Table of Contents

介绍

该文档主要面向iOS开发团队，以Coding Guidelines for Cocoa为基础针对编码、项目架构、开发实践等进行规范和约定。持续更新中.....

Summary

序言

> 本篇转载自为什么谷歌要执行严格的代码编写规范。原文：Stuff Everyone Should Do (part 2): Coding Standards

我们在谷歌所做事情中另外一个让我感到异常有效、有用的制度是严格的编码规范。

在到Google工作之前，我一直认为编码规范没有什么用处。我坚信这些规范都是官僚制度下产生的浪费大家的编程时间、影响人们开发效率的东西。

我是大错特错了。

在谷歌，我可以查看任何的代码，进入所有谷歌的代码库，我有权查看它们。事实上，这种权限是很少人能拥有的。但是，让我感到惊讶的却是，如此多的编码规范—缩进，命名，文件结构，注释风格—这一切让我出乎意料的轻松的阅读任意一段代码，并轻易的看懂它们。这让我震惊—因为我以为这些规范是微不足道的东西。它们不可能有这么大的作用—但它们却起到了这么大的作用。当你发现只通过看程序的基本语法结构就能读懂一段代码，这种时间上的节省不能不让人震撼！

反对编码规范的人很多，下面是一些常见的理由，对于这些理由，我以前是深信不疑。

> ## 这是浪费时间！
>
> 我是一个优秀的程序员，我不愿意浪费时间干这些愚蠢的事。我的技术很好，我可以写出清晰的、易于理解的代码。为什么我要浪费时间遵守这些愚蠢的规范？答案是：统一是有价值的。就像我前面说的—你看到的任何的一行代码—不论是由你写的，还是由你身边的同事，还是由一个跟你相差11个时区的距离人写的—它们都有统一的结构，相同的命名规范—这带来的效果是巨大的。你只需要花这么少的功夫就能看懂一个你不熟悉(或完全未见过)的程序，因为你一见它们就会觉得面熟。
>
> ## 我是个艺术家！
>
> 这种话很滑稽，但它反映了一种常见的抱怨。我们程序员对于自己的编码风格通常怀有很高的自负。我写出的的代码的确能反映出我的一些特质，它是我思考的一种体现。它是我的技能和创造力的印证。如果你强迫我遵守什么愚蠢的规范，这是在打压我的创造力。可问题是，你的风格里的重要的部分，它对你的思想和创造力的体现，并不是藏身于这些微不足道的句法形式里。(如果是的话，那么，你是一个相当糟糕的程序员。)规范事实上可以让人们可以更容易的看出你的创造力—因为他们看明白了你的作品，人们对你的认识不会因不熟悉的编码形式而受到干扰。
>
> ## 所有人都能穿的鞋不会合任何人的脚！
>
> 如果你使用的编码规范并不是为你的项目专门设计的，它对你的项目也许并不是最佳方案。这没事。同样，这只是语法：非最优并不表示是不好。对你的项目来说它不是最理想的，但并不能表明它不值得遵守。不错，对于你的项目，你并没有从中获得该有的好处，但对于一个大型公司来说，它带来的好处是巨大的。除此之外，专门针对某个项目制定编码规范一般效果会更好。一个项目拥有自己的编码风格无可厚非。但是，根据我的经验，在一个大型公司里，你最好有一个统一的编码规范，特定项目可以扩展自己特定的项目方言和结构。
>
> ## 我善长制定编码规范！
>
> 这应该是最常见的抱怨类型了。它是其它几种反对声音的混合体，但它却有自身态度的直接表现。有一部分反对者深信，他们是比制定编码规范的人更好的程序员，俯身屈从这些小学生制定的规范，将会降低代码的质量。对于此，客气点说，就是胡扯。纯属傲慢自大，荒唐可笑。事实上他们的意思就是，没有人配得上给他们制定规范，对他们的代码的任何改动都是一种破坏。如果参照任何一种合理的编码规范，你都不能写出合格的代码，那只能说你是个烂程序员。

当你按照某种编码规范进行编程时，必然会有某些地方让你摇头不爽。肯定会在某些地方你的编码风格会优于这些规范。但是，这不重要。在某些地方，编码规范也有优于你的编程风格的时候。但是，这也不重要。只要这规范不是完全的不可理喻，在程序的可理解性上得到

的好处会大大的补偿你的损失。

但是，如果编码规范真的是完全不可理喻呢？

如果是这样，那就麻烦了：你被糟蹋了。但这并不是因为这荒谬的编码规范。这是因为你在跟一群蠢货一起工作。想通过把编码规范制定的足够荒谬来阻止一个优秀的程序员写出优秀的代码，这需要努力。这需要一个执著的、冷静的、进了水的大脑。如果这群蠢货能强行颁布不可用的编码规范，那他们就能干出其它很多傻事情。如果你为这群蠢货干活，你的确被糟蹋了—不论你干什么、有没有规范。(我并不是说罕有公司被一群蠢货管理；事实很不幸，我们这个世界从来就不缺蠢货，而且很多蠢货都拥有自己的公司。)

代码命名基础

一般原则

清晰性

- 最好是既清晰又简短,但不要为简短而丧失清晰性

代码	点评
insertObject: atIndex:	好
insert: at:	不清晰：要插入什么？“at”表示什么？
removeObjectAtIndex:	好
removeObject:	这样也不错,因为方法是移除作为参数的对象
remove:	不清晰：要移除什么?

- 名称通常不缩写,即使名称很长,也要拼写完全。（使用US英文，禁止拼音）

代码	点评
destinationSelection:	好
destSel:	不好
setBackgroundColor:	好
setBkgdColor:	不好

你可能会认为某个缩写广为人知,但有可能并非如此,尤其是当你的代码被来自不同文化和语言背景的开发人员所使用时。

- 然而你可以使用少数非常常见,历史悠久的缩写。请参考可接受的缩略名一节
- 避免使用有歧义的 API 名称,如那些能被理解成多种意思的方法名称

代码	点评
sendPort:	是发送端口还是返回一个发送端口？
displayName:	是显示一个名称还是返回用户界面中控件的标题？

一致性

- 尽可能使用与Cocoa编程接口命名保持一致的名称。如果你不太确定某个命名的一致性，请浏览一下头文件或参考文档中的范例。
- 在使用多态方法的类中，命名的一致性非常重要，在不同类中实现相同功能的方法应该具有相同的名称。

代码	点评
- (int) tag	在 NSView，NSCell，NSControl 中有定义
- (void)setStringValue:(NSString *)	在许多Cocoa类中有定义

更多请看方法参数

前缀

通常，软件会被打包成一个框架或多个紧密相关的框架(如 Foundation 和 Application Kit 框架)。但由于Cocoa没有像C++一样的命名空间概念，所以我们只能用前缀来区分软件的功能范畴，防止命名冲突。

- 类名和常量应该始终使用三个字母的前缀（例如 NYT），因为两个字母的前缀是苹果SDK先使用的，但 Core Data 实体名称可以省略。为了代码清晰，常量应该使用相关类的名字作为前缀并使用驼峰命名法。

下面是常见的苹果官方的前缀

前缀	**Cocoa** 框架
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

比如在我们的APP中蓝牙模块（Bluetooth low energy）管理类

```
@interface BLEManager : NSObject
@property (assign) BLEDeviceType deviceType;
@end
```

书写约定

在为 API 元素命名时,请遵循如下一些简单的书写约定

- 对于包含多个单词的名称,不要使用标点符号作为名称的一部分或作为分隔符(下划线,破折号等); 此外,大写每个单词的首字符并将这些单词连续拼写在一起。请注意以下限制:
    - 方法名小写第一个单词的首字符,大写后续所有单词的首字符。方法名不使用前缀。如: fileExistsAtPath:isDirectory: 如果方法名以一个广为人知的大写首字母缩略词开头,该规则不适 用,如:NSImage 中的 TIFFRepresentation
    - 函数名和常量名使用与其关联类相同的前缀,并且要大写前缀后面所有单词的首字符。如: NSRunAlertPanel，NSCellDisabled
    - 避免使用下划线来表示名称的私有属性。苹果公司保留该方式的使用。如果第三方这样使用可能会导致命名冲突,他们可能会在无意中用自己的方法覆盖掉已有的私有方法,这会导致严重的后 果。请参考私有方法一节以了解私有 API 的命名约定的建议

类与协议命名

类名应包含一个明确描述该类(或类的对象)是什么或做什么的名词。类名要有合适的前缀(请参考前缀 一节)。Foundation 及 Application Kit 有很多这样例子,如:NSString, NSData, NSScanner, NSApplication, NSButton 以及 UIButton。

协议应该根据对方法的行为分组方式来命名。

- 大多数协议仅组合一组相关的方法,而不关联任何类,这种协议的命名应该使用动名词(ing),以不与 类名混淆。

代码	点评
NSLocking	good
NSLock	糟糕,它看起来像类名

- 有些协议组合一些彼此无关的方法(这样做是避免创建多个独立的小协议)。这样的协议倾向于与某个类关联在一起,该类是协议的主要体现者。在这种情形,我们约定协议的名称与该类同名。NSObject协议就是这样一个例子。这个协议组合一组彼此无关的方法,有用于查询对象在其类层次中位置的方法,有使之能调用特殊方法的方法以及用于增减引用计数的方法。由于 NSObject 是这些方法的主要体现者,所以我们用类的名称命名这个协议。

头文件

头文件的命名方式很重要,我们可以根据其命名知晓头文件的内容。

- 声明孤立的类或协议:将孤立的类或协议声明放置在单独的头文件中,该头文件名称与类或协议同名

头文件	声明
NSApplication.h	NSApplication 类

- 声明相关联的类或协议:将相关联的声明(类,类别及协议) 放置在一个头文件中,该头文件名称与主要的类,类别,协议的名字相同。

头文件	声明
NSString.h	NSString 和 NSMutableString 类
NSLock.h	NSLocking 协议和 NSLock, NSConditionLock, NSRecursiveLock 类

- 包含框架头文件:每个框架应该包含一个与框架同名的头文件,该头文件包含该框架所有公开的头文件。

头文件	声明
Foundation.h	Foundation.framework

- 为已有框架中的某个类扩展 API:如果要在一个框架中声明属于另一个框架某个类的范畴类的方法, 该头文件的命名形式为:原类名+“Additions”。如 Application Kit 中的 NSBundleAdditions.h
- 相关联的函数与数据类型:将相联的函数,常量,结构体以及其他数据类型放置到一个头文件中,并以合适的名字命名。如 Application Kit 中的 NSGraphics.h

方法(Method)命名

一般性原则

为方法命名时,请考虑如下一些一般性规则:

- 方法名不要使用 `new` 作为前缀。
- 小写第一个单词的首字符,大写随后单词的首字符,不使用前缀。请参考书写约定一节。有两种例外情况:1.方法名以广为人知的大写字母缩略词(如 TIFF or PDF)开头;2.私有方法可以使用统一的前缀来分组和辨识,请参考私有方法一节
- 表示对象行为的方法,名称以动词开头:

```
- (void) invokeWithTarget:(id)target:
- (void) selectTabViewItem:(NSTableViewItem *)tableViewItem
```

名称中不要出现 do或does,因为这些助动词没什么实际意义。也不要在动词前使用副词或形容词修饰。

- 如果方法返回方法接收者的某个属性,直接用属性名称命名。不要使用 get，除非是间接返回一个或多个值。请参考访问方法一节。

代码	点评
- (NSSize) cellSize;	对
- (NSSize) calcCellSize;	错
- (NSSize) getCellSize;	错

- 参数要用描述该参数的关键字命名

代码	点评
- (void) sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;	对
- (void) sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;	错

- 参数前面的单词要能描述该参数。

代码	点评
- (id) viewWithTag:(int)aTag;	对
- (id) taggedView:(int)aTag;	错

- 细化基类中的已有方法:创建一个新方法,其名称是在被细化方法名称后面追加参数关键词

```
- (id)initWithFrame:(CGRect)frameRect;//NSView, UIView.
- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode  cellClass:(Class)factoryId numb
erOfRows:(int)rowsHigh numberOfColumns (int)colsWide;//NSMatrix, a subclass of NSView
```

- 不要使用 and 来连接用属性作参数的关键字

代码	点评
- (int)runModalForDirectory:(NSString*)path file:(NSString*)name types:(NSArray *)fileTypes;	对
- (int)runModalForDirectory:(NSString*)path andFile:(NSString*)name andTypes:(NSArray *)fileTypes;	错

虽然上面的例子中使用 add 看起来也不错,但当你方法有太多参数关键字时就有问题。

- 如果方法描述两种独立的行为,使用 and 来串接它们

```
- (BOOL) openFile:(NSString *)fullPath withApplication:(NSString NSWorkspace *)appName
 andDeactivate:(BOOL)flag;//NSWorkspace.
```

访问方法

访问方法是对象属性的读取与设置方法。其命名有特定的格式依赖于属性的描述内容。

- 如果属性是用名词描述的,则命名格式为:

```
- (void) setNoun:(type)aNoun;
- (type) noun;
```

例如:

```
- (void) setgColor:(NSColor *)aColor;
- (NSColor *) color;
```

- 如果属性是用形容词描述的,则命名格式为:

```
- (void) setAdjective:(BOOL)flag;
- (BOOL) isAdjective;
```

例如:

```
- (void) setEditable:(BOOL)flag;
- (BOOL) isEditable;
```

- 如果属性是用动词描述的,则命名格式为:(动词要用现在时时态)

```
- (void) setVerbObject:(BOOL)flag;
- (BOOL) verbObject;
```

例如:

```
- (void) setShowAlpha:(BOOL)flag;
- (BOOL) showsAlpha;
```

- 不要使用动词的过去分词形式作形容词使用

```
- (void)setAcceptsGlyphInfo:(BOOL)flag;  //对
- (BOOL)acceptsGlyphInfo;                //对
- (void)setGlyphInfoAccepted:(BOOL)flag; //错
- (BOOL)glyphInfoAccepted;               //错
```

- 可以使用情态动词(can, should, will 等)来提高清晰性,但不要使用 do 或 does

```
- (void) setCanHide:(BOOL)flag;             //对
- (BOOL) canHide;                           //对
- (void) setShouldCloseDocument:(BOOL)flag; //对
- (void) shouldCloseDocument;               //对
- (void) setDoseAcceptGlyphInfo:(BOOL)flag; //错
- (BOOL) doseAcceptGlyphInfo;               //错
```

- 只有在方法需要间接返回多个值的情况下,才使用 get

```
- (void) getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;  //NSBe
zierPath
```

像上面这样的方法,在其实现里应允许接受 NULL 作为其 in-out 参数,以表示调用者对一个或多个返回值不感兴趣。

委托方法

委托方法是那些在特定事件发生时可被对象调用,并声明在对象的委托类中的方法。它们有独特的命名约定,这些命名约定同样也适用于对象的数据源方法。

- 名称以标示发送消息的对象的类名开头,省略类名的前缀并小写类第一个字符

```
- (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
```

- 冒号紧跟在类名之后(随后的那个参数表示委派的对象)。该规则不适用于只有一个 sender 参数的方法

```
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender;
```

- 上面的那条规则也不适用于响应通知的方法。在这种情况下,方法的唯一参数表示通知对象

```
- (void)windowDidChangeScreen:(NSNotification *)notification;
```

- 用于通知委托对象操作即将发生或已经发生的方法名中要使用 did 或 will

```
- (void)browserDidScroll:(NSBrowser *)sender;
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;
```

- 用于询问委托对象可否执行某操作的方法名中可使用 did 或 will,但最好使用 should

```
- (BOOL)windowShouldClose:(id)sender;
```

集合方法

管理对象(集合中的对象被称之为元素)的集合类,约定要具备如下形式的方法:

```
- (void)addElement:(elementType)adObj;
- (void)removeElement:(elementType)anObj;
- (NSArray *)elements;
```

例如:

```
- (void)addLayoutManager:(NSLayoutManager *)adObj;
- (void)removeLayoutManager:(NSLayoutManager *)anObj;
- (NSArray *)layoutManagers;
```

集合方法命名有如下一些限制和约定:

