Hello, iOS
5 Objective-C语言特性[1]

上一节我们了解了一个由Objective-C编写的程序的基本结构，这一节我们将深入地了解这一语言的特性。请确保阅读本节前已经对面向对象的知识有比较深入的了解，这对于理解本节介绍的概念起到很大作用。

1.协议(protocol)

在NeXT时期，Objective-C曾经希望引入多重继承[2]的概念。由于多重继承可能会产生C++中经常出现的“菱形继承”[3]这种逻辑上异常复杂的情形(而且这种特性往往被程序员用错)，因而出现了协议[4]。既然是为了弥补多重继承的空缺，同时又希望健全其逻辑性而存在的特性，所以协议可以理解为是**多个类共同分享的方法的列表**。

一开始，遵守协议的类需要实现这个协议的所有方法。在Objective-C 2.0中，可以使用@optional标记可选方法，即可选方法不必须实现。由此一来，**对于遵守协议的类，需要实现除了可选方法外的所有方法**。

```
17 @protocol NSCopying
18
19 - (id)copyWithZone:(nullable NSZone *)zone;
20
21 @end
```

这里我们可以参考NSObject.h中对NSCopying协议的定义，学习它的写法。协议以@protocol开头，以@end结束，中间定义了一个方法。

如果我希望遵守某个协议[5]，就需要在自己的父类后用尖括号包住遵守的协议，如果有多个则用逗号连接。

```
11 @interface TestClass : NSObject <NSCopying, NSCoding>
```

除了刚才我们提到的@optional表示可选方法外，还有@required表示必须实现的方法。我们认为协议类是一个公共的接口，规定了多个类之间的接口。

---

[1] 2015年9月3日，基于OS X 10.11 beta 8， iOS 9 beta 5， Xcode 7 beta 5.

[2] 多重继承(multiple inheritance)指一个子类可以有多个父类的情况。相对应的是单一继承(single inheritance)，即一个子类只能有一个父类。C++属于前者，而我们目前介绍的Objective-C和常见于Android开发中的Java都属于后者。

[3] 我们试举一例解释“菱形继承”的情形。例如存在类A，同时为类B和类C的父类。另外还有一个类D，其父类同时是类B和类C。那么对于类D，其重载方法应该继承类B还是类C的呢？一般在应用上我们会根据实际情况具体分析，但是这对于程序员的逻辑要求很高，同时代码维护性也较低。

[4] 对于有Java开发经验的开发者，协议这种特性与Java中的接口(interface)大致相似。

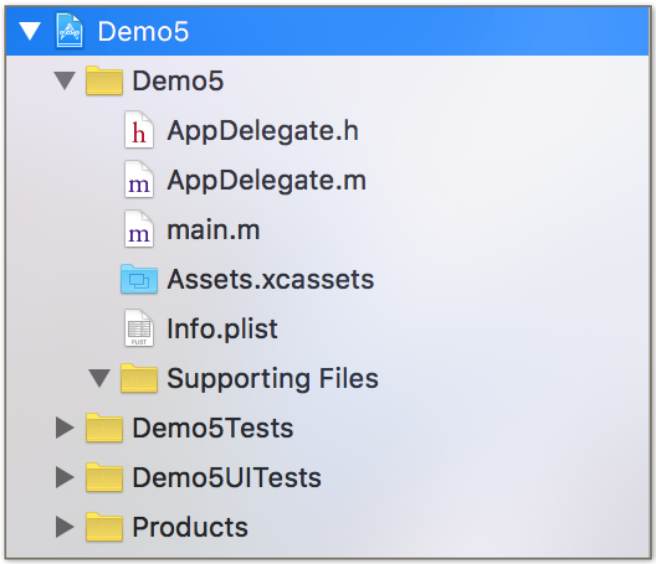
[5] 如果想要检查一个对象是否遵守某一协议，可以写如下代码：

```
if ([someObject conformsToProtocol:@protocol(SomeProtocol)] == YES) {
    //Codes
}
```

2.委托(delegate)

刚才我们了解的协议中，定义协议的类可以把协议定义的方法委托给实现它的类，这样可以使得协议具有更好的通用性。我们举一个例子，iOS中的“设置”采取了UITableViewController，即表格视图控制器。对于一个表格的试图UITabelView而言，它的对象定制个性化的内容是由UITableViewDataSource这个协议的对象提供的。也就是说表格对象把数据处理等委托给实现了UITableViewDataSource协议的对象处理。同时每用户对表格进行交互，例如点击、选择某一行等操作的事件处理都是由实现了UITableViewDelegate的对象处理。广泛地，一个应用程序的加载过程全程都是委托给相应代理对象来处理的。

