深入浅出 Cocoa 教程

作者：罗朝辉

2012/11/29 整理成册

简介

这是本人在学习和使用 Cocoa 开发过程中写过的一些文章，涵盖 runtime，class，message，KVO，多线程，core data，网络，framework，plugin 等各方面。不仅研究了应该如何使用这些技术，还深入底层探究这些技术是如何实现的，及其 runtime 分析。整体上来说还比较成系列，所以整理出来，希望对大家有帮助。

CSDN 移动开发专栏《深入浅出 Cocoa》包含了这个 pdf 中的所有文章，并会持续添加新的文章，欢迎大家访问该专栏(http://blog.csdn.net/column/details/cocoa.html)，查看最新情况。

文中错误之处难免，欢迎大家指出，指正。

版权声明

本人简介

罗朝辉，中文网名：飘飘白云，英文网名：kesalin，从事移动应用开发及游戏开发相关工作。对移动互联网，游戏领域感兴趣，C/C++/Objective-C 爱好者，也是 3D 图形学爱好者（虽然并不精通），欢迎大家与我交流，一起成长。

与我交流

Email： kesalin@gmail.com
微博： http://weiblo.com/kesalin
CSDN 博客： http://blog.csdn.com/kesalin

目录

[深入浅出 Cocoa] 之类与对象

罗朝辉（http://blog.csdn.com/kesalin/）

最近打算写一些 ObjC 中比较底层的东西，尤其是 runtime 相关的。苹果已经将 ObjC runtime 代码开源了，我们可以从：http://opensource.apple.com/source/objc4/objc4-493.9/runtime/ 浏览源代码，或点此下载源代码。

从哪里入手呢？那当然是最基本的类与对象。与 C++相比，ObjC 中的类与对象结构要简洁与一致得多（参考《深度探索 C++对象模型》，你就知道 C++中类与对象结构的复杂）。本文将详细讲解 ObjC 中类与对象的结构，下回将讲如何在 runtime 时操作类。

我们可以在/usr/include/objc/objc.h 和 runtime.h 中找到对 class 与 object 的定义：

```
typedef struct objc_class *Class;
typedef struct objc_object {
    Class isa;
} *id;
```

Class 是一个 objc_class 结构类型的指针；而 id（任意对象） 是一个 objc_object 结构类型的指针，其第一个成员是一个 objc_class 结构类型的指针。注意这里有一关键的引申解读：内存布局以一个 objc_class 指针为开始的所有东东都可以当做一个 object 来对待！ 那 objc_class 又是怎样一个结构体呢？且看：

```
struct objc_class
{
    struct objc_class* isa;
    struct objc_class* super_class;
    const char* name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list* ivars;
    struct objc_method_list** methodLists;
    struct objc_cache* cache;
    struct objc_protocol_list* protocols;
};
```

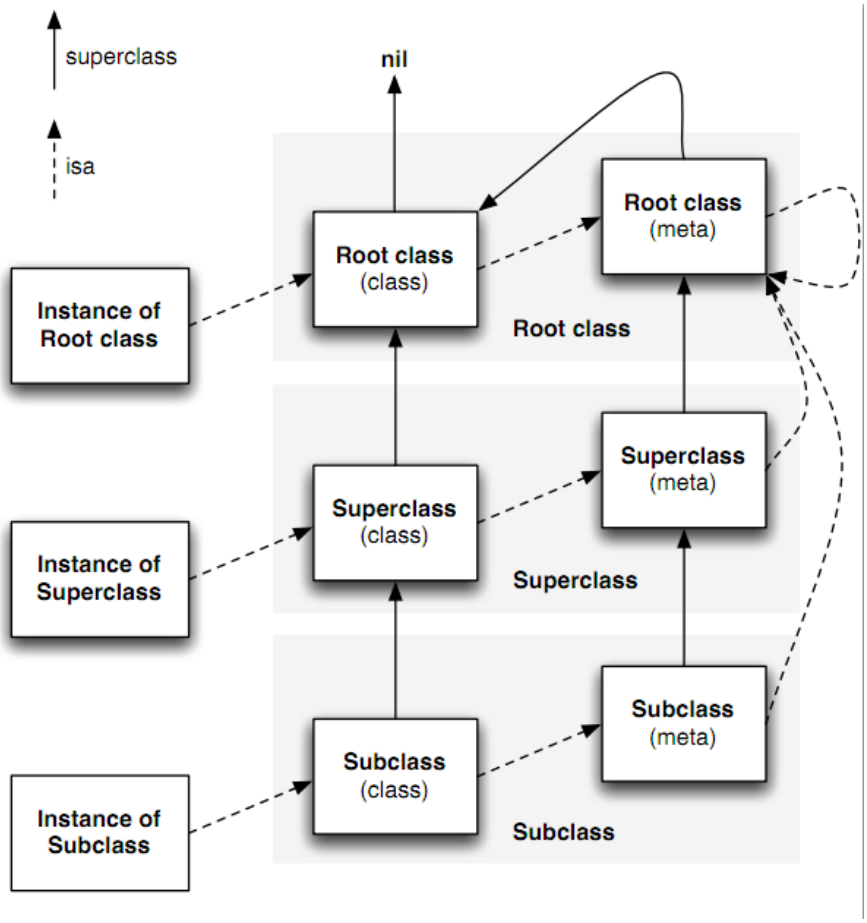
objc_class 结构体的各成员介绍如下：
isa：是一个 objc_class 类型的指针，看到这里，想起我前面的引申解读了没？**内存布局以一个 objc_class 指针为开始的所有东东都可以当做一个 object 来对待！** 这就是说 objc_class 或者说类其实也可以当做一个

objc_object 对象来对待！对象是对象，类也是对象，是不是有点混淆？别急，ObjC 发明（or 重用）了一个术语来区分这两种不同的对象：类对象（class object）与实例对象（instance object）。OK，名称混淆的问题解决，下面我将使用这两个术语来区分不同的对象，而使用"对象"这一术语来泛指所有的对象。ObjC 还对类对象与实例对象中的 isa 所指向的类结构作了不同的命名：类对象中的 isa 指向类结构被称作 metaclass，metaclass 存储类的 static 类成员变量与 static 类成员方法（+开头的方法）；实例对象中的 isa 指向类结构称作 class（普通的），class 结构存储类的普通成员变量与普通成员方法（-开头的方法）。

super_class：一看就明白，指向该类的父类呗！如果该类已经是最顶层的根类（如 NSObject 或 NSProxy），那么 super_class 就为 NULL。

好，先中断一下其他类结构成员的介绍，让我们厘清一下在继承层次中，子类，父类，根类（这些都是普通 class）以及其对应的 metaclass 的 isa 与 super_class 之间关系：
规则一：类的实例对象的 isa 指向该类；该类的 isa 指向该类的 metaclass；
规则二：类的 super_class 指向其父类，如果该类为根类则值为 NULL；
规则三：metaclass 的 isa 指向根 metaclass，如果该 metaclass 是根 metaclass 则指向自身；
规则四：metaclass 的 super_class 指向父 metaclass，如果该 metaclass 是根 metaclass 则指向该 metaclass 对应的类；

好吧，文字总是那么乏力，有图有真相！



<instance object，class，metaclass 的 isa 与 super_class 关系图>

那么 class 与 metaclass 有什么区别呢？

class 是 instance object 的类类型。当我们向实例对象发送消息（实例方法）时，我们在该实例对象的 class 结构的 methodlists 中去查找响应的函数，如果没找到匹配的响应函数则在该 class 的父类中的 methodlists 去查找（查找链为上图的中间那一排）。如下面的代码中，向 str 实例对象发送 lowercaseString 消息，会在 NSString 类结构的 methodlists 中去查找 lowercaseString 的响应函数。

```
NSString * str;
[str lowercaseString];
```

metaclass 是 class object 的类类型。当我们向类对象发送消息（类方法）时，我们在该类对象的 metaclass 结构的 methodlists 中去查找响应的函数，如果没有找到匹配的响应函数则在该 metaclass 的父类中的 methodlists 去查找（查找链为上图的最右边那一排）。如下面的代码中，向 NSString 类对象发送 stringWithString 消息，会在 NSString 的 metaclass 类结构的 methodlists 中去查找 stringWithString 的响应函数。

```
[NSString stringWithString:@"str"];
```

好，至此我们明白了类的结构层次，让我们接着看类结构中的其他成员。

name：一个 C 字符串，指示类的名称。我们可以在运行期，通过这个名称查找到该类（通过：id objc_getClass(const char *aClassName)）或该类的 metaclass（id objc_getMetaClass(const char *aClassName)）；

version：类的版本信息，默认初始化为 0。我们可以在运行期对其进行修改（class_setVersion）或获取（class_getVersion）。

info：供运行期使用的一些位标识。有如下一些位掩码：
CLS_CLASS (0x1L) 表示该类为普通 class ，其中包含实例方法和变量；
CLS_META (0x2L) 表示该类为 metaclass，其中包含类方法；
CLS_INITIALIZED (0x4L) 表示该类已经被运行期初始化了，这个标识位只被 objc_addClass 所设置；
CLS_POSING (0x8L) 表示该类被 pose 成其他的类；（poseclass 在 ObjC 2.0 中被废弃了）；
CLS_MAPPED (0x10L) 为 ObjC 运行期所使用
CLS_FLUSH_CACHE (0x20L) 为 ObjC 运行期所使用
CLS_GROW_CACHE (0x40L) 为 ObjC 运行期所使用
CLS_NEED_BIND (0x80L) 为 ObjC 运行期所使用
CLS_METHOD_ARRAY (0x100L) 该标志位指示 methodlists 是指向一个 objc_method_list 还是一个包含 objc_method_list 指针的数组；

instance_size：该类的实例变量大小（包括从父类继承下来的实例变量）；

ivars：指向 objc_ivar_list 的指针，存储每个实例变量的内存地址，如果该类没有任何实例变量则为 NULL；

methodLists：与 info 的一些标志位有关，CLS_METHOD_ARRAY 标识位决定其指向的东西（是指向单个 objc_method_list 还是一个 objc_method_list 指针数组），如果 info 设置了 CLS_CLASS 则 objc_method_list 存储实例方法，如果设置的是 CLS_META 则存储类方法；

cache：指向 objc_cache 的指针，用来缓存最近使用的方法，以提高效率；

protocols：指向 objc_protocol_list 的指针，存储该类声明要遵守的正式协议。

总结：

ObjC 为每个类的定义生成两个 objc_class ，一个即普通的 class，另一个即 metaclass。我们可以在运行期创建这两个 objc_class 数据结构，然后使用 objc_addClass 动态地创建新的类定义。这个够动态够强大的吧？下回讲演示如何在运行期动态创建新类。

[深入浅出 Cocoa] 之动态创建类

罗朝辉 (http://blog.csdn.net/kesalin/)

在前文《深入浅出 Cocoa 之类与对象》一文中，我已经详细介绍了 ObjC 中的 Class 与 Object 的概念，今天我们来如何在运行
时
动态创建类。下面这个函数就是应用前面讲到的 Class，MetaClass 的概念，在运行时动态创建一个类。这个函数来自《Inside Mac OS X-The Objective-C Programming Language》。

```
#import <objc/objc.h>
#import <objc/runtime.h>

BOOL CreateClassDefinition( const char * name, const char * superclassName)
{
    struct objc_class * meta_class;
    struct objc_class * super_class;
    struct objc_class * new_class;
    struct objc_class * root_class;
    va_list args;

    // 确保父类存在
    super_class = (struct objc_class *)objc_lookUpClass (superclassName);
    if (super_class == nil)
    {
        return NO;
    }

    // 确保要创建的类不存在
    if (objc_lookUpClass (name) != nil)
    {
        return NO;
    }

    // 查找 root class，因为 meta class 的 isa 指向 root class 的 meta class
    root_class = super_class;
    while( root_class->super_class != nil )
    {
        root_class = root_class->super_class;
    }
```

```
    // 为 class 及其 meta class 分配内存
    new_class = calloc( 2, sizeof(struct objc_class) );
    meta_class = &new_class[1];

    // 设置 class
    new_class->isa = meta_class;
    new_class->info = CLS_CLASS;
    meta_class->info = CLS_META;

    // 拷贝类名字，这里为了提高效率，让 class 与 meta class 都指向同一个类名字符串
    new_class->name = malloc (strlen (name) + 1);
    strcpy ((char*)new_class->name, name);
    meta_class->name = new_class->name;

    // 分配并置空 method lists，我们可以在之后使用
class_addMethods 向类中增加方法

    new_class->methodLists = calloc( 1, sizeof(struct objc_method_list *) );
    meta_class->methodLists = calloc( 1, sizeof(struct objc_method_list *) );

    // 将类加入到继承体系中去：
    // 1，设置类的 super class
    // 2，设置 meta class 的 super class
    // 3，设置 meta class 的 isa
     new_class->super_class = super_class;
     meta_class->super_class = super_class->isa;
    meta_class->isa = (void *)root_class->isa;

    // 最后，将 class 注册到运行时系统中
    objc_addClass( new_class );

    return YES;
}
```

如果要在代码中使用运行时相关的函数，我们需要导入 libobjc.dylib，并导入相关的头文件（比如这里的 runtime.h）。

在前文中总结到"ObjC 为每个类的定义生成两个 objc_class ，一个即普通的 class，另一个即 metaclass。我们可以在运行期创建这两个 objc_class 数据结构，然后使用 objc_addClass 动态地创建新的类定义。"，这在上面的代码中就体现出来了:new_class 和 meta_class 就是新类所必须的两个 objc_class。其他的代码就不解释了，注释以及代码足以自明了。

在实际的运用中，我们使用 ObjC 运行时函数来动态创建类：

```
Class objc_allocateClassPair(Class superclass, const char *name, size_t extraBytes);
```

譬如：

```
#import <objc/objc.h>
#import <objc/runtime.h>

void ReportFunction(id self, SEL _cmd)
{
    NSLog(@" >> This object is %p.", self);
    NSLog(@" >> Class is %@, and super is %@.", [self class], [self superclass]);

    Class prevClass = NULL;
    int count = 1;
    for (Class currentClass = [self class]; currentClass; ++count)
    {
        prevClass = currentClass;

        NSLog(@" >> Following the isa pointer %d times gives %p", count, currentClass);

        currentClass = object_getClass(currentClass);
        if (prevClass == currentClass)
            break;
    }

    NSLog(@" >> NSObject's class is %p", [NSObject class]);
    NSLog(@" >> NSObject's meta class is %p", object_getClass([NSObject class]));
}

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

    @autoreleasepool
    {
        Class newClass = objc_allocateClassPair([NSString class], "NSStringSubclass", 0);
        class_addMethod(newClass, @selector(report), (IMP)ReportFunction, "v@:");
        objc_registerClassPair(newClass);

        id instanceOfNewClass = [[newClass alloc] init];
        [instanceOfNewClass performSelector:@selector(report)];
        [instanceOfNewClass release];
    }

    return 0;
}
```

在上面的代码中，我们创建继承自 NSString 的子类 NSStringSubclass，然后向其中添加方法 report，

并在运行时系统中注册，这样我们就可以使用这个新类了。在这里使用 performSelector 来向新类的对象发送消息，可以避免编译警告信息（因为我们并没有声明该类及其可响应的消息）。

执行结果为：

```
>> This object is 0x100114710.
>> Class is NSStringSubclass, and super is NSString.
>> Following the isa pointer 1 times gives 0x100114410
>> Following the isa pointer 2 times gives 0x100114560
>> Following the isa pointer 3 times gives 0x7fff7e257b50
>> NSObject's class is 0x7fff7e257b78
>> NSObject's meta class is 0x7fff7e257b50
```

根据前文中的类关系图，我们不难从执行结果中分析出 NSStringSubclass 的内部类结构：

1，对象的地址为 ：0x100114710

2，class 的地址为：0x100114410

3，meta class 的地址为：0x100114560

4，meta class 的 class 地址为：0x7fff7e257b50 （也是 NSObject 的 meta class）

5，NSObject 的 meta class 的 meta class 是其自身

[深入浅出 Cocoa] 之消息

罗朝辉（http://blog.csdn.net/kesalin）

在入门级别的 ObjC 教程中，我们常对从 C++或 Java 或其他面向对象语言转过来的程序员说，ObjC 中的方法调用（ObjC 中的术语为消息）跟其他语言中的方法调用差不多，只是形式有些不同而已。

譬如 C++ 中的：

```
Bird * aBird = new Bird();
aBird->fly();
```

在 ObjC 中则如下：

```
Bird * aBird = [[Bird alloc] init];
[aBird fly];
```

初看起来，好像只是书写形式不同而已，实则差异大矣。C++中的方法调用可能是动态的，也可能是静态的；而 ObjC 中的消息都为动态的。下文将详细介绍为什么是动态的，以及编译器在这背后做了些什么事情。

要说清楚消息这个话题，我们必须先来了解三个概念 Class, SEL, IMP，它们在 objc/objc.h 中定义：

```
typedef struct objc_class *Class;
typedef struct objc_object {
    Class isa;
} *id;

typedef struct objc_selector    *SEL;
typedef id (*IMP)(id, SEL, ...);
```

Class 的含义

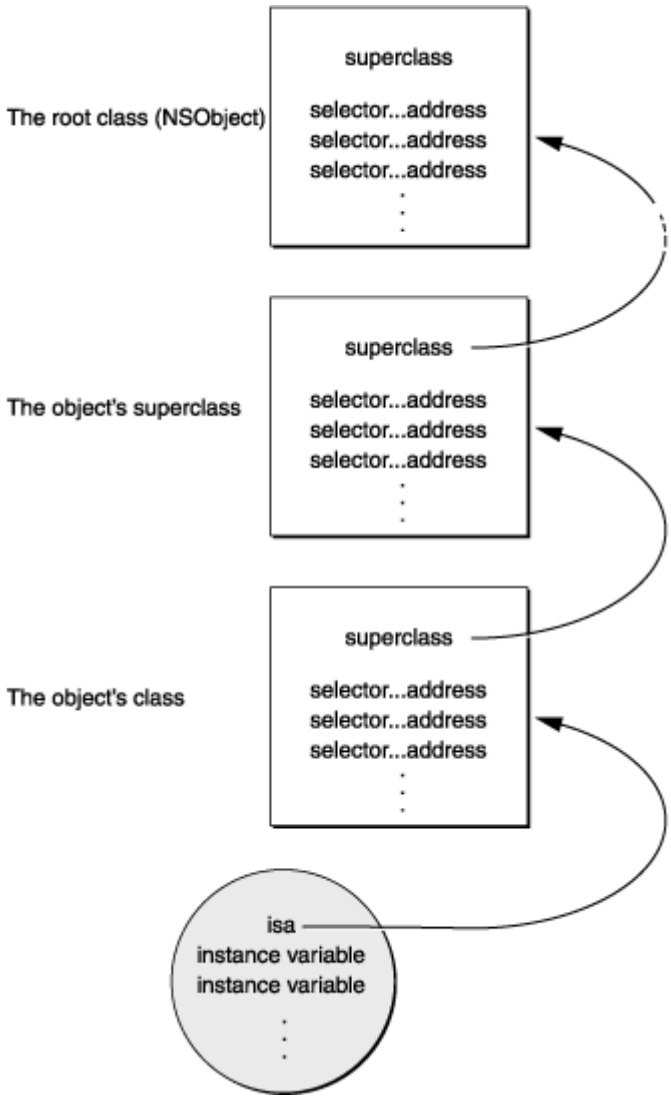
Class 被定义为一个指向 objc_class 的结构体指针，这个结构体表示每一个类的类结构。而 objc_class 在 objc/objc_class.h 中定义如下:

```
struct objc_class {
    struct objc_class super_class;  /*父类*/
    const char *name;               /*类名字*/
    long version;                   /*版本信息*/
```

```
    long info;                  /*类信息*/
    long instance_size;         /*实例大小*/
    struct objc_ivar_list *ivars;    /*实例参数链表*/
    struct objc_method_list **methodLists;  /*方法链表*/
    struct objc_cache *cache;           /*方法缓存*/
    struct objc_protocol_list *protocols;   /*协议链表*/
};
```

由此可见，Class 是指向类结构体的指针，该类结构体含有一个指向其父类类结构的指针，该类方法的链表，该类方法的缓存以及其他必要信息。

NSObject 的 class 方法就返回这样一个指向其类结构的指针。每一个类实例对象的第一个实例变量是一个指向该对象的类结构的指针，叫做 isa。通过该指针，对象可以访问它对应的类以及相应的父类。如图一所示：



如图一所示，圆形所代表的实例对象的第一个实例变量为 isa，它指向该类的类结构 The object's class。而该类结构有一个指向其父类类结构的指针 superclass， 以及自身消息名称(selector)/实现地址(address)的方法链表。

方法的含义：

注意这里所说的方法链表里面存储的是 Method 类型的。图一中 selector 就是指 Method 的 SEL， address 就是指 Method 的 IMP。 Method 在头文件 objc_class.h 中定义如下：

```
typedef struct objc_method *Method;

typedef struct objc_ method {
    SEL method_name;
    char *method_types;
    IMP method_imp;
};
```

一个方法 Method，其包含一个方法选标 SEL – 表示该方法的名称，一个 types – 表示该方法参数的类型，一个 IMP - 指向该方法的具体实现的函数指针。

SEL 的含义：

在前面我们看到方法选标 SEL 的定义为：

typedef struct objc_selector *SEL;

它是一个指向 objc_selector 指针，表示方法的名字/签名。如下所示，打印出 selector。

```
-(NSInteger)maxIn:(NSInteger)a theOther:(NSInteger)b
{
    return (a > b) ? a : b;
}

NSLog(@"SEL=%s", @selector(maxIn:theOther:));
```

输出：SEL=maxIn:theOther:

不同的类可以拥有相同的 selector，这个没有问题，因为不同类的实例对象 performSelector 相同的 selector 时，会在各自的消息选标(selector)/实现地址(address) 方法链表中根据 selector 去查找具体的方法实现 IMP，然后用这个方法实现去执行具体的实现代码。这是一个动态绑定的过程，在编译的时候，我们不知道最终会执行哪一些代码，只有在执行的时候，通过 selector 去查询，我们才能确定具体的执行代码。

IMP 的含义：

在前面我们也看到 IMP 的定义为：

typedef id (*IMP)(id, SEL, ...);

根据前面 id 的定义，我们知道 id 是一个指向 objc_object 结构体的指针，该结构体只有一个成员 isa，所以任何继承自 NSObject 的类对象都可以用 id 来指代，因为 NSObject 的第一个成员实例就是 isa。

至此，我们就很清楚地知道 IMP 的含义：IMP 是一个函数指针，这个被指向的函数包含一个接收消息的对象 id(self 指针)，调用方法的选标 SEL (方法名)，以及不定个数的方法参数，并返回一个 id。也就是说 IMP 是消息最终调用的执行代码，是方法真正的实现代码 。我们可以像在 C 语言里面一样使用这个函数指针。

NSObject 类中的 methodForSelector：方法就是这样一个获取指向方法实现 IMP 的指针，methodForSelector：返回的指针和赋值的变量类型必须完全一致，包括方法的参数类型和返回值类型。

下面的例子展示了怎么使用指针来调用 setFilled:的方法实现：

```
void (*setter)(id, SEL, BOOL);
int i;

setter = (void(*)(id, SEL, BOOL))[target methodForSelector:@selector(setFilled:)];

for (i = 0; i < 1000; i++)
    setter(targetList[i], @selector(setFilled:), YES);
```

使用 methodForSelector：来避免动态绑定将减少大部分消息的开销，但是这只有在指定的消息被重复发送很多次时才有意义，例如上面的 for 循环。

注意，methodForSelector：是 Cocoa 运行时系统的提供的功能，而不是 Objective-C 语言本身的功能。

消息调用过程：

至此我们对 ObjC 中的消息应该有个大致思路了：

示例

```
Bird * aBird = [[Bird alloc] init];
[aBird fly];
```

中对 fly 的调用，编译器通过插入一些代码，将之转换为对方法具体实现 IMP 的调用，这个 IMP 是通过在 Bird 的类结构中的方法链表中查找名称为 fly 的 选标 SEL 对应的具体方法实现找到的。

上面的思路还有一些没有提及的话题，比如说编译器插入了什么代码，如果在方法链表中没有找到对应的 IMP 又会如何，这些话题在下面展开。

消息函数 obj_msgSend：

编译器会将消息转换为对消息函数 objc_msgSend 的调用，该函数有两个主要的参数：消息接收者 id 和消息对应的方法选标 SEL，同时接收消息中的任意参数：

id objc_msgSend(id theReceiver, SELtheSelector, ...)

如上面的消息 [aBird fly]会被转换为如下形式的函数调用：

objc_msgSend(aBird, @selector(fly));

该消息函数做了动态绑定所需要的一切工作：

1，它首先找到 SEL 对应的方法实现 IMP。因为不同的类对同一方法可能会有不同的实现，所以找到的方法实现依赖于消息接收者的类型。

2， 然后将消息接收者对象(指向消息接收者对象的指针)以及方法中指定的参数传递给方法实现 IMP。

3， 最后，将方法实现的返回值作为该函数的返回值返回。

编译器会自动插入调用该消息函数 objc_msgSend 的代码，我们无须在代码中显示调用该消息函数。当 objc_msgSend 找到方法对应的实现时，它将直接调用该方法实现，并将消息中所有的参数都传递给方法实现，同时，它还将传递两个隐藏的参数：消息的接收者以及方法名称 SEL。这些参数帮助方法实现获得了消息表达式的信息。它们被认为是”隐藏“的是因为它们并没有在定义方法的源代码中声明，而是在代码编译时是插入方法的实现中的。

尽管这些参数没有被显示声明，但在源代码中仍然可以引用它们（就象可以引用消息接收者对象的实例变量一样）。在方法中可以通过 self 来引用消息接收者对象，通过选标_cmd 来引用方法本身。在下面的例子中，_cmd 指的是 strange 方法，self 指的收到 strange 消息的对象。

```
- strange
{
    id target = getTheReceiver();
    SEL method = getTheMethod();

    if (target == self || mothod == _cmd)
        return nil;

    return [target performSelector:method];
}
```

在这两个参数中，self 更有用一些。实际上，它是在方法实现中访问消息接收者对象的实例变量的途径。

查找 IMP 的过程：

前面说了，objc_msgSend 会根据方法选标 SEL 在类结构的方法列表中查找方法实现 IMP。这里头有一些文章，我们在前面的类结构中也看到有一个叫 objc_cache *cache 的成员，这个缓存为提高效率而存在的。每个类都有一个独立的缓存，同时包括继承的方法和在该类中定义的方法。。

下面来剖析一段苹果官方运行时源码：

```
static Method look_up_method(Class cls, SEL sel,
                             BOOL withCache, BOOL withResolver)
{
    Method meth = NULL;

    if (withCache) {
        meth = _cache_getMethod(cls, sel, &_objc_msgForward_internal);
        if (meth == (Method)1) {
            // Cache contains forward:: . Stop searching.
            return NULL;
        }
    }

    if (!meth) meth = _class_getMethod(cls, sel);

    if (!meth  &&  withResolver) meth = _class_resolveMethod(cls, sel);

    return meth;
}
```

通过分析上面的代码，可以看到，查找时：

1，首先去该类的方法 cache 中查找，如果找到了就返回它；

2，如果没有找到，就去该类的方法列表中查找。如果在该类的方法列表中找到了，则将 IMP 返回，并将它加入 cache 中缓存起来。根据最近使用原则，这个方法再次调用的可能性很大，缓存起来可以节省下次调用再次查找的开销。3，3，如果在该类的方法列表中没找到对应的 IMP，在通过该类结构中的 super_class 指针在其父类结构的方法列表中去查找，直到在某个父类的方法列表中找到对应的 IMP，返回它，并加入 cache 中；

4，如果在自身以及所有父类的方法列表中都没有找到对应的 IMP，则看是不是可以进行动态方法决议（后面有专文讲述这个话题）；

5，如果动态方法决议没能解决问题，进入下面要讲的消息转发流程。

便利函数：

我们可以通过 NSObject 的一些方法获取运行时信息或动态执行一些消息：

class　返回对象的类；

isKindOfClass 和 isMemberOfClass 检查对象是否在指定的类继承体系中；

respondsToSelector 检查对象能否相应指定的消息；

conformsToProtocol 检查对象是否实现了指定协议类的方法；

methodForSelector 返回指定方法实现的地址。

performSelector:withObject 执行 SEL 所指代的方法。

消息转发：

通常，给一个对象发送它不能处理的消息会得到出错提示，然而，Objective-C 运行时系统在抛出错误之前，会给消息接收对象发送一条特别的消息 forwardInvocation 来通知该对象，该消息的唯一参数是个 NSInvocation 类型的对象——该对象封装了原始的消息和消息的参数。

我们可以实现 forwardInvocation:方法来对不能处理的消息做一些默认的处理，也可以将消息转发给其他对象来处理，而不抛出错误。

关于消息转发的作用，可以考虑如下情景：假设，我们需要设计一个能够响应 negotiate 消息的对象，并且能够包括其它类型的对象对消息的响应。 通过在 negotiate 方法的实现中将 negotiate 消息转发给其它的对象来很容易的达到这一目的。

更进一步，假设我们希望我们的对象和另外一个类的对象对 negotiate 的消息的响应完全一致。一种可能的方式就是让我们的类继承其它类的方法实现。 然后，有时候这种方式不可行，因为我们的类和其它类可能需要在不同的继承体系中响应 negotiate 消息。

虽然我们的类无法继承其它类的 negotiate 方法，但我们仍然可以提供一个方法实现，这个方法实现只是简单的将 negotiate 消息转发给其他类的对象，就好像从其它类那儿“借”来的现一样。如下所示：

```
- negotiate
{
    if ([someOtherObject respondsToSelector:@selector(negotiate)])
        return [someOtherObject negotiate];

    return self;
}
```

这种方式显得有欠灵活，特别是有很多消息都希望传递给其它对象时，我们就必须为每一种消息提供方法实现。此外，这种方式不能处理未知的消息。当我们写下代码时，所有我们需要转发的消息的集合都必须确定。然而，实际上，这个集合会随着运行时事件的发生，新方法或者新类的定义而变化。

forwardInvocation:消息给这个问题提供了一个更特别的，动态的解决方案：当一个对象由于没有相应的方法实现而无法响应某消息时，运行时系统将通过 forwardInvocation:消息通知该对象。每个对象都从 NSObject 类中继承了 forwardInvocation:方法。然而，NSObject 中的方法实现只是简单地调用了 doesNotRecognizeSelector:。通过实现我们自己的 forwardInvocation:方法，我们可以在该方法实现中将消息转发给其它对象。

要转发消息给其它对象，forwardInvocation:方法所必须做的有：

1，决定将消息转发给谁，并且

2，将消息和原来的参数一块转发出去。

消息可以通过 invokeWithTarget:方法来转发：

```
- (void) forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:[anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];

    else
        [super forwardInvocation:anInvocation];
}
```

转发消息后的返回值将返回给原来的消息发送者。您可以将返回任何类型的返回值，包括: id，结构体，浮点数等。

forwardInvocation:方法就像一个不能识别的消息的分发中心，将这些消息转发给不同接收对象。或者它也可以象一个运输站将所有的消息都发送给同一个接收对象。它可以将一个消息翻译成另外一个消息，或者简单的"吃掉"某些消息，因此没有响应也没有错误。forwardInvocation:方法也可以对不同的消息提供同样的响应，这一切都取决于方法的具体实现。该方法所提供是将不同的对象链接到消息链的能力。

注意： forwardInvocation:方法只有在消息接收对象中无法正常响应消息时才会被调用。 所以，如果我们希望一个对象将 negotiate 消息转发给其它对象，则这个对象不能有 negotiate 方法，也不能在动态方法决议过程中为之提供实现。否则，forwardInvocation:将不可能会被调用。

参考资料：

Objective-CRuntime Reference:

http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html

Objective-C Runtime Programming Guide:

http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html

[深入浅出 Cocoa] 之消息（二）-详解动态方法决议(Dynamic Method Resolution)

罗朝辉 (http://blog.csdn.net/kesalin/)

序言

如果我们在 Objective C 中向一个对象发送它无法处理的消息，会出现什么情况呢？根据前文《深入浅出 Cocoa 之消息》的介绍，我们知道发送消息是通过 objc_send(id, SEL, ...) 来实现的，它会首先在对象的类对象的 cache，method list 以及父类对象的 cache, method list 中依次查找 SEL 对应的 IMP；如果没有找到且实现了动态方法决议机制就会进行决议，如果没有实现动态方法决议机制或决议失败且实现了消息转发机制就会进入消息转发流程，否则程序 crash。也就是说如果同时提供了动态方法决议和消息转发，那么动态方法决议先于消息转发，只有当动态方法决议依然无法正确决议 selector 的实现，才会尝试进行消息转发。在前文中，我并没有详细讲解动态方法决议，因此本文将详细介绍之。

本文代码下载：点此下载

一，向一个对象发送该对象无法处理的消息

如下代码：

```
@interface Foo : NSObject

-(void)Bar;

@end

@implementation Foo

-(void)Bar
{
    NSLog(@" >> Bar() in Foo");
}

@end
```

```
//////////////////////////////////////////////
#import "Foo.h"


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


    @autoreleasepool {


        Foo * foo = [[Foo alloc] init];


        [foo Bar];


        [foo MissMethod];


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

在编译时，XCode 会提示警告：

```
Instance method '-MissMethod' not found (return type defaults to 'id')
```

如果，我们忽视该警告运行之，一定会 crash：

```
>> Bar() in Foo
-[Foo MissMethod]: unrecognized selector sent to instance 0x10010c840
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[Foo MissMethod]: unrecognized
selector sent to instance 0x10010c840'
*** Call stack at first throw:
......
terminate called after throwing an instance of 'NSException'
```

下划线部分就是造成 crash 的原因：对象无法处理 MissMethod 对应的 selector，也就是没有相应的实现。

二，动态方法决议

Objective C 提供了一种名为动态方法决议的手段，使得我们可以在运行时动态地为一个 selector 提供实现。我们只要实现 +resolveInstanceMethod: 和/或 +resolveClassMethod: 方法，并在其中为指定的 selector 提供实现即可（通过调用运行时函数 class_addMethod 来添加）。这两个方法都是 NSObject 中的类方法，其原型为：

```
+ (BOOL)resolveClassMethod:(SEL)name;

+ (BOOL)resolveInstanceMethod:(SEL)name;
```

参数 name 是需要被动态决议的 selector；返回值文档中说是表示动态决议成功与否。但在上面的例子中（**不涉及消息转发的情况下**），如果在该函数内为指定的 selector 提供实现，无论返回 YES 还是 NO，编译运行都是正确的；但如果在该函数内并不真正为 selector 提供实现，无论返回 YES 还是 NO，运行都会 crash，道理很简单，selector 并没有对应的实现，而又没有实现消息转发。
resolveInstanceMethod 是为对象方法进行决议，而 resolveClassMethod 是为类方法进行决议。

下面我们用动态方法决议手段来修改上面的代码：

```
//

//  Foo.m

//  DeepIntoMethod

//

//  Created by 飘飘白云 on 12-11-13.

//  Copyright (c) 2012年 kesalin@gmail.com All rights reserved.

