深入理解KVO

By 张不坏

2015-04-29 更新日期:2015-08-16

写在前面

文章目录

程序设计语言中有各种各样的设计模式（pattern）和与此对应的反设计模式（anti-pattern），譬如singleton、factory、observer、MVC等等。对于基于Objective-C的iOS开发而言，有些设计模式几乎已经成为开发环境的一部分，譬如MVC，自打我们设计第一个页面开始就已经开始与之打交道了；KVO，即Key-Value Observing（根据我的理解它属于observer设计模式）也一样，只是它已经成为Objective-C事实标准了，作为一个iOS开发者，必须对它有相当的了解。

之前对KVO的了解仅限于使用层面，没有去想过它是如何实现的，更没有想过它会存在一些坑；甚至在刚接触它时，会尽可能创造机会使用它，譬如监听UITextField的text值的变化；但近几天接触了Objective-C的Runtime相关的知识，从runtime层面了解到了KVO的实现原理（即KVO的「消息转发机制」），也通过阅读各位大神的博客了解到了它的坑。

本文首先分析KVO和Runtime的关系，阐述KVO的实现原理；然后结合大神们的博客整理KVO存在的坑以及避免掉坑的正确使用姿势。

KVO和Runtime

关于KVO，即Key-Value Observing，官方文档《Key-Value Observing Programming Guide》里的介绍比较简短明了：

> Key-value observing is a mechanism that allows objects to be notified of changes to specified properties of other objects.

KVO的实现

KVO的实现也依赖于Objective-C的Runtime，官方文档《Key-Value Observing Programming Guide》中在《Key-Value Observing Implementation Details》部分简单提到它的实现：

> 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.

简单概述下KVO的实现：
当你观察一个对象（称该对象为「被观察对象」）时，一个新的类会动态被创建。这个类继承自「被观察对象」所对应类的，并**重写该被观察属性的setter方法**；针对setter方法的重写无非是在赋值语句前后加上相应的通知（或曰方法调用）；最后，把「被观察对象」的isa指针（isa指针告诉Runtime系统这个对象的类是什么）指向这个新创建的中间类，对象就神奇变成了新创建类的实例。

根据文档的描述，虽然**被观察对象**的isa指针被修改了，**但是调用其class方法得到的类信息仍然是它之前所继承类的类信息，而不是这个新创建类的类信息。**

补充：下面对isa指针和类方法class作以更多的说明。
isa指针和类方法class的返回值都是Class类型，如下：

```
@interface NSObject <NSObject> {
    Class isa  OBJC_ISA_AVAILABILITY;
}

+ (Class)class;
```

根据我的理解，一般情况下，isa指针和class方法返回值都是一样的；但KVO底层实现时，动态创建的类只是重写了被观察属性的setter方法，并未重写类方法 `class` ，因此向**被观察者**发送class消息实际上仍然调用的是**被观察者**原先类的类方法 `+ (Class)class` ，得到的类型信息当然是原先类的类信息，根据我的猜测， `isKindOfClass:` 和 `isMemberOfClass:` 与 `class` 方法紧密相关。

国外的大神Mike Ash早在2009年就做了关于KVO的实现细节的探究，更多详细参考这里。

KVO的槽点

AFNetworking作者Mattt Thompson在《Key-Value Observing》中说：

> Ask anyone who's been around the NSBlock a few times: Key-Value Observing has the worst API in all of Cocoa.

另一位不认识的大神在《KVO Considered Harmful》中也写道：

> KVO, or key-value observing, is a pattern that Cocoa provides for us for subscribing to changes to the properties of other objects. It's hands down the most poorly designed API in all of Cocoa, and even when implemented perfectly, it's still an incredibly dangerous tool to use, reserved only for when no other technique will suffice.

总之，两位大神都认为KVO的API非常差劲！

其中《KVO Considered Harmful》中对KVO的槽点有了比较详细的阐述，这一部分内容就取材于此。

为了更好说明这些槽点，假设一个应用场景：ZWTableViewController继承自UITableViewController，它现在需要做一件事情，即监测自己的tableView的contentSize，现采用典型的方式（即KVO）处理这么个需求。

所有的observe处理都放在一个方法里

对于上述示例「监听self.tableView的contentSize变化」，最基本处理方式是：

```
// register observer
- (void)viewDidLoad {
    [super viewDidLoad];
    [_tableView addObserver:self forKeyPath:@"contentSize" options:0 context:NULL];
    /* ... */
```

```
}

// 处理observe
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    [self configureView];
}
```

