**iOS平台开发规范**

修改记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | 修改后版本号 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | 0.5 | 完成初稿 | 2012-2-17 |  |
| 2 | 0.6 | 整理修改为浏览器初稿 | 2013-05-30 |  |
| 3 | 0.7 | 调整define的规范 | 2013-06-10 |  |

目录

[修改记录 2](#_Toc317164382)

[目录 3](#_Toc317164383)

[图表 5](#_Toc317164384)

[1 概述 6](#_Toc317164385)

[2 编码规范 7](#_Toc317164386)

[2.1 命名规范 7](#_Toc317164387)

[2.1.1 命名基础 7](#_Toc317164388)

[2.1.2 方法命名 8](#_Toc317164389)

[2.1.3 函数命名 10](#_Toc317164390)

[2.1.4 实例变量和数据类型命名 10](#_Toc317164391)

[2.1.5 可接受的简写和缩写 10](#_Toc317164392)

[2.2 代码注释 11](#_Toc317164393)

[2.2.1 文件注释 11](#_Toc317164394)

[2.2.2 声明注释 11](#_Toc317164395)

[2.2.3 实现注释 11](#_Toc317164396)

[2.3 代码格式化 11](#_Toc317164397)

[2.3.1 缩进 11](#_Toc317164398)

[2.3.2 空格 11](#_Toc317164399)

[2.3.3 行长度 12](#_Toc317164400)

[2.3.4 回行 12](#_Toc317164401)

[2.3.5 方法的申明和定义 12](#_Toc317164402)

[2.3.6 方法调用 12](#_Toc317164403)

[2.3.7 @public，@private，@package和@optional 12](#_Toc317164404)

[2.3.8 协议 13](#_Toc317164405)

[2.3.9 程序块 13](#_Toc317164406)

[2.3.10 没有实例变量的接口 13](#_Toc317164407)

[2.4 文件布局 14](#_Toc317164408)

[2.4.1 头文件布局 14](#_Toc317164409)

[2.4.2 实现文件布局 16](#_Toc317164410)

[2.5 分类管理 19](#_Toc317164411)

[2.5.1 本地化字符串 19](#_Toc317164412)

[2.5.2 Javascript语句管理 20](#_Toc317164413)

[2.5.3 Tag管理 20](#_Toc317164414)

[2.5.4 资源管理 21](#_Toc317164415)

[2.6 代码质量控制 21](#_Toc317164416)

[2.6.1 成员变量访问属性 21](#_Toc317164417)

[2.6.2 构造器 21](#_Toc317164418)

[2.6.3 初始化 23](#_Toc317164419)

[2.6.4 避免直接调用+new 23](#_Toc317164420)

[2.6.5 使公有接口尽可能简单 23](#_Toc317164421)

[2.6.6 #import 和 #include 23](#_Toc317164422)

[2.6.7 使用Root框架 23](#_Toc317164423)

[2.6.8 autorelease 23](#_Toc317164424)

[2.6.9 在构造和析构阶段避免调用访问器方法 23](#_Toc317164425)

[2.6.10 按照申明的次序析构实例变量 24](#_Toc317164426)

[2.6.11 避免抛出异常 24](#_Toc317164427)

[2.6.12 nil检查 25](#_Toc317164428)

[2.6.13 BOOL类陷阱 25](#_Toc317164429)

[2.6.14 自动合成实例变量 25](#_Toc317164430)

[2.6.15 Magic number 26](#_Toc317164431)

[3 SVN提交规范 27](#_Toc317164432)

[3.1 SVN能干什么 27](#_Toc317164433)

[3.2 提交原则 27](#_Toc317164434)

[3.3 Log message规范 28](#_Toc317164435)

[4 Code review规范 29](#_Toc317164436)

[4.1 通用原则 29](#_Toc317164437)

[4.2 参加人员 29](#_Toc317164438)

[4.3 Review时机 29](#_Toc317164439)

[4.4 被Review人员主要工作 29](#_Toc317164440)

[4.4.1 讲解流程 29](#_Toc317164441)

[4.4.2 注意事项 30](#_Toc317164442)

[4.5 Review人员主要工作 30](#_Toc317164443)

[5 项目提测规范 31](#_Toc317164444)

[5.1 提测阶段性划分 31](#_Toc317164445)

[5.2 提测流程 31](#_Toc317164446)

[6 项目发版规范 33](#_Toc317164447)

[6.1 提交前确认事项 33](#_Toc317164448)

[6.2 提交流程 33](#_Toc317164449)

[6.3 前期准备 33](#_Toc317164450)

[6.4 提交审核 33](#_Toc317164451)

[7 参考资料 34](#_Toc317164452)

图表

[图表 1 命名通用原则 7](#_Toc317164453)

[图表 2 文件名后缀 7](#_Toc317164454)

[图表 3 方法命名 8](#_Toc317164455)

[图表 4 返回属性的方法命名 8](#_Toc317164456)

[图表 5 访问器方法命名 9](#_Toc317164457)

[图表 6 代理方法命名 9](#_Toc317164458)

[图表 7 集合方法命名 9](#_Toc317164459)

[图表 8 方法参数命名 9](#_Toc317164460)

