07/26 KVC v.s. KVO

KVO & KVC:
Key-Value
Coding: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueCoding/

Other Cocoa Tech Rely on KVC

- Key-value observing (KVO): enables objects to register for asynchronous notifications driven by changes in another object's properties,
- cocoa bindings
- Core Data
- AppleScript

1. KVC Fundamentals

- Access object properties. — attributes/one to one relationship/one to many relationship

```
[myAccount setCurrentBalance:@(100.0)];
[myAccount setValue:@(100.0) forKey:@"currentBalance"];
[myAccount setValue:@(100.0) forKeyPath:@“account.currentBalance”];
setValuesForKeysWithDictionary:
```

- Access collection properties.

  `mutableArrayValueForKey:` and `mutableArrayValueForKeyPath:`
  `mutableSetValueForKey:` and `mutableSetValueForKeyPath:`
  `mutableOrderedSetValueForKey:` and `mutableOrderedSetValueForKeyPath:`

- Invoke collection operators on collection objects

  invoke @avg, @count, @max, @min, @sum

```
[self.transactions valueForKeyPath:@"@max.date"];
```

- Access non-object properties. (Because all properties in Swift are objects, this section only apples to Objective-C properties.) In objc, need to handling of non-object properties.

When you invoke one of the key-value coding protocol setters with a `nil` value for a non-object property, the setter has no obvious, general course of action to take. Therefore, it sends a `setNilValueForKey:` message to the object receiving the setter call.

```
- (void)setNilValueForKey:(NSString *)key

    {
        if ([key isEqualToString:@"age"]) {
```

```
        [self setValue:@(0) forKey:@”age”];
    } else {
        [super setNilValueForKey:key];
    }
}
```

```
typedef struct {
    float x, y, z;
    } ThreeFloats;
    @interface MyClass
    @property (nonatomic) ThreeFloats threeFloats;
    @end
      NSValue*   result      =      [   myClass
      valueForKey:   @"threeFloats”   ];
    ThreeFloats floats = {1., 2., 3.};
     NSValue* value = [NSValue valueWithBytes:&floats
      objCType:@encode(ThreeFloats)];
    [myClass setValue:value forKey:@"threeFloats"];
```

- Access properties by key path.

- Validating Properties

  When you call the `validateValue:forKey:error:` (or the `validateValue:forKeyPath:error:`) method, the default implementation of the protocol searches the object receiving the validation message

  You typically use the validation described here only in Objective-C. In Swift, property validation is more idiomatically handled by relying on compiler support for optionals and strong type checking, while using the built-in willSet and didSet property observers to test any run-time API contracts, as described in the Property Observers section of *The Swift Programming Language (Swift 4)*.

- Accessor Search Pattern
  - NOTE: The descriptions in this section use <key> or <Key> as a placeholder for the key string that appears as a parameter in one of the key-value coding protocol methods, which is then used by that method as part of a secondary method call or variable name lookup. The mapped property name obeys the placeholder's case. For example, for the getters <key> and is<Key>, the property named hidden maps to `hidden` and `isHidden`.
  - Search Pattern for basic Getter:
    - default implementation of `valueForKey:`, given a `key` parameter as input, carries out the following procedure, operating from within the class instance receiving the `valueForKey:` call.
    - 1. search instance for accessor method with name like `get<Key>, <key>,is<Key>,or _<key>. If found, invoke it and proceed to 5.`
    - `2. search instance for method countOf<Key> and objectIn<Key>AtIndex: (corresponding to`

the primitive methods defined by the NSArray class)
and <key>AtIndexes: (corresponding to
the NSArray method objectsAtIndexes:)
If above function works, create a collection proxy object
responds to all NSArray. Otherwise, step 3

- 3. If no simple accessor method or group of array access methods is found, look for a triple of methods
named countOf<Key>, enumeratorOf<Key>,
and memberOf<Key>: (corresponding to the primitive methods defined by the NSSet class).
return NSSet, otherwise, step 4.
- 4. If no simple accessor method or group of collection access methods is found, and if the receiver's class
method accessInstanceVariablesDirectly returns YES, search for an instance variable named _<key>, _is<Key>, <key>,
or is<Key>, in that order. If found, directly obtain the value of the instance variable and proceed to step 5.
Otherwise, proceed to step 6.
- 5. If the retrieved property value is an object pointer, simply return the result.
If the value is a scalar type supported by NSNumber, store it in an NSNumber instance and return that.
If the result is a scalar type not supported by NSNumber, convert to an NSValue object and return that.
- 6. If all else fails, invoke valueForUndefinedKey:. This raises an exception by default, but a subclass of NSObject may provide key-specific behavior.

