OC 底层面试题

一、通知（NSNotification）

使用观察者模式来实现的用于跨层传递信息的机制。传递方式是一对多的。

- 如果实现通知机制?

1.应用服务提供商从服务器端把要发送的消息和设备令牌（device token）发送给苹果的消息推送服务器APNs。
2.APNs 根据设备令牌在已注册的设备（iPhone、iPad、iTouch、mac 等）查找对应的设备，将消息发送给相应的设备。
3.客户端设备接将接收到的消息传递给相应的应用程序，应用程序根据用户设置弹出通知消息。

二、属性关键字

1.读写权限：readonly,readwrite(默认)
2.原子性: atomic(默认)，nonatomic。atomic 读写线程安全，但效率低，而且不是绝对的安全，比如如果修饰的是数组，那么对数组的读写是安全的，但如果是操作数组进行添加移除其中对象的还，就不保证安全了。
3.引用计数:

- retain/strong
- assign：修饰基本数据类型，修饰对象类型时，不改变其引用计数，会产生悬垂指针，修饰的对象在被释放后，assign 指针仍然指向原对象内存地址，如果使用 assign 指针继续访问原对象的话，就可能会导致内存泄漏或程序异常
- weak：不改变被修饰对象的引用计数，所指对象在被释放后，weak 指针会自动置为 nil
- copy：分为深拷贝和浅拷贝
  浅拷贝：对内存地址的复制，让目标对象指针和原对象指向同一片内存空间会增加引用计数
  深拷贝：对对象内容的复制，开辟新的内存空间

可变对象的 copy 和 mutableCopy 都是深拷贝
不可变对象的 copy 是浅拷贝，mutableCopy 是深拷贝
copy 方法返回的都是不可变对象

- @property (nonatomic, copy) NSMutableArray * array;这样写有什么影响?
  因为 copy 方法返回的都是不可变对象，所以 array 对象实际上是不可变的，如果对其进行可变操作如添加移除对象，则会造成程序 crash

三、分类、扩展、代理（Delegate）

一、分类

- 1.分类的作用？
  声明私有方法，分解体积大的类文件，把 framework 的私有方法公开

- 2.分类的特点
  运行时决议，可以为系统类添加分类 。
  说得详细些，在运行时时期，将 Category 中的实例方法列表、协议列表、属性列表添加到主类中后（所以 Category 中的方法在方法列表中的位置是在主类的同名方法之前的），然后会递归调用所有类的 load 方法，这一切都是在 main 函数之前执行的。

- 3.分类可以添加哪些内容？
  实例方法，类方法，协议，属性（添加 getter 和 setter 方法，并没有实例变量，添加实例变量需要用关联对象）

- 4.如果工程里有两个分类 A 和 B，两个分类中有一个同名的方法，哪个方法最终生效？
  取决于分类的编译顺序，最后编译的那个分类的同名方法最终生效，而之前的都会被覆盖掉(这里并不是真正的覆盖，因为其余方法仍然存在，只是访问不到，因为在动态添加类的方法的时候是倒序遍历方法列表的，而最后编译的分类的方法会放在方法列表前面，访问的时候就会先被访问到，同理如果声明了一个和原类方法同名的方法，也会覆盖掉原类的方法)。

- 5.如果声明了两个同名的分类会怎样？
  会报错，所以第三方的分类，一般都带有命名前缀

- 6.分类能添加成员变量吗？
  不能。只能通过关联对象(objc_setAssociatedObject)来模拟实现成员变量，但其实质是关联内容，所有对象的关联内容都放在同一个全局容器哈希表中:AssociationsHashMap,由 AssociationsManager 统一管理。

二、扩展

- 1.一般用扩展做什么？
  声明私有属性，声明方法（没什么意义），声明私有成员变量
- 2.扩展的特点
  编译时决议，只能以声明的形式存在，多数情况下寄生在宿主类的.m 中，不能为系统类添加扩展。

三、代理（Delegate）

代理是一种设计模式，以@protocol 形式体现，一般是一对一传递。
一般以 weak 关键词以规避循环引用。

四、KVO (Key-value observing)

KVO 是观察者模式的另一实现。
使用了 isa 混写(isa-swizzling)来实现 KVO

使用 setter 方法改变值 KVO 会生效，使用 setValue:forKey 即 KVC 改变值 KVO 也会生效，因为 KVC 会去调用 setter 方法

```
- (void)setValue:(id)value
{
    [self willChangeValueForKey:@"key"];

    [super setValue:value];

    [self didChangeValueForKey:@"key"];
}
```