啰嗦了这么多，主要的意思是，委托确定了特定的对象帮我做事情，以此来保证我作为任务的分发者可以减轻压力。



我写了一个Mac程序展示代理在程序运行过程当中发挥的作用。可以看到文件中的AppDelegate就是整个程序的代理，main是主程序。

```objc
#import <Cocoa/Cocoa.h>
#import "AppDelegate.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        AppDelegate *delegate = [[AppDelegate alloc] init];
        //创建AppDelegate的对象
        [NSApplication sharedApplication];
        //获取NSApplication的单例对象
        [NSApp setDelegate:delegate];
        //调用代理设置方法设置Cocoa应用代理，然后把事件委托给delegate处理
        return NSApplicationMain(argc, (const char **)argv);
        //开始运行程序
    }
}

```

main.h

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

@interface AppDelegate : NSObject <NSApplicationDelegate>

@property (strong) NSWindow *window;

@end

```

AppDelegate.h

```objc
#import "AppDelegate.h"

@implementation AppDelegate

@synthesize window;

- (void)applicationWillFinishLaunching:(NSNotification *)notification {
    self.window = [[NSWindow alloc] initWithContentRect:NSMakeRect(300, 300, 320, 200) styleMask:
        (NSTitledWindowMask | NSMiniaturizableWindowMask | NSClosableWindowMask) backing:
        NSBackingStoreBuffered defer:NO];
    self.window.title = @"委托测试程序";
    NSTextField *tf = [[NSTextField alloc] initWithFrame:NSMakeRect(60, 120, 200, 60)];
    [tf setSelectable:YES];
    [tf setBezeled:YES];
    [tf setDrawsBackground:YES];
    [tf setStringValue:@"Delegate Test"];
    NSButton *button = [[NSButton alloc] initWithFrame:NSMakeRect(120, 40, 80, 30)];
    button.title = @"OK";
    [button setBezelStyle:NSRoundedBezelStyle];
    [button setBounds:NSMakeRect(120, 40, 80, 30)];
    [self.window.contentView addSubview:tf];
    [self.window.contentView addSubview:button];
}

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self.window makeKeyAndOrderFront:self]; //把窗口显示到应用程序的前台
    // Insert code here to initialize your application
}

- (void)applicationWillTerminate:(NSNotification *)aNotification {
    // Insert code here to tear down your application
}

@end
```

其中AppDelegate.m文件比较多，需要解释的内容也都在这里面。注意第一个方法的名字，应用即将结束启动，这也就意味着是程序展现在你面前之前执行的事情。第二个方法的名字，应用已经结束启动，这时我们把这个窗口推到前台，也就是让用户看到。运行结果如左图。[6]

委托这个特性以后将会大量用到，它保证了协议的通用性，同时还让对象的使用更加灵活。比如iOS开发中让人头疼的输入框和输入法掉落问题，利用委托则可以找到一个比较经济的解决方案，会在将来为大家介绍。

[6] Mac下的开发并不是我们研究的重心，所以你不必要仔细研究上面的代码。当你对iOS的开发比较熟悉后，Mac的开发是大同小异的。

3.键值编码(KVC)[7]

键值编码，全称是Key Value Coding，简称KVC，它允许利用字符串操作对应变量或者属性的值。

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Student *stu = [[Student alloc] init];
        [stu setValue:@"李狗蛋" forKey:@"name"];
        [stu setValue:[[NSDate alloc] init] forKey:@"birth"];
        NSLog(@"学生姓名：%@", [stu valueForKey:@"name"]);
        NSLog(@"学生生日：%@", [stu valueForKey:@"bitrh"]);
    }
    return 0;
}
```

例如我们已经为学生类设计了姓名和生日两个属性，下面我们用KVC对其进行操作。

```
2015-09-03 07:36:52.263 Demo[2012:925813] 学生姓名：李狗蛋
2015-09-03 07:36:52.268 Demo[2012:925813] 学生生日：2015-09-02 23:36:52 +0000
Program ended with exit code: 0
```

得到这样的运行结果，证明了用”setValue: forKey:”和”valueForKey:”两个方法可执行setter和getter的功能。在这个样例中，属性的名称是键(key)，属性的值是值(value)，完成了一系列的键值操作。但是，如果我们操作了不存在的键会发生什么呢?

```
ueForKey:@"birth1"]);                                  Thread 1: signal SIGABRT
〉Thread 1〉12 main
2015-09-03 07:39:58.857 Demo[2020:932098] 学生姓名：李狗蛋
2015-09-03 07:39:58.859 Demo[2020:932098] *** Terminating app due to uncaught exception 'NSUnknownKeyException', reason: '[<Student 0x1001082d0> valueForUndefinedKey:]: this class is not key value coding-compliant for the key birth1.'
*** First throw call stack:
(
    0   CoreFoundation                      0x00007fff8df34bd2 __exceptionPreprocess + 178
    1   libobjc.A.dylib                     0x00007fff95c904fa objc_exception_throw + 48
    2   CoreFoundation                      0x00007fff8df34b19 - [NSException raise] + 9
    3   Foundation                          0x00007fff866c0f93 - [NSObject(NSKeyValueCoding) valueForUndefinedKey:] + 226
    4   Foundation                          0x00007fff8659a897 - [NSObject(NSKeyValueCoding) valueForKey:] + 414
    5   Demo                                0x0000000100000c6c main + 316
    6   libdyld.dylib                       0x00007fff907305ad start + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSException
(lldb)
```