[图表 9成员变量命名 10](#_Toc317164461)

[图表 10文件注释 11](#_Toc317164462)

[图表 11 实现代码注释 11](#_Toc317164463)

[图表 12 代码格式之空格 12](#_Toc317164464)

[图表 13 方法申明的格式 12](#_Toc317164465)

[图表 14 协议格式 13](#_Toc317164466)

[图表 15 程序块 13](#_Toc317164467)

[图表 16 特殊接口声明 14](#_Toc317164468)

[图表 17 头文件布局 16](#_Toc317164469)

[图表 18 实现文件 19](#_Toc317164470)

[图表 19 本地化字符串 20](#_Toc317164471)

[图表 20 简单JS语句 20](#_Toc317164472)

[图表 21 Tag定义举例 21](#_Toc317164473)

[图表 22 构造器实现 22](#_Toc317164474)

[图表 23 构造器中调用方法 24](#_Toc317164475)

[图表 24 异常格式 25](#_Toc317164476)

[图表 25 BOOL值 25](#_Toc317164477)

[图表 26 自动合成变量 26](#_Toc317164478)

[图表 27 SVN提交记录 27](#_Toc317164479)

[图表 28 修复QA BUG时log message示例 28](#_Toc317164480)

[图表 29 修复自测bug时log message示例 28](#_Toc317164481)

[图表 30 完成功能时log message示例 28](#_Toc317164482)

[图表 31 完成功能时log message示例 28](#_Toc317164483)

[图表 32 提测时log message示例 28](#_Toc317164484)

[图表 33 提测邮件示例 32](#_Toc317164485)

# 概述

规范开发流程对提高团队工作效率，降低项目风险有重大意义。此文档旨在制定一个为大家所接受和认可的开发规范，并严格执行。

本文涉及的规范有编码规范，SVN提交规范，Code review规范，提测流程规范，项目发版规范。

# 编码规范

规范代码中类、变量、函数的命名，代码格式编写，注释以及异常处理流程会提高代码的可读性和健壮性。代码规范使得团队成员编写的代码风格保持一致，便于维护和理解，保证代码质量。

## 命名规范

* 将命名分类，对应的类型不同，处理的方法不同；
* 命名必须用英文，不应使用拼音；
* 如不特殊说明，以下命名规则默认以驼峰法命名。

### 命名基础

#### 通用原则

* 清晰：简洁，表意；少用缩写，除非是业内长时间存在的名字；名字不怕长；

|  |
| --- |
| insertObject:atIndex: // 清晰  insert:at: // 不清晰  removeObjectAtIndex: // 清晰  remove: // 不清晰 |

图表 1 命名通用原则

* 一致性：可以参考Cocoa框架中的命名；
* 不要自引用。

#### 前缀

* 在给类、协议（protocols）、方法、常量和typedef结构命名时可以用前缀；
* 方法或是结构体中的域命名不要用前缀。

#### 后缀

扩展名是分类根入口，我们知道扩展名就知道文件存的是什么说明。

|  |  |
| --- | --- |
| .h | C/C++/Objective-C的头文件 |
| .m | Ojbective-C实现文件 |
| .mm | Ojbective-C++的实现文件 |
| .cpp | 纯C++的实现文件 |
| .c | 纯C的实现文件 |

图表 2 文件名后缀

特别说明：Xcode中设定文件编译的类型，不要主动修改，使用默认。应该通过修改文件名后缀来解决编译器发出的错误信息。

#### 类（class）和协议（protocol）命名

* 类型命名必须混合大小写且以大写开始。
* 单词的首字母大写，如UrlManager
* 以下几种类型特殊处理
  + 枚举：BD开头+枚举名
  + 结构体：S开头+结构名
  + 类：类名
  + 协议：类名+(子功能)+Delegate

#### 文件名

文件名应该与类名相同，如一个文件中有多个类，那以主类名为文件名。

### 方法命名

#### 通用规则

* 小写字母开头，嵌入单词首字母大写，不要用前缀。有两个特例需要注意：一是以一些业内公认的缩写开头，比如：URL,HTTP等；另一个是私有方法；
* 如果方法是对对象的某个操作，则以动词开头；

|  |
| --- |
| - (void)invokeWithTarget:(id)target;  - (void)selectTabViewItem:(NSTabViewItem \*)tabViewItem |

图表 3 方法命名

* 如果方法放回的是接收者的属性，则方法的后面要加上属性。不要在方法中加上“get”除非放回接收者的多个属性；

|  |
| --- |
| - (NSSize)cellSize; // right  - (NSSize)calcCellSize; // wrong  - (NSSize)getCellSize; // wrong |

图表 4 返回属性的方法命名

* 在所有的参数前面要加上关键词；
* 参数前面的关键词要对参数起描述作用；
* 继承的方法要比原方法更加的具体，增加的参数追加在已有的参数后面；
* 如果方法描述了两个不同类别的行为要以“and”来连接他们。
* 所有方法均需声明，私有方法在.m文件中进行声明。

#### 访问器(Accessor)方法

访问器方法是返回对象的属性的方法。根据属性的不同类型有如下几种命名方法：

* 属性是名词

- (void)setNoun:(type)aNoun;

- (type)noun;

