



冰凌天 Lv2

2018年08月13日 阅读 161

[关注](#)

小码哥iOS学习笔记第四天: KVO的本质

KVO的全称是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

一、KVO的使用

- 新建工程, 定义 `Person` 类继承自 `NSObject`, 并添加 `int` 类型的属性 `age`

ObjectiveC

```
@interface Person : NSObject
@property (nonatomic, assign) int age;
@end

@implementation Person
@end
```

- 在 `ViewController` 中添加两个 `Person` 类型的属性 `person1` 和 `person2`, 并给 `person1` 添加监听 `age` 属性的观察者, 当点击屏幕时修改这两个对象的 `age` 属性值

ObjectiveC

```
@interface ViewController ()

@property (nonatomic, strong) Person *person1;

@property (nonatomic, strong) Person *person2;

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    self.person1 = [[Person alloc] init];
    self.person1.age = 1;
}
```

```

    self.person2 = [[Person alloc] init];
    self.person2.age = 2;

    NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld;
    [self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];
}

- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    self.person1.age = 21;
    self.person2.age = 22;
}

/**
 当被观察的属性，使用`set`方法赋值时，触发观察者

@param keyPath 被监听的属性
@param object 添加监听的对象
@param change 属性改变前后的值
@param context 添加观察者时传入的参数
*/
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change {
    NSLog(@"%@ - %@ - %@ - %@", object, keyPath, change, context);
}

- (void)dealloc
{
    [self.person1 removeObserver:self forKeyPath:@"age"];
}

@end

```

- 运行程序, 点击屏幕, 会有如下打印:

```

<Person: 0x60c000014b20> - age - {
    kind = 1;
    new = 21;
    old = 1;
} - 年龄

```

ObjectiveC

- 打印中只有 `person1` 的属性值发生改变信息, 而没有 `person2` 的属性值改变的信息
- 这是因为给 `person1` 添加了观察者, 而 `person2` 没有添加

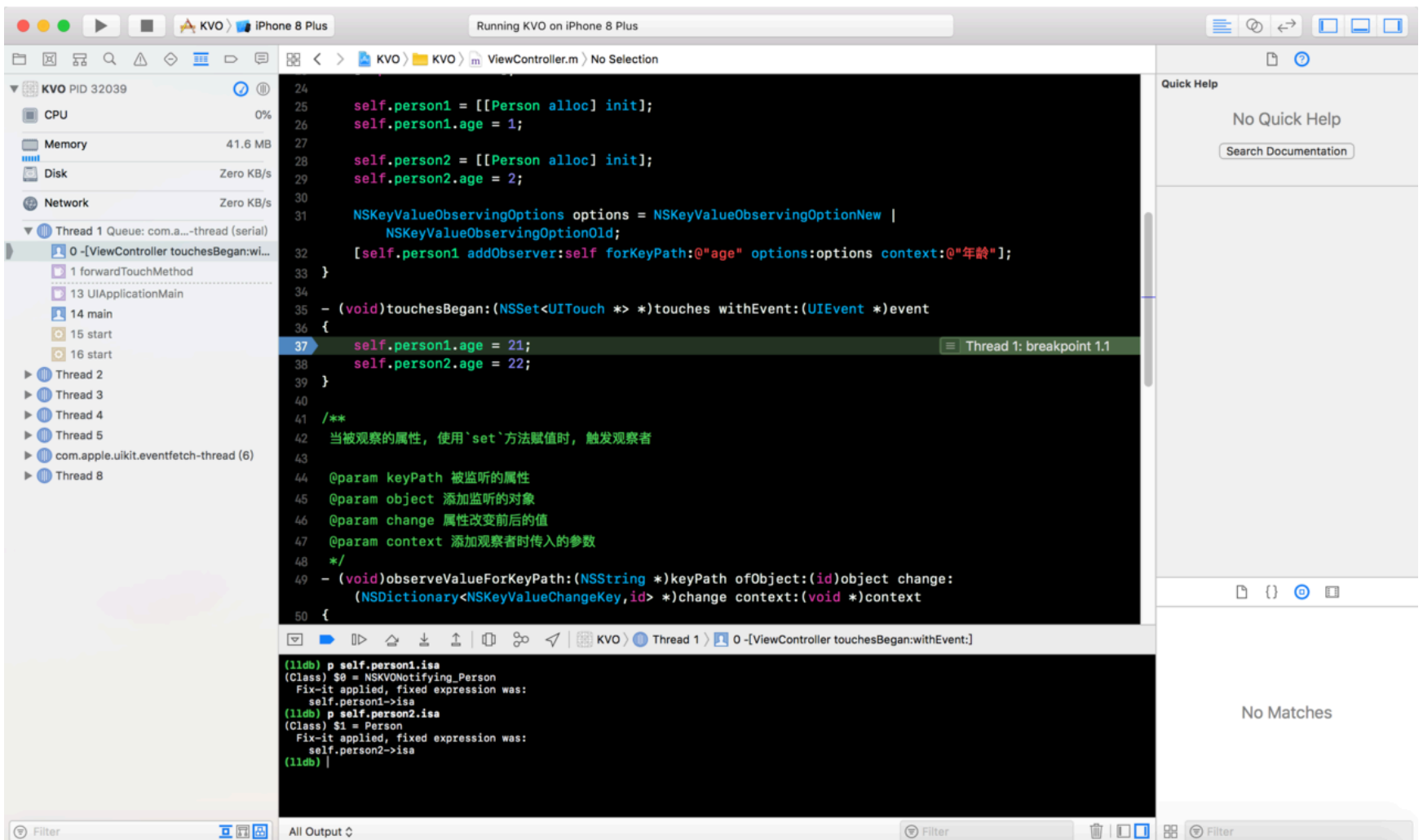
- 对象添加KVO监听属性, 类似于下图



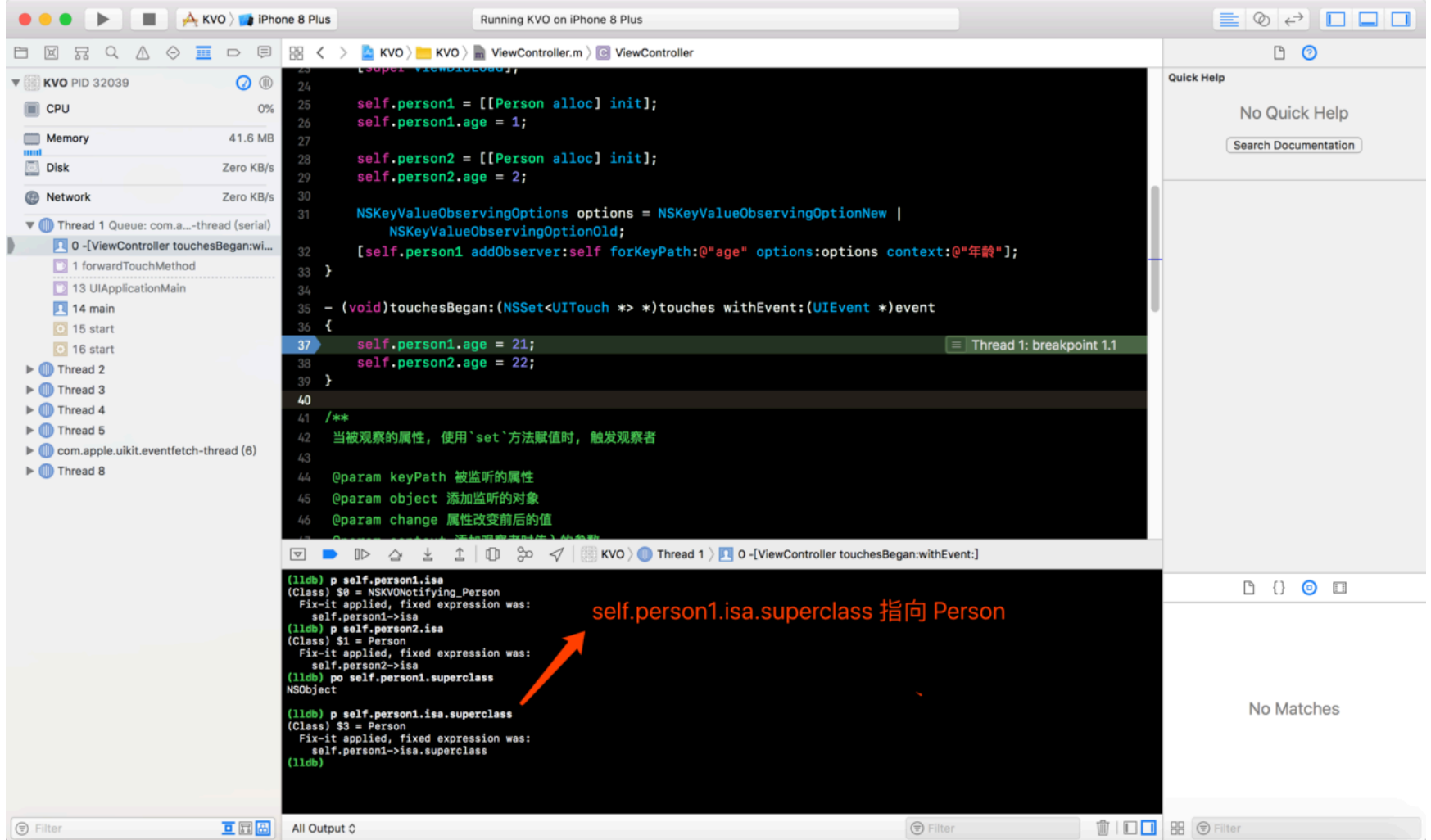
问: 为什么同样都是调用了 `setAge:` 方法, `person1` 却能监听 `age` 的属性值改变?

