iOS Design Pattern

文档修订记录

日期	修订版本	修改描述	作者
2016-07-22	0.1	添加 iOS 设计模式基本知识	Coder4869
2016-07-25	0.2	添加单例模式、代理模式、KVO 部分	Coder4869
2016-07-27	0.3	添加策略模式部分	Coder4869
2016-07-28	0.4	添加工厂模式部分	Coder4869

目 录

1. iOS 设计模式汇总

iOS设计模式包括：单例模式、委托模式(即代理模式)、观察者模式、MVC模式、策略模式、工厂模式等。各模式的特点如下：

1.1 单例模式

应用场景：确保程序运行期某个类，只有一份实例，用于进行资源共享控制。

优势：使用简单，延时求值，易于跨模块。

敏捷原则：单一职责原则

实例：[UIApplication sharedApplication]；

NSNotificationCenter 中的 defaultCenter 负责全局的消息分发；

NSFileManager 的 defaultManager 统一负责物理文件的管理；

NSUserDefaults 的 standardUserDefaults 统一管理用户的配置文件

注意事项：

[1].确保使用者只能通过getInstance方法才能获得，单例类的唯一实例。

[2].java，C++中使其没有公有构造函数，私有化并覆盖其构造函数。

[3].object-c中，重写allocWithZone方法，保证即使用户用alloc方法直接创建单例类的实例，返回的也只是此单例类的唯一静态变量。

1.2 代理模式

应用场景：当一个类的某些功能需要由别的类来实现，但是又不确定具体会是哪个类实现。

优势：解耦合

敏捷原则：开放-封闭原则

实例：tableview的数据源delegate，通过和protocol的配合，完成委托诉求。

1.3 观察者模式

简称为KVO，即Key-Value-Observer，基于KVC（Key-Value-Coding）机制实现的。键值对改变通知的观察者。

应用场景：一般为model层对，controller和view进行的通知方式，不关心谁去接收，只负责发布信息。

优势：解耦合

敏捷原则：接口隔离原则，开放-封闭原则

实例：Notification通知中心，注册通知中心，任何位置可以发送消息，注册观察者的对象可以接收。

1.4 MVC模式

应用场景：是一中非常古老的设计模式，通过数据模型，控制器逻辑，视图展示将应用程序进行逻辑划分。

优势：使系统，层次清晰，职责分明，易于维护

敏捷原则：对扩展开放-对修改封闭

实例：model-即数据模型，view-视图展示，controller进行UI展现和数据交互的逻辑控制。

MVVM是对MVC模式的变异。MVVM是指Model-View-ViewModel。

1.5 策略模式

应用场景：定义算法族，封装起来，使他们之间可以相互替换。

优势：使算法的变化独立于使用算法的用户

敏捷原则：接口隔离原则；多用组合，少用继承；针对接口编程，而非实现。

实例：排序算法，NSArray的sortedArrayUsingSelector；经典的鸭子会叫，会飞案例。

注意事项：

[1].剥离类中易于变化的行为，通过组合的方式嵌入抽象基类

[2].变化的行为抽象基类为，所有可变变化的父类

[3].用户类的最终实例，通过注入行为实例的方式，设定易变行为，防止了继承行为方式，导致无关行为污染子类。完成了策略封装和可替换性。

1.6 工厂模式

应用场景：工厂方式创建类的实例，多与proxy模式配合，创建可替换代理类。

优势：易于替换，面向抽象编程，application只与抽象工厂和易变类的共性抽象类发生调用关系。

敏捷原则：DIP依赖倒置原则

实例：项目部署环境中依赖多个不同类型的数据库时，需要使用工厂配合proxy完成易用性替换。

注意事项：项目初期，软件结构和需求都没有稳定下来时，不建议使用此模式，因为其劣势也很明显，增加了代码的复杂度，增加了调用层次，增加了内存负担。所以要注意防止模式的滥用。

1.7 Reference

iOS开发中的集中设计模式介绍：http://blog.csdn.net/liwei3gjob/article/details/8926862

2. 单例模式

本设计模式的特点参见第一部分。

2.1 示例代码

以下为单例模式常见的两种创建方式。

2.1.1 自定义示例

```
+(ClassName *)sharedInstance {
    static ClassName *instance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[ClassName alloc] init];
    });
    return instance;
}
```

2.1.2 修改allocWithZone方法示例

MRC实现：

```
static ClassName * instance = nil;
+ (ClassName *) sharedInstance
{
    @synchronized(self) {
        if (instance == nil) {
            [[self alloc] init]; // assignment not done here
        }
    }
    return instance;
}
+ (instancetype)allocWithZone:(NSZone *)zone
{
    @synchronized(self) {
        if (instance == nil) {
            instance = [super allocWithZone:zone];
            return instance; // assignment and return on first allocation
        }
    }
    return nil; //on subsequent allocation attempts return nil
}
```

ARC实现：

```
+ (ClassName *) shareInstance
{
    static ClassName * share = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        share = [[super allocWithZone:NULL] init];
```

```
    });
    return share;
}
+ (instancetype) allocWithZone:( NSZone *)zone
{
    return [self shareInstance];
}
```

注：用[[self alloc] init];初始化时，alloc方法会调用allocWithZone方法来完成初始化操作。

可以把**zone**看成一个内存池，alloc，allocWithZone或是dealloc这些操作，都是在这个内存池中操作的。cocoa总是会配置一个默认的NSZone，任何默认的内存操作都基于此。其缺陷是：它是全局范围的，时间一长，必然会导致内存的碎片化，在大量alloc对象时，影响性能。所有cocoa提供方法，可以重写allocWithZone自己生成一个NSZone，并将alloc，copy全部限制在此“zone”之内。

