Key-Value Observing(KVO)

——Cooci

1:Introduction to Key-Value Observing Programming(KVO的介绍)

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

Important: In order to understand key-value observing, you must first understand key-value coding.

2:At a Glance (KVO的大致过程)

Key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects. It is particularly useful for communication between model and controller layers in an application. (In OS X, the controller layer binding technology relies heavily on key-value observing.) A controller object typically observes properties of model objects, and a view object observes properties of model objects through a controller. In addition, however, a model object may observe other model objects (usually to determine when a dependent value changes) or even itself (again to determine when a dependent value changes).

You can observe properties including simple attributes, to-one relationships, and to-many relationships. Observers of to-many relationships are informed of the type of change made—as well as which objects are involved in the change.

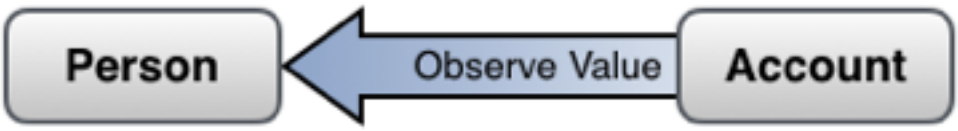
A simple example illustrates how KVO can be useful in your application. Suppose a Person object interacts with an Account object, representing the person's savings account at a bank. An instance of Person may need to be aware of when certain aspects of the Account instance change, such as the balance, or the interest rate.

If these attributes are public properties of Account, the Person could periodically poll the Account to discover changes, but this is of course inefficient, and often impractical. A better approach is to use KVO, which is akin to Person receiving an interrupt when a change occurs.

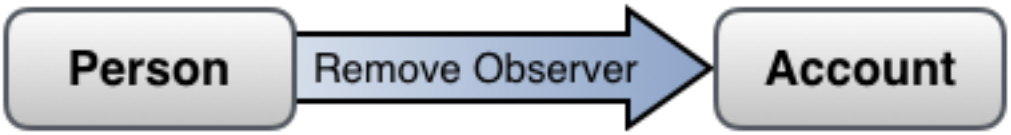
To use KVO, first you must ensure that the observed object, the Account in this case, is KVO compliant. Typically, if your objects inherit from NSObject and you create properties in the usual way, your objects and their properties will automatically be KVO Compliant. It is also possible to implement compliance manually. KVO Compliance describes the difference between automatic and manual key-value observing, and how to implement both.

Next, you must register your observer instance, the Person, with the observed instance, the Account. Person sends an addObserver:forKeyPath:options:context: message to the Account, once for each observed key path, naming itself as the observer

In order to receive change notifications from the Account, Person implements the observeValueForKeyPath:ofObject:change:context: method, required of all observers. The Account will send this message to the Person any time one of the registered key paths changes. The Person can then take appropriate action based upon the change notification



Finally, when it no longer wants notifications, and at the very least before it is deallocated, the Person instance must de-register by sending the message removeObserver:forKeyPath: to the Account.



Registering for Key-Value Observing describes the full lifecycle of registering for, receiving, and de-registering for key value observation notifications.

KVO's primary benefit is that you don't have to implement your own scheme to send notifications every time a property changes. Its well-defined infrastructure has framework-level support that makes it easy to adopt—typically you do not have to add any code to your project. In addition, the infrastructure is already full-featured, which makes it easy to support multiple observers for a single property, as well as dependent values.

Registering Dependent Keys explains how to specify that the value of a key is dependent on the value of another key.

Unlike notifications that use NSNotificationCenter, there is no central object that provides change notification for all observers. Instead, notifications are sent directly to the observing objects when changes are made. NSObject provides this base implementation of key-value observing, and you should rarely need to override these methods.

Key-Value Observing Implementation Details describes how key-value observing is implemented.

3:Registering for Key-Value Observing(KVO的注册KVO)

You must perform the following steps to enable an object to receive key-value observing notifications for a KVO-compliant property:

- Register the observer with the observed object using the method addObserver:forKeyPath:options:context:.

- Implement observeValueForKeyPath:ofObject:change:context: inside the observer to accept change notification messages.

- Unregister the observer using the method removeObserver:forKeyPath: when it no longer should receive messages. At a minimum, invoke this method before the observer is released from memory.

Important: Not all classes are KVO-compliant for all properties. You can ensure your own classes are KVO-compliant by following the steps described in KVO Compliance. Typically properties in Apple-supplied frameworks are only KVO-compliant if they are documented as such.

4:Registering as an Observer(KVO的注册观察者)

An observing object first registers itself with the observed object by sending an addObserver:forKeyPath:options:context: message, passing itself as the observer and the key path of the property to be observed. The observer additionally specifies an options parameter and a context pointer to manage aspects of the notifications.

Options(KVO的枚举)

The options parameter, specified as a bitwise OR of option constants, affects both the content of the change dictionary supplied in the notification, and the manner in which notifications are generated.

You opt to receive the value of the observed property from before the change by specifying option NSKeyValueObservingOptionOld. You request the new value of the property with option NSKeyValueObservingOptionNew. You receive both old and new values with the bitwise OR of these options.

You instruct the observed object to send an immediate change notification (before addObserver:forKeyPath:options:context: returns) with the option NSKeyValueObservingOptionInitial. You can use this additional, one-time notification to establish the initial value of a property in the observer.

You instruct the observed object to send a notification just prior to a property change (in addition to the usual notification just after the change) by including the option NSKeyValueObservingOptionPrior. The change dictionary represents a prechange notification by including the key **NSKeyValueChangeNotificationIsPriorKey** with the value of an NSNumber wrapping YES. That key is not otherwise present. You can use the prechange notification when the observer's own KVO compliance requires it to invoke one of the -willChange… methods for one of its properties that depends on an observed property. The usual post-change notification comes too late to invoke willChange… in time.

Context (KVO的上下文)

The context pointer in the addObserver:forKeyPath:options:context: message contains arbitrary data that will be passed back to the observer in the corresponding change notifications. You may specify NULL and rely entirely on the key path string to determine the origin of a change notification, but this approach may cause problems for an object whose superclass is also observing the same key path for different reasons.

A safer and more extensible approach is to use the context to ensure notifications you receive are destined for your observer and not a superclass.

The address of a uniquely named static variable within your class makes a good context. Contexts chosen in a similar manner in the super- or subclass will be unlikely to overlap. You may choose a single context for the entire class and rely on the key path string in the notification message to determine what changed. Alternatively, you may create a distinct context for each observed key path, which bypasses the need for string comparisons entirely, resulting in more efficient notification parsing. Listing 1 shows example contexts for the balance and interestRate properties chosen this way.

Listing 1 Creating context pointers

```
static void *PersonAccountBalanceContext =
&PersonAccountBalanceContext;
static void *PersonAccountInterestRateContext =
&PersonAccountInterestRateContext;
```

The example in Listing 2 demonstrates how a Person instance registers itself as an observer for an Account instance's balance and interestRate properties using the given context pointers.

