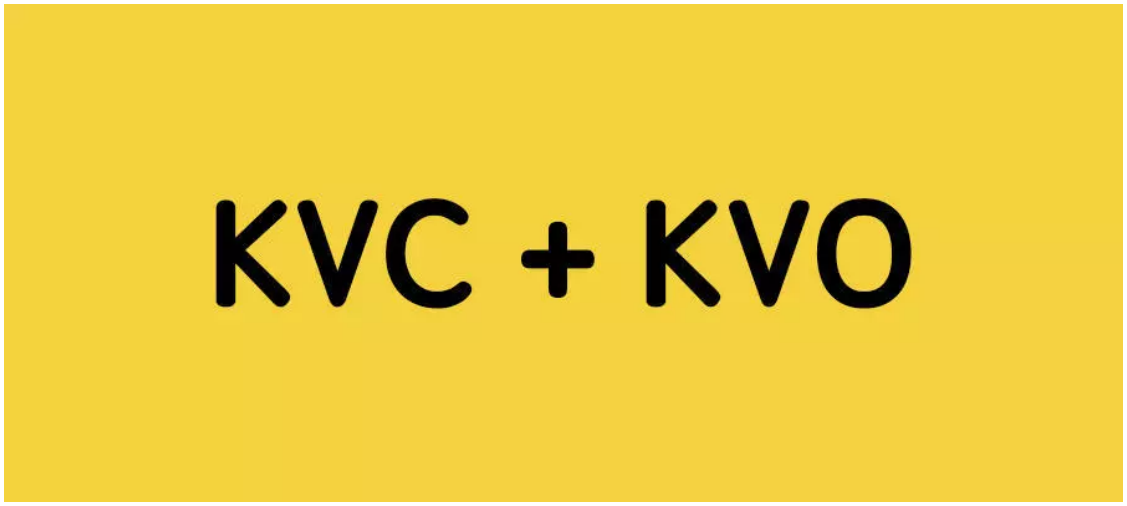

KVC

KVC定义

> KVC（Key-value coding）键值编码，就是指iOS的开发中，可以允许开发者通过Key名直接访问对象的属性，或者给对象的属性赋值。而不需要调用明确的存取方法。这样就可以在运行时动态地访问和修改对象的属性。而不是在编译时确定，这也是iOS开发中的黑魔法之一。很多高级的iOS开发技巧都是基于KVC实现的。

> 在实现了访问器方法的类中，使用点语法和KVC访问对象其实差别不大，二者可以任意混用。但是没有访问起方法的类中，点语法无法使用，这时KVC就有优势了。

KVC的定义都是对NSObject的扩展来实现的，Objective-C中有个显式的NSKeyValueCoding类别名，所以对于所有继承了NSObject的类型，都能使用KVC(一些纯Swift类和结构体是不支持KVC的，因为没有继承NSObject)，下面是KVC最为重要的四个方法：

```
- (nullable id)valueForKey:(NSString *)key;
//直接通过Key来取值

- (void)setValue:(nullable id)value forKey:(NSString *)key;
//通过Key来设值

- (nullable id)valueForKeyPath:(NSString *)keyPath;
//通过KeyPath来取值
```

```
- (void)setValue:(nullable id)value forKeyPath:(NSString *)keyPath;
//通过KeyPath来设值
```

NSKeyValueCoding类别中其他的一些方法：

```
+ (BOOL)accessInstanceVariablesDirectly;
//默认返回YES，表示如果没有找到Set<Key>方法的话，会按照_key，_iskey，key，iskey的顺序搜索成员，设置成NO就不这样搜索

- (BOOL)validateValue:(inout id __nullable * __nonnull)ioValue forKey:(NSString *)inKey error:(out NSError **)outError;
//KVC提供属性值正确性验证的API，它可以用来检查set的值是否正确、为不正确的值做一个替换值或者拒绝设置新值并返回错误原因。

- (NSMutableArray *)mutableArrayValueForKey:(NSString *)key;
//这是集合操作的API，里面还有一系列这样的API，如果属性是一个NSMutableArray，那么可以用这个方法来返回。

- (nullable id)valueForUndefinedKey:(NSString *)key;
//如果Key不存在，且没有KVC无法搜索到任何和Key有关的字段或者属性，则会调用这个方法，默认是抛出异常。

- (void)setValue:(nullable id)value forUndefinedKey:(NSString *)key;
//和上一个方法一样，但这个方法是设值。

- (void)setNilValueForKey:(NSString *)key;
//如果你在SetValue方法时面给Value传nil，则会调用这个方法

- (NSDictionary<NSString *, id> *)dictionaryWithValuesForKeys:(NSArray<NSString *> *)keys;
//输入一组key,返回该组key对应的Value，再转成字典返回，用于将Model转到字典。
```

同时苹果对一些容器类比如NSArray或者NSSet等，KVC有着特殊的实现。

有序集合对应方法如下：

```
-countOf<Key>//必须实现，对应于NSArray的基本方法count:2  -objectIn<Key>AtIndex:
```

```
-<key>AtIndexes://这两个必须实现一个，对应于 NSArray 的方法
objectAtIndex: 和 objectsAtIndexes:

-get<Key>:range://不是必须实现的，但实现后可以提高性能，其对应于 NSArray 方
法 getObjects:range:

-insertObject:in<Key>AtIndex:

-insert<Key>:atIndexes://两个必须实现一个，类似于 NSMutableArray 的方法
insertObject:atIndex: 和 insertObjects:atIndexes:

-removeObjectFrom<Key>AtIndex:

-remove<Key>AtIndexes://两个必须实现一个，类似于 NSMutableArray 的方法
removeObjectAtIndex: 和 removeObjectsAtIndexes:

-replaceObjectIn<Key>AtIndex:withObject:

-replace<Key>AtIndexes:with<Key>://可选的，如果在此类操作上有性能问题，就
需要考虑实现之
```