* 属性是形容词

- (void)setAdjective:(BOOL)flag;

- (BOOL)isAdjective;

* 属性是动词

- (void)setVerbObject:(BOOL)flag;

- (BOOL)verbObject;

* 不要用一个分词把一个动词变成形容词
* 可以用“can”、“should”、“will”等语气动词来表意，但是不要用“do”、“does”

|  |
| --- |
| - (void)setCanHide:(BOOL)flag; // right  - (BOOL)canHide; // right  - (void)setShouldCloseDocument:(BOOL)flag; // right  - (BOOL)shouldCloseDocument; // right  - (void)setDoesAcceptGlyphInfo:(BOOL)flag; // wrong  - (BOOL)doesAcceptGlyphInfo; // wrong |

图表 5 访问器方法命名

#### 代理(Delegate)方法

代理方法的命名有独特性。

* 以发送消息的对象类型的名字开头，同样第一个字母要小写；
* 如果只有一个参数，则参数名字命名为“sender”，参数的类型的名字是类名；
* 通知的delegate方法命名是个例外；

|  |
| --- |
| - (BOOL)applicationOpenUntitledFile:(NSApplication \*)sender;  - (void)windowDidChangeScreen:(NSNotification \*)notification; |

图表 6 代理方法命名

* 可以用“did”或是“will”来标明已经或是即将发生的事情；
* 如果是激发另一个对象做一些事情，那么优先考虑用“should”。

#### 集合方法

操作对象集合的方法的命名：

- (void)addElement:(elementType)anObj;

- (void)removeElement:(elementType)anObj;

- (NSArray \*)elements;

举例如下：

|  |
| --- |
| - (void)addLayoutManager:(NSLayoutManager \*)obj;  - (void)removeLayoutManager:(NSLayoutManager \*)obj;  - (NSArray \*)layoutManagers; |

图表 7 集合方法命名

#### 方法参数

* 以小写字母“a”开头后续单词的首字母大写，有一个例外，如果是Cocoa框架的delegate方法则保持SDK中方法声明时参数的原型不变；
* 名字中不要用“pointer”或“ptr”，由参数的类型而不是参数名字说明是否是一个指针；
* 避免用一两个字母来命名；
* 避免用缩写。

|  |
| --- |
| ...action:(SEL)aSelector;  ...alignment:(int)aMode;  ...content:(NSRect)aRect; |

图表 8 方法参数命名

#### 私有方法

* 私有方法的命名和公有方法的命名通用规则大体一致；
* 为了避免子类覆盖Cocoa框架的私有方法，规定不要用“\_”作为私有方法的前缀。
* 如果是从一个Cocoa框架的大类继承而来，为了绝对确保私有方法的唯一性。

### 函数命名

如果对象是一个singleton或者编写一个功能子系统时，优先使用函数而不是类方法。

函数命名有一些通用规则：

* 函数命名和方法一样，但有些意外情况：
  + 使用和类、常量一致的前缀；
  + 前缀后面的单词的第一个字母大写；
* 以描述函数功能的动词开头。

### 实例变量和数据类型命名

#### 常量

* const常量：以k开头，后面跟自命名；
* #define宏常量：以k开头，单词首字母大写；
* 本地化字串：以kString开头+自命名，自命名首字母需大写；
* 枚举常量：以BD开头，后面跟枚举类型的全写（要有明确区分的标识意义），加自命名。

typedef enum

{

BDFileTypeOther,

BDFileTypeAudio,

BDFileTypeVideo,

BDFileTypeDocument,

BDFileTypeImage

}BDFileType;

#### 变量

变量名必须混合大小写且以小写开始，无下划线。

* 参数变量：a+变量名，Cocoa框架中的方法参数除外；
* 成员变量：堆上分配新内存或对原变量产生引用计数修改： \_+变量名，实现Set及Get方法。这类变量，需要在析构函数中将对像释放，

|  |
| --- |
| NSTimer \*\_updateStatusBarTimer;  @property(nonatomic, retain) NSTimer \*updateStatusBarTimer;  @synthesize updateStatusBarTimer = \_updateStatusBarTimer; |

图表 9成员变量命名

* 成员变量，非堆上分配：变量名必须混合大小写且以小写开始，无下划线。

### 可接受的简写和缩写

通常情况下，在定义一个接口的时候不要用缩写。也有两种情况可以使用缩写：

* 如果缩写和标准C库缩写一致

比如：alloc、dealloc、init、max、msg、rect和temp等。

* 参数名字命名相对宽松，可以用缩写

如果行业内公认的缩写直接使用大写的，比如：ASCII、PDF、XML、HTTP、URL等。这类缩写如果是在变量、关键词等地开头可以用全小写，在其他情况使用时必须全用大写，禁止出现首字母大写其他字母小写的情况。

## 代码注释

* 养成写注释的好习惯，特别是在写接口给第三方使用时；
* 良好的变量、函数等命名可以增加阅读性，亦可减少注释；
* 你很可能就是注释的受益者。

### 文件注释

文件注释放在文件的开头，使用Xcode默认的注释风格。举例如下：

|  |
| --- |
| //  // HomePageViewController.h  // BaiduSearch  //  // Created by naonao on 6/7/11.  // Modified by tianbiao on 12/12/11. 注释了用户交互等相关代码，并没有删除以免后面用到。  // Copyright 2011 Baidu Inc. All rights reserved.  // |