//


#import "Foo.h"

#include <objc/runtime.h>


void dynamicMethodIMP(id self, SEL _cmd) {

    NSLog(@" >> dynamicMethodIMP");

}


@implementation Foo


-(void)Bar

{

    NSLog(@" >> Bar() in Foo");

}
```

```
+ (BOOL)resolveInstanceMethod:(SEL)name
{
    NSLog(@" >> Instance resolving %@", NSStringFromSelector(name));

    if (name == @selector(MissMethod)) {
        class_addMethod([self class], name, (IMP)dynamicMethodIMP, "v@:");
        return YES;
    }

    return [super resolveInstanceMethod:name];
}

+ (BOOL)resolveClassMethod:(SEL)name
{
    NSLog(@" >> Class resolving %@", NSStringFromSelector(name));

    return [super resolveClassMethod:name];
}

@end
```

在前文《深入浅出 Cocoa 之消息》中已经介绍过 Objective C 中的方法其实就是至少带有两个参数(self 和 _cmd)的普通 C 函数,因此在上面的代码中提供这样一个 C 函数 dynamicMethodIMP,让它来充当对象方法 MissMethod 这个 selector 的动态实现。因为 MissMethod 是被对象所调用,所以它被认为是一个对象方法,因而应该在 resolveInstanceMethod 方法中为其提供实现。通过调用

```
class_addMethod([self class], name, (IMP)dynamicMethodIMP, "v@:");
```

就能在运行期动态地为 name 这个 selector 添加实现:dynamicMethodIMP。class_addMethod 是运行时函数,所以需要导入头文件:objc/runtime.h。

再次编译运行前面的测试代码,输出如下:

```
>> Bar() in Foo.
>> Instance resolving MissMethod
>> dynamicMethodIMP called.
>> Instance resolving _doZombieMe
```

dynamicMethodIMP 被调用了,crash 没有了!万事大吉!

注意：这里两次调用了 resolveInstanceMethod，而且两次决议的 selector 在不同的系统下是不同的，上面演示的是 10.7 系统下第一个决议 MissMethod，第二个决议 _doZombieMe；在 10.6 系统下两次都是决议 MissMethod。

下面我把 resolveInstanceMethod 方法中为 selector 添加实现的那一行屏蔽了，消息转发就应该会进行：

```
//class_addMethod([self class], name, (IMP)dynamicMethodIMP, "v@:");
```

再次编译运行，此时输出：

```
>> Bar() in Foo.
>> Instance resolving MissMethod
+[Foo resolveInstanceMethod:MissMethod] returned YES, but no new implementation of -[Foo MissMethod] was found
 >> Instance resolving  _doZombieMe
objc[1223]: +[Foo resolveInstanceMethod:MissMethod] returned YES, but no new implementation of -[Foo MissMethod] was found
-[Foo MissMethod]: unrecognized selector sent to instance 0x10010c880
 *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[Foo MissMethod]: unrecognized selector sent to instance 0x10010c880'
*** Call stack at first throw:
......
```

在这里，resolveInstanceMethod 使诈了，它声称成功(返回 YES)决议了 selector，但是并没有真正提供实现，被编译器发觉而提示相应的错误信息。那它的返回值到底有什么作用呢，在它没有提供真正的实现，并且提供了消息转发机制的情况下，YES 表示不进行后续的消息转发，返回 NO 则表示要进行后续的消息转发。

三，源码剖析

让我们来看看运行时系统是如何进行动态方法决议的，下面的代码来自苹果官方公开的源码 objc-class.mm，我在其中添加了中文注释：

1，首先是判断是不是要进行类方法决议，如果不是或决议失败，则进行实例方法决议（请参考：《深入浅出 Cocoa 之类与对象》）：

```
/***********************************************************************
* _class_resolveMethod
* Call +resolveClassMethod or +resolveInstanceMethod and return
* the method added or NULL.
```

```
* Assumes the method doesn't exist already.
**********************************************************************/
__private_extern__ Method _class_resolveMethod(Class cls, SEL sel)
{
    Method meth = NULL;

    if (_class_isMetaClass(cls)) {
        meth = _class_resolveClassMethod(cls, sel);
    }
    if (!meth) {
        meth = _class_resolveInstanceMethod(cls, sel);
    }

    if (PrintResolving  &&  meth) {
        _objc_inform("RESOLVE: method %c[%s %s] dynamically resolved to %p",
                     class_isMetaClass(cls) ? '+' : '-',
                     class_getName(cls), sel_getName(sel),
                     method_getImplementation(meth));
    }

    return meth;
}
```

2，类方法决议与实例方法决议大体相似，在这里就只看实例方法决议部分了：

```
/**********************************************************************
 * _class_resolveInstanceMethod
 * Call +resolveInstanceMethod and return the method added or NULL.
 * cls should be a non-meta class.
 * Assumes the method doesn't exist already.
 **********************************************************************/
static Method _class_resolveInstanceMethod(Class cls, SEL sel)
{
    BOOL resolved;
    Method meth = NULL;

    // 是否实现了 resolveInstanceMethod，如果没有返回 NULL
    if (!look_up_method(((id)cls)->isa, SEL_resolveInstanceMethod,
```

```
                        YES /*cache*/, NO /*resolver*/))
    {
        return NULL;
    }

    // 调用 resolveInstanceMethod，并获取返回值
    resolved = ((BOOL(*)(id, SEL, SEL))objc_msgSend)((id)cls, SEL_resolveInstanceMethod, sel);

    if (resolved) {
        // 返回值为 YES，表示 resolveInstanceMethod 声称它已经成功添加实现，则再次查找 method list
        // +resolveClassMethod adds to self
        meth = look_up_method(cls, sel, YES/*cache*/, NO/*resolver*/);

        if (!meth) {
            // resolveInstanceMethod 使诈了，它声称成功添加实现了，但实际没有，给出警告信息，并返回 NULL
            // Method resolver didn't add anything?
            _objc_inform("+[%s resolveInstanceMethod:%s] returned YES, but "
                         "no new implementation of %c[%s %s] was found",
                         class_getName(cls),
                         sel_getName(sel),
                         class_isMetaClass(cls) ? '+' : '-',
                         class_getName(cls),
                         sel_getName(sel));
            return NULL;
        }
    }

    // 其他情况下返回 NULL
    return meth;
}
```

这段代码很容易理解：

1，首先判断是否实现了 resolveInstanceMethod，如果没有实现，返回 NULL，进入下一步处理；

2，如果实现了，调用 resolveInstanceMethod，获取返回值；

3，如果返回值为 YES，表示 resolveInstanceMethod 声称它已经提供了 selector 的实现，因此再次查找 method list，如果依然找到对应的 IMP，则返回该实现，否则提示警告信息，返回 NULL，进入下一步处理；

4，如果返回值为 NO，返回 NULL，进入下一步处理；

四，加入消息转发

在前文《深入浅出 Cocoa 之消息》一文中，我演示了一个消息转发的示例，下面我把动态方法决议部分去除，把消息转发部分添加进来：

```
// Proxy
@interface Proxy : NSObject

-(void)MissMethod;

@end

@implementation Proxy

-(void)MissMethod
{
    NSLog(@" >> MissMethod() called in Proxy.");
}

@end

// Foo
@interface Foo : NSObject

-(void)Bar;

@end

@implementation Foo

- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    SEL name = [anInvocation selector];
```

```
    NSLog(@" >> forwardInvocation for selector %@", NSStringFromSelector(name));

    Proxy * proxy = [[[Proxy alloc] init] autorelease];
    if ([proxy respondsToSelector:name]) {
        [anInvocation invokeWithTarget:proxy];
    }
    else {
        [super forwardInvocation:anInvocation];
    }
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector {
    return [Proxy instanceMethodSignatureForSelector:aSelector];
}

-(void)Bar
{
    NSLog(@" >> Bar() in Foo.");
}

@end
```

运行测试代码，输出如下：

```
 >> Bar() in Foo.
  >> forwardInvocation for selector MissMethod
  >> MissMethod() called in Proxy.
```

如果我把动态方法决议部分代码也加入进来输出又是怎样呢？下面只列出了 Foo 的实现代码，其他代码不变动。

```
@implementation Foo

+(BOOL)resolveInstanceMethod:(SEL)name
{
    NSLog(@" >> Instance resolving %@", NSStringFromSelector(name));
```

```
    if (name == @selector(MissMethod)) {
        class_addMethod([self class], name, (IMP)dynamicMethodIMP, "v@:");
        return YES;
    }

    return [super resolveInstanceMethod:name];
}

+(BOOL)resolveClassMethod:(SEL)name
{
    NSLog(@" >> Class resolving %@", NSStringFromSelector(name));
    return [super resolveClassMethod:name];
}

- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    SEL name = [anInvocation selector];
    NSLog(@" >> forwardInvocation for selector %@", NSStringFromSelector(name));

    Proxy * proxy = [[[Proxy alloc] init] autorelease];
    if ([proxy respondsToSelector:name]) {
        [anInvocation invokeWithTarget:proxy];
    }
    else {
        [super forwardInvocation:anInvocation];
    }
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector {
    return [Proxy instanceMethodSignatureForSelector:aSelector];
}

-(void)Bar
{
    NSLog(@" >> Bar() in Foo.");
}

@end
```

此时，输出为：

```
>> Bar() in Foo.
>> Instance resolving MissMethod
>> dynamicMethodIMP called.
>> Instance resolving _doZombieMe
```

注意到了没，消息转发没有进行！在前文中说过，消息转发只有在对象无法正常处理消息时才会调用，而在这里我在动态方法决议中为 selector 提供了实现，使得对象可以处理该消息，所以消息转发不会继续了。官方文档中说：

If you implement resolveInstanceMethod: but want particular selectors to actually be forwarded via the forwarding mechanism, you return NO for those selectors.

文档里的说法其实并不准确，只有在 resolveInstanceMethod 的实现中没有真正为 selector 提供实现，并返回 NO 的情况下才会进入消息转发流程；否则绝不会进入消息转发流程，程序要么调用正确的动态方法，要么 crash。这也与前面的源码不太一致，我猜测在比上面源码的更高层次的地方，再次查找了 method list，如果提供了实现就能够找到该实现。

下面我把 resolveInstanceMethod 方法中为 selector 添加实现的那一行屏蔽了，消息转发就应该会进行：

```
//class_addMethod([self class], name, (IMP)dynamicMethodIMP, "v@:");
```

再次编译运行，此时输出正如前面所推断的那样：

```
>> Bar() in Foo.
>> Instance resolving MissMethod
objc[1618]: +[Foo resolveInstanceMethod:MissMethod] returned YES, but no new implementation of -[Foo MissMethod] was found
>> forwardInvocation for selector MissMethod
>> MissMethod() called in Proxy.
>> Instance resolving _doZombieMe
```

进行了消息转发！而且编译器很善意地提示（见前面源码剖析）：哎呀，你不能欺骗我嘛，你说添加了实现（返回 YES），其实还是没有呀！然后编译器就无奈地去看能不能消息转发了。当然如果把返回值修改为 NO 就不会有该警告出现，其他的输出不变。

五，总结

从上面的示例演示可以看出，动态方法决议是先于消息转发的。

如果向一个 Objective C 对象对象发送它无法处理的消息（selector），那么编译器会按照如下次序进行处理：

1，首先看是否为该 selector 提供了动态方法决议机制，如果提供了则转到 2；如果没有提供则转到 3；

2，如果动态方法决议真正为该 selector 提供了实现，那么就调用该实现，完成消息发送流程，消息转发就不会进行了；如果没有提供，则转到 3；

3，其次看是否为该 selector 提供了消息转发机制，如果提供了消息了则进行消息转发，此时，无论消息转发是怎样实现的，程序均不会 crash。（因为消息调用的控制权完全交给消息转发机制处理，即使消息转发并没有做任何事情，运行也不会有错误，编译器更不会有错误提示。）；如果没提供消息转发机制，则转到 4；

4，运行报错：无法识别的 selector，程序 crash；

六，引用

官方运行时源代码：

http://www.opensource.apple.com/source/objc4/objc4-532/runtime/

Objective-C Runtime Programming Guide

[深入浅出 Cocoa]详解键值观察（KVO）及其实现机理

罗朝辉（http://blog.csdn.net/kesalin/）

一，前言

Objective-C 中的键(key)-值(value)观察(KVO)并不是什么新鲜事物，它来源于设计模式中的观察者模式，其基本思想就是：

一个目标对象管理所有依赖于它的观察者对象，并在它自身的状态改变时主动通知观察者对象。这个主动通知通常是通过调用各观察者对象所提供的接口方法来实现的。观察者模式较完美地将目标对象与观察者对象解耦。

在 Objective-C 中有两种使用键值观察的方式：手动或自动，此外还支持注册依赖键（即一个键依赖于其他键，其他键的变化也会作用到该键）。下面将一一讲述这些，并会深入 Objective-C 内部一窥键值观察是如何实现的。

本文源码下载：点此下载

二，运用键值观察

1，注册与解除注册

如果我们已经有了包含可供键值观察属性的类，那么就可以通过在该类的对象（被观察对象）上调用名为 NSKeyValueObserver**Registration** 的 category 方法将观察者对象与被观察者对象注册与解除注册：

```
- (void)addObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath opt
ions:(NSKeyValueObservingOptions)options context:(void *)context;
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath;
```

这两个方法的定义在 Foundation/NSKeyValueObserving.h 中，NSObject，NSArray，NSSet 均实现了以上方法，因此我们不仅可以观察普通对象，还可以观察数组或结合类对象。在该头文件中，我们还可以看到 NSObject 还实现了 NSKeyValueObserver**Notification** 的 category 方法（更多类似方法，请查看该头文件）：

```
- (void)willChangeValueForKey:(NSString *)key;
- (void)didChangeValueForKey:(NSString *)key;
```

这两个方法在手动实现键值观察时会用到，暂且不提。

值得注意的是：不要忘记解除注册，否则会导致资源泄露。

2，设置属性

将观察者与被观察者注册好之后，就可以对观察者对象的属性进行操作，这些变更操作就会被通知给观察者对象。注意，只有遵循 KVO 方式来设置属性，观察者对象才会获取通知，也就是说遵循使用属性的 setter 方法，或通过 key-path 来设置：

```
[target setAge:30];
[target setValue:[NSNumber numberWithInt:30] forKey:@"age"];
```

3，处理变更通知

观察者需要实现名为 NSKeyValueObserving 的 category 方法来处理收到的变更通知：

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object chan
ge:(NSDictionary *)change context:(void *)context;
```

在这里，change 这个字典保存了变更信息，具体是哪些信息取决于注册时的 NSKeyValueObservingOptions。

4，下面来看看一个完整的使用示例：

观察者类：

```
// Observer.h
@interface Observer : NSObject
@end

// Observer.m
#import "Observer.h"
#import <objc/runtime.h>
#import "Target.h"

@implementation Observer

- (void) observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    if ([keyPath isEqualToString:@"age"])
    {
```

```
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                encoding:NSUTF8StringEncoding];
        NSLog(@" >> class: %@, Age changed", className);

        NSLog(@" old age is %@", [change objectForKey:@"old"]);
        NSLog(@" new age is %@", [change objectForKey:@"new"]);
    }
    else
    {
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}

@end
```

注意：在实现处理变更通知方法 observeValueForKeyPath 时，要将不能处理的 key 转发给 super 的 observeValueForKeyPath 来处理。

使用示例：

```
Observer * observer = [[[Observer alloc] init] autorelease];

Target * target = [[[Target alloc] init] autorelease];
[target addObserver:observer
         forKeyPath:@"age"
            options:NSKeyValueObservingOption|New | NSKeyValueObservingOption|O
ld
            context:[Target class]];

[target setAge:30];
//[target setValue:[NSNumber numberWithInt:30] forKey:@"age"];

[target removeObserver:observer forKeyPath:@"age"];
```

在这里 observer 观察 target 的 age 属性变化，运行结果如下：

>> class: Target, Age changed

old age is 10

new age is 30

三，手动实现键值观察

上面的 Target 应该怎么实现呢？首先来看手动实现。

```
@interface Target : NSObject
{
    int age;
}

// for manual KVO - age
- (int) age;
- (void) setAge:(int)theAge;

@end

@implementation Target

- (id) init
{
    self = [super init];
    if (nil != self)
    {
        age = 10;
    }

    return self;
}

// for manual KVO - age
- (int) age
{
    return age;
```

```
}


- (void) setAge:(int)theAge
{
    [self willChangeValueForKey:@"age"];
    age = theAge;
    [self didChangeValueForKey:@"age"];
}


+ (BOOL) automaticallyNotifiesObserversForKey:(NSString *)key {
    if ([key isEqualToString:@"age"]) {
        return NO;
    }

    return [super automaticallyNotifiesObserversForKey:key];
}


@end
```

首先，需要手动实现属性的 setter 方法，并在设置操作的前后分别调用 **willChangeValueForKey:** 和 **didChangeValueForKey** 方法，这两个方法用于通知系统该 key 的属性值即将和已经变更了；

其次，要实现类方法 **automaticallyNotifiesObserversForKey**，并在其中设置对该 key 不自动发送通知（返回 NO 即可）。这里要注意，对其它非手动实现的 key，要转交给 super 来处理。

四，自动实现键值观察

自动实现键值观察就非常简单了，只要使用了自动属性即可。

```
@interface Target : NSObject
// for automatic KVO - age
@property (nonatomic, readwrite) int age;
@end


@implementation Target
@synthesize age; // for automatic KVO - age
```

```
- (id) init
{
    self = [super init];
    if (nil != self)
    {
        age = 10;
    }

    return self;
}

@end
```

五，键值观察依赖键

有时候一个属性的值依赖于另一对象中的一个或多个属性，如果这些属性中任一属性的值发生变更，被依赖的属性值也应当为其变更进行标记。因此，object 引入了依赖键。

1，观察依赖键

观察依赖键的方式与前面描述的一样，下面先在 Observer 的 observeValueForKeyPath:ofObject:change:context: 中添加处理变更通知的代码：

```
#import "TargetWrapper.h"

- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context
{
    if ([keyPath isEqualToString:@"age"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                  encoding:NSUTF8StringEncoding];
        NSLog(@" >> class: %@, Age changed", className);
```

```
        NSLog(@" old age is %@", [change objectForKey:@"old"]);
        NSLog(@" new age is %@", [change objectForKey:@"new"]);
    }
    else if ([keyPath isEqualToString:@"information"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                encoding:NSUTF8StringEncoding];
        NSLog(@" >> class: %@, Information changed", className);
        NSLog(@" old information is %@", [change objectForKey:@"old"]);
        NSLog(@" new information is %@", [change objectForKey:@"new"]);
    }
    else
    {
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

2，实现依赖键

在这里，观察的是 TargetWrapper 类的 information 属性，该属性是依赖于 Target 类的 age 和 grade 属性。为此，我在 Target 中添加了 grade 属性：

```
@interface Target : NSObject
@property (nonatomic, readwrite) int grade;
@property (nonatomic, readwrite) int age;
@end

@implementation Target
@synthesize age; // for automatic KVO - age
@synthesize grade;
@end
```

下面来看看 TragetWrapper 中的依赖键属性是如何实现的。

```
@class Target;
```

```
@interface TargetWrapper : NSObject
{
@private
    Target * _target;
}

@property(nonatomic, assign) NSString * information;
@property(nonatomic, retain) Target * target;

-(id) init:(Target *)aTarget;

@end

#import "TargetWrapper.h"
#import "Target.h"

@implementation TargetWrapper

@synthesize target = _target;

-(id) init:(Target *)aTarget
{
    self = [super init];
    if (nil != self) {
        _target = [aTarget retain];
    }

    return self;
}

-(void) dealloc
{
    self.target = nil;
    [super dealloc];
}
```

```
- (NSString *)information
{
    return [[[NSString alloc] initWithFormat:@"%d#%d", [_target grade], [_target age]] autorelease];
}

- (void)setInformation:(NSString *)theInformation
{
    NSArray * array = [theInformation componentsSeparatedByString:@"#"];
    [_target setGrade:[[array objectAtIndex:0] intValue]];
    [_target setAge:[[array objectAtIndex:1] intValue]];
}

+ (NSSet *)keyPathsForValuesAffectingInformation
{
    NSSet * keyPaths = [NSSet setWithObjects:@"target.age", @"target.grade", nil];
    return keyPaths;
}

//+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
//{
//    NSSet * keyPaths = [super keyPathsForValuesAffectingValueForKey:key];
//    NSArray * moreKeyPaths = nil;
//
//    if ([key isEqualToString:@"information"])
//    {
//        moreKeyPaths = [NSArray arrayWithObjects:@"target.age", @"target.grade", nil];
//    }
//
//    if (moreKeyPaths)
//    {
//        keyPaths = [keyPaths setByAddingObjectsFromArray:moreKeyPaths];
//    }
//
//    return keyPaths;
//}
```

```
@end
```

首先，要手动实现属性 information 的 setter/getter 方法，在其中使用 Target 的属性来完成其 setter 和 getter。

其次，要实现 **keyPathsForValuesAffectingInformation** 或 **keyPathsForValuesAffectingValueForKey**：方法来告诉系统 information 属性依赖于哪些其他属性，这两个方法都返回一个 key-path 的集合。在这里要多说几句，如果选择实现 **keyPathsForValuesAffectingValueForKey**，要先获取 super 返回的结果 set，然后判断 key 是不是目标 key，如果是就将依赖属性的 key-path 结合追加到 super 返回的结果 set 中，否则直接返回 super 的结果。

在这里，information 属性依赖于 target 的 age 和 grade 属性，target 的 age/grade 属性任一发生变化，information 的观察者都会得到通知。

3，使用示例：

```
Observer * observer = [[[Observer alloc] init] autorelease];
Target * target = [[[Target alloc] init] autorelease];

TargetWrapper * wrapper = [[[TargetWrapper alloc] init:target] autorelease];
[wrapper addObserver:observer
         forKeyPath:@"information"
            options:NSKeyValueObservingOptionNew | NSKeyValueObservingOption
Old
            context:[TargetWrapper class]];

[target setAge:30];
[target setGrade:1];
[wrapper removeObserver:observer forKeyPath:@"information"];
```

输出结果：

```
>> class: TargetWrapper, Information changed
 old information is 0#10
 new information is 0#30
 >> class: TargetWrapper, Information changed
 old information is 0#30
 new information is 1#30
```

六，键值观察是如何实现的

1，实现机理

键值观察用处很多，Core Binding 背后的实现就有它的身影，那键值观察背后的实现又如何呢？想一想在上面的自动实现方式中，我们并不需要在被观察对象 Target 中添加额外的代码，就能获得键值观察的功能，这很好很强大，这是怎么做到的呢？答案就是 Objective C 强大的 runtime 动态能力，下面我们一起来窥探下其内部实现过程。

当某个类的对象第一次被观察时，系统就会在运行期动态地创建该类的一个派生类，在这个派生类中重写基类中任何被观察属性的 setter 方法。

派生类在被重写的 setter 方法实现真正的通知机制，就如前面手动实现键值观察那样。这么做是基于设置属性会调用 setter 方法，而通过重写就获得了 KVO 需要的通知机制。当然前提是要通过遵循 KVO 的属性设置方式来变更属性值，如果仅是直接修改属性对应的成员变量，是无法实现 KVO 的。

同时派生类还重写了 class 方法以"欺骗"外部调用者它就是起初的那个类。然后系统将这个对象的 isa 指针指向这个新诞生的派生类，因此这个对象就成为该派生类的对象了，因而在该对象上对 setter 的调用就会调用重写的 setter，从而激活键值通知机制。此外，派生类还重写了 dealloc 方法来释放资源。

如果你对类和对象的关系不太明白，请阅读《深入浅出 Cocoa 之类与对象》；如果你对如何动态创建类不太明白，请阅读《深入浅出 Cocoa 之动态创建类》。

苹果官方文档说得很简洁：

Key-Value Observing Implementation Details

Automatic key-value observing is implemented using a technique called **isa-swizzling**.

The `isa` pointer, as the name suggests, points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements, among other data.

When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result the value of the isa pointer does not necessarily reflect the actual class of the instance.

You should never rely on the `isa` pointer to determine class membership. Instead, you should use the `class` method to determine the class of an object instance.

2，代码分析

由于派生类中被重写的 class 对我们撒谎（它说它就是起初的基类），我们只有通过调用 runtime 函数才能揭开派生类的真面目。 下面来看 Mike Ash 的代码：

首先是带有 x, y, z 三个属性的观察目标 Foo：

```
@interface Foo : NSObject
{
```

```
    int x;
    int y;
    int z;
}

@property int x;
@property int y;
@property int z;

@end

@implementation Foo
@synthesize x, y, z;
@end
```

下面是检验代码：

```
#import <objc/runtime.h>

static NSArray * ClassMethodNames(Class c)
{
    NSMutableArray * array = [NSMutableArray array];

    unsigned int methodCount = 0;
    Method * methodList = class_copyMethodList(c, &methodCount);
    unsigned int i;
    for(i = 0; i < methodCount; i++) {
        [array addObject: NSStringFromSelector(method_getName(methodList
[i]))];
    }

    free(methodList);

    return array;
}

static void PrintDescription(NSString * name, id obj)
{
    NSString * str = [NSString stringWithFormat:
```

```
                    @"\n\t%@: %@\n\tNSObject class %s\n\tlibobjc class %s\n\t
implements methods <%@>",
                    name,
                    obj,
                    class_getName([obj class]),
                    class_getName(obj->isa),
                    [ClassMethodNames(obj->isa) componentsJoinedByString:@",
"]];
    NSLog(@"%@", str);
}

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // Deep into KVO: kesalin@gmail.com
        //
        Foo * anything = [[Foo alloc] init];
        Foo * x = [[Foo alloc] init];
        Foo * y = [[Foo alloc] init];
        Foo * xy = [[Foo alloc] init];
        Foo * control = [[Foo alloc] init];

        [x addObserver:anything forKeyPath:@"x" options:0 context:NULL];
        [y addObserver:anything forKeyPath:@"y" options:0 context:NULL];

        [xy addObserver:anything forKeyPath:@"x" options:0 context:NULL];
        [xy addObserver:anything forKeyPath:@"y" options:0 context:NULL];

        PrintDescription(@"control", control);
        PrintDescription(@"x", x);
        PrintDescription(@"y", y);
        PrintDescription(@"xy", xy);

        NSLog(@"\n\tUsing NSObject methods, normal setX: is %p, overridden se
tX: is %p\n",
              [control methodForSelector:@selector(setX:)],
              [x methodForSelector:@selector(setX:)]);
```

```
        NSLog(@"\n\tUsing libobjc functions, normal setX: is %p, overridden s
etX: is %p\n",
              method_getImplementation(class_getInstanceMethod(object_getCla
ss(control),
                                                               @selector(setX:))),
              method_getImplementation(class_getInstanceMethod(object_getCla
ss(x),
                                                               @selector(setX:))));
    }


    return 0;
}
```

在上面的代码中，辅助函数 ClassMethodNames 使用 runtime 函数来获取类的方法列表，PrintDescription 打印对象的信息，包括通过 -class 获取的类名， isa 指针指向的类的名字以及其中方法列表。

在这里，我创建了四个对象，x 对象的 x 属性被观察，y 对象的 y 属性被观察，xy 对象的 x 和 y 属性均被观察，参照对象 control 没有属性被观察。在代码的最后部分，分别通过两种方式(对象方法和 runtime 方法)打印出参数对象 control 和被观察对象 x 对象的 setX 方面的实现地址，来对比显示正常情况下 setter 实现以及派生类中重写的 setter 实现。

编译运行，输出如下：

control: <Foo: 0x10010c980>

NSObject class Foo

libobjc class Foo

implements methods <x, setX:, y, setY:, z, setZ:>

x: <Foo: 0x10010c920>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

y: <Foo: 0x10010c940>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

xy: <Foo: 0x10010c960>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

Using NSObject methods, normal setX: is 0x100001df0, overridden setX: is 0x100001df0

Using libobjc functions, normal setX: is 0x100001df0, overridden setX: is 0x7fff8458e025

从上面的输出可以看到，如果使用对象的 -class 方面输出类名始终为：Foo，这是因为新诞生的派生类重写了 -class 方法声称它就是起初的基类，只有使用 runtime 函数 object_getClass 才能一睹芳容：NSKVONotifying_Foo。注意看：x，y 以及 xy 三个被观察对象真正的类型都是 NSKVONotifying_Foo，而且该类实现了：setY:, setX:, class, dealloc, _isKVOA 这些方法。其中 setX:, setY:, class 和 dealloc 前面已经讲到过，私有方法 _isKVOA 估计是用来标示该类是一个 KVO 机制声称的类。在这里 Objective C 做了一些优化，它对所有被观察对象只生成一个派生类，该派生类实现所有被观察对象的 setter 方法，这样就减少了派生类的数量，提供了效率。所有 NSKVONotifying_Foo 这个派生类重写了 setX，setY 方法（留意:没有必要重写 setZ 方法）。

接着来看最后两行输出，地址 0x100001df0 是 Foo 类中的实现，而地址是 0x7fff8458e025 是派生类 NSKVONotifying_Foo 类中的实现。那后面那个地址到底是什么呢？可以通过 GDB 的 info 命令加 symbol 参数来查看该地址的信息：

(gdb) info symbol 0x7fff8458e025
_NSSetIntValueAndNotify in section LC_SEGMENT.__TEXT.__text of /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation

看起来它是 Foundation 框架提供的私有函数：_NSSetIntValueAndNotify。更进一步，我们来看看 Foundation 到底提供了哪些用于 KVO 的辅助函数。打开 terminal，使用 nm -a 命令查看 Foundation 中的信息：

nm -a /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation

其中查找到我们关注的函数：

```
00000000000233e7 t __NSSetDoubleValueAndNotify
00000000000f32ba t __NSSetFloatValueAndNotify
0000000000025025 t __NSSetIntValueAndNotify
000000000007fbb5 t __NSSetLongLongValueAndNotify
00000000000f33e8 t __NSSetLongValueAndNotify
000000000002d36c t __NSSetObjectValueAndNotify
0000000000024dc5 t __NSSetPointValueAndNotify
00000000000f39ba t __NSSetRangeValueAndNotify
00000000000f3aeb t __NSSetRectValueAndNotify
00000000000f3512 t __NSSetShortValueAndNotify
00000000000f3c2f t __NSSetSizeValueAndNotify
```

```
00000000000f363b t __NSSetUnsignedCharValueAndNotify
000000000006e91f t __NSSetUnsignedIntValueAndNotify
0000000000034b5b t __NSSetUnsignedLongLongValueAndNotify
00000000000f3766 t __NSSetUnsignedLongValueAndNotify
00000000000f3890 t __NSSetUnsignedShortValueAndNotify
00000000000f3060 t __NSSetValueAndNotifyForKeyInIvar
00000000000f30d7 t __NSSetValueAndNotifyForUndefinedKey
```

Foundation 提供了大部分基础数据类型的辅助函数（Objective C 中的 Boolean 只是 unsigned char 的 typedef，所以包括了，但没有 C++中的 bool），此外还包括一些常见的 Cocoa 结构体如 Point, Range, Rect, Size，这表明这些结构体也可以用于自动键值观察，但要注意除此之外的结构体就不能用于自动键值观察了。对于所有 Objective C 对象对应的是 __NSSetObjectValueAndNotify 方法。

七，总结

KVO 并不是什么新事物，换汤不换药，它只是观察者模式在 Objective C 中的一种运用，这是 KVO 的指导思想所在。其他语言实现中也有“KVO”，如 WPF 中的 binding。而在 Objective C 中又是通过强大的 runtime 来实现自动键值观察的。至此，对 KVO 的使用以及注意事项，内部实现都介绍完毕，对 KVO 的理解又深入一层了。Objective 中的 KVO 虽然可以用，但却非完美，有兴趣的了解朋友请查看《KVO 的缺陷》以及改良实现 MAKVONotificationCenter 。

八，引用

Key-value observing：官方文档

Key-Value Observing Done Right：官方 KVO 实现的缺陷

MAKVONotificationCenter：一个改良的 Notification 实现，托管在 GitHub 上

[深入浅出 Cocoa] 之 Method Swizzling

罗朝辉(http://blog.csdn.net/kesalin)

在前文深入浅出 Cocoa 之消息中，我简要介绍了 ObjC 中消息的基本情况，包括 SEL 查找，缓存以及消息转发等。在本文中，我要介绍一个很有趣的技术，Method swizzling，通过这个手法，我们可以动态修改方法的实现，从而达到修改类行为的目的。当然，还有其他办法（如 ClassPosing，Category）也可以达到这个目的。ClassPosing 是针对类级别的，是重量级的手法，Category 也差不多，比较重量级，此外 Category 还无法避免下面的递归死循环（如果你的代码出现了如下形式的递归调用，应该考虑一下你的设计，而不是使用在这里介绍的 Method Swizzling 手法，：））。

```
// Bar
//
@implementation Bar

- (void) testMethod
{
    NSLog(@" >> Bar testMethod");
}

@end

// Bar(BarCategory)
//
@implementation Bar(BarCategory)

- (void) altRecursionMethod
{
    NSLog(@" >> Bar(BarCategory) recursionMethod");
    [self altRecursionMethod];
}

@end
```

在前文深入浅出 Cocoa 之消息中提到，ObjC 中的类(class)和实例(instance)都是对象，类对象有自己的类方法列表，实例对象有自己的实例方法列表，这些方法列表（struct objc_method_list）是存储在 struct objc_class 中的。每个方法列表存储近似 SEL:Method 的对，Method 是一个对象，包含方法的具体实现 impl。由此可知，我们只需要修改 SEL 对应的 Method 的 impl 既可以达到修改消息行为的目的。下面来看代码：

```
void PerformSwizzle(Class aClass, SEL orig_sel, SEL alt_sel, BOOL forInstance)
{
    // First, make sure the class isn't nil
    if (aClass != nil) {
        Method orig_method = nil, alt_method = nil;

        // Next, look for the methods
        if (forInstance) {
            orig_method = class_getInstanceMethod(aClass, orig_sel);
            alt_method = class_getInstanceMethod(aClass, alt_sel);
        } else {
            orig_method = class_getClassMethod(aClass, orig_sel);
            alt_method = class_getClassMethod(aClass, alt_sel);
        }

        // If both are found, swizzle them
        if ((orig_method != nil) && (alt_method != nil)) {
            IMP temp;

            temp = orig_method->method_imp;
            orig_method->method_imp = alt_method->method_imp;
            alt_method->method_imp = temp;
        } else {
#if DEBUG
            NSLog(@"PerformSwizzle Error: Original %@, Alternate %@",(orig_method
== nil)?@" not found":@" found",(alt_method == nil)?@" not found":@" found");
#endif
        }
    } else {
#if DEBUG
        NSLog(@"PerformSwizzle Error: Class not found");
#endif
    }
}

void MethodSwizzle(Class aClass, SEL orig_sel, SEL alt_sel)
{
    PerformSwizzle(aClass, orig_sel, alt_sel, YES);
```

```
}


void ClassMethodSwizzle(Class aClass, SEL orig_sel, SEL alt_sel)
{
    PerformSwizzle(aClass, orig_sel, alt_sel, NO);
}
```

让我们来分析上面代码：

1，首先，区分类方法和实例方法；

2，取得 SEL 对应的 Method；

3，修改 Method 的 impl，在这里是通过交换实现的。

上面的代码是可以工作的，但还不够完善。Apple 10.5 提供了交换 Method 实现的 API: `method_exchangeImplementations` 。下面我们使用这个新 API，并以 NSObject category 的形式给出新的实现方式：

```
#if TARGET_OS_IPHONE
#import <objc/runtime.h>
#import <objc/message.h>
#else
#import <objc/objc-class.h>
#endif

// NSObject (MethodSwizzlingCategory)
//
@interface NSObject (MethodSwizzlingCategory)

+ (BOOL)swizzleMethod:(SEL)origSel withMethod:(SEL)altSel;
+ (BOOL)swizzleClassMethod:(SEL)origSel withClassMethod:(SEL)altSel;

@end

@implementation NSObject (MethodSwizzlingCategory)

+ (BOOL)swizzleMethod:(SEL)origSel withMethod:(SEL)altSel
{
    Method origMethod = class_getInstanceMethod(self, origSel);
    if (!origSel) {
        NSLog(@"original method %@ not found for class %@",
NSStringFromSelector(origSel), [self class]);
        return NO;
```

```
    }

    Method altMethod = class_getInstanceMethod(self, altSel);
         if (!altMethod) {
       NSLog(@"original method %@ not found for class %@",
NSStringFromSelector(altSel), [self class]);
       return NO;
    }

    class_addMethod(self, origSel, class_getMethodImplementation(self, origSel),
method_getTypeEncoding(origMethod));
    class_addMethod(self, altSel, class_getMethodImplementation(self, altSel),
method_getTypeEncoding(altMethod));

    method_exchangeImplementations(class_getInstanceMethod(self, origSel),
class_getInstanceMethod(self, altSel));

    return YES;
}

+ (BOOL)swizzleClassMethod:(SEL)origSel withClassMethod:(SEL)altSel
{
    Class c = object_getClass((id)self);
    return [c swizzleMethod:origSel withMethod:altSel];
}

@end
```

代码就不用多解释了，下面我们来看如何使用。先看辅助类 Foo：

Foo.h

```
//
//  Foo.h
//  MethodSwizzling
//
//  Created by LuoZhaohui on 1/5/12.
//  Copyright (c) 2012 http://blog.csdn.net/kesalin/. All rights reserved.
//
```

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

// Foo
//
@interface Foo : NSObject

- (void) testMethod;
- (void) baseMethod;
- (void) recursionMethod;

@end

// Bar
//
@interface Bar : Foo

- (void) testMethod;

@end

// Bar(BarCategory)
//
@interface Bar(BarCategory)

- (void) altTestMethod;
- (void) altBaseMethod;
- (void) altRecursionMethod;

@end
```

Foo.m

```
//
//  Foo.m
//  MethodSwizzling
//
//  Created by LuoZhaohui on 1/5/12.
//  Copyright (c) 2012 http://blog.csdn.net/kesalin/. All rights reserved.
//
```

```
#import "Foo.h"

// Foo
//
@implementation Foo

- (void) testMethod
{
    NSLog(@" >> Foo testMethod");
}

- (void) baseMethod
{
    NSLog(@" >> Foo baseMethod");
}

- (void) recursionMethod
{
    NSLog(@" >> Foo recursionMethod");
}

@end

// Bar
//
@implementation Bar

- (void) testMethod
{
    NSLog(@" >> Bar testMethod");
}

@end

// Bar(BarCategory)
//
@implementation Bar(BarCategory)
```

```
- (void) altTestMethod
{
    NSLog(@" >> Bar(BarCategory) altTestMethod");
}

- (void) altBaseMethod
{
    NSLog(@" >> Bar(BarCategory) altBaseMethod");
}

- (void) altRecursionMethod
{
    NSLog(@" >> Bar(BarCategory) recursionMethod");
    [self altRecursionMethod];
}

@end
```

下面是具体的使用示例：

```
int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        Foo * foo = [[[Foo alloc] init] autorelease];
        Bar * bar = [[[Bar alloc] init] autorelease];

        NSLog(@"========= Method Swizzling test 1 =========");

        NSLog(@" Step 1");
        [foo testMethod];
        [bar testMethod];
        [bar altTestMethod];

        NSLog(@" Step 2");
        [Bar swizzleMethod:@selector(testMethod)
withMethod:@selector(altTestMethod)];
        [foo testMethod];
        [bar testMethod];
```

```
        [bar altTestMethod];


        NSLog(@"========= Method Swizzling test 2 =========");
        NSLog(@" Step 1");
        [foo baseMethod];
        [bar baseMethod];
        [bar altBaseMethod];


        NSLog(@" Step 2");
        [Bar swizzleMethod:@selector(baseMethod)
withMethod:@selector(altBaseMethod)];
        [foo baseMethod];
        [bar baseMethod];
        [bar altBaseMethod];


        NSLog(@"========= Method Swizzling test 3 =========");
        [Bar swizzleMethod:@selector(recursionMethod)
withMethod:@selector(altRecursionMethod)];
        [bar recursionMethod];
    }


    return 0;
}
```

输出结果为：注意，test 3 中调用了递归调用“自己”的方法，你能理解为什么没有出现死循环么？

========= Method Swizzling test 1 =========
Step 1
>> Foo testMethod
>> Bar testMethod
>> Bar(BarCategory) altTestMethod
Step 2
>> Foo testMethod
>> Bar(BarCategory) altTestMethod
>> Bar testMethod
========= Method Swizzling test 2 =========
Step 1
>> Foo baseMethod
>> Foo baseMethod
>> Bar(BarCategory) altBaseMethod
Step 2

>> Foo baseMethod

>> Bar(BarCategory) altBaseMethod

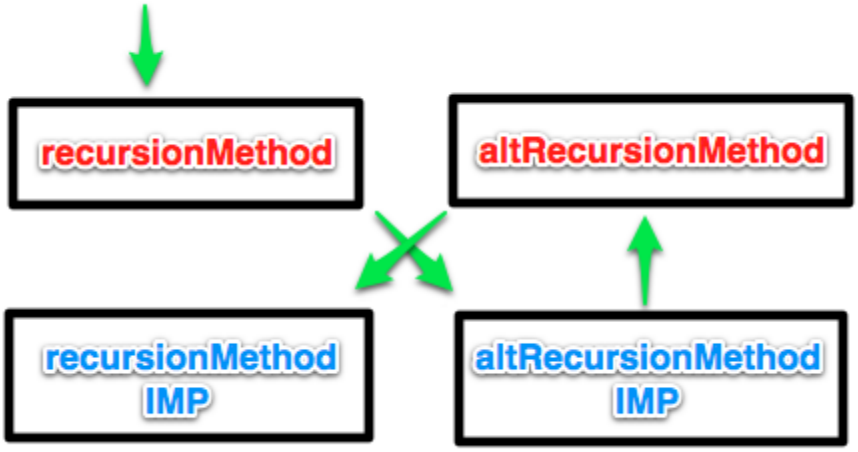
>> Foo baseMethod

========= Method Swizzling test 3 =========

>> Bar(BarCategory) recursionMethod

>> Foo recursionMethod

test3 解释：在函数体 {} 之间的部分是真正的 IMP，而在这之前的是 SEL。通常情况下，SEL 是与 IMP 匹配的，但在 swizzling 之后，情况就不同了。下图就是调用的时序图。



rentzsch 写了一个完善的开源类 jrswizzle 来处理 Method Swizzling，如果你在工程中使用到 Method Swizzling 手法，应该优先使用这个类库，：）。

Refference：

MethodSwizzling：http://www.cocoadev.com/index.pl?ExtendingClasses

jrswizzle：https://github.com/rentzsch/jrswizzle

[深入浅出 Cocoa] 之多线程 NSThread

罗朝辉(http://blog.csdn.net/kesalin)

iOS 支持多个层次的多线程编程，层次越高的抽象程度越高，使用起来也越方便，也是苹果最推荐使用的方法。下面根据抽象层次从低到高依次列出 iOS 所支持的多线程编程范式：

1, Thread;

2, Cocoa operations;

3, Grand Central Dispatch (GCD) (iOS4 才开始支持)

下面简要说明这三种不同范式：

Thread 是这三种范式里面相对轻量级的，但也是使用起来最负责的，你需要自己管理 thread 的生命周期，线程之间的同步。线程共享同一应用程序的部分内存空间，它们拥有对数据相同的访问权限。你得协调多个线程对同一数据的访问，一般做法是在访问之前加锁，这会导致一定的性能开销。在 iOS 中我们可以使用多种形式的 thread:

Cocoa threads: 使用 NSThread 或直接从 NSObject 的类方法 performSelectorInBackground:withObject: 来创建一个线程。如果你选择 thread 来实现多线程，那么 NSThread 就是官方推荐优先选用的方式。

POSIX threads: 基于 C 语言的一个多线程库，

Cocoa operations 是基于 Obective-C 实现的，类 NSOperation 以面向对象的方式封装了用户需要执行的操作，我们只要聚焦于我们需要做的事情，而不必太操心线程的管理，同步等事情，因为 NSOperation 已经为我们封装了这些事情。 NSOperation 是一个抽象基类，我们必须使用它的子类。iOS 提供了两种默认实现：NSInvocationOperation 和 NSBlockOperation。

Grand Central Dispatch (GCD): iOS4 才开始支持，它提供了一些新的特性，以及运行库来支持多核并行编程，它的关注点更高：如何在多个 cpu 上提升效率。

有了上面的总体框架，我们就能清楚地知道不同方式所处的层次以及可能的效率，便利性差异。下面我们先来看看 NSThread 的使用，包括创建，启动，同步，通信等相关知识。这些与 win32/Java 下的 thread 使用非常相似。

线程创建与启动

NSThread 的创建主要有两种直接方式：

```
[NSThread detachNewThreadSelector:@selector(myThreadMainMethod:) toTarget:self withObject:nil];
```

和

```
NSThread* myThread = [[NSThread alloc] initWithTarget:self
                                            selector:@selector(myThreadMainMethod:)
                                            object:nil];
[myThread start];
```

这两种方式的区别是：前一种一调用就会立即创建一个线程来做事情；而后一种虽然你 alloc 了也 init 了，但是要直到我们手动调用 start 启动线程时才会真正去创建线程。这种延迟实现思想在很多跟资源相关的地方都有用到。后一种方式我们还可以在启动线程之前，对线程进行配置，比如设置 stack 大小，线程优先级。

还有一种间接的方式，更加方便，我们甚至不需要显式编写 NSThread 相关代码。那就是利用 NSObject 的类方法

performSelectorInBackground:withObject: 来创建一个线程：

[myObj performSelectorInBackground:@selector(myThreadMainMethod) withObject:nil];

其效果与 NSThread 的 detachNewThreadSelector:toTarget:withObject: 是一样的。

线程同步

线程的同步方法跟其他系统下类似，我们可以用原子操作，可以用 mutex，lock 等。

iOS 的原子操作函数是以 OSAtomic 开头的，比如：OSAtomicAdd32, OSAtomicOr32 等等。这些函数可以直接使用，因为它们是原子操作。

iOS 中的 mutex 对应的是 NSLock，它遵循 NSLooking 协议，我们可以使用 lock, tryLock, lockBeforeData:来加锁，用 unLock 来解锁。使用示例：

```
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];
...
while (moreToDo) {
    /* Do another increment of calculation */
    /* until there's no more to do. */
    if ([theLock tryLock]) {
        /* Update display used by all threads. */
        [theLock unlock];
    }
}
```

