[《招聘一个靠谱的 iOS》](http://blog.sunnyxx.com/2015/07/04/ios-interview/)—参考答案（上）

说明：面试题来源是[微博@我就叫Sunny怎么了](http://weibo.com/u/1364395395)的这篇博文：[《招聘一个靠谱的 iOS》](http://blog.sunnyxx.com/2015/07/04/ios-interview/)，其中共55题，除第一题为纠错题外，其他54道均为简答题。

出题者简介： 孙源（sunnyxx），目前就职于百度，负责百度知道 iOS 客户端的开发工作，对技术喜欢刨根问底和总结最佳实践，热爱分享和开源，维护一个叫 forkingdog 的开源小组。

答案为[微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)整理，未经出题者校对，如有纰漏，请向[微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)指正。

----------

# 索引

1. [风格纠错题](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#1-风格纠错题)

1. [优化部分](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#优化部分)

2. [硬伤部分](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#硬伤部分)

2. [什么情况使用 weak 关键字，相比 assign 有什么不同？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#2-什么情况使用-weak-关键字相比-assign-有什么不同)

3. [怎么用 copy 关键字？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#3-怎么用-copy-关键字)

4. [这个写法会出什么问题： @property (copy) NSMutableArray \*array;](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#4-这个写法会出什么问题-property-copy-nsmutablearray-array)

5. [ 如何让自己的类用 copy 修饰符？如何重写带 copy 关键字的 setter？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#5-如何让自己的类用-copy-修饰符如何重写带-copy-关键字的-setter)

6. [@property 的本质是什么？ivar、getter、setter 是如何生成并添加到这个类中的](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#6-property-的本质是什么ivargettersetter-是如何生成并添加到这个类中的)

7. [@protocol 和 category 中如何使用 @property](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#7-protocol-和-category-中如何使用-property)

8. [ runtime 如何实现 weak 属性](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#8-runtime-如何实现-weak-属性)

9. [@property中有哪些属性关键字？/ @property 后面可以有哪些修饰符？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#9-property中有哪些属性关键字-property-后面可以有哪些修饰符)

10. [ weak属性需要在dealloc中置nil么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#10-weak属性需要在dealloc中置nil么)

11. [@synthesize和@dynamic分别有什么作用？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#11-synthesize和dynamic分别有什么作用)

12. [ARC下，不显式指定任何属性关键字时，默认的关键字都有哪些？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#12-arc下不显式指定任何属性关键字时默认的关键字都有哪些)

13. [用@property声明的NSString（或NSArray，NSDictionary）经常使用copy关键字，为什么？如果改用strong关键字，可能造成什么问题？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#13-用property声明的nsstring或nsarraynsdictionary经常使用copy关键字为什么如果改用strong关键字可能造成什么问题)

1. [对非集合类对象的copy操作](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#1-对非集合类对象的copy操作)

2. [集合类对象的copy与mutableCopy](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#2集合类对象的copy与mutablecopy)

14. [@synthesize合成实例变量的规则是什么？假如property名为foo，存在一个名为\_foo的实例变量，那么还会自动合成新变量么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#14-synthesize合成实例变量的规则是什么假如property名为foo存在一个名为\_foo的实例变量那么还会自动合成新变量么)

15. [在有了自动合成属性实例变量之后，@synthesize还有哪些使用场景？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#15-在有了自动合成属性实例变量之后synthesize还有哪些使用场景)

16. [objc中向一个nil对象发送消息将会发生什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#16-objc中向一个nil对象发送消息将会发生什么)

17. [objc中向一个对象发送消息[obj foo]和objc\_msgSend()函数之间有什么关系？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#17-objc中向一个对象发送消息obj-foo和objc\_msgsend函数之间有什么关系)

18. [什么时候会报unrecognized selector的异常？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#18-什么时候会报unrecognized-selector的异常)

19. [一个objc对象如何进行内存布局？（考虑有父类的情况）](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#19-一个objc对象如何进行内存布局考虑有父类的情况)

20. [一个objc对象的isa的指针指向什么？有什么作用？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#20-一个objc对象的isa的指针指向什么有什么作用)

21. [下面的代码输出什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（上）.md#21-下面的代码输出什么)

```Objective-C

@implementation Son : Father

- (id)init

{

self = [super init];

if (self) {

NSLog(@"%@", NSStringFromClass([self class]));

NSLog(@"%@", NSStringFromClass([super class]));

}

return self;

}

@end

```

22. 22--55题，请看下篇。

### 1. 风格纠错题

![enter image description here](http://i.imgur.com/O7Zev94.png)

修改完的代码：

修改方法有很多种，现给出一种做示例：

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 修改完的代码，这是第一种修改方法，后面会给出第二种修改方法

typedef NS\_ENUM(NSInteger, CYLSex) {

CYLSexMan,

CYLSexWoman

};

@interface CYLUser : NSObject<NSCopying>

@property (nonatomic, readonly, copy) NSString \*name;

@property (nonatomic, readonly, assign) NSUInteger age;

@property (nonatomic, readonly, assign) CYLSex sex;

- (instancetype)initWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

+ (instancetype)userWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

@end

```

下面对具体修改的地方，分两部分做下介绍：\*\*硬伤部分\*\* 和 \*\*优化部分\*\*

。因为\*\*硬伤部分\*\*没什么技术含量，为了节省大家时间，放在后面讲，大神请直接看\*\*优化部分\*\*。

#### \*\*优化部分\*\*

1. enum 建议使用 `NS\_ENUM` 和 `NS\_OPTIONS` 宏来定义枚举类型，参见官方的 [Adopting Modern Objective-C](https://developer.apple.com/library/ios/releasenotes/ObjectiveC/ModernizationObjC/AdoptingModernObjective-C/AdoptingModernObjective-C.html) 一文：

```objective-c

//定义一个枚举

typedef NS\_ENUM(NSInteger, CYLSex) {

CYLSexMan,

CYLSexWoman

};

```

（仅仅让性别包含男和女可能并不严谨，最严谨的做法可以参考 [这里](https://github.com/ChenYilong/iOSInterviewQuestions/issues/9) 。）

2. age 属性的类型：应避免使用基本类型，建议使用 Foundation 数据类型，对应关系如下：

```Objective-C

int -> NSInteger

unsigned -> NSUInteger

float -> CGFloat

动画时间 -> NSTimeInterval

```

同时考虑到 age 的特点，应使用 NSUInteger ，而非 int 。

这样做的是基于64-bit 适配考虑，详情可参考出题者的博文[《64-bit Tips》](http://blog.sunnyxx.com/2014/12/20/64-bit-tips/)。

3. 如果工程项目非常庞大，需要拆分成不同的模块，可以在类、typedef宏命名的时候使用前缀。

4. doLogIn方法不应写在该类中： <p><del>虽然`LogIn`的命名不太清晰，但笔者猜测是login的意思， （勘误：Login是名词，LogIn 是动词，都表示登陆的意思。见： [ \*\*\*Log in vs. login\*\*\* ](http://grammarist.com/spelling/log-in-login/)）</del></p>登录操作属于业务逻辑，观察类名 UserModel ，以及属性的命名方式，该类应该是一个 Model 而不是一个“ MVVM 模式下的 ViewModel ”：

> 无论是 MVC 模式还是 MVVM 模式，业务逻辑都不应当写在 Model 里：MVC 应在 C，MVVM 应在 VM。

（如果抛开命名规范，假设该类真的是 MVVM 模式里的 ViewModel ，那么 UserModel 这个类可能对应的是用户注册页面，如果有特殊的业务需求，比如： `-logIn` 对应的应当是注册并登录的一个 Button ，出现 `-logIn` 方法也可能是合理的。）

5. doLogIn 方法命名不规范：添加了多余的动词前缀。

请牢记：

> 如果方法表示让对象执行一个动作，使用动词打头来命名，注意不要使用 `do`，`does` 这种多余的关键字，动词本身的暗示就足够了。

应为 `-logIn` （注意： `Login` 是名词， `LogIn` 是动词，都表示登陆。 见[ \*\*\*Log in vs. login\*\*\* ](http://grammarist.com/spelling/log-in-login/)）

6. `-(id)initUserModelWithUserName: (NSString\*)name withAge:(int)age;`方法中不要用 `with` 来连接两个参数: `withAge:` 应当换为`age:`，`age:` 已经足以清晰说明参数的作用，也不建议用 `andAge:` ：通常情况下，即使有类似 `withA:withB:` 的命名需求，也通常是使用`withA:andB:` 这种命名，用来表示方法执行了两个相对独立的操作（\*从设计上来说，这时候也可以拆分成两个独立的方法\*），它不应该用作阐明有多个参数，比如下面的：

```objective-c

//错误，不要使用"and"来连接参数

- (int)runModalForDirectory:(NSString \*)path andFile:(NSString \*)name andTypes:(NSArray \*)fileTypes;

//错误，不要使用"and"来阐明有多个参数

- (instancetype)initWithName:(CGFloat)width andAge:(CGFloat)height;

//正确，使用"and"来表示两个相对独立的操作

- (BOOL)openFile:(NSString \*)fullPath withApplication:(NSString \*)appName andDeactivate:(BOOL)flag;

```

7. 由于字符串值可能会改变，所以要把相关属性的“内存管理语义”声明为 copy 。(原因在下文有详细论述：\*\*\*用@property声明的NSString（或NSArray，NSDictionary）经常使用copy关键字，为什么？\*\*\*)

8. “性别”(sex）属性的：该类中只给出了一种“初始化方法” (initializer)用于设置“姓名”(Name)和“年龄”(Age)的初始值，那如何对“性别”(Sex）初始化？

Objective-C 有 designated 和 secondary 初始化方法的观念。 designated 初始化方法是提供所有的参数，secondary 初始化方法是一个或多个，并且提供一个或者更多的默认参数来调用 designated 初始化方法的初始化方法。举例说明：

```Objective-C

// .m文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

//

@implementation CYLUser

- (instancetype)initWithName:(NSString \*)name

age:(NSUInteger)age

sex:(CYLSex)sex {

if(self = [super init]) {

\_name = [name copy];

\_age = age;

\_sex = sex;

}

return self;

}

- (instancetype)initWithName:(NSString \*)name

age:(NSUInteger)age {

return [self initWithName:name age:age sex:nil];

}

@end

```

上面的代码中initWithName:age:sex: 就是 designated 初始化方法，另外的是 secondary 初始化方法。因为仅仅是调用类实现的 designated 初始化方法。

因为出题者没有给出 `.m` 文件，所以有两种猜测：1：本来打算只设计一个 designated 初始化方法，但漏掉了“性别”(sex）属性。那么最终的修改代码就是上文给出的第一种修改方法。2：不打算初始时初始化“性别”(sex）属性，打算后期再修改，如果是这种情况，那么应该把“性别”(sex）属性设为 readwrite 属性，最终给出的修改代码应该是：

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 第二种修改方法（基于第一种修改方法的基础上）

typedef NS\_ENUM(NSInteger, CYLSex) {

CYLSexMan,

CYLSexWoman

};

@interface CYLUser : NSObject<NSCopying>

@property (nonatomic, readonly, copy) NSString \*name;

@property (nonatomic, readonly, assign) NSUInteger age;

@property (nonatomic, readwrite, assign) CYLSex sex;

- (instancetype)initWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

- (instancetype)initWithName:(NSString \*)name age:(NSUInteger)age;

+ (instancetype)userWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

@end

```

`.h` 中暴露 designated 初始化方法，是为了方便子类化 （想了解更多，请戳--》 [\*\*\*《禅与 Objective-C 编程艺术 （Zen and the Art of the Objective-C Craftsmanship 中文翻译）》\*\*\*](http://is.gd/OQ49zk)。）

- 按照接口设计的惯例，如果设计了“初始化方法” (initializer)，也应当搭配一个快捷构造方法。而快捷构造方法的返回值，建议为 instancetype，为保持一致性，init 方法和快捷构造方法的返回类型最好都用 instancetype。

- 如果基于第一种修改方法：既然该类中已经有一个“初始化方法” (initializer)，用于设置“姓名”(Name)、“年龄”(Age)和“性别”(Sex）的初始值:

那么在设计对应 `@property` 时就应该尽量使用不可变的对象：其三个属性都应该设为“只读”。用初始化方法设置好属性值之后，就不能再改变了。在本例中，仍需声明属性的“内存管理语义”。于是可以把属性的定义改成这样

```Objective-C

@property (nonatomic, readonly, copy) NSString \*name;

@property (nonatomic, readonly, assign) NSUInteger age;

@property (nonatomic, readonly, assign) CYLSex sex;

```

由于是只读属性，所以编译器不会为其创建对应的“设置方法”，即便如此，我们还是要写上这些属性的语义，以此表明初始化方法在设置这些属性值时所用的方式。要是不写明语义的话，该类的调用者就不知道初始化方法里会拷贝这些属性，他们有可能会在调用初始化方法之前自行拷贝属性值。这种操作多余而且低效。

9. `initUserModelWithUserName` 如果改为 `initWithName` 会更加简洁，而且足够清晰。

10. `UserModel` 如果改为 `User` 会更加简洁，而且足够清晰。

11. `UserSex`如果改为`Sex` 会更加简洁，而且足够清晰。

12. 第二个 `@property` 中 assign 和 nonatomic 调换位置。

推荐按照下面的格式来定义属性

```Objective-C

@property (nonatomic, readwrite, copy) NSString \*name;

```

属性的参数应该按照下面的顺序排列： 原子性，读写 和 内存管理。 这样做你的属性更容易修改正确，并且更好阅读。这在[《禅与Objective-C编程艺术 >》](https://github.com/oa414/objc-zen-book-cn#属性定义)里有介绍。而且习惯上修改某个属性的修饰符时，一般从属性名从右向左搜索需要修动的修饰符。最可能从最右边开始修改这些属性的修饰符，根据经验这些修饰符被修改的可能性从高到底应为：内存管理 > 读写权限 >原子操作。

#### \*\*\*硬伤部分\*\*\*

1. 在-和(void)之间应该有一个空格

3. enum 中驼峰命名法和下划线命名法混用错误：枚举类型的命名规则和函数的命名规则相同：命名时使用驼峰命名法，勿使用下划线命名法。

3. enum 左括号前加一个空格，或者将左括号换到下一行

4. enum 右括号后加一个空格

2. `UserModel :NSObject` 应为`UserModel : NSObject`，也就是`:`右侧少了一个空格。

2. `@interface` 与 `@property` 属性声明中间应当间隔一行。

2. 两个方法定义之间不需要换行，有时为了区分方法的功能也可间隔一行，但示例代码中间隔了两行。

9. `-(id)initUserModelWithUserName: (NSString\*)name withAge:(int)age;` 方法中方法名与参数之间多了空格。而且 `-` 与 `(id)` 之间少了空格。