图表 10文件注释

如果我们对这个文件做了大的修改或是添加了大地功能，需要在创建者后面加上修改者的信息并加上改动的描述。

### 声明注释

* 每个interface，protocol和category的声明都应该有一个注释用来描述它的作用；
* 公共接口的每个方法，都应该有注释来解释它的作用、参数、返回值以及其它成员的影响；
* 为类的线程安全性作注释，如果有的话。如果类的实例可以被多个线程访问，记得注释多线程条件下的使用规则；

### 实现注释

在实现的方法中加上注释，如果引用了成员变量或是方法则需要用“|”括起。比如

|  |
| --- |
| // Sometimes we need |count| to be less than zero.  // Remember to call |StringWithoutSpaces("foo bar baz")| |

图表 11 实现代码注释

## 代码格式化

### 缩进

使用制表符而不是空格来缩进对齐。在Xcode的菜单栏选择*Xcode->Preference->Text Editing->Indentation*中设置“*Tab width*”的宽度为**4个空格**

### 空格

* 在关键词for，do，while，if，else，switch，case，default 等后面要加上空格；
* 操作符&&，||，!=，==，=，>，<，/，%，+=，-=等的前后加上空格；
* 多个参数之间加上空格。

|  |
| --- |
| // for，if，==以及=号  for (UIView \*subView in self.view.subviews)  {  if (subView.tag == ETagSearchBarView)  {  subView.alpha = 0.0f;  break;  }  }  // 调用方法的多个参数之间加上空格  CGRectMake(point.x, point.y, width, height); |

图表 12 代码格式之空格

### 行长度

* 行宽不做硬性的规定。也即在Xcode的菜单栏选择*Xcode->Preference->Text Editing->Editing*中不要勾选“*Page guide at columm*” ，设置为110；
* 如果调整了字体大小或是调整了Xcode的宽度，则Xcode会自动回行，不要手动硬性回行。

### 回行

* “{”和“}”都要另起一行；
* 实现文件中的方法之间要回行。

### 方法的申明和定义

* 返回类型和“-”或是“+”号之间留一个空格；
* \*符号前面留一个空格
* 参数列表中的参数之间如果不换行只嵌入一个空格；

举例如下：

|  |
| --- |
| - (void)loadSearchResult:(NSString \*)aSearchText dingID:(NSString \*)aDingID; |

图表 13 方法申明的格式

* 方法的声明和定义建议写在同一行。如果有大于三个参数或者参数的类型和参数名字比较长则可以回行。
  + 多个参数回行，优先选择“：”对齐的方式；
  + 如果第一个参数的类型短，则以函数开头为起点缩进一个tab的方式对齐。

### 方法调用

* 和方法的声明对应；
* 尽量避免函数的参数是一个有几个参数的函数的调用方式。

### @public，@private，@package和@optional

* 类声明中@public，@private和@package前面不需要加一个空格；
* @optional前面不加空格。

### 协议

* 类型标识符和协议名之间的尖括号不能空格；
* 多个协议之间以“,”+空格分隔。

举例如下：

|  |
| --- |
| @interface HomePageViewController : UIViewController<WebToolBarDelegate, UIWebViewDelegate>  {  } |

图表 14 协议格式

* 多个protocol造成一行超过110字符时要手动换行

### 程序块

* 即便程序块能写在一行也需要{}分行写；
* “^”和“{”分行写，如果有参数参数紧随“^”；
* 如果程序块代码太大，可以单独抽出来。

举例如下：

|  |
| --- |
| // The entire block fits on one line.  [operation setCompletionBlock:^  {  [self onOperationDone];  }];  // An example where the parameter wraps and the block declaration fits  // on the same line. Note the spacing of |^(SessionWindow \*window) {|  // compared to |^{| above.  [[SessionService sharedService]  loadWindowWithCompletionBlock:^(SessionWindow \*window)  {  if (window)  [self windowDidLoad:window];  else  [self errorLoadingWindow];  }];  // Large blocks can be declared out-of-line.  void (^largeBlock)(void) = ^  {  // ...  };  [operationQueue\_ addOperationWithBlock:largeBlock]; |
|  |

图表 15 程序块

### 没有实例变量的接口

如果一个接口没有申明任何实例变量的话就可以忽略掉“{}”。

|  |
| --- |
| @interface MyClass : NSObject  // Does a lot of stuff  - (void)fooBarBam;  @end  @interface MyClass : NSObject  {  }  // Does a lot of stuff  - (void)fooBarBam;  @end |

