T} set data = #{template.data}, tag = #{template.tag} where
</update>
```

set 后面如果很多属性都要进行 update，那么 sql 语句写起来很复杂。

快捷语句用来解决这种问题：

```
%{'#{model.*}'}, msgId, time, content, uin, read, date}
```

等价于

```
#{{model.msgId}}, {{model.time}}, {{model.content}}, {{model.uin}}, {{model.read}}, {{model.date}}
```

```
%{* = #{model.*}', data, tag, id, version}
```

等价于

```
data = #{model.data}, tag = #{model.tag}, id = #{model.id}, version =
```

基本格式为

```
%{'format', ...}
```

第一个参数包在单引号里，表示格式，后面接的为需要格式化的数据数组。DAO 会对每个对象执行一次，把 format 里的 \* 替换为相应数据，拼接到一起。不用考虑性能的损耗，这个预处理只会执行一次。

上面两个例子可以修改为

```
<insert id="addMessages" arguments="messages" foreach="messages.model">
    insert or replace into ${T} (id, time, content, uin, read, date) .
</insert>

<update id="updateTemplate" arguments="template">
    update ${T} set %{* = #{template.*}'}, data, tag} where tplId = #.
</update>
```

## try except

```

<insert id="insertTemplates" arguments="templates" foreach="templates"
<try>
    insert into ${T} (tplId, tplVersion, time, data, tag) values(${model.*})
<except>
    update ${T} set ${'*' = '#{model.*}'}, data, tag where tplId = ${model.tplId}
</except>
</try>
</insert>

```

有时，同一个 model 可能多次插入数据库，用 insert or replace 会导致当 model 主键冲突（同主键的 model 已经在数据库存在）时，原先的数据被删除掉，重新 insert。这样会导致同条数据的 rowid 发生变化（通常不用关心 rowid，不过有些情况需要 rowid 在整个数据库文件的生命周期保持不变）。用 try except 语句块可以解决这个问题（当然不仅限于解决这种问题）。try except 只能出现在 DAO Method 定义里，前后不能再有其它语句。try 和 except 里面可以包含其它语句块。

当这条 DAO 方法执行时，如果 try 里面的语句执行失败，会自动尝试执行 except 里的语句。如果都失败，这次 DAO 调用才会返回失败。

## 非空参数

```

<insert id="addMessage" arguments="$model">
    insert or replace into ${T} ${table_columns} values(${model.msgId}, ${model.message})
</insert>

```

业务调用 DAO 接口，是允许传 nil 参数的。但当参数名前面有 \$ 符号时，表示该参数不接受 nil 值。当调用者不小心传入了 nil 值，DAO 调用会自动失败，防止发生不可预知的问题。

## 条件格式化

条件格式化类似 `foreach` 关键字，它会接收一个字典（`NSDictionary*`）参数，并把字典里的 `k-v` 对，格式化为“`k1=v1, k2=v2`”的格式。

```
<insert id="updateMessage" arguments="values, conditions">
    update ${T} <format prefix="set" pairs="values" join=","/><format
</insert>
```

如果传入的 `values` 是`@{@"uin":@(12345), @"content":@"12345"}`，  
`conditions` 为`@{@"id":@"100"}`

这个方法生成的 SQL 语句如下：

```
update table set uin = ?, content = ? where id = ?
```

并将`@(12345)`, `@"12345"`, `@"100"`绑定给 sqlite

当某个条件传入的 `pairs` 为空时，`prefix` 也不会拼接到 SQL 语句里。

@cnName 音频录制播放 @priority 2

# 音频录制播放

[TOC]

## 1 ALPAudioRecord

```
@interface ALPAudioRecord : NSObject <AudioRecordFinishDelegate>

//音量改变定时器
//@property(nonatomic, retain) NSTimer *voicePowerTimer;

//录音数据
@property(nonatomic, retain) NSData *cafData;

//最长录音时间， 默认 30 秒
@property(nonatomic, assign) double maxDuration;

//最短录音时间， 默认 1 秒
@property(nonatomic, assign) double minDuration;

//创建时间
@property(nonatomic, retain) NSString *createAt;

//filePath 为空， 代表不需要存 file.
@property(nonatomic, retain) NSString *filePath;

@property(nonatomic, assign) id <ALPAudioRecordDelegate> delegate;

//设为 YES， 转码及存储将在子线程里去做(delegate 仍在主线程回调)；为保证兼容
//建议使用 YES;
@property(nonatomic, assign) BOOL recordAsync;

/**
```

```
* 启动录制
* @param filePath 录音完后保存的路径，需要带文件名，并且当目录存在时才能使用
* @return 带有语音文件地址的语音类
*
*/
- (void)startRecordSavedInFile:(NSString *)filePath;

/***
* 取消录制
*/
- (void)cancelRecord;

/***
* 完成录音
*/
- (void)finishRecord;

/***
* 录音权限
*/
+ (void)requestMicrophonePermission:(void (^)(BOOL granted))block;
@end
```

## 实现代理

```

@protocol ALPAudioRecordDelegate <NSObject>
@optional
/** 
 * 录制结束的回调
 * 录制结束的情况：
 * 1. 调用 finishRecord 完成录制
 * 2. 调用 cancelRecord 取消录制
 * 3. 录制时间超过设定的最大时间
 * 4. 录制保存的文件路径无效
 * 5. 录制失败
 */
- (void)recordFinishWithALPAudioMessage:(ALPAudioMessage *)message re

/** 
 * 由于录音超时，将要进行格式转换和自动存储，供业务方 UI 控制使用。
*/
- (void)recordWillFinishForTimeOut;

@end

```

## 2. ALPAudioPlayer

```

@interface ALPAudioPlayer : NSObject

@property (nonatomic, retain) ALPAudioMessage *message;

//+ (ALPAudioPlayer *)sharedALPAudioPlayer;

/** 
 * 创建 ALPAudioPlayer 的页面在 viewWillDisappear 或退出时需要调用
*/
- (void)stop;

/** 

```

```
* 切换到扬声器状态下可以触发一些 UI 的显示用来提示用户
*/
- (void)switchToSpeaker:(LWAudioRecorderCallbackBlock)block;

/**
 * 播放音频文件
 * @param message 音频文件
*/
- (void)playWithAudioMessage:(ALPAudioMessage *)message;

/**
 * 停止播放音频文件
 * @param message 音频文件
*/
- (void)stopWithAudioMessage:(ALPAudioMessage *)message;

/**
 * 播放音频文件
 * @param message 音频文件
 * @param block 播放完毕后的回调
*/
- (void)playWithAudioMessage:(ALPAudioMessage *)message stopBlock:(LW
    ...


/**
 * 播放
*/
- (void)play;




/**
 * 当前音频文件是否正在播放
*/
- (BOOL)isPlayWithAudioMessage:(ALPAudioMessage *)message;




/**
 * 保持最后一次播放模式，通常在 playXXX 方法之前执行 如：最后一次是听筒，这
*/
- (void)keepLastPlayMode;
```

@end

---

@cnName 分享组件 @priority 3

# 分享组件

[TOC]

## 1 简介

分享组件 MPShareKit 提供微博、微信、支付宝钱包朋友、QQ、来往、短信等渠道的分享功能，提供给开发者统一的接口，无须处理各 SDK 的接口差异性。

## 2 接入渠道

接入各渠道之前须在各分享渠道官方网站申请账号，地址如下：

- 微博： <http://open.weibo.com/>
- 微信： <https://open.weixin.qq.com/>
- QQ： <http://open.qq.com/>
- 支付宝： <http://open.alipay.com/index.htm>

## 3 接入 SDK

分享组件会携带多个渠道的资源包，也需要加到应用里，同时在应用的 Info.plist 文件中加入 urlScheme（从各分享渠道获取）

- 微信、来往回调源 APP 的 scheme 均为分配的 key
- 微博的 scheme 为 "wb"+key
- QQ 的 scheme 为 "tencent"+APPID
- 支付宝的 Identifier 为 alipayShare， scheme 为分配的 appID

详情请参考各渠道的官方文档。

## 4 初始化分享渠道的密钥

移动分享组件提供两种注册密钥的方式，根据是否使用客户端框架而有所不同。

## 4.1 使用客户端框架时

实现协议 mPaaSAppInterface 的下列方法，并在此方法中以词典的形式返回 key、secret 的值。

## 4.2 不使用客户端框架时

在接入应用的启动方法中注册密钥。

```

- (BOOL)application:(UIApplication *)application didFinishLaunchingWith-
{
    // 此 key 和 secret 为 demo 所使用，仅能够进行分享，来往需要手动指定分
    NSDictionary *dic = @{@"laiwang" : @{@"key" : @"your_key", @"secre
    @"weixin" : @{@"key":@"wxc5c09c98c276ac86", @"secret": @"d56057d8a
    @"weibo" : @{@"key": @"1877934830", @"secret": @"1067b501c42f484262
    @"qq" : @{@"key": @"1104122330", @"secret": @"WyZkbNmE6d0rDTLf"}, 
    @"alipay" : @{@"key": @"2015060900117932"}/*该 key 对应的 bundleID

    [APSKClient registerAPPConfig:dic];
}

```

## 5 接口说明

### 5.1 唤起分享选择面板

在唤起分享面板时可以指定需要显示的渠道

```

NSArray *channelArr = @[kAPSKChannelQQ, kAPSKChannelLaiwangContacts, I

self.launchPad = [[APSKLaunchpad alloc] initWithChannels:channelArr se
self.launchPad.delegate = self;
[self.launchPad showForView:[[UIApplication sharedApplication] keyWindow]

```

### 5.2 完成分享操作

在 `@protocol APSKLaunchpadDelegate` 的 `sharingLaunchpad` 回调中处理分享

```
- (void)sharingLaunchpad:(APSLaunchpad *)launchpad didSelectChannel:  
{  
    [self shareUrl:channelName];  
    [self.launchPad dismissAnimated:YES];  
}  
  
- (void)shareUrl:(NSString*)channelName  
{  
    //生成数据，调用对应渠道分享  
    APSKMessage *message = [[APSKMessage alloc] init];  
    message.contentType = @"url";//类型分"text", "image", "url"三种  
    message.content = [NSURL URLWithString:@"www.sina.com.cn"];  
    message.icon = [UIImage imageNamed:@"1"];  
    message.title = @"标题";  
    message.description = @"描述";  
  
    APSKClient *client = [[APSKClient alloc] init];  
  
    [client shareMessage:message toChannel:channelName completionBlocl  
        //userInfo 为扩展信息  
        if(!error)  
        {  
            //your logistic  
        }  
        NSLog(@"error = %@", error);  
    }];  
}
```

## 5.3 从渠道应用跳回的处理

当使用客户端框架时不需要处理，会由框架负责。当不使用客户端框架时参照下列代码进行处理：

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
{
    //加入分享成功后，从渠道 APP 回到源 APP 的处理
    BOOL ret;
    ret = [APSKClient handleOpenURL:url];
    return ret;
}
```

@cnName 汉字拼音处理 @priority 4

# 汉字拼音处理

[TOC]

## 1 简介

汉字拼音处理是比较繁琐的工作，移动框架为开发者提供的汉字拼音处理模块可以大大简化开发者在 iOS 应用上的处理难度，具有以下功能特点：

(1) 汉字转拼音

- 根据汉字的 Unicode 码直接取出汉字的拼音

(2) 搜索功能

- 支持 2 个字段的搜索，其中主字段可以支持拼音，多音字搜索，副字段按字符串匹配搜索。
- 搜索结果会返回主字段匹配数据的索引，匹配的位置，副字段匹配数据的索引，2 个字段的匹配数据不重复。
- 建索引与搜索速度快。

## 2 功能介绍

### 2.1 汉字转拼音

调用 `APPinyinSearchManager` 的类方法

```

/**
 @brief 获取字符串的拼音串
 @param text: 输入的字符串
 @return 返回拼音串
 */
+ (NSString *)getPinYinWithText:(NSString *)text;

```

## 2.2 搜索功能

### 2.2.1 组织数据

待搜索的数据模型需要实现搜索数据协议，定义搜索的主字段和副字段

```

@protocol APPinyinSearchDataProtocol <NSObject>
@required
- (NSString *)primarySearchData;      //搜索主字段
@optional
- (NSString *)secondarySearchData;   //搜索副字段
@end

```

### 2.2.2 创建搜索索引

调用者创建并持有 `APPinyinSearchManager` 的实例对象。将组织好的数据传入 `APPinyinSearchManager` 的对象来创建索引，支持下面 2 种方式创建搜索索引：

1. Manager 不会持有传入的数据，只会持有数据在原始 Dict, Array 中的位置，主副字段。
2. 对一个 Manager 调用 `buildIndex` 方法时会覆盖原来的搜索索引，传空时会重置搜索索引。
3. 可以建立多个 Manager 实例来搜索不同维度的数据。
4. 建立索引是异步的，如果在建立索引的过程中使用搜索，会在索引完成后自动搜索并回调。

```

/**
@brief 建立搜索索引
@param dict: 实现协议的数据 (Ex: {A:[contact1,contact2,contact3], B:[
@param indexChar: 数据 Dictionary 的有序 keys, 用于返回数据的优先级 Ex:[
*/
- (void)buildSearchIndexWithDataDict:(NSDictionary *)dict indexChar:(I

/**
@brief 建立搜索索引
@param array: 数据 Array Ex:[contact1,contact2,contact3]
*/
- (void)buildSearchIndexWithDataArray:(NSArray *)array;

```

## 2.2.3 搜索

如果调用搜索时正在创建搜索索引，`APPinyinSearchManager` 会在索引建立好后顺序搜索并回调。

```

/**
@brief 搜索数据, 优先主字段, 主字段拼音匹配, 主副字段匹配数据不重复
@param searchText: 搜索串
@param owner: 调用的 owner, 一般传入 self, 为取消搜索使用
@param callBack: 搜索结果的回调
*/
- (void)search:(NSString*)searchText owner:(id)owner completionBlock:@

```

## 2.2.4 返回搜索结果

搜索结果通过下面的回调函数 Block 返回

```

/**
@brief 搜索字符串的回调
@param searchText: 搜索串
@param primaryMatchArrayWithPosition: 主字段匹配出来的数据，数组中的对象是 NSIndex
@param secondaryMatchArray: 副字段的匹配出来的数据，数组中的对象是 NSIndex
@param error: 预留错误字段
*/
typedef void (^ APPinyinSearchCallback) (NSString * searchText, NSMutable

```

其中 `primaryMatchArrayWithPosition` 数组中的数据结构如下

```

@interface APPinyinSearchPosition : NSObject
/**
@brief 名称匹配开始的位置
*/
@property (nonatomic, assign) int matchStart;
/**
@brief 名称匹配结束的位置
*/
@property (nonatomic, assign) int matchEnd;
/**
@brief 是否是全拼匹配
*/
@property (nonatomic, assign) BOOL matchAllInPy;
/**
@brief 是否是所有 word 匹配
*/
@property (nonatomic, assign) BOOL matchAllInWord;
/**
@brief 搜索数据对应 IndexPath。
    如果是字典，对应 section:key 的位置，row 是数据在 value 的位置。
    如果是数组，对应 section:0，row 是数据在数组中的位置。
*/
@property (nonatomic, retain) NSIndexPath * indexPath;

```

## 调用代码示范

```
[self.contactSearchManager search:searchText owner:self completionBlock:^(id<APContactSearchManagerCompletion> completion) {
    if (weakSelf.isSearchMode && [searchText isEqualToString:self.searchText]) {
        weakSelf.searchResultArray = [[NSMutableArray alloc] init];
        //找到主字段匹配到的数据
        for (int i = 0; i < [primaryMatchArrayWithPosition count]; i++) {
            APPinyinSearchPosition * position = [primaryMatchArrayWithPosition objectAtIndex:i];
            NSIndexPath * indexPath = position.indexPath;
            APContactInfo * contact = [weakSelf contactInfoInDataDictWithIndexPath:indexPath];
            //加入结果数组中
            [weakSelf.searchResultArray addObject:contact];
        }
        //记录主字段的位置匹配信息
        weakSelf.searchResultPositionArray = primaryMatchArrayWithPosition;
        //找到副字段的匹配的数据
        for (int i = 0; i < [secondaryMatchArray count]; i++) {
            NSIndexPath * indexPath = [secondaryMatchArray objectAtIndex:i];
            APContactInfo * contact = [self contactInfoInDataDictWithIndexPath:indexPath];
            //加入结果数组中
            [weakSelf.searchResultArray addObject:contact];
        }
        [weakSelf.tableView reloadData];
    }
}];
```

### 2.2.5 高亮显示匹配

使用 `APPinyinSearchPosition` 来展示主字段匹配位置

```
if (self.nameSearchPostion) {  
    NSUInteger start = self.nameSearchPostion.matchStart;  
    NSUInteger end = self.nameSearchPostion.matchEnd;  
    if (self.contactInfo.displayName.length > 0) {  
        isMatch = YES;  
        [self.contactNameLabel setText:self.contactInfo.displayName a-  
            NSRange redRange = NSMakeRange(start, (end-start+1));  
            [mutableAttributedString addAttribute:(NSString *)kCTForeg-  
            return mutableAttributedString;  
    }];  
}  
}
```

展示副字段匹配的位置

```
__weak APContactTableViewCell * weakSelf = self;  
[self.detailLabel setText:detailString afterInheritingLabelAttributes/  
    NSRange redRange = [detailString rangeOfString:weakSelf.searchText];  
    [mutableAttributedString addAttribute:(NSString *)kCTForegroundCo:  
    return mutableAttributedString;  
}];
```

## 2.2.6 其他功能

```
/**  
 * @brief 重置搜索索引  
 */  
- (void)resetSearchTree;  
/**  
 * @brief 取消响应搜索操作  
 * @param owner: 搜索方法调用者  
 */  
- (void)cancelSearchForOwner:(id)owner;
```



@cnName iOS 安全检测组件 @priority 5

# iOS 安全检测组件

[TOC]

## 1 简介

APSecurityUtility 提供一系列实用的安全检测接口，主要包括运行时 Hook 检查、外部调用检查、调试状态检查等。

调用 Hook 检测接口前，应先明确使用场景。根据编程语言、是否检测 iOS 系统 Framework API 函数、或自定义函数等条件，正确选择对应的检测接口。

## 2 接口介绍

### 2.1 APDetectClassSelectorSwizzling

```
/**  
 * 在调用一个(+)selector 前, 检查该 selector 是否被外部注入的库 hook  
 * 仅适用于本应用程序内(包含静态库)实现的 selector, 不能使用此方法检测 iOS  
 * 体系结构中其他类的 selector  
 *  
 * (BOOL)APDetectClassSelectorSwizzling(NSString *className, SEL sel)  
 *  
 *-----  
 * @param className selector 所属类名  
 * @param sel         selector  
 *  
 * @return 返回值类型 BOOL, 返回 YES 时表示该 selector 已被 hook  
 *-----  
 *  
 * 示例如下:  
 * BOOL isHooked = APDetectClassSelectorSwizzling(@"SomeClass", @selector(...))  
 */
```

## 2.2 APDetectInstanceSelectorSwizzling

```

/**
 * 在调用一个(-)selector 前, 检查该 selector 是否被外部注入的库 hook
 * 仅适用于本应用程序内(包含静态库)实现的 selector, 不能使用此方法检测 iOS
 *
 * (BOOL)APDetectInstanceSelectorSwizzling(NSString *className, SEL :
 *
 *
 * -----
 * @param className selector 所属类名
 * @param sel        selector
 *
 * -----
 * @return 返回值类型 BOOL, 返回 YES 时表示该 selector 已被 hook
 *
 * -----
 *
 * 示例如下:
 * BOOL isHooked = APDetectInstanceSelectorSwizzling(@"SomeClass", @:
 *
 */

```

## 2.3 APDetectCFuncCalledByOthers & APDetectNonFrameworkSelectorCalledByOthers

```

/**
 * 在 C 函数内部实现中加入此检测方法, 检查该函数是否被外部注入的程序调用, 例
 *
 * 示例如下:
 * bool isRisky = APDetectCFuncCalledByOthers;
 *
 */

```

## 2.4 APDetectUIKitInstanceSelectorSwizzling

```
/**  
 * 在调用一个 UIKit 里(-)selector 前，检查该 selector 是否被外部注入的库  
 * 仅适用于 iOS 系统 UIKit Framework 中定义的 API  
 *  
 * (BOOL)APDetectUIKitInstanceSelectorSwizzling(NSString *className,  
 *  
 *-----  
 * @param className selector 所属类名  
 * @param sel         selector  
 *  
 * @return 返回值类型 BOOL，返回 YES 时表示该 selector 已被 hook  
 *-----  
 *  
 * 示例如下：  
 * BOOL isHooked = APDetectUIKitInstanceSelectorSwizzling(@"UITextField"  
 *  
 */
```