2.2 特点解析

2.2.1 三个要点

一是某个类只能有一个实例；

二是它必须自行创建这个实例；

三是它必须自行向整个系统提供这个实例。

2.2.2 优缺点

[1]. 实例控制

Singleton 会阻止其他对象实例化其自己的 Singleton 对象的副本，从而确保所有对象都访问唯一实例。

[2]. 灵活性

因为类控制了实例化过程，所以类可以更加灵活修改实例化过程

[3]. 使用dispatch_once的好处（ARC）

使用dispatch_once体带原来的加锁操作，可以减少每次加锁的时间，优化了程序性能。dispatch_once函数接收一个dispatch_once_t用于检查该代码块是否已经被调度的谓词（是一个长整型，实际上作为BOOL使用）。它还接收一个希望在应用的生命周期内仅被调度一次的代码块，对于本例就用于share实例的实例化。dispatch_once不仅意味着代码仅会被运行一次，而且还是线程安全的。完全可以替代相对低效的加锁操作。

[4]. 使用synchronized的缺点（MRC）

只能保证线程内的安全，不能保证线程间的安全。多线程访问时，可能会造成多个实例的创建，比如线程A和线程B同时访问，在A进入创建流程，但是尚未创建完成时(instance == nil)，B访问单例，可能会创建两个单例。

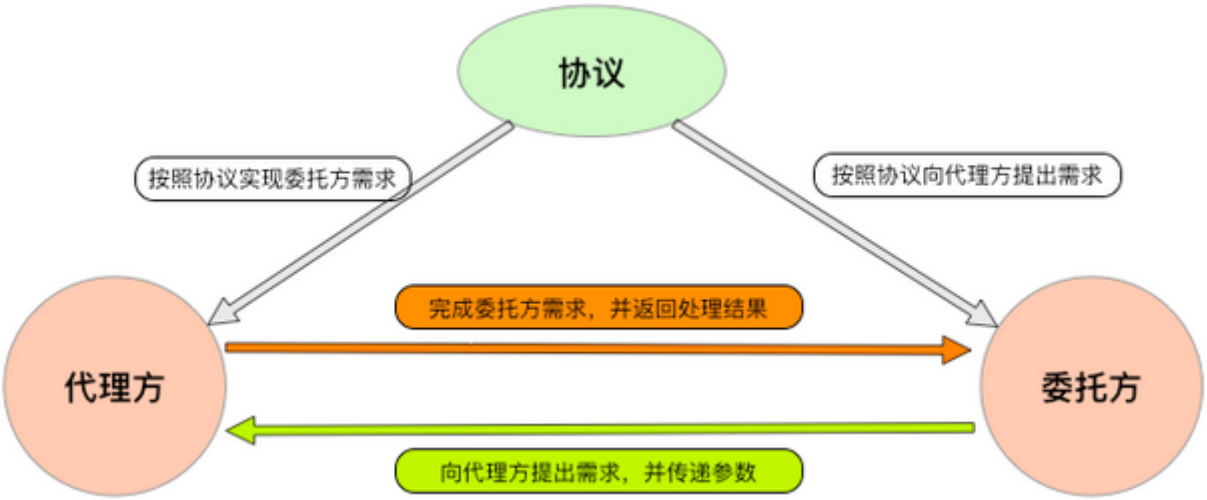
3. 代理模式

是一种通用的设计模式，iOS中对代理支持的很好，由代理对象、委托者、协议三部分组成。OC中用@protocol实现协议。

[1]. 协议：用来指定代理双方可以做什么，必须做什么。

[2]. 委托：根据指定的协议，指定代理去完成什么功能。

[3]. 代理：根据指定的协议，完成委托方需要实现的功能。



经常遇到两种类型的代理：在cocoa框架中的Delegate模式与自定义的委托模式。

3.1 协议-Protocol

协议规定代理双方的行为，协议中的内容一般都是方法列表，当然也可以定义属性。协议是公共的定义，如果只是某个类使用，一般将其写在某个类中。如果是多个类公用同一协议，一般创建专门的Protocol定义文件。遵循的协议可以被继承，例如UITableView继承自UIScrollView，便也继承了UIScrollViewDelegate，可以通过代理方法获取UITableView偏移量等状态参数。

协议与java中的interface类似。它只提供一套公用接口定义，接口的实现由代理对象完成。在iOS中，对象不支持多继承，而协议可以多继承。一个代理可以有多个委托方，而一个委托方也可以有多个代理。

协议有两个修饰符@optional和@required，默认为@required状态。这两个修饰符只是约定代理是否强制需要遵守协议，如果@required状态的方法代理没有遵守，会报一个黄色的警告，只是起一个约束的作用，没有其他功能。无论是@optional还是@required，在委托方调用代理方法时都需要做一个判断，判断代理是否实现当前方法，否则会导致崩溃。示例如下：

```
if ([self.delegate respondsToSelector:@selector(userLoginWithUsername:password:)]) {
    [self.delegate userLoginWithUsername:self.username.text password:self.password.text];
}
```

3.2 示例讲解

此部分以自定义的委托模式为例进行说明。

3.2.1 协议的定义与使用

```
@protocol DemoDelegate <NSObject>    //协议
```

```
- (void) demoDelegateMethod: (NSString*)fromValue; //含一个参数的协议方法
@end

@interface A：NSObject //协议委托者
@property(nonatomic, weak) id <DemoDelegate> delegate; //代理属性
@end

@implementation A
-(void)Call
{
    NSString* value = @"你好";
    [self.delegate demoDelegateMethod:value];   //代理属性调用协议方法
}
@end
```

3.2.2 Delegate方法的实现

```
@interface B：UIView < DemoDelegate > {   //代理对象，设置代理对象的代理资格
    NSString* value;
}
@end

@implementation B
- (void) demoDelegateMethod: (NSString*)fromValue   //实现协议方法
{
    value = fromValue;
    NSLog(@"%@ ,我是协议代理人",value);
}
@end
```