Listing 2 Registering the inspector as an observer of the balance and interestRate properties

```
- (void)registerAsObserverForAccount:(Account*)account {

    [account addObserver:self
              forKeyPath:@"balance"
                 options:(NSKeyValueObservingOptionNew |
                          NSKeyValueObservingOptionOld)
                 context:PersonAccountBalanceContext];

    [account addObserver:self
              forKeyPath:@"interestRate"
                 options:(NSKeyValueObservingOptionNew |
                          NSKeyValueObservingOptionOld)
                 context:PersonAccountInterestRateContext];
}
```

Note: The key-value observing addObserver:forKeyPath:options:context: method does not maintain strong references to the observing object, the observed objects, or the context. You should ensure that you maintain strong references to the observing, and observed, objects, and the context as necessary

5:Receiving Notification of a Change (KVO的接收到通知变化)

When the value of an observed property of an object changes, the observer receives an observeValueForKeyPath:ofObject:change:context: message. All observers must implement this method.

The observing object provides the key path that triggered the notification, itself as the relevant object, a dictionary containing details about the change, and the context pointer that was provided when the observer was registered for this key path.

The change dictionary entry NSKeyValueChangeKindKey provides information about the type of change that occurred. If the value of the observed object has changed, the NSKeyValueChangeKindKey entry returns NSKeyValueChangeSetting. Depending on the options specified when the observer was registered, the NSKeyValueChangeOldKey and NSKeyValueChangeNewKey entries in the change dictionary contain the values of the property before, and after, the change. If the property is an object, the value is provided directly. If the property is a scalar or a C structure, the value is wrapped in an NSValue object (as with key-value coding).

If the observed property is a to-many relationship, the NSKeyValueChangeKindKey entry also indicates whether objects in the relationship were inserted, removed, or replaced by returning NSKeyValueChangeInsertion, NSKeyValueChangeRemoval, or NSKeyValueChangeReplacement, respectively.

The change dictionary entry for NSKeyValueChangeIndexesKey is an NSIndexSet object specifying the indexes in the relationship that changed. If NSKeyValueObservingOptionNew or NSKeyValueObservingOptionOld are specified as options when the observer is registered, the NSKeyValueChangeOldKey and NSKeyValueChangeNewKey entries in the change dictionary are arrays containing the values of the related objects before, and after, the change.
The example in Listing 3 shows the observeValueForKeyPath:ofObject:change:context: implementation for the Person observer that logs the old and new values of the properties balance and interestRate, as registered in Listing 2.

Listing 3 Implementation of observeValueForKeyPath:ofObject:change:context:

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

    if (context == PersonAccountBalanceContext) {
        // Do something with the balance…
    } else if (context == PersonAccountInterestRateContext) {
        // Do something with the interest rate…
```

```
    } else {
        // Any unrecognized context must belong to super
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

If you specified a NULL context when registering an observer, you compare the notification's key path against the key paths you are observing to determine what has changed. If you used a single context for all observed key paths, you first test that against the notification's context, and finding a match, use key path string comparisons to determine what specifically has changed. If you have provided a unique context for each key path, as demonstrated here, a series of simple pointer comparisons tells you simultaneously whether or not the notification is for this observer, and if so, what key path has changed.

In any case, the observer should always call the superclass's implementation of observeValueForKeyPath:ofObject:change:context: when it does not recognize the context (or in the simple case, any of the key paths), because this means a superclass has registered for notifications as well.
Note: If a notification propagates to the top of the class hierarchy, NSObject throws an NSInternalInconsistencyException because this is a programming error: a subclass failed to consume a notification for which it registered.

6:Removing an Object as an Observer(KVO的移除观察者)

You remove a key-value observer by sending the observed object a removeObserver:forKeyPath:context: message, specifying the observing object, the key path, and the context. The example in Listing 4 shows Person removing itself as an observer of balance and interestRate.

Listing 4 Removing the inspector as an observer of balance and interestRate

```
- (void)unregisterAsObserverForAccount:
(Account*)account {
```

```
    [account removeObserver:self
                 forKeyPath:@"balance"
                    context:PersonAccountBalanceContext];
    [account removeObserver:self
                 forKeyPath:@"interestRate"
                    context:PersonAccountInterestRateContext];
}
```

After receiving a removeObserver:forKeyPath:context: message, the observing object will no longer receive any observeValueForKeyPath:ofObject:change:context: messages for the specified key path and object.

When removing an observer, keep several points in mind:(移除观察者注意事项)

- Asking to be removed as an observer if not already registered as one results in an NSRangeException. You either call removeObserver:forKeyPath:context: exactly once for the corresponding call to addObserver:forKeyPath:options:context:, or if that is not feasible in your app, place the removeObserver:forKeyPath:context: call inside a try/catch block to process the potential exception.

- An observer does not automatically remove itself when deallocated. The observed object continues to send notifications, oblivious to the state of the observer. However, a change notification, like any other message, sent to a released object, triggers a memory access exception. You therefore ensure that observers remove themselves before disappearing from memory.

- The protocol offers no way to ask an object if it is an observer or being observed. Construct your code to avoid release related errors. A typical pattern is to register as an observer during the observer's initialization (for example in init or viewDidLoad) and unregister during deallocation (usually in dealloc), ensuring properly paired and ordered add and remove messages, and that the observer is unregistered before it is freed from memory.

7:KVO Compliance (KVO需要注意的几点)

In order to be considered KVO-compliant for a specific property, a class must ensure the following:

The class must be key-value coding compliant for the property, as specified in Ensuring KVC Compliance.

KVO supports the same data types as KVC, including Objective-C objects and the scalars and structures listed in Scalar and Structure Support.

The class emits KVO change notifications for the property.

Dependent keys are registered appropriately (see Registering Dependent Keys).

There are two techniques for ensuring the change notifications are emitted. Automatic support is provided by NSObject and is by default available for all properties of a class that are key-value coding compliant. Typically, if you follow standard Cocoa coding and naming conventions, you can use automatic change notifications—you don't have to write any additional code.

Manual change notification provides additional control over when notifications are emitted, and requires additional coding. You can control automatic notifications for properties of your subclass by implementing the class method automaticallyNotifiesObserversForKey:

8:Automatic Change Notification(自动观察通知)

NSObject provides a basic implementation of automatic key-value change notification. Automatic key-value change notification informs observers of changes made using key-value compliant accessors, as well as the key-value coding methods. Automatic notification is also supported by the collection proxy objects returned by, for example, mutableArrayValueForKey:.

The examples shown in Listing 1 result in any observers of the property name to be notified of the change.

Listing 1 Examples of method calls that cause KVO change notifications to be emitted

```
// Call the accessor method.
[account setName:@"Savings"];
// Use setValue:forKey:.
[account setValue:@"Savings" forKey:@"name"];
// Use a key path, where 'account' is a kvc-compliant
property of 'document'.
[document setValue:@"Savings" forKeyPath:@"account.name"];
// Use mutableArrayValueForKey: to retrieve a relationship
proxy object.
Transaction *newTransaction = <#Create a new transaction
for the account#>;
NSMutableArray *transactions = [account
mutableArrayValueForKey:@"transactions"];
[transactions addObject:newTransaction];
```

9:Manual Change Notification(手动观察通知)

In some cases, you may want control of the notification process, for example, to minimize triggering notifications that are unnecessary for application specific reasons, or to group a number of changes into a single notification. Manual change notification provides means to do this.

Manual and automatic notifications are not mutually exclusive. You are free to issue manual notifications in addition to the automatic ones already in place. More typically, you may want to completely take control of the notifications for a particular property. In this case, you override the NSObject implementation of automaticallyNotifiesObserversForKey:. For properties whose automatic notifications you want to preclude, the subclass implementation of automaticallyNotifiesObserversForKey: should return NO. A subclass implementation should invoke super for any unrecognized keys. The example in Listing 2 enables manual notification for the balance property, allowing the superclass to determine the notification for all other keys.

