> 问题
>
> 1. iOS用什么方式实现对一个对象的KVO？（KVO的本质是什么？）
> 2. 如何手动触发KVO

首先需要了解KVO基本使用，KVO的全称 Key-Value Observing，俗称“键值监听”，可以用于监听某个对象属性值的改变。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Person *p1 = [[Person alloc] init];
    Person *p2 = [[Person alloc] init];
    p1.age = 1;
    p1.age = 2;
    p2.age = 2;
    // self 监听 p1的 age属性
    NSKeyValueObservingOptions options =
NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld;

    [p1 addObserver:self forKeyPath:@"age" options:options
context:nil];
    p1.age = 10;
    [p1 removeObserver:self forKeyPath:@"age"];
}

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:
(id)object change:(NSDictionary<NSKeyValueChangeKey,id> *)change
context:(void *)context
{
    NSLog(@"监听到%@的%@改变了%@", object, keyPath,change);
}

// 打印内容
监听到<Person: 0x604000205460>的age改变了{
    kind = 1;
    new = 10;
    old = 2;
}
```

上述代码中可以看出，在添加监听之后，age属性的值在发生改变时，就会通知到监听者，执行监听者的observeValueForKeyPath方法。

探寻KVO底层实现原理

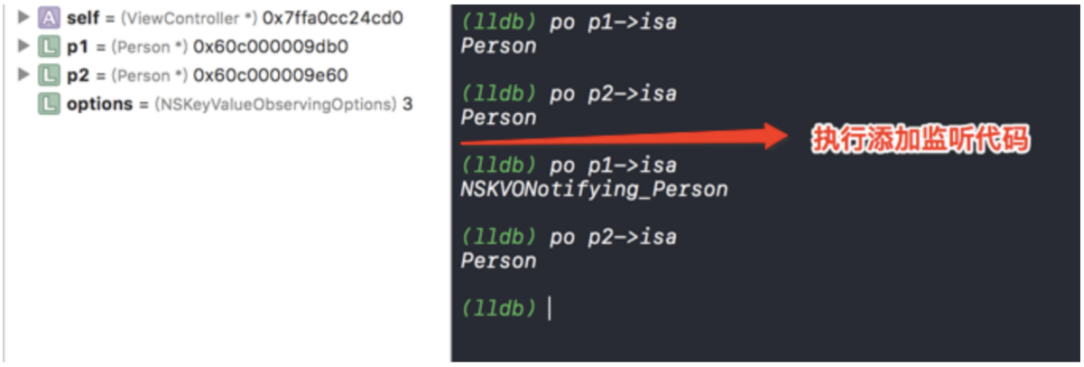
通过上述代码我们发现，一旦age属性的值发生改变时，就会通知到监听者，并且我们知道赋值操作都是调用 set方法，我们可以来到Person类中重写age的set方法，观察是否是KVO在set方法内部做了一些操作来通知监听者。

我们发现即使重写了set方法，p1对象和p2对象调用同样的set方法，但是我们发现p1除了调用set方法之外还会另外执行监听器的observeValueForKeyPath方法。

