# 代码命名基础

在面向对象的语言的使用过程中，我们经常忽略的一个点就是对类、方法、函数、常量和编程接口的其他元素的命名。本节主要讲述使用 cocoa 接口的大部分项目的通用命名约定。

## 一般原则

### 清晰

- 我们在编写过程中要尽可能的简单明了，但是不能因为简洁而使得清晰性受损。

| Code | Commentary |
| --- | --- |
| `insertObject:atIndex:` | Good. |
| `insert:at:` | Not clear; what is being inserted? what does “at” signify? |
| `removeObjectAtIndex:` | Good. |
| `removeObject:` | Good, because it removes object referred to in argument. |
| `remove:` | Not clear; what is being removed? |

- 一般情况下，不要对事物的名称进行缩写。即使名称有很长，我们也要把它们全部拼写出来。

| Code | Commentary |
| --- | --- |
| `destinationSelection` | Good. |
| `destSel` | Not clear. |
| `setBackgroundColor:` | Good. |
| `setBkgdColor:` | Not clear. |

  有时候我们可能会认为有些缩写是众所周知的，但是当开发人员具有与我们不同的文化和语言背景时，他在看待我们的方法名和函数名的时候就会感觉到其他不同的意思。

- 但是有一些缩写又真的非常的常见而且具有很长的使用历史，那么我们可以继续使用它。具体请看 Acceptable Abbreviations and Acronyms。
- 在 API 的命名过程中，不要使用含糊不清的名称。就比如说一个名称可能会有多种不同的解读。

| Code | Commentary |
| --- | --- |
| `sendPort` | Does it send the port or return it? |
| `displayName` | Does it display a name or return the receiver’s title in the user interface? |

## 一致性

- 在编程过程中，我们需要使用与编程接口中提供的名称保持一致。如果我们不确定该名称的话，可以浏览当前头文件或参考文档来进行确定。
- 当我们需要利用类的多态特性时，一致性会显得尤为重要。如果我们想在不同的类中执行相同操作的方法时，那我们就必须保证它们有相同的名称。

| Code | Commentary |
| --- | --- |
| `- (NSInteger)tag` | Defined in `NSView`, `NSCell`, `NSControl`. |
| `- (void)setStringValue:(NSString *)` | Defined in a number of Cocoa classes. |

## 不要以自我为参照

- 命名时不要自引用

| Code | Commentary |
| --- | --- |
| `NSString` | Okay. |
| `NSStringObject` | Self-referential. |

- 那些值为掩码的常量(使用掩码可以使它们根据实际情况进行按位组合)是一个例外，就像通知名称的常量一样。
    -

| Code | Commentary |
| --- | --- |
| `NSUnderlineByWordMask` | Okay. |
| `NSTableViewColumnDidMoveNotification` | Okay. |

# 前缀

前缀是编程接口中名称命名的一个重要组成部分。它们用于区分软件的功能领域。通常这个软件会被打包在一个框架中或（如 macOS 中的 Foundation 框架和 AppKit 框架，前缀都是 NS ）在紧密相关的框架中。前缀可以防止在第三方开发者定义的符号与 Apple 定义的符号（以及 Apple 自己框架中的符号）之间产生冲突。

- 前缀有其固定的格式。它由两个或三个大写字母组成，不要使用下划线或者 ”子前缀” 。下面是一些例子：

| Prefix | Cocoa Framework |
| --- | --- |
| NS | Foundation |
| NS | Application Kit |
| AB | Address Book |
| IB | Interface Builder |

- 在命名类、协议、函数、常量和结构体的时候要使用前缀。千万不要在命名方法的时候使用前缀；方法存在于定义他们的类所创建的命名空间中。同时也要注意，不要用前缀来命名结构体的字段。

