一、目的 3

1.0 统一编程风格 3

1.1 写有价值的代码 3

1.2 尊重代码 3

1.3 团队协作 3

1.4 信仰 3

二、项目开发流程规范 3

2.0 约定 3

2.1 命名 4

2.2 文件结构 6

2.3 APP开发初期（没接口、没UI）如何保证开发进度？ 8

2.4 第三方库引用 8

2.5 git使用规范 8

2.6 UI布局方式约定 9

2.7 本地缓存策略 9

2.8 网络通信安全策略 10

2.9 测试用例编写 11

2.10 调试代码标记 11

2.11 日志记录 11

2.12 APP远程控制策略 11

2.13 版本升级控制策略 12

2.14 Bug管理流程 12

2.15 提交测试的流程 12

2.16 提交审核、发布 13

三、架构规范 13

3.0 基础架构 13

3.1 业务架构（模块划分与通信规则） 14

3.2 类内部代码的组织 14

3.3 组件化开发之路 14

3.4 参考《APP应用架构谈》系列文章 14

四、编码规范 14

4.0 点语法 14

4.1 间距 15

4.2 条件判断 15

4.3 三目运算符 15

4.4 方法 15

4.5 变量 16

4.6 注释 16

4.7 init和dealloc 16

4.8 字面量 17

4.9 CGRect 函数 17

4.10 枚举与位掩码 18

4.11 私有属性 18

4.12 单例 18

4.13 @import 19

4.14 运算符前后都留一个空格，单目运算符除外 19

4.15 Control Structures 19

4.16方法命名 19

4.17 数字常量一定要定义名称 20

4.18 初始化Dictionary 20

4.19 BOOL类型陷阱 20

4.20 BOOL 21

# 一、目的

## 1.0 统一编程风格

提高代码的可读性与编码效率，避免团队开发可能带来的混乱。

## 1.1 写有价值的代码

稳定高效、优雅得体、可读性强、逻辑清晰易懂、易扩展、易维护… 代码的价值在于重用，不重复造轮子，提高工作效率。如果写出只能被自己维护的代码，这毫无价值可言！

## 1.2 尊重代码

外观干净、整洁，减少阅读障碍，bug的修复就自然变得容易查找和修复。所以说这是一个相互尊重的过程。把代码作为自己的名片！

## 1.3 团队协作

对项目负责、团队成员负责、对个人负责，不要只为完成工作任务而写代码。要记住自己写的每一行代码都会影响整个项目的进度、影响团队其它成员的引用和维护、影响个人素养的提升！

## 1.4 信仰

代码改变世界！

# 二、项目开发流程规范

## 2.0 约定

规则: 编程时必须遵守的约定

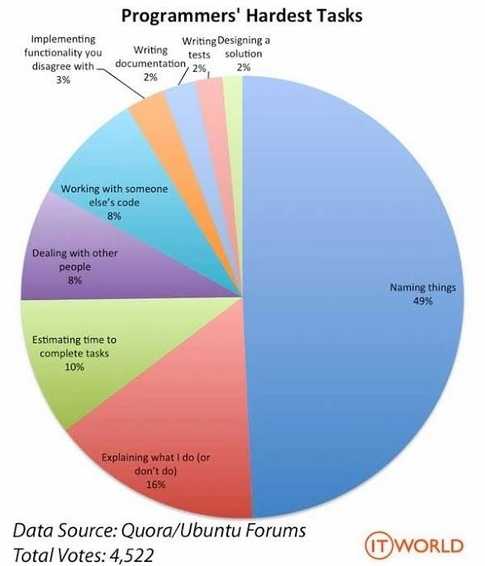
建议: 编程时需要考虑的约定

绿色代码: 对此规则或建议给出的正确例子

红色代码: 对此规则或建议给出的反面例子

## 2.1 命名

“计算机科学的两件难事：缓存失效和命名”

回顾我们平时的习惯：

1.查单词

2.比较单词语义

3.比较代码上下文

4.确定命名

只要把这几个步骤缩短就能节省大量时间。

变量命名神器——[Codelf](http://unbug.github.io/codelf/)