说明KVO在运行时获取对p1对象做了一些改变。相当于在程序运行过程中，对p1对象做了一些变化，使得p1对象在调用setage方法的时候可能做了一些额外的操作，所以问题出在对象身上，两个对象在内存中肯定不一样，两个对象可能本质上并不一样。接下来来探索KVO内部是怎么实现的。

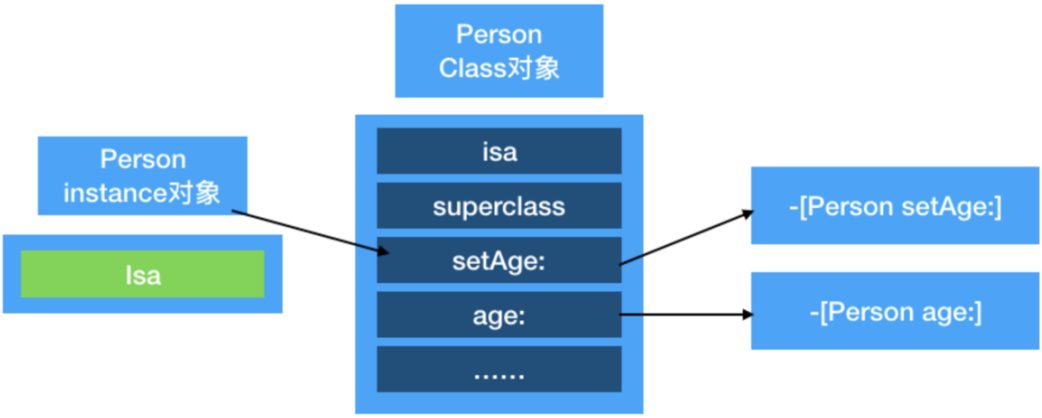
KVO底层实现分析

首先我们对上述代码中添加监听的地方打断点，看观察一下，addObserver方法对p1对象做了什么处理？也就是说p1对象在经过addObserver方法之后发生了什么改变，我们通过打印isa指针如下图所示



通过上图我们发现，p1对象执行过addObserver操作之后，p1对象的isa指针由之前的指向类对象Person变为指向NSKVONotifyin_Person类对象，而p2对象没有任何改变。也就是说一旦p1对象添加了KVO监听以后，其isa指针就会发生变化，因此set方法的执行效果就不一样了。

那么我们先来观察p2对象在内容中是如何存储的，然后对比p2来观察p1。 首先我们知道，p2在调用setage方法的时候，首先会通过p2对象中的isa指针找到Person类对象，然后在类对象中找到setage方法。然后找到方法对应的实现。如下图所示



但是刚才我们发现p1对象的isa指针在经过KVO监听之后已经指向了NSKVONotifyin_Person类对象，NSKVONotifyin_Person其实是Person的子类，那么也就是说其superclass指针是指向Person类对象的，NSKVONotifyin_Person是runtime在运行时生成的。那么p1对象在调用setage方法的时候，肯定会根据p1的isa找到NSKVONotifyin_Person，在NSKVONotifyin_Person中找setage的方法及实现。

经过查阅资料我们可以了解到。 NSKVONotifyin_Person中的setage方法中其实调用了 Fundation框架中C语言函数 _NSsetIntValueAndNotify，_NSsetIntValueAndNotify内部做的操作相当于，首先调用willChangeValueForKey将要改变方法，之后调用父类的setage方法对成员变量赋值，最后调用didChangeValueForKey已经改变方法。didChangeValueForKey中会调用监听器的监听方法，最终来到监听者的observeValueForKeyPath方法中。

那么如何验证KVO真的如上面所讲的方式实现?

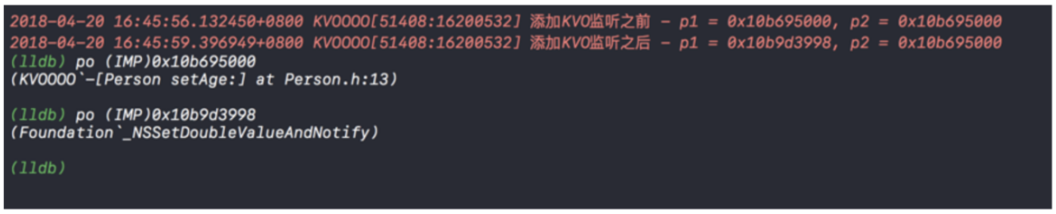
首先经过之前打断点打印isa指针，我们已经验证了，在执行添加监听的方法时，会将isa指针指向一个通过runtime创建的Person的子类NSKVONotifyin_Person。 另外我们可以通过打印方法实现的地址来看一下p1和p2的setage的方法实现的地址在添加KVO前后有什么变化。