但考虑到observeValueForKeyPath:ofObject:change:context:中可能会很多其他的observe事务，所以observeValueForKeyPath:ofObject:change:context:更好的逻辑是：

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

    if (object == _tableView && [keyPath isEqualToString:@"contentSize"]) {
        [self configureView];
    }
}
```

但如果KVO处理的事情种类多且繁杂，这会造成observeValueForKeyPath:ofObject:change:context:代码特别长，极不优雅。

严重依赖于string

KVO严重依赖string，换句话说，KVO中的keyPath必须是NSString这个事实使得编译器没办法在编译阶段将错误的keyPath给找出来；譬如很容易将「contentSize」写成「contentsize」；

需要自己处理superclass的observe事务

对于Objective-C，很多时候runtime系统都会自动帮助处理superclass的方法。譬如对于dealloc，假设类Father继承自NSObject，而类Son继承自Father，创建一个Son类对象aSon，在aSon被释放的时候，runtime会先调用Son的dealloc实例方法，之后会自动调用Father的dealloc实例方法，而无需在Son的dealloc中显式执行 `[super dealloc];` 。但对于KVO不会这样，所以为了保证父类（父类可能也会自己observe处理嘛）的observe事务也能被处理，上述observeValueForKeyPath:ofObject:change:context:代码得改成这样：

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

    if (object == _tableView && [keyPath isEqualToString:@"contentSize"]) {
        [self configureView];
    } else {
        [super observeValueForKeyPath:keyPath ofObject:object change:change context:cont
}
```

多次相同的removeObserver会导致crash

写过KVO代码的人都知道，对同一个对象执行两次removeObserver操作会导致程序crash。

在同一个文件中执行两次相同的removeObserver属于粗心，比较容易debug出来；但是跨文件执行两次相同的removeObserver就不是那么容易发现了。

我们一般会在dealloc中进行removeObserver操作（这也是Apple所推荐的）。

譬如，假设上述的ZWTableViewController的父类UITableViewController也对tableView的contentSize注册了相同的监听；那么UITableViewController的dealloc中常常会写出如下这样的代码：

```
[_tableView removeObserver:self forKeyPath:@"contentSize" context:NULL];
```

按照一般习惯，ZWTableViewController中的dealloc也会有相同的处理；那么当ZWTableViewController对象被释放时，ZWTableViewController的dealloc和其父类UITableViewController的dealloc都被调用，这样会导致相同的removeObserver被执行两次，自然会导致crash。

《KVO Considered Harmful》中还有很多其他的槽点，《Key-Value Observing Done Right》也描述了一些，这里就不多说了，更多信息还是建议看原文。

不过好在上述的槽点「严重依赖于string」和「多次相同的removeObserver会导致crash」有比较好的解决方案，如下『使用KVO』所述。

使用KVO

订阅

KVO中与订阅相关的API只有一个：

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

对于这四个参数：

- **observer**: The object to register for KVO notifications. The observer must implement the key-value observing method `observeValueForKeyPath:ofObject:change:context:` .
- **keyPath**: The key path, relative to the receiver, of the property to observe. This value must not be nil.
- **options**: A combination of the NSKeyValueObservingOptions values that specifies what is included in observation notifications. For possible values, see `NSKeyValueObservingOptions` .
- **context**: Arbitrary data that is passed to observer in `observeValueForKeyPath:ofObject:change:context:` .

大神们认为这个API丑陋的重要原因是因为后面两个参数：options和context。

下面来对这两个参数进行详细介绍。

options

options可选值是一个 `NSKeyValueObservingOptions` 枚举值，到目前为止，一共包括四个值，在介绍这四个值各自表示的意思之前，先得有一个概念，即KVO响应方法有一个NSDictionary类型参数change（下面『响应』中可以看到），这个字典中会有一个与被监听属性相关的值，譬如被改变之前的值、新值等，NSDictionary中有啥值由『订阅』时的options值决定，options可取值如下：

- NSKeyValueObservingOptionNew: 指示change字典中包含新属性值；
- NSKeyValueObservingOptionOld: 指示change字典中包含旧属性值；
- NSKeyValueObservingOptionInitial: 相对复杂一些，NSKeyValueObserving.h文件中有详细说明，此处略过；
- NSKeyValueObservingOptionPrior: 相对复杂一些，NSKeyValueObserving.h文件中有详细说明，此处略过；

现在细想，options这个参数也忒复杂了，难怪大神们觉得这个API丑陋（不过我等小民之前从未想过这个问题，=_=，没办法，Apple是个大帝国，我只是其中一个跪舔的小屁民）。

