

Objective-C 培训教程

主要章节提示:

- 第 一 章 程序整体语法结构
- 第 二 章 数据类型
- 第 三 章 字符串
- 第 四 章 内存管理
- 第 五 章 对象的初始化
- 第 六 章 存取器
- 第 七 章 继承
- 第 八 章 动态绑定和 id 类型
- 第 九 章 分类和协议
- 第 十 章 属性列表
- 第十一章 复制对象
- 第十二章 归档

第一章 程序整体语法结构

程序的头文件和源文件的扩展名分别为.h 和.m。注释语法和 C 一样。Object_C 中的 nil 相当于 NULL。Object_C 中的 YES 和 NO 相当于 true 和 false。

这里再讲解一下 YES 和 NO:

Object-c 提供了 BOOL 类型,但这个 BOOL 类型和 C++里的并不一样:在 C++里一切非 0 值的东西都为 true,而为 0 值的为 false。但是 Object-c 里 1 为 true 并被宏定义为 YES,0 为 false 并被宏定义为 NO。所以,如果读者写下面的代码,则肯定是错误的:

```
BOOL areIntsDifferent_faulty(int thing1,int thing2)
{
    return (thing1-thing2);
}
if(areIntsDifferent_faulty(23,5) == YES)
{
}
```

因为 areIntsDifferent_faulty 方法返回的是两个整数的差,如果这个差不为 1,那么永远不会为 YES。

先了解程序的结构:

```
#import <Foundation/Foundation.h>
int main(int argc,const char * argv[])
{
```

```

NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init ];
NSLog(@"Programming is fun!");

[pool drain];
return 0;
}

```

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

相当于#include 导入头文件 也有两种查找方式< ... > 和" ... "。导入该头文件是因为在程序结尾处用到的其他类和函数的有关信息

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

这条语句为自动释放池在内存中保留了空间，就是在释放内存池的时候同时释放调其中的所有对象，若对象要加入该池，只要发送一条 autorelease 消息。

```
NSLog(@"Programming is fun!");
```

将显示常量字符串，类似于 printf 函数，并且它会自动在文本后面添加'\n'。当然其中也可以使用转义字符。例如还有

```
NSLog(@"The sum of 50 and 25 is %i",sum);
```

```
[pool drain]; //释放内存池
```

```
[classOrInstance method];
```

左方括号是类的名称或者该类实例的名称，空格后面是方法（即消息）

获得对象:(从 Car 类获得其对象)

```
youCar = [Car new];
```

定义一个新类分为 2 部分：

@interface 部分

描述类、类的数据成分以及类的方法

@implementation 部分

实现这些方法的实际代码

@interface 部分的一般格式：

```

@interface NewClassName : ParentClassName
{
    memberDeclarations;
}
methoddeclarations;

```

@end

命名规则：以字母或下划线开头，之后可以是任何字母，下划线或者 0~9 数字组合，

约定：类名以大写字母开头，实例变量、对象以及方法的名称以小写字母开始。

每次创建新对象时，将同时创建一组新的实例变量且唯一。注意：在对象类型的右边都有一个*号，所有的对象变量都是指针类型。Id 类型已经被预定义为指针类型，所以不需要加一个*号。

函数开头的 (-) 号或者 (+) 号表示：

(-) 该方法是实例方法（对类的特定实例执行一些操作）；

(+) 是类方法（即对类本身执行某些操作的方法，例如创建类的新实例）

函数的声明示例：

- (void) setNumerator : (int) n

第一个表示方法类型，返回类型，接着是方法名，方法接受的参数，参数类型，参数名

注：如果不带参数则不使用 “:” 号

如果没有指定任何返回类型，那么默认是id类型，所有的输入参数默认也是id类型（id类型可用来引用任何类型的对象）。

或许到现在你会认为将对象赋给id类型变量会有问题。

注：无论在哪里，对象总是携带它的 isa 的保护成员（可以用来确定对象所属的类），所以即使将它存储在id类型的通用对象变量中，也总是可以确定它的类。

具有多个参数的方法：

-(return type) function_name : (parameter type) parameter1 otherParameter : (parameter_type) parameter2;

如果只有一个参数，在 : 后面声明参数的类型和名称；如果有多个参数的话，每个参数前面都要有一个 : , 然后接着是参数类型和参数名称。可是大家可能还是觉得很奇怪。比如上面这个例子中，otherParameter 这个是什么意思，在 objective c 中，对于有多个参数的函数，可以理解为将函数的名称拆成了几个部分，每个部分都是对紧接着的参数一个解释。

如在 C++中：

```
void initializeRectangle(int x1, int y1, int x2, int y2);
```

但并不知道这些参数都是什么意思；但在 objective c 中，可以这样声明：

```
void initializeRectangle: (int)x1 LeftUpY: (int)y1 RightBottomX: (int)x2  
                        RightBottomY:(int)y2;
```

@implementation 部分的一般格式：

```
@implementation NewClassName  
    methodDefinitions;  
@end
```

//NewClassName 表示的名称与@interface 部分的类名相同。

一个简单的示例：

```
/**
 *
 */
//Fraction.h 文件
#import <Foundation/Foundation.h>

@interface Fraction : NSObject
{
    int numerator;
    int denominator;
}

- (void) print;
- (void) setNumerator : (int) n;
- (void) setDenominator : (int) d;
@end

//Fraction.m 实现文件
@implementation Fraction

- (void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

- (void) setNumerator : (int) n
{
    numerator = n;
}

- (void) setDenominator : (int) d
{
    denominator = d;
}

@end

/**
 *
 */
```

```
Fraction * myFraction = [[Fraction alloc] init] ;
```

获得对象的实例并且初始化了其实例变量（可以这样理解：将 alloc 消息发送给 Fraction 类请求创建一个新实例，然后向新创建的实例对象发送 init 消息来初始化该对象）。

另外一种方法：

```
Fraction * myFraction = [Fraction new]; 但是通常使用第一种方式
```

对象调用方法 [myFraction setNumerator : 1];

用完释放 Fraction 对象的方法: [myFraction release];

注: 创建一个新对象, 都要请求分配内存, 在完成对该对象的操作时, 必须释放其所用的内存空间

i Phone 平台不支持垃圾回收机制

外部要访问实例变量需要通过类的方法来检索其值, 不能直接访问

示例:

```
/**
 * *****
 */
```

//Rectangle.h 文件

```
@interface Rectangle : NSObject
```

```
{
```

```
    int    width ;
```

```
    int    heigth ;
```

```
}
```

```
@property int width , heigth;
```

```
- (int) area ;
```

```
- (int) perimeter ;
```

```
- (void) setWidth : (int) w andHeigth : (int) h ;
```

```
@end
```

//Rectangle.m 文件

```
#import "Rectangle.h"
```

```
@implementation Rectangle
```

```
@synthesize width , heigth ;
```

```
- (void) setWidth : (int) w andHeigth : (int) h
```

```
{
```

```
    width = w ;
```

```
    heigth = h ;
```

```
}
```

```
- (int) area
```

```
{
```

```
    return width * heigth ;
```

```
}
```

```
- (int) perimeter
```

```
{
```

```
    return (width + heigth) * 2 ;
```

```
}  
@end
```

下面是 Rectangle 的子类 Square

//Square.h 文件

```
#import "Rectangle.h"  
  
@interface Square : Rectangle  
- (void) setSide : (int) s ;  
- (int) side ;  
  
@end ;
```

//Square.m 文件

```
#import "Square.h"  
  
@implementation Square : Rectangle  
- (void) setSide : (int) s  
{  
    [self setWidth : s andHeight:s]  
}  
- (int) side  
{  
    return width ;  
}  
  
@end ;  
  
/*****
```

self 关键字用来指明对象是当前方法的接收者。

例如下面是一个子类（正方形）的方法实现：

```
- (void) setSide: (int) s  
{  
    [self setWidth : s andHeight:s]  
}
```

利用其父类（长方形）的 setWidth: andHeight:方法来实现的。

调用消息的类可以不知道如何响应这个消息。如果它不知道如何处理这个消息，它会自动的将这个消息转给的父类，还不行就转给父类的父类，都没有找到就会报错。

与 C 语言兼容的地方：

预处理：

#define 语句和 c 一样

#运算符: `#define str(x) #x`

表示在调用该宏时, 预处理程序根据宏参数创建 C 风格的常量字符串。

例如: `str("hello")` 将产生 `"hello"`

##运算符:

表示用于把两个标记连在一起

`#import` 语句相当于 `#include` 语句, 但是 `#import` 可自动防止同一个文件被导入多次。

#条件编译语句 (`#ifdef`、`#endif`、`#else`、`#ifndef`) 和 C 一样

#undef 语句 消除特定名称的定义

其他基本的 C 语言特性:

数组、函数、指针、结构、联合的用法和 C 一样。

Compound Literal 是包含在括号之内的类型名称, 之后是一个初始化列表。

例如 如果 `intPtr` 为 `int *` 类型:

```
intPtr = (int[100]) {[0] = 1, [50] = 50, [99] = 99};
```

如果数组大小没有说明, 则有初始化列表确定。

其他如循环语句 (`do while`、`while`、`for`)、条件语句 (`if` 语句 (`if-else`、复合判断条件等)、`switch` 语句)、`Boolean`(YES NO)、条件运算符、`goto` 语句、空语句、逗号表达式、`sizeof` 运算符、命令行参数、位操作都和 C 一样。

第二章 数据类型

Object-c 提供基本数据类型：int 、float 、double 、char

Int:

八进制 整型第一位为 0， NSLog 的格式符为： %o 显示的八进制不带前导 0

%#o 显示的八进制带前导 0

十六进制 以 0x 开头的整型， NSLog 的格式符为： %x 显示的十六进制不带前导 0x

%#x 显示的十六进制带前导 0x

若（%X 或%#X） 显示的十六进制用大写

Float:

NSLog 的格式符： %f

NSLog 的格式符： %e 科学计数法显示值

NSLog 的格式符： %g 指数的值小于-4 大于 5， 采用%e， 否则采用%f

十六进制的浮点常量包含前导 0x 或 0X,后面紧跟一个或多个十进制或十六进制数字， 再后是 p 或 P， 最后是可以带符号的十进制指数。例： 0x0.3p10 表示的值为 $3/16 \times 2^{10}$

注： 若无特殊说明， Object-c 将所有的浮点常量看做 double 值， 要显示 double 值可使用和 float 一样的格式符。

Char:

NSLog 的格式符： %c

long double 常量写成尾部带有字母 l 或者 L 的浮点常量。 1.234e+7L

类型	NSLog 格式符		
	八进制	十六进制	十进制
long int	%lo	%lx	%li
long long int	%llo	%llx	%lli
long double	%Lg	%Le	%Lf
short int	%ho	%hx	%hi
unsigned short int	%ho	%hx	%hu
unsigned int	%o	%x	%u
unsigned long int	%lo	%lx	%lu
unsigned long long int	%llo	%llx	%llu
id	%p		

注： id 类型可以通过类型转化符可以将一般的 id 类型的对象转换成特定的对象。

<code>_Bool</code>	处理 Boolean（即 0 或 1）
<code>_Complex</code>	处理复数
<code>_Imaginary</code>	处理抽象数字

键盘输入：

```
int number;  
scanf("%i",&number);
```

实例变量的初始化值默认为 0

实例变量作用域的指令：

`@protected` 实例变量可被该类及任何子类中定义的方法直接访问（默认的情况）。

`@private` 实例变量可被定义在该类的方法直接访问，不能被子类定义的方法直接访问。

`@public` 实例变量可被该类中定义的方法直接访问，也可被其他类或模块中定义的方法访问。使得其他方法或函数可以通过（->）来访问实例变量（不推荐用）。

`@package` 对于 64 位图像，可以在实现该类的图像的任何地方访问这个实例变量。

在类中定义静态变量和 C 一样

`volatile` 说明符和 `const` 正好相反，明确告诉编译器，指定类型变量的值会改变。（I/O 端口）

比如要将输出端口的地址存储在 `outPort` 的变量中。

```
volatile char *outPort;  
*outPort = 'O';  
*outPort = 'N';
```

这样就可以避免编译器将第一个赋值语句从程序中删除

枚举数据类型、`typedef` 语法以及数据类型的转换和 C 也是一样。

第三章 字符串

一些有用的数据类型:

表示范围作用的结构体: NSRange:

有三种方式创建新的 NSRange

1、NSRange range;

range.location = 17;

range.length = 4;

2、NSRange range = {17, 4};

3、NSRange range = NSMakeRange (17, 4); (推荐)

表示用来处理几何图形的数据类型: NSPoint (点坐标) 和 NSSize (长度和宽度) 还有一个矩形数据类型 (由点和大小复合而成) NSRect

Cocoa 提供创建这些数据类型方法: NSMakePoint (), NSMakeSize () 和 NMakeRect ()

表示字符串的类 NSString

NSString *height = [NSString stringWithFormat:@"You height is %d feet,%d inches", 5,11];

创建的类对象包含了指向超类的指针、类名和指向类方法的列表的指针。类对象还包含一个 long 的数据, 为新创建的类对象指定大小。

返回字符串中的字符的个数:

unsigned int length = [height length];

返回 Bool 值的字符串比较函数:

- (BOOL) isEqualToString: (NSString *) aString; //比较两个字符串的内容是否相等

还可以使用 compare: 方法

- (NSComparisonResult) compare: (NSString *) string; //逐个字符比较

不区分大小写的比较:

- (NSComparisonResult) compare: (NSString *) string options: (unsigned) mask;

注意: NSComparisonResult 是一个枚举值

options 是一个位掩码, 即:

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

检查字符串是否以另一个字符串开头

- (BOOL) hasPrefix: (NSString *) aString;

判断字符串是否是以另一个字符串结尾

- (BOOL) hasSuffix : (NSString *)aString;

如果你想知道字符串内的某处是否包含其他字符串，使用 rangeOfString : 方法

- (NSRange) rangeOfString : (NSString *) aString;

NSString 是不可变的，NSMutableString 是可变的。用方法 stringWithCapacity : 来创建

NSMutableString *string = [NSMutableString stringWithCapacity : 42];

可以使用 appendString : 或 appendFormat : 来附加新字符串：

- (void) appendString : (NSString *) aString;

- (void) appendFormat : (NSString *) format , ... ;

可以使用 deleteCharactersInRange : 方法删除字符串中的字符

- (void) deleteCharactersInRange : (NSRange) range;

集合家族：

NSArray: 用来存储对象的有序列表（任意类型的对象）

限制: 只能存储 Objective—C 的对象, 不能存储 C 语言的基本数据类型(int, float, enum, struct、或者 NSArray 中的随机指针)。同时也不能存储 nil（对象的零值或 NULL 值）

//创建一个新的 NSArray

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

//获得包含的对象个数

- (unsigned) count;

//获得特定索引处的对象

- (id) objectAtIndex : (unsigned int) index ;

切分数组：

使用 componentsSeparatedByString : 来切分 NSArray,

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

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

使用 componentsJoinedByString : 来合并 NSArray 中的元素并创建字符串

string = [chunks componentsJoinedByString : @":-");

NSArray 是不可变数组，数组中包含的对象是可以改变的，但是数组对象本身是不会改变的。

可变数组 NSMutableArray 通过类方法 arrayWithCapacity : 来创建可变数组

+ (id) arrayWithCapacity : (unsigned) numItems ;

NSMutableArray *array = [NSMutableArray arrayWithCapacity : 17];

使用 addObject : 在数组末尾添加对象

- (void) addObject : (id) anObject

删除特定索引的对象

- (void) removeObjectAtIndex : (unsigned) index;

注：可变数组还可以在特定索引处插入对象，替换对象，为数组排序，NSArray 还提供了大量好用的功能。

枚举：

NSEnumerator 用来描述这种集合迭代器运算的方法：

要想使用 NSEnumerator，需要通过 objectEnumerator 向数组请求枚举器：

- (NSEnumerator *) objectEnumerator;

可以像这样使用这个方法：

NSEnumerator *enumerator;

enumerator = [array objectEnumerator];

注：若想从后向前枚举集合，使用方法 reverseobjectEnumerator；也可以使用。

获得枚举器以后，开始 while 循环，每次循环都向这个枚举器请求它的 nextObject

- (id) nextObject; //返回 nil 表明循环结束

注：在枚举的过程中不能改变数组容器。

快速枚举示例：

```
for (NSString *string in array) {  
    NSLog(@"I found %@", string);  
}
```