无序集合对应方法如下：

```
-countOf<Key>//必须实现，对应于NSArray的基本方法count:

-objectIn<Key>AtIndex:

-<key>AtIndexes://这两个必须实现一个，对应于 NSArray 的方法
objectAtIndex: 和 objectsAtIndexes:

-get<Key>:range://不是必须实现的，但实现后可以提高性能，其对应于 NSArray 方
法 getObjects:range:

-insertObject:in<Key>AtIndex:

-insert<Key>:atIndexes://两个必须实现一个，类似于 NSMutableArray 的方法
insertObject:atIndex: 和 insertObjects:atIndexes:

-removeObjectFrom<Key>AtIndex:

-remove<Key>AtIndexes://两个必须实现一个，类似于 NSMutableArray 的方法
removeObjectAtIndex: 和 removeObjectsAtIndexes:

-replaceObjectIn<Key>AtIndex:withObject:
```

```
-replace<Key>AtIndexes:with<Key>://这两个都是可选的，如果在此类操作上有性能问题，就需要考虑实现之
```

通过以下几个方面讲解KVC相关的技术概念以及使用：

- KVC设值
- KVC取值
- KVC使用keyPath
- KVC处理异常
- KVC处理数值和结构体类型属性
- KVC键值验证（Key-Value Validation）
- KVC处理集合
- KVC处理字典

KVC相关技术概念

KVC设值

KVC要设值，那么就要对象中对应的key，KVC在内部是按什么样的顺序来寻找key的。当调用setValue：属性值 forKey：@”name“的代码时，底层的执行机制如下：

- 程序优先调用set:属性值方法，代码通过setter方法完成设置。注意，这里的是指成员变量名，首字母大小写要符合KVC的命名规则，下同
- 如果没有找到setName：方法，KVC机制会检查+(BOOL)accessInstanceVariablesDirectly方法有没有返回YES，默认该方法会返回YES，如果你重写了该方法让其返回NO的话，那么在这一步KVC会执行setValue：forUndefinedKey：方法，不过一般开发者不会这么做。所以KVC机制会搜索该类里面有没有名为_的成员变量，无论该变量是在类接口处定义，还是在类实现处定义，也无论用了什么样的访问修饰符，只在存在以_命名的变量，KVC都可以对该成员变量赋值。
- 如果该类即没有set：方法，也没有_成员变量，KVC机制会搜索_is的成员变量。
- 和上面一样，如果该类即没有set：方法，也没有_和_is成员变量，KVC机制再会继续搜索和is的成员变量。再给它们赋值。

- 如果上面列出的方法或者成员变量都不存在，系统将会执行该对象的setValue：forUndefinedKey：方法，默认是抛出异常。

简单来说就是`如果没有找到Set<Key>方法的话，会按照_key，_iskey，key，iskey的顺序搜索成员并进行赋值操作`。

如果开发者想让这个类禁用KVC里，那么重写+(BOOL)accessInstanceVariablesDirectly方法让其返回NO即可，这样的话如果KVC没有找到set:属性名时，会直接用setValue：forUndefinedKey：方法。

下面看例子：

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

@interface Test: NSObject {
    NSString *_name;
}

@end

@implementation Test

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC赋值name
        [obj setValue:@"xiaoming" forKey:@"name"];
        //通过KVC取值name打印
        NSLog(@"obj的名字是%@", [obj valueForKey:@"name"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 15:36:52.354405+0800 KVCKVO[35231:6116188] obj的名字是xiaoming

可以看到通过`- (void)setValue:(nullable id)value forKey:(NSString`

`*)key;` 和 `- (nullable id)valueForKey:(NSString *)key;` 成功设置和取出obj对象的name值。

再看一下设置accessInstanceVariablesDirectly为NO的效果：

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

@interface Test: NSObject {
    NSString *_name;
}

@end

@implementation Test

+ (BOOL)accessInstanceVariablesDirectly {
    return NO;
}

- (id)valueForUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@",key);
    return nil;
}

- (void)setValue:(id)value forUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@", key);
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC赋值name
        [obj setValue:@"xiaoming" forKey:@"name"];
        //通过KVC取值name打印
        NSLog(@"obj的名字是%@", [obj valueForKey:@"name"]);

