Object-C 的语法与 Cocoa 框架

李海峰 QQ:61673110 邮箱:andrew830314@163.com

Object-C 是苹果 Mac OS X、iOS 平台的开发语言，Object-C 基于 C 语言的，增加面向对象的相关特性。你可以认为 Object-C 就是另一个版本的 C++，也就是它采用了与 C++不同的语法，但也实现了面向对象。

NextStep 是一个使用 Object-C 语言编写的功能强大的工具包，里面有大量的类库、结构体等，被苹果收购之后，更名为 Cocoa，但是苹果并未更改 NextStep 中的类库名称，因此你会看到大量的以 NS 为前缀的类名、结构体、枚举等。在 Object-C 中使用前缀可以有效的防止名称冲突。

Cocoa 框架由 Foundation Kit、App Kit 两部分组成，前者是基础工具库，是你必须首先要学会的，后者主要是 UI 库、高级对象等，我们这里只介绍 Foundation Kit。

本文档使用 Windows 上的 GNUStep 作为 Object-C 的编译器，不支持 Object-C 2.0 的相关新特性，但基本完全支持 Cocoa 的 Foundation Kit、App Kit 工具库。

1. GNUStep 的安装：

首先前往网址 http://www.gnustep.org/experience/Windows.html，下载文件：

Package	Required?	Stable	Unstable	Notes
GNUstep MSYS System	*Required*	0.25.1	0.27.0	MSYS/MinGW System
GNUstep Core	*Required*	0.25.0	0.27.0	GNUstep Core
GNUstep Devel	*Optional*	1.1.1	1.3.0	Developer Tools
Cairo Backend	*Optional*	0.22.1	None	Don't Use. Developer Only

然后按照下面的顺序安装这四个文件到同一个目录（例如：C:\GNUstep）：

(1.)gnustep-msys-system-xxx.exe

(2.)gnustep-core-xxx.exe

(3.)gnustep-devel-xxx.exe

(4.)gnustep-cairo-xxx.exe

安装完成后，进入开始---程序---GNUStep---Shell，你会看到一个在 Windows 上打开的命令行窗口，你可以在其中使用 Linux 的 Shell 命令 cd、ls、rm 等进行操作。启动 Shell 之后，它会在 GNUStep 的目录中建一个/home/xxx/的文件夹，xxx 为你当前登陆 Windows 系统的用户名称，Shell 默认进入的就是这个目录，也就是 Linux 上的 cd ~。

你可以在 Shell 中使用 vi 命令创建 Object-C 的源文件，但是推荐的方式是使用 UltraEdit 等编辑器编辑 Object-C 的源文件，然后在 Shell 中编译、运行。

GNUStep 使用 GCC 编译器，编译 Object-C 的命令：

```
gcc -o hello.exe hello.m
-I/GNUstep/System/Library/Headers
-fconstant-string-class=NSConstantString
```

-L/GNUstep/System/Library/Libraries
-lobjc -lgnustep-base
(1.)红色部分为编译生成的可运行文件，蓝色部分为要编译的源文件，可以有多个，使用空格分隔。
(2.) 参数-I 表示头文件查找的路径，-L 表示库文件查找路径，-l 表示需要链接的库文件，-fconstant-string-class=NSConstantString 主要是指定常量字符串所使用的 class。

2. 类定义：

我们定义一个类，这个类完成的功能是使用两个 int 类型的数字组成一个分数。在 Object-C 中必须首先定义一个接口，该接口用于描述这个类的组成，包含成员变量、类变量、类方法、成员方法。接口文件的扩展名为 h，也就是定义为 C 语言中的头文件。

Fraction.h

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

static int t=0;
@interface Fraction: NSObject{
        int numerator;//分子
        @public int denominator;//分母
}
-(void) setNumerator: (int) numerator;//numerator 的 setter 方法
-(void) setDenominator: (int) denominator;//denominator 的 setter 方法
-(void) setNumerator: (int) numerator andDenominator: (int) denominator;
//一个同时设置两个成员变量的快捷方法
-(int) numerator;//numerator 的 getter 方法
-(int) denominator;//denominator 的 getter 方法
-(void) print;
+(void) t;
@end
```

一个 Object-C 中的接口就是 C 语言中的一个 header 文件，这个文件结构如下所示：

```
#import Header

static 类型 变量名;
@interface 接口名: 父类名称{
        访问修饰符 类型 变量名;
        ... ...
}
-(返回值类型) 方法名: (参数类型) 参数名 标签 1: (参数类型) 参数名 ... ...
+(返回值类型) 方法名: (参数类型) 参数名 标签 1: (参数类型) 参数名 ... ...
@end
```

我们来逐行看一下上面的内容：
(1.) 这里与 C 语言不同的是导入头文件使用的是 import，而不是 include。另外与 C 语言一样

的地方是如果你想从当前目录查找 Header 文件，找不到就到系统的头文件库中查找，请使用#import "Header 文件"，如果你只想从系统的头文件库中查找，请使用#import <Header 文件>。Foundation/Foundation.h 包含了 Foundation Kit 中的所有的头文件定义，GNUStep 的 Object-C 的 Foundation 头文件在
\GNUStep 安装目录\GNUstep\System\Library\Headers\Foundation 文件夹。
GNUStep 的 Object-C 的 AppKit 头文件在
\GNUStep 安装目录\GNUstep\System\Library\Headers\ AppKit 文件夹。

(2.) static 标识的类变量定义在接口的外面，类变量只能本类访问，除非提供类方法给外部访问这个类变量。

(3.) Object-C 中的@+指令表示 C 语言之外的 Object-C 的衍生语法，因此@interface 表示定义了一个接口，接口名称之后紧跟了一个冒号，冒号后是父类的名字，Object-C 中的顶级父类是 NSObject。

(4.) 接口定义之后紧接着一对{ }，其中定义了成员变量，所谓的成员变量就相当于 JAVA 中的实例变量，从属于类的对象。Object-C 中的成员变量使用@public、@protected、@private 作为访问修饰符，默认是@protected。这里你要知道的是 Object-C 中只有成员变量有访问修饰符，类变量、类方法、成员方法是没有访问修饰符的，所有的方法都是 public 的，所有的类变量都是私有的。

(5.) 以-开头的方法为成员方法，以+开头的方法为类方法，方法中的类型描述（返回值类型、参数类型）都必须使用( )包围。如果方法有多个参数，每个参数都有一个标签名（可以省略，但不建议这样做），每个标签名之后使用冒号与参数描述分隔。在有多个参数的方法中，实际的方法名称为 **方法名:标签名 1:标签名 2:**... ...，上面的拥有两个参数的方法的方法名为 **setNumerator:andDenominator:**。与 JAVA 不同的是 Object-C 中的类方法只能类调用，如果你使用对象调用会报错，而 JAVA 仅仅是在编译期给出警告。

(6.) 以@end 表示接口定义结束。这是因为与 JAVA 不同的是 JAVA 的类型定义使用{ }包围，而 Object-C 中的{ }只包围成员变量，因此必须有个结束标志，一般 JAVA 程序员经常会忘记写这个结束标记。

这里你要知道 Object-C 的@interface 与 JAVA 的 interface 并不是一回事儿，后面你会看到 Object-C 中的@protocol 与 JAVA 中的 interface 才是等同的。这里你只需要记住的是 Object-C 中的@interface 只是类的一个描述，因为@interface 通常在独立的 h 文件中，你可以把它类比成 C 语言中的函数原型，也就是在 Object-C 里应该叫做类的原型。通过这个原型，编译器可以知道具体实现类有哪些功能。

上面的接口中的方法很简单，主要就是成员变量的 setter、getter 方法，这与 JAVA 没有什么不同的。但是你会发现 getter 方法没有以 get 作为方法名称前缀，这是因为 get 开头的方法在 Object-C 中有着特殊的含义，这在后面将会看到。

下面我们编写实现类，Object-C 的类文件使用扩展名 m。

Fraction.m

```
#import "Fraction.h"

@implementation Fraction
-(void) setNumerator: (int) n{
        numerator=n;
```

```
}
-(void) setDenominator: (int) d{
        denominator=d;
}
-(void) setNumerator: (int) n andDenominator: (int) d{
        numerator=n;
        denominator=d;
}
-(int) numerator{
        return numerator;
}
-(int) denominator{
        return denominator;
}
-(void) print{
        printf("%d/%d\n",numerator,denominator);
}
-(void) m{
        printf("-m:The class variable t is %d\n",++t);
}
+(void) t{
        printf("+t:The class variable t is %d\n",++t);
}
@end
```

因为我们将 Fraction.m 与 Fraction.h 放在一个文件夹下面，所以#import 使用了” ”，这个类的任务就是实现接口中的方法，因此与接口的结构不同的地方就是，你不能在这里定义变量，@interface 换成了@implementation，其余就没有什么特别的了。你不必在这里实现@interface 中的全部方法，这不会导致错误。这里有个-(void) m 方法比较特别，我们并没有在@interface 中声明，那么这个方法可以调用吗？因为 Object-C 是动态语言，即便是@interface 中没有定义的方法，依然可以被调用。
另外，你需要注意的是 setter 方法与接口中不同的是参数名缩写成了 n、d，这是因为在方法中，本地变量（参数、方法中定义的变量）在名称冲突的情况下，会隐藏成员变量，因此导致 **numerator=numerator** 变成了无意义的操作。当然你可以使用后面提到的 self 关键字，写成 **self->numerator=numerator**，也就是 JAVA 中的常用的 this.x=x 的写法。

下面我们编写调用代码，因为 Object-C 基于 C 语言，所以程序的入口依然是 main 函数。这里注意#import 的是 h，不是 m。
main.m

```
#import "Fraction.h"

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

```
        [frac setNumerator: 3 andDenominator: 5];
        [frac print];
        printf("The denominator of Fraction is %d\n",frac->denominator);
        [Fraction t];//调用类方法
        [frac m];
        [frac release];
        return 0;
}
```

(1.) 第一行我们创建了 Fraction 的实例（对象），Object-C 中实例只能使用指针作为变量，而不能使用值，所以你看到了*frac，而不是 frac，这与 JAVA 是一致的，JAVA 中的指向实例的变量（JAVA 中叫做引用）也是指针，只不过 JAVA 中没有指针的概念，所以你没有看到*。至于等号右侧的创建实例的代码，你可以在下面看到，这里先不用理会。

(2.) 第二行代码调用同时设置两个变量的方法，我们看到 Object-C 的调用方法的语法格式为**[类或者实例的指针 方法名：参数 1 标签 1：参数 2... ...]**。这种调用格式被称为中缀语法，初次看起来有点儿怪，但实际这样更加有效。举个例子，你接手了一个离职的人程序，其中的 JAVA 程序调用了一个有五个甚至更多的参数的方法，但是你手里没有这个方法的 API，那么你很难猜得出来这五个参数到底都干什么用的，但是 Object-C 调用的时候，每个参数前面都必须有方法的标签名，这样你便能很容易的从标签名看出这个参数是什么意思。

(3.) 第四行在 C 的 printf()函数中使用了**对象->成员变量**的语法访问实例的变量，但一般我们不推荐这么做，而是使用 getter 方法。这里你不能访问 numerator 变量，因为它是@protected 的，只能本类、子类直接访问。

(4.) 第五行我们调用了类方法 t，你也可以换成这样的写法：

[[Fraction class] t];

或者

Class clazz=[Fraction class];

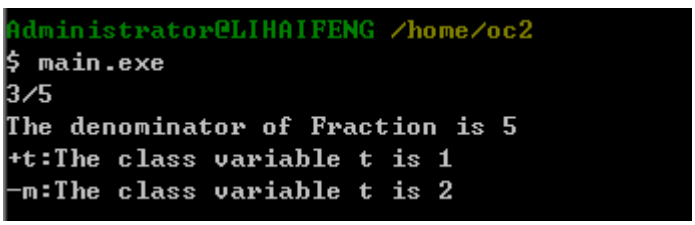
[clazz t];

class 方法是一个类方法，来自于 NSObject，相当于 JAVA 中的 getClass()方法，也就是获取这个类的 Class 对象，clazz 前面没有*。另外这种嵌套调用的方式，你也要习惯，这就和 JAVA 中的 **A.b().c()**没有什么区别。

(5.) 第六行我们调用了 m 方法，这个方法你会发现并没有在@interface 中声明，这里依然调用了，只是在编译的时候收到一个警告。这就是前面所有的 Object-C 是动态语言的原因。但是一般情况下，你都会给别人提供 h 文件，所以你在 m 文件中写的 h 文件中没有的方法，别人也是不会知道的，这个方法相当于变相的私有化了。

