留白和格式

空格 (Spaces) vs. 制表符（Tabs）

> **Tip**
>
> 只使用空格，一个Tab用4个Space代替。

应该将编辑器设置成自动将制表符替换成空格。

通过设置*Xcode > Preferences > Text Editing > Indentation*。

行宽

尽量让你的代码保持在 120 列之内。

通过设置 *Xcode > Preferences > Text Editing > Show page guide*。

方法声明和定义

> **Tip**
>
> - 在-(+) 和返回类型之间须使用一个空格，参数列表中只有参数之间可以有空格。

方法示例：

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

```
- (void)draw:(id)obj at:(Point *)where{
  ...
}
```

(指针*前的空格是可选的，使用自己一致的风格)

如果一行有非常多的参数，已经超过了单行的列限制，更好的方式是将每个参数单独拆成一行。如果使用多行，将每个参数前的冒号对齐。

```
- (void)doSomethingWith:(GTMFoo *)theFoo
                   rect:(NSRect)theRect
```

```
                interval:(float)theInterval {
  ...
}
```

当第一个关键字比其它的短时，保证下一行至少有 Tab的缩进。这样可以使关键字垂直对齐，而不是使用冒号对齐：

```
- (void)short:(GTMFoo *)theFoo
    longKeyword:(NSRect)theRect
    evenLongerKeyword:(float)theInterval {
  ...
}
```

方法调用

> **Tip**
>
> 方法调用应尽量保持与方法声明的格式一致。当格式的风格有多种选择时，新的代码要与已有代码保持一致。

调用时所有参数应该在同一行：

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

或者每行一个参数，以冒号对齐：

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

不要使用下面的缩进风格：

```
[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];
```

方法定义与方法声明一样，当关键字的长度不足以以冒号对齐时，下一行都要以四个空格进行缩进。

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

@public 和 @private

> **Tip**
>
> `@public` 和 `@private` 访问修饰符不需要缩进。

与 C++ 中的 `public, private` 以及 `protected` 非常相似。

```
@interface MyClass : NSObject {
@public
     ...
@private
     ...
}
@end
```

协议名

> **Tip**
>
> 类型标识符和尖括号内的协议名之间，不能有任何空格。

这条规则适用于类声明、实例变量以及方法声明。例如：

```
@interface MyProtocoledClass : NSObject<NSWindowDelegate> {
@private
     id<MyFancyDelegate> _delegate;
}
- (void)setDelegate:(id<MyFancyDelegate>)delegate;
@end
```

块（Blocks）

> **Tip**
>
> 块（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。
- 块内允许按两个空格缩进，但前提是和项目的其它代码保持一致的缩进风格。

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

// The block can be put on a new line, indented four spaces, with the
// closing brace aligned with the first character of the line on which
// block was declared.
[operation setCompletionBlock:^{
    [self.delegate newDataAvailable];
}];

// Using a block with a C API follows the same alignment and spacing
// rules as with Objective-C.
dispatch_async(fileIOQueue_, ^{
    NSString* path = [self sessionFilePath];
    if (path) {
      // ...
    }
});

// An example where the parameter wraps and the block declaration fits
// on the same line. Note the spacing of |^(SessionWindow *window) {|
// compared to |^{| above.
[[SessionService sharedService]
    loadWindowWithCompletionBlock:^(SessionWindow *window) {
        if (window) {
          [self windowDidLoad:window];
        } else {
          [self errorLoadingWindow];
        }
    }];

// An example where the parameter wraps and the block declaration does
// not fit on the same line as the name.
[[SessionService sharedService]
    loadWindowWithCompletionBlock:
        ^(SessionWindow *window) {
            if (window) {
              [self windowDidLoad:window];
            } else {
              [self errorLoadingWindow];
            }
        }];

// Large blocks can be declared out-of-line.
void (^largeBlock)(void) = ^{
    // ...
};
```

```
[operationQueue_ addOperationWithBlock:largeBlock];
```

命名

文件名

> **Tip**
>
> 文件名须反映出其实现了什么类 – 包括大小写。遵循你所参与项目的约定。

文件的扩展名应该如下：

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

类别的文件名应该包含被扩展的类名，如：`GTMNSString+Utils.h` 或 ``GTMNSTextView+Autocomplete.h``。
类文件应该和类名一致。

类名

> **Tip**
>
> 类名（以及类别、协议名）应首字母大写，并以驼峰格式分割单词。

应用层 的代码，应该尽量避免不必要的前缀。为每个类都添加相同的前缀无助于可读性。当编写的代码期望在不同应用程序间复用时，应使用前缀（如：`GTMSendMessage`）。

类别名(Category Names)

Tip

类别名应该有两三个字母的前缀以表示类别是项目的一部分或者该类别是通用的。类别名应该包含它所扩展的类的名字。

比如我们要基于 `NSString` 创建一个用于解析的类别，我们将把类别放在一个名为 `GTMNSString+Parsing.h` 的文件中。类别本身命名为`GTMStringParsingAdditions`（这个文件中可能存在多个不同的与解析有关类别，因此文件中扩展名字和Category名字不一样，通常情况下扩展名应该和Category一样）。

类名与包含类别名的括号之间，应该以一个空格分隔。

