介绍

使用公共API开发Cocoa框架、插件及其他可执行文件需要使用的方法和约定不同于应用程序开发。如果产品主要客户是开发人员，则保证产品的编程接口清晰明确，不至于让开发者产生疑惑十分重要。这种情况下，API命名约定就可以派上用场，它可以帮助您保持编程接口一致明确。另外，框架开发领域也存在一些特定的编程技术—或者说，这些技术对框架开发更加重要—例如版本管理，二进制兼容性，错误处理以及内存管理等。本节主题包括Cocoa命名约定以及我们提倡的框架编程实践。

本文档的组织方式

本主题的文章大致分成两类。第一类数量较多，讲述编程接口的命名约定。苹果公司的Cocoa框架使用了这里介绍的命名约定（只有一些小的例外）。该类文章包括如下：

“代码命名基础”
“为方法命名”
“为函数命名”
“为实例变量和数据类型命名”
“可以使用的缩略名称”

第二类的文章(目前只有一篇)讨论了框架编程方面的内容。

“框架开发者可以使用的技巧和技术”

代码命名基础

在面向对象软件库的设计过程中，开发人员经常忽视对类、方法、函数、常量以及其他编程接元素的命名。本节讨论大多数Cocoa接口通用的几条命名约定。

一般性原则

清晰

- 最好是既清晰又尽可能地简短，但不要为了追求简短而丧失清晰性：

`insertObject:atIndex:`	好的命名
`insert:at:`	不清晰；插入什么？“at”表示什么？
`removeObjectAtIndex:`	好的命名
`removeObject:`	这样命名也不错，因为方法将移除通过参数引用的对象。
`remove:`	不清晰：要移除什么？

- 通常情况下，请不要缩写事物的名称，即使名称很长，也应该把它完全拼写出来。

`destinationSelection`	好的命名
`destSel`	不清晰
`setBackgroundColor:`	好的命名
`setBkgdColor:`	不清晰

  您可能觉得某个缩写众所周知，但实际可能并非如此。特别是具有不同文化和语言背景的开发人员，在遇到您提供的方法或函数的名称缩写时，他们可能不明白其中的含义。
- 不过，有一些缩写确实很常见并且有很长的使用历史。因此，您可以继续使用。请参看“可以接受的缩略名称”一节以了解更多的信息。
- 要防止API的名称出现歧义。这里的歧义是指名称具有多种解释方式。

`sendPort`	该方法是把端口发送出去还是返回发送端口呢？
`displayName`	该方法是显示某个名称还是返回用户界面中接收者的标题呢？

一致性

- 请尽可能在Cocoa编程接口中保持名称一致性。如果不太有把握做到这一点，则请浏览一下头文件和参考文档中的范例。
- 如果类方法利用多态，一致性就显得尤其重要。因为在这种情况下，不同类用于完成同样事件的方法必须具有相同的名称。

`- (int)tag`	该方法同时定义在 NSView、 NSCell、 NSControl这三个类里面。
`- (void)setStringValue:(NSString *)`	该方法定义于数个Cocoa类中

您可以参看 “方法参数”一节。

不能自我指涉

- 名称不应该自我指涉。

NSString	可以使用
NSStringObject	该名称自我指涉

- 掩码的常量（可以使用位操作进行组合）不适用这条规则，作为通告的常量也不适用。

`NSUnderlineByWordMask`
`NSTableViewColumnDidMoveNotification`

前缀

前缀是编程接口名称的重要部分，它们可以区分软件的功能范畴。通常情况下，提供编程接口的软件会被打包成框架（Foundation框架以及Application Kit框架就是如此）或者是和框架紧密相关的产品，我们可以利用前缀来区分框架的功能范畴。另外，前缀可以防止第三方开发者定义的符号和苹果公司定义的符号发生冲突（以及防止苹果公司不同框架之间的符号发生冲突）。

- 前缀有规定的格式。它需要由两个或者三个大写字符组成，而且不能使用下划线或者“子前缀”。下面是一些例子：

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

- 在为类，协议，函数，常量以及通过`typedef`定义的结构命名时，请使用前缀。但在命名方法时，请不要使用前缀，因为方法已经存在于其定义类所创建的名称空间中。同理，在定义结构的字段时，也不要使用前缀。

书写约定

为API元素命名的时候， 请遵循下面这几条简单的书写约定：

- 对于含有多个单词的名称，请不要使用标点符号标志和分割符（下划线，破折号之类）；而是要大写每个单词的首字符并且把这些单词连续地拼写在一起。然而如下这些限定条件您也需要注意：
  方法的名称要以一个小写字符开头，而名称中单词的首字符应该大写。另外，请不要在方法的名称中使用前缀。

```
fileExistsAtPath:isDirectory:
```

  如果方法名称的开头是某个众所周知的缩略语，则该原则就不适用。例如`TIFFRepresentation` (NSImage)，该名称就不遵循该原则。
  函数或常量名称使用其关联类的前缀，并且要名称中单词的首字符要大写。

```
NSRunAlertPanel


NSCellDisabled
```

- 请不要使用下划线作为前缀来表示私有的属性，尤其是不要在类方法中使用。因为苹果公司保留使用这种方式，如果第三方再使用，就有可能会导致名称空间冲突。他们有可能在无意中用自己的方法覆盖了一个已经存在的私有方法，这样做将会带来灾难性的后果。请参看“私有方法”一节。您可以了解到我们提倡的可供私有API使用的约定。

类和协议的名称

类的名称应包含一个名词，这个名词明确地指示这个类(或者类对象)表示什么或者要做什么。此外，类名称还应该包含适当的前缀。（请参考“前缀”一节）。Foundation框架以及Application Kit框架就有很多这样的例子，例如NSString， NSDate，NSScanner，NSApplication，NSButton， 以及NSEvent。