(6.) 第七行我们释放了 frac 实例在第一行 alloc 所申请的内存空间，Object-C 的内存管理后面会看到。另外，你会发现 Fraction.h 中没有定义 alloc、init、release 方法，但是我们上面调用了，很显然，这些方法来自于父类 NSObject。

编译、运行上面的程序，你会看到 Shell 窗口输出如下内容：



3. Object-C 中的布尔类型：

早期的 C 语言中是没有布尔类型的（C99 增加了布尔类型），Object-C 中增加 BOOL 类型来表示 YES、NO，注意不是 TRUE、FALSE。BOOL 使用了一个 8 位（一个字节）的整数进行表示，8 位全 0 就是 NO。

我们知道 C 语言中非 0 值即为逻辑真，因此常常会有 int i=5;while(i){… …}的写法。在 Object-C 中一定要注意慎用 C 语言中的这种数字与逻辑真假混合对待的做法去操作 BOOL 类型变量。例如：

```
BOOL bi=8960;
if(bi==YES){
    printf("YES");
}
```

这里会输出 YES 吗？不会的。为什么呢？8960 是非 0 值，它不是逻辑真吗？还记得上面说过 BOOL 是一个 8 位的整数吗？因为 8960 用二进制表示是大于 8 位的，也就是说高位无效，只保留 8960 的低八位，8960 的低八位恰好全都是 0，因此 bi 就是 NO 了。因此在 Object-C 中一定要注意这个问题，非零值未必是 BOOL 的 YES，但是 0 一定是 NO。

所以有 C 语言编程经验的，最好不要把 BOOL 与整数掺合在一起作为布尔类型的判断，可能 C 语言的开发者认为直接用数字作为布尔值进行判断在写法上更为简洁。

4. Object-C 中的 null：

Object-C 中的对象使用 nil 表示 null，但是它与 null 是有区别的，如果你向 null 发送消息（使用 null 调用方法）会得到运行时错误，这在 JAVA 中司空见惯的空指针异常就已经知道了。但是 nil 是可以回应消息（respond to message），因此你不必再为空指针而烦恼。

5. 与 C 混合编写：

我们看一段代码：

```
BOOL differentInt(int m , int n){
        if(m!=n)
                return YES;
        else
                return NO;
}

NSString *boolString(BOOL yn){
        if(yn==YES){
                return @"YES";
        }
        else{
                return @"No";
        }
}

int main(int argc,const char *argv[]){
    NSLog(boolString(differentInt(5,3)));
```

```
    return 0;
}
```

这里我们定义了函数 differentInt()用于比较两个整数是否相等，boolString()函数用于将 BOOL 类型转换为字符串。这两个函数的返回值都是 Object-C 中的类型，其中的 BOOL 不是对象类型，所以不使用指针，因此方法名称前面没有*，NSString 是 Object-C 中的字符串对象，相当于 JAVA 中的 String 类型，@”... ...”是一种 NSString 的字面值的表示方法，与 JAVA 中的”... ...”可以直接表示字符串，而不必非得显示的用 String 来引用字符串是一样的。这里注意与 Object-C 的类型中的方法定义不同的是，函数的返回值如果是指针，*写在 C 语言的函数名称前面，而不是返回值的前面。

在 main 函数中我们使用了 Object-C 的函数 **NSLog(@”格式化字符串”,变量 1,变量 2,... ...)**，这与 C 语言的 **printf(”格式化字符串”,变量 1,变量 2,... ...)**很相似，不过它会像 JAVA 中的 LOG4J 一样，在输出语句之前增加日期戳、运行的类名、自动追加换行符\n 等信息。

你可能又会问 Object-C 不都是对象吗？怎么还出来个 NSLog()的函数呢？函数不是 C 语言面向过程编程的东西吗？其实 Cocoa 中有很多的东西都不是对象，而是 C 语言的函数、结构体等。例如：

```
struct NSRange{
    NSUInteger location;
    NSUInteger length;
}
```

结构体 NSRang 表示一个范围，location 是范围的起始点，length 是范围的长度。

```
struct NSRect{
    NSPoint point;
    NSSize size;
}
```

结构体 NSRect 表示一个矩形，point 是左顶点的坐标，size 是长宽。

```
struct NSPoint{
    float x;
    float y;
}
struct NSSzie{
    float width;
    float height;
}
```

NSRange 常用来做字符串处理。NSRect 常用来做图形处理，譬如：动画中不断的重新绘制矩形等。你很容易知道这是个频繁操作的处理过程，也就是矩形要不断绘制、擦除。假如 NSRect 是一个 Object-C 类型，由于类实例化对象要在堆内存中动态分配存储空间，这是个很消耗资源的操作，而动画又是频率较高的操作，反复的创建、销毁对象，效率将会极其低下。所以 Cocoa 这里将 NSRect 定义为 C 语言的结构体就大大提高了运行效率。相比较 Android 平台，有人说做游戏纯靠 JAVA 是不行的，JAVA 最多就是画画 UI 和做些简单的应用，因为 JAVA 不具备和 C 语言混合编程的能力，事事都要借助对象，所以必然要引入 C 去编写 Android 的*.SO 的图形引擎等，以提高处理能力。

6. 对象的初始化：

JAVA 的对象创建只有一个过程，就是调用构造方法，但是 Object-C 分为两个步骤：分配内存、初始化，如上面的例子中的如下语句：

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

(1.) alloc 是从 NSObject 继承而来的类方法，用于给对象分配存储空间，所有的成员变量在此时对确定了自己的内存位置，并被赋初值，整数类型为 0，浮点数为 0.0，BOOL 为 NO，对象类型为 nil，alloc 方法返回对象的指针。

(2.) init 是从 NSObject 继承而来的成员方法，这个方法是你在对象创建过程可以参与的方法，因此你可以定制自己的 init 方法。Object-C 对 init 方法没有特殊的要求，就是一个普通方法而已，只不过习惯以 init 作为方法前缀。一般 init 方法都返回当前类型的指针或者 id 类型。id 类型在 Object-C 中是一个泛型对象，与前面所说的 Class 对象一样，不使用指针作为变量，id 可以表示任意类型的 Object-C 对象。

下面我们定制 Fraction 的初始化方法。

在 ***Fraction.h*** 中增加如下两个方法原型：

```
-(id) init;
-(Fraction*) initWithNumerator: (int) numerator andDenominator: (int) denominator;
```

Fraction.m 的实现：

```
-(id) init{
        self=[super init];
}

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

(1.) 第一个方法我们覆盖了 NSObject 中的 init 初始化方法，返回值 id 你也可以写成 Fraction*，这里只是为了让你知道 id 可以代表任意类型。方法的实现中我们首先调用了父类的 init 方法，使用了关键字 super，这与 JAVA 没什么不同的。另外，你还看到了关键字 self，它相当于 JAVA 中的 this，用于指向当前实例的指针。但是奇怪的是我把[super init]的返回值赋给了 self，这在 JAVA 中是没有的操作。因为 Object-C 是动态语言，所以 init 方法很可能会返回一个不是当前类型的返回值，也就是 Fraction 的 init 方法是可能返回另一个类型的（Object-C 中的 NSString 的实现代码 init 方法返回的就是 NSCFString）。因此为了保证程序可以正常工作，你需要把父类的 init 方法产生的返回值赋给 self，让 self 指向父类 init 方法返回的对象，然后才能开始其它操作。

(2.) 第二个方法首先调用了第一个方法，这就相当于 JAVA 中的策略模式，重载方法互相调用，这没有什么特别的。需要知道的是首先判断了一下 self 是否为 nil（**if(slef)**与**if(self!=nil)**是相同的），因为父类可能因为某些原因导致初始化异常。

你可以在 main 函数中使用新的 init 方法：

```
Fraction *frac=[[Fraction alloc] initWithNumerator :2 denominator :3];
```

7. Object-C 的 description 方法：

JAVA 中的对象都有从 Object 中继承而来的 String toString()方法，用于获取对象的字符串表示，Object-C 中的这个方法的方法签名为：

-(NSString*) description;

由于这是 NSObject 中的成员方法，因此我们就不必在 Fraction.h 文件中声明它了，直接在 Fraction.m 中实现如下所示：

```
-(NSString*) description{
        return @"I am a fraction!";
}
```

编写 main 函数访问：

```
int main(int argc,const char *argv[]){
        Fraction *frac=[[Fraction alloc] initWithNumerator: 2 andDenominator: 3];
        NSLog(@"%@",frac);
        return 0;
}
```

这里我们看到 NSLog 输出对象用的格式化字符串是%@，这样 description 方法就会被调用。如果我们不实现自己的 description，那么会调用 NSObject 的实现，输出对象的首地址。

8. Object-C 的异常处理：

我们的 Fraction 忽略了一个问题，那就是没有对 denominator 为 0 的情况作处理，也就是分母不能为 0。我们可以使用 Object-C 的异常机制对这个问题进行处理。

Object-C 的异常都继承自 NSException，当然 NSException 本身也继承自 NSObject。

DenominatorNotZeroException.h

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

@interface DenominatorNotZeroException: NSException
@end
```

DenominatorNotZeroException.m

```
#import "DenominatorNotZeroException.h"

@implementation DenominatorNotZeroException
@end
```

我们定义了分母不能为 0 的异常，Object-C 中的异常定义起来很简单，其实除了写了类名之外什么都没有写。

下面我们改写 Fraction.m 的如下两个 setter 方法：

```
-(void) setDenominator: (int) d{
    if(d==0){
```

```
            NSException *e=[DenominatorNotZeroException
                    exceptionWithName: @"DenominatorNotZeroException"
                    reason: @"The denominator is not 0!"
                    userInfo:nil];
            @throw e;
    }
    denominator=d;
}

-(void) setNumerator: (int) n andDenominator: (int) d{
    if(d==0){
            NSException *e=[DenominatorNotZeroException
                    exceptionWithName: @"DenominatorNotZeroException"
                    reason: @"The denominator is not 0!"
                    userInfo:nil];
            @throw e;
    }
    numerator=n;
    denominator=d;
}
```

这两个方法中我们判断如果 denominator 为 0，则抛出 DenominatorNotZeroException，DenominatorNotZeroException 的创建使用了父类 NSException 的类方法 exceptionWithName:reason:userInfo:，前两个参数表示异常的名字、原因，参数类型为 NSString，第三个参数为用户信息，暂不清楚做什么的，我们传递 nil 就可以了。异常定义完成后使用@throw 抛出。

下面我们编写捕获异常的代码：

```
int main(int argc,const char *argv[]){
        NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
        @try{
            Fraction *frac=[[Fraction alloc] initWithNumerator: 3 andDenominator: 0];
        }@catch (DenominatorNotZeroException *dne){
                printf("%s\n",[[dne reason] cString]);
        }@catch (NSException *e){
                printf("%s\n",[[e name] cString]);
        }@finally{
                printf("finally run!");
        }
        [pool release];
        return 0;
}
```

这里的捕获代码基本与 JAVA 的意义相同，只不过是 try、catch、finally 前加@。另外，由于 NSException 的成员方法 name、reason 返回的是 NSString，因此又接着调用了 NSString

的成员方法 cString，得到 C 的字符串类型 char[]，然后才能传递给 C 的函数 printf()。
这里你还看到了一个 NSAutoreleasePool，先不用理会它，因为不写它会报错，后面会做讲解。
但你会发现上面的程序无法在 GNUStep 中运行，提示不认识@throw，所以你只需要知道上面的写法就可以了，运行可以在 Mac 电脑上操作。

9. id 类型：

下面我们演示一下 id 类型是如何可以指向任意类型的实例的。我们定义一个复数类型 Complex。

Complex.h

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

@interface Complex: NSObject {
        double real;//复数的实部
        double imaginary;//复数的虚部
}
-(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"

@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 );//输出复数 z=a+bi
}
@end
```

main.m

```
#import "Fraction.h"
#import "Complex.h"

int main(int argc,const char *argv[]){
        Fraction *frac=[[Fraction alloc] initWithNumerator: 3 andDenominator: 5];
        Complex *comp=[[Complex alloc] initWithReal: 1.5 andImaginary: 3.5];

        id number=frac;
        [number print];

        number=comp;
        [comp print];

        [frac release];
        [comp release];
        return 0;
}
```

我们看到 id 类型 number 首先指向了 frac 实例，调用了它的 print()方法，然后指向了 comp 实例，调用了它的 print()方法。所以 id 你可以把它理解为随便，也就是它表示任意的东西。

10. 类的继承：

下面我们以矩形、正方形的对象展示一下 Object-C 中的继承结构。

矩形 MyRectangle.h

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

