设计模式

从设计模式的角度分析Delegate、Notification、KVO的区别

三者优缺点：

delegate的优势：

1.非常严格的语法。所有将听到的事件必须是在delegate协议中有清晰的定义。

2.如果delegate中的一个方法没有实现那么就会出现编译警告/错误

3.协议必须在controller的作用域范围内定义

4.在一个应用中的控制流程是可跟踪的并且是可识别的；

5.在一个控制器中可以定义定义多个不同的协议，每个协议有不同的delegates

6.没有第三方对象要求保持/监视通信过程。

7.能够接收调用的协议方法的返回值。这意味着delegate能够提供反馈信息给controller

8.经常被用在存在父子关系的对象之间通信，例如控制器和控制器的view（自己加的理解）

缺点：

1.需要定义很多代码：1.协议定义；2.controller的delegate属性；3.在delegate本身中实现delegate方法定义

2.在释放代理对象时，需要小心的将delegate改为nil。一旦设定失败，那么调用释放对象的方法将会出现内存crash

3.在一个controller中有多个delegate对象，并且delegate是遵守同一个协议，但还是很难告诉多个对象同一个事件，不过有可能。

4.经常用在一对一的通信。（不知道是缺点还是优点，只能算是特点）（自己加的理解）

notification的优势：

1.不需要编写多少代码，实现比较简单

2.对于一个发出的通知，多个对象能够做出反应，即一对多的方式实现简单

3.controller能够传递context对象（dictionary），context对象携带了关于发送通知的自定义的信息

缺点：

1.在编译期不会检查通知是否能够被观察者正确的处理；

2.在释放注册的对象时，需要在通知中心取消注册；

3.在调试的时候应用的工作以及控制过程难跟踪；

4.需要第三方对象来管理controller与观察者对象之间的联系；

5.controller和观察者需要提前知道通知名称、UserInfo dictionary keys。如果这些没有在工作区间定义，那么会出现不同步的情况；

6.通知发出后，controller不能从观察者获得任何的反馈信息（相比较delegate）。

KVO的优势：

1.能够提供一种简单的方法实现两个对象间的同步。例如：model和view之间同步；

2.能够对非我们创建的对象，即内部对象的状态改变作出响应，而且不需要改变内部对象（SKD对象）的实现；

3.能够提供观察的属性的最新值以及先前值；

4.用key paths来观察属性，因此也可以观察嵌套对象；

5.完成了对观察对象的抽象，因为不需要额外的代码来允许观察值能够被观察

6.可以一对多。

缺点：

1.我们观察的属性必须使用strings来定义。因此在编译器不会出现警告以及检查；

2.对属性重构将导致我们的观察代码不再可用；

3.复杂的“IF”语句要求对象正在观察多个值。这是因为所有的观察代码通过一个方法来指向；

4.当释放观察者时不需要移除观察者。

1. 效率 肯定是delegate比NSNotification高。

delegate方法比notification更加直接，最典型的特征是，delegate方法往往需要关注返回值，

也就是delegate方法的结果。比如-windowShouldClose:，需要关心返回的是yes还是no。所以delegate方法往往包含

should这个很传神的词。也就是好比你做我的delegate，我会问你我想关闭窗口你愿意吗？你需要给我一个答案，我根据你的答案来决定如何做下一

步。相反的，notification最大的特色就是不关心接受者的态度，

我只管把通告放出来，你接受不接受就是你的事情，同时我也不关心结果。所以notification往往用did这个词汇，比如

NSWindowDidResizeNotification，那么NSWindow对象放出这个notification后就什么都不管了也不会等待接

受者的反应。

2、KVO和NSNotification的区别：

和delegate一样，KVO和NSNotification的作用也是类与类之间的通信，与delegate不同的是1）这两个都是负责发出通知，剩下的事情就不管了，所以没有返回值；2）delegate只是一对一，而这两个可以一对多。这两者也有各自的特点。

总结:

从上面的分析中可以看出3中设计模式都有各自的优点和缺点。在这三种模式中，我认为KVO有最清晰的使用案例，而且针对某个需求有清晰的实用性。而另外两种模式有比较相似的用处，并且经常用来给controller间进行通信。那么我们在什么情况使用其中之一呢?

