Google Objective-C Style Guide 中文版

译者的话：

一直想翻译这个style guide，终于在周末花了7个小时的时间用vim敲出了HTML。很多术语的翻译很难，平时看的中文技术类书籍有限，对很多术语的中文译法不是很清楚，难免有不恰当之处，请读者指出并帮我改进。

王轲"ewangke at gmail.com" 2011.03.27

目录

背景介绍

Objective-C是一种动态的面向对象的语言，它是C的扩展。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的主要的应用程序框架。它由一组支持Mac OS X全部特性的，并可用于快速开发的Objective-C类构成。

苹果公司已经撰写了非常全面的Objective-C编码指南。Google也为C++写了类型的编码指南。这个Objective-C指南是苹果和Google最佳实践的结合。因此，在阅读本指南之前，请确定你已经读过以下内容：

- Apple's Cocoa Coding Guidelines
- Google's Open Source C++ Style Guide

注意：所有在Google的C++编码指南中所禁止的事情，如未有特殊提及，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的代码提供编程指南及最佳实践。许多指南是在实际的项目及小组中经过长期的演进及验证的。Google开发的开源项目遵从本指南的要求。

Google已经发布了Google Toolbox for Mac project的部分开源代码，这些代码遵从本指南。跨多个项目的代码是这个库中的较好候选。

请注意，本指南不是Objective-C的教程。我们假定读者已经对Objective-C非常熟悉。如果你刚刚接触Objective-C或者需要提高，请阅读The Objective-C Programming Language。

例子

一个例子顶上一千句话，我们就从这样的一个例子开始，来感受一下编码的风格、空格以及命名等等。

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空格。

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

#import

// 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 GTMFoo : NSObject {
 @private
  NSString *foo_;
  NSString *bar_;
}

// Returns an autoreleased instance of GMFoo. See -initWithString: for details
// about the argument.
+ (id)fooWithString:(NSString *)string;
```

```
// Designated initializer. |string| will be copied and assigned to |foo_|.
- (id)initWithString:(NSString *)string;

// Gets and sets the string for |foo_|.
- (NSString *)foo;
- (void)setFoo:(NSString *)newFoo;

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

@end
```

一个实现文件的例子，展示了@implementation部分如何进行正确的注释、空格。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。

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

#import "GTMFoo.h"


@implementation GTMFoo

+ (id)fooWithString:(NSString *)string {
  return [[[self alloc] initWithString:string] autorelease];
}

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

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

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

- (NSString *)foo {
  return foo_;
}

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

- (BOOL)doWorkWithString:(NSString *)blah {
  // ...
```

```
  return NO;
}

@end
```

空格与格式

空格与制表符

我们使用空格进行缩进。不要在代码中使用制表符。你应该将你的文本编辑器设置成自动将制表符替换成空格。

总结：只使用空格，每次缩进两个空格。

行宽

即使Objective-C比C++更加冗长，为了保证本指南的可操作性，我们决定保持每行宽度为80列。这比你想的要简单。

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，因此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。

总结：代码中的每行最多有80个字符。

方法声明与定义

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，要与原的代码一致。

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 {
  ...
}
```

总结：-或者+与返回类型之间，需要有空格。参数列表中，只有参数之间有空格。

方法调用

方法调用时，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];
```

总结：方法定义的格式与方法声明的格式非常相似。当格式的风格有多种选择时，新的代码要与已经存在的代码保持一致。

@public与@private

与C++中的public, private以及protected非常相似。

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

总结：@public以及@private访问标识符应该以一个空格缩进。

异常

如果你必须使用Objective-C的异常，按下面的格式进行编码代码。然后，请参见避免抛出异常来了解不应该使用异常的原因。

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

总结：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个空格。

协议

这条规则也同样适用于类声明、成员变量以及方法声明。例如：

```
@interface MyProtocoledClass : NSObject {
 @private
  id delegate_;
}
- (void)setDelegate:(id)aDelegate;
@end
```

总结：尖括号所包括的协议名称与前面的类型标识之间不应该有空格。

命名

对于可维护的代码，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，不需要太多的注释修饰。

当编写纯Objective-C代码时，我们基本遵守标准的Objective-C naming rules，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camel命名法，这在Objective-C社区中非常普遍。