图表 16 特殊接口声明

## 文件布局

文件布局的合理性对提高可阅读性有很大的影响。总体原则如下：

* 功能区要明确清晰，一目了然；
* 私有方法以分类的方式在实现文件中声明和实现；
* 私有方法和其调用者放在一个功能区，并紧随调用者。

### 头文件布局

综合目前所遇到的情况举一个例子如下：

|  |
| --- |
| //  // HomePageViewController.h  // BaiduSearch  //  // Created by naonao on 6/7/11.  // Copyright 2011 Baidu Inc. All rights reserved.  //  #import <UIKit/UIKit.h>  #import <CoreLocation/CoreLocation.h>  #import "WebToolBarController.h"  #define ANIMATION\_TIME 0.5 // 动画持续时间  #define POP\_ADDVIEW\_KEEPTIME 3.0 // 弹出View持续时间  // 标识工具条上按钮内型  enum TToolbarItem  {  EItemBack,  EItemForward,  EItemRefresh,  EItemSearch,  EItemFullscreen  };  @class SearchResultViewController;  @class CustomUiWebViewController;  @protocol …  // 遵循工具栏，WebView和地理位置管理协议，管理其他视图控制器  @interface HomePageViewController : UIViewController<WebToolBarDelegate, UIWebViewDelegate>  {  id<WebToolBarDelegate> delegate;  // 地理位置信息  CLLocationManager \*\_locationManager; // 注释  CLLocation \*\_currentLocation;  BOOL updatingLocationFlag;    // OCR  OCRTabBarController \*\_tabBarController;  UIImageView \*\_dialog; // 没有检测到摄像头  BOOL fromOCRView;  NSString \*\_searchResultUrlString; // 图片搜索报内存警告时记录搜索结果页面中可见页面的url  }  @property (nonatomic, retain) CLLocationManager \*\_locationManager;  @property (nonatomic, retain) CLLocation \*\_currentLocation;  + (HomePageViewController \*)sharedInstance; // 本项目中，首页是单例  // --搜索结果页相关  - (void)loadSearchResultWithURL:(NSString \*)searchURL;  - (void)loadSearchResult:(NSString \*)searchTxt;  - (void)loadSearchResult:(NSString \*)searchTxt widgetId:(NSString\*)aDingId;  // --OCR  - (void)goOCRView;  - (void)closeOCRView;  @end // HomePageViewController |

图表 17 头文件布局

### 实现文件布局

|  |
| --- |
| //  // HomePageViewController.m  // BaiduSearch  //  // Created by naonao on 6/7/11.  // Modified by xyz on 12/11/12 增加了OCR功能  // Copyright 2011 Baidu Inc. All rights reserved.  //  #import <Cocoa/Cocoa.h>  #include <CoreFoundation/CoreFoundation.h>  #import "HomePageViewController.h"  #import "SearchBarViewController.h"  #include "base/basictypes.h"  @interface HomePageViewController ()  // 首页Ding更新相关接口  - (void)updateCurDingViewIfNeeded;  - (void)stopGettingLocationInfo;  - (void)startUpdatingLocation;  // 地理位置相关接口  - (BOOL)isUpdatingLocationInfo;  - (void)saveUpdatingLocationInfo;  @end // HomePageViewController  @implementation HomePageViewController  #pragma mark - alloc & dealloc  + (HomePageViewController \*)sharedInstance  {  static HomePageViewController \*\_sharedInstance = nil;  if (\_sharedInstance == nil)  \_sharedInstance = [[HomePageViewController alloc] initWithNibName:nil bundle:nil];  return \_sharedInstance;  }  - (void)dealloc  {  [widgetSettingViewController release];  [backgroundView release];    [super dealloc];  }  - (id)initWithNibName:(NSString \*)nibNameOrNil bundle:(NSBundle \*)nibBundleOrNil  {  self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];  if (self)  [DingLocalManager sharedInstance];  return self;  }  #pragma mark - View lifecycle  - (void)loadView  {  }  - (void)viewDidLoad  {  }  - (void)viewWillAppear:(BOOL)animated  {  [super viewWillAppear:animated];  }  - (void)viewDidAppear:(BOOL)animated  {  if (!updatingLocationFlag)  [self updateCurDingViewIfNeeded];  }  #pragma mark - ToolBar methods  - (void)goSetting  {  }  // 切换到help视图  - (void)goHelp  {  }  #pragma mark - location service delegate  - (void)locationManager:(CLLocationManager \*)manager didFailWithError:(NSError \*)error  {  // 只要出错就停止获取；取消定时器；更新丁信息  [self stopGettingLocationInfo];  [NSObject cancelPreviousPerformRequestsWithTarget:self selector:@selector(stopGettingLocationInfo:) object:nil];  [self updateCurDingViewIfNeeded];  }  // 私有方法  - (void)saveUpdatingLocationInfo  {  NSString \*filePath = [FilePathManager getLocationInfoUpdatingTime];  if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])  {  NSMutableDictionary \*locationInfoDict = [[NSMutableDictionary alloc] initWithContentsOfFile:filePath];  //  if (locationInfoDict) //  {  [locationInfoDict setValue:[NSDate date] forKey:@"updatingtime"];  [locationInfoDict writeToFile:filePath atomically:NO];  } // if (locationInfoDict) 超过半屏就上注释  [locationInfoDict release];  }  else // 第一次启动程序  {  NSMutableDictionary \*locationInfoDict = [[NSMutableDictionary alloc] init];  [locationInfoDict setValue:[NSDate date] forKey:@"updatingtime"];  [locationInfoDict writeToFile:filePath atomically:NO];  [locationInfoDict release];  }  } |