我们应根据协议对方法的分组方式来为其命名：

- 大部分协议会把一些彼此相关但又不合类关联的方法归结在一起，形成一个特殊的方法集合。这种协议要合理地命名，不要将其和类名混淆。一种常见的约定是使用动名词格式（“...ing”）：

NSLocking	好
NSLock	差（看起来像是个类名）

- 有一些协议会把一些彼此无关的方法归结在一起（不是创建几个独立的小协议）。对于这样的协议，我们倾向于把它和一个类联系起来，利用类来作为协议的主要表现。并且，我们约定让此种协议使用和类一样的名称。NSObject协议就是这种情况。它把一些不相关的方法组合在一起，这些方法有的用于查询任何对象在类层次中的位置，有的可以调用对象的特定方法，有的可以用来增加或者减少对象的引用计数。由于NSObject类提供了这些方法的主要表现，所以我们使用类名作为协议名称。

头文件

头文件的命名方式很重要，因为通过使用合理的命名约定，您利用文件名称来指示文件中包含的内容：

- **声明一个独立的类或协议**。.如果一个类或协议不属于某个群，则请将其声明放置在一份独立的文件，并使用其名称作为文件名。

头文件	声明
`NSApplication.h`	NSApplication类

- **声明相关联的类或者协议**：如果一群声明（类，类别以及协议）彼此相关，则请将它们放在一份文件，并使用主要的类或者协议名称作为文件名。

头文件	声明
`NSString.h`	NSString 和 NSMutableString这两个类
`NSLock.h`	NSLocking协议以及 NSLock， NSConditionLock，和 NSRecursiveLock这几个类

- **包含框架头文件**。每个框架都应该包含一份头文件，它的名称和框架名相同，而内容则包含了框架的全部公共头文件。

头文件	框架
`Foundation.h`	`Foundation`框架

- **为另一个框架里的某个类添加API**。如果您在一个框架中声明一些方法，而这些方法属于另一个框架中某个类的范畴类，则请在原始类的名称后加上“Additions”，然后将其作为头文件的名称。例如Application Kit框架中的`NSBundleAdditions.h`头文件就是这种处理方式。
- **相关联的函数和数据类型**。如果一群函数， 常量，结构以及其他数据类型彼此相互关联，则请将它们放入到合理命名的头文件，例如`NSGaphics.h`(位于Application Kit)。

为方法命名

方法可能是编程接口中最常见的元素了，因此对其命名要特别注意。本部分讨论方法命名的相关方面：

通用规则

为方法命名时，请记住下面这些通用的指导原则：

- 方法名称应以小写字符开头，名称中的单词首字符要大写。另外，请不要在方法名称中使用前缀。您可以参考“书写约定”一节，以了解更多信息。有两种特定的情况不适用该原则。其一，方法的名称可以使用某个众所周知的缩写开头，而该缩写可以大写（例如，TIFF 或者 PDF）。其二，您可以使用前缀来分组并区分私有方法（请参考“私有方法”一节）。
- 如果方法代表一个对象执行的动作，则其名称应该以一个动词开头：

```
- (void)invokeWithTarget:(id)target;

- (void)selectTabViewItem:(NSTabViewItem *)tabViewItem
```

- 请不要使用 “do”或者 “does”作为名称的一部分，因为这些辅助性的动词 不能为名称增加更多的含义。同时，请不要在动词之前使用副词或者形容词。
- 如果方法返回接收者的某个属性，则以属性名称作为方法名。如果方法没有间接地返回一个或多个值，您也无须使用”get“这样的单词。

`- (NSSize)cellSize;`	正确
`- (NSSize)calcCellSize;`	错误
`- (NSSize)getCellSize;`	错误

- 您可以参考 “存取方法”一节，以了解更多的信息。
- 所有参数前面都应使用关键字。

`- (void)sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;`	正确
`- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;`	错误

- 参数前面的单词应能够对参数进行描述。

`- (id)viewWithTag:(int)aTag;`	正确
`- (id)taggedView:(int)aTag;`	错误

- 如果您当前创建的方法比起它所继承的方法更有针对性，则您应该在已有的方法名称后面添加关键字，并将其作为新方法的名称。

`- (id)initWithFrame:(NSRect)frameRect;`	NSView
`- (id)initWithFrame:(NSRect)frameRect mode:(int)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide;`	NSMatrix是NSView的子类。

-
-
- 请不要使用”and“来连接两个表示接受者属性的关键字。

`- (int)runModalForDirectory:(NSString *)path file:(NSString *) name types:(NSArray *)fileTypes;`	正确
`- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes;`	错误

- 虽然上面的例子使用”and“这个词感觉还不错，但是随着创建的方法所带有的关键字越来越多，这种方式会引起问题。
- 如果方法描述了两个独立的动作，请使用”and“把它们连接起来。

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

-

存取方法

存取方法用于设置或返回对象的属性（也就是对象的实例变量）。由于属性的表示方法不同，我们提倡的存取方法的格式也有差异：

- 如果某个属性使用名词来表示，则方法的格式如下：`- (void)set`Noun`:(type)`aNoun`;`
  `- (type)noun;`
  例如：

```
- (void)setColor:(NSColor *)aColor;


- (NSColor *)color;
```

-
- 如果某个属性使用形容词表示， 则方法的格式为：`- (void)set`Adjective`:(BOOL)flag;`
  `- (BOOL)is`Adjective`;`
  例如：

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


- (BOOL)isEditable;
```

-
- 如果某个属性使用动词表示，则方法的格式为：`- (void)set`VerbObject`:(BOOL)flag;`
  `- (BOOL)`verbObject`;`
  例如：