Listing 2 Example implementation of automaticallyNotifiesObserversForKey:

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString
*)theKey {

    BOOL automatic = NO;
    if ([theKey isEqualToString:@"balance"]) {
        automatic = NO;
    }else {
        automatic = [super
automaticallyNotifiesObserversForKey:theKey];
    }
    return automatic;
}
```

To implement manual observer notification, you invoke willChangeValueForKey: before changing the value, and didChangeValueForKey: after changing the value. The example in

Listing 3 implements manual notifications for the balance property.

Listing 3 Example accessor method implementing manual notification

```
- (void)setBalance:(double)theBalance {
    [self willChangeValueForKey:@"balance"];
    _balance = theBalance;
    [self didChangeValueForKey:@"balance"];
}
```

You can minimize sending unnecessary notifications by first checking if the value has changed. The example in Listing 4 tests the value of balance and only provides the notification if it has changed.

Listing 4 Testing the value for change before providing notification

```
- (void)setBalance:(double)theBalance {

    if (theBalance != _balance) {
        [self willChangeValueForKey:@"balance"];
        _balance = theBalance;
        [self didChangeValueForKey:@"balance"];
    }
}
```

If a single operation causes multiple keys to change you must nest the change notifications as shown in Listing 5.

Listing 5 Nesting change notifications for multiple keys

```
- (void)setBalance:(double)theBalance {

    [self willChangeValueForKey:@"balance"];
    [self willChangeValueForKey:@"itemChanged"];
    _balance = theBalance;
    _itemChanged = _itemChanged+1;
    [self didChangeValueForKey:@"itemChanged"];
    [self didChangeValueForKey:@"balance"];
}
```

In the case of an ordered to-many relationship, you must specify not only the key that changed, but also the type of change and the indexes of the objects involved. The type of change is an NSKeyValueChange that specifies NSKeyValueChangeInsertion, NSKeyValueChangeRemoval, or NSKeyValueChangeReplacement. The indexes of the affected objects are passed as an NSIndexSet object.

The code fragment in Listing 6 demonstrates how to wrap a deletion of objects in the to-many relationship transactions.

Listing 6 Implementation of manual observer notification in a to-many relationship

```
- (void)removeTransactionsAtIndexes:(NSIndexSet
*)indexes {

    [self willChange:NSKeyValueChangeRemoval
        valuesAtIndexes:indexes forKey:@"transactions"];
    // Remove the transaction objects at the specified
indexes.
    [self didChange:NSKeyValueChangeRemoval
        valuesAtIndexes:indexes forKey:@"transactions"];
}
```

10:Registering Dependent Keys (注册依赖keys)

There are many situations in which the value of one property depends on that of one or more other attributes in another object. If the value of one attribute changes, then the value of the derived property should also be flagged for change. How you ensure that key-value observing

notifications are posted for these dependent properties depends on the cardinality of the relationship.

11:To-One Relationships (单一关系处理)

To trigger notifications automatically for a to-one relationship you should either override keyPathsForValuesAffectingValueForKey: or implement a suitable method that follows the pattern it defines for registering dependent keys.

For example, the full name of a person is dependent on both the first and last names. A method that returns the full name could be written as follows:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"%@ %@",firstName,
lastName];
}
```

An application observing the fullName property must be notified when either the firstName or lastName properties change, as they affect the value of the property.

One solution is to override keyPathsForValuesAffectingValueForKey: specifying that the fullName property of a person is dependent on the lastName and firstName properties. Listing 1 shows an example implementation of such a dependency:

Listing 1 Example implementation of keyPathsForValuesAffectingValueForKey:

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:
(NSString *)key {

    NSSet *keyPaths = [super
keyPathsForValuesAffectingValueForKey:key];
    if ([key isEqualToString:@"fullName"]) {
        NSArray *affectingKeys = @[@"lastName", @"firstName"];
        keyPaths = [keyPaths
setByAddingObjectsFromArray:affectingKeys];
    }
    return keyPaths;
}
```

Your override should typically invoke super and return a set that includes any members in the set that result from doing that (so as not to interfere with overrides of this method in superclasses).

You can also achieve the same result by implementing a class method that follows the naming convention keyPathsForValuesAffecting<Key>, where <Key> is the name of the attribute (first letter capitalized) that is dependent on the values. Using this pattern the code in Listing 1 could be rewritten as a class method named keyPathsForValuesAffectingFullName as shown in Listing 2.

Listing 2 Example implementation of the keyPathsForValuesAffecting<Key> naming convention

```
+ (NSSet *)keyPathsForValuesAffectingFullName {
    return [NSSet setWithObjects:@"lastName", @"firstName",
nil];
}
```

You can't override the keyPathsForValuesAffectingValueForKey: method when you add a computed property to an existing class using a category, because you're not supposed to override methods in categories. In that case, implement a matching keyPathsForValuesAffecting<Key> class method to take advantage of this mechanism.

Note: You cannot set up dependencies on to-many relationships by implementing keyPathsForValuesAffectingValueForKey:. Instead, you must observe the appropriate attribute of each of the objects in the to-many collection and respond to changes in their values by updating the dependent key yourself. The following section shows a strategy for dealing with this situation.

12:To-Many Relationships (对于多种关系的处理)

The keyPathsForValuesAffectingValueForKey: method does not support key-paths that include a to-many relationship. For example, suppose you have a Department object with a to-many relationship (employees) to a Employee, and Employee has a salary attribute. You might want the Department object have a totalSalary attribute that is dependent upon

the salaries of all the Employees in the relationship. You can not do this with, for example, keyPathsForValuesAffectingTotalSalary and returning employees.salary as a key.

1:There are two possible solutions in both situations:

You can use key-value observing to register the parent (in this example, Department) as an observer of the relevant attribute of all the children (Employees in this example). You must add and remove the parent as an observer as child objects are added to and removed from the relationship (see Registering for Key-Value Observing). In the observeValueForKeyPath:ofObject:change:context: method you update the dependent value in response to changes, as illustrated in the following code fragment:

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

    if (context == totalSalaryContext) {
        [self updateTotalSalary];
    }else{
    // deal with other observations and/or invoke super..
}

- (void)updateTotalSalary {
    [self setTotalSalary:[self
valueForKeyPath:@"employees.@sum.salary"]];
}

- (void)setTotalSalary:(NSNumber *)newTotalSalary {

    if (totalSalary != newTotalSalary) {
        [self willChangeValueForKey:@"totalSalary"];
        _totalSalary = newTotalSalary;
        [self didChangeValueForKey:@"totalSalary"];
    }
}

- (NSNumber *)totalSalary {
    return _totalSalary;
}
```

2. If you're using Core Data, you can register the parent with the application's notification center as an observer of its managed object

context. The parent should respond to relevant change notifications posted by the children in a manner similar to that for key-value observing.

13:Key-Value Observing Implementation Details(KVO实现的细节)

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.

14:Document Revision History (文档版本)

This table describes the changes to Key-Value Observing Programming Guide.