比如我们把birth改成了birth1，这是一个不存在的键，这时系统会抛出一个NSUnknownKeyException异常[8]，直译为未知key的异常。

---

[7] 关于KVC的更多信息，可以参考我的技术博客。文章地址：http://zhrmoe.iflab.org/2015/04/07/key-value-coding-fundamentals/

[8] 关于异常处理，我们将在后面讲到。

4.键值监听(KVO)

键值监听，全称是Key Value Observing，简称KVO，它可以在某个键对应的值发生改变的时候向消息中心[9]发送消息。

这里，我设计了一个类物品类Item和另一个审查物品类ItemView，物品有名称和价格两个属性，并在ItemView中注册了两个观察者。

```objc
#import "ItemView.h"

@implementation ItemView

@synthesize item = _item;

- (void)showItemInfo {
    NSLog(@"物品名：%@，价格为：%d", self.item.name, self.item.price);
}

- (void)setItem:(Item *)item {
    self->_item = item;
    [self.item addObserver:self forKeyPath:@"name" options:NSKeyValueObservingOptionNew context:nil];
    [self.item addObserver:self forKeyPath:@"price" options:NSKeyValueObservingOptionNew context:nil];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary<NSString *,id> *)change
                       context:(void *)context {
    NSLog(@"监听方法被调用");
    NSLog(@"修改的keyPath是%@",keyPath);
    NSLog(@"修改的对象是%@", object);
    NSLog(@"新的属性值是%@", [change objectForKey:@"new"]);
}

- (void)dealloc {
    [self.item removeObserver:self forKeyPath:@"name"];
    [self.item removeObserver:self forKeyPath:@"price"];
}

@end
```

这里我们看到可以使用”addObserver: forKeyPath: options: context:”方法注册一个观察者。析构[10]时，用”removeObserver: forKeyPath:”方法注销它。每次键值监听的功能发生作用时，这个名字非常长的方法也将被调用，在本程序中会打出4个log。

```objc
#import <Foundation/Foundation.h>
#import "Item.h"
#import "ItemView.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Item *item = [[Item alloc] init];
        item.name = @"李狗蛋的书包";
        item.price = 450;
        ItemView *itemView = [[ItemView alloc] init];
        itemView.item = item;
        [itemView showItemInfo];
        item.name = @"李狗蛋的笔袋";
        item.price = 25;
    }
    return 0;
}
```

这时我们在main.m中实例化Item和ItemView的对象。然后我们对item的属性做一些改变。

[9] 这个消息中心并不是从屏幕顶端下拉后看到的通知中心，而是系统层级上处理应用的各种动态的消息中心——NSNotificationCenter.

[10] 是与构造相反的行为，一般不特别地写代码，但是析构时有额外操作时需要给出代码。

```
2015-09-03 08:02:33.192 Demo[2061:972422] 物品名：李狗蛋的书包，价格为：450
2015-09-03 08:02:33.193 Demo[2061:972422] 监听方法被调用
2015-09-03 08:02:33.193 Demo[2061:972422] 修改的keyPath是name
2015-09-03 08:02:33.193 Demo[2061:972422] 修改的对象是<Item: 0x100100700>
2015-09-03 08:02:33.193 Demo[2061:972422] 新的属性值是李狗蛋的笔袋
2015-09-03 08:02:33.193 Demo[2061:972422] 监听方法被调用
2015-09-03 08:02:33.193 Demo[2061:972422] 修改的keyPath是price
2015-09-03 08:02:33.193 Demo[2061:972422] 修改的对象是<Item: 0x100100700>
2015-09-03 08:02:33.193 Demo[2061:972422] 新的属性值是25
Program ended with exit code: 0
```

结果如上图所示，监听的方法被成功调用了。这说明，监听器方法可以得到更新的数据，所以我们把这些数据体现在与用户的交互上，就完成了我们的需求。

本节我们介绍了协议、委托、KVC和KVO四个特性，理解难度比较高。如果阅读有困难，可以多读重点的段落，代码也可以自己实践体会。下一节我将简单介绍苹果在2014年WWDC上介绍的新语言Swift的一些基础知识。

第5节 EOF