任何的类、分类、方法以及变量的名字中都使用全大写缩写。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。许多的项目需要实现跨平台的C++代码，并与Objective-C以及Cocoa混合编写。或者以C++作为后端，Cocoa作为前端。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语句块中，就使用Objective-C的风格。如果实现一个C++的类，请使用C++的风格。这避免了成员变量与局部变量使用混合的命名风格，这会严重地影响可读性。

文件名

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

分类的文件名应该包含被扩展的类的名字，如：GTMNSString+Utils.h或GTMNSTextView+Autocomplete.h。

总结：文件名应该反映了它实现了什么类。遵守你的项目的惯例。

Objective-C++

为了最小化Cocoa/Objective-C与C++之间的命名风格的冲突，按照下面方法实现的风格编写代码。在实现@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];
}
```

总结：当编写源代码时，Ojbective-C++应该采用你正在实现的方法/函数的风格。

类名

应用程序级的代码，应该尽量避免不必要的前缀。为每个类都添加前缀不会提高任何的可读性。当

设计跨不同应用程序的代码时，应该使用前缀，例如：GTMSendMessage。

总结：类名、分类名、协议名应该以大写字母开始，并混合小写字母来分隔单词。(Camel命名法)

分类名

例如，录我们要创建一个NSString的类别以解析时，我们将把类别放在一个名为GTMNSString+Parsing.h的文件中。类别名本身的名字是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只有一个命名空间。如果代码不会被分享并且不会被运行在不同的地址空间中，方法名字就不那么重要。

类名与包含类别名的括号之间，应该以一个空格分隔。

总结：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。类别应该包含它所扩展的类的名字。

Objective-C方法名

方法名应该读起来就像句子，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: or replaceCharactersInRange:withString:)参见Apple's Guide to Naming Methods。

访问器方法应该与他们getting的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。

总结：方法名应该以小写字母开头，并混合大小写（Camel命名法）。每个命名的参数也应该以小写字母开头。

变量名

常用变量名

对于静态的类别，如int以及指针等，不要使用匈牙利命名法。要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];
```

成员变量

成员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KVO/KVC绑定成员变量时，我们允许这个例外（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、枚举、静态局部变量等）应该以小写字母k开头，使用混合大小写的格式来分隔单词，如：kInvalidHandle，kWritePerm。

总结：变量名应该以小写字母开头，并混合大小写。类的成员变量应该以下划线作为后缀。例如：myLocalVariable、myInstanceVariable_。如果不能使用Objective-C 2.0的@property，使用KVO/KVC绑定的成员变量可以以一个下划线作为前缀。

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：尽量注释很重要，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，记得你在给你的听众写，即下一个需要阅读你的代码的代码贡献者。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，不同的地方会在下面指出

文件注释

版权信息及作者

每个文件应该按顺序包括如下内容:

- 版权信息声明（如：Copyright 2008 Google Inc.）
- 授权样版。选择一个合适的项目所使用的授权样板（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。

总结：以版权信息作为文件头部，开始每一个文件，后接文件内容的描述。

声明注释

```
// 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。

总结：每个接口、类别以及协议应该注释，以描述它的目的及作用。

实现注释

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")|
```

总结：使用|来引用注释中的变量名及符号名而不是使用引号。

对象所有权

继承自NSObject的成员变量指针，通常被假定是强引用关系（retained）。如果它有没被类retained，应该注释为弱引用（weak）。然而，以IBOutlets作为标签的成员变量默认是不会被类retained的。如果是强引用关系，应该注释。

当成员变量指向CoreFoundation、C++或者其它非Objective-C对象时，无论是强引用还是弱引用，都需要注释说明。注意：Objective-C对象中的C++对象的自动封装，默认情况下是不允许的，参见。