图表 18 实现文件

## 分类管理

把一些同性质的资源、方法通过分类、归档以便于集中管理便于维护并且以免发生冲突。

### 本地化字符串

在Xcode中，本地化的字符串均定义在Localizable.strings文件中，使用时采用NSLocalizedString接口调用。遵循以下的规则：

* Localizable.strings中定义的本地化字符串按照功能分块组织，块与块之间间隔一行，块之前添加注释进行说明；
* 公用的本地化字符串应该统一定义到Common块中，放置在文件最上面；
* 本地化字符串的定义按照key-value形式。Key命名规则为“String+功能名+字符串名”；
* 必须保证文件中key值的唯一性；

举例如下：

|  |
| --- |
| // 公共字符串  "StringCommonFinish" = "完成";  "StringCommonCancel" = "取消";  "StringCommonEdit" = "编辑";  // OCR错误提示  "StringOCRErrorNodevice" = "没有检测到摄像头";  "StringOCRErrorNetError" = "网络连接错误，请重试";  "StringOCRErrorNoNetwork" = "没有网络连接";  "StringOCRErrorCantRecongnize" = "识别失败，请再试试"; |

图表 19 本地化字符串

### Javascript语句管理（待定）

JS脚本主要分为两类：一类是简单的JS语句，一般为JS的API；另一类为代码中自定义的JS函数。

#### 简单Javascript语句

* 所有JS语句定义为宏，统一定义到一个单独的头文件中；
* 代码中需要JS语句时均使用定义好的宏，不应该直接写相应的JS语句；
* JS语句对应的宏定义规则为KJS+函数名。函数名的生成规则：取整个代码首尾两个单词的全部，中间单词的首字母，将点号变为下划线进行相连；
* 定义按照接口功能分块组织；

举例如下：

|  |
| --- |
| #pragma mark - JavaScript Document Object  #define KJSDOCUMENT\_URL @"document.URL" // 得到当前页面的  #define KJSDOCUMENT\_AE\_BLUR @"document.activeElement.blur();" |

图表 20 简单JS语句

#### Javascript函数

* 函数调用是随用随读取，降低内存使用量；
* 定义统一的接口，以便代码中需要JS支持之处调用。接口统一定义到一个接口类中，按功能块划分；
* 接口命名统一采用C函数风格，函数前缀统一使用JS；

### 资源管理

* 项目工程中的Resources目录结构必须与本地文件目录一一对应；
* Resources中相同类型（图片，音频，视频等）的文件创建不同的文件夹进行管理，创建工程自动生成的文件位置不变；
* 对于在同一文件夹中的文件必须按照功能进行组织，尽量与代码中的目录结构保持一致；
* 文件或文件夹排序按照名称的字母序排序；
* Xib文件的命名与其对应的UIViewController类名相同；

## 代码质量控制

### 成员变量访问属性

成员变量的访问属性要严格控制，是@private的就不能申明为@protected。在设计一个类的时候要做充分考虑。

### 构造器

* 子类指定的构造器调用父类指定的构造器；
* 把父类指定的构造器返回值赋值给self；
* 加入判断父类指定的构造器返回的值是nil则不要访问实例变量；
* 在覆写的父类指定的构造器中调用子类指定的构造器；
* 在子类化时，确保每个非指定的构造器都要调用指定的构造器。

#### 明确指定构造器

一个类可以有多个构造器，也可以有多个指定的构造器。在声明一个类时，特别的如果这个类会被继承，明确指定的构造器可以确保被子类的构造器调用。

#### 覆写指定的构造器

如果子类需要一个初始化方法，那么确保要覆写父类指定的构造器。否则可能会有情况导致子类的构造器没有被调用。

|  |
| --- |
| @interface MYCircle : NSObject  {  NSPoint center;  float radius;  }  // Designated Initializer  - (id)initWithCenter:(NSPoint)aPoint radius:(float)aRadius;  @end  @implementation MYCircle  // Designated Initializer  - (id)initWithCenter:(NSPoint)aPoint radius:(float)aRadius  {  self = [super init];  if (nil != self)  {  center = aPoint;  radius = aRadius;  }  return self;  }  @end  // Overriden inherited Designated Initializer  - (id)init  {  static const float MYDefaultRadius = 1.0f;  // call Designated Initializer with default arguments  return [self initWithCenter:NSZeroPoint radius:MYDefaultRadius];  } |

图表 22 构造器实现

#### 覆写NSObject方法的位置

应该把构造器和析构器放在一起，而且是放在实现文件的前面，参考2.5小节实现文件布局。

### 初始化

在构造器中无需把实例变量初始化为0或是nil。为一个对象新分配的内存都会被初始化为0，除了isa。

### 避免直接调用+new

避免直接调用NSObject的new方法，也不要在子类中覆写此方法。应该使用alloc或是init方法。

现在OC代码都是通过alloc和init调用来创建一个对象。New很少被使用，这样便于统一，利于code review。

### 使公有接口尽可能简单

尽可能的使类简单化，没有必要公开的接口就不公开。可以使用一个私有的category防止造成public的头混乱。

和C++不一样，OC的方法都是public的所以更需要把非公开的接口隐藏起来，防止出现意外的调用。

私有方法其实并不是真正的私有。我们还是可能会覆写父类的私有方法，这样会造成非常哪里定位问题，所以通用的做法是对于私有方法用一个独特的命名方法以免被无意的覆写。

多用分类使实现模块化。

### #import 和 #include

#import是用来导入OC/OC++的头文件，#include是用来导入C/C++头文件。

### 使用Root框架

直接导入“#import <Foundation/Foundation.h>”而不是导入每个单独的头文件。导入最顶层框架的头文件对于编译器来说做的工作还会更少些，因为通常它是预编译的所以能够被更快的load。