- 如果集合中的元素无序,返回 NSSet,而不是 NSArray
- 如果将元素插入指定位置的功能很重要,则需具备如下方法:

```
- (void)insertElement:(elementType)anObj atIndex:(int)index;
- (void)removeElementAtIndex:(int)index;
```

集合方法的实现要考虑如下细节:

- 以上集合类方法通常负责管理元素的所有者关系,在 add 或 insert 的实现代码里会 retain 元素,在 remove 的实现代码中会 release 元素
- 当被插入的对象需要持有指向集合对象的指针时,通常使用 set... 来命名其设置该指针的方法,且不要 retain 集合对象。比如上面的 insertLayerManager:atIndex: 这种情形,NSLayoutManager 类使用如下方法:

```
- (void)setTextStorage:(NSTextStorage *)textStorage;
- (NSTextStorage *)textStorage;
```

通常你不会直接调用 setTextStorage:,而是覆写它。

另一个关于集合约定的例子来自 NSWindow 类:

```
- (void)addChildWindow:(NSWindow *)childWin ordered:(NSWindowOrderingMode)place;
- (void)removeChildWindow:(NSWindow *)childWin;
- (NSArray *)childWindows;
- (NSWindow *)parentWindow;
- (void)setParentWindow:(NSWindow *)window;
```

方法参数

命名方法参数时要考虑如下规则:

- 如同方法名,参数名小写第一个单词的首字符,大写后继单词的首字符。如:removeObject:(id)anObject
- 不要在参数名中使用 pointer 或 ptr,让参数的类型来说明它是指针
- 避免使用 one, two,...,作为参数名
- 避免为节省几个字符而缩写

按照 Cocoa 惯例,以下关键字与参数联合使用:

```
...action:(SEL)aSelector
...alignment:(int)mode
...atIndex:(int)index
...content:(NSRect)aRect
...doubleValue:(double)aDouble
...floatValue:(float)aFloat
...font:(NSFont *)fontObj
...frame:(NSRect)frameRect
...intValue:(int)anInt
...keyEquivalent:(NSString *)charCode
...length:(int)numBytes
...point:(NSPoint)aPoint
...stringValue:(NSString *)aString
...tag:(int)anInt
...target:(id)anObject
...title:(NSString *)aString
```

私有方法

大多数情况下,私有方法命名相同与公共方法命名约定相同,但通常我们约定给私有方法添加前缀,以便与公共方法区分开来。即使这样,私有方法的名称很容易导致特别的问题。当你设计一个继承自 Cocoa framework 某个类的子类时,你无法知道你的私有方法是否不小心覆盖了框架中基类的同名方法。

Cocoa framework 的私有方法名称通常以下划线作为前缀(如:_fooData),以标示其私有属性。基于这样的事实,遵循以下两条建议:

- 不要使用下划线作为你自己的私有方法名称的前缀,Apple 保留这种用法。
- 若要继承 Cocoa framework 中一个超大的类(如:NSView),并且想要使你的私有方法名称与基类中的区别开来,你可以为你的私有方法名称添加你自己的前缀。这个前缀应该具有唯一性, 建议用"p_Method"格式，p代表private。

尽管为私有方法名称添加前缀的建议与前面类中方法命名的约定冲突,这里的意图有所不同:为了防止不小心地覆盖基类中的私有方法。

函数(Function)命名

Objective-C 允许通过函数(C 形式的函数)描述行为,就如成员方法一样。如果隐含的类为单例或在处理函数子系统时,你应当优先使用函数,而不是类方法。

函数命名应该遵循如下几条规则:

- 函数命名与方法命名相似,但有两点不同:
  1. 它们有前缀,其前缀与你使用的类和常量的前缀相同
  2. 大写前缀后紧跟的第一个单词首字符
- 大多数函数名称以动词开头,这个动词描述该函数的行为

```
NSHighlightRect
NSDeallocateObject
```

查询属性的函数有个更多的规则要遵循:

- 查询第一个参数的属性的函数,省略动词

```
unsigned int NSEventMaskFromType(NSEventType type)
float NSHeight(NSRect rect)
```

- 返回值为引用的方法,使用 Get

```
const char *NSGetSizeAndAlignment(const char *typePtr, unsigned int *sizep, unsigned i
nt *alignp)
```

- 返回 boolean 值的函数,名称使用判断动词 is, does 开头

```
BOOL NSDecimalIsNotANumber(const NSDecimal *decimal)
```

属性(Property)与数据类型命名

这一节描述属性，实例变量，常量，异常以及通知的命名约定。

属性声明与实例变量

属性声明的命名大体上和访问方法的命名是一致的。假如属性是动词或名词，格式如下

```
@property (...) typenounOrVerb;
```

例如：

```
@property (strong) NSString *title;
@property (assign) BOOL showsAlpha;
```

如果属性是形容词，名字去掉"is"前缀，但是要特别说明一下符合规范的get访问方法，例如

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

多数情况下，你声明了一个属性，那么就会自动生成对应的实例变量。

确保实例变量名简明扼要地描述了它所代表的属性。通常，你应该使用访问方法，而不是直接访问实例变量（除了在init或者dealloc方法里）。为了便于标识实例变量，在名字前面加个下划线"_"，例如:

```
@implementation MyClass {
    BOOL _showsTitle;
}
```

在为类添加实例变量是要注意:

- 避免创建 public 实例变量
- 使用 `@private` , `@protected` 显式限定实例变量的访问权限
- 如果实例变量被设计为可被访问的,确保编写了访问方法

常量

常量命名规则根据常量创建的方式不同而不同。

~~枚举常量~~ (Deprecated)

- 使用枚举来定义一组相关的整数常量
- 枚举常量与其 `typedef` 命名遵守函数命名规则。如:来自 NSMatrix.h 中的例子

```
typedef enum _NSMatrixMode {
    NSRadioModeMatrix        = 0,
    NSHighlightModeMatrix    = 1,
    NSListModeMatrix         = 2,
    NSTrackModeMatrix        = 3,
} NSMatrixMode;
```

- 位掩码常量可以使用不具名枚举。如:

```
enum {
    NSBorderlessWindowMask      = 0,
    NSTitledWindowMask          = 1 << 0,
    NSClosableWindowMask        = 1 << 1,
    NSMiniaturizableWindowMask  = 1 << 2,
    NSResizableWindowMask       = 1 << 3
};
```

const常量

- 尽量用 `const` 来修饰浮点数常数，以及彼此没有关联的整数常量（否则使用枚举）
- const常量命名范例：

```
const float NSLightGray;
```

枚举常量命名规则与函数命名规则相同。

其他常量

- 通常不使用 `#define` 来创建常量。如上面所述，整数常量请使用枚举，浮点数常量请使用 `const`
- 使用大写字母来定义预处理编译宏。如：

```
#ifdef  DEBUG
```

- 编译器定义的宏名首尾都有双下划线。 如：

```
__MACH__
```

- 为 notification 名及 dictionary key 定义字符串常量,从而能够利用编译器的拼写检查,减少书写错误。Cocoa 框架提供了很多这样的范例:

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

实际的字符串值在实现文件中赋予。(注意: `APPKIT_EXTERN` 宏等价于 Objective-C 中 `extern`)

异常与通知

异常与通知的命名遵循相似的规则,但是它们有各自推荐的使用模式。

异常

虽然你可以出于任何目的而使用异常(由 NSException 类及相关类实现),Cocoa 通常不使用异常来处理常规的,可预料的错误。在这些情形下,使用诸如 `nil` , `NULL` , `NO` 或错误代码之类的返回值。异常的典型应用类似数组越界之类的编程错误。

异常由具有如下形式的全局 NSString 对象标识:

```
[Prefix] + UniquePartOfName + Exception
```

UniquePartOfName 部分是有连续的首字符大写的单词组成。例如:

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

通知

如果一个类有委托,那它的大部分通知可能由其委托的委托方法来处理。这些通知的名称应该能够反应其响应的委托方法。比如当应用程序提交 `NSApplicationDidBecomeActiveNotification` 通知时,全局 NSApplication 对象的委托会注册从而能够接收 `applicaitonDidBecomeActive:` 消息。

通知由具有如下形式的全局 NSString 对象标识:

```
[相关联类的名称] + [Did 或 Will] + [UniquePartOfName] + Notification
```

例如:

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

其它命名规范

图片命名

图片文件命名采用`type_location_identifier_state`规则，只需要`@2x`和`@3x`图片。

缩略前缀可用如下例子

- icon
- btn
- bg
- line
- logo
- pic
- img

参考:

```
UIImage *settingIcon = [UIImage imageNamed:@"icon_common_setting"];
```

图片目录中被用于类似目的的图片应归入各自的组中。

可接受的缩略名

在设计编程接口时,通常名称不要缩写。然而下面列出的缩写要么是固定下来的,要么是过去被广泛使用的,所以你可以继续使用。关于缩写有一些额外的注意事项:

- 标准 C 库中长期使用的缩写形式是可以接受的。如:"alloc", "getc"
- 你可以在参数名中更自由地使用缩写。如:"imageRep", "col"("column"), "obj", "otherWin"

常见的缩写

缩写	含义
alloc	Allocate
alt	Alternate
app	Application
calc	Calculate
dealloc	Deallocate
func	Function
horiz	Horizontal
info	Information
init	Initialize
int	Integer
max	Maximum
min	Minimum
msg	Message
nib	Interface Builder archive
pboard	Pasteboard
rect	Rectangle
Rep	Representation
temp	Temporary
vert	Vertical

常见的略写

```
ASCII
PDF
XML
HTML
URL
RTF
HTTP
TIFF
JPG
GIF
LZW
ROM
RGB
CMYK
MIDI
FTP
```

代码注释格式

当需要的时候，注释应该被用来解释为什么特定代码做了某些事情。所使用的任何注释必须保持最新，否则就删除掉。

通常应该避免一大块注释，代码就应该尽量作为自身的文档，只需要隔几行写几句说明。这并不适用于那些用来生成文档的注释。

文件注释

采用Xcode自动生成的注释格式，修改部分参数：

```
//
//  AppDelegate.m
//  LeFit
//
//  Created by Zhang Wen on 15-3-22.
//  Copyright (c) 2015 Appscomm LLC. All rights reserved.
//
```

其中项目名称、创建人、公司/组织版权需要填写正确。

import注释

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"
```

方法注释

~~采用Javadoc的格式，可以使用Xcode插件VVDocumenter-Xcode快速添加，只需输入 `///` 即可~~

Xcode 8后请使用自带扩展插件，快捷键 `Option` + `Command` + `/`

```
/**
 <#Description#>

 @param tableView <#tableView description#>
 @param section <#section description#>
 @return <#return value description#>
 */
- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)se
ction {
    return [self.familyNames objectAtIndex:section];
}
```

代码块注释

单行的用 `//` + `空格` 开头，多行的采用 `/* */` 注释

TODO注释

TODO 很不错, 有时候注释确实是为了标记一些未完成的或完成的不尽如人意的地方, 这样一搜索就知道还有哪些活要干, 日志都省了。

格式： `//TODO: 说明`

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDict
ionary *)launchOptions
{
    //TODO: 增加初始化
    return YES;
}
```

代码结构与排版

代码结构

实现文件中的代码结构，提倡以下约定：

- 用 `#pragma mark -` 将函数或方法按功能进行分组。
- 建议将dealloc方法放到实现文件的最顶部。

  > 这样是为了时刻提醒你要记得释放相关资源。

- delgate或协议相关方法放到一般内容之后。

```
#pragma mark - Lifecycle

- (void)dealloc {}
- (instancetype)init {}
- (void)viewDidLoad {}
- (void)viewWillAppear:(BOOL)animated {}
- (void)didReceiveMemoryWarning {}

#pragma mark - Custom Accessors

- (void)setCustomProperty:(id)value {}
- (id)customProperty {}

#pragma mark - Protocol conformance
#pragma mark - UITextFieldDelegate
#pragma mark - UITableViewDataSource
#pragma mark - UITableViewDelegate

#pragma mark - NSCopying

- (id)copyWithZone:(NSZone *)zone {}

#pragma mark - NSObject

- (NSString *)description {}
```

排版格式

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;
```

间距

- 一个缩进使用 4 个空格，永远不要使用制表符（tab）缩进。请确保在 Xcode 中设置了此偏好。
- 方法的大括号和其他的大括号(`if` `else` `switch` `while` 等)始终和声明在同一行开始，在新的一行结束。

推荐：

```
if (user.isHappy) {
// Do something
}
else {
// Do something else
}
```

- 方法之间应该正好空一行，这有助于视觉清晰度和代码组织性。在方法中的功能块之间应该使用空白分开，但往往可能应该创建一个新的方法。
- `@synthesize` 和 `@dynamic` 在实现中每个都应该占一个新行。

长度

- 每行代码的长度最多不超过100个字符
- 尝试将单个函数或方法的实现代码控制在30行内

> 如果某个函数或方法的实现代码过长，可以考量下是否可以将代码拆分成几个小的拥有单一功能的方法。
>
> 30行是在13寸macbook上XCode用14号字体时，恰好可以让一个函数的代码做到整屏完全显示的行数。

- 将单个实现文件里的代码行数控制在500～600行内

> 为了简洁和便于阅读，建议将单个实现文件的代码行数控制在500～600行以内最好。
>
> 当接近或超过800行时，就应当开始考虑分割实现文件了。
>
> 最好不要出现代码超过1000行的实现文件。
>
> 我们一般倾向于认为单个文件代码行数越长，代码结构就越不好。而且，翻代码翻的手软啊。
>
> 可以使用Objective-C的Category特性将实现文件归类分割成几个相对轻量级的实现文件。

条件判断

条件判断主体部分应该始终使用大括号 `{}` 括住来防止出错，即使它可以不用大括号（例如它只需要一行）。这些错误包括添加第二行（代码）并希望它是 if 语句的一部分时。还有另外一种更危险的，当 if 语句里面的一行被注释掉，下一行就会在不经意间成为了这个 if 语句的一部分。此外，这种风格也更符合所有其他的条件判断，因此也更容易检查。

推荐：

```
if (!error) {
    return success;
}
```

反对：

```
if (!error)
    return success;
```

或

```
if (!error) return success;
```

三目运算符

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;
```

错误处理

当引用一个返回错误参数（error parameter）的方法时，应该针对返回值，而非错误变量。

推荐：

```
NSError *error;
if (![self trySomethingWithError:&error]) {
    // 处理错误
}
```

反对：

```
NSError *error;
[self trySomethingWithError:&error];
if (error) {
    // 处理错误
}
```

一些苹果的 API 在成功的情况下会写一些垃圾值给错误参数（如果非空），所以针对错误变量可能会造成虚假结果（以及接下来的崩溃）。

方法

- 在方法声明中，在 `-` `+` 符号后应该有一个空格。方法片段之间也应该有一个空格。

推荐：

```
- (void)setExampleText:(NSString *)text image:(UIImage *)image;
```

- 实现文件中，方法的左花括号不另起一行，和方法名同行，并且和方法名之间保持1个空格

  > 此条是为了和Xcode6.1模板生成的文件的代码风格保持一致。

```
//赞成的
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

//不赞成的
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

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。除了初始化方法（ `init` ， `initWithCoder:` ，等），`dealloc` 方法和自定义的 setters 和 getters 内部，应避免直接访问实例变量。更多有关在初始化方法和 dealloc 方法中使用访问器方法的信息，参见这里。

推荐：

```
@interface NYTSection: NSObject

@property (nonatomic) NSString *headline;

@end
```

反对：

```
@interface NYTSection : NSObject {
    NSString *headline;
}
```

变量限定符

当在ARC 中引入变量限定符时， 限定符（ `__strong` , `__weak` , `__unsafe_unretained` , `__autoreleasing`) 应该位于星号和变量名之间，如： `NSString * __weak text` 。

block

`block` 适合用在 target，selector 模式下创建回调方法时，因为它使代码更易读。块中的代码应该缩进 `4` 个空格。 取决于块的长度，下列都是合理的风格准则：

- 如果一行可以写完块，则没必要换行。
- 如果不得不换行，关括号应与块声明的第一个字符对齐。
- 块内的代码须按 `4` 空格缩进。
- 如果块太长，比如超过 `20` 行，建议把它定义成一个局部变量，然后再使用该变量。
- 如果块不带参数，`^{` 之间无须空格。如果带有参数，`^(` 之间无须空格，但 `) {` 之间须有一个空格。