Date	Notes
2016-09-13	Clarified a number of topics and updated the code examples.
2012-07-17	Updated to use new Objective-C features.
	ARCification
2011-03-08	Clarified terminology in "Registering Dependent Keys."
2009-08-14	Added links to some key Cocoa definitions.
2009-05-09	Corrected minor typo.
2009-01-06	Clarified Core Data requirement in Registering Dependent Keys.
2009-01-04	Updated Registering Dependent Keys chapter.
2006-06-28	Updated code examples.
2005-07-07	Clarified that you should not release objects before calling willChangeValueForKey: methods. Noted that Java is not supported.
2004-08-31	Corrected minor typos.
	Clarified the need to nest manual key-value change notifications.
2004-03-20	Modified source example in Registering Dependent Keys.
2004-02-22	Corrected source example in Registering for Key-Value Observing. Added article Key-Value Observing Implementation Details .
2003-10-15	Initial publication of *Key-Value Observing*.

Key-Value Observing(KVO)中英文版本

//1:介绍键-值观察编码

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

//键-值观察是一种机制,允许将对象指定的更改通知其他对象的属性

Important: In order to understand key-value observing, you must first understand key-value coding.

//重要:为了了解键-值观察,您必须首先了解键值编码。

2:At a Glance (KVO的大致过程)

Key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects. It is particularly useful for communication between model and controller layers in an application. (In OS X, the controller layer binding technology relies heavily on key-value observing.) A controller object typically observes properties of model objects, and a view object observes properties of model objects through a controller. In addition, however, a model object may observe other model objects (usually to determine when a dependent value changes) or even itself (again to determine when a dependent value changes).

//键-值观察提供了一种机制,允许将对象更改通知其它对象的特定属性。是特别有用的模型和控制器层在应用程序之间的通信。(在OS X,控制器层绑定技术严重依赖键值观察。)控制器对象通常观察模型对象的性质,和一个视图对象观察通过控制器模型对象的属性。此外,然而,一个模型对象可以观察其他模型对象(通常是确定相关的值更改时),甚至本身(再次决定当一个依赖的价值变化)。

You can observe properties including simple attributes, to-one relationships, and to-many relationships. Observers of to-many relationships are informed of the type of change made—as well as which objects are involved in the change.

//你可以观察属性包括简单的属性,一个关系,和许多关系。许多关系的观察人士通知类型的改变,以及哪些对象参与变革。

A simple example illustrates how KVO can be useful in your application. Suppose a Person object interacts with an Account object, representing the person's savings account at a bank. An instance of Person may need to be aware of when certain aspects of the Account instance change, such as the balance, or the interest rate.

//一个简单的例子说明了KVO如何在应用程序中非常有用。假设一个Person对象与一个帐户对象,代表这个人的银行储蓄账户。实例的人可能需要知道当帐户实例的某些方面的变化,如平衡,或利率。

If these attributes are public properties of Account, the Person could periodically poll the Account to discover changes, but this is of course inefficient, and often impractical. A better approach is to use KVO, which is akin to Person receiving an interrupt when a change occurs.

//如果这些属性是公共账户的性质,人可以定期轮询账户发现变化,但这当然是低效的,往往不切实际。更好的方法是使用KVO,它类似于人收到一个中断发生变化时。

To use KVO, first you must ensure that the observed object, the Account in this case, is KVO compliant. Typically, if your objects inherit from NSObject and you create properties in the usual way, your objects and their properties will automatically be KVO Compliant. It is also possible to implement compliance manually. KVO Compliance describes the difference between automatic and manual key-value observing, and how to implement both.

//使用KVO,首先你必须确保观察对象,帐号在这种情况下,KVO兼容。一般来说,如果你的对象继承NSObject和创建属性以通常的方式,你的对象及其属性将自动KVO兼容。也可以手动实现遵从性。KVO合规描述了自动和手动键值观察之间的区别,以及如何实现。

Next, you must register your observer instance, the Person, with the observed instance, the Account. Person sends an addObserver:forKeyPath:options:context: message to the Account, once for each observed key path, naming itself as the observer

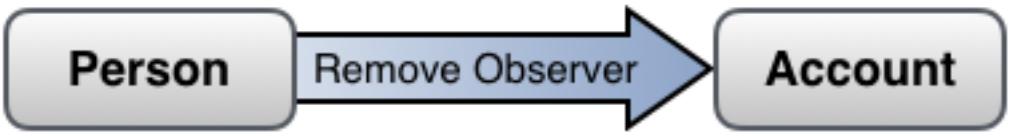
//接下来,您必须注册您的观察者,的人,观察到的实例,Account。Person发送一个addObserver:forKeyPath:选择:背景:Account信息,一旦为每个观察关键路径,命名本身作为观察者

In order to receive change notifications from the Account, Person implements the observeValueForKeyPath:ofObject:change:context: method, required of all observers. The Account will send this message to the Person any time one of the registered key paths changes. The Person can then take appropriate action based upon the change notification

//为了收到Account更改通知,人实现了observeValueForKeyPath:ofObject:改变:背景:方法,要求所有的观察者。帐户将发送此消息给人任何时候注册的一个关键路径的变化。人可以采取适当的行动基于更改通知

Finally, when it no longer wants notifications, and at the very least before it is deallocated, the Person instance must de-register by sending the message removeObserver:forKeyPath: to the Account.

//最后,当它不再想要通知,至少前一致,Person实例必须取消发送消息removeObserver:forKeyPath:Account。



Registering for Key-Value Observing describes the full lifecycle of registering for, receiving, and de-registering for key value observation notifications.

注册键值观察描述了注册的完整生命周期,键值观测接收,注销登记通知。

KVO's primary benefit is that you don't have to implement your own scheme to send notifications every time a property changes. Its well-defined infrastructure has framework-level support that makes it easy to adopt—typically you do not have to add any code to your project. In addition, the infrastructure is already full-featured, which makes it easy to support multiple observers for a single property, as well as dependent values.

//KVO的主要好处是,您不必实现自己的计划,发送通知每次属性变更。其定义良好的基础设施框架级支持,便于adopt-typically你不需要任何代码添加到您的项目。此外,基础设施已经全功能的,这使得它很容易为一个属性支持多个观察者,以及相关的值。

Registering Dependent Keys explains how to specify that the value of a key is dependent on the value of another key.

//注册相关的键解释了如何指定一个关键的价值依赖于另一个关键的价值。

Unlike notifications that use NSNotificationCenter, there is no central object that provides change notification for all observers. Instead, notifications are sent directly to the observing objects when changes are made. NSObject provides this base implementation of key-value observing, and you should rarely need to override these methods.

//与使用NSNotificationCenter的通知,没有中央对象为所有观察者提供了更改通知。相反,直接通知被发送到观察对象所做的修改。NSObject提供此基本实现键值的观察,你应该很少需要覆盖这些方法。

Key-Value Observing Implementation Details describes how key-value observing is implemented.
//键-值观察实现细节描述如何实现键值观察。

3:Registering for Key-Value Observing(KVO的注册KVO)

You must perform the following steps to enable an object to receive key-value observing notifications for a KVO-compliant property:

//你必须执行以下步骤启用对象为KVO-compliant属性接收键值观察通知:

- Register the observer with the observed object using the method addObserver:forKeyPath:options:context:

- 注册观察者与观察对象使用方法addObserver:forKeyPath:

- Implement observeValueForKeyPath:ofObject:change:context: inside the observer to accept change notification messages.

//实现observeValueForKeyPath:ofObject:改变:上下文:在观察者接受更改通知消息。

- Unregister the observer using the method removeObserver:forKeyPath: when it no longer should receive messages. At a minimum, invoke this method before the observer is released from memory.