我们可以使用指令 @synchronized 来简化 NSLock 的使用，这样我们就不必显示编写创建 NSLock,加锁并解锁相关代码。

```
- (void)myMethod:(id)anObj
{
    @synchronized(anObj)
    {
        // Everything between the braces is protected by the @synchronized directive.
    }
}
```

还有其他的一些锁对象，比如：循环锁 NSRecursiveLock，条件锁 NSConditionLock，分布式锁 NSDistributedLock 等等，在这里就不一一介绍了，大家去看官方文档吧。

用 NSCodition 同步执行的顺序

NSCodition 是一种特殊类型的锁，我们可以用它来同步操作执行的顺序。它与 mutex 的区别在于更加精准，等待某个 NSCondtion 的线程一直被 lock，直到其他线程给那个 condition 发送了信号。下面我们来看使用示例：

某个线程等待着事情去做，而有没有事情做是由其他线程通知它的。

```
[cocoaCondition lock];
while (timeToDoWork <= 0)
    [cocoaCondition wait];
```

```
timeToDoWork--;
// Do real work here.
[cocoaCondition unlock];
```

其他线程发送信号通知上面的线程可以做事情了：

```
[cocoaCondition lock];
timeToDoWork++;
[cocoaCondition signal];
[cocoaCondition unlock];
```

线程间通信

线程在运行过程中，可能需要与其它线程进行通信。我们可以使用 NSObject 中的一些方法：

在应用程序主线程中做事情：

performSelectorOnMainThread:withObject:waitUntilDone:

performSelectorOnMainThread:withObject:waitUntilDone:modes:

在指定线程中做事情：

performSelector:onThread:withObject:waitUntilDone:

performSelector:onThread:withObject:waitUntilDone:modes:

在当前线程中做事情：

performSelector:withObject:afterDelay:

performSelector:withObject:afterDelay:inModes:

取消发送给当前线程的某个消息

cancelPreviousPerformRequestsWithTarget:

cancelPreviousPerformRequestsWithTarget:selector:object:

如在我们在某个线程中下载数据，下载完成之后要通知主线程中更新界面等等，可以使用如下接口：-

```
(void)myThreadMainMethod
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    // to do something in your thread job
    ...
    [self performSelectorOnMainThread:@selector(updateUI) withObject:nil waitUntilDone:NO];
    [pool release];
}
```

RunLoop

说到 NSThread 就不能不说起与之关系相当紧密的 NSRunLoop。Run loop 相当于 win32 里面的消息循环机制，它可以让你根据事件/消息（鼠标消息，键盘消息，计时器消息等）来调度线程是忙碌还是闲置。

系统会自动为应用程序的主线程生成一个与之对应的 run loop 来处理其消息循环。在触摸 UIView 时之所以能够激发 touchesBegan/touchesMoved 等等函数被调用，就是因为应用程序的主线程在 UIApplicationMain 里面有这样一个 run loop 在分发 input 或 timer 事件。

参考资料：

NSRunLoop 概述和原理：http://xubenyang.me/384

官方文档：http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Multithreading/

[深入浅出 Cocoa] 多线程编程之 block 与 dispatch quene

罗朝辉(http://blog.csdn.net/kesalin)

block 是 Apple 在 GCC 4.2 中扩充的新语法特性，其目的是支持多核并行编程。我们可以将 dispatch_queue 与 block 结合起来使用，方便进行多线程编程。

本文源代码下载：点击下载

1，实验工程准备

在 XCode 4.0 中，我们建立一个 Mac OS X Application 类型的 Command Line Tool，在 Type 里面我们选择 Foundation 就好，工程名字暂且为 StudyBlocks.默认生成的工程代码 main.m 内容如下：

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

    // insert code here...
    NSLog(@"Hello, World!");

    [pool drain];
    return 0;
}
```

2，如何编写 block

在自动生成的工程代码中，默认打印一条语句"Hello, World!"，这个任务可以不可以用 block 语法来实现呢？答案是肯定的，请看：

```
    void (^aBlock)(void) = ^(void){ NSLog(@"Hello, World!"); };
    aBlock();
```

用上面的这两行语句替换 main.m 中的 NSLog(@"Hello, World!"); 语句，编译运行，结果是一样的。

这两行语句是什么意思呢？首先，等号左边的 void (^aBlock)(void) 表示声明了一个 block，这个 block 不带参数(void)且也无返回参数(void)；等号右边的 ^(void){ } 结构表示一个 block 的实现体，至于这个 block 具体要做的事情就都在 {} 之间了。在这里我们仅仅是打印一条语句。整个语句就是声明一个 block，并对其赋值。第二个语句就是调用这个 block 做实际的事情，就像我们调用函数一样。block 很有点像 C++0X 中的 Lambda 表达式。

我们也可以这么写：

```
void (^aBlock)(void) = 0;
aBlock = ^(void){
        NSLog(@" >> Hello, World!");
};
aBlock();
```

现在我们知道了一个 block 该如何编写了，那么 block 数组呢？也很简单，请看：

```
void (^blocks[2])(void) = {
    ^(void){ NSLog(@" >> This is block 1!"); },
    ^(void){ NSLog(@" >> This is block 2!"); }
};

blocks[0]();
blocks[1]();
```

谨记！

block 是分配在 stack 上的，这意味着我们必须小心里处理 block 的生命周期。

比如如下的做法是不对的，因为 stack 分配的 block 在 if 或 else 内是有效的，但是到大括号 } 退出时就可能无效了：

```
dispatch_block_t block;

if (x) {
    block = ^{ printf("true\n"); };
} else {
    block = ^{ printf("false\n"); };
}
block();
```

上面的代码就相当于下面这样的 unsafe 代码：

```
if (x) {
    struct Block __tmp_1 = ...; // setup details
    block = &__tmp_1;
} else {
    struct Block __tmp_2 = ...; // setup details
    block = &__tmp_2;
}
```

3，如何在 block 中修改外部变量

考虑到 block 的目的是为了支持并行编程，对于普通的 local 变量，我们就不能在 block 里面随意修改（原因很简单，block 可以被多个线程并行运行，会有问题的），而且如果你在 block 中修改普通的 local 变量，编译器也会报错。那么该如何修改外部变量呢？有两种办法，第一种是可以修改 static 全局变量；第二种是可以修改用新关键字 __block 修饰的变量。请看：

```
__block int blockLocal  = 100;
static int staticLocal  = 100;

void (^aBlock)(void) = ^(void){
    NSLog(@" >> Sum: %d\n", global + staticLocal);

    global++;
    blockLocal++;
    staticLocal++;
};

aBlock();

NSLog(@"After modified, global: %d, block local: %d, static local: %d\n", global, blockLocal, staticLocal);
```

相似的情况，我们也可以引用 static block 或 __block block。比如我们可以用他们来实现 block 递归：

```
// 1
void (^aBlock)(int) = 0;
static void (^ const staticBlock)(int) = ^(int i) {
    if (i > 0) {
        NSLog(@" >> static %d", i);
        staticBlock(i - 1);
    }
};

aBlock = staticBlock;
aBlock(5);

// 2
__block void (^blockBlock)(int);
blockBlock = ^(int i) {
    if (i > 0) {
        NSLog(@" >> block %d", i);
        blockBlock(i - 1);
    }
};

blockBlock(5);
```

4，上面我们介绍了 block 及其基本用法，但还没有涉及并行编程。block 与 Dispatch Queue 分发队列结合起来使用，是 iOS 中并行编程的利器。请看代码：

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


initData();


// create dispatch queue
//
dispatch_queue_t queue = dispatch_queue_create("StudyBlocks", NULL);


dispatch_async(queue, ^(void) {
    int sum = 0;
    for(int i = 0; i < Length; i++)
        sum += data[i];


    NSLog(@" >> Sum: %d", sum);


    flag = YES;
});


// wait util work is done.
//
while (!flag);
dispatch_release(queue);


[pool drain];
```

上面的 block 仅仅是将数组求和。首先，我们创建一个串行分发队列，然后将一个 block 任务加入到其中并行运行，这样 block 就会在新的线程中运行，直到结束返回主线程。在这里要注意 flag 的使用。flag 是 static 的，所以我们可以 block 中修改它。语句 while (!flag); 的目的是保证主线程不会 blcok 所在线程之前结束。

dispatch_queue_t 的定义如下：
typedef void (^dispatch_block_t)(void);
这意味着加入 dispatch_queue 中的 block 必须是无参数也无返回值的。

dispatch_queue_create 的定义如下：
dispatch_queue_t dispatch_queue_create(const char *label, dispatch_queue_attr_t attr);
这个函数带有两个参数：一个用于标识 dispatch_queue 的字符串；一个是保留的 dispatch_queue 属性，将其设置为 NULL 即可。

我们也可以使用

dispatch_queue_t dispatch_get_global_queue(long priority, unsigned long flags);

来获得全局的 dispatch_queue，参数 priority 表示优先级，值得注意的是：我们不能修改该函数返回的 dispatch_queue。

dispatch_async 函数的定义如下：

void dispatch_async(dispatch_queue_t queue, dispatch_block_t block);

它是将一个 block 加入一个 dispatch_queue，这个 block 会再其后得到调度时，并行运行。

相应的 dispatch_sync 函数就是同步执行了，一般很少用到。比如上面的代码如果我们修改为 dispatch_sync，那么就无需编写 flag 同步代码了。

5，dispatch_queue 的运作机制及线程间同步

我们可以将许多 blocks 用 dispatch_async 函数提交到到 dispatch_queue 串行运行。这些 blocks 是按照 FIFO(先入先出)规则调度的，也就是说，先加入的先执行，后加入的一定后执行，但在某一个时刻，可能有多个 block 同时在执行。

在上面的例子中，我们的主线程一直在轮询 flag 以便知晓 block 线程是否执行完毕，这样做的效率是很低的，严重浪费 CPU 资源。我们可以使用一些通信机制来解决这个问题，如：semaphore（信号量）。 semaphore 的原理很简单，就是生产-消费模式，必须生产一些资源才能消费，没有资源的时候，那我就啥也不干，直到资源就绪。下面来看代码：

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


initData();


// Create a semaphore with 0 resource
//
__block dispatch_semaphore_t sem = dispatch_semaphore_create(0);


// create dispatch semaphore
//
dispatch_queue_t queue = dispatch_queue_create("StudyBlocks", NULL);


dispatch_async(queue, ^(void) {
    int sum = 0;
    for(int i = 0; i < Length; i++)
        sum += data[i];


    NSLog(@" >> Sum: %d", sum);


    // signal the semaphore: add 1 resource
    //
    dispatch_semaphore_signal(sem);
```

```
});

// wait for the semaphore: wait until resource is ready.
//
dispatch_semaphore_wait(sem, DISPATCH_TIME_FOREVER);

dispatch_release(sem);
dispatch_release(queue);

[pool drain];
```

首先我们创建一个 __block semaphore，并将其资源初始值设置为 0 (不能少于 0)，在这里表示任务还没有完成，没有资源可用主线程不要做事情。然后在 block 任务完成之后，使用 dispatch_semaphore_signal 增加 semaphore 计数（可理解为资源数），表明任务完成，有资源可用主线程可以做事情了。而主线程中的 dispatch_semaphore_wait 就是减少 semaphore 的计数，如果资源数少于 0，则表明资源还可不得，我得按照 FIFO（先等先得）的规则等待资源就绪，一旦资源就绪并且得到调度了，我再执行。

6 示例：

下面我们来看一个按照 FIFO 顺序执行并用 semaphore 同步的例子：先将数组求和再依次减去数组。

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

initData();

__block int sum = 0;

// Create a semaphore with 0 resource
//
__block dispatch_semaphore_t sem = dispatch_semaphore_create(0);
__block dispatch_semaphore_t taskSem = dispatch_semaphore_create(0);

// create dispatch semaphore
//
dispatch_queue_t queue = dispatch_queue_create("StudyBlocks", NULL);

dispatch_block_t task1 = ^(void) {
    int s = 0;
    for (int i = 0; i < Length; i++)
        s += data[i];
    sum = s;
```

```
    NSLog(@" >> after add: %d", sum);

    dispatch_semaphore_signal(taskSem);
};

dispatch_block_t task2 = ^(void) {
    dispatch_semaphore_wait(taskSem, DISPATCH_TIME_FOREVER);

    int s = sum;
    for (int i = 0; i < Length; i++)
        s -= data[i];
    sum = s;

    NSLog(@" >> after subtract: %d", sum);
    dispatch_semaphore_signal(sem);
};

dispatch_async(queue, task1);
dispatch_async(queue, task2);

// wait for the semaphore: wait until resource is ready.
//
dispatch_semaphore_wait(sem, DISPATCH_TIME_FOREVER);

dispatch_release(taskSem);
dispatch_release(sem);
dispatch_release(queue);

[pool drain];
```

在上面的代码中，我们利用了 dispatch_queue 的 FIFO 特性，确保 task1 先于 task2 执行，而 task2 必须等待直到 task1 执行完毕才开始干正事，主线程又必须等待 task2 才能干正事。 这样我们就可以保证先求和，再相减，然后再让主线程运行结束这个顺序。

7，使用 dispatch_apply 进行并发迭代：

对于上面的求和操作，我们也可以使用 dispatch_apply 来简化代码的编写：

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

```
initData();

dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

__block int sum = 0;
__block int *pArray = data;

// iterations
//
dispatch_apply(Length, queue, ^(size_t i) {
    sum += pArray[i];
});

NSLog(@" >> sum: %d", sum);

dispatch_release(queue);

[pool drain];
```

注意这里使用了全局 dispatch_queue。

dispatch_apply 的定义如下：
dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)(size_t));

参数 iterations 表示迭代的次数，void (^block)(size_t) 是 block 循环体。这么做与 for 循环相比有什么好处呢？答案是：并行，这里的求和是并行的，并不是按照顺序依次执行求和的。

8, dispatch group
我们可以将完成一组相关任务的 block 添加到一个 dispatch group 中去，这样可以在 group 中所有 block 任务都完成之后，再做其他事情。比如 6 中的示例也可以使用 dispatch group 实现：

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

initData();

__block int sum = 0;

// Create a semaphore with 0 resource
//
__block dispatch_semaphore_t taskSem = dispatch_semaphore_create(0);
```

```
// create dispatch semaphore
//
dispatch_queue_t queue = dispatch_queue_create("StudyBlocks", NULL);
dispatch_group_t group = dispatch_group_create();

dispatch_block_t task1 = ^(void) {
    int s = 0;
    for (int i = 0; i < Length; i++)
        s += data[i];
    sum = s;

    NSLog(@" >> after add: %d", sum);

    dispatch_semaphore_signal(taskSem);
};

dispatch_block_t task2 = ^(void) {
    dispatch_semaphore_wait(taskSem, DISPATCH_TIME_FOREVER);

    int s = sum;
    for (int i = 0; i < Length; i++)
        s -= data[i];
    sum = s;

    NSLog(@" >> after subtract: %d", sum);
};

// Fork
dispatch_group_async(group, queue, task1);
dispatch_group_async(group, queue, task2);

// Join
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

dispatch_release(taskSem);
dispatch_release(queue);
dispatch_release(group);
```

```
[pool drain];
```

在上面的代码中，我们使用 dispatch_group_create 创建一个 dispatch_group_t，然后使用语句：dispatch_group_async(group, queue, task1); 将 block 任务加入队列中，并与组关联，这样我们就可以使用 dispatch_group_wait(group, DISPATCH_TIME_FOREVER); 来等待组中所有的 block 任务完成再继续执行。

至此我们了解了 dispatch queue 以及 block 并行编程相关基本知识，开始在项目中运用它们吧，

参考资料：

Concurrency Programming Guide：

http://developer.apple.com/library/ios/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html

[深入浅出 Cocoa] 之 Bonjour 网络编程

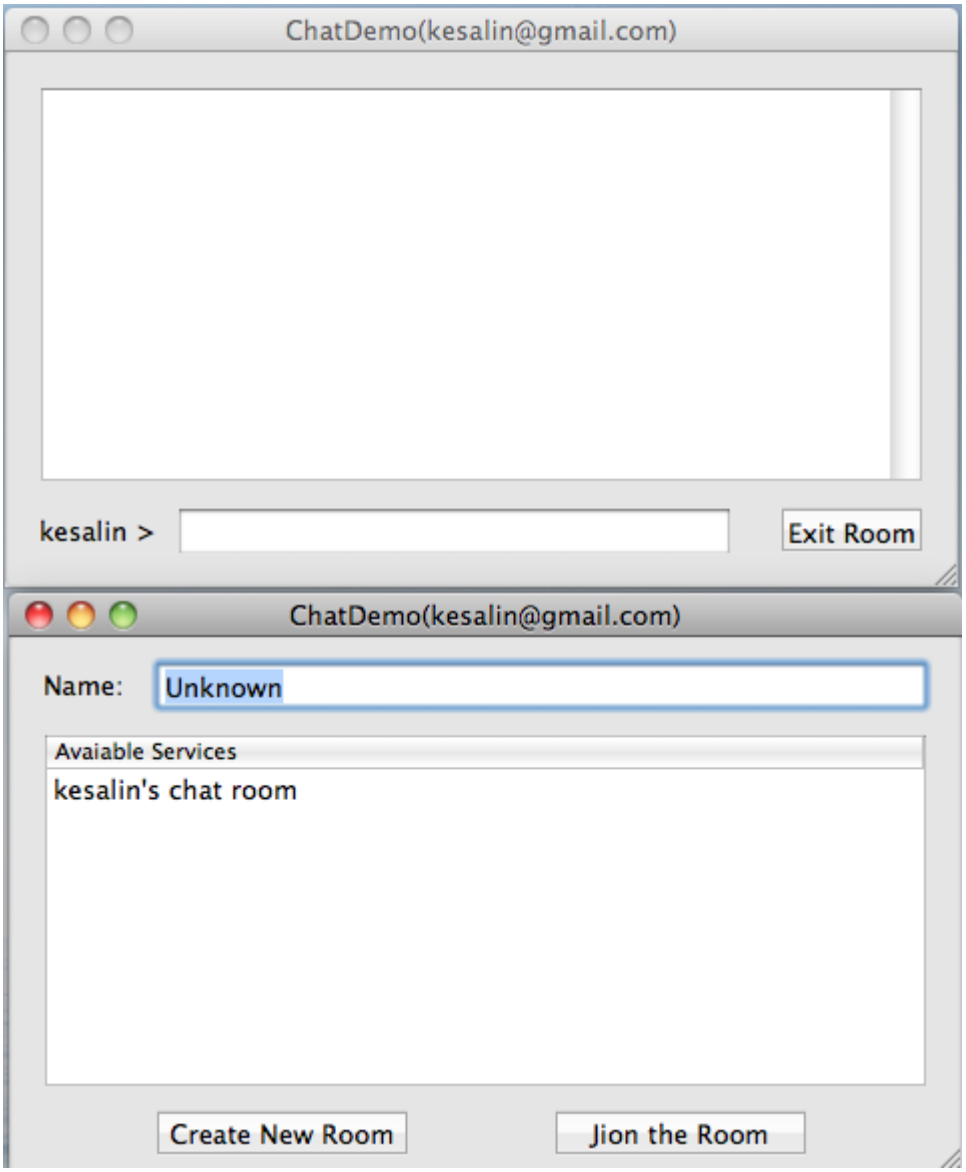
罗朝辉(http://blog.csdn.net/kesalin/)

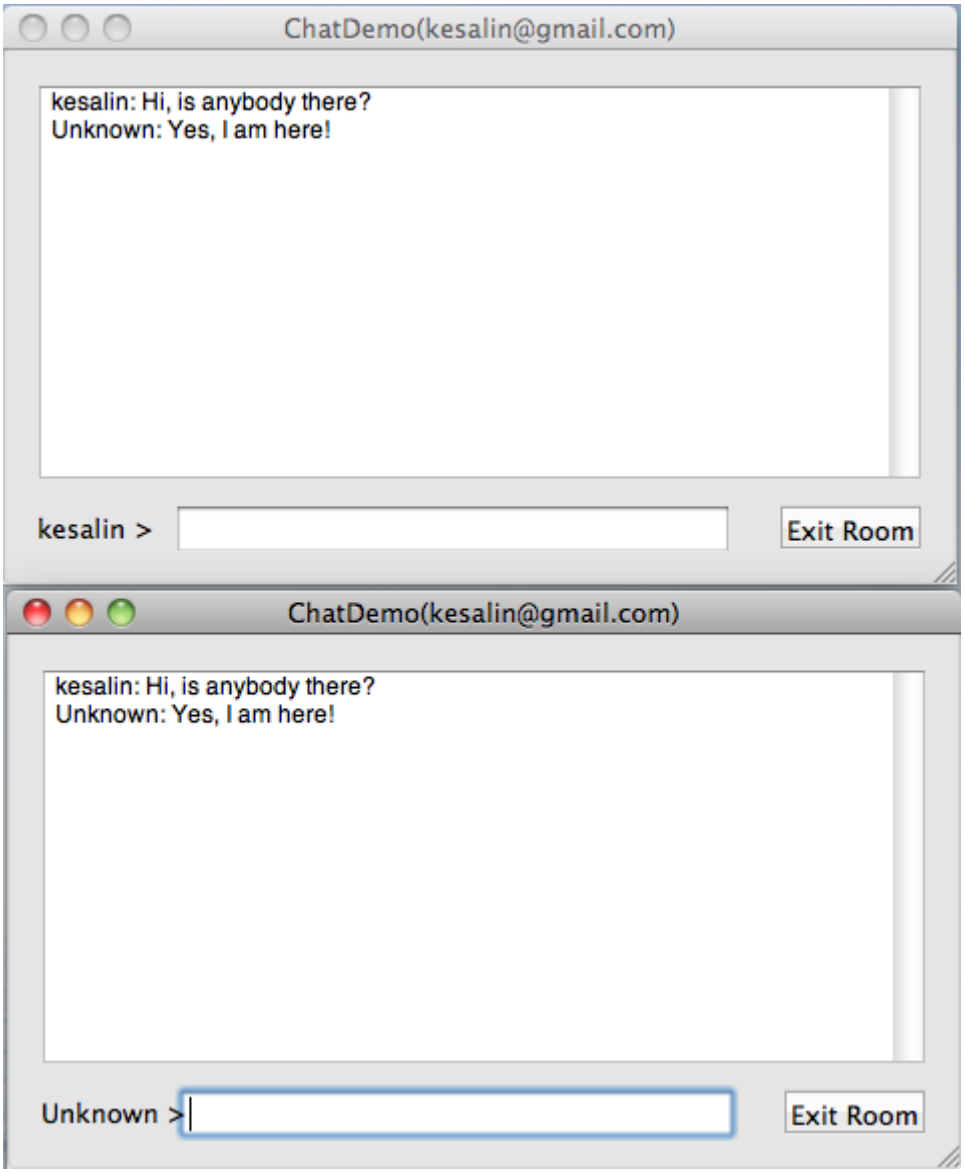
本文高度参考自 Tutorial: Networking and Bonjour on iPhone，在那个帖子里 iphone 版本的代码采用的是 MIT 开源协议，所以本例子中的 Mac 版本亦采用 MIT 开源协议。E 文较好的童鞋建议阅读原文。

本文通过使用 Bonjour 实现了一个简单的服务器/客户端聊天程序，演示了 CFSocket，NSNetService/NSNetServiceBrowser, NSInStream/NSOutStream 的用法。

代码下载：点击这里

效果图如下：





Cocoa 网络框架：

Cocoa 网络框架有三层，最底层的是基于 BSD socket 库，然后是 Cocoa 中基于 C 的 CFNetwork，最上面一层是 Cocoa 中 Bonjour。通常我们无需与 socket 打交道，我们会使用经 Cocoa 封装的 CFNetwork 和 Bonjour 来完成大多数工作。注：cocoa 很多组件都有两种实现，一种是基于 C 的以 CF 开头的类（CF=Core Foundation），这是比较底层的；另一种是基于 Obj-C 的以 NS 开头的类(NS=Next Step)，这种类抽象层次更高，易于使用。对于网络框架也一样。Bonjour 中 NSNetService 也有对应的 CFNetService，NSInputStream 有对应的 CFInputStream。

Sockets vs Streams：

Socket 相当于一条通信信道，应用程序通过创建 socket，然后使用这个 socket 连接到其他应用程序进行数据交换。我们可以通过同一个 socket 来发送数据或者接收数据。每个 socket 有一个 ip 地址和 port（通信端口，介于 1 ~ 65535 之间）。

Stream 是传送数据的单向通道，正因为是单向的，所以我们有输入/输出两种 streams：instream/outstream。stream 只是临时缓存数据，我们需要将它与文件或内存绑定，从而可以从/向文件或内存中读/写数据。在这个教程中，我们使用 stream 结合 socket 在网络上传送和接收数据。

Bonjour 简介：

Bonjour(法语中的你好)是一种能够自动查询接入网络中的设备或应用程序的协议。Bonjour 抽象掉 ip 和 port 的概念，让我们聚焦于更容易为人类思维理解的 service。通过 Bonjour，一个应用程序 publish 一个网络服务 service，然后网络中的其他程序就能自动发现这个 service，从而可以向这个 service 查询其 ip 和 port，然后通过获得的 ip 和 port 建立 socket 链接进行通信。通常我们是通过 NSNetService 和 NSNetServiceBrowser 来使用 Bonjour 的，前者用于建立与发布 service，后者用于监听查询网络上的 service。

同步与异步操作：

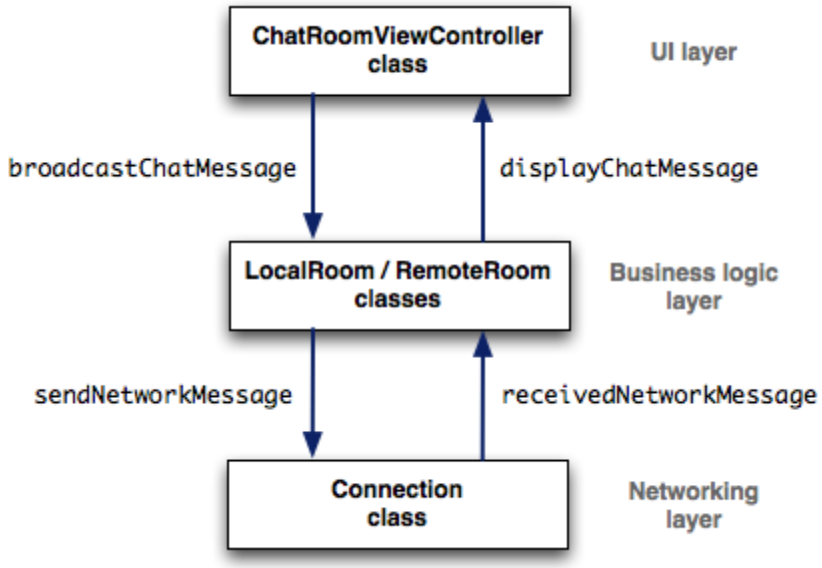
大多数网络操作是阻塞模式的，比如链接的建立，等待接收数据，或发送数据给网络另一端。因此如果我们不进行异步处理的话，当在进行网络通信时，我们的 UI 机会被阻塞。有两种办法来处理阻塞问题：启用多个线程或更有效地利用当前线程。在这个例子中，我们使用后一种办法，我们通过 cocoa 提供的 run loop 来做这个事情，其工作原理是：将网络消息当作普通的事件丢到当前的 run loop 中，从而我们可以异步处理它们。

Run loops 简介：

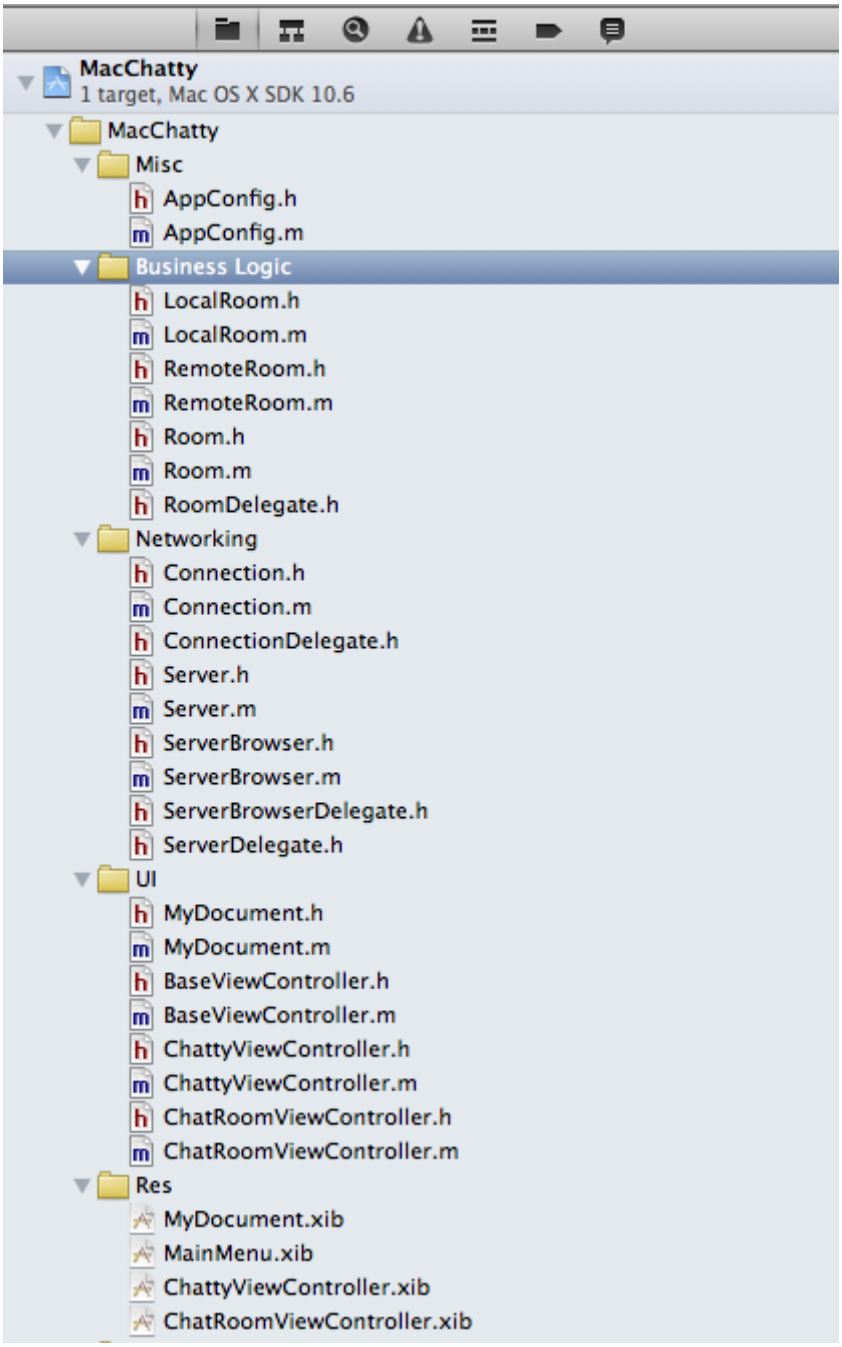
run loop 是 thread 中的消息处理循环，有事件来则处理，无事件则啥也不做。cocoa 中的 run loop 可以处理用户 UI 消息，网络连接消息，timer 消息等。我们也可以添加其他的消息来源，如 socket 和 stream，从而让 run loop 也可以处理它们。

程序框架：

理论介绍得差不多了，更多细节，请翻阅官方文档。下面我们来看看整个程序的框架设计图：



从上图可以清晰地看出，程序分为三个主要模块：UI 模块，逻辑模块，网络模块。下面我们打开工程，看看代码实现：



从工程图可以看出，代码结构相当清晰，所有的类被分为四个 group：

Networking: 网络相关的代码，包括 socket 的创建，连接的建立，service 的 publish 和 browser；

Business Logic：业务逻辑相关代码。在这个例子中，我们通过 room service 来提供聊天服务。我们通过建立一个 localroom 来创建服务器，并发布一个 room service，客户端（remoteroom）能够连接到一个已有的 room service，从而加入该 room 进行对话活动。

UI :在这个例子中，UI 很简单，只有两个 view，一个显示当前网络中的 service 列表，另一个显示 room，以及在该 room service 上进行的对话。

Misc:一个辅助类，用于存储用户设定名称。

网络类:

Server class:创建 server，并发布 service；

Connection class:决议 service；与服务器建立连接；通过 socket stream 交换数据；

ServerBrowser class:查询可用的 service；

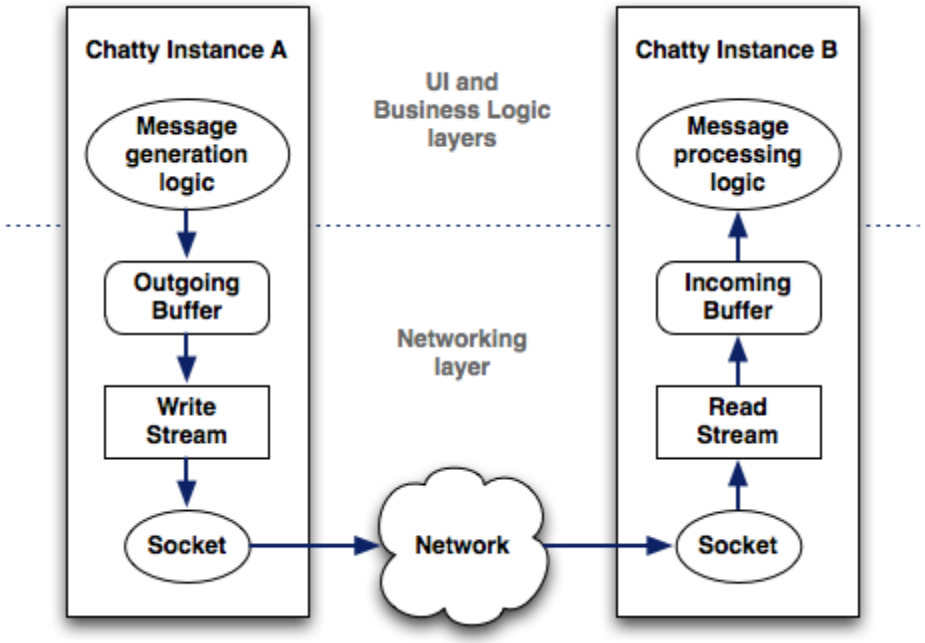
Room 类：

Room class: Room 基类

LocalRoom class: 创建服务器，发布 service，相应客户端的连接请求

RemoteRoom: 连接到服务器已有的 service，

网络数据传输过程：



从上图可以看出，数据从 A 的逻辑层，经 outgoing buffer 写入 write stream，然后经 socket 通过网络传输到 B 的网络层，然后 B 端的 read stream 从 socket 中读取数据，写入 incoming buffer，然后在 B 的逻辑层以及 UI 上显示出来。

用户交互操作都在 UI layer 上进行，当用户通过 broadcastChatMessage:fromUser: 发送一条聊天信息，由逻辑层来决定是发送给服务器（由 Remote room 处理），还是发送给连接到服务器自身的所有客户端（由 Local room 处理）。当从网络连接接收到一条聊天信息时，逻辑层会得到通知，客户端只会简单地将消息显示在 UI 上，而服务器首先将收到的聊天信息转发给所有连接到它的客户端，然后将该信息在 UI 上显示出来。

Socket+Streams+Buffers = Connection

Connection 类对一些的交互进行了封装:

两个 socket stream，一个用来写入，一个用来读取；两块 data buffer，每个 socket stream 对应一个 data buffer；以及各种控制 flag 和值

因为 stream 是单向的，所以我们需要为每一个 socket 建立两个 stream，一个用来从 socket 读取数据，一个用来向 socket 写入数据。我们在 connect 和 setupSocketStreams 中初始化它们。

在本例中，我们通过两种方式来创建 socket：

1，（客户端）通过创建 socket 连接到指定 ip 和 port 的服务器；

2，（服务器）通过接收来自客户端的连接请求，在这种情况下，OS 会自动创建一个用于响应的 socket，并通过 native socket

handle 传递给我们使用；

无论 socket 是由哪种方式建立的，我们都是通过相同的代码 setupSocketStreams 来初始化 stream。

创建 server

聊天至少需要同时运行两个 MacChatty 终端，其中至少有一个作为服务器，其他终端才能作为客户端连接到服务器进行对话。作为服务器的终端，需要创建一个 socket 来监听(listen)其他终端的连接请求（请参考 Sever class 中的 listeningSocket）。这项工作是在 Server 类中的 createServer 中完成的。

客户端如何知道怎样连接到服务器呢？每一个网络终端必须有独一无二的 ip 和 port，ip 地址是由动态获取的或由用户设定的，因此我们在这里无需操心 ip 地址问题，因此在代码中我们使用了 INADDR_ANY。那又如何设定我们想要监听的 port 呢？一些服务必须监听约定的 port 才能工作，比如 80，20， 21 等端口都是有约定用途的。在这里我们把端口设定问题交给 OS 来处理，OS 会为我们设定一个没有被占用的 port。为了实现这个目的，我们传入 port 为 0。为了让其他客户端能够连接到服务器，我们需要告知其他客户端服务器实际使用的 port，因此，我们在 createServer 方法 PART 3 中获取实际使用 port。

```
    //// PART 3: Find out what port kernel assigned to our socket
    //
    // We need it to advertise our service via Bonjour
    NSData *socketAddressActualData = [(NSData *)CFSocketCopyAddress(listeningSocket) autorelease];


    // Convert socket data into a usable structure
    struct sockaddr_in socketAddressActual;
    memcpy(&socketAddressActual, [socketAddressActualData bytes], [socketAddressActualData length]);


    self.port = ntohs(socketAddressActual.sin_port);
```

然后在 PART 4 中，我们将 listening socket 注册为 application run loop 的消息源，这样当有新连接到来的时候， OS 就会调用 serverAcceptCallback 这个回调函数通知我们。

```
    //// PART 4: Hook up our socket to the current run loop
    //
    CFRunLoopRef currentRunLoop = CFRunLoopGetCurrent();
    CFRunLoopSourceRef runLoopSource = CFSocketCreateRunLoopSource(kCFAllocatorDefault, listeningSocket, 0);
    CFRunLoopAddSource(currentRunLoop, runLoopSource, kCFRunLoopCommonModes);
    CFRelease(runLoopSource);
```

在 serverAcceptCallback 回调处理中，我们创建一个新的 Connection 对象，然后将它与 OS 自动创建的响应新连接的 socket 绑定起来。然后再将这个 Connection 对象传递给 Server delegate。

```
// Handle new connections
- (void) handleNewNativeSocket:(CFSocketNativeHandle)nativeSocketHandle
{
```

```
    Connection* connection = [[[Connection alloc] initWithNativeSocketHandle:nativeSocketHandle] autorelease];

    // In case of errors, close native socket handle
    if ( connection == nil ) {
        close(nativeSocketHandle);
        return;
    }

    // finish connecting
    BOOL succeed = [connection connect];
    if ( !succeed ) {
        [connection close];
        return;
    }

    // Pass this on to our delegate
    [delegate handleNewConnection:connection];
}



// This function will be used as a callback while creating our listening socket via 'CFSocketCreate'
static void serverAcceptCallback(CFSocketRef socket, CFSocketCallBackType type, CFDataRef address, const void *data,
void *info)
{
    // We can only process "connection accepted" calls here
    if ( type != kCFSocketAcceptCallBack ) {
        return;
    }

    // for an AcceptCallBack, the data parameter is a pointer to a CFSocketNativeHandle
    CFSocketNativeHandle nativeSocketHandle = *(CFSocketNativeHandle *)data;

    Server *server = (Server *)info;
    [server handleNewNativeSocket:nativeSocketHandle];

    NSLog(@" >> server accepted connection with socket %d", nativeSocketHandle);
}
```

通过 Bonjour 发布服务

Bonjour 并非在网络查找服务的唯一途径，但它是最容易使用的方法之一。我们在 publishService 方法中创建一个 NSNetService 对象来发布服务。我们根据服务类型在网络查找感兴趣的服务，本聊天服务使用“_chatty._tcp.”作为服务类型。在同一网络中，服务类型名必须唯一，这样才能精准定位服务，而不至于引发冲突。

Bonjour 操作也如 socket 一样需要异步进行，以避免长时间阻塞主线程。因此在实际发布服务时，我们将发布任务交给当前 run loop 去调度，然后设定其 delegate，由 delegate 来处理相关事件：“Publishing succeeded”, “Publishing failed”等。

```
- (BOOL) publishService
{
    // come up with a name for our chat room
    NSString* chatRoomName = [NSString stringWithFormat:@"%@'s chat room", [[AppConfig sharedInstance] name]];

    // create new instance of netService
            self.netService = [[NSNetService alloc] initWithDomain:@"" type:@"_chatty._tcp." name:chatRoomName
port:self.port];
            if (self.netService == nil)
                        return NO;