```
// The entire block fits on one line.
[operation setCompletionBlock:^{ [self onOperationDone]; }];

[operation setCompletionBlock:^{
    [self.delegate newDataAvailable];
}];

dispatch_async(fileIOQueue_, ^{
    NSString *path = [self sessionFilePath];
    if (path) {
        ...
    }
});


[[SessionService sharedService]
    loadWindowWithCompletionBlock:^(SessionWindow *window) {
        if (window) {
            [self windowDidLoad:window];
        }
        else {
            [self errorLoadingWindow];
        }
    }
];

[[SessionService sharedService]
    loadWindowWithCompletionBlock:
        ^(SessionWindow *window) {
            if (window) {
                [self windowDidLoad:window];
            }
            else {
                [self errorLoadingWindow];
            }
        }
];


void (^largeBlock)(void) = ^{
    ...
};
[operationQueue_ addOperationWithBlock:largeBlock];
```

开发实践

本章主要描述开发过程中一些比较固定的实践技巧，写代码时可以直接套用

初始化

- 初始化方法的返回类型用 `instancetype` ，不要用 `id`

> 关于instancetype的介绍参见NSHipster.com。

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;
}
```

这将会预防有时可能产生的许多崩溃。

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 We
b" : @"Bill"};
NSNumber *shouldUseLiterals = @YES;
NSNumber *buildingZIPCode = @10018;
```

反对：

```
NSArray *names = [NSArray arrayWithObjects:@"Brian", @"Matt", @"Chris", @"Alex", @"Ste
ve", @"Paul", nil];
NSDictionary *productManagers = [NSDictionary dictionaryWithObjectsAndKeys: @"Kate", @
"iPhone", @"Kamal", @"iPad", @"Bill", @"Mobile Web", nil];
NSNumber *shouldUseLiterals = [NSNumber numberWithBool:YES];
NSNumber *buildingZIPCode = [NSNumber numberWithInteger:10018];
```

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。苹果的 `CGGeometry` 参考中说到：

> All functions described in this reference that take CGRect data structures as inputs implicitly standardize those rectangles before calculating their results. For this reason, your applications should avoid directly reading and writing the data stored in the CGRect data structure. Instead, use the functions described here to manipulate rectangles and to retrieve their characteristics.

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;
```

常量

- 定义常量时，除非明确的需要将常量当成宏使用，否则优先使用 `const` ，而非 `#define`
- 只在某一个特定文件里面使用的常量，用 `static`

```
    static CGFloat const RWImageThumbnailHeight = 50.0;
```

枚举类型

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新的基础类型 `NS_ENUM`

```
typedef NS_ENUM(NSInteger, NYTAdRequestState) {
    NYTAdRequestStateInactive,
    NYTAdRequestStateLoading
};
```

位掩码

当用到位掩码时，建议使用 `NS_OPTIONS` 类型的宏

```
typedef NS_OPTIONS(NSUInteger, NYTAdCategory) {
    NYTAdCategoryAutos      = 1 << 0,
    NYTAdCategoryJobs       = 1 << 1,
    NYTAdCategoryRealState  = 1 << 2,
    NYTAdCategoryTechnology = 1 << 3
};
```

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。有名字的类目（例如 `ASMPrivate` 或 `private` ）永远都不应该使用，除非要扩展其他类。

```
@interface NYTAdvertisement ()

@property (nonatomic, strong) GADBannerView *googleAdView;
@property (nonatomic, strong) ADBannerView *iAdView;
@property (nonatomic, strong) UIWebView *adXWebView;

@end
```

布尔值

- Objective-C的布尔值只使用 `YES` 和 `NO`

- `true` 和 `false` 只能用于CoreFoundation，C或C++的代码中
- 禁止将某个值或表达式的结果与 `YES` 进行比较

> 因为BOOL被定义成signed char。这意味着除了YES(1)和NO(0)以外，它还可能是其他值。（32位系统）
>
> 因此**C**或**C++**中的非**0**为真并不一定就是**YES**

```
//以下都是被禁止的
- (BOOL)isBold {
    return [self fontTraits] & NSFontBoldTrait;
}

- (BOOL)isValid {
    return [self stringValue];
}

if ([self isBold] == YES) {
    //...
}


//以下才是赞成的方式
- (BOOL)isBold {
    return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}

- (BOOL)isValid {
    return [self stringValue] != nil;
}

- (BOOL)isEnabled {
    return [self isValid] && [self isBold];
}

if ([self isBold]) {
    //...
}
```

- 虽然 `nil` 会被直接解释成 `NO` ，但还是建议在条件判断时保持与nil的比较，因为这样代码更直观。

```
//比如，更直观的代码
if (someObject != nil) {
    //...
}

//没那么直观的代码
if (!someObject) {
    //...
}
```

- 在C或C++代码中，要注意NULL指针的检测。

> 向一个nil的Objective-C对象发送消息不会导致崩溃。但由于Objective-C运行时不会处理给NULL指针的情况，所以为了避免崩溃，需要自行处理对于C/C++的NULL指针的检测。

- 如果某个 `BOOL` 类型的property的名字是一个形容词，建议为getter方法加上一个"is"开头的别名。

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

- 在方法实现中，如果有block参数，要注意检测block参数为nil的情况。

```
- (void)exitWithCompletion:(void(^)(void))completion {
    // 错误。 如果外部调用此方法时completion传入nil，此处会发生EXC_BAD_ACCESS
    completion();

    // 正确。如果completion不存在则不调用。
    if (completion) {
        completion();
    }
}
```

Objective-C保留字

Steffen Itterheim 在他的博客中总结了 Objective-C 2.0 所有的编译器保留字，并且对这些保留字做了介绍和使用示例。这些保留字如下：

```
@class
@defs
@protocol @required @optional @end
@interface @public @package @protected @private @property @end
@implementation @synthesize @dynamic @end
@throw @try @catch @finally
@synchronized @autoreleasepool
@selector @encode
@compatibility_alias
@”string”
```

其中比较少用到的包括 `@dynamic` `@defs` `@encode` `@compatibility_alis` ，所以主要介绍这几个关键字。

@dynamic

`@dynamic` 是相对于 `@synthesize` 的，它们用样用于修饰 `@property` ，用于生成对应的的 getter 和 setter 方法。但是 `@ dynamic` 表示这个成员变量的 getter 和 setter 方法并不是直接由编译器生成，而是手工生成或者运行时生成。

示例如下：

```
@implementation ClassName
@synthesize aProperty, bProperty;
@synthesize cProperty=instanceVariableName;
@dynamic anotherProperty;
// method implementations
@end
```

@defs

`@defs` 用于返回一个 Objective-C 类的 struct 结构，这个 struct 与原 Objective-C 类具有相同的内存布局。就象你所知的那样，Objective-C 类可以理解成由基本的 C struct 加上额外的方法构成。

示例代码如下：

```
struct { @defs( NSObject) }
```

你可能会想，什么情况下才会需要使用这个关键字。答案是涉及非常底层的操作或优化的时候才会用到。像如这篇讨论Objective-C 如何做缓存优化的文章中，就用到了该关键字。

@encode

`@encode` 是用于表示一个类型的字符串，对此，苹果有专门的介绍文档

示例如下：

```
-(void) aMethod
{
    char *enc1 = @encode(int);                 // enc1 = "i"
    char *enc2 = @encode(id);                  // enc2 = "@"
    char *enc3 = @encode(@selector(aMethod));  // enc3 = ":"
    // practical example:
    CGRect rect = CGRectMake(0, 0, 100, 100);
    NSValue *v = [NSValue value:&rect withObjCType:@encode(CGRect)];
}
```

@compatibility_alis

`@compatibility_alis` 是用于给一个类设置一个别名。这样就不用重构以前的类文件就可以用新的名字来替代原有名字。

示例如下：

```
@compatibility_alias AliasClassName ExistingClassName
```

@autoreleasepool

`@autoreleasepool` 是用于 ARC 下代替 `NSAutoreleasePool` 的保留字，在苹果的这篇官方文档中有提到， `@autoreleasepool` 比 `NSAutoreleasePool` 快 6 倍。当然，文档中也提到，ARC 下不止 Autorelease Pool 的实现变快了， `retain` 和 `release` 也快很多。如果你还没有在工程中使用 ARC，推荐看看《是否应该使用 ARC》。

Xcode工程结构

为了避免文件杂乱，物理文件应该保持和 Xcode 项目文件同步。Xcode 创建的任何组（group）都必须在文件系统有相应的映射。为了更清晰，代码不仅应该按照类型进行分组，也可以根据功能进行分组。

如果可以的话，尽可能一直打开 Target Build Settings 中 "`Treat Warnings as Errors`" 以及一些额外的警告。如果你需要忽略指定的警告,使用Clang 的编译特性。

目录结构参考

- 按照类型分组

```
projectName/
    Resources                   图片等素材
    Sources                     代码
        /ViewController         控制器
        /Service                一些第三方比如支付宝，提供给controller的封装好的接口
        /External               引入的第三方库
        /Net                    网络请求
        /Model                  模型
        /Util                   工具类
        /Custom                 自定义的文件
            /Catergory          自定义的扩展类
            /Config             项目配置文件
            /View               视图
            /Util               工具类
```

- 按照功能分组

Git基本配置

配置用户信息

Git 提供了一个叫做 `git config` 的工具，专门用来配置或读取相应的工作环境变量。而正是由这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

- `/etc/gitconfig` 文件：系统中对所有用户都普遍适用的配置。若使用 `git config` 时用 `--system` 选项，读写的就是这个文件。
- `~/.gitconfig` 文件：用户目录下的配置文件只适用于该用户。若使用 `git config` 时用 `--global` 选项，读写的就是这个文件。
- 当前项目的 Git 目录中的配置文件（也就是工作目录中的 `.git/config` 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 `.git/config` 里的配置会覆盖 `/etc/gitconfig` 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 `.gitconfig` 文件。主目录即 $HOME 变量指定的目录，一般都是 `C:\Documents and Settings\$USER`。此外，Git 还会尝试找寻 `/etc/gitconfig` 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

用户信息

第一个要配置的是你个人的用户名称和电子邮件地址。这两条配置很重要，每次 Git 提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：

```
$ git config --global user.name "输入你的名字"
$ git config --global user.email abc@example.com
```

查看配置信息

要检查已有的配置信息，可以使用 ·git config --list· 命令：

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

.gitignore

```
# General
*.DS_Store
.AppleDouble
.LSOverride

# Icon must end with two \r
Icon


# Thumbnails
._*

# Files that might appear in the root of a volume
.DocumentRevisions-V100
.fseventsd
.Spotlight-V100
.TemporaryItems
.Trashes
.VolumeIcon.icns
.com.apple.timemachine.donotpresent

# Directories potentially created on remote AFP share
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk

# Xcode
#
# gitignore contributors: remember to update Global/Xcode.gitignore, Objective-C.gitig
nore & Swift.gitignore

## Build generated
build/
DerivedData/

## Various settings
*.pbxuser
!default.pbxuser
*.mode1v3
!default.mode1v3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3
xcuserdata/

## Other
```

```
*.moved-aside
*.xccheckout
*.xcscmblueprint

## Obj-C/Swift specific
*.hmap
*.ipa
*.dSYM.zip
*.dSYM

## Playgrounds
timeline.xctimeline
playground.xcworkspace

# Swift Package Manager
#
# Add this line if you want to avoid checking in source code from Swift Package Manage
r dependencies.
# Packages/
# Package.pins
.build/

# CocoaPods
#
# We recommend against adding the Pods directory to your .gitignore. However
# you should judge for yourself, the pros and cons are mentioned at:
# https://guides.cocoapods.org/using/using-cocoapods.html#should-i-check-the-pods-dire
ctory-into-source-control
#
# Pods/

# Carthage
#
# Add this line if you want to avoid checking in source code from Carthage dependencie
s.
# Carthage/Checkouts

Carthage/Build

# fastlane
#
# It is recommended to not store the screenshots in the git repo. Instead, use fastlan
e to re-generate the
# screenshots whenever they are needed.
# For more information about the recommended setup visit:
# https://docs.fastlane.tools/best-practices/source-control/#source-control

fastlane/report.xml
fastlane/Preview.html
fastlane/screenshots
fastlane/test_output

# Code Injection
```

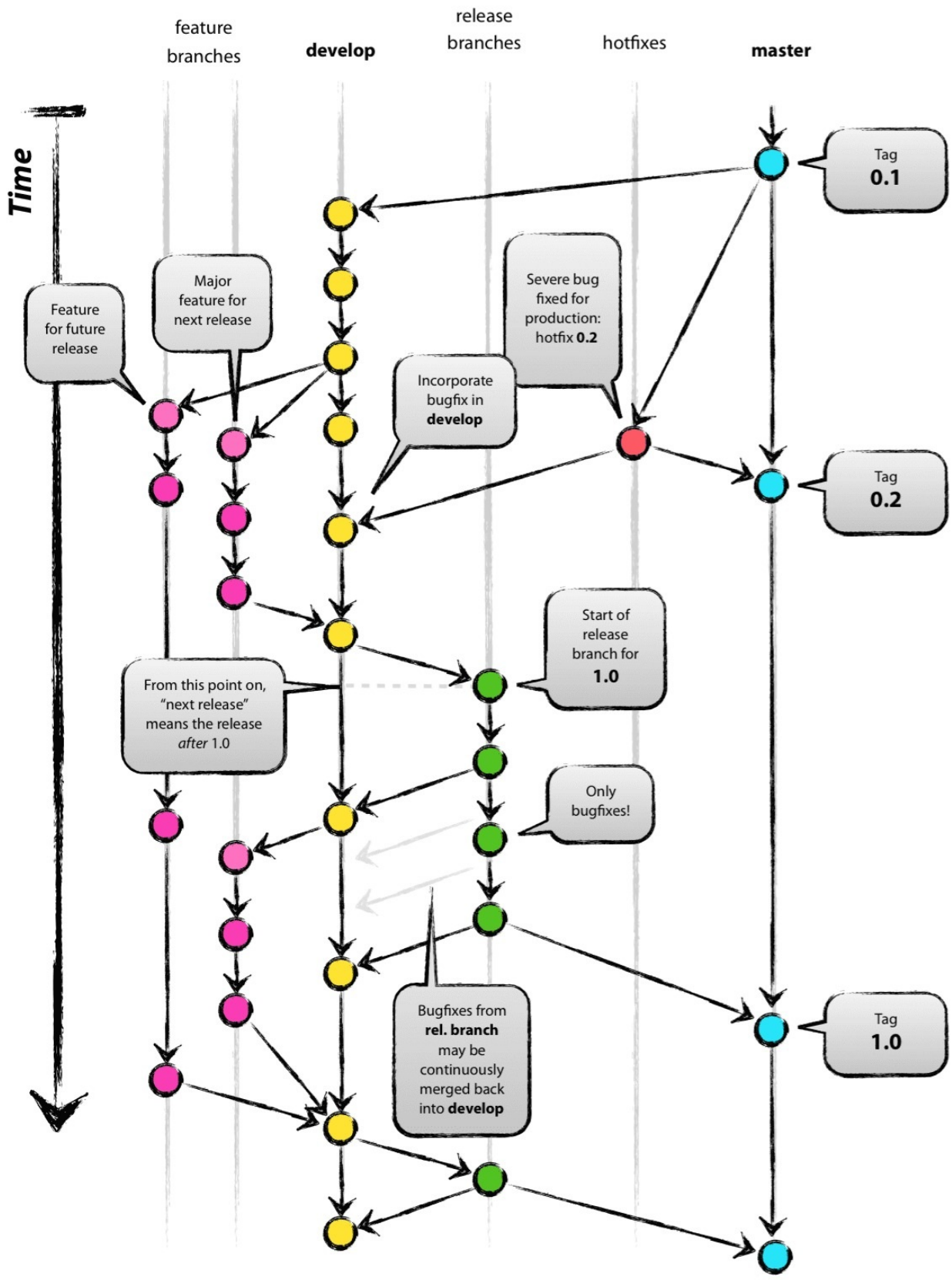
```
#
# After new code Injection tools there's a generated folder /iOSInjectionProject
# https://github.com/johnno1962/injectionforxcode

iOSInjectionProject/
```