数组排序：

例如：一条记录就是一条卡片的信息，包括 (NSString *name 和 NSString *email)

```
-(void) sort  
{  
    [book sortUsingSelector:@selector(compareNames:)]  
}
```

其中：

@selector(compareNames:)

//创建一个 SEL 类型的 selector，sortUsingSelector:使用该方法比较数组中的两个元素，

sortUsingSelector:方法需要完成这样的比较，它先调用这个指定的 selector 方法，然后向数组（接受者）的第一条记录发送消息，比较其参数和此记录。指定方法的返回值为 NSComparisonResult 类型，返回值为：若小于返回 NSOrderedAscending；相等返回 NSOrderedSame；大于返回 NSOrderedDescending

```
-(NSComparisonResult)compareNames:(id)element  
{  
    return [name compare:[element name]];  
}
```

NSDictionary：（关键字和定义的组合）

NSDictionary 通常在给定一个关键字（通常是一个 NSString 字符串）下存储一个数值（可以是任何类

型的对象)。然后你可以使用这个关键字查找相应的数值。

使用 dictionaryWithObjectsAndKeys : 来创建字典

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

例如:

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

```
NSDictionary *tires = [ NSDictionary dictionaryWithObjectsAndKeys : t1,@"front-left",nil];
```

使用方法 objectForKey : 来获取字典中的值

- (id) objectForKey : (id) akey;

查找轮胎可以这样:

```
Tire *tire = [tires objectForKey : @"front-left"];
```

创建新的 NSMutableDictionary 对象, 向类 NSMutableDictionary 发送 dictionaryWithCapacity : 方法来创建新的可变字典

+ (id) dictionaryWithCapacity : (unsigned int) numItems ;

可以使用方法 setObject : forKey : 方法给字典添加元素

setObject : forKey : (id) aKey

下面是另一种使用发送 dictionaryWithCapacity : 消息来创建字典的方法:

```
NSMutableDictionary *tires;
```

```
tires = [NSMutableDictionary dictionaryWithCapacity : 1];
```

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

```
...
```

注: 若对字典中已有的关键字使用 setObject : forKey : 方法, 则用新的替换

可以使用 removeObjectForKey : 方法来删除可变字典中的一个关键字

- (void) removeObjectForKey : (id) aKey ;

注: 不要去创建 NSString、NSArray 或 NSDictionary 的子类, 实在要的话可以用复合的方式来解决这个问题。

使用这种方法枚举词典:

```
for (NSString *key in g)
{
    ...
}
```

集合对象:

Set 是一组单值对象的集合, 有可变和不可变, 操作包括: 搜索、添加、删除集合中的成员 (仅用于可变集合)、比较两个集合, 计算两个集合的交集和并集等。

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

常用的 NSSet 方法

方法	说明
+ (id) initWithObjects:obj1,obj2,...,nil	使用一系列对象创建新集合
- (id) initWithObjects:obj1,obj2,...,nil	使用一系列对象初始化新分配的集合
- (NSUInteger) count	返回集合的成员个数
- (BOOL) containsObject: obj	确定集合是否包含 obj
- (BOOL) member: obj	使用 isEqual: 方法确定集合是否包含 obj
- (NSEnumerator *) objectEnumerator	为集合中的所有对象返回一个 NSEnumerator 对象
- (BOOL) isSubsetOfSet: nsset	确定 receiver 的每个成员是否都出现在 nsset 中
- (BOOL) intersectsSet: nsset	确定是否 receiver 中至少一个成员出现在对象 nsset 中
- (BOOL) isEqualToSet: nsset	确定两个集合是否相等

常用的 NSMutableSet 方法 (NSSet 的子类)

方法	说明
- (id) initWithCapacity:size	创建新集合，使其具有存储 size 个成员的初始空间
- (id) initWithCapacity:size	将新分配的集合设置为 size 个成员的存储空间
- (void) addObject:obj	将对象 obj 添加到集合中
- (void) removeObject:obj	从集合中删除对象 obj
- (void) removeAllObjects	删除接受者的所有成员
- (void) unionSet:nsset	将对象 nsset 的所有成员添加到接受者
- (void) minusSet:nsset	从接受者中删除 nsset 的左右成员
- (void) intersectSet:nsset	将接受者中的所有不属于 nsset 的元素删除

注：NSInteger 不是一个对象，基本数据类型的 typedef，被 typedef 成 64 位的 long 或 32 位 int

各种数值：

NSNumber：

可以使用对象来封装基本数值。

NSNumber 类来包装基本数据类型。

```
+ (NSNumber *) numberWithChar : ( char ) value ;
+ (NSNumber *) numberWithInt : ( int ) value ;
+ (NSNumber *) numberWithFloat : ( float ) value ;
+ (NSNumber *) numberWithBool : ( BOOL ) value ;
```

还有包括无符号版本和各种 long 型数据及 long long 整型数据

例如：NSNumber *number = [NSNumber numberWithInt: 42];

将一个基本类型封装到 NSNumber 后，可以使用下列方法重新获得：

- (char) charValue;
- (int) intValue;
- (float) floatValue;
- (BOOL) boolValue;
- (NSString *) stringValue;

NSNumber:

NSNumber 实际上是 NSNumber 的子类，NSNumber 可以封装任意值。可以用 NSNumber 将结构放入 NSArray 和 NSDictionary 中。

创建新的 NSNumber:

```
+ (NSNumber *) numberWithInt:(const void *) value
      objcType:(const char *)type;
```

@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。

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

```
NSNumber * value ;
```

```
value = [NSNumber numberWithInt:&rect objcType:@encode(CGRect)];
```

使用 getValue : 来提取数值 (传递的是要存储这个数值的变量的地址) (先找地址再取值)

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

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

注: Cocoa 提供了将常用的 struct 型数据转化成 NSNumber 的便捷方法:

- ```
+ (NSNumber *) numberWithInt:(NSPoint) point ;
+ (NSNumber *) numberWithInt:(NSSize) size;
+ (NSNumber *) numberWithInt:(CGRect) rect ;
- (NSSize) sizeValue;
- (CGRect) rectValue;
- (NSPoint) pointValue;
```

### **NSNumber:**

在关键字下如果属性是 NSNumber 表明没有这个属性,没有数值的话表明不知道是否有这个属性。[NSNumber null] //总返回一样的值

```
+ (NSNumber *) null;
```

例如:

```
[contact setObject:[NSNumber null] forKey:@"home"];
```

访问它:

```
id home = [contact objectForKey:@"home"];
```

```
if (home == [NSNumber null]) {
```

```
...
```

```
}
```

NSFileManager 允许对文件系统进行操作（创建目录、删除文件、移动文件或获取文件信息）

//创建一个属于自己的 NSFileManager 对象

```
NSFileManager *manager = [NSFileManager defaultManager];
```

//将代字符 ‘~’ 替换成主目录

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

//输出文件的扩展名

```
- (NSString *) pathExtension
```

示例：翻查主目录，查找.jpg 文件并输出找到的文件列表

```
//*****
```

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

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

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

```
 NSFileManager *manager;
```

```
 manager = [NSFileManager defaultManager];
```

```
 NSString *home;
```

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

```
 NSDirectoryEnumerator *direnum;
```

```
 direnum = [manager enumeratorAtPath: home];
```

```
 NSMutableArray *files;
```

```
 files = [NSMutableArray arrayWithCapacity: 42];
```

```
 NSString *filename;
```

```
 while (filename = [direnum nextObject]) {
```

```
 if ([[filename pathExtension] isEqualToString: @"jpg"]) {
```

```
 [files addObject: filename];
```

```
 }
```

```
 }
```

```
 NSEnumerator *fileenum;
```

```
 fileenum = [files objectEnumerator];
```

```
 while (filename = [fileenum nextObject]) {
```

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



```
}
```

```
[pool drain];
```

```
return 0;
```

```
}
```

```
//*****
```

## 第四章 内存管理

### 自动释放池：

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

事实上程序中可以有多个自动释放池。自动释放池其实并不包含实际的对象本身，仅仅是对释放的对象的引用。通过向目前的自动释放池发送一条 autorelease 消息，可以将一个对象添加到其中。

