Objective-C 代码开发规范

V1.0

版本号	状态	日期	摘要	撰稿人	校对	审核
V1.0	新建	2019-07-01	Objective-C 代码开发规范	张锦帅		

示例

命名

【强制】禁用非标准缩写（包括非标准首字母缩略词）

【强制】前缀被用来避免全局命名空间的命名冲突。类名、协议、全局方法和全局常量命名，要添加前缀，并且首字母大写，公司前缀为DZ

【推荐】任何类、扩展、方法、函数或者变量名中简称或者缩略语必须大写。对于名字，可参照苹果缩略语或简称大写字母标准，例如URL, ID, TIFF, EXIF。

【强制】C语言函数或者typedef命名须首字母大写，使用驼峰命名方式区分大小写

【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式

【强制】文件命名须与该文件中实现的类名保持一致，包括大小写

【强制】包含跨工程或者较大工程中共用的代码的文件，必须有一个清晰唯一的名字，一般包括工程或者类名作为前缀。

【强制】扩展类的文件名，须包含被扩展类的类名，例如DZNSString+Utils.h or NSTextView+DZAutocomplete.h

【强制】类命名（连同扩展和协议命名）需要使用首字母大写，遵循驼峰命名法。

【强制】Category名要使用合适的前缀，来表明该扩展是某一工程的一部分或可以通用。

【推荐】若类不被其他工程共用，扩展名和方法名可省略前缀。

【强制】Objective-C方法、参数、成员变量、局部变量，通常以小写开头，遵循驼峰命名法，但大写的简称或者缩略语可以沿用，即使位于开头。

【推荐】方法名应该尽量读起来像一个句子，告诉这个方法后面应该传入的参数名。Objective-C方法名倾向于非常长，但是这样的好处就是代码块读起来像文章，从而使得许多注释变得不必要。

【强制】只有在有必要说明方法意义或行为时，才在第二个参数名后，使用例如"with", "from", and "to"等介词和连词。

【强制】返回对象的方法名需要以名词开头，以表明返回的对象类型。

【强制】访问器方法的名称应与它获取的对象相同，不可有get前缀。例如：

【强制】返回BOOL类型的方法，要以is为开头，但属性名省略is

【强制】点语法只用于属性获取，不可用于方法调用

常量

【强制】不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

【强制】long 或者 Long 初始赋值时，必须使用大写的 L，不能是小写的 l，小写容易跟数字 1 混淆，造成误解。

【推荐】不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护

【强制】全局和文件作用域常量要带有合适的前缀。

【推荐】常量首选内联字符串字面量或数字，因为常量可以轻易重用并且可以快速改变而不需要查找和替换。常量应该声明为 `static` 常量而不是 `#define` ，除非非常明确地要当做宏来使用。

【强制】在.m文件中，小写字母k可以被用来作为声明独立的常量或者静态变量前缀：

类型和声明

【强制】方法声明，如最开始的示例所示，声明一个 @interface 内容的推荐顺序为：属性，类方法，初始化方法，实例方法。类方法区域，把便捷初始化方法放在前面。

【强制】除匹配系统接口调用外，避免使用无符号整型NSUInteger。

【强制】若能使用const变量，枚举，Xcode代码片段，或C函数，请不要使用宏。

注释

【强制】类、属性、成员变量、方法、扩展、协议声明及枚举注释必须使用 Xcode 注释规范，使用/*内容/格式，不得使用 //xxx 方式，因为前者在XCode中会有代码提示

【强制】所有的抽象方法（包括接口中的方法）必须要用 Xcode 注释、除了返回值、参数、 异常说明外，还必须指出该方法做什么事情，实现什么功能。

【强制】所有的类都必须添加创建者信息。

【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释 使用/* */注释，注意与代码对齐。

【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

【强制】对刁钻、微妙或复杂的代码添加注释，解释其中奥妙。

【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。

【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

【参考】注释掉的代码尽量要配合说明，而不是简单的注释掉。

【参考】对于注释的要求：

【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描， 经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。

【推荐】每个公开和私有的重要接口，要有附加的注释说明其作用和适用场景。

Cocoa & Objective-C特性

【强制】明确 designated initializer

【强制】若实现一个类需要init方法，必须重写父类的 designated initializer。

【强制】不必在 init 方法中初始化实例变量为 0 或 nil ，这是多余的!

【强制】不要使用 +new

【强制】公开API保持简单

【推荐】通常私有方法需要有一个较为独特的名字，这样可以防止子类无意的重写这些方法。

【强制】使用系统库Umbrella头文件

【强制】避免在初始化器或-dealloc中给当前对象发消息

【推荐】拆解指定初始化方法码为工具方法要谨慎!