3.3 原理讲解

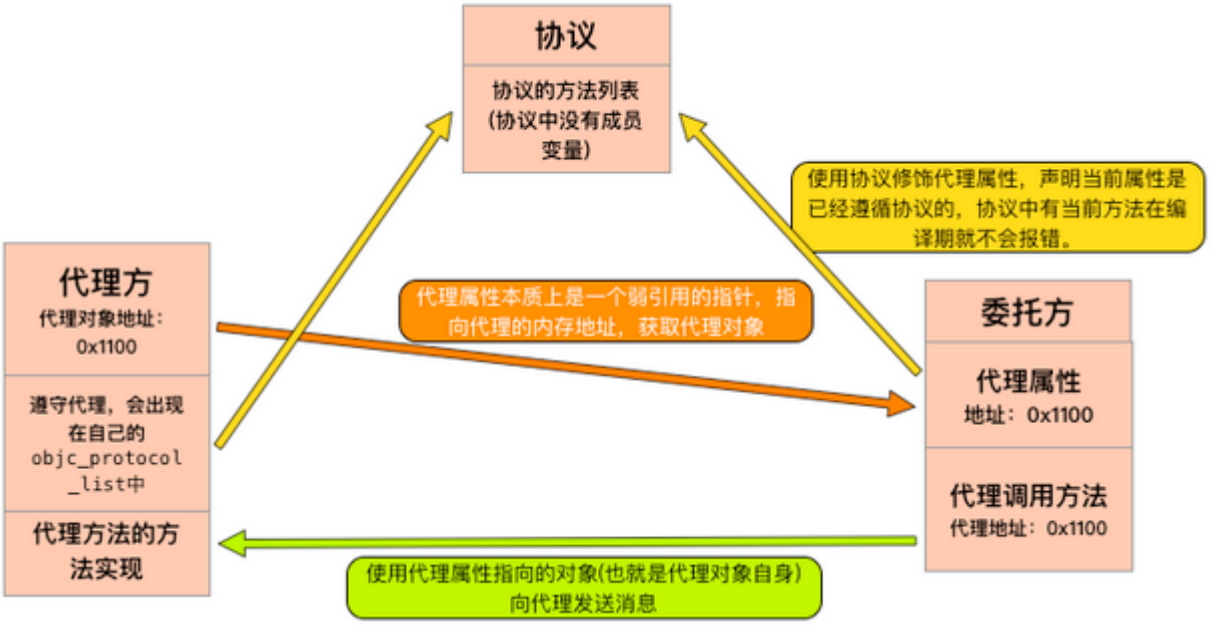
3.3.1 代理实现流程

在iOS中代理的本质就是代理对象内存的传递和操作，在委托类设置代理对象后，实际上只是用一个id类型的指针将代理对象进行了一个弱引用。委托方让代理方执行操作，实际上是在委托类中向这个id类型指针指向的对象发送消息，而这个id类型指针指向的对象，就是代理对象。

委托方的代理属性本质上就是代理对象自身，设置委托代理就是代理属性指针指向代理对象，相当于代理对象只是在委托方中调用自己的方法，如果方法没有实现就会导致崩溃。从崩溃的信息上来看，就可以看出来是代理方没有实现协议中的方法导致的崩溃。

而协议只是一种语法，是声明委托方中的代理属性可以调用协议中声明的方法，而协议中方法的实现还是有代理方完成，而协议方和委托方都不知道代理方有没有完成，也不需要知道怎么完成。

原理图如下：



3.3.2 代理内存管理

[1]. 为何设置代理属性使用weak?

如果用strong类型，有可能会造成循环引用。强引用示例图如下：

上图中，由于代理对象使用强引用指针，引用创建的委托方LoginVC对象，并且成为LoginVC的代理。这就会导致LoginVC的delegate属性强引用代理对象，导致循环引用的问题，最终两个对象都无法正常释放。

如果将LoginVC对象的delegate属性，设置为弱引用属性。在代理对象生命周期存在时，可以正常工作，如果代理对象被释放，委托方和代理对象都不会因为内存释放导致的Crash。

[2]. 两种弱引用方式

下面两种方式都是弱引用代理对象，但是第一种在代理对象被释放后不会导致崩溃，而第二种会导致崩溃。

@property (nonatomic, weak) id delegate;

@property (nonatomic, assign) id delegate;

weak和assign是一种“非拥有关系”的指针，通过这两种修饰符修饰的指针变量，都不会改变被引用对象的引用计数。但是在一个对象被释放后，weak会自动将指针指向nil，而assign

则不会。在iOS中，向nil发送消息时不会导致崩溃的，所以assign就会导致野指针的错误unrecognized selector sent to instance。在修饰代理属性时，建议用weak修饰，比较安全。

3.4 代理对象作用-控制器瘦身

3.4.1 控制器瘦身

项目的复杂化增加了业务增加，导致控制器的臃肿。MVVM模式不适合架构已确定的大中型项目。UITableView等控件的处理逻辑增加，也会增加控制器臃肿。可以使用代理对象方式对控制器瘦身。方案如下：

将UITableView的delegate和DataSource单独拿出来，由一个代理对象类进行控制，只将必须控制器处理的逻辑传递给控制器处理。

UITableView的数据处理、展示逻辑和简单的逻辑交互都由代理对象去处理，和控制器相关的逻辑处理传递出来，交由控制器来处理，这样控制器的工作少了很多，而且耦合度也大大降低了。这样一来，只需要将需要处理的工作交由代理对象处理，并传入一些参数即可。

使用代理对象类还有一个好处，就是如果多个UITableView逻辑一样或类似，代理对象是可以复用的。

3.5 非正式协议

3.5.1 简介

在iOS2.0之前尚未引入@Protocol正式协议，实现协议的功能主要是通过给NSObject添加Category的方式。这种通过Category的方式，相对于iOS2.0之后引入的@Protocol，就叫做非正式协议。