10. `-(id)initUserModelWithUserName: (NSString\*)name withAge:(int)age;` 方法中方法名与参数之间多了空格：`(NSString\*)name` 前多了空格。

10. `-(id)initUserModelWithUserName: (NSString\*)name withAge:(int)age;` 方法中 `(NSString\*)name`,应为 `(NSString \*)name`，少了空格。

7. <p><del>doLogIn方法中的 `LogIn` 命名不清晰：笔者猜测是login的意思，应该是粗心手误造成的。

（勘误： `Login` 是名词， `LogIn` 是动词，都表示登陆的意思。见： [ \*\*\*Log in vs. login\*\*\* ](http://grammarist.com/spelling/log-in-login/)）</del></p>

### 2. 什么情况使用 weak 关键字，相比 assign 有什么不同？

什么情况使用 weak 关键字？

1. 在 ARC 中,在有可能出现循环引用的时候,往往要通过让其中一端使用 weak 来解决,比如: delegate 代理属性

2. 自身已经对它进行一次强引用,没有必要再强引用一次,此时也会使用 weak,自定义 IBOutlet 控件属性一般也使用 weak；当然，也可以使用strong。在下文也有论述：\*\*\*《IBOutlet连出来的视图属性为什么可以被设置成weak?》\*\*\*

不同点：

1. `weak` 此特质表明该属性定义了一种“非拥有关系” (nonowning relationship)。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特质同assign类似，

然而在属性所指的对象遭到摧毁时，属性值也会清空(nil out)。

而 `assign` 的“设置方法”只会执行针对“纯量类型” (scalar type，例如 CGFloat 或

NSlnteger 等)的简单赋值操作。

2. assign 可以用非 OC 对象,而 weak 必须用于 OC 对象

### 3. 怎么用 copy 关键字？

用途：

1. NSString、NSArray、NSDictionary 等等经常使用copy关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary；

2. block 也经常使用 copy 关键字，具体原因见[官方文档：\*\*\*Objects Use Properties to Keep Track of Blocks\*\*\*](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html#//apple\_ref/doc/uid/TP40011210-CH8-SW12)：

block 使用 copy 是从 MRC 遗留下来的“传统”,在 MRC 中,方法内部的 block 是在栈区的,使用 copy 可以把它放到堆区.在 ARC 中写不写都行：对于 block 使用 copy 还是 strong 效果是一样的，但写上 copy 也无伤大雅，还能时刻提醒我们：编译器自动对 block 进行了 copy 操作。如果不写 copy ，该类的调用者有可能会忘记或者根本不知道“编译器会自动对 block 进行了 copy 操作”，他们有可能会在调用之前自行拷贝属性值。这种操作多余而低效。你也许会感觉我这种做法有些怪异，不需要写依然写。如果你这样想，其实是你“日用而不知”，你平时开发中是经常在用我说的这种做法的，比如下面的属性不写copy也行，但是你会选择写还是不写呢？

```Objective-C

@property (nonatomic, copy) NSString \*userId;

- (instancetype)initWithUserId:(NSString \*)userId {

self = [super init];

if (!self) {

return nil;

}

\_userId = [userId copy];

return self;

}

```

![enter image description here](http://i.imgur.com/VlVKl8L.png)

下面做下解释：

copy 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是将其“拷贝” (copy)。

当属性类型为 NSString 时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个 NSMutableString 类的实例。这个类是 NSString 的子类，表示一种可修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变” (immutable)的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的” (mutable)，就应该在设置新属性值时拷贝一份。

> 用 `@property` 声明 NSString、NSArray、NSDictionary 经常使用 copy 关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary，他们之间可能进行赋值操作，为确保对象中的字符串值不会无意间变动，应该在设置新属性值时拷贝一份。

该问题在下文中也有论述：\*\*\*用@property声明的NSString（或NSArray，NSDictionary）经常使用copy关键字，为什么？如果改用strong关键字，可能造成什么问题？\*\*\*

### 4. 这个写法会出什么问题： `@property (copy) NSMutableArray \*array;`

两个问题：1、添加,删除,修改数组内的元素的时候,程序会因为找不到对应的方法而崩溃.因为 copy 就是复制一个不可变 NSArray 的对象；2、使用了 atomic 属性会严重影响性能 ；

第1条的相关原因在下文中有论述\*\*\*《用@property声明的NSString（或NSArray，NSDictionary）经常使用 copy 关键字，为什么？如果改用strong关键字，可能造成什么问题？》\*\*\* 以及上文\*\*\*《怎么用 copy 关键字？》\*\*\*也有论述。

比如下面的代码就会发生崩溃

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 下面的代码就会发生崩溃

@property (nonatomic, copy) NSMutableArray \*mutableArray;

```

```Objective-C

// .m文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 下面的代码就会发生崩溃

NSMutableArray \*array = [NSMutableArray arrayWithObjects:@1,@2,nil];

self.mutableArray = array;

[self.mutableArray removeObjectAtIndex:0];

```

接下来就会奔溃：

```Objective-C

-[\_\_NSArrayI removeObjectAtIndex:]: unrecognized selector sent to instance 0x7fcd1bc30460

```

第2条原因，如下：

> 该属性使用了同步锁，会在创建时生成一些额外的代码用于帮助编写多线程程序，这会带来性能问题，通过声明 nonatomic 可以节省这些虽然很小但是不必要额外开销。

在默认情况下，由编译器所合成的方法会通过锁定机制确保其原子性(atomicity)。如果属性具备 nonatomic 特质，则不使用同步锁。请注意，尽管没有名为“atomic”的特质(如果某属性不具备 nonatomic 特质，那它就是“原子的”(atomic))。

在iOS开发中，你会发现，几乎所有属性都声明为 nonatomic。

一般情况下并不要求属性必须是“原子的”，因为这并不能保证“线程安全” ( thread safety)，若要实现“线程安全”的操作，还需采用更为深层的锁定机制才行。例如，一个线程在连续多次读取某属性值的过程中有别的线程在同时改写该值，那么即便将属性声明为 atomic，也还是会读到不同的属性值。

因此，开发iOS程序时一般都会使用 nonatomic 属性。但是在开发 Mac OS X 程序时，使用

atomic 属性通常都不会有性能瓶颈。

### 5. 如何让自己的类用 copy 修饰符？如何重写带 copy 关键字的 setter？

> 若想令自己所写的对象具有拷贝功能，则需实现 NSCopying 协议。如果自定义的对象分为可变版本与不可变版本，那么就要同时实现 `NSCopying` 与 `NSMutableCopying` 协议。

具体步骤：

1. 需声明该类遵从 NSCopying 协议

2. 实现 NSCopying 协议。该协议只有一个方法:

```Objective-C

- (id)copyWithZone:(NSZone \*)zone;

```

注意：一提到让自己的类用 copy 修饰符，我们总是想覆写copy方法，其实真正需要实现的却是 “copyWithZone” 方法。

以第一题的代码为例：

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 修改完的代码

typedef NS\_ENUM(NSInteger, CYLSex) {

CYLSexMan,

CYLSexWoman

};

@interface CYLUser : NSObject<NSCopying>

@property (nonatomic, readonly, copy) NSString \*name;

@property (nonatomic, readonly, assign) NSUInteger age;

@property (nonatomic, readonly, assign) CYLSex sex;

- (instancetype)initWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

+ (instancetype)userWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

@end

```

然后实现协议中规定的方法：

```Objective-C

- (id)copyWithZone:(NSZone \*)zone {

CYLUser \*copy = [[[self class] allocWithZone:zone]

initWithName:\_name

age:\_age

sex:\_sex];

return copy;

}

```

但在实际的项目中，不可能这么简单，遇到更复杂一点，比如类对象中的数据结构可能并未在初始化方法中设置好，需要另行设置。举个例子，假如 CYLUser 中含有一个数组，与其他 CYLUser 对象建立或解除朋友关系的那些方法都需要操作这个数组。那么在这种情况下，你得把这个包含朋友对象的数组也一并拷贝过来。下面列出了实现此功能所需的全部代码:

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 以第一题《风格纠错题》里的代码为例

typedef NS\_ENUM(NSInteger, CYLSex) {

CYLSexMan,

CYLSexWoman

};

@interface CYLUser : NSObject<NSCopying>

@property (nonatomic, readonly, copy) NSString \*name;

@property (nonatomic, readonly, assign) NSUInteger age;

@property (nonatomic, readonly, assign) CYLSex sex;

- (instancetype)initWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

+ (instancetype)userWithName:(NSString \*)name age:(NSUInteger)age sex:(CYLSex)sex;

- (void)addFriend:(CYLUser \*)user;

- (void)removeFriend:(CYLUser \*)user;

@end

```

// .m文件

```Objective-C

// .m文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

//

@implementation CYLUser {

NSMutableSet \*\_friends;

}

- (void)setName:(NSString \*)name {

\_name = [name copy];

}

- (instancetype)initWithName:(NSString \*)name

age:(NSUInteger)age

sex:(CYLSex)sex {

if(self = [super init]) {

\_name = [name copy];

\_age = age;

\_sex = sex;

\_friends = [[NSMutableSet alloc] init];

}

return self;

}

- (void)addFriend:(CYLUser \*)user {

[\_friends addObject:user];

}

- (void)removeFriend:(CYLUser \*)user {

[\_friends removeObject:user];

}

- (id)copyWithZone:(NSZone \*)zone {

CYLUser \*copy = [[[self class] allocWithZone:zone]

initWithName:\_name

age:\_age

sex:\_sex];

copy->\_friends = [\_friends mutableCopy];

return copy;

}

- (id)deepCopy {

CYLUser \*copy = [[[self class] alloc]

initWithName:\_name

age:\_age

sex:\_sex];

copy->\_friends = [[NSMutableSet alloc] initWithSet:\_friends

copyItems:YES];

return copy;

}

@end

```

以上做法能满足基本的需求，但是也有缺陷：

> 如果你所写的对象需要深拷贝，那么可考虑新增一个专门执行深拷贝的方法。

【注：深浅拷贝的概念，在下文中有介绍，详见下文的：\*\*\*用@property声明的 NSString（或NSArray，NSDictionary）经常使用 copy 关键字，为什么？如果改用 strong 关键字，可能造成什么问题？\*\*\*】

在例子中，存放朋友对象的 set 是用 “copyWithZone:” 方法来拷贝的，这种浅拷贝方式不会逐个复制 set 中的元素。若需要深拷贝的话，则可像下面这样，编写一个专供深拷贝所用的方法:

```Objective-C

- (id)deepCopy {

CYLUser \*copy = [[[self class] alloc]

initWithName:\_name

age:\_age

sex:\_sex];

copy->\_friends = [[NSMutableSet alloc] initWithSet:\_friends

copyItems:YES];

return copy;

}

```

至于\*\*\*如何重写带 copy 关键字的 setter\*\*\*这个问题，

如果抛开本例来回答的话，如下：

```Objective-C

- (void)setName:(NSString \*)name {

//[\_name release];

\_name = [name copy];

}

```

不过也有争议，有人说“苹果如果像下面这样干，是不是效率会高一些？”

```Objective-C

- (void)setName:(NSString \*)name {

if (\_name != name) {

//[\_name release];//MRC

\_name = [name copy];

}

}

```

这样真得高效吗？不见得！这种写法“看上去很美、很合理”，但在实际开发中，它更像下图里的做法：

![enter image description here](http://i.imgur.com/UwV9oSn.jpeg)

克强总理这样评价你的代码风格：

![enter image description here](http://i.imgur.com/N77Lkic.png)

我和总理的意见基本一致：

> 老百姓 copy 一下，咋就这么难？

你可能会说：

之所以在这里做`if判断` 这个操作：是因为一个 if 可能避免一个耗时的copy，还是很划算的。

(在刚刚讲的：《如何让自己的类用 copy 修饰符？》里的那种复杂的copy，我们可以称之为 “耗时的copy”，但是对 NSString 的 copy 还称不上。)

但是你有没有考虑过代价：

> 你每次调用 `setX:` 都会做 if 判断，这会让 `setX:` 变慢，如果你在 `setX:`写了一串复杂的 `if+elseif+elseif+...` 判断，将会更慢。

要回答“哪个效率会高一些？”这个问题，不能脱离实际开发，就算 copy 操作十分耗时，if 判断也不见得一定会更快，除非你把一个“ @property他当前的值 ”赋给了他自己，代码看起来就像：

```Objective-C

[a setX:x1];

[a setX:x1]; //你确定你要这么干？与其在setter中判断，为什么不把代码写好？

```

或者

```Objective-C

[a setX:[a x]]; //队友咆哮道：你在干嘛？！！

```

> 不要在 setter 里进行像 `if(\_obj != newObj)` 这样的判断。（该观点参考链接：[ \*\*\*How To Write Cocoa Object Setters： Principle 3: Only Optimize After You Measure\*\*\* ](http://vgable.com/blog/tag/autorelease/)

）

什么情况会在 copy setter 里做 if 判断？

例如，车速可能就有最高速的限制，车速也不可能出现负值，如果车子的最高速为300，则 setter 的方法就要改写成这样：

```Objective-C

-(void)setSpeed:(int)\_speed{

if(\_speed < 0) speed = 0;

if(\_speed > 300) speed = 300;

\_speed = speed;

}

```

回到这个题目，如果单单就上文的代码而言，我们不需要也不能重写 name 的 setter ：由于是 name 是只读属性，所以编译器不会为其创建对应的“设置方法”，用初始化方法设置好属性值之后，就不能再改变了。（ 在本例中，之所以还要声明属性的“内存管理语义”--copy，是因为：如果不写 copy，该类的调用者就不知道初始化方法里会拷贝这些属性，他们有可能会在调用初始化方法之前自行拷贝属性值。这种操作多余而低效）。

那如何确保 name 被 copy？在初始化方法(initializer)中做：

```Objective-C

- (instancetype)initWithName:(NSString \*)name

age:(NSUInteger)age

sex:(CYLSex)sex {

if(self = [super init]) {

\_name = [name copy];

\_age = age;

\_sex = sex;

\_friends = [[NSMutableSet alloc] init];

}

return self;