【强制】Setter方法中NSString、NSURLRequest或者容器要用Copy

【强制】用泛型标识元素类型

【推荐】如果指针检查只是为了防止给nil对象发消息，那这部分检查不必添加，给nil发消息的返回值是可靠的。

【推荐】可空性（Nullability）

【强制】注意BOOL陷阱

【强制】单例对象应该使用线程安全的模式创建共享的实例，这将会预防有时可能产生的许多崩溃。

【强制】注意`self`的循环引用，当使用代码块和异步分发的时候，要注意避免引用循环。 总是使用`weak`来引用对象，避免引用循环。

空间布局和格式

【强制】缩进时不使用制表符，只用4个空格

【强制】Objective-C最大行宽为120个字符。在XCode中，通过设置Preferences > Text Editing > Page guide at column:为120，可以轻松检查超宽地方

【强制】方法声明与定义中，- or + 后与返回值之间须有一个空格，参数名与参数类型间无空格，参数列表不同参数间有一个空格，指针定义的星号前有一个空格

【推荐】如果一个方法声明一行放不下，需要将不同参数各占一行

【强制】if, while, for, switch后有一个空格，比较运算符两边有空格，语句中必须使用大括号，即使只有一行代码

【强制】若 if 语句后有 else 语句，两部分都需要使用大括号。

【强制】除非两个case语句之间没有其他代码，有意添加的case连续执行情况，需要增加注释说明，在一个 switch 块内，都必须包含一个 default 语句并且放在 最后，即使它什么代码也没有。

【强制】二进制运算符、赋值运算符左右两边都需要添加空格。一元运算符可以省略空格。圆括号（左括号右边、右括号左边）不用空格。

【推荐】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。

【强制】方法调用的格式应与方法声明一致。

【推荐】推荐小而功能专注的函数，长方法或者函数某些情况下也是可以的，所以关于函数长度没有固定的限制。如果一个函数长度超过了40行，请考虑如何在不破坏代码结构的基础上，将它拆分。

【推荐】谨慎使用空白换行，为了让更多代码能够一屏展示，在函数大括号内，避免使用空行。不同函数间或不同代码逻辑组之间，空行限制在1-2行。

【强制】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】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。

【推荐】私有属性应该声明在类实现文件的延展（匿名的类目）中。

【推荐】为了避免文件杂乱，物理文件应该保持和 Xcode 项目文件同步

文档说明

适用范围

本文档适用于点掌iOS开发人员。

保密说明

本文档仅供点掌公司研发内部交流、学习和研究使用，禁止向外传播。

正文解释

本文档中【强制】是必须要遵循的，作为代码规范考核重要依据。 【推荐】正常情况下要遵循，如果不按要求使用需要在注释中写明理由。

1. 示例

一个例子胜过千言万语，下面的例子，演示如何声明一个@interface 正确的注释及代码空格空行编码风格

```
// GOOD:

#import <Foundation/Foundation.h>

@class Bar;

/**
 * A sample class demonstrating good Objective-C style. All interfaces,
 * categories, and protocols (read: all non-trivial top-level declarations
 * in a header) MUST be commented. Comments must also be adjacent to the
 * object they're documenting.
 */
@interface Foo : NSObject

/** The retained Bar. */
@property (nonatomic, strong) Bar *bar;

/** The current drawing attributes. */
@property (nonatomic, copy) NSDictionary<NSString *, NSNumber *> *attributes;

/**
 * Convenience creation method.
 * See -initWithBar: for details about @c bar.
 *
 * @param bar The string for fooing.
 * @return An instance of Foo.
 */
+ (instancetype)fooWithBar:(Bar *)bar;

/**
 * Initializes and returns a Foo object using the provided Bar instance.
 *
 * @param bar A string that represents a thing that does a thing.
 */
- (instancetype)initWithBar:(Bar *)bar NS_DESIGNATED_INITIALIZER;

/**
 * Does some work with @c blah.
 *
 * @param blah
 * @return YES if the work was completed; NO otherwise.
```

```
 */
- (BOOL)doWorkWithBlah:(NSString *)blah;

@end
```

实现一个@implementation 的正确编码规范示例，展示正确的注释和间距（包括空行，回车，空格等）

```
// GOOD:

#import "Foo.h"

@interface Foo ()

/** The string used for displaying "hi". */
@property (nonatomic, copy) NSString *string;

@end

@implementation Foo

+ (instancetype)fooWithBar:(Bar *)bar {
    return [[self alloc] initWithBar:bar];
}

- (instancetype)init {
    // Classes with a custom designated initializer should always override
    // the superclass's designated initializer.
    return [self initWithBar:nil];
}

- (instancetype)initWithBar:(Bar *)bar {
    self = [super init];
    if (self) {
        _bar = [bar copy];
        _string = [[NSString alloc] initWithFormat:@"hi %d", 3];
        _attributes = @{
            @"color" : [UIColor blueColor],
            @"hidden" : @NO
        };
    }
    return self;
}

- (BOOL)doWorkWithBlah:(NSString *)blah {
    // Work should be done here.
    return NO;
}
```

```
@end
```

