iOS开发·KVO用法，原理与底层实现: runtime模拟实现KVO监听机制（Blcok及Delgate方式）

本文Demo传送门：CMKVODemo

> **摘要**：这篇文章首先介绍KVO的基本用法，接着探究 KVO (Key-Value Observing) 实现机制，并利用 runtime 模拟实现 KVO的监听机制：一种Block方式回调，一种Delegate回调。同时，本文也会总结KVO实现过程中与 runtime 相关的API用法。

1. KVO理论基础

1.1 KVO的基本用法

> 步骤

❶ 注册观察者，实施监听

```
[self.person addObserver:self
              forKeyPath:@"age"
                 options:NSKeyValueObservingOptionNew
                 context:nil];
```

❷ 回调方法，在这里处理属性发生的变化

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary<NSString *,id>
*)change
                       context:(void *)context {
  //...实现监听处理
}
```

❸ 移除观察者

```
[self removeObserver:self forKeyPath:@“age"];
```

> 综合例子

```
//添加观察者
_person = [[Person alloc] init];
[_person addObserver:self
          forKeyPath:@"age"
             options:NSKeyValueObservingOptionNew |
NSKeyValueObservingOptionOld
             context:nil];
```

```
//KVO回调方法
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary<NSString *,id>
*)change
                       context:(void *)context
{
    NSLog(@"%@对象的%@属性改变了，change字典为:
%@",object,keyPath,change);
    NSLog(@"属性新值为：%@",change[NSKeyValueChangeNewKey]);
    NSLog(@"属性旧值为：%@",change[NSKeyValueChangeOldKey]);
}
```

```
//移除观察者
- (void)dealloc
{
    [self.person removeObserver:self forKeyPath:@"age"];
}
```

> 利用了KVO实现键值监听的**第三方框架**

AFNetworking，MJRresh

1.2 KVO的实现原理

KVO 是 Objective-C 对 **观察者模式**（Observer Pattern）的实现。当被观察对象的某个属性发生更改时，观察者对象会获得通知。有意思的是，你不需要给被观察的对象添加任何额外代码，就能使用 KVO 。这是怎么做到的?

KVO 的实现也依赖于 Objective-C 强大的 Runtime 。Apple 的文档有简单提到过 KVO 的实现。Apple 的文档唯一有用的信息是：**被观察对象的 isa 指针会指向一个中间类，而不是原来真正的类**。Apple 并不希望过多暴露 KVO 的实现细节。

不过，要是你用 runtime 提供的方法去深入挖掘，所有被掩盖的细节都会原形毕露。Mike Ash 早在 2009 年就做了这么个探究，了解更多 点这里。

> KVO 的实现：

当你观察一个对象时，一个**新的类**会动态被创建。这个类**继承**自该对象的原本的类，并**重写**了被观察属性的 `setter` 方法。自然，重写的 `setter` 方法会负责在调用原 `setter` 方法之前和之后，通知所有观察对象值的更改。最后把这个对象的 `isa` 指针 (isa 指针告诉 Runtime 系统这个对象的类是什么) 指向这个新创建的**子类**，对象就神奇的变成了新创建的**子类**的实例。

这个**中间类**，继承自**原本的那个类**。不仅如此，Apple 还重写了 `-class` 方法，企图欺骗我们这个类没有变，就是原本那个类。更具体的信息，去跑一下 Mike Ash 的那篇文章里的代码就能明白，这里就不再重复。

1.3 KVO的不足

KVO 很强大，没错。知道它内部实现，或许能帮助更好地使用它，或在它出错时更方便调试。但官方实现的 KVO 提供的 API 实在不怎么样。

比如，你只能通过重写 `-observeValueForKeyPath:ofObject:change:context:` 方法来获得通知。想要提供自定义的 selector，不行；想要传一个 block，门都没有。而且你还要处理父类的情况 - 父类同样监听同一个对象的同一个属性。但有时候，你不知道父类是不是对这个消息有兴趣。虽然 context 这个参数就是干这个的，也可以解决这个问题 - 在 `-addObserver:forKeyPath:options:context:` 传进去一个父类不知道的 context。但总觉得框在这个 API 的设计下，代码写的很别扭。至少至少，也应该支持 block 吧。

有不少人都觉得官方 KVO 不好使的。Mike Ash 的 Key-Value Observing Done Right，以及获得不少分享讨论的 KVO Considered Harmful 都把 KVO 拿出来吊打了一番。所以在实际开发中 KVO 使用的情景并不多，更多时候还是用 Delegate 或 NotificationCenter。

2. Block实现KVO

2.1 模拟实现

> 注意：以下都是同一个文件：NSObject+Block_KVO.m中写的

- 导入头文件，并定义两个静态变量

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

//as prefix string of kvo class
static NSString * const kCMkvoClassPrefix_for_Block =
@"CMObserver_";
static NSString * const kCMkvoAssiociateObserver_for_Block =
@"CMAssiociateObserver";
```

