Object-C 入门教程

分类： Sip&asterisk2009-05-04 16:34 16409人阅读 评论(2) 收藏 举报

大纲

开始吧

下载这篇教学

- 所有这篇初学者指南的原始码都可以由 objc.tar.gz 下载。这篇教学中的许多范例都是由 Steve Kochan 在 Programming in Objective-C. 一书中撰写。如果你想得到更多详细信息及范例，请直接参考该书。这个网站上登载的所有范例皆经过他的允许，所以请勿复制转载。

设定环境

- Linux/FreeBSD: 安装 GNUStep 为了编译 GNUstep 应用程序，必须先执行位于 /usr/GNUstep/System/Makefiles/GNUstep.sh 的 GNUstep.sh 这个档案。这个路径取决于你的系统环境，有些是在 /usr, some /usr/lib，有些是/usr/local。如果你的 shell 是以 csh/tcsh 为基础的 shell,则应该改用 GNUStep.csh。建议把这个指令放在 .bashrc 或 .cshrc 中。
- Mac OS X: 安装 XCode
- Windows NT 5.X: 安装 cygwin 或 mingw，然后安装 GNUStep

前言

- 这篇教学假设你已经有一些基本的 C 语言知识,包括 C 数据型别、什么是函式、什么是回传值、关于指针的知识以及基本的 C 语言内存管理。如果您没有这些背景知识，我非常建议你读一读 K&R 的书：The C Programming Language（译注：台湾出版书名为 C 程序语言第二版）这是 C 语言的设计者所写的书。
- Objective-C，是 C 的衍生语言，继承了所有 C 语言的特性。是有一些例外，但是它们不是继承于 C 的语言特性本身。
- nil：在 C/C++ 你或许曾使用过 NULL，而在 Objective-C 中则是 nil。不同之处是你可以传递讯息给 nil（例如 [nil message];），这是完全合法的，然而你却不能对 NULL 如法炮制。
- BOOL：C 没有正式的布尔型别，而在 Objective-C 中也不是「真的」有。它是包含在 Foundation classes（基本类别库）中（即 import NSObject.h；nil 也是包括在这个头文件内）。BOOL 在 Objective-C 中有两种型态:YES 或 NO,而不是 TRUE 或 FALSE。
- #import vs #include：就如同你在 hello world 范例中看到的，我们使用了#import。#import 由 gcc 编译程序支援。我并不建议使用 #include，#import 基本上跟 .h 档头尾的 #ifndef #define #endif 相同。许多程序员们都同意，使用这些东西这是十分愚蠢的。无论如何，使用 #import 就对了。这样不但可以避免麻烦，而且万一有一天 gcc 把它拿掉了，将会有足够的 Objective-C 程序员可以坚持保留它或是将它放回来。偷偷告诉你，Apple 在它们官方的程序代码中也使用了 #import。所以万一有一天这种事真的发生,不难预料 Apple 将会提供一个支持 #import 的 gcc 分支版本。
- 在 Objective-C 中，method 及 message 这两个字是可以互换的。不过 messages 拥有特别的特性，一个 message 可以

动态的转送给另一个对象。在 Objective-C 中，呼叫对象上的一个讯息并不一定表示对象真的会实作这个讯息，而是对象知道如何以某种方式去实作它，或是转送给知道如何实作的对象。

编译 hello world

- hello.m
- #import <stdio.h>
-
- int main(int argc, const char *argv[]) {
- printf("hello world/n");
- return 0;
  }

-
  -
    - 输出
      hello world
-
  -
    - 在 Objective-C 中使用 #import 代替 #include
    - Objective-C 的预设扩展名是 .m

创建 classes

@interface

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Fraction.h
- #import <Foundation/NSObject.h>
-
- @interface Fraction: NSObject {
- int numerator;
- int denominator;
- }
-
- -(void) print;
- -(void) setNumerator: (int) d;
- -(void) setDenominator: (int) d;
- -(int) numerator;
- -(int) denominator;
- @end

-
  -
    - NSObject：NeXTStep Object 的缩写。因为它已经改名为

OpenStep，所以这在今天已经不是那么有意义了。

- 继承（inheritance）以 Class: Parent 表示，就像上面的 Fraction: NSObject。
- 夹在 @interface Class: Parent { .... } 中的称为 instance variables。
- 没有设定访问权限（protected, public, private）时，预设的访问权限为 protected。设定权限的方式将在稍后说明。
    - Instance methods 跟在成员变数（即 instance variables）后。格式为：scope (returnType) methodName: (parameter1Type) parameter1Name;scope 有 class 或 instance 两种。instance methods 以 - 开头，class level methods 以 + 开头。
- Interface 以一个 @end 作为结束。

@implementation

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Fraction.m

```
#import "Fraction.h"
#import <stdio.h>

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

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

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

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
  @end
```

-
    -

- Implementation 以 @implementation ClassName 开始，以 @end 结束。
- Implement 定义好的 methods 的方式,跟在 interface 中宣告时很近似。

把它们凑在一起

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- main.m
- #import <stdio.h>
- #import "Fraction.m"
-
- int main(int argc, const char *argv[]) {
- // create a new instance
- Fraction *frac = [[Fraction alloc] init];
-
- // set the values
- [frac setNumerator: 1];
- [frac setDenominator: 3];
-
- // print it
- printf("The fraction is: ");
- [frac print];
- printf("/n");
-
- // free memory
- [frac release];
-
- return 0;
  }

•
  ○
    - output
      The fraction is: 1/3
•
  ○
    - Fraction *frac = [[Fraction alloc] init];这行程序代码中有很多重要的东西。
    - 在 Objective-C 中呼叫 methods 的方法是 [object method]，就像 C++的 object->method()。
    - Objective-C 没有 value 型别。所以没有像 C++ 的 Fraction frac; frac.print(); 这类的东西。在 Objective-C 中完全使用指针来处理对象。
    - 这行程序代码实际上做了两件事： [Fraction alloc]

呼叫了 Fraction class 的 alloc method。这就像 malloc 内存，这个动作也做了一样的事情。
- [object init] 是一个建构子（constructor）呼叫，负责初始化对象中的所有变量。它呼叫了 [Fraction alloc] 传回的 instance 上的 init method。这个动作非常普遍，所以通常以一行程序完成：Object *var = [[Object alloc] init];

- [frac setNumerator: 1] 非常简单。它呼叫了 frac 上的 setNumerator method 并传入 1 为参数。
- 如同每个 C 的变体，Objective-C 也有一个用以释放内存的方式： release。它继承自 NSObject，这个 method 在之后会有详尽的解说。

详细说明...

多重参数

- 目前为止我还没展示如何传递多个参数。这个语法乍看之下不是很直觉，不过它却是来自一个十分受欢迎的 Smalltalk 版本。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Fraction.h
- ...
- -(void) setNumerator: (int) n andDenominator: (int) d;
  ...

-
  -
    - Fraction.m
    - ...
    - -(void) setNumerator: (int) n andDenominator: (int) d
      {
    - numerator = n;
    - denominator = d;
    - }
      ...