非正式协议一般都是以NSObject的Category的方式存在的。由于是对NSObject进行的Category，所以所有基于NSObject的子类，都接受了所定义的非正式协议。对于@Protocol来说编译器会在编译期检查语法错误，而非正式协议则不会检查是否实现。

非正式协议中没有@Protocol的@optional和@required之分，和@Protocol一样在调用的时候，需要进行判断方法是否实现。

```
//由于是使用的Category，所以需要用self来判断方法是否实现
if ([self respondsToSelector:@selector(userLoginWithUsername:password:)]) {
    [self userLoginWithUsername:self.username.text password:self.password.text];
}
```

3.5.2 非正式协议示例

在iOS早期也使用了大量非正式协议，例如CALayerDelegate就是非正式协议的一种实现，非正式协议本质上就是Category。

```
@interface NSObject (CALayerDelegate)
- (void)displayLayer:(CALayer *)layer;
- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)ctx;
- (void)layoutSublayersOfLayer:(CALayer *)layer;
- (nullable id)actionForLayer:(CALayer *)layer forKey:(NSString *)event;
@ends
```

3.6 代理和block的选择

多个消息传递，应该使用delegate。在有多个消息传递时，用delegate实现更合适，看起来

也更清晰。block就不太好了，这个时候block反而不便于维护，而且看起来非常臃肿，很别扭。例如：UIKit的UITableView中有很多代理，如果都换成block实现，那简直看起来不能忍受。

一个委托对象的代理属性只能有一个代理对象（delegate只是一个保存某个代理对象的地址，如果设置多个代理相当于重新赋值，只有最后一个设置的代理才会被真正赋值。），如果想要委托对象调用多个代理对象的回调，应该用block。

单例对象最好不要用delegate。单例对象由于始终都只是同一个对象，如果使用delegate，就会造成delegate属性被重新赋值的问题，最终只能有一个对象可以正常响应代理方法。

这种情况可以使用block的方式，在主线程的多个对象中使用block都是没问题的。在多线程情况下因为是单例对象，我们对block中必要的地方加锁，防止资源抢夺的问题发生。

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
queue.maxConcurrentOperationCount = 10;   //设置最大并发数
for (int i = 0; i < 100; i++) {
    [queue addOperationWithBlock:^{
        [[LoginViewController shareInstance] userLoginWithSuccess:^(NSString *username) {
            NSLog(@"TestTableViewController : %d", i);
        }];
    }];
}
```

- 代理是可选的，而block在方法调用的时候只能通过将某个参数传递一个nil进去，只不过这并不是什么大问题，没有代码洁癖的可以忽略。

```
[self downloadTaskWithResumeData:resumeData
                  sessionManager:manager
                        savePath:savePath
                   progressBlock:nil
                    successBlock:successBlock
                    failureBlock:failureBlock];
```

- 从设计模式的角度来说，代理更加面向过程，block则更面向结果。例如用于XML解析的NSXMLParserDelegate代理，它有很多代理方法，NSXMLParser会不间断调用这些方法将一些转换的参数传递出来，这就是NSXMLParser解析流程，这些通过代理来展现比较合适。而例如一个网络请求回来，就通过success、failure代码块来展示就比较好。
- 从性能上来说，block的性能消耗要略大于delegate，因为block会涉及到栈区向堆区拷贝等操作，时间和空间上的消耗都大于代理。而代理只是定义了一个方法列表，在遵守协议对象的objc_protocol_list中添加一个节点，在运行时向遵守协议的对象发送消息即可。

3.7 Reference

Cocachina刘小壮：http://www.cocoachina.com/ios/20160317/15696.html

唐巧block：http://blog.devtang.com/2013/07/28/a-look-inside-blocks/

4. 观察者模式

观察者模式简称为KVO，即Key-Value-Observer，基于KVC（Key-Value-Coding）机制实现的。键值对改变通知的观察者。只有当调用KVC去访问key值的时候KVO才会起作用。

4.1 KVC(Key-Value-Coding)机制

KVC，即是指NSKeyValueCoding，一个非正式的Protocol，提供一种机制来间接访问对象的属性。

KVC主要通过isa-swizzling(类型混合指针机制)技术，来实现其内部查找定位的。isa指针（就是is a kind of的意思）指向维护分发表的对象的类。该分发表实际上包含了指向实现类中的方法的指针，和其它数据。

示例代码：[site setValue:@"sitename" forKey:@"name"];

编译器处理结果：