根据我开发iOS应用的经历，我发现有些过分的使用通知模式。我个人不是很喜欢使用通知中心。我发现用通知中心很难把握应用的执行流程。UserInfo dictionaries的keys到处传递导致失去了同步，而且在公共空间需要定义太多的常量。对于一个工作于现有的项目的开发者来说，如果过分的使用通知中心，那么很难理解应用的流程。

我觉得使用命名规则好的协议和协议方法定义对于清晰的理解controllers间的通信是很容易的。努力的定义这些协议方法将增强代码的可读性，以及更好的跟踪你的app。代理协议发生改变以及实现都可通过编译器检查出来，如果没有将会在开发的过程中至少会出现crash，而不仅仅是让一些事情异常工作。甚至在同一事件通知多控制器的场景中，只要你的应用在controller层次有着良好的结构，消息将在该层次上传递。该层次能够向后传递直至让所有需要知道事件的controllers都知道。当然会有delegation模式不适合的例外情况出现，而且notification可能更加有效。例如：应用中所有的controller需要知道一个事件。然而这些类型的场景很少出现。另外一个例子是当你建立了一个架构而且需要通知该事件给正在运行中应用。

根据经验，如果是属性层的事件，不管是在不需要编程的对象还是在紧紧绑定一个view对象的model对象，我只使用观察。对于其他的事件，我都会使用delegate模式。如果因为某种原因我不能使用delegate，首先我将估计我的app架构是否出现了严重的错误。如果没有错误，然后才使用notification。

什么是设计模式

- 设计模式是为特定场景下的问题而定制的解决方案。特定场景指问题所在的重复出现的场景，问题指特定环境下你想达成的目标。同样的问题在不同的环境下会有不同的限制和挑战。定制的解决方案是指在特定环境下克服了问题的限制条件而达成目标的一种设计。

设计模式的分类

- 设计模式分为三种类型，共23种。
    - 创建型模式：单例模式、抽象工厂模式、建造者模式、工厂模式、原型模式。
    - 结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
    - 行为型模式：模版方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模式、策略模式、职责链模式(责任链模式)、访问者模式。

类工厂方法是什么?

- 类工厂方法的实现是为了向客户提供方便，它们将分配和初始化合在一个步骤中， 返回被创建的对象，并
  进行自动释放处理。

- 这些方法的形式是+ (type)className...（其中 className不包括任何前缀）。

- 工厂方法可能不仅仅为了方便使用。它们不但可以将分配和初始化合在一起，还可以为初始化过程提供对
  象的分配信息。

- 类工厂方法的另一个目的是使类（比如NSWorkspace）提供单例。虽 然init...方法可以确认一个类在每次程序运行过程只存在一个实例，但它需要首先分配一个“生的”实例，然后还必须释放该实例。工厂方法则可以避免为可能没有用的对象盲目分配内存。

单例是什么?

单例模式的意思就是只有一个实例。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

1. 单例模式的要点：

显然单例模式的要点有三个；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. 单例模式的优点：

- 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。
- 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

手写一下单例方法(或者单例模式的设计：GCD 方式和同步锁方式的区别在哪里？unlock 呢？GCD 是怎么保证单例的？)

image

简要描述观察者模式，并运用此模式编写一段代码？

- 观察者模式（Observer）是指一个或多个对象对另一个对象进行观察，当被观察对象发生变化时，观察者可以直接或间接地得到通知，从而能自动地更新观察者的数据，或者进行一些操作。

- 具体到iOS的开发中，实现观察者模式常用的方式有KVO和Notification两种。

- 两者的不同在于，KVO是被观察者主动向观察者发送消息；Notification是被观察者向NotificationCenter发送消息，再由NotificationCenter post通知到每个注册的观察者。

谈谈你对MVC的理解?为什么要用MVC?在Cocoa中MVC是怎么实现的?你还熟悉其他的OC设计模式或别的设计模式吗?