Codelf通过搜索在线开源平台Github, Bitbucket, Google Code, Codeplex, Sourceforge, Fedora Projec的项目源码，帮开发者从中找出已有的匹配关键字的变量名，从而帮助为变量名苦恼的开发者命名。这个搜索服务支持直接搜索中文，你可以立即查看对应的源码，多少能学习到别人的设计，如果代码的业务与自己相投，对梳理自己的业务是有帮助的。

Codelf 由前网易前端工程师知名移动前端调试工具MIHTool的作者基于searchcode和有道词典开发完成，[目前开源在Github上](https://github.com/unbug/codelf)。

### （0）变量命名

1. 属性或成员变量：对象描述 + 类型

如 avatorImageView、loginButton、usernameLabel。

2. 常量：kXxxxXxxx

如 static NSInteger const kMaxRowCount = 4;

针对某些特殊用途的常量可以加固定前缀，这样做的好处是充分利用IDE或插件的代码自动联想（补全）功能，便于快速定位具体的名称，如

定义网络接口名称可以这样kMethodXxxx

定义参数名称可以这样 kParamXxxx

定义默认值（颜色、大小等）可以这样 kDefaultXxxx

3. 局部变量：

4. 宏定义变量：通常不建议这样做，因为变量的类型无法确定；重复宏定义变量如果处理不好会在运行时出问题。

其中每一对象的名称都要求有明确含义，一般都取对象名字的英文全称，只有当对象的名称有大家公认的、比较通用的简写字母代替时可以不用全称，比如APP、HTTP、IO等。

驼峰命名法：就是当变量名或函式名是由一个或多个单字连结在一起，而构成的唯一识别字时，驼峰命名法第一个单字以小写字母开始；第二个单字的首字母大写或每一个单字的首字母都采用大写字母。

### （1）项目命名

项目名称：KanPian

Bundle Identifier：com.公司名称.项目名称，如com.pisen.KanPian

### （2）版本号命名

主版本号.副版本号/次版本号/小版本号.修订版本号 (编译号) 如1.2.1 (13)。

主版本号：产品在核心功能上有较大改进。一般由产品经理或公司领导层确定。

副版本号/次版本号/小版本号：产品的核心功能基本没有变化，次要功能有改进。一般由产品经理确定。

修订版本号：从0开始计数，每次提交AppStore审核需要+1。

编译号：表示打包正式提交测试的次数。（由于iOS可以为任意字符串，而Android强制为数字，所以统一用数字）

上一级有变动，下一级要归0。当基础业务比较稳定，产品比较成熟时可以采用2段式（即取消修订版本号）的版本控制，新产品一般都建议用3段式的版本控制，方便版本bug跟踪、crash收集、用户反馈。

参考



### （3）类命名

业务相关类命名：PSXxxxViewController、PSXxxxCell、PSXxxxView、PSXxxxViewModel

单例类命名：XxxxManager、XxxxSingleData、XxxxConfigData

静态类命名：XxxxHelper、XxxxUtils

模型类命名：XxxxModel

### （4）图片命名

(模块名\_)前缀\_用途(\_方向)(\_颜色)(\_选中)@2x.png 如

User\_icon\_share\_white@2x.png

icon\_arrow\_up\_white@2x.png

Index\_button\_download\_blue@2x.png

background\_login@2x.png

本项目的图片资源全部放在Assets.xcassets中且必须有@2x和@3x，如果有独立封装的控件，图片可以放在【**控件名\_bundle】**文件夹中，且所有图片统一加控件名简称的前缀，方便写代码时自动查找。

### （5）模块命名

头字母大写，Xxxx ru

Home

MicroVideo

UserCenter

Common

Libraries

、、、

### （6）方法命名

见[4.16](#_4.16方法命名)

### （7）宏命名

主要用于代码段简写，单词大写字母\_单词大写字母。如

#define APP\_SHORT\_VERSION [[[NSBundle mainBundle] infoDictionary]

#define SCREEN\_WIDTH ([UIScreen mainScreen].bounds.size.width)

