

一：Objective-C 入门

1、Cocoa 的组成

苹果公司将 Cocoa、Carbon、QuickTime 和 OpenGL 等技术作为框架集提供

Cocoa 组成部分有：

Foundation 框架(有很多有用的，面向数据的低级类和数据结构)

Application Kit（也称 AppKit）框架（包含了所有的用户接口对象和高级类，例如 NS.....）

，还有一个支持框架的套件，包括 Core Animation 和 Core Image。

2、NSLog 相当于 printf()

```
NSLog(@"hello Objective-C");
```

//注：@是 Objective-C 在标准 C 语言基础上添加的特征之一，双引号的字符串前面有一个 @，这表示引用的字符串应该作为 Cocoa 的 NSString 元素处理

```
NSLog(@"are %d and %d different? %@",5,5,boolString(areTheyDifferent));
```

//注意%@：使用 NSLog 输出任何对象值时，都会使用这个格式说明

3、BOOL 使用 8 位存储，YES 定义为 1，NO 定义为 0，大于 1 不为 YES，跟标准 C 不同。

若不小心将一个长于 1 字节的整型值赋给 BOOL，则只截取低八位

Objective-C 中 1 不等于 YES，绝对不要将 BOOL 值和 YES 比较

二：面向对象的 Objective-C

4、使用间接从本地读取文件的例子

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

```
int main(int argc,const char * argv[])
```

```
{
```

```
if(argc == 1){
```

```
    NSLog(@"you need to provide a file name");
```

```
    return (1);
```

```
}
```

```
FILE *wordFile = fopen(argv[1] , "r");
```

```
char word[100];
```

```
while (fgets(word,100,wordFile)){
```

//fget 调用会保留分开每一行的换行符，我们不需要，把它替换为 0，表示字符串的结束

```
word[strlen(word)-1] = '\0';
```

```
NSLog(@"%s is %d characters long",word,strlen(word));
```

```
}
```

```
//运行用 ./Word-Length-4 /tmp/words.txt
```

若给了文件路径，那么 argc 会大于 1，然后我们可以查询 argv 数组得到文件路径。argv[1]

保存着用户提供的文件名，argv[0]保存着程序名。

在 XCode 中编译此程序需要在 XCode 文件列表中展开 Executables，双击程序名，在 Arguments 区域中添加启动参数

5、id

id 是一种泛型，用于表示任何类的对象，id 实际上是一个指针，指向其中的某个结构

6、[]

例[shape draw]

第一项是对象名，其余部分是要执行的操作

7、Objective-C 的 OOP 范例

1) @interface 部分(一般都作为.h 单独书写，声明部分)

@interface Circle: NSObject //说明这是为 Circle 的新类定义的接口

{

ShapeColor fillColor;

ShapeRect bounds;

} //括号内的是 Circle 对象需要的各种数据成员

- (void) setFillColor:(ShapeColor) fillColor; //先行短线表明“这是新方法的声明”如果是“+”则表示是类方法，也称工厂方法

```
- (void) setBounds:(ShapeRect) bounds;
```

```
- (void) draw;
```

```
@end //Circle
```

2) @implementation 部分(一般写为.m 文件，实现部分)

```
@implementation Circle //@implementation 是一个编译器指令，表明你将为某个类提供代码
```

```
- (void) setFillColor:(ShapeColor) c //在这里如果继续使用参数名 fillColor，就会隐藏 fillColor 实例变量，并且有警告
```

```
//我们已经定义了一个名为 fillColor 的实例变量,可以在该方法中引用该变量,如果使用相同的另一个变量,那么前一个会屏蔽
```

```
{
```

```
fillColor = c;
```

```
}
```

```
- (void) setBounds:(ShapeRect) b
```

```
{
```

```
bounds = b;
```

```
}
```

- (void) draw

{

NSLog("^^^")

}

@end //Circle

可以在@implementation 中定义那些在@interface 中无相应声明的方法，可以把他们看做是私有方法，仅在类的实现中使用。

注：Objective-C 不存在真正的私有方法，从而禁止其他代码调用它。这是 Objective-C 动态本质的副作用。

8、中缀符(infix notation)

方法的名称和及其参数都是合在一起的

例如

一个参数：

```
[circle setFillColor : KRedColor];
```

两个参数：

```
[circle setStringValue : @"hello there" color : KBlueColor];
```

9、继承(X 是一个 Y, isa)

1) Objective-C 不支持多继承，我们可以通过 Objective-C 的其他特性获取多继承的优点，例如分类和协议

2) 继承中方法的定义

可以使用空正文和一个虚(dummy)值都是可以的

3) 方法调度

当代码发送消息时，Objective-C 的方法调度将在当前分类中搜索相应的方法，如果找不到，则在该对象的超类中进行查找

4) 实例变量

10、复合(X 有一个 Y，has)

严格的讲，只有对象间的组合才叫做复合，诸如 int、float、enum 和 struct 等基本类型都认为是对象的一部分

BerwinZheng

2010-2-13 18:58:52

11、init

- (id) init

{

if (self = [super init]) { //将[super init]得结果赋给 self 是 Objective-C 的标准惯例，为了防止超类的初始化过程中返回的对象不同于原先创建的对象

//若要超类要完成所需的一次性初始化，需要调用[super init]，init 方法返回的值描述了被初始化的对象

engine = [Engine new];

```
tires[0] = [Tire new];
```

```
tires[1] = [Tire new];
```

```
tires[2] = [Tire new];
```

```
tires[3] = [Tire new];
```

```
}
```

```
return (self);
```

```
} // init
```

12、存取方法(accessor method)

setter 和 getter

setter 方法根据他所要更改的属性的名称来命名，并加上 set

getter 方法根据其返回的属性的名称来命名，不要加 get

三：源文件组织

13、@class * 和 import *.h

@class 创建一个类前声明，告诉编译器：相信我，以后你会知道这个到底是什么，但是现在，你只需要知道这些

继承一个类的时候不能用 @class，因为他们不是通过指针指向其他类，所以继承一个类时要用 import *.h

四：Xcode 的使用

14、更改自动注释中的公司名

终端中：

```
defaults write com.apple.apple.Xcode PBXCustomTemplateMacroDefinitions
```

```
{“ORGANIZATIONNAME” = “iPhone_xiaoyuan.com”};}
```

没有任何输出结果

15 键盘符号

1)Mac 按键符号

2)Microsoft 键盘和 Mac 键盘的对照

Alt->

徽标键->Option

16、Xcode 技巧

1) 同步显示

有时候两个窗口中显示的内容并不是同步的，只有分别单击了它们，才能同步更新内容

2) 首行缩进

选自，右键->Re-indent selection

Alt [和 Alt]可以把选中的代码左移和右移

3) 代码自动完成

Tab 键可以按频率最高的填充完成词

Esc 可以弹出提示列表 (E 表示枚举, f 代表函数, #代表@define, m 表示方法, C 表示类)

Ctrl+. 在各选项中切换

Shift+Ctrl+. 反向循环

control+/- 在占位符之间切换

4) 批量编辑

快照:File->Make Snapshot

查看快照: File->Snapshot

一次改变文件中的相同字符: 选定, Edit->Edit all in Scope, 更改的时候都会变

重构: 选定, Edit->Refactor, 弹出对话框, 输入要改成的字符 (选中 Snapshot 后可以看见改变)

5) 键盘代替鼠标

- control-F: Move forward, to the right (same as the right arrow).
- control-B: Move backwards, to the left (same as the left arrow).
- control-P: Move to the previous line (same as the up arrow).
- control-N: Move to the next line (same as the down arrow).
- control-A: Move to the beginning of a line (same as the as command- left arrow).
- control-E: Move to the end of a line (same as the as command- right arrow).