    // Add service to current run loop
            [self.netService scheduleInRunLoop:[NSRunLoop currentRunLoop] forMode:NSRunLoopCommonModes];

    // NetService will let us know about what's happening via delegate methods
            [self.netService setDelegate:self];

    // Publish the service
            [self.netService publish];

    return YES;
}
```

通过 Bonjour 查询服务

我们在 ServerBrowser 类中实现 Bonjour 查询网络服务的功能。我们创建一个 NSNetServiceBrowser 对象来查询类型为“_chatty._tcp.”的服务。当前网络中发现有服务被添加到或移除时，NSNetServiceBrowser 的 delegate 即我们的 ServerBrowser 就能得到通知，以进行相应的逻辑处理：更新服务列表，刷新 UI 等。

```
// New service was found
- (void) netServiceBrowser:(NSNetServiceBrowser *)netServiceBrowser
            didFindService:(NSNetService *)netService
                moreComing:(BOOL)moreServicesComing
```

```
{
    // Make sure that we don't have such service already (why would this happen? not sure)
    if ( ! [servers containsObject:netService] ) {
        // Add it to our list
        [servers addObject:netService];
    }

    // If more entries are coming, no need to update UI just yet
    if ( moreServicesComing ) {
        return;
    }

    // Sort alphabetically and let our delegate know
    [self sortServers];

    [delegate updateServerList];
}



// Service was removed
- (void)netServiceBrowser:(NSNetServiceBrowser *)netServiceBrowser
         didRemoveService:(NSNetService *)netService
               moreComing:(BOOL)moreServicesComing
{
    // Remove from list
    [servers removeObject:netService];

    // If more entries are coming, no need to update UI just yet
    if ( moreServicesComing ) {
        return;
    }

    // Sort alphabetically and let our delegate know
    [self sortServers];

    [delegate updateServerList];
}
```

通过 Bonjour 决议服务

当用户选择其中一个 chat room，并加入其中时，客户端将会连接到发布该 chat room 服务的服务器。这个连接过程在 ChattyViewController 类的 joinChatRoom: 方法中实现。首选我们通过选择的 NSNetService 发送 resolveWithTimeout: 消息来进行决议应该连接到哪个服务器（请参考 Connection 类的 connect 方法中最后一种情形），同时设定 NSNetService 的 delegate 来响应决议相关的事件：didNotResolve: 和 netServiceDidResolveAddress:。当决议完成之后，在 netServiceDidResolveAddress: 方法中，我们可以建立到服务的 socket 连接并创建用于数据传输的 stream 了。

```
// Called when net service has been successfully resolved
- (void)netServiceDidResolveAddress:(NSNetService *)sender
{
    if ( sender != netService ) {
        return;
    }

    // Save connection info
    self.host = netService.hostName;
    self.port = netService.port;

    // Don't need the service anymore
    self.netService = nil;

    // Connect!
    if ( ![self connect] ) {
        [delegate connectionAttemptFailed:self];
        [self close];
    }
}
```

至此，Bonjour 网络编程介绍就结束了，代码中的注释相当详细，细节就不多罗嗦了。

参考资料：

Tutorial: Networking and Bonjour on iPhone：
http://mobileorchard.com/tutorial-networking-and-bonjour-on-iphone/
Introduction to Bonjour Overview：
http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Introduction.html
Introduction to NSNetServices and CFNetServices Programming Guide：
http://developer.apple.com/library/mac/#documentation/Networking/Conceptual/NSNetServiceProgGuide/Introduction.html#//apple_ref/doc/uid/TP40002736

[深入浅出 Cocoa] 之 Framework

罗朝辉(http://blog.csdn.net/kesalin/)

Framework 简介

Mac OS X 扩展了 framework 的功能，让我们能够利用它来共享代码和资源。framework 在概念上有点像 Window 下的库，但是比库更加强大，通过 framework 我们可以共享所有形式的资源，如动态共享库，nib 文件，图像字符资源以及文档等。系统会在需要的时候将 framework 载入内存中，多个应用程序可以同时使用同一个 framework，而内存中的拷贝只有一份。一个 framework 同时也是一个 bundle，我们可以在 finder 里浏览其内容，也可以在代码中通过 NSBundle 访问它。利用 framework 我们可以实现动态或静态库的功能。与动态/静态库相比，framework 有如下优势：

第一， framework 能将不同类型的资源打包在一起，使之易于安装，卸载与定位；

第二， framework 能够进行版本管理，这使得 framework 能不断更新并向后兼容；

第三， 在同一时间，即使有多个应用程序使用同一 framework，但在内存中只有一份 framework 只读资源的拷贝，这减少了对内存的占用;

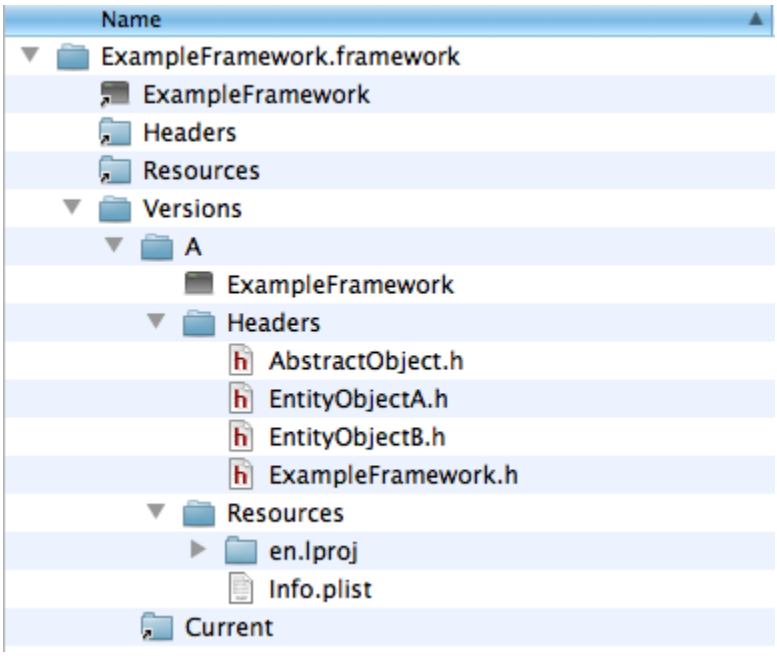
Framework 的结构

下面是一个带有 A，B 两个版本和一个 resources 目录的 framework 结构，并设定当前版本为 B：

```
MyFramework.framework/
    Headers      -> Versions/Current/Headers
    MyFramework  -> Versions/Current/MyFramework
    Resources    -> Versions/Current/Resources
    Versions/
        A/
            Headers/
                MyHeader.h
            MyFramework
            Resources/
                English.lproj/
                    Documentation
                    InfoPlist.strings
                Info.plist
        B/
            Headers/
                MyHeader.h
            MyFramework
            Resources/
                English.lproj/
                    Documentation
```

```
                InfoPlist.strings
            Info.plist
    Current -> B
```

结合上面的结构，下面我们来看本例中 ExampleFramework 的结构图：



Framework 存放位置

在 Mac OS 中有三个级别的位置来存放 framework。一般我们自己编写的 framework 都应该是应用程序级别。

1，系统级，/Library/Frameworks，放置到该级别，这需要管理员权限，整个系统都可以共享使用该级别的 framework；

2，用户级，/Users/用户名/Library/Frameworks，拥有用户权限的应用程序都可以共享使用该级别的 framework；

3，应用程序级。

在应用程序中内嵌 Framework

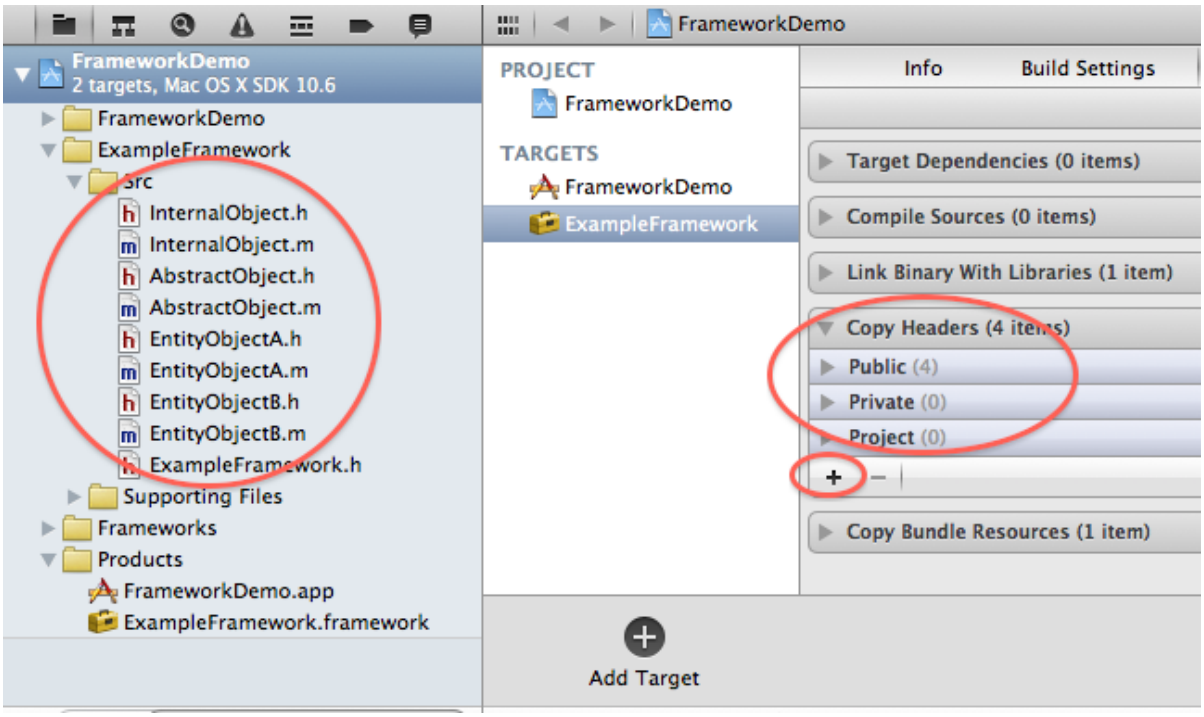
1，创建 Framework

新建一个名为 FrameworkDemo 的 Cocoa application 工程，然后选中项目名，向其中添加名为 ExampleFramework 的

Cocoa Framework。

2，添加内容

向 Framework 中添加源代码（请下载源代码），并导出需要向外部公开的头文件。

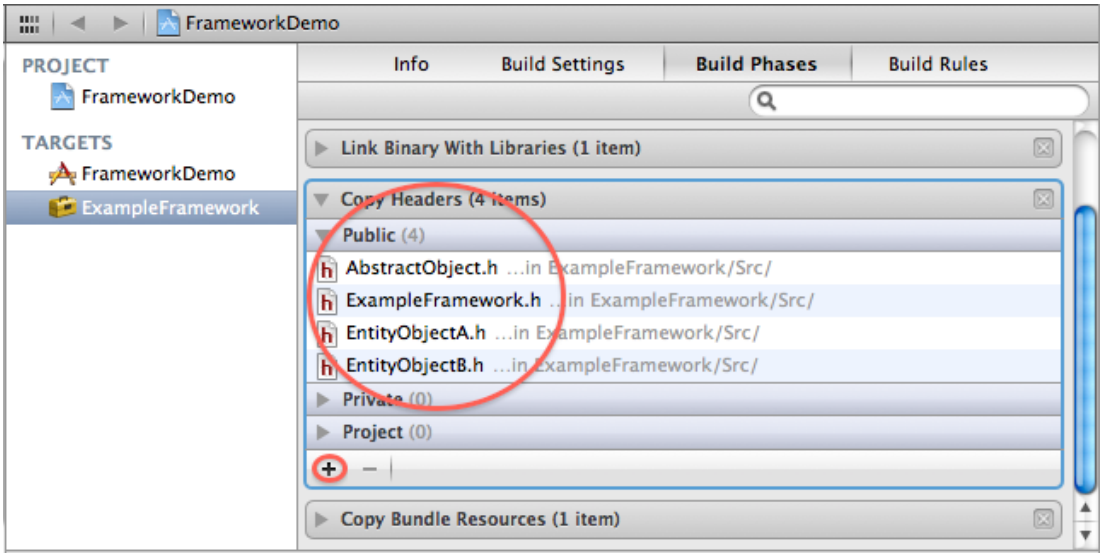


导出头文件有一些技巧：

1，如果有我们不想向用户公开的类名出现在必须公开的头文件中，我们可以使用 id 替代该类名或使用 @class 前置申明来避免导出该类的头文件，在本例中使用 id 替代 InternalObject，从而避免导出 InternalObject 类的头文件。

2，如果需要导出多个头文件，常见的做法是新建一个与 framework 同名的 .h 文件，将需要导出的头文件包含到该头文件中

来。如本例中的 ExampleFramework.h。

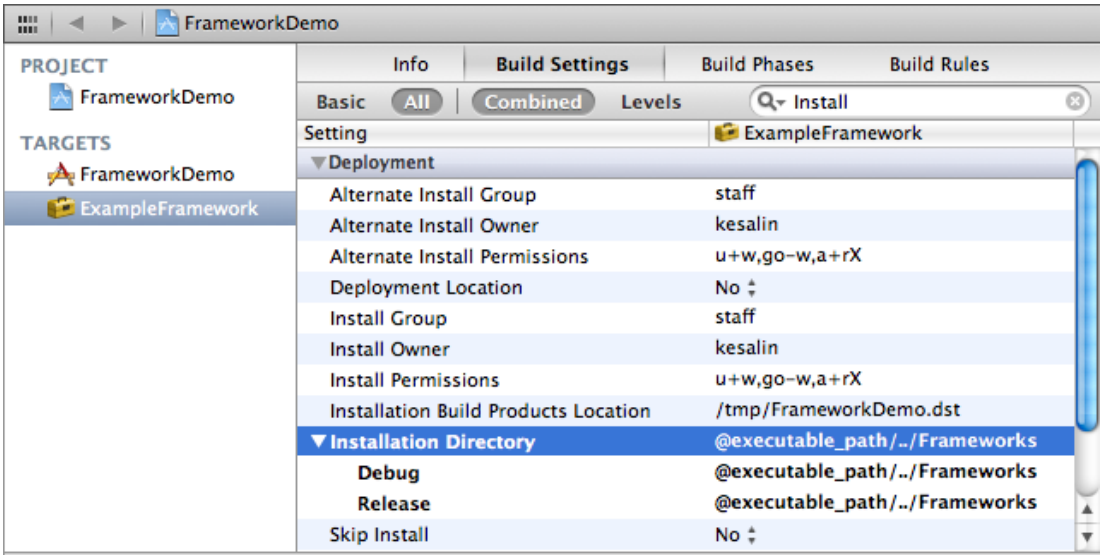


3，修改 framework build 选项

我们在使用自己编写的库时，常碰到下面的编译错误：

```
Library not loaded: path/to/framework

Referenced from: path/to/app/

Reason: image not found
```

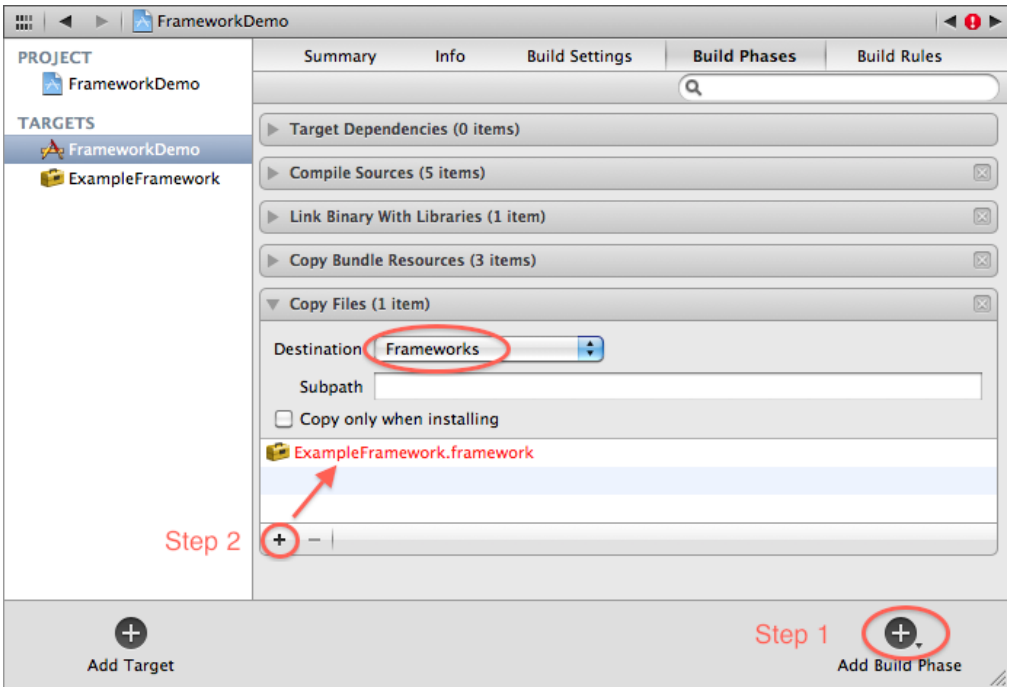
这多半是由于 framework 的 Installation Directory 编译选项设置不正确，导致应用程序无法正确定位 framework 所致。这需要我们设置编译选项 Installation Directory 为 @executable_path/../Frameworks。



4，使用 framework

至此，framework 编写完成，下面我们来在 FrameworkDemo 中来使用它。首先我们需要将 ExampleFramework 导入到 FrameworkDemo 中来，这样 FrameworkDemo 在运行时才能定位该 framework。新建一个 Add copy files 型的 build

phase，设置其 destination 为 framework，加入已经编写好的 ExampleFramework。



导入 framework 之后，我们就可以在工程中使用该 framework 了。编写如下代码：

```
//
//  FrameworkDemoAppDelegate.m
//  FrameworkDemo
//
//  Created by kesalin on 11-10-16.
//  Copyright 2011 年 kesalin@gmail.com. All rights reserved.
//

#import "FrameworkDemoAppDelegate.h"
#import <ExampleFramework/ExampleFramework.h>

@implementation FrameworkDemoAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
        EntityObjectA *objectA = [[EntityObjectA alloc] init];
            EntityObjectB *objectB = [[EntityObjectB alloc] init];

            NSLog(@"Object A called: %@", [objectA methodOne]);
            NSLog(@"Object B called: %@", [objectA methodTwo]);
```

```
            NSLog(@"Object B called: %@", [objectB methodOne]);
            NSLog(@"Object B called: %@", [objectB methodTwo]);
}

@end
```

注意：我们使用 framework 的方式为 ：framework 名/framework 名.h，这是约定的常规做法，Cocoa 自带的 framework 也都遵守这一约定，所以我们自己编写的库最后也遵守这一约定。

5，编译运行

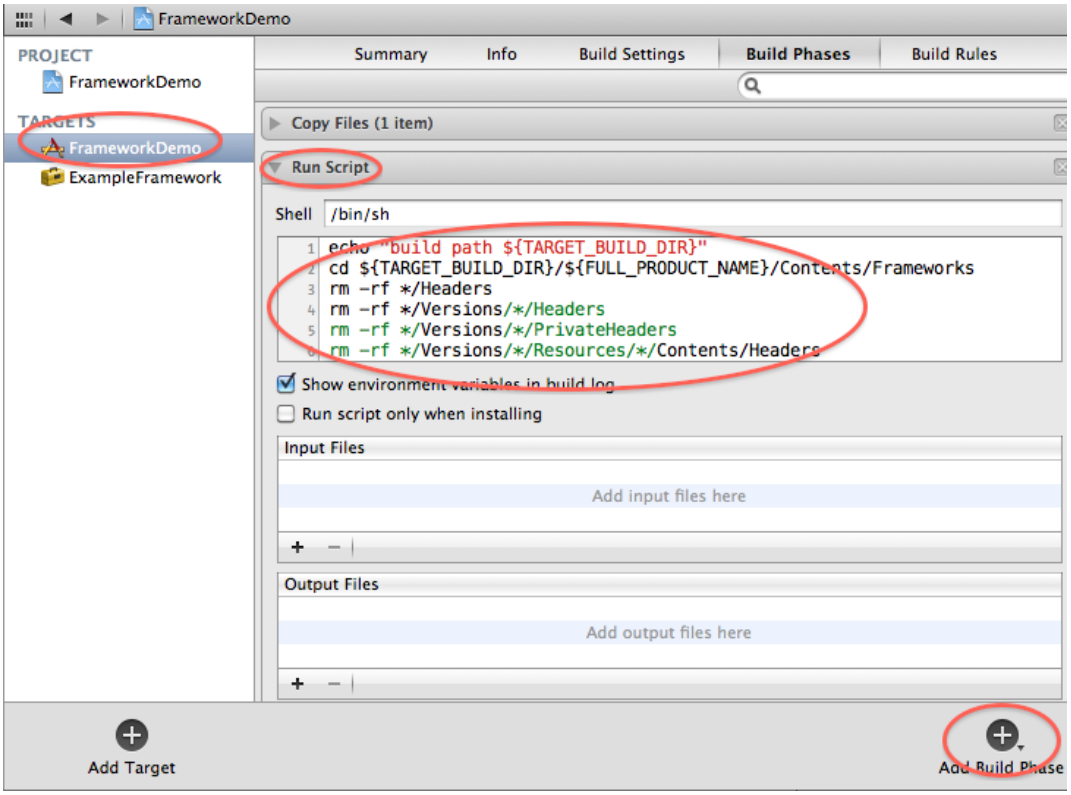
至此，工作完成，编译运行，应当输出如下：

```
Object A called: EntityObjectA:methodOne
Object B called: EntityObjectA:methodTwo - InternalObject:description
Object B called: EntityObjectB:methodOne
Object B called: EntityObjectB:methodTwo - InternalObject:description
```

6，清除冗余文件

这时可选项，且只对使用内嵌 framework 的应用程序有效。当我们拷贝导入 framework 之后，应用程序 bundle 已经拷贝了一份 framework，那么原本编译生成的那一份 framework 就变得多余了，我们可以将其清理掉。在使用内嵌 framework 的应用程序的 build phases 中加入 run script phase，脚本内容如下：

```
echo "build path ${TARGET_BUILD_DIR}"
cd ${TARGET_BUILD_DIR}/${FULL_PRODUCT_NAME}/Contents/Frameworks
rm -rf */Headers
rm -rf */Versions/*/Headers
rm -rf */Versions/*/PrivateHeaders
rm -rf */Versions/*/Resources/*/Contents/Headers
```

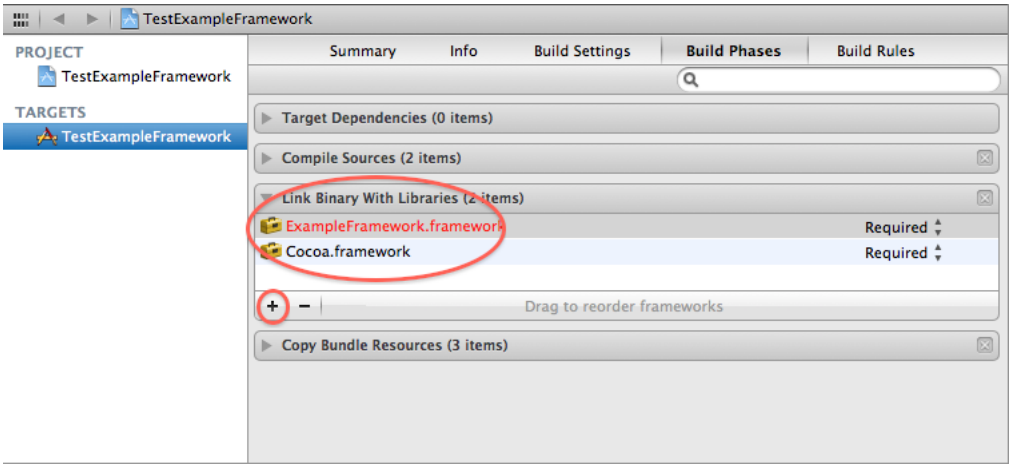


使用外部 framework

上面的示例是在应用程序内嵌 framework，供应用程序本身使用，很多时候，我们是使用第三方编写的 framework，下面接着来演示如何将 ExampleFramework 当做外部 framework。

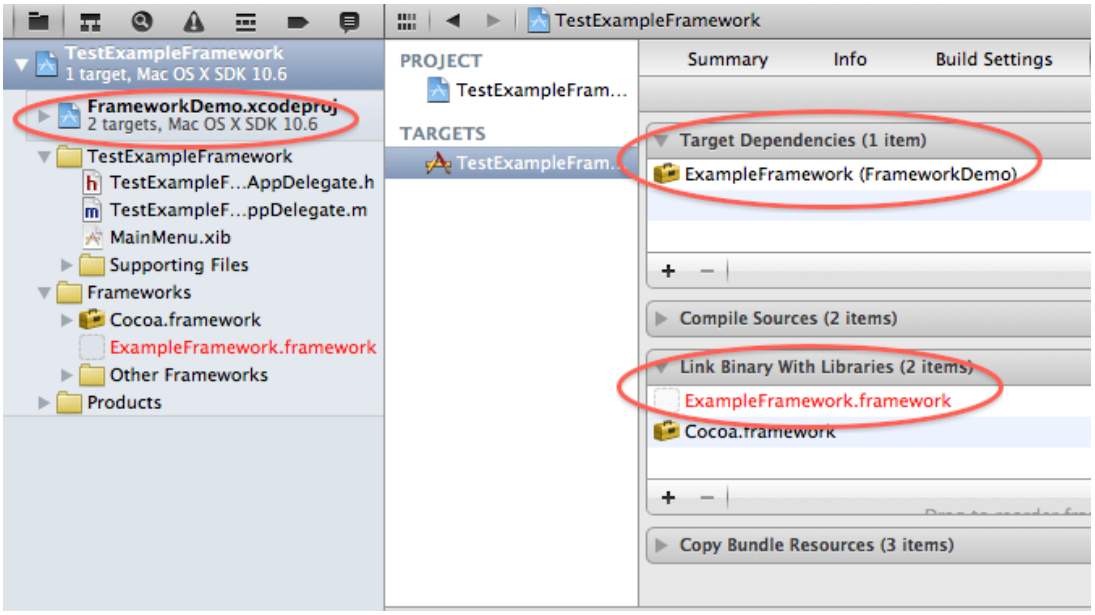
1，新建名为 TestExampleFramework 的 Cocoa Application 程序，在 TestExampleFrameworkAppDelegate.m 中添加如上步骤 4 中使用 framework 的代码。

2，编译运行，这时会报找不到头文件，类名的错误。这时因为我们还没有导入 framework。在 Build Phase 的 Link Binary With Libraries 中加入生成好的 ExampleFramework，该 framework 的默认生成路径在： /用户名/Library/Developer/XCode/DerivedData/FrameworkDemo-XXXX/Build/Products/Debug/下。至此，编译运行，输出应当如上步骤 5 相同。



此外还有一种方式使用第三方 framework，如果我们拥有第三方 framework 的源代码工程，想在我们的工程中编译该 framework，并使用它。我们可以将第三方 framework 的工程文件加入我们自己的工程，并在 Target Dependencies 和 Link Binary With Libraires 加入第三方 framework，这样我们就可以使用该 framework 了。

如下图所示：



[深入浅出 Cocoa] 之 Plugin

罗朝辉(http://blog.csdn.net/kesalin)

在前文 深入浅出 Cocoa 之 Framework 中讲解了 Framework，接下来讲解 plugin。如果你对 Framework 还不太熟悉的话，请阅读那篇文中，在本例中使用到了 framework，并在本文中没有详细讲述其创建和使用过程。

本文代码下载：点击这里

为什么要引入插件？

我们知道编译程序时，会连接相关 framework，通常我们所连接的框架是 Foundation 和 Application 框架。当程序启动运行时，每个被连接到的 framework 都会被加载到该程序的 objc 运行时环境中。如果我们想向正在运行的程序加载新的 framework，那该怎么办呢？答案之一就是使用 plugin 机制。cocoa 的 plugin 机制通常由 NSBundle 类来实现，而实现动态加载的功能由函数 objc_addClass 来完成。一般我们无需与 objc_addClass 这个函数打交道，我们使用 NSBundle 来完成绝大部分与 plugin 相关的工作。

plugin 机制能够让我们开发出高度模块化，可定制以及可扩展的应用程序，并能够让第三方为该应用程序添加新特性。想必很多人都熟悉 Eclipse，Eclipse 的 plugin 机制就非常方便与强大。

NSBundle 简介

束（bundle）是文件系统中的一个目录结构，它将程序会使用到的资源打包在一起。这些资源可包括编译好的代码，nib 文件，配置文件，图像，声音，本地化资源等等。束是 Mac OS X 的一个核心特性，应用程序，Framework，插件都是一个束，只是扩展名各异，如应用程序的扩展名为 .app；Framework 的扩展名是 .framework；插件的扩展名默认为 .bundle。

一个 plugin 就是一个 bundle(束)，xcode 默认以 .bundle 为扩展名。通常我们使用我们自己定义的扩展名，以便与系统或其他人编写的 plugin 区分开来。我们通过 NSBundle 来载入 bundle，并把其中经过编译的类注册到 objc 运行时中，然后我们就能在程序中使用这些类了；我们也可以使用 bundle 中的所有资源。

plugin 构架

我们可以通过多种途径来实现一个 plugin：

1，定义一个 objc protocol，让 plugin 遵守该 protocol；

2，定义一个基类，让 plugin 继承该基类；

3，定义一个 C 回调函数接口，让 plugin 实现改回调函数；

4，使用 CFPlugIn 来创建 plugin 接口；

在今天的例子中，使用的是第二种情况，这种情况稍稍复杂一些，我们需创建一个 framework 供宿主程序（使用插件的程序）和 plugin 使用，该 framework 的主要职责是提供基类接口。

plugin 的存放目录

通常 plugin 总是存放在以下三个位置：

1，应用程序名.app/Contents/Plug-ins　　这是程序的开发者存放随产品发布的插件的地方。

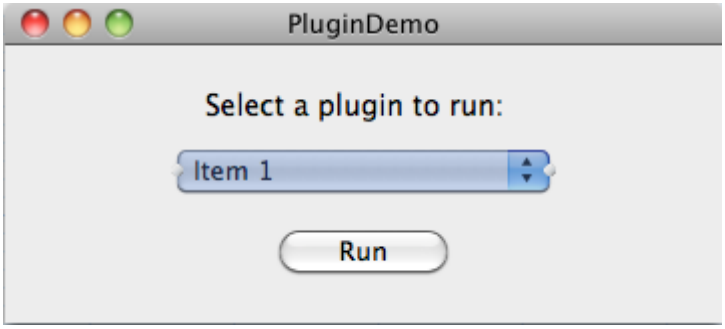
2，~/Library/Application Support/应用程序名/Plug-ins　　用户存放个人插件的地方。

3，/Library/Application Support/应用程序名/Plug-ins　　系统中供全部用户使用的插件。

在今天的例子中，使用的是第一种情况，即将插件存放在应用程序包中。

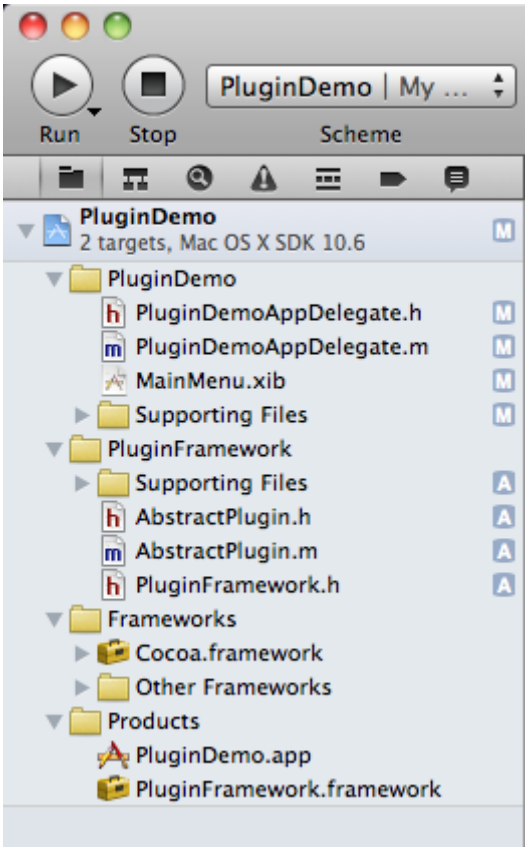
创建宿主程序

我们来创建一个名为 PluginDemo 的 cocoa application，该程序含有一个显示已安装 plugin 的 popup button 以及一个执行选中 plugin 的 button。



创建 framework

1，创建名为 PluginFramework 的 framework，向其中添加 plugin 基类：AbstractPlugin。如果你忘记怎样创建和使用 framework，请参看前文:深入浅出 Cocoa 之 Framework。



AbstractPlugin 类仅仅提供两个接口：

```
- (NSString *)name;


- (IBAction)run:(id)sender;
```

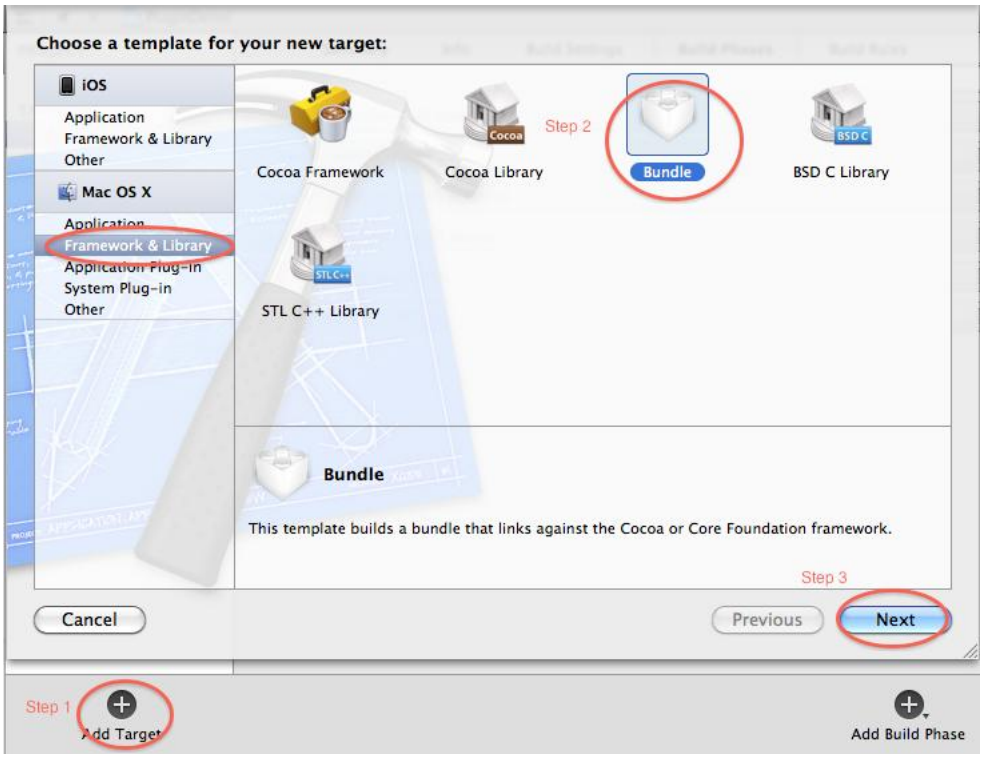
name 用来标识 plugin，run 用来供宿主程序运行插件。

2，在 PluginDemo 中连接和使用该 framework 来运行插件。如果你忘记怎样连接和使用 framework，请参看前文:深入浅出 Cocoa 之 Framework。我们在按钮响应函数中，运行选中的插件。

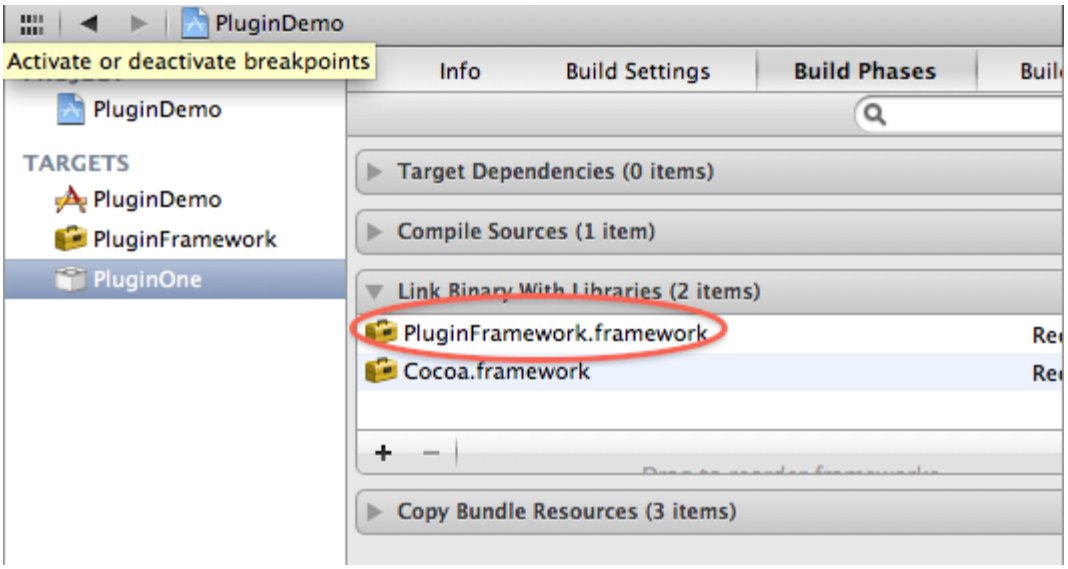
```
- (IBAction)runPlugin:(id)sender
{
    AbstractPlugin *plugin = [[pluginsController selectedObjects] lastObject];
            if (!plugin)
        return;

    [plugin run:sender];
}
```

创建 plugin

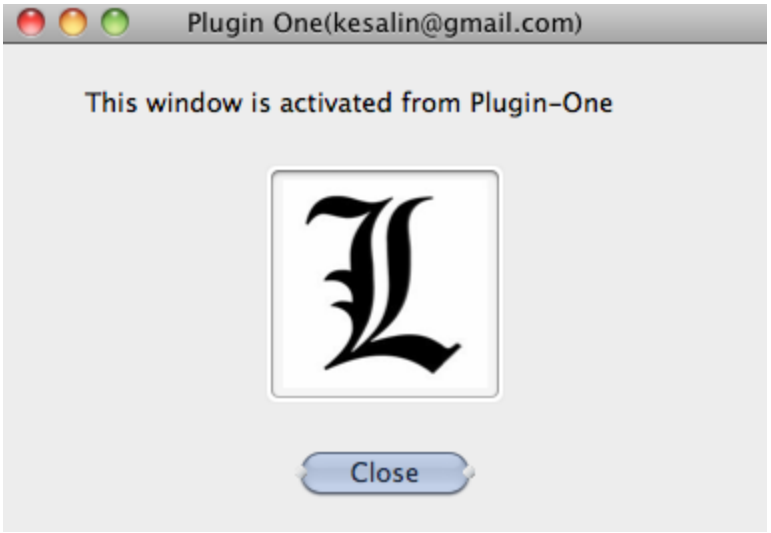
1，创建 plugin；



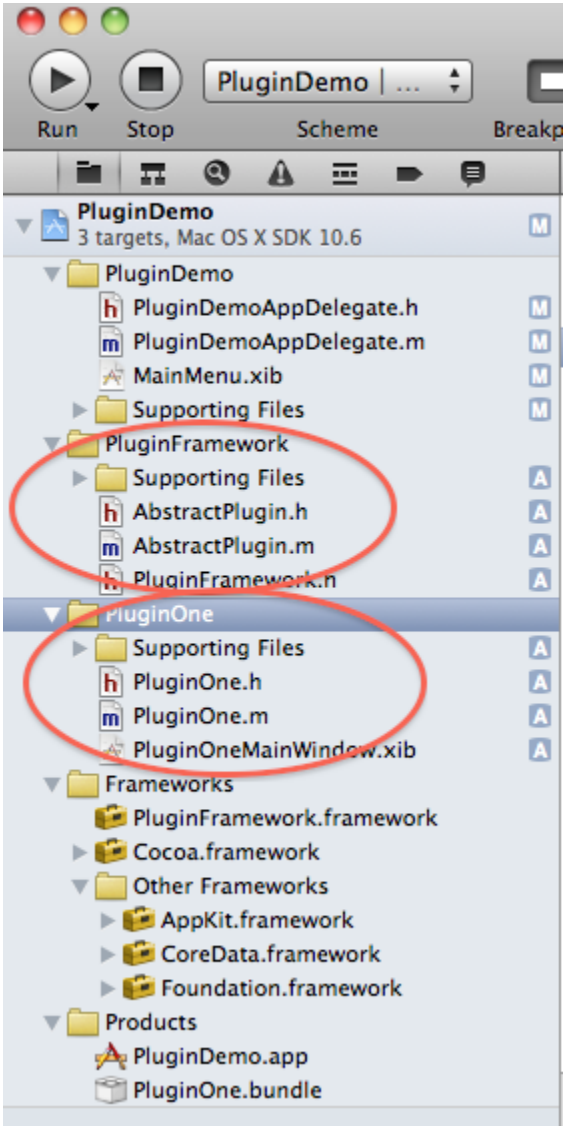
2，连接 PluginFramework；如果你忘记怎样连接和使用 framework，请参看前文:深入浅出 Cocoa 之 Framework。



3，创建 UI 界面；



4，创建继承自基类的 plugin 子类：PluginOne；



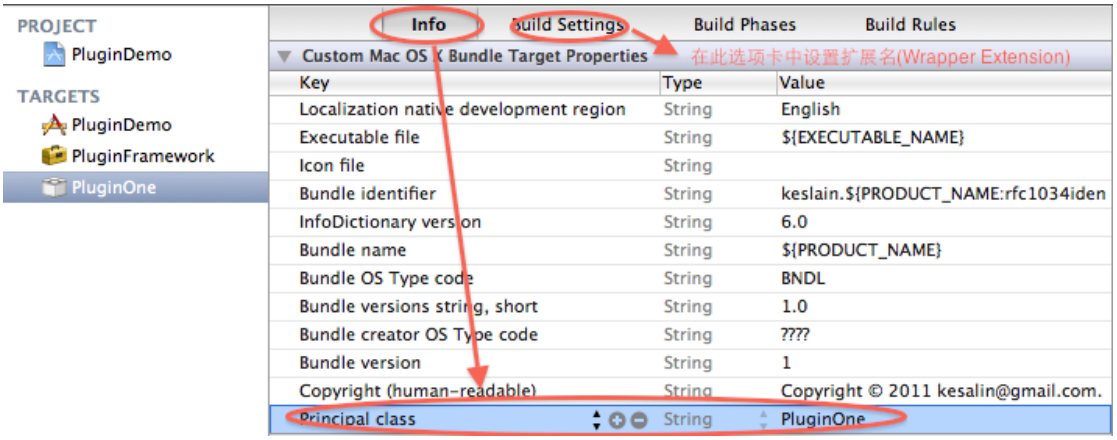
PluginOne 类继承自 AbstractPlugin，它仅仅是显示和隐藏一个 window，其实现如下：

```
#import "PluginOne.h"

@implementation PluginOne

@synthesize mainWindow;


- (id)init

{

    self = [super init];

    if (self) {

        // Initialization code here.