2. 命名

2.1. 【强制】禁用非标准缩写（包括非标准首字母缩略词）

相对节约空间来说，对于一个新的读者来说，能够立即理解意义会更为重要，例如：

```
// GOOD:

// Good names.
int numberOfErrors = 0;
int completedConnectionsCount = 0;
tickets = [[NSMutableArray alloc] init];
userInfo = [someObject object];
port = [network port];
NSDate *gAppLaunchDate;
```

```
// AVOID:

// Names to avoid.
int w;
int nerr;
int nCompConns;
tix = [[NSMutableArray alloc] init];
obj = [someObject object];
p = [network port];
```


2.2. 【强制】前缀被用来避免全局命名空间的命名冲突。类名、协议、全局方法和全局常量命名，要添加前缀，并且首字母大写，公司前缀为DZ

2.3. 【推荐】任何类、扩展、方法、函数或者变量名中简称或者缩略语必须大写。对于名字，可参照苹果缩略语或简称大写字母标准，例如URL, ID, TIFF, EXIF。

2.4. 【强制】C语言函数或者typedef命名须首字母大写，使用驼峰命名方式区分大小写

2.5. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式

说明:正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注 意，即使纯拼音命名方式 也要避免采用。 反例: DaZhePromotion [打折] / getPingfenByName [评分] / int 某变量=3 正例: taobao / youku / shanghai 等国际通用的名称，可视同英文。

2.6. 【强制】文件命名须与该文件中实现的类名保持一致，包括大小写

2.7. 【强制】包含跨工程或者较大工程中共用的代码的文件，必须有一个清晰唯一的名字，一般包括工程或者类名作为前缀。

2.8. 【强制】扩展类的文件名，须包含被扩展类的类名，例如DZNSString+Utils.h or NSTextView+DZAutocomplete.h

```
// GOOD:

/** An example error domain. */
extern NSString *DZExampleErrorDomain;

/** Gets the default time zone. */
extern NSTimeZone *DZGetDefaultTimeZone(void);

/** An example delegate. */
@protocol DZExampleDelegate <NSObject>
@end

/** An example class. */
@interface DZExample : NSObject
@end
```

2.9. 【强制】类命名（连同扩展和协议命名）需要使用首字母大写，遵循驼峰命名法。

2.10. 【强制】Category名要使用合适的前缀，来表明该扩展是某一工程的一部分或可以通用。

Category源文件名必须以被扩展类名开头，中间用+号衔接后面的扩展名字，例如：NSString+DZParsing.h。扩展中的方法以小写的扩展名为前缀，中间用下划线衔接方法名（例如：dz_myCategoryMethodOnAString:），以便避免全局命名范围内的命名冲突。类名和扩展名的左圆括号之间，须添加一个空格。

```
// GOOD:

// UIViewController+DZCrashReporting.h

/** A category that adds metadata to include in crash reports to
UIViewController. */
@interface UIViewController (DZCrashReporting)

/** A unique identifier to represent the view controller in crash reports. */
@property(nonatomic, setter=dz_setUniqueIdentifier:) int dz_uniqueIdentifier;

/** Returns an encoded representation of the view controller's current state.
*/
- (nullable NSData *)dz_encodedState;

@end
```

2.11. 【推荐】若类不被其他工程共用，扩展名和方法名可省略前缀。

```
// GOOD:

/** This category extends a class that is not shared with other projects. */
@interface XYZDataObject (Storage)
- (NSString *)storageIdentifier;
@end
```

2.12. 【强制】Objective-C方法、参数、成员变量、局部变量，通常以小写开头，遵循驼峰命名法，但大写的简称或者缩略语可以沿用，即使位于开头。

```
// GOOD:

NSString *myLocalVariable;

+ (NSURL *)URLWithString:(NSString *)URLString;
```

2.13. 【推荐】方法名应该尽量读起来像一个句子，告诉这个方法后面应该传入的参数名。Objective-C方法名倾向于非常长，但是这样的好处就是代码块读起来像文章，从而使得许多注释变得不必要。

2.14. 【强制】只有在有必要说明方法意义或行为时，才在第二个参数名后，使用例如"with", "from", and "to"等介词和连词。

```
- (void)addTarget:(id)target action:(SEL)action;                          //
GOOD; no conjunction needed
- (CGPoint)convertPoint:(CGPoint)point fromView:(UIView *)view;            //
GOOD; conjunction clarifies parameter
- (void)replaceCharactersInRange:(NSRange)aRange
            withAttributedString:(NSAttributedString *)attributedString;  //
GOOD.
```

2.15. 【强制】返回对象的方法名需要以名词开头，以表明返回的对象类型。

```
- (Sandwich *)sandwich;      // GOOD.

- (Sandwich *)makeSandwich;  // AVOID.
```

