Objective-C 2.0 的新特性与运行时编程

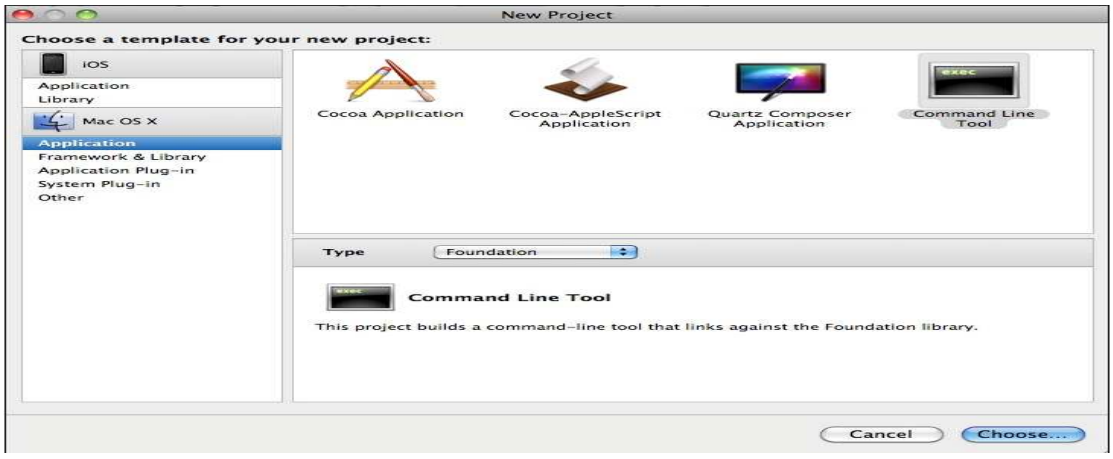
李海峰 QQ:61673110 邮箱:andrew830314@163.com

第一篇文档《Objective-C 的语法与 Cocoa 框架》中我们学习了 Objective-C 的语法与 Cocoa 框架中的 Foundation Kit，使用的是 Windows 操作系统上的 GNUStep 作为开发环境。现在我们转入 Mac OS X 系统，使用 Xcode 作为开发环境开始正统的学习 Objective-C 的编程。本文档主要讲解 Objective-C 2.0 的新特性与运行时编程，在此之前，你先要学会 Xcode 的简单使用。

1. Xcode 的使用：

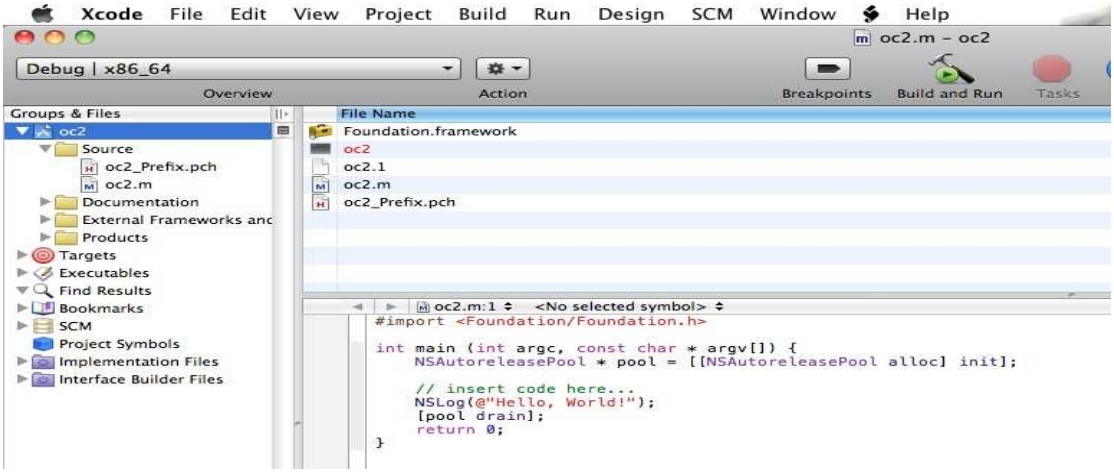
Xcode 是 Mac OS X 上的开发 Mac OS X、iOS 应用程序的一款工具，本文档使用 3.2.6 版本，这个工具极其强大，这里只是讲一些入门级别的使用方法，以便你可以从 GNUStep 快速过渡到 Xcode。Xcode 安装完成后，运行程序在/Developer/Applications/Xcode.app，建议你把它拖拽到 Dock，方便使用。

如果你是第一次打开 Xcode 会看到一个向导窗口，你点击右下角的 Cancel 忽略它即可。然后我们点击最顶端的菜单栏（Mac OS X 系统当前哪个程序的窗口获得鼠标焦点，最顶端的一栏就是哪个程序的菜单栏，为此我很 Farmer 的找菜单栏找了好半天！）File---New Project…，你会看到如下图所示的对话框：