- 暴露给调用者为被观察对象添加KVO方法

```
- (void)CM_addObserver:(NSObject *)observer forKey:(NSString *)key
withBlock:(CM_ObservingHandler)observedHandler
{
    //step 1 get setter method, if not, throw exception
    SEL setterSelector =
NSSelectorFromString(setterForGetter(key));
```

```
    Method setterMethod = class_getInstanceMethod([self class],
setterSelector);
    if (!setterMethod) {
        @throw [NSException exceptionWithName:
NSInvalidArgumentException reason: [NSString stringWithFormat:
@"unrecognized selector sent to instance %@", self] userInfo: nil];
        return;
    }

    //自己的类作为被观察者类
    Class observedClass = object_getClass(self);
    NSString * className = NSStringFromClass(observedClass);

    //如果被监听者没有CMObserver_，那么判断是否需要创建新类
    if (![className hasPrefix: kCMkvoClassPrefix_for_Block]) {
        //【代码①】
        observedClass = [self createKVOClassWithOriginalClassName:
className];
        //【API注解①】
        object_setClass(self, observedClass);
    }

    //add kvo setter method if its class(or superclass)hasn't
implement setter
    if (![self hasSelector: setterSelector]) {
        const char * types = method_getTypeEncoding(setterMethod);
        //【代码②】
        class_addMethod(observedClass, setterSelector,
(IMP)KVO_setter, types);
    }

    //add this observation info to saved new observer
    //【代码③】
    CM_ObserverInfo_for_Block * newInfo =
[[CM_ObserverInfo_for_Block alloc] initWithObserver: observer
forKey: key observeHandler: observedHandler];

    //【代码④】【API注解③】
    NSMutableArray * observers = objc_getAssociatedObject(self,
(__bridge void *)kCMkvoAssiociateObserver_for_Block);

    if (!observers) {
        observers = [NSMutableArray array];
        objc_setAssociatedObject(self, (__bridge void
*)kCMkvoAssiociateObserver_for_Block, observers,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    }
    [observers addObject: newInfo];
```

```
}
```

- 其中【代码①】的意思是，被观察的类如果是被观察对象本来的类，那么，就要专门依据本来的类新建一个新的**子类**，区分是否这个子类的标记是带有`kCMkvoClassPrefix_for_Block`的前缀。怎样新建一个**子类**？代码如下所示：

```
- (Class)createKVOClassWithOriginalClassName: (NSString *)className
{
    NSString * kvoClassName = [kCMkvoClassPrefix
stringByAppendingString: className];
    Class observedClass = NSClassFromString(kvoClassName);

    if (observedClass) { return observedClass; }

    //创建新类，并且添加CMObserver_为类名新前缀
    Class originalClass = object_getClass(self);
    //【API注解②】
    Class kvoClass = objc_allocateClassPair(originalClass,
kvoClassName.UTF8String, 0);

    //获取监听对象的class方法实现代码，然后替换新建类的class实现
    Method classMethod = class_getInstanceMethod(originalClass,
@selector(class));
    const char * types = method_getTypeEncoding(classMethod);
    class_addMethod(kvoClass, @selector(class), (IMP)kvo_Class,
types);
    objc_registerClassPair(kvoClass);
    return kvoClass;
}
```