    }
    return 0;
}
```

> 打印结果: 2018-05-05 15:45:22.399021+0800 KVCKVO[35290:6145826] 出现异常，该key不存在name 2018-05-05 15:45:22.399546+0800 KVCKVO[35290:6145826] 出现异常，该key不存在name 2018-05-05 15:45:22.399577+0800 KVCKVO[35290:6145826] obj的名字是(null)

可以看到accessInstanceVariablesDirectly为NO的时候KVC只会查询setter和getter这一层，下面寻找key的相关变量执行就会停止，直接报错。

设置accessInstanceVariablesDirectly为YES，再修改_name为_isName，看看执行是否成功。

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

@interface Test: NSObject {
    NSString *_isName;
}

@end

@implementation Test

+ (BOOL)accessInstanceVariablesDirectly {
    return YES;
}

- (id)valueForUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@",key);
    return nil;
}

- (void)setValue:(id)value forUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@", key);
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC赋值name
        [obj setValue:@"xiaoming" forKey:@"name"];
        //通过KVC取值name打印
```

```
        NSLog(@"obj的名字是%@", [obj valueForKey:@"name"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 15:49:53.444350+0800 KVCKVO[35303:6157671] obj的名字是xiaoming

从打印可以看到设置accessInstanceVariablesDirectly为YES，KVC会继续按照顺序查找，并成功设值和取值了。

KVC取值

当调用valueForKey：@”name“的代码时，KVC对key的搜索方式不同于setValue：属性值 forKey：@”name“，其搜索方式如下：

- 首先按get,,is的顺序方法查找getter方法，找到的话会直接调用。如果是BOOL或者Int等值类型， 会将其包装成一个NSNumber对象。
- 如果上面的getter没有找到，KVC则会查找countOf,objectInAtIndex或AtIndexes格式的方法。如果countOf方法和另外两个方法中的一个被找到，那么就会返回一个可以响应NSArray所有方法的代理集合(它是NSKeyValueArray，是NSArray的子类)，调用这个代理集合的方法，或者说给这个代理集合发送属于NSArray的方法，就会以countOf,objectInAtIndex或AtIndexes这几个方法组合的形式调用。还有一个可选的get:range:方法。所以你想重新定义KVC的一些功能，你可以添加这些方法，需要注意的是你的方法名要符合KVC的标准命名方法，包括方法签名。
- 如果上面的方法没有找到，那么会同时查找countOf，enumeratorOf,memberOf格式的方法。如果这三个方法都找到，那么就返回一个可以响应NSSet所的方法的代理集合，和上面一样，给这个代理集合发NSSet的消息，就会以countOf，enumeratorOf,memberOf组合的形式调用。
- 如果还没有找到，再检查类方法+ (BOOL)accessInstanceVariablesDirectly,如果返回YES(默认行为)，那么和先前的设值一样，会按_,_is,,is的顺序搜索成员变量名，这里不推荐这么做，因为这样直接访问实例变量破坏了封装性，使代码更脆弱。如果重写了类方法+ (BOOL)accessInstanceVariablesDirectly返回NO的话，那么会直接调用valueForUndefinedKey:方法，默认是抛出异常。

给Test类添加getAge方法，例如如下：

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

@interface Test: NSObject {
}

@end

@implementation Test

- (NSUInteger)getAge {
    return 10;
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC取值age打印
        NSLog(@"obj的年龄是%@", [obj valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:00:04.207857+0800 KVCKVO[35324:6188613] obj的年龄是10

可以看到`[obj valueForKey:@"age"]`，找到了getAge方法，并且取到了值。

下面把getAge改成age，例子如下：

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

@interface Test: NSObject {
}

@end
```

```
@implementation Test

- (NSUInteger)age {
    return 10;
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC取值age打印
        NSLog(@"obj的年龄是%@", [obj valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:02:27.270954+0800 KVCKVO[35337:6195086] obj的年龄是10

可以看到`[obj valueForKey:@"age"]`，找到了age方法，并且取到了值。

下面把getAge改成isAge，例子如下：

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

@interface Test: NSObject {
}

@end

@implementation Test

- (NSUInteger)isAge {
    return 10;
}

@end
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        Test *obj = [[Test alloc] init];
        //通过KVC取值age打印
        NSLog(@"obj的年龄是%@", [obj valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:03:56.234338+0800 KVCKVO[35345:6201242] obj的年龄是10

可以看到`[obj valueForKey:@"age"]`，找到了isAge方法，并且取到了值。

上面的代码说明了说明了KVC在调用`valueforKey:@"age"`时搜索key的机制。

KVC使用keyPath

在开发过程中，一个类的成员变量有可能是自定义类或其他的复杂数据类型，你可以先用KVC获取该属性，然后再次用KVC来获取这个自定义类的属性， 但这样是比较繁琐的，对此，KVC提供了一个解决方案，那就是键路径keyPath。顾名思义，就是按照路径寻找key。

```
- (nullable id)valueForKeyPath:(NSString *)keyPath;
//通过KeyPath来取值
- (void)setValue:(nullable id)value forKeyPath:(NSString *)keyPath;
//通过KeyPath来设值
```

用代码实现如下：

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

@interface Test1: NSObject {
    NSString *_name;
}
@end
```

```
@implementation Test1
@end

@interface Test: NSObject {
    Test1 *_test1;
}

@end

@implementation Test
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //Test生成对象
        Test *test = [[Test alloc] init];
        //Test1生成对象
        Test1 *test1 = [[Test1 alloc] init];
        //通过KVC设值test的"test1"
        [test setValue:test1 forKey:@"test1"];
        //通过KVC设值test的"test1的name"
        [test setValue:@"xiaoming" forKeyPath:@"test1.name"];
        //通过KVC取值age打印
        NSLog(@"test的\"test1的name\"是%@", [test
valueForKeyPath:@"test1.name"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:19:02.613394+0800 KVCKVO[35436:6239788] test的"test1的name"是xiaoming

从打印结果来看我们成功的通过keyPath设置了test1的值。 KVC对于keyPath是搜索机制第一步就是分离key，用小数点.来分割key，然后再像普通key一样按照先前介绍的顺序搜索下去。

KVC处理异常

KVC中最常见的异常就是不小心使用了错误的key，或者在设值中不小心传递了nil的值，KVC中有专门的方法来处理这些异常。

KVC处理nil异常

通常情况下，KVC不允许你要在调用setValue：属性值 forKey：(或者keyPath)时对非对象传递一个nil的值。很简单，因为值类型是不能为nil的。如果你不小心传了，KVC会调用setNilValueForKey:方法。这个方法默认是抛出异常，所以一般而言最好还是重写这个方法。

代码实现如下：

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

@interface Test: NSObject {
    NSUInteger age;
}

@end

@implementation Test

- (void)setNilValueForKey:(NSString *)key {
    NSLog(@"不能将%@设成nil", key);
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //Test生成对象
        Test *test = [[Test alloc] init];
        //通过KVC设值test的age
        [test setValue:nil forKey:@"age"];
        //通过KVC取值age打印
        NSLog(@"test的年龄是%@", [test valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:24:30.302134+0800 KVCKVO[35470:6258307] 不能将age设成nil 2018-05-05 16:24:30.302738+0800 KVCKVO[35470:6258307] test的年龄是0

KVC处理UndefinedKey异常

通常情况下，KVC不允许你要在调用setValue：属性值 forKey：(或者keyPath)时对不存在的key进行操作。 不然，会报错forUndefinedKey发生崩溃，重写forUndefinedKey方法避免崩溃。

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

@interface Test: NSObject {
}

@end

@implementation Test

- (id)valueForUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@",key);
    return nil;
}

- (void)setValue:(id)value forUndefinedKey:(NSString *)key {
    NSLog(@"出现异常，该key不存在%@", key);
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //Test生成对象
        Test *test = [[Test alloc] init];
        //通过KVC设值test的age
        [test setValue:@10 forKey:@"age"];
        //通过KVC取值age打印
        NSLog(@"test的年龄是%@", [test valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:30:18.564680+0800 KVCKVO[35487:6277523] 出现异常，该key不存在age 2018-05-05 16:30:18.565190+0800 KVCKVO[35487:6277523] 出现异常，该key不存在age 2018-05-05

> 16:30:18.565216+0800 KVCKVO[35487:6277523] test的年龄是(null)

KVC处理数值和结构体类型属性

不是每一个方法都返回对象，但是valueForKey：总是返回一个id对象，如果原本的变量类型是值类型或者结构体，返回值会封装成NSNumber或者NSValue对象。 这两个类会处理从数字，布尔值到指针和结构体任何类型。然后开以者需要手动转换成原来的类型。 尽管valueForKey：会自动将值类型封装成对象，但是setValue：forKey：却不行。你必须手动将值类型转换成NSNumber或者NSValue类型，才能传递过去。 因为传递进去和取出来的都是id类型，所以需要开发者自己担保类型的正确性，运行时Objective-C在发送消息的会检查类型，如果错误会直接抛出异常。

举个例子，Person类有个NSInteger类型的age属性，如下：

