Google Objective-C风格指

极客学院出版

前言

Objective-C 是 C 语言的扩展，增加了动态类型和面对对象的特性。它被设计成具有易读易用的，支持复杂的面向对象设计的编程语言。它是 Mac OS X 以及 iPhone 的主要开发语言。

Cocoa 是 Mac OS X 上主要的应用程序框架之一。它由一组 Objective-C 类组成，为快速开发出功能齐全的 Mac OS X 应用程序提供支持。

苹果公司已经有一份非常全面的 Objective-C 编码指南。Google 为 C++ 也写了一份类似的编码指南。而这份 Objective-C 指南则是苹果和 Google 常规建议的最佳结合。因此，在阅读本指南之前，请确定你已经阅读过：

- Apple’s Cocoa Coding Guidelines <http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/index.html> _
- Google’s Open Source C++ Style Guide <http://codinn.com/projects/google-cpp-styleguide/> _

Note
所有在 Google 的 C++ 风格指南中所禁止的事情，如未明确说明，也同样不能在Objective-C++ 中使用。

本文档的目的在于为所有的 Mac OS X 的代码提供编码指南及实践。许多准则是在实际的项目和小组中经过长期的演化、验证的。Google 开发的开源项目遵从本指南的要求。

Google 已经发布了遵守本指南开源代码，它们属于 Google Toolbox for Mac project <http://code.google.com/p/google-toolbox-for-mac/> _ 项目（本文以缩写 GTM 指代）。GTM 代码库中的代码通常为了可以在不同项目中复用。

注意，本指南不是 Objective-C 教程。我们假定读者对 Objective-C 非常熟悉。如果你刚刚接触 Objective-C 或者需要温习，请阅读 The Objective-C Programming Language <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html> _ 。

例子

都说一个例子顶上一千句话，我们就从一个例子开始，来感受一下编码的风格、留白以及命名等等。

一个头文件的例子，展示了在 @interface 声明中如何进行正确的注释以及留白。

```
// Foo.h
// AwesomeProject
//
```

```
//  Created by Greg Miller on 6/13/08.
//  Copyright 2008 Google, Inc. All rights reserved.
//

#import <Foundation/Foundation.h>

// A sample class demonstrating good Objective-C style. All interfaces,
// categories, and protocols (read: all top-level declarations in a header)
// MUST be commented. Comments must also be adjacent to the object they're
// documenting.
//
// (no blank line between this comment and the interface)
@interface Foo : NSObject {
 @private
  NSString *bar_;
  NSString *bam_;
}

// Returns an autoreleased instance of Foo. See -initWithBar: for details
// about |bar|.
+ (id)fooWithBar:(NSString *)bar;

// Designated initializer. |bar| is a thing that represents a thing that
// does a thing.
- (id)initWithBar:(NSString *)bar;

// Gets and sets |bar_|.
- (NSString *)bar;
- (void)setBar:(NSString *)bar;

// Does some work with |blah| and returns YES if the work was completed
// successfully, and NO otherwise.
- (BOOL)doWorkWithBlah:(NSString *)blah;

@end
```

一个源文件的例子，展示了 `@implementation` 部分如何进行正确的注释、留白。同时也包括了基于引用实现的一些重要方法，如 `getters` 、 `setters` 、 `init` 以及 `dealloc` 。

```
//
//  Foo.m
//  AwesomeProject
//
//  Created by Greg Miller on 6/13/08.
//  Copyright 2008 Google, Inc. All rights reserved.
```

```
//

#import "Foo.h"


@implementation Foo

+ (id)fooWithBar:(NSString *)bar {
  return [[[self alloc] initWithBar:bar] autorelease];
}

// Must always override super's designated initializer.
- (id)init {
  return [self initWithBar:nil];
}

- (id)initWithBar:(NSString *)bar {
  if ((self = [super init])) {
    bar_ = [bar copy];
    bam_ = [[NSString alloc] initWithFormat:@"hi %d", 3];
  }
  return self;
}

- (void)dealloc {
  [bar_ release];
  [bam_ release];
  [super dealloc];
}

- (NSString *)bar {
  return bar_;
}

- (void)setBar:(NSString *)bar {
  [bar_ autorelease];
  bar_ = [bar copy];
}

- (BOOL)doWorkWithBlah:(NSString *)blah {
  // ...
  return NO;
}

@end
```

不要求在 `@interface` 、 `@implementation` 和 `@end` 前后空行。如果你在 `@interface` 声明了实例变量，则须在关括号 `}` 之后空一行。