2.16. 【强制】访问器方法的名称应与它获取的对象相同，不可有get前缀。例如：

```
// GOOD:

- (id)delegate;     // GOOD.
```

```
// AVOID:

- (id)getDelegate;  // AVOID.
```

2.17. 【强制】返回BOOL类型的方法，要以is为开头，但属性名省略is

2.18. 【强制】点语法只用于属性获取，不可用于方法调用

```
// GOOD:

@property(nonatomic, getter=isGlorious) BOOL glorious;
- (BOOL)isGlorious;

BOOL isGood = object.glorious;       // GOOD.
BOOL isGood = [object isGlorious];   // GOOD.
```

```
// AVOID:

BOOL isGood = object.isGlorious;     // AVOID.
```

```
// GOOD:

NSArray<Frog *> *frogs = [NSArray<Frog *> arrayWithObject:frog];
NSEnumerator *enumerator = [frogs reverseObjectEnumerator];  // GOOD.
```

```
// AVOID: 避免使用.来调用方法
NSEnumerator *enumerator = frogs.reverseObjectEnumerator;    // AVOID.
```

3. 常量

3.1. 【强制】不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

3.2. 【强制】long 或者 Long 初始赋值时，必须使用大写的 L，不能是小写的 l，小写容易跟数字 1 混淆，造成误解。

说明：long int a = 2l; 写的是数字的 21，还是 Long 型的 2?

3.3. 【推荐】不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护

比如：缓存 相关的常量放在类：CacheConsts 下；系统配置相关的常量放在类：ConfigConsts 下。

说明：大而全的常量类，非得使用查找功能才能定位到修改的常量，不利于理解和维护。

3.4. 【强制】全局和文件作用域常量要带有合适的前缀。

```
// GOOD:

extern NSString *const GTLServiceErrorDomain;

typedef NS_ENUM(NSInteger, GTLServiceError) {
  GTLServiceErrorQueryResultMissing = -3000,
  GTLServiceErrorWaitTimedOut       = -3001,
};
```

3.5. 【推荐】常量首选内联字符串字面量或数字，因为常量可以轻易重用并且可以快速改变而不需要查找和替换。常量应该声明为 `static` 常量而不是 `#define` ，除非非常明确地要当做宏来使用。

```
// GOOD:

static NSString * const NYTAboutViewControllerCompanyName = @"The New York Times Company";

static const CGFloat NYTImageThumbnailHeight = 50.0;
```

```
// AVOID:

#define CompanyName @"The New York Times Company"

#define thumbnailHeight 2
```

3.6. 【强制】在.m文件中，小写字母k可以被用来作为声明独立的常量或者静态变量前缀：

```
// GOOD:

static const int kFileCount = 12;
static NSString *const kUserKey = @"kUserKey";
```


4. 类型和声明

4.1. 【强制】方法声明，如最开始的示例所示，声明一个 @interface 内容的推荐顺序为：属性，类方法，初始化方法，实例方法。类方法区域，把便捷初始化方法放在前面。

4.2. 【强制】除匹配系统接口调用外，避免使用无符号整型NSUInteger。

使用无符号整型进行数学计算或倒计数为0时，会出现微妙的错误。除匹配系统接口时使用NSUInteger外，数学计算时只使用带符号整型NSInteger。

```
// GOOD:

NSUInteger numberOfObjects = array.count;
for (NSInteger counter = numberOfObjects - 1; counter > 0; --counter)
```

```
// AVOID:

for (NSUInteger counter = numberOfObjects - 1; counter > 0; --counter)  //
AVOID.
```

4.3. 【强制】若能使用const变量，枚举，Xcode代码片段，或C函数，请不要使用宏。

宏会使你看到的代码和编译器看到的代码不同。现代C使用宏的的传统用途不再是常量和工具函数。现在宏只有在没有其他解决方案时才被使用。

需要宏的地方，使用唯一的名字以避免在汇编时符号冲突

宏的名字应该使用SHOUTY_SNAKE_CASE形式——所有字母大写，使用下划线链接不同单词。函数类的宏使用函数的命名规范。不要定义和C或Objective-C关键字同名的宏。

```
// GOOD:

#define DZ_EXPERIMENTAL_BUILD ...       // GOOD

// Assert unless X > Y
#define DZ_ASSERT_GT(X, Y) ...          // GOOD, macro style.

// Assert unless X > Y
#define DZAssertGreaterThan(X, Y) ...  // GOOD, function style.
// AVOID:

#define kIsExperimentalBuild ...         // AVOID

#define unless(X) if(!(X))               // AVOID
```


5. 注释

5.1. 【强制】类、属性、成员变量、方法、扩展、协议声明及枚举注释必须使用 Xcode 注释规范，使用/*内容/格式，不得使用 //xxx 方式，因为前者在XCode中会有代码提示