```
- (void)setShowsAlpha:(BOOL)flag;


- (BOOL)showsAlpha;
```

-
- 这种情况下，动词应使用一般现在时的格式。
- 请不要使用分词形式把动词转换为形容词：

`- (void)setAcceptsGlyphInfo:(BOOL)flag;`	正确
`- (BOOL)acceptsGlyphInfo;`	正确
`- (void)setGlyphInfoAccepted:(BOOL)flag;`	错误
`- (BOOL)glyphInfoAccepted;`	错误

-
- 您可以使用情态动词（在动词前冠以“can”,"should","will"等），使得方法的名称更加明确，但是请不要使用“do”或“does”这样的情态动词。

`- (void)setCanHide:(BOOL)flag;`	正确
`- (BOOL)canHide;`	正确
`- (void)setShouldCloseDocument:(BOOL)flag;`	正确
`- (BOOL)shouldCloseDocument;`	正确
`- (void)setDoesAcceptGlyphInfo:(BOOL)flag;`	错误
`- (BOOL)doesAcceptGlyphInfo;`	错误

- 只有当方法间接地返回对象或者数值，您才需要在方法名称中使用get"。这种格式只适用于需要返回多个数据项的方法。

`- (void)getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;`	NSBezierPath

- 如果方法格式和上面一样，则其实现应该能够接受`NULL`参数，这样调用者才能够表明他们对其中的一个或者多个返回值不感兴趣。

委托方法

委托方法是指当某些事件发生时，对象在委托里调用的处理方法（如果委托实现了它们）。委托方法的格式独特，但它也适用于在对象数据源里调用的方法：

- 方法名称的开头应标识出发送消息的对象所属的类：

```
- (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;


- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
```

- 在此，类的名称不需要使用前缀并且首字符要小写。
- 除非方法只有一个参数，并且该参数表示消息的发送者，否则类名称后面都要加上一个冒号（参数是委托对象的引用）。

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

- 如果是因为发送了一则通告而导致某个方法被调用，则上述原则不适用。在这种情况下，方法仅有的一个参数是通告对象。

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

- 如果调用某个方法是为了通知委托某个事件已经发生或者即将发生， 则请在方法名称中使用“did”或者“will”这样的助动词。

```
- (void)browserDidScroll:(NSBrowser *)sender;


- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;
```

- 如果调用某个方法是为了要求委托代表其他对象执行某件事，当然，您也可以在方法名称中使用“did”或者“will”，但我们倾向于使用“should”。

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

集合方法

对于管理一个对象集合的对象（每个被管理的对象称为集合的一个元素），习惯上，我们要求它具有如下格式的方法：

`- (void)add`Element`:(`elementType`)`anObj`;`

`- (void)remove`Element`:(`elementType`)`anObj`;`

`- (NSArray *)`elements`;`

例如:

```
- (void)addLayoutManager:(NSLayoutManager *)obj;


- (void)removeLayoutManager:(NSLayoutManager *)obj;


- (NSArray *)layoutManagers;
```

下述内容是该原则的条件和细化：

- 如果集合确实是无序的， 则应返回一个NSSet类型的对象，而不是返回NSArray对象。
- 如果把元素插入到集合的指定位置这一功能很重要，则应使用与下面类似的方法来替换或者补充前述的某些方法。

```
- (void)insertLayoutManager:(NSLayoutManager *)obj atIndex:(int)index;


- (void)removeLayoutManagerAtIndex:(int)index;
```

使用集合方法时， 您需要记住下面这两条实现细节：

- 上述方法通常隐含了它们对于被插入对象的所有权，因此，用于添加或者插入对象的代码必须增加对象的计数，而用于移除对象的代码也必须要释放对象。
- 如果被插入的对象需要有一个指针指向其幕后的主对象， 则通常情况下， 您应该使用`set...`这样方法，它可以设置对象的背后对象指针，但并不增加其引用计数。我们以`insertLayoutManager:atIndex:`方法为例，NSLayoutManager使用如下方法来实现这一功能：

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

```
- (NSTextStorage *)textStorage;
```

- 正常情况下， 您应该不会直接调用`setTextStorage:`方法，但可能需要对其进行重写。

我们还有另外一个示列用于展示集合方法的上述约定，它来自于NSWindow类：

```
- (void)addChildWindow:(NSWindow *)childWin ordered:(NSWindowOrderingMode)place;

- (void)removeChildWindow:(NSWindow *)childWin;

- (NSArray *)childWindows;



- (NSWindow *)parentWindow;

- (void)setParentWindow:(NSWindow *)window;
```

方法的参数

下面是数条和方法参数命名相关的通用规则：

- 和方法名称一样， 参数的名称也是以小写的字符开头，并且后续单词的首字符要大写。例如：`removeObject:(id)anObject`）。
- 请不要在参数名称中使用"pointer"或者"ptr"。您应该使用参数的类型来声明参数是否是一个指针。
- 请不要使用一到两个字符的名称作为参数名。
- 请不要使用只剩几个字符的缩写。

习惯上(在Cocoa中),我们把下面的关键字和参数应该组合在一起使用：

```
...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，私有方法遵循和公共方法一样的命名规则。但是，有一种常见的约定是为私有方法添加一个前缀，这样我们就很容易区分它们。但即便是利用这样的约定，私有方法的名称还是有可能导致奇怪的问题。当您为某个Cocoa框架类设计子类时，您无法知道您的某个私有方法是