除非接口和实现非常短，比如少量的私有方法或桥接类，空行方有助于可读性。

> 致谢
>
> 内容撰写：http://zh-google-styleguide.readthedocs.org/

更新日期	更新内容
2015-05-26	Objective-C 风格指南

目录

1

留白和格式

空格 vs. 制表符

Tip
只使用空格，且一次缩进两个空格。

我们使用空格缩进。不要在代码中使用制表符。你应该将编辑器设置成自动将制表符替换成空格。

行宽

尽量让你的代码保持在 80 列之内。

我们深知 Objective-C 是一门繁冗的语言，在某些情况下略超 80 列可能有助于提高可读性，但这也只能是特例而已，不能成为开脱。

如果阅读代码的人认为把把某行行宽保持在 80 列仍然有不失可读性，你应该按他们说的去做。

我们意识到这条规则是有争议的，但很多已经存在的代码坚持了本规则，我们觉得保证一致性更重要。

通过设置 -Xcode > Preferences > Text Editing > Show page guide-，来使越界更容易被发现。

方法声明和定义

Tip
/ + 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。

方法应该像这样：

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

星号前的空格是可选的。当写新的代码时，要与先前代码保持一致。

如果一行有非常多的参数，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 4 个空格的缩进。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`` 访问修饰符应该以一个空格缩进。

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

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

异常

Tip
每个 ``@`` 标签应该有独立的一行，在 ``@`` 与 ``{}`` 之间需要有一个空格， ``@catch`` 与被捕捉到的异常对象的声明之间也要有一

如果你决定使用 Objective-C 的异常，那么就按下面的格式。不过你最好先看看 :ref: 避免抛出异常 <avoid-throwing-exceptions> 了解下为什么不要使用异常。

```
@try {
  foo();
}
@catch (NSException *ex) {
  bar(ex);
}
@finally {
  baz();
}
```

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>)aDelegate;
@end
```

块（闭包）

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];
```

2

命名

对于易维护的代码而言，命名规则非常重要。Objective-C 的方法名往往十分长，但代码块读起来就像散文一样，不需要太多的代码注释。

当编写纯粹的 Objective-C 代码时，我们基本遵守标准的 Objective-C naming rules <http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html> _，这些命名规则可能与 C++ 风格指南中的大相径庭。例如，Google 的 C++ 风格指南中推荐使用下划线分隔的单词作为变量名，而(苹果的)风格指南则使用驼峰命名法，这在 Objective-C 社区中非常普遍。

任何的类、类别、方法以及变量的名字中都使用全大写的 首字母缩写 <http://en.wikipedia.org/wiki/Initialism> _。这遵守了苹果的标准命名方式，如 URL、TIFF 以及 EXIF。

当编写 Objective-C++ 代码时，事情就不这么简单了。许多项目需要实现跨平台的 C++ API，并混合一些 Objective-C、Cocoa 代码，或者直接以 C++ 为后端，前端用本地 Cocoa 代码。这就导致了两种命名方式直接不统一。

我们的解决方案是：编码风格取决于方法/函数以哪种语言实现。如果在一个 @implementation 语句中，就使用 Objective-C 的风格。如果实现一个 C++ 的类，就使用 C++ 的风格。这样避免了一个函数里面实例变量和局部变量命名规则混乱，严重影响可读性。

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 。

Objective-C++

Tip
源代码文件内，Ojbective-C++ 代码遵循你正在实现的函数/方法的风格。

为了最小化 Cocoa/Objective-C 与 C++ 之间命名风格的冲突，根据待实现的函数/方法选择编码风格。实现 `@implementation` 语句块时，使用 Objective-C 的命名规则；如果实现一个 C++ 的类，就使用 C++ 命名规则。

```
// file: cross_platform_header.h

class CrossPlatformAPI {
 public:
  ...
  int DoSomethingPlatformSpecific();  // impl on each platform
 private:
  int an_instance_var_;
};

// file: mac_implementation.mm
#include "cross_platform_header.h"

// A typical Objective-C class, using Objective-C naming.
@interface MyDelegate : NSObject {
 @private
  int instanceVar_;
  CrossPlatformAPI* backEndObject_;
}
- (void)respondToSomething:(id)something;
@end
@implementation MyDelegate
- (void)respondToSomething:(id)something {
  // bridge from Cocoa through our C++ backend
  instanceVar_ = backEndObject->DoSomethingPlatformSpecific();
  NSString* tempString = [NSString stringWithInt:instanceVar_];
  NSLog(@"%@", tempString);
}
@end

// The platform-specific implementation of the C++ class, using
// C++ naming.
int CrossPlatformAPI::DoSomethingPlatformSpecific() {
  NSString* temp_string = [NSString stringWithInt:an_instance_var_];
  NSLog(@"%@", temp_string);
  return [temp_string intValue];
}
```