```
// 通过methodForSelector找到方法实现的地址
NSLog(@"添加KVO监听之前 - p1 = %p, p2 = %p", [p1 methodForSelector:
@selector(setAge:)],[p2 methodForSelector: @selector(setAge:)]);

NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew |
NSKeyValueObservingOptionOld;
```

```
[p1 addObserver:self forKeyPath:@"age" options:options
context:nil];

NSLog(@"添加KVO监听之后 - p1 = %p, p2 = %p", [p1 methodForSelector:
@selector(setAge:)],[p2 methodForSelector: @selector(setAge:)]);
```

我们发现在添加KVO监听之前，p1和p2的setAge方法实现的地址相同，而经过KVO监听之后，p1的setAge方法实现的地址发生了变化，我们通过打印方法实现来看一下前后的变化发现，确实如我们上面所讲的一样，p1的setAge方法的实现由Person类方法中的setAge方法转换为了C语言的Foundation框架的_NSsetIntValueAndNotify函数。

Foundation框架中会根据属性的类型，调用不同的方法。例如我们之前定义的int类型的age属性，那么我们看到Foundation框架中调用的_NSset**Int**ValueAndNotify函数。那么我们把age的属性类型变为**double**重新打印一遍



我们发现调用的函数变为了_NSSet**Double**ValueAndNotify，那么这说明Foundation框架中有许多此类型的函数，通过属性的不同类型调用不同的函数。 那么我们可以推测Foundation框架中还有很多例如`_NSSetBoolValueAndNotify`、`_NSSetCharValueAndNotify`、`_NSSetFloatValueAndNotify`、`_NSSetLongValueAndNotify`等等函数。

我们可以找到Foundation框架文件，通过命令行查询关键字找到相关函数

```
~/Desktop mj$ nm Foundation | grep ValueAndNotify
0000000183080624 t __NSSetSizeValueAndNotify
0000000183080230 t __NSSetPointValueAndNotify
0000000183080378 t __NSSetRangeValueAndNotify
0000000183039cc4 t __NSSetFloatValueAndNotify
0000000183024f30 t __NSSetIntValueAndNotify
0000000182ffb37c t __NSSetCharValueAndNotify
0000000182ffb534 t __NSSetObjectValueAndNotify
0000000018307f5f8 t __NSSetBoolValueAndNotify
0000000018307f868 t __NSSetDoubleValueAndNotify
0000000018307fae0 t __NSSetLongValueAndNotify
0000000018307fc0 t __NSSetShortValueAndNotify
0000000018307fd50 t __NSSetLongLongValueAndNotify
00000001830804c0 t __NSSetRectValueAndNotify
```

首先我们知道，NSKVONotifyin_Person作为Person的子类，其superclass指针指向Person类，并且NSKVONotifyin_Person内部一定对setAge方法做了单独的实现，那么NSKVONotifyin_Person同Person类的差别可能就在于其内存储的对象方法及实现不同。 我们通过runtime分别打印Person类对象和NSKVONotifyin_Person类对象内存储的对象方法

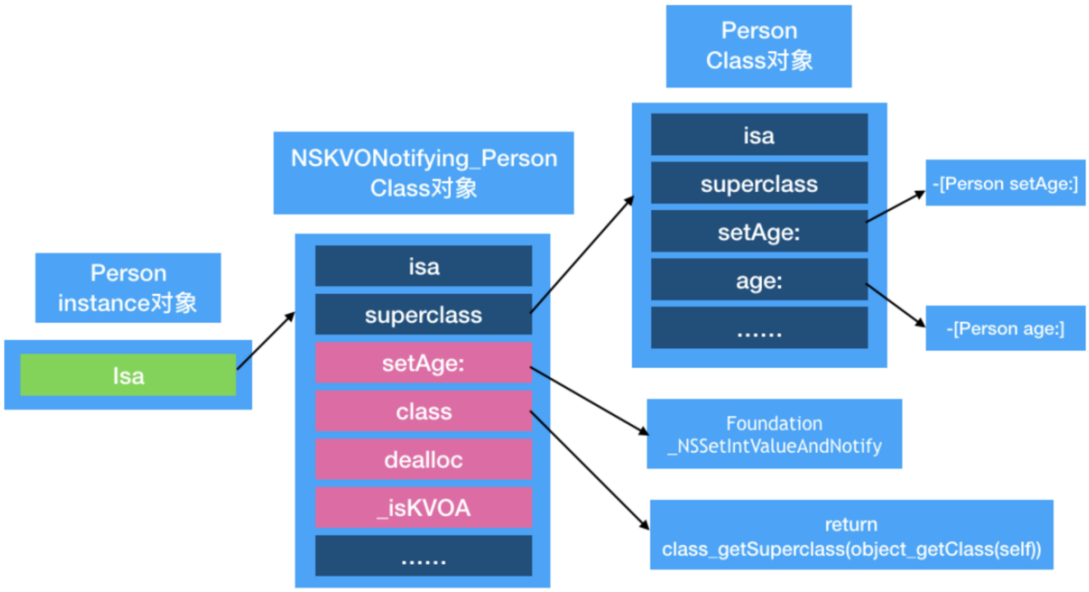
```
- (void)viewDidLoad {
    [super viewDidLoad];