### autorelease

对于一个新创建的临时变量，优先使用autorelease而不是release。这种方式可能会稍微慢些，但是会防止可能存在内存泄露。

避免在一个密集的调用和循环里面使用autorelease，这样会导致autorelease pool迅速被充满，从而导致效率降低。

### 在构造和析构阶段尽量调用访问器方法

|  |
| --- |
| - (id)init  {  self = [super init];  if (self)  {  self.bar = [[[NSMutableString alloc] init] autorelease]; // good  }  return self;  }  - (void)dealloc  {  self.bar = nil; // good  [super dealloc];  }  - (id)init  {  self = [super init];  if (self)  {  self.bar = [NSMutableString string]; // avoid  }  return self;  }  - (void)dealloc  {  self.bar = nil; // avoid  [super dealloc];  } |

图表 23 构造器中调用方法

### 按照申明的次序析构实例变量

* Dealloc中处理实例变量按照@interface中申明的顺序处理，便于review；
* 加入dealloc中调用了其他的方法来release实例变量，则需要加注释说明。

### 避免抛出异常

|  |
| --- |
| @try  {  foo();  }  @catch (NSException \*ex)  {  bar(ex);  }  @finally  {  baz();  } |

图表 24 异常格式

### nil检查

nil检查仅仅是为了逻辑流程而不是用来防止程序崩溃，OC的runtime处理了给一个nil的对象发送消息的情形。这一点不同于检查C/C++是否为NULL。

### BOOL类陷阱

把一个整型值转换为BOOL时要小心。避免直接和YES比较。通常会犯错的情形是把一个数组的大小，指针的值或是一个位的逻辑操作结果强制转换为BOOL，转换之后的结果取决于整数的最后一个字节可能会转换成NO。

举例如下：

|  |
| --- |
| BOOL great = [foo isGreat];  if (great == YES) // avoid  // ...be great!  BOOL great = [foo isGreat];  if (great) // good  // ...be great! |

图表 25 BOOL值

### 自动合成实例变量

如果代码只是在iOS上运行，优先使用自动合成实例变量。

|  |
| --- |
| // Header file  @interface Foo : NSObject  // A guy walks into a bar.  @property(nonatomic, copy) NSString \*bar;  @end  // Implementation file  @interface Foo ()  @property(nonatomic, retain) NSArray \*baz;  @end  @implementation Foo  @end |

图表 26 自动合成变量

### Magic number

一定要消灭除frame, color等以外的Magic number。Magic number不能表达任何意义给后续的开发人员维护和开发造成巨大的心理创伤和负担。

解决方案：

* 用类的枚举来解决magic number
* 用常量来解决magic number
* 改设计和标注消灭magic number

# SVN提交规范

目前开发以主干开发为主，分支开发为辅，保证主干代码的正确性很重要。

SVN会记录提交信息，但是团队成员在共同开发时需要遵循一些代码提交原则以免造成主干代码不能编译出有效的安装包。Log message是一个直观的反应提交修改点的记录，需要规范执行。

## SVN能干什么

下图是从SVN中执行show log命令后显示的SVN提交记录，由此看出SVN会记录如下信息：

* 提交人：SVN自动记录，是用户的登录名
* 提交时间：SVN自动记录
* Log message：由提交的用户填写
* 动作：修改，增加，删除等，SVN自动识别
* Revision：SVN自动分配

****

图表 27 SVN提交记录

## 提交原则

总原则：必须要保证代码在模拟器正常编译，运行才能提交。一次提交针对一个分散功能或是一个bug的改动。

具体步骤如下：

1. 告知团队成员你要提交代码；
2. Update SVN上最新的代码，如果需要merge或是resolve conflict必须先要解决，然后在编译运行。对于需要很多要merge或是很多conflict要解决的提交，必须要找上次提交人一起review；
3. 组内Code review，特别是新人或是功能比较复杂的提交，还有就是项目上线前期做的改动；
4. 提交。