```
SEL sel = sel_get_uid ("setValue:forKey:");   //根据方法名字获取SEL
IMP method = objc_msg_lookup (site->isa, sel);   //GNU运行时获取IMP
method(site, sel, @"sitename", @"name");
```

一个对象在调用setValue的时候，KVC内部的实现：

[1]. 首先根据方法名找到运行方法的时候所需要的环境参数。

[2]. 他会从自己isa指针结合环境参数，找到具体的方法实现的接口。

[3]. 再直接查找得来的具体的方法实现。

补充说明：

（1）SEL数据类型：它是编译器运行Objective-C里的方法的环境参数。

（2）IMP数据类型：他其实就是一个编译器内部实现时候的函数指针。当Objective-C编译器去处理实现一个方法的时候，就会指向一个IMP对象，这个对象是C语言表述的类型（事实上，在Objective-C的编译器处理的时候，基本上都是C语言的）。

4.2 KVO(Key-Value-Observing)机制

KVC机制上加上KVO的自动观察消息通知机制就是KVO的实现机制。

当观察者为一个对象的属性进行了注册，被观察对象的isa指针被修改的时候，isa指针就会指向一个中间类，而不是真实的类。所以isa指针其实不需要指向实例对象真实的类。所以我们的程序最好不要依赖于isa指针。在调用类的方法的时候，最好要明确对象实例的类名。

因为KVC的实现机制，可以很容易看到某个KVC操作的Key，而后也很容易的跟观察者注册表中的Key进行匹对。假如访问的Key是被观察的Key，则在内部就可以很容易的到观察者注册表中去找到观察者对象，而后给他发送消息。KVO的三种用法：

[1]. 使用KVC

如果有访问器方法，则运行时会在访问器方法中调用will/didChangeValueForKey:方法；

没用访问器方法，运行时会在setValue:forKey方法中调用will/didChangeValueForKey:方法。

[2]. 有访问器方法

运行时会重写访问器方法调用will/didChangeValueForKey:方法。因此，直接调用访问器方法改变属性值时，KVO也能监听到。

[3]. 显示调用will/didChangeValueForKey:方法。

4.3 案例分析

4.3.1 KVC

Person对象有name和address两个属性。以 KVC 说法，Person 对象分别有一个 value 对应他的 name 和 address 的 key。 key 只是一个字符串，它对应的值可以是任意类型的对象。从最基础的层次上看，KVC 有两个方法：一个是设置 key 的值，另一个是获取 key 的值。如下面的例子：

```
void changeName(Person *p, NSString *newName)
{
    // using the KVC accessor (getter) method
    NSString *originalName = [p valueForKey:@"name"];

    // using the KVC accessor (setter) method.
    [p setValue:newName forKey:@"name"];

    NSLog(@"Changed %@'s name to: %@", originalName, newName);
}
```

如果 Person 有一个 key 配偶（spouse），spouse 的 key 值是另一个 Person 对象，用 KVC 可以这样写：

```
void logMarriage(Person *p)
{
    // just using the accessor again, same as example above
    NSString *personsName = [p valueForKey:@"name"];

    // this line is different, because it is using
    // a "key path" instead of a normal "key"
    NSString *spousesName = [p valueForKeyPath:@"spouse.name"];

    NSLog(@"%@ is happily married to %@", personsName, spousesName);
}
```

key 与 key pat 要区分开来，key 可以从一个对象中获取值，而 key path 可以将多个 key 用点号 “.” 分割连接起来，比如：

```
[p valueForKeyPath:@"spouse.name"]; 等价于
[[p valueForKey:@"spouse"] valueForKey:@"name"];
```

4.3.2 KVO

用代码观察一个 person 对象的 address 变化，以下是实现的三个方法：

- watchPersonForChangeOfAddress: 实现观察
- observeValueForKeyPath:ofObject:change:context: 在被观察的 key path 的值变化时调用。
- dealloc 停止观察

```
static NSString *const KVO_CONTEXT_ADDRESS_CHANGED =
@"KVO_CONTEXT_ADDRESS_CHANGED"
```

```
@implementation PersonWatcher

-(void) watchPersonForChangeOfAddress:(Person *)p
{
    // this begins the observing
    [p addObserver:self
        forKeyPath:@"address"
           options:0
           context:KVO_CONTEXT_ADDRESS_CHANGED];

    // keep a record of all the people being observed,
    // because we need to stop observing them in dealloc
    [m_observedPeople addObject:p];    // m_observedPeople为NSMutableArray数组
}

// whenever an observed key path changes, this method will be called
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    // use the context to make sure this is a change in the address,
    // because we may also be observing other things
    if(context == KVO_CONTEXT_ADDRESS_CHANGED) {
        NSString *name = [object valueForKey:@"name"];
        NSString *address = [object valueForKey:@"address"];
        NSLog(@"%@ has a new address: %@", name, address);
    }
}

-(void) dealloc;
{
    // must stop observing everything before this object is deallocated to avoid crashes
    for(Person *p in m_observedPeople) {
        [p removeObserver:self forKeyPath:@"address"];
    }
    m_observedPeople = nil;
}
```

4.4 Reference

KVC, KVO实现原理剖析：http://www.jianshu.com/p/37a92141077e

KVC 与 KVO 理解：http://magicalboy.com/kvc_and_kvo/

5. 策略模式(Strategy)

策略模式：定义一系列的算法，把每一个算法封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。也称为政策模式 (Policy)。

策略模式把对象本身和运算规则区分开来，其功能非常强大，因为这个设计模式本身的核心思想就是面向对象编程的多形性的思想。

敏捷原则：接口隔离原则；多用组合，少用继承；针对接口编程，而非实现。

注意事项：

[1].剥离类中易于变化的行为，通过组合的方式嵌入抽象基类

[2].变化的行为抽象基类为，所有可变变化的父类

[3].用户类的最终实例，通过注入行为实例的方式，设定易变行为，防止了继承行为方式，导致无关行为污染子类。完成了策略封装和可替换性。

5.1 适用场景

[1]. 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。即一个系统需要动态地在几种算法中选择一种。

[2]. 需要使用一个算法的不同变体。例如，定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。

[3]. 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

[4]. 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

5.2 优缺点分析

5.2.1 优点

[1]. 相关算法系列

Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。

[2]. 提供了可以替换继承关系的办法

继承提供了另一种支持多种算法或行为的方法。可以直接生成一个Context类的子类，从而给它以不同的行为。但这会将行为硬行编制到Context中，而将算法的实现与Context的实现混合起来，从而使Context难以理解、难以维护和难以扩展，而且还不能动态地改变算法。最后得到一堆相关的类，它们之间的唯一差别是它们所使用的算法或行为。将算法封装在独立的Strategy类中使得你可以独立于其Context改变它，使它易于切换、易于理解、易于扩展。

[3]. 消除了一些if else条件语句

Strategy模式提供了用条件语句选择所需的行为以外的另一种选择。当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的Strategy类中消除了这些条件语句。含有许多条件语句的代码通常意味着需要使用Strategy模式。

[4]. 实现的选择

Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间/空间权衡取舍要求从不同策略中进行选择。

5.2.2 缺点

[1]. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类

客户要选择一个合适的Strategy就必须知道各Strategy的不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时，才需要使用Strategy模式。

[2]. Strategy和Context之间的通信开销

无论各Strategy实现的算法是简单还是复杂，它们都共享Strategy定义的接口。有时，某Strategy不会用到接口传入的全部信息，这使得有时Context会创建和初始化一些永远不会用到的参数。如果存在这样问题，那么将需要在Strategy和Context之间更进行紧密的耦合。

[3]. 策略模式将造成产生很多策略类

可以通过使用享元模式在一定程度上减少对象的数量。增加了对象的数目Strategy增加了一个应用中的对象的数目。有时你可以将 Strategy实现为可供各Context共享的无状态的对象来减少这一开销。任何其余的状态都由 Context维护。Context在每一次对Strategy对象的请求中都将这个状态传递过去。共享的 Strategy不应在各次调用之间维护状态。

5.3 案例分析

5.3.1 案例

在验证用户输入的表单的时候，加入包括电话输入框的验证和邮件输入框的验证，这两部分的验证算法是不同的，如果把这个算法看成一个函数，他几乎有相同的输入参数和返回参数。

5.3.2 策略

[1]. 把公共函数抽象为基类(InputValidator)的一个方法(bool validateInput(input,error))

[2]. 抽象出两个具体的策略类：电话验证类(PhoneValidator)和邮件验证类(EmailValidator)。策略类在各自的实现里面去复写父类的验证方法。

5.3.3 示例代码

接口方法定义：