Git客户端

- SourceTree
- GitHub Desktop

Git 分支模型

> 本篇转载自一个成功的 Git 分支模型 ，原文：A successful Git branching model



这篇文章围绕着Git做为我们所有的源代码版本控制工具而展开的。

为什么是Git

为了深入探讨git和集中式源码版本控制系统的利弊，参见这些文章。这方面有太多的激烈争论。作为一个开发者，相比其他工具，当前我更喜欢Git。Git的确改变了开发者关于合并与分支的思考方式。在那些经典的CVS/Subversion管理工具的世界中，合并/分支被认为是有些吓人的(“当心合并冲突，它们咬你!”)，而且偶尔你得做些操作解决一些问题。

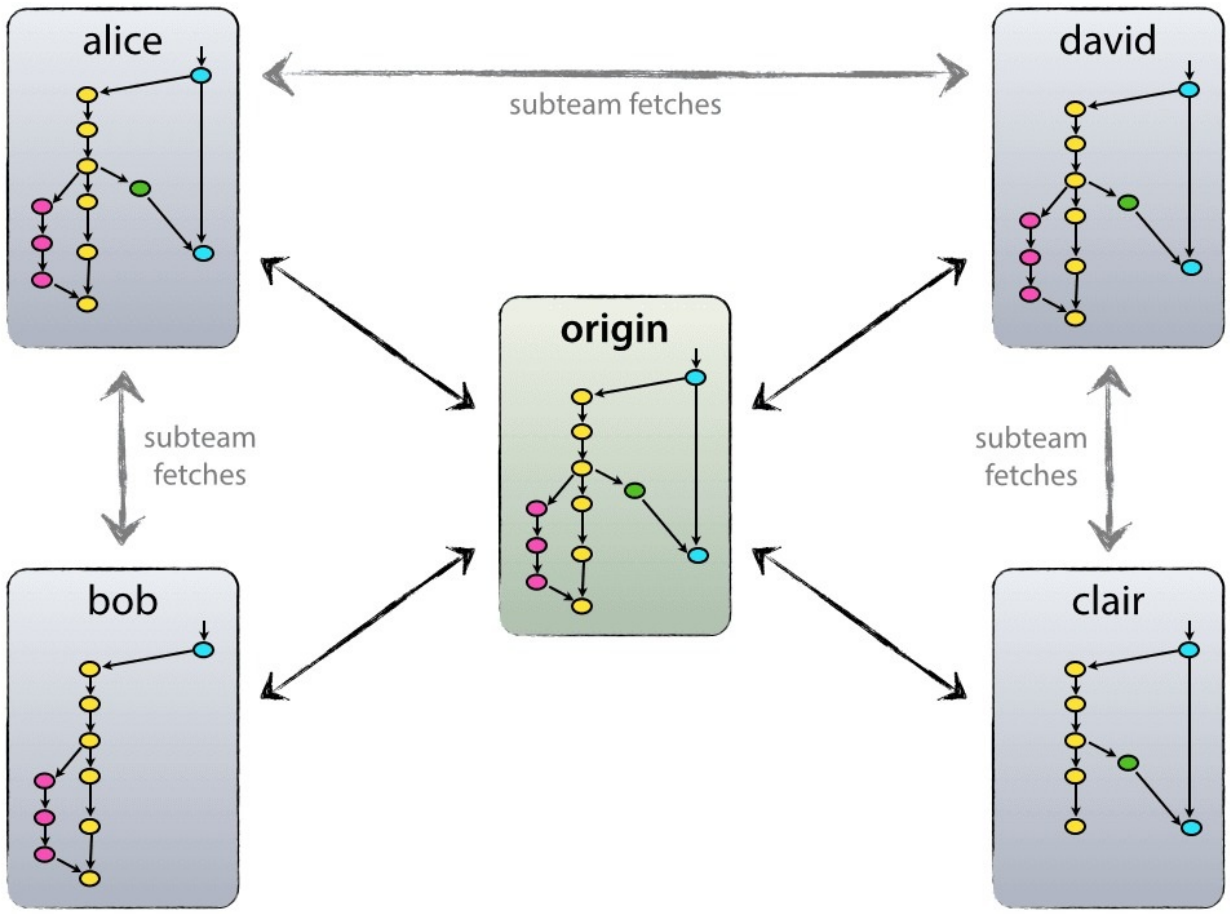
但是使用Git，这些操作都变得极度简单，这些操作被认为是你日常工作流程核心部分之一。例如，在CVS/Subversion 这本书中，分支与合并在很后的章节中才被第一次讨论(针对高级用户)。但是在每一本Git书籍中，在第三章就讲到了(基础部分)。

由于它的简单性和操作命令的重复性，分支与合并操作变得不再可怕。版本控制工具被认为在分支/合并方面提供操作便利性比什么都重要

关于工具本身，已经讨论的足够多了，下面针对开发模型进行展开。我将要介绍的这个模型不会比任何一套流程内容多，每个团队成员都必须遵守，这样便于管理软件开发过程。

既分散又集中

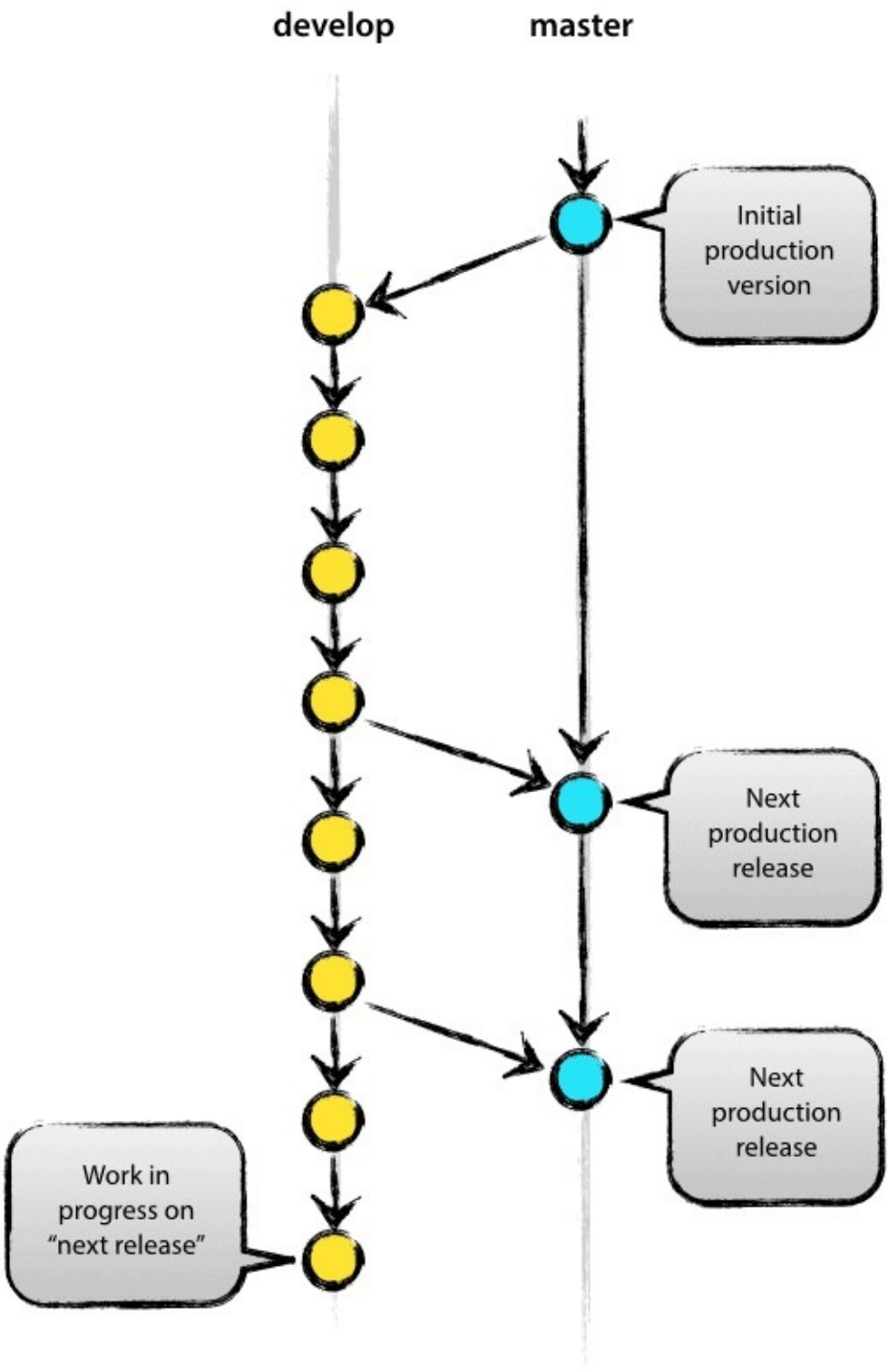
我们使用的，且与这个分支模型配合的非常好的库，他有一个“真正”的中央仓库。注意，这个库只是被认为是中央仓库(因为Git是一个分布式的版本控制工具，在技术层面没有所谓的中央仓库)。我们将会为这个仓库起名为 `origin` ，因为所有的Git用户对这个名字都比较熟悉。



每个开发者从origin拉取和推送代码。除了集中式的推送拉取关系，每个开发者也有可能从别的开发者处拉取代码，形成自己的团队。例如当与两个或者更多的人开发一个大的特性时，或者在将代码推送到 `origin` 之前，这种代码管理模式可能有用。在上图中，存在Alice和Bob，Alice和David，Clair 和David三个子团队

技术上而言，这只不过意味着Alice定义了一个远程Git仓库，起名为bob，实际上指向Bob的版本库，反之亦然(Bob定义了一个远程Git仓库，起名为alice，实际上指向Alice的版本库)。

主分支



老实说，我们讨论的开发模型受到了当前已存在模型的很大启发。集中式的版本库有两个永久存在的主分支：

- `master分支`
- `develop分支`

`origin` 的master分支每个Git用户都很熟悉。平行的另外一个分支叫做 `develop` 分支。

我们认为 `origin/master` 这个分支上 `HEAD` 引用所指向的代码都是可发布的。

我们认为 `origin/develop` 这个分支上 `HEAD` 引用所指向的代码总是反应了下一个版本所要交付特性的最新的代码变更。一些人管它叫“整合分支”。它也是自动构建系统执行构建命令的分支。

当 `develop` 分支上的代码达到了一个稳定状态，并且准备发布时，所有的代码变更都应该合并到 `master` 分支，然后打上发布版本号的tag。具体如何进行这些操作，我们将会讨论

因此，每次代码合并到 `master` 分支时，它就是一个人为定义的新的发布产品。理论上而言，在这我们应该非常严格，当 `master` 分支有新的提交时，我们应该使用Git的钩子脚本执行自动构建命令，然后将软件推送到生产环境的服务器中进行发布。

辅助性分支

紧邻 `master` 和 `develop` 分支，我们的开发模型采用了另外一种辅助性的分支，以帮助团队成员间的并行开发，特性的简单跟踪，产品的发布准备事宜，以及快速的解决线上问题。不同于主分支，这些辅助性分支往往只要有限的生命周期，因为他们最终会被删除。

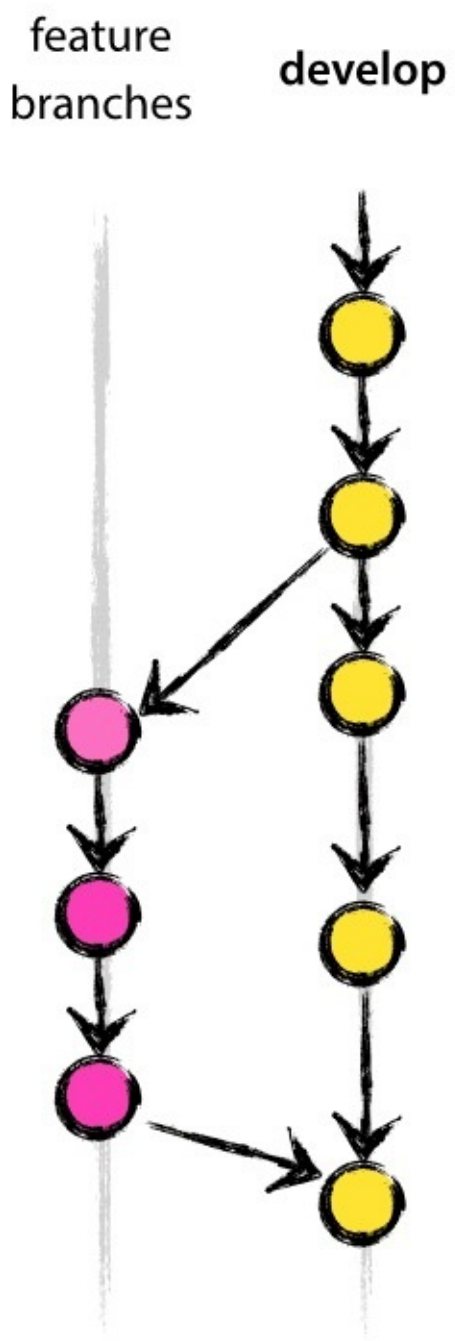
我们使用的不同类型分支包括:

- 特性分支
- Release分支
- Hotfix 分支

上述的每一个分支都有其特殊目的，也绑定了严格的规则：哪些分支是自己的拉取分支，哪些分支是自己的目标合并分支。

从技术角度看，这些分支的特殊性没有更多的含义。只是按照我们的使用方式对这些分支进行了归类。他们依旧是原Git分支的样子。

特性分支



特性分支可以从 `develop` 分支拉取建立，最终必须合并回 `develop` 分支。特性分支的命名，除了 `master` ， `develop` ， `release-*` ，或 `hotfix-*` 以外，可以随便起名。

特性分支(有时候也称主题分支)用于开发未来某个版本新的特性。当开始一个新特性的开发时，这个特性未来将发布于哪个目标版本，此刻我们是不得而知的。特性分支的本质特征就是只要特性还在开发，他就应该存在，但最终这些特性分支会被合并到develop分支(目的是在新版本中添加新的功能)或者被丢弃(它只是一个令人失望的试验)