- control-T: Transpose (swap) the characters on either side of the cursor.
- control-D: Delete the character to the right of the cursor.
- control-K: Kill (delete) the rest of the line. This is handy if you want to redo the end of a line of code.
- control-L: Center the insertion point in the window. This is great if you've lost your text cursor or want to quickly scroll the window so the insertion point is front and center.

6) 任意搜索

在菜单栏上面搜索

7) 快速打开

#import 后的文件选中，File->Open Quickly, Xcode 就会打开文件。若不选择，则会打开 Open Quickly 对话框

8) 打开文档

Option+双击

9) 调试时看数据

鼠标放在上面一会就可以看到

[BerwinZheng](#)

2010-2-13 18:59:15

五: Foundation Kit

17、一些有用的数据结构 (结构体能减少过程中的开销)

1) NSRange //用来表示相关事物的范围

```
typedef struct _NSRange {
```

```
unsigned int location;
```

```
unsigned int length;
```

```
} NSRange;
```

例如“Objective-C is a cool language”中，“cool”可以用 location 为 17，length 为 4 的范围来表示

有 3 种方式可以创建新的 NSRange

第一种：直接给字段赋值

```
NSRange range;
```

```
range.location = 17;
```

```
range.length = 4;
```

第二种：应用 C 语言的聚合结构赋值机制

```
NSRange range = { 17, 4 };
```

第三种：使用 Cocoa 提供的快捷函数 NSMakeRange():

```
NSRange range = NSMakeRange(17,4);
```

//使用 NSMakeRange()的好处是可以在任何使用函数的地方使用他

```
//例如 [anObject flarbulateWithRange: NSMakeRange (13, 15)];
```

2)几何数据类型

```
typedef struct _NSPoint {
```

```
float x;
```

```
float y;
```

```
} NSPoint;
```

```
typedef struct _NSSize {
```

```
float width;
```

```
float height;
```

```
} NSSize;
```

```
typedef struct _NSRect {
```

```
NSPoint origin;
```

```
NSSize size;
```

```
} NSRect;
```

//Cocoa 也为我们提供了这些类型的快捷函数：NSMakePoint()、NSMakeSize()和NSMakeRect()

18、字符串(NSString 和 NSMutableString)

A: 不可变的字符串(NSString)

1) 创建不可变的字符串

函数: + (id) stringWithFormat: (NSString *) format, ...;

使用方法:

```
NSString *height;
```

```
height = [NSString stringWithFormat:@"Your height is %d feet, %dinches", 5, 11];
```

2) NSString 类中的方法

① 大小

函数: - (unsigned int) length;

使用方法:

```
if ([height length] > 35) {
```

```
    NSLog(@"wow, you're really tall!");
```

```
}
```

② 比较

函数 1: - (BOOL) isEqualToString: (NSString *) aString;

使用方法:

```
NSString *thing1 = @"hello 5";
```

```
NSString *thing2;
```

```
thing2 = [NSString stringWithFormat:@"hello %d", 5];
```

```
if ([thing1 isEqualToString: thing2]) {
```

```
NSLog(@"They are the same!");
```

```
} //应用这个函数，不能用“==”，“==”只能比较字符串的指针值
```

函数 2: - (NSComparisonResult) compare: (NSString *) string;

其中

```
typedef enum _NSComparisonResult {
```

```
NSOrderedAscending = -1,
```

```
NSOrderedSame,
```

```
NSOrderedDescending
```

```
} NSComparisonResult;
```

使用方法:

```
[@"aardvark" compare: @"zygote"] return NSOrderedAscending;.
```

```
[@"zoinks" compare: @"jinkies"] return NSOrderedDescending. And,
```

```
[@"fnord" compare: @"fnord"] return NSOrderedSame.
```

不区分大小写的比较

函数: - (NSComparisonResult) compare: (NSString *) string

options: (unsigned) mask;

options 参数是一个位掩码，可以用位或运算符(|)来添加这些选项标记

一些常用的标记有

■ NSCaseInsensitiveSearch: 不区分大小写

■ NSLiteralSearch: 进行完全比较, 区分大小写

■ NSNumericSearch: 比较字符串的字符个数, 而不是字符值, 若没项, “100”会排在“99”前面(一定要加)

使用方法:

```
if ([thing1 compare: thing2
```

```
options: NSCaseInsensitiveSearch|NSNumericSearch]== NSOrderedSame) {
```

```
NSLog (@"They match!");
```

```
}
```

③ 包含字符串判断

函数:

- (BOOL) hasPrefix: (NSString *) aString; //判断开头

- (BOOL) hasSuffix: (NSString *) aString; //判断结尾

- (NSRange) rangeOfString: (NSString *) aString; //看字符串中是否包含其他字符串

使用方法:

```
NSString *filename = @"draft- chapter.pages";
```

```
if ([fileName hasPrefix: @"draft"]) {
```

```
// this is a draft
```

```
}
```

```
if ([fileName hasSuffix: @".mov"]) {
```

```
// this is a movie
```

```
}
```

```
NSRange range;
```

```
range = [fileName rangeOfString: @"chapter"];
```

//返回 range.start 为 6，range.length 为 7，若传递的参数在接受字符串中没有找到，那么 range.start 则等于 NSRangeNotFound

B)可变字符串(NSMutableString)

1)创建可变的字符串

方式 1:

函数: + (id) stringWithCapacity: (unsigned) capacity; //这个容量只是给 NSMutableString 的一个建议

使用方法:

```
NSMutableString *string;
```

```
string = [NSMutableString stringWithCapacity: 42];
```

方法 2:

继承 NSString 中的方法

```
NSMutableString *string;
```



```
string = [NSMutableString stringWithFormat: @"jo%dy", 2];
```

2)NSMutableString 中的方法

函数:

- (void) appendString: (NSString *) aString;
- (void) appendFormat: (NSString *) format, ...;
- (void) deleteCharactersInRange: (NSRange) range; //配合 rangeOfString: 一起连用

使用方法:

```
NSMutableString *string;
```

```
string = [NSMutableString stringWithCapacity: 50];
```

```
[string appendString: @"Hello there"];
```

```
[string appendFormat: @"human %d!",39];
```

```
NSMutableString *friends;
```

```
friends = [NSMutableString stringWithCapacity: 50];
```

```
[friends appendString: @"James BethLynn Jack Evan"];
```

```
NSRange jackRange;
```

```
jackRange = [friends rangeOfString: @"Jack"];
```

```
jackRange.length++; // eat the space that follows
```

```
[friends deleteCharactersInRange: jackRange];
```

19、NSArray 和 NSMutableArray

A) NSArray(不可改变的数组，是一个 Cocoa 类，用来存储对象的有序列表)

NSArray 的两个限制

首先：它只能存储 Objective-C 的对象，而不能存储 C 语言中的基本数据类型，如：int, float, enum, struct,或者是 NSArray 中的随机指针

然后：不能存储 nil

1)创建方法

通过类的方法 arrayWithObjects: 创建一个新的 NSArray

使用方法：

```
NSArray *array;
```

```
array = [NSArray arrayWithObjects:@"one", @"two", @"three",nil];
```

//array 是以 nil 结尾的,这是 nil 不能存储的原因

2)常用方法

- (unsigned) count; //获取数组包含的对象个数

- (id) objectAtIndex : (unsigned int) index ; //获取特定索引处的对象

- componentsSeparatedByString: //切分 NSArray

- componentsJoinedByString: //合并 NSString

使用方法:

```
int i;
```

```
for (i = 0; i < [array count]; i++) {
```

```
    NSLog(@"index %d has %@", i, [array objectAtIndex:i]);
```

```
}
```

```
NSString *string = @"oop:ack:bork:greeble:ponies";
```

```
NSArray *chunks = [string componentsSeparatedByString: @":"];
```

```
string = [chunks componentsJoinedByString: @" :- "];
```