- Search Pattern for basic Setter:
  - The default implementation of setValue:forKey:,
given key and value parameters as input, attempts to set a property named key to value
  - 1. Look for the first accessor named set<Key>: or _set<Key>, in that order. If found, invoke it with the input value (or unwrapped value, as needed) and finish.
  - 2. If no simple accessor is found, and if the class
method accessInstanceVariablesDirectly returns YES, look for an instance variable with a name like _<key>, _is<Key>, <key>,
or is<Key>, in that order. If found, set the variable directly with the input value (or unwrapped value) and finish.
  - 3. Upon finding no accessor or instance variable,
invoke setValue:forUndefinedKey:. This raises an exception by default, but a subclass of NSObject may provide key-specific behavior.
- Search Pattern for Mutable Arrays:

- The default implementation of `mutableArrayValueForKey:`, given a `key` parameter as input, returns a mutable proxy array for a property named `key` inside the object receiving the accessor call, using the following procedure:
- 1. Look for a pair of methods(mutable array methods) with names like `insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:` (corresponding to the `NSMutableArray` primitive methods`insertObject:atIndex:` and `removeObjectAtIndex:` respectively), or methods with names like `insert<Key>:atIndexes:` and `remove<Key>AtIndexes:` (corresponding to the `NSMutableArrayinsertObjects:atIndexes:` and `removeObjectsAtIndexes:` methods).
  If the object has at least one insertion method and at least one removal method, return a proxy object that responds to `NSMutableArray` messages by sending some combination of `insertObject:in<Key>AtIndex:`, `removeObjectFrom<Key>AtIndex:`, `insert<Key>:atIndexes:`, and `remove<Key>AtIndexes:` messages to the original receiver of `mutableArrayValueForKey:`.
  When the object receiving a `mutableArrayValueForKey:` message also implements an optional replace object method with a name like `replaceObjectIn<Key>AtIndex:withObject:` or `replace<Key>AtIndexes:with<Key>:`, the proxy object utilizes those as well when appropriate for best performance.
- 2. `performance` If the object does not have the mutable array methods, look instead for an accessor method whose name matches the pattern `set<Key>:`. In this case, return a proxy object that responds to `NSMutableArray` messages by issuing a `set<Key>:` message to the original receiver of `mutableArrayValueForKey:`.
  NOTE
  The mechanism described in this step is much less efficient than that of the previous step, because it may involve repeatedly creating new collection objects instead of modifying an existing one. Therefore, you should generally avoid it when designing your own key-value coding compliant objects.
- 3. If neither the mutable array methods, nor the accessor are found, and if the receiver's class responds `YES` to `accessInstanceVariablesDirectly`, search for

an instance variable with a name like _<key> or <key>, in that order.

If such an instance variable is found, return a proxy object that forwards each NSMutableArray message it receives to the instance variable's value, which typically is an instance of NSMutableArray or one of its subclasses.

- 4. If all else fails, return a mutable collection proxy object that issues a setValue:forUndefinedKey: message to the original receiver of the mutableArrayValueForKey: message whenever it receives an NSMutableArray message.

  The default implementation of setValue:forUndefinedKey: raises an NSUndefinedKeyException, but subclasses may override this behavior.

- Search Pattern for Mutable Ordered Sets:
  - 1. Search for methods(mutable ordered set method) with names like insertObject:in<Key>AtIndex: and removeObjectFrom<Key>AtIndex: (corresponding to the two most primitive methods defined by the NSMutableOrderedSet class), and also insert<Key>:atIndexes: and remove<Key>AtIndexes: (corresponding to insertObjects:atIndexes: and removeObjectsAtIndexes:).
  - 2. If the mutable set methods are not found, search for an accessor method with a name like set<Key>:. In this case, the returned proxy object sends a set<Key>: message to the original receiver of mutableOrderedSetValueForKey: every time it receives a NSMutableOrderedSet message.
  - 3. If neither the mutable set messages nor the accessor are found, and if the receiver's accessInstanceVariablesDirectly class method returns YES, search for an instance variable with a name like _<key> or <key>, in that order. If such an instance variable is found, the returned proxy object forwards any NSMutableOrderedSet messages it receives to the instance variable's value, which is typically an instance of NSMutableOrderedSet or one of its subclasses.
  - 4. If all else fails, the returned proxy object sends a setValue:forUndefinedKey: message to the original receiver of mutableOrderedSetValueForKey: whenever it receives a mutable set message.

