iOS开发·runtime原理与实践: 关联对象篇(Associated Object)(应用场景：为分类添加“属性”，为UI控件关联事件Block体)

1. 前言

1.1 关联对象

分类(category)与关联对象(Associated Object)作为objective-c的扩展机制的两个特性：分类，可以通过它来扩展方法；Associated Object，可以通过它来扩展属性；

在iOS开发中，可能category比较常见，相对的Associated Object，就用的比较少，要用它之前，必须导入<objc/runtime.h>的头文件。

1.2 如何关联对象

runtime提供了給我们3个API以管理关联对象（存储、获取、移除）：

```
//关联对象
void objc_setAssociatedObject(id object, const void *key, id value,
objc_AssociationPolicy policy)
//获取关联的对象
id objc_getAssociatedObject(id object, const void *key)
//移除关联的对象
void objc_removeAssociatedObjects(id object)
```

其中的参数:

- id object：被关联的对象

- const void *key：关联的key，要求唯一
- id value：关联的对象
- objc_AssociationPolicy policy ： 内存管理的策略

2. 关联对象：为分类添加“属性”

2.1 分类的限制

先来看 @property 的一个例子

```
@interface Person : NSObject

@property (nonatomic, strong) NSString *name;

@end
```

在使用上述 @property 时会做三件事:

- 生成实例变量 _property
- 生成 getter 方法 - property
- 生成 setter 方法 - setProperty :

```
@implementation DKObject {
    NSString *_property;
}

- (NSString *)property {
    return _property;
}

- (void)setProperty:(NSString *)property {
    _property = property;
}

@end
```

这些代码都是编译器为我们生成的，虽然你看不到它，但是它确实在这里。但是，如果我们在分类中写一个属性，则会给一个警告，分类中的 @property 并没有为我们生成实例变量以及存取方法，而需要我们手动实现。

因为在分类中 @property 并不会自动生成实例变量以及存取方法，所以一般使用关联对象为已经存在的类添加“属性”。解决方案：可以使用两个方法 objc_getAssociatedObject 以及 objc_setAssociatedObject 来模拟属性 的存取方法，而使用关联对象模拟实例变量。

2.2 实用举例：

- NSObject+AssociatedObject.m

```
#import "NSObject+AssociatedObject.h"
#import <objc/runtime.h>

@implementation NSObject (AssociatedObject)

- (void)setAssociatedObject:(id)associatedObject
{
    objc_setAssociatedObject(self, @selector(associatedObject),
associatedObject, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (id)associatedObject
{
    return objc_getAssociatedObject(self, _cmd);
}

@end
```

- ViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];

    NSObject *objc = [[NSObject alloc] init];
    objc.associatedObject = @"Extend Category";

    NSLog(@"associatedObject is = %@", objc.associatedObject);
}
```

其中， _cmd 代指当前方法的选择子，也就是 @selector(categoryProperty) 。

_cmd 在Objective-C的方法中表示当前方法的selector，正如同self表示当前方法调用的对象实例。这里强调当前， _cmd 的作用域只在当前方法里，直指当前方法名@selector。 因而，亦可以写成下面的样子：

```
- (id)associatedObject
{
    return objc_getAssociatedObject(self, @selector(associatedObject));
}
```

另外，查看 OBJC_ASSOCIATION_RETAIN_NONATOMIC ，可以发现它是一个枚举类型，完整枚举项如下所示：

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,           /**< Specifies a weak reference to
the associated object. */
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /**< Specifies a strong reference
to the associated object.
                                            *   The association is not made
atomically. */
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /**< Specifies that the associated
object is copied.
                                            *   The association is not made
atomically. */
    OBJC_ASSOCIATION_RETAIN = 01401,       /**< Specifies a strong reference
to the associated object.
                                            *   The association is made
atomically. */
    OBJC_ASSOCIATION_COPY = 01403          /**< Specifies that the associated
object is copied.
                                            *   The association is made
atomically. */
};
```

从这里的注释我们能看到很多东西，也就是说不同的 objc_AssociationPolicy 对应了不通的属性修饰符，整理成表格如下：

objc_AssociationPolicy	modifier
OBJC_ASSOCIATION_ASSIGN	assign
OBJC_ASSOCIATION_RETAIN_NONATOMIC	nonatomic, strong
OBJC_ASSOCIATION_COPY_NONATOMIC	nonatomic, copy
OBJC_ASSOCIATION_RETAIN	atomic, strong
OBJC_ASSOCIATION_COPY	atomic, copy