```
@interface MyRectangle: NSObject{
        int width;
        int height;
}
-(MyRectangle*) initWithWidth: (int) weight andHeight: (int) height;
-(void) setWidth: (int) width;
-(void) setHeight: (int) height;
-(int) width;
-(int) height;
-(void) area;//计算面积
@end
```

矩形 MyRectangle.m

```
#import "MyRectangle.h"

@implementation MyRectangle
-(MyRectangle*) initWithWidth: (int) w andHeight: (int) h{
        self=[super init];
        if(self){
                [self setWidth: w];
                [self setHeight: h];
        }
}
-(void) setWidth: (int) w{
        width=w;
}
-(void) setHeight: (int) h{
        height=h;
}
-(int) width{
        return width;
}
-(int) height{
        return height;
}
-(void) area{
        printf("%d\n",width*height);
}
@end
```

正方形 MySquare.h

```
#import "MyRectangle.h"
```

```
@interface MySquare: MyRectangle{
        int size;
}
-(MySquare*) initWithSize: (int) size;
-(void) setSize: (int) size;
-(int) size;
@end
```

这里正方形的父类是 MyRectangle。

正方形 MySquare.m

```
#import "MySquare.h"

@implementation MySquare
-(MySquare*) initWithSize: (int) s{
        self=[super init];
        if(self){
                [self setWidth: s];
                [self setHeight: s];
                //因为正方形的长宽相等，所以我们把正方形的边长赋给父类的长宽，以便复用计算面积的方法 area()。
        }
}
-(void) setSize: (int) s{
            size=s;
}
-(int) size{
        return size;
}
@end
```

main.m

```
#import "MySquare.h"

int main(int argc,const char *argv[]){
        MyRectangle *rec=[[MyRectangle alloc] initWithWidth: 2 andHeight: 5];
        [rec area];
        MySquare *squa=[[MySquare alloc] initWithSize: 6];
        [squa area];//使用父类的计算面积的方法

        [rec release];
        [squa release];
}
```

11. 动态判定与选择器：

第 9 节中我们使用了 id 这个泛型对象，假如别人传递给你一个 id，你如何动态判断这个在运行时传递过来的对象到底有没有 XXX()方法呢？因为 id 可以代表任意对象，你也不知道运行时传递的这个 id 里面到底有什么东西。其实 NSObject 中有一系列这样的方法用于动态判定，类似于 JAVA 的反射机制，但是 Object-C 的用起来更为简单。

-(BOOL) isMemberOfClass: (Class) clazz	用于判断对象是否是 clazz 类型的实例，但不包含子类的实例。
-(BOOL) isKindOfClass: (Class) clazz	用于判断对象是否是 clazz 类型的实例或者 clazz 的子类的实例。
-(BOOL) respondsToSelector: (SEL) selector	用于判断**类型**或者对象是否能够回应某个方法，这个方法使用选择器表示。
+(BOOL) instancesRespondToSelector: (SEL) selector	用于判断类型所产生的实例是否能够回应某个方法，这个方法使用选择器表示。
- (id) performSelector: (SEL) selector	用于动态调用**类型**或者对象上的一个方法。

上面的后三个方法中都涉及到了一个新类型选择器 SEL，它用于表示 Object-C 的一个方法，我们知道在 C 语言中有函数指针指向一个函数，也就是 Object-C 的选择器就相当于 C 语言中的函数指针，只不过选择器只能表示 Object-C 类型中定义的方法。选择器使用**@selector(方法名)**的形式获取，例如：**@selector(initWithWidth:andHeight:)** 、**@selector(alloc)**。同时，SEL 是在继 Class、id 之后第三个不需要在变量前使用*的类型。另外，注意上面的红色文字，你可以看到这两行的方法都是-开头的，也就是成员方法，但是为什么类也可以去掉用呢？其实是类的 Class 对象去掉用的，因为 Class 对象有着两个方法的回应能力。

我们以第 10 节的矩形、正方形的程序为基础，进行代码演示。

```
int main(int argc,const char *argv[]){
        MyRectangle *rec=[[MyRectangle alloc] initWithWidth: 2 andHeight: 5];
        [rec area];
        MySquare *squa=[[MySquare alloc] initWithSize: 6];
        [squa area];

        //-(BOOL) isMemberOfClass: (Class) clazz 用于判断对象是否是 clazz 的实例，但不包含子类的实例。
        if([squa isMemberOfClass: [MyRectangle class]]){
                printf("squa isMemberOfClass MyRectangle\n");
        }else{
                printf("squa not isMemberOfClass MyRectangle\n");
        }

        if([squa isMemberOfClass: [MySquare class]]){
                printf("squa isMemberOfClass MySquare\n");
```

```
}else{
    printf("squa not isMemberOfClass MySquare\n");
}
printf("-----------------------------------------\n");

//-(BOOL) isKindOfClass: (Class) clazz 用于判断对象是否是 clazz 的实例或者参数的子类的实例。
if([squa isKindOfClass: [MyRectangle class]]){
        printf("squa isKindOfClass MyRectangle\n");
}else{
        printf("squa not isKindOfClass MyRectangle\n");
}

if([squa isKindOfClass: [MySquare class]]){
        printf("squa isKindOfClass MySquare\n");
}else{
        printf("squa not isKindOfClass MySquare\n");
}
printf("-----------------------------------------\n");


//-(BOOL) respondsToSelector: (SEL) selector 用于判断对象或者类型是否有能力回应指定的方法。
if([squa respondsToSelector: @selector(initWithSize:)]){
        printf("squa respondsToSelector initWithSize:\n");
}else{
        printf("squa not respondsToSelector initWithSize:\n");
}

if([MySquare respondsToSelector: @selector(alloc)]){
        printf("MySquare respondsToSelector alloc\n");
}else{
        printf("MySquare not respondsToSelector alloc\n");
}

if([rec respondsToSelector: @selector(initWithWidth:andHeight:)]){
        printf("rec respondsToSelector initWithWidth:andHeight:\n");
}else{
        printf("rec not respondsToSelector initWithWidth:andHeight:\n");
}
printf("-----------------------------------------\n");

//+(BOOL) instancesRespondToSelector: (SEL) selector 用于判断类产生的实例是否是由有能力回应指定的方法。
```

```
        if([MySquare instancesRespondToSelector: @selector(initWithSize:)]){
                printf("MySquare instancesRespondToSelector initWithSize:\n");
        }else{
                printf("MySquare not instancesRespondToSelector initWithSize:\n");
        }

        if([MySquare instancesRespondToSelector: @selector(alloc)]){
                printf("MySquare instancesRespondToSelector alloc\n");
        }else{
                printf("MySquare not instancesRespondToSelector alloc\n");
        }
        printf("-----------------------------------------\n");

        //-(id) performSelector: (SEL) selector 用于动态调用类或者对象上的一个方法。
        id x_id=[rec performSelector: @selector(area)];
        [MyRectangle performSelector: @selector(alloc)];

        [rec release];
        [squa release];
}
```

12. 类别 Category：

如果你想扩充一个类的功能，但又不想使用继承，你可以选择类别。
下面我们写一个 Fraction 的类别，为 Fraction 类增加计算两个分数的加减乘除的方法。

FractionMath.h

```
#import "Fraction.h"

@interface Fraction (Math1)
-(Fraction*) mul: (Fraction*) f;//乘法，就是传入一个 Fraction 作为参数，与当前的 Fraction 进行计算
-(Fraction*) div: (Fraction*) f;//除法
@end

@interface Fraction (Math2)
-(Fraction*) add: (Fraction*) f;//加法
@end
```

其实类别就是在你想要扩展的@interface 的名字后面加个( )，里面写上类别的名字，这个名字必须是唯一的，也就是说一个@interface 只能有一个类别为 XXX 的 Category。

FractionMath.m

```
#import "FractionMath.h"

@implementation Fraction (Math1)
-(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]];
}
@end

@implementation Fraction (Math2)
-(Fraction*) add: (Fraction*) f{
        return [[Fraction alloc] initWithNumerator:
                numerator*[f denominator] + denominator*[f numerator]
                denominator: denominator*[f denominator]];
}
@end
```

类别的实现类也就是类型名后后面加个( )，里面写上类别的名字，其他的没有什么不同的。上面唯一可能会让你头晕的就是加减乘除的实现代码，实际上就是嵌套调用，写成了一个语句而已。拿 add 的实现来说，就是创建要返回的计算结果 Fraction 的实例，然后依据分数相加要先通分的规则,结果的分子 initWithNumber 就是第一个分数的分子*第二个分数的分母，再加上第二个分数的分子*第一个分数的分母，分母 denominator 参数就是两个分数的分母相乘。

main.m

```
#import "FractionMath.h"
int main( int argc, const char *argv[] ) {
        Fraction *frac1 = [[Fraction alloc] initWithNumerator: 1 denominator: 3];
        Fraction *frac2 = [[Fraction alloc] initWithNumerator: 2 denominator: 5];

        Fraction *frac3 = [frac1 mul: frac2];
        [frac1 print];
        printf( " * " );
        [frac2 print];
        printf( " = " );
        [frac3 print];
        printf( "\n" );

        Fraction *frac5 = [frac1 add: frac2];
```

```
        [frac1 print];
        printf( " + " );
        [frac2 print];
        printf( " = " );
        [frac5 print];
        printf( "\n" );

        [frac1 release];
        [frac2 release];
        [frac3 release];
        [frac5 release];
        return 0;
}
```

运行 GCC 之后，Shell 窗口输出如下内容：

```
lihaifeng@lihaifeng /home/oc/05
$ main.exe
1/3 * 2/5 = 2/15
1/3 + 2/5 = 11/15
```

我们看到没有使用继承语法，就为 Fraction 添加了数学运算的方法。另外，利用一个类可以有多个类别的特性（上面的 Math1、Math2），如果一个@interface 有上百个方法，那么你可以让编写的@implementation 什么都不实现，然后按照@interface 中的各个方法的功能的不同，在不同的类别中实现不同的方法，这样可以防止一个@implementation 实现全部的接口而显得臃肿。

那么类别与继承相比，有什么缺点吗？类别不可以声明新的成员变量，而且一旦你定义的方法与原始类中的方法名称相同，那么原始方法将被隐藏起来，因为不是继承结构，你不能在类别中的方法使用 super 激活原始类的同名方法。

类别还有一个功能，就是隐藏方法，我们在 Fraction.m 的最后增加如下的内容：

```
@interface Fraction (Math3)
-(Fraction*) sub: (Fraction*) f;//减法
@end

@implementation Fraction (Math3)
-(Fraction*) sub: (Fraction*) f{
        return [[Fraction alloc] initWithNumerator:
                numerator*[f denominator] - denominator*[f numerator]
                denominator: denominator*[f denominator]];
}
@end
```

在.m 文件中定义@interface？你没有看错，因为@interface 一旦定义在.m 文件中，它就不能以 Header 文件的形式被导入到其他的类了，也就是这样的@interface 中定义的方法相当于被你给隐藏了，只能这个.m 编译单元内看见。

此时如果你在 main 函数中调用 sub 做减法运算，会在编译器得到一个警告，但是实际上你还是能够访问到 sub 方法，因为 Object-C 是动态语言，只要运行期有这个方法，就能够调用。我们把@interface 定义在.m 文件中只能做到让别人不知道有这个方法，但是如果他有你的.m 文件的源码或者恰好猜到（这种撞大运的几率太低了）你有 sub 方法，还是可以调用的。

类别的应用比较广泛，譬如：第三方的 Object-C 库 RegexKitLite 就是对 NSString、NSMutableString 做的类别，为 Object-C 的字符类型增加了正则表达式的功能。

13. 协议@protocol：

前面说过@interface 相当于是 Object-C 的类的原型，与 JAVA 中的接口意义是不同的，Object-C 中的 @protocol 才是和 JAVA 中的接口等价的东西。例如：Object-C 的继承也是单继承，只允许有一个父类，但是@protocol 是允许多继承的（按照 Object-C 的说法叫做某类遵从了协议 A、协议 B，而不是继承），这些都与 JAVA 的接口一致。

Printing.h

```
@protocol Printing1
-(void) print1;
@end

@protocol Printing2
-(void) print2;
@end

@protocol Printing3 <Printing2>
-(void) print3;
@end
```

Printing3 <Printing2>的意思是 Printing3 遵从（继承）Printing2，<>是遵从@protocol 协议的语法。

Fraction.h

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

@interface Fraction: NSObject <Printing1,Printing3>{
        int numerator;
        int denominator;
}
-(Fraction*) initWithNumerator: (int) n denominator: (int) d;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
```

```
@end
```