//注销观察者使用方法removeObserver:forKeyPath:当它不再应该接收消息。至少,调用这个方法之前,观察者从内存中释放。

Important: Not all classes are KVO-compliant for all properties. You can ensure your own classes are KVO-compliant by following the steps described in KVO Compliance. Typically properties in Apple-supplied frameworks are only KVO-compliant if they are documented as such.

//重要:并不是所有的类都是KVO-compliant所有属性。你能保证自己的类是KVO-compliant遵循KVO合规中描述的步骤。通常在Apple-supplied属性框架只是KVO-compliant如果他们记录。

4:Registering as an Observer(KVO的注册观察者)

An observing object first registers itself with the observed object by sending an addObserver:forKeyPath:options:context: message, passing itself as the observer and the key path of the property to be observed. The observer additionally specifies an options parameter and a context pointer to manage aspects of the notifications.

//一个观察对象首先寄存器本身与观察对象发送一个addObserver:forKeyPath:选择:背景:消息,传递自己的观察者和关键路径属性被观察到。观察者另外指定一个选项参数和上下文指针管理方面的通知。

Options(KVO的枚举)

The options parameter, specified as a bitwise OR of option constants, affects both the content of the change dictionary supplied in the notification, and the manner in which notifications are generated.

You opt to receive the value of the observed property from before the change by specifying option NSKeyValueObservingOptionOld. You request the new value of the property with option NSKeyValueObservingOptionNew. You receive both old and new values with the bitwise OR of these options.

//选项参数,指定为一个常量按位或选项,将同时影响改变字典的内容提供通知,并生成通知的方式。你选择接受前观察到的属性的值改变NSKeyValueObservingOptionOld通过指定选项。你请求与选项NSKeyValueObservingOptionNew属性的新值。你收到新旧价值观的按位或这些选项。

You instruct the observed object to send an immediate change notification (before addObserver:forKeyPath:options:context: returns) with the option NSKeyValueObservingOptionInitial. You can use this additional, one-time notification to establish the initial value of a property in the observer.

//你指导观察对象发送立即更改通知(之前addObserver:forKeyPath:选择:背景:返回)NSKeyValueObservingOptionInitial的选项。您可以使用这些额外的,一次性通知在观察者建立属性的初始值。

You instruct the observed object to send a notification just prior to a property change (in addition to the usual notification just after the change) by including the option NSKeyValueObservingOptionPrior. The change dictionary represents a prechange notification by including the key **NSKeyValueChangeNotificationIsPriorKey** with the value of an NSNumber wrapping YES. That key is not otherwise present. You can use the prechange notification when the observer's own KVO compliance requires it to invoke one of the -willChange… methods for one of its properties that depends on an observed property. The usual post-change notification comes too late to invoke willChange… in time.

//你指导观察对象发送一个通知之前属性改变(除了通常的通知后变化),包括选择NSKeyValueObservingOptionPrior。改变字典代表了包括关键NSKeyValueChangeNotificationIsPriorKey prechange通知NSNumber包装的价值肯定的。关键是不存在。时您可以使用prechange通知观察者的KVO合规需要调用一个这么…它的一个属性,方法取决于一个观察到的属性。通常的记账的调用将通知来得太晚了…。

Context (KVO的上下文)

The context pointer in the addObserver:forKeyPath:options:context: message contains arbitrary data that will be passed back to the observer in the corresponding change notifications. You may specify NULL and rely entirely on the key path string to determine the origin of a change notification, but this approach may cause problems for an object whose superclass is also observing the same key path for different reasons.

//上下文指针addObserver:forKeyPath:选择:背景:消息包含任意数据将被传递回观察者在相应的更改通知。您可以指定NULL和完全依赖的关键路径字符串确定更改通知的起源,但是这种方法可能会导致问题的超类的一个对象也观察相同的关键路径是出于不同的原因。

A safer and more extensible approach is to use the context to ensure notifications you receive are destined for your observer and not a superclass.

//一个更安全、更可扩展的方法是使用上下文来确保通知你收到你的观察者,而不是一个超类。

The address of a uniquely named static variable within your class makes a good context. Contexts chosen in a similar manner in the super- or subclass will be unlikely to overlap. You may choose a single context for the entire class and rely on the key path string in the notification message to determine what changed. Alternatively, you may create a distinct context for each observed key path, which bypasses the need for string comparisons entirely, resulting in more efficient notification parsing. Listing 1 shows example contexts for the balance and interestRate properties chosen this way.

//惟一命名的静态变量的地址在您的类上下文。上下文选择以类似的方式在超级-或子类可能重叠。你可以选择一个为整个类和上下文依赖的关键路径字符串的通知消息来确定发生了变化。或者,您可能会创建一个不同的上下文为每个观察关键路径,绕过需要字符串比较完全,导致更有效的通知解析。清单1显示了示例环境平衡和贷款利率的属性选择这种方式。

Listing 1 Creating context pointers

```
static void *PersonAccountBalanceContext =
&PersonAccountBalanceContext;
static void *PersonAccountInterestRateContext =
&PersonAccountInterestRateContext;
```

The example in Listing 2 demonstrates how a Person instance registers itself as an observer for an Account instance's balance and interestRate properties using the given context pointers.

//清单2中的示例演示了如何一个人实例寄存器本身为一个帐户实例作为观察者的资产和贷款利率属性使用给定的上下文指针。

Listing 2 Registering the inspector as an observer of the balance and interestRate properties

//清单2注册审查员作为观察者的平衡和利率的属性

```
- (void)registerAsObserverForAccount:(Account*)account {

    [account addObserver:self
              forKeyPath:@"balance"
                 options:(NSKeyValueObservingOptionNew |
                          NSKeyValueObservingOptionOld)
                 context:PersonAccountBalanceContext];

    [account addObserver:self
              forKeyPath:@"interestRate"
                 options:(NSKeyValueObservingOptionNew |
                          NSKeyValueObservingOptionOld)
                 context:PersonAccountInterestRateContext];
}
```

Note: The key-value observing addObserver:forKeyPath:options:context: method does not maintain strong references to the observing object, the observed objects, or the context. You should ensure that you maintain strong references to the observing, and observed, objects, and the context as necessary

//注意:键值观察addObserver:forKeyPath:选择:背景:方法不保持强劲观察对象的引用,观察对象,或上下文。你应该确保你保持强引用观察,观察,必要时对象和上下文

5:Receiving Notification of a Change (KVO的接收到通知变化)

When the value of an observed property of an object changes, the observer receives an observeValueForKeyPath:ofObject:change:context: message. All observers must implement this method.

//时观察到的值属性对象的变化,观察者接收一个observeValueForKeyPath:ofObject:改变:背景:消息。所有观察家都必须实现这个方法。

The observing object provides the key path that triggered the notification, itself as the relevant object, a dictionary containing details about the change, and the context pointer that was provided when the observer was registered for this key path.

//观察对象提供了关键路径,引发了通知,本身为相关的对象,一个字典,其中包含的细节变化,和上下文指针,当观察者提供了注册这个关键路径。

The change dictionary entry NSKeyValueChangeKindKey provides information about the type of change that occurred. If the value of the observed object has changed, the NSKeyValueChangeKindKey entry returns NSKeyValueChangeSetting. Depending on the options specified when the observer was registered, the NSKeyValueChangeOldKey and NSKeyValueChangeNewKey entries in the change dictionary contain the values of the property before, and after, the change. If the property is an object, the value is provided directly. If the property is a scalar or a C

structure, the value is wrapped in an NSValue object (as with key-value coding).