        [NSBundle loadNibNamed:@"PluginOneMainWindow" owner:self];

    }
```

```
    return self;
}


- (void)dealloc
{
    mainWindow = nil;
    [super dealloc];
}


- (NSString *)name;
{
          return @"Plugin One";
}


- (IBAction)run:(id)sender;
{
          [mainWindow center];
          [mainWindow makeKeyAndOrderFront:sender];
}


- (IBAction)closeWindow:(id)sender;
{
          [mainWindow orderOut:sender];
}


@end
```

5，plugin 设置

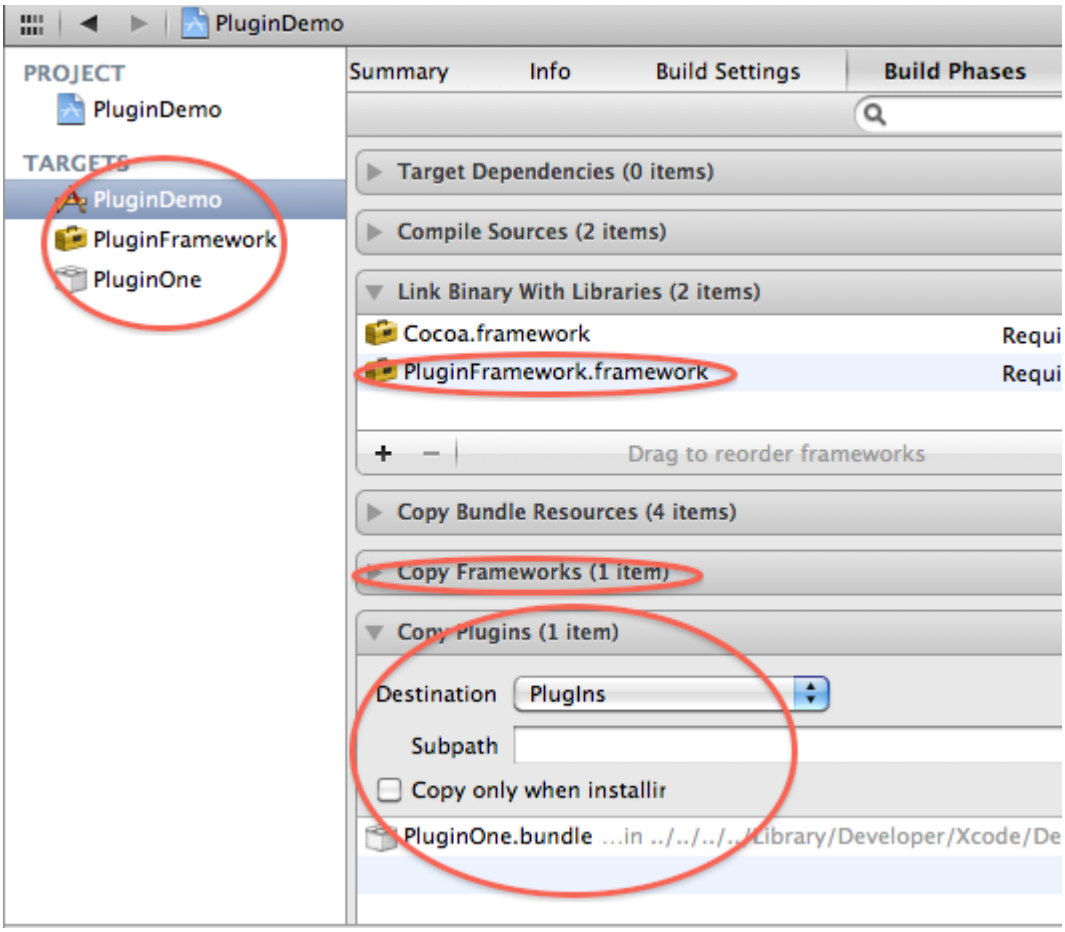
下面我们来对 plugin 进行设置，我们可以设置其 Principal class，Wrapper Extension（扩展名）。



使用 plugin

1，宿主程序设置

前面说了，在这个例子中，我们打算将插件随宿主程序一起发布，所以其存放位置就在宿主应用程序包中。因此我们需要在宿主程序种添加一个 Add Copy Files 的 build phase，如下所示：



2，载入 plugin

在正式的应用中，我们应该在前面提到的三个目录下去查找所有 plugin，因为这三个目录都是 Cocoa 所推荐的 plugin 目录。在这个例子中，演示的是随宿主应用程序一起发布的程序，所以我只扫描了应用程序包中的目录。

```
- (NSArray *)loadPlugins
{
          NSBundle *main = [NSBundle mainBundle];
          NSArray *allPlugins = [main pathsForResourcesOfType:@"bundle"
inDirectory:@"../PlugIns"];

          NSMutableArray *availablePlugins = [[[NSMutableArray alloc] init] autorelease];

          id plugin = nil;
          NSBundle *pluginBundle = nil;
```

```
          for (NSString *path in allPlugins) {
                    pluginBundle = [NSBundle bundleWithPath:path];
                    [pluginBundle load];

                    Class principalClass = [pluginBundle principalClass];
                    if (![principalClass isSubclassOfClass:[AbstractPlugin class]]) {
                              continue;
                    }

                    plugin = [[principalClass alloc] init];

      if ([plugin respondsToSelector:@selector(run:)])
      {
          [availablePlugins addObject:plugin];
          NSLog(@" >> loading plugin %@ from %@", [plugin name], path);
      }

                    [plugin release];
      plugin = nil;
                    pluginBundle = nil;
          }

          return availablePlugins;
}
```

该函数在 init 中被调用：

```
- (id)init
{
    self = [super init];
    if (self) {
        plugins = [[self loadPlugins] retain];
    }

    return self;
}
```

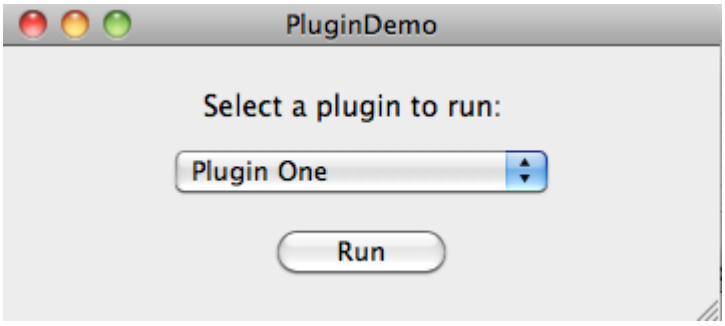
下面提供一个函数扫描前面提到的三个目录，你可以用这个函数提到上面代码中对 loadPlugins 的调用：

```
- (NSArray *)loadAllPlugins
```

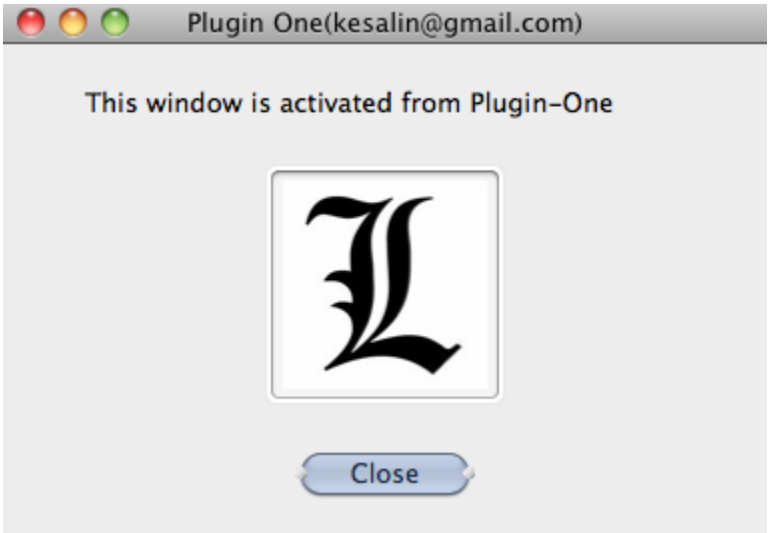
```
{
    NSString *appName      = @"PluginOne/Plugins";
    NSString *appSupport    = @"Library/Application Support";
    appSupport            = [appSupport stringByAppendingPathComponent:appName];

    NSString *appPath      = [[NSBundle mainBundle] builtInPlugInsPath];
    NSString *userPath     = [NSHomeDirectory() stringByAppendingPathComponent:appSupport];
    NSString *sysPath      = [@"/" stringByAppendingPathComponent:appSupport];

    NSArray* paths = [NSArray arrayWithObjects:appPath, userPath, sysPath, nil];

    NSMutableArray * availablePlugins = [[[NSMutableArray alloc] init] autorelease];
    for (NSString * path in paths)
    {
        NSLog(@" >> Search in directory: %@", path);

        NSArray *contents = [[NSFileManager defaultManager] contentsOfDirectoryAtPath:path
error:NULL];
        for (NSString *fileName in contents)
        {
            if ( [[fileName pathExtension] isEqualToString:@"plugin"] || [[fileName pathExtension]
isEqualToString:@"bundle"])
            {
                NSString *fullPath = [path stringByAppendingPathComponent:fileName];
                NSBundle *pluginBundle = [NSBundle bundleWithPath:fullPath];
                if (pluginBundle && [pluginBundle load])
                {
                    Class principalClass = [pluginBundle principalClass];
                    if (![principalClass isSubclassOfClass:[AbstractPlugin class]]) {
                        continue;
                    }

                    id plugin = [[principalClass alloc] init];

                    if ([plugin respondsToSelector:@selector(run:)])
                    {
                        [availablePlugins addObject:plugin];
                        NSLog(@" >> loading plugin %@ from %@", [plugin name], path);
                    }
```

```
                [plugin release];
                plugin = nil;
            }
        }
    }

    return availablePlugins;
}
```

显示 plugin 列表的 popupbutton 的内容被绑定到该 plugins 数组，所以程序启动之后，就能显示 plugin 的列表。运行结果如下：



点击运行之后，就能显示出插件主界面：



Reference：

Code Loading Programming Topics

provides information about writing plug-ins using the Objective-C language.

Bundle Programming Guide

provides an overview to bundles, including their purpose, types, structure, and the API for accessing bundle resources.

[深入浅出 Cocoa] 之 Core Data（1）- 框架详解

罗朝辉(http://blog.csdn.net/kesalin)

Core data 是 Cocoa 中处理数据，绑定数据的关键特性，其重要性不言而喻，但也比较复杂。Core Data 相关的类比较多，初学者往往不太容易弄懂。计划用三个教程来讲解这一部分：

框架详解：讲解 Core data 框架，运作过程，设计的类；

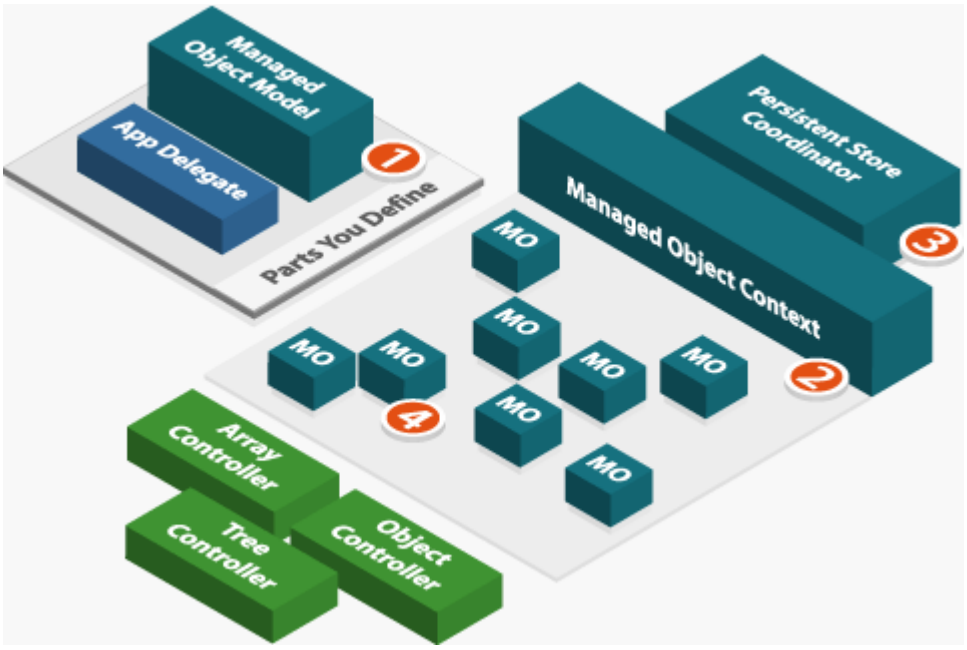
Core data 应用程序示例：通过生成一个使用 Core data 的应用程序来讲解如何 在 XCode 4 中使用 Core data。

手动创建 Core data 示例：不利用框架自动生成代码，完全自己编写所有的 Core data 相关代码的命令行应用程序来深入讲解 Core data 的使用。

本文为第一部份：框架详解

一，概观

下面先给出一张类关系图，让我们对它有个总体的认识。



在上图中，我们可以看到有五个相关模块：

1， Managed Object Model

Managed Object Model 是描述应用程序的数据模型，这个模型包含实体（Entity），特性（Property），读取请求（Fetch Request）等。（下文都使用英文术语。）

2，Managed Object Context

Managed Object Context 参与对数据对象进行各种操作的全过程，并监测数据对象的变化，以提供对 undo/redo 的支持及更

新绑定到数据的 UI。

3，Persistent Store Coordinator

Persistent Store Coordinator 相当于数据文件管理器，处理底层的对数据文件的读取与写入。一般我们无需与它打交道。

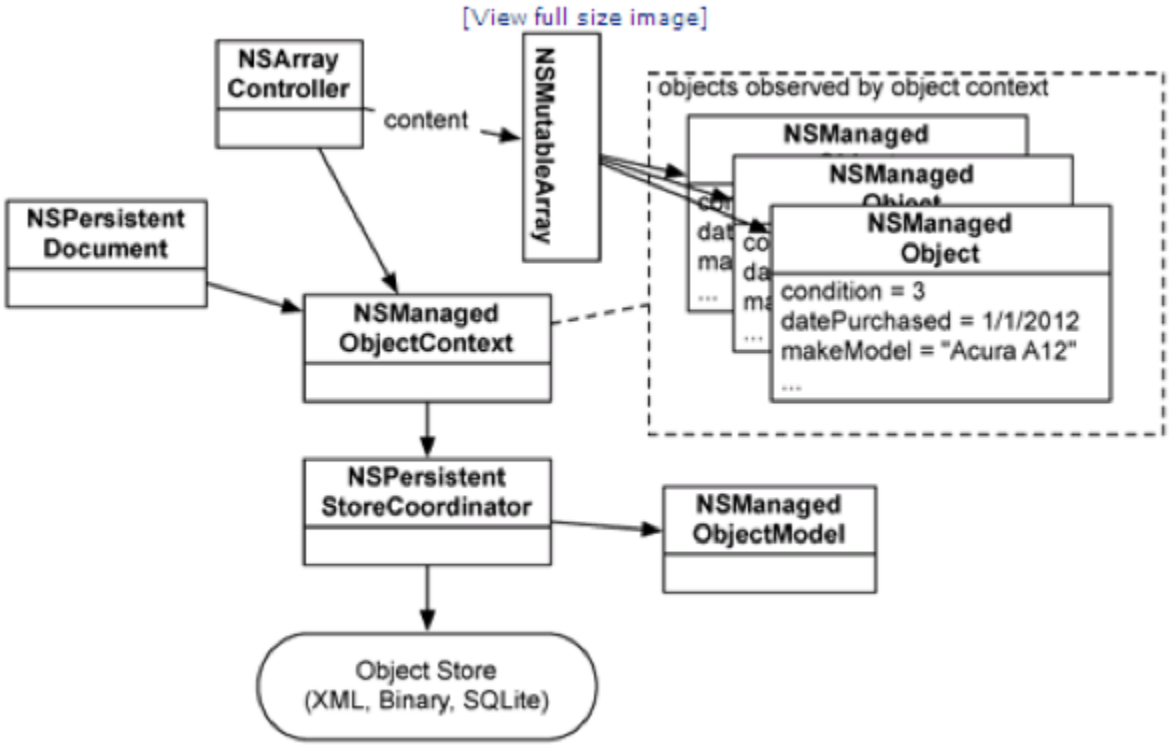
4，Managed Object

Managed Object 数据对象，与 Managed Object Context 相关联。

5，Controller

图中绿色的 Array Controller, Object Controller, Tree Controller 这些控制器，一般都是通过 control+drag 将 Managed Object Context 绑定到它们，这样我们就可以在 nib 中可视化地操作数据。

这写模块是怎样运作的呢？



1，应用程序先创建或读取模型文件（后缀为 xcdatamodeld）生成 NSManagedObjectModel 对象。Document 应用程序是一般是通过 NSDocument 或其子类 NSPersistentDocument）从模型文件（后缀为 xcdatamodeld）读取。

2，然后生成 NSManagedObjectContext 和 NSPersistentStoreCoordinator 对象，前者对用户透明地调用后者对数据文件进行读写。

3，NSPersistentStoreCoordinator 负责从数据文件(xml, sqlite,二进制文件等)中读取数据生成 Managed Object，或保存 Managed Object 写入数据文件。

4，NSManagedObjectContext 参与对数据进行各种操作的整个过程，它持有 Managed Object。我们通过它来监测 Managed Object。监测数据对象有两个作用：支持 undo/redo 以及数据绑定。这个类是最常被用到的。

5，Array Controller, Object Controller, Tree Controller 这些控制器一般与 NSManagedObjectContext 关联，因此我们可以通过它们在 nib 中可视化地操作数据对象。

二， Model class

模型有点像数据库的表结构，里面包含 Entry， 实体又包含三种 Property：Attribute（属性），RelationShip（关系）， Fetched Property（读取属性）。Model class 的名字多以 "Description" 结尾。我们可以看出：模型就是描述数据类型以及其关系的。

主要的 Model class 有：

Model Classes		
Managed Object Model	NSManagedObjectModel	数据模型
Entity	NSEntityDescription	抽象数据类型，相当于数据库中的表
Property	NSPropertyDescription	Entity 特性，相当于数据库表中的一列
> Attribute	NSAttributeDescription	基本数值型属性（如 Int16, BOOL, Date 等类型的属性）
> Relationship	NSRelationshipDescription	属性之间的关系
> Fetched Property	NSFetchedPropertyDescription	查询属性（相当于数据库中的查询语句）

1）Entity - NSEntityDescription

Entity 相当于数据库中的一个表，它描述一种抽象数据类型，其对应的类为 NSManagedObject 或其子类。

NSEntityDescription 常用方法:

+insertNewObjectForEntityForName:inManagedObjectContext: 工厂方法，根据给定的 Entity 描述，生成相应的 NSManagedObject 对象，并插入 ManagedObjectContext 中。

-managedObjectClassName 返回映射到 Entity 的 NSManagedObject 类名

-attributesByName 以名字为 key， 返回 Entity 中对应的 Attributes

-relationshipsByName 以名字为 key， 返回 Entity 中对应的 Relationships

2）Property - NSPropertyDescription

Property 为 Entity 的特性，它相当于数据库表中的一列，或者 XML 文件中的 value-key 对中的 key。它可以描述实体数据(Attribute)，Entity 之间的关系(RelationShip)，或查询属性(Fetched Property)。

> Attribute - NSAttributeDescription

Attribute 存储基本数据，如 NSString, NSNumber or NSDate 等。它可以有默认值，也可以使用正则表达式或其他条件对其值进行限定。一个属性可以是 optional 的。

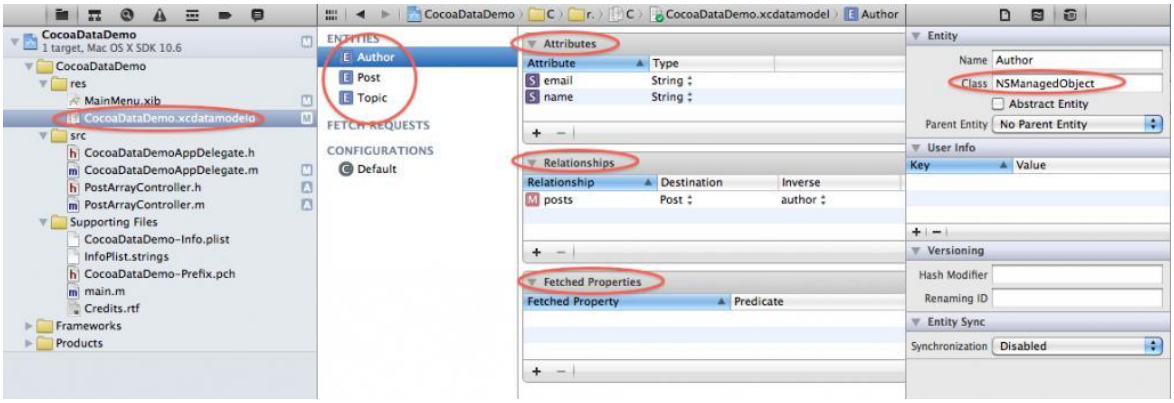
> Relationship - NSRelationshipDescription

Relationship 描述 Entity，Property 之间的关系，可以是一对一，也可以是一对多的关系。

> Fetched Property - NSFetchedPropertyDescription

Fetched Property 根据查询谓词返回指定 Entity 的符合条件的数据对象。

上面说的比较抽象，举个例子来说，见图：



我们有一个 CocoaDataDemo.xcdatamodeld 模型文件，应用程序根据它生成一个 NSManagedObjectModel 对象，这个模型有三个 Entity，每个 Entity 又可包含 Attribute Relationship, Feteched Property 三种类型的 Property。在本例中，Author Entity 包含两个 Attribute : name 和 email，它们对于的运行时类均为 NSManagedObject；还包含一个与 Post 的 Relationship；没有设置 Feteched Property。

我们通常使用 KVC 机制来访问 Property。下面来看代码：

```
NSManagedObjectContext * context = [[NSApp delegate] managedObjectContext];

NSManagedObject       * author  = nil;


author = [NSEntityDescription insertNewObjectForEntityForName: @"Author" inManagedObjectContext:
context];

[author setValue: @"nemo@pixar.com" forKey: @"email"];


NSLog (@"The Author's email is: %@", [author valueForKey:@"email"]);
```

在上面代码中，我们先取得 NSManagedObjectContext， 然后调用 NSEntityDescription 的方法，以 Author 为实体模型，生成对应的 NSManagedObject 对象，插入 NSManagedObjectContext 中，然后给这个对象设置特性 email 的值。

三，运行时类与对象

> Managed Object - NSManagedObject

Managed Object 表示数据文件中的一条记录，每一个 Managed Object 在内存中对应 Entity 的一个数据表示。Managed Object 的成员为 Entity 的 Property 所描述。

比如在上面的代码，author 这个 NSManagedObject，对应名为 Author 的 Entity。

每一个 Managed Object 都有一个全局 ID（类型为：NSManagedObjectID）。Managed Object 会附加到一个 Managed Object Context，我们可以通过这个全局 ID 在 Managed Object Context 查询对应的 Managed Object。

NSManagedObject 常用方法	
-entity	获取其 Entity
-objectID	获取其 Managed Object ID

-valueForKey:	获取指定 Property 的值
-setValue: forKey:	设定指定 Property 的值

> Managed Object Context - NSManagedObjectContext

Managed Object Context 的作用相当重要，对数据对象进行的操作都与它有关。当创建一个数据对象并插入 Managed Object Context 中，Managed Object Context 就开始跟踪这个数据对象的一切变动，并在合适的时候提供对 undo/redo 的支持，或调用 Persistent Store Coordinato 将变化保存到数据文件中去。

通常我们将 controller 类（如：NSArrayController，NSTreeController）或其子类与 Managed Object Context 绑定，这样就方便我们动态地生成，获取数据对象等。

NSManagedObjectContext 常用方法	
-save:	将数据对象保存到数据文件
-objectWithID:	查询指定 Managed Object ID 的数据对象
-deleteObject:	将一个数据对象标记为删除，但是要等到 Context 提交更改时才真正删除数据对象
-undo	回滚最后一步操作，这是都 undo/redo 的支持
-lock	加锁，常用于多线程以及创建事务。同类接口还有：-unlock and -tryLock
-rollback	还原数据文件内容
-reset	清除缓存的 Managed Objects。只应当在添加或删除 Persistent Stores 时使用
-undoManager	返回当前 Context 所使用的 NSUndoManager
-assignObject: toPersistantStore:	由于 Context 可以管理从不同数据文件而来的数据对象， 这个接口的作用就是指定数据对象的存储数据文件（通过指定 PersistantStore 实现）
-executeFetchRequest: error:	执行 Fetch Request 并返回所有匹配的数据对象

> Persistent Store Coordinator - NSPersistentStoreCoordinator

使用 Core Data document 类型的应用程序，通常会从磁盘上的数据文中中读取或存储数据，这写底层的读写就由 Persistent Store Coordinator 来处理。一般我们无需与它直接打交道来读写文件，Managed Object Context 在背后已经为我们调用 Persistent Store Coordinator 做了这部分工作。

NSPersistentStoreCoordinator 常用方法	
-addPersistentStoreForURL:configuration:URL:options:error:	装载数据存储，对应的卸载数据存储的接口为 -removePersistentStore:error:
-migratePersistentStore:toURL:options:withType:error:	迁移数据存储，效果与 "save as"相似，但是操作成功后， 迁移前的数据存储不可再使用
-managedObjectIDForURIRepresentation:	返回给定 URL 所指示的数据存储的 object id，如果找不到匹配的数据存储则
-persistentStoreForURL:	返回指定路径的 Persistent Store
-URLForPersistentStore:	返回指定 Persistent Store 的存储路径

> Persistent Document - NSPersistentDocument

NSPersistentDocument 是 NSDocument 的子类。 multi-document Core Data 应用程序使用它来简化对 Core Data 的操作。

通常使用 NSPersistentDocument 的默认实现就足够了，它从 Info.plist 中读取 Document types 信息来决定数据的存储格式（xml,sqlite, binary）。

NSPersistentDocument 常用方法	
-managedObjectContext	返回文档的 Managed Object Context，在多文档应用程序中，每个文档都有自己的 Context。
-managedObjectModel	返回文档的 Managed Object Model

四，Fetch Requests

Fetch Requests 相当于一个查询语句，你必须指定要查询的 Entity。我们通过 Fetch Requests 向 Managed Object Context 查询符合条件的数据对象，以 NSArray 形式返回查询结果，如果我们没有设置任何查询条件，则返回该 Entity 的所有数据对象。我们可以使用谓词来设置查询条件，通常会将常用的 Fetch Requests 保存到 dictionary 以重复利用。

示例：

```
NSManagedObjectContext * context  = [[NSApp delegate] managedObjectContext];
NSManagedObjectModel   * model    = [[NSApp delegate] managedObjectModel];
NSDictionary          * entities = [model entitiesByName];
NSEntityDescription    * entity   = [entities valueForKey:@"Post"];

NSPredicate * predicate;
predicate = [NSPredicate predicateWithFormat:@"creationDate > %@", date];

NSSortDescriptor * sort = [[NSSortDescriptor alloc] initWithKey:@"title"];
NSArray * sortDescriptors = [NSArray arrayWithObject: sort];

NSFetchRequest * fetch = [[NSFetchRequest alloc] init];
[fetch setEntity: entity];
[fetch setPredicate: predicate];
[fetch setSortDescriptors: sortDescriptors];

NSArray * results = [context executeFetchRequest:fetch error:nil];
[sort release];
[fetch release];
```

在上面代码中，我们查询在指定日期之后创建的 post，并将查询结果按照 title 排序返回。

NSFetchRequest 常用方法	
-setEntity:	设置你要查询的数据对象的类型（Entity）
-setPredicate:	设置查询条件
-setFetchLimit:	设置最大查询对象数目
-setSortDescriptors:	设置查询结果的排序方法

-setAffectedStores:	设置可以在哪些数据存储中查询

参考资料：

Core Data Reference API listing for the Core Data classes

http://developer.apple.com/documentation/Cocoa/Reference/CoreData_ObjC/index.html

NSPredicate Reference API listing for NSPredicate

http://developer.apple.com/documentation/Cocoa/Reference/Foundation/ObjC_classic/Classes/NSPredicate.html

[深入浅出 Cocoa] 之 Core Data（2）- 代码示例

罗朝辉(http://blog.csdn.net/kesalin)

前面详细讲解了 Core Data 的框架以及设计的类，下面我们来讲解一个完全手动编写代码-使用这些类的示例，这个例子来自-苹果官方示例。在这个例子里面，我们打算做这样一件事情：记录程序运行记录（时间与 process id），并保存到 xml 文件中。我们使用 Core Data 来做这个事情。

示例代码下载：点击这里

一，建立一个新的 Mac command-line tool application 工程，命名为 CoreDataTutorial。为支持垃圾主动回收机制，点击项目名称，在右边的 Build Setting 中查找 garbage 关键字，将找到的 Objective-C Garbage Collection 设置为 Required [-fobj-gc-only]。并将 main.m 中 的 main() 方法修改为如下：

```
int main (int argc, const char * argv[])
{
    NSLog(@" === Core Data Tutorial ===");

    // Enable GC
    //
    objc_startCollectorThread();

    return 0;
}
```

二，创建并设置模型类

在 main() 之前添加如下方法：

```
NSManagedObjectModel *managedObjectModel()
{
    static NSManagedObjectModel *moModel = nil;

    if (moModel != nil) {
        return moModel;
    }

    moModel = [[NSManagedObjectModel alloc] init];
```

```
    // Create the entity
    //
    NSEntityDescription *runEntity = [[NSEntityDescription alloc] init];
    [runEntity setName:@"Run"];
    [runEntity setManagedObjectClassName:@"Run"];

    [moModel setEntities:[NSArray arrayWithObject:runEntity]];

    // Add the Attributes
    //
    NSAttributeDescription *dateAttribute = [[NSAttributeDescription alloc] init];
    [dateAttribute setName:@"date"];
    [dateAttribute setAttributeType:NSDateAttributeType];
    [dateAttribute setOptional:NO];

    NSAttributeDescription *idAttribute = [[NSAttributeDescription alloc] init];
    [idAttribute setName:@"processID"];
    [idAttribute setAttributeType:NSInteger32AttributeType];
    [idAttribute setOptional:NO];
    [idAttribute setDefaultValue:[NSNumber numberWithInteger:-1]];

    // Create the validation predicate for the process ID.
    // The following code is equivalent to validationPredicate = [NSPredicate
predicateWithFormat:@"SELF > 0"]
    //
    NSExpression *lhs = [NSExpression expressionForEvaluatedObject];
    NSExpression *rhs = [NSExpression expressionForConstantValue:[NSNumber
numberWithInteger:0]];

    NSPredicate *validationPredicate = [NSComparisonPredicate
                                    predicateWithLeftExpression:lhs
                                    rightExpression:rhs
                                    modifier:NSDirectPredicateModifier
                                    type:NSGreaterThanPredicateOperatorType
                                    options:0];

    NSString *validationWarning = @"Process ID < 1";
    [idAttribute setValidationPredicates:[NSArray arrayWithObject:validationPredicate]
                withValidationWarnings:[NSArray arrayWithObject:validationWarning]];
```

```
    // set the properties for the entity.
    //
    NSArray *properties = [NSArray arrayWithObjects: dateAttribute, idAttribute, nil];
    [runEntity setProperties:properties];

    // Add a Localization Dictionary
    //
    NSMutableDictionary *localizationDictionary = [NSMutableDictionary dictionary];
    [localizationDictionary setObject:@"Date" forKey:@"Property/date/Entity/Run"];
    [localizationDictionary setObject:@"Process ID" forKey:@"Property/processID/Entity/Run"];
    [localizationDictionary setObject:@"Process ID must not be less than 1"
forKey:@"ErrorString/Process ID < 1"];

    [moModel setLocalizationDictionary:localizationDictionary];

    return moModel;
}
```

在上面的代码中：

1）我们创建了一个全局模型 moModel；

2）并在其中创建一个名为 Run 的 Entity，这个 Entity 对应的 ManagedObject 类名为 Run（很快我们将创建这样一个类）；

3）给 Run Entity 添加了两个必须的 Property：date 和 processID，分别表示运行时间以及进程 ID；并设置默认的进程 ID 为 -1；

4）给 processID 特性设置检验条件：必须大于 0；

5）给模型设置本地化描述词典；

本地化描述提供对 Entity，Property，Error 信息等的便于理解的描述，其可用的键值对如下表：

Key	Value
"Entity/NonLocalizedEntityName"	"LocalizedEntityName"
"Property/NonLocalizedPropertyName/Entity/EntityName"	"LocalizedPropertyName"
"Property/NonLocalizedPropertyName"	"LocalizedPropertyName"
"ErrorString/NonLocalizedErrorString"	"LocalizedErrorString"

三，创建并设置运行时类和对象

由于要用到存储功能，所以我们必须定义持久化数据的存储路径。我们在 main() 之前添加如下方法设置存储路径：

```
NSURL *applicationLogDirectory()
{
    NSString *LOG_DIRECTORY = @"CoreDataTutorial";
    static NSURL *ald = nil;

    if (ald == nil)
    {
        NSFileManager *fileManager = [[NSFileManager alloc] init];
        NSError *error = nil;
        NSURL *libraryURL = [fileManager URLForDirectory:NSLibraryDirectory
inDomain:NSUserDomainMask
                                 appropriateForURL:nil create:YES error:&error];
        if (libraryURL == nil) {
            NSLog(@"Could not access Library directory\n%@", [error localizedDescription]);
        }
        else
        {
            ald = [libraryURL URLByAppendingPathComponent:@"Logs"];
            ald = [ald URLByAppendingPathComponent:LOG_DIRECTORY];

            NSLog(@" >> log path %@", [ald path]);

            NSDictionary *properties = [ald resourceValuesForKeys:[NSArray
arrayWithObject:NSURLIsDirectoryKey] error:&error];
            if (properties == nil)
            {
                if (![fileManager createDirectoryAtPath:[ald path]
withIntermediateDirectories:YES attributes:nil error:&error])
                {
                    NSLog(@"Could not create directory %@\n%@",
                          [ald path], [error localizedDescription]);
                    ald = nil;
                }
            }
        }
    }

    return ald;
}
```

在上面的代码中，我们将持久化数据文件保存到路径：**/Users/kesalin/Library/Logs/CoreDataTutorial** 下。

下面，我们来创建运行时对象：ManagedObjectContext 和 PersistentStoreCoordinator。

```
NSManagedObjectContext *managedObjectContext()
{
    static NSManagedObjectContext *moContext = nil;
    if (moContext != nil) {
        return moContext;
    }

    moContext = [[NSManagedObjectContext alloc] init];

    // Create a persistent store coordinator, then set the coordinator for the context.
    //
    NSManagedObjectModel *moModel = managedObjectModel();
    NSPersistentStoreCoordinator *coordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:moModel];
    [moContext setPersistentStoreCoordinator: coordinator];

    // Create a new persistent store of the appropriate type.
    //
    NSString *STORE_TYPE = NSXMLStoreType;
    NSString *STORE_FILENAME = @"CoreDataTutorial.xml";

    NSError *error = nil;
    NSURL *url = [applicationLogDirectory() URLByAppendingPathComponent:STORE_FILENAME];

    NSPersistentStore *newStore = [coordinator addPersistentStoreWithType:STORE_TYPE
                                                            configuration:nil
                                                                      URL:url
                                                                  options:nil
                                                                    error:&error];
    if (newStore == nil) {
        NSLog(@"Store Configuration Failure\n%@", ([error localizedDescription] != nil) ? [error
localizedDescription] : @"Unknown Error");
    }

    return moContext;
}
```

在上面的代码中：

1）我们创建了一个全局 ManagedObjectContext 对象 moContext；

2）并在设置其 persistent store coordinator，存储类型为 xml，保存文件名为：CoreDataTutorial.xml，并将其放到前面定义的存储路径下。

好，至此万事具备，只欠 ManagedObject 了！下面我们就来定义这个数据对象类。向工程添加 Core Data->NSManagedObject subclass 的类，名为 Run (模型中 Entity 定义的类名) 。

Run.h

```
#import <CoreData/NSManagedObject.h>


@interface Run : NSManagedObject
{
    NSInteger processID;
}


@property (retain) NSDate *date;
@property (retain) NSDate *primitiveDate;
@property NSInteger processID;


@end
```

Run.m

```
//
//  Run.m
//  CoreDataTutorial
//
//  Created by kesalin on 8/29/11.
//  Copyright 2011 kesalin@gmail.com. All rights reserved.
//


#import "Run.h"


@implementation Run


@dynamic date;
@dynamic primitiveDate;


- (void) awakeFromInsert
```

```
{
    [super awakeFromInsert];

    self.primitiveDate = [NSDate date];
}

#pragma mark -
#pragma mark Getter and setter

- (NSInteger)processID
{
    [self willAccessValueForKey:@"processID"];
    NSInteger pid = processID;
    [self didAccessValueForKey:@"processID"];
    return pid;
}

- (void)setProcessID:(NSInteger)newProcessID
{
    [self willChangeValueForKey:@"processID"];
    processID = newProcessID;
    [self didChangeValueForKey:@"processID"];
}

// Implement a setNilValueForKey: method. If the key is “processID” then set processID to 0.
//
- (void)setNilValueForKey:(NSString *)key {

    if ([key isEqualToString:@"processID"]) {
        self.processID = 0;
    }
    else {
        [super setNilValueForKey:key];
    }
}

@end
```

注意：

1）这个类中的 date 和 primitiveDate 的访问属性为 @dynamic，这表明在运行期会动态生成对应的 setter 和 getter；

2）在这里我们演示了如何正确地手动实现 processID 的 setter 和 getter：为了让 ManagedObjecContext 能够检测 processID 的变化，以及自动支持 undo/redo，我们需要在访问和更改数据对象时告之系统，will/didAccessValueForKey 以及 will/didChangeValueForKey 就是起这个作用的。

3）当我们设置 nil 给数据对象 processID 时，我们可以在 setNilValueForKey 捕获这个情况，并将 processID 置 0；

4）当数据对象被插入到 ManagedObjectContext 时，我们在 awakeFromInsert 将时间设置为当前时间。

三，创建或读取数据对象，设置其值，保存

好，至此真正的万事具备，我们可以创建或从持久化文件中读取数据对象，设置其值，并将其保存到持久化文件中。本例中持久化文件为 xml 文件。修改 main() 中代码如下：