- MVC就是Model-View-Controller的缩写,M指的是业务模型,V指的是用户页面,C指的是控制器。MVC是架构模式,是讲M和V的代码分离,从而使同那个一个程序可以使用不同的表现形式。

- 单例,代理,观察者,工厂模式等

- 单例模式:上面有详细说明

- 代理模式:代理模式给某一个对象提供一个代理对象,并由代理对象控制对源对象的引用.比如一个工厂生产了产品,并不想直接卖给用户,而是搞了很多代理商,用户可以直接找代理商买东西,代理商从工厂进货.常见的如QQ的自动回复就属于代理拦截,代理模式在iphone中得到广泛应用.

MVC优点不正确的是

```
A 低耦合性
B 高重用性和可适用性
C 较低的生命周期成本
D 代码高效率
```

- 参考答案：D
- 理由：MVC只是一种设计模式，它的出现有比较久的历史了。Model-Controller-View是在开发中最常见到的设计模式，通过将Model、View、Controller三者相互联系，以Model作为数据加工厂，以Controller作为桥梁，处理业务，而View只是数据展示层，理应与业务无关。MVC设计模式降低了耦合性，提供了重用性和适用性，可有效地提高开发效率。

如何理解MVVM框架，它的优点和缺点在哪？运用此框架编写一段代码，建议采用ReactiveCocoa库实现；

- MVVM框架相对于传统的MVC来说，主要区别在于把原本在C中（ViewController）的业务逻辑、网络请求、数据存储等操作和表现逻辑，分离到ViewModel中，从而使ViewController得到精简
- MVC中，Controller同时操作Model和View；MVVM中，ViewModel作为一个过渡，Model的数据获取和加工由ViewModel负责，得到适合View的数据，利用绑定机制，使得View得以自动更新。

```
优点：
层次更加分明清晰
代码简洁优雅
减少VC的复杂性
代码和界面完全分离
方便测试

缺点：
MVVM需要使用数据绑定机制，对于OS X 开发，可以直接使用Coocoa Binding，对于iOS，没有太好的数据绑定方法，可以使用KV

ReactiveCocoa提供了一种很方便优雅的绑定机制。
```

ReactiveCocoa

- RAC具有函数式编程和响应式编程的特性
- 试图解决以下问题
- 传统iOS开发过程中，状态以及状态之间依赖过多的问题
- 传统MVC架构的问题：Controller比较复杂，可测试性差
- 提供统一的消息传递机制

哪些途径可以让 ViewController 瘦下来?

- 把 Data Source 和其他 Protocols 分离出来(将UITableView或者UICollectionView的代码提取出来放在其他类中)
- 将业务逻辑移到 Model 中(和模型有关的逻辑全部在model中写)
- 把网络请求逻辑移到 Model 层(网络请求依靠模型)
- 把 View 代码移到 View 层(自定义View)

你在你的项目中用到了哪些设计模式?

项目中使用了很多的设计模式，我相信面试官最好听到的不仅仅是设计模式的名字，更想听到的是这些设计模式在项目中如何应用
参考答案：
- 单例设计模式：在项目中，单例是必不可少的。比如`UIApplication`、`NSUserDefaults`就是苹果提供的单例。在项目中经常
- `MVC`设计模式：现在绝大部分项目都是基于`MVC`设计模式的，现在有一部分开发者采用`MVVM`、`MVP`等模式。
- 通知(`NSNotification`)模式：通知在开发中是必不可少的，对于跨模块的类交互，需要使用通知；对于多对多的关系，使用
- 工厂设计模式：在我的项目中使用了大量的工厂设计模式，特别是生成控件的`API`，都已经封装成一套，全部是扩展的类方法
- `KVC/KVO`设计模式：有的时候需要监听某个类的属性值的变化而做出相应的改变，这时候会使用`KVC/KVO`设计模式。在项目中

就说这么多吧，还有很多的设计模式，不过其它并不是那么常用。

如何实现单例，单例会有什么弊端?

单例在项目中的是必不可少的，它可以使我们全局都可共享我们的数据。这只是简单的问题，大家根据自己的情况回答。
参考答案：
- 首先，单例写法有好几种，通常的写法是基于线程安全的写法，结合`dispatch_once`来使用，保证单例对象只会被创建一次。
- 其次，由于单例是约定俗成的，因此在实际开发中通常不会去重写内存管理方法。

单例确实给我们带来的便利，但是它也会有代价的。单例一旦创建，整个`App`使用过程都不会释放，这会占用内存，因此不可滥用

你在你的项目中用到了哪些设计模式?