B)NSMutableArray(可变数组)

1)创建方法, 通过类方法 arrayWithCapacity 创建

```
+ (id) arrayWithCapacity: (unsigned) numItems;
```

使用方法:

```
NSMutableArray *array;
```

```
array = [NSMutableArray arrayWithCapacity: 17];
```

2)常用方法

```
- (void) addObject: (id) anObject;
```

```
- (void) removeObjectAtIndex: (unsigned) index;
```

使用方法:

```
for (i = 0; i < 4; i++) {
```

```
Tire *tire = [Tire new];
```

```
[array addObject: tire];
```

```
}
```

```
[array removeObjectAtIndex: 1]; //删除第二个
```

C)遍历数组的三种方式：通过索引、使用 `NAEnumerator` 和快速枚举

1)索引遍历 //只有在真的需要索引访问数组时才应使用`-objectAtIndex`，例如跳跃数组或者同时遍历多个数组时

```
int i;
```

```
for (i = 0; i < [array count]; i++) {
```

```
NSLog(@"index %d has %@", i, [array objectAtIndex: i]);
```

```
}
```

2)使用 `NSEnumerator` //Leopard 中被快速枚举替代

创建方法：通过函数 `-(NSEnumerator *) objectEnumerator;`

使用方法：

```
NSEnumerator *enumerator;
```

```
enumerator = [array objectEnumerator]; //如果想从后往前浏览集合，还有一个方法  
reverseEnumerator 可以使用
```

创建后通过 `while` 循环，条件是 `nextObject(方法原型 - (id) nextObject);`

循环遍历的程序为：

```
NSEnumerator *enumerator;

enumerator = [array objectEnumerator];

id thingie;

while(thingie = [enumerator nextObject ]) {

    NSLog(@"i found %@", thingie);

}
```

//注：对可变数组进行枚举操作时，不能通过添加和删除对象这类方式来改变数组容器，如果这样做了，枚举器会觉得困惑，为你将会得到未定义结果

3)快速枚举

在 Leopard 中才开始的，Tiger 中不能用

```
for (NSString *string in array ) {

    NSLog(@"i found %@", string);

}
```

[BerwinZheng](#)

2010-2-13 19:35:01

21、破除 NSArray 限制的方法

1)基本类型

a):Cocoa 提供了 NSNumber 类来包装基本类型

+ (NSNumber *) numberWithChar: (char) value;

+ (NSNumber *) numberWithInt: (int) value;

+ (NSNumber *) numberWithFloat: (float) value;

+ (NSNumber *) numberWithBool: (BOOL) value;

使用方法:

```
NSNumber *numner;
```

```
number = [NSNumber numberWithInt: 42];
```

```
[array addObject: number];
```

```
[dictionary setObject : number foyKey : @"Bork"];
```

只要将一些基本类型封装到 **NSNumber** 中, 就可以通过下面的实例方法重新获得其值

- (char) charValue;

- (int) intValue;

- (float) floatValue;

- (BOOL) boolValue;

- (NSString *) stringValue; //允许自动转换

Objective-C 不支持自动装箱, 要自己动手

b):NSNumber 是 NSValue 的子类, NSValue 可以包装任意值

创建新的 NSValue

+ (NSValue *) valueWithBytes: (const void *) value

objCType: (const char *) type;

使用方法:

```
NSRect rect = CGRectMake (1, 2, 30, 40);
```

```
NSValue *value;
```

```
value = [NSValue valueWithBytes: &rect
```

```
        objcType: @encode(NSRect)]; //encode 编译器指令可以接受数据类型的名称并为你  
        生成合适的字符串
```

```
[array addObject: value];
```

可以使用 `getValue:` 来提取数值（注意是 `get` 方法，指针）

```
- (void) getValue: (void *) value; //调用时，要传递的是要存储这个数值的变量的地址
```

使用方法

```
value = [array objectAtIndex: 0];
```

```
[value getValue: &rect];
```

Cocoa 提供了常用的 `struct` 型数据转换成 `NSValue` 的便捷方法

```
+ (NSValue *) valueWithPoint: (NSPoint) point;
```

```
+ (NSValue *) valueWithSize: (NSSize) size;
```

```
+ (NSValue *) valueWithRect: (NSRect) rect;
```

```
- (NSPoint) pointValue;
```

```
- (NSSize) sizeValue;
```

```
- (NSRect) rectValue;
```

使用方法:

```
value = [NSValue valueWithRect: rect];
```

```
[array addObject: value];
```

....

```
NSRect anotherRect = [value rectValue];
```

2)NSNull

NSNull 大概是 Cocoa 里最简单的类了，只有一个方法

```
+ (NSNull *) null;
```

可以这样添加到集合中

```
[contact setObject: [NSNull null]]
```

```
forKey: @"home fax machine"];
```

访问时:

```
id homefax;
```

```
homefax = [contact objectForKey: @"home fax machine"];
```

```
if (homefax == [NSNull null]) {
```

```
// ... no fax machine. rats.
```

```
}
```

//[NSNull null]总是返回一样份数值，所以你可以使用“==”讲该值与其他值进行比较.....

22、NSDictionary 和 NSMutableDictionary

A) NSDictionary

字典是关键字和其定义的集合，也被成为散列表或关联数组，使用的是键查询的优化存储方式

1)创建方法： 使用 dictionaryWithObjectsAndKeys: 来创建字典

```
+ (id) dictionaryWithObjectsAndKeys: (id) firstObject, ...;
```

使用方法：

```
Tire *t1 = [Tire new];
```

```
Tire *t2 = [Tire new];
```

```
Tire *t3 = [Tire new];
```

```
Tire *t4 = [Tire new];
```

```
NSDictionary *tires;
```

```
tires = [NSDictionary dictionaryWithObjectsAndKeys:
```

```
t1, @"front- left", t2, @"front- right",
```

```
t3, @"back- left", t4, @"back- right", nil];
```

2)常用方法

```
- (id) objectForKey: (id) aKey;
```

使用方法：

```
Tire *tire = [tires objectForKey: @"back- right"]; //如果没有则会返回 nil 值
```

B) NSMutableDictionary

1)创建方法:

可以向类 NSMutableDictionary 发送 dictionary 消息

也可以使用函数+ (id) dictionaryWithCapacity: (unsigned int) numItems;

2)常用方法

可以使用 setObject: forKey: 方法给字典添加元素:

- (void) setObject: (id) anObject forKey: (id) aKey;

- (void) removeObjectForKey: (id) aKey;

使用方法:

```
NSMutableDictionary *tires;
```

```
tires = [NSMutableDictionary dictionary];
```

```
[tires setObject: t1 forKey: @"front- left"];
```

```
[tires setObject: t2 forKey: @"front- right"];
```

```
[tires setObject: t3 forKey: @"back- left"];
```

```
[tires setObject: t4 forKey: @"back- right"];
```

//若已经存在，则会用新值替换原有的值

```
[tires removeObjectForKey: @"back- left"];
```

23、不要创建 NSString、NSArray 或 NSDictionary 的子类，因为在 Cocoa 中，许多类实际上是以类簇的方式实现的，即他们是一群隐藏在通用接口之下的与实现相关的类