-
  -
    - main.m
    - #import <stdio.h>
    - #import "Fraction.h"
    -
    - int main(int argc, const char *argv[]) {
    - // create a new instance

```
    Fraction *frac = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // combined set
    [frac2 setNumerator: 1 andDenominator: 5];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "/n" );

    // print it
    printf( "Fraction 2 is: " );
    [frac2 print];
    printf( "/n" );

    // free memory
    [frac release];
    [frac2 release];

    return 0;
}
```

- output

```
The fraction is: 1/3
Fraction 2 is: 1/5
```

- 这个 method 实际上叫做 setNumerator:andDenominator:
- 加入其他参数的方法就跟加入第二个时一样，即 method:label1:label2:label3:，而呼叫的方法是 [obj method: param1 label1: param2 label2: param3 label3: param4]
- Labels 是非必要的，所以可以有一个像这样的 method:method:::，简单的省略 label 名称，但以 : 区隔参数。并不建议这样使用。

建构子（Constructors）

- 基于 "Programming in Objective-C," Copyright © 2004 by

Sams Publishing 一书中的范例，并经过允许而刊载。

- Fraction.h
- ...
- -(Fraction*) initWithNumerator: (int) n denominator: (int) d;
  ...

-
  - Fraction.m
  - ...
  - -(Fraction*) initWithNumerator: (int) n denominator: (int) d {
  - self = [super init];
  -
  - if (self) {
  - [self setNumerator: n andDenominator: d];
  - }
  -
  - return self;
  - }
    ...

-
  - main.m
  - #import <stdio.h>
  - #import "Fraction.h"
  -
  - int main(int argc, const char *argv[]) {
  - // create a new instance
  - Fraction *frac = [[Fraction alloc] init];
  - Fraction *frac2 = [[Fraction alloc] init];
  - Fraction *frac3 = [[Fraction alloc] initWithNumerator: 3 denominator: 10];
  -
  - // set the values
  - [frac setNumerator: 1];
  - [frac setDenominator: 3];
  -
  - // combined set
  - [frac2 setNumerator: 1 andDenominator: 5];
  -
  - // print it
  - printf("The fraction is: ");
  - [frac print];

```
    printf( "/n" );

    printf( "Fraction 2 is: " );
    [frac2 print];
    printf( "/n" );

    printf( "Fraction 3 is: " );
    [frac3 print];
    printf( "/n" );

    // free memory
    [frac release];
    [frac2 release];
    [frac3 release];

    return 0;
}
```

- output
- The fraction is: 1/3
- Fraction 2 is: 1/5
  Fraction 3 is: 3/10

- @interface 里的宣告就如同正常的函式。
  - @implementation 使用了一个新的关键词：super 如同 Java，Objective-C 只有一个 parent class（父类别）。
  - 使用 [super init] 来存取 Super constructor，这个动作需要适当的继承设计。
  - 你将这个动作回传的 instance 指派给另一新个关键词：self。Self 很像 C++ 与 Java 的 this 指标。
- if (self) 跟 (self != nil) 一样，是为了确定 super constructor 成功传回了一个新对象。nil 是 Objective-C 用来表达 C/C++ 中 NULL 的方式，可以引入 NSObject 来取得。
- 当你初始化变量以后，你用传回 self 的方式来传回自己的地址。
- 预设的建构子是 -(id) init。
- 技术上来说，Objective-C 中的建构子就是一个 "init" method，而不像 C++ 与 Java 有特殊的结构。

访问权限

- 预设的权限是 @protected

- Java 实作的方式是在 methods 与变量前面加上 public/private/protected 修饰语，而 Objective-C 的作法则更像 C++ 对于 instance variable（译注：C++术语一般称为 data members）的方式。
- Access.h

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

@interface Access: NSObject {
@public
    int publicVar;
@private
    int privateVar;
    int privateVar2;
@protected
    int protectedVar;
}
 @end
```

-

  - Access.m

```
#import "Access.h"

@implementation Access
 @end
```

-

  - main.m

```
#import "Access.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Access *a = [[Access alloc] init];

    // works
    a->publicVar = 5;
    printf( "public var: %i/n", a->publicVar );

    // doesn't compile
    //a->privateVar = 10;
    //printf( "private var: %i/n", a->privateVar );

    [a release];
    return 0;
 }
```

-
  -
    - output
      public var: 5
-
  -
    - 如同你所看到的，就像 C++ 中 private: [list of vars] public: [list of vars] 的格式，它只是改成了@private, @protected, 等等。

Class level access

- 当你想计算一个对象被 instance 几次时，通常有 class level variables 以及 class level functions 是件方便的事。
- ClassA.h
- #import <Foundation/NSObject.h>
-
- static int count;
-
- @interface ClassA: NSObject
- +(int) initCount;
- +(void) initialize;
  @end
-
  -
    - ClassA.m
    - #import "ClassA.h"
    -
    - @implementation ClassA
    - -(id) init {
    - self = [super init];
    - count++;
    - return self;
    - }
    -
    - +(int) initCount {
    - return count;
    - }
    -
    - +(void) initialize {
    - count = 0;
    - }
      @end
-
  -

```
main.m
#import "ClassA.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    ClassA *c1 = [[ClassA alloc] init];
    ClassA *c2 = [[ClassA alloc] init];

    // print count
    printf("ClassA count: %i/n", [ClassA initCount]);

    ClassA *c3 = [[ClassA alloc] init];

    // print count again
    printf("ClassA count: %i/n", [ClassA initCount]);

    [c1 release];
    [c2 release];
    [c3 release];