## 2.5 APDetectLocalAuthInstanceSelectorSwizzling

```

/**
 * 在调用一个 LocalAuthentication.framework 里(-)selector 前，检查该 se
 * 仅适用于 iOS 系统 LocalAuthentication.framework 中定义的 API
 * 不适用于模拟器
 *
 * 注意：调用方需自行保证 link LocalAuthentication.framework (可设置 op
 * 当运行在 iOS8 以下的系统 link 不到 LocalAuthentication.framework)
 *
 * (BOOL)APDetectLocalAuthInstanceSelectorSwizzling(NSString *className,
 *
 *
 * @param className selector 所属类名
 * @param sel        selector
 *
 * @return 返回值类型 BOOL，返回 YES 时表示该 selector 已被 hook
 *
 *
 * 示例如下：
 * BOOL isHooked = APDetectLocalAuthInstanceSelectorSwizzling(@"LACo
 *
 */

```

## 2.6 APDetectJailbroken

```

/**
 * 越狱检测
 *
 * (BOOL)APDetectJailbroken;
*/

```

## 2.7 APDetectDebugger

```
/**  
 * 检查应用当前是否处于调试状态  
 * (BOOL)APDetectDebugger;  
 *  
 * @return 返回值类型 BOOL, 返回 YES 时表示当前为调试状态  
 */
```

## 2.8 APGetTextSectionDigest

```
/**  
 * 获取可执行程序二进制的代码段摘要  
 *  
 * (NSString *)APGetTextSectionDigest;  
 */
```

## 2.9 APDetectLinkerFlag

```
/**  
 * 检查二进制可执行程序的 linker flag 的 restrict 标志  
 *  
 * (BOOL)APDetectLinkerFlag;  
 * @return 返回值类型 BOOL, 返回 YES 时表示当前为存在  
 */
```

## 2.10 APCheckNotificationSecurity

```
/**  
 * 接收到通知后，检查通知发送者是否出自应用内部。当第三方插件发送通知名为 r  
 * 特别注意：需保证通知名字符串为静态字符串，若通知发送者以 format 拼接通知  
 *  
 * Sample:  
 *  
 * NSString *name = [notification name];  
 * if (!APCheckNotificationSecurity(name)) {  
 *     //不响应钱包外部发送的通知  
 *     return;  
 * }  
 *  
 * @return 返回值类型 BOOL，返回 YES 时表示安全  
 */
```

@cnName H5 容器 @priority 6

# H5 容器

[TOC]

## 1 简介

H5 容器由内核与壳两部分组成。

内核为 Poseidon.framework，是对 UIWebView 的再封装，提供了容器抽象对象、JSBridge、URL 统一拦截、事件分发、插件管理、JSAPI 管理等功能；

壳是内核的一个简单实例，提供一个 webViewController 和 webview 供外部使用，同时提供了通用的 JSAPI、通用的插件、简单的 UI 界面。

## 2 功能特点

1. 丰富的通用 JSAPI（详细文档见  
[http://ux.alipayinc.com/index.php/H5%AE%B9%99%A8%E6%96%87%E6%A1%A3](http://ux.alipayinc.com/index.php/H5%E5%AE%B9%E5%99%A8%E6%96%87%E6%A1%A3)）
2. 支持添加自定义 JSAPI
3. 可定制化的 UI 元素

## 3 模块接入

1. 将 mPaas.framework 添加到工程中。
2. 添加容器依赖的 iOS 系统框架：CoreTelephony、SystemConfiguration、MobileCoreServices、MessageUI、EventKit、AudioToolbox、CoreMotion、QuartzCore、CFNetwork。
3. 将 H5Service.bundle 和 Poseidon.bundle 链入到主工程根目录下，这两个 Bundle 会随着 mPaas.framework 一起分发给开发者。

接入模块后，可以使用下面代码唤起一个 H5 容器

```
H5ViewController *vc = [[H5Service sharedService] createWebViewController];
[self.navigationController pushViewController:vc animated:YES];
```

## 4 自定义 UserAgent

在容器启动参数里面配置 customUA 字段

```
H5ViewController *vc = [[H5Service sharedService] createWebViewController];
[self.navigationController pushViewController:vc animated:YES];
```

## 5 新增 JS API

### 5.1 修改 Poseidon-Config.plist

1. 新建一个类继承自 JsApiHandlerBase，会提供一个 handler:context:callback: 函数来处理参数输入和结果输出；
2. 在 Poseidon-Config.plist 的 JsApis 数组里新增一条记录，记录里包含类名和暴露给 JS 的 API 名称，声明在这里的 JS API 会在容器启动的时候一一注册进去。

不推荐这种做法，建议接入应用自己管理新增 API，运行时动态注册。因为修改了 Poseidon-Config 文件后，升级 H5 容器时需要注意覆盖问题。

### 5.2 运行时动态注册

1. 定义 JsAPI 名称，业务名.方法名，譬如： wealth.getMoney。
2. 获取 h5service 实例；
3. 通过 h5service 注册 JsAPI；

```

static NSString *jsApiName = @"wealth.getMoney";
H5Service *h5Service = [H5Service sharedService];
PSDJsApiHandlerBlock handlerBlock = ^(NSDictionary *data, PSDContext *context) {
    // data 是页面传递过来的参数
    // dosomething
    // ...
    // PSDContext 中可以获取h5webviewcontroller
    // 如果需要回调给页面，调用 responseCallbackBlock
};

// 先移除
[h5Service unregisterApiName:jsApiName];
// 再注册
[h5Service registerApis:@{jsApiName: handlerBlock}];

```

## 6 新增插件

1. 新建一个类继承自 `PluginBase` 或者实现 `PSDPluginProtocol`。每一个插件都会收到事件通知，可以通过实现 `handleEvent:`这个代理方法来接收事件通知，对某一个事件进行相应处理；
2. 插件的注册时机有两种。第一种是跟 `JSScript` 一样，在 `Poseidon-Config.plist` 的 `Plugins` 数组里新增记录，此类插件会在容器启动的时候进行注册并监听所有事件；第二种，在启动特定 H5 应用前，动态注册插件；
3. 插件是弱引用，请自己持有。

```

Plugin4SafePay *payPlugin = [[Plugin4SafePay alloc] init];
[self.plugins addObject:payPlugin];
[self.session addEventListener:kEvent_Navigation_All withListener:payPlugin];

```

@cnName 1 UI 控件 @priority 1

# 1 UI 控件

[TOC]

## APActionSheet

类名： APActionSheet， 继承 UIActionSheet

描述： 定义 ActionSheet 基类， 统一管理 ActionSheet

接口：

```
+(void)setIsBackGroundMode:(bool)isBackGroundMode;
```

属性：

```
@property(nonatomic,weak)id<UIActionSheetDelegate>dtDelegate;
```

示例：

```
- (void)setupActionSheetWithTitle:(NSString *)title
{
    if (self.mobileContact){
        _contactActionSheet = [[APActionSheet alloc] initWithTitle:title];
        NSArray *ary = [self.mobileContact phoneAry];
        ary = (ary != nil) ? ary : [self.mobileContact accountIdAry];
        for (NSString *content in ary) {
            [_contactActionSheet addButtonWithTitle:content];
        }
        // 取消按钮放到最后
        [_contactActionSheet addButtonWithTitle:TRANSFER_TEXT_TO_ACCOUNT];
        [_contactActionSheet showInView:self.view];
    }
}
```

## APActionSheetManager

类名： APActionSheetManager， 继承 NSObject

描述： 对 APActionSheet 进行管理。

接口：

```
/*
 * 构建一个单例类
*/
+ (APActionSheetManager *)sharedAPActionSheetManager;

/**
 *删除所有的 actionsheet
*/
-(void)dismissAllUIActionSheet;

/**
 *将 actionsheet 添加到显示队列里面
*/
-(void)addToShowedPool:(APActionSheet*)alertView;
`
```

属性：

```
@property(nonatomic,assign)bool backGroundMode;
```

示例：

```
APActionSheetManager *manager = [APActionSheetManager sharedAPActionS
```

## APAgreementBox

类名： APAgreementBox， 继承 UIControl

描述： 用于显示用户注册协议的组件

接口：

```
/*
 * 用于显示用户注册协议的组件
 * @param frame 视图在父视图的位置和大小
 * @param labelText 标题
 * @param linkText 超链接标题
 * @return 新创建并经过初始化的注册协议组件
 */
- (id)initWithFrame:(CGRect)frame labelText:(NSString *)labelText linkText:(NSString *)linkText checkBoxHidden:(BOOL)hidden;
```

属性：

```
// 用户点击单选框
@property(nonatomic, readonly) APCheckBox *checkBox;
// 用户点击超链接按钮
@property(nonatomic, readonly) APLinkButton *linkButton;
```

示例：

```
APAgreementBox *agreementBox = [APAgreementBox alloc] initWithFrame:CGRectMake(100, 100, 200, 300)
                                         labelText:@"我同意"
                                         linkText:@"取消"
                                         checkBoxHidden:NO];
[self.view addSubview:agreementBox];
```

## APAlertView

类名： APAlertView, 继承自 UIAlertView

描述： 定制的 UIAlertView, 只有两个按钮时按钮将上下排列

使用： 同UIAlertView； 请在需要两个按钮上下排列时使用此类， 其它情况下仍使用 UIAlertView

示例：

```
APAlertView *alertView = [[APAlertView alloc] initWithTitle:@"新消息"
                                                       message:@"我们将推
                                                       delegate:self
                                                  cancelButtonTitle:@"我知道了"
                                                  otherButtonTitles:@"使用遇到
[alertView show];
```

## APAlertViewManager

类名： APAlertViewManager， 继承 NSObject

描述： 对 APAlertView 进行管理。

接口：

```
/**  
 * 构建一个单例类  
 */  
  
+ (APAlertViewManager *)sharedAPAlertViewManager;  
  
/**  
 *删除所有的 alertView  
 */  
- (void)removeAllalertView;  
  
/**  
 *将 alertView 添加到显示队列里面  
 */  
- (void)pushAPAlertView:(UIAlertView *)alertView;
```

属性：

```
@property(nonatomic,assign)bool isBackGroundMode;
```

示例：

```
APAlertViewManager *manager = [APAlertViewManager sharedAPAlertViewMa
```

## APBankCardTextField

类名： APBankCardTextField， 继承 APTextField

描述： 银行卡文本输入框

接口： 参考 APTextField

示例：

```
APBankCardTextField *textField = [[APBankCardTextField alloc] initWithFrame:  
    textField.validator = [APTextValidator bankCardValidator];
```

## APBarButtonItem

类名： APBarButtonItem， 继承 UIBarButtonItem

描述： APBarButtonItem 是放置在 UIToolbar 或 UINavigationBar 上的按钮对象， 可以响应用户的操作。

接口：

```
/**  
 * @return 新创建 UIBarButtonItemFlexibleSpace 类型对象，用于调整  
 */  
+ (UIBarButtonItem *)flexibleSpaceItem;  
  
/**  
 * 用于创建并初始化一个“返回”的按钮对象  
 * @param title    锚钮标题，注意：设置没有效果，内部使用默认的返回图片  
 * @param target   响应按钮点击事件的对象  
 * @param action   响应按钮点击事件的函数  
 * @return 新创建并经过初始化的“返回”按钮对象  
 */  
+ (UIBarButtonItem *)backBarButtonItemWithTitle:(NSString *)title target:(id)target  
                                         action:(SEL)action;  
  
/**  
 * 用于创建并初始化一个指定图片的按钮对象。  
 * @param image    锚钮图片  
 * @param target   响应按钮点击事件的对象  
 * @param action   响应按钮点击事件的函数  
 * @return 新创建并经过初始化的指定图片的按钮对象。  
 */  
- (id)initWithImage:(UIImage *)image style:(UIBarButtonItemStyle)style  
               target:(id)target  
             action:(SEL)action;  
  
/**  
 * 用于创建并初始化一个指定标题的按钮对象。  
 * @param title    锚钮标题  
 * @param target   响应按钮点击事件的对象  
 * @param action   响应按钮点击事件的函数  
 * @return 新创建并经过初始化的指定标题的按钮对象。  
 */  
- (id)initWithTitle:(NSString *)title style:(UIBarButtonItemStyle)style  
               target:(id)target  
             action:(SEL)action;
```

示例：

```
APBarButtonItem *item = [[APBarButtonItem alloc] initWithImage:[UIImage imageNamed:@"账单.png"]];  
self.navigationItem.rightBarButtonItem = item;
```

```
APBarButtonItem *item = [[APBarButtonItem alloc] initWithTitle:@"账单" style: UIBarButtonItemStylePlain];  
self.navigationItem.rightBarButtonItem = item;
```

## APBorderedButton

类名： APBorderedButton， 继承 UIButton

描述： 目前默认实现了一下三种样式的按钮：

主按钮；

次按钮； 自定义按钮， 不会应用任何样式， 也不会初始化 frame

接口：

```
/**  
 * 一个方法的辅助方法，用于创建并初始化一个按钮的对象  
 *  
 * @param buttonType 按钮类型，必须是定义在<code>APBorderedButtonType</code>  
 * @param title       键钮标题  
 * @param target      响应按钮点击事件的对象  
 * @param action      响应按钮点击事件的函数  
 *  
 * @return 新创建并经过初始化的按钮对象  
 *  
 * 使用这个方法创建的按钮对象，其默认的 frame 为<code>CGRectMake(0.0, 0,  
 * 对于指定的 target 和 action 所对应事件为<code>UIControlEventTouchUpInside</code>  
 */  
+ (APBorderedButton *)buttonWithType:(APBorderedButtonType)buttonType  
                           title:(NSString *)title  
                         target:(id)target  
                        action:(SEL)action;  
  
/**  
 * 用于创建并初始化一个按钮的对象  
 *  
 * @param title       按钮标题  
 * @param buttonType 按钮类型，必须是定义在<code>APBorderedButtonType</code>  
 *  
 * @return 新创建并经过初始化的按钮对象  
 */  
- (id)initWithButtonTitle:(NSString *)title buttonType:(APBorderedButtonType)buttonType  
                     target:(id)target  
                    action:(SEL)action;
```

示例：

```
APBorderedButton *confirmBtn = [APBorderedButton buttonWithType:APBorderedButtonTypeNormal  
                                                               title:@"确认" target:self action:@selector(confirm)];
```

## APButton

类名： APButton， 继承 UIButton

描述： 目前默认实现了一下三种样式的按钮：

主按钮；

次按钮； 警示按钮， 通常按钮背景会显示为较醒目的红色； 自定义按钮， 不会应用任何样式， 也不会初始化 frame

接口：

```
/**  
  
 * 一个方法的辅助方法，用于创建并初始化一个按钮的对象。  
  
 * @param buttonType 按钮类型，必须是定义在 APButtonType 中的其中一个值。  
  
 * @param title      锚钮标题  
  
 * @param target     响应按钮点击事件的对象  
  
 * @param action     响应按钮点击事件的函数  
  
 * @return 新创建并经过初始化的按钮对象。  
  
 * 使用这个方法创建的按钮对象，其默认的 frame 为 CGRectMake(15.0, 0, 290, 50)。  
  
 * 对于指定的 target 和 action 所对应事件为 UIControlEventTouchUpInside。  
  
 */  
+ (APButton *)buttonWithType:(APButtonType)buttonType title:(NSString *)title  
    target:(id)target action:(SEL)action;  
  
/**  
  
 * 通过指定按钮类型来创建按钮  
  
 * @param buttonType 按钮类型，必须是定义在 APButtonType 中的其中一个值。  
  
 * @return 新创建并经过初始化的按钮对象  
  
 */  
- (id)initWithButtonType:(APButtonType)buttonType;
```

示例：

```
APButton *confirmBtn = [APButton buttonWithType:APButtonTypeDefault t:  
APButton *cancelBtn = [APButton buttonWithType:APButtonTypeSecondary t:
```

## APCheckBox

类名： APCheckBox 继承 UIControl

描述：该控件表明一个特定的状态（即选项）是选定 (on, 值为 true) 还是清除 (off, 值为 false)。 接口：

```
/**  
 * Returns an initialized <code>APCheckBox</code> object with the he:  
 * @return An initialized check box object.  
 */  
- (id)init;
```

属性：

```
/**  
 * A boolean value indicates whether the <code>APCheckBox</code> is checked.  
 *  
 * The default value is NO.  
 */  
@property(nonatomic, assign, getter = isChecked) BOOL checked;  
  
/** Gets or sets the image for check state. */  
@property(nonatomic, retain) UIImage *checkedImage;  
  
/** Gets or sets the image for unchecked state. */  
@property(nonatomic, retain) UIImage *uncheckedImage;  
  
/**  
 * The inset or outset margins for the rectangle around the check-box.  
 * The default value is <code>UIEdgeInsetsMake(0, 0, 0, 5.0)</code>.   
 */  
@property(nonatomic, assign) UIEdgeInsets imageEdgeInsets;  
  
/** Returns the label used for title of check box. */  
@property(nonatomic, strong, readonly) UILabel *titleLabel;
```

示例：

```
APCheckBox *checkBox = [[APCheckBox alloc] initWithFrame:CGRectZero];  
[checkBox sizeToFit];  
[self addSubview:checkBox];
```

## APCircleRefreshControl

类名： APCircleRefreshControl，继承 UIControl

描述：圆形加载刷新视图

接口：

```
// 创建视图在指定的 UIScrollView 上  
- (id)initInScrollView:(UIScrollView *)scrollView;  
  
// 开始刷新动画  
- (void)beginRefreshing;  
  
// 结束刷新动画  
- (void)endRefreshing;
```

示例：

```
self.refreshControl = [[APCircleRefreshControl alloc] initInScrollView:  
[self.refreshControl addTarget:self action:@selector(refreshEventRecie
```

## APCreditCardTextField

类名： APCreditCardTextField， 继承 APTextField

描述： 信用卡文本输入框

接口： 参考 APTextField

示例：

```
_textField = [[APCreditCardTextField alloc] initWithFrame: UIEdgeInsets:  
_textField.validator = [APTextValidator creditCardValidator];
```

## APExceptionView

类名： APExceptionView， 继承 UIView

描述：网络异常界面。

接口：

```
/**  
 * 初始化异常view 并设定异常类型  
 * @param frame view 的坐标, 必选  
 * @param type 异常类型, 必选  
 * @return APExceptionView  
 */  
  
- (id)initWithFrame:(CGRect)frame exceptionType:(APExceptionEnum) type;  
  
/**  
 * 获取大图标的只读 UIImageView  
 * @return UIImageView  
 */  
  
- (UIImageView *)getIconImageView;  
  
/**  
 * 获取说明文案的只读 UILabel  
 * @return UILabel  
 */  
  
- (UILabel *)getInfoLabel;  
  
/**  
 * 获取行动按钮 * @return UIButton  
 */  
  
- (UIButton *)getActionButton;
```

属性：

```
typedef enum {  
    APExceptionEnumNetworkError,  
    APExceptionEnumEmpty,  
    APExceptionEnumAlert  
} APExceptionEnum;
```

示例：

```
APExceptionView *exceptionView = APExceptionView alloc initWithFrame::
```







## APIImageAuthCodeBox

类名： APIImageAuthCodeBox，继承 APAuthCodeBox

描述： 图片验证码输入框

接口：

```
 /**
 * 创建图片验证码输入框
 * @param originY 组件的 y 坐标
 * @return 图片验证码输入框
 */
- (APIImageAuthCodeBox *)initWithOriginY:(CGFloat)originY;
```

属性：

```
@property(strong, nonatomic) UIImageView *authCodeImageView; // 图片马
@property(strong, nonatomic) UIImageView *circleImageView; // 图片马
```

示例：

```
- (void)viewDidLoad
{
    APIImageAuthCodeBox *imageBox = [[APIImageAuthCodeBox alloc] initWithFrame:CGRectMake(100, 100, 200, 100)];
    imageBox.inputBox.textField.placeholder = @"右侧验证码";
    imageBox.authCodeImageView.image = [UIImage imageNamed:@"auth_code"];
    [self.view addSubview:imageBox];
}

- (void)refreshAuthImage
{
    [imageBox startWaiting];
    [self performSelector:@selector(authImageReady) withObject:nil afterDelay:1.0];
}

- (void)authImageReady
{
    [imageBox stopWaiting];
}
```

## APInputBox

类名： APInputBox，继承自 UIView。描述：它内部包含一个 TextField(可通过 textField 属性获取），也可能包含一个显示在输入框内部右侧的按钮（可通过 iconButton 属性获取）。如果指定了 textFieldFormat，它将对 TextField 的文本自动用空格分组（一般用于证件号码输入）。创建：不带右侧按钮

```
/**  
 * 创建输入框组件  
 * @param originY 输入框的 y 坐标  
 * @param type 文本输入框的类型  
 * @return 输入框组件  
 */  
+ (APInputBox *)inputboxWithOriginY:(CGFloat)originY  
    inputboxType:(APInputBoxType)type;
```