特殊情况可以用宏定义变量名称。如基础框架中通过宏定义了默认值，但需要在外部可配置，此时只需定义宏变量来覆盖默认值。

宏定义实际的变量值：

#define XIB\_WIDTH 750.0f

基础框架定义的默认值

#ifndef XIB\_WIDTH

#define XIB\_WIDTH 640.0f

#endif

## 2.2 文件结构

Modules （本项目所有业务相关的代码）

Home（首页模块）

Controller

View(可选)

Model(可选)

ViewModel(可选)

MicroVideo（微片模块）

User（用户模块）

Login

Register

Common（公共模块）

View

Model

ViewModel

Resources（资源目录）

Images

Plists

Videos

Audios

Htmls

Sqlites

Libraries（第三方库，引用优先级 代码引用 > cocoapods > submodule）

SDWebImageView

AFNetworking

Monsary

…

Utils（工具类）

Managers

Categories

Custom

…

参考：http://www.tuicool.com/articles/QFjEju



## 2.3 APP开发初期（没接口、没UI）如何保证开发进度？

建议：由APP制定出整套接口定义，这样就确定了所有接口的输入与输出；然后模拟接口返回的model，就可以只根据线框图进行主要功能的开发；最后等接口和UI准备好了后再完善细节业务逻辑。

这样做的好处有：

（1）在具体写代码之前，帮助APP端对业务的理解更细致

（2）责任划分清晰。谁定义谁维护，其他人不能随意更改，若要修改需要相关人员一起讨论确定。

（3）有利于APP统一规范参数名称、数据模型、接口分解

（4）两端的开发可以并行不相互依赖

（5）APP开发过程中，测试人员可以协助跟进接口的测试

## 2.4 第三方库引用

CocoaPods：无需修改的、完全由第三方维护的代码。

优点：不影响本项目的工程文件引用，无需额外的工程配置。

缺点：在pod install后，如果代码是svn管理会产生文件夹丢失问题。git没有这样的困扰。

Submodule：仅仅只是对第三方代码库的引用。

优点：不影响本项目的工程文件引用，可与主项目同时开发维护。不存在svn管理问题。

缺点：需要引用相关配置信息。

直接代码引用：最原始的第三方代码引用方式。

优点：可以作为项目源码一部分进行维护，灵活。

缺点：如果有修改不能很好与第三方的最新库同步。

经团队集体讨论决定第三方库引用的优先级为：

直接代码引用 > CocoaPods > Submodule

## 2.5 git使用规范

（1）git提交注释

尽量将注释缩减为一句话，不要包含详细的内容。常用如下：

[Added] 新加入的需求、文件

[Fixed] 修复 bug 描述，如果有issues系统，带上id，如Issue #1234

[Updated] 完成的任务，或者由于第三方模块变化而做的变化

例子：

[Added] some files added.

（2）git合并

合并代码一律用git pull --rebase，以保证代码的变化在一条线上进行；

合并有冲突后一定要处理完冲突后继续pull，直到没有冲突再push；

只有编译通过的代码才push到服务器上，保证任何时刻从服务器拉回来的代码都能正常打包，为以后的持续集成做准备。

（3）git分支管理

master：已上线的、稳定的版本。

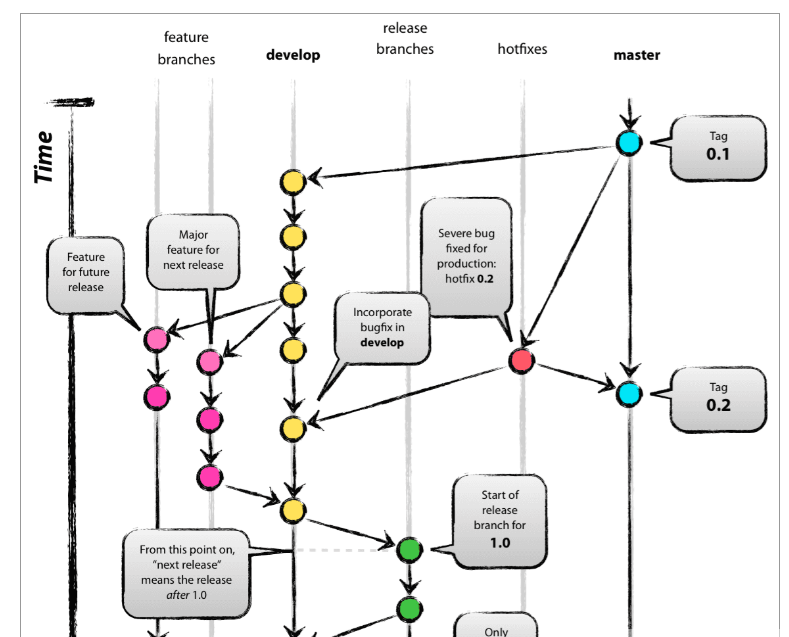
develop：当前开发分支。完成当前版本开发并在审核后合并到master，并打上当前版本号命名的tag，如tag\_v1.3.1。