    return 0;
}
```

-
  -
```
output
ClassA count: 2
ClassA count: 3
```

-
  -
    - static int count = 0; 这是 class variable 宣告的方式。其实这种变量摆在这里并不理想，比较好的解法是像 Java 实作 static class variables 的方法。然而，它确实能用。
    - +(int) initCount; 这是回传 count 值的实际 method。请注意这细微的差别!这里在 type 前面不用减号 - 而改用加号 +。加号 + 表示这是一个 class level function。(译注：许多文件中，class level functions 被称为 class functions 或 class method)
    - 存取这个变数跟存取一般成员变数没有两样，就像 ClassA 中的 count++ 用法。
    - +(void) initialize method is 在 Objective-C 开始执行你的程序时被呼叫，而且它也被每个 class 呼叫。这是初始化像我们的 count 这类 class level variables 的好地方。

异常情况（Exceptions）

- 注意：异常处理只有 Mac OS X 10.3 以上才支持。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- CupWarningException.h
- #import <Foundation/NSException.h>
-
- @interface CupWarningException: NSException
  @end

-
  -
    - CupWarningException.m
    - #import "CupWarningException.h"
    -
    - @implementation CupWarningException
      @end
-
  -
    - CupOverflowException.h
    - #import <Foundation/NSException.h>
    -
    - @interface CupOverflowException: NSException
      @end
-
  -
    - CupOverflowException.m
    - #import "CupOverflowException.h"
    -
    - @implementation CupOverflowException
      @end
-
  -
    - Cup.h
    - #import <Foundation/NSObject.h>
    -
    - @interface Cup: NSObject {
    -     int level;
    - }
    -
    - -(int) level;
    - -(void) setLevel: (int) l;
    - -(void) fill;
    - -(void) empty;
    - -(void) print;
      @end

-
  -
    - Cup.m
    - #import "Cup.h"
    - #import "CupOverflowException.h"
    - #import "CupWarningException.h"
    - #import <Foundation/NSException.h>
    - #import <Foundation/NSString.h>
    -
    - @implementation Cup
    - -(id) init {
    -     self = [super init];
    -
    -     if (self) {
    -         [self setLevel: 0];
    -     }
    -
    -     return self;
    - }
    -
    - -(int) level {
    -     return level;
    - }
    -
    - -(void) setLevel: (int) l {
    -     level = l;
    -
    -     if (level > 100) {
    -         // throw overflow
    -         NSException *e = [CupOverflowException
    -             exceptionWithName:
      @"CupOverflowException"
    -             reason: @"The level is above 100"
    -             userInfo: nil];
    -         @throw e;
    -     } else if (level >= 50) {
    -         // throw warning
    -         NSException *e = [CupWarningException
    -             exceptionWithName:
      @"CupWarningException"
    -             reason: @"The level is above or at 50"
    -             userInfo: nil];
    -         @throw e;
    -     } else if (level < 0) {

```
        // throw exception
        NSException *e = [NSException
            exceptionWithName:
  @"CupUnderflowException"
            reason: @"The level is below 0"
            userInfo: nil];
        @throw e;
    }
}

-(void) fill {
    [self setLevel: level + 10];
}

-(void) empty {
    [self setLevel: level - 10];
}

-(void) print {
    printf( "Cup level is: %i/n", level );
}
  @end
```

-
  -
```
 main.m
#import "Cup.h"
#import "CupOverflowException.h"
#import "CupWarningException.h"
#import <Foundation/NSString.h>
#import <Foundation/NSException.h>
#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

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

    // this will work
    for ( i = 0; i < 4; i++ ) {
        [cup fill];
        [cup print];
    }
```

```
    // this will throw exceptions
    for ( i = 0; i < 7; i++ ) {
        @try {
            [cup fill];
        } @catch ( CupWarningException *e ) {
            printf( "%s: ", [[e name] cString] );
        } @catch ( CupOverflowException *e ) {
            printf( "%s: ", [[e name] cString] );
        } @finally {
            [cup print];
        }
    }

    // throw a generic exception
    @try {
        [cup setLevel: -1];
    } @catch ( NSException *e ) {
        printf( "%s: %s/n", [[e name] cString], [[e
  reason] cString] );
    }

    // free memory
    [cup release];
    [pool release];
  }
```

-
  -
    - output
    - Cup level is: 10
    - Cup level is: 20
    - Cup level is: 30
    - Cup level is: 40
    - CupWarningException: Cup level is: 50
    - CupWarningException: Cup level is: 60
    - CupWarningException: Cup level is: 70
    - CupWarningException: Cup level is: 80
    - CupWarningException: Cup level is: 90
    - CupWarningException: Cup level is: 100
    - CupOverflowException: Cup level is: 110
      CupUnderflowException: The level is below 0
-
  -
    - NSAutoreleasePool 是一个内存管理类别。现在先别管它是

干嘛的。

- Exceptions（异常情况）的丢出不需要扩充（extend）NSException 对象，你可简单的用 id 来代表它： @catch（id e）{ ... }
- 还有一个 finally 区块，它的行为就像 Java 的异常处理方式，finally 区块的内容保证会被呼叫。
- Cup.m 里的 @"CupOverflowException" 是一个 NSString 常数物件。在 Objective-C 中，@ 符号通常用来代表这是语言的衍生部分。C 语言形式的字符串（C string）就像 C/C++ 一样是 "String constant" 的形式，型别为 char *。

继承、多型（Inheritance, Polymorphism)以及其他面向对象功能

id 型别

- Objective-C 有种叫做 id 的型别，它的运作有时候像是 void*,不过它却严格规定只能用在对象。Objective-C 与 Java 跟 C++ 不一样，你在呼叫一个对象的 method 时，并不需要知道这个对象的型别。当然这个 method 一定要存在，这称为 Objective-C 的讯息传递。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Fraction.h
- #import <Foundation/NSObject.h>
-
- @interface Fraction: NSObject {
-     int numerator;
-     int denominator;
- }
-
- -(Fraction*) initWithNumerator: (int) n denominator: (int) d;
- -(void) print;
- -(void) setNumerator: (int) d;
- -(void) setDenominator: (int) d;
- -(void) setNumerator: (int) n andDenominator: (int) d;
- -(int) numerator;
- -(int) denominator;

```
@end
```

-
  -
    - Fraction.m
    - #import "Fraction.h"
    - #import <stdio.h>
    -
    - @implementation Fraction
    - -(Fraction*) initWithNumerator: (int) n denominator: (int) d {
    - self = [super init];
    -
    - if (self) {
    - [self setNumerator: n andDenominator: d];
    - }
    -
    - return self;
    - }
    -
    - -(void) print {
    - printf("%i / %i", numerator, denominator);
    - }
    -
    - -(void) setNumerator: (int) n {
    - numerator = n;
    - }
    -
    - -(void) setDenominator: (int) d {
    - denominator = d;
    - }
    -
    - -(void) setNumerator: (int) n andDenominator: (int) d {
    - numerator = n;
    - denominator = d;
    - }
    -
    - -(int) denominator {
    - return denominator;
    - }
    -
    - -(int) numerator {
    - return numerator;
    - }

```
@end
```

-
  -
    - Complex.h
    - #import <Foundation/NSObject.h>
    -
    - @interface Complex: NSObject {
    -     double real;
    -     double imaginary;
    - }
    -
    - -(Complex*) initWithReal: (double) r andImaginary: (double) i;
    - -(void) setReal: (double) r;
    - -(void) setImaginary: (double) i;
    - -(void) setReal: (double) r andImaginary: (double) i;
    - -(double) real;
    - -(double) imaginary;
    - -(void) print;
    -
      @end
-
  -
    - Complex.m
    - #import "Complex.h"
    - #import <stdio.h>
    -
    - @implementation Complex
    - -(Complex*) initWithReal: (double) r andImaginary: (double) i {
    -     self = [super init];
    -
    -     if (self) {
    -         [self setReal: r andImaginary: i];
    -     }
    -
    -     return self;
    - }
    -
    - -(void) setReal: (double) r {
    -     real = r;
    - }
    -
    - -(void) setImaginary: (double) i {

```
    imaginary = i;
}

-(void) setReal: (double) r andImaginary: (double) i {
    real = r;
    imaginary = i;
}