```
//  Person.m
#import "Person.h"

@interface Person ()

@property (nonatomic,assign) NSInteger age;

@end

@implementation Person

@end
```

修改值

我们通过KVC技术使用如下方式设置age属性的值：

```
[person setValue:[NSNumber numberWithInteger:5] forKey:@"age"];
```

我们赋给age的是一个NSNumber对象，KVC会自动的将NSNumber对象转换成NSInteger对象，然后再调用相应的访问器方法设置age的值。

获取值

同样，以如下方式获取age属性值：

```
[person valueForKey:@"age"];
```

这时，会以NSNumber的形式返回age的值。

```
//  ViewController.m
#import "ViewController.h"
#import "Person.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    Person *person = [[Person alloc]init];
    [person setValue:[NSNumber numberWithInteger:5] forKey:@"age"];
    NSLog(@"age=%@",[person valueForKey:@"age"]);

}

@end
```

> 打印结果： 2017-01-16 16:31:55.709 Test[28586:2294177] age=5

需要注意的是我们不能直接将一个数值通过KVC赋值的，我们需要把数据转为NSNumber和NSValue类型传入，那到底哪些类型数据要用NSNumber封装哪些类型数据要用NSValue封装呢？看下面这些方法的参数类型就知道了：

可以使用NSNumber的数据类型有：

```
+ (NSNumber*)numberWithChar:(char)value;
+ (NSNumber*)numberWithUnsignedChar:(unsignedchar)value;
+ (NSNumber*)numberWithShort:(short)value;
+ (NSNumber*)numberWithUnsignedShort:(unsignedshort)value;
+ (NSNumber*)numberWithInt:(int)value;
+ (NSNumber*)numberWithUnsignedInt:(unsignedint)value;
+ (NSNumber*)numberWithLong:(long)value;
```

```
+ (NSNumber*)numberWithUnsignedLong:(unsignedlong)value;
+ (NSNumber*)numberWithLongLong:(longlong)value;
+ (NSNumber*)numberWithUnsignedLongLong:(unsignedlonglong)value;
+ (NSNumber*)numberWithFloat:(float)value;
+ (NSNumber*)numberWithDouble:(double)value;
+ (NSNumber*)numberWithBool:(BOOL)value;
+ (NSNumber*)numberWithInteger:
(NSInteger)valueNS_AVAILABLE(10_5,2_0);
+ (NSNumber*)numberWithUnsignedInteger:
(NSUInteger)valueNS_AVAILABLE(10_5,2_0);
```

就是一些常见的数值型数据。

可以使用NSValue的数据类型有：

```
+ (NSValue*)valueWithCGPoint:(CGPoint)point;
+ (NSValue*)valueWithCGSize:(CGSize)size;
+ (NSValue*)valueWithCGRect:(CGRect)rect;
+ (NSValue*)valueWithCGAffineTransform:
(CGAffineTransform)transform;
+ (NSValue*)valueWithUIEdgeInsets:(UIEdgeInsets)insets;
+ (NSValue*)valueWithUIOffset:
(UIOffset)insetsNS_AVAILABLE_IOS(5_0);
```

NSValue主要用于处理结构体型的数据，它本身提供了如上集中结构的支持。任何结构体都是可以转化成NSValue对象的，包括其它自定义的结构体。

KVC键值验证（Key-Value Validation）

KVC提供了验证Key对应的Value是否可用的方法：

```
- (BOOL)validateValue:(inoutid*)ioValue forKey:(NSString*)inKey
error:(outNSError**)outError;
```

该方法默认的实现是调用一个如下格式的方法：