}

```

### 6. @property 的本质是什么？ivar、getter、setter 是如何生成并添加到这个类中的

\*\*@property 的本质是什么？\*\*

> @property = ivar + getter + setter;

下面解释下：

> “属性” (property)有两大概念：ivar（实例变量）、存取方法（access method ＝ getter + setter）。

“属性” (property)作为 Objective-C 的一项特性，主要的作用就在于封装对象中的数据。 Objective-C 对象通常会把其所需要的数据保存为各种实例变量。实例变量一般通过“存取方法”(access method)来访问。其中，“获取方法” (getter)用于读取变量值，而“设置方法” (setter)用于写入变量值。这个概念已经定型，并且经由“属性”这一特性而成为 `Objective-C 2.0` 的一部分。

而在正规的 Objective-C 编码风格中，存取方法有着严格的命名规范。

正因为有了这种严格的命名规范，所以 Objective-C 这门语言才能根据名称自动创建出存取方法。其实也可以把属性当做一种关键字，其表示:

> 编译器会自动写出一套存取方法，用以访问给定类型中具有给定名称的变量。

所以你也可以这么说：

> @property = getter + setter;

例如下面这个类：

```Objective-C

@interface Person : NSObject

@property NSString \*firstName;

@property NSString \*lastName;

@end

```

上述代码写出来的类与下面这种写法等效：

```Objective-C

@interface Person : NSObject

- (NSString \*)firstName;

- (void)setFirstName:(NSString \*)firstName;

- (NSString \*)lastName;

- (void)setLastName:(NSString \*)lastName;

@end

```

\*\*更新\*\*：

property在runtime中是`objc\_property\_t`定义如下:

```objective-c

typedef struct objc\_property \*objc\_property\_t;

```

而`objc\_property`是一个结构体，包括name和attributes，定义如下：

```objective-c

struct property\_t {

const char \*name;

const char \*attributes;

};

```

而attributes本质是`objc\_property\_attribute\_t`，定义了property的一些属性，定义如下：

```objective-c

/// Defines a property attribute

typedef struct {

const char \*name; /\*\*< The name of the attribute \*/

const char \*value; /\*\*< The value of the attribute (usually empty) \*/

} objc\_property\_attribute\_t;

```

而attributes的具体内容是什么呢？其实，包括：类型，原子性，内存语义和对应的实例变量。

例如：我们定义一个string的property`@property (nonatomic, copy) NSString \*string;`，通过 `property\_getAttributes(property)`获取到attributes并打印出来之后的结果为`T@"NSString",C,N,V\_string`

其中T就代表类型，可参阅[Type Encodings](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple\_ref/doc/uid/TP40008048-CH100-SW1)，C就代表Copy，N代表nonatomic，V就代表对于的实例变量。

\*\*ivar、getter、setter 是如何生成并添加到这个类中的?\*\*

> “自动合成”( autosynthesis)

完成属性定义后，编译器会自动编写访问这些属性所需的方法，此过程叫做“自动合成”(autosynthesis)。需要强调的是，这个过程由编译

器在编译期执行，所以编辑器里看不到这些“合成方法”(synthesized method)的源代码。除了生成方法代码 getter、setter 之外，编译器还要自动向类中添加适当类型的实例变量，并且在属性名前面加下划线，以此作为实例变量的名字。在前例中，会生成两个实例变量，其名称分别为

`\_firstName` 与 `\_lastName`。也可以在类的实现代码里通过

`@synthesize` 语法来指定实例变量的名字.

```Objective-C

@implementation Person

@synthesize firstName = \_myFirstName;

@synthesize lastName = \_myLastName;

@end

```

我为了搞清属性是怎么实现的,曾经反编译过相关的代码,他大致生成了五个东西

1. `OBJC\_IVAR\_$类名$属性名称` ：该属性的“偏移量” (offset)，这个偏移量是“硬编码” (hardcode)，表示该变量距离存放对象的内存区域的起始地址有多远。

2. setter 与 getter 方法对应的实现函数

2. `ivar\_list` ：成员变量列表

2. `method\_list` ：方法列表

2. `prop\_list` ：属性列表

也就是说我们每次在增加一个属性,系统都会在 `ivar\_list` 中添加一个成员变量的描述,在 `method\_list` 中增加 setter 与 getter 方法的描述,在属性列表中增加一个属性的描述,然后计算该属性在对象中的偏移量,然后给出 setter 与 getter 方法对应的实现,在 setter 方法中从偏移量的位置开始赋值,在 getter 方法中从偏移量开始取值,为了能够读取正确字节数,系统对象偏移量的指针类型进行了类型强转.

### 7. @protocol 和 category 中如何使用 @property

1. 在 protocol 中使用 property 只会生成 setter 和 getter 方法声明,我们使用属性的目的,是希望遵守我协议的对象能实现该属性

2. category 使用 @property 也是只会生成 setter 和 getter 方法的声明,如果我们真的需要给 category 增加属性的实现,需要借助于运行时的两个函数：

1. `objc\_setAssociatedObject`

2. `objc\_getAssociatedObject`

### 8. runtime 如何实现 weak 属性

要实现 weak 属性，首先要搞清楚 weak 属性的特点：

> weak 此特质表明该属性定义了一种“非拥有关系” (nonowning relationship)。为这种属性设置新值时，设置方法既不保留新值，也不释放旧值。此特质同 assign 类似， 然而在属性所指的对象遭到摧毁时，属性值也会清空(nil out)。

那么 runtime 如何实现 weak 变量的自动置nil？

> runtime 对注册的类， 会进行布局，对于 weak 对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为 key，当此对象的引用计数为0的时候会 dealloc，假如 weak 指向的对象内存地址是a，那么就会以a为键， 在这个 weak 表中搜索，找到所有以a为键的 weak 对象，从而设置为 nil。

（注：在下文的《使用runtime Associate方法关联的对象，需要在主对象dealloc的时候释放么？》里给出的“对象的内存销毁时间表”也提到`\_\_weak`引用的解除时间。）

先看下 runtime 里源码的实现：

```Objective-C

/\*\*

\* The internal structure stored in the weak references table.

\* It maintains and stores

\* a hash set of weak references pointing to an object.

\* If out\_of\_line==0, the set is instead a small inline array.

\*/

#define WEAK\_INLINE\_COUNT 4

struct weak\_entry\_t {

DisguisedPtr<objc\_object> referent;

union {

struct {

weak\_referrer\_t \*referrers;

uintptr\_t out\_of\_line : 1;

uintptr\_t num\_refs : PTR\_MINUS\_1;

uintptr\_t mask;

uintptr\_t max\_hash\_displacement;

};

struct {

// out\_of\_line=0 is LSB of one of these (don't care which)

weak\_referrer\_t inline\_referrers[WEAK\_INLINE\_COUNT];

};

};

};

/\*\*

\* The global weak references table. Stores object ids as keys,

\* and weak\_entry\_t structs as their values.

\*/

struct weak\_table\_t {

weak\_entry\_t \*weak\_entries;

size\_t num\_entries;

uintptr\_t mask;

uintptr\_t max\_hash\_displacement;

};

```

具体完整实现参照 [objc/objc-weak.h](https://opensource.apple.com/source/objc4/objc4-646/runtime/objc-weak.h) 。

我们可以设计一个函数（伪代码）来表示上述机制：

`objc\_storeWeak(&a, b)`函数：

`objc\_storeWeak`函数把第二个参数--赋值对象（b）的内存地址作为键值key，将第一个参数--weak修饰的属性变量（a）的内存地址（&a）作为value，注册到 weak 表中。如果第二个参数（b）为0（nil），那么把变量（a）的内存地址（&a）从weak表中删除，

你可以把`objc\_storeWeak(&a, b)`理解为：`objc\_storeWeak(value, key)`，并且当key变nil，将value置nil。

在b非nil时，a和b指向同一个内存地址，在b变nil时，a变nil。此时向a发送消息不会崩溃：在Objective-C中向nil发送消息是安全的。

而如果a是由 assign 修饰的，则：

在 b 非 nil 时，a 和 b 指向同一个内存地址，在 b 变 nil 时，a 还是指向该内存地址，变野指针。此时向 a 发送消息极易崩溃。

下面我们将基于`objc\_storeWeak(&a, b)`函数，使用伪代码模拟“runtime如何实现weak属性”：

```Objective-C

// 使用伪代码模拟：runtime如何实现weak属性

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

id obj1;

objc\_initWeak(&obj1, obj);

/\*obj引用计数变为0，变量作用域结束\*/

objc\_destroyWeak(&obj1);

```

下面对用到的两个方法`objc\_initWeak`和`objc\_destroyWeak`做下解释：

总体说来，作用是：

通过`objc\_initWeak`函数初始化“附有weak修饰符的变量（obj1）”，在变量作用域结束时通过`objc\_destoryWeak`函数释放该变量（obj1）。

下面分别介绍下方法的内部实现：

`objc\_initWeak`函数的实现是这样的：在将“附有weak修饰符的变量（obj1）”初始化为0（nil）后，会将“赋值对象”（obj）作为参数，调用`objc\_storeWeak`函数。

```Objective-C

obj1 = 0；

obj\_storeWeak(&obj1, obj);

```

也就是说：

> weak 修饰的指针默认值是 nil （在Objective-C中向nil发送消息是安全的）

然后`obj\_destroyWeak`函数将0（nil）作为参数，调用`objc\_storeWeak`函数。

`objc\_storeWeak(&obj1, 0);`

前面的源代码与下列源代码相同。

```Objective-C

// 使用伪代码模拟：runtime如何实现weak属性

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

id obj1;

obj1 = 0;

objc\_storeWeak(&obj1, obj);

/\* ... obj的引用计数变为0，被置nil ... \*/

objc\_storeWeak(&obj1, 0);

```

`objc\_storeWeak` 函数把第二个参数--赋值对象（obj）的内存地址作为键值，将第一个参数--weak修饰的属性变量（obj1）的内存地址注册到 weak 表中。如果第二个参数（obj）为0（nil），那么把变量（obj1）的地址从 weak 表中删除，在后面的相关一题会详解。

使用伪代码是为了方便理解，下面我们“真枪实弹”地实现下：

> 如何让不使用weak修饰的@property，拥有weak的效果。

我们从setter方法入手：

（注意以下的 `cyl\_runAtDealloc` 方法实现仅仅用于模拟原理，如果想用于项目中，还需要考虑更复杂的场景，想在实际项目使用的话，可以使用我写的一个小库，可以使用 CocoaPods 在项目中使用： [CYLDeallocBlockExecutor](https://github.com/ChenYilong/CYLDeallocBlockExecutor) ）

```Objective-C

- (void)setObject:(NSObject \*)object

{

objc\_setAssociatedObject(self, "object", object, OBJC\_ASSOCIATION\_ASSIGN);

[object cyl\_runAtDealloc:^{

\_object = nil;

}];

}

```

也就是有两个步骤：

1. 在setter方法中做如下设置：

```Objective-C

objc\_setAssociatedObject(self, "object", object, OBJC\_ASSOCIATION\_ASSIGN);

```

2. 在属性所指的对象遭到摧毁时，属性值也会清空(nil out)。做到这点，同样要借助 runtime：

```Objective-C

//要销毁的目标对象

id objectToBeDeallocated;

//可以理解为一个“事件”：当上面的目标对象销毁时，同时要发生的“事件”。

id objectWeWantToBeReleasedWhenThatHappens;

objc\_setAssociatedObject(objectToBeDeallocted,

someUniqueKey,

objectWeWantToBeReleasedWhenThatHappens,

OBJC\_ASSOCIATION\_RETAIN);

```

知道了思路，我们就开始实现 `cyl\_runAtDealloc` 方法，实现过程分两部分：

第一部分：创建一个类，可以理解为一个“事件”：当目标对象销毁时，同时要发生的“事件”。借助 block 执行“事件”。

// .h文件

```Objective-C

// .h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 这个类，可以理解为一个“事件”：当目标对象销毁时，同时要发生的“事件”。借助block执行“事件”。

typedef void (^voidBlock)(void);

@interface CYLBlockExecutor : NSObject

- (id)initWithBlock:(voidBlock)block;

@end

```

// .m文件

```Objective-C

// .m文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 这个类，可以理解为一个“事件”：当目标对象销毁时，同时要发生的“事件”。借助block执行“事件”。

#import "CYLBlockExecutor.h"

@interface CYLBlockExecutor() {

voidBlock \_block;

}

@implementation CYLBlockExecutor

- (id)initWithBlock:(voidBlock)aBlock

{

self = [super init];

if (self) {

\_block = [aBlock copy];

}

return self;

}

- (void)dealloc

{

\_block ? \_block() : nil;

}

@end

```

第二部分：核心代码：利用runtime实现`cyl\_runAtDealloc`方法

```Objective-C

// CYLNSObject+RunAtDealloc.h文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 利用runtime实现cyl\_runAtDealloc方法

#import "CYLBlockExecutor.h"

const void \*runAtDeallocBlockKey = &runAtDeallocBlockKey;

@interface NSObject (CYLRunAtDealloc)

- (void)cyl\_runAtDealloc:(voidBlock)block;

@end

// CYLNSObject+RunAtDealloc.m文件

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// 利用runtime实现cyl\_runAtDealloc方法

#import "CYLNSObject+RunAtDealloc.h"

#import "CYLBlockExecutor.h"

@implementation NSObject (CYLRunAtDealloc)

- (void)cyl\_runAtDealloc:(voidBlock)block

{

if (block) {

CYLBlockExecutor \*executor = [[CYLBlockExecutor alloc] initWithBlock:block];

objc\_setAssociatedObject(self,

runAtDeallocBlockKey,

executor,

OBJC\_ASSOCIATION\_RETAIN);

}

}

@end

```

使用方法：

导入

```Objective-C

#import "CYLNSObject+RunAtDealloc.h"

```

然后就可以使用了：

```Objective-C

NSObject \*foo = [[NSObject alloc] init];

[foo cyl\_runAtDealloc:^{

NSLog(@"正在释放foo!");

}];

```

如果对 `cyl\_runAtDealloc` 的实现原理有兴趣，可以看下我写的一个小库，可以使用 CocoaPods 在项目中使用： [CYLDeallocBlockExecutor](https://github.com/ChenYilong/CYLDeallocBlockExecutor)

参考博文： [\*\*\*Fun With the Objective-C Runtime: Run Code at Deallocation of Any Object\*\*\*](http://stackoverflow.com/a/31560217/3395008)

### 9. @property中有哪些属性关键字？/ @property 后面可以有哪些修饰符？

属性可以拥有的特质分为四类:

1. 原子性--- `nonatomic` 特质

在默认情况下，由编译器合成的方法会通过锁定机制确保其原子性(atomicity)。如果属性具备 nonatomic 特质，则不使用自旋锁。请注意，尽管没有名为“atomic”的特质(如果某属性不具备 nonatomic 特质，那它就是“原子的” ( atomic) )，但是仍然可以在属性特质中写明这一点，编译器不会报错。若是自己定义存取方法，那么就应该遵从与属性特质相符的原子性。

2. 读/写权限---`readwrite(读写)`、`readonly (只读)`

3. 内存管理语义---`assign`、`strong`、 `weak`、`unsafe\_unretained`、`copy`

4. 方法名---`getter=<name>` 、`setter=<name>`

`getter=<name>`的样式：