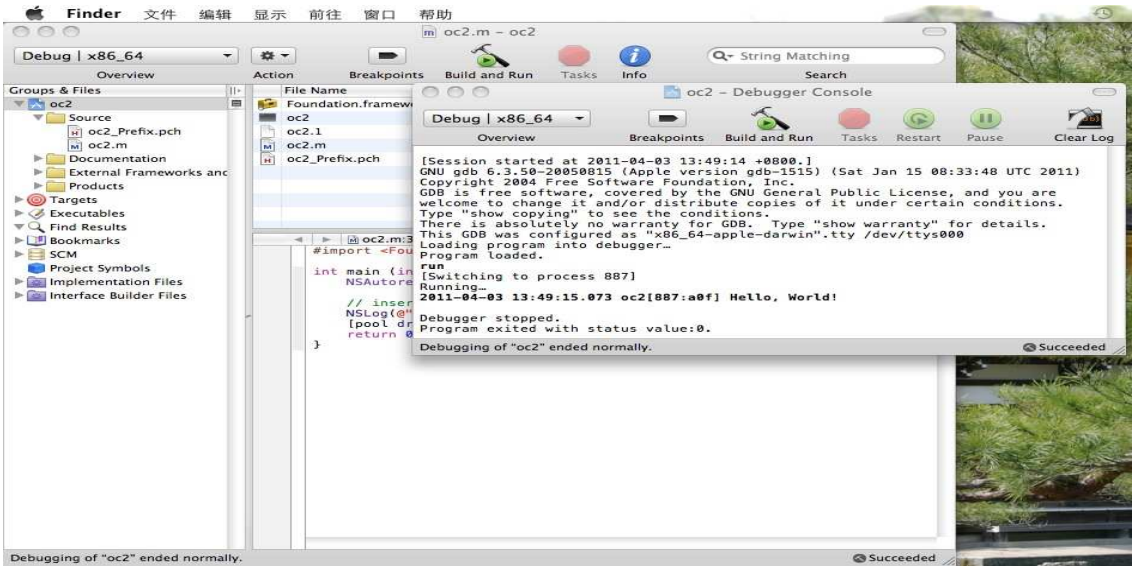
我们在左侧选择系统类型为 Mac OS X，应用类型选择 Application，然后选择右侧的 Command Line Tool，下方的 type 选择 Foundation，最后点击 choose。你会看到又弹出了一个对话框，如下图所示：

这一步要求你输入 Project 的名称和保存 Project 的位置，我们取名为 oc2，保存位置为默认的“文稿”，也就是/Users/登录用户/Documents/。至此我们创建好了一个名称为 oc2 的命令行工程，这是 Xcode 中最简单的工程类型。如下图所示：



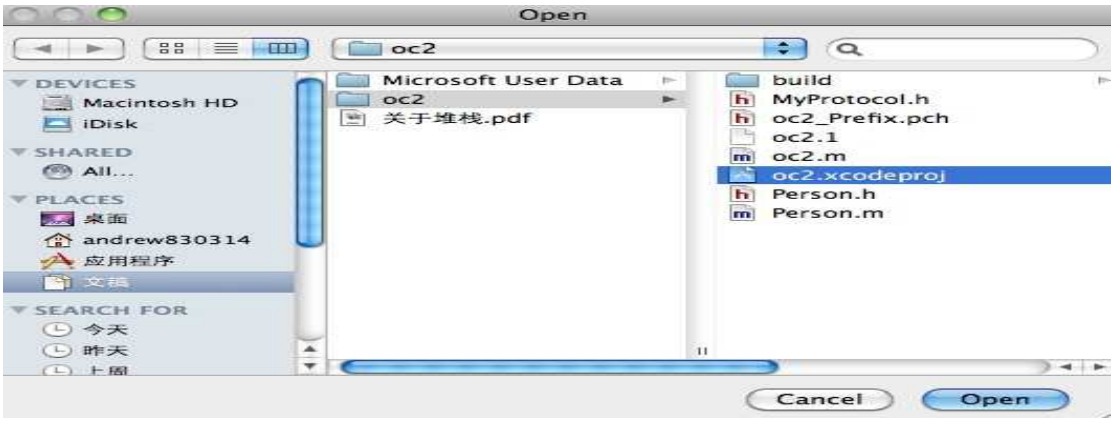
左侧为 Groups & Files，用来显示组成你的程序的所有元素，包括源代码、图片等。这里的 Groups 表示将你散落在硬盘各处的文件组成一个虚拟的组，便于你在此处阅览所有的工程资源。右侧的上半部分为浏览器栏，当左侧的选中位置不同时，浏览器栏会显示左侧当前选中位置中的所有的文件，例如：上图中左侧选中工程 oc2，所以 oc2 下的文件都会显示出来。那么如果左侧选中 Source，就会只显示 Source 下的文件。浏览器栏的右上方有个搜索框（上图中没有展示出来），你可以在里面输入 oc2，那么浏览器栏中就会显示 oc2 开头的文件。Groups & Files 中找到 oc2，并打开其下的 Source 文件夹，你会看到 oc2.m 文件，这就是此工程的 main 函数所在的源码文件。单击 oc2.m 会在右侧浏览器栏的下方区域打开，双击会在新窗口中打开。在 oc2.m 源码中，我们唯一不熟悉的代码就是[pool drain]，它与[pool release]的区别就是在有 GC 机制的内存管理的时候，drain 会触发 gc 回收操作，但是 release 不会，至于什么时候回收，由 gc 决定。

下面我们运行 oc2 工程，点击 Xcode 上方的 Build and Run 按钮，控制台（如果你没有看到控制台，选择菜单栏中的 Run---Console 就可以打开控制台）输出如下内容：



我们看到 Hello, Wolrd!在控制台输出了。你可以尝试让某一行出错，譬如：删除结尾的;，Xcode 会在这一行的前面使用红色的叹号标注，黄色的叹号表示警告信息。

如果你下次打开 Xcode 的时候，没有出现向导对话框，你可以选择菜单栏 File---Open...，在弹出的对话框中，找到你的工程所在的文件夹，进入之后选择*.xcodeproj 文件即可打开指定的工程，如下图所示：