-(double) real {
    return real;
}

-(double) imaginary {
    return imaginary;
}

-(void) print {
    printf( "%_f + %_fi", real, imaginary );
}

@end
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "Complex.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] initWithNumerator: 1 denominator: 10];
    Complex *comp = [[Complex alloc] initWithReal: 10 andImaginary: 15];
    id number;

    // print fraction
    number = frac;
    printf( "The fraction is: " );
    [number print];
    printf( "/n" );

    // print complex
    number = comp;
```

```
    printf( "The complex number is: " );
    [number print];
    printf( "/n" );

    // free memory
    [frac release];
    [comp release];

    return 0;
}
```

-
  -
    - output
    - The fraction is: 1 / 10
      The complex number is: 10.000000 + 15.000000i
-
  -
    - 这种动态链接有显而易见的好处。你不需要知道你呼叫 method 的那个东西是什么型别，如果这个对象对这个讯息有反应，那就会唤起这个 method。这也不会牵涉到一堆繁琐的转型动作，比如在 Java 里呼叫一个整数对象的 .intValue() 就得先转型，然后才能呼叫这个 method。

继承（Inheritance）

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Rectangle.h
- #import <Foundation/NSObject.h>
-
- @interface Rectangle: NSObject {
- int width;
- int height;
- }
-
- -(Rectangle*) initWithWidth: (int) w height: (int) h;
- -(void) setWidth: (int) w;
- -(void) setHeight: (int) h;
- -(void) setWidth: (int) w height: (int) h;
- -(int) width;
- -(int) height;
- -(void) print;
  @end

-
  -

- Rectangle.m

```
#import "Rectangle.h"
#import <stdio.h>

@implementation Rectangle
-(Rectangle*) initWithWidth: (int) w height: (int) h {
    self = [super init];

    if ( self ) {
        [self setWidth: w height: h];
    }

    return self;
}

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

-(void) setHeight: (int) h {
    height = h;
}

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

-(int) width {
    return width;
}

-(int) height {
    return  height;
}

-(void) print {
    printf( "width = %i, height = %i", width, height );
}
@end
```

-
  - Square.h

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

```
@interface Square: Rectangle
-(Square*) initWithSize: (int) s;
-(void) setSize: (int) s;
-(int) size;
@end
```

- Square.m

```
#import "Square.h"

@implementation Square
-(Square*) initWithSize: (int) s {
    self = [super init];

    if ( self ) {
        [self setSize: s];
    }

    return self;
}

-(void) setSize: (int) s {
    width = s;
    height = s;
}

-(int) size {
    return width;
}

-(void) setWidth: (int) w {
    [self setSize: w];
}

-(void) setHeight: (int) h {
    [self setSize: h];
}
@end
```

- main.m

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

```
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Rectangle *rec = [[Rectangle alloc] initWithWidth: 10 height: 20];
    Square *sq = [[Square alloc] initWithSize: 15];

    // print em
    printf( "Rectangle: " );
    [rec print];
    printf( "/n" );

    printf( "Square: " );
    [sq print];
    printf( "/n" );

    // update square
    [sq setWidth: 20];
    printf( "Square after change: " );
    [sq print];
    printf( "/n" );

    // free memory
    [rec release];
    [sq release];

    return 0;
}
```

-
  - output
  - Rectangle: width = 10, height = 20
  - Square: width = 15, height = 15
    Square after change: width = 20, height = 20
-
  - 继承在 Objective-C 里比较像 Java。当你扩充你的 super class（所以只能有一个 parent），你想自定义这个 super class 的 method，只要简单的在你的 child class implementation 里放上新的实作内容即可。而不需要 C++ 里呆呆的 virtual table。
  - 这里还有一个值得玩味的地方，如果你企图像这样去呼叫 rectangle 的 constructor： Square *sq = [[Square alloc] initWithWidth: 10 height: 15]，会发生什么事？答案是会产

生一个编译程序错误。因为 rectangle constructor 回传的型别是 Rectangle*，而不是 Square*，所以这行不通。在某种情况下如果你真想这样用，使用 id 型别会是很好的选择。如果你想使用 parent 的 constructor，只要把 Rectangle* 回传型别改成 id 即可。

动态识别（Dynamic types）

- 这里有一些用于 Objective-C 动态识别的 methods（说明部分采中英并列,因为我觉得英文比较传神,中文怎么译都怪）:

-(BOOL) isKindOfClass: classObj	is object a descendent or member of classObj 此对象是否是 classObj 的子孙或一员
-(BOOL) isMemberOfClass: classObj	is object a member of classObj 此对象是否是 classObj 的一员
-(BOOL) respondsToSelector: selector	does the object have a method named specifiec by the selector 此对象是否有叫做 selector 的 method
+(BOOL) instancesRespondToSelector: selector	does an object created by this class have the ability to respond to the specified selector 此对象是否是由有能力响应指定 selector 的对象所产生
-(id) performSelector: selector	invoke the specified selector on the object 唤起此对象的指定 selector

-
  -
    - 所有继承自 NSObject 都有一个可回传一个 class 物件的 class method。这非常近似于 Java 的 getClass() method。这个 class 对象被使用于前述的 methods 中。
    - Selectors 在 Objective-C 用以表示讯息。下一个范例会秀出建立 selector 的语法。
    - 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
    - main.m
    - #import "Square.h"
    - #import "Rectangle.h"
    - #import <stdio.h>
    -
    - int main(int argc, const char *argv[]) {
    - Rectangle *rec = [[Rectangle alloc] initWithWidth: 10 height: 20];
    - Square *sq = [[Square alloc] initWithSize: 15];
    -
    - // isMemberOfClass

```
    // true
    if ( [sq isMemberOfClass: [Square class]] == YES ) {
        printf( "square is a member of square class/n" );
    }

    // false
    if ( [sq isMemberOfClass: [Rectangle class]] == YES ) {
        printf( "square is a member of rectangle class/n" );
    }

    // false
    if ( [sq isMemberOfClass: [NSObject class]] == YES ) {
        printf( "square is a member of object class/n" );
    }

    // isKindOfClass

    // true
    if ( [sq isKindOfClass: [Square class]] == YES ) {
        printf( "square is a kind of square class/n" );
    }

    // true
    if ( [sq isKindOfClass: [Rectangle class]] == YES ) {
        printf( "square is a kind of rectangle class/n" );
    }

    // true
    if ( [sq isKindOfClass: [NSObject class]] == YES ) {
        printf( "square is a kind of object class/n" );
    }

    // respondsToSelector

```

```
    // true
    if ( [sq respondsToSelector: @selector( setSize: )]
== YES ) {
        printf( "square responds to setSize:
method/n" );
    }

    // false
    if ( [sq respondsToSelector:
@selector( nonExistant )] == YES ) {
        printf( "square responds to nonExistant
method/n" );
    }

    // true
    if ( [Square respondsToSelector: @selector( alloc )]
== YES ) {
        printf( "square class responds to alloc
method/n" );
    }

    // instancesRespondToSelector

    // false
    if ( [Rectangle instancesRespondToSelector:
@selector( setSize: )] == YES ) {
        printf( "rectangle instance responds to
setSize: method/n" );
    }