```Objective-C

@property (nonatomic, getter=isOn) BOOL on;

```

<p><del>（ `setter=<name>`这种不常用，也不推荐使用。故不在这里给出写法。）

</del></p>

`setter=<name>`一般用在特殊的情境下，比如：

在数据反序列化、转模型的过程中，服务器返回的字段如果以 `init` 开头，所以你需要定义一个 `init` 开头的属性，但默认生成的 `setter` 与 `getter` 方法也会以 `init` 开头，而编译器会把所有以 `init` 开头的方法当成初始化方法，而初始化方法只能返回 self 类型，因此编译器会报错。

这时你就可以使用下面的方式来避免编译器报错：

```Objective-C

@property(nonatomic, strong, getter=p\_initBy, setter=setP\_initBy:)NSString \*initBy;

```

另外也可以用关键字进行特殊说明，来避免编译器报错：

```Objective-C

@property(nonatomic, readwrite, copy, null\_resettable) NSString \*initBy;

- (NSString \*)initBy \_\_attribute\_\_((objc\_method\_family(none)));

```

3. 不常用的：`nonnull`,`null\_resettable`,`nullable`

注意：很多人会认为如果属性具备 nonatomic 特质，则不使用

“同步锁”。其实在属性设置方法中使用的是自旋锁，自旋锁相关代码如下：

```Objective-C

static inline void reallySetProperty(id self, SEL \_cmd, id newValue, ptrdiff\_t offset, bool atomic, bool copy, bool mutableCopy)

{

if (offset == 0) {

object\_setClass(self, newValue);

return;

}

id oldValue;

id \*slot = (id\*) ((char\*)self + offset);

if (copy) {

newValue = [newValue copyWithZone:nil];

} else if (mutableCopy) {

newValue = [newValue mutableCopyWithZone:nil];

} else {

if (\*slot == newValue) return;

newValue = objc\_retain(newValue);

}

if (!atomic) {

oldValue = \*slot;

\*slot = newValue;

} else {

spinlock\_t& slotlock = PropertyLocks[slot];

slotlock.lock();

oldValue = \*slot;

\*slot = newValue;

slotlock.unlock();

}

objc\_release(oldValue);

}

void objc\_setProperty(id self, SEL \_cmd, ptrdiff\_t offset, id newValue, BOOL atomic, signed char shouldCopy)

{

bool copy = (shouldCopy && shouldCopy != MUTABLE\_COPY);

bool mutableCopy = (shouldCopy == MUTABLE\_COPY);

reallySetProperty(self, \_cmd, newValue, offset, atomic, copy, mutableCopy);

}

```

### 10. weak属性需要在dealloc中置nil么？

不需要。

> 在ARC环境无论是强指针还是弱指针都无需在 dealloc 设置为 nil ， ARC 会自动帮我们处理

即便是编译器不帮我们做这些，weak也不需要在 dealloc 中置nil：

正如上文的：\*\*\*runtime 如何实现 weak 属性\*\*\* 中提到的：

我们模拟下 weak 的 setter 方法，应该如下：

```Objective-C

- (void)setObject:(NSObject \*)object

{

objc\_setAssociatedObject(self, "object", object, OBJC\_ASSOCIATION\_ASSIGN);

[object cyl\_runAtDealloc:^{

\_object = nil;

}];

}

```

如果对 `cyl\_runAtDealloc` 的实现原理有兴趣，可以看下我写的一个小库，可以使用 CocoaPods 在项目中使用： [CYLDeallocBlockExecutor](https://github.com/ChenYilong/CYLDeallocBlockExecutor)

也即:

> 在属性所指的对象遭到摧毁时，属性值也会清空(nil out)。

### 11. @synthesize和@dynamic分别有什么作用？

1. @property有两个对应的词，一个是 @synthesize，一个是 @dynamic。如果 @synthesize和 @dynamic都没写，那么默认的就是`@syntheszie var = \_var;`

2. @synthesize 的语义是如果你没有手动实现 setter 方法和 getter 方法，那么编译器会自动为你加上这两个方法。

3. @dynamic 告诉编译器：属性的 setter 与 getter 方法由用户自己实现，不自动生成。（当然对于 readonly 的属性只需提供 getter 即可）。假如一个属性被声明为 @dynamic var，然后你没有提供 @setter方法和 @getter 方法，编译的时候没问题，但是当程序运行到 `instance.var = someVar`，由于缺 setter 方法会导致程序崩溃；或者当运行到 `someVar = var` 时，由于缺 getter 方法同样会导致崩溃。编译时没问题，运行时才执行相应的方法，这就是所谓的动态绑定。

### 12. ARC下，不显式指定任何属性关键字时，默认的关键字都有哪些？

1. 对应基本数据类型默认关键字是

atomic,readwrite,assign

2. 对于普通的 Objective-C 对象

atomic,readwrite,strong

参考链接：

1. [ \*\*\*Objective-C ARC: strong vs retain and weak vs assign\*\*\* ](http://stackoverflow.com/a/15541801/3395008)

2. [ \*\*\*Variable property attributes or Modifiers in iOS\*\*\* ](http://rdcworld-iphone.blogspot.in/2012/12/variable-property-attributes-or.html)

### 13. 用@property声明的NSString（或NSArray，NSDictionary）经常使用copy关键字，为什么？如果改用strong关键字，可能造成什么问题？

1. 因为父类指针可以指向子类对象,使用 copy 的目的是为了让本对象的属性不受外界影响,使用 copy 无论给我传入是一个可变对象还是不可对象,我本身持有的就是一个不可变的副本.

2. 如果我们使用是 strong ,那么这个属性就有可能指向一个可变对象,如果这个可变对象在外部被修改了,那么会影响该属性.

copy 此特质所表达的所属关系与 strong 类似。然而设置方法并不保留新值，而是将其“拷贝” (copy)。

当属性类型为 NSString 时，经常用此特质来保护其封装性，因为传递给设置方法的新值有可能指向一个 NSMutableString 类的实例。这个类是 NSString 的子类，表示一种可修改其值的字符串，此时若是不拷贝字符串，那么设置完属性之后，字符串的值就可能会在对象不知情的情况下遭人更改。所以，这时就要拷贝一份“不可变” (immutable)的字符串，确保对象中的字符串值不会无意间变动。只要实现属性所用的对象是“可变的” (mutable)，就应该在设置新属性值时拷贝一份。

举例说明：

定义一个以 strong 修饰的 array：

```Objective-C

@property (nonatomic ,readwrite, strong) NSArray \*array;

```

然后进行下面的操作：

```Objective-C

NSArray \*array = @[ @1, @2, @3, @4 ];

NSMutableArray \*mutableArray = [NSMutableArray arrayWithArray:array];

self.array = mutableArray;

[mutableArray removeAllObjects];;

NSLog(@"%@",self.array);

[mutableArray addObjectsFromArray:array];

self.array = [mutableArray copy];

[mutableArray removeAllObjects];;

NSLog(@"%@",self.array);

```

打印结果如下所示：

```Objective-C

2015-09-27 19:10:32.523 CYLArrayCopyDmo[10681:713670] (

)

2015-09-27 19:10:32.524 CYLArrayCopyDmo[10681:713670] (

1,

2,

3,

4

)

```

（详见仓库内附录的 Demo。）

为了理解这种做法，首先要知道，两种情况：

1. 对非集合类对象的 copy 与 mutableCopy 操作；

2. 对集合类对象的 copy 与 mutableCopy 操作。

#### 1. 对非集合类对象的copy操作：

在非集合类对象中：对 immutable 对象进行 copy 操作，是指针复制，mutableCopy 操作时内容复制；对 mutable 对象进行 copy 和 mutableCopy 都是内容复制。用代码简单表示如下：

- [immutableObject copy] // 浅复制

- [immutableObject mutableCopy] //深复制

- [mutableObject copy] //深复制

- [mutableObject mutableCopy] //深复制

比如以下代码：

```Objective-C

NSMutableString \*string = [NSMutableString stringWithString:@"origin"];//copy

NSString \*stringCopy = [string copy];

```

查看内存，会发现 string、stringCopy 内存地址都不一样，说明此时都是做内容拷贝、深拷贝。即使你进行如下操作：

```Objective-C

[string appendString:@"origion!"]

```

stringCopy 的值也不会因此改变，但是如果不使用 copy，stringCopy 的值就会被改变。

集合类对象以此类推。

所以，

> 用 @property 声明 NSString、NSArray、NSDictionary 经常使用 copy 关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary，他们之间可能进行赋值操作，为确保对象中的字符串值不会无意间变动，应该在设置新属性值时拷贝一份。

#### 2、集合类对象的copy与mutableCopy

集合类对象是指 NSArray、NSDictionary、NSSet ... 之类的对象。下面先看集合类immutable对象使用 copy 和 mutableCopy 的一个例子：

```Objective-C

NSArray \*array = @[@[@"a", @"b"], @[@"c", @"d"]];

NSArray \*copyArray = [array copy];

NSMutableArray \*mCopyArray = [array mutableCopy];

```

查看内容，可以看到 copyArray 和 array 的地址是一样的，而 mCopyArray 和 array 的地址是不同的。说明 copy 操作进行了指针拷贝，mutableCopy 进行了内容拷贝。但需要强调的是：此处的内容拷贝，仅仅是拷贝 array 这个对象，array 集合内部的元素仍然是指针拷贝。这和上面的非集合 immutable 对象的拷贝还是挺相似的，那么mutable对象的拷贝会不会类似呢？我们继续往下，看 mutable 对象拷贝的例子：

```Objective-C

NSMutableArray \*array = [NSMutableArray arrayWithObjects:[NSMutableString stringWithString:@"a"],@"b",@"c",nil];

NSArray \*copyArray = [array copy];

NSMutableArray \*mCopyArray = [array mutableCopy];

```

查看内存，如我们所料，copyArray、mCopyArray和 array 的内存地址都不一样，说明 copyArray、mCopyArray 都对 array 进行了内容拷贝。同样，我们可以得出结论：

在集合类对象中，对 immutable 对象进行 copy，是指针复制， mutableCopy 是内容复制；对 mutable 对象进行 copy 和 mutableCopy 都是内容复制。但是：集合对象的内容复制仅限于对象本身，对象元素仍然是指针复制。用代码简单表示如下：

```Objective-C

[immutableObject copy] // 浅复制

[immutableObject mutableCopy] //单层深复制

[mutableObject copy] //单层深复制

[mutableObject mutableCopy] //单层深复制

```

这个代码结论和非集合类的非常相似。

参考链接：[iOS 集合的深复制与浅复制](https://www.zybuluo.com/MicroCai/note/50592)

### 14. @synthesize合成实例变量的规则是什么？假如property名为foo，存在一个名为`\_foo`的实例变量，那么还会自动合成新变量么？

在回答之前先说明下一个概念：

> 实例变量 = 成员变量 ＝ ivar

这些说法，笔者下文中，可能都会用到，指的是一个东西。

正如

[Apple官方文档 \*\*\*You Can Customize Synthesized Instance Variable Names\*\*\*](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple\_ref/doc/uid/TP40011210-CH5-SW6) 所说：

![enter image description here](http://i.imgur.com/D6d0zGJ.png)

如果使用了属性的话，那么编译器就会自动编写访问属性所需的方法，此过程叫做“自动合成”( auto synthesis)。需要强调的是，这个过程由编译器在编译期执行，所以编辑器里看不到这些“合成方法” (synthesized method)的源代码。除了生成方法代码之外，编译器还要自动向类中添加适当类型的实例变量，并且在属性名前面加下划线，以此作为实例变量的名字。

```Objective-C

@interface CYLPerson : NSObject

@property NSString \*firstName;

@property NSString \*lastName;

@end

```

在上例中，会生成两个实例变量，其名称分别为

`\_firstName` 与 `\_lastName`。也可以在类的实现代码里通过 `@synthesize` 语法来指定实例变量的名字:

```Objective-C

@implementation CYLPerson

@synthesize firstName = \_myFirstName;

@synthesize lastName = \_myLastName;

@end

```

上述语法会将生成的实例变量命名为 `\_myFirstName` 与 `\_myLastName` ，而不再使用默认的名字。一般情况下无须修改默认的实例变量名，但是如果你不喜欢以下划线来命名实例变量，那么可以用这个办法将其改为自己想要的名字。笔者还是推荐使用默认的命名方案，因为如果所有人都坚持这套方案，那么写出来的代码大家都能看得懂。

总结下 @synthesize 合成实例变量的规则，有以下几点：

1. 如果指定了成员变量的名称,会生成一个指定的名称的成员变量,

2. 如果这个成员已经存在了就不再生成了.

2. 如果是 `@synthesize foo;` 还会生成一个名称为foo的成员变量，也就是说：

> 如果没有指定成员变量的名称会自动生成一个属性同名的成员变量,

2. 如果是 `@synthesize foo = \_foo;` 就不会生成成员变量了.

假如 property 名为 foo，存在一个名为 `\_foo` 的实例变量，那么还会自动合成新变量么？

不会。如下图：

![enter image description here](http://i.imgur.com/t28ge4W.png)

### 15. 在有了自动合成属性实例变量之后，@synthesize还有哪些使用场景？

回答这个问题前，我们要搞清楚一个问题，什么情况下不会autosynthesis（自动合成）？

1. 同时重写了 setter 和 getter 时

2. 重写了只读属性的 getter 时

2. 使用了 @dynamic 时

2. 在 @protocol 中定义的所有属性

2. 在 category 中定义的所有属性

2. 重载的属性

当你在子类中重载了父类中的属性，你必须 使用 `@synthesize` 来手动合成ivar。

除了后三条，对其他几个我们可以总结出一个规律：当你想手动管理 @property 的所有内容时，你就会尝试通过实现 @property 的所有“存取方法”（the accessor methods）或者使用 `@dynamic` 来达到这个目的，这时编译器就会认为你打算手动管理 @property，于是编译器就禁用了 autosynthesis（自动合成）。

因为有了 autosynthesis（自动合成），大部分开发者已经习惯不去手动定义ivar，而是依赖于 autosynthesis（自动合成），但是一旦你需要使用ivar，而 autosynthesis（自动合成）又失效了，如果不去手动定义ivar，那么你就得借助 `@synthesize` 来手动合成 ivar。

其实，`@synthesize` 语法还有一个应用场景，但是不太建议大家使用：

可以在类的实现代码里通过 `@synthesize` 语法来指定实例变量的名字:

```Objective-C

@implementation CYLPerson

@synthesize firstName = \_myFirstName;

@synthesize lastName = \_myLastName;

@end

```

上述语法会将生成的实例变量命名为 `\_myFirstName` 与 `\_myLastName`，而不再使用默认的名字。一般情况下无须修改默认的实例变量名，但是如果你不喜欢以下划线来命名实例变量，那么可以用这个办法将其改为自己想要的名字。笔者还是推荐使用默认的命名方案，因为如果所有人都坚持这套方案，那么写出来的代码大家都能看得懂。

举例说明：应用场景：

```Objective-C

//

// .m文件

// http://weibo.com/luohanchenyilong/ (微博@iOS程序犭袁)

// https://github.com/ChenYilong

// 打开第14行和第17行中任意一行，就可编译成功

@import Foundation;

@interface CYLObject : NSObject

@property (nonatomic, copy) NSString \*title;

@end

@implementation CYLObject {

// NSString \*\_title;

}

//@synthesize title = \_title;

- (instancetype)init

{

self = [super init];

if (self) {

\_title = @"微博@iOS程序犭袁";

}

return self;

}

- (NSString \*)title {

return \_title;

}

- (void)setTitle:(NSString \*)title {

\_title = [title copy];

}

@end

```

结果编译器报错：

![enter image description here](http://i.imgur.com/fAEGHIo.png)

当你同时重写了 setter 和 getter 时，系统就不会生成 ivar（实例变量/成员变量）。这时候有两种选择：

1. 要么如第14行：手动创建 ivar

2. 要么如第17行：使用`@synthesize foo = \_foo;` ，关联 @property 与 ivar。

更多信息，请戳- 》[ \*\*\*When should I use @synthesize explicitly?\*\*\* ](http://stackoverflow.com/a/19821816/3395008)

### 16. objc中向一个nil对象发送消息将会发生什么？

在 Objective-C 中向 nil 发送消息是完全有效的——只是在运行时不会有任何作用:

1. 如果一个方法返回值是一个对象，那么发送给nil的消息将返回0(nil)。例如：

```Objective-C

Person \* motherInlaw = [[aPerson spouse] mother];

```

如果 spouse 对象为 nil，那么发送给 nil 的消息 mother 也将返回 nil。

2. 如果方法返回值为指针类型，其指针大小为小于或者等于sizeof(void\*)，float，double，long double 或者 long long 的整型标量，发送给 nil 的消息将返回0。

2. 如果方法返回值为结构体,发送给 nil 的消息将返回0。结构体中各个字段的值将都是0。

2. 如果方法的返回值不是上述提到的几种情况，那么发送给 nil 的消息的返回值将是未定义的。

具体原因如下：

> objc是动态语言，每个方法在运行时会被动态转为消息发送，即：objc\_msgSend(receiver, selector)。

那么，为了方便理解这个内容，还是贴一个objc的源代码：

```Objective-C

// runtime.h（类在runtime中的定义）

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

struct objc\_class {

Class isa OBJC\_ISA\_AVAILABILITY; //isa指针指向Meta Class，因为Objc的类的本身也是一个Object，为了处理这个关系，runtime就创造了Meta Class，当给类发送[NSObject alloc]这样消息时，实际上是把这个消息发给了Class Object

#if !\_\_OBJC2\_\_

Class super\_class OBJC2\_UNAVAILABLE; // 父类

const char \*name OBJC2\_UNAVAILABLE; // 类名

long version OBJC2\_UNAVAILABLE; // 类的版本信息，默认为0

long info OBJC2\_UNAVAILABLE; // 类信息，供运行期使用的一些位标识

long instance\_size OBJC2\_UNAVAILABLE; // 该类的实例变量大小

struct objc\_ivar\_list \*ivars OBJC2\_UNAVAILABLE; // 该类的成员变量链表

struct objc\_method\_list \*\*methodLists OBJC2\_UNAVAILABLE; // 方法定义的链表

struct objc\_cache \*cache OBJC2\_UNAVAILABLE; // 方法缓存，对象接到一个消息会根据isa指针查找消息对象，这时会在method Lists中遍历，如果cache了，常用的方法调用时就能够提高调用的效率。

struct objc\_protocol\_list \*protocols OBJC2\_UNAVAILABLE; // 协议链表

#endif

} OBJC2\_UNAVAILABLE;

```

objc在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，然后在发送消息的时候，objc\_msgSend方法不会返回值，所谓的返回内容都是具体调用时执行的。

那么，回到本题，如果向一个nil对象发送消息，首先在寻找对象的isa指针时就是0地址返回了，所以不会出现任何错误。

### 17. objc中向一个对象发送消息[obj foo]和`objc\_msgSend()`函数之间有什么关系？

具体原因同上题：该方法编译之后就是`objc\_msgSend()`函数调用.

我们用 clang 分析下，clang 提供一个命令，可以将Objective-C的源码改写成C++语言，借此可以研究下[obj foo]和`objc\_msgSend()`函数之间有什么关系。

以下面的代码为例，由于 clang 后的代码达到了10万多行，为了便于区分，添加了一个叫 iOSinit 方法，

```Objective-C

//

// main.m

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

// Copyright (c) 2015年 微博@iOS程序犭袁. All rights reserved.

//

#import "CYLTest.h"

int main(int argc, char \* argv[]) {

@autoreleasepool {

CYLTest \*test = [[CYLTest alloc] init];

[test performSelector:(@selector(iOSinit))];

return 0;

}

}

```

在终端中输入

```Objective-C

clang -rewrite-objc main.m

```

就可以生成一个`main.cpp`的文件，在最低端（10万4千行左右）

![enter image description here](http://i.imgur.com/eAH5YWn.png)

我们可以看到大概是这样的：

```Objective-C

((void ()(id, SEL))(void )objc\_msgSend)((id)obj, sel\_registerName("foo"));

```

也就是说：

> [obj foo];在objc编译时，会被转意为：`objc\_msgSend(obj, @selector(foo));`。

### 18. 什么时候会报unrecognized selector的异常？

简单来说：

> 当调用该对象上某个方法,而该对象上没有实现这个方法的时候，

可以通过“消息转发”进行解决。

简单的流程如下，在上一题中也提到过：

> objc是动态语言，每个方法在运行时会被动态转为消息发送，即：objc\_msgSend(receiver, selector)。

objc在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果，在最顶层的父类中依然找不到相应的方法时，程序在运行时会挂掉并抛出异常unrecognized selector sent to XXX 。但是在这之前，objc的运行时会给出三次拯救程序崩溃的机会：

1. Method resolution

objc运行时会调用`+resolveInstanceMethod:`或者 `+resolveClassMethod:`，让你有机会提供一个函数实现。如果你添加了函数，那运行时系统就会重新启动一次消息发送的过程，否则 ，运行时就会移到下一步，消息转发（Message Forwarding）。

2. Fast forwarding

如果目标对象实现了`-forwardingTargetForSelector:`，Runtime 这时就会调用这个方法，给你把这个消息转发给其他对象的机会。

只要这个方法返回的不是nil和self，整个消息发送的过程就会被重启，当然发送的对象会变成你返回的那个对象。否则，就会继续Normal Fowarding。

这里叫Fast，只是为了区别下一步的转发机制。因为这一步不会创建任何新的对象，但下一步转发会创建一个NSInvocation对象，所以相对更快点。

3. Normal forwarding

这一步是Runtime最后一次给你挽救的机会。首先它会发送`-methodSignatureForSelector:`消息获得函数的参数和返回值类型。如果`-methodSignatureForSelector:`返回nil，Runtime则会发出`-doesNotRecognizeSelector:`消息，程序这时也就挂掉了。如果返回了一个函数签名，Runtime就会创建一个NSInvocation对象并发送`-forwardInvocation:`消息给目标对象。

为了能更清晰地理解这些方法的作用，git仓库里也给出了一个Demo，名称叫“ `\_objc\_msgForward\_demo` ”,可运行起来看看。

### 19. 一个objc对象如何进行内存布局？（考虑有父类的情况）

- 所有父类的成员变量和自己的成员变量都会存放在该对象所对应的存储空间中.

- 每一个对象内部都有一个isa指针,指向他的类对象,类对象中存放着本对象的

1. 对象方法列表（对象能够接收的消息列表，保存在它所对应的类对象中）

2. 成员变量的列表,

2. 属性列表,

它内部也有一个isa指针指向元对象(meta class),元对象内部存放的是类方法列表,类对象内部还有一个superclass的指针,指向他的父类对象。

每个 Objective-C 对象都有相同的结构，如下图所示：

![enter image description here](http://i.imgur.com/7mJlUj1.png)

翻译过来就是

| Objective-C 对象的结构图 |

------------- |

ISA指针 |

根类的实例变量 |

倒数第二层父类的实例变量 |

... |

父类的实例变量 |

类的实例变量 |

- 根对象就是NSObject，它的superclass指针指向nil

- 类对象既然称为对象，那它也是一个实例。类对象中也有一个isa指针指向它的元类(meta class)，即类对象是元类的实例。元类内部存放的是类方法列表，根元类的isa指针指向自己，superclass指针指向NSObject类。

如图:

![enter image description here](http://i.imgur.com/w6tzFxz.png)

### 20. 一个objc对象的isa的指针指向什么？有什么作用？

指向他的类对象,从而可以找到对象上的方法

### 21. 下面的代码输出什么？

```Objective-C

@implementation Son : Father

- (id)init

{

self = [super init];

if (self) {

NSLog(@"%@", NSStringFromClass([self class]));

NSLog(@"%@", NSStringFromClass([super class]));

}

return self;

}

@end

```

\*\*答案：\*\*

都输出 Son

NSStringFromClass([self class]) = Son

NSStringFromClass([super class]) = Son

这个题目主要是考察关于 Objective-C 中对 self 和 super 的理解。

我们都知道：self 是类的隐藏参数，指向当前调用方法的这个类的实例。那 super 呢？

很多人会想当然的认为“ super 和 self 类似，应该是指向父类的指针吧！”。这是很普遍的一个误区。其实 super 是一个 Magic Keyword， 它本质是一个编译器标示符，和 self 是指向的同一个消息接受者！他们两个的不同点在于：super 会告诉编译器，调用 class 这个方法时，要去父类的方法，而不是本类里的。

上面的例子不管调用`[self class]`还是`[super class]`，接受消息的对象都是当前 `Son ＊xxx` 这个对象。

当使用 self 调用方法时，会从当前类的方法列表中开始找，如果没有，就从父类中再找；而当使用 super 时，则从父类的方法列表中开始找。然后调用父类的这个方法。

这也就是为什么说“不推荐在 init 方法中使用点语法”，如果想访问实例变量 iVar 应该使用下划线（ `\_iVar` ），而非点语法（ `self.iVar` ）。

点语法（ `self.iVar` ）的坏处就是子类有可能覆写 setter 。假设 Person 有一个子类叫 ChenPerson，这个子类专门表示那些姓“陈”的人。该子类可能会覆写 lastName 属性所对应的设置方法：

```Objective-C

//

// ChenPerson.m

//

//

// Created by https://github.com/ChenYilong on 15/8/30.

// Copyright (c) 2015年 http://weibo.com/luohanchenyilong/ 微博@iOS程序犭袁. All rights reserved.

//

#import "ChenPerson.h"

@implementation ChenPerson

@synthesize lastName = \_lastName;

- (instancetype)init

{

self = [super init];

if (self) {

NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, NSStringFromClass([self class]));

NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, NSStringFromClass([super class]));

}

return self;

}

- (void)setLastName:(NSString\*)lastName

{

//设置方法一：如果setter采用是这种方式，就可能引起崩溃

// if (![lastName isEqualToString:@"陈"])

// {

// [NSException raise:NSInvalidArgumentException format:@"姓不是陈"];

// }

// \_lastName = lastName;

//设置方法二：如果setter采用是这种方式，就可能引起崩溃

\_lastName = @"陈";

NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, @"会调用这个方法,想一下为什么？");

}

@end

```

在基类 Person 的默认初始化方法中，可能会将姓氏设为空字符串。此时若使用点语法（ `self.lastName` ）也即 setter 设置方法，那么调用将会是子类的设置方法，如果在刚刚的 setter 代码中采用设置方法一，那么就会抛出异常，

为了方便采用打印的方式展示，究竟发生了什么，我们使用设置方法二。

如果基类的代码是这样的：

```Objective-C

//

// Person.m

// nil对象调用点语法

//

// Created by https://github.com/ChenYilong on 15/8/29.

// Copyright (c) 2015年 http://weibo.com/luohanchenyilong/ 微博@iOS程序犭袁. All rights reserved.