```
- (BOOL)validate<Key>:error:
```

例如：

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

@interface Test: NSObject {
    NSUInteger _age;
}

@end

@implementation Test

- (BOOL)validateValue:(inout id  _Nullable __autoreleasing
*)ioValue forKey:(NSString *)inKey error:(out NSError * _Nullable
__autoreleasing *)outError {
    NSNumber *age = *ioValue;
    if (age.integerValue == 10) {
        return NO;
    }

    return YES;
}

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //Test生成对象
        Test *test = [[Test alloc] init];
        //通过KVC设值test的age
        NSNumber *age = @10;
        NSError* error;
        NSString *key = @"age";
        BOOL isValid = [test validateValue:&age forKey:key
error:&error];
        if (isValid) {
            NSLog(@"键值匹配");
            [test setValue:age forKey:key];
        }
        else {
            NSLog(@"键值不匹配");
        }
        //通过KVC取值age打印
```

```
        NSLog(@"test的年龄是%@", [test valueForKey:@"age"]);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 16:59:06.111671+0800 KVCKVO[35777:6329982] 键值不匹配 2018-05-05 16:59:06.112215+0800 KVCKVO[35777:6329982] test的年龄是0

> 这样就给了我们一次纠错的机会。需要指出的是，KVC是不会自动调用键值验证方法的，就是说我们如果想要键值验证则需要手动验证。但是有些技术，比如CoreData会自动调用。

KVC处理集合

KVC同时还提供了很复杂的函数，主要有下面这些：

简单集合运算符

简单集合运算符共有@avg， @count ， @max ， @min ，@sum5种，都表示什么直接看下面例子就明白了， 目前还不支持自定义。

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

@interface Book : NSObject
@property (nonatomic, copy)  NSString* name;
@property (nonatomic, assign)  CGFloat price;
@end

@implementation Book
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Book *book1 = [Book new];
        book1.name = @"The Great Gastby";
        book1.price = 10;
        Book *book2 = [Book new];
        book2.name = @"Time History";
        book2.price = 20;
        Book *book3 = [Book new];
```

```
        book3.name = @"Wrong Hole";
        book3.price = 30;

        Book *book4 = [Book new];
        book4.name = @"Wrong Hole";
        book4.price = 40;

        NSArray* arrBooks = @[book1,book2,book3,book4];
        NSNumber* sum = [arrBooks valueForKeyPath:@"@sum.price"];
        NSLog(@"sum:%f",sum.floatValue);
        NSNumber* avg = [arrBooks valueForKeyPath:@"@avg.price"];
        NSLog(@"avg:%f",avg.floatValue);
        NSNumber* count = [arrBooks valueForKeyPath:@"@count"];
        NSLog(@"count:%f",count.floatValue);
        NSNumber* min = [arrBooks valueForKeyPath:@"@min.price"];
        NSLog(@"min:%f",min.floatValue);
        NSNumber* max = [arrBooks valueForKeyPath:@"@max.price"];
        NSLog(@"max:%f",max.floatValue);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 17:04:50.674243+0800 KVCKVO[35785:6351239] sum:100.000000 2018-05-05 17:04:50.675007+0800 KVCKVO[35785:6351239] avg:25.000000 2018-05-05 17:04:50.675081+0800 KVCKVO[35785:6351239] count:4.000000 2018-05-05 17:04:50.675146+0800 KVCKVO[35785:6351239] min:10.000000 2018-05-05 17:04:50.675204+0800 KVCKVO[35785:6351239] max:40.000000

对象运算符

比集合运算符稍微复杂，能以数组的方式返回指定的内容，一共有两种：

- @distinctUnionOfObjects
- @unionOfObjects

它们的返回值都是NSArray，区别是前者返回的元素都是唯一的，是去重以后的结果；后者返回的元素是全集。

例如：

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

@interface Book : NSObject
@property (nonatomic, copy)  NSString* name;
@property (nonatomic, assign)  CGFloat price;
@end

@implementation Book
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Book *book1 = [Book new];
        book1.name = @"The Great Gastby";
        book1.price = 40;
        Book *book2 = [Book new];
        book2.name = @"Time History";
        book2.price = 20;
        Book *book3 = [Book new];
        book3.name = @"Wrong Hole";
        book3.price = 30;

        Book *book4 = [Book new];
        book4.name = @"Wrong Hole";
        book4.price = 10;

        NSArray* arrBooks = @[book1,book2,book3,book4];

        NSLog(@"distinctUnionOfObjects");
        NSArray* arrDistinct = [arrBooks
valueForKeyPath:@"@distinctUnionOfObjects.price"];
        for (NSNumber *price in arrDistinct) {
            NSLog(@"%f",price.floatValue);
        }
        NSLog(@"unionOfObjects");
        NSArray* arrUnion = [arrBooks
valueForKeyPath:@"@unionOfObjects.price"];
        for (NSNumber *price in arrUnion) {
            NSLog(@"%f",price.floatValue);
        }

    }
    return 0;
}
```

> 打印结果： 2018-05-05 17:06:21.832401+0800 KVCKVO[35798:6358293] distinctUnionOfObjects 2018-05-05 17:06:21.833079+0800 KVCKVO[35798:6358293] 10.000000 2018-05-05 17:06:21.833112+0800 KVCKVO[35798:6358293] 20.000000 2018-05-05 17:06:21.833130+0800 KVCKVO[35798:6358293] 30.000000 2018-05-05 17:06:21.833146+0800 KVCKVO[35798:6358293] 40.000000 2018-05-05 17:06:21.833165+0800 KVCKVO[35798:6358293] unionOfObjects 2018-05-05 17:06:21.833297+0800 KVCKVO[35798:6358293] 40.000000 2018-05-05 17:06:21.833347+0800 KVCKVO[35798:6358293] 20.000000 2018-05-05 17:06:21.833371+0800 KVCKVO[35798:6358293] 30.000000 2018-05-05 17:06:21.833388+0800 KVCKVO[35798:6358293] 10.000000

KVC处理字典

当对NSDictionary对象使用KVC时，valueForKey:的表现行为和objectForKey:一样。所以使用valueForKeyPath:用来访问多层嵌套的字典是比较方便的。

KVC里面还有两个关于NSDictionary的方法:

```
- (NSDictionary<NSString *, id> *)dictionaryWithValuesForKeys:
(NSArray<NSString *> *)keys;
- (void)setValuesForKeysWithDictionary:(NSDictionary<NSString *,
id> *)keyedValues;
```

dictionaryWithValuesForKeys:是指输入一组key，返回这组key对应的属性，再组成一个字典。 setValuesForKeysWithDictionary是用来修改Model中对应key的属性。下面直接用代码会更直观一点:

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

@interface Address : NSObject

@end

@interface Address()

@property (nonatomic, copy)NSString* country;
@property (nonatomic, copy)NSString* province;
@property (nonatomic, copy)NSString* city;
@property (nonatomic, copy)NSString* district;
```