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

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`（是的，我们知道类别名和文件名不一样，但是这个文件中可能存在多个不同的与解析有关类别）。类别中的方法应该以 `gtm_myCategoryMethodOnAString:` 为前缀以避免命名冲突，因为 Objective-C 只有一个名字空间。如果代码不会分享出去，也不会运行在不同的地址空间中，方法名字就不那么重要了。

类名与包含类别名的括号之间，应该以一个空格分隔。

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 <http://developer.apple.com/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingMethods.html> _`。

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
```

这仅限于 Objective-C 的方法名。C++ 的方法与函数的命名规则应该遵从 C++ 风格指南中的规则。

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后缀。例如：``myLocalVariable``、``myInstance

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，并以下划线作为后缀，如 usernameTextField_ 。然而，如果不能使用 Objective-C 2.0（操作系统版本的限制），并且使用了 KVO/KVC 绑定成员变量时，我们允许例外（译者注： KVO=Key Value Observing，KVC=Key Value Coding ）。这种情况下，可以以一个下划线作为成员变量名字的前缀，这是苹果所接受的键/值命名惯例。如果可以使用 Objective-C 2.0， @property 以及 @synthesize 提供了遵从这一命名规则的解决方案。

常量

常量名（如宏定义、枚举、静态局部变量等）应该以小写字母 `k` 开头，使用驼峰格式分隔单词，如：`kInvalidHandle`，`kWritePerm` 。

3

注释

虽然写起来很痛苦，但注释是保证代码可读性的关键。下面的规则给出了你应该什么时候、在哪进行注释。记住：尽管注释很重要，但最好的代码应该自成文档。与其给类型及变量起一个晦涩难懂的名字，再为它写注释，不如直接起一个有意义的名字。

当你写注释的时候，记得你是在给你的听众写，即下一个需要阅读你所写代码的贡献者。大方一点，下一个读代码的人可能就是你！

记住所有 C++ 风格指南里的规则在这里也同样适用，不同的之处后续会逐步指出。

文件注释

Tip
每个文件的开头以文件内容的简要描述起始，紧接着是作者，最后是版权声明和/或许可证样板。

版权信息及作者

每个文件应该按顺序包括如下项：

- 文件内容的简要描述
- 代码作者
- 版权信息声明（如：`Copyright 2008 Google Inc.`）
- 必要的话，加上许可证样板。为项目选择一个合适的授权样板（例如，`Apache 2.0, BSD, LGPL, GPL`）。

如果你对其他人的原始代码作出重大的修改，请把你自己的名字添加到作者里面。当另外一个代码贡献者对文件有问题时，他需要知道怎么联系你，这十分有用。

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
```

如果你已经在文件头部详细描述了接口，可以直接说明 “完整的描述请参见文件头部”，但是一定要有这部分注释。

另外，公共接口的每个方法，都应该有注释来解释它的作用、参数、返回值以及其它影响。

为类的线程安全性作注释，如果有的话。如果类的实例可以被多个线程访问，记得注释多线程条件下的使用规则。

实现部分的注释

Tip
使用 | 来引用注释中的变量名及符号名而不是使用引号。

这会避免二义性，尤其是当符号是一个常用词汇，这使用语句读起来很糟糕。例如，对于符号 `count` ：

```
// Sometimes we need |count| to be less than zero.
```

或者当引用已经包含引号的符号：

```
// Remember to call |StringWithoutSpaces("foo bar baz")|
```

对象所有权

Tip
当与 Objective-C 最常规的作法不同时，尽量使指针的所有权模型尽量明确。

继承自 `NSObject` 的对象的实例变量指针，通常被假定是强引用关系（retained），某些情况下也可以注释为弱引用（weak）或使用 `__weak` 生命周期限定符。同样，声明的属性如果没有被类 `retained` ，必须指定是弱引用或赋予 `@property` 属性。然而，Mac 软件中标记上 `IBOutlets` 的实例变量，被认为是不会被类 `retained` 的。