### 引用计数：

概念：创建对象时，将它的引用次数设置为 1，每一次必须保持该对象时，就发送一条 retain 消息，使其引用次数加 1。

```
[myFraction retain];
```

不再需要对象时，可以通过发送 release 消息，使对象的引用次数减 1。

```
[myFraction release];
```

当引用计数为 0 的时候，系统就会释放它的内存，通过向对象发送 dealloc 消息。通过向对象发送 retainCount 消息可以获得这个对象的引用计数，返回的是 NSUInteger 整数。只要对象的引用计数不为 0，系统就不会释放对象使用的内存。

将对象添加到任何类型的集合中都会使该对象的引用计数增加。从任何集合中删除对象都能够使其引用计数减少。

内存中常量字符串没有引用计数机制，因为永远不能释放这些对象。这也适用于使用常量字符串初始化的不可变字符串对象。

当某段代码需要访问一个对象的时候，将对象的引用计数器加 1，当其为 0 表明不再有代码访问该对象了，即对象将被销毁（通过调用 dealloc 方法）。

一个简单的示例：

```
int main (int argc, const char * argv[])
{
 NSAutoreleasePool *pool;
 pool = [[NSAutoreleasePool alloc] init];

 RetainTracker *tracker;
 tracker = [RetainTracker new]; // count: 1
```

```

[tracker retain]; // count: 2
[tracker autorelease]; // count: still 2
[tracker release]; // count: 1

NSLog(@"releasing pool");
[pool release];
// gets nuked, sends release to tracker
return (0);
}

```

当使用 `alloc`、`new` 或者通过 `copy` 消息（生成接受对象的一个副本）创建一个对象，对象的引用计数器被设置成 1。发送 `retain` 消息将增加引用计数器，`release` 消息减 1。

要获得引用计数器的当前值，可以发送 `retainCount` 消息

```

- (id) retain ;
- (void) release ;
- (unsigned) retainCount ;

```

访问方法中的保留和释放：

最好的方法设置的原则是（保持新的释放旧的，谁拥有对象谁就复制释放对象）

```

- (void) setEngine : (Engine *) newEngine
{
 [newEngine retain];
 [engine release];
 engine = newEngine;
}

```

### 内存管理规则

| 获得途径                        | 临时对象      | 拥有对象                                   |
|-----------------------------|-----------|----------------------------------------|
| <code>alloc/new/copy</code> | 不再使用时释放对象 | 在 <code>dealloc</code> 方法中释放对象         |
| 任何其他方法                      | 不需要执行任何操作 | 获得对象时保留，在 <code>dealloc</code> 方法中释放对象 |

### 内存管理规则摘要：

- 1、释放对象，可以释放其所占的内存，规则是：不再使用创建或者保持的对象时，就释放它们。
- 2、发送一条 `release` 消息不一定销毁对象，只有引用计数变为 0 时，才销毁这个对象。系统向对象发送一条 `dealloc` 消息来释放它所占的内存。
- 3、自动释放池在释放池本身的时候自动释放池中的对象。系统向每个对象发送一条 `release` 消息，对引用计数变为 0 的对象发送一条 `dealloc` 消息来释放它所占的内存。
- 4、若你的方法中不再需要一个对象时，但需要返回它，可发送一条 `autorelease` 消息来加入自

动释放池。

- 5、若使用 `alloc` 或 `copy` 方法（或使用 `allocWithZone:`、`copyWithZone:` 或 `mutableCopy` 方法来直接创建对象，则由你负责释放它。每次 `retain` 对象时，应该 `release` 或 `autorelease` 它。
- 6、除了上以规则中提到的方法之外，不必费心地释放其他方法返回的对象，这不是你的责任。

## 第五章 对象的初始化

```
/** *****
```

```
//Tire.h 文件
```

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

```
@interface Tire : NSObject {
```

```
 float pressure;
```

```
 float treadDepth;
```

```
}
```

```
- (id) initWithPressure: (float) pressure;
```

```
- (id) initWithTreadDepth: (float) treadDepth;
```

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

```
 treadDepth: (float) treadDepth; //指定的初始化函数
```

```
- (void) setPressure: (float) pressure;
```

```
- (float) pressure;
```

```
- (void) setTreadDepth: (float) treadDepth;
```

```
- (float) treadDepth;
```

```
@end // Tire
```

```
//Tire.m 文件
```

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

```
@implementation Tire
```

```
- (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
```

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

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

```
{
```

```
 if (self = [super init]) {
 pressure = p;
 treadDepth = td;
 }
```

```
return (self);
```

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

```
- (void) setPressure: (float) p
```

```
{
```

```
 pressure = p;
```

```

} // setPressure
- (float) pressure
{
 return (pressure);
} // pressure

- (void) setTreadDepth: (float) td
{
 treadDepth = td;
} // setTreadDepth

- (float) treadDepth
{
 return (treadDepth);
} // treadDepth

- (NSString *) description
{
 NSString *desc;
 desc = [NSString stringWithFormat:
 @"Tire: Pressure: %.1f TreadDepth: %.1f",
 pressure, treadDepth];
 return (desc);
} // description

@end // Tire
//*****

```

通常的写法: - (id) init

```

{
 if (self = [super init]) {
 ...
 }
 return (self);
}

```

注：在自己的初始化方法中，需要调用自己的指定的初始化函数或者超类的指定的初始化函数。一定要将超类的初始化函数的值赋给 self 对象，并返回你自己的初始化方法的值。超类可能决定返回一个完全不同的对象。

有些类包含多个以 `init` 开头的方法：

例如 `NSString` 类中的一些初始化方法：

```
NSString *emptyString = [[NSString alloc] init];
```

//返回一个空的字符串

```
NSString *string = [[NSString alloc] initWithFormat : @"%d or %d",25,624];
```

//返回一个字符串,其值为 25or624

```
NSString *string = [[NSString alloc] initWithContentOfFile : @"/tmp/words.txt"];
```

//使用指定路径上的文件中的内容初始化一个字符串

### 初始化函数的规则：

- 1、若不需要为自己的类创建初始化函数方法，只需要 `alloc` 方法将内存清 0 的默认行为，不需要担心 `init` 方法。
- 2、若构造一个初始化函数，则一定要在自己的初始化函数中调用超类的指定的初始化函数。
- 3、若初始化函数不止一个，则需要选定一个指定的初始化函数，被选定的方法应该调用超类的指定的初始化函数。



## 第六章 存取器

```
/**
 *
 */
//Car.h 文件
#import <Cocoa/Cocoa.h>

@class Tire;
@class Engine;

@interface Car : NSObject {
 NSString *name ;
 NSMutableArray *tires ;
 Engine *engine ;
}

@property (copy) NSString *name ;
@property (retain) Engine *engine ;

- (void) setTire : (Tire *) tire atIndex : (int) index
- (Tire *) tireAtIndex : (int) index ;
- (void) print ;
@end //Car
/**
 *
 */

/**
 *
 */
//Car.m 文件
#import "Car.h"

@implementation Car

@synthesize name ;
@synthesize engine ;

- (id) init
{
 if(self = [super init]){
 name = @"Car" ;
 tires = [[NSMutableArray alloc] init] ;
 int i ;
 for (i = 0;i < 4;i++){
```

```

 [tires addObject : [NSNull null]] ;
 }
}

return (self) ;
}

- (void) dealloc
{
 [name release] ;
 [tires release] ;
 [engine release] ;
 [super dealloc] ;
} //dealloc

- (void) setTire : (Tire *) tire atIndex : (int) index
{
 [tires replaceObjectAtIndex : index withObject : tire] ;
} //setTire:atIndex

- (Tire *) tireAtIndex : (int) index
{
 Tire *tire;
 tire = [tires objectAtIndex : index] ;
 return (tire) ;
} //tireAtIndex

- (void) print
{
 NSLog(@"%@ has:", [self tireAtIndex : i]) ;
 int i ;
 for (i = 0 ; i < 4 ; i++){
 NSLog(@"%@", [self tireAtIndex : i]) ;
 }
 NSLog(@"%@", engine) ;
} //print

@end //Car

//*****

```

最后就可以在 main ( ) 函数中使用点表示法给对象赋值

```
Car *car = [[Car alloc] init] ;
```

```
car.name = @"Herbie";
```

```
car.engine = [[Slant6 alloc] init]; //Slant6 是 Engine 的子类
```