24、Foundation 实例 //查找文件

A)使用枚举遍历

```
int main (int argc, const char *argv[])
```

```
{
```

```
    NSAutoreleasePool *pool;
```

```
    pool = [[NSAutoreleasePool alloc] init]; //自动释放池
```

```
    NSFileManager *manager; //Cocoa 中有很多类都是单实例构架，即只需要一个实例，你真的只需要一个文件管理器
```

```
    manager = [NSFileManager defaultManager]; // defaultManager 方法创建一个属于我们的 NSFileManager 对象
```

```
    NSString *home;
```

```
    home = [@"~" stringByExpandingTildeInPath]; // stringByExpandingTildeInPath 方法可将~替换成当前用户的主目录
```

```
    NSDirectoryEnumerator *direnum; //NSEnumerator 的子类
```

```
    direnum = [manager enumeratorAtPath: home]; //创建一个枚举条件
```

```
    NSMutableArray *files;
```

```
    files = [NSMutableArray arrayWithCapacity: 42]; //把搜索的结果作为文件存储
```

```
    NSString *filename;
```

```
    while (filename = [direnum nextObject]) { //调用 nextObject 时，都会返回该目录中的一个文件的另一个路径，也可搜索子目录
```

```
        if ([[filename pathExtension] // pathExtension 输出文件的扩展名(去掉了前面的点.)
```

```

isEqualTo: @"jpg"]) {

[files addObject: filename];

}

}

NSEnumerator *fileenum;

fileenum = [files objectEnumerator];

while (filename = [fileenum nextObject]) {

NSLog ("%@", filename);

}

[pool drain];

return (0);

} // main

```

B)使用快速遍历

```

int main (int argc, const char * argv[]) {

NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

NSFileManager *manager;

manager = [NSFileManager defaultManager];

NSString *home;

```

```
home = [@"~" stringByExpandingTildeInPath];

NSMutableArray *files;

files = [NSMutableArray arrayWithCapacity: 42];

for (NSString *filename

in [manager enumeratorAtPath: home]) {

if ([[filename pathExtension]

isEqualTo: @"jpg"]) {

[files addObject: filename];

}

}

for (NSString *filename in files) {

NSLog(@"%@", filename);

}

}
```

[BerwinZheng](#)

2010-2-13 19:35:36

六：内存管理

25、Cocoa 采用引用计数(reference counting)的技术，有时称为保留计数。

每个对象有一个与之相关联的整数，称做为他的引用计数器或保留计数器

- (id) retain;

- (void) release;

- (unsigned) retainCount; //当前值

26、对象所有权的处理

- (void) setEngine: (Engine *) newEngine

{

[newEngine retain];

[engine release];

engine = newEngine;

} // setEngine

原则：先保存新对象，再释放旧对象

27、自动释放

程序会自动建立一个自动释放池(autorelease pool)，他是一个存放实体的池(集合)，这些实体可能是对象，能够被自动释放。

自动释放池创建代码

NSAutoreleasePool *pool;

pool = [[NSAutoreleasePool alloc] init];

.....

[pool release];

NSObject 类提供了一个 autorelease 方法：

- (id) autorelease;

//该方法预先定义了一条在将来某个时间发送的 **release** 消息，其返回值是接收消息的对象，**retain** 采用了相同的技术，使嵌套调用更加容易。

//当给一个对象发送 **autorelease** 消息时，实际上是将对象添加到 **NSAutoreleasePool** 方法中，当自动释放池销毁了，会像该池中的所有对象发送 **release** 消息

例如

```
- (NSString *) description
```

```
{
```

```
    NSString *description;
```

```
    description = [[NSString alloc]
```

```
        initWithFormat: @"I am %d years old", 4];
```

```
    return ([description autorelease]); //因为 descriptor 方法首先创建了一个新的字符串对象，  
    然后自动释放该对象，最后将其返回给 NSLog()函数
```

```
} // description
```

29、Cocoa 内存管理原则

如果使用 **new**，**alloc** 或 **copy** 操作获得一个对象，则该对象的保留计数器值加 1，**release** 减 1

如果通过任何其他方法获得一个对象，则假设该对象的保留计数器值为 1，而且已经被设置为自动释放

如果保留了某个对象，则必须保持 **retain** 方法和 **release** 方法使用的次数相同

30、NSColor 的 blueColor 方法返回一个全局单例对象

31、一直拥有对象

希望在多个代码段中一直拥有某个对象常见的方法有：在其他对象中使用这些变量，将它们加入到诸如 `NSArray` 或 `NSDictionary` 等集合中，或将其作为全局变量使用(罕见)

如果你使用 `new`, `alloc` 或 `copy` 方法获得一个对象，则不需要执行任何其他操作，他将一直存在，你只要在拥有该对象的 `dealloc` 方法中释放该对象就可

```
- (void) doStuff
```

```
{
```

```
// flonkArray is an instance variable
```

```
flonkArray = [NSMutableArray new]; // count: 1
```

```
} // doStuff
```

```
- (void) dealloc
```

```
{
```

```
[flonkArray release]; // count: 0
```

```
[super dealloc];
```

```
} // dealloc
```

32、Cocoa 程序才开始处理事件之前创建一个自动释放池，并在事件处理结束后销毁该自动释放池

33、保证内存占用比较小的一种方法，分段处理

```
int i;
```

```
for (i = 0; i < 1000000; i++) {
```

```
id object = [someArray objectAtIndex:i];
```



```
NSString *desc = [object description];

// and do something with the description

}
```

节省内存的方法：

```
NSAutoreleasePool *pool;

pool = [[NSAutoreleasePool alloc] init];

int i;

for (i = 0; i < 1000000; i++) {

    id object = [someArray objectAtIndex:i];

    NSString *desc = [object description];

    // and do something with the description

    if (i % 1000 == 0) {

        [pool release];

        pool = [[NSAutoreleasePool alloc] init];

    }

}

[pool release]

//自动释放池以栈的形式存在
```

34、Objective-C 2.0 的垃圾回收机制，是一个可选择启用的功能，项目信息属性转到 **Build**

选项卡，在 Objective-C Garbage Collection 选项选成 Required [-fobjc-gc- only]即可

但注意：iPhone 里面不能用

七：对象的初始化

35、两种方法

[类名 new] //不熟悉 Cocoa 的开发人员使用的辅助方法

[[类名 alloc] init] //主要使用方法

36、分配(allocation)

向某个发送

内存区域：全部初始化为 0

BOOL：NO

int：0

float：0.0

指针：nil

37、两种格式

Car *car = [[Car alloc] init]; //推荐使用，这种嵌套调用非常重要，因为初始化方法返回的对象可能与分配的对象不同，虽然很奇怪，但是它的确会发生

Car *car = [Car alloc];

[car init]; //不推荐使用

38、编写 init 方法

- (id) init

```
{  
  
if (self = [super init]) {  
  
engine = [Engine new];  
  
tires[0] = [Tire new];  
  
tires[1] = [Tire new];  
  
tires[2] = [Tire new];  
  
tires[3] = [Tire new];  
  
}  
  
return (self);  
  
} // init
```

//注意首行的 **self = [super init]**，从根类 **NSObject** 继承的类调用超类的初始化方法，可以使 **NSObject** 执行所需的任何操作，以便对象能够响应消息并处理保留计数器，而从其他类继承的类调用超类的初始化方法，可以使子类有机会实现自己全新的初始化

//实例变量所在的位置到隐藏的 **self** 参数的距离是固定的，如果从 **init** 方法返回一个新对象，则需要更新 **self**，以便其后的任何实例变量的引用可以被映射到正确的位置，这也是 **self = [super init]** 使用的原因，记住，这个赋值操作只影响 **init** 方法中 **self** 的值，而不影响该范围以外的任何内容