release：开发完成后准备发布上线的分支。如果有bug随时合并到develop分支上进行修复。

hotfixes：针对已发布上线的版本修复bug所创建的临时分支。修复完成后合并到master。

branch\_feature：新功能分支。如果开发的新功能与master上相同功能有冲突则该分支就完全独立存在；如果是与master上相同功能的补充和完善则合并到master上去。

参考：http://jiongks.name/blog/a-successful-git-branching-model/



## 2.6 UI布局方式约定

在iOS开发过程中，通常有三种方式进行UI布局：Xib、Storyboard、纯代码。不论哪种方式都必须考虑兼容不同Size的设备。

Xib优点：可视化布局约束、开发效率高、维护容易、重用性强

缺点：容易合并冲突，开发iPad不方便，不方便动态修改

Storyboard优点：流程清晰、转场特效、其它与Xib类似

缺点：太多的VC打开缓慢，更容易合并冲突，不方便重用，其它与Xib类似

纯代码布局优点：减少合并冲突、可灵活控制位置大小、通用的UI组件(前提是足够稳定)

缺点：大量UI布局代码，维护不方便，开发效率低

建议：小型中型项目用Storyboard；大型项目用Xib+代码；Storyboard分模块创建

## 2.7 本地缓存策略

### （1）缓存的目的

    在没有网络的情况下依然能保持APP的UI完整性，提升用户体验

### （2）哪些数据需要缓存

    1. 登陆的用户对象

    2. 一些全局单例对象

    3. 网络图片

    4. app运行过程中的参数：开关变量、基本类型的变量

5. 页面数据

### （3）如何缓存数据

1. NSUserDefault

通常存储简单业务逻辑的参数，如首次启动判断、下载最新版判断等。

2. StorageManager

页面缓存、复杂对象的序列化和反序列化、用户的个性化设置、token等。

3. sqlite/CoreData

查询比较频繁的大数据、排序要求较复杂的数据。

4.keychain

保存比较重要的数据，比如密码、秘钥等。与ADP相关，相同账号的不同app可以共享数据，且app在卸载后也不能清除（除非全部卸载该账号下所有app）。升级ios不会清除，重置和重装系统要清除。可以保证相同开发账号的所有程序在相同设备能够获取到不变的UDID 。

### （4）缓存数据的清理

一般情况下

1.  app运行过程中长期驻留内存的缓存数据，不能在设置里清除，如单例。

2. 网络图片随时都可以清除

3. 页面数据随时都可以清除

4. 基本类型的变量、开关变量需要视情况而定

参考：http://www.tuicool.com/articles/jmuuia7

### （5）缓存数据放在什么位置

需要在升级后继续使用的数据放在：

公共目录：Documents/APP\_NAME\_CACHE/

用户目录：Documents/ APP\_NAME\_CACHE /UserId/

不需要备份，app升级后直接丢弃的数据放在：

公共目录：Library/Caches/ APP\_NAME\_CACHE /

用户目录：Library/Caches/ APP\_NAME\_CACHE /UserId

## 2.8 网络通信安全策略

常用加密方式：AES加密、DES加密、RSA加密、MD5、POST传输、动态秘钥

## 2.9 测试用例编写

规范待补充……

## 2.10 调试代码标记

开发过程中，为了提高开发效率，减少一些不必要的bug出现，建议：在需要的地方打个标记，可以方便自己对代码流程进行管理。

TODO：表示功能未完成，用于开发功能框架时简化处理细节问题的提示语。

FIXME：业务逻辑可能考虑不完整，有bug。

TEST：打破常规业务流程只测试单个功能，组装假数据。如，

//>>>>>>>>>>>TEST:组装detailModel>>>>>>>>>>>>>

DetailModel \*model = [DetailModel new];