//改变字典条目NSKeyValueChangeKindKey提供信息的类型发生变化。如果观察到的值对象发生了变化,NSKeyValueChangeSetting NSKeyValueChangeKindKey条目的回报。指定的选项取决于观察者注册时,改变字典中的NSKeyValueChangeOldKey和NSKeyValueChangeNewKey条目包含属性的值之前,和之后,改变。如果属性是一个对象,直接提供的值。如果属性是一个标量或C结构,包裹在一个价值NSValue对象(与键值编码)。

If the observed property is a to-many relationship, the NSKeyValueChangeKindKey entry also indicates whether objects in the relationship were inserted, removed, or replaced by returning NSKeyValueChangeInsertion, NSKeyValueChangeRemoval, or NSKeyValueChangeReplacement, respectively.

//如果观察到的属性是一个很多的关系,NSKeyValueChangeKindKey条目还指示对象的关系是否插入,删除,或返回NSKeyValueChangeInsertion所取代,NSKeyValueChangeRemoval,或NSKeyValueChangeReplacement,分别。

The change dictionary entry for NSKeyValueChangeIndexesKey is an NSIndexSet object specifying the indexes in the relationship that changed. If NSKeyValueObservingOptionNew or NSKeyValueObservingOptionOld are specified as options when the observer is registered, the NSKeyValueChangeOldKey and NSKeyValueChangeNewKey entries in the change dictionary are arrays containing the values of the related objects before, and after, the change.
The example in Listing 3 shows the observeValueForKeyPath:ofObject:change:context: implementation for the Person observer that logs the old and new values of the properties balance and interestRate, as registered in Listing 2.

//改变NSKeyValueChangeIndexesKey字典条目是一个NSIndexSet对象指定索引的关系改变了。如果NSKeyValueObservingOptionNew或NSKeyValueObservingOptionOld作为选项指定注册观察者时,改变字典中

的NSKeyValueChangeOldKey和NSKeyValueChangeNewKey条目数组包含相关对象的值之前,和之后,改变。清单3中的示例显示了observeValueForKeyPath:ofObject:改变:背景:实现日志的人观察者的新旧值属性平衡和贷款利率,如清单2注册。

Listing 3 Implementation of observeValueForKeyPath:ofObject:change:context:

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

    if (context == PersonAccountBalanceContext) {
        // Do something with the balance...
    } else if (context == PersonAccountInterestRateContext) {
        // Do something with the interest rate...
    } else {
        // Any unrecognized context must belong to super
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

If you specified a NULL context when registering an observer, you compare the notification's key path against the key paths you are observing to determine what has changed. If you used a single context for all observed key paths, you first test that against the notification's context, and finding a match, use key path string comparisons to determine what specifically has changed. If you have provided a unique context for each key path, as demonstrated here, a series of simple pointer comparisons tells you simultaneously whether or not the notification is for this observer, and if so, what key path has changed.

//如果你指定一个空上下文当注册一个观察者,你比较通知的关键路径和关键路径你观察来确定发生了什么变化。如果您使用一个上下文所有观察到的关键路径,你第一次测试,对通知的上下文,并找到一个匹配,使用关键路径字符串比较来确定具体发生了变化。如果你提供了一个独特的背景下为每

个关键路径,是本文所演示的,一系列的简单指针比较同时告诉你是否通知观察者,,如果是这样,关键路径改变了什么。

In any case, the observer should always call the superclass's implementation of observeValueForKeyPath:ofObject:change:context: when it does not recognize the context (or in the simple case, any of the key paths), because this means a superclass has registered for notifications as well.
Note: If a notification propagates to the top of the class hierarchy, NSObject throws an NSInternalInconsistencyException because this is a programming error: a subclass failed to consume a notification for which it registered.

//在任何情况下,观察者observeValueForKeyPath应该调用超类的实现:ofObject:改变:背景:当它不能识别上下文(或在简单的情况下,任何的关键路径),因为这意味着一个超类注册通知。注意:如果一个通知传播类层次结构的顶部,NSObject抛出NSInternalInconsistencyException因为这是一个编程错误:没有一个子类使用它注册的通知。

6:Removing an Object as an Observer(KVO的移除观察者)

You remove a key-value observer by sending the observed object a removeObserver:forKeyPath:context: message, specifying the observing object, the key path, and the context. The example in Listing 4 shows Person removing itself as an observer of balance and interestRate.

//你删除一个键-值观察者通过发送观察对象removeObserver:forKeyPath:背景:信息,指定观察对象,关键路径和上下文。清单4中的示例显示了人删除本身作为观察者的平衡和贷款利率。

Listing 4 Removing the inspector as an observer of balance and interestRate

```
- (void)unregisterAsObserverForAccount:
(Account*)account {

    [account removeObserver:self
                 forKeyPath:@"balance"
                    context:PersonAccountBalanceContext];
```

```
    [account removeObserver:self
                 forKeyPath:@"interestRate"
                    context:PersonAccountInterestRateContext];
}
```

After receiving a removeObserver:forKeyPath:context: message, the observing object will no longer receive any observeValueForKeyPath:ofObject:change:context: messages for the specified key path and object.

//在收到removeObserver:forKeyPath:背景:消息,观察对象将不再接收任何observeValueForKeyPath:ofObject:改变:背景:消息指定关键路径和对象。

When removing an observer, keep several points in mind:(移除观察者注意事项)

- Asking to be removed as an observer if not already registered as one results in an NSRangeException. You either call removeObserver:forKeyPath:context: exactly once for the corresponding call to addObserver:forKeyPath:options:context:, or if that is not feasible in your app, place the removeObserver:forKeyPath:context: call inside a try/catch block to process the potential exception.

//作为观察员要求删除如果不是NSRangeException已经注册为一个结果。你要么叫removeObserver:forKeyPath:背景:到底一次相应的调用addObserver:forKeyPath:选择:背景:,或如果在你的应用程序,这不是可行的地方removeObserver:forKeyPath:背景:在一个try / catch块来处理潜在的例外。

- An observer does not automatically remove itself when deallocated. The observed object continues to send notifications, oblivious to the state of the observer. However, a change notification, like any other message, sent to a released object, triggers a memory access exception. You therefore ensure that observers remove themselves before disappearing from memory.

//一个观察者本身不会自动删除时收回。观察对象继续发送通知,无视的状态观测器。然而,更改通知,像其他任何信息,发送给发布对象,触发一个内存访问异常。因此确保观察员从内存中删除自己消失之前。

- The protocol offers no way to ask an object if it is an observer or being observed. Construct your code to avoid release related errors. A typical pattern is to register as an observer during the observer's initialization (for example in init or viewDidLoad) and unregister during deallocation (usually in dealloc), ensuring properly paired and ordered add and remove messages, and that the observer is unregistered before it is freed from memory.

//协议提供了没有办法问一个对象是一个观察者或被观察到。构建你的代码,以避免释放相关的错误。典型的模式是登记为一个观察者在观察者的初始化(例如在init或viewDidLoad)和注销在回收dealloc(通常),确保正确配对和命令添加和删除消息,观察者是未注册前从内存中释放。

7:KVO Compliance (KVO需要注意的几点)

In order to be considered KVO-compliant for a specific property, a class must ensure the following:

- The class must be key-value coding compliant for the property, as specified in Ensuring KVC Compliance.