若在类中定义属性: (接口中)

`@property float rainHandling` //表明类的对象具有 float 类型的属性, 其名称: rainHandling, 而且可以调用 `- setRainHandling:` 来设置属性, 调用 `- rainHandling` 来访问属性。`@property` 的作用是自动声明属性的 setter 和 getter 方法。

实现中:

```
@synthesize rainHandling //表示创建该属性的访问器
```

有时你可能希望实例变量有另一个名称, 而公开的属性有另一个名称:

方法:

只要在.h 文件中修改实例变量, 然后修改 `@synthesize name = appel;` 编译器还将创建 `-setName:` 和 `-name` 方法, 但在其实现中使用 `appel`。

添加特性:

```
@property (readwrite, copy) NSString *name; //对象可读写, 对象将被复制
```

```
@property (readwrite, retain) NSString *name; //对象可读写, 对象将被保持
```

```
@property (readonly) NSString *name; ////对象只读
```

点表达式的妙用:

点表达式 (.) 在等号左边, 该属性名称的 setter 方法将被调用。若在右边, 则可以调用属性名称的 getter 方法。

注意: 在使用特性的时候经常出现, 提示访问的对象不是 struct 类型, 请检查你是否包含了使用的类所需要的所有必须的头文件

该技术同样适用于 int、char、BOOL、struct 甚至可以定义一个 NSRect 类的对象的特性。

补充：

- 1、C/C++中支持的内存方式 Objective-C 都支持（例如 new,delete 或 malloc,free），Objective-C 也有自 己对象分配内存的方法：alloc,allocWithZone。如果出现内存警告，需要手动清除不必要的内存对象。如果还不够用，内存继续增长，系统会强制应用退出。
- 2、数据类型的字节数对应表：

| 类型标识符                  | 长度（字节） | 范围                                              | 备注                          |
|------------------------|--------|-------------------------------------------------|-----------------------------|
| char                   | 1      | -128 ~ 127                                      | $-2^7 \sim (2^7 - 1)$       |
| int                    | 4      | -2147483648 ~ 2147483647                        | $-2^{31} \sim (2^{31} - 1)$ |
| short int              | 2      | -32768 ~ 32767                                  | $-2^{15} \sim (2^{15} - 1)$ |
| long int               | 4      | -2147483648 ~ 2147483647                        | $-2^{31} \sim (2^{31} - 1)$ |
| long long int          | 8      |                                                 | $-2^{63} \sim (2^{63} - 1)$ |
| long double            | 10     |                                                 | 19 位有效位                     |
| double                 | 8      |                                                 | 15 位有效位                     |
| float                  | 4      | $1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$ | 7 位有效位                      |
| unsigned short int     | 2      |                                                 | $0 \sim (2^{16} - 1)$       |
| unsigned int           | 4      |                                                 | $0 \sim (2^{32} - 1)$       |
| unsigned long int      | 4      |                                                 | $0 \sim (2^{32} - 1)$       |
| unsigned long long int | 8      |                                                 | $0 \sim (2^{64} - 1)$       |
| BOOL                   | 1      | YES 或 NO                                        |                             |

## 第七章 继承

关于继承，主要讲一下三点：

1. Objective-C 不支持多继承。
2. Square 类继承于 Rectangle 类的继承示例

```

//*****//
//Rectangle 类 声明
#import<Foundation/Foundation.h>
@interface Rectangle: NSObject //继承于根类 NSObject
{
 int width;
 int height;
}
@property int width, height; // 存取器属性
-(int) area;
-(int) perimeter;
@end

// Rectangle 类 定义
#import "Rectangle.h"
@implementation Rectangle
@synthesize width, height;
- (int) area
{
 return width * height;
}
-(int) perimeter
{
 return (width + height) * 2;
}
@end

// Square 类 声明
#import<Foundation/Foundation.h>
#import "Rectangle.h"

@interface Square: Rectangle // 继承
//Objective-C 不支持多继承，如果将该语句改为
//@interface Square: Rectangle, Rectangle1 编译器将不能识别
//可以通过 Objective-C 的分类和协议特性获取多继承的优点。
-(void) setSide: (int) s; //因为没有利用存取器

```

```

-(int) side;
@end

// Square 类 定义
#import "Square.h"
@implementation Square

-(void) setSide: (int) s
{
 [self setWidth: s andHeight: s];
 // self 指令在自身类中查找 setWidth: andHeight: 方法，查找不到，则调用其父类中的该方法
}

-(int) side
{
 return width;
}
@end
//*****//

```

### 3. @class 指令

Rectangle 类只存储了矩形大小。现在要添加原点(x,y)的概念。因此，定义一个名为 XYPoint 的类。

```

//*****//

// XYPoint 类 声明

#import<Foundation/Foundation.h>

@interface XYPoint: NSObject
{
 int x;

 int y;
}

@property int x, y; //存取器属性

-(void) setX: (int)xVal andY: (int) yVal;

@end

// XYPoint 类 定义

#import "XYPoint.h"

@implementation XYPoint

@synthesize x, y;

```

```
-(void) setX: (int) xVal andY: (int) yVal
```

```
{
```

```
 x = xVal;
```

```
 y = yVal;
```

```
}
```

```
@end
```

```
//声明
```

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

```
@class XYPoint; //代替#import "XYPoint.h"
```

//使用@class 指令提高效率，编译器不需要处理整个 XYPoint.h 文件，//只需要知道 XYPoint 是一个类名，但是如果需要引用 XYPoint 类中方//法，@class 指令是不够的，必须用#import "XYPoint.h"。

```
@interface Rectangle: NSObject
```

```
{
```

```
 int width;
```

```
 int height;
```

```
 XYPoint *origin;
```

```
}
```

```
-(int) area;
```

```
-(int) perimeter;
```

```
@end
```

```
/**/
```

## 第八章 动态绑定和 **id** 类型

```
/**
 *
 */
#import "Fraction.h"

#import "Complex.h" //两个类中都含有 print 方法

int main(int argc, char *argv[])
{
 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

 Fraction *f = [[Fraction alloc] init];

 Complex *c = [[Complex alloc] init];

 id dataValue; //声明 dataValue 为 id 类型

 //对对象的成员变量赋值

 [f setTo: 2 over: 5];

 [c setReal:10.0 andImaginary: 2.5];

 dataValue = f; //将 Fraction f 存储到 dataValue 中

 [dataValue print]; //现在 dataValue 可以调用用于 Fraction 对象的任何方法

 dataValue = c; //将 Complex c 存储到 dataValue 中

 [dataValue print]; //调用用于 Complex 对象的任何方法

 //问题：两次遇到 [dataValue print]; 并且 Fraction 和 Complex 类中都定义有 print 方法，系统如何知道
 调用哪个？

 //答案：在程序执行期间，当系统准备将 print 消息发送给 dataValue 时，它首先检查 dataValue 中存储
 的对象所属的类。

 [f release];

 [c release];

 [pool drain];

 return 0;

}
/**
 *
 */
```



## 第九章 分类和协议

### 9.1 分类

通过分类(category)可以以模块的方式向现有的类添加方法。它提供了一种简单的方式，用它可以类定义模块化到相关方法的组或分类中。它还提供了扩展现有类定义的简便方式，并且不必访问类的源代码，也无需创建子类。

对于 Fraction 类，除了将两个分数相加的 add:方法外，还想要拥有将两个分数相减、相乘和相除的方法。

```
/**
 *
 */
// Fraction 类声明
#import<Foundation/Foundation.h>

@interface Fraction : NSObject
{
 int numerator;
 int denominator;
}

@property int numerator, denominator;

-(void) setTo: (int) n over: (int) d;

-(Fraction *) add: (Fraction *) f; // 声明分数的 加法函数

-(void) print;

@end
/**
 *
 */
```

现在，从接口部分删除 add:方法，并将其添加到新分类，同时添加其他三种要实现的数学运算。看一下新 MathOps 分类的接口部分。

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

#import "Fraction.h"

@interface Fraction (MathOps)

-(Fraction *) add: (Fraction *) f; // 加法函数
```

```

-(Fraction *) mul: (Fraction *) f; // 乘法函数
-(Fraction *) sub: (Fraction *) f; // 减法函数
-(Fraction *) div: (Fraction *) f; // 除法函数

@end

//*****

// #import "Fraction.h" 这里既是分类接口部分的定义，也是对现有接口部分的扩展，所以必须包括原始接口部分

// @interface Fraction (MathOps) 告诉编译器正在为 Fraction 类定义新的分类，名称为 MathOps。

```

可以在一个实现文件中定义 Fraction.h 接口部分中的所有方法，以及 MathOps 分类中的所有方法。

也可以在单独的实现部分定义分类的方法。在这种情况下，这些方法的实现部分还必须找出方法所属的分类。与接口部分一样，通过将分类名称括在类名称之后的圆括号中来确定方法所属的分类，如下所示：

```

@implementation Fraction (MathOps)

 // code for category methods

...

