Key-Value Coding

key-valueCoding 是一种间接访问对象属性的途径。它使用字符串来辨识属性，而不是通过存取方法或者实例变量来访问。key-valueCoding需要的方法定义在**NSKeyValueCoding.h**,默认由**NSObject**实现。

和基本类型、结构体作为属性一样，key-valueCoding也支持对象作为属性。非对象类型的参数和返回类型会自动进行装箱和拆箱操作。

key-value coding的API提供了通过key查询对象和设置对象值的通用方法。

在你设计应用中的对象的时候你应该定义一个包含你模型所有property key的集合 ， 同时实现相符合的存取方法。

使用Key_Value Coding让代码更简洁

通过使标识符和你想要显示的属性键相同，你可以明显的简化你的代码。

未使用key-value coding

```
-(id)tableView:(NSTableView *)tableview
objectValueForTableColumn:(id)column row:(NSInteger)row
{

    ChildObject *child = [childrenArray objectAtIndex:row];

    if ([[column identifier] isEqualToString:@"name"]) {
        return [child name];
    }
    if ([[column identifier] isEqualToString:@"age"]) {
        return [child age];
    }
    if ([[column identifier] isEqualToString:@"favoriteColor"]) {
        return [child favoriteColor];
    }
}
```

使用key-value coding

```
- (id)tableView:(NSTableView *)tableview
      objectValueForTableColumn:(id)column row:(NSInteger)row {

    ChildObject *child = [childrenArray objectAtIndex:row];
    return [child valueForKey:[column identifier]];
}
```

Terminology

key-value coding可以用来存取三种不同的对形象类型：属性、一对一的关系、一对多的关系，属性可以是以上的三种类型。

- attributes:包含NSString、BOOL等基础类型，值对象，例如NSNumber还有其他不可变的对象类型(NSColor等)
- 一对一关系：属性拥有自己的属性，自己的属性的改变不会影响自身，比如UIView实例的父view就是一个一对一的关系
- 一对多的关系: 由对象的集合类型构成，例如NSArray、NSSet。key-value coding 允许使用元素为自定义类的集合。

Key-Value Coding的原理

键和键路径

key

一个键为一个字符串明确的标志了一个对象的某个属性。典型的就是一个键对应接受对象的存取方法或者实例变量的名字。**key必须是ASCII编码，使用小写字母开头，不能包含空格。**(例如：payee, openingBalance, transactions and amount)

key path

keypath是一个由点和多个key值组成的字符串。其中的第一个key属于接受者，后面的每一个key属于前面的keypath取出的值。例如，keypath address.street会从接收者中取出key为address的值，然后street属性会确定属于address对象

使用Key-Value Coding获取属性的值

方法 `valueForKey:` 返回接收者中key对应的值。如果存取器或者实例变量与key对应， `valueForKey:` 的接收者会向自己发送一个 `valueForUndefinedKey:` 消息， `valueForUndefinedKey:` 的默认实现会抛出一个NSUndefinedKeyException,子类可以重写这个方

法。

同样的 `valueForKeyPath:` 返回消息接收者中keypath对应的值，如果keypath中任意一个对应key的对象不符合key-value coding，接收者会向自身发送 `valueForUndefinedKey:` 消息。

方法 `dictionaryWithValuesForKeys:` 会查找消息接收者中所有的和数组中key对应的value，返回的NSDictionary包含了所有的key对应的value和key本身。

> 集合对象，NSArray、NSSet和NSDictionary，不能包含nil值，不过可以用NSNull代替。dictionaryWithValuesForKeys:和setValuesForKeysWithDictionary:会自动在nil和NSNull两者中转换。

如果一个keypath中包含一个key对应一对多的属性，同时这个key不是keypath中的最后一个key，那么返回值会是一个集合类型，包含了这个key对应的值。例如：accounts.transactions.payee ,accounts.transactions的value是一个NSArray，那么accounts.transactions.payee将返回一个数组，包含所有transactions中每一个对象对应的payee的值。