特性分支只存在开发者本地版本库，不在远程版本库。

创建特性分支

当开始开发一个新特性时，从 `develop` 分支中创建特性分支

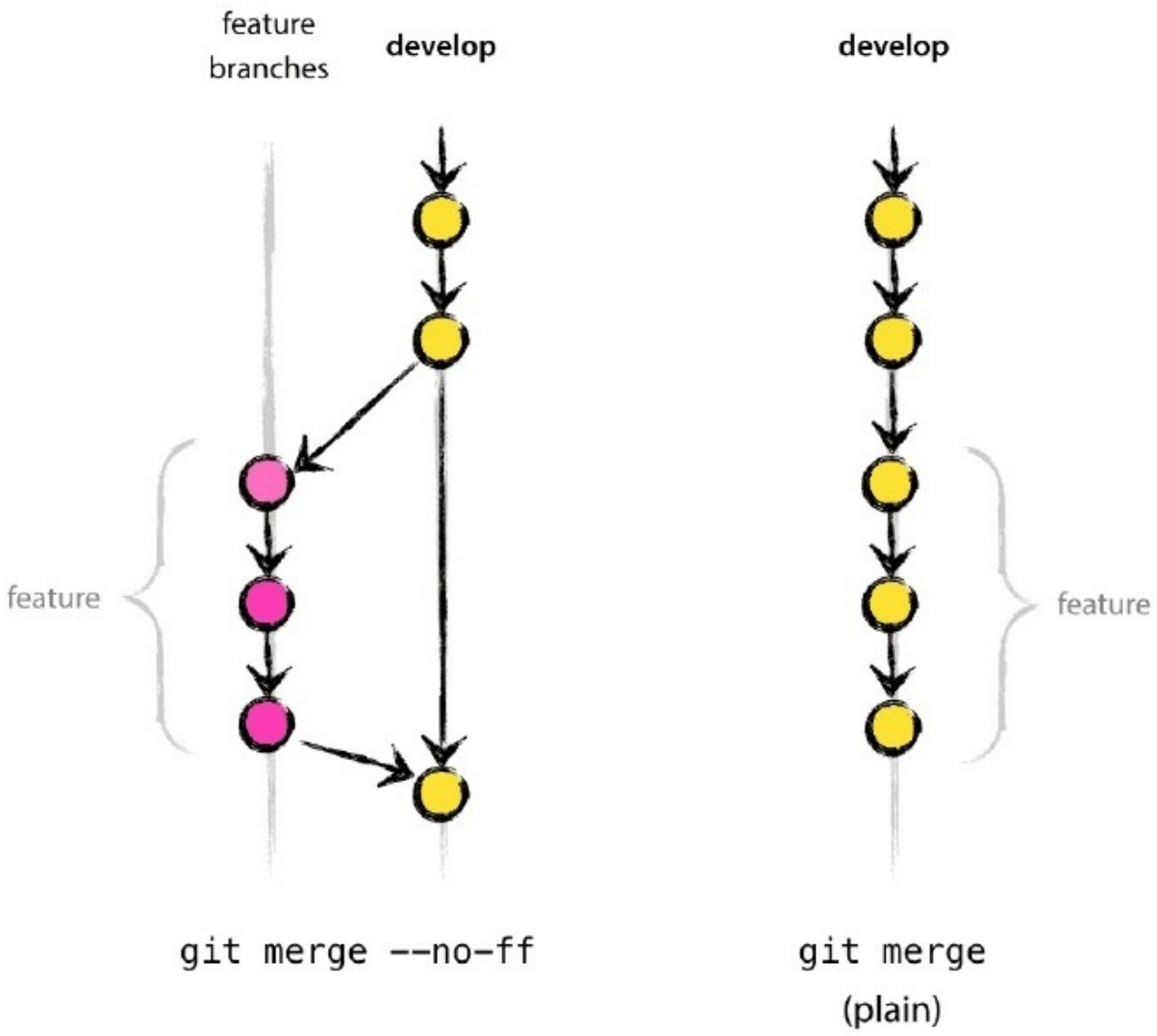
```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

在**develop**分支整合已经开发完成的特性

开发完成的特性必须合并到 `develop` 分支，即添加到即将发布的版本中。

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

`--no-ff` 参数的作用是在合并的时候，会创建一个新的提交对象，即使是fast-forward方式的合并。这就避免了丢失特性分支的历史记录信息以及提交记录信息。比较一下



在右面的例子中，是不可能从Git历史记录中看到一个已经实现了的特性的所有提交对象-除非你去查看所有的日志信息。要想获取整个特性分支信息，在右面的例子中的确是一个头疼的问题，但是如果使用 `--no-ff` 参数就没有这个问题。

使用这个参数后，的确创建了一些新的提交对象(那怕是空提交对象)，但是很值得。

不幸的是，我还没有找到一种方法使Git默认的merge操作带着 `--no-ff` 参数，但的确应该这样。

发布分支

从 `develop` 分支去建立Release分支，Release分支必须合并到 `develop` 分支和 `master` 分支，Release分支名可以这样起名: `release-*`。

Release分支用于支持一个新版本的发布。他们允许在最后时刻进行一些小修小改。甚至允许进行一些小bug的修改，为新版本的发布准要一些元数据(版本号，构建时间等)。通过在release分支完成这些工作， `develop` 分支将会合并这些特性以备下一个大版本的发布。

从 `develop` 分支拉取新的release分支的时间点是当开发工作已经达到了新版本的期望值。至少在这个时间点，下一版本准备发布的所有目标特性必须已经合并到了 `develop` 分支。更远版本的目标特性不必合并回develop分支。这些特性必须等到个性分支创建后，才能合并回develop分支

在release分支创建好后，就会获取到一个分配好即将发布的版本号，不能更早，就在这个时间点。在此之前， `develop` 分支代码反应出了下一版本的代码变更，但是到底下一版本是 0.3 还是 1.0，不是很明确，直到release分支被建立后一切都确定了。这些决定在release分支开始建立，项目版本号等项目规则出来后就会做出。

创建**release**分支

从 `develop` 分支创建release分支。例如1.1.5版本是当前产品的发布版本，我们即将发布一个更大的版本。 `develop` 分支此时已经为下一版本准备好了，我们决定下一版的版本号是 1.2(1.1.6或者2.0也可以)。所以我们创建release分支，并给分支赋予新的版本号:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

创建好分支并切到这个分支后，我们给分支打上版本号。 `bump-version.sh` 是一个虚构的shell脚本，它更改了工作空间的某些文件来反映新版本特征。(当然也可以手动改变这些文件)，然后版本就被提交了。

新的分支会存在一段时间，直到新版本最终发布。在这段时间里，bug的解决可以在这个分支进行(不要在 `develop` 分支进行)。此时是严禁添加新的大特性。这些修改必须合并回 `develop` 分支，之后就等待新版本的发布。

结束一个**release**分支

当release分支的准备成为一个眞正的发布版本时，一些操作必须需要执行。首先，将release分支合并回 `master` 分支(因为 `master` 分支的每一次提交都是预先定义好的一个新版本，谨记)。然后为这次提交打tag，为将来去查看历史版本。最后在release分支做的更改也合并到 `develop` 分支，这样的话，将来的其他版本也会包含这些已经解决了的bug。

在Git中需要两步完成:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

这样release分支已经完成工作，tag也已经打了。

备注:你可以使用 `-s` or `-u <key>` 参数为你的tag设置标签签名。

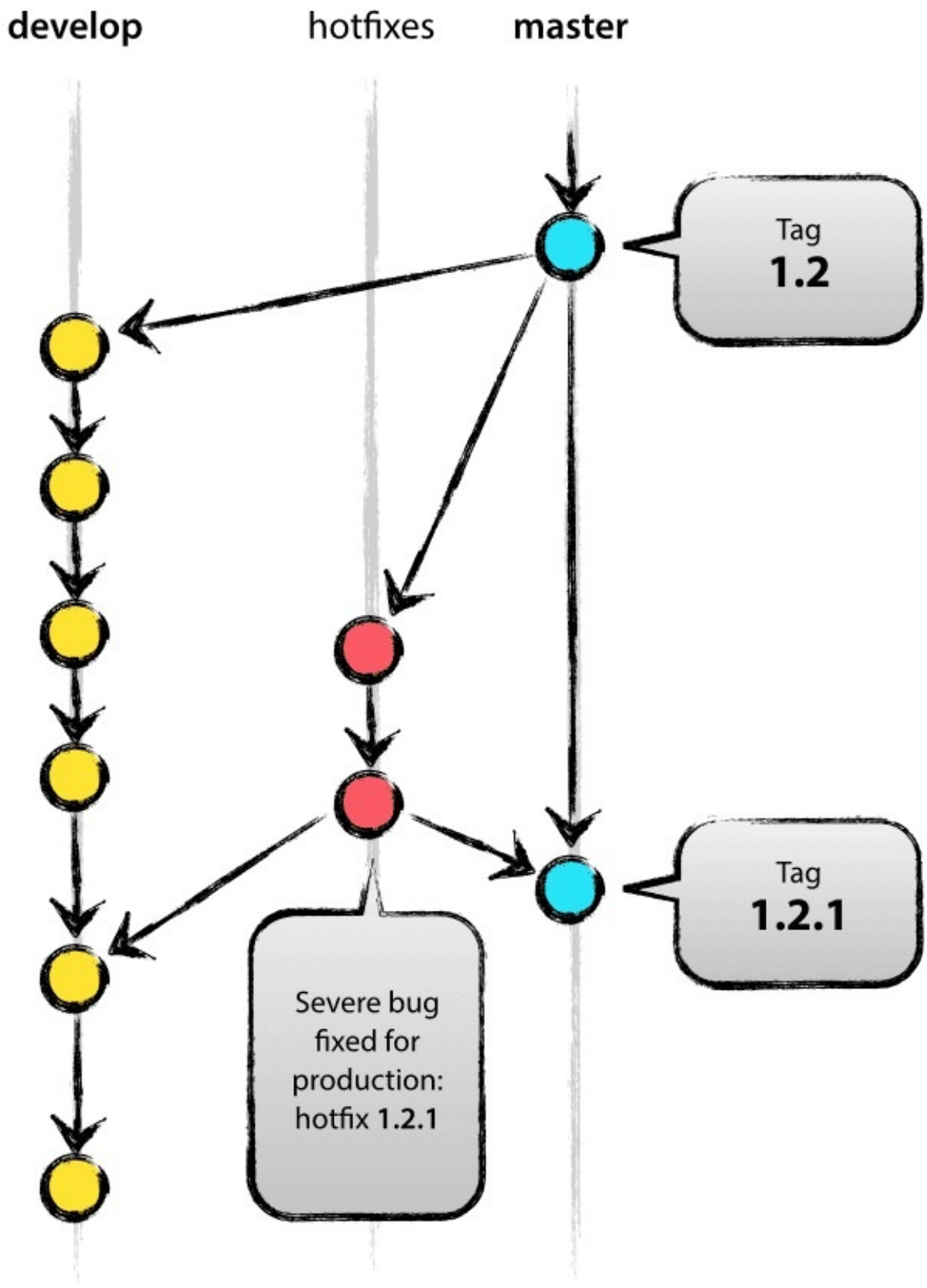
为了保存这些在release分支所做的变更，我们需要将这些变更合并回 `develop` 分支。执行如下Git命令:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

这步有可能会有合并冲突(极有可能，因为我们已经改变了版本号)。如果有冲突，解决掉他，然后提交。
现在我们已经完成了工作，release分支可以删除了，因为我们不在需要他:

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Hotfix分支



Hotfix分支从 `master` 分支建立，必须合并回 `develop` 分支和 `master` 分支，为Hotfix分支可以这样起名: `hotfix-*`

Hotfix分支在某种程度上非常像release分支，他们都意味着为某个新版本发布做准备，并且都是预先不可知的。Hotfix分支是基于当前生产环境的产品的一个bug急需解决而必须创建的。当某个版本的产品有一个严重bug需要立即解决，Hotfix分支需要从 `master` 分支上该版本对应

的tag上进行建立，因为这个tag标记了产品版本。

创建hotfix分支

Hotfix分支从 `master` 分支进行创建。例如当前线上1.2版本产品因为server端的一个Bug导致系统有问题。但是在 `develop` 分支进行更改是不靠谱的，所以我们需要建立hotfix分支，然后开始解决问题:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

千万别忘记在创建分支后修改版本号。

然后解决掉bug，提交一次或多次。

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

结束hotfix 分支

完成工作后，解决掉的bug代码需要合并回 `master` 分支，但同时也需要合并到 `develop` 分支，目的是保证在下一版中该bug已经被解决。这多么像release分支啊。

首先，对 `master` 分支进行合并更新，然后打tag

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

备注:你可以使用 `-s` or `-u <key>` 参数为你的tag设置标签签名。

紧接着，在 `develop` 分支合并bugfix代码

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

这里可能会有一些异常情况，当一个**release**分支存在时，**hotfix** 分支需要合并到**release** 分支，而不是 `develop` 分支。当release分支的使命完成后，合并回release分支的bugfix代码最终也会被合并到 `develop` 分支。(当 `develop` 分支急需解决这些bug，而等不到release分支的结束，你可以安全的将这些bugfix代码合并到develop分支，这样做也是可以的)。

最后删除这些临时分支

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

总结

这个分支模型其实没有什么震撼人心的新东西，这篇文章开始的那个“最大图片”已经证明了他在我们工程项目中的巨大作用。它会形成一种优雅的理想模型，而且很容易理解，该模型也允许团队成员形成一个关于分支和版本发布过程的相同理念。

这里有提供一个高质量的分支模型图的PDF版本。去吧，把它挂在墙上随时快速参考。

Git-branching-model.pdf

更新：任何需要他的人,这里有一个主图的gitflow-model.src.key文件

Xcode扩展插件

Xcode Editor Extension

> New Xcode extensions for the source editor let you customize your coding experience. Use extensions to navigate within your editor's text, and select, modify, and transform your code. Bind your favorite extensions to a keyboard shortcut to make common reformatting tasks a snap. Xcode includes a new template so you can easily create editor extensions and distribute them on the Mac App Store, or sign your extensions with your Developer ID to share them online. And because extensions run in a separate process, Xcode stays safe and stable.

Xcode 8 开始支持一套官方的扩展 API，第一个支持的扩展类型就是源码编辑器的扩展。另一方面， Xcode 采用了系统完整性保护功能（即 SIP） 。这意味着想要将代码注入到 Xcode 进程已经不可能了，现在Alcatraz也只能在无正式签名的Xcode中生效了。因此在Xcode 8后应该使用**Editor Extension**，下面推荐一些第三方Xcode扩展。

The list

- JSON-to-Swift-Converter - an open-source Xcode Source Editor extension for creating Swift code from JSON-formatted text.
- JSON2Swift - a flexible, fast, customizable tool which allows you to generate swift code based on JSON file.
- Swiftify Xcode extension - allows you to convert Objective-C code to Swift right from Xcode.
- EasyCode-Xcode - "legal" Xcode Plugin for inserting code in super duper fast lazy way.
- CodeGenerator - Xcode editor extension for swift code generation.
- GithubIssuesExtension - Xcode editor extension that adds an ability to post and edit github issues through comment templates
- XAlign - An amazing Xcode Source Editor extension to align regular code.
- Alignment -This Xcode source editor extension align your assignment statement.
- CleanClosureXcode - An Xcode Source Editor extension to clean the closure syntax.
- xTextHandler - Xcode Source Editor Extension based tools to improve the text editing experience of Xcode 8 and provide extensions with simple code.
- Xcode-Search - A Xcode Source Editor Extension that searches external sources
- SwiftLintForXcode - SwiftLint for Xcode is a Xcode Extension that was created to run SwiftLint.

- XcodeCComment - Xcode Source Editor Extension for C Style Comment /**/
- Duplicate Line - Xcode's source editor extension to duplicate selected line or lines.
- Dotify - Convert [[AnObject method1] method2] to AnObject.method1.method2 with Xcode Extension
- XcodeEditorPlus - Bring some convenient editor shortcuts to Xcode using Xcode Source Editor Extension, inspired from AppCode.
- XcodeWay - Navigate to many places from Xcode
- XcodeColorSense2 - An Xcode source editor extension that makes working with color easier
- strimmer - Strimmer is an Xcode 8 Source Code Extension that quickly strips all trailing whitespace from the current file.
- Jumpy - Jumpy is an Xcode source editor extension for jumping across multiple lines of code.
- SETools - Xcode 8 Extension - Figlet Font Titles
- GenerateSwiftInit - Proof of concept for Xcode 8 source extensions; generate a Swift init from current selection
- XTExtension - Comment lines.
- VS-Key-Bindings-For-Xcode - Visual Studio Comment Selection and Uncomment Selection for Xcode 8 extension.
- Localizer - It then breaks apart any String declarations it founds by splitting the selection based off " and then wrap the resulting String inside NSLocalizedString.
- EmojifySourceEditorExtension - A Proof-Of-Concept implementation of the new Xcode Source Editor Extension
- EmojifyApplication - Same as above.
- xcsort - Sort text or code lines from a text selection. An Xcode extension by @battlmonstr.
- SwiftLintXcodePlugin - Xcode source extension to help with some SwiftLint rules
- BDDShortcuts - BDD Shortcuts is an Xcode 8+ plugin that adds handy shortcuts for Swift and BDD frameworks such as Cedar, Quick, Kiwi or Specta.
- Jump - XCode source editor extension for quick navigating
- XcodeTopComment - Xcode Source Editor Extension to remove or modify the top comment of a file
- PAXcodePlugin - Example of Xcode Source Editor Extension
- XcodeExtension-TotsuzenNoShi - A small "sudden" example of Xcode source code extension
- ClangFormatter - Xcode Source Editor Extension for clang-format
- CwlWhitespace - The first command uses multiple selections to select every text range in your file that it believes is violating a whitespace rule. If a line contains a zero-length problem (missing whitespace or missing indent) then the whole line will be selected.
- DemoXcodeExtension - Select the current scope / block.

- TestXcodeSourceExtensions - Replace entire files with literals
- Swimat - An Xcode formatter plug-in to format your swift code
- FastCommentLine - Moves cursor to next line after commenting/uncommenting current line (like AppCode).
- SwiftInitializerGenerator - This Xcode 8 Source Code Extension will generate a Swift initializer based on the lines you've selected.
- Import☝ - Add imports from anywhere in the code.
- XcodeEquatableGenerator - Xcode 8 Source Code Extension will generate conformance to Swift Equatable protocol based on type and fields selection.
- Mark - Generates MARK comments from protocol conformance in class declaration.
- Propertizer - Autocompletes @property statements with desired type.
- Eric's Mark - Identifies IBOutlets, IBActions, Super Classes, Properties (and more) and MARK them.
- Quick Add - A Xcode Source Editor Extension to quickly add a method implementation with comment from selected text
- CleanHeaders-Xcode - A Xcode Source Editor Extension to sort your header imports and remove duplicates, similar to iSort.
- FBXCodeExtension - An Xcode Source Editor Extension providing convenient line manipulation such like **line deletion** and **duplication**.
- Basics - Generates **NSCopying**, **isEquals**, **hash**, and more.
- HandyXcode - A few 'Handy' Xcode commands (insert code placeholder, multi-line comment, etc.)
- LocalizedString - Xcode Source Editor Extension that helps to localize Swift and Objective-C source files
- Xgist - Xcode Source Editor Extension that sends code to GitHub's Gist
- PlayAlways - Create Xcode playgrounds from your menu bar.
- CodeCows - Add hundreds of ASCII cows to your source code
- DeclareType - Generate the type declaration in your file based on the file name
- Switch Enum Case Generator - Instant switch with selected enum cases
- XShared - Xcode extension which allows you copying the code with special formatting quotes for social (Slack, Telegram)
- TabifyIndents - This application adds two feature that Tabify and Untabify to Xcode Source Editor.
- Literals - Converts UIColor, NSColor, UIImage to literals
- Pragmatic - An Xcode 8._x _Source Editor extension for simplifying common pragma driven tasks
- Xcode-NSCoding - Automatically creates NSCoding-implementations from properties
- LineEscapeEx - Duplicate lines as comment
- LanguageTranslator - A Xcode Source Editor extension to translate selected text into other language.

- NamingTranslator - A Xcode Source Editor extension to translate selected variable or method between PascalCase, camelCase and snake_case.
- CodeGenerator - A Xcode Source Editor extension to generate lazy getter methods from property.
- MGTextPlus - A Xcode Source Editor extension to duplicate line, delete line, join lines and more.
- Rubicon - Swift parser + Spy generator.
- BExtension - Xcode Source extension for delivering enums and variables from enum cases, see this article

其它**Extension**

> # Alcatraz 插件管理器 (Deprecated)
>
> ## 安装
>
> 在终端输入下面命令
>
> ```
> curl -fsSL https://raw.github.com/alcatraz/Alcatraz/master/Scripts/install.sh | s
> h
> ```
>
> 成功之后会给你一杯啤酒
>
> > Alcatraz successfully installed!!1! Please restart your Xcode.
>
> 然后重启一下Xcode。就可以看到在Menu — Windows 看到 `Package Manager` ,打开它，然后就长这个样子
>
> 里面有三种东西：Plugins(插件)，Color Themes(颜色主题)，Templates(模板)
>
> 选择你想要的东西，点击==INSTALL==按钮开始安装，完成之后会显示红色的==REMOVE==按钮。
>
> ## 卸载
>
> 打开终端粘贴下面的命令：
>
> ```
> rm -rf ~/Library/Application\ Support/Developer/Shared/Xcode/Plug-ins/Alcatraz.xc
> plugin
> ```
>
> 删除掉Alcatraz安装的所有插件：