```
int main (int argc, const char * argv[])
{
    NSLog(@" === Core Data Tutorial ===");

    // Enable GC
    //
    objc_startCollectorThread();

    NSError *error = nil;

    NSManagedObjectModel *moModel = managedObjectModel();
    NSLog(@"The managed object model is defined as follows:\n%@", moModel);

    if (applicationLogDirectory() == nil) {
        exit(1);
    }

    NSManagedObjectContext *moContext = managedObjectContext();

    // Create an Instance of the Run Entity
    //
    NSEntityDescription *runEntity = [[moModel entitiesByName] objectForKey:@"Run"];
    Run *run = [[Run alloc] initWithEntity:runEntity insertIntoManagedObjectContext:moContext];
    NSProcessInfo *processInfo = [NSProcessInfo processInfo];
    run.processID = [processInfo processIdentifier];

    if (![moContext save: &error]) {
```

```
        NSLog(@"Error while saving\n%@", ([error localizedDescription] != nil) ? [error
localizedDescription] : @"Unknown Error");
        exit(1);
    }

    // Fetching Run Objects
    //
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:runEntity];

    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"date"
ascending:YES];
    [request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];

    error = nil;
    NSArray *array = [moContext executeFetchRequest:request error:&error];
    if ((error != nil) || (array == nil))
    {
        NSLog(@"Error while fetching\n%@", ([error localizedDescription] != nil) ? [error
localizedDescription] : @"Unknown Error");
        exit(1);
    }

    // Display the Results
    //
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateStyle:NSDateFormatterMediumStyle];
    [formatter setTimeStyle:NSDateFormatterMediumStyle];

    NSLog(@"%@ run history:", [processInfo processName]);

    for (run in array)
    {
        NSLog(@"On %@ as process ID %ld", [formatter stringForObjectValue:run.date],
run.processID);
    }

    return 0;
}
```

在上面的代码中：

1）我们先获得全局的 NSManagedObjectModel 和 NSManagedObjectContext 对象：moModel 和 moContext；

2）并创建一个 Run Entity，设置其 Property processID 为当前进程的 ID；

3）将该数据对象保存到持久化文件中：[moContext save: &error]。我们无需与 PersistentStoreCoordinator 打交道，只需要给 ManagedObjectContext 发送 save 消息即可，NSManagedObjectContext 会透明地在后面处理对持久化数据文件的读写；

4）然后我们创建一个 FetchRequest 来查询持久化数据文件中保存的数据记录，并将结果按照日期升序排列。查询操作也是由 ManagedObjectContext 来处理的：[moContextexecuteFetchRequest:request error:&error];

5）将查询结果打印输出；

大功告成！编译运行，我们可以得到如下显示：

```
2011-09-03 21:42:47.556 CoreDataTutorial[992:903] CoreDataTutorial run history:
2011-09-03 21:42:47.557 CoreDataTutorial[992:903] On 2011-9-3 下午 09:41:56 as process ID 940
2011-09-03 21:42:47.557 CoreDataTutorial[992:903] On 2011-9-3 下午 09:42:16 as process ID 955
2011-09-03 21:42:47.558 CoreDataTutorial[992:903] On 2011-9-3 下午 09:42:20 as process ID 965
2011-09-03 21:42:47.558 CoreDataTutorial[992:903] On 2011-9-3 下午 09:42:24 as process ID 978
2011-09-03 21:42:47.559 CoreDataTutorial[992:903] On 2011-9-3 下午 09:42:47 as process ID 992
```

通过这个例子，我们可以更好理解 Core Data 的运作机制。在 Core Data 中我们最常用的就是 ManagedObjectContext，它几乎参与对数据对象的所有操作，包括对 undo/redo 的支持；而 Entity 对应的运行时类为 ManagedObject，我们可以理解为抽象数据结构 Entity 在内存中由 ManagedObject 来体现，而 Perproty 数据类型在内存中则由 ManagedObject 类的成员属性来体现。一般我们不需要与 PersistentStoreCoordinator 打交道，对数据文件的读写操作都由 ManagedObjectContext 为我们代劳了。

[深入浅出 Cocoa] 之 Core Data（3）- 使用绑定

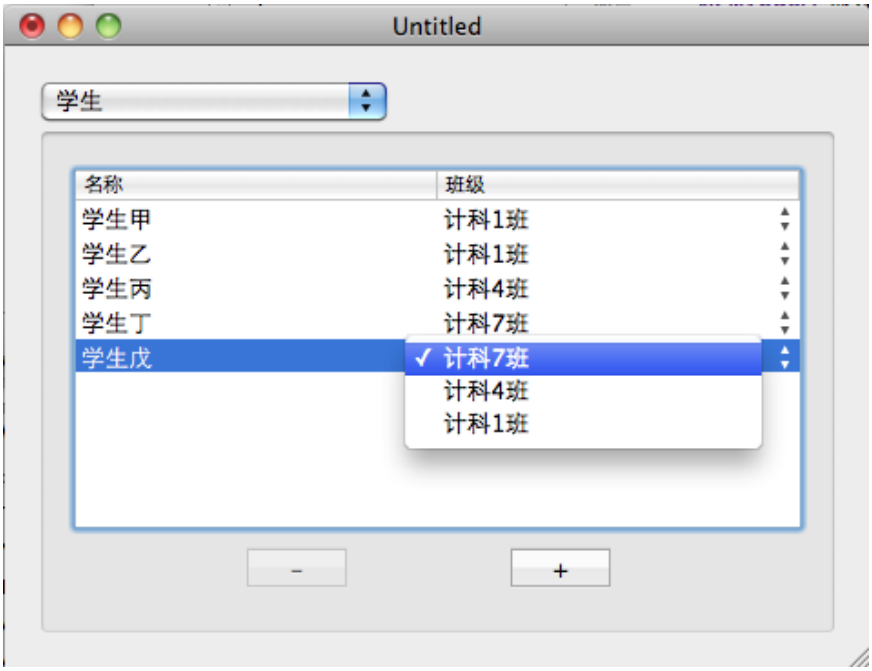
罗朝辉(http://blog.csdn.net/kesalin)

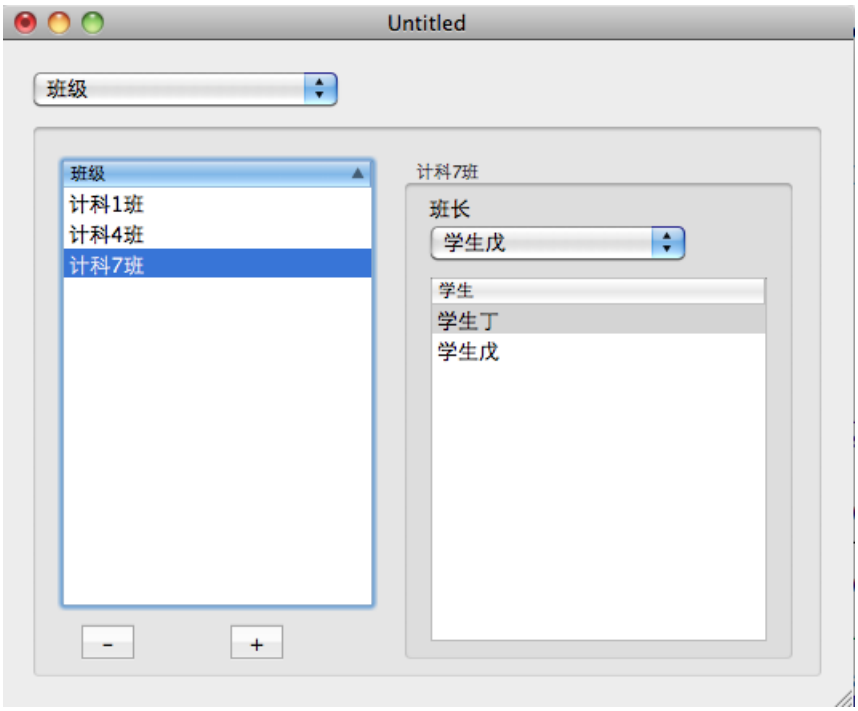
前面讲解了 Core Data 的框架，并完全手动编写代码演示了 Core Data 的运作过程。下面我们来演示如何结合 XCode 强大的可视化编辑以及 Cocoa 键值编码，绑定机制来使用 Core Data。有了上面提到的哪些利器，在这个示例中，我们无需编写 NSManagedObjectModel 代码，也无需编写 NSManagedObjectContext，工程模版在背后为我们做了这些事情。

今天要完成的这个示例，有两个 Entity：StudentEntity 与 ClassEntity，各自有一个名为 name 的 Attribute。其中 StudentEntity 通过一个名为 inClass 的 relationship 与 ClassEntity 关联，而 ClassEntity 也有一个名为 students 的 relationship 与 StudentEntity 关联，这是一个一对多的关系。此外 ClassEntity 还有一个名为 monitor 的 relationship 关联到 StudentEntity，表示该班的班长。

代码下载：点此下载

最终的效果图如下：

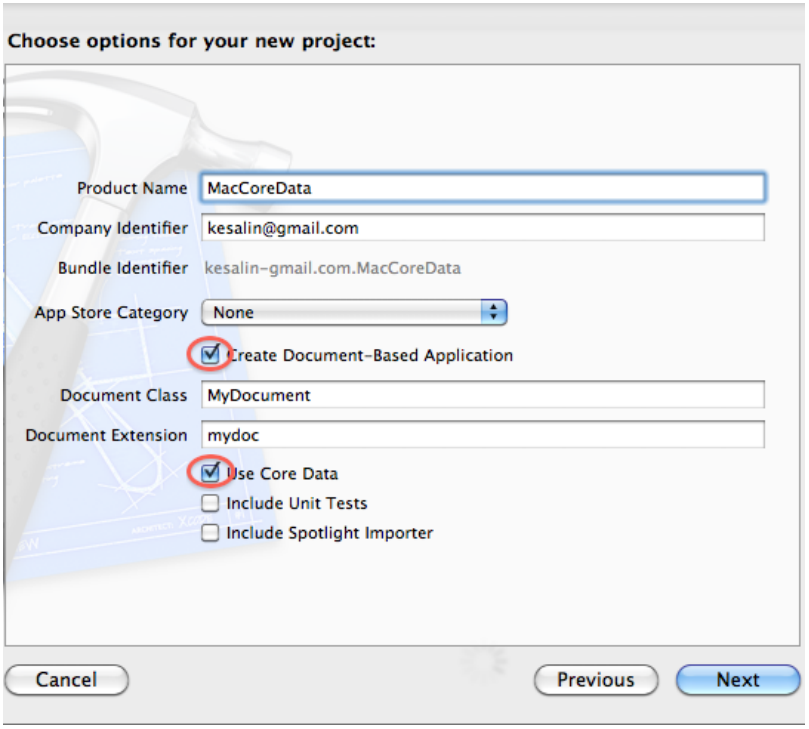




下面我们一步一步来完成这个示例：

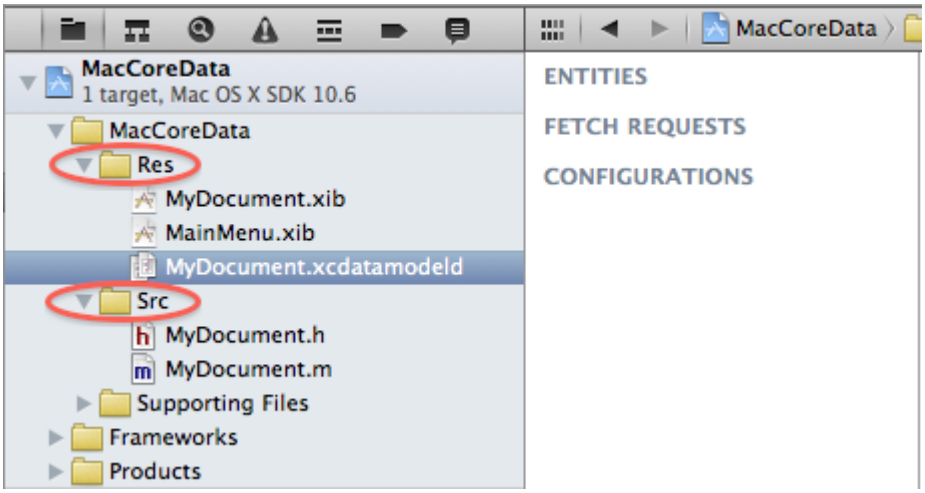
1，创建工程：

创建一个 Cocoa Application，工程名为：MacCoreData，并勾选 Create Document-Based Application 和 Use Core Data，在这里要用到 Core Data 和 Document 工程模版，以简化代码的编写。



2，分类文件：

在 MacCoreData 下新建 Src 和 Res 两个 Group，并将 MyDocument.h 和 MyDocument 拖到 Src 下，将其他 xib 和 xcdatamodeld 拖到 Res 中。将文件分类是个好习惯，尤其是对大项目来说。后面请自觉将文件分类~~

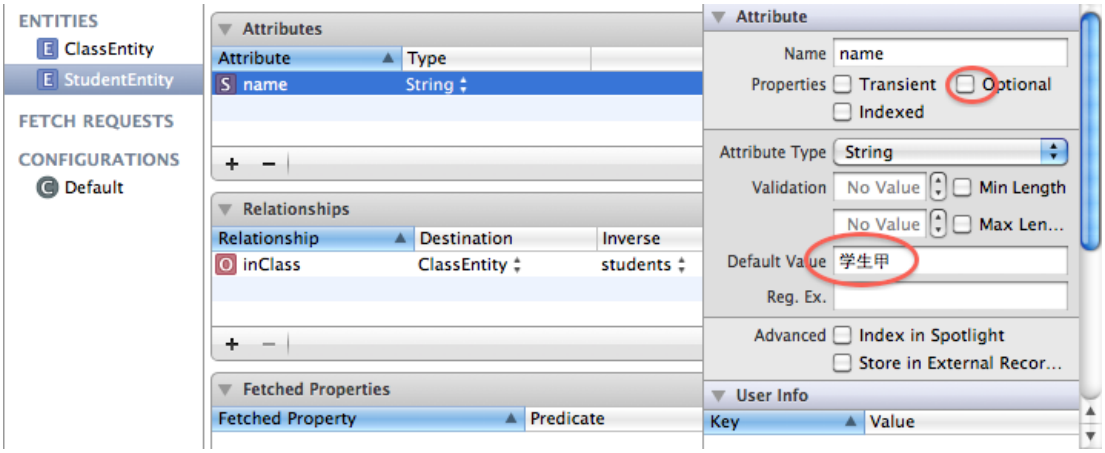


3，创建 Entity:

在工程中，我们可以看到名为 MyDocument.xcdatamodeld 的文件，其后缀表明这是一个 core data model 文件，框架就是读取该模型文件生成模型的。下面我们选中这个文件，向其中添加两个实体。点击下方的 Add Entity 增加两个新 Entity: ClassEntity 和 StudentEntity。

向 StudentEntity 中添加名为 name 的 string 类型的 Attribute，并设置其 Default Value 为学生甲，去除 Optional 前勾选状态；

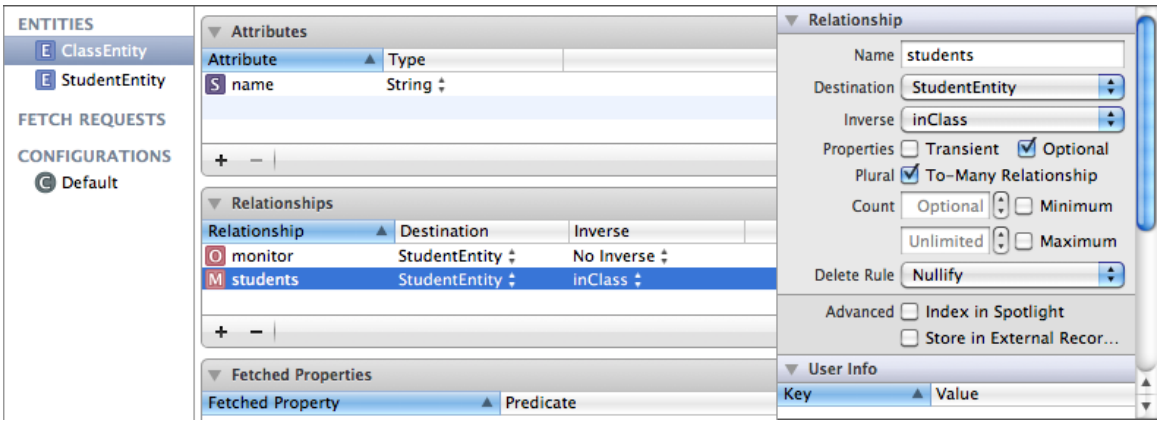
向 ClassEntity 中添加名为 name 的 string 类型的 Attribute，并设置其 Default Value 为 XX 班，去除 Optional 前勾选状态；选项 Optional 是表示该 Attribute 可选与否的，在这里 name 都是必须的。



向 StudentEntity 中添加名为 inClass 指向 ClassEntity 的 Relationship，其 Inverse 栏要等 ClassEntity 添加了反向关系才能选择，后面回提到；

向 ClassEntity 中添加名为 students 指向 StudentEntity 的 Relationship，其 Inverse 栏选择 inClass，表明这是一个双向关系，勾选 To-Many Relationship，因为一个班级可以有多名学生，这是一对多关系。设定之后，我们可以可以将 StudentEntity 的 inClass 关系的 Inverse 设置为 students 了。

再向 ClassEntity 中添加名为 monitor 指向 StudentEntity 的 Relationship，表示该班的班长。



4，生成 NSManagedObject 类：

选中 StudentEntity，然后点击菜单 File-> New -> New file…，添加 Core Data -> NSManagerObject subclass， XCode 就会自动为我们生成 StudentEntity.h 和 StudentEntity.m 文件，记得将这两个文件拖放到 Src Group 下。下面我们来看看这两个文件中有什么内容：

StudentEntity.h

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

#import <CoreData/CoreData.h>


@class ClassEntity;


@interface StudentEntity : NSManagedObject {

@private

}

@property (nonatomic, retain) NSString * name;

@property (nonatomic, retain) ClassEntity * inClass;


@end
```

StudentEntity.m

```
#import "StudentEntity.h"

#import "ClassEntity.h"


@implementation StudentEntity

@dynamic name;

@dynamic inClass;


@end
```

在前面手动代码的示例中，我们是自己编写 Run NSManagedObject 的代码，而现在，XCode 已经根据模型文件的描述，自动

为我们生成了，方便吧。有时候自动生成的代码不一定满足我们的需要，我们就得对代码进行修改，比如对 ClassEntity 来说，班长只能是其 students 中的一员，如果我们在 students 中移除了班长那个学生，那么该班级的班长就应该置空。

选中 ClassEntity，重复上面的步骤，自动生成 ClassEntity.h 和 ClassEntity.m，下面我们根据需求来修改 ClassEntity.m。
在 - (void)removeStudentsObject:(StudentEntity *)value 的开头添加如下代码：

```
if (value == [self monitor])
    [self setMonitor:nil];
```

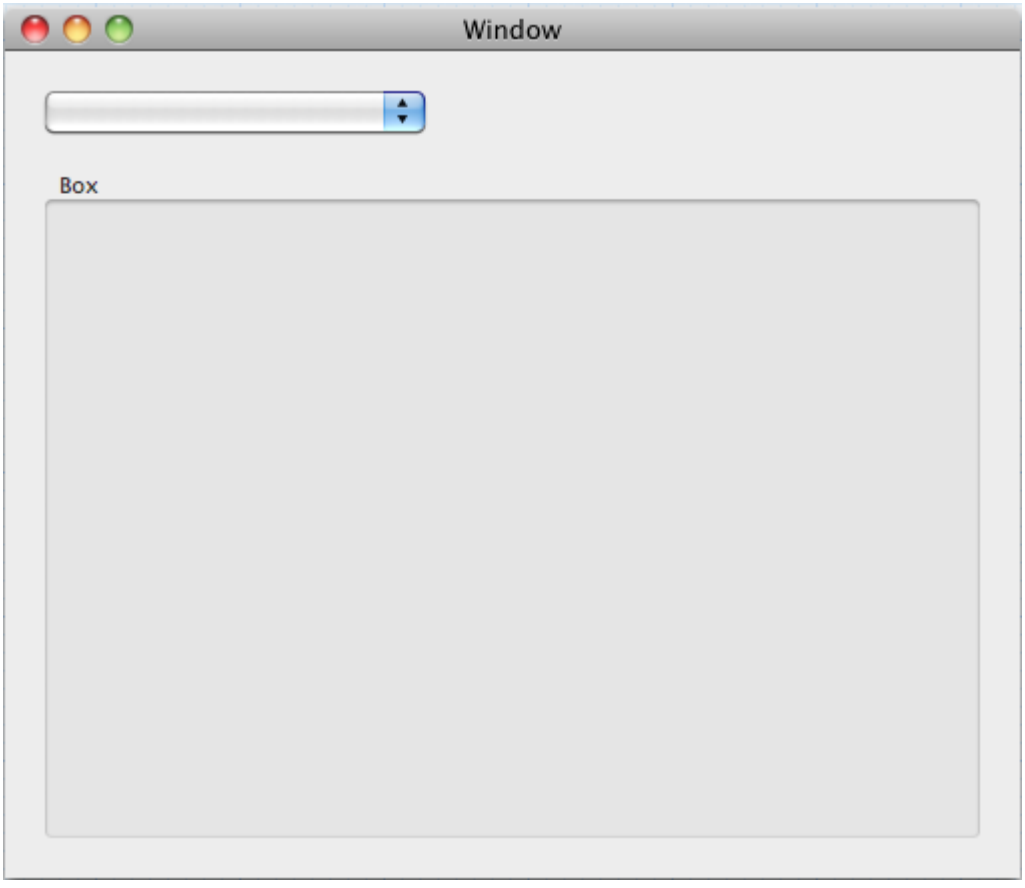
在 - (void)removeStudents:(NSSet *)value 的开头添加如下代码：

```
if ([value containsObject:[self monitor]])
    [self setMonitor:nil];
```

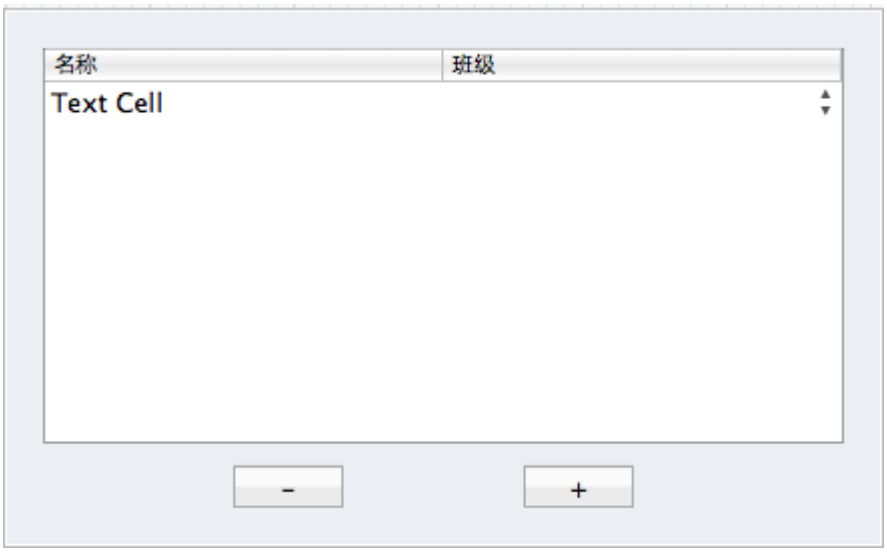
这样当我们在 students 中删除一个学生时，就会检测该学生是不是班长，如果是，就将该班的班长置空。

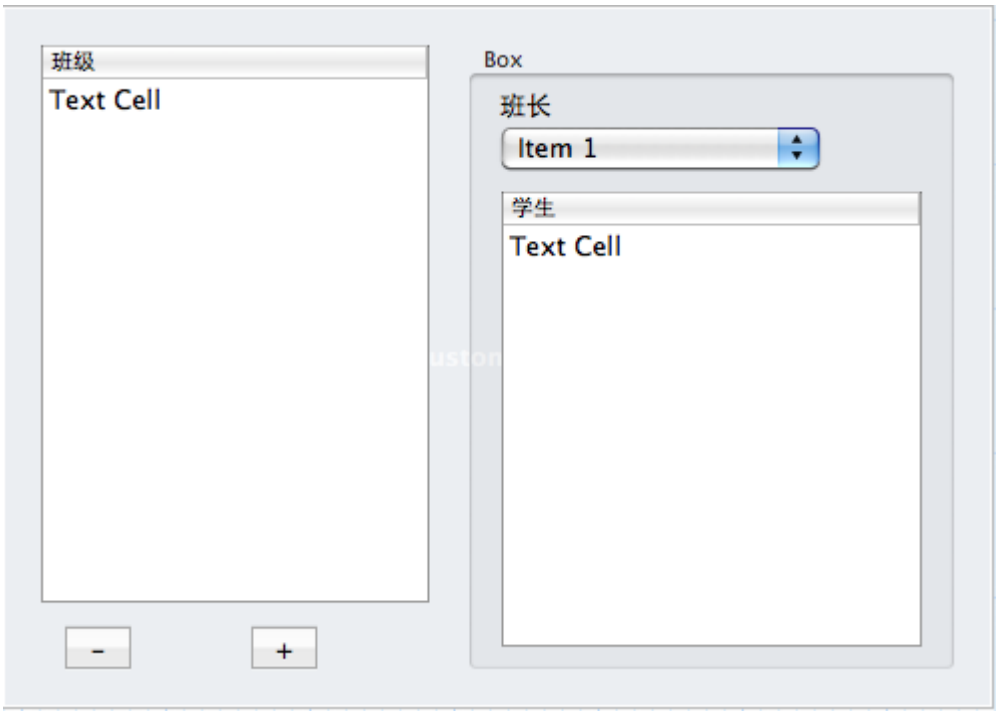
5，下面来生成 UI 界面：
在这里，我们是通过切换 view 的方法来显现学生与班级两个界面，因此我们需要主界面，班级以及学生共三个界面。

向 MyDocument.xib 中添加如下一个 popup button 和一个 NSBox。并删除 popup 控件中的 menu item，因为我们要通过代码来添加班级，学生项的。



然后在 Res 中添加两个新 Empty xib 文件：StudentView.xib 和 ClassView.xib，分别向这两个 xib 文件中拖入一个 Custom View，然后在这个 view 添加相关控件构成 UI。记得设置 ClassView 中两个 tableView 的列数为 1，拖入一个 PopupButtonCell 到 StudentView 中班级那一列。效果如下：





6，添加 ViewController：

下面我们创建 ViewController 来在程序中转载 xib 文件，显示和切换 view。为了便于切换 view，我们创建一个继承自 NSViewController 的名为：ManagedViewController 的类（记得不要创建该类对应的 xib 文件！创建一个 NSObject 子类，然后修改其父类为 NSViewController），然后让 StudentViewController 和 ClassViewController 从它继承。

ManagedViewController 类的代码如下：

ManagedViewController.h

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

```
@interface ManagedViewController : NSViewController {
@private
    NSManagedObjectContext * managedObjectContext;
    NSArrayController * contentArrayController;
}

@property (nonatomic, retain) NSManagedObjectContext * managedObjectContext;
@property (nonatomic, retain) IBOutlet NSArrayController *contentArrayController;

@end
```

ManagedViewController.m

```
#import "ManagedViewController.h"

@implementation ManagedViewController

@synthesize managedObjectContext;
@synthesize contentArrayController;

- (void)dealloc
{
    self.contentArrayController = nil;
    self.managedObjectContext = nil;

    [super dealloc];
}

// deal with "Delete" key event.
//
- (void) keyDown:(NSEvent *)theEvent
{
    if (contentArrayController) {
        if ([theEvent keyCode] == 51) {
            [contentArrayController remove:nil];
        }
        else {
            [super keyDown:theEvent];
        }
    }
```

```
    else {
        [super keyDown:theEvent];
    }
}

@end
```

在上面代码中，我们有一个 NSManagedObjectContext * managedObjectContext 指针，它指向 MyDocument 框架中的 NSManagedObjectContext 对象，后面我们会说到，至于 NSArrayController * contentArrayController，它是一个 IBOutlet，将与 xib 中创建的 NSArrayController 关联，后面也会说到。在这里引入 contentArrayController 是为了让 delete 能够删除记录。

ClassViewController 类的代码如下：

```
#import "ManagedViewController.h"

@interface ClassViewController : ManagedViewController {
@private
}

@end

#import "ClassViewController.h"

@implementation ClassViewController

- (id)init
{
    self = [super initWithNibName:@"ClassView" bundle:nil];
    if (self) {
        [self setTitle:@"班级"];
    }

    return self;
}

- (void)dealloc
{
    [super dealloc];
}

@end
```

StudentViewController 类的代码如下：

```
#import "ManagedViewController.h"


@interface StudentViewController : ManagedViewController {

@private

}


@end


#import "StudentViewController.h"


@implementation StudentViewController


- (id)init

{

    self = [super initWithNibName:@"StudentView" bundle:nil];

    if (self) {

        [self setTitle:@"学生"];

    }


    return self;

}


- (void)dealloc

{

    [super dealloc];

}


@end
```

在这两个子类中，我们在 init 方法中载入 xib 文件，然后设置其 title。

===前半部分完===

[深入浅出 Cocoa] 之 Core Data（4）- 使用绑定

罗朝辉(http://blog.csdn.net/kesalin)

前面讲解了 Core Data 的框架，并完全手动编写代码演示了 Core Data 的运作过程。下面我们来演示如何结合 XCode 强大的可视化编辑以及 Cocoa 键值编码，绑定机制来使用 Core Data。有了上面提到的哪些利器，在这个示例中，我们无需编写 NSManagedObjectModel 代码，也无需编写 NSManagedObjectContext，工程模版在背后为我们做了这些事情。

今天要完成的这个示例，有两个 Entity：StudentEntity 与 ClassEntity，各自有一个名为 name 的 Attribute。其中 StudentEntity 通过一个名为 inClass 的 relationship 与 ClassEntity 关联，而 ClassEntity 也有一个名为 students 的 relationship 与 StudentEntity 关联，这是一个一对多的关系。此外 ClassEntity 还有一个名为 monitor 的 relationship 关联到 StudentEntity，表示该班的班长。

代码下载：点此下载

==========

接前半部分

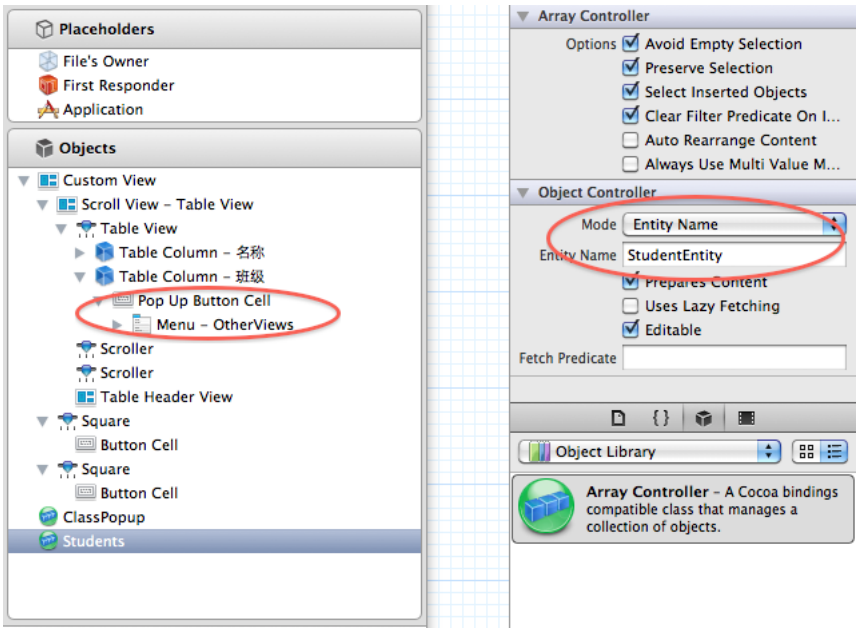
7，创建 NSArrayController，关联对象

现在回到 xib 中来，选中 StudentView.xib，设置 StudentView 的 File's Owner 的类为 StudentViewController；使用 Control-Drag 将 File's Owner 的 view 指向 custom view。

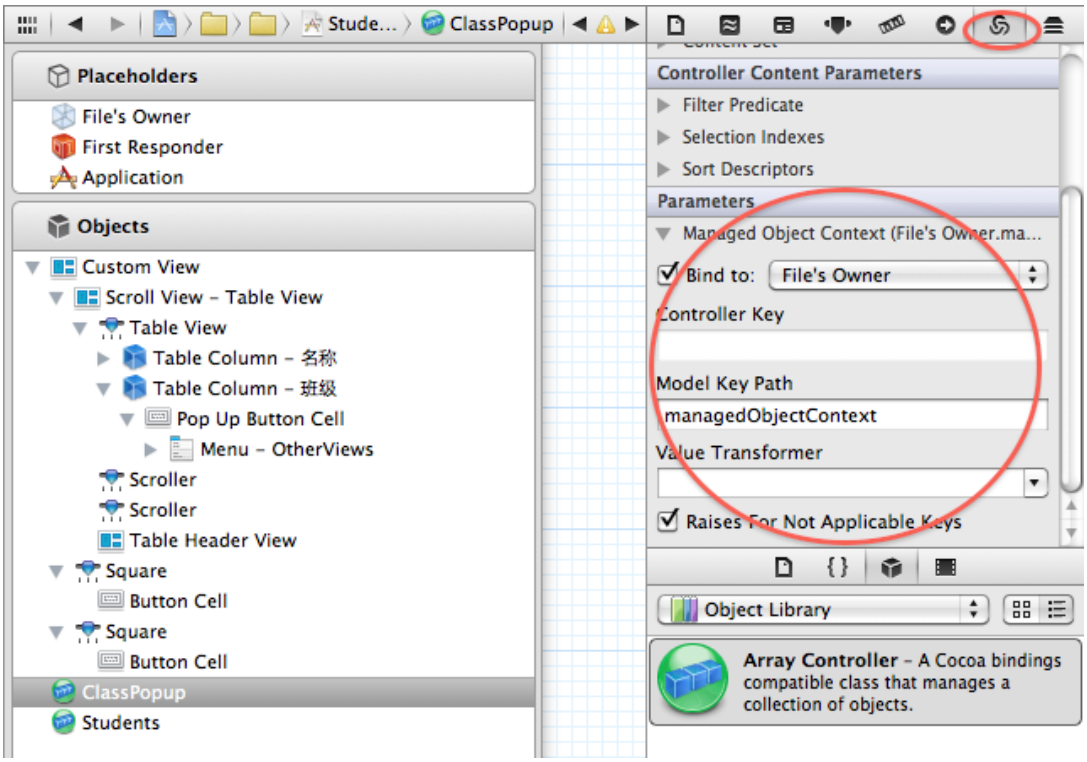
向其中拖入两个 NSArrayController：ClassPopup 和 Students。

设置 ClassPopup 的 Object Controller Mode 为 Entity Name，实体名为：ClassEntity，并勾选 Prepare Content。

设置 Students 的 Object Controller Mode 为 Entity Name，实体名为：StudentEntity，并勾选 Prepare Content。



上面的这些操作，ClassPopup ArrayController 管理 ClassEntity 的数据，Students ArrayController 管理 StudentEntity 的数据，后面我们就要将控件与这些 ArrayController 绑定起来。下面我们将这两个 NSArrayController 的 ManagedObjectContext 参数与 ManagedViewController(File's Owner) 中的 managedObjectContext 绑定起来，这样 NSDocuments 的 NSManagedObjectContext 就作用到的 ArrayController 中来了。下面只演示了 ClassPopup，请自行完成 Students 的绑定：



前面我们在 ManagedViewController 创建了一个 IBOutlet contentArrayController，现在是将它关联的时候了，使用 Control-Drag 将 File's Owner 的 contentArrayController 关联到 Students。重复上面的过程，选中 ClassView.xib，将 File's Owner 的类为 ClassViewController，并将其 view 指向 custom view。向其中拖入三个 NSArrayController：Classes，MonitorPopup 和 Students。

设置 Classes 的 Object Controller Mode 为 Entity Name，实体名为：ClassEntity，并勾选 Prepare Content。

将 Classes 的 ManagedObjectContext 参数与 ManagedViewController(File's Owner) 中的 managedObjectContext 绑定起来。

注意：这里没有对 MonitorPopup 和 Students 进行修改。

使用 Control-Drag 将 File's Owner 的 contentArrayController 关联到 Classes。

将 Students 和 MonitorPopup 的 Content set 绑定到 Classes 的 Model key path: students，表示这两个 ArrayController 是管理对应 ClassEntity 的 students 的数据。

至此，模型， ArrayController 都准备好了，下面我们将控件绑定到这些对象上。上面已经够繁琐的了，下面我们得更加仔细，很容易出错的。

选中 StudentView.xib，展开 Custom View 中的 TableView，直到我们看到名称和班级两个 Table Column。

选中名称列，将其 value 绑定到 Students，model key path 为：name，表明第一列显示学生的名称；

选择班级列，注意这一列是 popup button cell，将其 Content 绑定到 ClassPopup；

将其 ContentValues 绑定到 ClassPopup，model key path 为：name，表明第二列的选项为班级的名称；

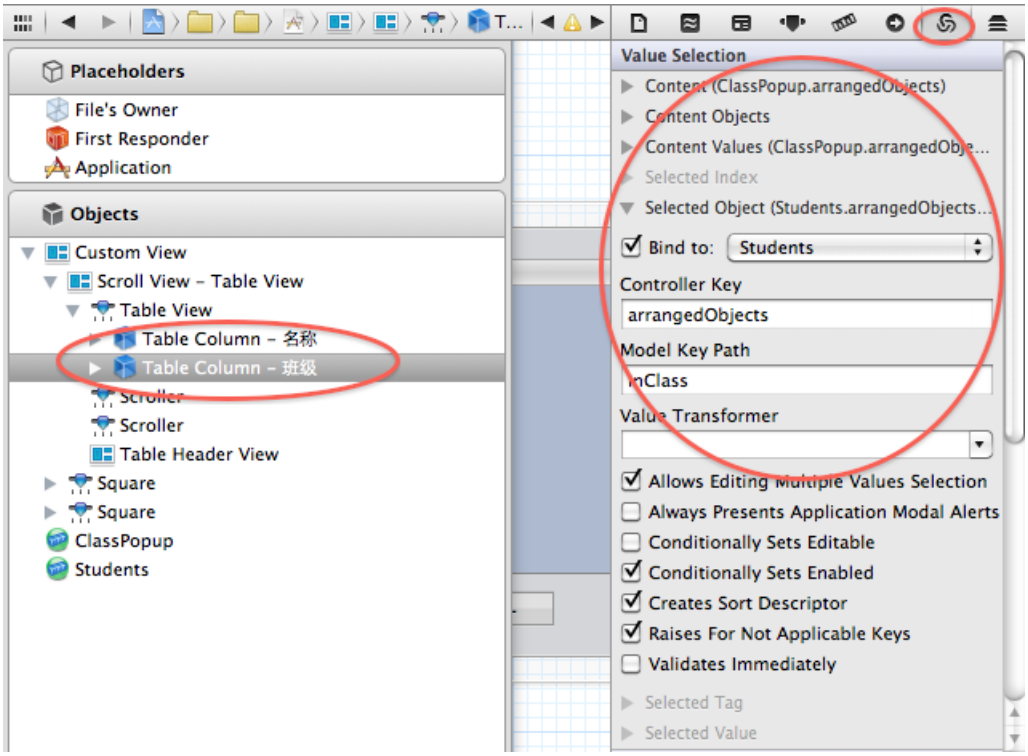
将其 Selected Object 绑定到 Students，model key path 为：inClass；表明将学生添加为选中班级的一员；

选中 + button，使用 Control+Drag 将其托拽到 Students 上，选择 add: 动作关联；

选中 - button，使用 Control+Drag 将其托拽到 Students 上，选择 remove: 动作关联；

选中 - button，将其 Eanbled 绑定到 Students, ctroller key 为：canRemove；

以上操作是将添加，删除学生的操作直接与 Students ArrayController 绑定，无需编写一点儿代码！



选中 ClassView.xib

展开 Custom View 中的班级表，，直到我们看到班级 Table Column：选择班级列，将其 value 绑定到 Classes，model key path 为：name，表明这一列显示班级的名称；

选中 Box，将其 Title 绑定到 Classed，model key path 为：name，并设置下方的 No Selection Placeholder 为：No Selection，Null Placeholder 为：Unnamed Class。表明 box 显示的信息为选中班级的信息，如果没有选中任何班级，则显示 No Selection。