…

//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

标记代码在发布测试之前一定要处理完！

## 2.11 日志记录

方便对测试人员提出的非必现的bug进行跟踪，Crash日志可以通过第三方平台（如UMeng）搜集。

日志记录在正式发布上线时需要关闭！

## 2.12 APP远程控制策略

一般Native APP发布出去后，业务逻辑都由本地代码控制的，如果要修改就得重新修改代码、编译、测试、上线新版本（iOS还得走AppStore的审核流程）。这样就极大地延长了新版本的发布周期，特别是针对某些不影响主要业务逻辑的、非常小的修改（比如功能的隐藏与否、提示语的更改等），或者当APP出现致命或影响严重的bug，需要立即马上修复，如果仍然还要走复杂的流程的话就显得非常不合理。除此之外，远程控制还可以：

1. AppStore审核时可以临时关闭某些功能，避免审核被拒。
2. 测试阶段可以灵活修改某台设备对应的接口地址，方便后台调试接口。
3. 当接口的域名被劫持时可以远程修改为接口的IP地址。
4. 可以控制某些功能在特殊时间内才放出来，这对运营非常有用。
5. 可以内嵌消息推送、强制版本升级等功能。

针对以上需求，现提供两种解决方案：

1. 在线参数配置

参考了UMeng、ShareSDK等第三方常用平台提供的在线参数配置功能，但无法满足多版本、多设备、多APP、多平台的控制要求，故提出以下在线参数配置方案（仅供参考）：

AppParamName



AppParamValue:



优势：可以只针对某类版本（如1.\*、1.2.\*）的参数进行特殊设置；可以只针对某个设备（udid）的参数进行单独设置，而其它所有版本和设备不受影响。一般用于开关变量、常量、枚举变量的定义。

劣势：需要预先定义好针对不同变量的业务逻辑并设置好默认值；无法修改或新增业务逻辑。

1. [JSPatch](http://www.jspatch.com/)

通过分发JS代码到APP客户端，利用iOS的runtime特性（swizzling）替换原来的方法来达到改变代码逻辑的目的。

## 2.13 版本升级控制策略

一般走AppStore自动升级流程，但为了更优化用户体验，当有新版本发布时，app一启动就能发现并及时提醒用户。

版本升级分为：用户自己确定是否升级、出现重大漏洞需要强制升级。一般情况下，主版本号的更新需要强制升级，其它版本的变化由用户自己决定。

接口需要考虑多版本同时运行的情况。建议通过接口地址后添加版本号（v1/）来控制，不建议在已有的接口里进行兼容性修改，因为这样容易影响已上线的版本，而且增加测试人员不必要的工作量。

## 2.14 Bug管理流程

通过禅道系统进行管理。bug一般分为以下等级：

1级：致命。必须立刻马上处理

2级：非常重要。

3级：重要

4级：一般

原则上必须修改完1、2、3级的所有bug才允许上线。修改完成一个bug必须更新禅道，同时提交代码库。

## 2.15 提交测试的流程

删除所有标记的调试代码 ->关闭DEBUG -> BundleIdentifier+1 -> 修改为生产证书 -> Archive -> 上传itunesConnect -> 手动发布外测（内测是自动发布的）。

提前itunesConnect上添加外测人员的AppleId，并加入该测试项目；针对iOS7系统的测试，由于iOS7不支持testFlight的安装，所以只能打包成ipa文件上传到禅道的版本管理列表。

当testflight上可以下载测试版本时，在禅道上新建一条对应的版本记录用来保存该版本的所有的bug。提交内测阶段，每天下班前必须合并所有代码并打包提交itunesConnect，方便测试人员第二天能快速进入验证前一天修改的bug的工作状态。