//

#import "Person.h"

@implementation Person

- (instancetype)init

{

self = [super init];

if (self) {

self.lastName = @"";

//NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, NSStringFromClass([self class]));

//NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, self.lastName);

}

return self;

}

- (void)setLastName:(NSString\*)lastName

{

NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", \_\_PRETTY\_FUNCTION\_\_, \_\_LINE\_\_, @"根本不会调用这个方法");

\_lastName = @"炎黄";

}

@end

```

那么打印结果将会是这样的：

```Objective-C

🔴类名与方法名：-[ChenPerson setLastName:]（在第36行），描述：会调用这个方法,想一下为什么？

🔴类名与方法名：-[ChenPerson init]（在第19行），描述：ChenPerson

🔴类名与方法名：-[ChenPerson init]（在第20行），描述：ChenPerson

```

我在仓库里也给出了一个相应的 Demo（名字叫：Demo\_21题\_下面的代码输出什么）。有兴趣可以跑起来看一下，主要看下他是怎么打印的，思考下为什么这么打印。

接下来让我们利用 runtime 的相关知识来验证一下 super 关键字的本质，使用clang重写命令:

```Objective-C

$ clang -rewrite-objc test.m

```

将这道题目中给出的代码被转化为:

```Objective-C

NSLog((NSString \*)&\_\_NSConstantStringImpl\_\_var\_folders\_gm\_0jk35cwn1d3326x0061qym280000gn\_T\_main\_a5cecc\_mi\_0, NSStringFromClass(((Class (\*)(id, SEL))(void \*)objc\_msgSend)((id)self, sel\_registerName("class"))));

NSLog((NSString \*)&\_\_NSConstantStringImpl\_\_var\_folders\_gm\_0jk35cwn1d3326x0061qym280000gn\_T\_main\_a5cecc\_mi\_1, NSStringFromClass(((Class (\*)(\_\_rw\_objc\_super \*, SEL))(void \*)objc\_msgSendSuper)((\_\_rw\_objc\_super){ (id)self, (id)class\_getSuperclass(objc\_getClass("Son")) }, sel\_registerName("class"))));

```

从上面的代码中，我们可以发现在调用 [self class] 时，会转化成 `objc\_msgSend`函数。看下函数定义：

```Objective-C

id objc\_msgSend(id self, SEL op, ...)

```

我们把 self 做为第一个参数传递进去。

而在调用 [super class]时，会转化成 `objc\_msgSendSuper`函数。看下函数定义:

```Objective-C

id objc\_msgSendSuper(struct objc\_super \*super, SEL op, ...)

```

第一个参数是 `objc\_super` 这样一个结构体，其定义如下:

```Objective-C

struct objc\_super {

\_\_unsafe\_unretained id receiver;

\_\_unsafe\_unretained Class super\_class;

};

```

结构体有两个成员，第一个成员是 receiver, 类似于上面的 `objc\_msgSend`函数第一个参数self 。第二个成员是记录当前类的父类是什么。

所以，当调用 ［self class] 时，实际先调用的是 `objc\_msgSend`函数，第一个参数是 Son当前的这个实例，然后在 Son 这个类里面去找 - (Class)class这个方法，没有，去父类 Father里找，也没有，最后在 NSObject类中发现这个方法。而 - (Class)class的实现就是返回self的类别，故上述输出结果为 Son。

objc Runtime开源代码对- (Class)class方法的实现:

```Objective-C

- (Class)class {

return object\_getClass(self);

}

```

而当调用 `[super class]`时，会转换成`objc\_msgSendSuper函数`。第一步先构造 `objc\_super` 结构体，结构体第一个成员就是 `self` 。

第二个成员是 `(id)class\_getSuperclass(objc\_getClass(“Son”))` , 实际该函数输出结果为 Father。

第二步是去 Father这个类里去找 `- (Class)class`，没有，然后去NSObject类去找，找到了。最后内部是使用 `objc\_msgSend(objc\_super->receiver, @selector(class))`去调用，

此时已经和`[self class]`调用相同了，故上述输出结果仍然返回 Son。

参考链接：[微博@Chun\_iOS](http://weibo.com/junbbcom)的博文[刨根问底Objective－C Runtime（1）－ Self & Super](http://chun.tips/blog/2014/11/05/bao-gen-wen-di-objective%5Bnil%5Dc-runtime(1)%5Bnil%5D-self-and-super/)

### 22. runtime如何通过selector找到对应的IMP地址？（分别考虑类方法和实例方法）

每一个类对象中都一个方法列表,方法列表中记录着方法的名称,方法实现,以及参数类型,其实selector本质就是方法名称,通过这个方法名称就可以在方法列表中找到对应的方法实现.

### 23. 使用runtime Associate方法关联的对象，需要在主对象dealloc的时候释放么？

- 在ARC下不需要。

- <p><del> 在MRC中,对于使用retain或copy策略的需要 。</del></p>在MRC下也不需要

> 无论在MRC下还是ARC下均不需要。

[ \*\*\*2011年版本的Apple API 官方文档 - Associative References\*\*\* ](https://web.archive.org/web/20120818164935/http://developer.apple.com/library/ios/#/web/20120820002100/http://developer.apple.com/library/ios/documentation/cocoa/conceptual/objectivec/Chapters/ocAssociativeReferences.html) 一节中有一个MRC环境下的例子：

```Objective-C

// 在MRC下，使用runtime Associate方法关联的对象，不需要在主对象dealloc的时候释放

// http://weibo.com/luohanchenyilong/ (微博@iOS程序犭袁)

// https://github.com/ChenYilong

// 摘自2011年版本的Apple API 官方文档 - Associative References

static char overviewKey;

NSArray \*array =

[[NSArray alloc] initWithObjects:@"One", @"Two", @"Three", nil];

// For the purposes of illustration, use initWithFormat: to ensure

// the string can be deallocated

NSString \*overview =

[[NSString alloc] initWithFormat:@"%@", @"First three numbers"];

objc\_setAssociatedObject (

array,

&overviewKey,

overview,

OBJC\_ASSOCIATION\_RETAIN

);

[overview release];

// (1) overview valid

[array release];

// (2) overview invalid

```

文档指出

> At point 1, the string `overview` is still valid because the `OBJC\_ASSOCIATION\_RETAIN` policy specifies that the array retains the associated object. When the array is deallocated, however (at point 2), `overview` is released and so in this case also deallocated.

我们可以看到，在`[array release];`之后，overview就会被release释放掉了。

既然会被销毁，那么具体在什么时间点？

> 根据[ \*\*\*WWDC 2011, Session 322 (第36分22秒)\*\*\* ](https://developer.apple.com/videos/wwdc/2011/#322-video)中发布的内存销毁时间表，被关联的对象在生命周期内要比对象本身释放的晚很多。它们会在被 NSObject -dealloc 调用的 object\_dispose() 方法中释放。

对象的内存销毁时间表，分四个步骤：

// 对象的内存销毁时间表

// http://weibo.com/luohanchenyilong/ (微博@iOS程序犭袁)

// https://github.com/ChenYilong

// 根据 WWDC 2011, Session 322 (36分22秒)中发布的内存销毁时间表

1. 调用 -release ：引用计数变为零

\* 对象正在被销毁，生命周期即将结束.

\* 不能再有新的 \_\_weak 弱引用， 否则将指向 nil.

\* 调用 [self dealloc]

2. 子类 调用 -dealloc

\* 继承关系中最底层的子类 在调用 -dealloc

\* 如果是 MRC 代码 则会手动释放实例变量们（iVars）

\* 继承关系中每一层的父类 都在调用 -dealloc

3. NSObject 调 -dealloc

\* 只做一件事：调用 Objective-C runtime 中的 object\_dispose() 方法

4. 调用 object\_dispose()

\* 为 C++ 的实例变量们（iVars）调用 destructors

\* 为 ARC 状态下的 实例变量们（iVars） 调用 -release

\* 解除所有使用 runtime Associate方法关联的对象

\* 解除所有 \_\_weak 引用

\* 调用 free()

对象的内存销毁时间表：[参考链接](http://stackoverflow.com/a/10843510/3395008)。

### 24. objc中的类方法和实例方法有什么本质区别和联系？

类方法：

1. 类方法是属于类对象的

2. 类方法只能通过类对象调用

2. 类方法中的self是类对象

2. 类方法可以调用其他的类方法

2. 类方法中不能访问成员变量

2. 类方法中不能直接调用对象方法

实例方法：

1. 实例方法是属于实例对象的

2. 实例方法只能通过实例对象调用

2. 实例方法中的self是实例对象

2. 实例方法中可以访问成员变量

2. 实例方法中直接调用实例方法

2. 实例方法中也可以调用类方法(通过类名)

下一篇文章将发布在[这里](https://github.com/ChenYilong/iOSInterviewQuestions)，会对以下问题进行总结，并将本篇文章的勘误一并列出，欢迎指正！请持续关注[微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)

----------

@property部分主要参考

[Apple官方文档：Properties Encapsulate an Object’s Values](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple\_ref/doc/uid/TP40011210-CH5-SW2)

runtime部分主要参考[Apple官方文档：Declared Properties](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtPropertyIntrospection.html)

### 25. `\_objc\_msgForward`函数是做什么的，直接调用它将会发生什么？

### 26. runtime如何实现weak变量的自动置nil？

### 27. 能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？

### 28. runloop和线程有什么关系？

### 29. runloop的mode作用是什么？

### 30. 以+ scheduledTimerWithTimeInterval...的方式触发的timer，在滑动页面上的列表时，timer会暂定回调，为什么？如何解决？

### 31. 猜想runloop内部是如何实现的？

### 32. objc使用什么机制管理对象内存？

### 33. ARC通过什么方式帮助开发者管理内存？

### 34. 不手动指定autoreleasepool的前提下，一个autorealese对象在什么时刻释放？（比如在一个vc的viewDidLoad中创建）

### 35. `BAD\_ACCESS`在什么情况下出现？

### 36. 苹果是如何实现autoreleasepool的？

### 37. 使用block时什么情况会发生引用循环，如何解决？

### 38. 在block内如何修改block外部变量？

### 39. 使用系统的某些block api（如UIView的block版本写动画时），是否也考虑引用循环问题？

### 40. GCD的队列（`dispatch\_queue\_t`）分哪两种类型？

### 41. 如何用GCD同步若干个异步调用？（如根据若干个url异步加载多张图片，然后在都下载完成后合成一张整图）

### 42. `dispatch\_barrier\_async`的作用是什么？

### 43. 苹果为什么要废弃`dispatch\_get\_current\_queue`？

### 44. 以下代码运行结果如何？

- (void)viewDidLoad

{

[super viewDidLoad];

NSLog(@"1");

dispatch\_sync(dispatch\_get\_main\_queue(), ^{

NSLog(@"2");

});

NSLog(@"3");

}

### 45. addObserver:forKeyPath:options:context:各个参数的作用分别是什么，observer中需要实现哪个方法才能获得KVO回调？

### 46. 如何手动触发一个value的KVO

### 47. 若一个类有实例变量`NSString \*\_foo`，调用setValue:forKey:时，可以以foo还是`\_foo`作为key？

### 48. KVC的keyPath中的集合运算符如何使用？

### 49. KVC和KVO的keyPath一定是属性么？

### 50. 如何关闭默认的KVO的默认实现，并进入自定义的KVO实现？

### 51. apple用什么方式实现对一个对象的KVO？

### 52. IBOutlet连出来的视图属性为什么可以被设置成weak?

### 53. IB中User Defined Runtime Attributes如何使用？

### 54. 如何调试`BAD\_ACCESS`错误

### 55. lldb（gdb）常用的调试命令？

----------

[《招聘一个靠谱的 iOS》](http://blog.sunnyxx.com/2015/07/04/ios-interview/)—参考答案（下）

说明：面试题来源是[微博@我就叫Sunny怎么了](http://weibo.com/u/1364395395)的这篇博文：[《招聘一个靠谱的 iOS》](http://blog.sunnyxx.com/2015/07/04/ios-interview/)，其中共55题，除第一题为纠错题外，其他54道均为简答题。

出题者简介： 孙源（sunnyxx），目前就职于百度，负责百度知道 iOS 客户端的开发工作，对技术喜欢刨根问底和总结最佳实践，热爱分享和开源，维护一个叫 forkingdog 的开源小组。

答案为[微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)整理，未经出题者校对，如有纰漏，请向[微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)指正。

----------

# 索引

1. [ 25. `\_objc\_msgForward` 函数是做什么的，直接调用它将会发生什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#25-\_objc\_msgforward函数是做什么的直接调用它将会发生什么)

2. [26. runtime如何实现weak变量的自动置nil？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#26-runtime如何实现weak变量的自动置nil)

3. [27. 能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#27-能否向编译后得到的类中增加实例变量能否向运行时创建的类中添加实例变量为什么)

4. [28. runloop和线程有什么关系？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#28-runloop和线程有什么关系)

5. [29. runloop的mode作用是什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#29-runloop的mode作用是什么)

6. [30. 以+ scheduledTimerWithTimeInterval...的方式触发的timer，在滑动页面上的列表时，timer会暂定回调，为什么？如何解决？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#30-以-scheduledtimerwithtimeinterval的方式触发的timer在滑动页面上的列表时timer会暂定回调为什么如何解决)

7. [31. 猜想runloop内部是如何实现的？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#31-猜想runloop内部是如何实现的)

8. [32. objc使用什么机制管理对象内存？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#32-objc使用什么机制管理对象内存)

9. [33. ARC通过什么方式帮助开发者管理内存？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#33-arc通过什么方式帮助开发者管理内存)

10. [34. 不手动指定autoreleasepool的前提下，一个autorealese对象在什么时刻释放？（比如在一个vc的viewDidLoad中创建）](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#34-不手动指定autoreleasepool的前提下一个autorealese对象在什么时刻释放比如在一个vc的viewdidload中创建)

11. [35. BAD\_ACCESS在什么情况下出现？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#35-bad\_access在什么情况下出现)

12. [36. 苹果是如何实现autoreleasepool的？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#36-苹果是如何实现autoreleasepool的)

13. [37. 使用block时什么情况会发生引用循环，如何解决？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#37-使用block时什么情况会发生引用循环如何解决)

14. [38. 在block内如何修改block外部变量？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#38-在block内如何修改block外部变量)

15. [39. 使用系统的某些block api（如UIView的block版本写动画时），是否也考虑引用循环问题？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#39-使用系统的某些block-api如uiview的block版本写动画时是否也考虑引用循环问题)

16. [40. GCD的队列（dispatch\_queue\_t）分哪两种类型？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#40-gcd的队列dispatch\_queue\_t分哪两种类型)

17. [41. 如何用GCD同步若干个异步调用？（如根据若干个url异步加载多张图片，然后在都下载完成后合成一张整图）](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#41-如何用gcd同步若干个异步调用如根据若干个url异步加载多张图片然后在都下载完成后合成一张整图)

18. [42. dispatch\_barrier\_async的作用是什么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#42-dispatch\_barrier\_async的作用是什么)

19. [43. 苹果为什么要废弃dispatch\_get\_current\_queue？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#43-苹果为什么要废弃dispatch\_get\_current\_queue)

0. [44. 以下代码运行结果如何？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#44-以下代码运行结果如何)

```Objective-C

- (void)viewDidLoad

{

[super viewDidLoad];

NSLog(@"1");

dispatch\_sync(dispatch\_get\_main\_queue(), ^{

NSLog(@"2");

});

NSLog(@"3");