```
@protocol ValidatorInterface <NSObject>
-(BOOL) validateInput:(id)input error:(NSError**)error;
@end
```

PhoneValidator类定义：

```
@interface PhoneValidator:NSObject <ValidatorInterface>
@end
@implementation PhoneValidator
-(BOOL) validateInput:(id)input error:(NSError**)error
{
    NSLog(@"Phone Validator Action!");
    return true;
}
@end
```

EmailValidator类定义：

```
@interface EmailValidator:NSObject< ValidatorInterface >
@end
@implementation EmailValidator
```

```
-(BOOL) validateInput:(id)input error:(NSError**)error
{
    NSLog(@"Email Validator Action!");
    return true;
}
@end
```

InputValidator类定义：

```
@interface InputValidator:NSObject
{
    @public
    id< ValidatorInterface > validator;
}
-(void)setValidator:(id<ValidatorInterface >)myValidator;
-(BOOL) validateInput:(id)input error:(NSError**)error;
@end
@implementation InputValidator
-(void)setValidator:(id<ValidatorInterface >)myValidator {
     validator = myValidator;
}
-(BOOL) validateInput:(id)input error:(NSError**)error
{
    NSLog(@"Input Validator Action!");
    [validator validateInput:input error:error];
    return true;
}
@end
```

调用案例：