使用Key-Value Coding设置属性的值

`setValue:forKey:` 使用传入的值设置消息接收者中对应key的值。默认的 `setValue:forKey:` 会自动将NSValue对象解包为数值类型或者结构体并分配给对应的属性。

如果key不存在receiver会被发送一个 `setValue:forUndefinedKey:` 消息，默认的 `setValue:forUndefinedKey:` 会抛出一个NSUndefinedKeyException，子类可以重写这个方法操作这个请求。 `setValue:forKeyPath:` 相似，可以handle keypath。

`setValuesForKeysWithDictionary:` 使用传入的字典设置消息接收者的值，使用传入的dictionary中的key去识别接收者中的属性。默认的实现会对应每一个键值对调用 `setValue:forKey:` ,同时调用中会将value为nil的值转换成NSNull。

当设置一个非对象类型的值为nil的时候，接收者会给自己发送 `setNilValueForKey:` 消息, `setNilValueForKey:` 默认实现会抛出NSInvalidArgumentException。有特殊需求的话可以重写 `setValuesForKeysWithDictionary:` ，替换value,然后调用 `setValue:forKey:`

点语法与Key-Value Coding

kvc和点语法是正交的，使用kvc不一定要使用点语法，也可以在kvc之外使用点语法。在kvc中点语法用来在keypath中界定节点。如果你使用点语法访问了一个属性，那么你就调用了接收者的标准存取方法。

example:

```
@interface MyClass
@property NSString *stringProperty;
@property NSInteger integerProperty;
@property MyClass *linkedInstance;
@end
```

在KVC中你可以像下面这样使用

```
MyClass *myInstance = [[MyClass alloc] init];
NSString *string = [myInstance valueForKey:@"stringProperty"];
[myInstance setValue:@2 forKey:@"integerProperty"];
```

点语法

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;
```

与上面的结果相同

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:@2 forKeyPath:@"linkedInstance.integerProperty"];
```

KVC存取方法

为了让KVC能找出对应的存取方法
供`valueForKey:`、`setValue:forKey:`、`mutableArrayValueForKey:`、`mutableSetValueForKey:`调用，你需要实现kvc的存取方法。格式为`-set<Key>/-<key>`（key是占位符，代表属性的名称）。 example:属性name，存取方法应该为`-setName:`和`-name.`

存取方法的通用模式

通常的获取方法是-,返回值为对象、数值或者数据结构，对BOOL类型的属性来说-is也是支持的。

@property BOOL hidden; `- (BOOL)hidden { return ...; }`
`- (BOOL)isHidden { return ...; }`这两个getter方法都可是合法的。

为了让属性和一对一的关系支持`setValue:forKey:`方法，`-set<key>:`必须实现：`- (void)setHidden:(BOOL)flag { return; }`

如果属性是非对象类型，必须实现对应的设置nil的方法，kvc中`setNilValueForKey:`方法在设置属性的

值为nil时调用.我们可以在这个方法中去设置某个key默认值，或者处理没有存取方法的key。

```
@interface CustomModel : NSObject
@property (nonatomic,assign)int count;
@end

//调用
CustomModel* model = [[CustomModel alloc] init];
[model setValue:nil forKey:@"count"];
```

会出现如下错误:

```
2015-10-21 13:35:03.978 KVCDemo[5781:167289] *** Terminating app due to uncaught
exception 'NSInvalidArgumentException', reason: '[<CustomModel 0x7f9203ccdd80>
setNilValueForKey]: could not set nil as the value for the key count.'
```

这种情况，我们应该重写类的 `-setNilValueForKey:` 方法

```
- (void)setNilValueForKey:(NSString *)theKey {
    if ([theKey isEqualToString:@"count"]) {
        [self setValue:@YES forKey:@"count"];
    }
    else {
        [super setNilValueForKey:theKey];
    }
}
```

集合存取器(一对多的属性）

虽然可以使用 `-<key>` 和 `-set<Key>:` 的方式来存取collection，但是通常你是操作返回的collection对象。但是如果需要通过 KVC 来操作collection中的内容的话就需要实现 collection额外的存取方法 `mutableArrayValueForKey:` 、 `mutableSetValueForKey:` 。