    // true
    if ( [Square instancesRespondToSelector:
@selector( setSize: )] == YES ) {
        printf( "square instance responds to setSize:
method/n" );
    }

    // free memory
    [rec release];
    [sq release];

    return 0;
}
```

- output
- square is a member of square class
- square is a kind of square class
- square is a kind of rectangle class
- square is a kind of object class
- square responds to setSize: method
- square class responds to alloc method
  square instance responds to setSize: method

Categories

- 当你想要为某个 class 新增 methods，你通常会扩充（extend，即继承）它。然而这不一定是个完美解法，特别是你想要重写一个 class 的某个功能，但你却没有原始码时。Categories 允许你在现有的 class 加入新功能，但不需要扩充它。Ruby 语言也有类似的功能。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- FractionMath.h
- #import "Fraction.h"
-
- @interface Fraction (Math)
- -(Fraction*) add: (Fraction*) f;
- -(Fraction*) mul: (Fraction*) f;
- -(Fraction*) div: (Fraction*) f;
- -(Fraction*) sub: (Fraction*) f;
  @end

- FractionMath.m
- #import "FractionMath.h"
-
- @implementation Fraction (Math)
- -(Fraction*) add: (Fraction*) f {
- return [[Fraction alloc] initWithNumerator:
  numerator * [f denominator] +
-
  denominator * [f numerator]
- denominator:
  denominator * [f denominator]];
- }
-
- -(Fraction*) mul: (Fraction*) f {

```
    return [[Fraction alloc] initWithNumerator:
  numerator * [f numerator]
                              denominator:
  denominator * [f denominator]];

}

-(Fraction*) div: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator:
  numerator * [f denominator]
                              denominator:
  denominator * [f numerator]];
}

-(Fraction*) sub: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator:
  numerator * [f denominator] -

  denominator * [f numerator]
                              denominator:
  denominator * [f denominator]];
}
  @end
```

-
  -
```
    main.m
#import <stdio.h>
#import "Fraction.h"
#import "FractionMath.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac1 = [[Fraction alloc]
  initWithNumerator: 1 denominator: 3];
    Fraction *frac2 = [[Fraction alloc]
  initWithNumerator: 2 denominator: 5];
    Fraction *frac3 = [frac1 mul: frac2];

    // print it
    [frac1 print];
    printf( " * " );
    [frac2 print];
    printf( " = " );
    [frac3 print];
```

```
    printf( "/n" );

    // free memory
    [frac1 release];
    [frac2 release];
    [frac3 release];

    return 0;
}
```

output
1/3 * 2/5 = 2/15

重点是 @implementation 跟 @interface 这两行：@interface Fraction (Math)以及 @implementation Fraction (Math).

（同一个 class）只能有一个同名的 category，其他的 categories 得加上不同的、独一无二的名字。

Categories 在建立 private methods 时十分有用。因为 Objective-C 并没有像 Java 这种 private/protected/public methods 的概念，所以必须要使用 categories 来达成这种功能。作法是把 private method 从你的 class header (.h) 档案移到 implementation (.m) 档案。以下是此种作法一个简短的范例。

MyClass.h

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

@interface MyClass: NSObject
-(void) publicMethod;
@end
```

MyClass.m

```
#import "MyClass.h"
#import <stdio.h>

@implementation MyClass
-(void) publicMethod {
    printf( "public method/n" );
}
@end
```

```
// private methods
@interface MyClass (Private)
-(void) privateMethod;
@end

@implementation MyClass (Private)
-(void) privateMethod {
    printf( "private method/n" );
}
@end
```

```
main.m
#import "MyClass.h"

int main( int argc, const char *argv[] ) {
    MyClass *obj = [[MyClass alloc] init];

    // this compiles
    [obj publicMethod];

    // this throws errors when compiling
    //[obj privateMethod];

    // free memory
    [obj release];

    return 0;
}
```

```
output
public method
```

Posing

Posing 有点像 categories，但是不太一样。它允许你扩充一个 class，并且全面性地的扮演（pose）这个 super class。例如：你有一个扩充 NSArray 的 NSArrayChild 物件。如果你让 NSArrayChild 扮演 NSArray，则在你的程序代码中所有的 NSArray 都会自动被替代为 NSArrayChild。

基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。

FractionB.h

- #import "Fraction.h"
-
- @interface FractionB: Fraction
- -(void) print;
- @end

-
  -
    - FractionB.m
    - #import "FractionB.h"
    - #import <stdio.h>
    -
    - @implementation FractionB
    - -(void) print {
    - printf("(%i/%i)", numerator, denominator);
    - }
      @end
-
  -
    - main.m
    - #import <stdio.h>
    - #import "Fraction.h"
    - #import "FractionB.h"
    -
    - int main(int argc, const char *argv[]) {
    - Fraction *frac = [[Fraction alloc]
      initWithNumerator: 3 denominator: 10];
    -
    - // print it
    - printf("The fraction is: ");
    - [frac print];
    - printf("/n");
    -
    - // make FractionB pose as Fraction
    - [FractionB poseAsClass: [Fraction class]];
    -
    - Fraction *frac2 = [[Fraction alloc]
      initWithNumerator: 3 denominator: 10];
    -
    - // print it
    - printf("The fraction is: ");
    - [frac2 print];
    - printf("/n");
    -
    - // free memory

```
    [frac release];
    [frac2 release];

    return 0;
}
```

-
  - output
  - The fraction is: 3/10
    The fraction is: (3/10)
-
  - 这个程序的输出中，第一个 fraction 会输出 3/10，而第二个会输出 (3/10)。这是 FractionB 中实作的方式。
  - poseAsClass 这个 method 是 NSObject 的一部份,它允许 subclass 扮演 superclass。

Protocols

- Objective-C 里的 Protocol 与 Java 的 interface 或是 C++ 的 purely virtual class 相同。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- Printing.h
- @protocol Printing
- -(void) print;
  @end
-
  - Fraction.h
  - #import <Foundation/NSObject.h>
  - #import "Printing.h"
  -
  - @interface Fraction: NSObject <Printing, NSCopying> {
  - int numerator;
  - int denominator;
  - }
  -
  - -(Fraction*) initWithNumerator: (int) n denominator: (int) d;
  - -(void) setNumerator: (int) d;
  - -(void) setDenominator: (int) d;
  - -(void) setNumerator: (int) n andDenominator: (int) d;
  - -(int) numerator;
  - -(int) denominator;

```
@end
```