```
@end

@implementation Address

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        //模型转字典
        Address* add = [Address new];
        add.country = @"China";
        add.province = @"Guang Dong";
        add.city = @"Shen Zhen";
        add.district = @"Nan Shan";
        NSArray* arr =
@[@"country",@"province",@"city",@"district"];
        NSDictionary* dict = [add dictionaryWithValuesForKeys:arr];
//把对应key所有的属性全部取出来
        NSLog(@"%@",dict);

        //字典转模型
        NSDictionary* modifyDict =
@{@"country":@"USA",@"province":@"california",@"city":@"Los
angle"};
        [add setValuesForKeysWithDictionary:modifyDict];
//用key Value来修改Model的属性
        NSLog(@"country:%@  province:%@
city:%@",add.country,add.province,add.city);

    }
    return 0;
}
```

> 打印结果： 2018-05-05 17:08:48.824653+0800 KVCKVO[35807:6368235] { city = "Shen Zhen"; country = China; district = "Nan Shan"; province = "Guang Dong"; } 2018-05-05 17:08:48.825075+0800 KVCKVO[35807:6368235] country:USA province:california city:Los angle

打印出来的结果完全符合预期。

KVC使用

KVC在iOS开发中是绝不可少的利器，这种基于运行时的编程方式极大地提高了灵活性，简化了代码，甚至实现很多难以想像的功能，KVC也是许多iOS开发黑魔法的基础。 下面列举iOS开发中KVC的使用场景.

动态地取值和设值

利用KVC动态的取值和设值是最基本的用途了。

用KVC来访问和修改私有变量

对于类里的私有属性，Objective-C是无法直接访问的，但是KVC是可以的。

Model和字典转换

这是KVC强大作用的又一次体现，KVC和Objc的runtime组合可以很容易的实现Model和字典的转换。

修改一些控件的内部属性

这也是iOS开发中必不可少的小技巧。众所周知很多UI控件都由很多内部UI控件组合而成的，但是Apple度没有提供这访问这些控件的API，这样我们就无法正常地访问和修改这些控件的样式。 而KVC在大多数情况可下可以解决这个问题。最常用的就是个性化UITextField中的placeHolderText了。

操作集合

Apple对KVC的valueForKey:方法作了一些特殊的实现，比如说NSArray和NSSet这样的容器类就实现了这些方法。所以可以用KVC很方便地操作集合。

用KVC实现高阶消息传递

当对容器类使用KVC时，valueForKey:将会被传递给容器中的每一个对象，而不是容器本身进行操作。结果会被添加进返回的容器中，这样，开发者可以很方便的操作集合来返回另一个集合。

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        NSArray* arrStr = @[@"english",@"franch",@"chinese"];
```

```
        NSArray* arrCapStr = [arrStr
valueForKey:@"capitalizedString"];
        for (NSString* str  in arrCapStr) {
            NSLog(@"%@",str);
        }
        NSArray* arrCapStrLength = [arrStr
valueForKeyPath:@"capitalizedString.length"];
        for (NSNumber* length  in arrCapStrLength) {
            NSLog(@"%ld",(long)length.integerValue);
        }

    }
    return 0;
}
```

> 打印结果： 2018-05-05 17:16:21.975983+0800 KVCKVO[35824:6395514] English 2018-05-05 17:16:21.976296+0800 KVCKVO[35824:6395514] Franch 2018-05-05 17:16:21.976312+0800 KVCKVO[35824:6395514] Chinese 2018-05-05 17:16:21.976508+0800 KVCKVO[35824:6395514] 7 2018-05-05 17:16:21.976533+0800 KVCKVO[35824:6395514] 6 2018-05-05 17:16:21.976550+0800 KVCKVO[35824:6395514] 7

方法capitalizedString被传递到NSArray中的每一项，这样，NSArray的每一员都会执行capitalizedString并返回一个包含结果的新的NSArray。 从打印结果可以看出，所有String都成功以转成了大写。 同样如果要执行多个方法也可以用valueForKeyPath:方法。它先会对每一个成员调用 capitalizedString方法，然后再调用length，因为lenth方法返回是一个数字，所以返回结果以NSNumber的形式保存在新数组里。

实现KVO

KVO是基于KVC实现的，下面讲一下KVO的概念和实现。

KVO

KVO定义

KVO 即 Key-Value Observing，翻译成键值观察。它是一种观察者模式的衍生。其基本思想是，对目标对象的某属性添加观察，当该属性发生变化时，通过触发观察者对象实现的KVO接口方法，来自动的通知观察者。

> 观察者模式是什么 一个目标对象管理所有依赖于它的观察者对象，并在它自身的状态改变时主动通知观察者对象。这个主动通知通常是通过调用各观察者对象所提供的接口方法来实现的。观察者模式较完美地将目标对象与观察者对象解耦。

简单来说KVO可以通过监听key，来获得value的变化，用来在对象之间监听状态变化。KVO的定义都是对NSObject的扩展来实现的，Objective-C中有个显式的NSKeyValueObserving类别名，所以对于所有继承了NSObject的类型，都能使用KVO(一些纯Swift类和结构体是不支持KVC的，因为没有继承NSObject)。

KVO使用

注册与解除注册

如果我们已经有了包含可供键值观察属性的类，那么就可以通过在该类的对象（被观察对象）上调用名为 NSKeyValueObserverRegistration 的 category 方法将观察者对象与被观察者对象注册与解除注册：

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

```
observer:观察者，也就是KVO通知的订阅者。订阅着必须实现
observeValueForKeyPath:ofObject:change:context:方法
keyPath：描述将要观察的属性，相对于被观察者。
options：KVO的一些属性配置；有四个选项。
context: 上下文，这个会传递到订阅着的函数中，用来区分消息，所以应当是不同的。
```

options所包括的内容

```
NSKeyValueObservingOptionNew：change字典包括改变后的值
NSKeyValueObservingOptionOld:change字典包括改变前的值
NSKeyValueObservingOptionInitial:注册后立刻触发KVO通知
NSKeyValueObservingOptionPrior:值改变前是否也要通知（这个key决定了是否在改变前改变后通知两次）
```