集合的存取方法有两类，一种是NSArray为代表的有序的集合存取方法，一种是NSSet代表的无序集合存取方法。

有序存取器

Getter

有序集合的读取

- `-countOf<Key>` ,必须实现，和NSArray的count方法相似
- `-objectIn<Key>AtIndex: or -<key>AtIndexes:` ,这两个方法必须有一个实现，类似NSArray

的 `objectAtIndex:` 和 `objectsAtIndexes:` 。

- `-get<Key>:range:` ,可选的实现,实现这个方法会增加性能。？？？？

> `-countOf<Key>` 返回一对多关系中对象的个数，比如有一个属性类型为 `@property (nonatomic,strong)NSArray* list;` -countOfList返回值为list中对象的个数。

example:

```
- (NSUInteger)countOfEmployees {
    return [self.employees count];
}
```

> `-objectIn<Key>AtIndex:` 返回一对多关系中index位置中的对象。 `-<key>AtIndexes:` 返回接收者中NSIndexSet标明的位置中的一个array对象。这两个方法中实现任意一个就可以了。当然也可以两个都实现。 example:

```
- (id)objectInEmployeesAtIndex:(NSUInteger)index {
    return [employees objectAtIndex:index];
}
- (NSArray *)employeesAtIndexes:(NSIndexSet *)indexes {
    return [self.employees objectsAtIndexes:indexes];
}
```

如果想要性能提升，你也可以实现 `-get<Key>:range:` 获取该对象range标明的范围内对应所有对象，并保存在传入的一个缓冲池中。

example:

```
- (void)getEmployees:(Employee * __unsafe_unretained *)buffer range:(NSRange)inRan
ge {
    // Return the objects in the specified range in the provided buffer.
    // For example, if the employees were stored in an underlying NSArray
    [self.employees getObjects:buffer range:inRange];
}
```

可变的有序集合存取

实现可变的有序存取方法 `-mutableArrayValueForKey:` 可以更容易的去管理一个可变的集合。实现了这些方法你的对象中的相对应的property就能支持KVO。

> 相比直接返回一个可变集合对象，实现可变存取器更加有优势，可变存储器在修改属性的数据方面更加有效率。

要让一个可变有序集合支持KVC必须实现以下方法：

- `-insertObject:in<Key>AtIndex:` 、 `-insert<Key>:atIndexes:` 这两个方法至少要有一个实现，类似NSMutalbeArray的 `insertObject:atIndex:` 、 `insertObjects:atIndexes:` 方法。

```
- (void)insertObject:(Employee *)employee inEmployeesAtIndex:(NSUInteger)index {
    [self.employees insertObject:employee atIndex:index];
    return;
}
- (void)insertEmployees:(NSArray *)employeeArray atIndexes:(NSIndexSet *)indexes
{
    [self.employees insertObjects:employeeArray atIndexes:indexes];
return; }
```

- `-removeObjectFrom<Key>AtIndex:` 、 `-remove<Key>AtIndexes:.` 这两个方法至少要有一个实现，类似NSMutalbeArray的 `removeObjectAtIndex:` 、 `removeObjectsAtIndexes:` 方法。

```
- (void)removeObjectFromEmployeesAtIndex:(NSUInteger)index {
    [self.employees removeObjectAtIndex:index];
}
- (void)removeEmployeesAtIndexes:(NSIndexSet *)indexes {
    [self.employees removeObjectsAtIndexes:indexes];
}
```

- `-replaceObjectIn<Key>AtIndex:withObject:` 、 `-replace<Key>AtIndexes:with<Key>:` 可选的实现，如果对性能要求高，则实现更好。

```
- (void)replaceObjectInEmployeesAtIndex:(NSUInteger)index
                             withObject:(id)anObject {
    [self.employees replaceObjectAtIndex:index withObject:anObject];
}
- (void)replaceEmployeesAtIndexes:(NSIndexSet *)indexes
                    withEmployees:(NSArray *)employeeArray {
    [self.employees replaceObjectsAtIndexes:indexes withObjects:employeeArray];
}
```