```
@interface NSString (GTMStringParsingAdditions)
- (NSString *)gtmFoobarString;
@end
```

Objective-C 方法名

Tip

方法名应该以小写字母开头，并混合驼峰格式。每个具名参数也应该以小写字母开头。

方法名应尽量读起来就像句子，这表示你应该选择与方法名连在一起读起来通顺的参数名。（例如，`convertPoint:fromRect:` 或`replaceCharactersInRange:withString:`）。详情参见 **Apple's Guide to Naming Methods**。

访问器方法应该与他们 要获取的 成员变量的名字一样，但不应该以get作为前缀。例如：

```
- (id)getDelegate;  // AVOID
- (id)delegate;     // GOOD
```

变量名

Tip

变量名应该以小写字母开头，并使用驼峰格式。类的成员变量应该以下划线作为前缀。例如：`myLocalVariable`、`_myInstanceVariable`。

普通变量名

对于静态的属性（`int` 或指针），不要使用匈牙利命名法。尽量为变量起一个描述性的名字。不要担心浪费列宽，因为让新的代码阅读者立即理解你的代码更重要。例如：

- 错误的命名：

```
int w;
int nerr;
int nCompConns;
tix = [[NSMutableArray alloc] init];
obj = [someObject object];
p = [network port];
```

- 正确的命名：

```
int numErrors;
int numCompletedConnections;
tickets = [[NSMutableArray alloc] init];
userInfo = [someObject object];
port = [network port];
```

实例变量

实例变量应该混合大小写，并以下划线作为前缀，如 `_userNameTextField`。

常量

常量名应该以小写字母 `k` 开头，使用驼峰格式分隔单词，如：`kInvalidHandle`，`kWritePerm`。

宏

宏使用全大写命名，如：`SCREEN_WIDTH, SCREEN_HEIGHT, DECLARE_SINGLETON(className)`。

注释

声明部分的注释

> **Tip**
>
> 每个接口、类别以及协议应辅以注释，以描述它的目的及与整个项目的关系。

```
// A delegate for NSApplication to handle notifications about app
// launch and shutdown. Owned by the main app controller.
@interface MyAppDelegate : NSObject {
  ...
}
@end
```

当一个类的作用或者与其他类之间的关系比较负载时，应当添加注释予以说明。对于接口方法根据功能的复杂性进行说明，包括参数和返回值，以及一些注意事项。

Cocoa 和 Objective-C 特性

初始化

> **Tip**
>
> 不要在 init 方法中，将成员变量初始化为 `0` 或者 `nil`；毫无必要。

刚分配的对象，默认值都是 0，除了 `isa` 指针（译者注：`NSObject` 的 `isa` 指针，用于标识对象的类型）。所以不要在初始化器里面写一堆将成员初始化为`0` 或者 `nil` 的代码。

避免 +new

Tip

不要调用 `NSObject` 类方法 `new`，也不要在子类中重载它。使用 `alloc` 和 `init` 方法创建并初始化对象。

现代的 Ojbective-C 代码通过调用 `alloc` 和 `init` 方法来创建并 retain 一个对象。由于类方法 `new` 很少使用，这使得有关内存分配的代码审查更困难。

使用根框架

Tip

`#import` 根框架而不是单独的零散文件

当你试图从框架（如 Cocoa 或者 Foundation）中包含若干零散的系统头文件时，实际上包含顶层根框架的话，编译器要做的工作更少。根框架通常已经经过预编译，加载更快。另外记得使用 `#import` 而不是 `#include` 来包含 Objective-C 的框架。

```
#import <Foundation/Foundation.h>      // good

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

构建时即设定 `autorelease`

Tip

当创建临时对象时，在同一行使用 `autolease`，而不是在同一个方法的后面语句中使用一个单独的 `release`。

尽管运行效率会差一点，但避免了意外删除 `release` 或者插入 `return` 语句而导致内存泄露的可能。例如：

```
// AVOID (unless you have a compelling performance reason)
MyController* controller = [[MyController alloc] init];
// ... code here that might return ...
[controller release];

// BETTER
MyController* controller = [[[MyController alloc] init] autorelease];
```

autorelease 优先 retain 其次

> **Tip**
>
> 给对象赋值时遵守 `autorelease``之后 ``retain` 的模式。

当给一个变量赋值新的对象时，必须先释放掉旧的对象以避免内存泄露。有很多“正确的”方法可以处理这种情况。我们则选择“`autorelease` 之后 `retain`”的方法，因为事实证明它不容易出错。注意大的循环会填满 `autorelease` 池，并且可能效率上会差一点，但权衡之下我们认为是可以接受的。

```
- (void)setFoo:(GMFoo *)aFoo {
  [foo_ autorelease];  // Won't dealloc if |foo_| == |aFoo|
  foo_ = [aFoo retain];
}
```