二、添加KVO的对象的isa指针指向何处

- 下图就是上面创建的工程, 我们在 `touchesBegan:withEvent:` 方法中
- 打断点, 当点击控制器的view后, 查看 `person1` 和 `person2` 的 `isa` 指针指向何处



- 根据结果可以知道
 - `person1` 的 `isa` 指向类对象 `NSKeyValueObserving_Person`
 - `person2` 的 `isa` 指向类对象 `Person`
- 通过 `person1` 的 `isa` 找到 `NSKeyValueObserving_Person` 后, 再次调用 `superclass`, 可以看到 `NSKeyValueObserving_Person` 的父类是 `Person`



- 说明: 添加了观察者(KVO)的对象, 它的 `isa` 指针发生了改变, 指向了系统动态生成的子类 `NSKVONotifying_Person`
- 已经知道对象调用方法的过程:
 - 首先通过 `isa` 指针, 找到类对象
 - 在类对象中查找方法, 如果方法存在就会调用
- 所以 `person1` 调用的 `setAge:` 方法, 是子类 `NSKVONotifying_Person` 中重写的 `setAge:` 方法
- 这就是为什么, 明明 `person1` 和 `person2` 都调用了 `setAge:` 方法, 而 `person1` 会有属性监听

1、未使用KVO监听的Person对象

2、使用KVO监听的Person对象

- 上面是通过 `isa` 验证了 `person1` 指向了 `NSKeyValueObservingPerson` 类, 下面使用代码进行验证
- 在给 `person1` 添加观察者的前后, 分别打印 `person1` 和 `person2` 的类型

3、通过代码验证 `person1` 的类型是 `NSKeyValueObservingPerson`

ObjectiveC

```
NSLog(@"%@ - %@",
      object_getClass(self.person1),
      object_getClass(self.person2));

NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld;
[self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];

NSLog(@"%@ - %@",
      object_getClass(self.person1),
      object_getClass(self.person2));
```

- 执行后打印如下:

```
Person - Person
NSKeyValueObservingPerson - Person
```

- 根据打印结果, 可以证明, 在给 `person1` 添加观察者之后, `person1` 的类型是 `NSKeyValueObservingPerson`

三、验证 `NSKeyValueObserving_Person` 中 `setAge:` 的方法实现, 是 `_NSSetIntValueAndNotify` 函数

- 在 `person1` 添加观察者的前后, 设置打印 `setAge` 方法地址的代码

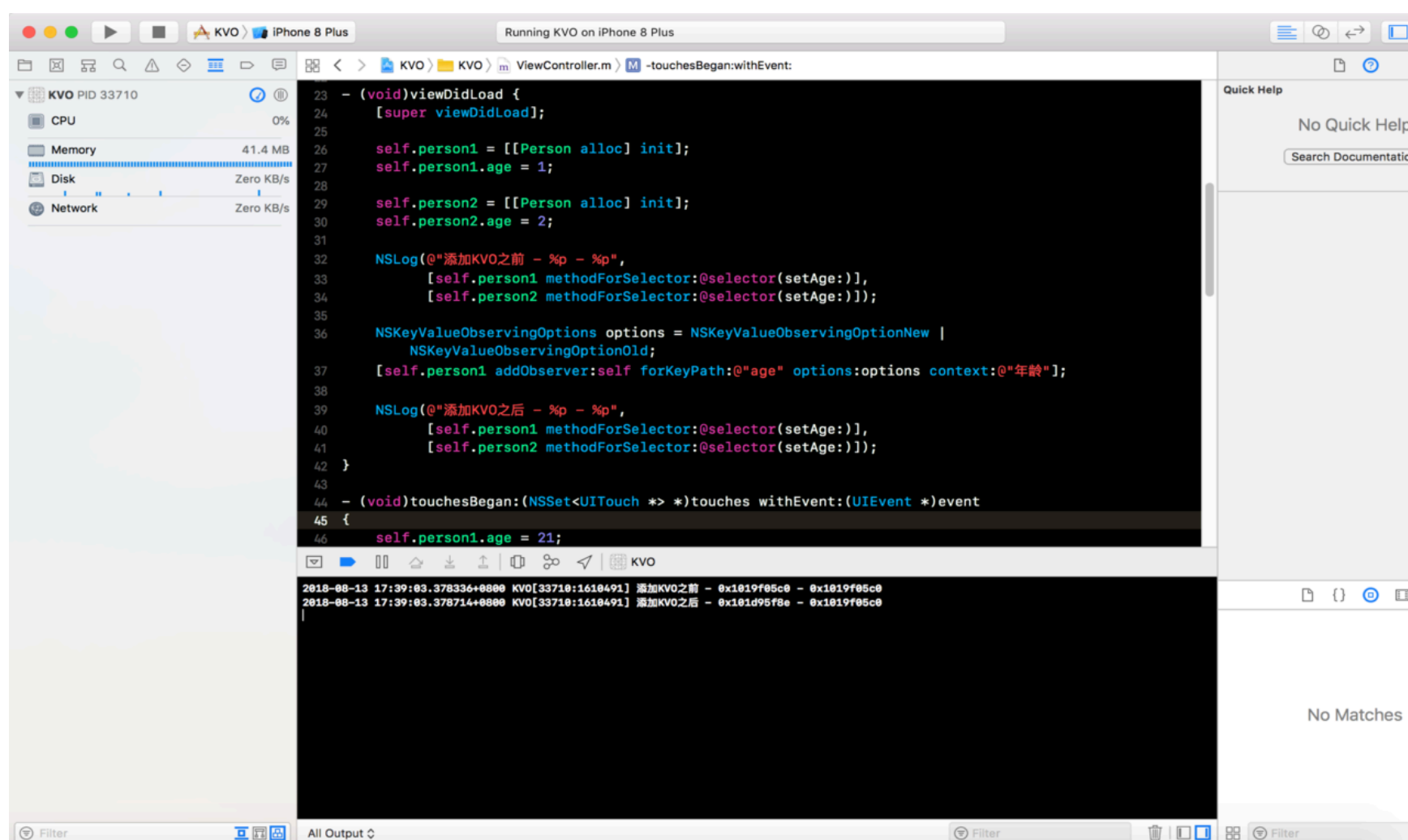
ObjectiveC

```
NSLog(@"添加KVO之前 - %p - %p",
      [self.person1 methodForSelector:@selector(setAge:)],
      [self.person2 methodForSelector:@selector(setAge:)]);
```