- 设计模式有很多,面试官肯定不想听你把项目里的设计模式名字报给他,他想听得肯定是你是怎么去用这些设计模式的
- 参考答案:
- MVC:这个设计模型大部分应用应该都在用,介绍下MVC就好
- 单例:单例在项目中用的还是蛮多的,像登录界面,对一些第三方框架二次封装等等
- KVC/KVO:这个用的应该也很多,KVC用来替换掉系统的tabbar,用KVO来监听偏移量来完成下拉刷新,改变导航条背景颜色这些

- 工厂方法:这个用的更多了,设置一些自定义View肯定要用到这个设计模式
- 如何实现单例，单例会有什么弊端?
- 这个问题还是蛮简单的,说下单例是怎么写的,单例的缺点就好
- 参考答案:

```
// OC版
+(instancetype)sharedInstance
{
static id sharedInstance = nil;

static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{

sharedInstance = [[self alloc] init];
});

return sharedInstance;
}

// Swift版
static let sharedInstance : <#SingletonClass#> = <#SingletonClass#>()
```

- 单例的缺点也就是会一直占着这块内存,不会被释放

单例书写

- 伪单例

```
1、 获取单例对象的方法

+ (DataHandle *)sharedDataHandle; // 创建单例对象的方法。类方法 命名规则： shared + 类名
2、 方法的实现

// 因为实例是全局的 因此要定义为全局变量，且需要存储在静态区，不释放。不能存储在栈区。
static DataHandle *handle = nil;

// 伪单例 和 完整的单例。 以及线程的安全。
// 一般使用伪单例就足够了 每次都用 sharedDataHandle 创建对象。
+ (DataHandle *)sharedDataHandle
{
    // 添加同步锁，一次只能一个线程访问。如果有多个线程访问，等待。一个访问结束后下一个。
    @synchronized(self){
        if (nil == handle) {
            handle = [[DataHadle alloc] init];
        }
    }
  return handle;
}
```

- 完整单例

完整的单例
完整的单例要求比较高，不仅要求我们通过方法获取的对象是单例，如果有 对该对象进行copy mutableCopy copyWithZone 等

完整的单例要做到四个方面：

为单例对象实现一个静态实例,然后设置成nil，
构造方法检查静态实例是否为nil，是则新建并返回一个实例，
重写allocWithZone方法，用来保证其他人直接使用alloc和init试图获得一个新实例的时候不会产生一个新实例，
适当实现copyWithZone，,retain,retainCount,release和autorelease 等方法

1、 获取单例对象的方法

```
+ (DataHandle *)sharedDataHandle; // 创建单例对象的方法。类方法 命名规则： shared + 类名
2、 方法的实现
@synchronized(self){
        if (nil == handle) {
            handle = [[super allocWithZone:nil] init]; // 避免死循环
            // 如果 在单例类里面重写了 allocWithZone 方法 ，在创建单例对象时 使用 [[DataHandle alloc] init]

        }
    }
 return handle;
```

3、 重写 allocWithZone copy mutableCopy copyWithZone

防止外界拷贝造成多个实例， 保证实例的唯一性。
注意：如果自己重写了 allocWithZone 就不要再调用自身的 alloc 方法，否则会出现死循环。

```
+ (instancetype)allocWithZone:(struct _NSZone *)zone
{
    return [DataHandle sharedDataHandle];
}

- (id)copy
{
    return self;
}

- (id)mutableCopy
{
    return self;
}

+ (id)copyWithZone:(struct _NSZone *)zone
{
    return self;
}
4、 重写 alloc retain release autorelease retainCount

+ (instancetype)alloc
{
    return [DataHandle sharedDataHandle];
}
```

```
//  因为只有一个实例，  一直不释放，所以不增加引用计数。无意义。
- (instancetype)retain
{
    return self;
}

- (oneway void)release
{
    // nothing
}

- (instancetype)autorelease
{
    return self;
}

- (NSUInteger)retainCount
{
    return NSUIntegerMax; // 返回整形最大值。
}
```

文章如有问题，请留言，我将及时更正。

满地打滚卖萌求赞，如果本文帮助到你，轻点下方的红心，给作者君增加更新的动力。