(1.)代码辅助提示：

下面我们改造一下这段代码，如下所示：

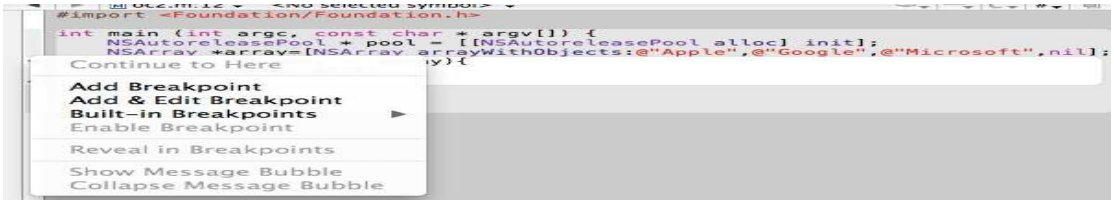
```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSArray *array=[NSArray arrayWithObjects:
                                    @"Apple",@"Google",@"Microsoft",nil];
    for(NSString *str in array){
        NSLog(@"%@",str);
    }
    [pool drain];
    return 0;
}
```

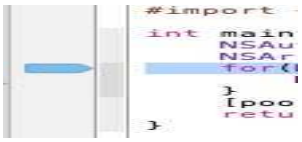
你会看到 Xcode 与 Eclipse 一样，在你输入 NSAr 的时候，就会自动补全为 NSArray，你只需要按下 tab 键确认即可。如果你想看到提示的列表，譬如：你输入了 NS 之后就想看到有哪些类是 NS 开头的或者你输入了 NSArray 之后想看看它有哪些方法，你可以使用 control 与，组合键，注意要在英文输入法下，因为,是半角的。

(2.)调试代码：

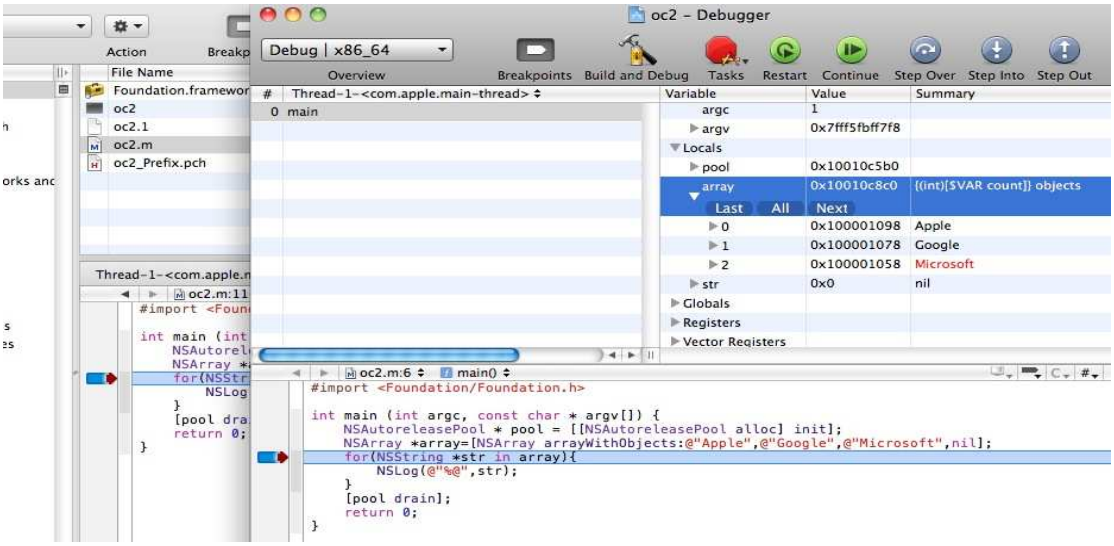
我们在 for 这一行的前面紧挨着的较窄的竖边栏鼠标右键，点击 Add Breakpoint 如下图所示：



你也可以在 for 这一行的前面的第二条较宽的竖边栏单击左键设置断点，如下图所示：



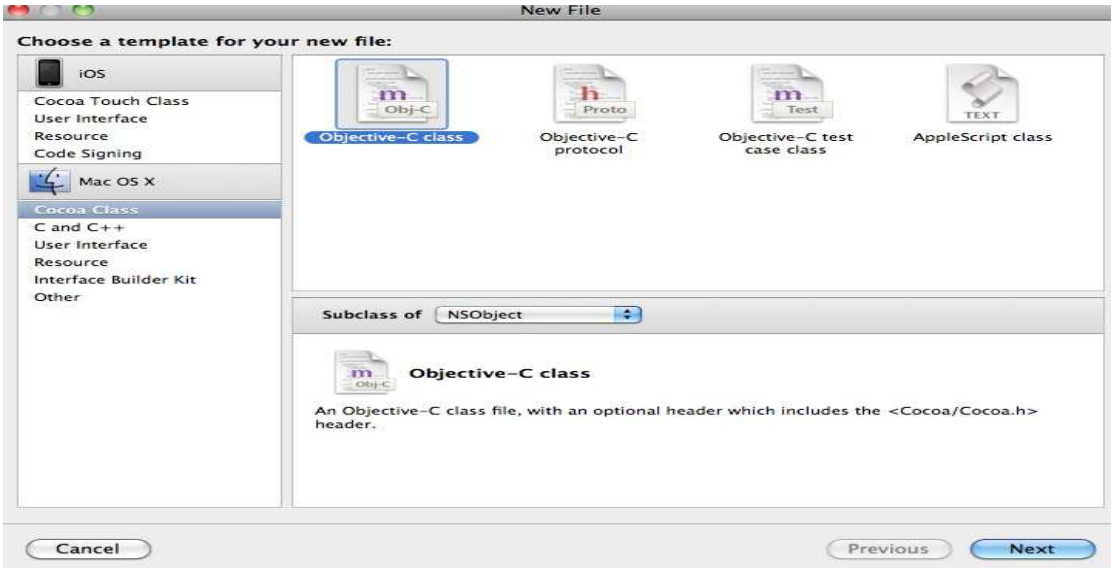
我们点击 Build and Debug 按钮，你会看到程序暂停在了 for 这一行，如果你没有看到调试窗口，可以打开菜单栏中的 Run---Debugger，如下图所示：