```
InputValidator * IV = [[InputValidator alloc] init];
[IV setValidator:[[EmailValidator alloc] init]];
//[IV setValidator:[[PhoneValidator alloc] init]];
[IV validateInput:nil error:nil];
```

5.3.4 其他实例

排序算法，NSArray的sortedArrayUsingSelector；

经典的鸭子会叫，会飞案例。

5.4 Reference

iOS设计模式 - (4)策略模式：http://blog.csdn.net/hitwhylz/article/details/40583525

6. 工厂模式

工厂模式包括简单工厂、工厂方法、抽象工厂三种模式。

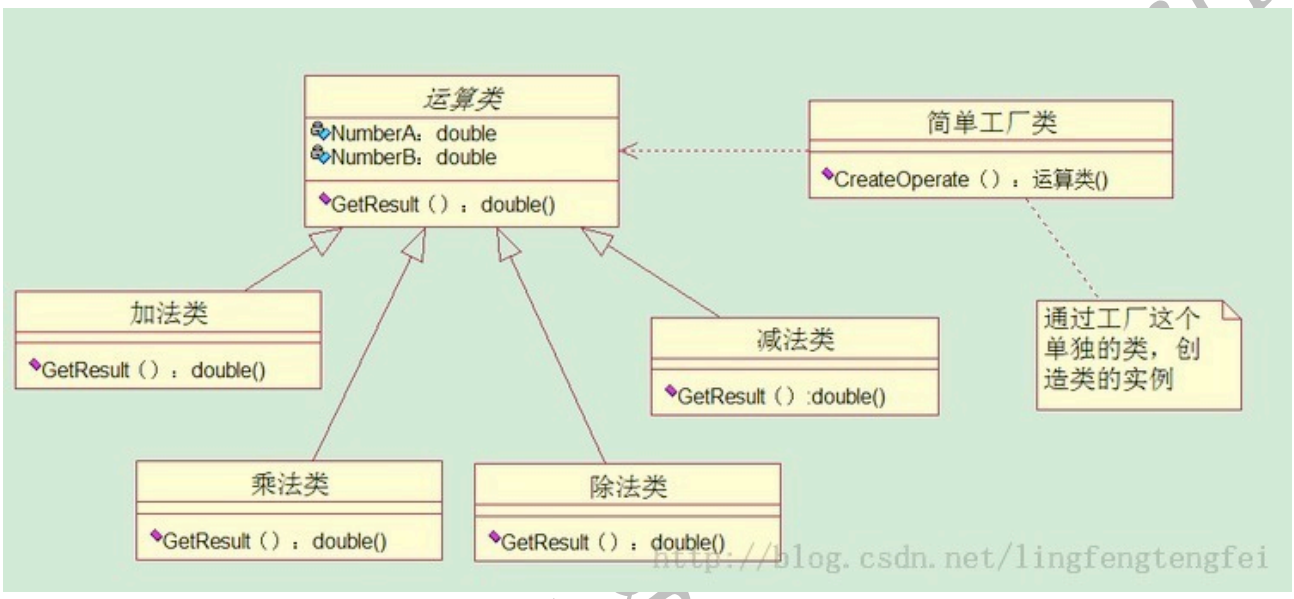
6.1 简单工厂

实现了算法和界面的分离，也就是将业务逻辑和界面逻辑分开了，降低了耦合度。

算法的封装：定义一个抽象的算法接口，提供不同算法的公共接口方法。其他具体算法继承这个抽象类，并实现具体的算法。

简单工厂类：作为一个独立的类，实现了针对不同的算法进行实例化。

简单工厂的UML图：



简单工厂模式参与者：

[1]. 工厂(Factory)角色：接受客户端的请求，通过请求负责创建相应的产品对象。

[2]. 抽象产品(Abstract Product)角色：是工厂模式所创建对象的父类或是共同拥有的接口。可是抽象类或接口。

[3]. 具体产品(ConcreteProduct)对象：工厂模式所创建的对象都是这个角色的实例。

简单工厂模式的演变：

[1]. 当系统中只有唯一的产品时，可以省略抽象产品。即：工厂角色与具体产品可以合并。

简单工厂模式的优缺点：

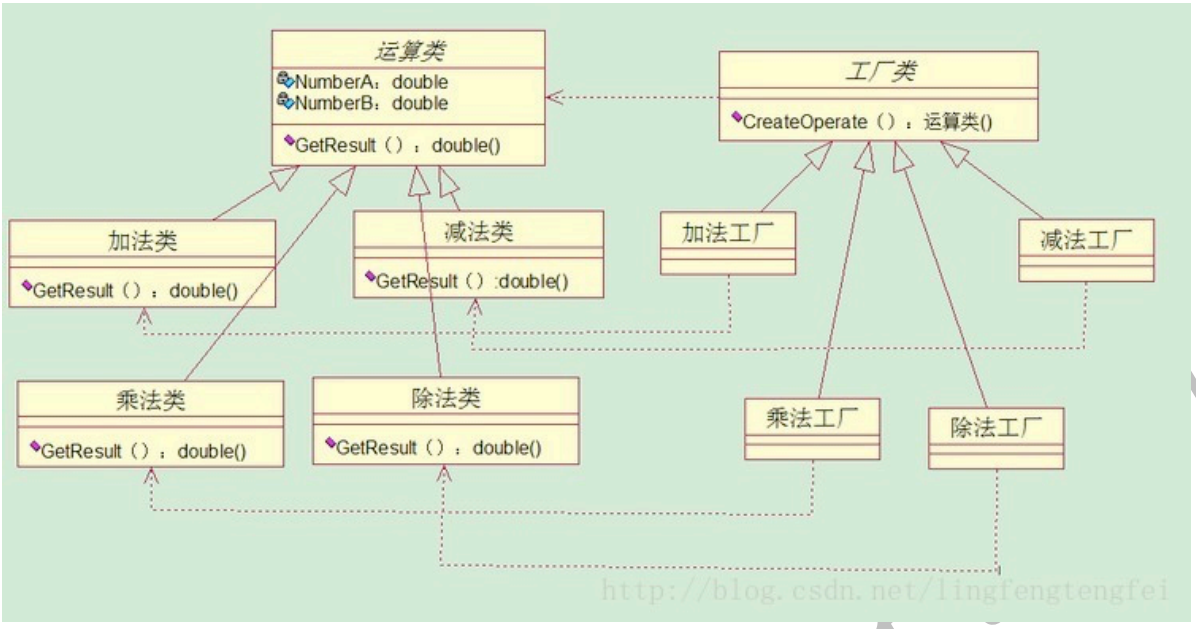
[1]. 工厂类含有必要的创建何种产品的逻辑，这样客户端只需要请求需要的产品，而不需要理会产品的实现细节。

[2]. 工厂类只有一个，它集中了所有产品创建的逻辑，它将是整个系统的瓶颈，同时造成系统难以拓展。

[3]. 简单工厂模式通常使用静态工厂方法，这使得工厂类无法由子类继承，这使得工厂角色无法形成基于继承的等级结构。

6.2 工厂方法

工厂方法使用OOP的多态性，将工厂和产品都抽象出一个基类，在基类中定义统一的接口，然后在具体的工厂中创建具体的产品。即：定义一个用于创建对象的接口，让子类决定实例化哪一个类。遵循了“开放-封闭”原则(即开放接口，封闭修改)。工厂模式的UML图：



工厂方法的参与者：

[1]. 抽象工厂角色：与应用程序无关，任何在模式中创建对象的工厂必须实现这个接口。

[2]. 具体工厂角色：实现了抽象工厂接口的具体类，含有与引用密切相关的逻辑，并且受到应用程序的调用以创建产品对象。

[3]. 抽象产品角色：工厂方法所创建产品对象的超类型，也就是产品对象的共同父类或共同拥有的接口。

[4]. 具体产品角色：这个角色实现了抽象产品角色所声名的接口。工厂方法所创建的每个具体产品对象都是某个具体产品角色的实例。

简单工厂与工厂模式的共同优点：

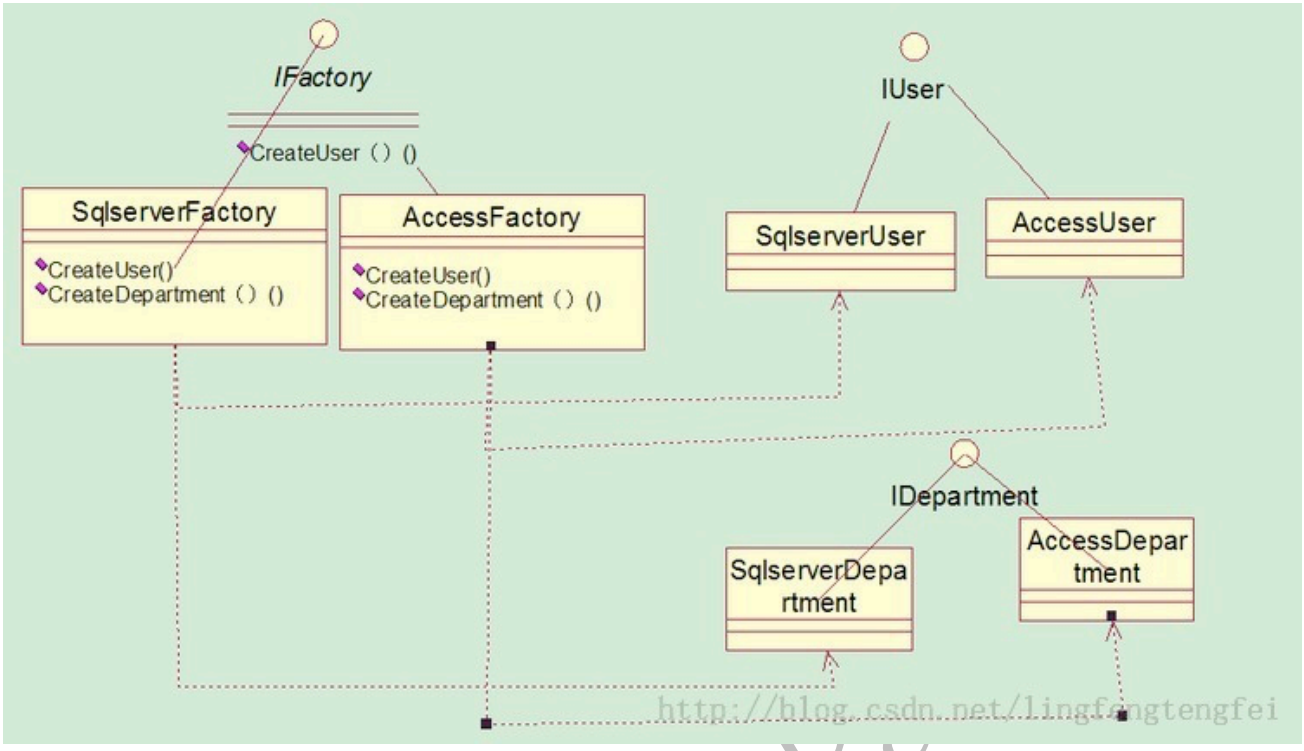
都集中封装了对象的创建，使得要更换对象时不需要做大的改动就可实现，降低了客户端程序与产品对象的耦合。

工厂模式PK简单工厂模式表

模式	简单工厂模式	工厂模式
优点	工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。	简单工厂模式的进一步抽象和推广。遵循“开放-封闭”原则。降低了工厂类的内聚，满足了类之间的层次关系，又很好的符合了面向对象设计中的单一职责原则，有利于程序的拓展。