- 另外【代码②】的意思是，将原来的setter方法替换一个新的setter方法（这就是runtime的黑魔法，Method Swizzling）。那么新的setter方法又是什么呢？如下所示：

```
#pragma mark -- Override setter and getter Methods
static void KVO_setter(id self, SEL _cmd, id newValue)
{
    NSString * setterName = NSStringFromSelector(_cmd);
    NSString * getterName = getterForSetter(setterName);
    if (!getterName) {
```

```
        @throw [NSException exceptionWithName:
NSInvalidArgumentException reason: [NSString stringWithFormat:
@"unrecognized selector sent to instance %p", self] userInfo: nil];
        return;
    }

    id oldValue = [self valueForKey: getterName];
    struct objc_super superClass = {
        .receiver = self,
        .super_class = class_getSuperclass(object_getClass(self))
    };

    [self willChangeValueForKey: getterName];
    void (*objc_msgSendSuperKVO)(void *, SEL, id) = (void
*)objc_msgSendSuper;
    objc_msgSendSuperKVO(&superClass, _cmd, newValue);
    [self didChangeValueForKey: getterName];

    //获取所有监听回调对象进行回调
    NSMutableArray * observers = objc_getAssociatedObject(self,
(__bridge const void *)kCMkvoAssiociateObserver_for_Block);
    for (CM_ObserverInfo_for_Block * info in observers) {
        if ([info.key isEqualToString: getterName]) {

dispatch_async(dispatch_queue_create(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0), ^{
                info.handler(self, getterName, oldValue, newValue);
            });
        }
    }
}
```

- 【代码③】是新建一个观察者类。这个类的实现写在同一个class，相当于导入一个类：CM_ObserverInfo_for_Block。这个类的作用是观察者，并在初始化的时候负责调用者传过来的Block回调。如下，`self.handler = handler;`即负责回调。

```
@interface CM_ObserverInfo_for_Block : NSObject

@property (nonatomic, weak) NSObject * observer;
@property (nonatomic, copy) NSString * key;
@property (nonatomic, copy) CM_ObservingHandler handler;

@end
```

```
@implementation CM_ObserverInfo_for_Block

- (instancetype)initWithObserver: (NSObject *)observer forKey:
(NSString *)key observeHandler: (CM_ObservingHandler)handler
{
    if (self = [super init]) {

        _observer = observer;
        self.key = key;
        self.handler = handler;
    }
    return self;
}

@end
```

- 【代码④】的作用是，以及已知的“属性名”，类型为NSString的静态变量`kCMkvoAssiociateObserver_for_Block`来获取这个“属性”观察者数组（这个其实并不是真正意义的属性，属于runtime关联对象的知识范畴，可理解成**观察者数组**这样一个属性）。其中，关于`(__bridge void *)`的知识后面会讲到。

> **调用者**：利用上面的API为被观察者添加KVO

- VC调用API

```
#import "NSObject+Block_KVO.h"
//...........

- (void)viewDidLoad {
    [super viewDidLoad];

    ObservedObject * object = [ObservedObject new];
    object.observedNum = @8;

#pragma mark - Observed By Block
    [object CM_addObserver: self forKey: @"observedNum" withBlock:
^(id observedObject, NSString *observedKey, id oldValue, id
newValue) {
        NSLog(@"Value had changed yet with observing Block");
        NSLog(@"oldValue---%@",oldValue);
        NSLog(@"newValue---%@",newValue);
    }];
```

```
    object.observedNum = @10;
}
```

2.2 runtime关键API解析

> 【API注解①】：object_setClass

我们可以在运行时创建新的class，这个特性用得不多，但其实它还是很强大的。你能通过它创建新的子类，并添加新的方法。

但这样的一个子类有什么用呢？别忘了Objective-C的一个关键点：object内部有一个叫做isa的变量指向它的class。这个变量可以被改变，而不需要重新创建。然后就可以添加新的ivar和方法了。可以通过以下命令来修改一个object的class

```
object_setClass(myObject, [MySubclass class]);
```

这可以用在Key Value Observing。当你开始observing an object时，Cocoa会创建这个object的class的subclass，然后将这个object的isa指向新创建的subclass。

> 【API注解②】：objc_allocateClassPair

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

- 看起来一切都很简单，运行时创建类只需要三步： 1、为"class pair"分配空间（使用`objc_allocateClassPair`）. 2、为创建的类添加方法和成员（上例使用`class_addMethod`添加了一个方法）。 3、注册你创建的这个类，使其可用（使用`objc_registerClassPair`）。

为什么这里1和3都说到pair，我们知道pair的中文意思是一对，这里也就是一对类，那这一对类是谁呢？他们就是Class、MetaClass。

- 需要配置的参数为： 1、第一个参数：作为新类的超类,或用Nil来创建一个新的根类。 2、第二个参数：新类的名称 3、第三个参数：一般传0

> 【API注解③】：(__bridge void *)

在 ARC 有效时，通过 `(__bridge void *)` 转换 id 和 void * 就能够相互转换。为什么转换？这是因为 `objc_getAssociatedObject` 的参数要求的。先看一下它的API：

```
objc_getAssociatedObject(id _Nonnull object, const void * _Nonnull
key)
```