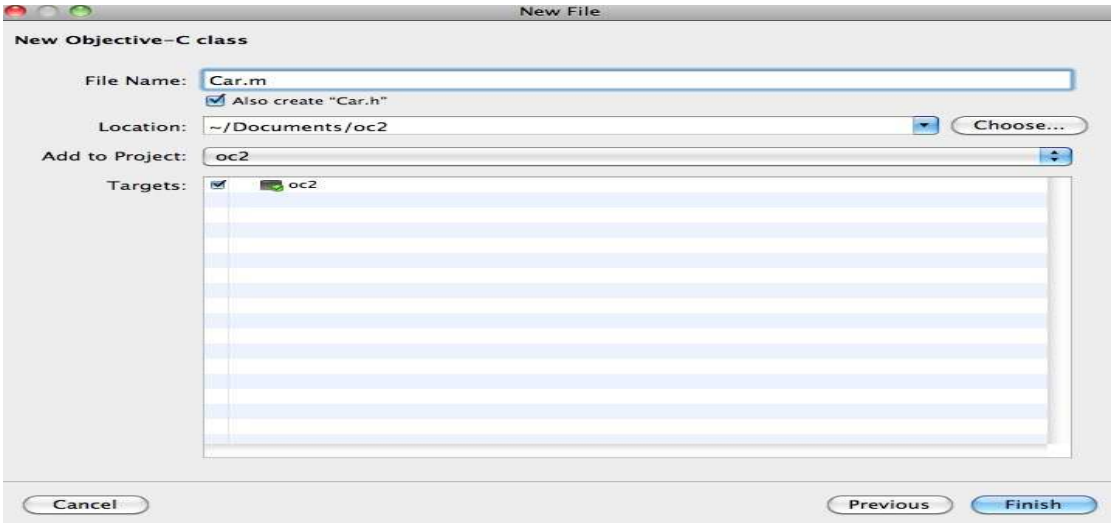
我们看到这个调试窗口与 Ecipse 的布局基本差不多，左侧是当前正在挂起的线程，右侧是变量的情况。单击 Step Over 单步向前，单击 Continue 运行到下一个断点。

(3.)定义公司名称：

下面我们在 Source 文件夹上右键---Add New File...，你会看到如下的对话框：



我们选择 Mac OS X---Objective-C class，点击 Next，如下图所示：



点击 Finish，你就会看到 Car.h、Car.m 被创建了。你会看到每个源码的上方都是自动生成的源码创建时间、创建者（当前用户）、公司名称，其中的公司名称为__myCompanyName__，如何把它改为我们需要的值呢？请右键单击工程 oc2，选择 Get Info，弹出对话框的 General 选项卡如下图所示：

我们修改 Organization Name 为你想要的内容，然后再重新创建 Car.h、Car.m 文件。

(4.)格式化代码：

选中要格式化的代码区域，然后按下右键，选择 Re-intent selection，Xcode 将会缩进你选择的代码。

(5.)使用快照：

如果你打算对工程做一些比较大或者复杂的改动，你需要更改许多的地方，但是你又不确定你改完了是不是会出问题，此时你可以在工程还没有改动之前，处于最佳状态的时候，点击菜单栏的 File---Make Snapshot，为工程制作一个快照，一旦修改出现了问题，可以点击菜单栏 File---Snapshots，然后在弹出的对话框中点击 Restore 直接回滚到快照保存的状态。

(6.)批量修改：

如果你想修改一个变量的名字，而这个名字可能出现在 N 个地方，你可以选中要修改的变量，然后鼠标右键选择 Edit All in Scope，此时变量的名字会被一个矩形圈中，你只需要在这里写入新的名字，所有文件里的这个名字都会变为你修改后的名字。

Eclipse 有重构代码的功能，就是你把一个类的名字改了，它会修改工程中的所有引用的地方。在 Xcode 中，你需要在引用声明这个类的地方（不能在源码文件上选择 Rename），选中类名，然后鼠标右键---Refactor...，你会看到下面的对话框，如下图所示：

我们输入新的名字，点击 Preview 可以预览修改后的情况，然后再点击 Apply 确认修改结果。

(7.)快速查找文件：

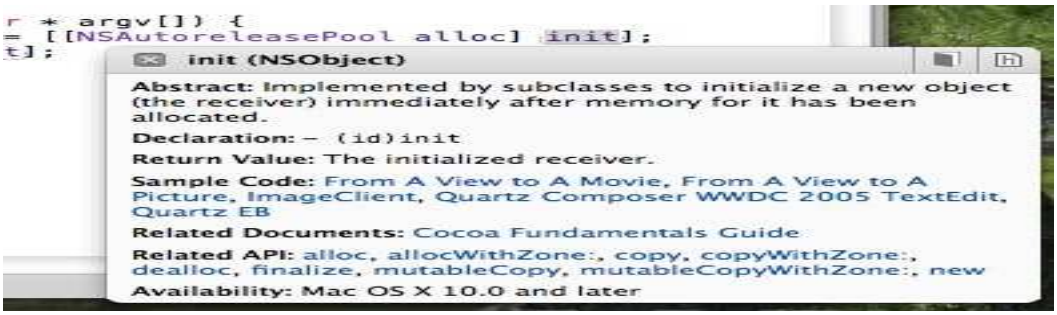
如果想看看一个类型长得什么模样，譬如：import 的 h 文件，你可以选中你要查看的类名，然后按下 Command+Shift+D，就会看到一个搜索对话框。当然，如果你什么都不选择，按下这个快捷键，打开搜索对话框之后，再输入你要查找的类名也可以。