而我们在代码中实现的属性 associatedObject 就相当于使用了 nonatomic 和 strong 修饰符。

3. 关联对象：为UI控件关联事件Block体

3.1 UIAlertView

开发iOS时经常用到UIAlertView类，该类提供了一种标准视图，可向用户展示警告信息。当用户按下按钮关闭该视图时，需要用委托协议（delegate protocol）来处理此动作，但是，要想设置好这个委托机制，就得把创建警告视图和处理按钮动作的代码分开。由于代码分作两块，所以读起来有点乱。

方案1：传统方案

比方说，我们在使用UIAlertView时，一般都会这么写：

- Test2ViewController

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self.view setBackgroundColor:[UIColor whiteColor]];
    self.title = @"Test2ViewController";

    [self popAlertViews1];
}

#pragma mark - way1
- (void)popAlertViews1{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Question"
message:@"What do you want to do?" delegate:self cancelButtonTitle:@"Cancel"
otherButtonTitles:@"Continue", nil];
    [alert show];
}

// UIAlertViewDelegate protocol method
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex
{
    if (buttonIndex == 0) {
        [self doCancel];
    } else {
        [self doContinue];
    }
}
```

如果想在同一个类里处理多个警告信息视图，那么代码就会变得更为复杂，我们必须在delegate方法中检查传入的alertView参数，并据此选用相应的逻辑。 要是能在创建UIAlertView的时候直接把处理每个按钮的逻辑都写好，那就简单多了。这可以通过关联对象来做。创建完警告视图之后，设定一个与之关联的“块”（block），等到执行delegate方法时再将其读出来。下面对此方案进行改进。

方案2：关联Block体

除了上一个方案中的传统方法，我们可以利用关联对象为UIAlertView关联一个Block：首先在创建UIAlertView的时候设置关联一个回调 （objc_setAssociatedObject） ，然后在UIAlertView的代理方法中取出关联相应回调 （objc_getAssociatedObject ）。

- Test2ViewController.m

```
#pragma mark - way2
```

```
- (void)popAlertViews2 {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Question"
message:@"What do you want to do?" delegate:self cancelButtonTitle:@"Cancel"
otherButtonTitles:@"Continue", nil];
    void (^clickBlock)(NSInteger) = ^(NSInteger buttonIndex){
        if (buttonIndex == 0) {
            [self doCancel];
        } else {
            [self doContinue];
        }
    };

objc_setAssociatedObject(alert,CMAlertViewKey,clickBlock,OBJC_ASSOCIATION_COPY);
    [alert show];
}

// UIAlertViewDelegate protocol method
- (void)alertView:(UIAlertView*)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex{

    void (^clickBlock)(NSInteger) = objc_getAssociatedObject(alertView,
CMAlertViewKey);
    clickBlock(buttonIndex);
}
```

方案3：继续改进：封装关联的Block体，作为属性

上面方案，如果需要的位置比较多，相同的代码会比较冗余地出现，所以我们可以将设置Block的代码封装到一个UIAlertView的分类中去。

- UIAlertView+Handle.h

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

// 声明一个button点击事件的回调block
typedef void (^ClickBlock)(NSInteger buttonIndex) ;

@interface UIAlertView (Handle)

@property (copy, nonatomic) ClickBlock callBlock;

@end
```

- UIAlertView+Handle.m

```
#import "UIAlertView+Handle.h"
```

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

@implementation UIAlertView (Handle)

- (void)setCallBlock:(ClickBlock)callBlock
{
    objc_setAssociatedObject(self, @selector(callBlock), callBlock,
OBJC_ASSOCIATION_COPY_NONATOMIC);
}

- (ClickBlock )callBlock
{
    return objc_getAssociatedObject(self, _cmd);
    //    return objc_getAssociatedObject(self, @selector(callBlock));
}

@end
```

- Test2ViewController.m

```
#pragma mark - way3
- (void)popAlertViews3 {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Question"
message:@"What do you want to do?" delegate:self cancelButtonTitle:@"Cancel"
otherButtonTitles:@"Continue", nil];
    [alert setCallBlock:^(NSInteger buttonIndex) {
        if (buttonIndex == 0) {
            [self doCancel];
        } else {
            [self doContinue];
        }
    }];