// if (**self = [super init]**) 使用的原因，如果 **[super init]** 返回的结果是 **nil**，则主体不会执行，只是赋值和检测非零值结合的方法，沿袭自 C 风格

39、便利初始化函数 //也可以自己构建

- (id) initWithFormat: (NSString *) format, ...;

- (id) initWithContentsOfFile: (NSString *) path; //打开指定路径上的文件，读取文件内容，并使用文件类内容初始化一个字符串

使用方法：

```
string = [[NSString alloc]
```

```
initWithFormat: @"%d or %d", 25, 624];
```

```
string = [[NSString alloc]
```

```
initWithContentsOfFile: @"/tmp/words.txt"];
```

构造便利初始化函数

例如

在@interface Tire: NSObject 中添加方法声明

- (id) initWithPressure : (float) pressure

treadDepth: (float) treadDepth;

在@implementation Tire 中实现该方法

- (id) initWithPressure: (float) p

treadDepth: (float) td

{

if (self = [super init]) {

pressure = p;

```
treadDepth = td;
```

```
}
```

```
return (self);
```

```
} // initWithPressure:treadDepth:
```

这样就完成了初始化函数的定义，分配，初始化一体完成

```
Tire *tire;
```

```
tire = [[Tire alloc]
```

```
initWithPressure: 23 + i
```

```
treadDepth: 33 - i];
```

[BerwinZheng](#)

2010-2-13 19:36:05

39、如果用 NSMutableArray 代替 C 数组，则就不用执行边界检查

40、指定初始化函数

有的时候定义了太多的初始化函数时，会出现一些细微的问题

例如下面的程序

```
@interface Tire : NSObject
```

```
{
```

```
float pressure;
```

```
float treadDepth;
```

```
}
```

- (id) initWithPressure: (float) pressure;

- (id) initWithTreadDepth: (float) treadDepth; //新增加的两个初始化函数

- (id) initWithPressure: (float) pressure

treadDepth: (float) treadDepth;

- (void) setPressure: (float) pressure;

- (float) pressure;

- (void) setTreadDepth: (float) treadDepth;

- (float) treadDepth;

@end // Tire

//声明了三个初始化函数

//新声明的初始化函数的实现

- (id) initWithPressure: (float) p

{

if (self = [super init]) {

pressure = p;

treadDepth = 20.0;

}

return (self);

```
} // initWithPressure
```

```
- (id) initWithTreadDepth: (float) td
```

```
{
```

```
if (self = [super init]) {
```

```
pressure = 34.0;
```

```
treadDepth = td;
```

```
}
```

```
return (self);
```

```
} // initWithTreadDepth
```

问题来了

子类化问题:

```
@interface AllWeatherRadial : Tire
```

```
{
```

```
float rainHandling;
```

```
float snowHandling;
```

```
}
```

```
- (void) setRainHandling: (float) rainHandling;
```

```
- (float) rainHandling;
```

```
- (void) setSnowHandling: (float) snowHandling;
```

- (float) snowHandling;

@end // AllWeatherRadial

//枯燥的存取函数

- (void) setRainHandling: (float) rh

{

rainHandling = rh;

} // setRainHandling

- (float) rainHandling

{

return (rainHandling);

} // rainHandling

- (void) setSnowHandling: (float) sh

{

snowHandling = sh;

} // setSnowHandling

- (float) snowHandling

{

return (snowHandling);


```

} // snowHandling

- (NSString *) description

{

    NSString *desc;

    desc = [[NSString alloc] initWithFormat:

@"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",

[self pressure], [self treadDepth],

[self rainHandling],

[self snowHandling]];

    return (desc);

} // description

//main.m 函数中实现

int i;

for (i = 0; i < 4; i++) {

    AllWeatherRadial *tire;

    tire = [[AllWeatherRadial alloc] init];

    [car setTire: tire

    atIndex: i];

    [tire release];

```

```
}
```

运行的结果是下面这个样子的

```
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
```

```
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
```

```
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
```

```
AllWeatherRadial: 34.0 / 20.0 / 0.0 / 0.0
```

```
I am a slant- 6. VROOOM!
```

//注：默认情况下初始化函数只会按最容易实现的方式去运行，这不是我要的结果，并且是错误的结果

解决办法：指定初始化函数(designated initializer)

```
- (id) init
```

```
{
```

```
if (self = [self initWithPressure: 34
```

```
treadDepth: 20]) {
```

```
}
```

```
return (self);
```

```
} // init
```

```
- (id) initWithPressure: (float) p
```

```
{
```

```

if (self = [self initWithPressure: p
    treadDepth: 20.0]) {

}

return (self);

} // initWithPressure

- (id) initWithTreadDepth: (float) td

{

if (self = [self initWithPressure: 34.0
    treadDepth: td]) {

}

return (self);

} // initWithTreadDepth

```

添加到 AllWeatherRadial 类的初始化函数

```

- (id) initWithPressure: (float) p

    treadDepth: (float) td

{

if (self = [super initWithPressure: p
    treadDepth: td]) {

```

```
rainHandling = 23.7;
```

```
snowHandling = 42.5;
```

```
}
```

```
return (self);
```

```
} // initWithPressure:treadDepth
```

此时我们再运行可以得到这样的结果

```
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
```

```
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
```

```
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
```

```
AllWeatherRadial: 34.0 / 20.0 / 23.7 / 42.5
```

```
I am a slant- 6. VROOOM!
```

```
BerwinZheng
```

```
2010-2-13 19:36:38
```

```
八： 属性
```

41、属性(property)是 Objective-C 2.0 中引入的,为了方便的编写存取方法

42、属性的使用方法

1)声明方法的简化

```
//旧的表示方法
```

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

```
#import "Tire.h"
```

```
@interface AllWeatherRadial : Tire {
```

```
float rainHandling;
```

```
float snowHandling;
```

```
}
```

```
- (void) setRainHandling: (float) rainHandling;
```

```
- (float) rainHandling;
```

```
- (void) setSnowHandling: (float) snowHandling;
```

```
- (float) snowHandling;
```

```
@end // AllWeatherRadial
```

```
//用属性表示后
```

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

```
#import "Tire.h"
```

```
@interface AllWeatherRadial : Tire {
```

```
float rainHandling;
```

```
float snowHandling;
```

```
@property float rainHandling; //表明该类有一个名为 rainHandling 的 float 型属性，你可以通过-setRainHandling: 来设置属性，通过-rainHandling 来访问属性
```

```
@property float snowHandling;
```

```
@end // AllWeatherRadial
```

```
}
```

//@property 预编译命令的作用是自动声明属性的 **setter** 和 **getter** 方法

//属性的名称不必与实例变量名称相同，但是一般都是相同的

2)实现方法的简化

//百年老字号

```
#import "AllWeatherRadial.h"
```

```
@implementation AllWeatherRadial
```

```
- (id) initWithPressure: (float) p
```

```
    treadDepth: (float) td
```

```
{
```

```
    if (self = [super initWithPressure: p
```

```
        treadDepth: td]) {
```

```
        rainHandling = 23.7;
```

```
        snowHandling = 42.5;
```

```
    }
```

```
    return (self);
```

```
} // initWithPressure:treadDepth
```

```
- (void) setRainHandling: (float) rh
```

```
{

rainHandling = rh;

} // setRainHandling

- (float) rainHandling

{

return (rainHandling);

} // rainHandling

- (void) setSnowHandling: (float) sh

{

snowHandling = sh;

} // setSnowHandling

- (float) snowHandling

{

return (snowHandling);

} // snowHandling

- (NSString *) description

{

NSString *desc;

desc = [[NSString alloc] initWithFormat:
```

```
@ "AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
```

```
[self pressure], [self treadDepth],
```

```
[self rainHandling],
```

```
[self snowHandling]];
```

```
return (desc);
```

```
} // description
```

```
@end // AllWeatherRadial
```

```
//改进后的方法
```

```
#import "AllWeatherRadial.h"
```

```
@implementation AllWeatherRadial
```

```
@synthesize rainHandling;
```

```
@synthesize snowHandling;
```

```
- (id) initWithPressure: (float) p
```

```
    treadDepth: (float) td
```

```
{
```

```
    if (self = [super initWithPressure: p
```

```
        treadDepth: td]) {
```

```
        rainHandling = 23.7;
```



```

snowHandling = 42.5;

}

return (self);

} // initWithPressure:treadDepth

- (NSString *) description

{

NSString *desc;

desc = [[NSString alloc] initWithFormat:

@"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",

[self pressure], [self treadDepth],

[self rainHandling],

[self snowHandling]];

return (desc);

} // description

@end // AllWeatherRadial