-
  -
    - Fraction.m
    - #import "Fraction.h"
    - #import <stdio.h>
    -
    - @implementation Fraction
    - -(Fraction*) initWithNumerator: (int) n denominator: (int) d {
    -     self = [super init];
    -
    -     if (self) {
    -         [self setNumerator: n andDenominator: d];
    -     }
    -
    -     return self;
    - }
    -
    - -(void) print {
    -     printf("%i/%i", numerator, denominator);
    - }
    -
    - -(void) setNumerator: (int) n {
    -     numerator = n;
    - }
    -
    - -(void) setDenominator: (int) d {
    -     denominator = d;
    - }
    -
    - -(void) setNumerator: (int) n andDenominator: (int) d {
    -     numerator = n;
    -     denominator = d;
    - }
    -
    - -(int) denominator {
    -     return denominator;
    - }
    -
    - -(int) numerator {
    -     return numerator;
    - }

```
-(Fraction*) copyWithZone: (NSZone*) zone {
    return [[Fraction allocWithZone: zone]
 initWithNumerator: numerator
 denominator: denominator];
}
 @end
```

-
  -
```
 Complex.h
#import <Foundation/NSObject.h>
#import "Printing.h"

@interface Complex: NSObject <Printing> {
    double real;
    double imaginary;
}

-(Complex*) initWithReal: (double) r andImaginary:
 (double) i;
-(void) setReal: (double) r;
-(void) setImaginary: (double) i;
-(void) setReal: (double) r andImaginary: (double) i;
-(double) real;
-(double) imaginary;
 @end
```

-
  -
```
 Complex.m
#import "Complex.h"
#import <stdio.h>

@implementation Complex
-(Complex*) initWithReal: (double) r andImaginary:
 (double) i {
    self = [super init];

    if ( self ) {
        [self setReal: r andImaginary: i];
    }

    return self;
}
```

```
-(void) setReal: (double) r {
    real = r;
}

-(void) setImaginary: (double) i {
    imaginary = i;
}

-(void) setReal: (double) r andImaginary: (double) i {
    real = r;
    imaginary = i;
}

-(double) real {
    return real;
}

-(double) imaginary {
    return imaginary;
}

-(void) print {
    printf( "%_f + %_fi", real, imaginary );
}
@end
```

-
  - main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "Complex.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc]
initWithNumerator: 3 denominator: 10];
    Complex *comp = [[Complex alloc] initWithReal: 5
andImaginary: 15];
    id <Printing> printable;
    id <NSCopying, Printing> copyPrintable;

    // print it
    printable = frac;
```

```
    printf( "The fraction is: " );
    [printable print];
    printf( "/n" );

    // print complex
    printable = comp;
    printf( "The complex number is: " );
    [printable print];
    printf( "/n" );

    // this compiles because Fraction comforms to both
Printing and NSCopyable
    copyPrintable = frac;

    // this doesn't compile because Complex only
conforms to Printing
    //copyPrintable = comp;

    // test conformance

    // true
    if ( [frac conformsToProtocol:
@protocol( NSCopying )] == YES ) {
        printf( "Fraction conforms to NSCopying/n" );
    }

    // false
    if ( [comp conformsToProtocol:
@protocol( NSCopying )] == YES ) {
        printf( "Complex conforms to NSCopying/n" );
    }

    // free memory
    [frac release];
    [comp release];

    return 0;
}
```

- output
- The fraction is: 3/10
- The complex number is: 5.000000 + 15.000000i
  Fraction conforms to NSCopying

-
  -
    - protocol 的宣告十分简单，基本上就是 @protocol ProtocolName (methods you must implement) @end。
    - 要遵从（conform）某个 protocol，将要遵从的 protocols 放在 <> 里面，并以逗点分隔。如：@interface SomeClass <Protocol1, Protocol2, Protocol3>
    - protocol 要求实作的 methods 不需要放在 header 档里面的 methods 列表中。如你所见，Complex.h 档案里没有 -(void) print 的宣告，却还是要实作它，因为它（Complex class）遵从了这个 protocol。
    - Objective-C 的接口系统有一个独一无二的观念是如何指定一个型别。比起 C++或 Java 的指定方式，如：Printing *someVar = (Printing *) frac; 你可以使用 id 型别加上 protocol：id <Printing> var = frac;。这让你可以动态地指定一个要求多个 protocol 的型别，却从头到尾只用了一个变数。如：<Printing, NSCopying> var = frac;
    - 就像使用@selector 来测试对象的继承关系，你可以使用 @protocol 来测试对象是否遵从接口。如果对象遵从这个接口，[object conformsToProtocol: @protocol(SomeProtocol)] 会回传一个 YES 型态的 BOOL 对象。同样地，对 class 而言也能如法炮制 [SomeClass conformsToProtocol: @protocol(SomeProtocol)]。

内存管理

- 到目前为止我都刻意避开 Objective-C 的内存管理议题。你可以呼叫对象上的 dealloc，但是若对象里包含其他对象的指针的话，要怎么办呢？要释放那些对象所占据的内存也是一个必须关注的问题。当你使用 Foundation framework 建立 classes 时，它如何管理内存？这些稍后我们都会解释。注意：之前所有的范例都有正确的内存管理，以免你混淆。

Retain and Release（保留与释放）

- Retain 以及 release 是两个继承自 NSObject 的对象都会有的 methods。每个对象都有一个内部计数器，可以用来追踪对象的 reference 个数。如果对象有 3个 reference 时，不需要 dealloc 自己。但是如果计数器值到达 0 时，对象就得 dealloc 自己。[object retain] 会将计数器值加 1（值从 1 开始），[object release] 则将计数器值减 1。如果呼叫 [object release] 导致计数器到达 0，就会自动 dealloc。
- Fraction.m
- ...
- -(void) dealloc {

```
    printf( "Deallocing fraction/n" );
    [super dealloc];
}
...
```

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。

```
main.m
#import "Fraction.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Fraction *frac1 = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // print current counts
    printf( "Fraction 1 retain count: %i/n", [frac1
  retainCount] );
    printf( "Fraction 2 retain count: %i/n", [frac2
  retainCount] );

    // increment them
    [frac1 retain]; // 2
    [frac1 retain]; // 3
    [frac2 retain]; // 2

    // print current counts
    printf( "Fraction 1 retain count: %i/n", [frac1
  retainCount] );
    printf( "Fraction 2 retain count: %i/n", [frac2
  retainCount] );

    // decrement
    [frac1 release]; // 2
    [frac2 release]; // 1

    // print current counts
    printf( "Fraction 1 retain count: %i/n", [frac1
  retainCount] );
    printf( "Fraction 2 retain count: %i/n", [frac2
  retainCount] );

    // release them until they dealloc themselves
```

```
    [frac1 release]; // 1
    [frac1 release]; // 0
    [frac2 release]; // 0
}
```

-
  -
    - output
    - Fraction 1 retain count: 1
    - Fraction 2 retain count: 1
    - Fraction 1 retain count: 3
    - Fraction 2 retain count: 2
    - Fraction 1 retain count: 2
    - Fraction 2 retain count: 1
    - Deallocing fraction
      Deallocing fraction
-
  -
    - Retain call 增加计数器值，而 release call 减少它。你可以呼叫 [obj retainCount] 来取得计数器的 int 值。 当当 retainCount 到达 0，两个对象都会 dealloc 自己，所以可以看到印出了两个 "Deallocing fraction"。

Dealloc

- 当你的对象包含其他对象时，就得在 dealloc 自己时释放它们。Objective-C 的一个优点是你可以传递讯息给 nil，所以不需要经过一堆防错测试来释放一个对象。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- AddressCard.h
- #import <Foundation/NSObject.h>
- #import <Foundation/NSString.h>
-
- @interface AddressCard: NSObject {
-     NSString *first;
-     NSString *last;
-     NSString *email;
- }
-
- -(AddressCard*) initWithFirst: (NSString*) f
-                 last: (NSString*) l
-                 email: (NSString*) e;
- -(NSString*) first;
- -(NSString*) last;
- -(NSString*) email;