```
rm -rf ~/Library/Application\ Support/Alcatraz/
```

常用插件

Alcatraz里面常用插件说明

插件	用途
XToDo	注释辅助插件，主要用于收集并列出项目中的TODO,FIXME,`???
CocoaPods plugin	为CocoaPods添加了一个菜单项
Peckham	添加引用文件有时候非常麻烦，如果你需要引入一个pod头文件，Xcode自带的自动补全自然帮不了你，这时候你可以用Peckham插件解决这个问题。Command+Control+P解决所有的引入
FuzzyAutocomplete	代替Xcode的autocomplete，它利用模式匹配算法来解决问题。
ACCodeSnippetRepository	使用它和你的Git库同步，如果你想手动导入一个Snippet需要很麻烦的步骤，通过这个插件你只需要点击几下鼠标。
XcodeColors	改变调试控制台颜色，这个插件配合CocoaLumberjack使用效果非常好
VVDocumenter	输入三个斜线“VVV”，自动生成规范化的注释
SCXcodeMiniMap	可以在当前的窗口内创建一个代码迷你地图，并在屏幕上高亮提示
XAlign	代码对齐，Shift + Cmd + X
HOStringSense	经常输入大段文本的时候，如果文本里面有各种换行和特殊字符，经常会让人很头疼，有了HOStringSense，再也不不用为这个问题犯愁了，顺便附送字数统计功能。
OMColorSense	代码里的那些冷冰冰的颜色数值，到底时什么颜色？如果你经常遇到这个问题，每每不得不运行下模拟器去看看，那么这个插件绝对不容错过。更彪悍的是你甚至可以点击显示的颜色面板，直接通过系统的ColorPicker来自动生成对应颜色代码，再也不用做各种颜色代码转换了！
ClangFormat	CLang代码格式化
CodePilot	代码、图片、文件搜索利器，快捷键CMD + SHIFT +X

> ## Xcode更新后插件不能用的解决办法
>
> 运行下面的命令
>
> ```
> find ~/Library/Application\ Support/Developer/Shared/Xcode/Plug-ins -name Info.pl
> ist -maxdepth 3 | xargs -I{} defaults write {} DVTPlugInCompatibilityUUIDs -array
> -add `defaults read /Applications/Xcode.app/Contents/Info.plist DVTPlugInCompatib
> ilityUUID`
> ```

第三方开源库

CocoaPods

第三方开源库可以使用CocoaPods做iOS工程的依赖管理。每种语言发展到一个阶段，就会出现相应的依赖管理工具，例如Java语言的Maven，nodejs的npm。随着iOS开发者的增多，业界也出现了为iOS程序提供依赖管理的工具，它的名字叫做：**CocoaPods**。 CocoaPods项目的源码在Github上管理。该项目开始于2011年8月12日，经过多年发展，现在已经成为iOS开发事实上的依赖管理标准工具。开发iOS项目不可避免地要使用第三方开源库，CocoaPods的出现使得我们可以节省设置和更新第三方开源库的时间。

常用命令

```
1.先升级Gem
    sudo gem update --system
2.切换cocoapods的数据源
    【先删除，再添加，查看】
    gem sources --remove https://rubygems.org/
    gem sources -a http://ruby.taobao.org/
    gem sources -l
3.安装cocoapods
    sudo gem install cocoapods
4.将Podspec文件托管地址从github切换到国内的oschina
    【先删除，再添加，再更新】
    pod repo remove master
    pod repo add master http://git.oschina.net/akuandev/Specs.git
    pod repo add master https://gitcafe.com/akuandev/Specs.git
    pod repo update
5.设置pod仓库
    pod setup
6.测试
    【如果有版本号，则说明已经安装成功】
    pod --version
7.利用cocoapods来安装第三方框架
    01 进入要安装框架的项目的.xcodeproj同级文件夹
    02 在该文件夹中新建一个文件Podfile
    03 在文件中告诉cocoapods需要安装的框架信息
        a.该框架支持的平台
        b.适用的iOS版本
        c.框架的名称
        d.框架的版本
8.安装
pod install --no-repo-update
pod update --no-repo-update
```

CocoaPods安装

常用指令区别

- **pod install**

会根据Podfile.lock文件中列举的版本号来安装第三方框架

如果一开始Podfile.lock文件不存在, 就会按照Podfile文件列举的版本号来安装第三方框架

- **pod update**

将所有第三方框架更新到最新版本, 并且创建一个新的Podfile.lock文件

安装框架之前, 默认会执行pod repo update指令

- **pod install --no-repo-update**

- **pod update --no-repo-update**

安装框架之前, 不会执行pod repo update指令

Podfile.lock 文件

最后一次更新Pods时, 所有第三方框架的版本号及引用关系。

Podfile 文件

标准格式

```
platform :ios, "9.0"

target 'BSBDJ' do
    pod "AFNetworking"
    pod "SDWebImage"
    pod "MJExtension"
end
```

pod install 提速

每次执行~~pod install~~和pod update的时候，cocoapods都会默认更新一次spec仓库。这是一个比较耗时的操作。在确认spec版本库不需要更新时，给这两个命令加一个参数跳过spec版本库更新,可以明显提高这两个命令的执行速度。

```
pod install --verbose --no-repo-update
pod update --verbose --no-repo-update
```

第三方库版本号的格式

Besides no version, or a specific one, it is also possible to use operators:

- `= 0.1` Version 0.1.
- `> 0.1` Any version higher than 0.1.
- `>= 0.1` Version 0.1 and any higher version.
- `< 0.1` Any version lower than 0.1.
- `<= 0.1` Version 0.1 and any lower version.
- `~> 0.1.2` Version 0.1.2 and the versions up to 0.2, not including 0.2. This operator works based on *the last component* that you specify in your version requirement. The example is equal to `>= 0.1.2` combined with `< 0.2.0` and will always match the latest known version matching your requirements.

A list of version requirements can be specified for even more fine grained control.

```
pod ‘AFNetworking’ //不显式指定依赖库版本，表示每次都获取最新版本
pod ‘AFNetworking’, ‘2.0’ //只使用2.0版本
pod ‘AFNetworking’, ‘>2.0' //使用高于2.0的版本
pod ‘AFNetworking’, ‘>=2.0' //使用大于或等于2.0的版本
pod ‘AFNetworking’, ‘<2.0' //使用小于2.0的版本
pod ‘AFNetworking’, ‘<=2.0' //使用小于或等于2.0的版本
pod ‘AFNetworking’, ‘~>0.1.2' //使用大于等于0.1.2但小于0.2的版本，相当于>=0.1.2并且<0.2.0
pod ‘AFNetworking’, ‘~>0.1' //使用大于等于0.1但小于1.0的版本
pod ‘AFNetworking’, ‘~>0' //高于0的版本，写这个限制和什么都不写是一个效果，都表示使用最新版本
```

常用第三方库汇总

```
target 'Target Name' do
#facebook pop开源动画库
pod 'pop', '~> 1.0'

#SDWebImage
platform:ios, '6.1'
pod 'SDWebImage', '3.7'

#空视图
pod 'DZNEmptyDataSet', '1.7.2'

#MJRefresh
pod 'MJRefresh', '3.1.0'

#MJExtension
#pod 'MJExtension', '2.5.12'

#YYText 一个支持富文本的pods
pod 'YYText', '0.9.9'

#FMDB 数据库
pod 'FMDB', '2.6'

#AFNetworking
pod 'AFNetworking', '3.1.0'

#Xib动态桥接
pod 'XXNibBridge', '2.3.0'

#二维码扫描
pod 'ZBarSDK', '1.3.1'

#FPS监控
pod 'JPFPSStatus', '0.1'

#MagicalRecord 优化CoreData
pod 'MagicalRecord', '2.3.2'

end
```

参考

- Apple Coding Guidelines for Cocoa
- Github objective-c-style-guide
- Raywenderlich objective-c-style-guide 【中文】
- Google Objective-C Style Guide 【中文】
- NYTimes Objective-C Style Guide 【中文】
- Swift API Design Guidelines
- LinkedIn's Official Swift Style Guide 【中文】
- Style guide & coding conventions for Swift projects 【中文】
- The Official raywenderlich.com Swift Style Guide 【中文】

iOS开发优化

省电

- 如果使用定位，需要在定位完毕之后关闭定位，或者降低定位的频率，不停的定位会消耗电量。
- 如果使用蓝牙，需要使用的时候再开启蓝牙，用完之后关闭蓝牙，蓝牙也很耗电。
- 优化算法，减少循环次数，大量循环会让CPU一直处于忙碌状态，特别费电。
- 尽量不要使用网络轮询（心跳包、定时器），推荐使用推送。
- timer的时间间隔不宜太短，满足需求即可。
- 不要频繁刷新页面，能刷新1行 `cell` 的最好刷新一行，尽量不要 `reloadData` 。
- 线程适量，不宜过多。

性能优化

- 避免使用庞大的XIB、Storyboard，尽量使用纯代码开发界面，尤其对于App启动界面、首页。
- 使用懒加载的方式延迟加载界面。
- 避免反复处理数据。
- 避免使用 `NSDateFormatter` 和 `NSCalendar` 。
- 图片缓存的取舍 `UIImage` 加载图片方式一般有两种:A. `imagedNamed` 初始化、B. `imageWithContentsOfFile` 初始化二者不同之处在于, `imageNamed` 默认加载图片成功后会内存中缓存图片,这个方法用一个指定的名字在系统缓存中查找并返回一个图片对象.如果缓存中没有找到相应的图片对象,则从指定地方加载图片然后缓存对象，并返回这个图片对象.而 `imageWithContentsOfFile` 则仅只加载图片,不缓存.大量使用 `imageNamed` 方式会在不需要缓存的地方额外增加开销CPU的时间来做这件事.当应用程序需要加载一张比较大的图片并且使用一次性，那么其实是没有必要去缓存这个图片的，
  用 `imageWithContentsOfFile` 是最为经济的方式,这样不会因为 `UIImage` 元素较多情况下，CPU会被逐个分散在不必要缓存上浪费过多时间.使用场景需要编程时，应该根据实际应用场景加以区分， `UIImage` 虽小，但使用元素较多问题会有所凸显。

Swift编码规范

正确性 (Correctness)

把警告当做错误处理。这条规则从根本禁止了一些文法使用，如推荐使用**#selector**文而不是用字符串(更多请阅读Swift 3为什么推荐使用#selector)。

命名 (Naming)

使用驼峰式的描述性命名方式，为类，方法，变量等命名。类名的首字母应该大写，而方法和变量的首字母使用小写字符。

推荐做法：

```
private let maximumWidgetCount = 100
class WidgetContainer {
  var widgetButton: UIButton
  let widgetHeightPercentage = 0.85
}
```

不推荐做法：

```
let MAX_WIDGET_COUNT = 100
class app_widgetContainer {
  var wBut: UIButton
  let wHeightPct = 0.85
}
```

缩写和简写应该要尽量避免，遵守苹果命名规范，缩写和简写中的所有字符大小写要一致。

推荐：

```
let urlString:  URLString
let userID:  UserID
```

不推荐：

```
let uRLString:  UrlString
let userId:  UserId
```

对于函数和初始化方法，推荐对所有的参数进行有意义的命名，除非上下文已经非常清楚。如果外部参数命名可以使得函数调用更加可读，也应该把外部参数命名包含在内。

```
func dateFromString(dateString: String) -> NSDate
func convertPointAt(#column: Int, #row: Int) -> CGPoint
func timedAction(#delay: NSTimeInterval, perform action: SKAction) -> SKAction!
// 调用方式如下：
dateFromString("2014-03-14")
convertPointAt(column: 42, row: 13)
timedAction(delay: 1.0, perform: someOtherAction)
```

对于方法来说，参照标准的苹果惯例，方法命名含义要引用到第一个参数：

```
class Guideline {
  func combineWithString(incoming: String, options: Dictionary?) { ... }
  func upvoteBy(amount: Int) { ... }
}
```

协议 (Protocol)

根据苹果接口设计指导准则，协议名称用来描述一些东西是什么的时候是名词，例如：Collection,WidgetFactory。若协议名称用来描述能力应该以-ing, -able, 或 -ible结尾，例如：Equatable,Resizing。

枚举 (Enumerations)

使用首字母大写的驼峰命名规则来命名枚举值：