    Person *p1 = [[Person alloc] init];
    p1.age = 1.0;
    Person *p2 = [[Person alloc] init];
    p1.age = 2.0;
    // self 监听 p1的 age属性
    NSKeyValueObservingOptions options =
NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld;
    [p1 addObserver:self forKeyPath:@"age" options:options
context:nil];

    [self printMethods: object_getClass(p2)];
    [self printMethods: object_getClass(p1)];

    [p1 removeObserver:self forKeyPath:@"age"];
}

- (void) printMethods:(Class)cls
```

```
{
    unsigned int count ;
    Method *methods = class_copyMethodList(cls, &count);
    NSMutableString *methodNames = [NSMutableString string];
    [methodNames appendFormat:@"%@ - ", cls];

    for (int i = 0 ; i < count; i++) {
        Method method = methods[i];
        NSString *methodName  =
NSStringFromSelector(method_getName(method));

        [methodNames appendString: methodName];
        [methodNames appendString:@" "];

    }

    NSLog(@"%@",methodNames);
    free(methods);
}
```

上述打印内容如下

```
KVOOOO[56283:16530334] Person - setAge: age
KVOOOO[56283:16530334] NSKVONotifying_Person - setAge: class dealloc _isKVOA
```

通过上述代码我们发现NSKVONotifyin_Person中有4个对象方法。分别为setAge: class dealloc _isKVOA，那么至此我们可以画出NSKVONotifyin_Person的内存结构以及方法调用顺序。



这里NSKVONotifyin_Person重写class方法是为了隐藏NSKVONotifyin_Person。不被外界所看到。我们在p1添加过KVO监听之后，分别打印p1和p2对象的class可以发现他们都返回Person。

```
NSLog(@"%@,%@",[p1 class],[p2 class]);
// 打印结果 Person,Person
```

如果NSKVONotifyin_Person不重写class方法，那么当对象要调用class对象方法的时候就会一直向上找来到nsobject，而nsobect的class的实现大致为返回自己isa指向的类，返回p1的isa指向的类那么打印出来的类就是NSKVONotifyin_Person，但是apple不希望将NSKVONotifyin_Person类暴露出来，并且不希望我们知道NSKVONotifyin_Person内部实现，所以在内部重写了class类，直接返回Person类，所以外界在调用p1的class对象方法时，是Person类。这样p1给外界的感觉p1还是Person类，并不知道NSKVONotifyin_Person子类的存在。

那么我们可以猜测NSKVONotifyin_Person内重写的class内部实现大致为