//类必须键值编码的属性,指定在确保现有的遵从性。

- KVO supports the same data types as KVC, including Objective-C objects and the scalars and structures listed in Scalar and Structure Support.

//KVO支持与现有的相同的数据类型,包括objective - c对象和标量中列出的标量和结构和结构的支持。

- The class emits KVO change notifications for the property.

//类排放KVO属性更改通知。

- Dependent keys are registered appropriately (see Registering Dependent Keys).

//依赖键注册适当的(参见注册相关的键)。

There are two techniques for ensuring the change notifications are emitted. Automatic support is provided by NSObject and is by default available for all properties of a class that are key-value coding compliant. Typically, if you follow standard Cocoa coding and naming conventions, you can use automatic change notifications—you don't have to write any additional code.

//有两个技术确保发出的变更通知。自动默认NSObject和提供的支持是可用于类的所有属性键值编码兼容。通常,如果你遵循标准可可编码和命名约定,您可以使用自动改变用户不需要编写额外的代码。

Manual change notification provides additional control over when notifications are emitted, and requires additional coding. You can control automatic notifications for properties of your subclass by implementing the class method automaticallyNotifiesObserversForKey:

//手动更改通知通知发出时,提供了额外的控制,需要额外的编码。你可以控制自动通知你的子类实现的属性类方法 automaticallyNotifiesObserversForKey:

8:Automatic Change Notification(自动观察通知)

NSObject provides a basic implementation of automatic key-value change notification. Automatic key-value change notification informs observers of changes made using key-value compliant accessors, as well as the key-value coding methods. Automatic notification is also supported by the collection proxy objects returned by, for example, mutableArrayValueForKey:.

//NSObject提供了一个基本实现自动键-值变化的通知。自动键-值更改通知通知观察者使用键值的读写方法的更改,以及键值编码方法。还支持自动通知返回的代理对象集合,例如,mutableArrayValueForKey:。

The examples shown in Listing 1 result in any observers of the property name to be notified of the change.

//示例如清单1所示的结果在任何观察员的属性名变更的通知。

Listing 1 Examples of method calls that cause KVO change notifications to be emitted

```
// Call the accessor method.
[account setName:@"Savings"];
// Use setValue:forKey:.
[account setValue:@"Savings" forKey:@"name"];
// Use a key path, where 'account' is a kvc-compliant
property of 'document'.
[document setValue:@"Savings" forKeyPath:@"account.name"];
// Use mutableArrayValueForKey: to retrieve a relationship
proxy object.
Transaction *newTransaction = <#Create a new transaction
for the account#>;
NSMutableArray *transactions = [account
mutableArrayValueForKey:@"transactions"];
[transactions addObject:newTransaction];
```

9:Manual Change Notification(手动观察通知)

In some cases, you may want control of the notification process, for example, to minimize triggering notifications that are unnecessary for application specific reasons, or to group a number of changes into a single notification. Manual change notification provides means to do this.

//在某些情况下,您可能希望通知过程的控制,例如,减少引发不必要的特定于应用程序的原因,通知或组的数量变化成一个通知。手动更改通知提供了手段。

Manual and automatic notifications are not mutually exclusive. You are free to issue manual notifications in addition to the automatic ones already in place. More typically, you may want to completely take control of the notifications for a particular property. In this case, you override the NSObject implementation of automaticallyNotifiesObserversForKey:. For properties whose automatic notifications you want to preclude, the subclass implementation of automaticallyNotifiesObserversForKey:

should return NO. A subclass implementation should invoke super for any unrecognized keys. The example in Listing 2 enables manual notification for the balance property, allowing the superclass to determine the notification for all other keys.

//手动和自动通知并不是相互排斥的。你自由发行手册除了自动的通知已经到位。通常情况下,你可能想要完全控制通知特定属性。在这种情况下,您覆盖的NSObject实现automaticallyNotifiesObserversForKey:。属性的自动通知你想排除,automaticallyNotifiesObserversForKey的子类实现:应该没有回来。一个子类实现应该为任何未被调用超键。清单2中的示例允许手动通知资产产权,允许超类决定通知所有其他键。

Listing 2 Example implementation of automaticallyNotifiesObserversForKey:

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString
*)theKey {

    BOOL automatic = NO;
    if ([theKey isEqualToString:@"balance"]) {
        automatic = NO;
    }else {
        automatic = [super
automaticallyNotifiesObserversForKey:theKey];
    }
    return automatic;
}
```

To implement manual observer notification, you invoke willChangeValueForKey: before changing the value, and didChangeValueForKey: after changing the value. The example in

//实现手动通知观察者,你调用willChangeValueForKey:改变价值之前,和didChangeValueForKey:改变后的值。中的例子

Listing 3 implements manual notifications for the balance property.

Listing 3 Example accessor method implementing manual notification

```
- (void)setBalance:(double)theBalance {
    [self willChangeValueForKey:@"balance"];
    _balance = theBalance;
    [self didChangeValueForKey:@"balance"];
}
```

You can minimize sending unnecessary notifications by first checking if the value has changed. The example in Listing 4 tests the value of balance and only provides the notification if it has changed.

//发送不必要的通知可以最小化首先检查如果值已经改变了。清单4中的示例测试资产的价值,如果它改变了只提供通知。

Listing 4 Testing the value for change before providing notification

```
- (void)setBalance:(double)theBalance {

    if (theBalance != _balance) {
        [self willChangeValueForKey:@"balance"];
        _balance = theBalance;
        [self didChangeValueForKey:@"balance"];
    }
}
```

If a single operation causes multiple keys to change you must nest the change notifications as shown in Listing 5.

//如果单个操作导致多个键改变必须巢通知如清单5所示。

Listing 5 Nesting change notifications for multiple keys

```
- (void)setBalance:(double)theBalance {

    [self willChangeValueForKey:@"balance"];
    [self willChangeValueForKey:@"itemChanged"];
    _balance = theBalance;
    _itemChanged = _itemChanged+1;
    [self didChangeValueForKey:@"itemChanged"];
    [self didChangeValueForKey:@"balance"];
}
```

In the case of an ordered to-many relationship, you must specify not only the key that changed, but also the type of change and the indexes

of the objects involved. The type of change is an NSKeyValueChange that specifies NSKeyValueChangeInsertion, NSKeyValueChangeRemoval, or NSKeyValueChangeReplacement. The indexes of the affected objects are passed as an NSIndexSet object.

The code fragment in Listing 6 demonstrates how to wrap a deletion of objects in the to-many relationship transactions.

//在一个有序的情况下许多关系,您必须指定不仅改变的关键,但也改变的类型和对象的索引。的变化是一个NSKeyValueChange类型指定NSKeyValueChangeInsertion,NSKeyValueChangeRemoval或NSKeyValueChangeReplacement。受影响对象的索引作为NSIndexSet传递对象。清单6中的代码片段演示了如何删除对象封装在许多关系事务。

Listing 6 Implementation of manual observer notification in a to-many relationship