那么通过直接赋值成员变量会触发 KVO 吗?
不会，因为不会调用 setter 方法，需要加上
willChangeValueForKey 和 didChangeValueForKey 方法来手动触发才行

五、KVC(Key-value coding)

```
-(id)valueForKey:(NSString *)key;

-(void)setValue:(id)value forKey:(NSString *)key;
```

KVC 就是指 iOS 的开发中，可以允许开发者通过 Key 名直接访问对象的属性，或者给对象的属性赋值。而不需要调用明确的存取方法。这样就可以在运行时动态地访问和修改对象的属性。而不是在编译时确定，这也是 iOS 开发中的黑魔法之一。很多高级的 iOS 开发技巧都是基于 KVC 实现的

当调用 setValue：属性值 forKey：@”name“的代码时，，底层的执行机制如下：

- 程序优先调用 set<Key>:属性值方法，代码通过 setter 方法完成设置。注意，这里的<key>是指成员变量名，首字母大小写要符合 KVC 的命名规则，下同
- 如果没有找到 setName：方法，KVC 机制会检查+ (BOOL)accessInstanceVariablesDirectly 方法有没有返回 YES，默认该方法会返回 YES，如果你重写了该方法让其返回 NO 的话，那么在这一步 KVC 会执行 setValue：forUndefinedKey：方法，不过一般开发者不会这么做。所以 KVC 机制会搜索该类里面有没有名为<key>的成员变量，无论该变量是在类接口处定义，还是在类实现处定义，也无论用了什么样的访问修饰符，只在存在以<key>命名的变量，KVC 都可以对该成员变量赋值。
- 如果该类即没有 set<key>：方法，也没有_<key>成员变量，KVC 机制会搜索_is<Key>的成员变量。
- 和上面一样，如果该类即没有 set<Key>：方法，也没有_<key>和_is<Key>成员变量，KVC 机制再会继续搜索<key>和 is<Key>的成员变量。再给它们赋值。

- 如果上面列出的方法或者成员变量都不存在，系统将会执行该对象的 setValue：forUndefinedKey：方法，默认是抛出异常。

即如果没有找到 Set<Key>方法的话，会按照_key，_iskey，key，iskey 的顺序搜索成员并进行赋值操作。

如果开发者想让这个类禁用 KVC，那么重写+ (BOOL)accessInstanceVariablesDirectly 方法让其返回 NO 即可，这样的话如果 KVC 没有找到 set<Key>:属性名时，会直接用 setValue：forUndefinedKey：方法。

当调用 valueForKey：@”name“的代码时，KVC 对 key 的搜索方式不同于 setValue：属性值 forKey：@”name“，其搜索方式如下：

- 首先按 get<Key>,<key>,is<Key>的顺序方法查找 getter 方法，找到的话会直接调用。如果是 BOOL 或者 Int 等值类型， 会将其包装成一个 NSNumber 对象。
- 如果上面的 getter 没有找到，KVC 则会查找 countOf<Key>,objectIn<Key>AtIndex 或<Key>AtIndexes 格式的方法。如果 countOf<Key>方法和另外两个方法中的一个被找到，那么就会返回一个可以响应 NSArray 所有方法的代理集合(它是 NSKeyValueArray，是 NSArray 的子类)，调用这个代理集合的方法，或者说给这个代理集合发送属于 NSArray 的方法，就会以 countOf<Key>,objectIn<Key>AtIndex 或<Key>AtIndexes 这几个方法组合的形式调用。还有一个可选的 get<Key>:range:方法。所以你想重新定义 KVC 的一些功能，你可以添加这些方法，需要注意的是你的方法名要符合 KVC 的标准命名方法，包括方法签名。
- 如果上面的方法没有找到，那么会同时查找 countOf<Key>，enumeratorOf<Key>,memberOf<Key>格式的方法。如果这三个方法都找到，那么就返回一个可以响应 NSSet 所的方法的代理集合，和上面一样，给这个代理集合发 NSSet 的消息，就会以 countOf<Key>，enumeratorOf<Key>,memberOf<Key>组合的形式调用。
- 如果还没有找到，再检查类方法+

(BOOL)accessInstanceVariablesDirectly,如果返回 YES(默认行为)，那么和先前的设值一样，会按_<key>,_is<Key>,<key>,is<Key>的顺序搜索成员变量名，这里不推荐这么做，因为这样直接访问实例变量破坏了封装性，使代码更脆弱。如果重写了类方法+ (BOOL)accessInstanceVariablesDirectly 返回 NO 的话，那么会直接调用 valueForUndefinedKey:方法，默认是抛出异常。