```

//@synthesize 也是一种新的编译器功能，表示“创建该属性的访问器”

//当遇到@synthesize rainHandling;时，编译器将输出-setRainHandling:和- rainHandling 方法的已编译代码

43、点表达式

点表达式(.), 若出现在等号(=)左边, 该属性名称的 **setter** 方法将被调动, 多出现在对象变量右边, 则该属性名册和那个的 **getter** 方法将被调用

//注: 特性的点表达式和流行的键/值编码的后台工作没有联系

44、特性扩展

特性同样适用于 int、char、BOOL 和 struct 类型

(所有者对象保留被拥有的对象, 而不是被拥有的对象保留所有者对象)

//可以使用一些声明, 用于内存处理(那个是用垃圾回收机制的路过)

```
@property (copy) NSString *name;
```

```
@property (retain) Engine *engine;
```

45、特性名和实例变量名字不相同的情况

```
@interface Car : NSObject {
```

```
NSString *appellation;
```

```
NSMutableArray *tires;
```

```
Engine *engine;
```

```
}
```

```
@property (copy) NSString *name;
```

```
@property (retain) Engine *engine;
```

//然后, 修改@synthesize 指令

```
@synthesize name = appellation;
```

编译器仍将创建 `-setName:` 和 `-name` 方法，但是在实现中却是用实例变量 `application`

//这样做会有错误，因为我们直接访问的实例变量 `name` 已经被修改了，我们既可以选择搜索替换 `name`，也可以将直接的实例变量访问修改为使用访问器访问，在 `init` 方法中，将

```
name = @"Car";
```

修改为：

```
self.name = @"Car" ;  // [self setName : @"Car"];
```

在 `dealloc` 中，使用一种高明的技巧：

```
self.name = nil; //使用 nil 参数调用 setName:方法
```

生成的访问器将自动释放以前的 `name` 对象，并使用 `nil` 替代 `name`

最后修改 `-description` 方法需要使用第一次被修改的 `NSLog()` 函数：

```
NSLog(@"%@ has:", self.name);
```

[BerwinZheng](#)

2010-2-13 19:37:00

46、只读特性

//默认的特性是支持可写可读的，原型如下

```
@property (readwrite, copy) NSString *name;
```

```
@property (readwrite, retain) Engine *engine;
```

//但为了简便，为了消除重复

//只读属性的设置

```
@interface Me : NSObject {
```

```
float shoeSize;
```

```
NSString *licenseNumber;
```

```
}
```

```
@property (readonly) float shoeSize;
```

```
@property (readonly) NSString *licenseNumber;
```

```
@end
```

//这类编译器只会生成 **getter** 方法，不会有 **setter** 方法

47、特性的局限性

//不支持那么需要接受额外参数的方法

```
- (void) setTire: (Tire *) tire
```

```
atIndex: (int) index;
```

```
- (Tire *) tireAtIndex: (int) index;
```

//这样的只能使用百年老字号

九：类别

48、可以利用 **Objective-C** 的动态运行时分配机制，为现有的类添加新方法---这就叫类别 (**category**)

//特别是那些不能创建之类的类，很是 **cool**

49、创建类别

//如果你希望想一个 **array** 或者 **dictionary** 里面添加一个个数字，你需要一个个的封装，如果多，你会疯掉，可以为 **string** 类添加一个类别来完成这项工作

1)声明对象 //与类的声明格式类似

```
@interface NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber;
```

```
@end // NumberConvenience
```

//我们正在向 **String** 类里面添加一个 **NumberConvenience** 方法，可以添加很多个，只要名称不相同

2)实现部分

```
@implementation NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber
```

```
{
```

```
    unsigned int length = [self length]; //获得字符串的长度
```

```
    return ([NSNumber numberWithInt: length]);
```

```
} // lengthAsNumber
```

```
@end // NumberConvenience
```

现在就可以用了

```
int main (int argc, const char *argv[])
```

```
{
```

```

NSAutoreleasePool *pool;

pool = [[NSAutoreleasePool alloc] init];

NSMutableDictionary *dict;

dict = [NSMutableDictionary dictionary];

[dict setObject: @"hello" lengthAsNumber]

forKey: @"hello"];

[dict setObject: @"iLikeFish" lengthAsNumber]

forKey: @"iLikeFish"];

[dict setObject: @"Once upon a time" lengthAsNumber]

forKey: @"Once upon a time"];

NSLog (@"%@", dict);

[pool release];

return (0);

} // main

```

//任何 NSString 类都将响应 lengthAsNumber 消息,正式这种兼容性使类别称为一个非常伟大的概念,不需要创建 NSString 的之类,类别同样可以完成同样的工作

50、类别的局限性

第一: 无法向类别里面添加新的实例变量,类别里面没有位置容纳实例变量//也可以是用 dictionary 封装,但是不划算

第二：若名称冲突，类别的优先级更高(一般都是加一个前缀避免名称冲突)

51、类别的作用

Cocoa 中类别主要用于 3 个目的

1)将类的实现分散到多个不同的文件或不同构架中

用类别分离文件时注意类别的写法，一个类的类别才能实现这个类的方法

2)创建对私有方法的前向引用

有些声明不需要写在.h 文件中，因为有的时候这个只是本类的一个小的实现，声明太麻烦，而且让 code reader 比较难理解，就可以在.m 中用类别声明一下

```
@interface Car (PrivateMethods)
```

```
- (void) moveTireFromPosition: (int) pos1
```

```
toPosition: (int) pos2;
```

```
@end //private Methods
```

3)向对象添加非正式协议

非正式协议表示这里有一些你可能希望实现的方法，因此你可以使用它们更好的完成工作

52、将类的实现分散到多个不同的文件或不同架构中

看文档中的 NSWindows 类

```
@interface NSWindow : NSResponder
```

然后是一大堆类别:

```
@interface NSWindow(NSKeyboardUI)
```

```
@interface NSWindow(NSToolbarSupport)
```

```
@interface NSWindow(NSDrag)
```

```
@interface NSWindow(NSCarbonExtensions)
```

```
@interface NSObject(NSWindowDelegate)
```

这样就可以把一个大的文件分开使用,看起来方便,实用

53、run 循环

```
[[NSRunLoop currentRunLoop] run];
```

是一种 cocoa 构造,它一直处于阻塞状态(即不执行任何处理),知道某些有趣的事情发生为止

//这个 run 方法将一直运行而不会返回,后面的代码将一直不执行

54、委托和类别

委托强调类别的另一种应用:被发送给委托对象的方法可以声明为一个 NSObject 的类别。
NSNetService 委托方法的声明如下

```
@interface NSObject
```

```
(NSNetServiceBrowserDelegateMethods)
```