# 一些代码书写约定（排版约定）

在命名 API 的时候我们需要遵循一些简单的约定：

- 对于由多个单词组成的名称，不要将标点符号或者分隔符作为名称的一部分（下划线，破折号等）；取而代之的是，我们需要将每个单词的首字母大写，然后将所有的单词直接拼在一起（例如：runTheWordsTogether ）—— 这就是所谓的 驼峰命名 。但是，我们要注意以下的几种情况：
  - 命名方法的时候，我们需要以小写字母开头（将第一个单词的首字母小写），然后将其他的嵌入单词的首字母大写，不要使用前缀。

    ```
    fileExistsAtPath:isDirectory:
    ```

    这条约定有个例外，就是在命名那些以众所周知的首字母缩略词作为开头的方法名的时候，例如， TIFFRepresentation ( NSImage )。
  - 对于函数和常量的命名，我们需要使用与相关类相同的前缀，并将嵌入单词的首字母大写。

    ```
    NSRunAlertPanel
    NSCellDisabled
    ```

- 我们需要避免使用下划线作为方法名称中的前缀（使用下划线作为实例变量的前缀是可以的）。苹果保留了使用下划线作为方法前缀的权利。第三方在使用下划线作为方法前缀的过程中会导致命名空间冲突，他们可能无意中重写了现有的私有方法，并带来了灾难性的后果。我们可以查看私有方法以获取有关私有 API 遵循约定的建议。

# 类和协议名称

类的名称应该包含一个名字，这个名词应该能清楚的表达类（或者类的对象）代表什么或者是能做什么。类的名称需要一个适当的前缀（Prefixes）。Foundation 和 Application 框架中有很多例子；一些有代表性的就是 NSString，NSDate，NSScanner，NSApplication，UIApplication，NSButton 和 UIButton。

协议应该根据他们的群体行为进行命名：

- 大多数协议中定义的方法与其他类并没有什么直接的关联关系。这种类型的协议在命名的时候需要考虑到不能与类有混淆关系，很常见的一种约定就是使用动名词（ ing ）形式。

| Code | Commentary |
| --- | --- |
| NSLocking | Good. |
| NSLock | Poor (seems like a name for a class). |

- 一些协议将许多不想关的方法组合在了一起（而不是分成几个独立的小协议）。这些协议通常与协议的表达式的类相关联。在这些情况下，我们可以给予协议一个与类相同的名称。

这种协议的一个例子就是 NSObject 协议。这个协议将查看对象的类的层级、执行特定的方法和增加或减少引用计数组合了起来。因为 NSObject 类提供了这些方法的主要表达式，所以该协议以 NSObject 类的名称命名。

# 头文件

如何命名头文件非常重要，因为我们的命名规则会表明该文件包含的内容:

- 一个独立的类或协议的头文件命名。如果某个类或协议不属于某个功能组，则将其声明放到一个单独的文件中，该文件的名称是声明的类或协议的名称。

| Header file | Declares |
| --- | --- |
| `NSLocale.h` | The `NSLocale` class. |

- 相关的类和协议的头文件命名。对于一组相关的声明（类，类别和协议），我们需要将声明放到一个带有主类，类别或协议名称的文件中。

| Header file | Declares |
| --- | --- |
| `NSString.h` | `NSString` and `NSMutableString` classes. |
| `NSLock.h` | `NSLocking` protocol and `NSLock`, `NSConditionLock`, and `NSRecursiveLock` classes. |

- 包含框架中所有功能头文件的头文件命名。每个框架 ( framework ) 都应该有一个头文件，该文件以框架命名，其中包含了框架中所有需要用到的头文件。

| Header file | Framework |
| --- | --- |
| `Foundation.h` | `Foundation.framework`. |

- 为另外一个框架中的类添加 API 时头文件的命名。

```
Adding API to a class in another framework. If you declare methods in one framework that are in a category on a class in another framework, append “Additions” to the name of the original class; an example is the NSBundleAdditions.h header file of the Application Kit.
//如果我们需要为另外一个框架中的某个类添加一些 API 。我们需要创建一个该类的分类，并在分类中声明相关拓展的方法，我们在定义头文件的时候需要在该类的类名后加入一个 “additions” 来作为头文件名。一个例子就是 Application Kit 框架中的 NSBundleAdditions.h 头文件。
```