## 2.16 提交审核、发布

由开发人员负责上传APP并填好审核需要的相关信息，然后由测试人员控制什么时候提交审核，由产品经理或运营控制何时发布。

# 三、架构规范

## 3.0 基础架构

处理除了业务实现以外的所有逻辑，至少应该包括：网络请求、模型映射、缓存策略、安全策略、常用宏定义、日志记录、基础类的常用扩展。

一套稳定的基础架构本身的代码在任何项目中都不应该被直接修改。基于该思想，基础框架必须实现动态配置策略。常用配置策略有：

1. 宏定义

在基础架构中判断是否定义过某变量，如果没有就定义一个默认值。优点是简单、直接。缺点是容易被滥用，不能指定变量的类型。

1. 缓存本地配置文件

类似NSUserDefaults的作用。

1. 统一用一个单例处理配置信息，用之前先设置这个单例

推荐该方法，优点：可以统一管理所有配置信息；可以在单例初始化设置默认值；变量的类型明确。

1. 控制反转+依赖注入+工厂方法

将代码的业务逻辑控制权上交给配置文件，通过依赖注入的方式实例化具体的对象，来达到改变业务逻辑的目的。其中可以采用工厂模式（或抽象工厂模式）统一实例化对象。

1. category重写class的同名方法（iOS常用）

特别注意不要在多个category对相同方法进行重写，因为无法确定最终起作用的category是哪个。

1. if (respondsToSelector:@selector(@"className"))

performSelector:@selecor(@"className") withObject:object

iOS常用，本质和方法（1）类似，缺点也是容易被滥用。常用在判断iOS不同版本某些方法是否存在。

1. 方法（6）的扩展：用protocol定义好规范，框架中只处理delegate

虽然有protocol定义规范，但在具体调用的时候还是需要先判断该方法是否存在，感觉有些多余。

1. UI\_APPEARANCE\_SELECTOR

与方法（3）类似，但仅限于某个特定的类。

1. 链式编程（更加优雅的配置方式），如masonry。

## 3.1 业务架构（模块划分与通信规则）

参考：http://www.tuicool.com/articles/3iArAnm

## 3.2 类内部代码的组织

代码组织的原则：

同一个对象的配置放在一起；

同一个区域的功能放在一起；

按照页面布局从上往下组织代码；

按照业务逻辑先准备view，再渲染数据，然后封装业务逻辑的思路自上而下组织代码；

私有方法是内部某项单一功能的封装，仅被内部调用；

共有方法仅仅只是对内部业务的封装，提供对外的统一入口；

参考类结构：

Life cycle + setup subviews + data > events > notification > delegate > getter & setter > private method > public method

另外，如果页面的功能比较复杂，可以分功能模块进行独立封装，独立模块的类结构也应该遵循基本类结构。

参考：<http://www.cocoachina.com/ios/20150525/11919.html>

## 3.3 组件化开发之路

探索中…

## 3.4 参考《APP应用架构谈》系列文章

参考：http://www.tuicool.com/articles/byIjaa

# 四、编码规范

## 4.0 点语法

应该 始终 使用点语法来访问或者修改属性，访问其他实例时首选括号。

推荐：

view.backgroundColor = [UIColor orangeColor];

[UIApplication sharedApplication].delegate;

反对：

[view setBackgroundColor:[UIColor orangeColor]];

UIApplication.sharedApplication.delegate;

## 4.1 间距

缩进用tab一次性缩进4个空格；方法的大括号和其他的大括号（if/else/switch/while 等等）始终和声明在同一行开始，在新的一行结束

if (user.isHappy) {

// Do something

}

else {

// Do something else

}

方法之间空一行；在方法中的功能块之间用空白分开并做适当注释

## 4.2 条件判断

推荐：

if (!error) {

return success;

}

反对：

if (!error)

return success;

if (!error) return success;

## 4.3 三目运算符

只有当它可以增加代码清晰度或整洁时才使用。单一的条件都应该优先考虑使用。多条件时通常使用 if 语句会更易懂，或者重构为实例变量。

推荐：

result = a > b ? x : y;

反对：

result = a > b ? x = c > d ? c : d : y;