(8.)使用书签：

有一段代码你可能要在稍后作出修改，但是你可能怕一会儿就忘了，你可以选中这段代码，然后按下 Command+D，在弹出的对话框中起个名字，它就会存储到 Xcode 左侧的 Groups and Files 中的 Bookmarks。

(9.)查看 API 文档：

按住 option(alt)键，双击你要查看的方法，就可以打开 API 文档，如下图所示：



(10.)查看类成员：

使用 control 与 2 组合键，可以查看一个类有哪些变量、方法，并选择一个快速定位，这在类文件代码行数很多的时候很方便。

2. Objective-C 2.0 的新特性：

(1.)属性访问器：

第一篇文档中我们都要给成员变量编写样板式的 getter、setter 方法，Objective-C 2.0 中给出了一些新的指令，简化这些代码。我们先看一个示例：

Person.h:

```
@interface Person : NSObject{
        NSString *name;
        float weight;
}
-(Person*) initWithWeight: (int) weight;
```

```
@property (retain,readwrite) NSString* name;
@property (readonly)float weight;
@end
```

这里的@property (parameter1,parameter2)在 h 文件中，自动声明属性，也就是在编译之后的代码中添加成员变量的 setter、getter 方法。如果你希望控制属性是只读的（不生成 setter 方法），请使用 readonly 参数，默认是 readwrite 的。如果属性是对象类型，你需要使用 retain、assign、copy 参数，表示 setter 方法内部实现的时候，持有对象的方式。其中 retain 就是增加引用计数，强引用类型；assign 就是变量的直接赋值，弱引用类型，也是默认值；copy 就是把 setter 的参数复制一份，再赋给成员变量，也就是第一篇文档中的对象复制一节中最后面所说的 setter 的第三种赋值策略。如果你不给出持有对象的方式，编译器会给出警告。另外，@property 的( )中的参数还有一种不常用的 atomic、nonatomic，前者是默认值，表示属性是原子的，支持多线程并发访问（实际就是 setter 的实现中加入了同步锁），后者是非原子的，也就是适合在非多线程的环境提升效率（因为 setter 中没有同步锁的代码）。

Person.m:

```
@implementation Person
-(Person*) initWithWeight: (int) w{
        self=[super init];
        if (self) {
            weight=w;//我们看到这里不会报错，因为我们不是通过setter方法访问的。
        }
        return self;
}
@synthesize name;
@synthesize weight;
-(void) dealloc{
        [self setName:nil];
        [super dealloc];
}
@end
```

这里的@synthesize 表示在 m 文件中，自动实现访问器方法，注意这里不需要再声明类型。另外，dealloc 中的第一行，我们没有像之前一样[name release]，而是使用编译器为我们生成的 setter 方法释放 name 的所有权，这样对于 assign、retain、copy 三种情况都是适用的。

main.m:

```
int main (int argc, const char * argv[]) {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        Person *person=[[Person alloc] initWithWeight:68];
        person.name=@"Jetta";
        //person.weight=68;//此行会报错，因为weight是只读的。
        printf("The person's name is %s, weight is %f!",
            [person.name cStringUsingEncoding:NSUTF8StringEncoding],
            person.weight);
```

```
        [person release];
        [pool drain];
        return 0;

}
```

这里我们看到访问 setter、getter 的方法，居然用 JAVA 中的.语法，这也是 Objective-C 2.0 的新特性，支持用.语法访问属性，自然也不用使用中缀语法的[ ]进行包围。同样是 person.name，它在=左侧就是调用 setter 方法，在=右侧或者作为参数传递的就是调用 getter 方法。

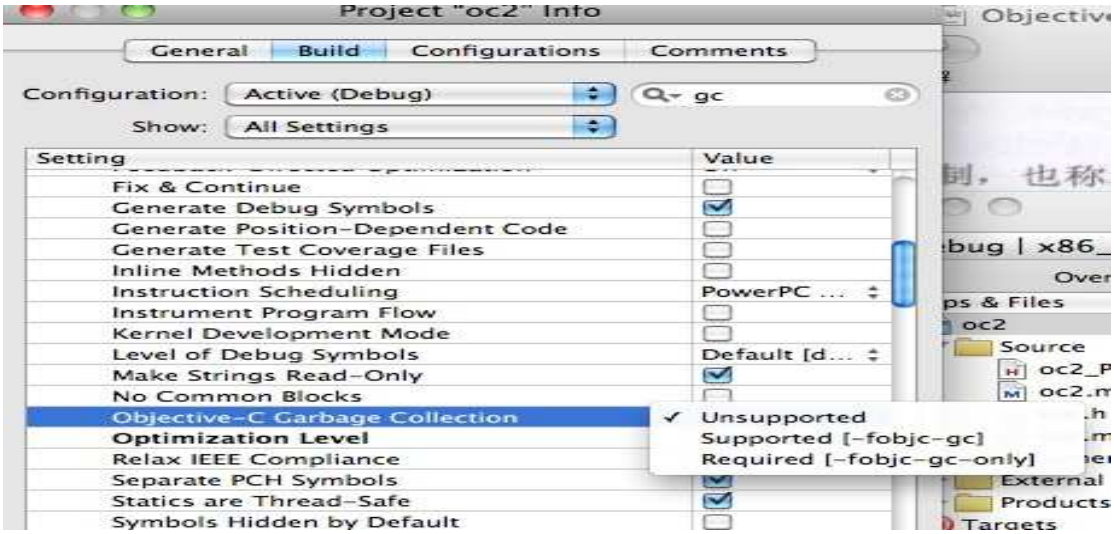
这里还需要注意的是我们使用 cStringUsingEncoding 方法把 NSString 转换为 C 语言的字符串，因为第一篇文档里用的 cString 方法已经过时，不推荐使用。NSUTF8StringEncoding 是一个枚举类型，你可以选中它，然后按下 control+shift+D 查看还有其他哪些枚举值。