### 带右侧按钮

```
/**  
 * 创建带图标按钮的输入框组件  
 * @param originY 输入框的 y 坐标  
 * @param icon 按钮上的图标, 44x44  
 * @param type 文本输入框的类型  
 * @return 带按钮的输入框组件  
 */  
+ (APInputBox *)inputboxWithOriginY:(CGFloat)originY  
    buttonIcon:(UIImage *)icon  
    inputboxType:(APInputBoxType)type;
```

### 属性：

#### APInputBox 内部属性

@property(strong, nonatomic)	APTextField	*textField;
@property(strong, nonatomic)	NSString	*textFieldText
@property(strong, nonatomic)	NSString	*textFieldFormat
@property(strong, nonatomic)	UIButton	*iconButton
@property(assign, nonatomic)	BOOL	hidesButtonWhileNe
@property(nonatomic, readonly)	UILabel	*titleLabel
@property(assign, nonatomic)	APInputBoxStyle	style
@property(assign, nonatomic)	APInputBoxType	inputBoxType

## 示例： 创建带图标按钮的示例代码

```
APIInputBox *phoneNumberBox = [APIInputBox inputboxWithOriginY:0 button:[phoneNumberBox.iconButton addTarget:self action:@selector(onShowAddress) forControlEvents:UIControlEventTouchUpInside];  
phoneNumberBox.titleLabel.text = @"手机号码";  
phoneNumberBox.textField.placeholder = @"联系人手机号";  
phoneNumberBox.hidesButtonWhileNotEmpty = YES;
```

## 创建不带图标按钮、按手机号码自动分段的示例代码

```
APIInputBox *phoneNumberBox = [APIInputBox inputboxWithOriginY:0 inputboxType:kInputboxTypePhone];  
phoneNumberBox.titleLabel.text = @"手机号码";  
phoneNumberBox.textFieldFormat = @"### #### ###";  
phoneNumberBox.textField.placeholder = @"联系人手机号";
```

## 方法： 方法说明

```

/**
 * 按照指定格式对文本添加空格
 * @param 文本内容
 * @return 添加空格后的文本
 */
- (NSString *)formatText:(NSString *)text;

/**
 * 对于没有在初始化时指定 icon 的 inputBox，可以使用此方法添加
 * @param icon 按钮上的图标，44x44
 *
 */
- (void)setRightButtonIcon:(UIImage *)icon;

/**
 * 检查输入的有效性。
 */
- (BOOL)checkInputValidity;

/**
 * 过滤文本，只可输入数字，限定最大长度
 * 参数为相应 delegate 参数，maxLength 为最大长度
 */
- (BOOL)shouldChangeRange:(NSRange)range
    replacementString:(NSString *)string
    withMaxLength:(int)maxLength;

/**
 * 限定最大长度
 * @maxLength 最大长度，不包括 format 的空格
 */
- (BOOL)shouldChangeRange:(NSRange)range
    replacementString:(NSString *)string
    withFormatStringMaxLength:(int)maxLength;

```

## APLinkButton

类名：APLinkButton，继承 UIControl

描述：超链接风格的按钮

接口：

没有自定义初始化函数，使用父类定义的初始化函数

```
/**  
 * 设置指定状态时标题显示的颜色  
 * @param color 标题显示的颜色  
 * @param state 指定的状态  
 */  
- (void)setTitleColor:(UIColor *)color forState:(UIControlState)state;  
  
/**  
 * 获取指定状态时标题显示的颜色  
 * @param state 指定的状态  
 * @return 标题显示的颜色  
 */  
- (UIColor *)titleColorForState:(UIControlState)state;
```

属性：

```
/** 链接按钮显示标题的视图 */  
@property(nonatomic, readonly) UILabel *titleLabel;  
  
/** 链接按钮显示的标题 */  
@property(nonatomic, strong) NSString *title;  
  
/** 标示标题下是否显示横线 */  
@property(nonatomic, assign) BOOL underline;
```

示例：

```
APLinkButton *forgotButton = [[APLinkButton alloc] initWithFrame:shadowFrame];
[forgotButton setTitle:@"忘记登录密码? "];
[forgotButton sizeToFit];
```

## APLoginBox

类名： APLoginBox， 继承 UIView

描述： 登录框控件

接口：

```
/**  
 * 输入框与密码框  
 */  
- (APInputBox *)usernameInputBox;  
- (APInputBox *)passwordInputBox;  
  
/**  
 * 构造函数  
 *  
 * @param originY 指定 frame.origin.y  
 * @param flag 是否带注册按钮  
 * @return 返回登录组件  
 *  
 */  
+ (APLoginBox *)loginBoxWithOriginY:(CGFloat)originY registerButton:(  
  
  
#pragma mark -  
#pragma mark 信用卡账单邮箱登陆定制方法  
/**  
 * 设置下拉框提示内容,并且直接显示下拉列表  
 *  
 * @param historyItems 下拉框中提示的内容  
 * @return 调整后需要增加的高度  
 */  
- (NSInteger)setHistoryItemsAndDisplayHistory:(NSArray *)historyItems;  
  
/**  
 * 设置文本框变更,修改通知逻辑  
 */  
- (void)setupDidBeginEditNotification;
```

### 属性：

```
**  
* 帐号输入框
```

```
/*
@property(nonatomic, readonly) UITextField *usernameField;
/** 
 * 密码输入框
 */
@property(nonatomic, readonly) UITextField *passwordField;
/** 
 * 验证码输入框
 */
@property(nonatomic, readonly) APAAuthCodeBox *authCodeBox;
/** 
 * 忘记密码"链接"
 */
@property(nonatomic, readonly) APLinkButton *forgotButton;
/** 
 * 注册按钮
 */
@property(nonatomic, readonly) UIButton *registerButton;
/** 
 * 登录按钮
 */
@property(nonatomic, readonly) UIButton *loginButton;

/**
 * 历史登录账号
 *
 * 数组，每个元素为一个 NSString，将显示在历史帐号列表上
 */
@property(nonatomic, copy) NSArray *historyItems;
/** 
 * 历史账号列表是否可见
 */
@property(nonatomic, assign) BOOL historyTableVisible;
/** 
 * 验证码控件是否可见
 */
@property(nonatomic, assign) BOOL authCodeBoxVisible;
/**
```

```
* 代理  
*/  
@property(nonatomic, assign) id<APLoginBoxDelegate> delegate;
```

示例：

```
_loginBox = [ALPLoginBox loginBoxWithOriginY:90 registerButton:YES];  
_loginBox.historyItems = @[@"13127995722", @"mozart0@tom.com", @"hbhft"];  
_loginBox.delegate = self;  
[_loginBox setLoginBoxOriginXCommonPix];  
[self.view addSubview:_loginBox];
```

## APMobileTextField

类名： APMobileTextField， 继承 APTextField

描述： 手机号输入框控件

接口： 查看 APTextField

示例：

```
_textField = [[APMobileTextField alloc] initWithFrame:UIEdgeInsetsInsetByEdgeInsets(0, 0, 0, 0)];  
_textField.validator = [APTextValidator mobileNumberValidator];
```

## APNetErrorView

类名： APNetErrorView， 继承 UIView

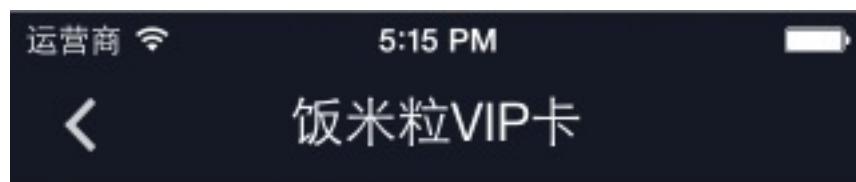
描述： 网络错误提示

接口：

```
/**  
 * 网络异常老控件样式，建议使用新控件 APExceptionView  
 * @param parent 显示在指定父视图  
 * @param target 响应按钮点击事件的对象  
 * @param action 响应按钮点击事件的函数  
 * @return 新创建并经过初始化视图对象。  
 */  
+ (id)showInParentView:(UIView *)parent withTarget:(id)target action:(SEL)action;  
  
/**  
 * 取消显示  
 */  
- (void)dismiss;  
  
/**  
 * 设置指定点击处理  
 * @param target 响应按钮点击事件的对象  
 * @param action 响应按钮点击事件的函数  
 */  
- (void)setTarget:(id)target action:(SEL)action;
```

示例：

```
self.netErrorView = [APNetErrorView showInParentView:self.mainView wi
```



## APNextPagePullView

类名： APNextPagePullView，继承 UIView

描述： 实现点击更多出现旋转框，加载下一页成功，旋转框消失。

接口：

```
/**  
 * 视图已经发生滚动的方法  
*/  
- (void)refreshScrollViewDidScroll:(UIScrollView *)scrollView;  
  
/**  
 * 视图已经发生拖动的方法  
*/  
- (void)refreshScrollViewDidEndDragging:(UIScrollView *)scrollView;  
  
/**  
 *数据加载完成  
*/  
- (void)refreshScrollViewDataSourceDidFinishedLoading:(UIScrollView *)  
  
/**  
 *数据加载完成后延迟一段时间旋转框消失  
- (void)refreshScrollViewDataSourceDidFinishedLoadingAndDelay:(UIScrollView *)
```

属性：

```
@property(nonatomic, weak) id delegate;

@protocol APNextPagePullViewDelegate
/**
 *判断是否正在加载下一页
*/
- (BOOL)isTableViewLoadingNextPage:(APNextPagePullView *)pullView;

/**
 *视图需要更新的通知
*/
- (void)upPullTableDidTriggerRefresh:(APNextPagePullView *)pullView;
```
示例：
```
APNextPagePullView *view = [[APNextPagePullView alloc] initWithFrame:...]
```

## APNumericKeyboard

类名： APNumericKeyboard， 继承 UIView

描述： 通过以下静态方法提供一个数字键盘。 键盘的确定按钮被点击时，  
textField 的 delegate 将收到 textFieldShouldReturn: 消息

接口：

```
+ (APNumericKeyboard *)sharedKeyboard;
```

属性：

```
//手动设置 textinput，外部需要设置 keyboard 的 y 轴
@property (nonatomic, strong) id<UITextInput> textField;
//是否是身份证输入键盘
@property (nonatomic, assign, getter = isIdNumber) BOOL idNumber;
//是否隐藏小数点按钮
@property (nonatomic, assign, getter = isDotHidden) BOOL dotHidden;
//是否隐藏收键盘按钮
@property (nonatomic, assign, getter = isDismissHidden) BOOL dismissH;
//是否隐藏确定按钮
@property (nonatomic, assign, getter = isSubmitEnable) BOOL submitEna
```

示例：

```
APIInputBox *inputBox1 = [APIInputBox inputBoxWithOriginY:15 placeholder:@"请输入身份证号"];
inputBox1.textField.inputView = [ALPNumericKeyboard sharedKeyboard];
[self.view addSubview:inputBox1];
```

## APNumPwdInputView

类名： APNumPwdInputView，继承 UIView

描述： 6 位数字密码输入框控件

接口：

```
/**  
 * 初始化 6 位数字密码  
 * quadWidth: 单个方格宽度  
 * quadHeight: 单个方格高度  
 */  
- (id)initWithQuadWidth:(CGFloat)quadWidth quadHeight:(CGFloat)quadHe:  
  
/**  
 * 初始化 6 位数字密码  
 *  
 * quadWidth: 单个方格宽度  
 * quadHeight: 单个方格高度  
 * password: 初始密码  
 */  
- (id)initWithQuadWidth:(CGFloat)quadWidth quadHeight:(CGFloat)quadHe:  
  
/**  
 * 清空密码框，不会触发delegate 回调  
 */  
- (void)reset;  
  
/**  
 * 显示键盘  
 */  
- (BOOL)showKeyBoard;  
  
/**  
 * 隐藏键盘  
 */  
- (BOOL)hideKeyBoard;  
  
/**  
 * 设置背景图片  
 */  
- (void)setBackgroundImage:(UIImage *)image;
```

属性：

```
/** 用户输入的密码 */
@property (nonatomic, strong, readonly) NSString *password;

/**
 * APNumPwdInputViewDelegate 代理，代理可以实现密码修改回调函数：
 * - (void)onPasswordDidChange:(APNumPwdInputView*)sender;
 */

@property (nonatomic, assign) id<APNumPwdInputViewDelegate> delegate;

/**
 * 是否是 6 位数字密码控件。如果是 NO，则显示为普通的密码输入框。
 */
@property (nonatomic, assign, getter = isNumericPassword) BOOL numeric
```

示例：

```
APNumPwdInputView *pwdInputView = [[APNumPwdInputView alloc] initWithFrame:CGRectMake(100, 100, 200, 100)];
pwdInputView.delegate = self;
[self.view addSubview:pwdInputView];
```

## APNumPwdPopupView

类名： APNumPwdPopupView，继承 UIView

描述：

一个弹出框，用于显示 6 位数字密码控件，还有以下元素：

返回按钮（可选） 标题 副标题（可选） 取消按钮（可选） 确定按钮 接口：

```
/** 创建弹出框，用于显示 6 位数字密码控件
```

```

* title          标题
* cancelButtonTitle 取消按钮标题
* confirmButtonTitle 确定按钮标题
*/
- (id)initWithTitle:(NSString*)title
    cancelButtonTitle:(NSString*)cancelButtonTitle
    confirmButtonTitle:(NSString*)confirmButtonTitle;

/** 创建弹出框，用于显示 6 位数字密码控件
* title          标题
* subtitleView   副标题
* cancelButtonTitle 取消按钮标题
* confirmButtonTitle 确定按钮标题
*/
- (id)initWithTitle:(NSString*)title
    subtitleView:(UIView *)subtitleView
    cancelButtonTitle:(NSString*)cancelButtonTitle
    confirmButtonTitle:(NSString*)confirmButtonTitle;

/** 如果没有设置 target 或者 selector，执行默认行为 */
- (void)addLeftBackButtonTarget:(id)target selector:(SEL)selector even

/** 获取subtitle 的固定宽度 */
+ (CGFloat )subtitleViewWidth;

/*
 *自定义的 view，用于显示副标题。
 *这个 view 中可以显示任何业务需要自定义的副标题，在调用这个函数之前，subtitleViewWidth 需要被设置。
 *如果设置了副标题，在主标题和副标题之间会显示一条分割线。
*/
- (void)showSubtitleView:(UIView *)subtitleView;

/*
 和 UIAlertView 不同，按钮点击后不会自动 dismiss，需要在 apNumPwdPopupViewDidDismiss: 方法中手动调用 dismiss。
*/
- (void)dismiss;
- (void)dismissWithCompletionBlock:(void (^)(void))block;
- (void)clearBlock;

```

示例：

```
- (void)showALPNumPwdInputView:(NSInteger)tag {  
  
    _pwdPopup = [[APNumPwdPopupView alloc] initWithTitle:@"输入手机支付密码"];  
  
    self.alertViewTag = tag;  
  
    __weak ALPNumPwdPopupView * weekpop = _pwdPopup;  
    __weak SWPhonePasswordViewController * weekvc = self;  
  
    _pwdPopup.cancelBlock = ^(ALPNumPwdPopupView * sender, int button){  
        [weekpop dismiss];  
        [weekvc reloadData];  
    };  
  
    _pwdPopup.confirmBlock = ^(ALPNumPwdPopupView * sender, int button){  
        [MBProgressHUD showHUDAddedTo:weekpop animated:YES];  
        [weekvc rpcForModifyGesture:sender.pwdInputView.password];  
    };  
  
    [_pwdPopup presentWithinWindow:DTContextGet().window];  
}
```

## APPickerView

类名： APPickerView， 继承 UIView

描述： 封装 UIPickerView， 带有“取消”和“完成”按钮的组合控件

接口：

```
/*
 * 创建组件
 * @param title 标题, 可为 nil
 * @return 创建的组件, 默认不显示, 需调用 show
 */
+ (APPickerView *)pickerViewWithTitle:(NSString *)title;

/*
 * 初始化对象
 * @param frame 显示位置
 * @param title 显示标题, 不显示可设 nil
 * @return 默认返回对象不显示, 要显示需要调 show
 */
- (id)initWithFrame:(CGRect)frame initWithTitle:(NSString *)title;

/*
 * 显示
 */
- (void)show;

/*
 * 隐藏
 */
- (void)hide;

/***
 * 重载数据
 */
- (void)reload;
```

属性：

```
/** UIPickerView 控件，用于设置它的 dataSource 和 delegate 属性 */
@property(nonatomic, strong) UIPickerView *pickerView;

/**
 *设置 APPickerView 代理，代理必须实现代理函数：
 * 点取消消息时回调
 *- (void)cancelPickerView:(APPickerView *)pickerView;
 * 点完成时回调，选中项可通过 pickerView selectedRowInComponent 返回
 *- (void)selectedPickerView:(APPickerView *)pickerView;
*/
@property(nonatomic, weak) id<APPickerDelegate> delegate;
```

示例：

```
_pickView = [APPickerView pickerViewWithTitle:nil];
_pickView.delegate = self;
_pickView.pickerView.dataSource = self;
[self.view addSubview:_pickView];
// 设置选中位置
[_pickView.pickerView selectRow:0 inComponent:0 animated:YES];
[_pickView show];
```

展示：



## APSegmentedControl

类名： APSSegmentedControl， 继承 UISegmentedControl

描述： 指定了分段控件 frame 为 CGRectMake(0, 0, 320.0, 44.0)

接口：

```
// 创建并初始化内部 frame 为 CGRectMake(0, 0, 320.0, 44.0)
- (id)initWithItems:(NSArray *)items;
```

## APSmsAuthCodeBox

类名： APSmsAuthCodeBox， 继承 APAuthCodeBox

描述： 短信验证码输入框

接口：

```
/**
 *  创建短信验证码输入框
 *  @param originY    组件的 y 坐标
 *  @param interval   发送短信前的等待时间
 *  @return            短信验证码输入框
 */
- (APSmsAuthCodeBox *)initWithOriginY:(CGFloat)originY
                                Interval:(NSTimeInterval)interval;
```

属性：

```
// 有效时间
@property (assign, nonatomic) NSTimeInterval interval;
// 开始时间，在计时，内部会使用当前时间为属性赋值
// 所以在使用短信验证控件时这个属性并不使用
@property (assign, nonatomic) NSTimeInterval startTime;
```

示例：

```
CGRect frame = CGRectMake(10, originY, 300, 49);
APSmsAuthCodeBox *box = [[APSmsAuthCodeBox alloc] initWithFrame:frame];
box.interval = interval;
box.inputBox.textField.keyboardType = UIKeyboardTypeDecimalPad;
box.inputBox.textField.placeholder = @"短信校验码";
```

## APTextField

类名： APTextField， 继承 UITextField

描述： 输入框控件， 增加了 APTextValidator 验证器的功能。

接口：

```
/**  
 * 创建输入框组件  
 * @param style 文本框样式  
 * @param originY 输入框的 y 坐标  
 * @return 返回输入框组件  
 */  
+ (APTextField *)textFieldWithStyle:(APTextFieldStyle)style originY:(  
  
/**  
 * Initializes and returns an newly created text field object with the  
 *  
 * @return An initialized text field object.  
 */  
- (id)init;  
  
/**  
 * 检查输入的有效性  
 * @return 返回验证结果  
 */  
- (BOOL)checkInputValidity;  
  
/**  
 * 格式化文本  
 * @return 返回格式化后的文本  
 */  
- (NSString *)formatText:(NSString *)text;  
  
/**  
 * 格式化之前的原始文本  
 * @param textField 文本框  
 * @return 返回格式化之前的文本  
 */  
- (NSString *)getOriginTextWithFormatText:(NSString *)formatText;
```

属性：

```
@property(strong, nonatomic) APTextValidator *validator;
@property(strong, nonatomic) NSString *normalizedText;
@property(nonatomic, strong) APTextFieldDelegateProxy *delegate;
//以下两个属性不在推荐使用，因为身份证建议使用 APNumericKeyboard
@property(strong, nonatomic) UIButton *buttonX;
@property(assign, nonatomic) BOOL keyboardTypeIDCard;
```

```
typedef NS_ENUM(NSInteger, APTextFieldStyle){
    APTextFieldStylePlain,
    APTextFieldStyleBordered,
};

typedef NS_ENUM(NSInteger, APKeyboardType){
    APKeyboardTypeIDCard = 100,
}
```

示例：

```
_textField = [[APTextField alloc] initWithFrame:UIEdgeInsetsInsetRect(self.view.bounds, UIEdgeInsetsZero)];
_textField.validator = [APTextValidator mobileNumberValidator];
```