强引用及弱引用注释的例子：

```
@interface MyDelegate : NSObject {
 @private
  IBOutlet NSButton* okButton_;  // normal NSControl
  IBOutlet NSMenu* myContextMenu_;  // manually-loaded menu (strong)

  AnObjcObject* doohickey_;  // my doohickey
  MyController* controller_;  // so we can send msgs back (weak, owns me)

  // non-NSObject pointers...
  CWackyCPPClass* wacky_;  // some cross-platform object (strong)
  CFDictionaryRef* dict_;  // (strong)
}
@end
```

强引用：对象被类retained。

弱引用：对象没有被类retained。（如委托）

总结：当与Objective-C最通常的作法不同时，尽量使用指针的所有权模型尽量直观。

Cocoa以及Objective-C特性

成员变量应该为@private

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

总结：成员变量应该声明为私有。

指明指定的初始化器

对于需要继承你的类的人来说，指定的初始化器十分重要。这样他们就可以只重写一个初始化器（可能是几个）来保证他们的子类的初始化器会被调用。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。

总结：注释并且明确指出你的类的指定的初始化器。

重写指定的初始化器

如果我没有重写父类的指定的初始化器，你的初始化器有时可能不会被调用，这会导致很微妙而且难以解决的bug。

总结：当你写子类的时候，如果需要init...方法，记得重写父类的指定的初始化器。

初始化

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。

总结：不要在init方法中，将成员变量初始化为0或者nil，这是冗余的。

避免使用+new

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

总结：不要调用NSObject类的类方法new，也不要在子类重写。相反，你应该使用alloc和init方法来创建并初始化一个对象。

保持公有的API尽量简单

与C++不同，Objective-C没有方法来区分公有的方法和私有的方法，所有的方法都是公有的（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有的API中。这降低了你不希望被调用的方法被调用的可能性。这包括重写父类的方法。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
```

在Objective-C 2.0之前，如果你在私有的@interface中声明了某个方法，但在@implementation中忘记定义了这个方法，编译器不会反对（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来声明你的私有类别，例如：

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

这会保证如果声明的方法如果没有在@implementation中实现，会产生一个编译器警告。

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写了父类的私有方法，这很难调试。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写它们。

Ojbective-C的类别是一种很好的方法来将一个大的@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、基于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，比创建任意的一个新类把方法放进里面要好得多。

#import与#include

基于你所包括的头文件的编程语言，选择使用#import还是#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。头文件应该提供自己的。

一些Ojbective-C的头文件缺少#define保护，只期望被使用#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的标准C、C++的头文件，可能会被普通的C、C++包含。既然标准的C、C++里面没有#import的用法，这些文件将被#include包含。Objective-C源文件中使用#include包含，意味着这些头文件会永远以相同的语义被包含。

这条规则帮助跨平台的项目中产生无意的错误。一个忘记使用#define保护的Mac开发者，在引用了新的C、C++头文件时，如果新的头文件使用#import被引用，在Mac上会没有问题。但在其它使用#include平台上可能会使构造失败。一致地在所有的平台上使用#include，意味着构造更可能一致地成功或者失败。这避免了文件只能在某些平台上使用的尴尬。

下面是正确的使用方式：

```
#import
#include
#import "GTMFoo.h"
#include "base/basictypes.h"
```

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

使用根框架

当你试图从框架（如Cocoa或者Foundation）中包含单独的系统头文件时，实际上包含顶级根框架编译器要作更少的工作。根框架通常被预编译，并且加载得更快。另外记得使用#import而不是#include来包含Objective-C的框架。

正确的做法：

```
#import      // good
```

错误的做法：

```
#import         // avoid
#import
...
```

总结：包含根框架而不是单独的文件。

创建对象时尽量使用autorelease

尽管运行效率会差一点，这避免了意外地删除了release或者插入return语句而产生内存泄露的可能。例如：

错误的做法：

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

正确的做法：

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

总结：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。

Autolease之后retain

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。有一些正确的方法来处理它。我们选择autorelease之后retain的方法因为事实证明它更少地出错。注意大的循环会填满autorelease pool，并且可能效率上会差一点，但这点折衷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];
}
```

总结：给对象赋值时遵守autorelease之后retain的模式。

Dealloc中应该按声明的顺序处理成员变量

代码审查者在审查新的或者修改过的dealloc实现时，需要保证每个retained的对象都得到了释放。

为了简单的审查dealloc，对象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，注释这个方法处理了哪些成员变量的释放。