否在无意中覆盖了具有相同名称的私有的框架方法。

大部分Cocoa框架中私有方法的名称都带有一个下划线前缀（例如，`_fooData`），这个前缀把方法标记为私有。根据这样的实际情况，我们给出两条建议：

- 请不要在您的私有方法中使用下划线作为前缀，因为苹果公司保留使用这种命名约定。
- 在为某个很大的Cocoa框架类（例如NSView）派生子类时，如果需要绝对保证子类私有方法名称不会和超类发生冲突，则您可以为子类私有方法添加自己的前缀。前缀应该尽可能地具有唯一性，也许您的前缀可以基于公司或者项目名称，并且使用"XX_"这样的格式。例如，如果您的项目叫做Byte Flogger，则前缀可以是`BF_addObject:`这样的格式。

尽管为私有方法名称添加前缀似乎和早前我们对类方法的命名要求相矛盾，但这是因为此处的目的和早前不同：我们这么做是为了避免在无意中重写了超类中的私有方法。

为函数命名

Objective-C 允许使用函数或者方法来表达行为。如果底层对象总为单例或者处理的事物明显是功能性子系统，则您应该使用函数而非类方法。

请遵守下述几条函数通用命名规则：

- 函数名称和方法名称格式相似，但是有两种情况例外：
  函数要使用前缀开头，并且这个前缀和类或者常量所使用的一样。
  前缀后面的单词首字符要大写。
- 大多数函数名称以动词开头，该动词描述了函数的作用：

```
NSHighlightRect


NSDeallocateObject
```

用于查询属性的函数有一套更细致的命名规则：

- 如果函数返回其首个参数的某个属性，则请省略掉函数名称中的动词。

```
unsigned int NSEventMaskFromType(NSEventType type)


float NSHeight(NSRect aRect)
```

-
- 如果函数返回的值是个引用，则请在函数名称中使用“Get”。

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

-
- 如果函数返回值是布尔类型，则它应以曲折动词开头。

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

为实例变量和数据类型命名

本节描述实例变量、常量、异常、以及通知的命名约定。

实例变量

在为某个类添加实例变量时，请记住下面几个因素：

- 避免创建公共实例变量。开发人员应该关心对象的接口，而不是对象的数据存储方式这样的细节。
- 请把实例变量显式声明为`@private`或者`@protected`。如果您预期所提供的类会被子类化，并且子类可能需要父类的数据， 则请使用`@protected`指令来修饰实例变量。
- 请确保实例变量的名称能够扼要地描述它所保存的属性。

如果实例变量将作为类对象的可访问属性，则请务必为其编写存取方法。

常量

根据常量创建方式不同，其命名规则也有所差异。

枚举常量

- 请使用枚举类型来表示一群相互关联的整数值常量。
- 枚举常量及其所属的通过typedef定义的数据类型遵循和函数一样的命名约定（请查看“为函数命名”一节）。下面是一个取自`NSMatrix.h`文件的例子

```
typedef enum _NSMatrixMode {


NSRadioModeMatrix           = 0,


NSHighlightModeMatrix       = 1,
```

```
NSListModeMatrix            = 2,

NSTrackModeMatrix           = 3

} NSMatrixMode;
```

- 请注意，在上述例子中，typedef标签不是一定要具有的。
- 您也可以创建匿名的枚举类型来表示诸如位掩码这样的事物。例如：

```
enum {

    NSBorderlessWindowMask      = 0,

    NSTitledWindowMask          = 1 << 0,

    NSClosableWindowMask        = 1 << 1,

    NSMiniaturizableWindowMask  = 1 << 2,

    NSResizableWindowMask       = 1 << 3



};
```

使用const 创建的常量

- 请使用const 来创建浮点值常量。如果某个整数值常量和其他的常量不相关，您也可以使用const来创建，否则，则应使用枚举类型。
- 下面的声明展示了const常量的格式：

```
const float NSLightGray;
```

- 使用枚举类型声明的常量遵循和函数相同的命名约定。（请参考"为函数命名"一节）。

其他类型的常量

- 通常情况下， 请不要使用#define预处器理命令创建常量。对于整数值常量，请使用枚举类型创建，而对于浮点值常量，请使用const修饰符创建，这和前述的原则一样。
- 有些符号，预处理器需要对其进行计算，以便决定是否要对某一代码块进行处理，则它们应该使用大写字符表示。例如：

```
#ifdef DEBUG
```

- 请注意，编译器定义的宏，其开头和结尾要具有两个下划线字符。例如：

```
__MACH__
```

- 对于通告名称或字典关键字的字符串，请将其定义为常量。通过使用字符串常量，编译器可以验证字符串是否被正确赋值（也就是说， 编译器将执行拼写检查）。Cocoa框架提供很多字符串常量的例子，例如：

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

- 在实现文件中，NSString的实际值被指定为常量。（请注意，APPKIT_EXTERN宏经过计算之后是Objective-C中的extern关键字）。

异常和通告

异常和通告的名称遵循相似的命名规则，但是我们为二者推荐的使用模式并不相同。

异常

尽管您可以随意地将异常（即NSException类和一些相关联的函数所提供的机制）用于任何目的，但是通常情况下，cocoa不会利用他们来处理常规的、可预期的错误条件。这类错误应使用诸如nil、 NULL、 NO这样的返回值或者错误码来表示。通常，Cocoa把异常用于表示诸如数组索引越界这样的编程错误。

异常使用全局的NSString对象来标识，其名称按如下的方式进行组合：

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

异常名称中的具有唯一性的那部分，其组成词应该拼写在一起， 并且每个单词的首字符要大写。下面是一些例子：

```
NSColorListIOException

NSColorListNotEditableException

NSDraggingException

NSFontUnavailableException

NSIllegalSelectorException
```

通告

如果某个类含有委托，则通过所定义的委托方法，类的委托可以收到大部分通告。通告的名称应该反映相应的委托方法。例如，一旦应用程序发送一则`NSApplicationDidBecomeActiveNotification`的通告，则全局NSApplication对象的委托就会自动进行注册，这样它就可以接收到一条`applicationDidBecomeActive:`的消息。