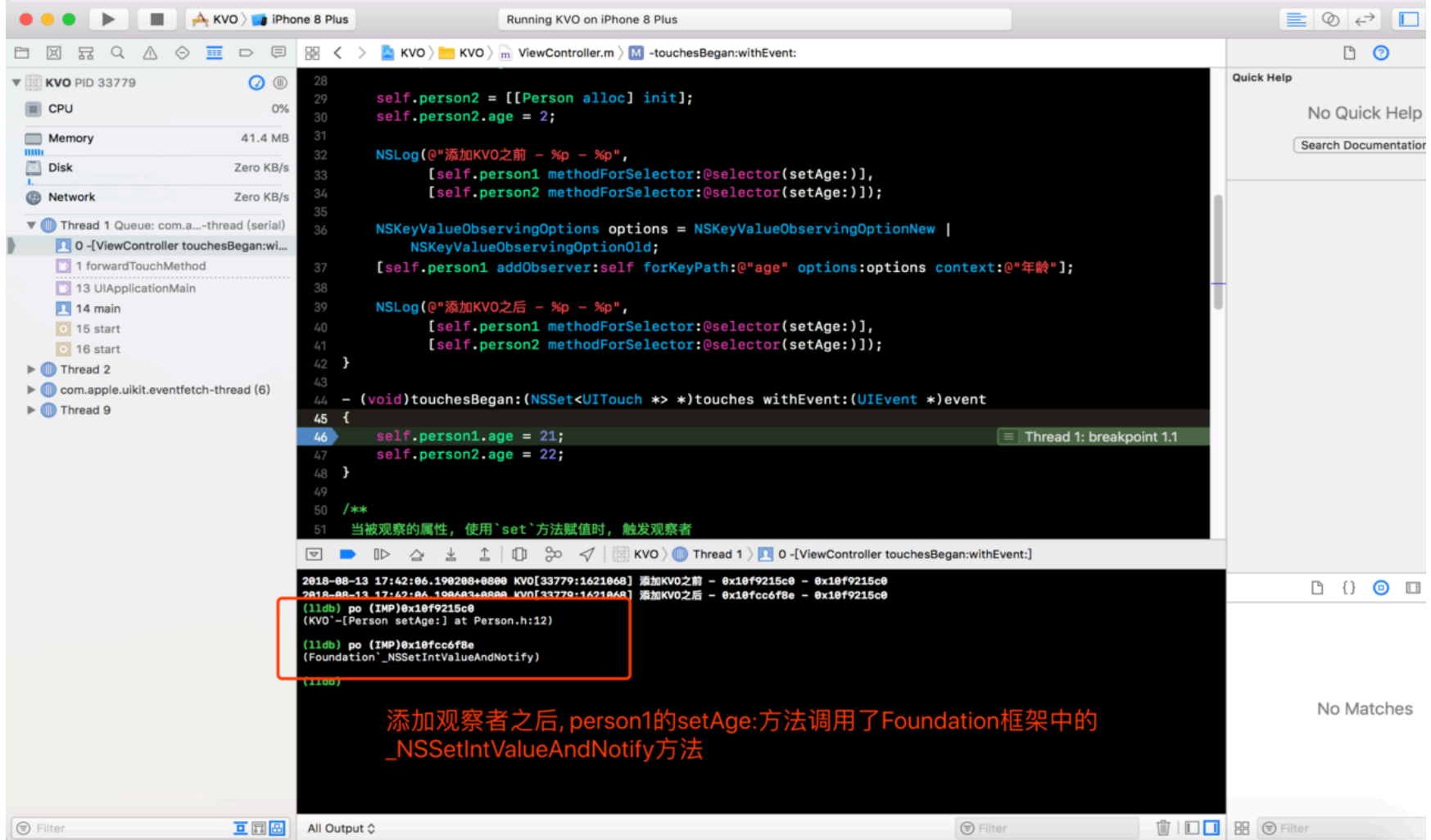
```
NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld;
[self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];
```

```
NSLog(@"添加KVO之后 - %p - %p",
      [self.person1 methodForSelector:@selector(setAge:)],
      [self.person2 methodForSelector:@selector(setAge:)]);
```

- 打印结果如下图:



- 很明显, 在添加KVO的前后, `person1` 调用的 `setAge:` 方法已经改变
- 下面使用lldb打印一下 `setAge:` 方法



四、探索 `NSKVONotifying_Person` 类对象的 `isa` , 指向何处

- 在添加KVO前后, 添加如下代码, 打印 `person1` 的类对象和元类对象

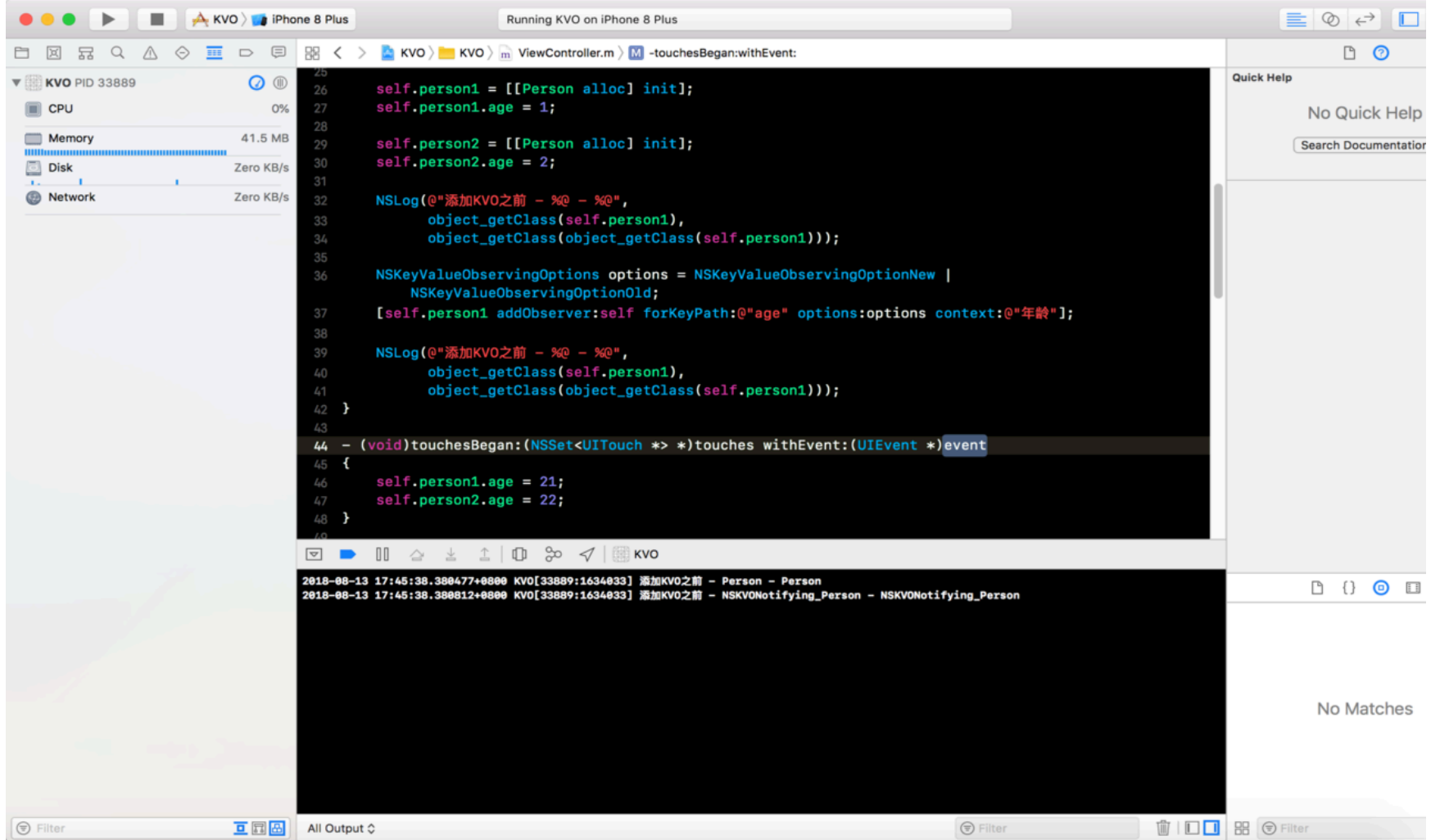
ObjectiveC

```
NSLog(@"添加KVO之前 - %@ - %@",
      object_getClass(self.person1),
      object_getClass(object_getClass(self.person1)));

NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObservingO
[self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];

NSLog(@"添加KVO之前 - %@ - %@",
      object_getClass(self.person1),
      object_getClass(object_getClass(self.person1)));
```