不过更糟心的是下面的context参数。

context

options信息量稍大，但其实蛮好理解的，然而对于context，在写这篇博客之前，一直不知道context参数有啥用（也没在意）。

context作用大了去了，在上文『KVO的槽点』提到一个槽点『多次相同的removeObserver会导致crash』。导致『多次调用相同的removeObserver』一个很重要的原因是我们经常在addObserver时为context参数赋值NULL，关于如何使用context参数，下面的『响应』中会提到。

响应

iOS的UI交互（譬如UIButton的一次点击）有一个非常不错的消息转发机制 — Target-Action模型，简单来说，为指定的event指定target和action处理方法。

```
UIButton *button = [UIButton new];
[button addTarget:self action:@selector(buttonDidClicked:) forControlEvents:UIControlEve
```

这种target-action模型逻辑非常清晰。作为对比，KVO的响应处理就非常糟糕了，所有的响应都对应是同一个方法 `- (void)observeValueForKeyPath:ofObject:change:context:` ，其原型如下：

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

除了NSDictionary类型参数change之外，其余几个参数都能在 `–addObserver:forKeyPath:options:context:` 找到对应。

change参数上文已经讲过了，这里不多说了。下面将针对「严重依赖于string」和「多次相同的removeObserver会导致crash」这两个槽点对keyPath和context参数进行阐述。

keyPath

keyPath的类型是NSString，这导致了我们使用了错误的keyPath而不自知，譬如将@”contentSize”错误写成@”contentsize”，一个更好的方法是不直接使用@”xxxoo”，而是积极使用 `NSStringFromSelector(SEL aSelector)` 方法，即改 `@"contentSize"` 为 `NSStringFromSelector(@selector(contentSize))` 。

context

对于context，上文已经提到一种场景：假如父类（设为ClassA）和子类（设为ClassB）都监听了同一个对象肿么办？是ClassB处理呢还是交给父类ClassA的 `observeValueForKeyPath:ofObject:change:context:` 处理呢？更复杂一点，如果子类的子类（设为ClassC）也监听了同一个对象，当ClassB接收到ClassC的 `[super observeValueForKeyPath:keyPath ofObject:object change:change context:context];` 消息时又该如何处理呢？

这么一想，KVO的API还真的是设计非常糟糕。一般来说，比较靠谱的做法是自己的屁股自己擦。ClassB的observe事务在ClassB中处理，怎么知道是自己的事务还是ClassC传上来的事务呢？用context参数判断！

在addObserver时为context参数设置一个独一无二的值即可，在responding处理时对这个context值进行检验。如此就解决了问题，但这需要靠用户（各个层级类的程序员用户）自觉遵守。

取消订阅

和『订阅』以及『响应』不同，『取消订阅』有两个方法：

```
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath context:(voic
```

```
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath;
```

个人觉得应该尽可能使用第一个方法，保持『订阅』-『响应』-『取消订阅』一致性嘛，养成好习惯！

此外，为了避免『取消订阅』时造成的crash，可以把『取消订阅』代码放在@try-@catch语句中，如下是一个比较全面的的KVO使用示例：

```
static void * zwContentSize = &zwContentSize;

- (void)viewDidLoad {

    [super viewDidLoad];

    // 1. subscribe
    [_tableView addObserver:self
                 forKeyPath:NSStringFromSelector(@selector(contentSize))
                    options:NSKeyValueObservingOptionNew
                    context:zwContentSize];
}

// 2. responding
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {
    if (context == zwContentSize) {
        // configure view
    } else {
        [super observeValueForKeyPath:keyPath ofObject:object change:change context:cont
    }
}

- (void)dealloc {
    @try {
        // 3. unsubscribe
        [_tableView removeObserver:self
                        forKeyPath:NSStringFromSelector(@selector(contentSize))
                           context:zwContentSize];
    }
    @catch (NSException *exception) {

    }
}
```

总之，KVO很强大，但也挺坑，使用它要养成好习惯，避免入坑！

参考资料

1. 《Key-Value Observing Programming Guide》；
2. 《如何自己动手实现KVO》；
3. 《KVO Implementation》；
4. 《Creating Classes at Runtime in Objective-C》；
5. 《Key-Value Observing Done Right》；
6. 《KVO Considered Harmful》；
7. 《Key-Value Observing》；
8. 《KVC和KVO》

KVO

iOS

PREVIOUS:
« 关联对象
NEXT:
» Objective-C中的+initialize和+load

分类

归档