5.2. 【强制】所有的抽象方法（包括接口中的方法）必须要用 Xcode 注释、除了返回值、参数、 异常说明外，还必须指出该方法做什么事情，实现什么功能。

说明：对子类的实现要求，或者调用注意事项，请一并说明。

5.3. 【强制】所有的类都必须添加创建者信息。

5.4. 【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

5.5. 【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

5.6. 【强制】对刁钻、微妙或复杂的代码添加注释，解释其中奥妙。

5.7. 【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。

反例：“TCP 连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

5.8. 【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

5.9. 【参考】注释掉的代码尽量要配合说明，而不是简单的注释掉。

说明：代码被注释掉有两种可能性： 1）后续会恢复此段代码逻辑。 2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码仓库保存了历史代码）。

5.10. 【参考】对于注释的要求：

第一、能够准确反应设计思想和代码逻辑； 第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。

5.11. 【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

5.12. 【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描， 经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。

1） 待办事宜（TODO）：（标记人，标记时间，[预计处理时间]） 表示需要实现，但目前还未实现的功能。这实际上是一个标签，目前还没有实现，但已经被广泛使用。只能应用于类，接口和方法。 2） 错误，不能工作（FIXME）：（标记人，标记时间，[预计处理时间]） 在注释中用 FIXME 标记某代码是错误的，而且不能工作，需要及时纠正的情况。

5.13. 【推荐】每个公开和私有的重要接口，要有附加的注释说明其作用和适用场景。

6. Cocoa & Objective-C特性

6.1. 【强制】明确 designated initializer

清晰的指明你创建的类的 designated initializer，这一点在其他人创建继承自你的类时显得非常重要。这样的话，他们只需要重写某一个或几个初始化器就可以保证他们的子类初始化方法被调用。这也可以帮助后面可能调试代码的人，更清晰的了解你的类初始化流程。请使用注释或者`NS_DESIGNATED_INITIALIZER`宏指明你的 designated initializer。如果使用`NS_DESIGNATED_INITIALIZER`宏，请将其他不支持的初始化器用`NS_UNAVAILABLE`来标注。

6.2. 【强制】若实现一个类需要init方法，必须重写父类的 designated initializer。

如果不重写父类的 designated initializer，你自己的初始化器可能不会保证在所有情况下都被调用，这会导致bug很诡异，难以被发现。 一个典型的例子是你是否创建了一个`UIViewController`子类重写`initWithNibName:bundle:`

```
// GOOD:

@implementation ZOCViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle
*)nibBundleOrNil {
    // call to the superclass designated initializer
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization （自定义的初始化过程）
    }
    return self;
}

@end
```

在`UIViewController`子类的例子里面如果重写`init`会是一个错误，这个情况下调用者会尝试调用`initWithNib:bundle`初始化你的类，你的类实现不会被调用。这同样违背了它应该是合法调用任何 designated initializer 的规则。 在你希望提供你自己的初始化函数的时候，你应该遵守这三个步骤来保证获得正确的行为：

1. 定义你的 designated initializer，确保调用了直接父类的 designated initializer。
2. 重写直接父类的 designated initializer。调用你的新的 designated initializer。
3. 为新的 designated initializer 写文档。

6.3. 【强制】不必在 init 方法中初始化实例变量为 0 或 nil ，这是多余的!

所有新创建的实例变量都会被初始化为0（isa除外），所以不必散乱的在init初始化方法中将实例变量初始化为 0 或 nil。

6.4. 【强制】不要使用 +new

不可调用NSObject的类方法new或在子类中重写该方法。+new方法很少使用，与初始化器的使用有很大的不同。相反，一般使用+alloc和-init方法来创建和初始化实例对象。

6.5. 【强制】公开API保持简单

保持类简单,如果一个方法不需要作为公开方法，请不要暴露在公开接口中。和C++不同，Objective-C在公开方法和私有方法之间并没有什么太多区分，任何消息都可能被发送给对象。因此，只有使用者需要调用某一个方法时，才把这个方法放置在公开的API中。这样可以降低一些你不希望外界调用的方法被调用的可能性，这也包括重写的父类方法。

6.6. 【推荐】通常私有方法需要有一个较为独特的名字，这样可以防止子类无意的重写这些方法。

因为内部方法并不是真正的私有方法，也容易碰巧重写一个父类的私有方法，这样会导致很难排除由此引起的bug。

6.7. 【强制】使用系统库Umbrella头文件

引用系统库的Umbrella头文件，而不是引用某几个单独的头文件。（关于umbrella头文件，更像是一个库中专门暴露出来，供外界使用者去引用的头文件，而不是引用库中的某些头文件）