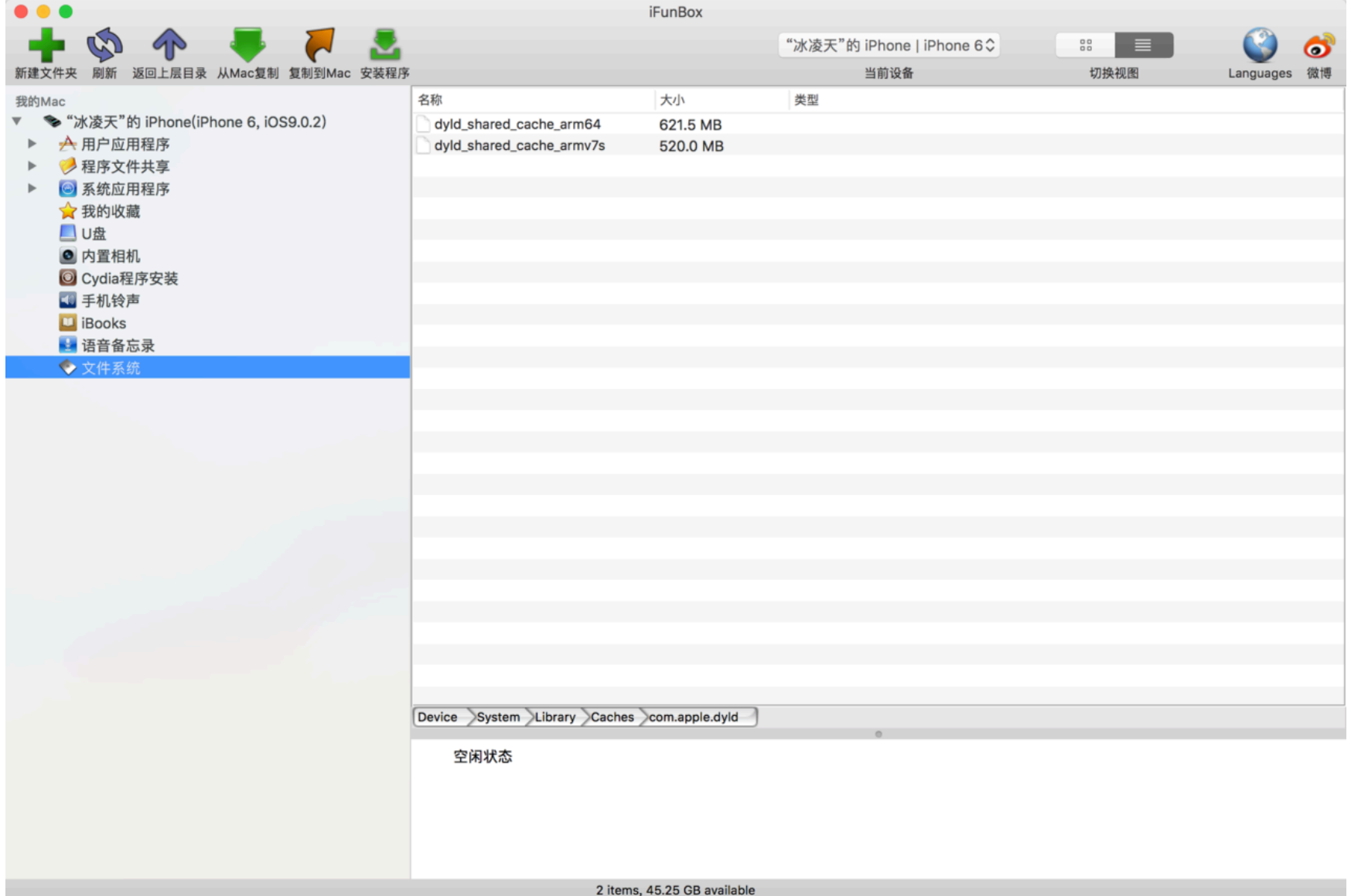
- 打印结果如下



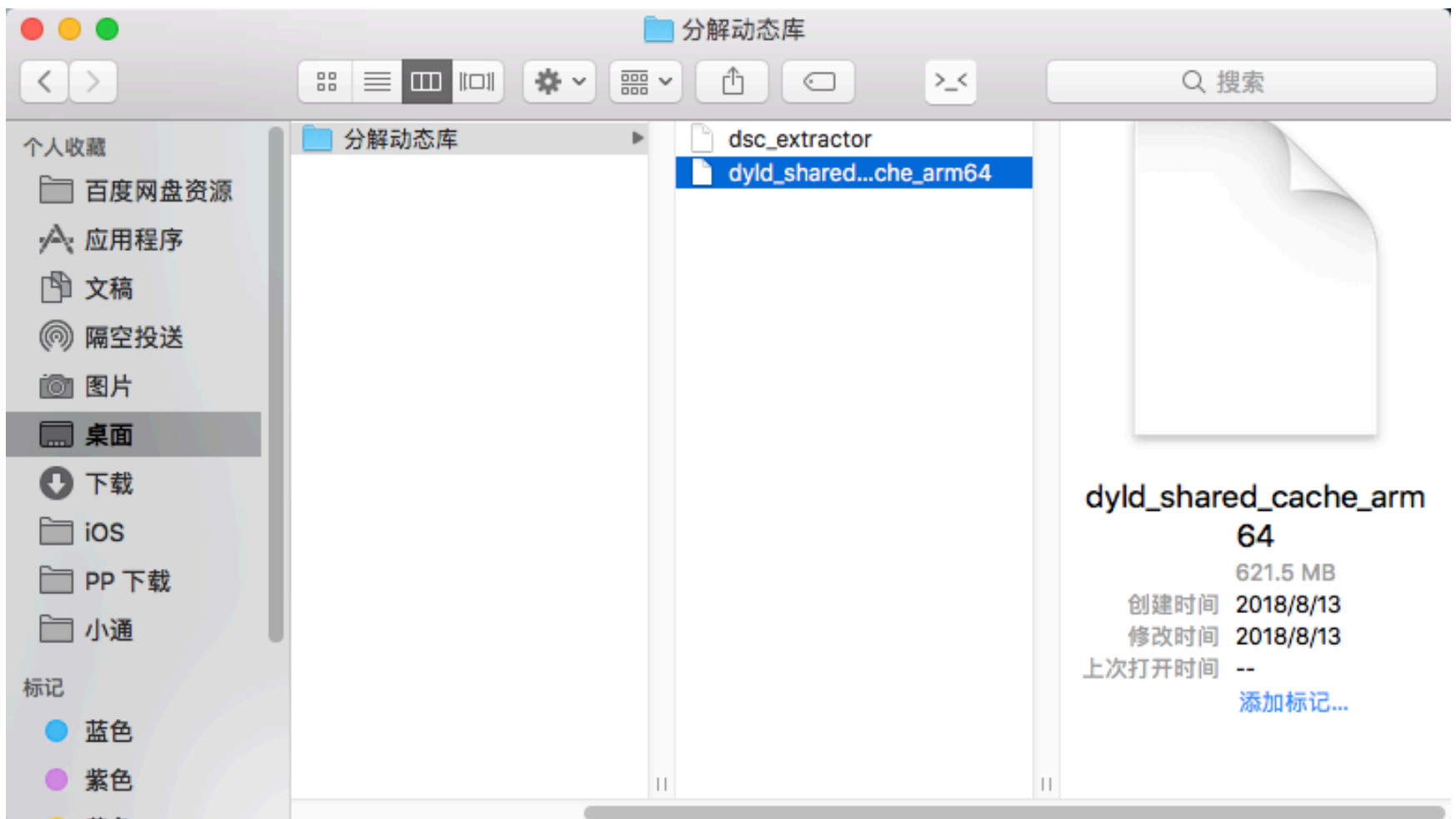
- 由打印可知: `NSKV0Notifying_Person` 类对象的 `isa` 指向 `NSKV0Notifying_Person` 的元类对象

五、通过逆向, 查看 Foundation 中的 _NSSetIntValueAndNotify 函数

- 使用 **iFunBox** 查看越狱手机中的动态库文件



- 我使用的iPhone6, 所以这里查看arm64架构下的动态库文件
- 将 `dyld_shared_cache_arm64` 文件托至电脑(复制)