Fraction.m

```
#import "Fraction.h"

@implementation Fraction
-(Fraction*) initWithNumerator: (int) n denominator: (int) d{
        self=[super init];
        if(self){
                [self setNumerator: n];
                [self setDenominator: d];
        }
}
-(void) setNumerator: (int) n{
        numerator=n;
}
-(void) setDenominator: (int) d{
        denominator=d;
}
-(int) numerator{
        return numerator;
}
-(int) denominator{
        return denominator;
}
-(void) print1{
        printf("1:%d/%d\n",numerator,denominator);
}
-(void) print2{
        printf("2:%d/%d\n",numerator,denominator);
}
-(void) print3{
        printf("3:%d/%d\n",numerator,denominator);
}
@end
```

main.m

```
#import "Fraction.h"

int main (int argc , const char *argv[]){
        Fraction *frac=[[Fraction alloc] initWithNumerator: 3 denominator: 5];
```

```
        <Printing1> p1=frac;//使用 protocol 类型，相当于 JAVA 中使用接口类型作为对象的引用 List list=ArrayList 的实例。
        //也可以写作  id <Printing1> p1=frac;
        [p1 print1];

        id<Printing1,Printing2,Printing3> p2=frac;
        //从这里可以看出 id 是一个泛型对象，在 id 后面使用<>作为泛型参数可以明确的告诉别人你想把 id 当作哪些种协议去使用，当然你可以不写泛型参数。
        [p2 print2];
        [p2 print3];

        //-(BOOL) conformsToProtocol: (Protocol*) prot 用于判断对象是否遵从某个 protocol。
        if([frac conformsToProtocol: @protocol(Printing1)]
            &&[frac conformsToProtocol: @protocol(Printing2)]
            &&[frac conformsToProtocol: @protocol(Printing3)]){
                printf("YES");
        }else{
                printf("NO");
        }

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

14. 内存管理：

JAVA 使用 GC 机制自动管理内存的，Object-C 支持手动管理内存，也支持 GC 机制，但是 GC 机制对于 iOS 设备无效，也就是仅对 Mac OS X 电脑有效。这是合理的，因为 iPhone、iPod、iPad 等的内存、CPU 肯定要比电脑低很多，你必须谨慎对待内存的使用，而不能肆无忌惮的等着 GC 帮你去收拾烂摊子。

Object-C 在使用 alloc、new、copy 的时候为对象分配内存，然后返回分配的内存的首地址存入指针变量，使用 dealloc 释放内存。new 是 alloc 和 init 的合写形式，也就是[[Fraction alloc] init]与[Fraction new]是相同的，copy 我们会在第 16 节讲解，

我们知道 Object-C 中的对象都是使用指针引用的，也就是在方法调用的时候传递的都是指针，那么一个对象的引用在一次调用过程中，可能被传递到了多处，也就是有多个地方在引用这个对象。

例如：

```
main(){
    A *a=[A new];
    B *b=[B new];
    C *c=[C new];
    [b m: a];
    [c m: a];
}
```

上面的代码就将 a 传递给了 b、c 两个实例，而 a 实例本身又是在 main 函数中定义的，因此一共有 main 函数、b、c 三个地方引用了 a 实例。
那么由上面的代码引出了这样的问题：你在 alloc 一个对象之后，什么时候 dealloc 它呢？回收早了可能导致有些还在引用的地方会报错，不回收自然会导致内存泄漏。
Object-C 的解决办法是采用一个引用计数器 retainCount 来表示还有多少个地方在引用这个对象。一个对象在被 alloc 之后就 retainCount 就是 1，之后每调用一次调用 retain 方法都会使 retainCount 加 1，调用 release 都会使 retainCount 减 1。当 Object-C 发现一个对象的引用计数器 retainCount 为 0 时，就会立即调用这个对象从 NSObject 继承而来的 dealloc 方法回收内存，这个调用动作是 Object-C 运行环境完成的，你需要关心的就是把 retainCount 在恰当的时候减为 0 就可以了。

```
int main(){
    Fraction *frac=[[Fraction alloc] initWithNumerator: 3 denominator: 5];
    printf("%d\n",[frac retainCount]);//1---alloc 分配内存并使引用计数器从 0 变为 1


    [frac retain];//2---引用计数器加 1
    printf("%d\n",[frac retainCount]);

    [frac retain];//3---引用计数器加 1
    printf("%d\n",[frac retainCount]);

    [frac release];//2---引用计数器减 1
    printf("%d\n",[frac retainCount]);

    [frac release];//1---引用计数器减 1
    printf("%d\n",[frac retainCount]);

    [frac release];//0---引用计数器减 1
    //此时 frac 的 dealloc 方法自动被调用，Object-C 回收 frac 对象被回收。你可以在 Fraction 中覆盖-(void) dealloc 方法中加个输出语句观察一下。此时你再去调用 frac 的方法都会导致程序崩溃，因为那块内存去被清理了。但是你可以像下面这样做。

    frac=nil;
    [frac print];//记得前面说过 nil 与 null 的区别了吗？因此此行为空操作，但是不会报错。
}
```

这段代码输出了 frac 的引用计数器的变化 1---2---3---2---1---0，当然这段代码的 retain 操作毫无意义，仅仅是演示 retain、release 之后引用计数器的变化情况。

我们定义一个住址类型的类：

Address.h

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

```
@interface Address: NSObject{
        NSString *city;
        NSString *street;
}
-(void) setCity: (NSString*) c;
-(void) setStreet: (NSString*) s;
-(void)setCity: (NSString*) c andStreet: (NSString*) s;
-(NSString*) city;
-(NSString*) street;
@end
```

与前面的示例不同的是 Address 的成员变量 city、street 都是 NSString 的对象类型，不是基本数据类型。

Address.m

```
#import "Address.h"

@implementation Address
-(void) setCity: (NSString*) c{
        [c retain];
        [city release];
        city=c;
}
-(void) setStreet: (NSString*) s{
        [s retain];
        [street release];
        street=s;
}
-(void)setCity: (NSString*) c andStreet: (NSString*) s{
        [self setCity: c];
        [self setStreet: s];
}
-(NSString*) city{
        return city;
}
-(NSString*) street{
        return street;
}
-(void) dealloc{
        [city release];
        [street release];
        [super dealloc];
```

```
}
@end
```

你可以先不理会这里的一堆 retain、release 操作，一会儿会做讲解。

main 函数

```
NSString *city=[[NSString alloc] initWithString: @"Beijing"];
// initWithString 是 NSString 的一个使用 NSString 字面值初始化的方法。与直接使用下面的 C 语言字符序列初始化相比，@” ”支持 Unicode 字符集。
NSString *street=[[NSString alloc] initWithCString: "Jinsongzhongjie"];
// initWithCString 是 NSString 的一个使用 C 语言的字符序列初始化的方法。

Address *address=[[Address alloc] init];
[address setCity: city andStreet: street];

[city release];
[street release];
[address release];
```

我们在 main 函数中创建了 city、street 两个 NSString 的实例，然后 setter 到 address 实例，由于 city、street 你使用了 alloc 分配内存，你势必就要 release 它们。首先要确定的是在 main 函数里 release 还是在 address 里 release 呢？因为你确实把他们两个传递给 Address 的实例 address 了。我们一般按照谁创建的就谁回收的原则（对象的拥有权），那么自然你要在 main 方法里 release，我们恰当的选择了在调用完 address 的 setter 方法之后 release，因为 city、street 在 main 函数中的任务（给 address 里的两个成员变量赋值）就此结束。
此时，新的问题来了，我们在 main 函数中 setter 完之后 release 对象 city、street，因此 city、street 的 retainCount 会由 1 变为 0，这就会导致你传递到 address 里的 city、street 指针指向的对象被清理掉，而使 address 出错。很显然，你需要在 address 的 setter 方法里 retain 一下，增加 city、street 的引用计数为 2，这样即便在 main 函数 release 之后，city、street 指向的内存空间也不会被 dealloc，因为 2-1=1，还有一个引用计数。因此就有了 Address 中的 setter 的写法，我们拿 setCity 方法为例：

```
-(void) setCity: (NSString*) c{
        [c retain];//---1
        [city release];//---2
        city=c;//---3
}
```

在 JAVA 这种使用 GC 机制的语言中，我们只需要写第三条语句。其实 Object-C 中你也可以只写第三条语句，我们管这种方式获得的 city 叫做弱引用，也就是 city 只是通过赋值操作，把自己指向了 c 指向的对象，但是对象本身的引用计数器没有任何变化，此时 city 就要承担[c release]之后所带来的风险，不过有些情况下，你可能确实需要这种弱引用。这里你需要注意的是有些文档把这种弱引用叫做 assign（指派，这很形象，把一个指针直接指派到另一个指针指向的对象，但不拥有这个对象）方式，强引用叫做 retain 方式。
当然，大多数情况下，我们使用的都是强引用，也就是使用第一行代码首先 retain 一下，使引用计数器加 1，再进行赋值操作，再进一步说就是先通过 retain 方法拿到对象的拥有权，

再安全的使用对象。
第二行代码又是为什么呢？其实这是为了防止 city 可能已经指向了一个对象，如果不先对 city 进行一次 release，而直接把 city 指向 c 指向的对象，那么 city 原来指向的对象可能会出现内存泄漏，因为 city 在改变指向的时候，没有将原来指向的对象的引用计数器减 1，违反了你 retain 对象之后，要在恰当的时刻 release 对象的要求。
第三行代码毋庸置疑的要在最后一行出现，但按照前面的阐述，貌似第一行、第二行的代码是没有先后顺序的，也就是可以先[city release]，再[c retain]，但真的是这样吗？有一种较为少见的情况，那就是把自己作为参数赋给自己（这听起来很怪），也就是说参数 c 和成员变 city 是一个指针变量，那么此时如果你先调用[city release]，就会导致对象的 retainCount 归 0，对象被 dealloc 了，那么[c retain]就会报错，因为对象没有了。
综上所述，上面所示的三行代码是比较好的组织方式。但其实你可能也会看到有些人用其他的方式书写 setter 方法，这也没什么好奇怪的，只要保证对象正常的使用、回收就是没有问题的代码。
最后我们再看一下 Address 中的 dealloc 方法，因为你在 setter 中持有了 city、street 指向的对象的拥有权，那么你势必要和 main 函数一样，负责在恰当的时刻 release 你的拥有权，由于你在 address 实例中可能一直要用到 city、street，因此 release 的最佳时刻就是 address 要被回收的时候。另外，我们知道对象被 dealloc 之后，原来指向它的指针们依然指向这块内存区域，Object-C 并不会把这些指着垃圾的指针们都指向 nil，因此你如果还调用这些指针们的方法，就会报错。因此有些人喜欢在 dealloc 里把对象 release 之后紧接着指向 nil，防止它会被意外调用而出现空指针异常。但我认为你这样可能屏蔽了错误，不是一个好办法。

Object-C 的手动管理内存还提供了一种半自动的方式，就是第八节用到的 NSAutoreleasePool 类型。只要实例是 alloc、new、copy 出来的，你就需要选择完全手动的管理内存，也就是你要负责 release。第八节的 DenominatorNotZeroException 异常我们是通过父类 NSException 的类方法 exceptionWithName:reason:userInfo 创建的，我们并没有使用到 alloc、new、copy 关键字，这种情况下，你只需要定义一个 NSAutoreleasePool 就可以了。为什么是这样呢？
我们先看下面的代码：

```
NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
Fraction *frac=[[Fraction alloc] initWithNumerator: 3 denominator: 5];
printf("frac 的引用次数  %d\n",[frac retainCount]);//1
[frac retain];
printf("frac 的引用次数  %d\n",[frac retainCount]);//2
[frac autorelease];
printf("frac 的引用次数  %d\n",[frac retainCount]);//2
[frac autorelease];
printf("frac 的引用次数  %d\n",[frac retainCount]);//2
[pool release];
```

这里我们对 frac 用的是 autorelease 方法，不是 release 方法，autorelease 方法并不对引用计数器减 1，而是将对象压入离它最近的 NSAutoreleasePool 的栈顶（所谓离得最近就表示多个 NSAutoreleasePool 是可以嵌套存在的），等到 NSAutoreleasePool 的 release 方法被调用后，NSAutoreleasePool 会将栈内存放的指针指向的对象的 release 方法。

其实 exceptionWithName:reason:userInfo 方法的内部实现就是把创建好的 NSException 的实例的指针返回之前，首先调用了指针的 autorelease 方法，把它放入了自动回收池。

其实在 iOS 开发中，苹果也是不建议你使用半自动的内存管理方式，但不像 GC 机制一样被禁用，原因是这种半自动的内存管理，容易在某些情况下导致内存溢出。我们看一下如下的代码：