当实例变量指向 `CoreFoundation` 、C++ 或者其它非 Objective-C 对象时，不论指针是否会被 `retained` ，都需要使用 `__strong` 和 `__weak` 类型修饰符明确指明。`CoreFoundation` 和其它非 Objective-C 对象指针需要显式的内存管理，即便使用了自动引用计数或垃圾回收机制。当不允许使用 `__weak` 类型修饰符（比如，使用 clang 编译时的 C++ 成员变量），应使用注释替代说明。

注意：Objective-C 对象中的 C++ 对象的自动封装，缺省是不允许的，参见 `这里 <http://chanson.livejournal.com/154253.html>` _ 的说明。

强引用及弱引用声明的例子：

```
@interface MyDelegate : NSObject {
 @private
  IBOutlet NSButton -okButton_;  // normal NSControl; implicitly weak on Mac only

  AnObjcObject- doohickey_;  // my doohickey
  __weak MyObjcParent -parent_;  // so we can send msgs back (owns me)

  // non-NSObject pointers...
  __strong CWackyCPPClass -wacky_;  // some cross-platform object
  __strong CFDictionaryRef -dict_;
}
@property(strong, nonatomic) NSString -doohickey;
@property(weak, nonatomic) NSString -parent;
@end
```

（译注：强引用 - 对象被类 retained 。弱引用 - 对象没有被类 retained ，如委托 ）

4

Cocoa 和 Objective-C 特性

成员变量应该是 @private

Tip
成员变量应该声明为 ``@private``

```
@interface MyClass : NSObject {
 @private
  id myInstanceVariable_;
}
// public accessors, setter takes ownership
- (id)myInstanceVariable;
- (void)setMyInstanceVariable:(id)theVar;
@end
```

明确指定构造函数

Tip
注释并且明确指定你的类的构造函数。

对于需要继承你的类的人来说，明确指定构造函数十分重要。这样他们就可以只重写一个构造函数（可能是几个）来保证他们的子类的构造函数会被调用。这也有助于将来别人调试你的类时，理解初始化代码的工作流程。

重载指定构造函数

Tip
当你写子类的时候，如果需要 ``init…`` 方法，记得重载父类的指定构造函数。

如果你没有重载父类的指定构造函数，你的构造函数有时可能不会被调用，这会导致非常隐秘而且难以解决的 bug。

重载 NSObject 的方法

Tip
如果重载了 ``NSObject`` 类的方法，强烈建议把它们放在 ``@implementation`` 内的起始处，这也是常见的操作方法。

通常适用（但不局限）于 `init...`，`copyWithZone:`，以及 `dealloc` 方法。所有 `init...` 方法应该放在一起，`copyWithZone:` 紧随其后，最后才是 `dealloc` 方法。

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。

保持公共 API 简单

Tip
保持类简单；避免 “厨房水槽（kitchen-sink）” 式的 API。如果一个函数压根没必要公开，就不要这么做。用私有类别保证公共头

与 C++ 不同，Objective-C 没有方法来区分公共的方法和私有的方法 -- 所有的方法都是公共的（译者注：这取决于 Objective-C 运行时的方法调用的消息机制）。因此，除非客户端的代码期望使用某个方法，不要把这个方法放进公共 API 中。尽可能的避免了你你不希望被调用的方法却被调用到。这包括重载父类的方法。对于内部实现所需要的方法，在实现的文件中定义一个类别，而不是把它们放进公有的头文件中。

```
// GTMFoo.m
#import "GTMFoo.h"

@interface GTMFoo (PrivateDelegateHandling)
- (NSString *)doSomethingWithDelegate;  // Declare private method
@end

@implementation GTMFoo(PrivateDelegateHandling)
...
- (NSString *)doSomethingWithDelegate {
  // Implement this method
}
```

```
...
@end
```

Objective-C 2.0 以前，如果你在私有的 `@interface` 中声明了某个方法，但在 `@implementation` 中忘记定义这个方法，编译器不会抱怨（这是因为你没有在其它的类别中实现这个私有的方法）。解决文案是将方法放进指定类别的 `@implemenation` 中。

如果你在使用 Objective-C 2.0，相反你应该使用 `类扩展 <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_4_section_5.html>` _ 来声明你的私有类别，例如：

```
@interface GMFoo () { ... }
```

这么做确保如果声明的方法没有在 `@implementation` 中实现，会触发一个编译器告警。

再次说明，“私有的” 方法其实不是私有的。你有时可能不小心重载了父类的私有方法，因而制造出很难查找的 Bug。通常，私有的方法应该有一个相当特殊的名字以防止子类无意地重载它们。