另外，在commit时有两个情况容易出错：

* 增加新文件

增加了新文件，但是commit的时候并没有先在SVN上执行add操作，导致新增文件并没有能够提交到SVN上。结果是编译不能通过，缺少文件。

* 删除文件

只是在IDE里面删除了文件，并没有在SVN上delete掉文件，导致本地代码和SVN代码不同步，结果是文件冗余。

## Log message规范

Log message

  SVN代码提交注释格式：【动作】【模块】备注

  注释样例如下图：



  其中【】使用中文全角

  【动作】可以如下：

  优化

  功能

  提测

  重构

  BUG

  【模块】是该动作影响的模块名，如果是BUG，请填写BUG号，如【BUG】【FLYFLOW-9583】

  备注部分请填写较为详尽的提交缘由

# Code review规范（暂定）

引入Code review机制对提高代码质量有很大的意义。

## 通用原则

Code review过程中需要把控的一些原则：

* Code review时长

对于Code review的时长，建议控制在2个小时之内。

* Code review人数

包括发起code review的人员，至少要有2个人参加，提测阶段应该全体参加。

## 参加人员

* 项目负责人

项目负责人一定要参加每次code review。

* 模块相关RD

与该模块相关的RD一定要参加code review。

* + 直接关系，如A模块使用B模块，一定要参加
    - A代码Review时，B要参加
    - B代码Review时，A要参加
  + 间接关系，A，B，C三模块，A->B->C，C Review时A跟据情况可以不用参加

## Review时机

* 全新模块，第一次提交SVN。
* 全新模块，模块功能完成

针对比较大的功能，建议分开提交，分开做code review。

* 因不可预知的情况，导致代码被调整（非本人修改）
* 代码出现问题，修改后，不明确原因
* 产品上线前，临时代码修改
* 对其它模块会存在风险（调用其它模块的方式改变，更改接口参数）
* 更改实现方案
* 代码重构

## 被Review人员主要工作

介绍该模块完成的功能及工作流程，要求明确说明实现功能。如果是比较大的功能模块需要配以实现文档。

### 讲解流程

* 功能简介
* 实现简介
* 头文件讲解
* 实现文件讲解
* 记录发现的问题，本人尽量使用笔记记录
* 问题修改
* 提交到SVN

### 注意事项

一些特别的情况要引起注意并且review做要特别说明：

* 如有多个实现方案，需要说明各方案的优缺点，说明选取方案的优缺点
* 说明项目共享资源，全局变量，配置文件等，是否修改或有依赖
* 实现的代码中，对最低的系统版本要求
* 证书的修改，更新证书
* 如出现重大问题代码调整，重新再作次Review
* 对QA说明测试重点，及有可能存在的风险及测试注意事项

## Review人员主要工作

* 是否符合预期需求
* 是否符合代码规范
* 是否可以适用在项目中，符合项目整休实现流程
* 是否存在风险，对性能是否有影响
* QA要了解测试重点及实现策略

# 项目提测规范（待定）

指定提测规范细化提测流程，请项目负责人重点关注。

## 提测阶段性划分

提测大致可以分为两个阶段：

1. 开发阶段，亦可称为分功能提测阶段，此时未全功能提测。
2. Bug fix阶段，此时已经全功能提测。

在项目排期的时候项目负责人给出分功能提测的时间点，各功能模块的负责人应当熟悉这些提测的时间点，按时完成相应的功能，以保证分功能提测能够按时进行。

在全功能提测后的bug fix阶段，提测时间动态协商，由项目负责人统筹，但是至少每周要提测两个版本。

## 提测流程

目前iOS客户端产品没有统一的编译机，所有的提测是项目负责人线下编译。以此指定如下提测流程：

1. 提测人员提前告知项目组内人员，今天在什么时候会走提测流程，特别是在项目的第一次提测和全功能提测时。每个功能模块负责人应当确保各自的模块已经通过自测并及时提交到了SVN上；
2. 提测人员Update SVN上最新代码，在本地编译生成真机版本，并确保运行基本功能正常；
3. 更新必要的版本号和日期信息并提交到SVN，提测人员要确保这次提交是这次提测版本的最后一次提交，log message格式如图表 32所示；
4. 编译生成最新的app包，通过itunes打包成ipa包，然后提交到项目icafe的项目文档中，文件命名中要包含提测的版本号；
5. monkey版为debug版，提测版是adhoc版，发布版是distribution版。
6. 给QA接口人和PM发邮件，抄送给RD/PM/UE/QA邮件组。邮件格式如下图所示：

图表 33 提测邮件示例

1. 在项目群里说明已经提测，请QA跟进。

# 项目发版规范

## 提交前确认事项

* 证书及版本

指定证书为发布版证书，Build安装包版本为Distribution版。

* 服务器地址确认为线上地址
* 确认最新代码中没有测试代码
* 确认最低支持系统版本
* 版本号信息

和PM确认发布版本的版本号

* 渠道号信息

和PM确认发布版本的渠道号

* 帮助信息

需要与PM确认本次提测试中的帮助信息是否需要修改。

* QA反馈

活动BUG数是否为0

## 提交流程

## 前期准备

在<https://itunesconnect.apple.com>网站上添写提交版本的相关信息，主要为简介，截图，官网的设置，本步不需要开发人员处理。掌百及百度搜索项目提交孙昱同学在负责，如需要提交版本，需提前沟通。在提交的过程中，有一步需要选对应的产品，就是前期设置过才会有对应的产品可选的。

## 提交审核

目前两个产品的提交都是在Xcode中用archive方式。苹果相关流程和注意实现，请参考官方文档：<https://itunesconnect.apple.com/docs/iTunesConnect_DeveloperGuide_CN.pdf>

# 参考资料

1. http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml, Google Objective-C Style Guide, Google
2. https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html#//apple\_ref/doc/uid/10000146-SW1, Introduction to Coding Guidelines for Cocoa, Apple