人们似乎更倾向引用Cocoa或Foundation系统库中单独的头文件，这样看起来引用更简单，但事实上，引用系统库的顶层头文件会让编译器做更少的工作。根框架通常会预处理，从而加载速度更快。另外，对于Objective-C框架，记住使用@import和#import，而不使用 #include 。

```
// GOOD:

@import UIKit;      // GOOD.
#import <Foundation/Foundation.h>      // GOOD.
```

```
// AVOID:

#import <Foundation/NSArray.h>         // AVOID.
#import <Foundation/NSString.h>
...
```

6.8. 【强制】避免在初始化器或-dealloc中给当前对象发消息

初始化器或 -dealloc方法中，避免调用该类对象的实例方法。 父类的初始化方法先于子类初始化方法执行。在所有类都被初始化，使该对象的所有属性被初始化之前调用该类的示例方法，都可能导致子类直接操作未初始化的属性。 -dealloc方法中也存在类似问题，例如在-dealloc调用一些实例方法，这些方法操作的某些属性，可能早已被释放掉了。 属性访问是一种不太明显的案例。属性访问器可以像其他selector一样被重写。一旦切实可行，直接在初始化器和-dealloc方法中对成员变量进行赋值和释放，而不是通过属性访问器来完成。

```
// GOOD:

- (instancetype)init {
  self = [super init];
  if (self) {
    _bar = 23;  // GOOD.
  }
  return self;
}
```

6.9. 【推荐】拆解指定初始化方法码为工具方法要谨慎!

工具方法可能会有意或无意的被子类重写，或者碰巧产生了命名冲突； 当编辑一个工具方法时，可能不会知晓这段代码是被初始化方法调用的；

```
// AVOID:

- (instancetype)init {
  self = [super init];
  if (self) {
    self.bar = 23;  // AVOID.
    [self sharedMethod];  // AVOID. Fragile to subclassing or future
extension.
  }
  return self;
}
```

```
// GOOD:

- (void)dealloc {
  [_notifier removeObserver:self];  // GOOD.
}
```

```
// AVOID:

- (void)dealloc {
  [self removeNotifications];  // AVOID.
}
```

6.10. 【强制】Setter方法中NSString、NSURLRequest或者容器要用Copy

Setter方法中对传入的NSString参数必须copy一份再使用，同样，对于容器类参数，例如 NSArray、NSDictionary，也需要copy再使用。

永远不要只考虑是string类型，也可能（传入的）是NSMutableString类型。这可以避免在你不知情的情况下，调用方修改该值。

持有容器对象参数也需要考虑该对象可能也是可变的，因此，使用该容器参数的复制对象，相对更安全。

```
// GOOD:

@property (nonatomic, copy) NSString *name;

- (void)setZigfoos:(NSArray<Zigfoo *> *)zigfoos {
  // Ensure that we're holding an immutable collection.
  _zigfoos = [zigfoos copy];
}
```

6.11. 【强制】用泛型标识元素类型

在Xcode7或更新版本编译的工程，应该使用Objective-C轻量级泛型符号来标识其包含的对象类别。NSArray, NSDictionary, 或 NSSet对象都需要轻量级泛型符号标明，这可以改善类型安全以及明确文档使用说明。

```
// GOOD:

@property(nonatomic, copy) NSArray<Location *> *locations;
@property(nonatomic, copy, readonly) NSSet<NSString *> *identifiers;

NSMutableArray<MyLocation *> *mutableLocations = [otherObject.locations
mutableCopy];
```

若要标识的泛型比较复杂，可以考虑使用typedef定义新类型，以保持可读性。

```
// GOOD:

typedef NSSet<NSDictionary<NSString *, NSDate *> *> TimeZoneMappingSet;
TimeZoneMappingSet *timeZoneMappings = [TimeZoneMappingSet
setWithObjects:...];
```

使用最具描述性的常用父类或协议。通常情况下，当不知道容器中对象类型时，使用id显式声明。

```
// GOOD:

@property(nonatomic, copy) NSArray<id> *unknowns;
```

6.12. 【推荐】如果指针检查只是为了防止给nil对象发消息，那这部分检查不必添加，给nil发消息的返回值是可靠的。

```
// AVOID:

if (dataSource) {  // AVOID.
  [dataSource moveItemAtIndex:1 toIndex:0];
}
// GOOD:

[dataSource moveItemAtIndex:1 toIndex:0];  // GOOD.
```