Ojbective-C 的类别可以用来将一个大的 `@implementation` 拆分成更容易理解的小块，同时，类别可以为最适合的类添加新的、特定应用程序的功能。例如，当添加一个 “middle truncation” 方法时，创建一个 `NSString` 的新类别并把方法放在里面，要比创建任意的一个新类把方法放进里面好得多。

`#import` and `#include`

```
``#import`` Ojbective-C/Objective-C++ 头文件，``#include`` C/C++ 头文件。
```

基于你所包括的头文件的编程语言，选择使用 `#import` 或是 `#include`：

- 当包含一个使用 Objective-C、Objective-C++ 的头文件时，使用 `#import` 。
- 当包含一个使用标准 C、C++ 头文件时，使用 `#include` 。头文件应该使用 `#define 保护 <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml?showone=The__define_Guard#The__define_Guard>` _。

一些 Ojbective-C 的头文件缺少 `#define` 保护，需要使用 `#import` 的方式包含。由于 Objective-C 的头文件只会被 Objective-C 的源文件及头文件包含，广泛地使用 `#import` 是可以的。

文件中没有 Objective-C 代码的标准 C、C++ 头文件，很可能会被普通的 C、C++ 包含。由于标准 C、C++ 里面没有 `#import` 的用法，这些文件将被 `#include` 。在 Objective-C 源文件中使用 `#include` 包含这些头文件，意味着这些头文件永远会在相同的语义下包含。

这条规则帮助跨平台的项目避免低级错误。某个 Mac 开发者写了一个新的 C 或 C++ 头文件，如果忘记使用 `#define` 保护，在 Mac 下使用 `#import` 这个头文件不回引起问题，但是在其它平台下使用 `#include` 将可能编译

失败。在所有的平台上统一使用 `#include`，意味着构造更可能全都成功或者失败，防止这些文件只能在某些平台下能够工作。

```
#import <Cocoa/Cocoa.h>
#include <CoreFoundation/CoreFoundation.h>
#import "GTMFoo.h"
#include "base/basictypes.h"
```

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

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];
}
```

init 和 dealloc 内避免使用访问器

Tip
在 ``init`` 和 ``dealloc`` 方法执行的过程中，子类可能会处在一个不一致的状态，所以这些方法中的代码应避免调用访问器。

子类尚未初始化，或在 init 和 dealloc 方法执行时已经被销毁，会使访问器方法很可能不可靠。实际上，应在这些方法中直接对 ivals 进行赋值或释放操作。

正确：

```
- (id)init {
  self = [super init];
  if (self) {
    bar_ = [[NSMutableString alloc] init];  // good
  }
  return self;
}

- (void)dealloc {
  [bar_ release];                           // good
  [super dealloc];
}
```

错误：

```
- (id)init {
  self = [super init];
  if (self) {
    self.bar = [NSMutableString string];  // avoid
```

```
  }
  return self;
}

- (void)dealloc {
  self.bar = nil;                    // avoid
  [super dealloc];
}
```

按声明顺序销毁实例变量

> ``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。

`setter` 应复制 NSStrings

> 接受 ``NSString`` 作为参数的 ``setter``，应该总是 ``copy`` 传入的字符串。

永远不要仅仅 `retain` 一个字符串。因为调用者很可能在你不知情的情况下修改了字符串。不要假定别人不会修改，你接受的对象是一个 `NSString` 对象而不是 `NSMutableString` 对象。

```
- (void)setFoo:(NSString *)aFoo {
  [foo_ autorelease];
  foo_ = [aFoo copy];
}
```

避免抛异常

> Tip
> 不要 ``@throw`` Objective-C 异常，同时也要时刻准备捕获从第三方或 OS 代码中抛出的异常。

我们的确允许 `-fobjc-exceptions` 编译开关（主要因为我们要用到 `@synchronized` ），但我们不使用 `@throw` 。为了合理使用第三方的代码， `@try` 、 `@catch` 和 `@finally` 是允许的。如果你确实使用了异常，请明确注释你期望什么方法抛出异常。

不要使用 NS_DURING 、NS_HANDLER 、NS_ENDHANDLER 、NS_VALUERETURN 和 NS_VOIDRETURN 宏，除非你写的代码需要在 Mac OS X 10.2 或之前的操作系统中运行。

注意：如果抛出 Objective-C 异常，Objective-C++ 代码中基于栈的对象不会被销毁。比如：