通告使用全局的NSString对象进行标识，其名称按如下的方式组合：

```
[Name of associated class] + [Did | Will] + [UniquePartOfName] + Notification
```

例如：

```
NSApplicationDidBecomeActiveNotification

NSWindowDidMiniaturizeNotification

NSTextViewDidChangeSelectionNotification

NSColorPanelColorDidChangeNotification
```

可以使用的缩略名称

设计编程接口时，通常不应使用名称缩写（请参考 “通用规则”一节）。但是下列缩写可以继续用，因为它们要么已得到广泛认可，要么从过去就开始使用了。另外，请注意下面两件事情，它们也跟名称缩写有关：

- 标准C库里已使用很长时间的缩写格式—例如，“alloc”和“getc”—可以复制到Cocoa中。
- 参数名称可以更加随意地使用缩写（例如，“imageRep”、 “col” (表示“column”)、“obj”、 以及“otherWin”）。

缩写	意义和注释
alloc	分配内存
alt	可选的
app	应用程序。例如，NSApp表示全局的应用程序对象。但是在委托方法中，需要把“application”整个拼写出来。
calc	计算
dealloc	释放内存
func	函数
horiz	水平的
info	信息
init	初始化（表示初始化新对象的方法）
int	整数
max	最大值
min	最小值
msg	消息
nib	Interface Buidler档案

pboard	粘贴板（只能粘贴常量）
rect	矩形
Rep	表现形式（用于诸如NSBitmapImageRep这样的类）
temp	暂时性
vert	垂直

您可以用计算机行业中很常见的缩略语来代替其所表示的单词。下面是一些知名度比较高的简写：

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

框架开发者可以使用的技巧和技术

相对于其他的开发者而言，框架开发者要更加注意编写代码的方式。因为许多客户应用程序可能链接到框架，而这样宽泛地暴露接口，就导致框架的任何缺点都可能通过系统放大。下面的条款讨论一些框架编程技术，框架开发者可以利用它们来确保框架的高效性和完整性。

请注意：此处所讨论的一些技术并不局限于框架开发。将之用于应用程序开发同样卓有成效。

初始化

下述意见和建议涵盖框架初始化方面的内容。

类的初始化

`initialize`类方法中的代码只执行一次，它是类里面第一个被调用的方法。我们通常利用该方法来设置类的版本号（请参考"版本化和兼容性"一节）。

对于继承链中的每一个类，不论其是否实现`initialize`方法，运行时都会向它发送`initialize`消息。这可能导致一个类的`initialize`方法被多次调用（举个例子,如果子类没有实现`initialize`方法,则其父类的方法将被调用两次）。但通常您希望始化代码仅执行一次，为确保如此，您可以执行如下检查：

```
if (self == [NSFoo class]) {

    // the initializing code

}
```

您不应该显式地调用`initialize`方法。如果需要触发初始化行为，则请调用一些无害的方法，例如：

```
[NSImage self];
```

指定初始化函数

指定初始化函数是类的一个`init`方法，它调用超类的某个`init`方法（其他的初始化函数调用类自己定义的`init`方法）。每个公共类都应包含一个或多个指定初始化函数。举些例子，NSView的`initWithFrame:`方法以及NSResponder的`init`方法都是指定初始化函数。在某些情况下，类的`init`方法并不想被重载，比如NSString和其他面向类簇的抽象类，因而其子类应该实现自己的初始化方法。

您应该明确标示出指定初始化函数，因为该信息对于想根据您的类来派生子类的开发者有重要的意义。一个子类可以只重载指定初始化函数，这对其他所有初始化函数没有影响，它们仍将按其原先设计的行为工作。

在实现一个框架类时，您经常需要为其实现诸如`initWithCoder:`以及`encodeWithCoder:`这样的归档方法。请注意，对象解档时未发生的事情不要放在初始化代码路径里执行。对于实现归档功能的类，我们有一个比较好的方法可以做到这一点，那就是在类的指定初始化方法以及`initWithCoder:`方法（该方法也是个指定初始化函数）中调用一个公共的例程。

初始化过程中的错误检测

为确保能够恰当地检测并传播错误，一个设计良好的初始化方法应完成如下步骤：

1. 调用`super`的`init`方法对self重新赋值。
2. 检测指定初始化方法的返回值是否为`nil`。返回值为`nil`表明超类的初始化过程出现错误。
3. 如果当前类在初始化的过程中出现错误，则请释放对象并且返回`nil`值。

列表1 描述的方法可以完成上述步骤。

列表1 初始化过程中的错误检测

```
- (id)init {
```

```
    if ((self = [super init]) != nil) {   // call a designated initializer here

        // initialize object  ...

        if (someError) {

            [self release]; // [self dealloc] or [super dealloc] might be

            self = nil;     // better if object is malformed

        }

    }

    return self;

}
```

版本化和兼容性

在向框架添加新类或新方法时，您通常没必要为每个新功能群指定新的版本号。因为一般情况下，开发者会执行（或者说，应该执行）诸如`respondsToSelector:`这种Objective-C的运行时检测来判断给定的系统是否存在某种功能。开发者比较喜欢使用这种方式来检测新功能，同时它也是最动态的方式。

不论如何，您可以使用数种技术以确保新版本的框架能被正确标志并尽可能地兼容早期版本。

框架的版本

如果已存在的新功能或者错误改正不容易通过运行时进行检测，则您应该为开发者提供检测这些变更的办法。有一种办法是把确切的框架版本号保存起来，然后让该号码对开发者可见：