    [alert show];
}

// UIAlertViewDelegate protocol method
- (void)alertView:(UIAlertView*)alertView clickedButtonAtIndex:
(NSInteger)buttonIndex{

    void (^block)(NSInteger) = alertView.callBlock;
    block(buttonIndex);
}
```

方案4：继续改进：封装关联的Block体，跟初始化方法绑在一起

> 练习：可以对这个分类进一步改进，将设置Block属性的方法与初始化方法写在一起。

3.2 UIButton

除了上述的UIAlertView，这节以UIButton为例，使用关联对象完成一个功能函数：为UIButton增加一个分类，定义一个方法，使用block去实现button的点击回调。

- UIButton+Handle.h

```
#import <UIKit/UIKit.h>
#import <objc/runtime.h>    // 导入头文件

// 声明一个button点击事件的回调block
typedef void(^ButtonClickCallBack)(UIButton *button);

@interface UIButton (Handle)

// 为UIButton增加的回调方法
- (void)handleClickCallBack:(ButtonClickCallBack)callBack;

@end
```

- UIButton+Handle.m

```
#import "UIButton+Handle.h"

// 声明一个静态的索引key，用于获取被关联对象的值
static char *buttonClickKey;

@implementation UIButton (Handle)

- (void)handleClickCallBack:(ButtonClickCallBack)callBack {
    // 将button的实例与回调的block通过索引key进行关联：
    objc_setAssociatedObject(self, &buttonClickKey, callBack,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);

    // 设置button执行的方法
    [self addTarget:self action:@selector(buttonClicked)
forControlEvents:UIControlEventTouchUpInside];
}

- (void)buttonClicked {
    // 通过静态的索引key，获取被关联对象（这里就是回调的block）
    ButtonClickCallBack callBack = objc_getAssociatedObject(self,
&buttonClickKey);

    if (callBack) {
        callBack(self);
    }
}
```

```
@end
```

在Test3ViewController中，导入我们写好的UIButton分类头文件，定义一个button对象，调用分类中的这个方法：

- Test3ViewController.m

```
    [self.testButton handleClickCallBack:^(UIButton *button) {
        NSLog(@"block --- click UIButton+Handle");
    }];
```

4. 关联对象：关联观察者对象

有时候我们在分类中使用NSNotificationCenter或者KVO，推荐使用关联的对象作为观察者，尽量避免对象观察自身。

例如AFNetworking里面的为菊花控件监听NSURLSessionTask以获取网络进度的分类：

- UIActivityIndicatorView+AFNetworking.m

```
@implementation UIActivityIndicatorView (AFNetworking)

- (AFActivityIndicatorViewNotificationObserver *)af_notificationObserver {

    AFActivityIndicatorViewNotificationObserver *notificationObserver =
objc_getAssociatedObject(self, @selector(af_notificationObserver));
    if (notificationObserver == nil) {
        notificationObserver = [[AFActivityIndicatorViewNotificationObserver
alloc] initWithActivityIndicatorView:self];
        objc_setAssociatedObject(self, @selector(af_notificationObserver),
notificationObserver, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    }
    return notificationObserver;
}

- (void)setAnimatingWithStateOfTask:(NSURLSessionTask *)task {
    [[self af_notificationObserver] setAnimatingWithStateOfTask:task];
}

@end
```

```
@implementation AFActivityIndicatorViewNotificationObserver

- (void)setAnimatingWithStateOfTask:(NSURLSessionTask *)task {
    NSNotificationCenter *notificationCenter = [NSNotificationCenter
defaultCenter];
```

```
    [notificationCenter removeObserver:self
name:AFNetworkingTaskDidResumeNotification object:nil];
    [notificationCenter removeObserver:self
name:AFNetworkingTaskDidSuspendNotification object:nil];
    [notificationCenter removeObserver:self
name:AFNetworkingTaskDidCompleteNotification object:nil];

    if (task) {
        if (task.state != NSURLSessionTaskStateCompleted) {
            UIActivityIndicatorView *activityIndicatorView =
self.activityIndicatorView;
            if (task.state == NSURLSessionTaskStateRunning) {
                [activityIndicatorView startAnimating];
            } else {
                [activityIndicatorView stopAnimating];
            }

            [notificationCenter addObserver:self
selector:@selector(af_startAnimating)
name:AFNetworkingTaskDidResumeNotification object:task];
            [notificationCenter addObserver:self
selector:@selector(af_stopAnimating)
name:AFNetworkingTaskDidCompleteNotification object:task];
            [notificationCenter addObserver:self
selector:@selector(af_stopAnimating)
name:AFNetworkingTaskDidSuspendNotification object:task];
        }
    }
}
```