这两个方法的定义在 Foundation/NSKeyValueObserving.h 中，NSObject，NSArray，NSSet均实现了以上方法，因此我们不仅可以观察普通对象，还可以观察数组或结合类对象。在该头文件中，我们还可以看到 NSObject 还实现了 NSKeyValueObserverNotification 的 category 方法（更多类似方法，请查看该头文件NSKeyValueObserving.h）：

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

这两个方法在手动实现键值观察时会用到。注意在不用的时候，不要忘记解除注册，否则会导致内存泄露。

处理变更通知

每当监听的keyPath发生变化了，就会在这个函数中回调。

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

在这里，change 这个字典保存了变更信息，具体是哪些信息取决于注册时的 NSKeyValueObservingOptions。

手动KVO(禁用KVO)

KVO的实现，是对注册的keyPath中自动实现了两个函数，在setter中，自动调用。

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

可能有时候，我们要实现手动的KVO，或者__我们实现的类库不希望被KVO__。 这时候需要关闭自动生成KVO通知，然后手动的调用，手动通知的好处就是，可以灵

活加上自己想要的判断条件。下面看个例子如下：

```
@interface Target : NSObject
{
    int age;
}

// for manual KVO - age
- (int) age;
- (void) setAge:(int)theAge;

@end

@implementation Target

- (id) init
{
    self = [super init];
    if (nil != self)
    {
        age = 10;
    }

    return self;
}

// for manual KVO - age
- (int) age
{
    return age;
}

- (void) setAge:(int)theAge
{
    [self willChangeValueForKey:@"age"];
    age = theAge;
    [self didChangeValueForKey:@"age"];
}

+ (BOOL) automaticallyNotifiesObserversForKey:(NSString *)key {
    if ([key isEqualToString:@"age"]) {
        return NO;
    }

    return [super automaticallyNotifiesObserversForKey:key];
}
```

```
@end
```

首先，需要手动实现属性的 setter 方法，并在设置操作的前后分别调用 willChangeValueForKey: 和 didChangeValueForKey方法，这两个方法用于通知系统该 key 的属性值即将和已经变更了； 其次，要实现类方法 automaticallyNotifiesObserversForKey，并在其中设置对该 key 不自动发送通知（返回 NO 即可）。这里要注意，对其它非手动实现的 key，要转交给 super 来处理。 如果需要__禁用该类KVO__的话直接automaticallyNotifiesObserversForKey返回NO，实现属性的 setter 方法，不进行调用willChangeValueForKey: 和 didChangeValueForKey方法。

键值观察依赖键

有时候一个属性的值依赖于另一对象中的一个或多个属性，如果这些属性中任一属性的值发生变更，被依赖的属性值也应当为其变更进行标记。因此，object 引入了依赖键。

观察依赖键

观察依赖键的方式与前面描述的一样，下面先在 Observer 的 observeValueForKeyPath:ofObject:change:context: 中添加处理变更通知的代码：

```
#import "TargetWrapper.h"

- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context
{
    if ([keyPath isEqualToString:@"age"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString
stringWithCString:object_getClassName(classInfo)

encoding:NSUTF8StringEncoding];
        NSLog(@" >> class: %@, Age changed", className);

        NSLog(@" old age is %@", [change objectForKey:@"old"]);
        NSLog(@" new age is %@", [change objectForKey:@"new"]);
    }
    else if ([keyPath isEqualToString:@"information"])
```

```
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString
stringWithCString:object_getClassName(classInfo)

encoding:NSUTF8StringEncoding];
        NSLog(@" >> class: %@, Information changed", className);
        NSLog(@" old information is %@", [change
objectForKey:@"old"]);
        NSLog(@" new information is %@", [change
objectForKey:@"new"]);
    }
    else
    {
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

实现依赖键

在这里，观察的是 TargetWrapper 类的 information 属性，该属性是依赖于 Target 类的 age 和 grade 属性。为此，我在 Target 中添加了 grade 属性：

```
@interface Target : NSObject
@property (nonatomic, readwrite) int grade;
@property (nonatomic, readwrite) int age;
@end

@implementation Target
@synthesize age; // for automatic KVO - age
@synthesize grade;
@end
```

下面来看看 TragetWrapper 中的依赖键属性是如何实现的。

```
@class Target;

@interface TargetWrapper : NSObject
```

```
{
@private
    Target * _target;
}

@property(nonatomic, assign) NSString * information;
@property(nonatomic, retain) Target * target;

-(id) init:(Target *)aTarget;

@end

#import "TargetWrapper.h"
#import "Target.h"

@implementation TargetWrapper

@synthesize target = _target;

-(id) init:(Target *)aTarget
{
    self = [super init];
    if (nil != self) {
        _target = [aTarget retain];
    }

    return self;
}

-(void) dealloc
{
    self.target = nil;
    [super dealloc];
}

- (NSString *)information
{
    return [[[NSString alloc] initWithFormat:@"%d#%d", [_target
grade], [_target age]] autorelease];
}

- (void)setInformation:(NSString *)theInformation
{
    NSArray * array = [theInformation
componentsSeparatedByString:@"#"];
    [_target setGrade:[[array objectAtIndex:0] intValue]];
    [_target setAge:[[array objectAtIndex:1] intValue]];
}
```

```
+ (NSSet *)keyPathsForValuesAffectingInformation
{
    NSSet * keyPaths = [NSSet setWithObjects:@"target.age",
@"target.grade", nil];
    return keyPaths;
}