- 把变更归档在一个版本号下面（例如，归档在发布记录中）。
- 设置框架的当前版本号并且提供某种方法使之全局可见。您可以把版本号保存在框架的信息属性列表（`Info.plist`），这样就可以从该列表获取版本号。

基于键的归档

如果框架对象需要被写入到nib文件，则它们必须能够自我归挡 。另外，如果文档使用归挡机制来保存文档数据，则您也要对它们做归挡。在归挡时，您可以使用“老风格”（利用`initWithCoder:`和`encodeWithCoder:`这样的方法）。但是，为了更好地兼容过去、现在、以及未来的框架版本，您应该基于健进行归档。

基于键进行归档，对象就可以使用键来读取或写入被归挡值。相对以往的归挡机制，该方法可以在前向和后向兼容性上提供更多的灵活性。因为老归档机制要求代码和读取或写入的值维持相同的顺序，而且它也没有什么好办法来改变已写到档案的数据。如果您需要了解更多基于键的归档机制，请参考 Cocoa归档和序列化编程指南 （Archives and Serializations Programming Guide for Cocoa）一文。

对于正在编写的新的类，请为其使用基于键的归档机制。如果之前已发布的类使用了老归档机制，您也无需再采取任何措施。如果对象实现了Mac OS X 10.2版本之前的归档机制，则它必须能从档案中读取内容并能其内容写入到档案。但如果您在Mac OS X v10.2及之后的平台上为该对象添加新属性，则您不必，实际上是不应该，将这些属性保存到老档案中（这样做可能会使老档案在更早的系统中变得不可读取），这种情况下，新属性应使用键值归档机制。

请注意下列和基于键归档相关的事实：

- 如果档案中的某个键丢失，则在获取这个键对应值的时候，依据所要求的类型，其返回值可能是`nil`、 `NULL`、`NO`、0、或者0。通过对该返回值进行测试，您可以减少写到档案中的数据。同时，还可以检测某个键是否已被写入到档案中。
- 如果使用旧式归档，则`initWithCoder:`的实现需要独自挑起兼容性的重担。但如果使用键值归档，则归档方法和解档方法都可以采取一些措施以保证兼容性。举个例子，一个新版本的类的归档方法可能使用键来写入一个新值，而依旧把早期的字段写入到档案中，这样类的旧版本仍然可以理解该对象。与此同时，我们还可以在解档方法中使用某种合理的方式以处理数值缺失的情况，从而为将来的版本保留一些灵活性。
- 在命名框架类的档案键时，我们提倡使用和框架其他API元素一样的前缀，前缀后面再使用实例变量名称。您只要确保它的名称不会和任意的子类或者超类名称发生冲突即可。
- 如果您使用一个工具函数向档案中写入一个基本数据类型（换句话说，就是一个非对象的值），则请务必为该数值使用一个唯一键。举个例子，如果您有一个对矩形归档的“archiveRect” 例程，则您应该为该函数传入一个键作为参数。您可以直接把它作为档案键；或者，如果这个例程向档案写入多个值（例如，写入四个浮点数值），则它应该把每个数值自身独有的位添加到所提供的键上面。
- 位字段对于编译器和比特序有依赖关系，按照位字段现有的格式进行归档可能会有危险。只有当有多个位元需要被写入档案多次时，我们才会对位字段进行归档，这主要是为了提高性能。请参看 “位字段”一节 以获取相关的建议。

对象的尺寸和保留字段

每个Objective-C的对象都有一个尺寸，它由对象自身实例变量加上对象所有超类具有的实例变量得到的总尺寸决定。如果改变了一个类的尺寸，则其拥有实例变量的子类必须重新编译。为了保持二进制兼容性，通常情况下，我们不能通过向类添加新的实例变量或者去除类里面不必要的实例变量来改变对象的尺寸。

因此，对于新的类，为其留下几个额外的”保留“字段以便于将来扩展是个不错的想法。如果一个类只会有少数的几个实例，这个想法显然不成问题。但如果这个类会被实例化数千次，则您可能需要让保留的单个变量的尺寸小一些（也就是说，任意对象都占用四个字节）。

对于较早的类的对象，如果它们的空间已经用完（并且假定实例变量没有被导出成为公共变量），则您可以移动实例变量，或者把它们捆绑在一起,使之成为一个更小的字段。通过对实例变量进行重新排布，您就有可能添加新的数据而不会导致对象的总尺寸发生改变。或者您可以把一个剩余的保留槽作为指针，使它可以指向一块额外的内存，然后您在对象初始化的时候分配这块内存（并且在对象释放的时候销毁它）。又或者您可以把额外的数据放入到一张外部的哈希表（例如放入NSDictionary）；这种方法对于那些很少创建使用的实例变量具有很好的效果。

异常和错误

大多数Cocoa框架的方法不会强制要求开发者捕捉处理异常，因为程序正常执行时不会产生异常，而且我们通常也不使用异常来表示可预期的运

行时错误或用户错误。下面这些例子属于可预期的运行时错误或者用户错误：

- 文件找不到
- 用户不存在
- 试图打开应用程序中一个错误类型的文档
- 把字符串转化为特定编码时出现错误

但是不管怎么样， Cocoa确实会引发异常以指示下面这些编程错误或者逻辑错误：

- 数组索引越界
- 试图改变一个不可改变的对象
- 错误参数类型

我们认为应用程序推向市场之前，开发者会对其进行测试，发现并解决这些类型的错误。因此，应用程序不需要在运行时处理上述异常。如果出现一个异常发生但应用程序没有捕捉，则最顶层的缺省处理器通常会捕捉并且报告该异常，然后程序将继续执行。开发者可以选择替换掉这个缺省的异常处理器，新的处理器可以更详细地描述什么地方发生了错误，并且还能够让用户选择是否保存数据并且推出应用程序。