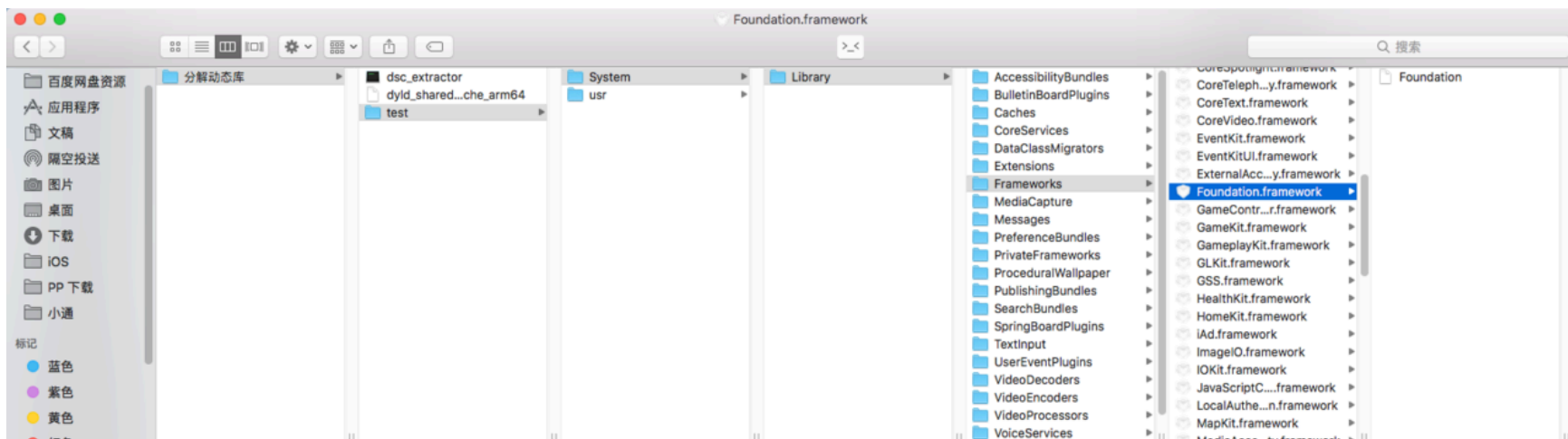


- 通过终端, 使用 `dsc_extractor` 对 `dyld_shared_cache_arm64` 进行分解

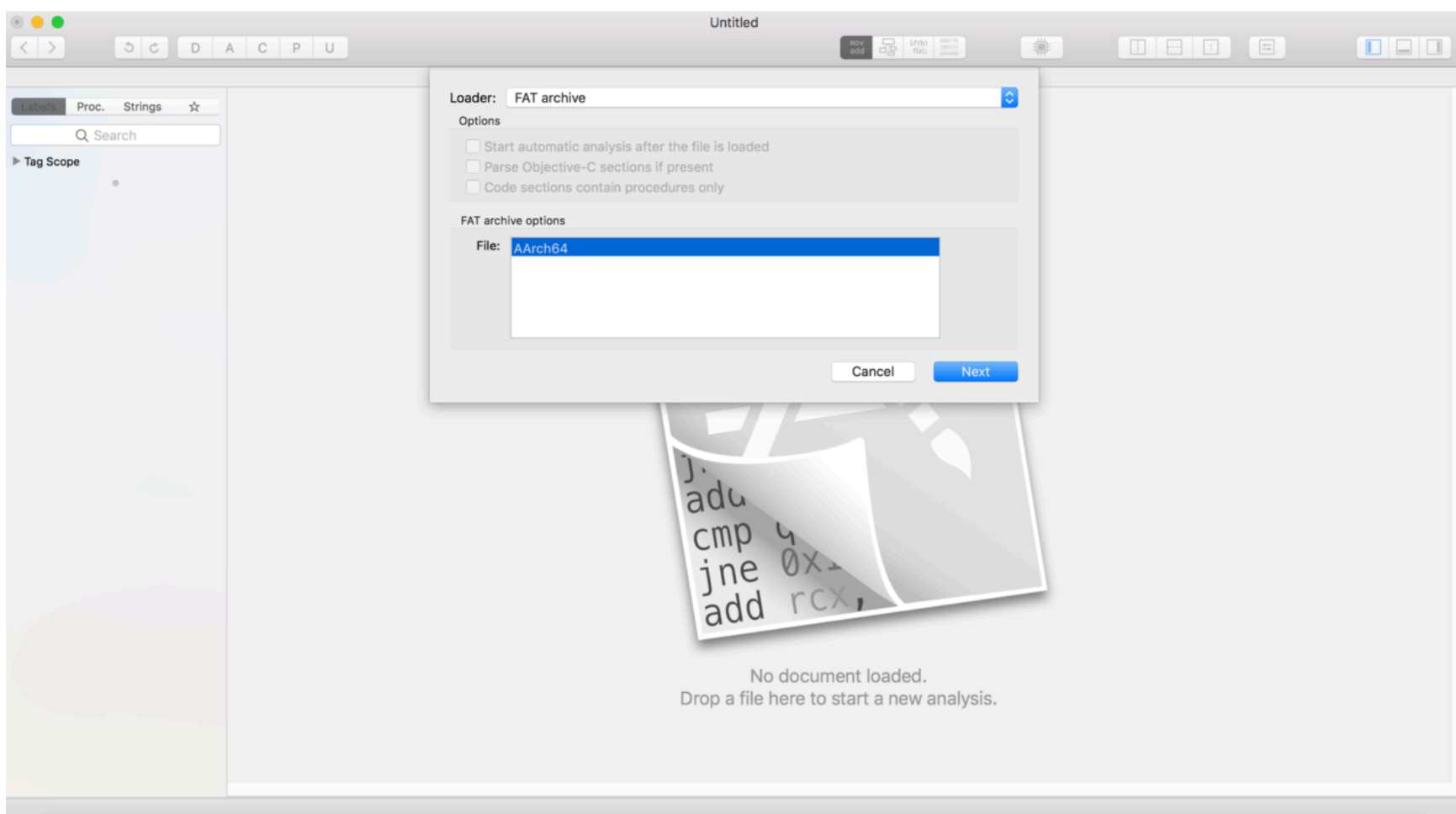
终端命令

```
./dsc_extractor dyld_shared_cache_arm64 test
```