## APTipView

类名： APTipView， 继承 UIView

描述： 提示视图

接口：

```
/**  
 * 创建一个有点击按钮的提示视图  
 * @param frame 在父视图中的位置和大小  
 * @param tip 提示信息  
 * @param title 按钮标题  
 * @return 有点击按钮的提示视图对象  
 */  
- (id)initWithFrame:(CGRect)frame tipMessage:(NSString *)tip title:(NSString *)title;  
  
/**  
 * 创建提示视图  
 * @param frame 在父视图中的位置和大小  
 * @param tip 提示信息  
 * @return 提示视图对象  
 */  
- (id)initWithFrame:(CGRect)frame tipMessage:(NSString *)tip;
```

### 属性：

```
/** APTipViewDelegate 代理，必须实现点击按钮代理方法  
 * - (void)tipButtonClick:(UIButton *)sender;  
 */  
@property (nonatomic, weak) id <APTipViewDelegate> delegate;
```

### 示例：

```
self.tipView = [[APTipView alloc] initWithFrame:self.view.frame tipMessage:@"  
[self.view addSubview:self.tipView];
```

### 展示：



## APToastView

类名： APToastView， 继承 UIView

描述： 提示信息

接口：

```
/**
 * 显示 Toast
 *
 * @param superview 要在其中显示 toast 的视图
 * @param icon 图标类型，支持以下几种
 *   APToastIconNone 无图标
 *   APToastIconSuccess 成功图标
 *   APToastIconFailure 失败图标
 * @param text 显示文本
 * @param duration 显示时长
 *
 */
+ (void)presentToastWithin:(UIView *)superview withIcon:(APToastIcon):
```

```
/**
 * @param superview 要在其中显示 toast 的视图
```

```
* @param text 显示文本
* @return 返回显示的 toast 对象
*/
+ (APToastView *)presentToastWithin:(UIView *)superview text:(NSString *)text;

/*
 * 模态显示提示，此时屏幕不响应用户操作
*/
+ (APToastView *)presentToastWithText:(NSString *)text;

/*
 * 模态toast
 * @param superview 父视图
*/
+ (APToastView *)presentModelToastWithin:(UIView *)superview text:(NSString *)text;

/*
 * 使 toast 消失
*/
- (void)dismissToast;

/***
 * 显示 Toast
 *
 * @param superview 要在其中显示 toast 的视图
 * @param icon 图标类型，支持以下几种
 *   APTToastIconNone 无图标
 *   APTToastIconSuccess 成功图标
 *   APTToastIconFailure 失败图标
 * @param text 显示文本
 * @param duration 显示时长
 * @param completion Toast 自动消失后的回调
 *
 */
+ (void)presentToastWithin:(UIView *)superview withIcon:(APToastIcon):text
duration:(NSTimeInterval)duration completion:(void(^)(void))completion;

/***
 * 显示模态Toast
*/
```

```
*  
* @param superview 要在其中显示 toast 的视图  
* @param icon 图标类型，支持以下几种  
*   APTToastIconNone 无图标  
*   APTToastIconSuccess 成功图标  
*   APTToastIconFailure 失败图标  
* @param text 显示文本  
* @param duration 显示时长  
* @param completion Toast 自动消失后的回调  
*  
*/  
+ (void)presentModalToastWithin:(UIView *)superview withIcon:(APToastIcon)icon  
    text:(NSString *)text duration:(NSTimeInterval)duration completion:(void (^)())completion
```

示例：

```
[APToastView presentToastWithin:weakSelf.view withIcon:ALPToastIconNone  
    text:@"该卡暂不能开通快捷支付,请用其他卡" duration:3.0 completion:nil]
```

展示：



该卡暂不能开通快捷支付,请用其他卡

## APRefreshTableHeaderView

描述：基于开源 EGOResfreshTableHeaderView 封装，下拉刷新控件。

## APTableViewCell

接口：

```
/**  
 * 构造，派生类调用基类构造  
*/  
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)  
    reuseIdentifier;
```

属性：

```
typedef NS_ENUM(NSUInteger, APTableViewCellStyle) {  
    /** Simple cell with text label and optional image view. */  
    APTableViewCellStyleSimple,  
    APTableViewCellStyleSubtitle = UITableViewCellStyleSubtitle,  
};  
  
/**  
 * Default UIEdgeInsetsZero. Add additional padding area around content.  
 */  
@property(nonatomic) UIEdgeInsets contentInset;
```

## APTableViewCellBaseCell



1.1 类名称：APTableViewCellBaseCell

1.2 继承关系： 1.3 使用范围:实现了单行 cell 的大多情况下的需求

1. 操作和行为： 3.1 操作接口：

```
+ (float)cellHeight;  
  
/**  
 * 重置所有子控件，reuse 时应调此方法  
 */  
- (void)reset;  
  
/**  
 * 设置头像图标  
 */  
- (void)setLogoImg:(UIImage *)img;
```

```
/*
 * 注意：需要引入 SDWebImage.framework 才能有效！设置头像为指定 url，并设
 *
 * @param imgUrl      图片url
 * @param defaultImg 默认图片UIImage 实例
 */
- (void)setLogoImgWithUrl:(NSString *)imgUrl withDefault:(UIImage *)de

/*
 * 设置主标题
 */
- (void)setTitle:(NSString* )title;

/*
 * 设置是否显示展开图标
 */
- (void)setExtendLogo;

/*
 * 设置默认背景色
 */
- (void)setNormalBackground:(UIColor *)normalColor;

/*
 * 设置选中背景色
 */
- (void)setSelectedBackground:(UIColor *)selectedColor;

/*
 * 设置提示信息，对标题信息进行补充，靠右显示，有展开图标则显示于展开图标左边
 */
- (void)setInfo:(NSString *)info;

/*
 * 设置提示信息字体颜色
 */
- (void)setInfoFontColor:(UIColor *)color;
```

```
/**  
 * 设计提示图标，位置同上，提示信息和提示图标同时只能设一个  
 */  
- (void)setInfoImg:(UIImage *)img;  
  
/**  
 * 设置开关  
 * @param isOpen 默认开关设置  
 * @param target 开关事件响应对象  
 * @param selector 开关事件响应方法  
 */  
  
- (void)setSwitchWithDefault:(BOOL) isOpne withTarget:(id) target withSelector:(SEL) selector;  
  
/**  
 * 设置具体详情信息，此类无此效果，richcell 可设置  
 */  
- (void)setInfoDetail:(NSString *)detail;  
  
- (BOOL)isShowExtend;  
  
- (void)addView:(UIView *)subView;  
  
- (void)enableLineImageView:(BOOL)yesOrNo;  
  
- (NSInteger )contentViewRightGap;
```

2.2 回调事件： 2.3 其他： 注意：如要使用接口(void)setLogoImgWithURL:(NSString )imgUrl withDefault:(UIImage )defaultImg; 需要引入 SDWebImage.framework 才能有效！设置头像为指定 url，并设置默认头像,defaultImg 不可为空

### 1. 使用示例：

```
- (UITableViewCell*) tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    APTableViewBaseCell* cell = [tableView dequeueReusableCellWithIdentifier:@"APButtonCell"];
    if (cell) {
        [cell reset];
    }
    else {
        cell = [[APTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"APButtonCell"];
        [cell setTitle:@"支付宝"];
        UIImage *img = [UIImage imageNamed:@"icon_58.png"];
        [cell setInfo:@"alipay"];
        [cell setLogoImg:img];
    }
    return cell;
}
```

## APButtonCell

类名： APButtonCell， 继承 UITableviewCell

描述： 带按钮的表格。

接口：

```
/**  
 * 在表格中添加 button 方法  
 */  
- (void)addButton:(APButton *)button;  
  
/**  
 * 在表格中添加一个制定样式，点击事件的 button  
 */  
- (APButton *)addButtonType:(APButtonType)buttonType title:(NSString *)title target:(id)target selector:(SEL)selector;
```

示例：

```
APButtonCell *buttonCell = ??[APButtonCell alloc] initWithFrame:CGRectMake(10, 10, 100, 100);  
APButton *button = ??[APButton alloc] initWithFrame:CGRectMake(10, 10, 100, 100);  
[button setTitle:@"确定"];  
?[buttonCell addButton:button];  
[buttonCell addButtonWithType:APButtonTypeCustom title:@"确定" target:nil selector:@selector(buttonClicked)];
```

## APInputBoxCell

类名： APInputBoxCell， 继承 UITableviewCell

描述： 带按钮的表格。

接口：

```
/**  
 *控件高度  
 */  
+ (float)cellHeight;  
  
/**  
 * 返回 cell 中的输入框  
 */  
- (APIInputBox *)textFieldInCell;
```

示例：

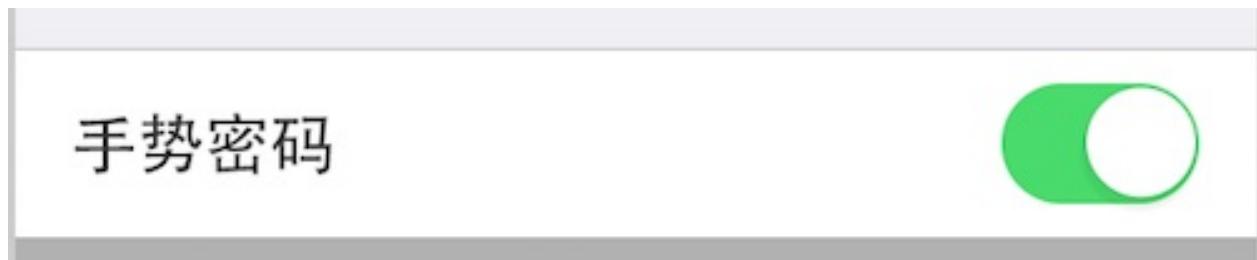
```
APIInputBoxCell *inputBoxCell = ??[APIInputBox alloc] initWithFrame:UIT:  
APIInputBox *box = [inputBoxCell textFieldInCell];
```

## APSwitchCell

1. 控件的描述： 1.1 类名称： APSwitchCell

1.2 继承关系： 1.3 使用范围:用于设置

1. 界面属性 2.1 展现效果：



1. 操作和行为： 3.1 操作接口：

```
/*
 *  返回 cell 高度
 *
 *  @return 高度值
 */
+ (float)cellHeight;

/**
 *  初始化
 *
 *  @param reuseIdentifier cell 复用 id
 *
 *  @return APSwitchCell
 */
- (id)initWithReuseIdentifier:(NSString *)reuseIdentifier;

/**
 *  UISwitch 控件
 */
@property (nonatomic, readonly) UISwitch *switchControl;
```

### 3.2 回调事件: 3.3 其他:

#### 1. 使用示例:

```
APSwitchCell *cell = nil;
identifier = @"switch";
cell = [_tableView dequeueReusableCellWithIdentifier:identifier];
if (!cell) {
    cell = [[APSwitchCell alloc] initWithReuseIdentifier:identifier];
    cell.textLabel.text = @"手势密码";
    cell.selectionStyle = UITableViewCellStyleNone;
    _tableView.separatorStyle = UITableViewCellStyleSingleLine;
}
```

## APITableViewDoubleLineCell

类名： APITableViewDoubleLineCell， 继承 APITableViewCell

描述： 带两条线的 cell

接口：

```
/**  
 * APITableViewDoubleLineCell 初始化函数，设置的 Style 为 UITableViewCe  
 * 推荐使用该接口初始化  
 * @param reuseIdentifier 重用标记  
 * @return 返回初始化的实例  
 */  
- (id)initWithReuseIdentifier:(NSString *)reuseIdentifier;  
  
/**  
 * 返回 cell 的高度  
 * @return cell 的高度  
 */  
+ (float)cellHeight;  
  
/**  
 * cell 的重置 */  
- (void)reset;
```

示例：

```
APITableViewDoubleLineCell *cell = nil;  
identifier = @"DoubleLineCell";  
cell = [tableView dequeueReusableCellWithIdentifier:identifier];  
if (!cell) {  
    cell = [APITableViewDoubleLineCell alloc initWithReuseIdentifier:ident:  
    cell.selectionStyle = UITableViewCellSelectionStyleNone;  
}
```

## APITableViewTwoTextCell

1.1 类名称： APITableViewTwoTextCell

1.2 继承关系： 父类： APITableViewCell

1.3 使用范围： 含有左右两个文字标签的 cell。

两边的标签会根据文字的长度自动左右对齐。

1. 界面属性 2.1 展现效果：

卡类型

招商银行 储蓄卡

1. 使用示例：

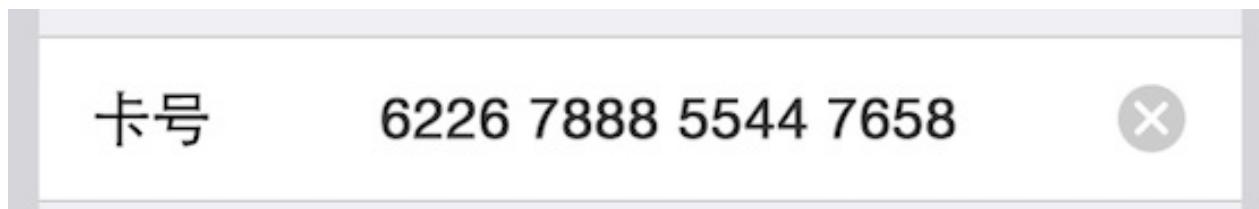
```
cell = [[APITableViewTwoTextCell alloc]  
initWithStyle:UITableViewCellStyleDefault  
reuseIdentifier:cellId  
leftText:@"左边的文字"  
rightText:@"右边的文字"]];
```

## APTextFieldCell

1.1 类名称： APTextFieldCell

1.2 使用范围:用于表格单元中的输入框场景

1.3 展现效果：



1.4 接口：

```
@property(nonatomic, strong, readonly) APTextField *textField;  
@property(nonatomic, assign) CGFloat textLabelWidth;
```

@cnName 描述文件 @priority 1

# 描述文件

[TOC]

## 1 简介

通用控件支持主题式管理，每个主题为一个独立的 bundle。这个目录里面包含主题描述文件 `theme.plist` 和其它文件、图片资源。

主题目录可以放在Main Bundle里随应用下发，也可以后期下载后放在 Documents里。

## 2 类接口介绍

### 2.1 APTHEMEManager

```
#define APDefaultTheme [APThemeManager sharedInstance].defaultTheme
#define APCurrentTheme [APThemeManager sharedInstance].currentTheme

@interface APThemeManager : NSObject

@property (nonatomic, strong, readonly) APTheme* defaultTheme;
@property (nonatomic, strong, readonly) APTheme* currentTheme;

+ (instancetype)sharedInstance;

/***
 * 加载主题，并作为Current Theme
 *
 * @param path 主题.bundle的资源路径
 */
- (void)loadThemeWithPath:(NSString*)path;

@end
```

`defaultTheme` : 默认主题，放在Main Bundle里，名字为 `UIDefaultTheme.bundle` 。 `currentTheme` : 当使用默认主题时同 `defaultTheme` 。

## 2.2 APThemeValuePath

```
@protocol APThemeValuePath <NSObject>

- (NSString*)stringForPath:(NSString*)path;
- (UIColor*)colorForPath:(NSString*)path;
- (NSInteger)intForPath:(NSString*)path;
- (float)floatForPath:(NSString*)path;
- (BOOL)boolForPath:(NSString*)path;
- (UIImage*)imageForPath:(NSString*)path;
- (UIFont*)fontForPath:(NSString*)path;

// 某个路径的值是否在配置文件里定义了
- (BOOL)definedForPath:(NSString*)path;

@end
```

@path参数为 . 分隔的路径，对应 theme.plist 的字典树结构。

## 2.3 APTheme

```
@interface APTheme : NSBundle <APThemeValuePath>

@property (nonatomic, strong, readonly) NSString* name;
@property (nonatomic, strong, readonly) NSDictionary* theme;
@property (nonatomic, assign, readonly) BOOL inherited; // 当某个值找不到父级时返回这个值

- (instancetype)initWithPath:(NSString*)path;

- (APThemeFetch*)fetchForPrefix:(NSString*)prefix;

@end
```

APTheme实现 APThemeValuePath 这个接口的方法。

## 2.4 APThemeFetch

```
@interface APThemeFetch : NSObject <APThemeValuePath>
@end
```

APThemeFetch实现 APThemeValuePath 这个接口的方法。这个类由 APTheme 的 fetchForPrefix 方法创建。用法如下：

```
APThemeFetch* fetch = [APCurrentTheme fetchForPrefix:@"APNavigationBar"]

if ([fetch definedForPath:@"Image"]) {
    UIImage* image = [[fetch imagePath:@"Image"] stretchableImageWithLeftCapWidth:0 topCapHeight:0];
    if (image)
        [[UINavigationBar appearance] setBackgroundImage:image forBarMetrics:UIBarMetricsDefault];
}

if ([fetch definedForPath:@"BarTintColor"]) {
    [[UINavigationBar appearance] setBarTintColor:[fetch colorForPath:@"BarTintColor"]];
}
```

上面 fetch 获取的数据实际为：

```
APNavigationBar.Global.Background.Image
APNavigationBar.Global.Background.BarTintColor
```

### 3 描述文件格式

描述文件为一个 plist 文件，根节点下面预定义了 Name 和 Constants 两项。其它项目可以自定义，但通常为每个 UI 组件建立一个节点，并保证下级属性定义结构合理。



Key	Type	Value
Root	Dictionary	(4 items)
Name	String	AlipayDefault
Constants	Dictionary	(1 item)
AntBlue	String	#00aaee
StatusBarStyle	Number	1
APNavigationBar	Dictionary	(3 items)
Global	Dictionary	(2 items)

**Name** : 主题的名称

**Constants** : 这个配置文件可以使用的常量，可以理解为代码中的宏。例如主题会有一些公共的颜色值，字号定义，建议将这些定义抽出来放在 Constants里。这样修改起来会更加方便。比如上图定义了AntBlue这个颜色值的常量，那么配置文件里就可以使用 \$AntBlue 来引用这个常量，而不需要每个使用这个颜色值的地方都写 #00aaee。

## 3.1 逻辑判断

有些时候，我们对不同设备可能使用不同的图片，不同的字号。只需要在配置文件里将指定路径的值的类型修改为 **Dictionary**，并在这个字典里定义 **Default** 值和其它逻辑判断条件满足时的值即可。比如：

▼ Font	Dictionary	(2 items)
Bold	Boolean	YES
▼ Size	Dictionary	(2 items)
screen>=phone6p	String	20
Default	String	17

这个图里定义的字体，当设备屏幕不小于iPhone6 plus时，使用20号字，其它情况都使用17号。如果定义了多个判断条件，那么当设备参数都符合时结果是不确定的。

目前支持的判断方法如下：A?B。其中A支持的情况为

主语 (A)	支持 的运 算符 (?)	值的情 况 (B)	例子	说明
screen	<, <=, =, > >=, !=	pad_low pad_retina phone_low phone4 phone5 phone6 phone6p	screen>phone5 screen<=phone6p	pad_low表示低分辨率的iPad屏幕， phone_low表示低分辨率的iPhone屏幕。 iPhone屏幕和iPad屏幕不能比较大小。比如本设备是iPad4， 虽然为高分屏，但screen>phone4这个表达式是不会被解析为True的。
ios	<, <=, =, > >=, !=	8 8.2 9.0.1	ios<8.0 ios>=9.0.1	表达式右边的版本表达式最多支持三段结构，只写一段或两段也可以。逐位进行比较。
device	=, !=	iphone ipad	device=iphone device!=ipad	只支持等于或不等于两个运算符

## 3.2 颜色定义

支持 #rrggbba 和 #rrggbbaaa 两种格式定义颜色，使用16进制。如果没指定 aa 分量，会默认为1.0，也就是不透明。

## 3.3 字体定义

可以使用 `APThemeValuePath` 接口的 `fontForPath:` 方法直接获取一个`UIFont*` 对象。字体下面目前支持 `Bold` 和 `Size` 两个属性。其中 `Bold` 为可选的，默认为NO； `Size` 为字号，定义为`String`或`Number`都行。如图：

▼ Font	Dictionary	(2 items)
Bold	Boolean	YES
▼ Size	Dictionary	(2 items)
screen>=phone6p	String	20
Default	String	17

## 3.4 图片定义

你可以从配置文件里获得字符串并自己加载图片，不过推荐使用 `imageForPath` 接口来加载图片。在配置文件里只需要配置图片相对主题 `Bundle` 的路径即可。