```
- (Class) class {
    // 得到类对象，在找到类对象父类
    return class_getSuperclass(object_getClass(self));
}
```

验证didChangeValueForKey:内部会调用observer的observeValueForKeyPath:ofObject:change:context:方法

我们在Person类中重写willChangeValueForKey:和didChangeValueForKey:方法，模拟他们的实现。

```
- (void)setAge:(int)age
{
    NSLog(@"setAge:");
    _age = age;
}
- (void)willChangeValueForKey:(NSString *)key
{
    NSLog(@"willChangeValueForKey: - begin");
    [super willChangeValueForKey:key];
    NSLog(@"willChangeValueForKey: - end");
}
- (void)didChangeValueForKey:(NSString *)key
{
    NSLog(@"didChangeValueForKey: - begin");
    [super didChangeValueForKey:key];
    NSLog(@"didChangeValueForKey: - end");
}
```

再次运行来查看didChangeValueForKey的方法内运行过程，通过打印内容可以看到，确实在didChangeValueForKey方法内部已经调用了observer的observeValueForKeyPath:ofObject:change:context:方法。

```
2018-04-21 09:29:07.118237+0800 KVOOOO[54016:16399197] setAge
2018-04-21 09:29:09.327826+0800 KVOOOO[54016:16399197] willChangeValueForKey: - begin
2018-04-21 09:29:09.328074+0800 KVOOOO[54016:16399197] willChangeValueForKey: - end
2018-04-21 09:29:09.328187+0800 KVOOOO[54016:16399197] setAge
2018-04-21 09:29:09.328282+0800 KVOOOO[54016:16399197] didChangeValueForKey: - begin
2018-04-21 09:29:09.328536+0800 KVOOOO[54016:16399197] 监听到<Person: 0x6000000037a0>的age改变了{
    kind = 1;
    new = 37123;
    old = 0;
}
2018-04-21 09:29:09.328652+0800 KVOOOO[54016:16399197] didChangeValueForKey: - end
```

回答问题：

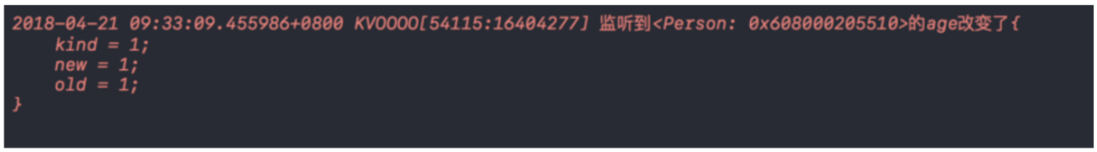
> 1. iOS用什么方式实现对一个对象的KVO？（KVO的本质是什么？）答. 当一

> 个对象使用了KVO监听，iOS系统会修改这个对象的isa指针，改为指向一个全新的通过Runtime动态创建的子类，子类拥有自己的set方法实现，set方法实现内部会顺序调用**willChangeValueForKey方法、原来的setter方法实现、didChangeValueForKey方法，而didChangeValueForKey方法内部又会调用监听器的observeValueForKeyPath:ofObject:change:context:监听方法。**

> 2. 如何手动触发KVO 答. 被监听的属性的值被修改时，就会自动触发KVO。如果想要手动触发KVO，则需要我们自己调用**willChangeValueForKey和didChangeValueForKey**方法即可在不改变属性值的情况下手动触发KVO，并且这两个方法缺一不可。

通过以下代码可以验证

```
Person *p1 = [[Person alloc] init];
p1.age = 1.0;

NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew |
NSKeyValueObservingOptionOld;
[p1 addObserver:self forKeyPath:@"age" options:options
context:nil];

[p1 willChangeValueForKey:@"age"];
[p1 didChangeValueForKey:@"age"];

[p1 removeObserver:self forKeyPath:@"age"];
```



通过打印我们可以发现，didChangeValueForKey方法内部成功调用了observeValueForKeyPath:ofObject:change:context:，并且age的值并没有发生改变。