(2.)for-each 循环：

第一篇文档已经介绍过，主要用于快速迭代集合类。

(3.)GC 机制：

Xcode 中的工程默认是不启用 gc 机制的，你可以在工程上鼠标右键---Get Info，然后切换到第二个选项卡 Build，在搜索框中输入 gc，然后找到下图中被选中的一行，我们看到是 Unsupported 的，你可以选择为 Supported、Required 启用 gc 机制。在 gc 模式下，我们第一篇文档说的 retain、release、autorelease 都干什么呢？实际上这几个操作在 gc 模式下会被忽略，也就是执行的是空操作。这也就是说你基于手动管理内存的代码，在 gc 机制下不需要做特别的改动。



另外，再重申一下第一篇文档中提到的内容，就是 iOS 平台不支持 GC 机制。

(4.)协议的必选与可选方法：

在 Objective-C 2.0 中增加了如下两个指令，如下所示：

```
@protocol MyProtocol
@required -(void) requiredMethod;
@optional- (void) optionalMethod;
@end
```

协议中的方法声明前使用@required 的表示这个方法必须实现，否则将会得到编译器的警告，

而@optional 为可选实现，为默认值。

3. Objective-C 2.0 的运行时编程：

Objective-C 2.0 的运行时环境叫做 Morden Runtime，iOS 和 Mac OS X 64-bit 的程序都运行在这个环境，也就是说 Mac OS X 32-bit 的程序运行在旧的 Objective-C 1.0 的运行时环境 Legacy Runtime，这里我们只讲解 Morden Runtime。
同运行时交互主要在三个不同的地方，分别是 **A.**Objective-C 源码（譬如：你定义的 Category 中的新方法会在运行时自动添加到原始类）、**B.**NSObject 的方法（isMemberClassOf 等动态判定的方法）、**C.**运行时函数。由于前两者在第一篇文档中讲解过，这里我们讲一下运行时函数的相关内容。

(1.)isa 指针：
NSObject 中有一个 Class isa 的指针类型的成员变量，因为我们的对象大都直接或者间接的从 NSObject 继承而来，因此都会继承这个 isa 成员变量，isa 在运行时会指向对象的 Class 对象，一个类的所有对象的 Class 对象都是同一个（JAVA 也是如此），这保证了在内存中每一个类型都有唯一的类型描述。这个 Class 对象中也有个 isa 指针，它指向了上一级的父类的 Class 对象。
在明白了这个 isa 之后，你就可以明白在继承的时候，A extends B，你调用 A 的方法 a()，首先 A 的 isa 到 A 的 Class 对象中去查找 a()方法，找到了就调用，如果没找到，就驱使 A 的 Class 对象中的 isa 到父类 B 的 Class 对象中去查找。

(2.)SEL 与 IMP：
第一篇文档中，我们提到了方法选择器 SEL，它可以通过如下两种方式获得：
(SEL) @selector(方法的名字)
(SEL) NSSelectorFromString(方法的名字的字符串)
另外，你还可以通过(NSString*) NSStringFromSelector(SEL)函数来获取 SEL 所指定的方法名称字符串。
其实 Objective-C 在编译的时候，会依据每一个定义的方法的名字、参数序列，生成一个唯一的整数标识，这个标识就是 SEL。因此，在运行时查找方法都是通过这个唯一的标识，而不是通过方法的名字。

Objective-C 又提供了 IMP 类型，IMP 表示指向实现方法的指针（函数指针），通过它，你可以直接访问一个实现方法，从而避免了[xxx message]的静态调用方式，需要首先通过 SEL 确定方法，然后再通过 IMP 找到具体的实现方法，最后再发送消息所带来的执行效率问题。一般，如果你在多次循环中反复调用一个方法，用 IMP 的方式，会比直接向对象发送消息高效一些。
例：
Person.m:

```
#import "Person.h"

@implementation Person
@synthesize name;
```

```
@synthesize weight;
-(Person*) initWithWeight: (int) w{
        self=[super init];
        if (self) {
            weight=w;
        }
        return self;
}
-(void) print: (NSString*) str{
        NSLog(@"%@ %@",str,name);
}
-(void) dealloc{
        [self setName:nil];
        [super dealloc];
}
@end
```

main.m:

```
int main (int argc, const char * argv[]) {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        Person *person=[[Person alloc] initWithWeight:68];
        person.name=@"Jetta";
        SEL print_sel=NSSelectorFromString(@"print:");
        IMP imp=[person methodForSelector: print_sel];
        imp(person,print_sel,@"**********");
        [pool drain];
        return 0;
}
```