上面是官方文档中的描述。假如 A 框架中有个 B 类。现在 B 类不能满足我们的业务需求，我们需要为 B 类新增一些 API。我们通常会采用分类的手段，为 B 类拓展一些 API 。这个时候，我们需要用一个名为 BAdditions 的头文件来包含这些对 B 类的拓展。

- 对一些相关函数和数据类型的整合头文件的命名。如果我们有一组相关的函数、常量、结构和其他数据类型，我们需要将他们放到一个适当命名（功能类型）的头文件中，比如 Application Kit 框架中的 NSGraphics.h 头文件。

# 方法命名

方法是我们编程接口中最常见的元素，因此应该特别注意如何命名他们。本节讨论方法命名的以下规则：

## 一般规则

命名方法的时候有一些通用规则：

- 用小写字母开始，并将其他嵌入单词的首字母大写。不要使用前缀，可以参考Typographic Conventions.（也可以看上面的代码约定）

  这个准则有两个例外情况，我们可以使用一些众所周知的大写首字母缩写（ TIFF，PDF ）作为方法的开头，并且我们可以使用前缀来对私有方法进行分组和标识（可以参考 Private Methods ）。

- 对于那些表示对象行为的方法，要以动词开头

  ```
  - (void)invokeWithTarget:(id)target;
  - (void)selectTabViewItem:(NSTabViewItem *)tabViewItem
  ```

  不要使用 do 和 does 作为方法名的一部分，因为这些辅助动词基本不会增加什么实际上的意义。另外，在动词之前不要使用辅助动词和形容词。

- 如果一个方法返回的是接收者的属性，直接以属性名命名该方法。除非是间接的返回一个或者多个值，否则 “get” 是不必要写的。

  | **- (NSSize)cellSize;** | **Right.** |
  | --- | --- |
  | - (NSSize)calcCellSize; | Wrong. |
  | - (NSSize)getCellSize; | Wrong. |

  可以参考 Accessor Methods.

- 在所有参数前使用关键字。

  | **- (void)sendAction:(SEL)aSelector toObject:(id)anObject forAllCells: (BOOL)flag;** | **Right.** |
  | --- | --- |
  | - (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag; | Wrong. |

- 在参数的描述之前对参数进行一个简单的说明。

  | **- (id)viewWithTag:(NSInteger)aTag;** | **Right.** |
  | --- | --- |
  | - (id)taggedView:(int)aTag; | Wrong. |

- 当我们需要创建一个比继承下来的方法更具体的方法时，我们需要将新加的参数放到原有方法的末尾。

| **- (id)initWithFrame:(CGRect)frameRect;** | **NSView, UIView.** |
| --- | --- |
| - (id)initWithFrame:(NSRect)frameRect mode:(int)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide; | NSMatrix, a subclass of NSView |

- 不要使用 “and” 来链接属于接受者属性的关键字。

| **- (int)runModalForDirectory:(NSString *)path file:(NSString *) name types:(NSArray *)fileTypes;** | **Right.** |
| --- | --- |
| - (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andTypes:(NSArray *)fileTypes; | Wrong. |

  尽管在这个例子中，使用 “and” 可能听起来还蛮不错的，但当我们使用越来越多的关键字创建方法时，会导致一些问题（语义上的不明确...）。

- 如果该方法描述了两个不同的动作，那么我们可以使用 “and” 来连接他们。

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

# 存取器（访问器，set 和 get）方法的命名

存取器方法是那些设置和返回对象属性的方法。他们有一定的推荐形式，具体要看这个属性的表达方式来定。

- 如果这个属性是个名词，格式为:

  `- (type)noun;`

  `- (void)setNoun:(type)aNoun;`

  例如：

```
- (NSString *)title;
- (void)setTitle:(NSString *)aTitle;
```

- 如果这个属性是个形容词，格式为：

  `- (BOOL)isAdjective;`

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

  例如：

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

- 如果这个属性是个动词，格式为：

  `- (BOOL)verbObject;`

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

  例如：

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

  动词应该是简单的现在时。

- 不要用过去分词将动词转换成形容词：

| **- (void)setAcceptsGlyphInfo:(BOOL)flag;** | **Right.** |
| --- | --- |
| - (BOOL)acceptsGlyphInfo; | Right. |
| - (void)setGlyphInfoAccepted:(BOOL)flag; | Wrong. |
| - (BOOL)glyphInfoAccepted; | Wrong. |

- 我们可以使用情态动词（以 can ，should， will 等为前缀的动词）来使我们表达的意思更加清晰，但是不要使用 do 或者 does 。

| **- (void)setCanHide:(BOOL)flag;** | **Right.** |
| --- | --- |
| - (BOOL)canHide; | Right. |
| - (void)setShouldCloseDocument:(BOOL)flag; | Right. |
| - (BOOL)shouldCloseDocument; | Right. |
| - (void)setDoesAcceptGlyphInfo:(BOOL)flag; | Wrong. |
| - (BOOL)doesAcceptGlyphInfo; | Wrong. |

- 仅对间接返回对象和值的方法使用 “get” 。只有当返回多个值的情况下，我们才有必要使用这条规则。

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

在诸如此类的方法中，这些输入输出参数的实现应该可以接受值为 NULL 的入參，来表明，调用者并不关心这些属性是不是真的有返回值。

# 委托方法的命名

委托方法指的是某个对象在某些事件发生的时候，在其委托中调用的方法（如果委托实现了他们）。他们具有独特的形式，一个很好的例子就是对象数据源中的方法。

- 通过一个能表明触发者类型的名称做开头：

```
- (BOOL)tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;
```

  这里的类型要省略前缀，并且首字母要小写。

- 除非该方法只有一个参数，触发者，否则冒号应该紧跟在类名的后面（这个参数是对触发者的引用）。

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

- 对此的例外就是通过接受通知而调用的方法。这种情况下，方法唯一的参数是 NSNotification 对象。

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

- 被调用的方法可以使用 “did” 或 “will” 来通知委托某些事情已经发生或者即将发生。

```
- (void)browserDidScroll:(NSBrowser *)sender;
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)window;
```

- 有些情况下，尽管我们可以在方法名称中插入 “did” 或 “will” 来告诉委托对象去做一些事情，但是这种情况最好用 “should”。

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

# 集合方法的命名

对于那些用来管理对象集合（集合中的每个对象称为该集合的元素）的对象，必须拥有以下格式的方法：

`- (void)addElement:( elementType )anObj;`

`- (void)removeElement:( elementType )anObj;`

`- (NSArray *)elements;`

例如：

```
- (void)addLayoutManager:(NSLayoutManager *)obj;
- (void)removeLayoutManager:(NSLayoutManager *)obj;
- (NSArray *)layoutManagers;
```

以下是对该指南的一些条件限定和改进：

- 如果集合确实是无序的，那么返回值应该是 NSSet 类型，而不是 NSArray 类型。
- 如果将元素插入到集合中的指定位置很重要，我们应该使用跟以下类似的方法进行代替：

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

下面是关于集合方法命名的一些需要牢记在心的实现细节：

- 这些方法通常意味着插入对象的所有权，因此那些添加或者插入对象的代码必须要持有该对象，删除对象的代码也必须要释放对象。(如果我们在添加或插入时不持有对象的话，万一对象在别的地方被释放，就会丢失对该对象的所有权，如果我们持有了，却不在删除的时候释放，就会导致内存泄漏，因为该对象一直没有被销毁)
- 如果插入的元素需要有一个指向集合对象的指针，那么我们通常会使用 set... 方法来为元素设置一个指向集合对象但不持有它的指针。在 `insertLayoutManager:atIndex:` 方法中，NSLayoutManager 类就用到了这个方法。