展开 Box

选中 Pop up button

将其 Content 绑定到 MonitorPopup；

将其 ContentValues 绑定到 MonitorPopup，model key path 为：name，表明其选项为班级中的学生的名称；

将其 Selected Object 绑定到 Classes，model key path 为：monitor；表明将选中的学生当作该班级的班长；

展开学生 tabel view，直到我们看到学生这个 Table Column。

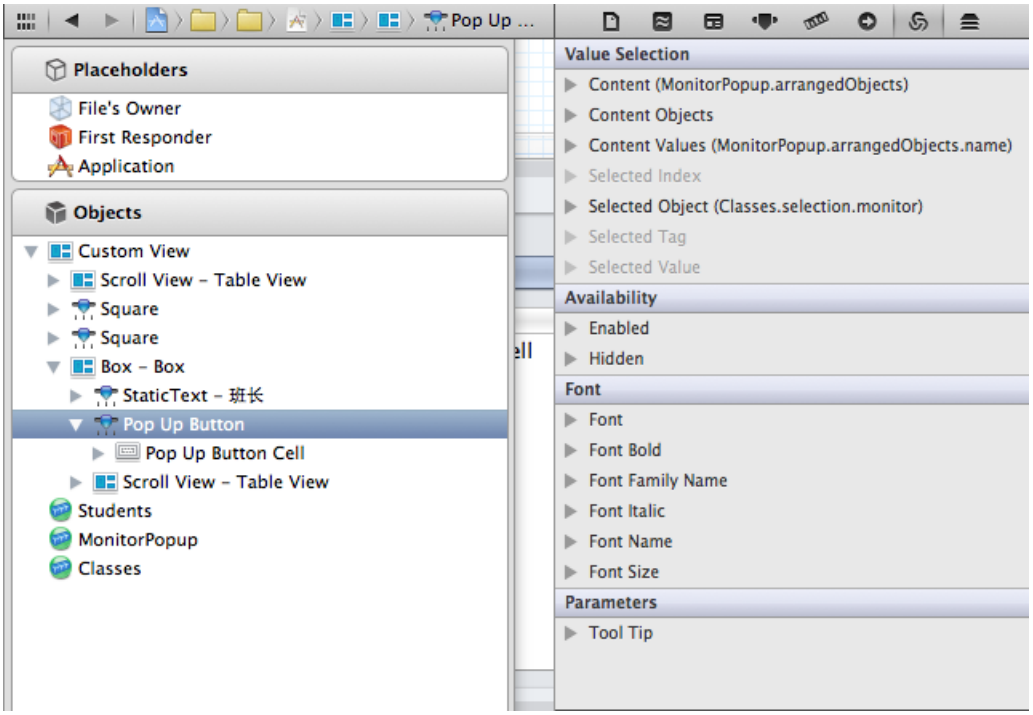
选择学生列，将其 Value 绑定到 Students，Model key path 为：name，表明学生列表显示该班级中所有学生的名称。

选中 + button，使用 Control+Drag 将其托拽到 Classes 上，选择 add: 动作关联；

选中 - button，使用 Control+Drag 将其托拽到 Classes 上，选择 remove: 动作关联；

选中 - button，将其 Eanbled 绑定到 Classes, ctroller key 为：canRemove；

以上操作是将添加，删除班级的操作直接与 Classes ArrayController 绑定。



至此，绑定也大功告成，如果你的程序运行不正确，多半是这地方的关联与绑定错了，请回到这部分，仔细检查每一项。

8，显示，切换 view。

现在到了设置主界面的时候，修改 MyDocument.h 中的代码如下：

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


@class ManagedViewController;


@interface MyDocument : NSPersistentDocument {
@private
    NSBox *        box;
    NSPopUpButton * popup;


    NSMutableArray *viewControllers;
    NSInteger      currentIndex;
}


@property (nonatomic, retain) IBOutlet NSBox *        box;
```

```
@property (nonatomic, retain) IBOutlet NSPopUpButton *  popup;


- (IBAction) changeViewController:(id)sender;
- (void) displayViewController:(ManagedViewController *)mvc;


@end
```

修改 MyDocument.m 中的代码如下：

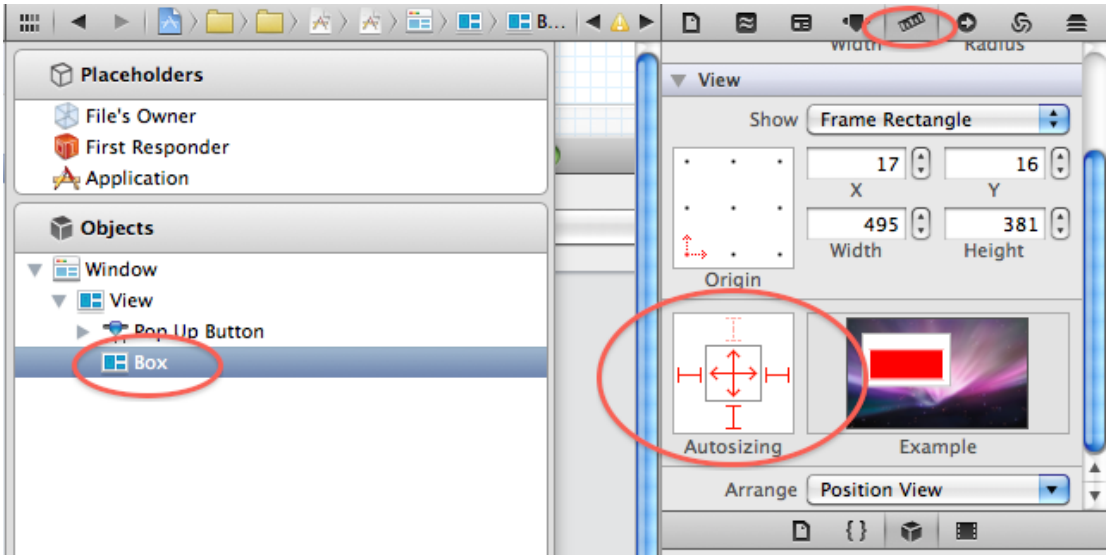
```
#import "MyDocument.h"
#import "ClassViewController.h"
#import "StudentViewController.h"


@implementation MyDocument
@synthesize popup;
@synthesize box;


- (id)init
{
    self = [super init];
    if (self) {
        // create view controllers
        //
        viewControllers = [[NSMutableArray alloc] init];


        ManagedViewController * mvc;
        mvc = [[ClassViewController alloc] init];
        [mvc setManagedObjectContext:[self managedObjectContext]];
        [viewControllers addObject:mvc];
        [mvc release];


        mvc = [[StudentViewController alloc] init];
        [mvc setManagedObjectContext:[self managedObjectContext]];
        [viewControllers addObject:mvc];
        [mvc release];
    }
    return self;
}


- (void) dealloc
```

```
{
    self.box = nil;
    self.popup = nil;
    [viewControllers release];
    [super dealloc];
}

- (NSString *)windowNibName
{
    // Override returning the nib file name of the document
    // If you need to use a subclass of NSWindowController or if your document supports multiple
NSWindowControllers, you should remove this method and override -makeWindowControllers instead.
    return @"MyDocument";
}

- (void)windowControllerDidLoadNib:(NSWindowController *)aController
{
    [super windowControllerDidLoadNib:aController];

    // init popup
    //
    NSMenu *menu = [popup menu];
    NSInteger itemCount = [viewControllers count];

    for (NSInteger i = 0; i < itemCount; i++) {
        NSViewController *vc = [viewControllers objectAtIndex:i];
        NSMenuItem *item = [[NSMenuItem alloc] initWithTitle:[vc title]
                                                      action:@selector(changeViewController:)
                                               keyEquivalent:@""];
        [item setTag:i];
        [menu addItem:item];
        [item release];
    }

    // display the first controller
    //
    currentIndex = 0;
    [self displayViewController:[viewControllers objectAtIndex:currentIndex]];
    [popup selectItemAtIndex:currentIndex];
```

```
}


#pragma mark -
#pragma mark Change Views


- (IBAction) changeViewController:(id)sender
{
    NSInteger tag = [sender tag];
    if (tag == currentIndex) {
        return;
    }


    currentIndex = tag;
    ManagedViewController *mvc = [viewControllers objectAtIndex:currentIndex];
    [self displayViewController:mvc];
}


- (void) displayViewController:(ManagedViewController *)mvc
{
    NSWindow *window = [box window];
    BOOL ended = [window makeFirstResponder:window];
    if (!ended) {
        NSBeep();
        return;
    }


    NSView *mvcView = [mvc view];


    // Adjust window's size and position
    //
    NSSize currentSize     = [[box contentView] frame].size;
    NSSize newSize         =  [mvcView frame].size;
    float deltaWidth       = newSize.width - currentSize.width;
    float deltaHeight      = newSize.height - currentSize.height;


    NSRect windowFrame     = [window frame];
    windowFrame.size.width  += deltaWidth;
    windowFrame.size.height += deltaHeight;
```

```
    windowFrame.origin.y    -= deltaHeight;


    [box setContentView:nil];
    [window setFrame:windowFrame display:YES animate:YES];


    [box setContentView:mvcView];


    // add viewController to the responder-chain
    //
    [mvcView setNextResponder:mvc];
    [mvc setNextResponder:box];
}


@end
```

在 MyDocument 中，我们创建了两个 ManagedViewController，并将 managedObjectContext 传入其中。这两个 ViewController 分别代表班级与学生两个界面，然后通过 popup button 的选择在他们之间切换显示；在 displayViewController 中，我们还根据当前界面的大小来调整主界面的大小。这需要我们设置主界面中 box 的自动大小。打开 MyDocument.xib，作如下设置：



然后，使用 Control+Drag，将 File's Owner 的 popup 和 popup button 相联，box 与 box 相联，并将 popup button 的 action 设置为 File's Owner 的 - (IBAction) changeViewController:(id)sender。

至此，所有的工作都完成了。编译运行程序，如果不出意外的话，我们应该可以添加学生，班级，并设置学生的班级，班级的班长等信息了。

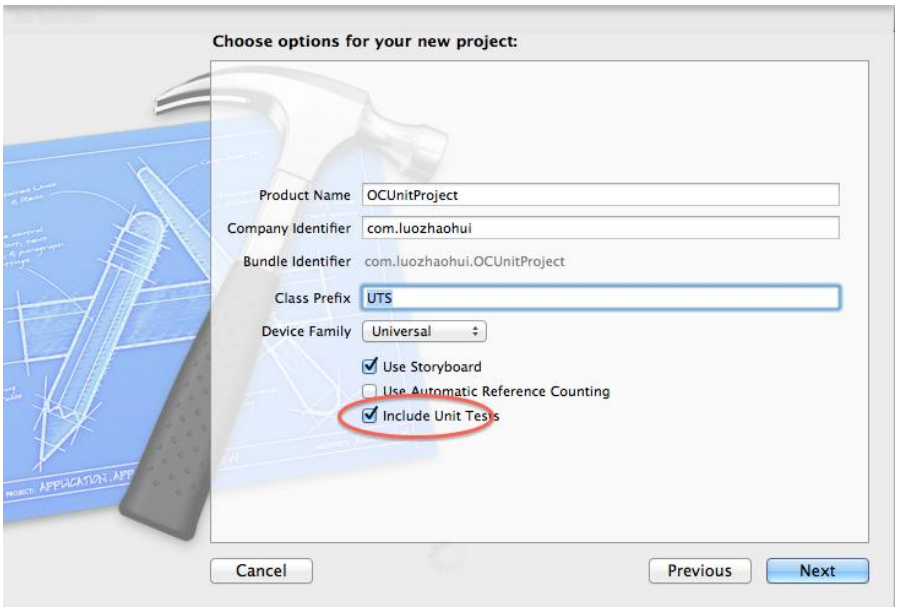
[调试]XCode 下的 iOS 单元测试

罗朝辉 (http://blog.csdn.net/kesalin)

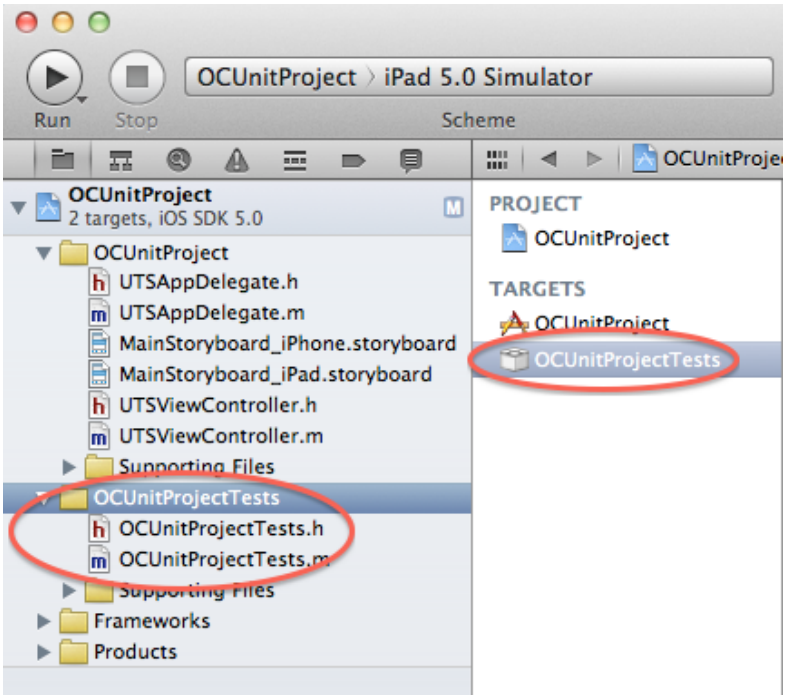
XCode 内置了 OCUnit 单元测试框架，但目前最好用的测试框架应该是 GHUnit。通过 GHUnit + OCMock 组合，我们可以在 iOS 下进行较强大的单元测试功能。本文将演示如何在 XCode 4.2 下使用 OCUnit，GHUnit 和 OCMock 进行单元测试。

OCUnit

在 XCode 下新建一个 OCUnitProject 工程，选中 Include Unit Tests 选择框，



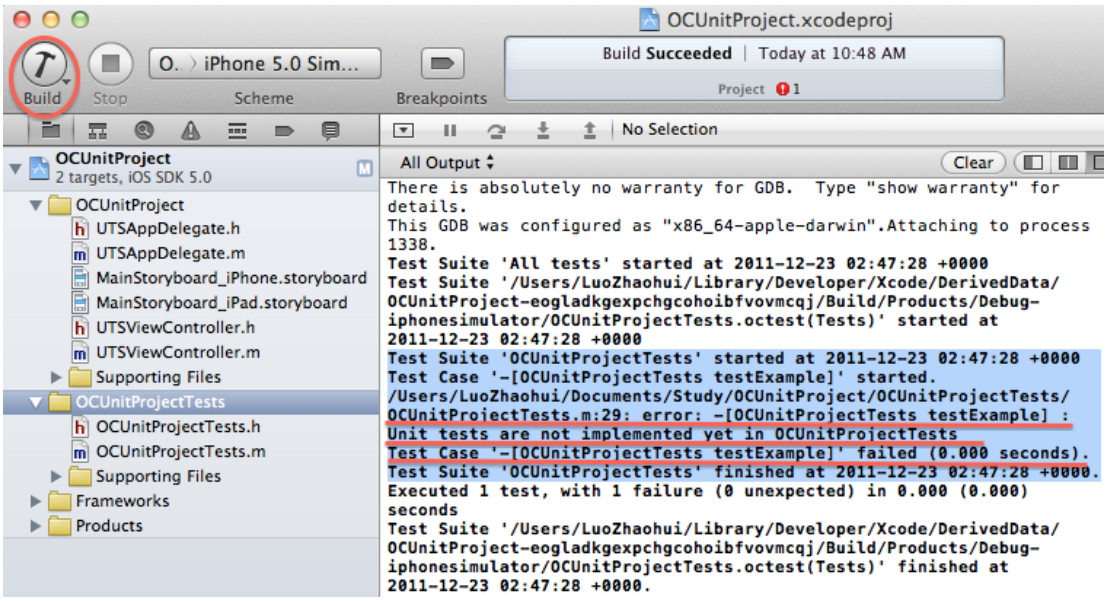
OCUnit 框架则会为我们自动添加 Unit Test 框架代码：



XCode 在 OCUnitProjectTests.m 中为我们自动生成了一个 Fail 的测试：

```
- (void)testExample

{

    STFail(@"Unit tests are not implemented yet in OCUnitProjectTests");

}
```

让我们来运行 Test，看看效果：



从图中的红色下划线部分可以看出，测试没有通过，符合预期。我们只要像类 OCUnitProjectTests 一样编写继承自 SenTestCase 类的子类，在其中添加形式如：- (void) testXXX(); 的测试函数既可，注意必须是一个无参无返回类型且名称是以 test 为前缀的函数。

OCUnit 的优点是官方支持，与 XCode 集成的比较好。

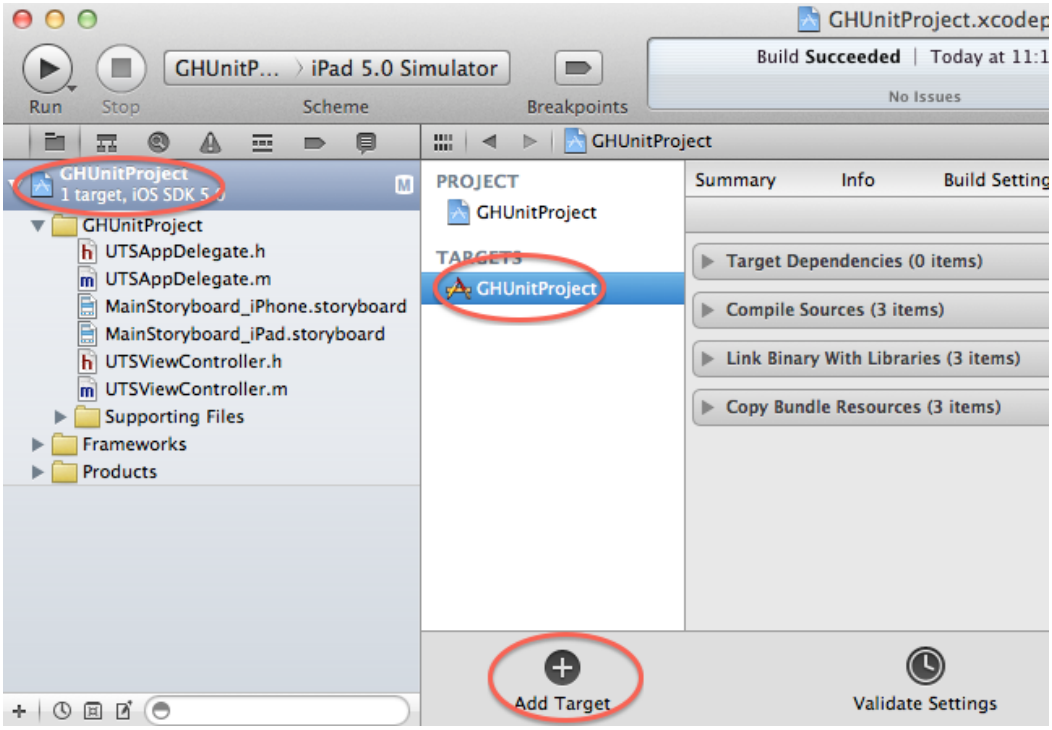
GHUnit

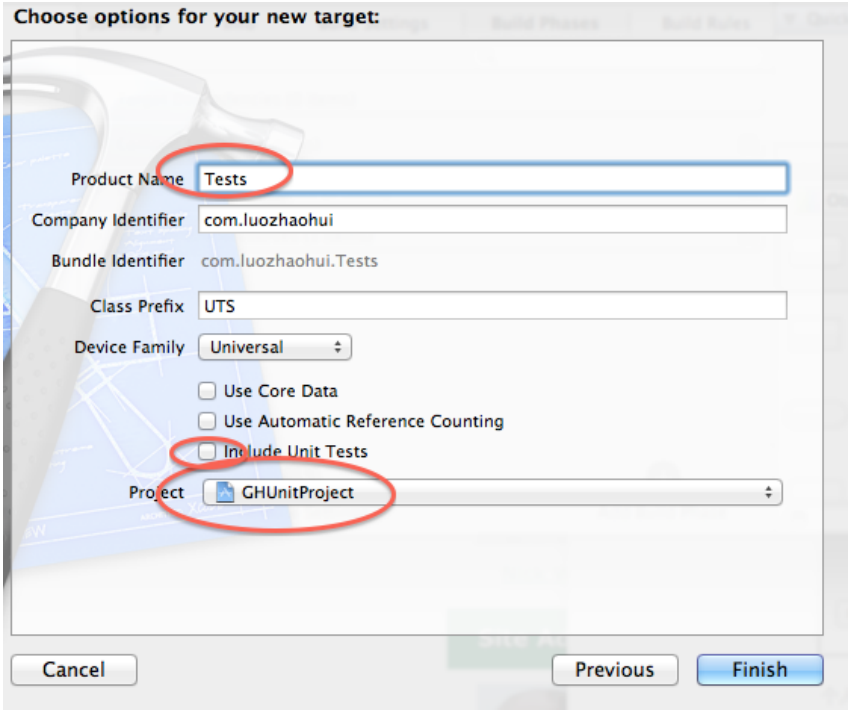
GHUnit 是一个开源的单元测试框架，具有可视化界面，功能亦相当强大。Mark 写了一篇 OCUnit vs GHUnit 的文章，有兴趣的童鞋可以看一看。OCMock 是由 Mulle Kybernetik 为 OS X 和 iOS 平台编写的遵循 mock object 理念的单元测试框架。

下面来介绍如何配置 GHUnit 和 OCMock

1，首先，创建一个名为 GHUnitProject 的单视图应用程序，注意：不要选中 Include Unit Tests 选择框。然后运行，应该出现白屏。

2，添加新的 test target，选中左边的工程名，点击右侧的 Add Target，新增一个名为 Tests 的 Empty Application 应用程序，让其附属于 GHUnitProject 注意：不要选中 Include Unit Tests 选择框。

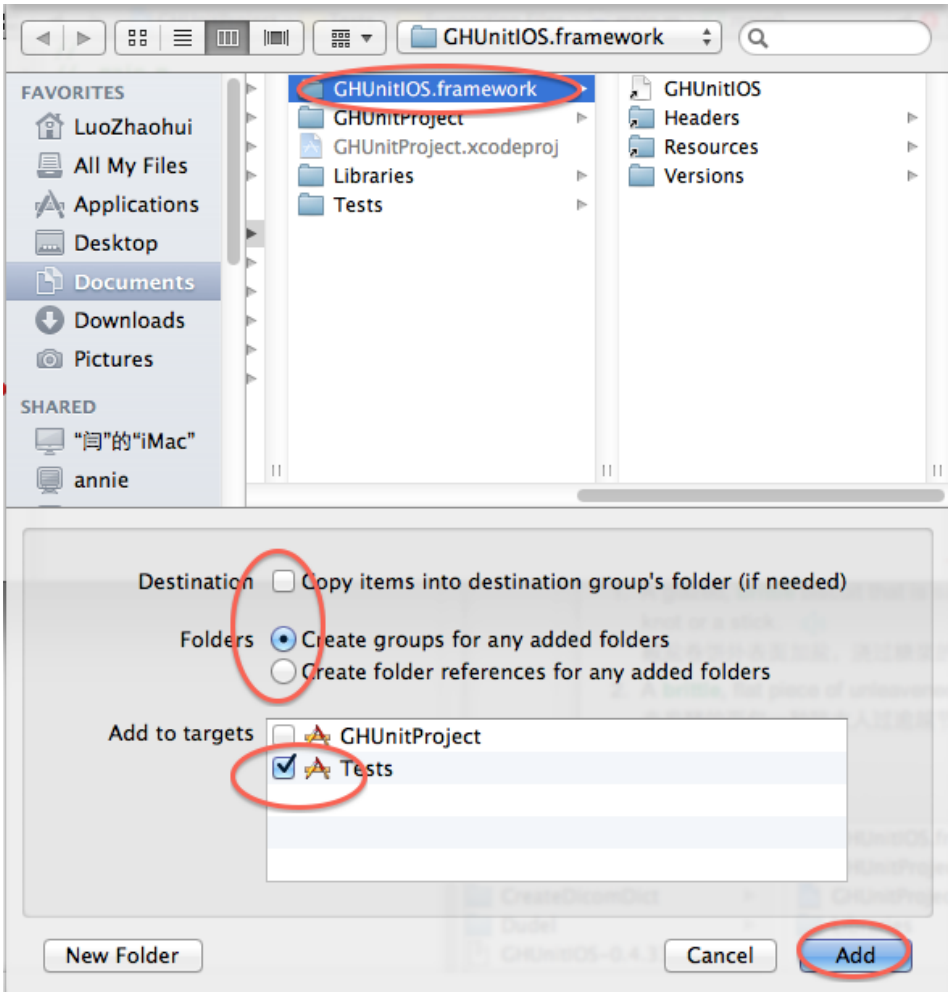




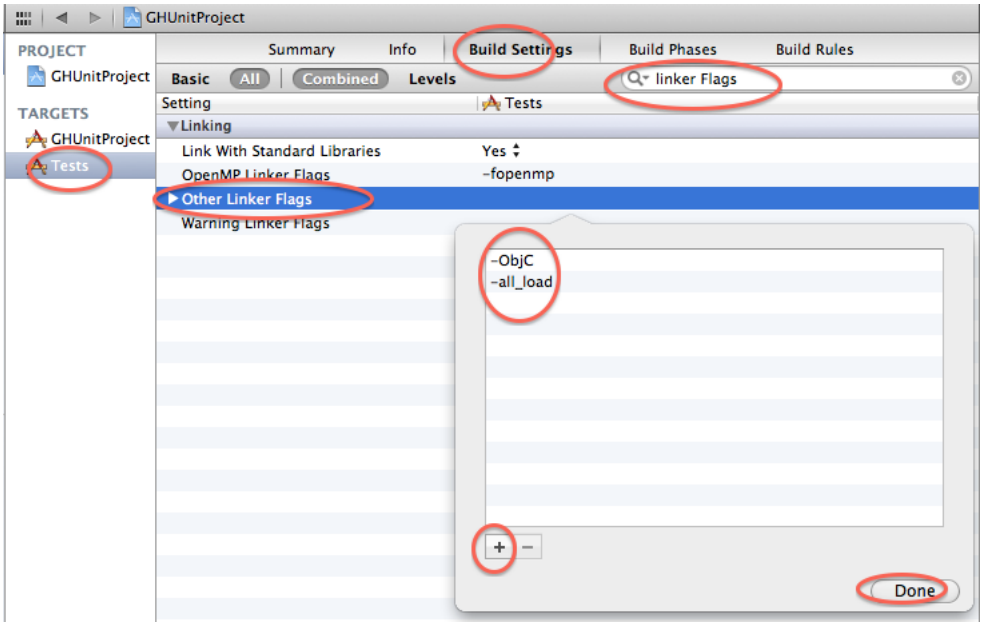
3，向 Tests 工程中（注意是 Tests 工程）添加 GHUnitIOS Framework。首先下载与 XCode 版本对应的 GHUnitIOS Framework。英文好的可以直接查看官方 iOS 版的安装文档：点此查看，跳过此第 3 节；否则请接着看。

3.1，解压 GHUnitIOS 框架到 GHUnitProject 下，让 GHUnitIOS.framework 与 Tests 在同一目录下。

3.2，回到 XCode，右击工程中的 Frameworks group，选中 Add Files to...菜单，选取 GHUnitIOS.framework ，注意 targets 要选择 Tests。



3.3，设置 Tests 的 Build Settings：在 Other Linker Flags 中增加两个 flag： -ObjC 和 -all_load。

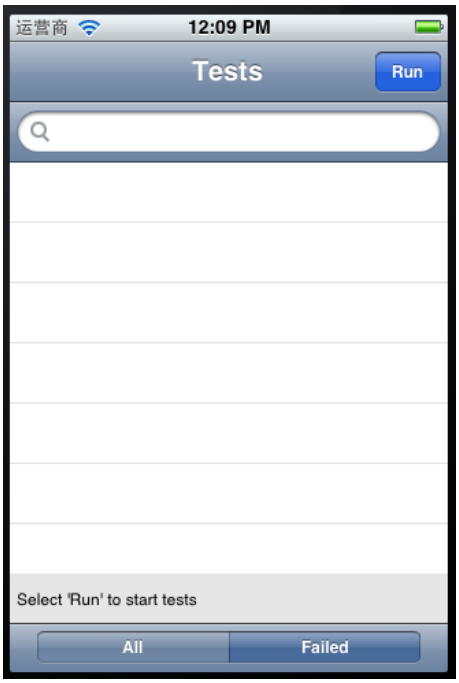


3.1，删除 Tests 工程中的 UTSAppDelegate.h 和 UTSAppDelegate.m 两个文件；

3.2，修改 Tests 工程中的 main.m 为:

```
#import <UIKit/UIKit.h>
#import <GHUnitIOS/GHUnitIOSAppDelegate.h>

int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([GHUnitIOSAppDelegate class]));
    }
}
```

3.3，选择编译目标 Tests>iPhone 5.0 Simulator，编译运行，应该能得到如下效果。目前我们还没有编写任何实际测试，所以列表为空。



4，编写 GHUnit 测试。向 Tests 工程中添加名为 GHUnitSampleTest 的 Objective C class。其内容如下：

GHUnitSampleTest.h

```
#import <GHUnitIOS/GHUnit.h>
@interface GHUnitSampleTest: GHTestCase
{}
@end
```

GHUnitSampleTest.m

```
#import "GHUnitSampleTest.h"
```

```
@implementation GHUnitSampleTest

- (void)testStrings

{

    NSString *string1 = @"a string";

    GHTestLog(@"I can log to the GHUnit test console: %@", string1);


    // Assert string1 is not NULL, with no custom error description

    GHAssertNotNULL(string1, nil);


    // Assert equal objects, add custom error description

    NSString *string2 = @"a string";

    GHAssertEqualObjects(string1, string2, @"A custom error message. string1 should be equal
to: %@.", string2);

}

@end
```

然后编译运行，点击 Run，效果如下：

图中的 All 栏显示所以的测试，Failed 栏显示没有通过的测试。强大吧，GHUnit。你可以向 GHUnitSampleTest 添加新的测试，比如：

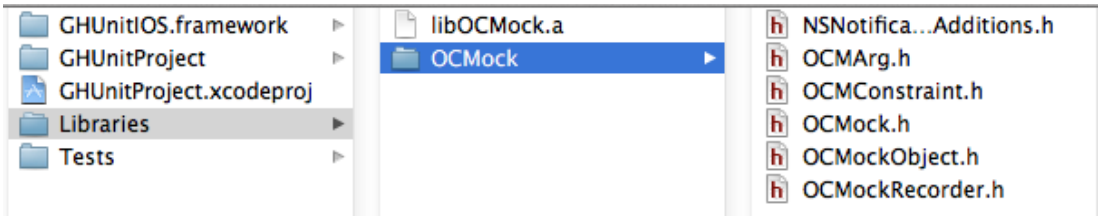
```
- (void)testSimpleFail

{

          GHAssertTrue(NO, nil);

}
```

我们可以向 Tests 添加更多测试类，只要该类是继承自 GHTestCase，且其中的测试方法都是无参无返回值且方法名字是以 test 为前缀即可。

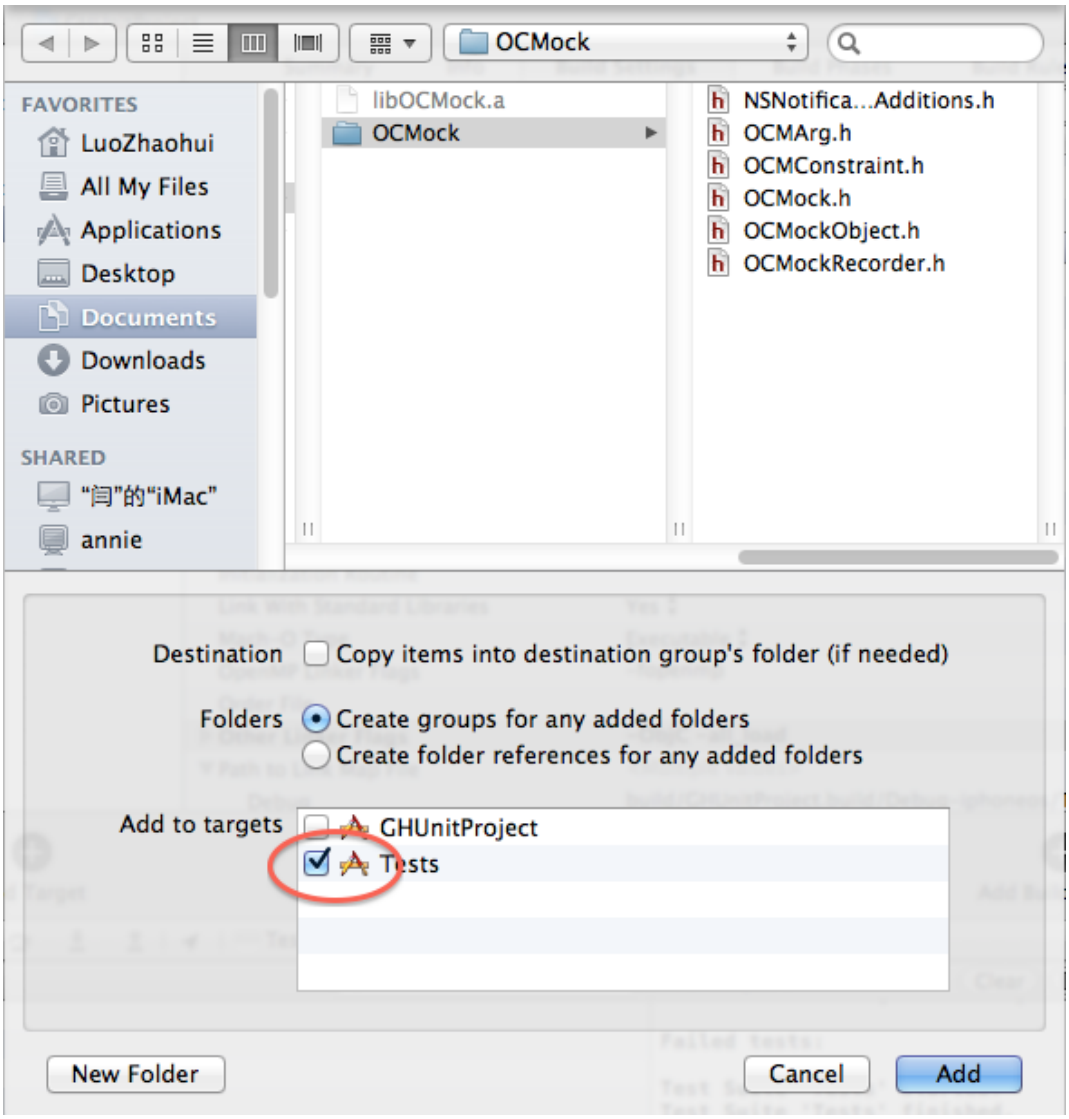
OCMock

下面我们来添加 OCMock。

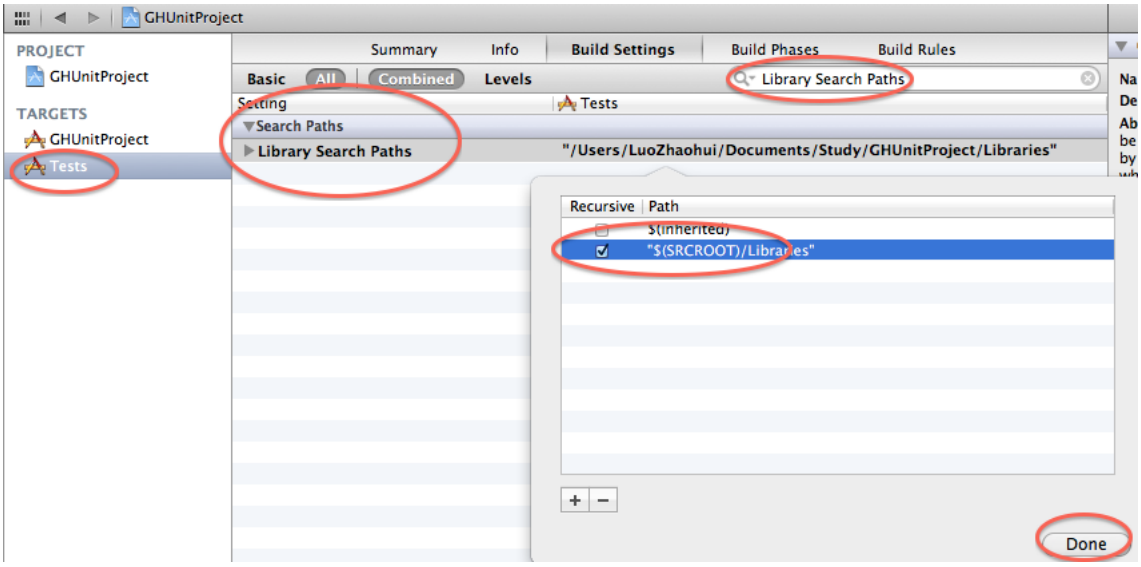
1， 我们只能以静态库的方式来添加 OCMock。在 GHUnitTest 目录下新建 Libraries 目录，该目录是与 Tests 目录平级的。下载静态库文件，解压头文件至该目录下。文件下载：头文件 libOCMock.a ，framework 文件：OCMock framework ，打开下载好的 ocmock-1.77.dmg，拷贝其中的'Release/Library/Headers/OCMock' 目录至 Libraries 下。最终目录结构如下：



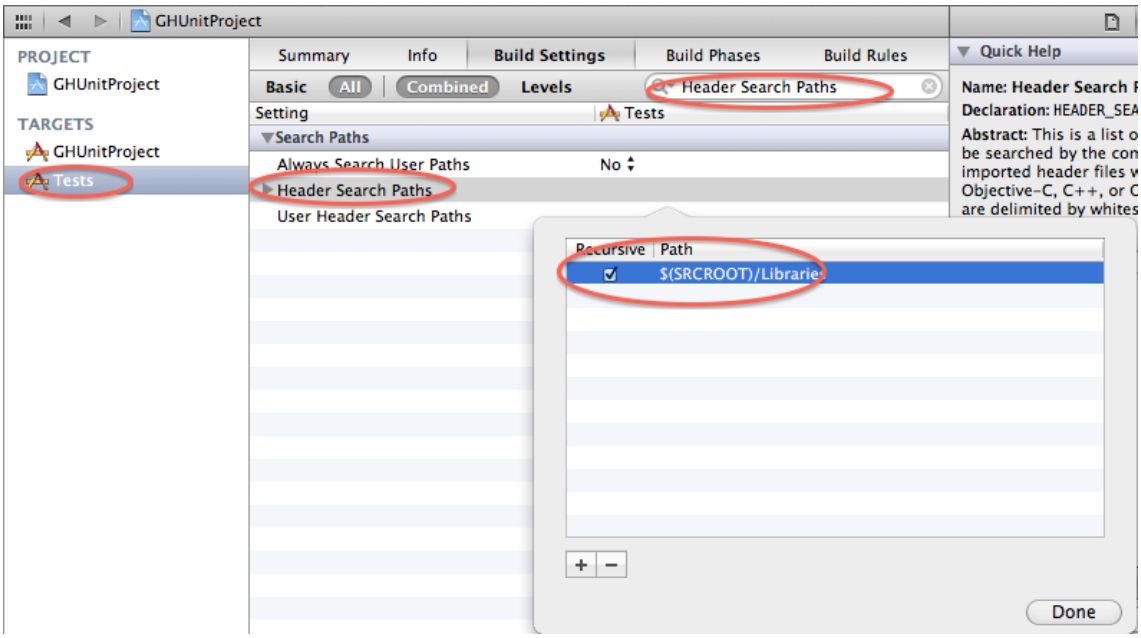
2，在 GHUnitTest 工程中新建名为 Libraries 的 group，导入 libOCMock.a 和目录 OCMock，注意 target 是 Tests。



3，设置 Tests 的 Build Setting。让 Libray Search Paths 包含 $(SRCROOT)/Libraries：



在 Header Search Paths 中增加 $(SRCROOT)/Libraries，并选中 Recursive 选择框。



4，编写 OCMock 测试。向 Tests 工程中添加名为 OCMockSampleTest 的 Objective C class。其内容如下：

OCMockSampleTest.h

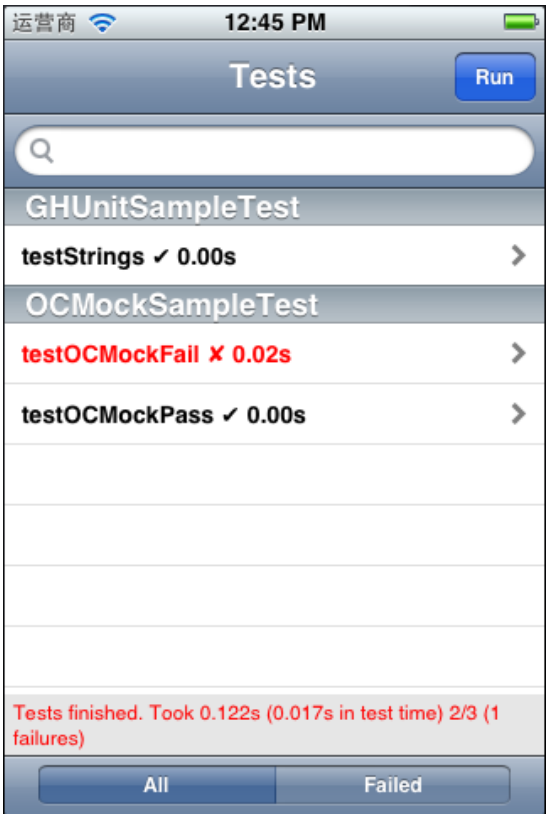
```
#import <GHUnitIOS/GHUnit.h>