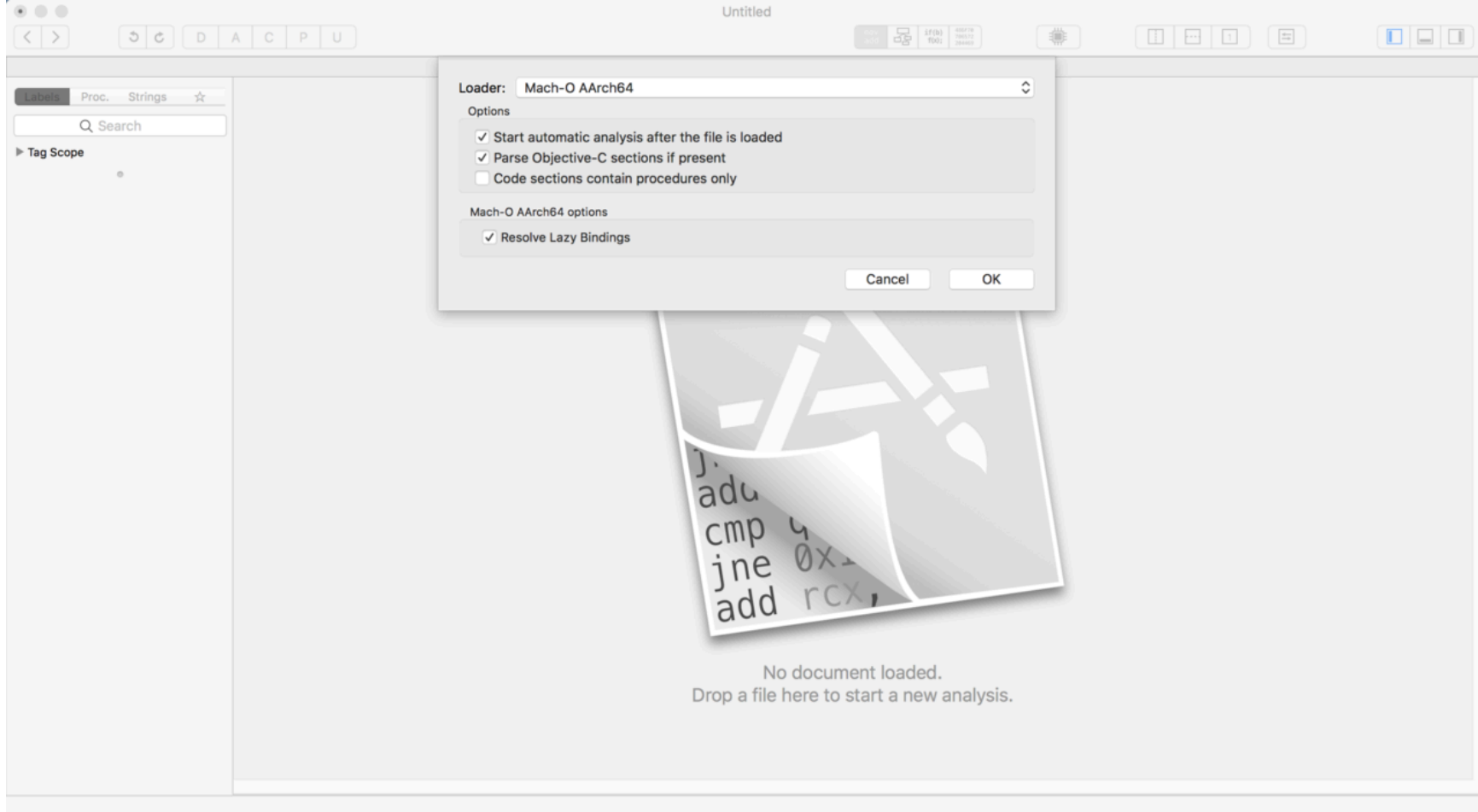
- 分解出的 **Foundation** 动态库如下



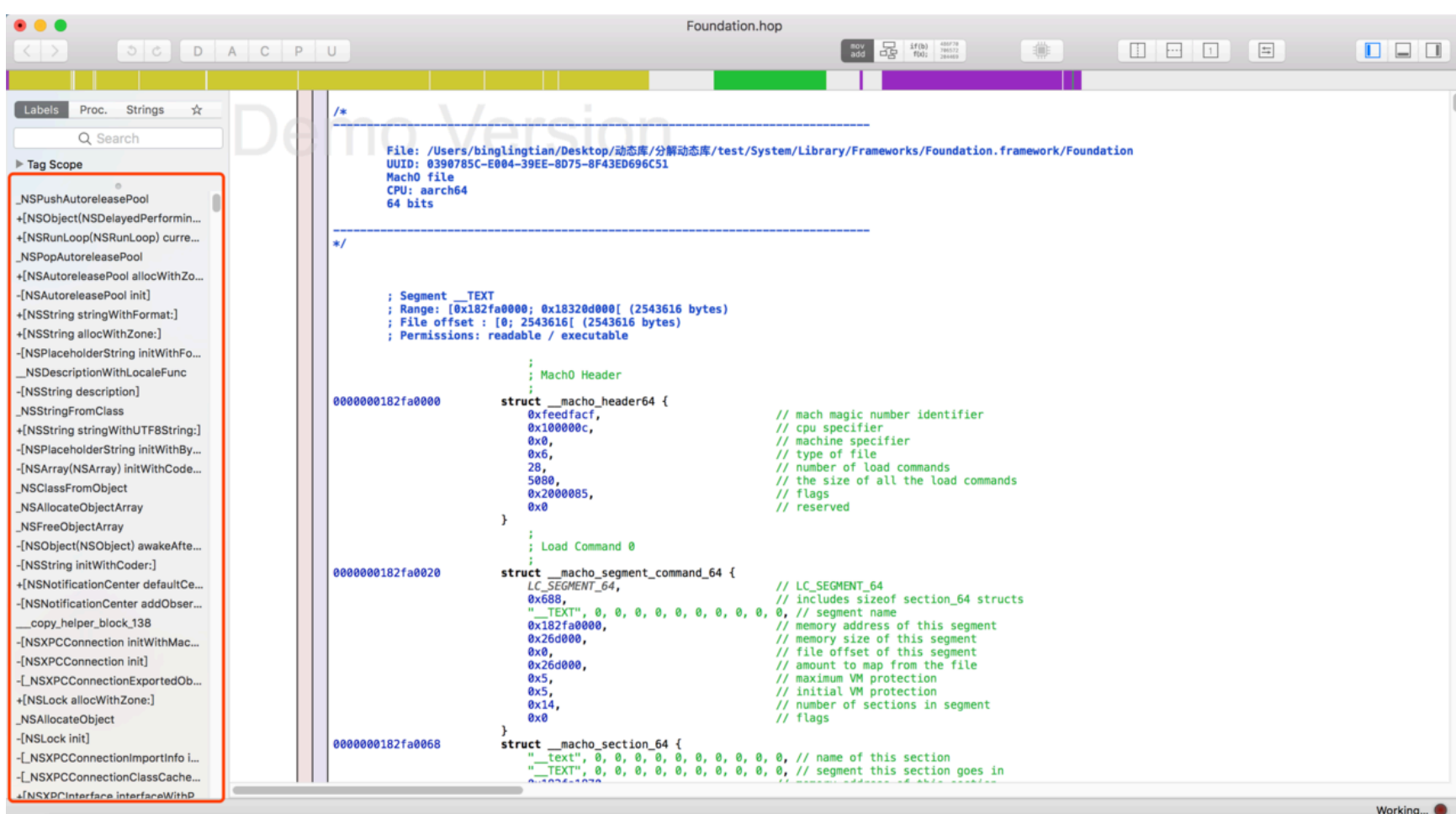
- 接下来使用反编译工具 **hopper** , 对 **Foundation** 反编译
- 使用 **hopper** 打开 **Foundation** 动态库文件