按声明顺序销毁实例变量

> **Tip**
>
> `dealloc` 中实例变量被释放的顺序应该与它们在 `@interface` 中声明的顺序一致，这有助于代码审查。

代码审查者在评审新的或者修改过的 `dealloc` 实现时，需要保证每个 `retained` 的实例变量都得到了释放。

为了简化 `dealloc` 的审查，`retained` 实例变量被释放的顺序应该与他们在 `@interface` 中声明的顺序一致。如果 `dealloc` 调用了其它方法释放成员变量，添加注释解释这些方法释放了哪些实例变量。

nil 检查

> **Tip**
>
> `nil` 检查只用在逻辑流程中。

使用 `nil` 的检查应该侧重在逻辑，而不是程序是否会崩溃。Objective-C 运行时会处理向 `nil` 对象发送消息的情况(不做任何处理)。如果方法没有返回值，就没关系。如果有返回值，可能由于运行时架构、返回值类型以及 OS X 版本的不同而不同(根据XCode5.0文档：对象类型为nil,数值类型为0,BOOL类型为NO, struct类型的所有成员初始化为0)

BOOL 若干陷阱

> **Tip**
>
> 将普通整形转换成 `BOOL` 时要小心。不要直接将 `BOOL` 值与 `YES` 进行比较。

Ojbective-C 中把 `BOOL` 定义成无符号字符型，这意味着 `BOOL` 类型的值远不止 `YES``(1)`或 ``NO``(0)`。不要直接把整形转换成 ``BOOL`。常见的错误包括将数组的大小、指针值及位运算的结果直接转换成 `BOOL`，取决于整型结果的最后一个字节，很可能会产生一个 `NO` 值。当转换整形至 `BOOL` 时，使用三目操作符来返回 `YES` 或者 `NO`。（译者注：读者可以试一下任意的 256 的整数的转换结果，如 256、512 …）

你可以安全在 `BOOL`、`_Bool` 以及 `bool` 之间转换（参见 C++ Std 4.7.4, 4.12 以及 C99 Std 6.3.1.2）。你不能安全在 `BOOL` 以及 `Boolean` 之间转换，因此请把`Boolean` 当作一个普通整形，就像之前讨论的那样。但 Objective-C 的方法标识符中，只使用 `BOOL`。

对 `BOOL` 使用逻辑运算符（`&&`，`||` 和 `!`）是合法的，返回值也可以安全地转换成 `BOOL`，不需要使用三

目操作符。

错误的用法：

```
- (BOOL)isBold {
  return [self fontTraits] & NSFontBoldTrait;
}
- (BOOL)isValid {
  return [self stringValue];
}
```

正确的用法：

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

同样，不要直接比较 `YES/NO` 和 `BOOL` 变量。不仅仅因为影响可读性，更重要的是结果可能与你想的不同。

错误的用法：

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

正确的用法：

```
BOOL great = [foo isGreat];
```

```
if (great)
  // ...be great!
```

属性（Property）

> **Tip**
>
> 属性（Property）通常允许使用，但需要清楚的了解：属性（Property）是 Objective-C 2.0 的特性，会限制你的代码只能跑在 iPhone 和 Mac OS X 10.5 (Leopard) 及更高版本上。点引用只允许访问声明过的 `@property`。

命名

属性所关联的实例变量名必须以下划线作为前缀。属性的名字应该与成员变量去掉下划线的名字一模一样。

使用 `@synthesize` 指示符来正确地重命名属性。

```
@interface MyClass : NSObject {
@private
  NSString * _name;
}
@property(copy, nonatomic) NSString *name;
@end

@implementation MyClass
@synthesize name = _name;
@end
```

位置

属性的声明必须紧靠着类接口中的实例变量语句块。属性的定义必须在 `@implementation` 的类定义的最上方。他们的缩进与包含他们的 `@interface` 以及`@implementation` 语句一样。

```
@interface MyClass : NSObject {
@private
  NSString * _name;
}
@property(copy, nonatomic) NSString *name;
@end

@implementation MyClass
@synthesize name = _name;
- (id)init {
...
}
@end
```

字符串应使用 copy 属性（Attribute）

应总是用 `copy` 属性（attribute）声明 `NSString` 属性（property）。

从逻辑上，确保遵守 `NSString` 的 `setter` 必须使用 `copy` 而不是 `retain` 的原则。

原子性

一定要注意属性（property）的开销。缺省情况下，所有 `synthesize` 的 `setter` 和 `getter` 都是原子的。这会给每个 `get` 或者 `set` 带来一定的同步开销。将属性（property）声明为 `nonatomic`，除非你需要原子性。