无序存取器

一般不实现无序集合的getter方法，而是直接使用NSSet或其子类的实例对象作为model来操作属性。在kvc中如果没有找到可变集合的存取方法，会直接获取该集合。只有在你操作的对象为自定义的集合对象的时候，无需集合的存取方法才有必要实现。

- `countOf<Key>:` 必须实现
- `-enumeratorOf<Key>:` 必须实现

- `-memberOf<Key>:` 必须实现

无序集合存取

- `-add<Key>Object:` or `-add<Key>:` 至少实现一个
- `-remove<Key>Object:` or `-remove<Key>:` 至少实现一个
- `-intersect<Key>:` 可选的，实现能提升性能

Key-Value Validation

kvc提供了API验证属性的值，kvc的认可基础给类提供了机会去接收/替换/拒绝一个设置给属性的值，并返回错误原因。

命名: `-validate<Key>:error:.`

```
-(BOOL)validateName:(id *)ioValue error:(NSError * __autoreleasing *)outError {
    // Implementation specific code.
    return ...;
}
```

实现验证方法

Validation methods需要传入两个参数，一个是需要被验证的值，一个是NSError对象用于返回错误信息。方法有三种可能：

- value是合法的，不改变value和error，return YES
- 值是不合法的，一个新的值不能被正确的创建和返回,此时，方法会返回NO，然后设置错误信息到NSError对象
- 值是不合法的，但是一个新的值正确的创建和返回,此时，方法会返回YES，这种情况不修改NSError对象。必须返回一个新创建的对象并返回,即便新创建的值是可能改变的。不能返回修改过后的传入的对象

```
-(BOOL)validateName:(id *)ioValue error:(NSError * __autoreleasing *)outError
{
    // The name must not be nil, and must be at least two characters long.
    if ((*ioValue == nil) || ([(NSString *)*ioValue length] < 2)) {
        if (outError != NULL) {
            NSString *errorString = NSLocalizedString(
                                                      @"A Person's name must be at
 least two characters long",
                                                      @"validation: Person, too sh
ort name error");
            NSDictionary *userInfoDict = @{ NSLocalizedDescriptionKey : errorStrin
g};
                                                *outError = [[NSError alloc] initWithD
omain:PERSON_ERROR_DOMAIN
                                                code:PERSON_INVALID_NAME_CODE
                                                userInfo:userInfoDict];
        }
        return NO;
    }
    return YES;
}
```

> 当方法返回NO的时候必须首先检查outError参数是否为NULL，如果outError不空，应该讲outError设置为一个有效的NSError对象。

触发验证方法

你可以通过调用 `-validateValue:forKey:error:` 正确的触发验证方法。 `validateValue:forKey:error:` 默认的实现会再接收者的Class中去寻找方法名和 `validate<Key>:error:` 匹配的方法，如果找到了这个方法，会触发这个方法并返回结果，如果没找到 `validateValue:forKey:error:` 会返回YES,确认这个设置的值;

属性的 `-set<Key>:` 不能调用验证方法 `validate<Key>:error:`

自动验证

一半来说，kvc不会自动触发验证方法，需要应用自己去触发验证方法。但是有些情况下，可以通过某些技术让验证方法自动触发，例如Core Data中当managed object context保存的时候，验证方法就会自动的触发，Cocoa bindings allow you to specify that validation should occur automatically。

纯量的验证

验证方法期望传入的参数是一个object，如果返回值是一个非对象类型的属性，那么值会被装箱在一个NSValue或者NSNumber的对象中并返回。

```
-(BOOL)validateXnum:(id*)inValue error:(out NSError * _Nullable __autoreleasing *)
outError
{
    if(*inValue == nil)
        return YES;

    if([*inValue floatValue] < 1){

        NSNumber* num = [NSNumber numberWithFloat:1];
        if(num){
            *inValue = num;
            return YES;
        }
        if(outError != NULL){
            NSString* errorStr = @"errooooo";
            NSDictionary* userinfo = @{NSLocalizedDescriptionKey:errorStr};
            NSError* err = [[NSError alloc] initWithDomain:@"PERSON_ERROR_DOMAIN"
code:1 userInfo:userinfo];
            *outError = err;
        }
        return NO;
    }
    return YES;
}
```