    The default implementation of setValue:forUndefinedKey: raises an NSUndefinedKeyException, but objects may override this behavior.
- Search Pattern for Mutable Sets:
  - 1. Search for methods with names like add<Key>Object: and remove<Key>Object: (corresponding to the NSMutableSet primitive

methods `addObject:` and `removeObject:` respectively) and also `add<Key>:` and `remove<Key>:`

- 2. If the receiver of the mutableSetValueForKey: call is a managed object, the search pattern does not continue as it would for non-managed objects. See Managed Object Accessor Methods in *Core Data Programming Guide* for more information.
- 3. If the mutable set methods are not found, and if the object is not a managed object, search for an accessor method with a name like set<Key>:. If such a method is found, the returned proxy object sends a set<Key>: message to the original receiver of mutableSetValueForKey: for each NSMutableSetmessage it receives.
- 4.If the mutable set methods and the accessor method are not found, and if the accessInstanceVariablesDirectly class method returns YES, search for an instance variable with a name like _<key> or <key>, in that order. If such an instance variable is found, the proxy object forwards each NSMutableSet message it receives to the instance variable's value, which is typically an instance of NSMutableSet or one of its subclasses.
- 5.If all else fails, the returned proxy object responds to any NSMutableSet message it receives by sending a setValue:forUndefinedKey: message to the original receiver of mutableSetValueForKey:.

2.Adopting Key-Value Coding

- Achieving Basic Key-Value Coding Compliance
  - Basic Getters:
    - ```
      - (NSString*)title
      {
          // Extra getter logic…
          return _title;
      }
      ```
    - ```
      - (BOOL)isHidden
      {
          // Extra getter logic…
          return _hidden;
      }
      ```
  - Basic Setters:
    - ```
      - (void)setHidden:(BOOL)hidden
      {
      // Extra setter logic… Never call validation methods in
        `Validating Properties`
      _hidden = hidden;
      }
      ```
  - Instance Variables
      ```

- it queries its class's `accessInstanceVariablesDirectly` method to see if the class allows direct use of instance variables. By default, this class method returns `YES`, although you can override this method to return `NO`.
- If you do allow use of ivars, ensure that they are named in the usual way, using the property name prefixed by an underscore (_). Normally, the compiler does this for you when automatically synthesizing properties, but if you use an explicit `@synthesize` directive, you can enforce this naming yourself: @synthesize title = _title;
- You use a `@dynamic` directive to inform the compiler that you will provide getters and setters at runtime.

- Defining Collection Methods
  - Accessing Indexed Collections (indexed accessor methods, an instance of NSArray or NSMutableArray)
    - Indexed Collection Getters
      - `countOf<Key>`

```
- (NSUInteger)countOfTransactions {
 return [self.transactions count];
 }
```

      - `objectIn<Key>AtIndex:` or `<key>AtIndexes:`

```
- (id)objectInTransactionsAtIndex:(NSUInteger)index {
 return [self.transactions objectAtIndex:index];
 }
 - (NSArray *)transactionsAtIndexes:(NSIndexSet *)indexes
   {
 return [self.transactions objectsAtIndexes:indexes];
 }
```

      - `get<Key>:range: (optional but can result improved performance)`

```
- (void)getTransactions:(Transaction * __unsafe_unretained
*)buffer
 range:(NSRange)inRange {
 [self.transactions getObjects:buffer range:inRange];
 }
```

    - Indexed Collection Mutators (Supporting a mutable to-many relationship with indexed accessors requires implementing a different group of methods.)
      - `insertObject:in<Key>AtIndex:` or `insert<Key>:atIndexes:`

```
- (void)insertObject:(Transaction *)transaction
 inTransactionsAtIndex:(NSUInteger)index {
 [self.transactions insertObject:transaction
   atIndex:index];
 }
 - (void)insertTransactions:(NSArray *)transactionArray
 atIndexes:(NSIndexSet *)indexes {
 [self.transactions insertObjects:transactionArray
   atIndexes:indexes];
 }
```