缺点	没遵守“开放-封闭”原则。如果要添加开放的算法，需要在简单工厂类中添加相应的判断语句。在简单工厂类中利用了Switch语句，不利于程序的扩展。	工厂方法把简单工厂的内部逻辑判断转移到了客户端代码来执行；每增加一产品就要增加一个产品工厂的类，增加了额外的开发量。

6.3 抽象工厂

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。即：一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。UML结构图：



只有一个User类和User操作类的时候，只需要工厂方法模式；但数据库中有很多的表，而Sql和access又是两大不同的分类，所以就延伸到了抽象工厂模式

抽象工厂中的参与者：

[1]. 抽象工厂（Abstract Factory）角色：担任这个角色的是工厂方法模式的核心，它是与应用系统商业逻辑无关的。

[2]. 具体工厂（Concrete Factory）角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。

[3]. 抽象产品（Abstract Product）角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。

[4]. 具体产品（Concrete Product）角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统的商业逻辑。

抽象工厂模式的优点：

[1]. 易于交换产品系列，由于具体工厂类，在一个应用程序中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需改变具体工厂即可使用不同的产品配置。

[2]. 它让具体的创建实例过程与客户端分离，客户端是通过他们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户端代码中。

抽象模式的缺点：

[1]. 抽象模式虽然便于两数据库之间的切换，但是不便于增加需求功能。

[2]. 如果有100个调用数据库访问的类，就需要多次实例化100此具体工厂类。

抽象工厂模式、反射以及配置文件：

反射方法的实质是在对象实例化的时候传引用，将程序由编译时转为运行时，通过字符串变量来处理，去除了、switch判断的麻烦。但是如果数据库在更换时，还需要去修改程序（字符串的值）重编译。

通过添加配置文件可以解决更改DataAccess的问题。

抽象工厂的使用场景：

一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。

这个系统有多于一个的产品族，而系统只消费其中某一产品族。

同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。

系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

抽象工厂＝多个工厂方法＋各工厂方法之间的关系。

6.4 示例代码

可以用策略模式部分的示例代码来理解工厂方法模式。

6.5 Reference

http://lvxingzhelimin.blog.163.com/blog/static/17071655020111085295654 2/

http://blog.csdn.net/casablaneca/article/details/39851457

7. MVC 模式

应用场景：是一中非常古老的设计模式，通过数据模型，控制器逻辑，视图展示将应用程序进行逻辑划分。

优势：使系统，层次清晰，职责分明，易于维护

敏捷原则：对扩展开放-对修改封闭

实例：model-即数据模型，view-视图展示，controller进行UI展现和数据交互的逻辑控制。

MVVM是对MVC模式的变异。MVVM是指Model-View-ViewModel。