@end

```

### 关于分类的一些注意事项

- A、 尽管分类可以访问原始类的实例变量，但是它不能添加自身的任何变量。如果需要添加变量，可以考虑创建子类。
- B、 分类可以重载该类中的另一个方法，但是通常认为这种做法不可取。因为，重载之后，再不能访问原来的方法。
- C、 可以拥有很多分类。
- D、 和一般接口部分不同的是，不必实现分类中的所有方法。这对于程序扩展很有用，可以在该分类中声明所有方法，然后在一段时间之后才实现它。
- E、 通过使用分类添加新方法来扩展类不仅会影响这个类，同时也会影响它的所有子类。

## 9.2 协议

协议的声明类似于类接口的声明，有一点不同的是，协议没有父类，并且不能定义成员变量。下面的例子演示了只有一个方法的协议的声明：

```

@protocol MyProtocol

```

```
- (void)myProtocolMethod;
```

```
@end
```

协议是多个类共享的一个方法列表，协议中列出的方法没有相应的实现。如果一个类采用 `MyProtocol` 协议，则必须实现名为 `myProtocolMethod` 的方法。

通过在 `@interface` 行的一对尖括号 `<...>` 内列出协议名称，可以告知编译器你正在采用一个协议。这项协议的名称放在类名和它的父类名称之后，如下所示：

```
@interface AddressBook: NSObject <myProtocol>
```

这说明，`AddressBook` 是父类为 `NSObject` 的对象，并且它遵守 `myProtocolMethod` 协议。在 `AddressBook` 的实现部分，编译器期望找到定义的 `myProtocolMethod` 方法。

如果采用多项协议，只需把它们都列在尖括号中，用逗号分开：

```
@interface AddressBook: NSObject < myProtocol , yourProtocol >
```

以上代码告知编译器 `AddressBook` 类采用 `myProtocolMethod` 和 `yourProtocolMethod` 协议。这次，编译器将期望在 `AddressBook` 的实现部分看到为这些协议列出的所有方法的实现。

### 有关协议的注意事项：

A、如果一个类遵守某项协议，那么它的子类也遵守该协议。

B、协议不引用任何类，它是无类的（`classless`）。任何类都可以遵守某项协议。

C、通过在类型名称之后的尖括号中添加协议名称，可以借助编译器的帮助来检查变量的一致性，如下：

```
id <Drawing> currentObject;
```

这告知编译器 `currentObject` 将包含遵守 `Drawing` 协议的对象。如果向 `currentObject` 指派静态类型的对象，这个对象不遵守 `Drawing` 协议，编译器将给出 `warning`。

再次提到 `id` 类型，如果向 `currentObject` 指派一个 `id` 变量，不会产生这条消息，因为编译器不知道存储在 `id` 变量中的对象是否遵守 `Drawing` 协议。

D、如果这个变量保存的对象遵守多项协议，则可以列出多项协议，如下：

```
id <Drawing, Drawing 1> currentObject;
```

E、定义一项协议时，可以扩展现有协议的定义。以下协议

```
@protocol Drawing3D <Drawing>
```

说明 `Drawing3D` 协议也采用了 `Drawing` 协议。因此采用 `Drawing3D` 协议的类都必须实现此协议列出的方法，以及 `Drawing` 协议的方法。

F、分类也可以采用一项协议，如：

```
@interface Fraction (stuff) <NSCopying, NSCoding>
```

此处，Fraction 拥有一个分类 stuff，这个分类采用了 NSCopying 和 NSCoding 协议。

## 第十章 属性列表

说明：“Objective-C 编程人员可以使用与 C 绑定的所有工具，例如标准 C 库函数。可以使用 `malloc()` 和 `free()` 函数处理动态内存管理问题，或者使用 `open()`, `read()`, `write()`, `fopen()` 和 `fread()` 函数处理文件。”

属性列表类包括 `NSArray`、`NSDictionary`、`NSString`、`NSNumber`、`NSDate` 和 `NSData`。

### 10.1 NSDate

`NSDate` 是用于处理日期和时间的基础类。可以使用 `[NSDate date]` 获取当前的日期和时间，它是一个自动释放对象。以下代码：

```
NSDate *date = [NSDate date];
NSLog(@"today is %@", date);
```

将输出：

```
Today is 2009-11-10 19:23:02
```

还可以获取与当前时间相隔一定时差的日期。例如，24 小时之间的确切日期

```
NSDate *yesterday = [NSDate dateWithTimeIntervalSinceNow: -(24*60*60)];
NSLog(@"yesterday is %@", yesterday);
```

将输出

```
Yesterday is 2009-11-19 19:23:02
```

`+dateWithTimeIntervalSinceNow:` 接受一个 `NSTimeInterval` 参数，该参数是一个双精度值，表示以秒为单位的时间间隔。通过该参数可以指定时间偏移的方式：对于将来的时间，使用正的时间间隔；对于过去的时间，使用负的时间间隔。

### 10.2 NSData

`NSData` 类包装了大量字节。我们可以获得数据的长度和指向字节起始位置的指针。下面的 `NSData` 对象将保存一个普通的 C 字符串（一个字节序列），然后输出数据：

```
const char *string = "Hi there, this is a C string!";
NSData *data = [NSData dataWithBytes: string length: strlen(string) + 1];
NSLog(@"data is %@", data);
```

输出结果：

```
data is <4869..... 2100>
```

这是一个十六进制数据块实际上就是上面的字符串，0x48 代表字符 H，0x69 代表字符 i，等等。

strlen(string) + 1 中的 “+1” 用于包含 C 字符串所需的尾部的零字节（输出结果末尾的 00）。

可以使用 %s 格式的说明符输出字符串：

```
NSLog(@"%d bytes string is '%s'", [data length], [data bytes]);
```

//-length 方法给出字节数    -bytes 方法给出指向字符串起始位置的指针

输出结果如下所示：

```
30 bytes string is 'Hi there, this is a C string! '
```

NSData 对象是不可改变的，它们被创建后就不能改变。NSMutableData 支持在数据内容中添加和删除字节。

### 10.3 写入和读取属性列表

集合属性列表类（NSArray、NSDictionary）具有一个 -writeToFile:atomically: 方法，用于将属性列表写入文件。NSString 和 NSData 也具有 writeToFile:atomically: 方法，但是只能写出字符串和数据块。

因此，我们可以将字符串存入一个数组，然后保存该数组：

```
NSArray *phrase;
```

```
phrase = [NSArray arrayWithObjects: @"I", @"seem", @"to", @"be", @"a", @"verb", nil];
```

```
[phrase writeToFile: @"tmp/verbiage.txt" atomically: YES];
```

现在如果看一下文件/tmp/verbiage.txt，应该可以看到如下代码：

```
/**
 *
 *
 */
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<array>

 <string>I<string>

 < string>seem<string>

 < string>to< string>

 < string>be< string>
```

```

 < string >a< string >

 < string >verb< string >

 </array>

</plist>

//*****//

```

虽然繁琐，但是正是我们要保存的内容：一个字符串数组。这些属性列表文件可以为任意复杂的形式，可以包含字符串、数字和日期数组的字典数组。

现在已经将 verbiage.txt 文件保存在了磁盘上，可以使用+arrayWithContentsOfFile: 方法读取该文件。代码如下所示：

```

NSArray *phrase2 = [NSArray arrayWithContentsOfFile: @" /tmp/verbiage.txt"];