        - removeObjectFrom<Key>AtIndex: or remove<Key>AtIndexes:
          ```
          - (void)removeObjectFromTransactionsAtIndex:
          (NSUInteger)index {
           [self.transactions removeObjectAtIndex:index];
           }
           - (void)removeTransactionsAtIndexes:(NSIndexSet
             *)indexes {
           [self.transactions removeObjectsAtIndexes:indexes];
           }
          ```
        - replaceObjectIn<Key>AtIndex:withObject: or replace<Key>AtIndexes:with<Key>: (optional when profiling of your app reveals performance issues.)
    - Accessing Unorfered Collections (unordered collection accessor methods, NSSet or NSMutableSet)
        - Getter
            - countOf<Key>
              ```
              - (NSUInteger)countOfEmployees {
               return [self.employees count];
               }
              ```
            - enumeratorOf<Key> – return NSEnumerator
              ```
              - (NSEnumerator *)enumeratorOfEmployees {
               return [self.employees objectEnumerator];
               }
              ```
            - memberOf<Key>:
              ```
              - (Employee *)memberOfEmployees:(Employee *)anObject {
               return [self.employees member:anObject];
               }
              ```
        - Mutators
            - add<Key>Object: or add<Key>:
              ```
              - (void)addEmployeesObject:(Employee *)anObject {
               [self.employees addObject:anObject];
               }
               - (void)addEmployees:(NSSet *)manyObjects {
               [self.employees unionSet:manyObjects];
               }
              ```
            - remove<Key>Object: or remove<Key>:
              ```
              - (void)removeEmployeesObject:(Employee *)anObject {
               [self.employees removeObject:anObject];
               }
               - (void)removeEmployees:(NSSet *)manyObjects {
               [self.employees minusSet:manyObjects];
               }
              ```
            - intersect<Key>: (optional for performance)
              ```
              - (void)intersectEmployees:(NSSet *)otherObjects {
               return [self.employees intersectSet:otherObjects];
              ```

```
}
```