## 4.4 方法

在方法签名中，在 -/+ 符号后应该有一个空格，方法声明与第一个{之间应该有一个空格，方法片段之间也应该有一个空格。

推荐：

- (void)setExampleText:(NSString \*)text image:(UIImage \*)image {

}

## 4.5 变量

变量名应该尽可能命名为描述性的。除了 for() 循环外，其他情况都应该避免使用单字母的变量名。 星号表示指针属于变量，例如：NSString \*text 不要写成 NSString\* text 或者 NSString \* text ，常量除外。 尽量定义属性来代替直接使用实例变量。

推荐：

@interface NYTSection: NSObject

@property (nonatomic, strong) NSString \*headline;

int numErrors;

int numCompletedConnections;

tickets = [[NSMutableArray alloc] init];

userInfo = [someObject object];

port = [network port];

@end

反对：

@interface NYTSection : NSObject {

NSString \*headline;

}

int w;

int nerr;

int nCompConns;

tix = [[NSMutableArray alloc] init];

obj = [someObject object];

p = [network port];

## 4.6 注释

/\*\*

\* 功能描述

\*

\* @param

\* @return

\* @sample

\* @author

\*

\*/

## 4.7 init和dealloc

dealloc 方法应该放在实现文件的最上面，并且刚好在 @synthesize 和 @dynamic 语句的后面。在任何类中，init 都应该直接放在 dealloc 方法的下面。

- (void)dealloc {

NSLog(@“%@ is reallocing…”, [self class]);

}

- (instancetype)init {

self = [super init]; // 或者调用指定的初始化方法

if (self) {

// Custom initialization

}

return self;

}

## 4.8 字面量

每当创建 NSString， NSDictionary， NSArray，和 NSNumber 类的不可变实例时，都应该使用字面量。要注意 nil 值不能传给 NSArray 和 NSDictionary 字面量，这样做会导致崩溃。

推荐：

NSArray \*names = @[@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul"];

NSDictionary \*productManagers = @{@"iPhone" : @"Kate", @"iPad" : @"Kamal", @"Mobile Web" : @"Bill"};

NSNumber \*shouldUseLiterals = @YES;

NSNumber \*buildingZIPCode = @10018;

反对：

NSArray \*names = [NSArray arrayWithObjects:@"Brian", @"Matt", @"Chris", @"Alex", @"Steve", @"Paul", nil];

NSDictionary \*productManagers = [NSDictionary dictionaryWithObjectsAndKeys: @"Kate", @"iPhone", @"Kamal", @"iPad", @"Bill", @"Mobile Web", nil];

NSNumber \*shouldUseLiterals = [NSNumber numberWithBool:YES];

NSNumber \*buildingZIPCode = [NSNumber numberWithInteger:10018];

## 4.9 CGRect 函数

当访问一个 CGRect 的 x， y， width， height 时，应该使用CGGeometry 函数代替直接访问结构体成员。

推荐：

CGRect frame = self.view.frame;

CGFloat x = CGRectGetMinX(frame);

CGFloat y = CGRectGetMinY(frame);

CGFloat width = CGRectGetWidth(frame);

CGFloat height = CGRectGetHeight(frame);

反对：

CGRect frame = self.view.frame;

CGFloat x = frame.origin.x;

CGFloat y = frame.origin.y;

CGFloat width = frame.size.width;

CGFloat height = frame.size.height;

## 4.10 枚举与位掩码

当使用 enum 时，建议使用新的基础类型规范，因为它具有更强的类型检查和代码补全功能。现在 SDK 包含了一个宏来鼓励使用使用新的基础类型 - NS\_ENUM()

typedef NS\_ENUM(NSInteger, NYTAdRequestState) {

NYTAdRequestStateInactive,

NYTAdRequestStateLoading

};

当用到位掩码时，使用 NS\_OPTIONS 宏。

typedef NS\_OPTIONS(NSUInteger, NYTAdCategory) {

NYTAdCategoryAutos = 1 << 0,

NYTAdCategoryJobs = 1 << 1,

NYTAdCategoryRealState = 1 << 2,

NYTAdCategoryTechnology = 1 << 3

};

## 4.11 私有属性

私有属性应该声明在类实现文件的延展（匿名的类目）中。如

@interface NYTAdvertisement ()