```
class exceptiontest {
 public:
  exceptiontest() { NSLog(@"Created"); }
  ~exceptiontest() { NSLog(@"Destroyed"); }
};

void foo() {
  exceptiontest a;
  NSException *exception = [NSException exceptionWithName:@"foo"
                                                   reason:@"bar"
                                                 userInfo:nil];
  @throw exception;
}

int main(int argc, char *argv[]) {
  GMAutoreleasePool pool;
  @try {
    foo();
  }
  @catch(NSException *ex) {
    NSLog(@"exception raised");
  }
  return 0;
}
```

会输出：

注意：这里析构函数从未被调用。这主要会影响基于栈的 smartptr ，比如 shared_ptr 、linked_ptr ，以及所有你可能用到的 STL 对象。因此我们不得不痛苦的说，如果必须在 Objective-C++ 中使用异常，就只用 C++ 的异常机制。永远不应该重新抛出 Objective-C 异常，也不应该在 @try 、@catch 或 @finally 语句块中使用基于栈的 C++ 对象。

nil 检查

```
``nil`` 检查只用在逻辑流程中。
```

使用 nil 的检查来检查应用程序的逻辑流程，而不是避免崩溃。Objective-C 运行时会处理向 nil 对象发送消息的情况。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，参见 Apple’s documentation <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_2_section_3.html> _ 。

注意，这和 C/C++ 中检查指针是否为 ``NULL`` 很不一样，C/C++ 运行时不做任何检查，从而导致应用程序崩溃。因此你仍然需要保证你不会对一个 C/C++ 的空指针解引用。

BOOL 若干陷阱

> 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）

```
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 iP
```

命名

属性所关联的实例变量的命名必须遵守以下划线作为后缀的规则。属性的名字应该与成员变量去掉下划线后缀的名字一模一样。

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。

点引用

点引用是地道的 Objective-C 2.0 风格。它被使用于简单的属性 `set`、`get` 操作，但不应该用它来调用对象的其它操作。

正确的做法：

```
NSString *oldName = myObject.name;
myObject.name = @"Alice";
```

错误的做法：

```
NSArray *array = [[NSArray arrayWithObject:@"hello"] retain];

NSUInteger numberOfItems = array.count;  // not a property
array.release;                           // not a property
```

没有实例变量的接口

```
没有声明任何实例变量的接口，应省略空花括号。
```

正确的做法：

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

错误的做法：

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

自动 synthesize 实例变量

```
Tip
  只运行在 iOS 下的代码，优先考虑使用自动 ``synthesize`` 实例变量。
```

`synthesize` 实例变量时，使用 `@synthesize var = var_;` 防止原本想调用 `self.var = blah;` 却不慎写成了 `var = blah;` 。

不要synthesize CFType的属性 CFType应该永远使用@dynamic实现指示符。 尽管CFType不能使用retain属性特性，开发者必须自己处理retain和release。很少有情况你需要仅仅对它进行赋值，因此最好显示地实现getter和setter，并作出注释说明。 列出所有的实现指示符 尽管@dynamic是默认的，显示列出它以及其它的实现指示符会提高可读性，代码阅读者可以一眼就知道类的每个属性是如何实现的。

```
// Header file
@interface Foo : NSObject
// A guy walks into a bar.
@property(nonatomic, copy) NSString *bar;
@end

// Implementation file
@interface Foo ()
@property(nonatomic, retain) NSArray *baz;
@end

@implementation Foo
@synthesize bar = bar_;
@synthesize baz = baz_;
@end
```

Cocoa 模式

委托模式

> 委托对象不应该被 ``retain``

实现委托模式的类应：

1. 拥有一个名为 `delegate_` 的实例变量来引用委托。
2. 因此，访问器方法应该命名为 `delegate` 和 `setDelegate:` 。
3. `delegate_` 对象不应该被 `retain` 。

模型/视图/控制器（MVC）

> Tip
> 分离模型与视图。分离控制器与视图、模型。回调 API 使用 ``@protocol``。

- 分离模型与视图：不要假设模型或者数据源的表示方法。保持数据源与表示层之间的接口抽象。视图不需要了解模型的逻辑（主要的规则是问问你自己，对于数据源的一个实例，有没有可能有多种不同状态的表示方法）。
- 分离控制器与模型、视图：不要把所有的 “业务逻辑” 放进跟视图有关的类中。这使代码非常难以复用。使用控制器类来处理这些代码，但保证控制器不需要了解太多表示层的逻辑。
- 使用 `@protocol` 来定义回调 API，如果不是所有的方法都必须实现，使用 `@optional`（特例：使用 Objective-C 1.0 时，`@optional` 不可用，可使用类别来定义一个 “非正规的协议”）。