```
-(void) setFirst: (NSString*) f;
-(void) setLast: (NSString*) l;
-(void) setEmail: (NSString*) e;
-(void) setFirst: (NSString*) f
        last: (NSString*) l
        email: (NSString*) e;
-(void) setFirst: (NSString*) f last: (NSString*) l;
-(void) print;
@end
```

-
  -
```
AddressCard.m
#import "AddressCard.h"
#import <stdio.h>

@implementation AddressCard
-(AddressCard*) initWithFirst: (NSString*) f
                last: (NSString*) l
                email: (NSString*) e {
    self = [super init];

    if ( self ) {
        [self setFirst: f last: l email: e];
    }

    return self;
}

-(NSString*) first {
    return first;
}

-(NSString*) last {
    return last;
}

-(NSString*) email {
    return email;
}

-(void) setFirst: (NSString*) f {
    [f retain];
    [first release];
    first = f;
```

```
}

-(void) setLast: (NSString*) l {
    [l retain];
    [last release];
    last = l;
}

-(void) setEmail: (NSString*) e {
    [e retain];
    [email release];
    email = e;
}

-(void) setFirst: (NSString*) f
        last: (NSString*) l
        email: (NSString*) e {
    [self setFirst: f];
    [self setLast: l];
    [self setEmail: e];
}

-(void) setFirst: (NSString*) f last: (NSString*) l {
    [self setFirst: f];
    [self setLast: l];
}

-(void) print {
    printf( "%s %s <%s>", [first cString],
                                [last cString],
                                [email cString] );
}

-(void) dealloc {
    [first release];
    [last release];
    [email release];

    [super dealloc];
}
  @end
```

-
  - main.m

```
#import "AddressCard.h"
#import <Foundation/NSString.h>
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    NSString *first =[[NSString alloc] initWithCString:
  "Tom"];
    NSString *last = [[NSString alloc] initWithCString:
  "Jones"];
    NSString *email = [[NSString alloc]
  initWithCString: "tom@jones.com"];
    AddressCard *tom = [[AddressCard alloc]
  initWithFirst: first
                                          last:
  last
                                          email:
  email];

    // we're done with the strings, so we must dealloc
  them
    [first release];
    [last release];
    [email release];

    // print to show the retain count
    printf( "Retain count: %i/n", [[tom first]
  retainCount] );
    [tom print];
    printf( "/n" );

    // free memory
    [tom release];

    return 0;
  }
```

-
  -
    - output
    - Retain count: 1
      Tom Jones <tom@jones.com>
-
  -
    - 如 AddressCard.m，这个范例不仅展示如何撰写一个 dealloc method，也展示了如何 dealloc 成员变量。

- 每个 set method 里的三个动作的顺序非常重要。假设你把自己当参数传给一个自己的 method（有点怪，不过确实可能发生）。若你先 release，「然后」才 retain，你会把自己给解构（destruct，相对于建构）！这就是为什么应该要 1) retain 2) release 3) 设值 的原因。
- 通常我们不会用 C 形式字符串来初始化一个变量，因为它不支持 unicode。下一个 NSAutoreleasePool 的例子会用展示正确使用并初始化字符串的方式。
- 这只是处理成员变量内存管理的一种方式，另一种方式是在你的 set methods 里面建立一份拷贝。

Autorelease Pool

- 当你想用 NSString 或其他 Foundation framework classes 来做更多程序设计工作时，你需要一个更有弹性的系统，也就是使用 Autorelease pools。
- 当开发 Mac Cocoa 应用程序时，autorelease pool 会自动地帮你设定好。
- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- main.m
- #import <Foundation/NSString.h>
- #import <Foundation/NSAutoreleasePool.h>
- #import <stdio.h>
-
- int main(int argc, const char *argv[]) {
- NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
- NSString *str1 = @"constant string";
- NSString *str2 = [NSString stringWithString: @"string managed by the pool"];
- NSString *str3 = [[NSString alloc] initWithString: @"self managed string"];
-
- // print the strings
- printf("%s retain count: %x/n", [str1 cString], [str1 retainCount]);
- printf("%s retain count: %x/n", [str2 cString], [str2 retainCount]);
- printf("%s retain count: %x/n", [str3 cString], [str3 retainCount]);
-
- // free memory
- [str3 release];
-

- // free pool
- [pool release];
- return 0;
  }

-
  -
    - output
    - constant string retain count: ffffffff
    - string managed by the pool retain count: 1
      self managed string retain count: 1

-
  -
    - 如果你执行这个程序，你会发现几件事：第一件事，str1 的 retainCount 为 ffffffff。
    - 另一件事，虽然我只有 release str3，整个程序却还是处于完美的内存管理下，原因是第一个常数字符串已经自动被加到 autorelease pool 里了。还有一件事，字符串是由 stringWithString 产生的。这个 method 会产生一个 NSString class 型别的字符串，并自动加进 autorelease pool。
    - 千万记得，要有良好的内存管理，像 [NSString stringWithString: @"String"]这种 method 使用了 autorelease pool，而 alloc method 如 [[NSString alloc] initWithString: @"String"] 则没有使用 auto release pool。
    - 在 Objective-C 有两种管理内存的方法， 1) retain and release or 2) retain and release/autorelease。
    - 对于每个 retain，一定要对应一个 release 「或」一个 autorelease。
    - 下一个范例会展示我说的这点。
    - 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
    - Fraction.h
    - ...
    - +(Fraction*) fractionWithNumerator: (int) n
      denominator: (int) d;
    - ...

-
  -
    - Fraction.m
    - ...
    - +(Fraction*) fractionWithNumerator: (int) n
      denominator: (int) d {
    - Fraction *ret = [[Fraction alloc]