这里我们看到要获得 IMP 的指针，可以通过 NSObject 中的 methodForSelector: (SEL)方法，访问这个指针函数，我们使用 imp(id,SEL,argument1,… …)，第一个参数是调用方法的对象，第二个方法是方法的选择器对象，第三个参数是可变参数，表示传递方法需要的参数。

(3.)objc_msgSend 函数：

通过 isa 指针的讲解，我们知道 Objective-C 中的方法调用是在运行时才去绑定的，再进一步看，编译器会把对象消息发送[xxx method]转换为 objc_msgSend(id receiver,SEL selector,参数…)的函数调用。因此上面例子中的 print 方法你也可以像下面这样调用：

```
objc_msgSend(person,print_sel,@"++++++++");
```

当然，这是编译器要做的事情，你在写代码的时候，是不需要直接使用这种写法的。

综合 isa、SEL、IMP 的讲解，实际上 objc_msgSend 的调用过程就应该是这样的。

A.首先通过第一个参数的 receiver，找到它的 isa 指针，然后在 isa 指向的 Class 对象中使用第二个参数 selector 查找方法；

B.如果没有找到，就使用当前 Class 对象中的新的 isa 指针到上一级的父类的 Class 对象中查找；

C.当找到方法后，再依据 receiver 的中的 self 指针找到当前的对象，调用当前对象的具体实现的方法（IMP 指针函数），然后传递参数，调用实现方法。
D.假如一直找到 NSObject 的 Class 对象，也没有找到你调用的方法，就会报告不能识别发送消息的错误。

(4.)动态方法解析：
我们在 Objective-C 2.0 的新特性中的属性访问器一节中，实际忽略了一个内容，那就是动态属性。Objective-C 2.0 中增加了@dynamic 指令，表示变量对应的属性访问器方法，是动态实现的，你需要在 NSObject 中继承而来的+(BOOL) resolveInstanceMethod:(SEL) sel 方法中指定动态实现的方法或者函数。
例：
Person.h:

```
@interface Person : NSObject{
        NSString *name;
        float weight;
}
-(Person*) initWithWeight: (int) weight;
@property (retain,readwrite) NSString* name;
@property (readonly)float weight;
@property float height;
-(void) print: (NSString*) str;
@end
```

Person.m:

```
void dynamicMethod(id self,SEL _cmd,float w){
        printf("dynamicMethod-%s\n",[NSStringFromSelector(_cmd)
cStringUsingEncoding:NSUTF8StringEncoding]);
        printf("%f\n",w);
}

@implementation Person
@synthesize name;
@synthesize weight;
@dynamic height;
-(Person*) initWithWeight: (int) w{
        self=[super init];
        if (self) {
            weight=w;
        }
        return self;
}
-(void) print: (NSString*) str{
```

```
    NSLog(@"%@%@",str,name);
}
+(BOOL) resolveInstanceMethod: (SEL) sel{
    NSString *methodName=NSStringFromSelector(sel);
    BOOL result=NO;
    //看看是不是我们要动态实现的方法名称
    if ([methodName isEqualToString:@"setHeight:"]) {
        class_addMethod([self class], sel, (IMP) dynamicMethod,
                                                        "v@:f");
        result=YES;
    }
    return result;
}
-(void) dealloc{
    [self setName:nil];
    [super dealloc];
}
@end
```

这里我们对于接口中的height在实现类中使用了@dynamic指令，紧接着，你需要指定一个函数或者其他类的方法作为height的setter、getter方法的运行时实现。为了简单，我们指定了Person.m中定义的函数（注意这是C语言的函数，不是Objective-C的方法）dynamicMethod作为height的setter方法的运行时实现。被指定为动态实现的方法的dynamicMethod的参数有如下的要求：

A.第一个、第二个参数必须是id、SEL；

B.第三个参数开始，你可以按照原方法（例如：setHeight:(float)）的参数定义。

再接下来，你需要覆盖 NSObject 的类方法 resolveInstanceMethod，这个方法会把需要动态实现的方法（setHeight:）的选择器传递进来，我们判断一下是否是需要动态实现的选择器，如果是就把处理权转交给 dynamicMethod。如何转交呢？这里我们就要用到运行时函数 class_addMethod(Class,SEL,IMP,char[])。

运行时函数位于 objc/runtime.h，正如名字一样，这里面都是 C 语言的函数。按照这些函数的功能的不同，主要分为如下几类：操作类型、操作对象、操作协议等。大多数的函数都可以通过名字看出是什么意思，例如：class_addProtocol 动态的为一个类型在运行时增加协议、objc_getProtocol 把一个字符串转换为协议等。具体这些运行时函数都是做什么用的，你可以参看 Apple 官方页面：

http://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html#//apple_ref/doc/uid/TP40001418

言归正传，我们来解释一下这里需要用到的 class_addmethod 方法，这个方法有四个参数，Class 表示你要为哪个类型增加方法，SEL 参数表示你要增加的方法的选择器，IMP 表示你要添加的方法的运行时的具体实现的函数指针。其实在这里你能够看出 SEL 并不能在运行时找到真正要调用的方法，IMP 才可以真正的找到实现方法的。

在讲解第四个参数 char[]之前，我们先看一下第一篇文档中提到的@encode 指令，在把任意非 Objective-C 对象类型封装为 NSValue 类型的时候使用到了@encode 指令，但当时我们没有详细说明这个指令的含义。实际上@encode()可以接受任何类型，Objective-C 中用这个指