}

```

1. [45. addObserver:forKeyPath:options:context:各个参数的作用分别是什么，observer中需要实现哪个方法才能获得KVO回调？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#45-addobserverforkeypathoptionscontext各个参数的作用分别是什么observer中需要实现哪个方法才能获得kvo回调)

2. [46. 如何手动触发一个value的KVO](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#46-如何手动触发一个value的kvo)

3. [47. 若一个类有实例变量 NSString \*\_foo ，调用setValue:forKey:时，可以以foo还是 \_foo 作为key？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#47-若一个类有实例变量-nsstring-\_foo-调用setvalueforkey时可以以foo还是-\_foo-作为key)

4. [48. KVC的keyPath中的集合运算符如何使用？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#48-kvc的keypath中的集合运算符如何使用)

5. [49. KVC和KVO的keyPath一定是属性么？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#49-kvc和kvo的keypath一定是属性么)

6. [50. 如何关闭默认的KVO的默认实现，并进入自定义的KVO实现？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#50-如何关闭默认的kvo的默认实现并进入自定义的kvo实现)

7. [51. apple用什么方式实现对一个对象的KVO？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#51-apple用什么方式实现对一个对象的kvo)

8. [52. IBOutlet连出来的视图属性为什么可以被设置成weak?](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#52-iboutlet连出来的视图属性为什么可以被设置成weak)

9. [53. IB中User Defined Runtime Attributes如何使用？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#53-ib中user-defined-runtime-attributes如何使用)

0. [54. 如何调试BAD\_ACCESS错误](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#54-如何调试bad\_access错误)

1. [55. lldb（gdb）常用的调试命令？](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#55-lldbgdb常用的调试命令)

----------

### 25. `\_objc\_msgForward`函数是做什么的，直接调用它将会发生什么？

> `\_objc\_msgForward`是 IMP 类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，`\_objc\_msgForward`会尝试做消息转发。

我们可以这样创建一个`\_objc\_msgForward`对象：

IMP msgForwardIMP = \_objc\_msgForward;

在[上篇](https://github.com/ChenYilong/iOSInterviewQuestions)中的《objc中向一个对象发送消息`[obj foo]`和`objc\_msgSend()`函数之间有什么关系？》曾提到`objc\_msgSend`在“消息传递”中的作用。在“消息传递”过程中，`objc\_msgSend`的动作比较清晰：首先在 Class 中的缓存查找 IMP （没缓存则初始化缓存），如果没找到，则向父类的 Class 查找。如果一直查找到根类仍旧没有实现，则用`\_objc\_msgForward`函数指针代替 IMP 。最后，执行这个 IMP 。

Objective-C运行时是开源的，所以我们可以看到它的实现。打开[ \*\*\*Apple Open Source 里Mac代码里的obj包\*\*\* ](http://www.opensource.apple.com/tarballs/objc4/)下载一个最新版本，找到 `objc-runtime-new.mm`，进入之后搜索`\_objc\_msgForward`。

![enter image description here](http://i.imgur.com/rGBfaoL.png)

里面有对`\_objc\_msgForward`的功能解释：

![enter image description here](http://i.imgur.com/vcThcdA.png)

```Objective-C

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* lookUpImpOrForward.

\* The standard IMP lookup.

\* initialize==NO tries to avoid +initialize (but sometimes fails)

\* cache==NO skips optimistic unlocked lookup (but uses cache elsewhere)

\* Most callers should use initialize==YES and cache==YES.

\* inst is an instance of cls or a subclass thereof, or nil if none is known.

\* If cls is an un-initialized metaclass then a non-nil inst is faster.

\* May return \_objc\_msgForward\_impcache. IMPs destined for external use

\* must be converted to \_objc\_msgForward or \_objc\_msgForward\_stret.

\* If you don't want forwarding at all, use lookUpImpOrNil() instead.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

```

对 `objc-runtime-new.mm`文件里与`\_objc\_msgForward`有关的三个函数使用伪代码展示下：

```Objective-C

// objc-runtime-new.mm 文件里与 \_objc\_msgForward 有关的三个函数使用伪代码展示

// Created by https://github.com/ChenYilong

// Copyright (c) 微博@iOS程序犭袁(http://weibo.com/luohanchenyilong/). All rights reserved.

// 同时，这也是 obj\_msgSend 的实现过程

id objc\_msgSend(id self, SEL op, ...) {

if (!self) return nil;

IMP imp = class\_getMethodImplementation(self->isa, SEL op);

imp(self, op, ...); //调用这个函数，伪代码...

}

//查找IMP

IMP class\_getMethodImplementation(Class cls, SEL sel) {

if (!cls || !sel) return nil;

IMP imp = lookUpImpOrNil(cls, sel);

if (!imp) return \_objc\_msgForward; //\_objc\_msgForward 用于消息转发

return imp;

}

IMP lookUpImpOrNil(Class cls, SEL sel) {

if (!cls->initialize()) {

\_class\_initialize(cls);

}

Class curClass = cls;

IMP imp = nil;

do { //先查缓存,缓存没有时重建,仍旧没有则向父类查询

if (!curClass) break;

if (!curClass->cache) fill\_cache(cls, curClass);

imp = cache\_getImp(curClass, sel);

if (imp) break;

} while (curClass = curClass->superclass);

return imp;

}

```

虽然Apple没有公开`\_objc\_msgForward`的实现源码，但是我们还是能得出结论：

> `\_objc\_msgForward`是一个函数指针（和 IMP 的类型一样），是用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，`\_objc\_msgForward`会尝试做消息转发。

> 在[上篇](https://github.com/ChenYilong/iOSInterviewQuestions)中的《objc中向一个对象发送消息`[obj foo]`和`objc\_msgSend()`函数之间有什么关系？》曾提到`objc\_msgSend`在“消息传递”中的作用。在“消息传递”过程中，`objc\_msgSend`的动作比较清晰：首先在 Class 中的缓存查找 IMP （没缓存则初始化缓存），如果没找到，则向父类的 Class 查找。如果一直查找到根类仍旧没有实现，则用`\_objc\_msgForward`函数指针代替 IMP 。最后，执行这个 IMP 。

为了展示消息转发的具体动作，这里尝试向一个对象发送一条错误的消息，并查看一下`\_objc\_msgForward`是如何进行转发的。

首先开启调试模式、打印出所有运行时发送的消息：

可以在代码里执行下面的方法：

```Objective-C

(void)instrumentObjcMessageSends(YES);

```

或者断点暂停程序运行，并在 gdb 中输入下面的命令：

```Objective-C

call (void)instrumentObjcMessageSends(YES)

```

以第二种为例，操作如下所示：

![enter image description here](http://i.imgur.com/uEwTCC4.png)

之后，运行时发送的所有消息都会打印到`/tmp/msgSend-xxxx`文件里了。

终端中输入命令前往：

```Objective-C

open /private/tmp

```

![enter image description here](http://i.imgur.com/Fh5hhCw.png)

可能看到有多条，找到最新生成的，双击打开

在模拟器上执行执行以下语句（这一套调试方案仅适用于模拟器，真机不可用，关于该调试方案的拓展链接：[ \*\*\*Can the messages sent to an object in Objective-C be monitored or printed out?\*\*\* ](http://stackoverflow.com/a/10750398/3395008)），向一个对象发送一条错误的消息：

```Objective-C

//

// main.m

// CYLObjcMsgForwardTest

//

// Created by http://weibo.com/luohanchenyilong/.

// Copyright (c) 2015年 微博@iOS程序犭袁. All rights reserved.

//

#import <UIKit/UIKit.h>

#import "AppDelegate.h"

#import "CYLTest.h"

int main(int argc, char \* argv[]) {

@autoreleasepool {

CYLTest \*test = [[CYLTest alloc] init];

[test performSelector:(@selector(iOS程序犭袁))];

return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));

}

}

```

![enter image description here](http://i.imgur.com/UjbmVvB.png)

你可以在`/tmp/msgSend-xxxx`（我这一次是`/tmp/msgSend-9805`）文件里，看到打印出来：

![enter image description here](http://i.imgur.com/AAERz1T.png)

```Objective-C

+ CYLTest NSObject initialize

+ CYLTest NSObject alloc

- CYLTest NSObject init

- CYLTest NSObject performSelector:

+ CYLTest NSObject resolveInstanceMethod:

+ CYLTest NSObject resolveInstanceMethod:

- CYLTest NSObject forwardingTargetForSelector:

- CYLTest NSObject forwardingTargetForSelector:

- CYLTest NSObject methodSignatureForSelector:

- CYLTest NSObject methodSignatureForSelector:

- CYLTest NSObject class

- CYLTest NSObject doesNotRecognizeSelector:

- CYLTest NSObject doesNotRecognizeSelector:

- CYLTest NSObject class

```

结合[《NSObject官方文档》](https://developer.apple.com/library/prerelease/watchos/documentation/Cocoa/Reference/Foundation/Classes/NSObject\_Class/#//apple\_ref/doc/uid/20000050-SW11)，排除掉 NSObject 做的事，剩下的就是`\_objc\_msgForward`消息转发做的几件事：

1. 调用`resolveInstanceMethod:`方法 (或 `resolveClassMethod:`)。允许用户在此时为该 Class 动态添加实现。如果有实现了，则调用并返回YES，那么重新开始`objc\_msgSend`流程。这一次对象会响应这个选择器，一般是因为它已经调用过`class\_addMethod`。如果仍没实现，继续下面的动作。

2. 调用`forwardingTargetForSelector:`方法，尝试找到一个能响应该消息的对象。如果获取到，则直接把消息转发给它，返回非 nil 对象。否则返回 nil ，继续下面的动作。注意，这里不要返回 self ，否则会形成死循环。

3. 调用`methodSignatureForSelector:`方法，尝试获得一个方法签名。如果获取不到，则直接调用`doesNotRecognizeSelector`抛出异常。如果能获取，则返回非nil：创建一个 NSlnvocation 并传给`forwardInvocation:`。

4. 调用`forwardInvocation:`方法，将第3步获取到的方法签名包装成 Invocation 传入，如何处理就在这里面了，并返回非ni。

5. 调用`doesNotRecognizeSelector:` ，默认的实现是抛出异常。如果第3步没能获得一个方法签名，执行该步骤。

上面前4个方法均是模板方法，开发者可以override，由 runtime 来调用。最常见的实现消息转发：就是重写方法3和4，吞掉一个消息或者代理给其他对象都是没问题的

也就是说`\_objc\_msgForward`在进行消息转发的过程中会涉及以下这几个方法：

1. `resolveInstanceMethod:`方法 (或 `resolveClassMethod:`)。

2. `forwardingTargetForSelector:`方法

3. `methodSignatureForSelector:`方法

4. `forwardInvocation:`方法

5. `doesNotRecognizeSelector:` 方法

为了能更清晰地理解这些方法的作用，git仓库里也给出了一个Demo，名称叫“ `\_objc\_msgForward\_demo` ”,可运行起来看看。

下面回答下第二个问题“直接`\_objc\_msgForward`调用它将会发生什么？”

直接调用`\_objc\_msgForward`是非常危险的事，如果用不好会直接导致程序Crash，但是如果用得好，能做很多非常酷的事。

就好像跑酷，干得好，叫“耍酷”，干不好就叫“作死”。

正如前文所说：

> `\_objc\_msgForward`是 IMP 类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，`\_objc\_msgForward`会尝试做消息转发。

如何调用`\_objc\_msgForward`？

`\_objc\_msgForward`隶属 C 语言，有三个参数 ：

|--| `\_objc\_msgForward`参数| 类型 |

-------------|-------------|-------------

1.| 所属对象 | id类型

2. |方法名 | SEL类型

3. |可变参数 |可变参数类型

首先了解下如何调用 IMP 类型的方法，IMP类型是如下格式：

为了直观，我们可以通过如下方式定义一个 IMP类型 ：

```Objective-C

typedef void (\*voidIMP)(id, SEL, ...)

```

一旦调用`\_objc\_msgForward`，将跳过查找 IMP 的过程，直接触发“消息转发”，

如果调用了`\_objc\_msgForward`，即使这个对象确实已经实现了这个方法，你也会告诉`objc\_msgSend`：

> “我没有在这个对象里找到这个方法的实现”

想象下`objc\_msgSend`会怎么做？通常情况下，下面这张图就是你正常走`objc\_msgSend`过程，和直接调用`\_objc\_msgForward`的前后差别：

![enter image description here](http://ww1.sinaimg.cn/bmiddle/6628711bgw1eecx3jef23g206404tkbi.gif)

有哪些场景需要直接调用`\_objc\_msgForward`？最常见的场景是：你想获取某方法所对应的`NSInvocation`对象。举例说明：

[JSPatch （Github 链接）](https://github.com/bang590/JSPatch)就是直接调用`\_objc\_msgForward`来实现其核心功能的：

> JSPatch 以小巧的体积做到了让JS调用/替换任意OC方法，让iOS APP具备热更新的能力。

作者的博文[《JSPatch实现原理详解》](http://blog.cnbang.net/tech/2808/)详细记录了实现原理，有兴趣可以看下。

同时 [ \*\*\*RAC(ReactiveCocoa)\*\*\* ](https://github.com/ReactiveCocoa/ReactiveCocoa) 源码中也用到了该方法。

### 26. runtime如何实现weak变量的自动置nil？

> runtime 对注册的类， 会进行布局，对于 weak 对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为 key，当此对象的引用计数为0的时候会 dealloc，假如 weak 指向的对象内存地址是a，那么就会以a为键， 在这个 weak 表中搜索，找到所有以a为键的 weak 对象，从而设置为 nil。

在[上篇](https://github.com/ChenYilong/iOSInterviewQuestions)中的《runtime 如何实现 weak 属性》有论述。（注：在[上篇](https://github.com/ChenYilong/iOSInterviewQuestions)的《使用runtime Associate方法关联的对象，需要在主对象dealloc的时候释放么？》里给出的“对象的内存销毁时间表”也提到`\_\_weak`引用的解除时间。）

我们可以设计一个函数（伪代码）来表示上述机制：

`objc\_storeWeak(&a, b)`函数：

`objc\_storeWeak`函数把第二个参数--赋值对象（b）的内存地址作为键值key，将第一个参数--weak修饰的属性变量（a）的内存地址（&a）作为value，注册到 weak 表中。如果第二个参数（b）为0（nil），那么把变量（a）的内存地址（&a）从weak表中删除，

你可以把`objc\_storeWeak(&a, b)`理解为：`objc\_storeWeak(value, key)`，并且当key变nil，将value置nil。

在b非nil时，a和b指向同一个内存地址，在b变nil时，a变nil。此时向a发送消息不会崩溃：在Objective-C中向nil发送消息是安全的。

而如果a是由assign修饰的，则：

在b非nil时，a和b指向同一个内存地址，在b变nil时，a还是指向该内存地址，变野指针。此时向a发送消息极易崩溃。

下面我们将基于`objc\_storeWeak(&a, b)`函数，使用伪代码模拟“runtime如何实现weak属性”：

```Objective-C

// 使用伪代码模拟：runtime如何实现weak属性

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

id obj1;

objc\_initWeak(&obj1, obj);