@property (nonatomic, strong) GADBannerView \*googleAdView;

@property (nonatomic, strong) ADBannerView \*iAdView;

@property (nonatomic, strong) UIWebView \*adXWebView;

@end

推荐:

@property (weak, nonatomic) id<SGOAnalyticsDelegate> analyticsDelegate;

反对:

@property (nonatomic) id <SGOAnalyticsDelegate> analyticsDelegate;

## 4.12 单例

单例对象应该使用线程安全的模式创建共享的实例。

+ (instancetype)sharedInstance {

static id sharedInstance = nil;

static dispatch\_once\_t onceToken;

dispatch\_once(&onceToken, ^{

sharedInstance = [[self alloc] init];

});

return sharedInstance;

}

## 4.13 @import

如果有一个以上的 import 语句，就对这些语句进行分组。每个分组的注释是可选的。

注：对于模块使用 @import 语法。

// Frameworks

@import QuartzCore;

// Models

#import "NYTUser.h"

// Views

#import "NYTButton.h"

#import "NYTUserView.h"

## 4.14 运算符前后都留一个空格，单目运算符除外

NSInteger index = rand() % 50 + 25;

index++;

index += 1;

index--;

if (alpha + beta <= 0) && (kappa + phi > 0) {

}

## 4.15 Control Structures

if ( ! goodCondition) return;

if (condition is YES) {

// do stuff

} else {

// do other stuff

}

## 4.16方法命名

避免使用’andXxxx’、’withXxxx’ 类似的方式来连接参数。

推荐:

- (NSInteger)runModalForDirectory:(NSString \*)path file:(NSString \*)name types:(NSArray \*)fileTypes;

反对:

- (NSInteger)runModalForDirectory:(NSString \*)path andFile:(NSString \*)name andTypes:(NSArray \*)fileTypes;

方法参数类型的小括号前后都不能有空格。

Good:

- (void)setExample:(NSString \*)text;

Bad:

- (void)setExample: (NSString \*)text;

- (void)setExample:(NSString \*) text;

## 4.17 数字常量一定要定义名称

推荐:

if ([pin length] > RBKPinSizeMax)

反对:

if ([pin length] < 5) What is this checking for?

## 4.18 初始化Dictionary

NSDictionary \*keyedStuff = @{

@"this.key": @"corresponds to this value",

@"otherKey": @"remoteData.payload",

@"some": @"more",

@"JSON": @"keys",

@"and": @"stuff",

};

## 4.19 BOOL类型陷阱

BOOL在Objective-C里被定义为unsignedchar，这意味着它不仅仅只有YES(1)和NO(0)两个值。不要直接把整形强制转换为BOOL型，把整形转为BOOL型的时候请使用ternery操作符，保证返回YES或NO值。

对BOOL值使用逻辑运算(&&，||，!)都是有效的，返回值也可以安全的转为BOOL型而不需要ternery操作符。

- (BOOL)isBold {

return [self fontTraits] & NSFontBoldTrait;

}

- (BOOL)isValid {

return [self stringValue];

}

- (BOOL)isBold {

return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;

}

- (BOOL)isValid {

return [self stringValue] != nil;

}

- (BOOL)isEnabled {

return [self isValid] && [self isBold];

}

## 4.20 BOOL

因为 nil 解析为 NO，所以没有必要在条件中与它进行比较。永远不要直接和 YES 进行比较，因为 YES 被定义为 1，而 BOOL 可以多达 8 位。这使得整个文件有更多的一致性和更大的视觉清晰度。

推荐：

if (!someObject) {

}

反对：

if (someObject == nil) {

}

推荐：

if (isAwesome)

if ( ! [someObject boolValue])

反对：

if ([someObject boolValue] == NO)

if (isAwesome == YES) // 永远别这么做

如果一个 BOOL 属性名称是一个形容词，属性可以省略 “is” 前缀，但为 get 访问器指定一个惯用的名字，例如：

@property (assign, getter=isEditable) BOOL editable;

参考：

https://github.com/NYTimes/objective-c-style-guide/blob/master/README\_zh-Hans.md

https://github.com/RobotsAndPencils/objective-c-style-guide