令做类型编码，它可以把任何一个类型转换为字符串，譬如：void 类型被编码之后为 v，对象类型为@，SEL 类型为:等，具体的你可以参看 Apple 官方页面关于 Type Encoding 的描述：http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple_ref/doc/uid/TP40008048-CH100-SW
现在我们来正式的看以下第四个参数 v@:f 的含义，它描述了 IMP 指向的函数的描述信息，按照@encode 指令编译之后的字符说明，第一个字符 v 表示返回值为 void，剩余的字符为 dynamicMethod 函数的参数描述，@表示第一个参数 id，:自然就是第二个参数 SEL，f 就是第三个参数 float。由于前面说过动态方法的实现的前两个参数必须是 id、SEL，所以第四个参数中的字符串的第二、三个字符一定是@:。
我们看到 resolveInstanceMethod 方法的返回值为 BOOL，也就是这个方法返回 YES 表示找到了动态方法的具体实现，否则就表示没有在运行时找到真实的实现，程序就汇报错。

经过了上面的处理，Objective-C 的运行时只要发现你调用了@dynamic 标注的属性的 setter、getter 方法，就会自动到 resolveInstanceMethod 里去寻找真实的实现。这也就是说你在 main.m 中调用 peson.height 的时候，实际上 dynamicMethod 函数被调用了。

实际上除了@dynamic 标注的属性之外，如果你调用了类型中不存在的方法，也会被 resolveInstanceMethod 或者 resolveClassMethod 截获，但由于你没有处理，所以会报告不能识别的消息的错误。

你可能在感叹一个@dynamic 指令用起来真是麻烦，我也是研究了半天 Apple 官方的晦涩的鸟语才搞明白的。不过好在一般 Objective-C 的运行时编程用到的并不多，除非你想设计一个动态化的功能，譬如：从网络下载一个升级包，不需要退出原有的程序，就可以动态的替换掉旧的功能等类似的需求。

(5.)消息转发：
在前面的 objc_msgSend()函数的最后，我们总结了 Objective-C 的方法调用过程，在最后一步我们说如果一路找下来还是没有找到调用的方法，就会报告错误，实际上这里有个细节，那就是最终找不到调用的方法的时候，系统会调用-(void) forwardInvocation: (NSInvocation*) invocation 方法，如果你的对象没有实现这个方法，就调用 NSObject 的 forwardInvocation 方法，那句不能识别消息的错误，实际就是 NSObject 的 forwardInvocation 抛出来的异常。
我们这里告诉你这个系统内部的实现过程，实际是要告诉你，你可以覆盖 forwardInvocation 方法，来改变 NSObject 的抛异常的处理方式。譬如：你可以把 A 不能处理的消息转发给 B 去处理。

NSInvocation 是一个包含了 receiver、selector 的对象，也就是它包含了向一个对象发送消息的所有元素：对象、方法名、参数序列，你可以调用 NSInvocation 的 invoke 方法将这个消息激活。

例：

main.m:

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

```
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        Person *person=[[Person alloc] init];
        person.name=@"Jetta";
        [person fly];
        [person release];
        [pool drain];
        return 0;
}
```

这里我们调用了一个 Person 中不存在的方法 fly。

Bird.m:

```
#import "Bird.h"

@implementation Bird
-(void) fly{
        printf("Bird Can fly!");
}
@end
```

Person.m

```
@implementation Person
@synthesize name;
@synthesize weight;
-(NSMethodSignature*) methodSignatureForSelector:(SEL)selector{
        //首先调用父类的方法
        NSMethodSignature *signature=
                        [super      methodSignatureForSelector: selector];
        //如果当前对象无法回应此selector，那么selector构造的方法签名必然为nil
        if (!signature) {
            //首先判断Bird的实例是否有能力回应此selector
            if ([Bird instancesRespondToSelector:selector]) {
                //获取Bird的selector的方法签名对象
                signature=[Bird instanceMethodSignatureForSelector:
                                selector];
            }
        }
        return signature;
}
-(void) forwardInvocation: (NSInvocation*) invocation{
        //首先验证Bird是否有能力回应invocation中包含的selector
        if ([Bird instancesRespondToSelector:[invocation selector]]) {
            //创建要移交消息响应权的实例bird
```

```
        Bird *bird=[Bird new];
        //激活invocation中的消息，但是消息的响应者是bird，而不是默认的self。
        [invocation invokeWithTarget:bird];
    }
}
-(void) dealloc{
    [self setName:nil];
    [super dealloc];
}
@end
```

下面我们来详细分析一下如果你想把不能处理的消息转发给其他的对象，需要经过哪个几个步骤：

A. 首先，你要覆盖NSObject中的methodSignatureForSelector方法。这是因为你如果想把消息fly从Person转发给Bird处理，那么你必须将NSInvocation中包含的Person的fly的方法签名转换为Bird的fly的方法签名，也就是把方法签名纠正一下。

由此，你也看出来NSInvocation的创建，内部使用了两个对象，一个是receiver，一个是NSMethodSignature，而NSMethodSignature是由SEL创建的。NSInvocation确实存在一个类方法invocationWithMethodSignature返回自身的实例。

B.然后我们覆盖forwardInvocation方法，使用的不是invoke方法，而是invokeWithTarget方法，也就是把调用权由self转交给bird。

实际上消息转发机制不仅可以用来处理找不到方法的错误，你还可以变相的实现多继承。假如我们的 Person 想要拥有 Bird、Fish 的所有功能，其实你可以尽情的用 Person 的实例调用 Bird、Fish 的方法，只要在 Person 的 forwardInvocation 里，把消息的响应权转交给 Bird 或者 Fish 的实例就可以了。不过这种做法实在有点儿 BT，除非万不得已，否则千万不要这么做，但是你也从这里能够看出来 Objective-C 这种语言有多么的灵活、强大，这是 JAVA 所完全不能相比的。