确保支持KVC

为了支持kvc，你的property必须要实现 `valueForKey:` 、 `setValue:forKey:` 。

单一的属性和一对一的关系

- 需要实现方法 `-<key>` 或 `-is<key>` ,或者拥有一个实例变量 `key` 或 `_key` ，KVC支持首字母大写的key。
- 如果property是可变的还需要实现 `-set<Key>`
- `在-set<Key>` 中不应该执行验证方法
- 如果你的类适用验证方法那么你应该实现 `-validate<Key>:error:`

有序一对多的关系（不可变）

- 实现 `-<key>` 返回一个array或者拥有一个名为 `key` 或 `_key` 的数组实例变量。
- 实现 `-countOf<Key>` 必须实现, `-objectIn<Key>AtIndex:` 、 `-<key>AtIndexes:` 两者至少有一个被实现。

- 可以选择实现 `-get<Key>:range:` 以提高性能. ##有序一对多的关系（可变）
- `-insertObject:in<Key>AtIndex:` 、 `-insert<Key>:atIndexes:` 两者至少有一个被实现。
- 实现 `-removeObjectFrom<Key>AtIndex` 、 `-remove<Key>AtIndexes:` 两者至少有一个被实现。
- 实现 `-replaceObjectIn<Key>AtIndex:withObject:`
  或 `-replace<Key>AtIndexes:with<Key>:` 以提高性能

无序一对多的关系（不可变）

- Implement a method named `-<key>` that returns a set.
- Or have a set instance variable named `<key>` or `_<key>` .
- Or implement the methods `-countOf<Key>` , `-enumeratorOf<Key>` , and `-memberOf<Key>:`

无序一对多的关系（可变）

- Implement one or both of the methods `-add<Key>Object:` or `-add<Key>:` .
- Implement one or both of the methods `-remove<Key>Object:` or `-remove<Key>:` .
- Optionally, you can also implement `-intersect<Key>:` and `-set<Key>:` to improve performance.

对纯量和结构体的支持

KVC通过自动的将纯量和数据结构装箱和拆箱为NSNumber和NSValue来提供对纯量和数据结构的支持。

Non-Object Values

`-valueForKey:` 和 `-setValue:forKey:` 默认的实现会自动的将非对象类型转换成对象类型

`-valueForKey:` 会查找对应key的存取方法或者实例变量，它会检查返回值的类型，如果这个值不是一个对象，那么他会使用返回值创建一个NSValue或NSNumber并返回。example： `numberWithBool:`

相似是 `-setValue:forKey` 会根据存取方法或者实例变量来确定数据类型，如果key对应的值不是一个object，这个值会 `-<type>Value` 方法转为对应的类型。比如设置一个BOOL类型的属性的值，传入的值会调用 `-boolValue` 转换为一个BOOL类型的值。

处理nil值

当调用 `-setValue:forKey` ,为一个非对象类型的属性设置一个nil值的时候会出现其他问题。当设置空值的时候，消息的接受者会被发送一个 `-setNilValueForKey:` 消息， `-setNilValueForKey:` 默认的

实现会抛出一个**NSInvalidArgumentException**异常，子类可以重写这个方法去处理这种情况。

> 如果消息接收者重写了NSObject的 `unableToSetNilForKey:` 方法， `unableToSetNilForKey:` 会替代 `setNilValueForKey:` 被调用，

```
- (void)setNilValueForKey:(NSString *)theKey
{
    if ([theKey isEqualToString:@"age"]) {
        [self setValue:[NSNumber numberWithFloat:0.0] forKey:@”age”];
    } else
        [super setNilValueForKey:theKey];
}
```

装箱和拆箱结构体