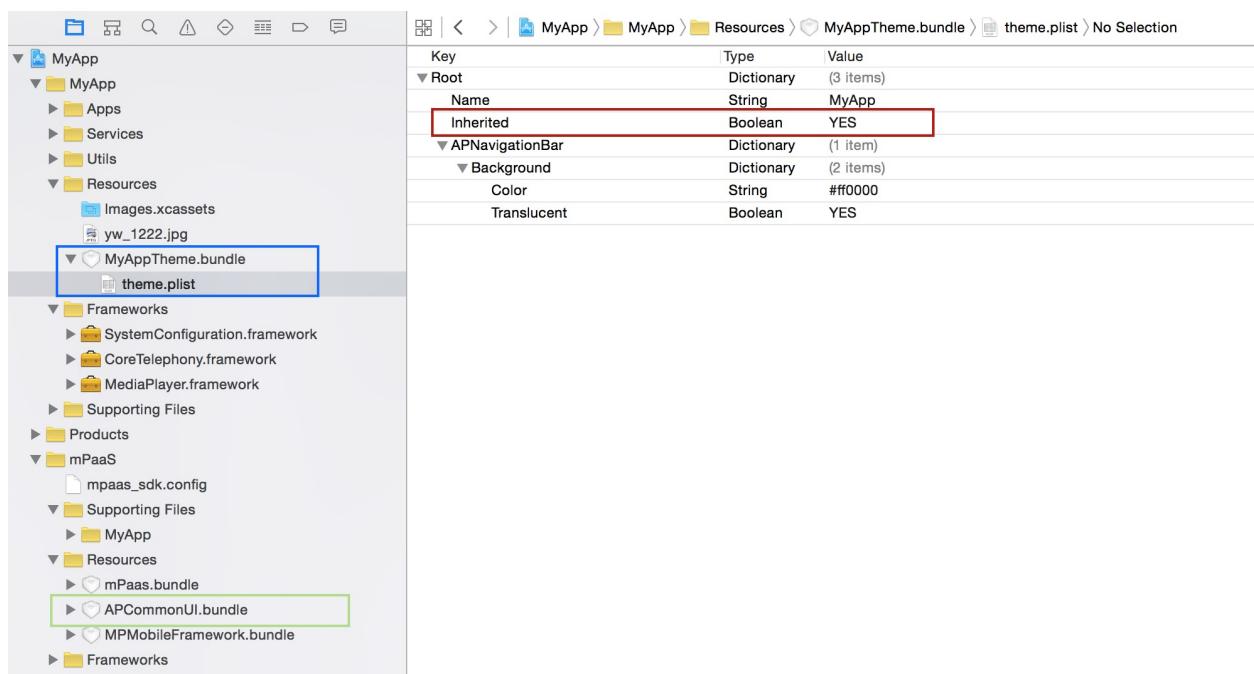
@cnName 自定义主题 @priority 2

## 自定义主题

MPCommonUI 模块中会自带一份默认主题。当接入方应用有自己的样式需求时，可以创建自己的主题资源目录，并使用 APTHEMEManager 加载相应主题即可。

APCommonUI.bundle 这个目录是通用控件模块自带的资源包，里面包含了默认主题。请尽量不要修改这个目录里的图片与主题文件，以防升级 MPCommonUI 模块时带来不必要的资源覆盖问题。

如下图：



绿色部分为移动框架自动维护的目录，里面有默认的资源包，这里不要修改。我们在自己的工程中添加一个资源包，同时创建一个`theme.plist`文件，内容如图蓝色。在红色部分定义：

```
Inherited = YES
```

表示这个主题是继承自默认主题的，对该主题文件中未定义的值，会使用 APCommonUI.bundle 中的默认主题的值。

在程序启动的时候加载我们的主题（这个代码通常可以放在应用的 mPaasAppInterface 实现类的 init 方法中）：

```
[ [APThemeManager sharedInstance] loadThemeWithPath:[ [NSBundle mainBundle] bundlePath] themeName:@"APTheme"]];
```

@cnName APNavigationBar @priority 3

# APNavigationBar

当使用通用控件库时，客户端框架会使用APNavigationBar作为自己的导航条类。下面 [ ] 中间为可选属性。

```
StatusBarStyle
APNavigationBar
    Global
        Title
            TextColor
            Font
                [Bold]
            Size
        Background
            [Image]
            [TintColor]
            [BarTintColor]
            [Translucent]
    Button
        [BackImage]
        TextColor
        Font
            [Bold]
        Size
    Background
        Color
        Translucent
        [Image]
```

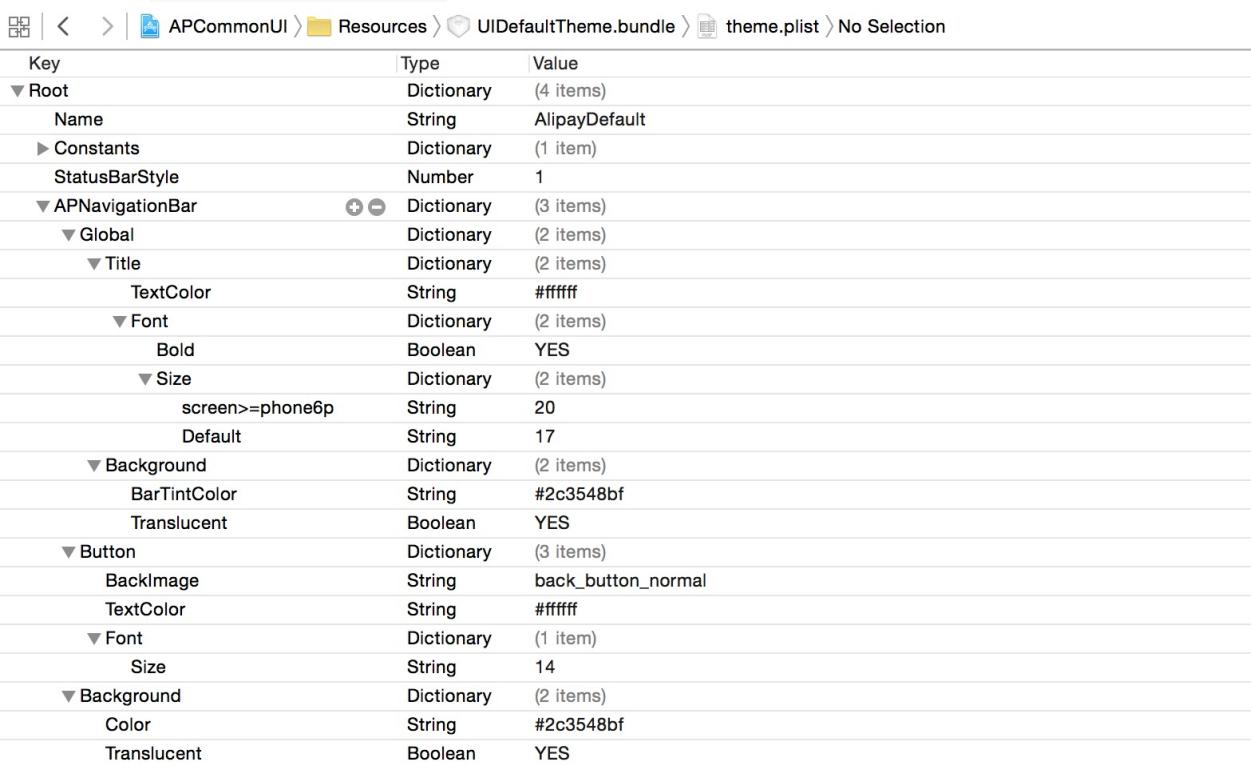
`StatusBarStyle`：因为在iOS上导航条与状态条的颜色搭配息息相关，所以与APNavigationBar并列可以定义状态条的样式。取值详见系统 `UIStatusBarStyle` 枚举的定义。0为深色样式，用于淡颜色的导航条；1为浅色样式，用于深色的导航条。

**Global** : `UINavigationBar`控件全局设置，影响应用的所有`UINavigationBar`及其的子类。通常用来统一系统提供的窗口的导航条样式，如通讯录联系人选择、选择相册照片的窗口。`Global.Title` 设置导航条中间的标题字体与颜色，`Global.Background` 设置导航条背景，可以设置图片或指定颜色。

**Button** : 导航条的返回按钮属性。`BackImage` 为返回按钮的图片，可以不指定而使用系统默认样式。

**Background** : 除了上面 `Global` 设置全局的导航条外，可以单独指定 `APNavigationBar` 的背景。

一个定义 `APNavigationBar` 的样例如下图：



The screenshot shows the Xcode Property List Editor with the file `theme.plist` open. The structure is as follows:

Key	Type	Value
Root	Dictionary	(4 items)
Name	String	AlipayDefault
Constants	Dictionary	(1 item)
StatusBarStyle	Number	1
APNavigationBar	Dictionary	(3 items)
Global	Dictionary	(2 items)
Title	Dictionary	(2 items)
TextColor	String	#ffffff
Font	Dictionary	(2 items)
Bold	Boolean	YES
Size	Dictionary	(2 items)
screen>=phone6p	String	20
Default	String	17
Background	Dictionary	(2 items)
BarTintColor	String	#2c3548bf
Translucent	Boolean	YES
Button	Dictionary	(3 items)
BackImage	String	back_button_normal
TextColor	String	#ffffff
Font	Dictionary	(1 item)
Size	String	14
Background	Dictionary	(2 items)
Color	String	#2c3548bf
Translucent	Boolean	YES

@cnName 3 IconFont @priority 3

## 3 IconFont

[TOC]

### 3.1 简介

iOS 应用会携带许多 PNG 格式的图标资源，还要对不同屏幕尺寸打包多套资源。为了减少图标资源、提升显示效果，IconFont 基于适量图形，提供部分图标资源的替代方案，有效减少安装包大小。

IconFont 的源文件为 ttf 格式的字体文件，ttf 文件里描述了每个图标的矢量信息。在使用时，mPaaS 会将 ttf 字体加载出来，业务代码便可以绘制图标。

IconFont 主页，内有大量资源下载：<http://iconfont.cn/>

### 3.2 使用方法

#### 3.2.1 添加 IconFont

##### 3.2.1.1 描述文件

引入 mPaaS 通用控件模块 MPCommonUI 后，需要将 APCommonUI.bundle 添加到工程里。这个 Bundle 下有 `ap_iconfont_name.plist` 文件来配置 IconFont。

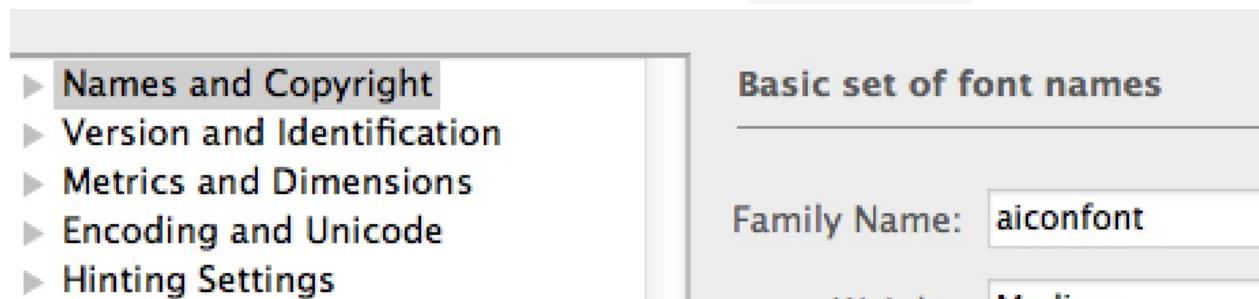
每个资源包括两个字段，分别为字体文件与配置文件，如图：

Key	Type	Value
Root	Dictionary	(1 item)
iconfont	Dictionary	(2 items)
file	String	iconfont
config	String	iconfontconfig

资源名为字体的 Family Name， file 为字体 ttf 文件相对 main bundle 的位置， config 是描述文件相对 main bundle 的位置，注意都不需要添加扩展名。

### 3.2.1.2 查看字体的 Family Name

使用 FontLab Studio 打开 ttf 文件，查询它的 Family Name。



### 3.2.2 IconFont 接口

绘制 IconFont 时的 text 文本值为 8 位的 Unicode 字符。mPaaS 提供了 UIView 扩展来方便的显示 IconFont。

```
@interface APIIconFont : NSObject

//使用默认 ttf(配置文件第一项)
+ (UIFont *)iconFont;
+ (UIFont *)iconFontWithSize:(NSInteger)fontSize;

+ (UIFont *)iconFontWithFontName:(NSString *)fontName;
+ (UIFont *)iconFontWithFontName:(NSString *)fontName size:(NSInteger)fontSize;

@end

@interface UIView (APIIconFont)

//使用默认 ttf(配置文件第一项)
+ (UIView *)iconFontViewWithPoint:(CGPoint)point name:(NSString *)name;
+ (UIView *)iconFontViewWithPoint:(CGPoint)point name:(NSString *)name size:(NSInteger)fontSize;

+ (UIView *)iconFontViewWithPoint:(CGPoint)point name:(NSString *)name fontName:(NSString *)fontName;
+ (UIView *)iconFontViewWithPoint:(CGPoint)point name:(NSString *)name fontName:(NSString *)fontName size:(NSInteger)fontSize;

- (BOOL)setIconFontColor:(UIColor *)color;
- (BOOL)setIconFontName:(NSString *)name;

@end
```

### 3.2.3 绘制 IconFont

```
UILabel * label = [[UILabel alloc] init];
label.text = @"\U00003439";
label.font = [APIIconFont iconFont];
[self.view addSubview:label];

UIView *fontView = [UIView iconFontViewWithPoint:CGPointMake(100, 100)
//通过 frame 动态设置 icon 位置和大小
fontView.frame = CGRectMake(100, 100, 48, 48);
[self.view addSubview:fontView];

fontView = [UIView iconFontViewWithPoint:CGPointMake(130, 100) name:@"yuebao"]
//重置颜色
[fontView setIconFontColor:[UIColor redColor]];
//重置图片
[fontView setIconFontName:@"yuebao"];
[self.view addSubview:fontView];
```

@cnName 声波通讯组件 @priority 8

# 声波通讯组件

[TOC]

## 1 用途与功能

声波近场通讯技术，使具备喇叭、麦克风的终端实现了通过声波进行数据通讯的功能。

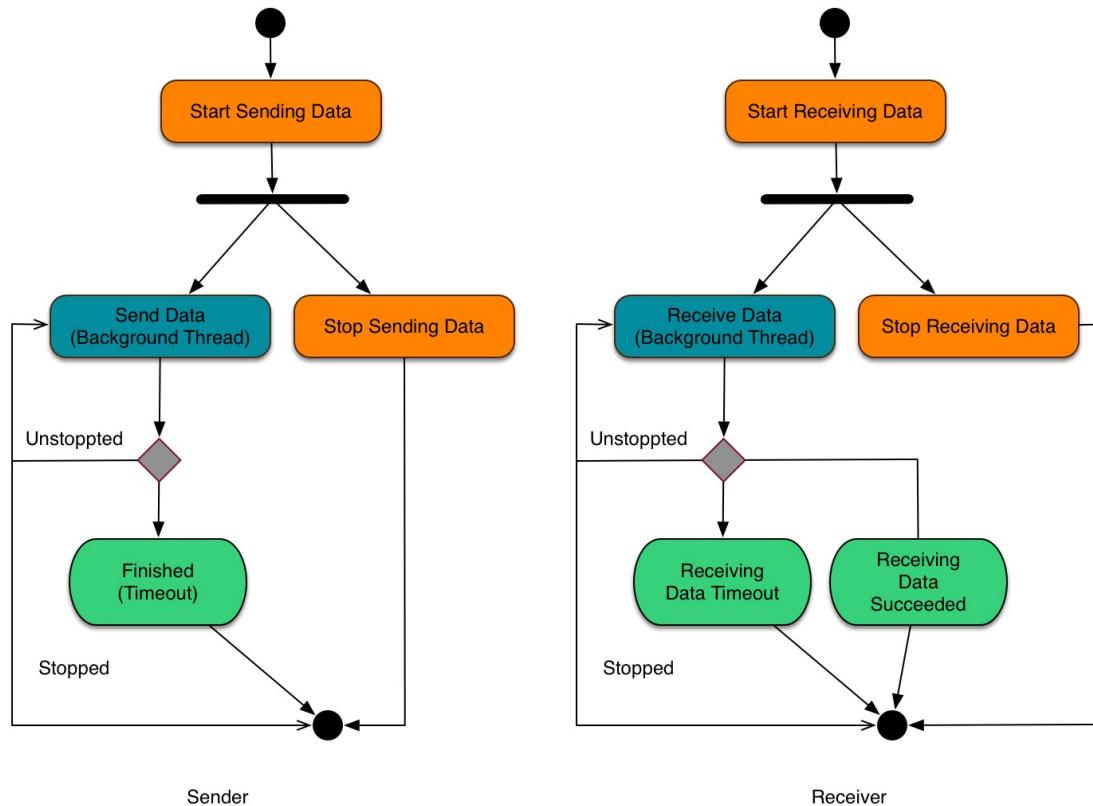
声波近场通讯技术不需要改造硬件，采用纯软件方式，部署成本低，能够快速推广。可广泛应用于各个应用场景中，示例如但不限于：

- 手机与手机之间建立会话，实现现场支付的收款、付款功能；
- 手机与收银 PC（或平板电脑）之间建立会话，实现现场支付的收款、付款功能；
- 手机与 PC 之间建立会话，实现通过手机支付网购货款的功能；
- 手机与商户的签到 PC（或签到模块）之间建立会话，实现精准签到功能；
- 手机与手机的蓝牙配对；
- 手机与手机之间建立会话，交换名片、照片、音乐等信息。

SonicWaveNFC SDK（后简称 SDK）是为开发声波近场通讯技术相关应用提供的开发工具。为开发创新性的移动互联应用、解决方案提供了最佳的功能。

## 2 通讯方式

程序采用半双工（half-duplex）方式通讯，允许两台终端之间的双向数据传输。同一时间只由一台终端发送数据，若另一台终端要发送数据，需等原来发送数据的终端发送完成后再处理。下图表示了程序进行一次通讯时的处理过程：



发送方为使用喇叭发出声波信号发送数据的终端，接收方为使用麦克风录取声波信号接收数据的终端。

**发送方的处理过程如下：**

- 1、应用系统调用开始发送数据方法；
- 2、程序将在后台线程中执行发送数据处理，循环执行；
- 3、应用系统在超时或其他应用事件触发时，调用停止发送数据方法；
- 4、程序在后台线程中收到停止发送数据指令，停止执行；
- 5、结束发送数据过程。

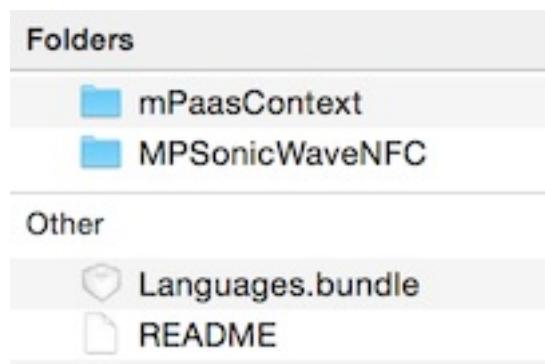
**接收方的处理过程如下：**

- 1、应用系统调用开始接收数据方法；

- 2、程序在后台线程中执行接收数据处理，循环执行；
- 3、若程序在执行接收数据处理时，数据被成功接收，则停止执行，向应用系统发出数据被成功接收的事件回调，结束接收数据过程；
- 4、若程序在执行接收数据处理时，数据接收超时，则停止执行，向应用系统发出数据接收超时的事件回调，结束接收数据过程；
- 5、应用系统在其他事件应用事件触发时，调用停止接收数据方法；
- 6、程序在后台线程中收到停止接收数据指令，停止执行；
- 7、结束接收数据过程。

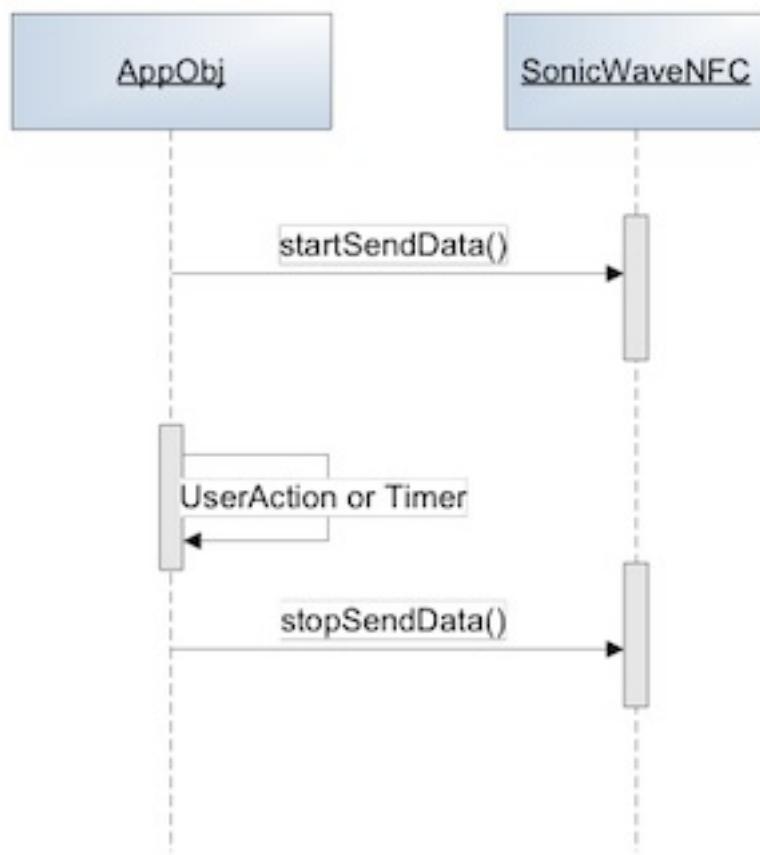
## 3 使用方法

声波通讯组件为 MPSonicWaveNFC，打包时勾选后生成 mPaas.framework，同时需要将 mPaas.bundle 添加到工程中，里面有默认的音效文件。