- Handling Non-object values (only objective-C)
  - ```
    - (void)setNilValueForKey:(NSString *)key
    {
        if ([key isEqualToString:@"age"]) {
            [self setValue:@(0) forKey:@”age”];
        } else {
            [super setNilValueForKey:key];
        }
    }
    ```
- Adding Validation (only objective-C)
  - Note: swift handle by replying compiler support for optionals and strong type checking, while using build-in willSet and didSet to test any runtime.
  - `validateValue:forKey:error:`
- Describing Property Relationship
- Designing for Performance
  - Overriding Key-Value Coding Methods
    - rarely need to override default implementation of key-value coded accessors, such as valueForKey:and setValue:forKey:, or the key-based validation methods like validateValue:forKey:
    - if do override them, make sure invoke default implementation in superclass.
  - Optimizing To-Many Relationship
- Compliance Checklist !!! (https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueCoding/Compliant.html#//apple_ref/doc/uid/20002172-BAJEAIEE)
  - Attribute and To-One Relationship Compliance
    - For each property that is an attribute or a to-one relationship:
      - 1. Implement a method named <key> or is<Key>, or create an instance variable <key> or _<key>. The compiler typically does this for you when it automatically synthesizes properties.
        - NOTE
        - Although property names frequently begin with a lowercase letter, the default implementation of the protocol also works with names that begin with an uppercase letter, such as URL.
      - 2. If the property is mutable, implement the set<Key>: method. The compiler typically does this for you when you allow it to automatically synthesize your properties.
        - IMPORTANT
        - If you override the default setter, be sure not to invoke any of the protocol’s validation methods.
      - 3. If the property is a scalar, override the setNilValueForKey: method to gracefully handle the case where a nil value is assigned to the scalar property.
  - Indexed To-Many Relationship Compliance
    - For each property that is an ordered, to-many relationship (such as an NSArray object):
    ```

            - 1. Implement a method named <key> that returns an array, or have an array instance variable named <key> or _<key>. The compiler typically does this for you when it automatically synthesizes properties.
            - 2. Alternatively, implement the method countOf<Key> and one or both of objectIn<Key>AtIndex: and <key>AtIndexes:.
            - 3. Optionally, implement get<Key>:range: to improve performance.
        - In addition, if the property is mutable:
            - 1. Implement one or both of the methods insertObject:in<Key>AtIndex: and insert<Key>:atIndexes:.
            - 2. Implement one or both of the methods removeObjectFrom<Key>AtIndex: and remove<Key>AtIndexes:.
            - 3. Optionally, implement replaceObjectIn<Key>AtIndex:withObject: or replace<Key>AtIndexes:with<Key>: to improve performance.
            -
    - Unordered To-Many Relationship Compliance
        - For each property that is an unordered, to-many relationship (such as an NSSet object):
            - 1. Implement the <key> that returns a set, or have an NSSet instance variable named <key> or _<key>. The compiler typically does this for you when it automatically synthesizes properties.
            - 2. Alternatively, implement the methods countOf<Key>, enumeratorOf<Key>, and memberOf<Key>:.
        - In addition, if the property is mutable:
            - Implement one or both of the methods add<Key>Object: and add<Key>:.
            - Implement one or both of the methods remove<Key>Object: and remove<Key>:.
            - Optionally, implement intersect<Key>: to improve performance.
    - Validation
        - Opt in to validation for properties that need it:
            - Implement the validate<Key>:error: method, returning a boolean indicating the validity of the value, and a reference to an error object when appropriate.

`.fference between forKey and forKeyPath?`

ɔth methods are part of Key-Value Coding and should not generally be used for element access in dictionaries. They only work on keys of type `NSString` and require specific syntax.

The difference between both of them is that specifying a (single) key would just look up that item.

Specifying a key path on the other hand follows the path through the objects. If you had a dictionary of dictionaries you could look up an element in the second level by using a key path like `"key1.key2"`.

If you just want to access elements in a dictionary you should use `objectForKey:` and `setObject:forKey:`.

Edit to answer why `valueForKey:` should not be used:

`valueForKey:` works only for string keys. Dictionaries can use other objects for keys.
`valueForKey:` Handles keys that start with an "@" character differently. You cannot access elements whose keys start with `@`.
Most importantly: By using `valueForKey:` you are saying: "I'm using KVC". There should be a reason for doing this. When using `objectForKey:` you are just accessing elements in a dictionary through the natural, intended API.

KVO

Useful for communication between model and controller.

- Example:
- ```
  static void *PersonAccountBalanceContext = &PersonAccountBalanceContext;
  static void *PersonAccountInterestRateContext =
  &PersonAccountInterestRateContext;

  - (void)registerAsObserverForAccount:(Account*)account {
      [account addObserver:self
                forKeyPath:@"balance"
                   options:(NSKeyValueObservingOptionNew |
                            NSKeyValueObservingOptionOld)
                   context:PersonAccountBalanceContext];

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

  - (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];
    }
}

- (void)unregisterAsObserverForAccount:(Account*)account {
    [account removeObserver:self
                 forKeyPath:@"balance"
                    context:PersonAccountBalanceContext];

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

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.

- using technique called *isa-swizzling*.

@escaping @autoclosure

用法

此属性关键字用在函数或者方法的闭包参数前面，但是闭包类型被限定在无参闭包上：`() -> T`。例如以下的一个例子：

```
1func doSomeOperation(@autoclosure op: () -> Bool) {
2op()
3}
4// 调用如下：
5doSomeOperation(2 > 3)
6
7
```

作用

看了以上代码，大家估计会比较困惑，这样定义了去掉`@autoclosure`有什么区别？我们先直接抛出`@autoclosure`的全部作用，然后再来分析，有和没有的区别：

使用`@autoclosure`关键字能简化闭包调用形式
使用`@autoclosure`关键字能延迟闭包的执行

关于第一点，大家可能看上面的例子代码，调用带`@autoclosure`的函数形式很自然，其实如果去掉`@autoclosure`，我们就不能这样调用了：

```
1func doSomeOperationWithoutAutoclosure(op: () -> Bool) {
2op()
3}
4doSomeOperationWithoutAutoclosure({2 > 3})
5doSomeOperationWithoutAutoclosure{2 > 3} //尾闭包的简化
6
```