- (void) netServiceBrowser: (NSNetServiceBrowser *) browser

didFindService: (NSNetService *) service

moreComing: (BOOL) moreComing;

- (void) netServiceBrowser: (NSNetServiceBrowser *) browser

didRemoveService: (NSNetService *) service

moreComing: (BOOL) moreComing;

@end

通过这些方法声明为 **NSObject** 的类型，**NSNetServiceBrowser** 的实现可以将这些消息之一发送给任何对象，无论这些对象实际上属于哪个类。这意味着，只要实现了委托方法，任何类的对象都可以成为委托对象

通过这种方法可以不继承（c++）和不实现某个特定的接口（java），就可以作为委托对象使用

55、**NSObject** 提供了一个名为 **respondsToSelector:** 的方法，该方法访问对象以确定其是否能够响应某个特定的消息

56、复制的种类有 **Shallow Copy** 和 **deep copy**

Shallow Copy 不复制引用对象，新复制的对象只指向指向现有的引用对象

deep copy 将复制所有的引用对象

57, [self class]妙用

```
Car *carCopy = [[[self class] allocWithZone: zone] init];
```

可以通过 **self** 的类型来判断运行结果，子类的用这个函数就是子类的结果

58, NSDate *yesterday = [NSDate dateWithTimeIntervalSinceNow: -(24 * 60 * 60)];

获取一个时间段以前的一个时间，dateWithTimeIntervalSinceNow 接受一个 NSTimeInterval 参数，是一个双精度值，以秒为单位

59.NSData 和 char*的转化

```
const char *string = "Hi there, this is a C string!";
```

```
NSData *data = [NSData dataWithBytes: string
```

```
length: strlen(string) + 1];
```

```
NSLog(@"data is %@", data); //输出为 ascii 码
```

```
NSLog(@"%d byte string is '%s'", [data length], [data bytes]); //格式化输出要的内容
```

60, 有些属性文件（特别是首选项文件）是以二进制格式存储的，通过使用 plutil 命令：plutil -convert xml1 filename.plist 可以转化成人们可读的形式

[BerwinZheng](#)

2010-2-13 19:37:32

61, [phrase writeToFile: @"/tmp/verbiage.txt" atomically: YES];的 atomically 是用于通知 cocoa 是否应该首先将文件内容保存在临时文件中，当文件保存成功后，再将该临时文件和原始文件交换

62, 编码对象

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

```
@interface Thingie : NSObject <NSCoding> {
```

```
    NSString *name;
```

```
    int magicNumber;
```

```
    float shoeSize;
```

```
    NSMutableArray *subThingies;
```

```
}
```

```
@property (copy) NSString *name;
```

```
@property int magicNumber;
```

```
@property float shoeSize;
```

```
@property (retain) NSMutableArray *subThingies;
```

```
- (id)initWithName: (NSString *) n
```

```
    magicNumber: (int) mn
```

```
    shoeSize: (float) ss;
```

```
@end // Thingie
```

```
@implementation Thingie
```

```
@synthesize name;
```

```
@synthesize magicNumber;
```

```
@synthesize shoeSize;
```

```
@synthesize subThingies;
```

```
- (id)initWithName: (NSString *) n
```

```
    magicNumber: (int) mn
```

```
    shoeSize: (float) ss {
```

```
    if (self = [super init]) {
```

```
        self.name = n;
```

```
        self.magicNumber = mn;
```

```
        self.shoeSize = ss;
```

```
        self.subThingies = [NSMutableArray array];
```

```
    }
```

```
    return (self);
```

```
}
```

```
- (void) dealloc {
```

```
    [name release];
```

```
    [subThingies release];
```

```
[super dealloc];
```

```
} // dealloc
```

```
- (NSString *) description {
```

```
    NSString *description =
```

```
    [NSString stringWithFormat: @"%@: %d/%.1f %@",
```

```
    name, magicNumber, shoeSize, subThingies];
```

```
    return (description);
```

```
} // description
```

```
- (void) encodeWithCoder: (NSCoder *) coder {
```

```
    [coder encodeObject: name
```

```
    forKey: @"name"];
```

```
    [coder encodeInt: magicNumber
```

```
    forKey: @"magicNumber"];
```

```
    [coder encodeFloat: shoeSize
```

```
    forKey: @"shoeSize"];
```

```

        [coder encodeObject: subThingies

forKey: @"subThingies"];

} // encodeWithCoder

- (id) initWithCoder: (NSCoder *) decoder {

    if (self = [super init]) {

        self.name = [decoder decodeObjectForKey: @"name"];

        self.magicNumber = [decoder decodeIntForKey: @"magicNumber"];

        self.shoeSize = [decoder decodeFloatForKey: @"shoeSize"];

        self.subThingies = [decoder decodeObjectForKey: @"subThingies"];

    }

    return (self);

} // initWithCoder

@end // Thingie

int main (int argc, const char * argv[]) {

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

```

```
Thingie *thing1;

thing1 = [[Thingie alloc

initWithName: @"thing1"

magicNumber: 42

shoeSize: 10.5];

NSLog(@"some thing: %@", thing1);

// 使用 NSData 的两个子类 NSKeyedArchiver 和 NSKeyedUnarchiver

NSData *freezeDried;

freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];

[thing1 release];

thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];

NSLog(@"reconstituted thing: %@", thing1);


Thingie *anotherThing;

anotherThing = [[[Thingie alloc

initWithName: @"thing2"

magicNumber: 23

shoeSize: 13.0] autorelease];
```

```
[thing1.subThingies addObject: anotherThing];
```

```
anotherThing = [[[Thingie alloc]
```

```
initWithName: @"thing3"
```

```
magicNumber: 17
```

```
shoeSize: 9.0] autorelease];
```

```
[thing1.subThingies addObject: anotherThing];
```

```
NSLog (@"thing with things: %@", thing1);
```

```
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
```

```
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
```

```
NSLog (@"reconstituted multithing: %@", thing1);
```

```
[thing1.subThingies addObject: thing1];
```

```
// You really don't want to do this...
```

```
// NSLog (@"infinite thinging: %@", thing1);
```

```
freezeDried = [NSKeyedArchiver archivedDataWithRootObject: thing1];
```

```
[freezeDried writeToFile:@" /tmp/xiaoyuan" atomically:YES];
```

```
thing1 = [NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
```

```
[pool release];
```

```
return (0);
```



```
} // main
```

63, KVC(Key Value Code)

Advantage one:

```
NSLog(@"horsepower is %@", [engine valueForKey: @"horsepower"]); //这个会自动打  
包成 NSNumber 或 NSValue
```

```
[engine setValue: [NSNumber numberWithInt: 150] //这个使用的时候要自己打包
```

```
 forKey: @"horsepower");
```

```
NSLog(@"horsepower is %@", [engine valueForKey: @"horsepower"]);
```

```
[car setValue: [NSNumber numberWithInt: 155]
```

```
 forKeyPath: @"engine.horsepower");//路径的获得
```

```
NSLog(@"horsepower is %@", [car valueForKeyPath: @"engine.horsepower"]);
```

```
NSArray *pressures = [car valueForKeyPath: @"tires.pressure");//要是路径是一个数组就  
只能获取一个数组，不能只获取一个
```

```
NSLog(@"pressures %@", pressures);
```