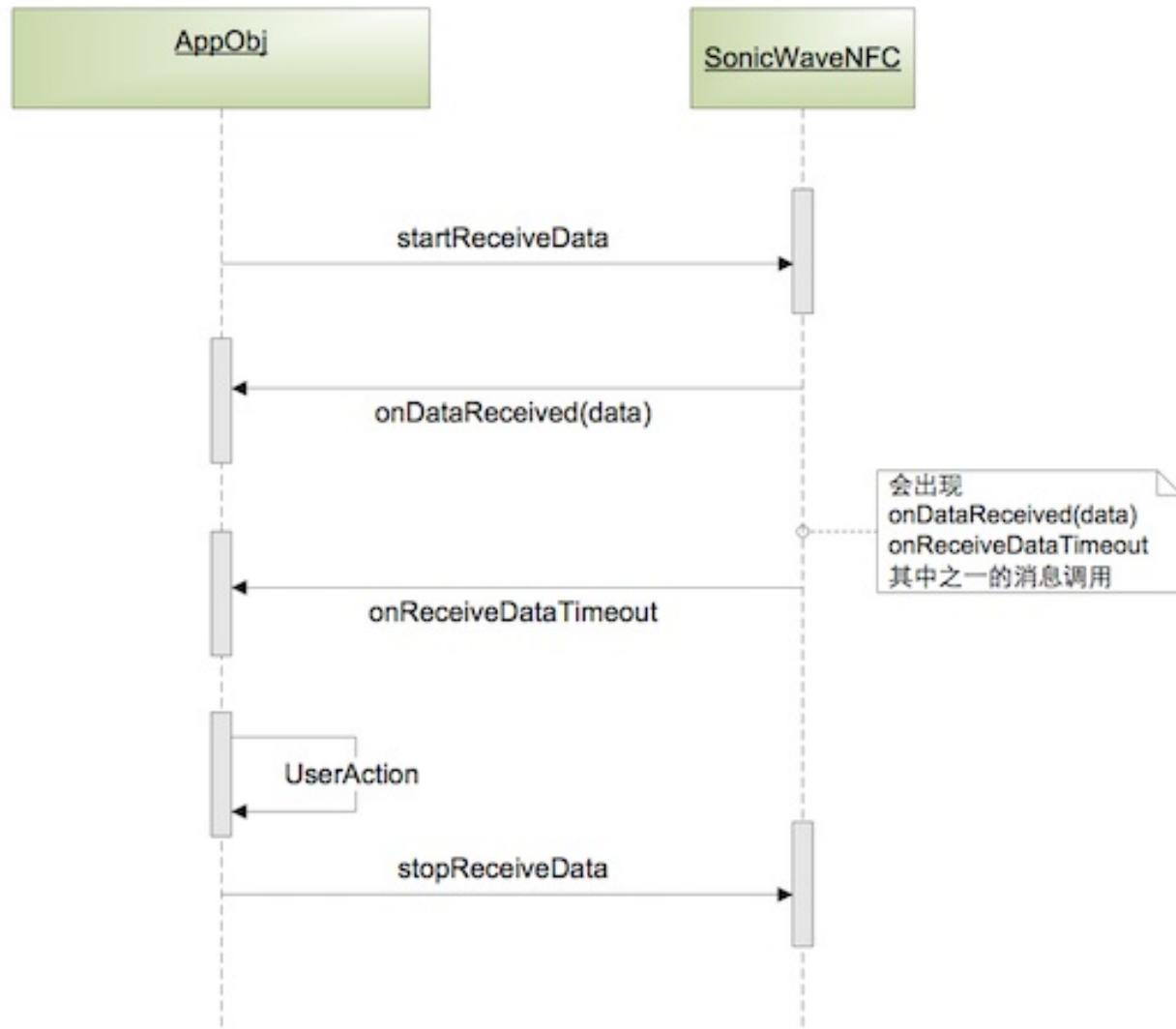
### 3.1 发送数据

其中，AppObj 为应用系统中某容器类的实例或某页面对象。



## 3.2 接收数据

其中， AppObj 为应用系统中某容器类的实例或某页面对象。



## 4 接口说明

### 4.1 SonicWaveNFC

#### hasHeadset

描述：判断外放耳麦是否插入

返回：外放耳麦是否插入

#### startSendData

描述：开始发送数据

参数：

`strData` 需要发送的数据，最长 64 个，该值为空或 `null` 时，则只发背景音，不发送数据。

`iTimeoutSeconds` 发送数据的超时秒数，建议值为 5 秒

`iSoundType` 发送数据时的发声模式

`NFC_SENDDATA_SOUNDTYPE_SILENT` 0 无声模式

`NFC_SENDDATA_SOUNDTYPE_NOISY` 1 有声模式

`NFC_SENDDATA_SOUNDTYPE_MIX` 2 混合背景提示音模式

`iVolume` 发送数据时的音量，有效值为 0-100，建议值为 80

`handler` 声波近场通讯的回调处理接口 `SonicWaveNFCHandler` 实现对象

返回：开始发送数据是否正常

提示：当发送的数据都为数字、小写字母时，传输速度更快。

## **stopSendData**

描述：停止发送数据

## **setBkSoundWave4MixFromRes**

描述：设置使用的混合背景提示音

参数：

`resName` 资源名称，参见 `pathForResource` 的同名参数

`ofType` 资源类型（扩展名），参见 `pathForResource` 的同名参数

返回：是否设置成功

## **startReceiveData**

描述：开始接收数据

参数：

`iTimeoutSeconds` 接收数据的超时秒数

`iMinAmplitude` 接收数据时幅度值的最小门限值，低于该门限值的音频内容将被忽略，

`handler` 声波近场通讯的回调处理接口 `SonicWaveNFCHandler` 实现对象

返回：开始接收数据是否正常

## **stopReceiveData**

描述：停止接收数据

## **4.2 SonicWaveNFCHandler**

### **onSendDataStarted**

描述：发送数据正常开始

## **onSendDataTimeout**

描述：发送数据超时

## **onSendDataFailed**

描述：发送数据失败

参数：

`iErrorCode` 失败的原因码

## **onReceiveDataStarted**

描述：接收数据正常开始

## **onDataReceived**

描述：接收数据成功

参数：

`strData` 接收到的数据

## **onReceiveDataTimeout**

描述：接收数据超时

## **onReceiveDataFailed**

参数：

`iErrorCode` 失败的原因码

## **onHeadsetOn**

描述：发送数据或接收数据过程中，外放耳麦被插入了

## **onHeadsetOff**

描述：发送数据或接收数据过程中，外放耳麦被拔出了

@cnName 多语言组件 @priority 9

# 多语言组件

[TOC]

## 1 简介

MPLanguageSetting 用于实现 App 内语言环境切换，不依赖于系统语言设置。具有以下特点：

1. 可无需调用方关心地自动切换常量字符的语言.
2. 对于参数化的字符串,提供给调用方回调方法.
3. 如还有特殊需求，也可以监听语言切换时框架发出的通知.

## 2 使用方法

### 2.1 将单一语言应用切换为多语言

1. 将 mPaas.framework，以及资源文件 MPLanguage.bundle 引入工程文件中，这个 bundle 在 mPaas.framework 目录下，需要应用拷贝出来。
2. 代码中需要本地化的字符串使用 NSLocalizedString 宏编写，以便于生成.strings 文件. 如:

```
// 建议 key 值使用 "module.content" 这种形式,而不要直接用 "content" ,  
  
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    CustomTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"CustomCell" forIndexPath:indexPath];  
  
    cell.titleLabel.text = NSLocalizedString(@"app1.table.cell.title");  
    cell subTitleLabel.text = NSLocalizedString(@"app1.table.cell.subtitle");  
    cell.textField.placeholder = NSLocalizedString(@"app1.table.cell.textFieldPlaceholder");  
  
    return cell;  
}
```

使用 genstrings 抽取代码中需要本地化的文本，生成.strings 文件.

```
// 打开终端，找到工程所在目录根目录下 执行：  
  
find ./ -name "*.m" -print0 | xargs -0 genstrings -o .
```

3.执行完成后,会在当前目录下生成一个 Localizable.strings 文件 . 内容类似如下:

```
/* comment here. */  
"app1.table.cell.subTitle" = "app1.table.cell.subtitle";  
  
/* comment here. */  
"app1.table.cell.textfield" = "app1.table.cell.textfield";  
  
/* comment here. */  
"app1.table.cell.title" = "app1.table.cell.title";
```

4.将 Localizable.strings 文件中的内容复制到 MPLanguage.bundle/下的.strings 文件下, 并翻译成对应语言. 如,将 Localizable.strings 文件内容复制到 MPLanguage.bundle/zh\_CN.strings 中(后续我们会提供工具帮助完成),

并翻译成中文：

```
// Filename: zh_CN.strings

/* comment here. */
"app1.table.cell.subTitle" = "子标题";

/* comment here. */
"app1.table.cell.textfield" = "点击后输入";

/* comment here. */
"app1.table.cell.title" = "标题";
```

5.唤起语言设置页面. 有两种方式：（1）通过 startApp 方式，（2）通过 MPLanguageSetting 提供的语言设置 VC

（1）通过 startApp 方式：

在 MobileRuntime.plist 中添加一个 `MPLanguageSettingrAppDelegate`，并设置 AppID

MobileRuntime.plist		
Key	Type	Value
Root	Dictionary	(3 items)
Launcher	String	20000001
Services	Array	(1 item)
Applications	Array	(3 items)
Item 0	Dictionary	(3 items)
delegate	String	DemoAppDelegate
description	String	Demo
name	String	20000002
Item 1	Dictionary	(3 items)
delegate	String	LauncherAppDelegate
description	String	启动窗
name	String	20000001
Item 2	Dictionary	(3 items)
delegate	String	MPLanguageSettingrAppDelegate
description	String	LanguageSetting
name	String	20000003

在需要唤起设置语言页面的位置调用 `[DTContextGet()  
startApplication:@"20000003" params:@{} animated:YES];`

（2）使用提供的 `MPLanguageManageController` 方式：

```
MPLanguageManageController *settingVC = [[MPLanguageManageController alloc] init];
// push settingVC 或 present settingVC
```

至此，工程已经可以自动处理常量字符串的语言切换了。

## 3 进一步使用

1. 获得当前语言设置环境下的文本内容：

```
//使用 MPLanguageSettings 提供的宏 __TEXT(key, comment)

- (void)tabBarController:(UITabBarController *)tabBarController didSelectViewController:(UIViewController *)viewController
{
    self.title = __TEXT(viewController.title,@""); // 以 viewController 为基类
}
```

1. 实现 - (void)languageSettingDidChange:(NSDictionary \*)info

```
NS_REQUIRE_SUPER; .
```

```
//DTVViewController 类中实现了对语言切换通知的监听，子类可以实现 `languageSettingDidChange:userInfo` 方法

// DemoAppViewController.m

- (void)languageSettingDidChange:(NSDictionary *)info
{
    [super languageSettingDidChange:info];
    if (![info[APLanguageSettingInfoNewKey] isEqualToString:info[APLanguageSettingInfoOldKey]])
        self.title = __TEXT(@"app.sub.app1", @"子 App");
    // update constraints
    // update image resources , etc.
}
 NSLog(@"DemoAppVC");
}
```

## 1. 监听语言切换时的通知:

字段	意义	备注
APLanguageSettingDidChangeNotification	语言切换时的通知名	
APLanguageSettingInfoOldKey	通知中 userInfo 获取原语言名的 key	
APLanguageSettingInfoNewKey	通知中 userInfo 获取新的语言名的 key	新的语言名可能与原语言名一致

@cnName 扫码组件 @priority 10

# 扫码组件

[TOC]

## 1 简介

MPScanCode 模块整合了条码、二维码、信用卡等扫描功能，提供快速、低 CPU 开销的扫码服务，所有功能整合进入一个界面，应用可以选择开启的扫码方式。

## 2 使用方法

推荐使用移动客户端框架为扫码功能创建一个微应用，并通过框架接口启动扫码服务。

```
@interface ScanCodeDelegate () <BumpControllerDelegate>

@property(nonatomic, strong) UIViewController *rootController;

@end

@implementation ScanCodeDelegate

- (UIViewController *)rootControllerInApplication:(DTMicroApplication *)
{
    return self.rootController;
}

- (void)application:(DTMicroApplication *)application willStartLaunch:
{
    self.rootController = [[ScanCodeViewController alloc] initWithNibName:
        ((ScanCodeViewController*)self.rootController).bumpDelegate = self;
}

@end
```

## 配置 MobileRuntime.plist

The screenshot shows the Xcode Property List Editor with the following details:

- Path: MobileFramework > MobileFramework > Supporting Files > MobileRuntime.plist
- Search bar: scan
- Table view:

Key	Type	Value
Root	Dictionary	(3 items)
Applications	Array	(18 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
Item 2	Dictionary	(3 items)
delegate	String	ScanCodeDelegate
description	String	扫码
name	String	20000016

## 启动扫码

```
[DTContextGet() startApplication:@"20000016" params:nil animated:YES]
```

@cnName 1 概述 @priority 1

# 1 概述

移动开发者工具是一套组件，包括命令行工具，Xcode插件等。辅助开发者进行开发。

在终端里运行下面命令行安装移动开发者工具

```
sh

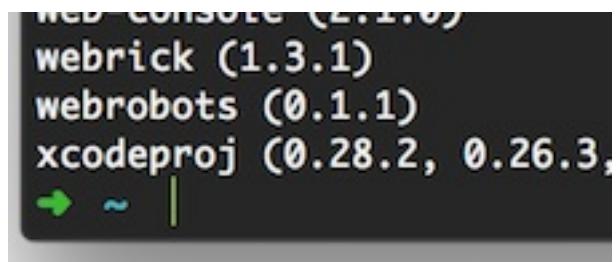
shenmo — shenmo@shenmoMacBookPro: ~ — zsh — 99x44
Last login: Wed Oct 14 20:57:17 on ttys000
→ sh <(curl -s http://code.taobao.org/svn/mpaaskit/trunk/install.sh)
mpaaskit_installer
* 欢迎使用mpaaskit (海纳平台iOS开发组件)
- 1. 执行环境检测
- 检查 git: git version 2.3.8 (Apple Git-58)
done.
- 2. 安装 mpaaskit
- 最新版本为 0.0.1
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload   Total Spent  Left Speed
100 3153  100 3153    0      0  5707      0 --:--:-- --:--:-- --:--:--  5711
/Users/shenmo
mpaaskit install_path: /Users/shenmo/.mpaaskit
http://code.taobao.org/svn/mpaaskit/tags/0.0.1.zip
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload   Total Spent  Left Speed
100 10.4M  100 10.4M    0      0  552k      0  0:00:19  0:00:19 --:--:--  733k
- install mPaaS templates....
- install mPaaS plugins
```

请同时运行一下下面命令，更新一下 Ruby Gems

```
sudo gem update
```

保证 xcdeproj 处于比较新的版本（2015年10月最新版本为0.28.2），才可以编辑修改最新的Xcode工程文件。运行命令：

```
gem list
```

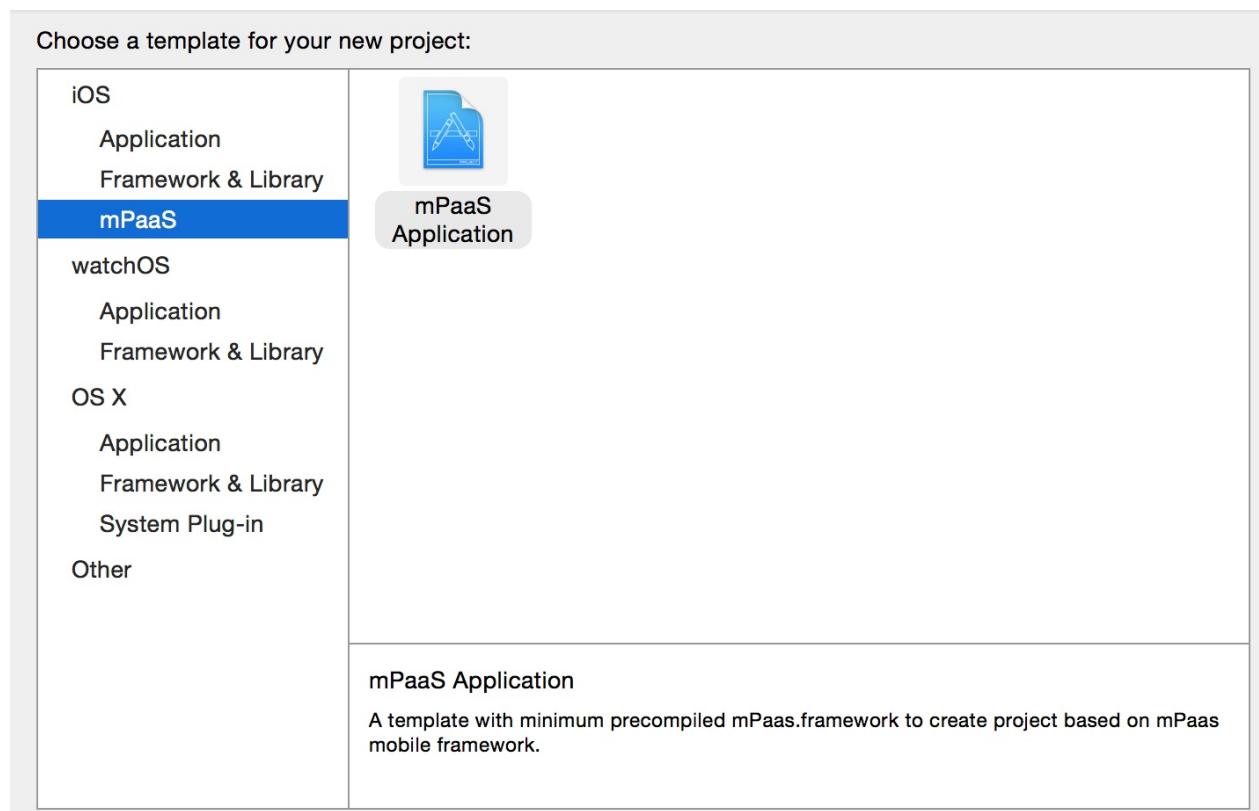


此外，如果你使用了 [taobao.org](http://taobao.org) 提供的 **Ruby** 源，需要确保使用 `https` 的方式。他们已经不支持 `http` 了。

@cnName 2 Xcode工程模板 @priority 2

## 2 Xcode工程模板

安装移动开发者工具后，会同时自动安装Xcode工程模板。在Xcode->File->New Project菜单下打开新建工程对话框，输入工程名字便可以创建一个基于移动框架的Xcode工程。



## Table of Contents |

Choose options for your new project:

Product Name:

Organization Name:

Organization Identifier:

Bundle Identifier:

mPaaS Application Key:

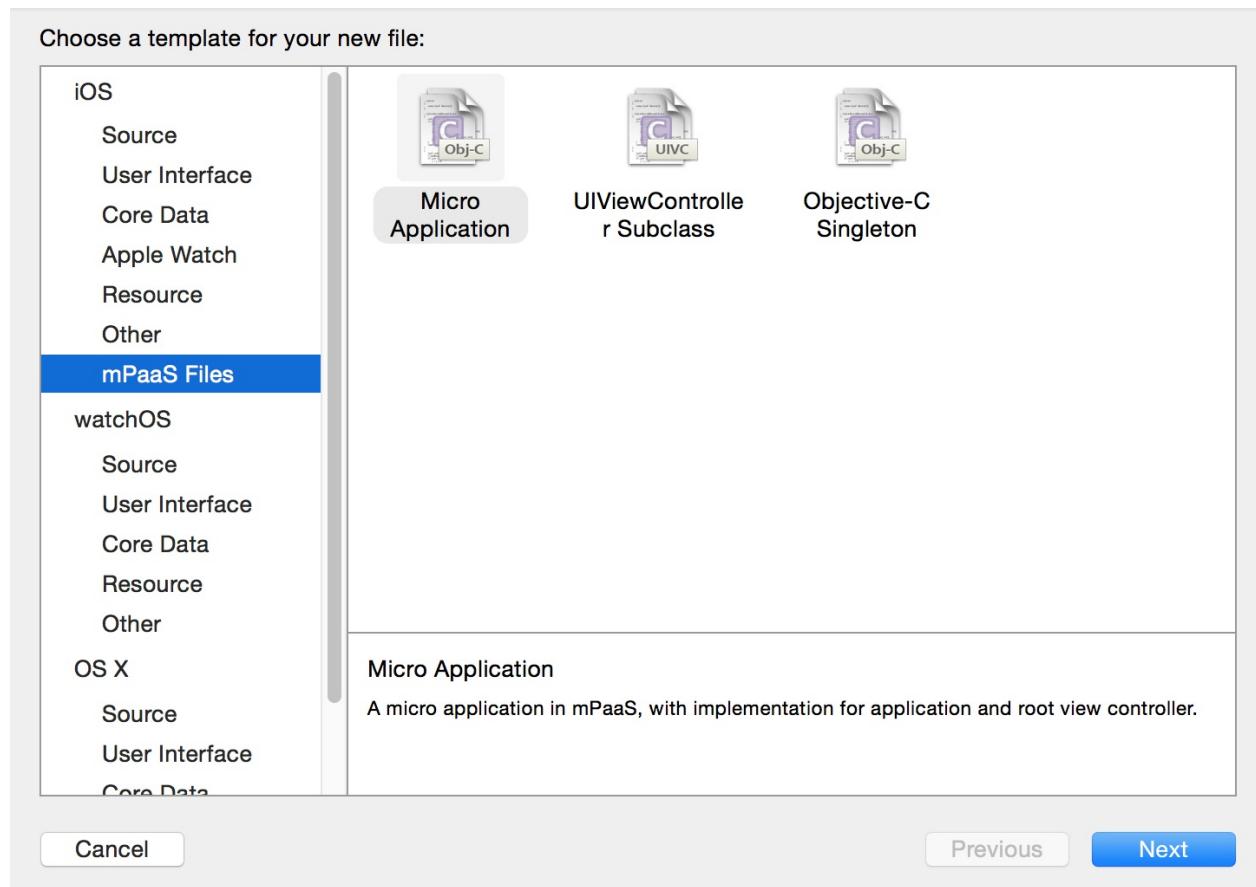
[Cancel](#) [Previous](#) [Next](#)

## @cnName 3 Xcode文件模板 @priority 3

## 3 Xcode文件模板

安装移动开发者工具后，会同时自动安装Xcode文件模板。在Xcode->File->New File菜单下打开新建文件对话框，选择需要创建的文件模板并输入名称即可。目前提供了以下三种模板：

1. 微应用代码模板
2. UIViewController模板
3. 单例模板

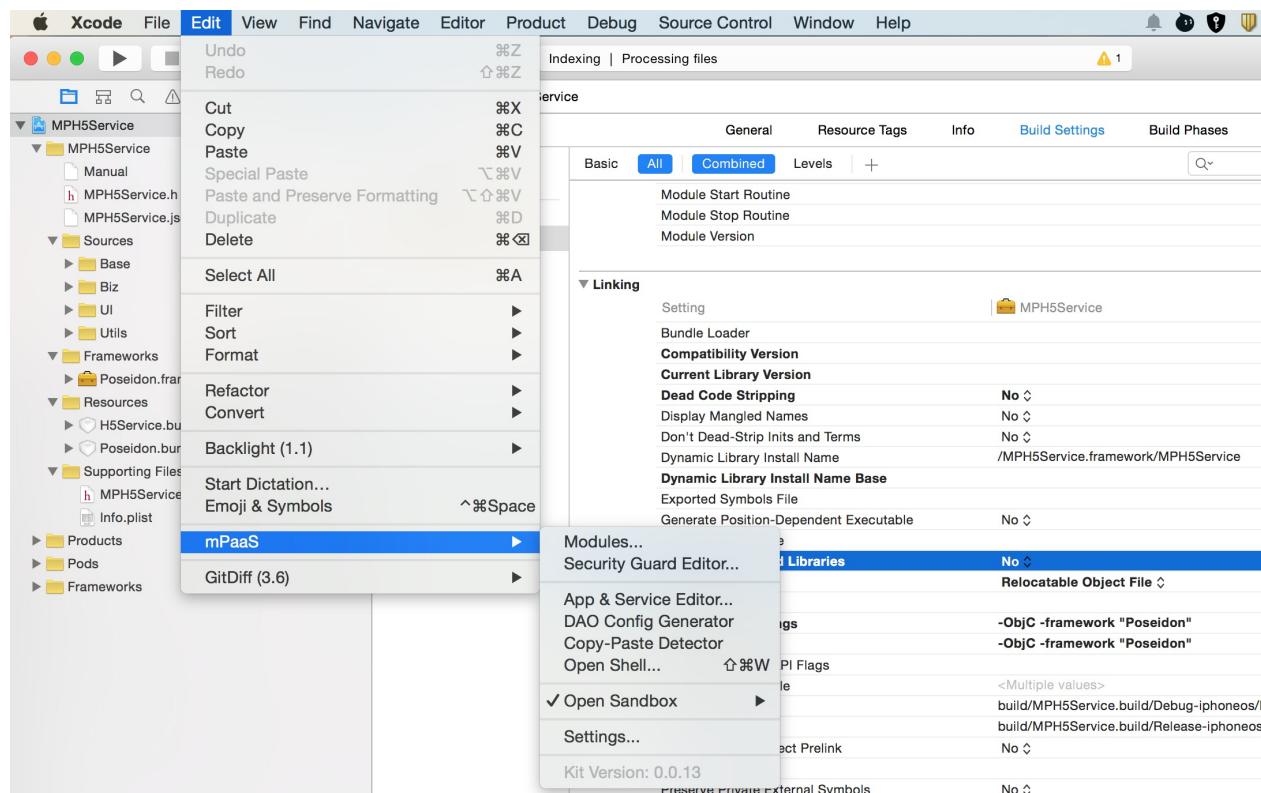


@cnName 4 Xcode插件 @priority 4

## 4 Xcode插件

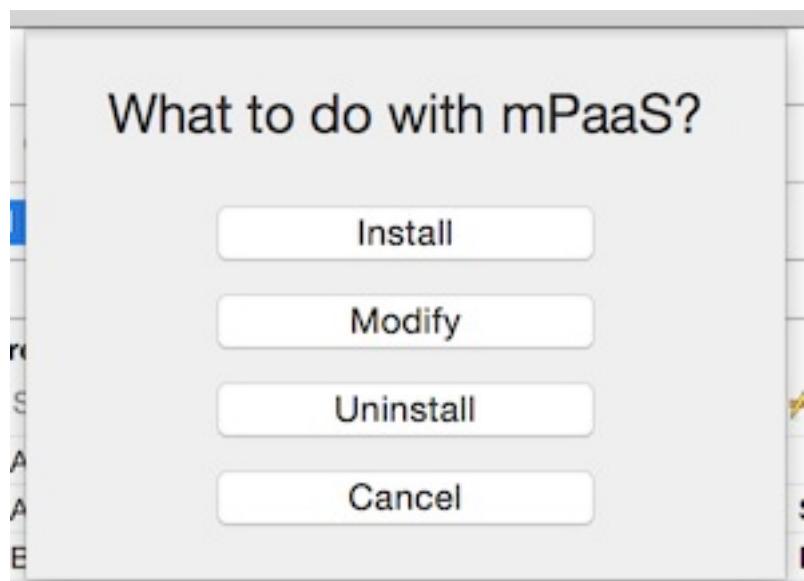
[TOC]

开发者工具包括 Xcode 插件，安装完成后会自动出现在 Xcode > Edit 菜单下。目前提供的功能如图：



### 4.1 移动模块管理器

菜单项为 `Modules`，点击后出现如下界面：

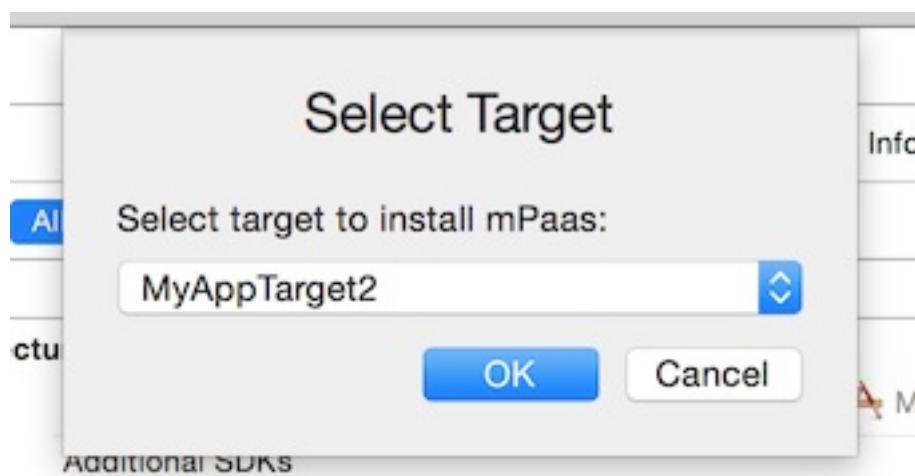


Install 安装移动模块

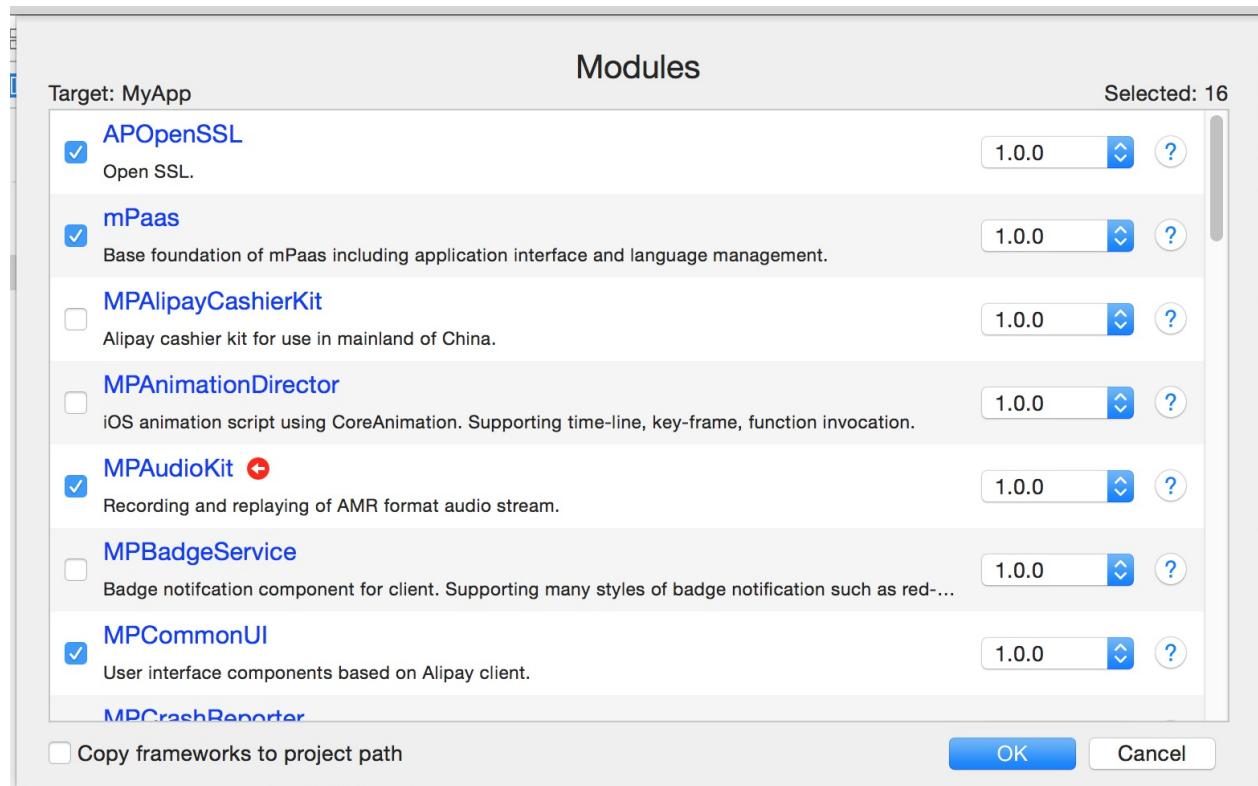
Modify 修改已经安装了移动的目标使用的模块

Uninstall 卸载移动模块

一个Xcode工程里可能有很多编译 Targets，插件支持对 Target 进行单独的设置。当工程里有多个 Target 时会出现选择框：



选择需要操作的 Target 后进入模块编辑窗口：



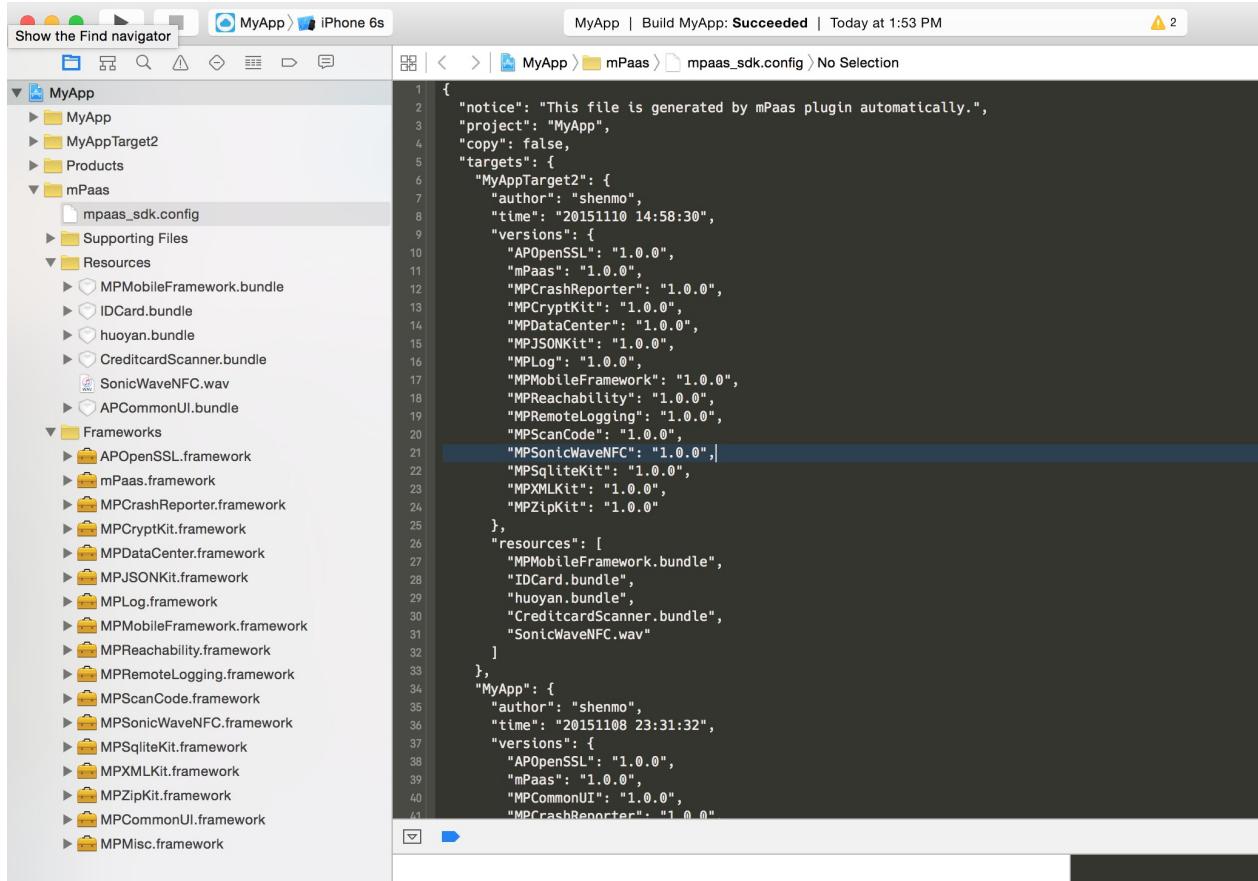
勾选需要使用的移动模块，并选择相应的版本后，点击 OK 即可。

红色箭头 本次编辑修改变更过的模块。

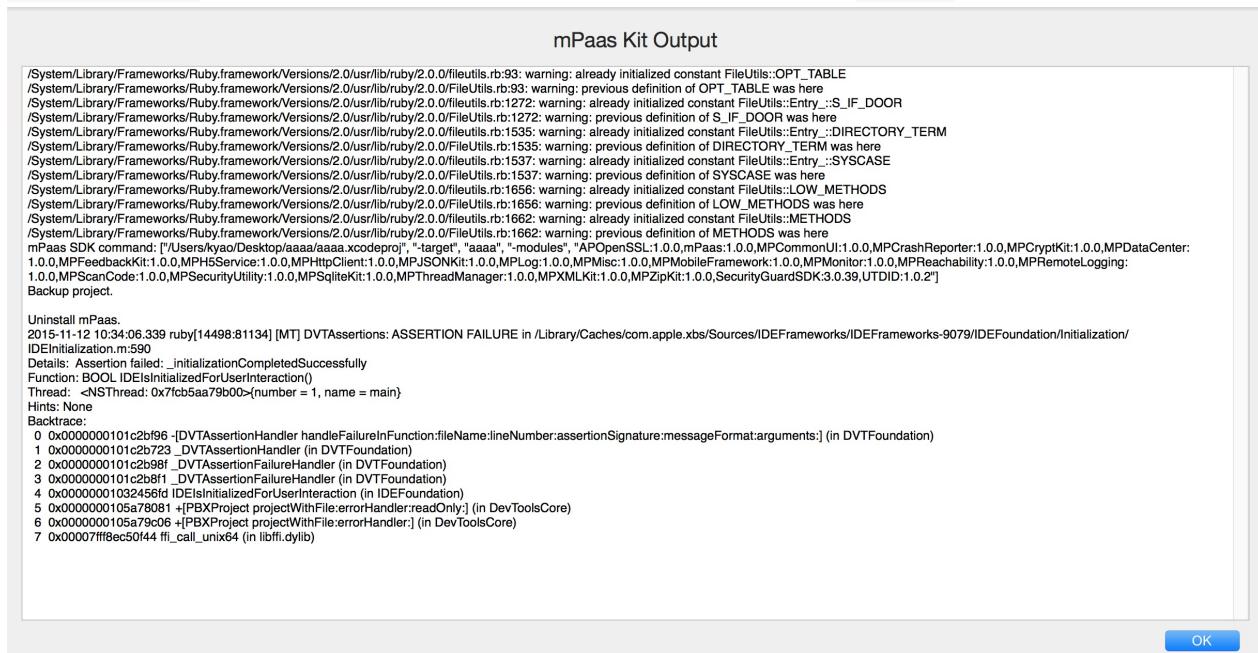
问号按钮 显示模块的版本和升级信息。

`Copy frameworks to project path` 移动 SDK 会统一安装到用户机器的公共目录，并由插件自动维护。当不勾选时，会使用这个 sdk 目录里公共的包。这样可以减小工程的大小，用户只需要提交自己的代码即可。当工程在其它机器上运行时，只要也安装了移动插件，即使还没安装移动SDK，插件会自动下载SDK，保证程序正常运行。当然，你也可以勾选，这样相应模块的包会拷贝到工程的 mPaaS 目录下，你可以将这些模块包作为代码一起提交。