```
    initWithNumerator: n denominator: d];
    [ret autorelease];

    return ret;
}
    ...
```

-
  -
    - main.m
    - #import <Foundation/NSAutoreleasePool.h>
    - #import "Fraction.h"
    - #import <stdio.h>
    -
    - int main(int argc, const char *argv[]) {
    - NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    - Fraction *frac1 = [Fraction fractionWithNumerator: 2 denominator: 5];
    - Fraction *frac2 = [Fraction fractionWithNumerator: 1 denominator: 3];
    -
    - // print frac 1
    - printf("Fraction 1: ");
    - [frac1 print];
    - printf("/n");
    -
    - // print frac 2
    - printf("Fraction 2: ");
    - [frac2 print];
    - printf("/n");
    -
    - // this causes a segmentation fault
    - //[frac1 release];
    -
    - // release the pool and all objects in it
    - [pool release];
    - return 0;
      }
-
  -
    - output
    - Fraction 1: 2/5
      Fraction 2: 1/3
-

- 在这个例子里，此 method 是一个 class level method。在对象建立后，在它上面呼叫 了 autorelease。在 main method 里面，我从未在此对象上呼叫 release。
- 这样行得通的原因是：对任何 retain 而言，一定要呼叫一个 release 或 autorelease。对象的 retainCount 从 1 起跳 ，然后我在上面呼叫 1 次 autorelease，表示 1 - 1 = 0。当 autorelease pool 被释放时，它会计算所有对象上的 autorelease 呼叫次数，并且呼叫相同次数的 [obj release]。
- 如同批注所说，不把那一行批注掉会造成分段错误(segment fault)。因为对象上已经呼叫过 autorelease，若再呼叫 release，在释放 autorelease pool 时会试图呼叫一个 nil 对象上的 dealloc，但这是不允许的。最后的算式会变为：1 (creation) - 1 (release) - 1 (autorelease) = -1
- 管理大量暂时对象时，autorelease pool 可以被动态地产生。你需要做的只是建立一个 pool，执行一堆会建立大量动态对象的程序代码，然后释放这个 pool。你可能会感到好奇，这表示可能同时有超过一个 autorelease pool 存在。

Foundation framework classes

- Foundation framework 地位如同 C++ 的 Standard Template Library。不过 Objective-C 是真正的动态识别语言(dynamic types)，所以不需要像 C++ 那样肥得可怕的样版 (templates)。这个 framework 包含了对象组、网络、线程，还有更多好东西。

NSArray

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- main.m
- #import <Foundation/NSArray.h>
- #import <Foundation/NSString.h>
- #import <Foundation/NSAutoreleasePool.h>
- #import <Foundation/NSEnumerator.h>
- #import <stdio.h>
-
- void print(NSArray *array) {
-     NSEnumerator *enumerator = [array objectEnumerator];
-     id obj;
-
-     while (obj = [enumerator nextObject]) {
-         printf("%s/n", [[obj description] cString]);
-     }

```
}

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSArray *arr = [[NSArray alloc] initWithObjects:
                    @"Me", @"Myself", @"I", nil];
    NSMutableArray *mutable = [[NSMutableArray alloc] init];

    // enumerate over items
    printf( "----static array/n" );
    print( arr );

    // add stuff
    [mutable addObject: @"One"];
    [mutable addObject: @"Two"];
    [mutable addObjectsFromArray: arr];
    [mutable addObject: @"Three"];

    // print em
    printf( "----mutable array/n" );
    print( mutable );

    // sort then print
    printf( "----sorted mutable array/n" );
    [mutable sortUsingSelector: @selector( caseInsensitiveCompare: )];
    print( mutable );

    // free memory
    [arr release];
    [mutable release];
    [pool release];

    return 0;
}
```

-
  -
    ```
    output
    ----static array
    Me
    Myself
    I
    ```

- ----mutable array
- One
- Two
- Me
- Myself
- I
- Three
- ----sorted mutable array
- I
- Me
- Myself
- One
- Three

  Two

-
  -
    - 数组有两种（通常是 Foundation classes 中最数据导向的部分），NSArray 跟 NSMutableArray，顾名思义，mutable（善变的）表示可以被改变，而 NSArray 则不行。这表示你可以制造一个 NSArray 但却不能改变它的长度。
    - 你可以用 Obj, Obj, Obj, ..., nil 为参数呼叫建构子来初始化一个数组，其中 nil 表示结尾符号。
    - 排序（sorting）展示如何用 selector 来排序一个对象，这个 selector 告诉数组用 NSString 的忽略大小写顺序来排序。如果你的对象有好几个排序方法，你可以使用这个 selector 来选择你想用的方法。
    - 在 print method 里，我使用了 description method。它就像 Java 的 toString，会回传对象的 NSString 表示法。
    - NSEnumerator 很像 Java 的列举系统。while (obj = [array objectEnumerator]) 行得通的理由是 objectEnumerator 会回传最后一个对象的 nil。在 C 里 nil 通常代表 0，也就是 false。改用 ((obj = [array objectEnumerator]) != nil) 也许更好。

NSDictionary

- 基于 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一书中的范例，并经过允许而刊载。
- main.m
- #import <Foundation/NSString.h>
- #import <Foundation/NSAutoreleasePool.h>
- #import <Foundation/NSDictionary.h>
- #import <Foundation/NSEnumerator.h>
- #import <Foundation/Foundation.h<
- #import <stdio.h>

```
▪
▪ void print( NSDictionary *map ) {
▪     NSEnumerator *enumerator = [map keyEnumerator];
▪     id key;
▪
▪     while ( key = [enumerator nextObject] ) {
▪         printf( "%s => %s/n",
▪                 [[key description] cString],
▪                 [[[map objectForKey: key] description]
   cString] );
▪     }
▪ }
▪
▪ int main( int argc, const char *argv[] ) {
▪     NSAutoreleasePool *pool = [[NSAutoreleasePool
   alloc] init];
▪     NSDictionary *dictionary = [[NSDictionary alloc]
   initWithObjectsAndKeys:
▪         @"one", [NSNumber numberWithInt: 1],
▪         @"two", [NSNumber numberWithInt: 2],
▪         @"three", [NSNumber numberWithInt: 3],
▪         nil];
▪     NSMutableDictionary *mutable =
   [[NSMutableDictionary alloc] init];
▪
▪     // print dictionary
▪     printf( "----static dictionary/n" );
▪     print( dictionary );
▪
▪     // add objects
▪     [mutable setObject: @"Tom" forKey:
   @"tom@jones.com"];
▪     [mutable setObject: @"Bob" forKey:
   @"bob@dole.com" ];
▪
▪     // print mutable dictionary
▪     printf( "----mutable dictionary/n" );
▪     print( mutable );
▪
▪     // free memory
▪     [dictionary release];
▪     [mutable release];
▪     [pool release];
▪
```

```
    return 0;
}
```

```
output
----static dictionary
1 => one
2 => two
3 => three
----mutable dictionary
bob@dole.com => Bob
tom@jones.com => Tom
```

优点与缺点

优点

- Cateogies
- Posing
- 动态识别
- 指标计算
- 弹性讯息传递
- 不是一个过度复杂的 C 衍生语言
- 可透过 Objective-C++ 与 C++ 结合

缺点

- 不支持命名空间
- 不支持运算符多载（虽然这常常被视为一个优点，不过**正确地**使用运算符多载可以降低程序代码复杂度）
- 语言里仍然有些讨厌的东西，不过不比 C++ 多。

更多信息

- Object-Oriented Programming and the Objective-C Language
- GNUstep mini tutorials
- Programming in Objective-C
- Learning Cocoa with Objective-C
- Cocoa Programming for Mac OS X