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

  我们通常不会直接调用 `setTextStorage:` ，但有可能需要重写它。

上述手机方法命名的另一个典型的例子就是 NSWindow 类：

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

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

# 方法参数的命名规则

有几个关于方法参数命名的基本规则：

- 与方法命名一样，参数需要以小写字母开头，其他连续单词的首字母需要大写（例如 removeObject:(id)anObject ）。
- 不要在参数名中使用 pointer 或 ptr ，参数是否是一个指针要通过它的类型而不是名称来表示。
- 避免使用单字母或者双字母参数名。
- 避免使用几个字母的缩写来命名。

在传统上，以下关键字和参数名需要一起使用：

```
...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
```

## 私有方法命名

在大多数情况下，私有方法的命名通常会遵循公共方法的命名规则。但是，为了对公共方法和私有方法进行区分，通常约定是对私有方法加一个前缀。但是即使有了这个约定，在私有方法的命名上也会出现一些特殊的问题。当我们在设计 Cocoa 框架类的子类的时候，我们并不知道我们在子类中声明的私有方法是否无意中覆盖掉了具有相同名称的私有框架方法。

Cocoa 框架中的大部分私有方法都会用下划线 _ 做前缀（ _fooData ）来标记其私有性。从这个事实上来看，有两个建议：

- 不要使用下划线来做我们私有方法的前缀，苹果保留用 _ 来标记私有方法的使用权。
- 如果我们继承了一个很庞大的 Cocoa 框架类，并且我们想要保证自己的私有方法的名称与父类中的不同，我们可以将我们自己的前缀加入到我们的私有方法的命名中。前缀应该尽可能的唯一，可能是一个基于公司或项目或“XX_”格式的前缀。如果我们的项目被定义为 Byte Flogger ，那么加了前缀之后的方法可能是 `BF_addObject:`。

尽管为私有方法的名称提供前缀的建议可能与先前声明方法存在于其类中的命名空间中相互矛盾，但这里的意图是不同的，我们的目的是为了防止无意中重写父类方法。

# 函数命名：

Objective-C 允许我们通过函数和方法来表达我们的行为。当我们需要使用单例作为基础对象来实现一些通用的功能子系统的时候，我们应该使用函数而不是类方法。

函数同样有一些需要我们遵循的命名规则：

- 函数命名大体上和方法命名差不多，但是有一些区别：
  - 它们需要以类和常量相同的前缀开始
  - 前缀后的单词需要将其首字母大写
- 大多数的函数名需要以描述函数作用的动词作为开头：

```
NSHighlightRect
NSDeallocateObject
```

对于那些用来查询属性的函数，有另外一组命名规则：

- 如果该函数返回的是第一个参数的属性，需要省略动词。

```
unsigned int NSEventMaskFromType(NSEventType type)
float NSHeight(NSRect aRect)
```

- 如果返回的值是个引用类型，那么需要使用 “Get”。

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

- 如果返回值是个布尔值，函数需要以一个间接的动词做开头（If the value returned is a boolean, the function should begin with an inflected verb）

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

# 命名属性和数据类型

本节介绍声明属性、实例变量、常量、通知和异常的命名规范。

## 属性和实例变量的声明

一个已声明的属性会有效地声明该属性的存取器方法，因此属性的命名规范跟存取器方法的命名规范大体相同（Accessor Methods）。如果该属性表示为名词或者动词，那么格式是：

@property (...) type nounOrVerb;

例如：

```
@property (strong) NSString *title;
@property (assign) BOOL showsAlpha;
```

如果一个已声明的属性表示为形容词，那么这个属性的名称需要省略 is 前缀，但是 get 访问器的名称需要加 is 前缀：

```
@property (assign, getter=isEditable) BOOL editable;
```