很容易看的出来，使用括号来写起来更加自然。其实如果是尾闭包的形式，也可以接受。只是尾闭包只能是放到参数列表的最后才能这样使用。而`@autoclosure`是可以修饰任何位置的参数：

```
1func doSomeOperationWithTwoAutoclosure(@autoclosure op1: () -> Bool, @autoclosure op2: () ->
2Bool) {
3op1()
4op2()
5}
6doSomeOperationWithTwoAutoclosure(2 > 3, op2: 3 > 2)
```

`@autoclosure`本身取名也有体现出这种语法的意思。直译为自动闭包，也就是会把`(2 > 3)`这样的语法自动转换为闭包执行。

我们再来看看延迟执行这事。其实延迟这个特性，本身不是`@autoclosure`带来的，而是本来闭包本身就带有这样的特性。以上的`op1`和`op2`都是在调用的时候才去执行。

@noescape 和 @escape

对于`autoclosure`属性来讲，还有2个相关的属性不得不提。也就是`@noescape`和`@escape`。

这2个属性都是用来修饰闭包的。`@noescape`意思是非逃逸的闭包，而`@escape`则相反。

默认情况下，闭包是`@escape`的。表示此闭包还可以被其他闭包调用。比如我们常用的异步操作：

```
1func executeAsyncOp(asyncClosure: () -> ()) -> Void {
2dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)) {
3asyncClosure()
4}
5}
```

其中`asyncClosure`在`dispatch_async`中的闭包中调用，完成异步的操作。因为闭包默认是`@escape`的，以上代码是可以运行的。但是当我们在`asyncClosure`前面加入`@noescape`属性时候，编译器就会报错:

```
1closure use of @noesape parameter `asyncClosure` may allow it to escape
```

`@noescape`属性是在 Swift 1.2 中引入的，把传入闭包参数的调用限制在调用的函数体内，对性能有一定的提升，同时将闭包标注为`@noescape`使你能在闭包中隐式地引用self。

在 Swift 标准库中很多方法，都用了`@noescape`属性，比如 Array 对应的方法 map，filter 和 reduce：

```
1func map<T>(@noescape transform: (Self.Generator.Element) -> T) -> [T]
2func filter(@noescape includeElement: (Self.Generator.Element) -> Bool) -> [Self.Generator.Element]
3func reduce<T>(initial: T, @noescape combine: (T, Self.Generator.Element) -> T) -> T
4
5
```

而`@autoclosure`默认是`@noescape`的，要使用逃逸特性，请使用`@autoclosure(escaping)`

用法

官方库中在这些地方用到了`@autoclosure`：

断言, 官方博客的介绍

注意这篇文章很老了，Swift 1.2 中，`@autoclosure`的语法形式已经改变了。最新的`@autoclosure`语法是把`@autoclosure`放到了参数名的前面，之前是放到了参数和类型中间的。
?? 操作符, 参见喵神的文章

同时在项目当中，一些比较耗时的操作，使用函数把操作封装起来，作为闭包传入到处理函数中。这时就可以用到`@autoclosure`, 简化调用的形式，比如以下一个例子：

```
1 /* 在文件读或写操作后，做一些其他操作 */
2 func doSomeOpAfterFileOp(@autoclosure fileOp: () -> Bool) {
3 if fileOp() == true {
4 //做其他操作
5 }
6 }
7 func fileOp() -> Bool {
8 return true
9 }
10doSomeOpAfterFileOp(fileOp())
11
12
```

```
 var completionHandlers: [() -> Void] = []
 func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void)
   {
      completionHandlers.append(completionHandler)
 }

func someFunctionWithNonescapingClosure(closure: () -> Void) {
     closure()
}

class SomeClass {
 var x = 10
 func doSomething() {
     someFunctionWithEscapingClosure { self.x = 100 }
     someFunctionWithNonescapingClosure { x = 200 }
 }
}
let instance = SomeClass()
instance.doSomething()
print(instance.x)
```

```
// Prints “200"
completionHandlers.first?()
print(instance.x)
// Prints “100”

utoclosure default as @noescape, so we need @autoclosure @escaping sometime.
 customersInLine is ["Barry", "Daniella"]
 var customerProviders: [() -> String] = []
 func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () ->
   String) {
 customerProviders.append(customerProvider)
 }
 collectCustomerProviders(customersInLine.remove(at: 0))
 collectCustomerProviders(customersInLine.remove(at: 0))
 print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
 Prints "Now serving Daniella!"
```