**mPaaS 目录会出现在工程中，不要手动修改这个目录里的内容。**

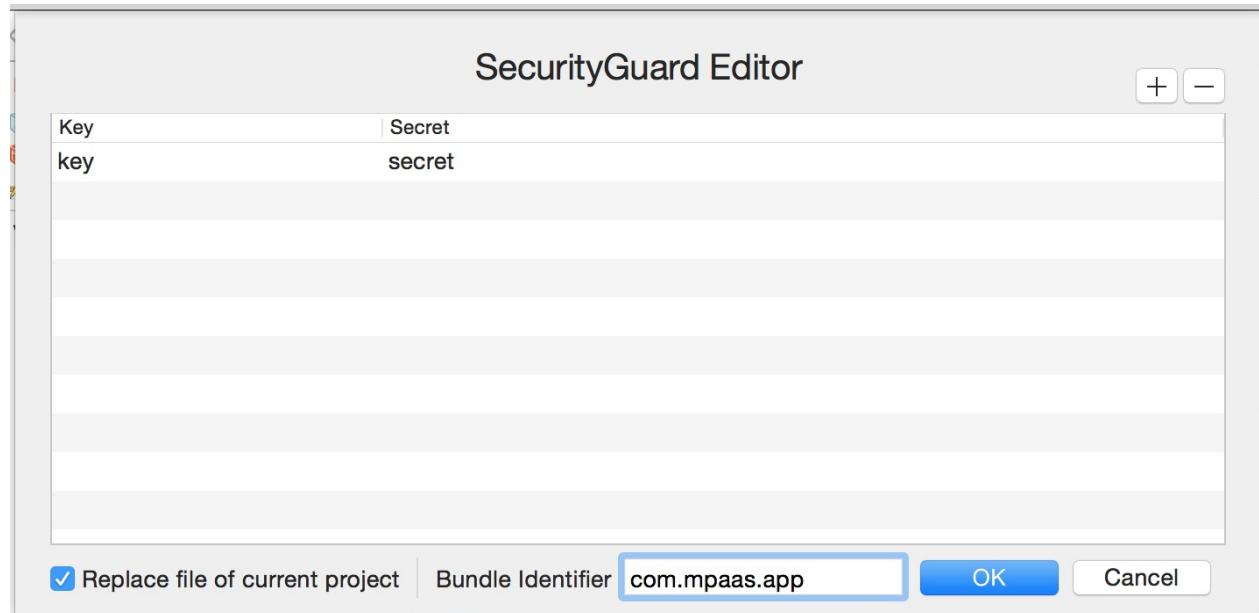


如果使用模块管理器时遇到下面错误，说明你的机器上还没有安装  
xcodeproj 这个 Ruby Gem 或版本比较老。请参考 [概述](#)



## 4.2 无线保镖密钥图片生成器

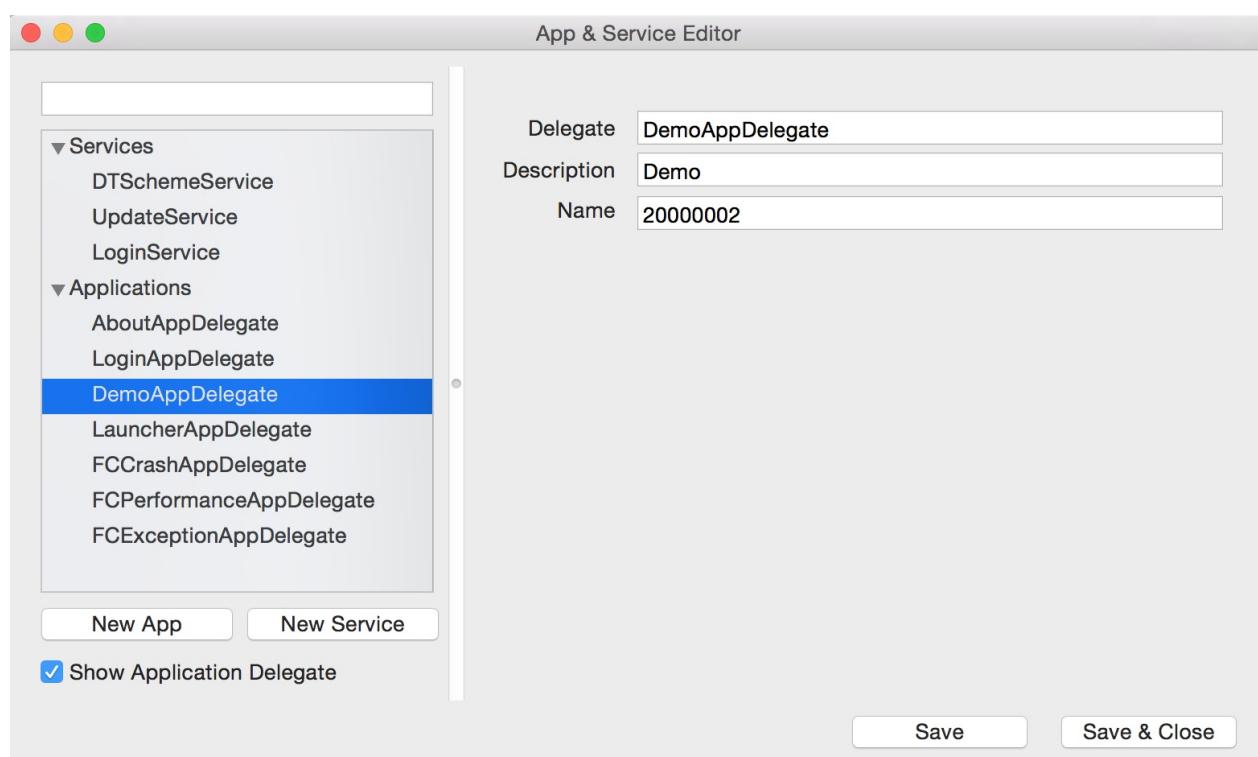
菜单项为 `Security Guard Editor` , 界面如图:



点击右上角的 `+` , `-` 添加或删除密钥对，直接在列表里编辑内容。密钥图片需要与应用的 `Bundle Identifier` 相匹配。勾选 `Replace file of current project` 会在生成图片后自动替换本工程内的 `yw_1222.jpg` 图片。

## 4.3 微应用与服务编辑器

菜单项为 `App & Service Editor` , 在使用移动客户端框架的工程内点击后，会唤起编辑器。方便开发者浏览已经注册的微应用或微服务，并提供可视化编辑功能。



## 4.4 统一存储DAO功能配置文件生成

菜单项为 `DAO Config Generator`，在代码编辑器中选中需要生成DAO配置文件的类声明。

```

39     curData = curData ?: @""
40
41 @class PubChatMessage;
42
43 /**
44 * 消息基类
45 */
46 @interface PPChatMessage : NSObject <NSCoding>
47
48 @property(nonatomic, strong) NSString *nameForOwner;//消息所属方名字
49 @property(nonatomic, assign) int cellHeight;//保存动态计算的高度,增加加载速度
50 @property(nonatomic, strong) NSString *msgId;
51 @property(nonatomic, strong) NSString *msgTypeString;
52 @property(nonatomic, assign) MessageType msgType;
53 @property(nonatomic, strong) NSString *msgTime;
54 @property(nonatomic, strong) NSString *msgTimeMisecs;
55 @property(nonatomic, strong) NSString *newsMsgTime;
56 @property(nonatomic, strong) NSString *newsMsgTimeMisecs;
57 @property(nonatomic, assign) BOOL deleteOnLongPress;
58 @property(nonatomic, assign) BOOL deleteOnClick;
59 @property(nonatomic, strong) NSString *thirdAccountId;
60 @property(nonatomic, strong) NSString *publicId;
61 @property(nonatomic, strong) NSString *msgDataId;
62
63 @end
64

```

点击 `DAO Config Generator`，会自动生成DAO存储配置文件。文件会创建在临时目录中，复制到工程目录下并添加到工程中即可。关于统一存储DAO功能的使用方法请参考 [工具组件集->统一存储](#)

```
<module name="PPChatMessage" initializer="createTable" tableName="PPChatMessage_table" version="1.0">
<const table_columns="(nameForOwner, cellHeight, msgId, msgTypeString, msgType, msgTime, msgTimeMisecs, newsMsgTime, newsMsgTimeMisecs, deleteOnLongPress, deleteOnClick, thirdAccountId, publicId, msgDataId)" />
<const table_values="%{m.*}" nameForOwner, cellHeight, msgId, msgTypeString, msgType, msgTime, msgTimeMisecs, newsMsgTime, newsMsgTimeMisecs, deleteOnLongPress, deleteOnClick, thirdAccountId, publicId, msgDataId" />
<const table_update="update ${T} set %{* = #m.*}" nameForOwner, cellHeight, msgId, msgTypeString, msgType, msgTime, msgTimeMisecs, newsMsgTime, newsMsgTimeMisecs, deleteOnLongPress, deleteOnClick, thirdAccountId, publicId, msgDataId" />
<update id="createTable">
<!—在第二个step里输入建索引的语句，如果有更多索引，可以添加step。如果不需索引，可以把step里的SQL语句直接写在这里。—>
<step>
    create table if not exists ${T} (nameForOwner text, cellHeight integer, msgId integer, msgTypeString text, msgType integer, msgTime text, msgTimeMisecs text, newsMsgTime text, newsMsgTimeMisecs integer, deleteOnLongPress integer, deleteOnClick integer, thirdAccountId integer, publicId text, msgDataId text)
</step>
<step>
    create index if not exists <!—输入你需要创建的索引—>
</step>

```

## 4.5 其它插件

### 4.5.1 Open Sandbox

快速打开应用模拟器沙箱

### 4.5.2 Copy-Paste Detector

基于CPD的代码重复检测工具

### 4.5.3 Open Shell

快速打开终端，并定位到当前工程的路径

@cnName 5 命令行工具 @priority 5

## 5 命令行工具

[TOC]

安装开发者工具后，除了工程模板、文件模板、Xcode插件外，还有一套命令行工具。

### 5.1 更新或重新安装开发者工具

在终端运行下面命令，可以强行重新安装最新版的开发者工具。

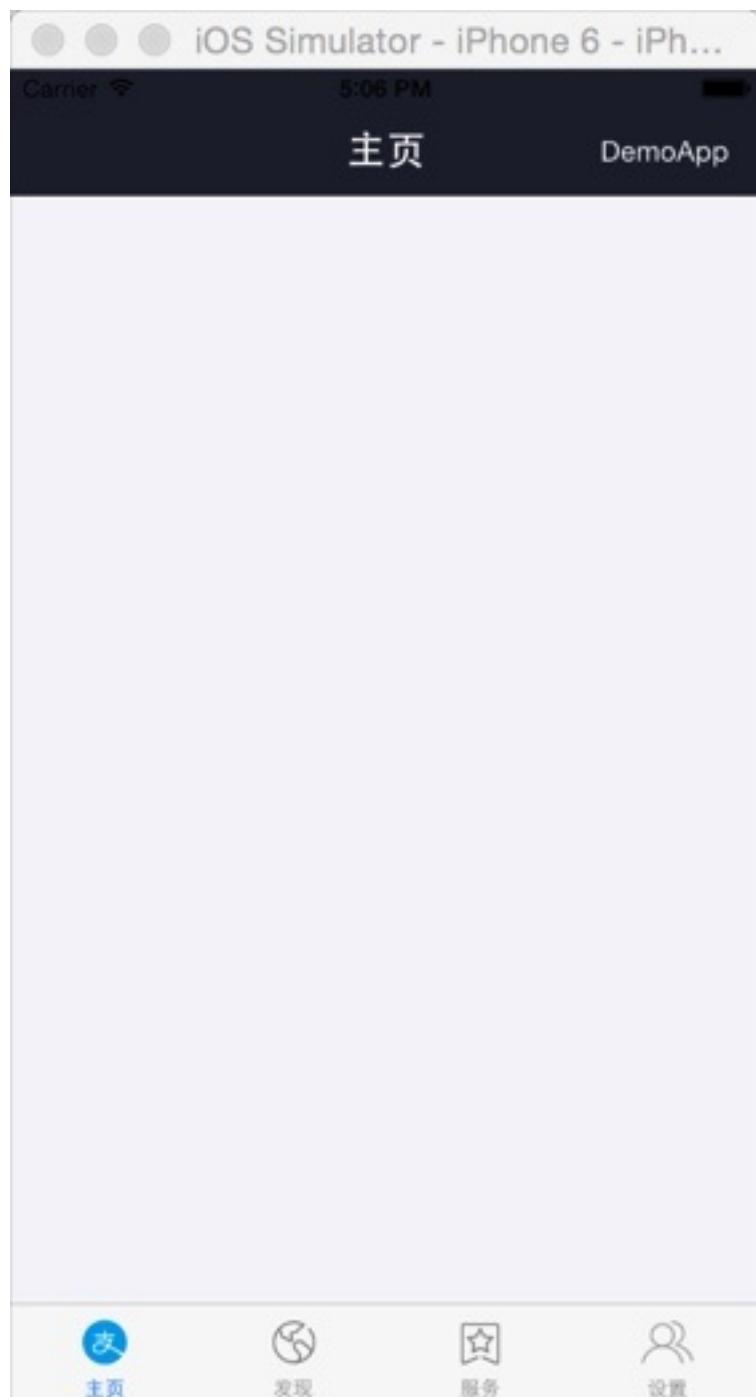
```
mpaas kitupdate
```

### 5.2 创建应用

进入目的路径，使用下面命令创建一个名字为 **XXXX** 的工程

```
mpaas createapp XXXX
```

这个工程可以直接运行，并且提供了一个 Demo 子应用和服务的例子，整个 App 是基于多 tab。第三方 App 可以在这上面进行开发，并且使用 Xcode 插件修改使用的模块。



## 5.3 查看SDK版本信息

在终端运行下面命令，会打印出本地的 SDK 里模块的信息。

```
mpaas sdk
```

其中 `Local` 为模块在本地的版本号，`Remote` 为仓库中最新的版本号，`Missing` 表示这个模块在本地还没下载，青色底表示这个模块与仓库的版本不一致。

Framework	Local	Remote
APOpenSSL	1.0.0	1.0.0
mPaaS	1.0.0	1.0.1
MPAlipayCashierKit	1.0.0	1.0.0
MPAnimationDirector	1.0.0	1.0.0
MPAudioKit	1.0.0	1.0.0
MPBadgeService	1.0.0	1.0.0
MPCrashReporter	1.0.0	1.0.0
MPCryptKit	1.0.0	1.0.0
MPDataCenter	Missing	1.0.0
MPFeedbackKit	Missing	1.0.0
MPFileTransferService	1.0.0	1.0.0
MPH5Service	1.0.0	1.0.0
MPHHttpClient	1.0.1	1.0.1
MPIImageCodeGenerator	1.0.0	1.0.0
MPJSONKit	1.0.0	1.0.0

## 5.4 更新本地SDK

在终端运行

```
mpaas sdk update
```

会同步本地与仓库的SDK，未下载的模块会自动下载，已经下载的模块会更新到最新版本。

```

Updating mPaaS SDK [标题: 5 命令行工具 | 权值: 5 | 查看页面]
Downloading mPaaS.framework [Succeeded]
mPaaS updated from 1.0.0 to 1.0.1
Downloading MPDataCenter.framework [Succeeded]
Downloading MPFeedbackKit.framework [Succeeded]
Done
→ ~ |

```

## 5.5 查看某个模块的本地版本与更新信息

在终端运行，`XXXX` 为模块名

```
mpaas sdk XXXX
```



```
➔ ~ mpaas sdk MPHttpClient
MPHttpClient.framework is 1.0.1 in local mPaaS SDK.

对NSURLRequest的封装，提供下载、上传，RPC调用等功能。
Encapsulation of NSURLRequest providing uploading, downloading, RPC functionality.

Versions:

1.0.0
    初始版本
    Initial version.

    Dependencies: mPaas, MPZipKit, MPJSONKit, MPCryptKit, MPLog, MPReachability, SecurityGuardSDK, UTID

1.0.1
    RPC模块可以指定请求加签名时使用的密钥。
    You can specify a request signing key for RPC.

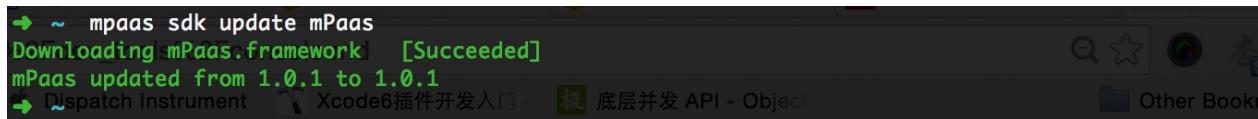
    Dependencies: mPaas, MPZipKit, MPJSONKit, MPCryptKit, MPLog, MPReachability, SecurityGuardSDK, UTID

➔ ~ |
```

## 5.6 升级某个模块到最新版本

在终端运行，**XXXX** 为模块名。这条命令会重新下载模块，即使本地已经是最新版本。

```
mpaas sdk update XXXX
```

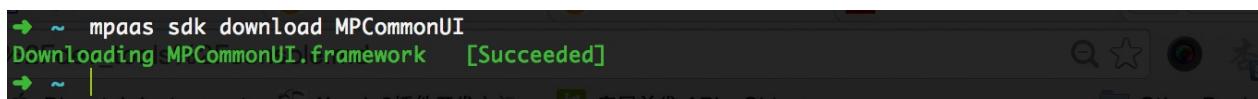


```
➔ ~ mpaas sdk update mPaas
Downloading mPaas.framework [Succeeded]
mPaas updated from 1.0.1 to 1.0.1
➔ Dispatch Instrument Xcode6插件开发入门 极 底层并发 API - ObjC Other Books
```

## 5.7 下载或重新下载某个模块

在终端运行，**XXXX** 为模块名。这条命令会重新下载模块，即使本地已经存在。

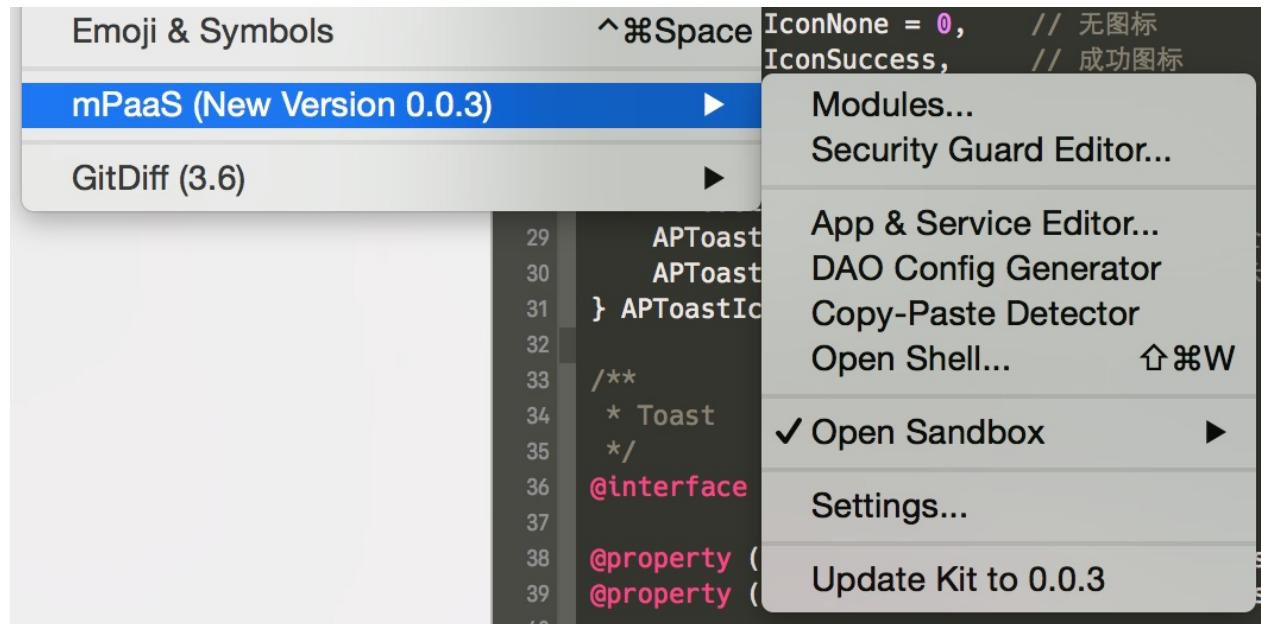
```
mpaas sdk download XXXX
```



```
➔ ~ mpaas sdk download MPCommonUI
Downloading MPCommonUI.framework [Succeeded]
➔ ~ |
Dispatch Instrument Xcode6插件开发入门 极 底层并发 API - ObjC Other Books
```

## 5.8 查看开发者工具版本

你可以在 Xcode 里，mPaaS 插件菜单中看到开发者工具版本，如果有更新，会提示。



也可以运行命令行查看到前开发者工具的版本

```
mpaas kitversion
```

A terminal window is shown with the command 'mpaas kitversion' entered. The output of the command, '0.0.2', is displayed below the command line. The terminal has a dark theme with white text and icons.

@cnName 6 发布记录 @priority 6

## 6 发布记录

## @cnName 附录 @priority 8

# 附录

## 1 开源代码使用说明

iOS 客户端代码里引用或修改了以下开源代码

开源项目	用处
OpenCore	MPAudioKit
MBProgressHUD	MPCommonUI
ODRefreshControl	MPCommonUI
TTTAttributedLabel	MPCommonUI
EGOResfreshTableHeaderView	MPCommonUI
PLCrashReport	MPCrashReporter
libpng	MPImageCodeGenerator
Barcode	MPImageCodeGenerator
QR_Encode	MPImageCodeGenerator
Reachability	MPReachability
FMDB	MPSqliteKit
SDWebImage	MPWebImage
KissXML	MPXMLKit
MiniZip	MPZipKit

插件里使用了以下开源插件的代码

开源项目
Lin
VVDocumenter