```
- (void)removeTransactionsAtIndexes:(NSIndexSet
*)indexes {

    [self willChange:NSKeyValueChangeRemoval
        valuesAtIndexes:indexes forKey:@"transactions"];
    // Remove the transaction objects at the specified
indexes.
    [self didChange:NSKeyValueChangeRemoval
        valuesAtIndexes:indexes forKey:@"transactions"];
}
```

10:Registering Dependent Keys (注册依赖keys)

There are many situations in which the value of one property depends on that of one or more other attributes in another object. If the value of one attribute changes, then the value of the derived property should also be flagged for change. How you ensure that key-value observing notifications are posted for these dependent properties depends on the cardinality of the relationship.

//有许多情况下,一个属性的值取决于其他一个或多个属性的另一个对象。如果一个属性的值变化,那么派生属性的值也应该改变的标记。你如何确保键值观察通知发布这些依赖属性取决于关系的基数。

11:To-One Relationships (单一关系处理)

To trigger notifications automatically for a to-one relationship you should either override keyPathsForValuesAffectingValueForKey: or implement a suitable method that follows the pattern it defines for registering dependent keys.

//自动触发通知你应该覆盖keyPathsForValuesAffectingValueForKey:一个关系或实现一个合适的方法,遵循模式定义注册相关的键。

For example, the full name of a person is dependent on both the first and last names. A method that returns the full name could be written as follows:

//例如,一个人的全名是依赖于第一个和最后一个名称。一个方法,返回全名可以写成:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"%@ %@",firstName,
lastName];
}
```

An application observing the fullName property must be notified when either the firstName or lastName properties change, as they affect the value of the property.

//应用程序观察fullName财产时,必须通知firstName和lastName属性的改变,因为他们影响房地产的价值。

One solution is to override keyPathsForValuesAffectingValueForKey: specifying that the fullName property of a person is dependent on the lastName and firstName properties. Listing 1 shows an example implementation of such a dependency:

//一个解决方案是覆盖keyPathsForValuesAffectingValueForKey:指定fullName属性依赖于人的名和姓的属性。清单1显示了一个示例实现的依赖:

Listing 1 Example implementation of keyPathsForValuesAffectingValueForKey:

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:
(NSString *)key {

    NSSet *keyPaths = [super
keyPathsForValuesAffectingValueForKey:key];
    if ([key isEqualToString:@"fullName"]) {
        NSArray *affectingKeys = @[@"lastName", @"firstName"];
        keyPaths = [keyPaths
setByAddingObjectsFromArray:affectingKeys];
    }
    return keyPaths;
}
```

Your override should typically invoke super and return a set that includes any members in the set that result from doing that (so as not to interfere with overrides of this method in superclasses).

//覆盖应该通常调用超级并返回一组包括的任何成员组,由于这样做(以免干扰覆盖超类的方法)。

You can also achieve the same result by implementing a class method that follows the naming convention keyPathsForValuesAffecting<Key>, where <Key> is the name of the attribute (first letter capitalized) that is dependent on the values. Using this pattern the code in Listing 1 could be rewritten as a class method named keyPathsForValuesAffectingFullName as shown in Listing 2.

//你也可以实现相同的结果通过实现一个类方法,遵循命名约定keyPathsForValuesAffecting <键>、<键>在哪里属性的名称(首字母大写),取决于价值观。清单1中的代码使用这个模式可以写成一个类方法命名keyPathsForValuesAffectingFullName如清单2所示。

Listing 2 Example implementation of the keyPathsForValuesAffecting<Key> naming convention

```
+ (NSSet *)keyPathsForValuesAffectingFullName {
    return [NSSet setWithObjects:@"lastName", @"firstName",
nil];
}
```

You can't override the keyPathsForValuesAffectingValueForKey: method when you add a computed property to an existing class using a category, because you're not supposed to override methods in categories. In that case, implement a matching keyPathsForValuesAffecting<Key> class method to take advantage of this mechanism.

//你不能覆盖keyPathsForValuesAffectingValueForKey:方法当你将一个计算的属性添加到现有类使用一个类别,因为你不应该覆盖方法类别。在这种情况下,实现一个匹配keyPathsForValuesAffecting <键>类方法利用这种机制。

Note: You cannot set up dependencies on to-many relationships by implementing keyPathsForValuesAffectingValueForKey:. Instead, you must observe the appropriate attribute of each of the objects in the to-many collection and respond to changes in their values by updating the dependent key yourself. The following section shows a strategy for dealing with this situation.

//注意:您不能设置许多依赖关系通过实现keyPathsForValuesAffectingValueForKey:。相反,您必须遵守适当的每个对象的属性很多收集和响应值的变化通过更新自己依赖的关键。以下部分显示了一个策略来处理这种情况。

12:To-Many Relationships (对于多种关系的处理)

The keyPathsForValuesAffectingValueForKey: method does not support key-paths that include a to-many relationship. For example, suppose you have a Department object with a to-many relationship (employees) to a Employee, and Employee has a salary attribute. You might want the Department object have a totalSalary attribute that is dependent upon the salaries of all the Employees in the relationship. You can not do this with, for example, keyPathsForValuesAffectingTotalSalary and returning employees.salary as a key.

//keyPathsForValuesAffectingValueForKey:不支持关键路径方法,包括许多关系。例如,假设您有一个使用许多部门对象关系(员工)员工,和员工工资属性。您可能希望部门对象有一个totalSalary属性依赖于所有的员工的工资关系。你不能这样做,例如,keyPathsForValuesAffectingTotalSalary和返回的员工。工资作为一个关键。

1:There are two possible solutions in both situations:

You can use key-value observing to register the parent (in this example, Department) as an observer of the relevant attribute of all the children (Employees in this example). You must add and remove the parent as an observer as child objects are added to and removed from the relationship (see Registering for Key-Value Observing). In the observeValueForKeyPath:ofObject:change:context: method you update the dependent value in response to changes, as illustrated in the following code fragment:

> //您可以使用键值观察注册父(在这个例子中,部门)作为观察者的相关属性的所有的孩子(员工在本例中)。作为观察员必须添加和删除父在子对象添加和删除的关系(参见注册键值观察)。observeValueForKeyPath:ofObject:改变:背景:方法依赖价值变化而更新,如以下代码片段:

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

```
        if (context == totalSalaryContext) {
            [self updateTotalSalary];
        }else{
        // deal with other observations and/or invoke super..
    }

    - (void)updateTotalSalary {
        [self setTotalSalary:[self
valueForKeyPath:@"employees.@sum.salary"]];
    }

    - (void)setTotalSalary:(NSNumber *)newTotalSalary {

        if (totalSalary != newTotalSalary) {
            [self willChangeValueForKey:@"totalSalary"];
            _totalSalary = newTotalSalary;
            [self didChangeValueForKey:@"totalSalary"];
        }
    }

    - (NSNumber *)totalSalary {
        return _totalSalary;
    }
```

2. If you're using Core Data, you can register the parent with the application's notification center as an observer of its managed object context. The parent should respond to relevant change notifications posted by the children in a manner similar to that for key-value observing.

//2。如果你使用核心数据,您可以注册父与应用程序的通知中心作为观察者的管理对象上下文。父母应该孩子应对相关变化通知发布的键值观察的方式类似。

13:Key-Value Observing Implementation Details(KVO实现的细节)

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.

//自动键值观察使用一种叫做isa-swizzling的技术实现。isa指针,顾名思义,指向的对象的类维护调度表。这个调度表基本上包含指向类实现方法,以及其他数据。当一个观察者注册对象的一个属性isa观察对象的指针被修改,指着一个中间类而不是在真正的类。由于isa指针的值并不一定反映实际的类的实例。你不应该依靠isa指针来确定类会员。相反,您应该使用类方法来确定类对象的实例。