大多数情况下，我们使用一个已声明的属性的时候，我们可以为其合成一个相应的实例变量。

我们要确保实例变量的名称可以简洁地将存储的属性描述出来。通常，我们不应该直接访问实例变量，相反，我们应该使用存取器方法（但是在 init 和 dealloc 方法中我们应该直接访问实例变量）。为了更好地区分实例变量和属性，我们需要给实例变量加一个下划线前缀(_)。例如：

```
@implementation MyClass {
    BOOL _showsTitle;
}
```

如果要为已声明的实例变量合成一个实例变量，我们需要在 @synthesize 语句中指定实例变量的名称。

```
@implementation MyClass
@synthesize showsTitle=_showsTitle;
```

为类添加实例变量的时候，我们需要注意一些注意事项：

- 避免显示地声明公共的实例变量。

  开发人员应该关注对象的接口，而不是他们存储数据的细节。我们可以通过使用属性，并为属性合成相应的实例变量来避免声明公共实例变量。

- 如果我们需要声明一个实例变量，我们需要通过 @private 和 @protected 来显示地声明它。

  如果我们已经预料到这个类会有子类，并且子类中会直接访问父类的数据，我们应该使用 @protected 指令。

- 如果实例变量是该类的实例的可访问属性，那么我们需要为这个实例变量声明存取器方法（如果可能的话，尽量使用属性）。

# 常量

常量的规则会根据常量的创建方式而有所不同。

## 枚举常量

- 将枚举用于具有整数值的相关常量组。
- 枚举常量和 typedef 需要按照函数命名规范（ Naming Functions）进行分组。下面的示例来自 NSMatrix.h：

  ```
  typedef enum _NSMatrixMode {
      NSRadioModeMatrix           = 0,
      NSHighlightModeMatrix       = 1,
      NSListModeMatrix            = 2,
      NSTrackModeMatrix           = 3
  } NSMatrixMode;
  ```

  需要注意的是，typedef 标签（上面例子中的 _NSMatrixMode ）可以忽略。

- 我们可以为位掩码等内容创建未命名的枚举，比如：

  ```
  enum {
      NSBorderlessWindowMask      = 0,
      NSTitledWindowMask          = 1 << 0,
      NSClosableWindowMask        = 1 << 1,
      NSMiniaturizableWindowMask  = 1 << 2,
      NSResizableWindowMask       = 1 << 3
  };
  ```

## 用 const 创建的常量

- 使用 const 来创建一个浮点值常量。我们可以使用 const 来创建一个整型常量，前提是这个常量跟其他常量没有关系，否则的话，使用枚举会更好一些。
- const 常量的格式以下面的声明为例：

```
const float NSLightGray;
```

和枚举常量一样，用 const 创建的常量需要遵循函数命名规范（Naming Functions）。

## 其他类型的常量

- 原则上，不要使用 #define 预处理命令来创建常量。对于整数常量最好使用枚举，而对于浮点数常量最好使用 const 修饰词，如上所述。
- 对于那些预处理器用来评估是否需要处理代码块的符号，要用大写字母表示。例如：

```
#ifdef DEBUG
```

- 那些由编译器定义的宏具有前导和后导双下划线字符。例如：

```
__MACH__
```

- 我们需要为通知名称和字典键值定义字符串常量。对于字符串常量，编译器会验证其值的正确性（即执行拼写检查）。在 cocoa 框架中，提供的字符串常量的例子有很多，如：

```
APPKIT_EXTERN NSString * NSPrintCopies;
```