可以知道，这个“属性名”的key是必须是一个 `void *` 类型的参数。所以需要转换。关于这个转换，下面给一个转换的例子：

```
id obj = [[NSObject alloc] init];

void *p = (__bridge void *)obj;
id o = (__bridge id)p;
```

关于这个转换可以了解更多：ARC 类型转换：显示转换 id 和 void *

当然，如果不通过转换使用这个API，就需要这样使用：

- 方式1：

```
objc_getAssociatedObject(self, @"AddClickedEvent");
```

- 方式2：

```
static const void *registerNibArrayKey = &registerNibArrayKey;
```

```
NSMutableArray *array = objc_getAssociatedObject(self,
registerNibArrayKey);
```

- 方式3：

```
static const char MJErrorKey = '\0';
```

```
objc_getAssociatedObject(self, &MJErrorKey);
```

- 方式4：

```
+ (instancetype)cachedPropertyWithProperty:
(objc_property_t)property
{
    MJProperty *propertyObj = objc_getAssociatedObject(self,
property);
    //省略
}
```

其中`objc_property_t`是runtime的类型

```
typedef struct objc_property *objc_property_t;
```

2.3 runtime其它API解析

剩下的就是runtime的比较常见API了，这里就不按照上面代码的顺序的讲解了。这里只做按runtime的知识范畴将这些API做一个分类：

- runtime：关联对象相关API

```
objc_getAssociatedObject(id _Nonnull object, const void * _Nonnull
key)
objc_setAssociatedObject(id _Nonnull object, const void * _Nonnull
key,
                         id _Nullable value, objc_AssociationPolicy
policy)
```

- runtime：方法替换相关API

```
BOOL class_addMethod(Class cls, SEL name, IMP imp, const char
*types);
object_getClass(id _Nullable obj)
Method class_getInstanceMethod(Class cls, SEL name);
const char * method_getTypeEncoding(Method m);
FOUNDATION_EXPORT SEL NSSelectorFromString(NSString
*aSelectorName);
```

- runtime：消息机制相关API

```
objc_msgSendSuper
```

- KVO

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

3. 拓展：Delegate实现KVO

> 注意：以下都是同一个文件：NSObject+Block_Delegate.m中写的

- 观察类CM_ObserverInfo需要改一个属性，将Block改为一个Delegate。

```
@interface CM_ObserverInfo : NSObject

@property (nonatomic, weak) NSObject * observer;
@property (nonatomic, copy) NSString * key;
//修改这里
@property (nonatomic, assign) id <ObserverDelegate>
observerDelegate;

@end
```

- 同样，观察类CM_ObserverInfo初始化的时候也需要相应初始这个新属性。

```
@implementation CM_ObserverInfo

- (instancetype)initWithObserver: (NSObject *)observer forKey:
(NSString *)key
{
    if (self = [super init]) {

        _observer = observer;
        self.key = key;
        //修改这里
        self.observerDelegate = (id<ObserverDelegate>)observer;
    }
    return self;
}
@end
```

- 暴露给调用者为被观察对象添加KVO方法：不需要传Block了。

```
#pragma mark -- NSObject Category(KVO Reconstruct)
@implementation NSObject (Block_KVO)

- (void)CM_addObserver:(NSObject *)observer forKey:(NSString *)key
withBlock:(CM_ObservingHandler)observedHandler
{
   //...省略
   //add this observation info to saved new observer
   //修改这里
   CM_ObserverInfo * newInfo = [[CM_ObserverInfo alloc]
initWithObserver: observer forKey: key];
   //...省略
}
```

> **调用者**：利用上面的API为被观察者添加KVO

- VC调用API

```
#import "NSObject+Delegate_KVO.h"
//...........
```

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

    ObservedObject * object = [ObservedObject new];
    object.observedNum = @8;

#pragma mark - Observed By Delegate
    [object CM_addObserver: self forKey: @"observedNum"];

    object.observedNum = @10;
}
```

- VC实现代理方法

```
#pragma mark - ObserverDelegate
-(void)CM_ObserveValueForKeyPath:(NSString *)keyPath ofObject:
(id)object oldValue:(id)oldValue newValue:(id)newValue{
    NSLog(@"Value had changed yet with observing Delegate");
    NSLog(@"oldValue---%@",oldValue);
    NSLog(@"newValue---%@",newValue);
}
```