```
enum Shape {
  case Rectangle
  case Square
  case Triangle
  case Circle
}
```

文章 (Prose)

当我们在文章中（教程，图书，注释等）需要引用到函数时，需要从调用者的视角考虑，包含必要的参数命名，或者使用 `_` 表示不需要命名的参数。

> 从你自身实现的 `init` 中调用 `convertPointAt(column:row:)` 。
>
> 如果你调用 `dateFromString(_:)` ,需要保证你提供的输入字符串格式是”yyyy-MM-dd”。
>
> 如果你需要在 `viewDidLoad()` 中调用 `timedAction(delay:perform:)` ，记得提供调整后的延迟值和需要处理的动作。
>
> 你不能直接调用数据源方法 `tableView(_:cellForRowAtIndexPath:)`

当你遇到疑问时，可以看看Xcode在 `jump bar` 中是如何列出方法名的 —— 我们的风格与此匹配。

Methods in Xcode jump bar

类的前缀 (Class Prefixes)

Swift类型自动被模块名设置了名称空间，所以你不需要加一个类的前缀。如果两个来自不同模块的命名冲突了，你可以附加一个模块名到类型命名的前面来消除冲突。

```
import SomeModule
let myClass = MyModule.UsefulClass()
```

委托 (Delegate)

在定义委托方法时，第一个未命名参数应是委托数据源。(为了保持参数声明的一致性在Swift3引入的)

推荐：

```
func namePickerView(_ namePickerView: NamePickerView, didSelectName name: String)
func namePickerViewShouldReload(_ namePickerView: NamePickerView) -> Bool
```

不推荐：

```
func didSelectName(namePicker: NamePickerViewController, name: String)
func namePickerShouldReload() -> Bool
```

泛型 (Generics)

泛型类参数应具有描述性，遵守“大骆驼命名法”。如果一个参数名没有具体的含义，可以使用传统单大写字符，如T, U, 或V等。

推荐：

```
struct Stack<Element> { ... }
func write<Target: OutputStream>(to target: inout Target)
func swap<T>(_ a: inout T, _ b: inout T)
```

不推荐：

```
struct Stack<T> { ... }
func write<target: OutputStream>(to target: inout target)
func swap<Thing>(_ a: inout Thing, _ b: inout Thing)
```

语言 (Language)

使用美式英语拼音符合Apple API的标准。

推荐做法：

```
let color = "red"
```

不推荐做法：

```
let colour = "red"
```

代码组织结构 (Code Organization)

使用 `extension` 来组织你的功能逻辑块中的代码结构。每个extension应该使用注释 `// MARK: -` 分割以保持代码的良好组织。

协议遵守（Protocol Conformance）

当我们对一个类添加协议时，推荐使用一个单独的类扩展 `extension` 来实现协议的方法。这可以保持协议相关的方法聚合在一起，同时也可以简单的标识出一个协议对应类中需要实现哪些对应的方法。

同时，别忘了添加 `// MARK:` ，注释可以使得代码组织的更好！

推荐做法：

```
class MyViewcontroller: UIViewController {
  // class stuff here
}
// MARK: - UITableViewDataSource
extension MyViewcontroller: UITableViewDataSource {
  // table view data source methods
}
// MARK: - UIScrollViewDelegate
extension MyViewcontroller: UIScrollViewDelegate {
  // scroll view delegate methods
}
```

不推荐做法：

```
class MyViewcontroller: UIViewController, UITableViewDataSource, UIScrollViewDelegate
{
  // all methods
}
```

对于UIKit view controllers, 建议用 `extension` 定义不同的类，按照生命周期，自定义方法，IBAction分组。

无用代码 (Unused Code)

无用的代码，包括Xcode生成的模板代码和占位符注释应该删除，除非是有目的性的保留这些代码。

一些方法内只是简单地调用了父类里面的方法也需要删除，包括UIApplicationDelegate内的空方法和无用方法。

推荐：

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
  return Database.contacts.count
}
```

不推荐：

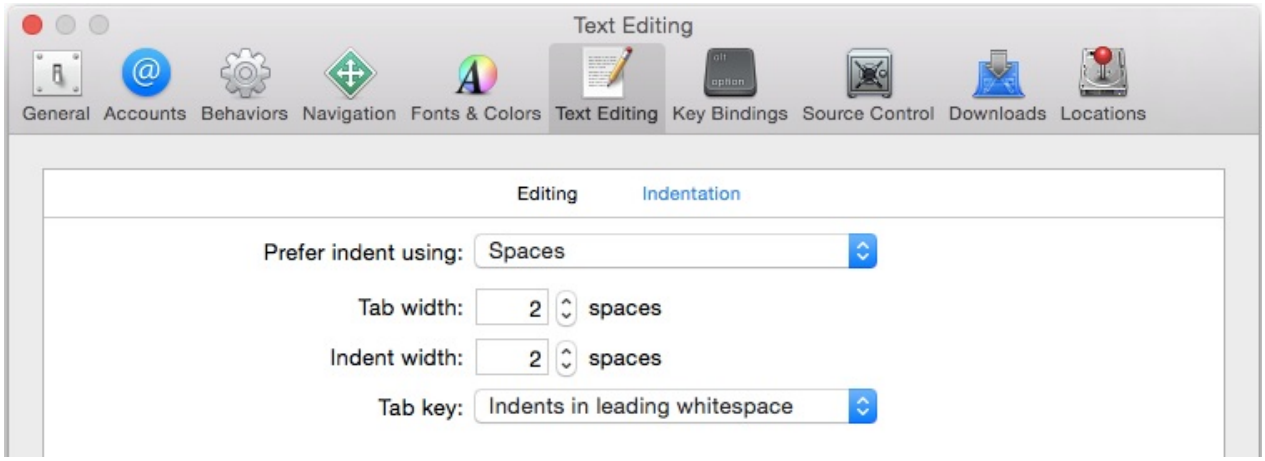
```
override func didReceiveMemoryWarning() {
  super.didReceiveMemoryWarning()
  // Dispose of any resources that can be recreated.
}

override func numberOfSections(in tableView: UITableView) -> Int {
  // #warning Incomplete implementation, return the number of sections
  return 1
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
  // #warning Incomplete implementation, return the number of rows
  return Database.contacts.count
}
```

最少引入(Minimal Imports)

减少不必要的引入，例如引入 `Foundation` 能满足的情况下不用引入 `UIKit` 。

空格（Spacing）

- 使用2个空格的缩进比使用tabs更好，可以减少占用空间和帮助防止多次换行。确保在Xcode进行了下图的偏好设置：



Xcode indent settings

- 方法定义的大括号或者其他大括号（ `if` / `else` / `switch` / `while` 等）—— 一般都放在定义名称的同一行，并且使用一个新的行来结束。
- 提示：你可以通过以下方法重新进行缩进：选择一些代码（或者使用⌘A选择所有），然后按Control-I(或者点击菜单栏 Editor\Structure\Re-Indent）。一些Xcode模板代码使用的缩进是4个空格，所以这种方法可以很好的修复缩进。

推荐做法：

```
if user.isHappy {
  // Do something
} else {
  // Do something else
}
```

不推荐做法：

```
if user.isHappy
{
    // Do something
}
else {
    // Do something else
}
```

- 应该在方法之间空出一行，从视觉上有更好的区分和组织。方法内的空白行隔开不同的功能，但是当一个方法中有很多段落时，也意味着你应该将该方法重构成几个方法。
- 冒号总是在左边不留空格，右边留有一个空格。例外有三元运算符`? :`, 空字典`[:]`和`#selector`语法中的未命名参数`(_:)`

推荐：

```
class TestDatabase: Database {
  var data: [String: CGFloat] = ["A": 1.2, "B": 3.2]
}
```

不推荐：

```
class TestDatabase : Database {
  var data :[String:CGFloat] = ["A" : 1.2, "B":3.2]
}
```

- 长行代码应该被限制在大约70个字符内。硬限制是故意不指定的。
- 避免在每行的末尾留有空格。
- 在每个文件的末尾添加一个换行符。

注释（Comments）

当你需要时，使用注释来解释一段特定的代码段的作用。注释必须保证更新或者及时删除。

避免在代码中使用块注释，代码尽可能自己能表达含义。以下情况除外：当使用注释来生成文档时。

类和结构体（Classes and Structures）

选择使用谁？（Which one to use?）

请记住，结构体是值类型。使用结构体并没有一个标识。一个数组包含 `[a, b, c]` 和另外一个数组同样包含 `[a, b, c]` 是完全一样的，它们完全可以交换使用。使用第一个还是使用第二个无关紧要，因为它们代表的是同一个东西。这就是为什么数组是结构体。

类是引用类型。使用类是有一个标识或者有一个特定的生命周期。你需要对一个人类建模为一个类，因为两个不同的人的实例，是两个不同的东西。只是因为两个人有同样的名字和生日，也不能断定这两个人是一样的。但是人的生日是一个结构体，因为日期1950-3-3和另外一个日期1950-3-3是相同的。日期不需要一个标识。

有时，一些事物应该定义为结构体，但是需要兼容AnyObject或者已经在以前的历史版本中定义为类（ `NSDate` ， `NSSet` ）。尽可能的尝试遵守这些规则。

定义的案例（Example definition）

以下是一个风格很好的类定义:

```
class Circle: Shape {
  var x: Int, y: Int
  var radius: Double
  var diameter: Double {
    get {
      return radius * 2
    }
    set {
      radius = newValue / 2
    }
  }
  init(x: Int, y: Int, radius: Double) {
    self.x = x
    self.y = y
    self.radius = radius
  }
  convenience init(x: Int, y: Int, diameter: Double) {
    self.init(x: x, y: y, radius: diameter / 2)
  }
  func describe() -> String {
    return "I am a circle at \(centerString()) with an area of \(computeArea())"
  }
  override func computeArea() -> Double {
    return M_PI * radius * radius
  }
  private func centerString() -> String {
    return "(\(x),\(y))"
  }
}
```

以上例子遵循了以下风格规范：

- 指定属性，变量，常量，参数定义或者其他定义的类型，在冒号后面，紧跟着一个空格，而不是把空格放在冒号前面。比如：`x: Int` 和 `Circle: Shape` 。
- 如果能表示相同的目的和上下文，可以在同一行定义多个变量和结构体。
- 缩进getter，setter的定义和属性观察器的定义。
- 不需要添加 `internal` 这样的默认的修饰符。同样的，不需要在重写一个方法时添加访问修饰符。
- 在扩展中组织额外的功能（如打印）
- 隐藏非共享的实现细节，如在扩展中的 `centerString` 内使用 `private` 访问控制。

Self的使用（Use of Self）

为了保持简洁，避免使用 self 关键词，Swift 不需要使用 `self` 来访问对象属性和调用对象方法。

必须使用 `self` 来区分构造器中属性命名和参数命名，还有在闭包表达式中引用属性值(编译器需要区分):

```
class BoardLocation {
  let row: Int, column: Int
  init(row: Int, column: Int) {
    self.row = row
    self.column = column
    let closure = {
      println(self.row)
    }
  }
}
```

计算属性（Computed Properties）

为了保持简洁，如果一个计算属性是只读的，请忽略掉get语句。只有在需要定义set语句的时候，才提供get语句。

推荐做法：

```
var diameter: Double {
  return radius * 2
}
```

不推荐做法：

```
var diameter: Double {
  get {
    return radius * 2
  }
}
```

Final

给那些不打算被继承的类使用 `final` 修饰符， 例如:

```
// Turn any generic type into a reference type using this Box class.
final class Box<T> {
  let value: T
  init(_ value: T) {
    self.value = value
  }
}
```

函数声明（Function Declarations）

保证短的函数定义在同一行中，并且包含左大括号：

```
func reticulateSplines(spline: [Double]) -> Bool {
  // reticulate code goes here
}
```

在一个长的函数定义时，在适当的地方进行换行，同时在下一行中添加一个额外的缩进：

```
func reticulateSplines(spline: [Double], adjustmentFactor: Double,
    translateConstant: Int, comment: String) -> Bool {
  // reticulate code goes here
}
```

闭包表达式（Closure Expressions）

如果闭包表达式参数在参数列表中的最后一个时，使用尾部闭包表达式。给定闭包参数一个描述性的命名。

推荐做法：

```
UIView.animateWithDuration(1.0) {
  self.myView.alpha = 0
}
UIView.animateWithDuration(1.0,
  animations: {
    self.myView.alpha = 0
  },
  completion: { finished in
    self.myView.removeFromSuperview()
  }
)
```

不推荐做法：

```
UIView.animateWithDuration(1.0, animations: {
  self.myView.alpha = 0
})
UIView.animateWithDuration(1.0,
  animations: {
    self.myView.alpha = 0
  }) { f in
    self.myView.removeFromSuperview()
}
```

当单个闭包表达式上下文清晰时，使用隐式的返回值：

```
attendeeList.sort { a, b in
  a > b
}
```

链式方法使用尾随闭包会更清晰易读，至于如何使用空格，换行，还是使用命名和匿名参数不做具体要求。

```
let value = numbers.map { $0 * 2 }.filter { $0 % 3 == 0 }.index(of: 90)

let value = numbers
  .map {$0 * 2}
  .filter {$0 > 50}
  .map {$0 + 10}
```

类型（Types）

尽可能使用 Swift 原生类型。Swift 提供到 Objective-C 类型的桥接，所以你仍然可以使用许多需要的方法。

推荐做法：

```
let width = 120.0                                    // Double
let widthString = (width as NSNumber).stringValue    // String
```

不推荐做法：

```
let width: NSNumber = 120.0                          // NSNumber
let widthString: NSString = width.stringValue        // NSString
```

在 Sprite Kit 代码中，使用 `CGFloat` 可以使得代码更加简明，避免很多转换。

常量（Constants）

常量定义使用 `let` 关键字，变量定义使用 `var` 关键字，如果变量的值不需要改变，请尽量使用 `let` 关键字。

提示：一个好的技巧是，使用 `let` 定义任何东西，只有在编译器告诉我们值需要改变的时候才改成 `var` 定义。

你可以使用类型属性来定义类型常量而不是实例常量，使用static let 可以定义类型属性常量。 这样方式定义类别属性整体上优于全局常量，因为更容易区分于实例属性. 比如:

推荐：

```
enum Math {
  static let e = 2.71828182845904523536028747
  static let root2 = 1.41421356237309504880168872
}

let hypotenuse = side * Math.root2
```

注意: 使用枚举的好处是变量不会被无意初始化，且全局有效。
不推荐：

```
let e = 2.71828182845904523536028747  // pollutes global namespace
let root2 = 1.41421356237309504880168872

let hypotenuse = side * root2 // what is root2?
```

静态方法和变量类型属性 (Static Methods and Variable Type Properties)

静态方法和类型属性的工作原理类似于全局方法和全局属性，应该节制使用。它们的使用场景在于如果某些功能局限于特别的类型或和Objective-C 互相调用。

可选类型（Optionals）

当nil值是可以接受的时候时，定义变量和函数返回值为可选类型 `?` 。

当你确认变量在使用前已经被初始化时，使用 `!` 来显式的拆包类型，比如在 `viewDidLoad` 中会初始化 `subviews` 。

当你访问一个可选值时，如果只需要访问一次或者在可选值链中有多个可选值时，请使用可选值链：

```
self.textContainer?.textLabel?.setNeedsDisplay()
```

当需要很方便的一次性拆包或者添加附加的操作时，请使用可选值绑定：

```
if let textContainer = self.textContainer {
  // do many things with textContainer
}
```

当我们命名一个可选变量和属性时，避免使用诸如 `optionalString` 和 `maybeView` 这样的命名，因为可选值的表达已经在类型定义中了。

在可选值绑定中，直接映射原始的命名比使用诸如 `unwrappedView` 和 `actualLabel` 要好。

推荐做法：

```
var subview: UIView?
var volume: Double?
// later on...
if let subview = subview, volume = volume {
  // do something with unwrapped subview and volume
}
```