注意：这适用于 nil 作为消息接受者，而不是作为参数值。

个别方法可能会，也可能不会安全处理nil参数值。

也请注意这区别于检查C/C++指针和block指针是否为NULL, 这些运行时不会处理，这些会导致应用崩溃。你仍然需要检查指针不能为NULL。

6.13. 【推荐】可空性（Nullability）

Interface中可以使用可空性注解来描述接口行为和如何使用。使用可控性作用域（例如NS_ASSUME_NONNULL_BEGIN 和 NS_ASSUME_NONNULL_END）或使用可控性注解，这两种方式都是可以的。相对 **nullable 和** nonnull 关键字，更推荐使用 _Nullable和 _Nonnull。对于Objective-C方法和属性，推荐使用无下划线的关键字，例如nonnull 和 nullable。

```
// GOOD:

/** A class representing an owned book. */
@interface DZBook : NSObject

/** The title of the book. */
@property (readonly, copy, nonnull) NSString *title;

/** The author of the book, if one exists. */
@property (readonly, copy, nullable) NSString *author;

/** The owner of the book. Setting nil resets to the default owner. */
@property (copy, null_resettable) NSString *owner;

/** Initializes a book with a title and an optional author. */
- (nonnull instancetype)initWithTitle:(nonnull NSString *)title
                               author:(nullable NSString *)author
    NS_DESIGNATED_INITIALIZER;

/** Returns nil because a book is expected to have a title. */
- (nullable instancetype)init;

@end

/** Loads books from the file specified by the given path. */
NSArray<DZBook *> *_Nullable DZLoadBooksFromFile(NSString *_Nonnull path);
NSArray<DZBook *> *__nullable DZLoadBooksFromTitle(NSString *__nonnull path);
```

```
// AVOID:

NSArray<DZBook *> *__nullable DZLoadBooksFromTitle(NSString *__nonnull path);
```

6.14. 【强制】注意BOOL陷阱

将整型数值转换为BOOL要小心。避免直接和 YES直接进行比较。

在OSX和32位iOS系统版本中，BOOL 类型被定义为有符号char类型，所以，它可以是除了YES (1) and NO (0)之外的其他值。切勿将整数值直接赋值或强制类型转换后赋值给BOOL值。

常见错误包括强转或转换数组大小、指针值或者通过逻辑位运算结果赋值给BOOL变量，根据整数最后一个字节的值，结果仍然有可能会产生NO值。在将一个整数转换为BOOL值时，需要使用三元运算符来保证返回值是YES 或 NO。

也可以使用逻辑运算符(&&、 ||、 !)来获取 BOOL 值，这种方法也可以不使用三元运算符来安全转换 BOOL 值。

```
// AVOID:

- (BOOL)isBold {
  return [self fontTraits] & NSFontBoldTrait;  // AVOID.
}
- (BOOL)isValid {
  return [self stringValue];  // AVOID.
}
// GOOD:

- (BOOL)isBold {
  return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}
- (BOOL)isValid {
  return [self stringValue] != nil;
}
- (BOOL)isEnabled {
  return [self isValid] && [self isBold];
}
```

同样，不要将 BOOL 值和YES进行直接比较。这样做不仅仅是因为对于精通C语言的人可能难以理解，而是返回值可能并不总是和预期一致（容易引起逻辑错误），这才是真正原因。

```
// AVOID:

BOOL great = [foo isGreat];
if (great == YES) {  // AVOID.
  // ...be great!
}
```

```
// GOOD:

BOOL great = [foo isGreat];
if (great) {         // GOOD.
  // ...be great!
}
```

6.15. 【强制】单例对象应该使用线程安全的模式创建共享的实例，这将会预防有时可能产生的许多崩溃。

```
// GOOD:

+ (instancetype)sharedInstance {
    static id sharedInstance = nil;

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });

    return sharedInstance;
}
```

6.16. 【强制】注意`self`的循环引用，当使用代码块和异步分发的时候，要注意避免引用循环。总是使用`weak`来引用对象，避免引用循环。

单个语句的例子:

```
// GOOD:

__weak __typeof(self) weakSelf = self;
[self executeBlock:^(NSData *data, NSError *error) {
    [weakSelf doSomethingWithData:data];
}];
```

```
// AVOID:

[self executeBlock:^(NSData *data, NSError *error) {
    [self doSomethingWithData:data];
}];
```

多个语句的例子:

```
// GOOD:

__weak __typeof(self)weakSelf = self;
[self executeBlock:^(NSData *data, NSError *error) {
    __strong __typeof(weakSelf) strongSelf = weakSelf;
    if (strongSelf) {
        [strongSelf doSomethingWithData:data];
        [strongSelf doSomethingWithData:data];
    }
}];
```

```
// AVOID:

__weak __typeof(self)weakSelf = self;
[self executeBlock:^(NSData *data, NSError *error) {
    [weakSelf doSomethingWithData:data];
    [weakSelf doSomethingWithData:data];
}];
```

你应该把这两行代码作为 snippet 加到 Xcode 里面并且总是这样使用它们。

```
__weak __typeof(self)weakSelf = self;
__strong __typeof(weakSelf)strongSelf = weakSelf;
```

7. 空间布局和格式

7.1. 【强制】缩进时不使用制表符，只用4个空格

7.2. 【强制】Objective-C最大行宽为120个字符。在XCode中，通过设置Preferences > Text Editing > Page guide at column:为120，可以轻松检查超宽地方