Advantage two

```
NSNumber *count;
```

```
count = [garage valueForKeyPath: @"cars.@count"];
```

```
NSLog(@"We have %@ cars", count);
```

```

NSNumber *sum;

sum = [garage valueForKeyPath: @"cars.@sum.mileage"];

NSLog(@"We have a grand total of %@ miles", sum);

NSNumber *avgMileage;

avgMileage = [garage valueForKeyPath: @"cars.@avg.mileage"];

NSLog(@"average is %.2f", [avgMileage floatValue]);

NSNumber *min, *max;

min = [garage valueForKeyPath: @"cars.@min.mileage"];

max = [garage valueForKeyPath: @"cars.@max.mileage"];

NSLog(@"minimax: %@ / %@", min, max);

NSArray *manufacturers;

manufacturers = [garage valueForKeyPath: @"cars.@distinctUnionOfObjects.make"];

NSLog(@"makers: %@", manufacturers);

//另外，union 运算符指一组对象的并集

//distinct 用于删除重复的内容

//遗憾的一点就是不能添加自的运算符

Advantage three

car = [[garage valueForKeyPath: @"cars"] lastObject];

```

```
NSArray *keys = [NSArray arrayWithObjects: @"make", @"model", @"modelYear", nil];
```

```
NSDictionary *carValues = [car dictionaryWithValuesForKeys: keys];
```

```
NSLog(@"Car values : %@", carValues);
```

```
NSDictionary *newValues =
```

```
[NSDictionary dictionaryWithObjectsAndKeys:
```

```
@"Chevy", @"make",
```

```
@"Nova", @"model",
```

```
[NSNumber numberWithInt:1964], @"modelYear",
```

```
[NSNull null], @"mileage",
```

```
nil];
```

```
[car setValuesForKeysWithDictionary: newValues];
```

```
NSLog(@"car with new values is %@", car);
```

```
//新装配过的 car
```

KVC 的一些特殊情况处理

case one: nil 的处理

```
[car setValue:nil forKey: @"mileage"];
```

```
NSLog(@"Nil miles are %@", car.mileage);
```

这里的标量值 `mileage` 中的 `nil` 表示的是什么呢？0？-1？pi？cocoa 无法知道，可以再 `car` 类里面重写

```
- (void) setNilValueForKey: (NSString *) key {
```

```
    if ([key isEqualToString: @"mileage"]) {
```

```
        mileage = 0;
```

```
    } else {
```

```
        [super setNilValueForKey: key];
```

```
    }
```

```
} // setNilValueForKey
```

case two: 未定义的键的处理

在控制类里面写

```
- (void) setValue: (id) value forKey: (NSString *) key {
```

```
    if (stuff == nil) {
```

```
        stuff = [[NSMutableDictionary alloc] init];
```

```
    }
```

```
    [stuff setValue: value forKey: key];
```

```
} // setValueForKey
```

```
- (id) valueForKey:(NSString *)key {
```

```
    id value = [stuff valueForKey: key];
```

```
return (value);
```

```
} // valueForKey
```

[BerwinZheng](#)

2010-2-13 19:37:51

64、Cocoa 中提供 `NSPredicate` 的类，它用于指定过滤的条件

Cocoa 用 `NSPredicate` 描述查询的方式，原理类似于在数据库中进行查询

计算谓词：

```
//基本的查询
```

```
NSPredicate *predicate;
```

```
predicate = [NSPredicate predicateWithFormat: @"name == 'Herbie'"];
```

```
BOOL match = [predicate evaluateWithObject: car];
```

```
NSLog (@"%s", (match) ? "YES" : "NO");
```

```
//在整个 cars 里面循环比较
```

```
predicate = [NSPredicate predicateWithFormat: @"engine.horsepower > 150"];
```

```
NSArray *cars = [garage cars];
```

```
for (Car *car in [garage cars]) {
```

```
    if ([predicate evaluateWithObject: car]) {
```

```
        NSLog (@"%@", car.name);
```

```
    }
```

```
}
```

```
//输出完整的信息
```

```
predicate = [NSPredicate predicateWithFormat: @"engine.horsepower > 150"];
```

```
NSArray *results;
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@", results);
```

```
//含有变量的谓词
```

```
NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:@"name == $NAME"];
```

```
NSDictionary *varDict;
```

```
varDict = [NSDictionary dictionaryWithObjectsAndKeys:
```

```
    @"Herbie", @"NAME", nil];
```

```
predicate = [predicateTemplate predicateWithSubstitutionVariables: varDict];
```

```
NSLog(@"SNORGL: %@", predicate);
```

```
match = [predicate evaluateWithObject: car];
```

```
NSLog(@"%s", (match) ? "YES" : "NO");
```

```
//注意不能使用$VARIABLE 作为路径名，因为它值代表值
```

```
//谓词字符串还支持 c 语言中一些常用的运算符
```

/*

*>: 大于

*>=和<=: 大于或等于

*<: 小于

*<=和<=: 小于或等于

*!=和<>: 不等于

*括号运算符, AND, OR, NOT, &&, ||, ! (可以不区分大小写, 但建议一致)

*/

```
predicate = [NSPredicate predicateWithFormat:
```

```
    @"(engine.horsepower > 50) AND (engine.horsepower < 200)"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"oop %@", results);
```

```
predicate = [NSPredicate predicateWithFormat: @"name < 'Newton'"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@", [results valueForKey: @"name"]);
```

//强大的数组运算符

```
predicate = [NSPredicate predicateWithFormat:
```

```

        @"engine.horsepower BETWEEN { 50, 200 }"];

results = [cars filteredArrayUsingPredicate: predicate];

NSLog (@"%@", results);


NSArray *betweens = [NSArray arrayWithObjects:

        [NSNumber numberWithInt: 50], [NSNumber numberWithInt: 200],
nil];

predicate = [NSPredicate predicateWithFormat: @"engine.horsepower
BETWEEN %@", between];

results = [cars filteredArrayUsingPredicate: predicate];

NSLog (@"%@", results);


predicateTemplate = [NSPredicate predicateWithFormat: @"engine.horsepower
BETWEEN $POWERS"];

varDict = [NSDictionary dictionaryWithObjectsAndKeys: between, @"POWERS", nil];

predicate = [predicateTemplate predicateWithSubstitutionVariables: varDict];

results = [cars filteredArrayUsingPredicate: predicate];

NSLog (@"%@", results);


//IN 运算符

predicate = [NSPredicate predicateWithFormat: @"name IN { 'Herbie', 'Snugs',

```



```
'Badger', 'Flap' }"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog (@"%@", [results valueForKey: @"name"]);
```

```
predicate = [NSPredicate predicateWithFormat: @"SELF.name IN { 'Herbie', 'Snugs',  
'Badger', 'Flap' }"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog (@"%@", [results valueForKey: @"name"]);
```

```
names = [cars valueForKey: @"name"];
```

```
predicate = [NSPredicate predicateWithFormat: @"SELF IN { 'Herbie', 'Snugs',  
'Badger', 'Flap' }"];
```

```
results = [names filteredArrayUsingPredicate: predicate];//这里限制了 SELF 的范围
```

```
NSLog (@"%@", results);
```

```
//BEGINSWITH,ENDSWITH,CONTAINS
```

//附加符号, [c],[d],[cd],c 表示不区分大小写, d 表示不区分发音字符, cd 表示什么都不区分

```
predicate = [NSPredicate predicateWithFormat: @"name BEGINSWITH 'Bad'"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog (@"%@", results);
```

```
predicate = [NSPredicate predicateWithFormat: @"name BEGINSWITH 'HERB'"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@@", results);
```

```
predicate = [NSPredicate predicateWithFormat: @"name BEGINSWITH[cd] 'HERB'"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@@", results);
```

//LIKE 运算符（通配符）

```
predicate = [NSPredicate predicateWithFormat: @"name LIKE[cd] '*er*'];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@@", results);
```

```
predicate = [NSPredicate predicateWithFormat: @"name LIKE[cd] '???er*'];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@@", results);
```