```
NSAutoreleasePool *pool=[[NSAutoreleasePool alloc]init];
for(int i=0;i<1000000;i++){
    NSString *s=@"...";
    if(i%1000==0){//执行代码}
}
[pool release];
```

这里我们在循环 100 万次的代码中每次都创建一个 NSString 实例，由于 NSString 没有使用 alloc 创建实例，因此我们使用了自动回收池。但这段代码的问题是这 100 万个 NSString 要在 for 循环完毕，最后的 pool release 方法调用后才会回收，这就会造成在 for 循环调用过程中，内存占用一直上涨。当然，你可以改进上面的代码如下所示：

```
NSAutoreleasePool *pool=[[NSAutoreleasePool alloc]init];
for(int i=0;i<1000000;i++){
    NSString *s=@"...";
    if(i%1000==0){
        [pool release];
        pool=[[NSAutoreleasePool alloc] init];
    }
}
```

上面的代码每循环 1000 次，就回收自动回收池，然后再紧接着重新创建一个自动回收池给下 1000 个 NSString 对象使用。其实这个问题很像 Hibernate 的一个问题：给你一张 1000 万行记录的 Excel，让你导入到数据库，你该怎么做呢？直接的做法如下所示：

```
Session session=获取 Hibernate 的 JDBC 连接对象
for(int i=0;i<Excel 的行数;i++){
    Object obj=每一行的 Excel 记录对应的 JAVA 对象;
    session.save(obj);
}
Transaction.commit();
```

由于 Hibernate 的一级缓存是在你提交事务的时候才清空，并且刷新到数据库上的，因此在循环结束前，你的一级缓存里会积累大量的 obj 对象，使得内存占用越来越大。

解决办法与 Object-C 的自动回收池差不多，就是如下的做法：

```
Session session=获取 Hibernate 的 JDBC 连接对象
for(int i=0;i<Excel 的行数;i++){
    Object obj=每一行的 Excel 记录对应的 JAVA 对象;
    session.save(obj);
    if(i%1000==0){
        session.flush();
    }
}
Transaction.commit();
```

我们看到每隔 1000 次就刷新一级缓存，它的作用就是清理一级缓存，把数据都发送到数据

库上面去，但是我并没有提交事务，也就是数据都从 JVM 转移到数据库的缓冲区累积了，数据库依然会等待循环结束后的 commit()操作才会提交事务。

15. 常用的类型：

(1.) 字符串：

Cocoa 中使用 NSString 表示字符串，字面值使用@”“的形式。除了前面看到 initWithString、initWithCString 的初始化方法外，NSString 还有其他的初始化方法和类方法创建 NSString 的实例，例如：

+(NSString*) stringWithFormat: @””,…

与 NSLog 一样，使用格式化模版格式化字符串。

NSString 也有一些其他的成员方法：

-(BOOL) isEqualToString: (NSString*) s

比较两个字符串是否相等，与 JAVA 一致的地方是==比较指针，比较对象是否相同要用到 equal 方法。

-(NSComparisonResult) compare: (NSString*) s options: (NSStringCompareOptions) options

这个方法用于详细的比较两个字符串是否相等，譬如：是否忽略大小写等。

返回值 NSComparisonResult 为 C 语言的 enum 类型，常用的枚举值为 NSOrderedSame，表示比较结果相等。

第二个参数 NSStringCompareOptions 也是个 enum 类型，常用的枚举值为 NSCaseInsensitiveSearch、NSNumericSearch，分别表示忽略大小写、字数是否相等。由于 NSStringCompareOptions 的枚举值都是 2 的指数，所以你可以使用位或运算表示同时具备两个条件，例如：NSCaseInsensitiveSearch|NSNumericSearch 表示忽略大小写并且字数相同。

-(BOOL) hasPrefix: (NSString) s

表示字符串是否以参数 s 作为前缀，当然也有 hasSufix 犯法判断后缀。

-(NSRange) rangeOfString: (NSString*) s

判断是否包含参数 s，这里返回前面提到过的结构体 NSRange。如果包含字符串 s，则 NSRange 的 location 为包含的字符串 s 所在的起始位置，length 为长度。如果不包含字符串 s，则 location 为 NSNotFound，length 为 0。

例：

```
NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
NSString *s1=[NSString stringWithFormat: @"You height is %d weight is %d!",168,68];
NSLog(s1);
//长度
NSLog(@"The str length is %d!",[s1 length]);

NSString *s2=@"You Height is 168 Weight is 68!";

//比较
if([s2 isEqualToString: s1]==YES)
```

```
    NSLog(@"Equal YES");
else
    NSLog(@"Equal NO");

//详细比较
NSComparisonResult cr=[s2 compare: s1
          options: NSCaseInsensitiveSearch|NSNumericSearch];
if(cr==NSOrderedSame){
    NSLog(@"Compare YES");
}
else{
    NSLog(@"Compare NO");
}

//判断前缀
if([s1 hasPrefix: @"The"]){
    NSLog(@"hasPrefix YES");
}
else{
    NSLog(@"hasPrefix NO");
}

//判断 s1 中是否含有字符串 height
NSRange range=[s1 rangeOfString: @"height"];
if(!(range.location==NSNotFound)){
    NSLog(@"The \"height\" is located in %u!",range.location);
    NSLog(@"The \"height\" length is %u!",range.length);
}
[pool release];
```

NSString 是长度不可变的字符串，NSMutableString 继承自 NSString，实现了可变字符串。NSMutableString 一般使用类方法

+(NSMutableString*) stringWithCapacity: (int) capacity 进行创建，capacity 指定预先分配的字符串长度。

-(NSMutableString*) appendString: (NSString*) s

这与 JAVA 的 StringBuffer 的 append 没什么区别。

-(void) deleteCharactersInRange: (NSRange) range

这个方法删除指定范围的字符串，常与 rangeOfString 方法联用。

例：

```
NSMutableString *ms1=[NSMutableString stringWithCapacity: 100];
[ms1 appendString: @"You height is "];
```

```
[ms1 appendFormat: @"%d weight is %d!", 168, 68];
NSLog(@"%@",ms1);

NSRange range=[ms1 rangeOfString: @" weight is 68"];
[ms1 deleteCharactersInRange: range];
NSLog(@"%@",ms1);
```

(2.) 数组：

Cocoa 使用 NSArray 表示数组，但是它不能存储基本数据类型、enum、struct、nil，只能存储 Object-C 的对象。

例：

```
NSArray *array=[NSArray arrayWithObjects:   @"One", @"Two", @"Three", nil];
//从这个类方法 arrayWithObjects 的定义可以看出，它使用 nil 表示数组元素结束，这也是 nil 不能存储在 NSArray 中的原因。
int count=[array count];//获取数组元素的格式，本例中为 3。
int i;
for(i=0; i<count;i++){
    NSLog(@"%@",[array objectAtIndex: i]);
    //遍历数组元素，objectAtIndex 方法获取指定索引位置的元素。
}

NSString *s=@"iPhone,Android,Windows Phone 7";
array=[s componentsSeparatedByString: @","];//按照一个字符串将字符串拆分为数组。
s=[array componentsJoinedByString: @" "];//按照一个字符串将数组连接为字符串。
NSLog(s);
```

同理，NSMutableArray 为长度可变的数组，相当于 JAVA 中的 List。

例：

```
NSMutableArray *mArray=[NSMutableArray arrayWithCapacity: 10];
[mArray addObject: @"Apple"];//添加数组元素
[mArray addObject: @"Google"];
[mArray addObject: @"MicroSoft"];
[mArray removeObjectAtIndex: 2];//移除指定索引位置的数组元素
s=[mArray componentsJoinedByString: @","];//拼接数组元素为字符串
NSLog(s);

NSEnumerator *e = [mArray objectEnumerator];
//获取数组的迭代器，相当于 JAVA 中的 Iterator，reserveObjectEnumerator 用于获取反转之后的数组迭代器。与 JAVA 一致的地方是你在使用迭代器时，不能对数组进行添加、删除操作。
id obj;
while(obj=[e nextObject]){
    NSLog(@"%@",obj);
}
```

```
printf("----------------------------\n");
/*
for(NSString *ms in mArray){
    NSLog(@"%@",ms);
}
*/
//for-each 快速枚举为 Object-C 2.0 的新特性，Windows 上的 GNUStep 并不支持。
```

(3.) 字典（哈希表）：

NSDictionary 用于存储 key-value 的数据结构，与 JAVA 中的 Map 类似。

例：

```
NSDictionary *dic=[NSDictionary dictionaryWithObjectsAndKeys: @"Apple", @"A", @"Google", @"G", nil];
//dictionaryWithObjectAndKeys 后的可变参数，每两个为一个 value-key，以 nil 表示结束。
NSLog(@"%@",[dic objectForKey: @"A"]);//按照指定的 key 查询 value
```

同样的有 NSMutableDictionary 表示长度可变的字典。

例：

```
NSMutableDictionary *mDic=[NSMutableDictionary dictionaryWithCapacity: 10];
[mDic setObject: @"Apple" forKey: @"A"];//添加 value-key 对
[mDic setObject: @"Google" forKey: @"G"];
[mDic setObject: @"Windows Phone 7" forKey: @"W"];
[mDic removeObjectForKey: @"W"];//移除指定 key 的 value
/*
    for(id key in mDic){
        NSLog(@"%@ : %@",key,[mDic objectForKey: key]);
    }
*/
//快速迭代的 for-each 循环

NSEnumerator *keyEnum=[mDic keyEnumerator];//获得 key 的枚举器
id key;
while(key=[keyEnum nextObject]){
        NSLog(@"%@ : %@",key,[mDic objectForKey: key]);
}
```

(4.) 哈希 Set：

NSSet 表示以 hash 方式计算存储位置的集合，与 JAVA 中的 HashSet 是一致的。在 NSSet 中的每个对象都有一个唯一的 hash 值，重复的对象将只能保留一个。因此，这引出了 Object-C 中的对象比较问题，这需要你实现从 NSObject 继承而来的如下两个方法：

```
- (BOOL) isEqual: (id) anObject;
- (NSUInteger) hash;
```

这与 JAVA 的对象比较没有什么区别，两个相等的对象必须有相同的 hashCode，所以这两个

方法必须同时实现。下面我们看一个示例。

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

@interface Person: NSObject{
        int pid;
        NSString *name;
}
-(void) setPid: (int) pid;
-(void) setName: (NSString*) name;
-(int) pid;
-(NSString*) name;
@end

@implementation Person
-(void) setPid: (int) p{
        pid=p;
}
-(void) setName: (NSString*) n{
        [n retain];
        [name release];
        name=n;
}
-(int) pid{
        return pid;
}
-(NSString*) name{
        return name;
}
- (NSUInteger) hash{
        return pid+[name hash];
}
-(BOOL) isEqual: (id) p{
        if(pid==[p pid]&&[name isEqualToString: [p name]]){
                return YES;
        }else{
                return NO;
        }
}
-(void) dealloc{
        [name release];
        [super dealloc];
}
@end
```

```
int main (int argc , const char * argv[]){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Person *person1=[[Person alloc] init];
    [person1 setPid: 1];
    [person1 setName: @"Name1"];

    Person *person2=[[Person alloc] init];
    [person2 setPid: 1];
    [person2 setName: @"Name1"];

    Person *person3=[[Person alloc] init];
    [person3 setPid: 3];
    [person3 setName: @"Name3"];

    NSSet *set=[NSSet setWithObjects: person1, person2, person3, nil];
    NSEnumerator *e=[set objectEnumerator];
    Person *person;
    while(person=[e nextObject]){
            NSLog(@"%d",[person pid]);
    }
    [pool release];
    return 0;
}
```

我们看到 person1 与 person2 实例相同，所以最后输出两个 Person 的 pid，而不是三个。

同样的，NSMutableSet 表示长度可变的哈希 Set。

(5.) 封装类：

前面的几个容器类的对象都不能存放基本数据结构、enum、struct、nil，怎么办呢？在 JAVA 中我们知道所有的基本数据类型都有对应的封装类，例如：int---Integer、boolean---Boolean，使用封装类可以把基本数据类型包装为对象，而封装类本身又有方法可以返回原来的基本数据类型。Cocoa 使用 NSValue 作为封装类。