@interface OCMockSampleTest : GHTestCase


@end
```

OCMockSampleTest.m

```
#import "OCMockSampleTest.h"
```

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


@implementation OCMockSampleTest


// simple test to ensure building, linking,
// and running test case works in the project
- (void)testOCMockPass
{
    id mock = [OCMockObject mockForClass:NSString.class];
    [[[mock stub] andReturn:@"mocktest"] lowercaseString];


    NSString *returnValue = [mock lowercaseString];
    GHAssertEqualObjects(@"mocktest", returnValue,
                         @"Should have returned the expected string.");
}


- (void)testOCMockFail
{
    id mock = [OCMockObject mockForClass:NSString.class];
    [[[mock stub] andReturn:@"mocktest"] lowercaseString];


    NSString *returnValue = [mock lowercaseString];
    GHAssertEqualObjects(@"thisIsTheWrongValueToCheck",
                         returnValue, @"Should have returned the expected string.");
}


@end
```

编译运行，点击 Run，效果如下图。



至此，iOS 下的 OCUnit，GHUnit，OCMock 单元测试介绍就到此结束了。当然还有其他一些测试框架，比如 google 出品的 GTM。

参考资料：

OCMock: http://ocmock.org/

Unit Testing in Xcode 4- use OCUnit and SenTest instead of GHUnit

GHUnit Reference: http://gabriel.github.com/gh-unit/

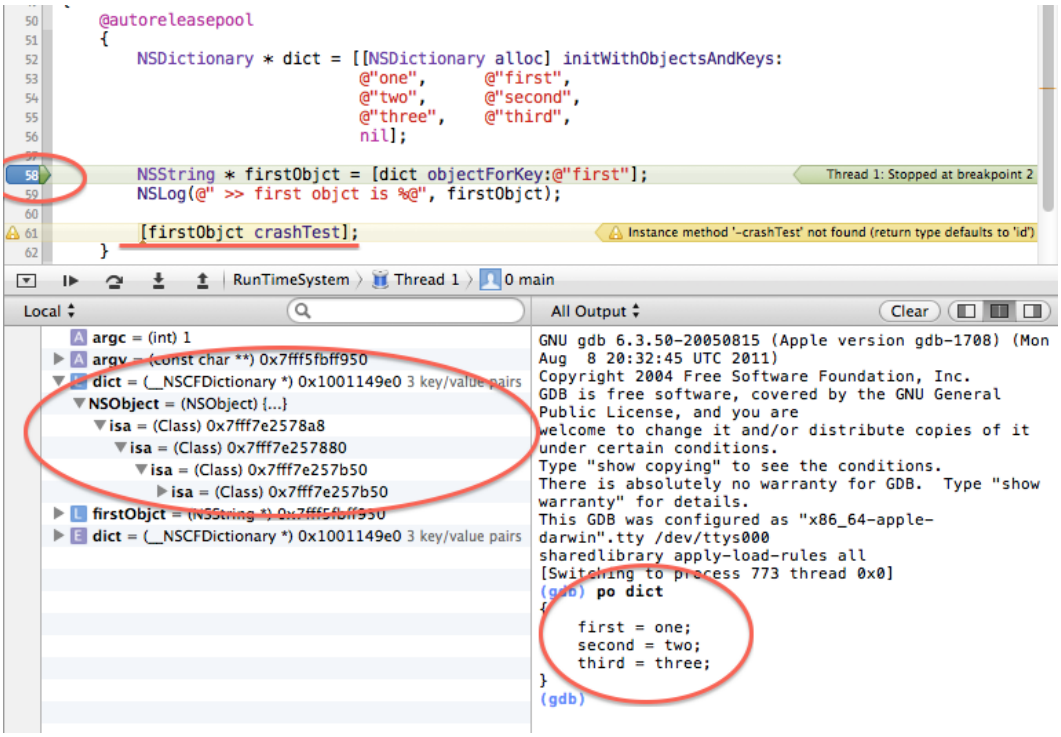
[调试]XCode 的一些调试技巧

罗朝辉 (http://blog.csdn.net/kesalin/)

XCode 内置 GDB，我们可以在命令行中使用 GDB 命令来调试我们的程序。下面将介绍一些常用的命令以及调试技巧。

po 命令：为 print object 的缩写，显示对象的文本描述（显示从对象的 description 消息获得的字符串信息）。

比如：



上图中，我使用 po 命令显示一个 NSDictionary 的内容。注意在左侧我们可以看到 dict 的一些信息：3 key/value pairs，显示该 dict 包含的数据量，而展开的信息显示 isa 层次体系（即 class 和 metaclass 结构关系）。我们可以右击左侧的 dict，选中“Print Description of "dict"”，则可以在控制台输出 dict 的详细信息：

```
Printing description of dict:
<CFBasicHash 0x1001149e0 [0x7fff7e27ff40]>{type = immutable dict, count = 3,
entries =>
        0 : <CFString 0x100002458 [0x7fff7e27ff40]>{contents = "first"} = <CFString 0x100002438 [0x7fff7e27ff40]>{contents = "one"}
        1 : <CFString 0x100002498 [0x7fff7e27ff40]>{contents = "second"} = <CFString 0x100002478 [0x7fff7e27ff40]>{contents = "two"}
        2 : <CFString 0x1000024d8 [0x7fff7e27ff40]>{contents = "third"} = <CFString 0x1000024b8 [0x7fff7e27ff40]>{contents = "three"}
}
(gdb)
```

print 命令：有点类似于格式化输出，可以输出对象的不同信息：

如：

```
(gdb) print (char *)[[dict description] cStringUsingEncoding:4]
$1 = 0x1001159c0 "{\n    first = one;\n    second = two;\n    third = three;\n}"
(gdb) print (int)[dict retainCount]
$2 = 1
(gdb)
```

注：4 是 NSUTF8StringEncoding 的值。

info 命令：我们可以查看内存地址所在信息

比如 "**info symbol 内存地址**" 可以获取内存地址所在的 symbol 相关信息：

```
(gdb) info symbol 0x00000001000017f7
main + 343 in section LC_SEGMENT.__TEXT.__text of
/Users/LuoZhaohui/Library/Developer/Xcode/DerivedData/RunTimeSystem-anzdlhiwvlbizpfureuvenvm
atnp/Build/Products/Debug/RunTimeSystem
```

比如 "**info line *内存地址**" 可以获取内存地址所在的代码行相关信息：

```
(gdb) info line *0x00000001000017f7
Line 62 of "/Users/LuoZhaohui/Documents/Study/RunTimeSystem/RunTimeSystem/main.m" starts at
address 0x1000017f7 <main+343> and ends at 0x10000180a <main+362>.
```

show 命令：显示 GDB 相关的信息。如：show version 显示 GDB 版本信息

```
(gdb) show version
GNU gdb 6.3.50-20050815 (Apple version gdb-1708) (Mon Aug  8 20:32:45 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".
```

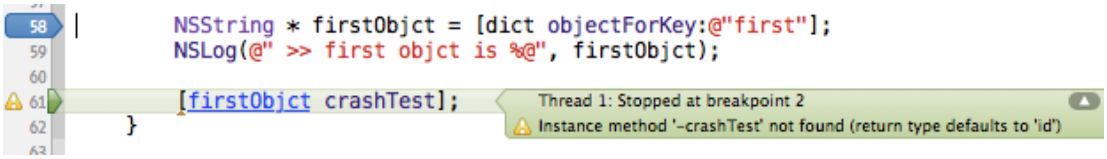
help 命令：如果忘记某条命令的语法了，可以使用 help 命令名 来获取帮助信息。如：help info 显示 info 命令的用法。

```
(gdb) help info
Generic command for showing things about the program being debugged.

List of info subcommands:
```

```
info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info args -- Argument variables of current stack frame
info auxv -- Display the inferior's auxiliary vector
info breakpoints -- Status of user-settable breakpoints
info catch -- Exceptions that can be caught in the current stack frame
info checkpoints -- Help
info classes -- All Objective-C classes
......

Type "help info" followed by info subcommand name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

在系统抛出异常处设置断点

有时候我们的程序不知道跑到哪个地方就 crash 了，而 crash 又很难重现。保守的做法是在系统抛出异常之前设置断点，具体来说是在 **objc_exception_throw** 处设置断点。设置步骤为：首先在 XCode 按 CMD + 6，进入断点管理窗口；然后点击右下方的 +，增加新的 Symbolic Breakpoint，在 Symbol 一栏输入：**objc_exception_throw，**然后点击 done，完成。这样在 Debug 模式下，如果程序即将抛出异常，就能在抛出异常处中断了。比如在前面的代码中，我让 [firstObjctcrashTest]; 抛出异常。在 objc_exception_throw 处设置断点之后，程序就能在该代码处中断了，我们从而知道代码在什么地方出问题了。



[版本管理]Mac 下配置 Git 服务器

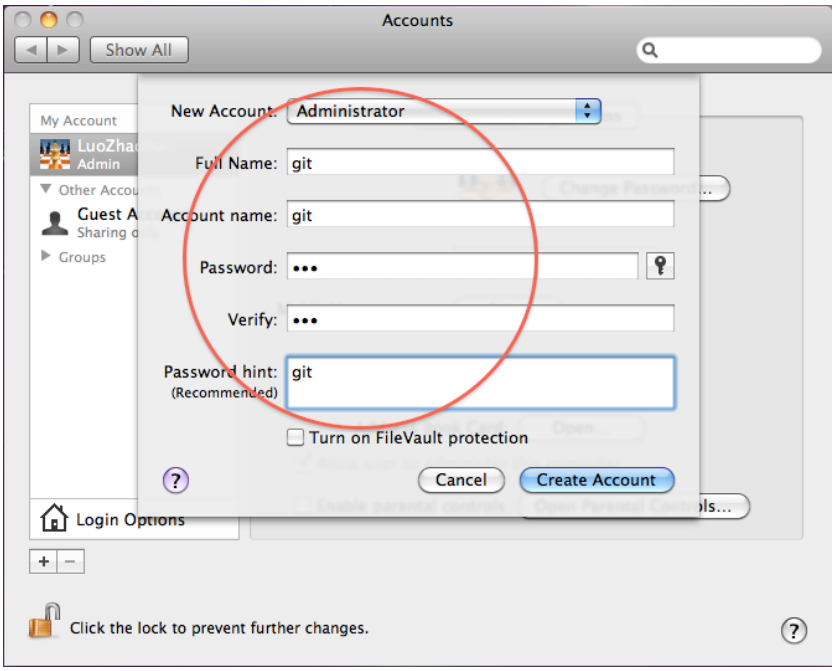
罗朝辉 (http://blog.csdn.net/kesalin)

XCode 4 默认支持 Git 作为代码仓库，当我们新建一个仓库的时候，可以勾选创建默认仓库，只不过这个仓库是在本地的。本文介绍如何在 mac 机器上创建 Git 服务器，总体思路是：使用 gitosis 来简化创建过程，在用作服务器的机器上创建一个名为 git 的账户来创建 git 服务器，其他客户端通过 ssh 机制访问 git 服务器。

本文文档：点此下载

一，创建 git 账户

1，在用作服务器的机器 Server 上创建 git 账户。我们可以通过 System Preferences->accounts 来添加。在这里我添加一个 git 的 administrator 账户，administrator 不是必须的，在这里仅仅为了方便。



2，设置远程访问

logout 当前账户，使用 git 账户登录；在 System Preferences->Sharing 中，勾选：Web Sharing 和 Remote Logig。

二，下载安装 gitosis

1， Mac Snow 默认已经为我们安装了 Git 和 Python，可以使用如下命令查看其版本信息：

```
yourname:~ git$ git --version
git version 1.7.3.4
yourname:~ git$ python --version
Python 2.6.1
```

2，通过命令 "git clone git://eagain.net/gitosis.git" 来下载 gitosis

```
yourname:~ git$ git clone git://eagain.net/gitosis.git

Cloning into gitosis ...
remote: Counting objects: 614, done.
remote: Compressing objects: 100% (183/183), done.
remote: Total 614 (delta 434), reused 594 (delta 422)
Receiving objects: 100% (614/614), 93.82 KiB | 45 KiB/s, done.
Resolving deltas: 100% (434/434), done.
```

3，进入 gitosis 目录，使用命令 "sudo python setup.py install" 来执行 python 脚本来安装 gitosis。

```
yourname:~ git$ cd gitosis/
yourname:gitosis git$ sudo python setup.py install
running install
running bdist_egg
running egg_info
creating gitosis.egg-info
……
Using /Library/Python/2.6/site-packages/setuptools-0.6c9-py2.6.egg
Finished processing dependencies for gitosis==0.2
```

三，制作 ssh rsa 公钥

1，回到 client 机器上，制作 ssh 公钥。在这里我的使用同一台机器上的另一个账户作为 client。如果作为 client 的机器与作为 server 的机器不是同一台，也是类型的流程：制作公钥，放置到服务的 /tmp 目录下。只不过在同一台机器上，我们可以通过开启另一个 terminal，使用 su 切换到 local 账户就可以同时操作两个账户。

```
yourname:~ git$ su local_account
Password:
bash-3.2$ cd ~
bash-3.2$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/local_account/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/local_account/.ssh/id_rsa.
Your public key has been saved in /Users/local_account/.ssh/id_rsa.pub.
```

```
bash-3.2$ cd .ssh
bash-3.2$ ls
id_rsa        id_rsa.pub
bash-3.2$ cp id_rsa.pub /tmp/yourame.pub
```

在上面的命令里，首先通过 su 切换到 local 账户（只有在同一台机器上才有效），然后进入到 local 账户的 home 目录，使用 ssh-keygen -t rsa 生成 id_rsa.pub，最后将该文件拷贝放置到 /tmp/yourname.pub,这样 git 账户就可以访问 yourname.pub 了,在这里改名是为了便于在 git 中辨识多个 client。

四，使用 ssh 公钥初始化 gitosis

1，不论你是以那种方式（邮件，usb 等等）拷贝 yourname.pub 至服务器的 /tmp/yourname.pub。下面的流程都是一样，登入服务器机器的 git 账户，进入先前提到 gitosis 目录，进行如下操作初始化 gitosis，初始化完成后，会在 git 的 home 下创建 repositories 目录。

```
yourname:gitosis git$ sudo -H -u git gitosis-init < /tmp/yourname.pub
Initialized empty Git repository in /Users/git/repositories/gitosis-admin.git/
Reinitialized existing Git repository in /Users/git/repositories/gitosis-admin.git/
```

在这里，会将该 client 当做认证受信任的账户，因此在 git 的 home 目录下会有记录，文件 authorized_keys 的内容与 yourname.pub 差不多。

```
yourname:~ git$ cd ~
yourname:~ git$ cd .ssh
yourname:.ssh git$ ls
authorized_keys
```

我们需要将 authorizd_keys 稍做修改，用文本编辑器打开它，删除里面的"command="gitosis-serve yourname",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty " 这一行：

```
yourname:.ssh git$ open -e authorized_keys
```

然后，我们对 post-update 赋予可写权限，以便 client 端可以提交更改。

```
yourname:gitosis git$ sudo chmod 755 /Users/git/repositories//gitosis-admin.git/hooks/post-update
Password:
yourname:.ssh git$ cd ~
yourname:~ git$ cd repositories/
yourname:repositories git$ ls
gitosis-admin.git
yourname:repositories git$
```

在上面的命令中可以看到，gitosis 也是作为仓库的形式给出，我们可以在其他账户下 checkout，然后对 gitosis 进行配置管理等等，而无需使用服务器的 git 账户进行。

最后一步，修改 git 账户的 PATH 路径。

```
yourname:gitosis git$ touch ~/.bashrc
yourname:gitosis git$ echo PATH=/usr/local/bin:/usr/local/git/bin:\$PATH > .bashrc
yourname:gitosis git$ echo export PATH >> .bashrc
yourname:gitosis git$ cat .bashrc
PATH=/usr/local/bin:/usr/local/git/bin:$PATH
export PATH
```

至此，服务器的配置完成。

五，client 配置

1，回到 local 账户，首先在 terminal 输入如下命令修改 local 的 git 配置：

```
bash-3.2$ git config --global user.name "yourgitname"
bash-3.2$ git config --global user.email "yourmail@yourcom.com"
```

2，测试服务器是否连接正确，将 10.1.4.211 换成你服务的名称或服务器地址即可。

```
yourname:~ local_account$ ssh git@10.1.4.211
Last login: Mon Nov  7 13:11:38 2011 from 10.1.4.211
```

3，在本地 clone 服务器仓库，下面以 gitosis-admin.git 为例：

```
bash-3.2$ git clone git@10.1.4.211:repositories/gitosis-admin.git

Cloning into gitosis-admin ...
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 5 (delta 0)
Receiving objects: 100% (5/5), done.
bash-3.2$ ls
Desktop        InstallApp     Music          Sites
Documents      Library        Pictures       gitosis-admin
Downloads      Movies         Public
bash-3.2$ git
```

在上面的输出中可以看到，我们已经成功 clone 服务器的 gitosis-admin 仓库至本地了。

4，在本地管理 gitosis-admin：

进入 gitosis-admin 目录，我们来查看一下其目录结构：gitosis.conf 文件是一个配置文件，里面定义哪些用户可以访问哪些仓库，我们可以修改这个配置；keydir 是存放 ssh 公钥的地方。

```
bash-3.2$ cd gitosis-admin/
bash-3.2$ ls
gitosis.conf keydir
bash-3.2$ cd keydir/
```

```
bash-3.2$ ls
yourname.pub
```

我们只需要将其他 client 产生的 ssh 公钥添加到 keydir 目录下，并在 gitosis.conf 文件中配置这些用户可以访问的仓库(用户名与放置在 keydir 下 sh 公钥名相同，这就是在前面我们要修改 ssh 公钥名的原因)，然后将改动提交至服务器，这样就可以让其他的 client 端访问服务器的代码仓库了。

[翻译]苹果 Cocoa 编码规范

罗朝辉(http://blog.csdn.net/kesalin)

本文档下载：点击这里

> Code Naming Basics 代码命名基础

在面向对象软件库的设计过程中，开发人员经常忽视对类，方法，函数，常量以及其他编程接口元素的命名。本节讨论大多数 Cocoa 接口的一些命名约定。

>> General Principles 一般性原则

>>> Clarity 清晰性

- 最好是既清晰又简短，但不要为简短而丧失清晰性

代码	点评
insertObject:atIndex:	good
insert:at:	不清晰；要插入什么？“at”表示什么？
removeObjectAtIndex:	good
removeObject:	这样也不错，因为方法是移除作为参数的对象
remove	不清晰；要移除什么？

- 名称通常不缩写，即使名称很长，也要拼写完全

代码	点评
destinationSelection	good
destSel	不清晰
setBackgroundColor:	good
setBkgdColor:	不清晰

你可能会认为某个缩写广为人知，但有可能并非如此，尤其是当你的代码被来自不同文化和语言背景的开发人员所使用时。

- 然而，你可以使用少数非常常见，历史悠久的缩写。请参考：”可接受的缩略名“一节

- 避免使用有歧义的 API 名称，如那些能被理解成多种意思的方法名称

代码	点评
sendPort	是发送端口还是返回一个发送端口？

displayName	是显示一个名称还是返回用户界面中控件的标题？

>>> Consistency 一致性

● 尽可能使用与 Cocoa 编程接口命名保持一致的名称。如果你不太确定某个命名的一致性，请浏览一下头文件或参考文档中的范例

● 在使用多态方法的类中，命名的一致性非常重要。在不同类中实现相同功能的方法应该具有相同的名称

代码	点评
- (int) tag	在 NSView, NSCell, NSControl 中有定义
- (void) setStringValue:(NSString *)	在许多 Cocoa classes 中有定义

请参考“方法参数”一节。

>>> No Self Reference 不要自我指涉

● 不要名称自我指涉

代码	点评
NSString	okey
NSStringObject	自我指涉

● 掩码（可使用位操作进行组合）和用作通知名称的常量不受该约定限制

代码	点评
NSUnderlineByWordMask	okey
NSTableViewColumnDidMoveNotification	okey

>> Prefixes 前缀

前缀是名称的重要组成部分。它们可以区分软件的功能范畴。通常，软件会被打包成一个框架或多个紧密相关的框架（如 Foundation 和 Application Kit 框架）。前缀可以防止第三方开发者与苹果公司之间的命名冲突（同样也可防止苹果内部不同框架之间的命名冲突）

● 前缀有规定的格式。它由两到三个大写字符组成，不能使用下划线与子前缀

前缀	Cocoa 框架
NS	Foundation
NS	Application Kit
AB	Address Book
IB	Interface Builder

● 命名 class，protocol，structure，函数，常量时使用前缀；命名成员方法时不使用前缀，因为方法已经在它所在类的命名空间种；同理，命名结构体字段时也不使用前缀

>> Typographic Conventions 书写约定

在为 API 元素命名时，请遵循如下一些简单的书写约定

● 对于包含多个单词的名称，不要使用标点符号作为名称的一部分或作为分隔符（下划线，破折号等）；此外，大写每个单词的首字符并将这些单词连续拼写在一起。请注意以下限制：

- 方法名小写第一个单词的首字符，大写后续所有单词的首字符。方法名不使用前缀。如： fileExistsAtPath:isDirectory:
  如果方法名以一个广为人知的大写首字母缩略词开头，该规则不适用，如：NSImage 中的 TIFFRepresentation
- 函数名和常量名使用与其关联类相同的前缀，并且要大写前缀后面所有单词的首字符。如：NSRunAlertPanel，NSCellDisabled
- 避免使用下划线来表示名称的私有属性。苹果公司保留该方式的使用。如果第三方这样使用可能会导致命名冲突，他们可能会在无意中用自己的方法覆盖掉已有的私有方法，这会导致严重的后果。请参考“私有方法”一节以了解私有 API 的命名约定的建议

>> Class and Protocol Names 类与协议命名

类名应包含一个明确描述该类（或类的对象）是什么或做什么的名词。类名要有合适的前缀（请参考“前缀”一节）。Foundation 及 Application Kit 有很多这样例子，如：NSString, NSData, NSScanner, NSApplication, NSButton 以及 NSEvent。

协议应该根据对方法的行为分组方式来命名。

● 大多数协议仅组合一组相关的方法，而不关联任何类，这种协议的命名应该使用动名词(ing)，以不与类名混淆。

代码	点评
NSLocking	good
NSLock	糟糕，它看起来像类名

● 有些协议组合一些彼此无关的方法（这样做是避免创建多个独立的小协议）。这样的协议倾向于与某个类关联在一起，该类是协议的主要体现者。在这种情形，我们约定协议的名称与该类同名。NSObject 协议就是这样一个例子。这个协议组合一组彼此无关的方法，有用于查询对象在其类层次中位置的方法，有使之能调用特殊方法的方法以及用于增减引用计数的方法。由于 NSObject 是这些方法的主要体现者，所以我们用类的名称命名这个协议。

>> Header Files 头文件

头文件的命名方式很重要，我们可以根据其命名知晓头文件的内容。

● 声明孤立的类或协议：将孤立的类或协议声明放置在单独的头文件中，该头文件名称与类或协议同名

头文件	声明
NSApplication.h	NSApplication 类

● 声明相关联的类或协议：将相关联的声明（类，类别及协议）放置在一个头文件中，该头文件名称与主要的类/类别/协议的名字相同。

头文件	声明
NSString.h	NSString 和 NSMutableString 类
NSLock.h	NSLocking 协议和 NSLock, NSConditionLock, NSRecursiveLock 类

● 包含框架头文件：每个框架应该包含一个与框架同名的头文件，该头文件包含该框架所有公开的头文件。

头文件	框架
Foundation.h	Foundation.framework

● 为已有框架中的某个类扩展 API：如果要在一个框架中声明属于另一个框架某个类的范畴类的方法，该头文件的命名形式为：原类名+“Additions”。如 Application Kit 中的 NSBundleAdditions.h

● 相关联的函数与数据类型：将相联的函数，常量，结构体以及其他数据类型放置到一个头文件中，并以合适的名字命名。如 Application Kit 中的 NSGraphics.h

> Naming Methods 方法命名

>> General Rules 一般性规则

为方法命名时，请考虑如下一些一般性规则：

● 小写第一个单词的首字符，大写随后单词的首字符，不使用前缀。请参考“书写约定”一节。有两种例外情况：1，方法名以广为人知的大写字母缩略词（如 TIFF or PDF）开头；2，私有方法可以使用统一的前缀来分组和辨识，请参考“私有方法”一节

● 表示对象行为的方法，名称以动词开头：

```
- (void) invokeWithTarget:(id)target:
```

```
- (void) selectTabViewItem:(NSTableViewItem *)tableViewItem
```

名称中不要出现 do 或 does，因为这些助动词没什么实际意义。也不要在动词前使用副词或形容词修饰

● 如果方法返回方法接收者的某个属性，直接用属性名称命名。不要使用 get，除非是间接返回一个或多个值。请参考“访问方法”一节。

- (NSSize) cellSize;	对
- (NSSize) calcCellSize;	错
- (NSSize) getCellSize;	错

● 参数要用描述该参数的关键字命名

- (void) sendAction:(SEL)aSelector to:(id)anObject forAllCells:(BOOL)flag;	对
- (void) sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;	错

● 参数前面的单词要能描述该参数。

- (id) viewWithTag:(int)aTag;	对
- (id) taggedView:(int)aTag;	错

● 细化基类中的已有方法：创建一个新方法，其名称是在被细化方法名称后面追加参数关键词

- (id) initWithFrame:(NSRect)frameRect;	NSView
- (id) initWithFrame:(NSRect)frameRect mode:(ind)aMode cellClass:(Class)factoryId numberOfRows:(int)rowsHigh numberOfColumns:(int)colsWide;	NSMatrix - NSView 的子类

● 不要使用 and 来连接用属性作参数关键字

- (int) runModalForDirectory:(NSString *)path file:(NSString *)name types:(NSArray *)fileTypes;	对
- (int) runModalForDirectory:(NSString *)path addFile:(NSString *)name addTypes:(NSArray *)fileTypes;	错

虽然上面的例子中使用 add 看起来也不错，但当你方法有太多参数关键字时就有问题。

● 如果方法描述两种独立的行为，使用 and 来串接它们

- (BOOL) openFile:(NSString *)fullPath withApplication:(NSString *)appName andDeactivate:(BOOL)flag;	NSWorkspace

>> Accessor Methods 访问方法

访问方法是对象属性的读取与设置方法。其命名有特定的格式依赖于属性的描述内容。

● 如果属性是用名词描述的，则命名格式为：

- (void) setNoun:(type)aNoun;

- (type) noun;

例如：

- (void) setgColor:(NSColor *)aColor;

- (NSColor *) color;

● 如果属性是用形容词描述的，则命名格式为：

- (void) setAdjective:(BOOL)flag;

- (BOOL) isAdjective;

例如：

- (void) setEditable:(BOOL)flag;

- (BOOL) isEditable;

● 如果属性是用动词描述的，则命名格式为：（动词要用现在时时态）

- (void) setVerbObject:(BOOL)flag;

- (BOOL) verbObject;

例如：

- (void) setShowAlpha:(BOOL)flag;

- (BOOL) showsAlpha;

● 不要使用动词的过去分词形式作形容词使用

- (void) setAcceptsGlyphInfo:(BOOL)flag;	对
- (BOOL) acceptsGlyphInfo;	对
- (void) setGlyphInfoAccepted:(BOOL)flag;	错
- (BOOL) glyphInfoAccepted;	错

● 可以使用情态动词（can, should, will 等）来提高清晰性，但不要使用 do 或 does

- (void) setCanHide:(BOOL)flag;	对
- (BOOL) canHide;	对
- (void) setShouldCloseDocument:(BOOL)flag;	对
- (void) shouldCloseDocument;	对
- (void) setDoseAcceptGlyphInfo:(BOOL)flag;	错
- (BOOL) doseAcceptGlyphInfo;	错

● 只有在方法需要间接返回多个值的情况下，才使用 get

- (void) getLineDash:(float *)pattern count:(int *)count phase:(float *)phase;	NSBezierPath

像上面这样的方法，在其实现里应允许接受 NULL 作为其 in/out 参数，以表示调用者对一个或多个返回值不感兴趣。

>> Delegate Methods 委托方法

委托方法是那些在特定事件发生时可被对象调用，并声明在对象的委托类中的方法。它们有独特的命名约定，这些命名约定同样也适用于对象的数据源方法。

● 名称以标示发送消息的对象的类名开头，省略类名的前缀并小写类第一个字符

- (BOOL) tableView:(NSTableView *)tableView shouldSelectRow:(int)row;
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;

● 冒号紧跟在类名之后（随后的那个参数表示委派的对象）。该规则不适用于只有一个 sender 参数的方法

- (BOOL) applicationOpenUntitledFile:(NSApplication *)sender;

- 上面的那条规则也不适用于响应通知的方法。在这种情况下，方法的唯一参数表示通知对象

```
- (void) windowDidChangeScreen:(NSNotification *)notification;
```

- 用于通知委托对象操作即将发生或已经发生的方法名中要使用 did 或 will

```
- (void) browserDidScroll:(NSBrowser *)sender;
```

```
- (NSUndoManager *) windowWillReturnUndoManager:(NSWindow *)window;
```

- 用于询问委托对象可否执行某操作的方法名中可使用 did 或 will，但最好使用 should

```
- (BOOL) windowShouldClose:(id)sender;
```

>> Collection Methods 集合方法

管理对象（集合中的对象被称之为元素）的集合类，约定要具备如下形式的方法：

```
- (void) addElement:(elementType)adObj;
```

```
- (void) removeElement:(elementType)anObj;
```

```
- (NSArray *)elements;
```

例如：

```
- (void) addLayoutManager:(NSLayoutManager *)adObj;
```

```
- (void) removeLayoutManager:(NSLayoutManager *)anObj;
```

```
- (NSArray *)layoutManagers;
```

集合方法命名有如下一些限制和约定：

- 如果集合中的元素无序，返回 NSSet，而不是 NSArray
- 如果将元素插入指定位置的功能很重要，则需具备如下方法：

```
- (void) insertElement:(elementType)anObj atIndex:(int)index;
```

```
- (void) removeElementAtIndex:(int)index;
```

集合方法的实现要考虑如下细节：

- 以上集合类方法通常负责管理元素的所有者关系，在 add 或 insert 的实现代码里会 retain 元素，在 remove 的实现代码中会 release 元素
- 当被插入的对象需要持有指向集合对象的指针时，通常使用 set... 来命名其设置该指针的方法，且不要 retain 集合对象。比如上面的 insertLayerManager:atIndex: 这种情形，NSLayoutManager 类使用如下方法：

```
- (void) setTextStorage:(NSTextStorage *)textStorage;
```

```
- (NSTextStorage *)textStorage;
```

通常你不会直接调用 setTextStorage:，而是覆写它。

另一个关于集合约定的例子来自 NSWindow 类：

```
- (void) addChildWindow:(NSWindow *)childWin ordered:(NSWindowOrderingMode)place;
- (void) removeChildWindow:(NSWindow *)childWin;
- (NSArray *)childWindows;
- (NSWindow *) parentWindow;
- (void) setParentWindow:(NSWindow *)window;
```

>> Method Arguments 方法参数

命名方法参数时要考虑如下规则：

- 如同方法名，参数名小写第一个单词的首字符，大写后继单词的首字符。如：removeObject:(id)anObject
- 不要在参数名中使用 pointer 或 ptr，让参数的类型来说明它是指针
- 避免使用 one， two，...，作为参数名
- 避免为节省几个字符而缩写

按照 Cocoa 惯例，以下关键字与参数联合使用：

```
...action:(SEL)aSelector
..alignment:(int)mode
...atIndex:(int)index
...content:(NSRect)aRect
...doubleValue:(double)aDouble
...floatValue:(float)aFloat
...font:(NSFont *)fontObj
...frame:(NSRect)frameRect
...intValue:(int)anInt
...keyEquivalent:(NSString *0charCode
...length:(int)numBytes
...point:(NSPoint)aPoint
...stringValue:(NSString *)aString
...tag:(int)anInt
...target:(id)anObject
...title:(NSString *)aString
```

>> Private Methods 私有方法

大多数情况下，私有方法命名相同与公共方法命名约定相同，但通常我们约定给私有方法添加前缀，以便与公共方法区分开来。即使这样，私有方法的名称很容易导致特别的问题。当你设计一个继承自 Cocoa framework 某个类的子类时，你无法知道你的私有方法是否不小心覆盖了框架中基类的同名方法。

Cocoa framework 的私有方法名称通常以下划线作为前缀（如：_fooData），以标示其私有属性。基于这样的事实，遵循以下两条建议：

- 不要使用下划线作为你自己的私有方法名称的前缀，Apple 保留这种用法。
- 若要继承 Cocoa framework 中一个超大的类（如：NSView），并且想要使你的私有方法名称与基类中的区别开来，你可以为你的私有方法名称添加你自己的前缀。这个前缀应该具有唯一性，如基于你公司的名称，或工程的名称，并以“XX_”形式给出。比如你的工程名为"Byte Flogger"，那么就可以是“BF_addObject:”

尽管为私有方法名称添加前缀的建议与前面类中方法命名的约定冲突，这里的意图有所不同：为了防止不小心地覆盖基类中的私有方法。

> Naming Functions 函数命名

Objective-C 允许通过函数（C 形式的函数）描述行为，就如成员方法一样。 你应当优先使用函数，而不是类方法，如果隐含的类为单例或在处理函数子系统时。

函数命名应该遵循如下几条规则：

- 函数命名与方法命名相似，但有几点不同：
    - 它们有前缀，其前缀与你使用的类和常量的前缀相同
    - 大写前缀后紧跟的第一个单词首字符
- 大多数函数名称以动词开头，这个动词描述该函数的行为

```
NSHighlightRect
```

```
NSDeallocateObject
```

查询属性的函数有个更多的规则要遵循：

- 查询第一个参数的属性的函数，省略动词

```
unsigned int NSEventMaskFromType(NSEventType type)
```

```
float NSHeight(NSRect rect)
```

- 返回值为引用的方法，使用 Get

```
const char *NSGetSizeAndAlignment(const char *typePtr, unsigned int *sizep, unsigned int *alignp)
```

- 返回 boolean 值的函数，名称使用判断动词 is/does 开头

```
BOOL NSDecimalIsNotANumber(const NSDecimal *decimal)
```

> Naming Instance Variables and Data Types 实例变量与数据类型命名

这一节描述实例变量，常量，异常以及通知的命名约定。

>> Instance Variables 实例变量

在为类添加实例变量是要注意：

- 避免创建 public 实例变量
- 使用 @private，@protected 显式限定实例变量的访问权限
- 确保实例变量名简明扼要地描述了它所代表的属性

如果实例变量别设计为可被访问的，确保编写了访问方法

>> Constants 常量

常量命名规则根据常量创建的方式不同而大不同。

>>> 枚举常量

- 使用枚举来定义一组相关的整数常量
- 枚举常量与其 typedef 命名遵守函数命名规则。如：来自 NSMatrix.h 中的例子：（本例中的 typedef tag（_NSMatrixMode）不是必须的）

```
typedef enum _NSMatrixMode {
    NSRadioModeMatrix       = 0,
    NSHighlightModeMatrix = 1;
    NSListModeMatrix          = 2,
    NSTrackModeMatrix       = 3
} NSMatrixMode;
```

- 位掩码常量可以使用不具名枚举。如：

```
enum {
    NSBorderlessWindowMask          = 0,
    NSTitledWindowMask                 = 1 << 0,
    NSClosableWindowMask            = 1 << 1,
    NSMiniaturizableWindowMask  = 1 << 2,
    NSResizableWindowMask          = 1 << 3
};
```

>>> const 常量

- 使用 const 来修饰浮点常量或彼此没有关联的整数常量
- 枚举常量命名规则与函数命名规则相同。const 常量命名范例：

```
const float NSLightGray;
```

>>> 其他常量

- 通常不使用 #define 来创建常量。如上面所述，整数常量请使用枚举，浮点数常量请使用 const

● 使用大写字母来定义预处理编译宏。如：#ifdef DEBUG

● 编译器定义的宏名首尾都有双下划线。如：__MACH__

● 为 notification 名及 dictionary key 定义字符串常量，从而能够利用编译器的拼写检查，减少书写错误。Cocoa 框架提供了很多这样的范例：

```
APPKIT_EXTERN NSString *NSPrintCopies;
```

实际的字符串值在实现文件中赋予。（注意：APPKIT_EXTERN 宏等价于 Objective-C 中 extern）

>> Exceptions and Notifications 异常与通知

异常与通知的命名遵循相似的规则，但是它们有各自推荐的使用模式。

>>> 异常

虽然你可以处于任何目的而使用异常（由 NSException 类及相关类实现），Cocoa 通常不使用异常来处理常规的，可预料的错误。在这些情形下，使用诸如 nil, NULL, NO 或错误代码之类的返回值。异常的典型应用类似数组越界之类的编程错误。

异常由具有如下形式的全局 NSString 对象标识：

[Prefix] + UniquePartOfName + Exception

UniquePartOfName 部分是有连续的首字符大写的单词组成。例如：

NSColorListIOException
NSColorListNotEditableException
NSDraggingException
NSFontUnavailableException
NSIllegalSelectorException

>>> 通知

如果一个类有委托，那它的大部分通知可能由其委托的委托方法来处理。这些通知的名称应该能够反应其响应的委托方法。比如，当应用程序提交 NSApplicationDidBecomeActiveNotification 通知时，全局 NSApplication 对象的委托会注册从而能够接收 applicaitonDidBecomeActive: 消息。

通知由具有如下形式的全局 NSString 对象标识：

[相关联类的名称] + [Did 或 Will] + [UniquePartOfName] + Notification

例如：

NSApplicationDidBecomeActiveNotification
NSWindowDidMiniaturizeNotification
NSTextViewDidChangeSelectionNotification
NSColorPanelColorDidChangeNotification

> 可接受的缩略语

在设计编程接口时，通常名称不要缩写。然而，下面列出的缩写要么是固定下来的要么是过去被广泛使用的，所以你可以继续使用。关于缩写有一些额外的注意事项：

● 标准 C 库中长期使用的缩写形式是可以接受的。如："alloc"，"getc"

- 你可以在参数名中更自由地使用缩写。如：imageRep，col（column），obj，otherWin

常见的缩写

缩写	含义	缩写	含义
alloc	Allocate	msg	Message
alt	Alternate	nib	Interface Builder archive
app	Application	pboard	Pasteboard
calc	Calculate	rect	Rectangle
dealloc	Deallocate	Rep	Representation
func	Function	temp	Temporary
horiz	Horizontal	vert	Vertical
info	Information	init	Initialize
max	Maximum		

常见的缩写：

ASCII，PDF，XML，HTML，URL，RTF，HTTP，TIFF
JPG，GIF，LZW，ROM，RGB，CMYK，MIDI，FTP

> 框架开发者小贴士与技巧

>> Initialize 初始化

>>> 类初始化

在 initialize 类方法中，能够编写实现一些延迟执行且只被一次的代码，initialize 类方法是由运行时系统在该类响应任何其他消息之前调用的。典型的应用是在其中设置类的版本信息。运行时系统向每个类发送 initialize 消息，即使该类没有实现 initialize，也会调用其基类的某个 initialize 方法。因此一个类的 initialize 方法可能会因为存在继承类的缘故被执行多次。因此有必要使用一定的技巧来防止只执行一次的代码被多次执行。如：NSFoo 类的 initialize 方法实现可能如下：

```
+ (id) initialize
{
    if (self == [NSFoo class])
    {
        // 初始化代码
    }
    return self;
}
```

不应当显式调用 initialize 方法。如果你需要激活 initialize 方法，使用 [NSFoo self] 形式的调用。

Ref：Coding Guidelines for Cocoa

后记