错误是Cocoa框架不同于其他软件库的又一个地方。Cocoa的方法通常不返回错误码。某些情况下，一个错误具有一个合理或者可能的原因，方法通过对一个布尔值或者对象返回值（nil/non-nil）进行简单测试以判断该情况；但是返回NO或者nil值的原因则被记录在文档中。另外，您不应该使用错误码来指示需要在运行时处理的编程错误，相反您应该引发一个异常，或者在某些情况下，您也可以只记录下该错误而不引发异常。

举个例子，NSDictionary的objectForKey:方法会返回所找到的对象，如果对象找不到，则它会返回nil值。NSArray的objectAtIndex:法不可能返回nil值（除非我们把通用的编程语言约定重载成向nil对象发送消息都会返回nil值），因为NSArray对象不能保存nil值，并且所有的越界访问都被定义成编程错误，这种错误引发异常而并不返回nil对象。如果对象不能使用用户提供的参数进行初始化，许多init方法都会返回nil值

在少数情况下，一个方法需要多个不同的错误码是合理的。这时，该方法需要将错误码指定到一个传引用参数，然后您可以利用该参数返回一个错误码，也可以返回一个本地化的错误字符串，或者一些其他的可以描述错误的信息。举个例子，您可以把错误转换成NSError对象返回（请参看Foundation框架的NSError.h头文件以获取更多的细节）。而除了这个NSError对象，方法还能直接返回相对简单的BOOL值或者nil值。另外要注意，这种方法的所有传引用参数都是可选的。因此，如果发送者不想了解错误原因，它们可以传一个NULL 值给错误码参数。

重要： NSError类在Mac OS X v10.3之后的版本对外公开。

框架数据

框架数据的处理方式可能会对性能、跨平台兼容性、以及其他某些方面产生影响。本节讨论一些和框架数据相关的技术。

常量数据

出于性能的原因，您应尽可能多地把框架数据标志为常量，因为这样可以减小Mach-O二进制文件的__DATA 段的尺寸。没有const标记的全局或静态的数据最终会存放在__DATA段的__DATA节。对于每一个使用该框架的应用程序实例，这种类型的数据都会占用内存。尽管多出500字节（举个例子）好像没那么糟糕，但是这有可能导致应用程序所需的内存页面的数量增多—即每个应用程序都需要额外的四千字节的内存。

您应该为所有不变的数据添加const标记。具有const标记的数据块，如果其中没有char *类型的指针，则该数据会被存放在__TEXT段中（这会使数据成为真正的常量）； 否则数据就会被存放在__DATA段中，但是这些数据不可以被写入（除非预绑定未完成，如果预邦定已经完成，则只能在加载的时候，通过移动二进制文件的方式来写入）。

您应该初始化静态变量，这样可以确保该变量被合并到__DATA段的__data节中而非__bss节。如果没有明显的值用于初始化静态变量，则请使用0、NULL、0.0、或者任何恰当的值。

位字段

如果使用有符号的值来表示位字段，而代码又假定这个位字段是布尔值，则可能会导致未定义的行为。只有一个位的位字段尤为如此。因为在这种情况下，这个位字段只能存储0和 -1（取决于编译器的实现），把它和1做比较，其结果总是不相等。因此，只有一个位的位字段应该是无符号的。举个例子，如果您在代码中遇到如下情况：

```
BOOL isAttachment:1;


int startTracking:1;
```

您应该把上述代码中的类型改为unsigned int

和位字段相关的另一个问题是归档。通常情况下，您不应该按照当前格式将其写入到磁盘或者档案中，因为当我们在另外的架构或者编译器中读取这些字段时，它们的格式可能发生变化。

内存分配

在框架代码中，如果可以的话，最好是完全避免分配内存。如果出于某种原因，您需要一块临时的缓冲区，则通常情况下，使用栈比分配缓冲区更好。但是，栈的大小有限（栈总的大小通常为512千字节）， 因此是否使用栈取决于函数和您所需要的缓冲区的大小。通常情况下，如果您需要的缓冲区的大小不超过1000（或者是MAXPATHLE定义的值）字节，使用栈是合适的。

一个比较精细的方法是在开始的时候使用栈，但是如果所需要的内存的大小超过了栈缓冲区的大小，则切换到malloc内存分配方式。列表 2展示的代码片段就是这么做的。

列表 2 使用栈内存和通过malloc分配的缓冲区

```
#define STACKBUFSIZE (1000 / sizeof(YourElementType))

 YourElementType stackBuffer[STACKBUFSIZE];

 YourElementType *buf = stackBuffer;

 int capacity = STACKBUFSIZE;  // In terms of YourElementType

 int numElements = 0;  // In terms of YourElementType
```

```
while (1) {
    if (numElements > capacity) {  // Need more room
        int newCapacity = capacity * 2;  // Or whatever your growth algorithm is
        if (buf == stackBuffer) {  // Previously using stack; switch to allocated memory
            buf = malloc(newCapacity * sizeof(YourElementType));
            memmove(buf, stackBuffer, capacity * sizeof(YourElementType));
        } else {  // Was already using malloc; simply realloc
            buf = realloc(buf, newCapacity * sizeof(YourElementType));
        }
        capacity = newCapacity;
    }
    // ... use buf; increment numElements ...
  }
  // ...
  if (buf != stackBuffer) free(buf);
```