例：

```
NSRect rect=NSMakeRect(1,2,30,50);//这个是在前面提到过的一个表示矩形的结构体
NSValue *v=[NSValue valueWithBytes: &rect objCType: @encode(NSRect)];
//valueWithBytes 要求传入包装数据的地址，也就是变量中存储的数据的首地址，我们依然使用 C 语言的&作为地址运算符，计算指针存储的首地址。
//objCType 要求传入用于描述数据的类型、大小的字符串，使用@encode 指令包装数据所属的类型。
NSMutableArray *mArray=[NSMutableArray arrayWithCapacity: 3];
[mArray addObject: v];
NSRect rect2;
[[mArray objectAtIndex: 0] getValue: &rect2];
```

```
//NSValue 的-(void) getValue: (void*) value 方法可以将 NSValue 中的数据内容取出，要求传入的参数是一个指针，也就是说 getValue 方法将你传入的指针指向存储的数据。
//一般 Cocoa 中的 getXXX 方法都是这个作用，这也是为什么前面的 getter 方法不要以 get 作为方法前缀的原因。
printf("%f\n",rect2.size.width);
```

对于基本数据类型，你可以使用简便的 NSNumber 来封装，它是 NSValue 的子类。

例：

```
NSNumber *n1=[NSNumber numberWithChar: 'A'];
NSNumber *n2=[NSNumber numberWithInt: 100];
NSNumber *n3=[NSNumber numberWithFloat: 99.9F];
NSNumber *n5=[NSNumber numberWithBool: YES];
printf("%d\n",[n2 intValue]);
```

如果你想存储控制到集合类，可以使用 NSNull，它使用 NSNull *n =[NSNull null];的方式创建。

(6.) 日期类型：

Cocoa 中使用 NSDate 类型表示日期。

例：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
//获取当前时间
NSDate *date=[NSDate date];
NSLog(@"Today is %@!",date);

//格式化一个指定的字符串为日期类型
NSCalendarDate *date2=[NSCalendarDate dateWithString:@"26 Apr 2008"
                                      calendarFormat:@"%d %b %Y"];
NSLog(@"Today is %@!",date2);

//获取当前时间的前一天的时间
NSDate *date3=[NSDate dateWithTimeIntervalSinceNow: -24*60*60];
NSLog(@"Yestoday is %@!",date3);
[pool release];
```

你会看到 Shell 窗口输出如下内容：

```
2011-03-31 16:18:43.101 Date[2540] Today is 2011-03-31 16:18:43 +0800!
2011-03-31 16:33:51.624 Date[488] Today is 26 Apr 2008!
2011-03-31 16:38:17.720 Date[4008] Yestoday is 2011-03-30 16:38:17 +0800!
```

(7.) 数据缓冲区：

Cocoa 中使用 NSData 类型来实现缓冲区，其实你可以把 NSData 当作 JAVA 中的字节数组，用于存储二进制的数据类型，譬如：从网络下载回来的文件等。

NSData 的创建使用如下的类方法：

+(NSData*) dataWithBytes: (const void*) bytes length: (NSUInteger) length;

第一参数指定你要缓冲的数据的起始位置，也就是指针里存储的首地址，第二个参数指定数据的长度。

例：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
const char *cs="He is very heigh!";
NSData *data=[NSData dataWithBytes:cs length:strlen(cs)+1 ];
NSLog(@"%@",data);
[pool release];
```

首先我们定义了一个 C 语言的字符序列，然后把它的指针、长度赋给了 NSData 的创建方法。这里需要注意的是由于 C 语言的字符序列的指针地址本身就是数组的首地址，所以不需要再加 &计算指针的地址了。另外，长度要再加 1 也是 C 语言的特性。C 语言里的字符串以\0 结尾。

你会看到 Shell 窗口输出一组 16 进制的数字，也就是缓冲区中存储的内容。

NSData 是长度不可变的数据缓冲区，还有一个 NSMutableData 用来存储长度可变的数据缓冲区。

16. 写入和读取属性：

在 iPhone 的*.ipa 文件中，你经常可以看到*.plist 文件，它保存了程序的相关属性，叫做属性列表。其实它就是 NSArray、NSDictionary、NSString、NSData 持久化之后的文件。

这几个类型都有一个成员方法 **writeToFile: (NSString*) file atomically: BOOL** 用于将自己写入到一个文件。atomically 为 YES 表示文件先存储到临时文件区，如果文件保存成功，再替换掉原始文件，这样的好处是可以防止在保存文件过程中出错，但是仅适用于小文件，因为你相当于是在临时文件存储区也放了一份，再加上原始文件，就是两份文件，如果文件比较大，将会占用较多的用户的磁盘空间。

例：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
NSArray *array=[NSArray arrayWithObjects: @"Apple", @"Google" , @"Microsoft" ,nil];
[array writeToFile: @"plist.txt" atomically: NO];
[pool release];
```

你会在当前文件夹下面找到 plist.txt，打开之后内容如下所示：

```
(
    Apple,
    Google,
    Microsoft
)
```

因为我们用的是 Windows 上的 GNUStep，你在 Mac 电脑上看到的会是 XML 的格式。

这几个对象还提供了相应的方法，将文件还原为对象。

例：

```
NSArray *array2=[NSArray arrayWithContentsOfFile: @"plist.txt"];
```

NSLog(@"%@",[array2 objectAtIndex: 1]);
这个还原文件为对象的方法有个陷阱，就是即便加载文件失败，也不会报告任何的错误，仅仅是返回的指针指向 nil。

如果你要持久化的类型不是上述的数组、字典、缓冲区，那该怎么办呢？Object-C 中也有和 JAVA 一样的序列化、反序列化支持，使用 NSCoding 协议。NSCoding 协议定义了如下两个方法：

-(void) encodeWithCoder: (NSCoder*) coder;

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

第一个方法相当于编码对象，第二个方法是解码回对象，也就相当于给类型定义了一个新的 init 方法而已。

例：

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

@interface Person: NSObject <NSCoding>{
        int pid;
        NSString *name;
        float height;
}
-(void) setPid: (int) pid;
-(void) setName: (NSString*) name;
-(void) setHeight: (float) height;
-(int) pid;
-(NSString*) name;
-(float) height;
@end

@implementation Person
-(void) setPid: (int) p{
        pid=p;
}
-(void) setName: (NSString*) n{
        [n retain];
        [name release];
        name=n;
}
-(void) setHeight: (float) h{
        height=h;
}
-(int) pid{
        return pid;
}
-(NSString*) name{
```

```
        return name;
}
-(float) height{
        return height;
}
-(void)   encodeWithCoder:   (NSCoder*) coder{
        [coder encodeObject: name forKey: @"name"];
        [coder encodeInt: pid forKey: @"pid"];
        [coder encodeFloat: height forKey: @"height"];
}
-(id) initWithCoder: (NSCoder*) decoder{
        self=[super init];
        if(self){
                name=[decoder decodeObjectForKey: @"name"];
                [self setName: [decoder decodeObjectForKey: @"name"]];
                //这里千万不能写成 name=[decoder  decodeObjectForKey:  @"name"],
            因为 name 是对象，你需要调用 setter 方法，运行里面的 retain 操作。
                pid=[decoder decodeIntForKey: @"pid"];
                height=[decoder decodeFloatForKey: @"height"];
        }
        return self;
}
-(void) dealloc{
        [name release];
        [super dealloc];
}
@end

int main (int argc , const char * argv[]){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Person *person=[Person new];
    [person setName: @"张明"];
    [person setPid: 1];
    [person setHeight: 185.5];
    //归档对象
    NSData *data=[NSKeyedArchiver archivedDataWithRootObject: person];
    [data writeToFile: @"person.db" atomically: NO];
    [person release];

    //还原数据为对象
    Person *person2=[NSKeyedUnarchiver unarchiveObjectWithData: data];
    NSLog(@"%@",[person2 name]);
    [pool release];
    return 0;
```

}

红色部分的方法是编码对象的实现，其实就是用参数 NSCoder 的 encodeXXX:forKey:方法给你要编码的字段取个名字。绿色部分的方法是解码对象的实现，就是用参数 NSCoder 的 decodeObjectForKey:解码指定的字段。在 main 函数中使用 NSKeyedArchiver 的类方法 archivedDataWithRootObject 将要序列化的对象转换为 NSData 的数据缓冲区类型，然后再通过前面提到过的 writeToFile:atomically:方法将数据写入到硬盘。我们还原数据为对象使用的是 NSKeyedUnarchiver 的类方法 unarchiveObjectWithData。

17. 对象的复制:

对象的复制就相当于 JAVA 中的 clone()方法，也就是对象的深度复制，所谓深度复制就是重新分配一个存储空间，并将原对象的内容都复制过来，从这些描述可以看出，复制也会分配空间，那就是你要对复制出来的对象 release，就是前面所说的 alloc、new、copy 操作创建的对象，要手工 release。

Object-C 中的一个对象是否可以被复制，要看它的类型是否遵循 NSCopying 协议，这个协议中有个复制方法**-(id) copyWithZone: (*NSZone) zone** 需要我们去实现。

我们这里使用 Car（汽车）对象进行演示，Car 里有 Engine（引擎）、Tire（轮胎）两个成员变量。

Engine.h

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

@interface Engine: NSObject <NSCopying>
@end
```

Engine.m

```
#import "Engine.h"

@implementation Engine
-(id) copyWithZone: (NSZone*) zone{
        Engine *engine=[[[self class] allocWithZone: zone] init];
        return engine;
}
@end
```

这里我们看一下 NSCopying 的 copyWithZone 方法，这个方法的调用是 Object-C 的运行环境来完成的，我们的程序在想复制一个对象的时候，调用的是 copy 方法，这个方法是从 NSObject 继承而来的。copyWithZone 只有一个参数，就是系统传进来的一个表示内存空间的 NSZone 对象，也就是在这个空间里复制原来的对象。复制一个对象需要经过如下几个步骤：

(1.) 首先调用类方法 allocWithZone 传入 NSZone 参数，为即将产生的新对象分配空间。你要注意这个类方法必须用[self class]来调用，而不能用 Engine 来调用。因为可能 Engine 会有子类，子类会有新的成员变量。那么子类在调用它的 copyWithZone 的时候，由于 allocWithZone 用的是 Engine 来调用的，那么也就是分配的空间也是按照 Engine 的结构

来分配的（分配空间前面说过，主要就是给成员变量确定在内存中的位置），这样根本就不会考虑子类新增加的成员变量，从而导致内存溢出，因为子类的新的成员变量根本就没有存储位置。[self class]的意思是获取当前实例的 Class 对象，也就是运行时动态获取的，这样就可以保证子类复制的时候，这个地方返回的是子类的 Class 对象，而不是父类 Engine。

(2.) 然后调用初始化方法，完成新实例的创建。

(3.) 最后是要把原有实例中的内容都 setter 到新实例。

Tire.h

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

@interface Tire: NSObject <NSCopying>{
        int count;//轮胎的数量
}
-(Tire*) initWithCount: (int) count;
-(void) setCount: (int) count;
-(int) count;
@end
```

Tire.m

```
#import "Tire.h"

@implementation Tire
-(Tire*) initWithCount: (int) c{
        self=[super init];
        if(self){
                count=c;
        }
        return self;
}
-(void) setCount: (int) c{
        count=c;
}
-(int) count{
        return count;
}
-(id) copyWithZone: (NSZone*) zone{
        Tire *tire=[[[self class] allocWithZone: zone] initWithCount: count];
        return tire;
}
@end
```

Car.h

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

@interface Car: NSObject <NSCopying>{
        Engine *engine;
        Tire *tire;
}
-(void) setEngine: (Engine*) engine;
-(void) setTire: (Tire*) tire;
-(Engine*) engine;
-(Tire*) tire;
@end
```

Car.m

```
#import "Car.h"

@implementation Car
-(void) setEngine: (Engine*) e{
        [e retain];
        [engine release];
        engine=e;
}
-(void) setTire: (Tire*) t{
        [t retain];
        [tire release];
        tire=t;
}
-(Engine*) engine{
        return engine;
}
-(Tire*) tire{
        return tire;
}
-(id) copyWithZone: (NSZone*) zone{
        Car *car=[[[self class] allocWithZone: zone] init];
        Engine *engineCopy=[engine copy];
        Tire *tireCopy=[tire copy];
        [car setEngine: engineCopy];
        [car setTire: tireCopy];
        [engineCopy release];
        [tireCopy release];
        return car;
}
-(void) dealloc{
```