不推荐做法：

```
var optionalSubview: UIView?
var volume: Double?
if let unwrappedSubview = optionalSubview {
  if let realVolume = volume {
    // do something with unwrappedSubview and realVolume
  }
}
```

延迟初始化 (Lazy Initialization)

延迟初始化用来细致地控制对象的生命周期，这对于想实现延迟加载视图的UIViewController特别有用，你可以使用即时被调用闭包或私有构造方法：

```
lazy var locationManager: CLLocationManager = self.makeLocationManager()

private func makeLocationManager() -> CLLocationManager {
  let manager = CLLocationManager()
  manager.desiredAccuracy = kCLLocationAccuracyBest
  manager.delegate = self
  manager.requestAlwaysAuthorization()
  return manager
}
```

注意：

- `[unowned self]` 在这里不是必须的，应为没有产生引用循环。
- Location manager 的负面效果会弹出对话框要求用户提供权限，这是做延时加载的原因。

结构体构造器（Struct Initializers）

使用原生的 Swift 结构体构造器，比老式的几何类（CGGeometry）的构造器要好。

推荐做法：

```
let bounds = CGRect(x: 40, y: 20, width: 120, height: 80)
let centerPoint = CGPoint(x: 96, y: 42)
```

不推荐做法：

```
let bounds = CGRectMake(40, 20, 120, 80)
let centerPoint = CGPointMake(96, 42)
```

推荐使用结构体限定的常量 `CGRect.infiniteRect`，`CGRect.nullRect` 等，来替代全局常量 `CGRectInfinite`，`CGRectNull` 等。对于已经存在的变量，可以直接简写成 `.zeroRect` 。

类型推断（Type Inference）

推荐使用更加紧凑的代码，让编译器能够推断出常量和变量的类型。除非你需要定义一个特定的类型(比如 `CGFloat` 和 `Int16`)，而不是默认的类型。

推荐做法：

```
let message = "Click the button"
let currentBounds = computeViewBounds()
var names = ["Mic", "Sam", "Christine"]
let maximumWidth: CGFloat = 106.5
```

不推荐做法：

```
let message: String = "Click the button"
let currentBounds: CGRect = computeViewBounds()
let names = [String]()
```

类型注解对空的数组和字典

对空的数据和字典，使用类型注解。

推荐：

```
var names: [String] = []
var lookup: [String: Int] = [:]
```

不推荐：

```
var names = [String]()
var lookup = [String: Int]()
```

注意：遵守这条规则意味选择描述性命名比之前变得更加重要。

语法糖（**Syntactic Sugar**）

推荐使用类型定义简洁的版本，而不是全称通用语法。

推荐做法：

```
var deviceModels: [String]
var employees: [Int: String]
var faxNumber: Int?
```

不推荐做法：

```
var deviceModels: Array<String>
var employees: Dictionary<Int, String>
var faxNumber: Optional<Int>
```

函数 vs 方法 (Functions vs Methods)

自由函数不依附于任何类或类型，应该节制地使用。如果可能优先使用方法而不是自由函数，这有助于代码的可读性和易发现性。

自由函数使用的场景是当不确定它和具体的类别或实例相关。

推荐:

```
let sorted = items.mergeSorted()  // easily discoverable
rocket.launch()  // acts on the model
```

不推荐：

```
let sorted = mergeSort(items)  // hard to discover
launch(&rocket)
```

自由函数：

```
let tuples = zip(a, b)  // feels natural as a free function (symmetry)
let value = max(x, y, z)  // another free function that feels natural
```

内存管理 (Memory Management)

代码应避免指针循环引用，分析对象图谱，使用弱引用 `weak` 和未知引用 `unowned` 避免强引用循环。另外使用值类型(`struct` 和 `enum`)可以避免循环引用。

延长对象生命周期

延长对象生命周期习惯上使用 `[weak self]` 和 `guard let strongSelf = self else { return }` 。 `[weak self]` 优于 `[unowned self]` 因为前者更更能明显地 `self` 生命周期长于闭包块。显式延长生命周期优先于可选性拆包。

推荐：

```
resource.request().onComplete { [weak self] response in
  guard let strongSelf = self else {
    return
  }
  let model = strongSelf.updateModel(response)
  strongSelf.updateUI(model)
}
```

不推荐：

```
// might crash if self is released before response returns
resource.request().onComplete { [unowned self] response in
  let model = self.updateModel(response)
  self.updateUI(model)
}
```

不推荐：

```
// deallocate could happen between updating the model and updating UI
resource.request().onComplete { [weak self] response in
  let model = self?.updateModel(response)
  self?.updateUI(model)
}
```

访问控制 (Access Control)

合理的使用 `private` 和 `fileprivate`，推荐使用 `private`，在使用 `extension` 时可使用 `fileprivate`。

访问控制符一般放在属性修饰符的最前面. 除非需要使用 `static` 修饰符，`@IBAction`，`@IBOutlet` 或 `@discardableResult`。

推荐：

```
private let message = "Great Scott!"

class TimeMachine {
  fileprivate dynamic lazy var fluxCapacitor = FluxCapacitor()
}
```

不推荐：

```
fileprivate let message = "Great Scott!"

class TimeMachine {
  lazy dynamic fileprivate var fluxCapacitor = FluxCapacitor()
}
```

控制流（Control Flow）

推荐循环使用 `for-in` 表达式，而不使用 `for-condition-increment` 表达式。

推荐做法：

```
for _ in 0..<3 {
  print("Hello three times")
}

for (index, person) in attendeeList.enumerated() {
  print("\(person) is at position #\(index)")
}

for index in stride(from: 0, to: items.count, by: 2) {
  print(index)
}

for index in (0...3).reversed() {
  print(index)
}
```

不推荐做法：

```
var i = 0
while i < 3 {
  print("Hello three times")
  i += 1
}


var i = 0
while i < attendeeList.count {
  let person = attendeeList[i]
  print("\(person) is at position #\(i)")
  i += 1
}
```

黄金路径 (Golden Path)

当编码遇到条件判断时，左边的距离是黄金路径或幸福路径，因为路径越短，速度越快。不要嵌套 `if` 循环，多个返回语句是可以的。 `guard` 就为此而生的。
推荐：

```
func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {

  guard let context = context else {
    throw FFTError.noContext
  }
  guard let inputData = inputData else {
    throw FFTError.noInputData
  }

  // use context and input to compute the frequencies
  return frequencies
}
```

不推荐：

```
func computeFFT(context: Context?, inputData: InputData?) throws -> Frequencies {

  if let context = context {
    if let inputData = inputData {
      // use context and input to compute the frequencies

      return frequencies
    } else {
      throw FFTError.noInputData
    }
  } else {
    throw FFTError.noContext
  }
}
```

当有多个条件需要用 `guard` 或 `if let` 解包，可用复合语句避免嵌套。

推荐：

```
guard let number1 = number1,
      let number2 = number2,
      let number3 = number3 else {
  fatalError("impossible")
}
// do something with numbers
```

不推荐：

```
if let number1 = number1 {
  if let number2 = number2 {
    if let number3 = number3 {
      // do something with numbers
    } else {
      fatalError("impossible")
    }
  } else {
    fatalError("impossible")
  }
} else {
  fatalError("impossible")
}
```

失败防护

防护语句的退出有很多方式，一般都是单行语句，如 `return`，`throw`，`break`，`continue` 和 `fatalError()` 等。避免出现大的代码块，如果清理代码需要多个退出点，可以用 `defer` 模块避免重复清理代码。

分号（Semicolons）

Swift 不需要在你代码中的每一句表达式之后添加分号。只有在你需要在一行中连接多个表达式中，使用分号来区隔。

不要在同一行编写多个使用分号区隔的表达式。

唯一的例外是在使用 `for-conditional-increment` 架构。然而，尽可能使用 `for-in` 架构来替代它。

推荐做法：

```
let swift = "not a scripting language"
```

不推荐做法：

```
let swift = "not a scripting language";
```

注意：Swift与JavaScript有很大的不同，JavaScript认为忽略分号通常认为是不安全的。

圆括号 (Parentheses)

条件判断时圆括号不是必须的，建议省略。

推荐：

```
if name == "Hello" {
  print("World")
}
```

不推荐：

```
if (name == "Hello") {
  print("World")
}
```

在较大的表达式中，可选的括号有时可以使代码更清楚。

```
let playerMark = (player == current ? "X" : "O")
```

组织和包标识符 (Organization and Bundle Identifier)

版权声明（Copyright Statement）

以下的版权声明应该被包含在所有源文件的顶部：

/*

- Copyright (c) 2017 `Your Organization`
-
- Permission is hereby granted, free of charge, to any person obtaining a copy
- of this software and associated documentation files (the “Software”), to deal
- in the Software without restriction, including without limitation the rights
- to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
- copies of the Software, and to permit persons to whom the Software is
- furnished to do so, subject to the following conditions:
-
- The above copyright notice and this permission notice shall be included in
- all copies or substantial portions of the Software.
-
- THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND,

EXPRESS OR
- IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
- FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
- AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
- LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
- OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
- THE SOFTWARE.

*/

Objective-C新特性

Modules

新的编译符号 `@import`

```
@import UIKit.UIView;
```

编译优化，避免编译时重复引用，增加编译速度。

Nullability

常规用法

```
@property (nonatomic, strong, nonnull) Sark *sark;
@property (nonatomic, copy, readonly, nullable) NSArray *friends;
+ (nullable NSString *)friendWithName:(nonnull NSString *)name;
```

修饰变量，前面需要加下划线，比如 block 内用法：

```
- (void)startWithCompletionBlock:(nullable void (^)(NSError * _Nullable error))block;
```

setter 用法，参见 UIViewController 中的 view 属性，它可以被设成 nil，但是调用 getter 时会触发 -loadView 从而创建并返回一个非 nil 的 view。

```
@property (null_resettable, nonatomic, strong) UIView *view;
```

Audited Regions宏的用法（包在宏里面的对象默认加 `nonnull` 修饰符，只需要把 `nullable` 的指出来就行）：

```
NS_ASSUME_NONNULL_BEGIN
@interface Sark : NSObject
@property (nonatomic, copy, nullable) NSString *workingCompany;
@property (nonatomic, copy) NSArray *friends;
- (nullable NSString *)gayFriend;
@end
NS_ASSUME_NONNULL_END
```

Nullability 主要作用是在编译器层面提供了空值的类型检查，在类型不符时给出 warning，方便开发者第一时间发现潜在问题。

- 对于属性、方法返回值、方法参数的修饰，使用：`nonnull` / `nullable` ；
- 对于 C 函数的参数、Block 的参数、Block 返回值的修饰，使用：`_Nonnull` / `_Nullable` ，建议弃用~~ `__nonnull` / `__nullable` ~~；
- 通过 `typedef` 定义的类型的 nullability 特性通常依赖于上下文，即使是在 Audited Regions 中，也不能假定它为 `nonnull` ；
- 对于复杂的指针类型（如 `id *` ）必须显式去指定是 `nonnull` 还是 `nullable` 。例如，指定一个指向 `nullable` 对象的 `nonnull` 指针，可以使用 `__nullable id * __nonnull` ；
- 我们经常使用的 `NSError **` 通常是被假定为一个指向 `nullable` NSError 对象的 `nullable` 指针。

__kindof

主要作用还是编译器层面的类型检查

```
//UIView 的写法
@property (nonatomic, readonly, copy) NSArray<__kindof UIView *> *subviews;

// UITableView 的写法
- (nullable __kindof UITableViewCell *)dequeueReusableCellWithIdentifier:(NSString *)i
dentifier;
```

泛型

带泛型的容器

```
NSArray<NSString *> *strings = @[@"sun", @"yuan"];
NSDictionary<NSString *, NSNumber *> *mapping = @{@"a": @1, @"b": @2};
```

自定义泛型

```
@interface Stack<ObjectType> : NSObject
- (void)pushObject:(ObjectType)object;
- (ObjectType)popObject;
@property (nonatomic, readonly) NSArray<ObjectType> *allObjects;
@end
```

还可以增加限制

```
// 只接受 NSNumber * 的泛型
@interface Stack<ObjectType: NSNumber *> : NSObject
// 只接受满足 NSCopying 协议的泛型
@interface Stack<ObjectType: id<NSCopying>> : NSObject
```

covariant && contravariant

- `__covariant`：子类型可以强转到父类型（里氏替换原则）
- `__contravariant`：父类型可以强转到子类型

参考 NSArray 和 NSMutableArray 的定义

```
// NSArray
@interface NSArray<__covariant ObjectType> : NSObject <NSCopying, NSMutableCopying, NS
SecureCoding, NSFastEnumeration>

@property (readonly) NSUInteger count;
- (ObjectType)objectAtIndex:(NSUInteger)index;
- (instancetype)init NS_DESIGNATED_INITIALIZER;
- (instancetype)initWithObjects:(const ObjectType [])objects count:(NSUInteger)cnt NS_
DESIGNATED_INITIALIZER;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder NS_DESIGNATED_INITIALIZER;

@end

// NSMutableArray
@interface NSMutableArray<ObjectType> : NSArray<ObjectType>

- (void)addObject:(ObjectType)anObject;
- (void)insertObject:(ObjectType)anObject atIndex:(NSUInteger)index;
- (void)removeLastObject;
- (void)removeObjectAtIndex:(NSUInteger)index;
- (void)replaceObjectAtIndex:(NSUInteger)index withObject:(ObjectType)anObject;
- (instancetype)init NS_DESIGNATED_INITIALIZER;
- (instancetype)initWithCapacity:(NSUInteger)numItems NS_DESIGNATED_INITIALIZER;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder NS_DESIGNATED_INITIALIZER;

@end
```

Designated Initializer

Objective-C 中主要通过 `NS_DESIGNATED_INITIALIZER` 宏来实现指定构造器的。

参考 UIViewController 的两个指定构造器：

```
- (instancetype)initWithNibName:(nullable NSString *)nibNameOrNil bundle:(nullable NSB
undle *)nibBundleOrNil NS_DESIGNATED_INITIALIZER;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder NS_DESIGNATED_INITIALIZER;
```

类似 Swift 的指定初始化，原则如下（类比 Swift 的构造构成）：

- 每个类的正确初始化过程应当是按照从子类到父类的顺序，依次调用每个类的Designated Initializer。并且用父类的Designated Initializer初始化一个子类对象，也需要遵从这个过程。
- 如果子类指定了新的初始化器，那么在这个初始化器内部必须调用父类的Designated Initializer。并且需要重写父类的Designated Initializer，将其指向子类新的初始化器。
- 你可以不自定义Designated Initializer，也可以重写父类的Designated Initializer，但需要调用直接父类的Designated Initializer。
- 如果有多个Secondary initializers(次要初始化器)，它们之间可以任意调用，但最后必须指向Designated Initializer。在Secondary initializers内不能直接调用父类的初始化器。
- 如果有多个不同数据源的Designated Initializer，那么不同数据源下的Designated Initializer应该调用相应的 `[super (designated initializer)]` 。如果父类没有实现相应的方法，则需要根据实际情况来决定是给父类补充一个新的方法还是调用父类其他数据源的Designated Initializer。比如UIView的 `initWithCoder` 调用的是NSObject的 `init` 。
- 需要注意不同数据源下添加额外初始化动作的时机。

iOS生命周期

应用生命周期

iOS的应用程序有5种状态:

- **Not Running(**非运行状态**)**

应用没有运行或被系统终止。

- **Inactive(**前台非活动状态**)**

应用正在进入前台状态，但是还不能接受事件处理。

- **Active(**前台活动状态**)**

应用进入前台状态，能接受事件处理。

- **Background(**后台状态**)**

应用进入后台后，依然能够执行代码。如果有可执行的代码，就会执行代码，如果没有可执行的代码或者将可执行的代码执行完毕，应用会马上进入挂起状态。有的程序经过特殊的请求后可以长期处于Backgroud状态。

- **Suspended(**挂起状态**)**

处于挂起的应用进入一种“冷冻”状态,不能执行代码。如果系统内存不够,系统就把挂起的程序清除掉，为前台程序提供更多的内存，应用会被终止。