NSLog(@"'%@'", phrase2);

```

输出结果正好与前面保存的形式相匹配：

```

(
 I,
 seem,
 to,
 be,
 a,
 verb
)

```

writeToFile: 方法中的 atomically: 参数的值为 BOOL 类型，用于通知是否应该首先将文件保存在临时文件中，当文件成功保存后，再将该临时文件和原始文件交换。这是一种安全机制。

## 第十一章 复制对象

### 11.1 首先回顾继承部分

```
/**
 *
 */
// XYPoint 类 声明

#import<Foundation/Foundation.h>

@interface XYPoint: NSObject

{

 int x;

 int y;

}

@property int x, y; //存取器属性

-(void) setX: (int)xVal andY: (int) yVal;

@end

/**
 *
 */

// XYPoint 类 定义

#import "XYPoint.h"

@implementation XYPoint

@synthesize x, y;

-(void) setX: (int) xVal andY: (int) yVal

{

 x = xVal;

 y = yVal;

}

@end

/**
 *
 */

// Rectangle 类 声明

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



```

@class XYPoint;

@interface Rectangle: NSObject
{
 int width;

 int height;

 XYPoint *origin;
}

@property int width, height; // 存取器属性

-(XYPoint *) origin;

-(void) setOrigin: (XYPoint *) pt;

-(void) setWidth: (int) w andHeight: (int) h;

-(int) area;

-(int) perimeter;

@end

//*****//

//*****//

// Rectangle 类 定义

#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h
{
 width = w;

 height = h;
}

-(void) setOrigin: (XYPoint *) pt
{
 origin = pt;
}

- (int) area
{

```

```

 return width * height;
 }

 -(int) perimeter
 {
 return (width + height) * 2;
 }

 -(CGPoint *) origin
 {
 return origin;
 }

@end

//*****

//*****

#import "Rectangle.h"

#import "CGPoint.h"

int main(int argc, char *argv[])
{
 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

 Rectangle *myRect = [[Rectangle alloc] init];

 [myRect setWidth: 5 andHeight: 8];

 CGPoint *myPoint = [[CGPoint alloc] init];

 [myPoint setX: 100 andY: 200];

 myRect.origin = myPoint; //赋值

 NSLog(@"Origin at (%i, %i)", myRect.origin.x, myRect.origin.y);

 [myPoint setX: 50 andY: 50];

 NSLog(@"Origin at (%i, %i)", myRect.origin.x, myRect.origin.y);

 [myRect release];

 [myPoint release];

 [pool drain];

 return 0;
}

```

```
//*****//
```

```
myRect.origin = myPoint;
```

这样赋值的结果仅仅是将对象 myPoint 的地址复制到 myRect.origin 中。在赋值操作结束时，两个变量都指向内存中的同一个地址。

所以，将一个变量赋值给另一个对象仅仅是创建另一个对这个对象的引用。如果 dataArray 和 dataArray2 都是 NSMutableArray 对象，那么语句

```
dataArray2 = dataArray;
```

```
[dataArray2 removeObjectAtIndex: 0];
```

将从这两个变量引用的同一个数组中删除第一个元素。

```
//*****//
```

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

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

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

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

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

```
{
```

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

```
 NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
```

```
 @"one", @"two", @"three", @"four", nil];
```

```
 NSMutableArray *dataArray2;
```

```
 dataArray2 = dataArray;
```

```
 [dataArray2 removeObjectAtIndex: 0];
```

```
 NSLog(@"dataArray: ");
```

```
 for(NSString *elem in dataArray)
```

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

```
 NSLog(@"dataArray2: ");
```

```
 for(NSString *elem in dataArray2)
```

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

//以下输出结果是:

```
// dataArray:
// two
// three
// four

// dataArray2:
// two
// three
// four
```

//下面开始 Copy

```
dataArray2 = [dataArray mutableCopy];
[dataArray2 removeObjectAtIndex: 0];

NSLog(@"dataArray: ");

for(NSString *elem in dataArray)
 NSLog(@"%@", elem);

NSLog(@"dataArray2: ");

for(NSString *elem in dataArray2)
 NSLog(@"%@", elem);
```

//以下输出结果是:

```
// dataArray:
// two
// three
// four

//

// dataArray2:
// three
// four

[dataArray2 release];

[pool drain];

return 0;

}
```

```
/*******//
```

## 11.2 copy 和 mutableCopy

利用名为 copy 和 mutableCopy 的方法,可以创建对象的副本。结合 NSMutableArray 对象 dataArray 和 dataArray2, 语句

```
dataArray2 = [dataArray mutableCopy];
```

在内存中创建了一个新的 dataArray 副本,并复制了它的所有元素。随后,执行语句 [dataArray2 removeObjectAtIndex: 0];

删除了 dataArray2 中的第一个元素,但是不会删除 dataArray 中的。

### 注意:

A、产生一个对象的可变副本并不要求被复制的对象本身是可变的。也可以创建可变对象的不可变副本。

B、在产生数组的副本时,数组中每个元素的保持计数将通过复制操作自动增 1。所以,需要 [dataArray2 release]; 释放它的内存。

## 11.3 浅复制和深复制

浅复制代码分析:

```
/*******//
```

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

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

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

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

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

```
{
```

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

```
NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
```

```
 [NSMutableString stringWithstring: @"one"],
```

```
 [NSMutableString stringWithstring: @"two"],
```

```
 [NSMutableString stringWithstring: @"three"],
```

```
 nil
```

```
];
```

```
NSMutableArray *dataArray2;

NSMutableString *mStr;

NSLog(@"dataArray: ");

for(NSString *elem in dataArray)

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

//输出结果:

```
// dataArray:

// one

// two

// three

dataArray2 = [dataArray mutableCopy];

mStr = [dataArray objectAtIndex: 0]; //检索 dataArray 的第一个元素

[mStr appendString: @"ONE"]; //将字符串附加到这个元素

NSLog(@"dataArray: ");

for(NSString *elem in dataArray)

 NSLog(@"%@@", elem);

NSLog(@"dataArray2: ");

for(NSString *elem in dataArray2)

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

//以下输出结果是:

```
// dataArray:

// oneONE

// two

// three

// dataArray2:

// oneONE

// two

// three

[dataArray2 release];

[pool drain];
```

```
return 0;

}

//*****//
```

`dataArray` 的第一元素发生改变：从集合中获取元素时，得到了这个元素的一个新引用，但并不是一个新副本。所以，对 `dataArray` 调用 `objectAtIndex:` 方法时，返回的对象与 `dataArray` 中的第一个元素都指向内存中的同一个对象。随后，修改 `string` 对象的 `mStr` 的副作用就是同时改变了 `dataArray` 的第一个元素。

`dataArray2` 的第一元素发生改变：这与默认的浅复制方式有关。使用 `mutableCopy` 方法复制数组时，在内存中为新的数组对象分配了空间，并且将单个元素复制到新数组中。但是，这仅仅是将一个数组的元素复制到另一个数组。那么，这两个数组中的每个元素指向内存中的同一个字符串。这与将一个对象复制给另一个对象没有什么区别。

要为数组中每个元素创建完全不同的副本，需要执行所谓的深复制，也就是要创建数组中的每个对象内容的副本。例如，假设想要更改上面 `dataArray2` 的第一个元素，但是不更改 `dataArray` 的第一个元素，可以创建一个新字符串，并将它存储到 `dataArray2` 的第一个位置，如下所示：

```
mStr = [NSMutableString stringWithstring: [dataArray2 objectAtIndex: 0]];
```

然后，可以更改 `mStr`，并使用 `replaceObjectAtIndex: withObject:` 方法将它添加到数组中，如下所示：

```
[mStr appendString @"ONE"];
```

```
[dataArray2 replaceObjectAtIndex: 0 withObject: mStr];
```

这样，替换数组中的对象后，`mStr` 和 `dataArray2` 的第一个元素仍指向内存中的同一个对象。这意味着随后在程序中对 `mStr` 做的修改也将更改数组的第一个元素。

有关深复制的内容，将在下一章继续学习。

## 第十二章 归档

归档是指用某种格式来保存一个或多个对象，以便以后还原这些对象的过程。即包括将多个对象写入文件，以便以后读回该对象。

两种归档的方法：属性列表和带键值的编码

若对象是 NSString、NSDictionary、NSArray、NSData、NSNumber 对象时，可以使用 `writeToFile:atomically:` 方法将数据写到文件中，是以属性列表的形式写到文件中的。