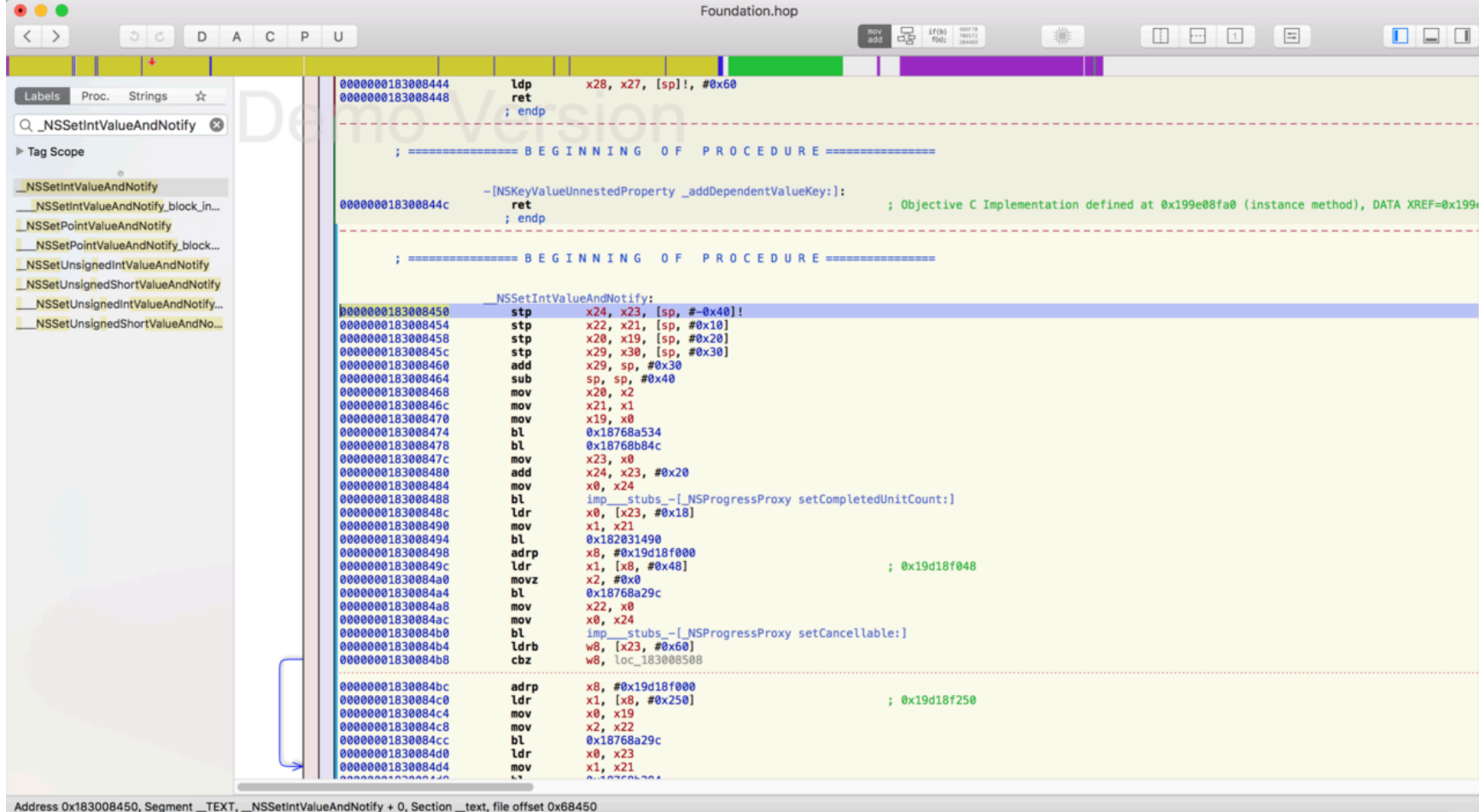
- Nest



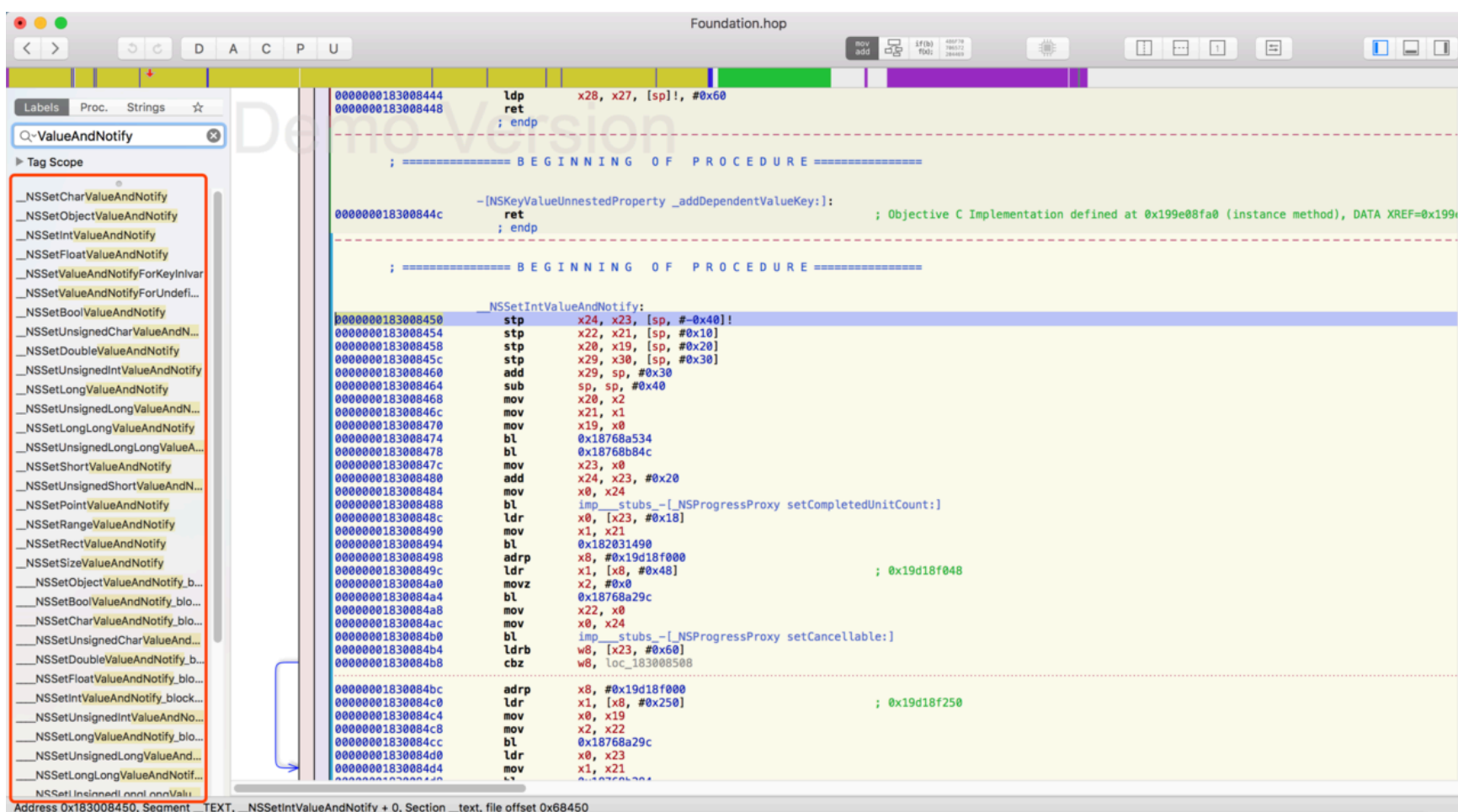
- OK, 可以看到反编译成功



- 搜索 `_NSSetIntValueAndNotify` , 可以看到 `Foundation` 中确实有 `_NSSetIntValueAndNotify` 函数



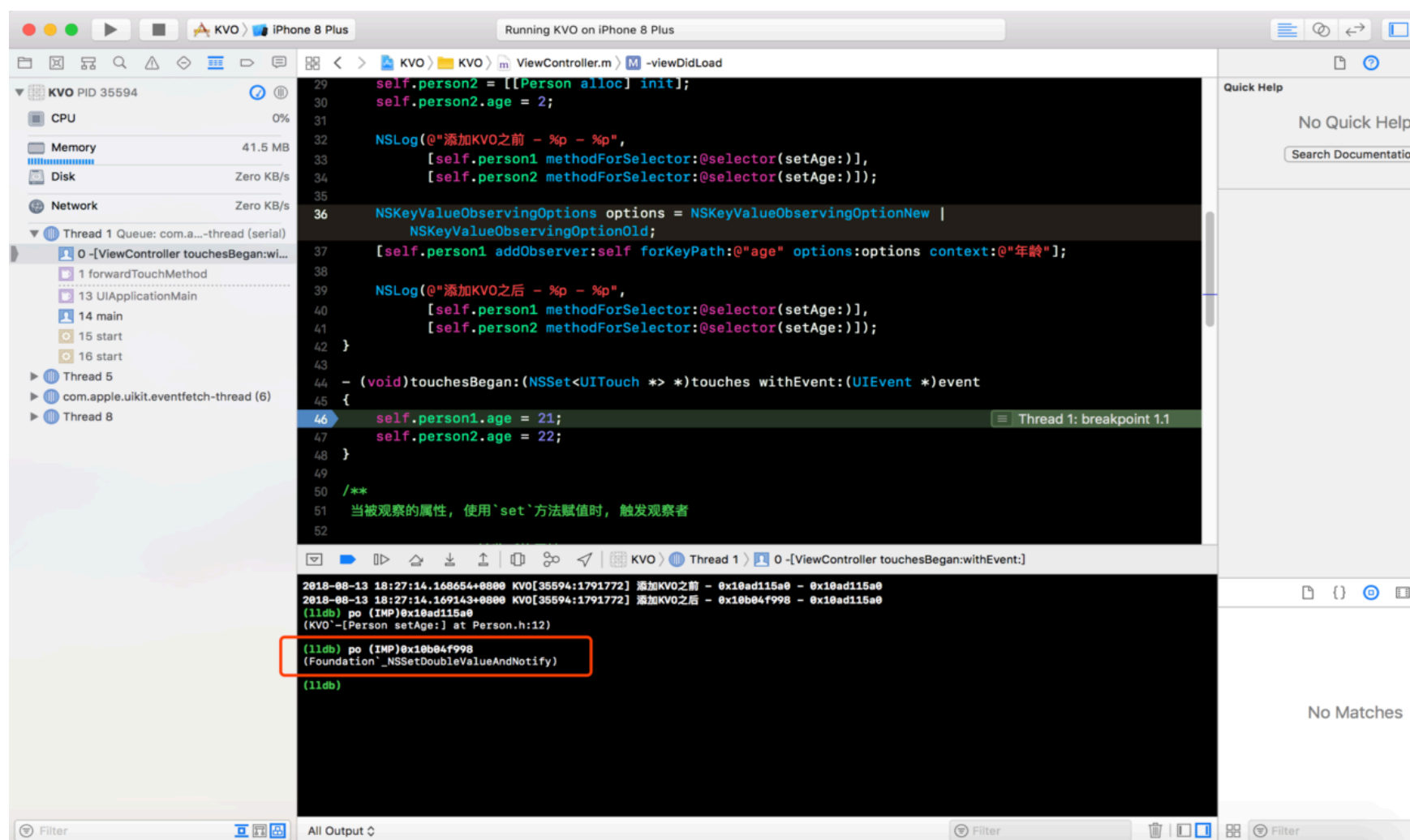
- 搜索 **ValueAndNotify** , 可以看到有很多类似的方法



- 根据搜索结果可以推断出, 对不同类型的属性添加观察, 就会调用对应属性类型的 **_NSSet*ValueAndNotify** 方法, * 表示类型
- 验证这个推断, 将 **Person** 的 **age** 属性类型, 从 **int** 改为 **double**

```
@interface Person : NSObject
@property (nonatomic, assign) double age;
@end
```

- 再次打印 `setAge:` 方法实现:



- 根据结果, 验证推断正确

六、_NSSet*ValueAndNotify的内部实现

```
[self willChangeValueForKey:@"age"];
// 原来的setter实现
[self didChangeValueForKey:@"age"];
```

- 调用顺序:
 - 调用 `willChangeValueForKey:`
 - 调用原来的 `setter` 实现
 - 调用 `didChangeValueForKey:`
 - `didChangeValueForKey:` 内部会调用 `observeValueForKeyPath:ofObject:change:context:`

- 通过代码验证, 在 `Person.m` 中手动实现 `willChangeValueForKey` 和 `didChangeValueForKey` 以及 `setAge:` 方法, 代码如下:

ObjectiveC

```
@implementation Person
- (void)setAge:(int)age
{
    _age = age;
    NSLog(@"setAge:");
}

- (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");
}
@end
```

- 运行程序, 点击控制器的View, 修改 `person1` 的 `age` 属性, 有如下打印:

ObjectiveC

```
// 1
willChangeValueForKey - begin
// 2
willChangeValueForKey - end
// 3
setAge:
// 4
didChangeValueForKey - begin
// 5
<Person: 0x60800001a510> - age - {
    kind = 1;
    new = 21;
    old = 1;
} - 年龄
// 6
didChangeValueForKey - end
```

- 根据打印: 可以证明以上的 `_NSSet*ValueAndNotify` 的内部实现

七、子类内部的方法

- 在 使用KVO监听的Person对象 的图片中, `NSKeyValueObserving_Person` 的类对象中, 一共有两个指针 `isa` 和 `superclass`, 四个方法 `setAge:`, `class`, `dealloc` 和 `_isKVOA`
- 下面使用 `Runtime` 代码, 来验证 `NSKeyValueObserving_Person` 中确实存在这四个方法
- 首先在 `ViewController` 中添加下面的方法:

ObjectiveC

```
- (void)printMethodNamesOfClass:(Class)cls
{
    unsigned int count;
    Method *methodList = class_copyMethodList(cls, &count);

    NSMutableArray *array = [NSMutableArray array];
    for (int i = 0; i < count; i++) {
        Method method = methodList[i];

        NSString *methodName = NSStringFromSelector(method_getName(method));

        [array addObject:methodName];
    }
    NSLog(@"%@ - %@", cls, array);
}
```

- 同时删除 `Person.m` 中的所有方法

ObjectiveC

```
@implementation Person

@end
```

- 接着在给 `person1` 添加观察者的后面调用方法, 传入 `cls`

ObjectiveC

```
self.person1 = [[Person alloc] init];
self.person1.age = 1;

self.person2 = [[Person alloc] init];
self.person2.age = 2;
```



```
NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObserving0
[self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];

[self printMethodNamesOfClass:object_getClass(self.person1)];
[self printMethodNamesOfClass:object_getClass(self.person2)];
```

- 运行程序后, 有以下打印:

```
NSKVONotifying_Person - (
    "setAge:",
    class,
    dealloc,
    "_isKVOA"
)
Person - (
    "setAge:",
    age
)
```

ObjectiveC

- 可以看到NSKVONotifying_Person中有四个方法:
 - setAge:
 - class
 - dealloc
 - _isKVOA

1、推断 NSKVONotifying_Person 中 class 方法的实现

- 首先在给 person1 添加观察者的后面添加打印 [self.person1 class] 的代码

```
self.person1 = [[Person alloc] init];
self.person1.age = 1;

self.person2 = [[Person alloc] init];
self.person2.age = 2;
```

```
NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew | NSKeyValueObserving0
[self.person1 addObserver:self forKeyPath:@"age" options:options context:@"年龄"];

NSLog(@"%@@", [self.person1 class]);
```

ObjectiveC

- 执行后, 打印结果如下:

```
// 打印结果:
```

```
Person
```

- 打印结果是 `Person` , 而 `person1` 的isa指向是 `NSKVONotifying_Person` , 这说明在 `NSKVONotifying_Person` 中, 对 `class` 进行了重写
- 现在推断 `NSKVONotifying_Person` 中的 `class` 方法实现如下:

```
- (Class)class {
    return [Person class];
}
```

2、关于 `dealloc` 和 `_isKVOA` 方法

- 由于没办法看到 `NSKVONotifying_Person` 中具体的源码, 所以只能模糊推断
- 因为 `NSKVONotifying_Person` 是为了观察 `age` 属性, 才创建出来的, 所以在 `dealloc` 中会进行一些结尾操作
- 而 `_isKVOA` 方法, 则推断为:

```
- (BOOL)_isKVOA {
    return YES
}
```

八、面试题

1、iOS用什么方式实现对一个对象的KVO?(KVO的本质是什么)

- 利用RuntimeAPI动态生成一个子类, 并且让instance对象的isa指向这个全新的子类
- 当修改instance对象的属性时, 会调用Foundation的`_NSSet*ValueAndNotify`函数
 - `willChangeValueForKey:`
 - 父类原来的setter方法
 - `didChangeValueForKey:`
 - 内部会触发监听器 `Observer` 的监听方法
(`observeValueForKeyPath:ofObject:change:context:`)

2、如果直接修改对象的成员变量, 是否会触发监听器的(`observeValueForKeyPath:ofObject:change:context:`)方法?

- 将 `Person` 类的 `_age` 暴露出来

```
@interface Person : NSObject
{
    @public
    int _age;
}
@property (nonatomic, assign) int age;
@end
```

- 将 `ViewController` 中的 `touchesBegan:withEvent:` 方法修改如下

ObjectiveC

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    self.person1->_age = 21;
}
```

- 运行程序后, 发现并没有触发 `observeValueForKeyPath:ofObject:change:context:` 方法
- 所以, 直接修改对象的成员变量, 而不调用 `set` 方法, 将不会触发观察者的 `observeValueForKeyPath:ofObject:change:context:` 方法

3、如何手动触发KVO?

- 已知实例对象被观察的属性, 在调用 `set` 方法进行修改时, 会触发 `_NSSet*ValueAndNotify` 函数
- 并触发 `willChangeValueForKey:` 和 `didChangeValueForKey:` 这两个方法, 所以我们可以手动添加这两个方法, 来触发KVO
- 现在已知直接修改 `成员变量` 时, 不会触发KVO, 那么就在修改 `成员变量` 的前后添加这两个方法

ObjectiveC

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    [self.person1 willChangeValueForKey:@"age"];
    self.person1->_age = 21;
    [self.person1 didChangeValueForKey:@"age"];
}
```

- 运行程序, 点击ViewController的view, 有如下打印:

```
<Person: 0x60000001c6c0> - age - {
    kind = 1;
    new = 21;
    old = 1;
```

```
} - 年龄
```

- 所以, 通过调用 `willChangeValueForKey:` 和 `didChangeValueForKey:` 方法, 就可以手动的调用KVO

注意:

`willChangeValueForKey:` 和 `didChangeValueForKey:` , 两个方法必须同时出现, 如果只有一个, 将不会触发KVO

ObjectiveC

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    self.person1->_age = 21;
    [self.person1 didChangeValueForKey:@"age"];
}
```

- 运行程序, 点击屏幕后, 没有任何打印

关注下面的标签, 发现更多相似文章

iOS

架构

逆向



冰凌天

Lv2 iOS

获得点赞 189 · 获得阅读 9,820

关注

安装掘金浏览器插件

打开新标签页发现好内容, 掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧!

评论

输入评论...

相关推荐

专栏_森宇_ · 1天前 · iOS

iOS 13 适配要点总结

 19

 2

专栏知识小集 · 3天前 · iOS

Xcode 11 的那些新东西

 45



专栏零壹技术栈 · 4天前 · Redis / 架构

一篇文章让你明白你多级缓存的分层架构


 114

 2

专栏tigerAndBull酱 · 1天前 · iOS

TABAnimated骨架屏缓存策略

 3



专栏_sx_ · 9天前 · 架构 / 前端

透过现象看本质: 常见的前端架构风格和案例

 268

 35

荐 · 知识小集 · 6天前 · iOS

专栏

iOS 13 正式发布，来看看有哪些 API 变动

 66

 7

专栏QiShare · 2天前 · iOS

iOS13 DarkMode适配（一）

 22

 16

专栏iOS桃子 · 1天前 · iOS

iOS | 面试知识整理 - 网络相关 (七)

 6



专栏知识小集 · 2天前 · iOS

今年的 Swift，有哪些新的东西呢？

 5



Xcode11新变化：SceneDelegate



5