总结：dealloc中对象被释放的顺序应该与他们在@interface中声明的顺序一致，这有助于代码审查。

Setters中对NSString进行copy

永远不要仅仅retain一个字符串。这避免了调用者在你不知道的情况下对字符串作出了修改。不要作出那样的假设：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];
}
```

总结：接受NSString作为参数的setter，应该copy它所接受的字符串。

避免抛出异常

我们确实在编译时允许-fobjc-exceptions（主要我们得到了使用@synchronized的好处），但我们不使用@throw。当正确地使用第三方的代码时，使用@try、@catch、和@finally是允许的。如果你确实使用了异常，请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 这些宏，除非你写的代码需要在Mac OS X 10.2或者之前的操作系统中运行。

注意：当使用Objective-C++写基于栈的对象的代码时，如果抛出Objective-C异常，对象不会被清理。例如：

```
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：

```
2006-09-28 12:34:29.244 exceptiontest[23661] Created
```

```
2006-09-28 12:34:29.244 exceptiontest[23661] exception raised
```

注意：这里析构函数从未被调用。这主要会影响基于栈的对象如shared_ptr、linked_ptr和所有你可能使用的STL对象。你永远不应该重新抛出Objective-C异常，也不应该在@try, @catch, @finally 语句块中使用基于栈的C++对象。

总结：不要抛出Objective-C异常，但准备从第三方的调用或者系统调用捕捉异常。

nil的检查

使用nil的检查来检查应用程序的逻辑，而不是避免崩溃。Objective-C运行时会处理向一个nil的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。

注意，这与检查C/C++的指针是否为NULL非常不同，运行时不会检查空的情况，并导致你的应用程序崩溃。你仍然需要保证你不会对一个C/C++的空指针解引用。

总结：nil检查只用于逻辑流的判断。

BOOL陷阱

Ojbective-C中定义BOOL为无符号字符型，这意味着BOOL类型可以有不同于YES(1)或者NO(0）的值。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，这取决于整型结果的最后一个字节，可能产生一个NO的值。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 and 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当作一个普通的整形，就像我产在上面讨论的那样。Objective-C的方法签名中，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];
}
```

同样的，不要直接比较BOOL变量与YES/NO。不仅仅这影响可读性，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!
```

总结：将常规整形转换成BOOL时要小心，不要直接将BOOL值与YES进行比较。

属性

命名

属性所关联的成员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

类接口中的属性的声明必须紧跟着成员变量语句块。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
```

NSString使用copy特性

NSString属性应该永远被声明为copy特性。

这从逻辑上遵守了NSString的setter必须使用copy而不是retain。

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。

错误的做法:

```
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
```

正确的做法:

```
@interface MyClass : NSObject
@property(readonly) NSString *name;
@end

@implementation MyClass
@dynamic name;
.
.
.
- (NSString*)name {
  return @"foo";
}
@end
```

原子性

一定要注意属性的开销。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。显示将你的属性声明为nonatomic除非你需要原子操作。

点引用

点引用是地道的Objective-C 2.0的风格。它被使用于简单的属性set、get操作，但对象的其它行为不应该使用它。

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
```

总结：需要注意的是，使用点引用语法必须需要Objective-C 2.0的支持，这意味着你的代码只能运行于iPhone或者Mac OS X 10.5(Leopard)及以后的版本。点引用只允许访问声明的属性。

Cocoa模式

委托模式

实现委托模式的类应该：

- 拥有一个名为delegate_的成员变量来引用委托。
- 因此，访问器方法应该名为delegate和setDelegate:。
- delegate_对象不应该被retained。

总结：委托对象不应该被retained。

模型-视图-控制器

- 模型与视图分离：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）。
- 控制器与模型、视图分离：不要把所有的“领域逻辑”放进跟视图有关的类中。这命名得代码非常难以重用。使用控制器来写这些代码，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（例外：当使用Objective-C 1.0，@optional不可用，因此请使用类别来定义“非正式的协议”）。

总结：分离模型与视图。分离控制器与视图、模型。回调API使用@protocol。