//+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
//{
//    NSSet * keyPaths = [super
keyPathsForValuesAffectingValueForKey:key];
//    NSArray * moreKeyPaths = nil;
//
//    if ([key isEqualToString:@"information"])
//    {
//        moreKeyPaths = [NSArray arrayWithObjects:@"target.age",
@"target.grade", nil];
//    }
//
//    if (moreKeyPaths)
//    {
//        keyPaths = [keyPaths
setByAddingObjectsFromArray:moreKeyPaths];
//    }
//
//    return keyPaths;
//}

@end
```

首先，要手动实现属性 information 的 setter/getter 方法，在其中使用 Target 的属性来完成其 setter 和 getter。

其次，要实现 keyPathsForValuesAffectingInformation 或 keyPathsForValuesAffectingValueForKey: 方法来告诉系统 information 属性依赖于哪些其他属性，这两个方法都返回一个key-path 的集合。在这里要多说几句，如果选择实现 keyPathsForValuesAffectingValueForKey，要先获取 super 返回的结果 set，然后判断 key 是不是目标 key，如果是就将依赖属性的 key-path 结合追加到 super 返回的结果 set 中，否则直接返回 super的结果。

在这里，information 属性依赖于 target 的 age 和 grade 属性，target 的 age/grade 属性任一发生变化，information 的观察者都会得到通知。

```
Observer * observer = [[[Observer alloc] init] autorelease];
Target * target = [[[Target alloc] init] autorelease];

TargetWrapper * wrapper = [[[TargetWrapper alloc] init:target]
autorelease];
[wrapper addObserver:observer
          forKeyPath:@"information"
             options:NSKeyValueObservingOptionNew |
NSKeyValueObservingOptionOld
             context:[TargetWrapper class]];

[target setAge:30];
[target setGrade:1];
[wrapper removeObserver:observer forKeyPath:@"information"];
```

> 打印结果： class: TargetWrapper, Information changed old information is 0#10 new information is 0#30 class: TargetWrapper, Information changed old information is 0#30 new information is 1#30

KVO和线程

一个需要注意的地方是，KVO 行为是同步的，并且发生与所观察的值发生变化的同样的线程上。没有队列或者 Run-loop 的处理。手动或者自动调用 -didChange... 会触发 KVO 通知。

所以，当我们试图从其他线程改变属性值的时候我们应当十分小心，除非能确定所有的观察者都用线程安全的方法处理 KVO 通知。通常来说，我们不推荐把 KVO 和多线程混起来。如果我们要用多个队列和线程，我们不应该在它们互相之间用 KVO。

KVO 是同步运行的这个特性非常强大，只要我们在单一线程上面运行（比如主队列 main queue），KVO 会保证下列两种情况的发生：

首先，如果我们调用一个支持 KVO 的 setter 方法，如下所示：

```
self.exchangeRate = 2.345;
```

KVO 能保证所有 exchangeRate 的观察者在 setter 方法返回前被通知到。

其次，如果某个键被观察的时候附上了 NSKeyValueObservingOptionPrior 选项，直到 -observe... 被调用之前， exchangeRate 的 accessor 方法都会返回同样的值。

KVO实现

> KVO 是通过 isa-swizzling 实现的。 基本的流程就是编译器自动为被观察对象创造一个派生类，并将被观察对象的isa 指向这个派生类。如果用户注册了对某此目标对象的某一个属性的观察，那么此派生类会重写这个方法，并在其中添加进行通知的代码。Objective-C 在发送消息的时候，会通过 isa 指针找到当前对象所属的类对象。而类对象中保存着当前对象的实例方法，因此在向此对象发送消息时候，实际上是发送到了派生类对象的方法。由于编译器对派生类的方法进行了 override，并添加了通知代码，因此会向注册的对象发送通知。注意派生类只重写注册了观察者的属性方法。

苹果官方文档的说明如下：

> **Key-Value Observing Implementation Details**
>
> 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.

KVO的实现依赖于Runtime的强大动态能力。

即当一个类型为 ObjectA 的对象，被添加了观察后，系统会生成一个 NSKVONotifying_ObjectA 类，并将对象的isa指针指向新的类，也就是说这个对象的类型发生了变化。这个类相比较于ObjectA，会重写以下几个方法。

重写setter

在 setter 中，会添加以下两个方法的调用。

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

然后在 `didChangeValueForKey:` 中，去调用：

```
- (void)observeValueForKeyPath:(nullable NSString *)keyPath
                      ofObject:(nullable id)object
                        change:(nullable
NSDictionary<NSKeyValueChangeKey, id> *)change
                       context:(nullable void *)context;
```

包含了新值和旧值的通知。

于是实现了属性值修改的通知。因为 KVO 的原理是修改 setter 方法，因此使用 KVO 必须调用 setter 。若直接访问属性对象则没有效果。

重写class

当修改了isa指向后，class的返回值不会变，但isa的值则发生改变。

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

@interface ObjectA: NSObject

@property (nonatomic) NSInteger age;

@end

@implementation ObjectA
@end

@interface ObjectB: NSObject
@end

@implementation ObjectB
```

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

@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...

        //生成对象
        ObjectA *objA = [[ObjectA alloc] init];
        ObjectB *objB = [[ObjectB alloc] init];

        // 添加Observer之后
        [objA addObserver:objB forKeyPath:@"age"
options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
context:nil];

        // 输出ObjectA
        NSLog(@"%@", [objA class]);
        // 输出NSKVONotifying_ObjectA（object_getClass方法返回isa指向）
        NSLog(@"%@", object_getClass(objA));

    }
    return 0;
}
```

> 打印结果： 2018-05-06 22:47:05.538899+0800
> KVCKVO[38474:13343992] ObjectA 2018-05-06 22:47:05.539242+0800
> KVCKVO[38474:13343992] NSKVONotifying_ObjectA

重写dealloc

系统重写 dealloc 方法来释放资源。

重写_isKVOA

这个私有方法是用来标示该类是一个 KVO 机制声称的类。

如何证明被观察的类被重写了以上方法

参考用代码探讨 KVC/KVO 的实现原理这篇文章，通过代码一步步分析，从断点截图来看，可以很好证明以上被重写的方法。