语言问题

下面的条款讨论和Objective-C语言相关的问题，包括协议、对象比较、以及向对象发送`autorelease`消息的时机。

向nil对象发送消息

在Objective-C中，只要消息的返回值是对象、任意的指针类型、或是其尺寸小于等于`sizeof(void*)`的整数型标量，我们就可以将其发送给`nil`对象，该消息将返回`nil`值。这个特性一种很有价值的编程资产，但是有一个问题需要注意。如果发送给`nil`对象的消息的返回值非上述类型（例如，消息返回任意的`struct`类型，或者浮点类型，或者任意的向量类型），则消息的返回值未定义。通常情况下，如果消息的返回值不是一个对象，则不要依赖这种行为，因为那是很糟糕的做法。在Power PC系统中，向`nil`对象发送上述类型的消息不会有问题，但对于其他架构来说，这种行为行不通。

对象比较

通用的对象比较方法`isEqual:`和与对象类型关联在一起的比较方法（诸如`isEqualToString:`）有一个重要的不同。`isEqual:`方法允许任何对象作为参数，如果用于比较的对象属于不同的类，则该方法将返回`NO`值。而诸如`isEqualToString:`以及`isEqualToArray:`这样的方法通常都假定参数是某种特定的类型（和接收者一样的类型）。因此，这种函数不会执行类型检查，这使得它们运行速度比普通的对象比较方法快，但是安全性则不如之。对于那些从外部源获取的值，例如从应用程序的的信息属性表（`Info.plist`）或者应用程序的偏好设置获取的值，在对其进行比较时，我们倾向于使用`isEqual:`方法，因为这种方法更安全；但如果我们已知道要进行比较的值的类型，则应使用`isEqualToString:`方法。

关于`isEqual:`方法，还有一点就是它和`hash`方法的联系。存放在基于散列的Cocoa集合类里的对象（例如存放在NSDictionary或者NSSet中的对象）有一个基本的不变式，即如果`[A isEqual:B] == YES`，则`[A hash] == [B hash]`也成立。因此，如果您重写了类的`isEqual:`方法，则您也应重写`hash`方法以保持不变式成立。缺省情况下，`isEqual:`方法判断对象地址指针是否相等，而`hash`方法则返回一个基于对象地址产生的hash数值，因此，这个不变式就可以保持成立。

协议

协议是Objective-C中一个有趣的概念，但是它们在Cocoa API中用得有限。之所以这样做，有一个原因是协议有严格的设计要求。以某个含有十个方法的NSDataSource协议为例。如果某个开发人员遵循该协议并且实现其所有方法，而之后您又向该协议添加了一个新的方法，这就会破坏开发者和原有协议的一致性。因此，协议往往仅能包含它们首次公开时所含有的方法集（除非您不期望其他开发者实现该协议）。因此，只有方法集不可能再增长时，才应使用协议。如果您必须扩展某个协议，则您应该添加一个新的协议，利用新协议来对原来的协议进行扩展，或者您可以把新方法添加在协议外部，并且在使用前检查它是否存在。

基本上，上述的原因也可以解释为何您不能使用常规协议来声明委托方法。而把委托方法声明为NSObjec上的范畴类— 即非正式的协议— 的另一个原因是我们可以选择是否实现某个方法。

自动释放的对象

返回对象值的方法或者函数，如果它们不是用于创建对象或者复制对象(`new`,`alloc`,copy 以及这些方法的变体)，则请务必让其返回的对象可以`autoreleased` 。在此处，“自动释放”并不一定表示对象应该显式自动释放-即在返回对象之前，向对象发送`autorelease` 消息。一般情况下，它仅表示返回值不是由调用者释放。

出于性能方面的原因，请尽可能不要在方法实现中使用自动释放的对象，特别是那些可能会在短时间内频繁执行的代码（例如在循环结构中，循环次数未知并且可能是很多的情况）使用。举个例子，对于下面的情况，您不应该发送如下的消息：

```
[NSString stringWithCharacters:]
```

而是应该发送下面的消息：

```
[[NSString alloc] initWithCharacters:]
```

当您不再需要该字符串对象时，请显示地释放它。但是，请记住有些时候方法或者函数会返回自动释放的对象，这时您需要向对象发送`autorelease`消息。

存取方法

在存取方法中作什么事情才是正确呢，这是个重要的问题。举个例子，假设您在一个获取方法中直接返回一个实例变量，而后立刻调用设置方法，则在释放先前实例变量的时候就可能会有危险，因为它有可能把您之前返回的值给释放掉。Cocoa框架的原则是让设置方法自动释放实例变量先前的值,但某些情况下,所涉及的某个设置方法会被频繁的调用（例如在一个很密集的循环中），这时候Cocoa的原则就不适用了。但在实践中，这种情况非常罕见，除非是一些底层的对象才会如此。另外，诸如NSAttributedString、NSArray、以及NSDictionary这样的通用集合不会自动释放对象，这主要是为了维护对象的存在时间。它们只是简单地留存或者释放其含有的对象。另外，它们也应该对这一事实进行归档，这样客户程序就可以了解这些对象的行为。

对于现在正在编写的框架代码， 我们建议在get方法中使用自动释放的对象，因为这是最安全的方法：

```
- (NSString *)title {

    return [[instanceVar retain] autorelease];

}



- (void)setTitle:(NSString *)newTitle {

    if (instanceVar != newTitle) {

        [instanceVar release];

        instanceVar = [newTitle copy];

        // or retain, depending on object & usage

    }

}
```

另外，我们还需要考虑设置方法是使用copy方式还是使用retain方式。如果您所感兴趣的是对象的值而非实际对象本身，则请使用`copy`方式。一个一般性的经验法则是对实现NSCopying协议的对象使用`copy`方式（您不应该在运行时检测对象是否实现NSCopying协议，而应该直接查找参考文档）。通常情况下,诸如字符串、颜色、URL这样的对象应该能被复制；而像视图、窗口这样的对象则应该可以被保持。而至于其他的对象（例如数组），是使用`copy`还是使用`retain`，则要根据具体情况决定。