```
        [engine release];
        [tire release];
        [super dealloc];
}
@end
```

这里的 copyWithZone 稍显麻烦一些，因为 Car 的成员变量都是对象类型，所以 copyWithZone 里的 engineCopy、tireCopy 的创建都使用了 copy 方法，这是因为 Engine、Tire 本身都支持复制操作。

main.m

```
#import "Car.h"

int main(int argc,const char *argv[]){
        Engine *engine=[Engine new];
        Tire *tire=[[Tire alloc] initWithCount: 4];
        Car *car=[Car new];
        [car setEngine: engine];
        [car setTire: tire];
        [engine release];
        [tire release];
        Car *newCar=[car copy];//复制 car 对象
        [car release];
        //至此，原有的对象 car 已经被内存回收，如果下面的代码正常运行并输出，就说明我们的复制操作成功。
        printf("The car has %d tires!",[[newCar tire] count]);
        [newCar release];
        return 0;
}
```

18. 多线程：

Object-C 的多线程编程与 JAVA 语言极其类似分为原始的线程操作、线程池两种，后者其实就是使用队列等机制对前者的封装。JAVA 也一样，原始的线程操作使用 Thread 类，线程池的操作使用 java.util.concurrent.*中的类库。

(1.) NSThread：

Object-C 中 NSThread 类型表示线程，线程对象的创建主要使用以下几个方法：

***A.*- (id) init;**

***B.*- (id) initWithTarget: (id) target selector: (SEL)selector object: (id) argument;**

第一个参数指定目标对象，一般情况下就是当前的实例 self。第二个参数指定要运行的方法，相当于 JAVA 中的 run 方法。第三个参数一般为 nil。

***C.*+(void) detachNewThreadSelector:(SEL)aSelector**
toTarget:(id)aTarget withObject:(id)anArgument

这个方法是类方法，与第二个方法其实在参数上是一致的，区别就是你不需要显示的 release

这个线程，因为没有使用 alloc 关键字。

通常我们使用后两种方法，因为不但 init 完成，还设置好了一个线程所必需的一些元素（例如：运行的 run 方法）。

启动一个线程使用 start 方法，结束一个线程使用 exit 方法，但要注意如果你使用了 exit 方法，首先要将这个线程的引用计数器归 0。

(2.) NSCondition：

NSCondition 用于执行同步操作，在功能上就相当于 synchronized 关键字，在用法上相当于 JAVA 中的 java.util.concurrent.*包中的 Lock 对象。一段代码如果需要同步就用 NSCondition 先 lock 一下，同步的部分执行完毕再 unlock 一下，也就是说 lock、unlock 之间的代码是加锁的。

(3.) 示例程序：

下面我们依然通过 JAVA 中最经典的售票案例演示 Object-C 的多线程代码。

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

//定义售票接口
@interface SellTickets: NSObject{
        int tickets;//未售出的票数
        int count;//售出的票数
        NSThread* ticketsThread1;//线程 1
        NSThread* ticketsThread2;//线程 2
        NSThread* ticketsThread3;//线程 3
        NSCondition* ticketsCondition;//同步锁
}
- (void) ticket;//售票方法
@end

//实现售票类
@implementation SellTickets

//这个方法初始化了成员变量，并启动了三个线程，其中前两个线程执行售票方法 run，第三个线程执行用于测试线程之间是否异步执行的 asyn 方法。
- (void) ticket{
        tickets = 100;
        count = 0;
        ticketsCondition = [[NSCondition alloc] init];

        ticketsThread1 = [[NSThread alloc] initWithTarget:self
                                        selector:@selector(run) object:nil];
        [ticketsThread1 setName:@"Thread-1"];
        [ticketsThread1 start];
```

```
        ticketsThread2 = [[NSThread alloc] initWithTarget:self
                                        selector:@selector(run) object:nil];
        [ticketsThread2 setName:@"Thread-2"];
        [ticketsThread2 start];

        ticketsThread3 = [[NSThread alloc] initWithTarget:self
                                        selector:@selector(asyn) object:nil];
        [ticketsThread3 setName:@"Thread-3"];
        [ticketsThread3 start];

        [NSThread sleepForTimeInterval:80];
}
//测试线程之间是否异步执行。
-(void) asyn{
        [NSThread sleepForTimeInterval:5];
        printf("**********************************\n");
}
- (void)run{
        while (YES) {
                //上锁
                [ticketsCondition lock];
                if(tickets > 0){
                    [NSThread sleepForTimeInterval:0.5];
                    count = 100 - tickets;
                    NSLog(@"当前票数是:%d,售出:%d,线程名:%@",tickets,count,
                                    [[NSThread currentThread] name]);
                    tickets--;
                }else{
                    break;
                }
                //解锁
                [ticketsCondition unlock];
        }
}

- (void)dealloc {
        //回收线程、同步锁的实例
        [ticketsThread1 release];
        [ticketsThread2 release];
        [ticketsThread3 release];
        [ticketsCondition release];
        [super dealloc];
}
```

```
@end

int main(int argc,const char *argv[]){
        SellTickets *st=[[SellTickets alloc] init];
        [st ticket];
        [st release];
        return 0;
}
```

注意红色部分的代码，我们在这里让主线程暂停 80 秒，因为如果你不这么做，由于三个线程是异步执行的，ticket 方法被主线程执行完毕后，由于主线程的退出，会导致三个线程也被结束掉，你会发现售票过程根本没有被执行。实际开发中你是不需要这么做的，例如：你在一个 UI 窗体上异步加载图片，由于主线程 UI 窗体不会退出，因此你发起的线程也就不会被结束掉，你就不必加这么一句暂停主线程的垃圾代码了。

另外就是线程 3，它没有参与售票，用意就是想看一下三个线程是否异步执行，按照 asyn 中的代码，在程序 5 秒钟后，线程 3 会夹杂在线程 1、2 的售票输出语句中输出一串*。

(4.) 与主线程的交互：

如果你想更新 UI 上的某一个部件，就需要在发起的新线程里调用 UI 所在的主线程上的一个方法，新线程不能直接访问主线程的方法，需要在 run 方法中使用如下的方法：

```
- (void)  performSelectorOnMainThread: (SEL)  aSelector
        withObject: (id)  arg  waitUntilDone: (BOOL)  wait
```

例如：

```
- (void) run{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    [self performSelectorOnMainThread: @selector(moveText)
                        withObject: nil waitUntilDone: NO];
    //移动窗体上的一个文本标签
    [pool release];
}
```

(5.) 线程池：

Object-C 使用 NSOperation、NSOperationQueue 两个类型实现线程池的操作，NSOpertion 就好比 JAVA 中的 Runnable，其中的 main（名字有点儿奇怪）方法也就是你要执行的操作，NSOperationQueue 是一个线程池队列，你只需要把 NSOperation 添加到 NSOperationQueue，队列就会 retain 你的 NSOperation，然后执行其中的操作，直到执行完毕。队列中会有多个操作，队列会按照你添加的顺序逐个执行，也就说队列中的任务是串行的。

例：

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

//定义一个任务对象，它要实现 NSOperation，这与 JAVA 中的任务要实现 Runnable 接口是一样的。
@interface MyOperation: NSOperation{
        int num;
```

```
}
//我们自定义的 init 方法。
-(MyOperation*) initWithNum: (int) num;
@end


@implementation MyOperation
-(MyOperation*) initWithNum: (int) n{
        self=[super init];
        if(self){
            num=n;
        }
        return self;
}
-(BOOL) isConcurrent{
        return YES;
}
//任务方法，从 NSOperation 继承
-(void) main{
        //[NSThread sleepForTimeInterval: 5];
        NSLog(@"%d\n",++num);
}
@end

int main(int argc,const char *argv[]){
        NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
        NSOperation *operation1=[[MyOperation alloc] initWithNum: 1];
        [operation1 autorelease];
        NSOperation *operation2=[[MyOperation alloc] initWithNum: 2];
        [operation2 autorelease];
        NSOperationQueue *queue=[[NSOperationQueue alloc] init];
        [queue setMaxConcurrentOperationCount: 2];
        [queue addOperation: operation1];//将任务添加到队列
        [queue addOperation: operation2];
        [NSThread sleepForTimeInterval: 15];
        [queue release];
        [pool release];
        return 0;
}
```

这里的红色代码不用说明了，与前面的例子一样，为了不让主线程结束。蓝色的方法 isConcurrent 是表明该 Operation 可以并发执行，也就是它在 Queue 中与其他任务是并行执行的，而不是一开始说的 NSOperationQueue 中的 Operation 是串行执行的，默认 Operation 串行执行的原因是 NSOperation 中的 isConcurrent 方法返回 NO。绿色的代码表示 Queue 中最多可以并行支持的 Operation 数量，如果超过这个值，则放入等待队列。但是 GNUStep 似

乎不支持并行执行，只要 isConcurrent 方法返回 YES，运行时就会报错。

如果线程池可以满足你的业务需求，尽量使用线程池，而不是原始的 NSThread，因为使用线程池屏蔽了许多线程自身需要处理的问题，代码也更加简洁。

19. KVC 与 KVO：

KVC 是 NSKeyValueCoding 的缩写，它是 Foundation Kit 中的一个 NSObject 的 Category，作用你可以类比 JAVA 中的反射机制，就是动态访问一个对象中的属性。我们以第 16 节中的 Person 为例进行代码演示。

我们知道正常访问 Person 中的 name 的方式为：

```
[person setName: @"豆豆"];//setter
NSLog(@"%@",[person name]);//getter
```

KVC 中可以这样访问：

```
[person setValue: @"太狼" forKey: @"name"];
NSLog(@"%@",[person valueForKey: @"name"]);
```

我们看到 KVC 使用了和 NSDictionary 差不多的 key-value 的操作方法，也就是说它是按照一个字符串 key 在对象中查找属性，然后给它设置值、获取值。如果 key 为 xxx，则对于获取属性的值的查找规则如下所示：

A. 首先查找.h 文件中声明的成员方法 getXxx 或者 xxx；
B. 然后查找.m 文件中声明的变相的私有成员方法_getXxx 或者_xxx
C. 最后查找成员变量_xxx 或者 xxx。

KVC 还支持对象导航图的方式访问属性，首先看一段伪代码：

```
Person{
    Address *address;
}
-(id) init{
    self=[super init];
        if(self){
            address=[Address new];
        }
        return self;
}
@end

Address{
    NSString *city;
}
@end

Person person=[[Person alloc] init;
```

如果我们想访问 person 中的 address 中的 city，你可以如下操作：
[person **setValue**: @"北京" **forKeyPath**: @"address.city"];
NSLog(@"%@",[person **valueForKeyPath**: @"address.city"]);
我们注意到与前面的相比，这里多了个 Path，也就是按照路径搜索，所谓的路径就和 JAVA 中的 EL 表达式差不多，你可以使用.操作符，一级一级的查找属性。这里请注意红色代码 address=[Address new];，如果没有这一句，你会发现上面输出 nil，也就是设置值没有成功，原因是 person 的 address 是对象类型，初始化的时候指向了 nil，还没有分配存储空间，因此你也就不能用 KVC 去给 address 里面的 city 设置值。

你还要注意的是 KVC 有个和 EL 表达式不一样的地方，就是你不能对数组使用索引，也就是 Person 里有一个 NSArray *schools 的成员变量，你不能使用 schools[0]这种形式，只能获取全部的 schools 再自己分拣。

虽然 KVC 不支持对数组进行索引操作，但是支持运算符@count、@sum、@avg、@min、@max 操作。假设 Person 的 NSArray *school 的数组中存放的是 School 类型，School 有学校的名字 name、入学时间 date、课程数目 lessons。那么[person valueForKeyPath: @"schools.@count"]可以计算数组中有多少个学校，[person valueForKeyPath: @"schools.@sum.lessons"]可以计算这个人读过的所有学校的课程总数。

KVC 在解析 key 的字符串的时候，是会比你正常调用 setter、getter 要慢的，而且编译器无法在编译器对你的方法调用做出检查（因为你的属性名都是字符串，只有运行时才会知道你有没有写错），出错的几率也会提高，所以请不要随意使用 KVC，而省去 setter、getter 方法。KVC 一般用于动态绑定，也就是运行时才能确定谁调用哪个方法，编译期并不确定。

KVO 就是 NSKeyValueObserving 的缩写，它也是 Foundation Kit 中的一个 NSObject 的 Category，KVO 基于 KVC 实现，基于观察者设计模式（Observer Pattern）实现的一种通知机制，你可以类比 JAVA 中的 JMS，通过订阅的方式，实现了两个对象之间的解耦，但又可以让他们相互调用。
按照观察者模式的订阅机制，KVO 中必然有如下三个方法：

A. 订阅（Subscribe）