参数 `atomically` 为 YES，表示先将数据写到临时备份文件中，一旦成功，再转移到文件中。

示例：

```
/**
 *
 */
#import<Foundation/NSObject.h>
#import<Foundation/NSString.h>
#import<Foundation/NSDictionary.h>
#import<Foundation/NSAutoreleasePool.h>
int main(int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSDictionary *glossary1 =
 [NSDictionary dictionaryWithObjectsAndKeys :
 @"A class defined so other classes can inherit",@"abstract class",
 @"To implement all the method defined in a protocol",@"adopt",
 @"Storing an object for later use",@"archiving",
 nil
];
 if ([glossary1 writeToFile:@"glossary" atomically:YES] == NO)
 NSLog(@"Save to file failed !");
 NSDictionary *glossary2 = [NSDictionary dictionaryWithContentOfFile:@"glossary"];
 for (NSString *key in glossary2)
 NSLog(@"%@: %@",key,[glossary2 objectForKey:key]);
 [pool drain];
 return 0;
}
```

glossary 文件里保存的数据格式是<key>...</key> <string>...</string>

要读回数据使用 `dataWithContentOfFile:` 方法

要读回字符串对象使用 `stringWithContentOfFile:` 方法

要读回根据字典创建的属性列表使用 `dictionaryWithContentOfFile:` 方法或者  
`arrayWithContentOfFile:` 方法

**注：**属性列表可以来自任何的源，可以来自文本编辑器或者 Property List Editor 程序来



创建的属性列表。

## NSKeyedArchiver 类

使用 NSKeyedArchiver 类创建带键的档案，在带键的档案中，每个归档的字段都有一个名称。归档某个对象的时候，会为他提供一个名称，即键。从归档中检索该对象的时候，是根据这个键来检索它的。这样，可以按照任意的顺序将对象写入归档并进行检索。另外，如果向类添加了新的实例变量或删除了实例变量，程序也可以进行处理。

NSKeyedArchiver 类中的 archiveRootObject: toFile: 方法将数据对象存储到磁盘上。

例如：

```
NSDictionary *glossary = [...];
[NSKeyedArchiver archiveRootObject:glossary toFile:@"file"];
```

通过 NSKeyedUnarchiver 类中的 unArchiveObjectWithFile:方法将创建的归档文件读入执行的程序中。

例如：

```
NSDictionary *glossary = [NSKeyedUnarchiver unArchiveObjectWithFile:@"file"]
//将指定的文件打开并读取文件的内容。该文件必须是前面归档操作的结果。可以为文件指定完整路径名或相对路径名。
```

## 编码方法和解码方法：

按照<NSCoding>协议，在类定义中添加编码方法 encodeWithCoder: 方法和解码方法 initWithCoder: 方法实现的。

对于基本 objective-C 类（NSString、NSArray、NSDictionary、NSSet、NSDate、NSNumber、NSData）使用 encodeObject: forKey: 编码方法和 decodeObject: forKey: 解码方法

| 在带键的档案中编码和解码基本数据类型    |                       |
|-----------------------|-----------------------|
| 编码方法                  | 解码方法                  |
| encodeBool: forKey:   | decodeBool: forKey:   |
| encodeInt: forKey:    | decodeInt: forKey:    |
| encodeInt32: forKey:  | decodeInt32: forKey:  |
| encodeInt64: forKey:  | decodeInt64: forKey:  |
| encodeFloat: forKey:  | decodeFloat: forKey:  |
| encodeDouble: forKey: | decodeDouble: forKey: |

若要确保继承的实例变量也被编码：

```
[super encodeWithCoder:encoder];
```

若要确保继承的实例变量也被解码：

```
[super initWithCoder:encoder];
```

encodeObject: forKey: 方法可以用于任何在其类中实现对 encodeWithCoder: 方法的对象。同样 decodeObject: forKey: 方法传递在编码时用的相同的键，就可以解码每个实例。

## 注：

还有一些基本数据类型，如（char、short、long、long long）在上表中没有列出，此时你必须确定数据对象的大小并使用相应的例程。例如：要归档 short int 的数据，首先将其保存在 int 中，然后使用 encodeInt:

forKey: 归档它，反向使用 decodeInt: forKey: 可以恢复它，最后将其赋值给 short int 的数据。  
示例:

```
/**
 *
 */
#import<Foundation/NSObject.h>
#import<Foundation/NSString.h>
#import<Foundation/NSKeyedArchiver.h>
@interface AddressCard: NSObject<NSCoding,NSCopying>
{
 NSString *name;
 NSString *email;
}
@property (copy, nonatomic) NSString *name,*email ;
-(void)setName:(NSString *)theName andEmail:(NSString *)theEmail;
...
@end

//下面是添加到其中的编码和解码的方法
- (void) encodeWithCoder: (NSCoder*) encoder
{
 [encoder encodeObject:name forKey:@"AddressCardName"];
 [encoder encodeObject:email forKey:@"AddressCardEmail"];
}

- (id) initWithCoder: (NSCoder*) decoder
{
 name = [[decoder decodeObjectForKey:@"AddressCardName"] retain];
 email = [[decoder decodeObjectForKey:@"AddressCardEmail"] retain];
}
/**
 *
 */
```

使用 NSData 创建自定义档案:

```
NSMutableData *dataArea = [NSMutableData data];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
 initWithWritingMutableData: dataArea];
```

//以指定要写入归档数据的存储空间，即 dataArea。此时就可以向 archiver 发送编码消息，以归档程序中的对象。所有编码消息在收到 finishEncoding 消息之前都被归档并存储在指定的数据空间中。

最后向 dataArea 发送 writeToFile: atomically: 消息。请求它把它的数 据写到指定的文件中（若文件名为 file）

```
[dataArea writeToFile: @"file" atomically: YES];
```

相反若要从档案文件中恢复数据和归档工作相反：首先，分配一个数据空间。然后，把档案中的数据读入该空间，接着创建一个 NSKeyedUnarchiver 对象，告知它从空间解码数据。必须调用解码方法来提取和解码归档的对象，做完以后，向 NSKeyedUnarchiver 对象发送一条 finishEncoding 消息。

使用归档文件复制对象（深复制）:

使用 Foundation 的归档能力来创建对象的深复制。在归档和解归档的过程中产生的字符串的新副本。

需要生成一个对象（或不支持 NSCopying 协议的对象）的深复制时，记住使用这项技术。

例如：

实现复制

```
NSMutableArray *dataArray = [...];
NSMutableArray *dataArray2;
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:dataArray];
dataArray2 = [NSKeyedUnarchiver unarchivedObjectWithData:data];
```

甚至可以避免中间赋值即，

```
NSMutableArray *dataArray2 = [NSKeyedUnarchiver unarchivedObjectWithData:
 [NSKeyedArchiver archivedDataWithRootObject:dataArray]];
```

示例：

```
//*****
import<Foundation/NSObject.h>
import<Foundation/NSString.h>
import<Foundation/NSArray.h>
import<Foundation/NSAutoreleasePool.h>
import<Foundation/NSKeyedArchiver.h>
int main(int argc ,char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSData *data ;
 NSMutableArray *dataArray = [NSMutableArray arrayWithObjects :
 [NSMutableArray stringWithString : @"one"] ,
 [NSMutableArray stringWithString : @"two"] ,
 [NSMutableArray stringWithString : @"three"] ,
 nil
];
 NSMutableArray *dataArray2 ;
 NSMutableString *mStr ;
 data = [NSKeyedArchiver archivedDataWithRootObject:dataArray];
 dataArray2 = [NSKeyedUnarchiver unarchivedObjectWithData:data];
 mStr = [dataArray2 objectAtIndex : 0];
 [mStr appendString :@"ONE"];
 NSLog(@"dataArray:");
 for(NSString *elem in dataArray)
 NSLog(@"%@@",elem);
 NSLog(@"\ndataArray2:");
 for(NSString *elem in dataArray2)
 NSLog(@"%@@",elem);
 [pool drain];
 return 0;
}
//*****
```

结果如下：

```
//-----
dataArray:
one
two
three

dataArray2:
oneONE
two
Three
//-----
```