7.3. 【强制】方法声明与定义中，- or + 后与返回值之间须有一个空格，参数名与参数类型间无空格，参数列表不同参数间有一个空格，指针定义的星号前有一个空格

方法示例如下：

```
// GOOD:

- (void)doSomethingWithString:(NSString *)theString {
  ...
}
```

7.4. 【推荐】如果一个方法声明一行放不下，需要将不同参数各占一行

除首行外，其余所有行都要至少缩进4个空格。不同参数前的冒号要对齐。如果方法声明第一行中参数前的冒号所在位置，对齐所有冒号后，后面的参数前面缩进的空格数少于4个，这种情况只需要将函数声明中第一行之外的其他行冒号对齐即可

```
// GOOD:

- (void)doSomethingWithFoo:(DZFoo *)theFoo
                      rect:(NSRect)theRect
                  interval:(float)theInterval {
  ...
}

- (void)shortKeyword:(DZFoo *)theFoo
```

```
            longerKeyword:(NSRect)theRect
    someEvenLongerKeyword:(float)theInterval
                    error:(NSError **)theError {
  ...
}
```

7.5. 【强制】if, while, for, switch后有一个空格，比较运算符两边有空格，语句中必须使用大括号，即使只有一行代码

```
// GOOD:

for (int i = 0; i < 5; ++i) {
}

// AVOID:

if (hasSillyName)
  LaughOutLoud();

for (int i = 0; i < 10; i++)
  BlowTheHorn();
```

7.6. 【强制】若 if 语句后有 else 语句，两部分都需要使用大括号。

```
// GOOD:

if (hasBaz) {
    foo();
} else {
    bar();
}

// AVOID:

if (hasBaz) foo();
else bar();         // AVOID.

if (hasBaz) {
  foo();
} else bar();       // AVOID.
```

7.7. 【强制】除非两个case语句之间没有其他代码，有意添加的case连续执行情况，需要增加注释说明，在一个 switch 块内，都必须包含一个 default 语句并且放在 最后，即使它什么代码也没有。

```
// GOOD:

switch (i) {
```

```
    case 1:
        ...
        break;
    case 2:
        j++;
        // Falls through.
    case 3: {
        int k;
        ...
        break;
    }
    case 4:
    case 5:
    case 6:
        break;
    default:
        break;
}
```

7.8. 【强制】二进制运算符、赋值运算符左右两边都需要添加空格。一元运算符可以省略空格。圆括号（左括号右边、右括号左边）不用空格。

```
// GOOD:

x = 0;
v = w * x + y / z;
v = -y * (x + z);
```

7.9. 【推荐】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。

```
// GOOD:

result = a > b ? x : y;
```

```
// AVOID:

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

7.10. 【强制】方法调用的格式应与方法声明一致。

若源码中有现有的编码规范使用惯例，请保持一致继续使用。方法调用的所有参数都放在一行。

```
// GOOD:

[myObject doFooWith:arg1 name:arg2 error:arg3];
```

或每个参数放在单独一行，冒号对齐。

```
// GOOD:

[myObject doFooWith:arg1
               name:arg2
              error:arg3];
```

不要使用下列编码风格：

```
// AVOID:

[myObject doFooWith:arg1 name:arg2  // some lines with >1 arg
              error:arg3];

[myObject doFooWith:arg1
               name:arg2 error:arg3];

[myObject doFooWith:arg1
          name:arg2  // aligning keywords instead of colons
          error:arg3];
```

与声明和定义一样，当第一个关键字比其他字段短，将后面的参数缩进至少4个空格，并将冒号对齐。

```
// GOOD:

[myObj short:arg1
          longKeyword:arg2
    evenLongerKeyword:arg3
                error:arg4];
```

方法调用包含多个内嵌block时，将这些参数缩进4个空格，并左对齐。

7.11. 【推荐】推荐小而功能专注的函数，长方法或者函数某些情况下也是可以的，所以关于函数长度没有固定的限制。如果一个函数长度超过了40行，请考虑如何在不破坏代码结构的基础上，将它拆分。

即便你现在的长方法可以完美运行，可能几个月后的某些修改会给他添加新特性。这也导致出现问题难以发现。保持函数简练可以使代码易读，利于修改。

7.12. 【推荐】谨慎使用空白换行，为了让更多代码能够一屏展示，在函数大括号内，避免使用空行。不同函数间或不同代码逻辑组之间，空行限制在1-2行。

7.13. 【强制】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

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）。

```
// GOOD:

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

```
// AVOID:

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

7.14. 【推荐】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。

```
// GOOD:

@interface NYTSection : NSObject

@property (nonatomic) NSString *headline;

@end
```

```
// AVOID:

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

7.15. 【推荐】私有属性应该声明在类实现文件的延展（匿名的类目）中。

```
// GOOD:

@interface NYTAdvertisement ()

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

@end
```

7.16. 【推荐】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。