```
- (void) addObserver: (NSObject*) anObserver
                     forKeyPath: (NSString*) aPath
                     options: (NSKeyValueObservingOptions) options
                     context: (void*) aContext;
```

参数 options 为 NSKeyValueObservingOptionOld、NSKeyValueObservingOptionNew。

B. 取消订阅（Unsubscribe）

```
- (void) removeObserver: (NSObject*) anObserver
                          forKeyPath: (NSString*) aPath;
```

C. 接收通知（Receive notification）

```
- (void) observeValueForKeyPath: (NSString*) aPath
                          ofObject: (id) anObject
```

change: (NSDictionary*) aChange
context: (void*) aContext;

这三个方法的参数不大容易直接说清楚都是什么意思，下面我们通过实际的代码来理解。这段代码的业务含义就是警察一直监视犯人的名字是否发生变化，只要发生变化，警察就会收到通知。

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

//犯人类型
@interface Prisoner: NSObject{
        int pid;
        NSString *name;
}
-(void) setPid: (int) pid;
-(void) setName: (NSString*) name;
-(int) pid;
-(NSString*) name;
@end

@implementation Prisoner
-(void) setPid: (int) p{
        pid=p;
}
-(void) setName: (NSString*) n{
        [n retain];
        [name release];
        name=n;
}
-(int) pid{
        return pid;
}
-(NSString*) name{
        return name;
}
-(void) dealloc{
        [name release];
        [super dealloc];
}
@end

//警察类型
@interface Police: NSObject
@end
```

```
@implementation Police
//接收通知的方法，继承自 NSObject 父类。
//请先看 main 函数中的 addObserver 方法参数的解释再来这个方法的解释。
//第一个参数是你监视的对象上的属性，第二个参数是你监视的对象，第三个参数存放了你监视的属性的值，最后一个参数我们传递 nil。
- (void) observeValueForKeyPath: (NSString*) aPath
                        ofObject: (id) anObject
                                change: (NSDictionary*) aChange
                        context: (void*) aContext{
    if([aPath isEqualToString: @"name"]){
            NSLog(@"Police : %@",[aChange objectForKey: @"old"]);
            NSLog(@"Police : %@",[aChange objectForKey: @"new"]);
            //因为 main 函数中我们监听 name 的新旧两个值，所以 aChange 这个字典对象里就存放了@"old"、@"new"两个 key-value 对。
    }
}
@end


int main (int argc , const char * argv[]){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Prisoner *prisoner=[[Prisoner alloc] init];
    Police *police=[[Police alloc] init];
    //为犯人添加观察者警察，警察关注犯人的 name 是否发生变化，如果发生变化就立即通知警察，也就是调用 Police 中的 observeValueForKeyPath 方法。
    //换句话说就是警察对犯人的名字很感兴趣，他订阅了对犯人的名字变化的事件，这个事件只要发生了，警察就会收到通知。
    [prisoner   addObserver: police
            forKeyPath: @"name"
            options: NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
            context: nil];
    //addObserver 的调用者是要被监视的对象，第一个参数是谁要监视它，第二个参数是监视它的哪个属性的变化（使用 KVC 机制，也就是前面所说的 KVO 基于 KVC），第三个参数是监视属性值改变的类型，我们这里监听 Old、New，也就是 Cocoa 会把 name 属性改变之前的旧值、改变之后的新值都传递到 Police 的处理通知的方法，最后一个参数我们传递 nil。
    //这里有一个陷阱，如果你不小心把 forKeyPath 的属性名写错了，prisoner 里根本就不存在，那么 Cocoa 不会报告任何的错误。所以当你发现你的处理通知的 observeValueForKeyPath 没有任何反应的时候，首先看一下是不是这个地方写错了。
    [prisoner setName: @"豆豆"];
    //警察取消订阅犯人名字变化的事件。
    [prisoner removeObserver: police
                        forKeyPath: @"name"];
```

```
    [prisoner setName: @"太狼"];
    [prisoner release];
    [police release];
    [pool release];
    return 0;
}
```

运行之后，Shell 窗口输出：

```
2011-04-01 15:20:33.467 Kvo[2004] Police : <null>//因为 name 没有 old 值
2011-04-01 15:09:18.479 Kvo[4004] Police : 豆豆
```

Notification 是 Object-C 中的另一种事件通知机制，我们依然以犯人、警察的示例进行演示。

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

//犯人类型
@interface Prisoner: NSObject{
        int pid;
        NSString *name;
}
-(void) setPid: (int) pid;
-(void) setName: (NSString*) name;
-(int) pid;
-(NSString*) name;
@end

@implementation Prisoner
-(void) setPid: (int) p{
        pid=p;
}
-(void) setName: (NSString*) n{
        [n retain];
        [name release];
        name=n;
}
-(int) pid{
        return pid;
}
-(NSString*) name{
        return name;
}
-(void) dealloc{
        [name release];
        [super dealloc];
}
@end
```

```
//警察类型
@interface Police: NSObject
-(void) handleNotification:(NSNotification *) notification;
@end

@implementation Police
-(id) init{
        self=[super init];
        if(self){
            //获取通知中心，它是单例的。
            NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
            //向通知中心把自己添加为观察者，第一个参数是观察者，也就是警察自己，第二个参数是观察的事件的标识符 prisioner_name，这就是一个标识性的名字，不是特指哪一个属性，第三个参数指定 handleNotification 为处理通知的方法。
            [nc addObserver: self selector:@selector(handleNotification:)
                                name:@" prisioner_name" object:nil];
    }
}
//接收通知，这个方法的名字任意，只有参数是 Notification 就可以了。
-(void) handleNotification:(NSNotification *) notification{
        Prisoner *prisoner=[notification object];//获得通知中心传递过来的事件源对象
        NSLog(@"%@",[prisoner name]);
}
@end

int main (int argc , const char * argv[]){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Prisoner *prisoner=[[Prisoner alloc] init];
    Police *police=[[Police alloc] init];
    [prisoner setName: @"豆豆"];

    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];

    //向通知中心发送通知，告知通知中心有一个 prisioner_name 的事件发生了，并把自己作为事件源传递给通知中心。
    //通知中心随后就会查找是谁监听了 prisioner_name 事件呢？找到之后就调用观察者指定的处理方法，也就是 Police 中的 handleNotification 方法被调用。
    [nc postNotificationName: @"prisioner_name" object: prisoner];

    //从通知中心移除观察者
    [nc removeObserver: police];
    [prisoner setName: @"太狼"];
    [prisoner release];
```

```
    [police release];
    [pool release];
    return 0;
}
```

Shell 窗口输出如下所示：

2011-04-01 16:20:58.995 Notification[2564] 豆豆

我们看到这种通知方式的实现非常不优雅，观察者 Police 中耦合了通知中心的 API，而且最重要的是看到红色的一行代码了吗？通知的发送是需要被监视的对象主动触发的。

20. 谓词 NSPredicate：

Cocoa 提供了 NSPredicate 用于指定过滤条件，谓词是指在计算机中表示计算真假值的函数，它使用起来有点儿像 SQL 的查询条件，主要用于从集合中分拣出符合条件的对象，也可以用于字符串的正则匹配。首先我们看一个非常简单的例子，对谓词有一个认知。

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

@interface Person: NSObject{
        int pid;
        NSString *name;
        float height;
}
-(void) setPid: (int) pid;
-(void) setName: (NSString*) name;
-(void) setHeight: (float) height;
-(int) pid;
-(NSString*) name;
-(float) height;
@end

@implementation Person
-(void) setPid: (int) p{
        pid=p;
}
-(void) setName: (NSString*) n{
        [n retain];
        [name release];
        name=n;
}
-(void) setHeight: (float) h{
        height=h;
}
-(int) pid{
        return pid;
}
-(NSString*) name{
```

```
        return name;
}
-(float) height{
        return height;
}
-(void) dealloc{
        [name release];
        [super dealloc];
}
@end

int main (int argc , const char * argv[]){
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    //实例化三个 Person，并放入数组。
    NSMutableArray *array=[NSMutableArray arrayWithCapacity: 5];
    Person *person1=[[Person alloc] init];
    [person1 setPid: 1];
    [person1 setName: @"Name1"];
    [person1 setHeight: 168];
    [array addObject: person1];

    Person *person2=[[Person alloc] init];
    [person2 setPid: 2];
    [person2 setName: @"Name2"];
    [person2 setHeight: 178];
    [array addObject: person2];

    Person *person3=[[Person alloc] init];
    [person3 setPid: 3];
    [person3 setName: @"Name3"];
    [person3 setHeight: 188];
    [array addObject: person3];

    //创建谓词，条件是 pid>1 并且 height<188.0。其实谓词也是基于 KVC 的，也就是如果 pid 在 person 的成员变量 xxx 中，那么此处要写成 xxx.pid>1。
    NSPredicate *pre = [NSPredicate predicateWithFormat:
                                                @" pid>1 and height<188.0"];

    int i=0;
    for(;i<[array count];i++){
            Person *person=[array objectAtIndex: i];
            //遍历数组，输出符合谓词条件的 Person 的 name。
            if ([pre evaluateWithObject: person]) {
                    NSLog(@"%@",[person name]);
```

```
            }
        }
        [person1 release];
        [person2 release];
        [person3 release];
        [pool release];
        return 0;
}
```

Shell 窗口输出如下所示：

2011-04-01 16:51:18.382 Predicate[2400] Name2

我们看到创建谓词使用类方法 predicateWithFormat: (NSString*) format，format 里的东西真的和 SQL 的 where 条件差不多。另外，参数 format 与 NSLog 的格式化模版差不多，如果 1 和 188.0 是传递过来的参数，你可以写成如下的形式：

@"pid>%d and height<%f",1,188.0

(1.) 逻辑运算符：AND、OR、NOT

这几个运算符计算并、或、非的结果。

(2.) 范围运算符：BETWEEN、IN

例：

@”pid BETWEEN {1,5}”

@"name IN {'Name1','Name2'}"

(3.) 占位符：

```
NSPredicate *preTemplate = [NSPredicate predicateWithFormat:@"name==$NAME"];
NSDictionary *dic=[NSDictionary dictionaryWithObjectsAndKeys:
                                                    @"Name1", @"NAME",nil];
NSPredicate *pre=[preTemplate predicateWithSubstitutionVariables: dic];
```

占位符就是字段对象里的 key，因此你可以有多个占位符，只要 key 不一样就可以了。

(4.) 快速筛选数组：

前面我们都是使用谓词逐个判断数组内的对象是否符合，其实数组本身有更为便捷的方法，直接筛选出一个符合谓词的新数组。

```
NSPredicate *pre = [NSPredicate predicateWithFormat:@"pid>1"];
NSMutableArray *arrayPre=[array filteredArrayUsingPredicate: pre];
NSLog(@"%@",[[arrayPre objectAtIndex: 0] name]);
```

(5.) 字符串运算符：

BEGINSWITH、ENDSWITH、CONTAINS 分别表示是否以某字符串开头、结尾、包含。

他们可以与 c、d 连用，表示是否忽略大小写、是否忽略重音字母（字母上方有声调标号）。

例：

@”name BEGINSWITH[cd] ‘He’”

判断 name 是否以 He 开头，并且忽略大小写、忽略重音字母。

(6.) LIKE 运算符：

LIKE 使用?表示一个字符，*表示多个字符，也可以与 c、d 连用。

例：

@”name LIKE ‘???er*’” 与 Paper Plane 相匹配。

(7.) SELF：

前面的数组中放的都是对象，如果数组放的都是字符串（或者是其他没有属性的类型），该怎么写谓词呢？这里我们使用 SELF。

例：

```
NSArray *arrays=[NSArray arrayWithObjects: @"Apple", @"Google", @"MircoSoft", nil];
NSPredicate *pre2 = [NSPredicate predicateWithFormat:@"SELF=='Apple'"];
```

(8.) 正则表达式：

NSPredicate 使用 MATCHES 匹配正则表达式，正则表达式的写法采用 international components for Unicode (ICU)的正则语法。

例：

```
NSString *regex = @"^A.+e$";//以 A 开头，以 e 结尾的字符。
NSPredicate *pre= [NSPredicate predicateWithFormat:@"SELF MATCHES %@", regex];
if([pre evaluateWithObject: @"Apple"]){
    printf("YES\n");
}else{
    printf("NO\n");
}
```

21. Object-C 2.0 的新特性：

略。