实际的 NSString 的值会在实现文件中被赋予一个常量（需要注意的是，在 Objective-C 中，APPKIT_EXTERN 等于 extern）。

```
//Q&A
Q:为什么我们需要为通知名称和字典键值定义字符串常量？不定义会有什么问题？
A:字典键值和通知名称会被重复利用，比如通知名称，post的时候我们会用，addObserver的时候我们也会用到，如果每次都执行硬编码的话出错率会有点高，所以使用字符串常量是一个比较好的方式。

Q:对于字符串常量，编译器会验证其值的正确性？有什么用？
A:
    我们可以想一下利用宏定义和创建常量这两种方式有什么区别。宏，只是一个代码替换的功能，在我们定义宏的时候，并不能确定值的正确性，可能我们需要的是个字符串，被赋予一个整型或者其他类型的话也不会报错，只有在真正执行代码的时候才会出现问题。这就会引发一些不容易发现的问题。（虽然这种情况不常见）。
    创建常量的话，首先我们需要声明常量的类型，进行赋值操作之后，编译器会执行代码检查，如果类型出错，编译器会报错，就能保证我们在使用的时候，永远拿到的都是我们需要使用的类型。
```

# 通知和异常

通知和异常的名称的命名规则大体上差不多，但两者都有自己的推荐使用模式。

## 通知

如果一个类有一个委托，那么它的大部分通知可能会被委托的委托方法进行接受。这些通知的名称应该反应相应的委托方法。例如：全局 NSApplication 对象的委托会自动注册以便于在发送 NSApplicationDidBecomeActiveNotification 通知的同时，委托会接受 applicationDidBecomeActive: 消息。

通知由全局 NSString 对象标示，其名称要以下面这种方式组成：

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

例如：

```
NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification
```

## 异常

尽管我们可以为了达成目的而自由的选择抛出哪种异常（ NSException 类和相关函数所提供的机制），但是 Cocoa 会保留编程错误导致的异常，例如数组索引超出范围。Cocoa 不会使用异常来处理常规的，预期的错误条件。对于这些情况，通过返回 nil，NULL，NO 或错误代码进行处理。更多细节可以参考 编程错误处理指南。

异常由全局 NSString 对象标示，其名称要以下面这种方式组成：

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

其中，UniquePartOfName 由几个首字母大写的相连单词组成。下面是一些例子：

```
NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException
```

# 可接受的缩略语和首字母缩略词

一般来说，当我们设计编程接口的时候，不应该使用缩略词（General Principles）。但是，下面列出的缩写要么已经很成熟，要么已经被使用过，所以，我们可以在编程中使用他们。关于缩略词还有一些要注意的地方：

- 在标准 C 语言库中长期使用的重复表单的缩写——例如 “alloc” 和 “getc”——是允许的。
- 我们可以在参数名中更自由的使用缩写（例如，“imageRep”， “col”（collum），“obj” 和 “otherWin”）。

| Abbreviation | Meaning and comments |
| --- | --- |
| alloc | Allocate. |
| alt | Alternate. |
| app | Application. For example, NSApp the global application object. However, “application” is spelled out in delegate methods, notifications, and so on. |
| calc | Calculate. |
| dealloc | Deallocate. |
| func | Function. |
| horiz | Horizontal. |
| info | Information. |
| init | Initialize (for methods that initialize new objects). |
| int | Integer (in the context of a C `int`—for an `NSInteger` value, use `integer`). |
| max | Maximum. |
| min | Minimum. |
| msg | Message. |
| nib | Interface Builder archive. |
| pboard | Pasteboard (but only in constants). |
| rect | Rectangle. |
| Rep | Representation (used in class name such as `NSBitmapImageRep`). |
| temp | Temporary. |
| vert | Vertical. |

我们可以使用计算机行业中常用的缩略词和首字母缩略词来代替他们所代表的词语。下面是一些常用的首字母缩略词：

- ASCII

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

# 针对框架开发人员的提示和技巧

框架开发人员必须比其他开发人员更仔细地编写代码。许多客户端应用程序都有可能会链接到这个框架，正是因为这种公开性，框架中的任何细小的缺陷都有可能在整个 app 系统中被放大。为了确保框架的高效性和完整性，我们需要明白一些框架可以采用的技。术。

> 注意：其中一些技术并不仅仅局限于框架中使用，我们可以在应用程序的开发过程中有效地应用他们。

## 初始化

下面的一些建议和推荐规范涵盖了框架的初始化。

### 类初始化

initialize 类方法为我们提供了一个只执行一次的入口，方便我们在调用该类的其他方法之前执行一些操作。他通常用于设置类的版本号（可以参考 Versioning and Compatibility ）。

runtime 会向继承链中的每一个类发送 initialize 消息，即使我们没有实现它。因此，它可能会多次调用一个类的 initialize 方法（例如，子类没有实现它的时候，就会多次调用父类的 initialize 方法）。通常，我们的目的是让初始化代码只执行一次，那么，我们可以使用 `dispatch_once()`：

```
+ (void)initialize {
    static dispatch_once_t onceToken = 0;
    dispatch_once(&onceToken, ^{
        // the initializing code
    }
}
```

```
//注意
因为 runtime 会向每个类都发送 initialize ，所以有些情况下我们可能需要根据类的上下文来在 initialize 方法中执行一些操作，但是如果子类不实现 initialize 方法的话，会默认调用父类的实现。如果我们特别想根据相关类的上下文而进行一些初始化操作的话，我们最好使用下面的方法而不是 dispatch_once()：
if (self == [NSFoo class]) {
    // the initializing code
}
```

我们永远都不应该显式地去调用 initialize 方法，如果我们需要出发初始化操作的话，我们可以采用一些无害的方法，比如：

```
[NSImage self];
```

## 指定的初始化器

一个常见的指定的初始化器方法就是一个类的 init 方法，在 init 方法中默认会调用父类的 init 方法。（其他初始化器的实现中需要调用类的 init 方法）。每个公共类都应该有一个或者多个指定的初始化器。例子有 NSView 的 initWithFrame: 方法和 NSResponder 的 init 方法。在有些场景，我们是不需要调用父类的初始化方法的，就像 NSString 和其他抽象类在实现类簇时那样，子类（这里的子类指的是 NSString，NSArray， NSDictionary 等抽象类）应该实现他们自己的初始化方法。

指定的初始化器应该清楚地标识出来，因为这些信息对那些想用这个类做父类的子类非常重要。子类只能覆盖指定的初始化器，其他所有的初始化器都应该按照原本的设计正常运行。

当我们实现一个框架的类的时候，我们必须实现它的归档方法：initWithCoder: 和 encodeWithCoder:。我们需要注意的是，不要在一个对象还没有解档成功之前做一些初始化的操作。如果我们实现了归档，那么一个好的解决办法就是在指定的初始化器方法和 initWithCoder: （本身也是一个指定的初始化器）方法中调用同一个初始化操作。

## 初始化期间的错误检测

一个设计良好的初始化方法应该完成以下步骤来确保错误能进行一个正确的检测和传播。

1. 通过调用 super 指定的初始化器来对 self 进行重新分配。
2. 检测返回值是否为 nil，如果是，说明父类的初始化方法中产生了一些错误。
3. 如果在初始化当前类时发生错误，释放对象并返回 nil 。

以下是对上面这个规则进行的一些说明：

```
- (id)init {
    self = [super init];  // Call a designated initializer here.
    if (self != nil) {
        // Initialize object  ...
        if (someError) {
            [self release];
            self = nil;
        }
    }
    return self;
}
```

# 版本和兼容性

当我们向框架中添加新的类或者方法的时候，通常不需要为每个新的功能组都指定新的版本号。开发人员通常会执行（或者说应该执行）Objective-C 的运行时检查，像 respondsToSelector: 来确保在给定的系统中一个功能组是否可用。这些运行时测试是检查新版本特性的首选方式，同时也是最动态的方式。

然而，我们可以使用多种技术来确保框架中的每个新版本都能被正确地标记到，并与早期版本经可能的兼容。

## 框架版本