/\*obj引用计数变为0，变量作用域结束\*/

objc\_destroyWeak(&obj1);

```

下面对用到的两个方法`objc\_initWeak`和`objc\_destroyWeak`做下解释：

总体说来，作用是：

通过`objc\_initWeak`函数初始化“附有weak修饰符的变量（obj1）”，在变量作用域结束时通过`objc\_destoryWeak`函数释放该变量（obj1）。

下面分别介绍下方法的内部实现：

`objc\_initWeak`函数的实现是这样的：在将“附有weak修饰符的变量（obj1）”初始化为0（nil）后，会将“赋值对象”（obj）作为参数，调用`objc\_storeWeak`函数。

```Objective-C

obj1 = 0；

obj\_storeWeak(&obj1, obj);

```

也就是说：

> weak 修饰的指针默认值是 nil （在Objective-C中向nil发送消息是安全的）

然后`obj\_destroyWeak`函数将0（nil）作为参数，调用`objc\_storeWeak`函数。

`objc\_storeWeak(&obj1, 0);`

前面的源代码与下列源代码相同。

```Objective-C

// 使用伪代码模拟：runtime如何实现weak属性

// http://weibo.com/luohanchenyilong/

// https://github.com/ChenYilong

id obj1;

obj1 = 0;

objc\_storeWeak(&obj1, obj);

/\* ... obj的引用计数变为0，被置nil ... \*/

objc\_storeWeak(&obj1, 0);

```

`objc\_storeWeak`函数把第二个参数--赋值对象（obj）的内存地址作为键值，将第一个参数--weak修饰的属性变量（obj1）的内存地址注册到 weak 表中。如果第二个参数（obj）为0（nil），那么把变量（obj1）的地址从weak表中删除。

### 27. 能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？

- 不能向编译后得到的类中增加实例变量；

- 能向运行时创建的类中添加实例变量；

解释下：

- 因为编译后的类已经注册在 runtime 中，类结构体中的 `objc\_ivar\_list` 实例变量的链表 和 `instance\_size` 实例变量的内存大小已经确定，同时runtime 会调用 `class\_setIvarLayout` 或 `class\_setWeakIvarLayout` 来处理 strong weak 引用。所以不能向存在的类中添加实例变量；

- 运行时创建的类是可以添加实例变量，调用 `class\_addIvar` 函数。但是得在调用 `objc\_allocateClassPair` 之后，`objc\_registerClassPair` 之前，原因同上。

### 28. runloop和线程有什么关系？

总的说来，Run loop，正如其名，loop表示某种循环，和run放在一起就表示一直在运行着的循环。实际上，run loop和线程是紧密相连的，可以这样说run loop是为了线程而生，没有线程，它就没有存在的必要。Run loops是线程的基础架构部分， Cocoa 和 CoreFundation 都提供了 run loop 对象方便配置和管理线程的 run loop （以下都以 Cocoa 为例）。每个线程，包括程序的主线程（ main thread ）都有与之相应的 run loop 对象。

runloop 和线程的关系：

1. 主线程的run loop默认是启动的。

iOS的应用程序里面，程序启动后会有一个如下的main()函数

```Objective-C

int main(int argc, char \* argv[]) {

@autoreleasepool {

return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));

}

}

```

重点是UIApplicationMain()函数，这个方法会为main thread设置一个NSRunLoop对象，这就解释了：为什么我们的应用可以在无人操作的时候休息，需要让它干活的时候又能立马响应。

2. 对其它线程来说，run loop默认是没有启动的，如果你需要更多的线程交互则可以手动配置和启动，如果线程只是去执行一个长时间的已确定的任务则不需要。

3. 在任何一个 Cocoa 程序的线程中，都可以通过以下代码来获取到当前线程的 run loop 。

```Objective-C

NSRunLoop \*runloop = [NSRunLoop currentRunLoop];

```

参考链接：[《Objective-C之run loop详解》](http://blog.csdn.net/wzzvictory/article/details/9237973)。

### 29. runloop的mode作用是什么？

model 主要是用来指定事件在运行循环中的优先级的，分为：

\* NSDefaultRunLoopMode（kCFRunLoopDefaultMode）：默认，空闲状态

\* UITrackingRunLoopMode：ScrollView滑动时

\* UIInitializationRunLoopMode：启动时

\* NSRunLoopCommonModes（kCFRunLoopCommonModes）：Mode集合

苹果公开提供的 Mode 有两个：

1. NSDefaultRunLoopMode（kCFRunLoopDefaultMode）

2. NSRunLoopCommonModes（kCFRunLoopCommonModes）

### 30. 以+ scheduledTimerWithTimeInterval...的方式触发的timer，在滑动页面上的列表时，timer会暂定回调，为什么？如何解决？

RunLoop只能运行在一种mode下，如果要换mode，当前的loop也需要停下重启成新的。利用这个机制，ScrollView滚动过程中NSDefaultRunLoopMode（kCFRunLoopDefaultMode）的mode会切换到UITrackingRunLoopMode来保证ScrollView的流畅滑动：只能在NSDefaultRunLoopMode模式下处理的事件会影响ScrollView的滑动。

如果我们把一个NSTimer对象以NSDefaultRunLoopMode（kCFRunLoopDefaultMode）添加到主运行循环中的时候,

ScrollView滚动过程中会因为mode的切换，而导致NSTimer将不再被调度。

同时因为mode还是可定制的，所以：

Timer计时会被scrollView的滑动影响的问题可以通过将timer添加到NSRunLoopCommonModes（kCFRunLoopCommonModes）来解决。代码如下：

```objective-c

//

// http://weibo.com/luohanchenyilong/ (微博@iOS程序犭袁)

// https://github.com/ChenYilong

//将timer添加到NSDefaultRunLoopMode中

[NSTimer scheduledTimerWithTimeInterval:1.0

target:self

selector:@selector(timerTick:)

userInfo:nil

repeats:YES];

//然后再添加到NSRunLoopCommonModes里

NSTimer \*timer = [NSTimer timerWithTimeInterval:1.0

target:self

selector:@selector(timerTick:)

userInfo:nil

repeats:YES];

[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];

```

### 31. 猜想runloop内部是如何实现的？

> 一般来讲，一个线程一次只能执行一个任务，执行完成后线程就会退出。如果我们需要一个机制，让线程能随时处理事件但并不退出，通常的代码逻辑

是这样的：

function loop() {

initialize();

do {

var message = get\_next\_message();

process\_message(message);

} while (message != quit);

}

或使用伪代码来展示下:

//

// http://weibo.com/luohanchenyilong/ (微博@iOS程序犭袁)

// https://github.com/ChenYilong

int main(int argc, char \* argv[]) {

//程序一直运行状态

while (AppIsRunning) {

//睡眠状态，等待唤醒事件

id whoWakesMe = SleepForWakingUp();

//得到唤醒事件

id event = GetEvent(whoWakesMe);

//开始处理事件

HandleEvent(event);

}

return 0;

}

参考链接：

1. [《深入理解RunLoop》](http://blog.ibireme.com/2015/05/18/runloop/#base)

2. 摘自博文[\*\*\*CFRunLoop\*\*\*](https://github.com/ming1016/study/wiki/CFRunLoop)，原作者是[微博@我就叫Sunny怎么了](http://weibo.com/u/1364395395)

### 32. objc使用什么机制管理对象内存？

通过 retainCount 的机制来决定对象是否需要释放。

每次 runloop 的时候，都会检查对象的 retainCount，如果retainCount 为 0，说明该对象没有地方需要继续使用了，可以释放掉了。

### 33. ARC通过什么方式帮助开发者管理内存？

<p><del>编译时根据代码上下文，插入 retain/release

</del></p>

ARC相对于MRC，不是在编译时添加retain/release/autorelease这么简单。应该是编译期和运行期两部分共同帮助开发者管理内存。

在编译期，ARC用的是更底层的C接口实现的retain/release/autorelease，这样做性能更好，也是为什么不能在ARC环境下手动retain/release/autorelease，同时对同一上下文的同一对象的成对retain/release操作进行优化（即忽略掉不必要的操作）；ARC也包含运行期组件，这个地方做的优化比较复杂，但也不能被忽略。【TODO:后续更新会详细描述下】

### 34. 不手动指定autoreleasepool的前提下，一个autorealese对象在什么时刻释放？（比如在一个vc的viewDidLoad中创建）

分两种情况：手动干预释放时机、系统自动去释放。

1. 手动干预释放时机--指定autoreleasepool

就是所谓的：当前作用域大括号结束时释放。

2. 系统自动去释放--不手动指定autoreleasepool

Autorelease对象出了作用域之后，会被添加到最近一次创建的自动释放池中，并会在当前的 runloop 迭代结束时释放。

释放的时机总结起来，可以用下图来表示：

![autoreleasepool与 runloop 的关系图](http://i61.tinypic.com/28kodwp.jpg)

下面对这张图进行详细的解释：

从程序启动到加载完成是一个完整的运行循环，然后会停下来，等待用户交互，用户的每一次交互都会启动一次运行循环，来处理用户所有的点击事件、触摸事件。

我们都知道：

\*\*所有 autorelease 的对象，在出了作用域之后，会被自动添加到最近创建的自动释放池中。\*\*

但是如果每次都放进应用程序的 `main.m` 中的 autoreleasepool 中，迟早有被撑满的一刻。这个过程中必定有一个释放的动作。何时？

在一次完整的运行循环结束之前，会被销毁。

那什么时间会创建自动释放池？运行循环检测到事件并启动后，就会创建自动释放池。

子线程的 runloop 默认是不工作，无法主动创建，必须手动创建。

自定义的 NSOperation 和 NSThread 需要手动创建自动释放池。比如： 自定义的 NSOperation 类中的 main 方法里就必须添加自动释放池。否则出了作用域后，自动释放对象会因为没有自动释放池去处理它，而造成内存泄露。

但对于 blockOperation 和 invocationOperation 这种默认的Operation ，系统已经帮我们封装好了，不需要手动创建自动释放池。

@autoreleasepool 当自动释放池被销毁或者耗尽时，会向自动释放池中的所有对象发送 release 消息，释放自动释放池中的所有对象。

如果在一个vc的viewDidLoad中创建一个 Autorelease对象，那么该对象会在 viewDidAppear 方法执行前就被销毁了。

参考链接：[《黑幕背后的Autorelease》](http://blog.sunnyxx.com/2014/10/15/behind-autorelease/)

### 35. BAD\_ACCESS在什么情况下出现？

访问了悬垂指针，比如对一个已经释放的对象执行了release、访问已经释放对象的成员变量或者发消息。

死循环

### 36. 苹果是如何实现autoreleasepool的？

autoreleasepool 以一个队列数组的形式实现,主要通过下列三个函数完成.

1. `objc\_autoreleasepoolPush`

2. `objc\_autoreleasepoolPop`

3. `objc\_autorelease`

看函数名就可以知道，对 autorelease 分别执行 push，和 pop 操作。销毁对象时执行release操作。

举例说明：我们都知道用类方法创建的对象都是 Autorelease 的，那么一旦 Person 出了作用域，当在 Person 的 dealloc 方法中打上断点，我们就可以看到这样的调用堆栈信息：

![enter image description here](http://i60.tinypic.com/15mfj11.jpg)

### 37. 使用block时什么情况会发生引用循环，如何解决？

一个对象中强引用了block，在block中又强引用了该对象，就会发射循环引用。

解决方法是将该对象使用\_\_weak或者\_\_block修饰符修饰之后再在block中使用。

1. id weak weakSelf = self;

或者 weak \_\_typeof(&\*self)weakSelf = self该方法可以设置宏

2. id \_\_block weakSelf = self;

或者将其中一方强制制空 `xxx = nil`。

检测代码中是否存在循环引用问题，可使用 Facebook 开源的一个检测工具 [\*\*\*FBRetainCycleDetector\*\*\*](https://github.com/facebook/FBRetainCycleDetector) 。

### 38. 在block内如何修改block外部变量？

默认情况下，在block中访问的外部变量是复制过去的，即：\*\*写操作不对原变量生效\*\*。但是你可以加上 `\_\_block` 来让其写操作生效，示例代码如下:

```Objective-C

\_\_block int a = 0;

void (^foo)(void) = ^{

a = 1;

};

foo();

//这里，a的值被修改为1

```

这是 [微博@唐巧\_boy](http://weibo.com/tangqiaoboy)的《iOS开发进阶》中的第11.2.3章节中的描述。你同样可以在面试中这样回答，但你并没有答到“点子上”。真正的原因，并没有书这本书里写的这么“神奇”，而且这种说法也有点牵强。面试官肯定会追问“为什么写操作就生效了？”真正的原因是这样的：

> 我们都知道：\*\*Block不允许修改外部变量的值\*\*，这里所说的外部变量的值，指的是栈中指针的内存地址。`\_\_block` 所起到的作用就是只要观察到该变量被 block 所持有，就将“外部变量”在栈中的内存地址放到了堆中。进而在block内部也可以修改外部变量的值。

\*\*Block不允许修改外部变量的值\*\*。Apple这样设计，应该是考虑到了block的特殊性，block也属于“函数”的范畴，变量进入block，实际就是已经改变了作用域。在几个作用域之间进行切换时，如果不加上这样的限制，变量的可维护性将大大降低。又比如我想在block内声明了一个与外部同名的变量，此时是允许呢还是不允许呢？只有加上了这样的限制，这样的情景才能实现。于是栈区变成了红灯区，堆区变成了绿灯区。

我们可以打印下内存地址来进行验证：

```Objective-C

\_\_block int a = 0;

NSLog(@"定义前：%p", &a); //栈区

void (^foo)(void) = ^{

a = 1;

NSLog(@"block内部：%p", &a); //堆区

};

NSLog(@"定义后：%p", &a); //堆区

foo();

```

```Objective-C

2016-05-17 02:03:33.559 LeanCloudChatKit-iOS[1505:713679] 定义前：0x16fda86f8

2016-05-17 02:03:33.559 LeanCloudChatKit-iOS[1505:713679] 定义后：0x155b22fc8

2016-05-17 02:03:33.559 LeanCloudChatKit-iOS[1505:713679] block内部： 0x155b22fc8

```

“定义后”和“block内部”两者的内存地址是一样的，我们都知道 block 内部的变量会被 copy 到堆区，“block内部”打印的是堆地址，因而也就可以知道，“定义后”打印的也是堆的地址。

那么如何证明“block内部”打印的是堆地址？

把三个16进制的内存地址转成10进制就是：

1. 定义后前：6171559672

2. block内部：5732708296

3. 定义后后：5732708296

中间相差438851376个字节，也就是 418.5M 的空间，因为堆地址要小于栈地址，又因为iOS中一个进程的栈区内存只有1M，Mac也只有8M，显然a已经是在堆区了。

这也证实了：a 在定义前是栈区，但只要进入了 block 区域，就变成了堆区。这才是 `\_\_block` 关键字的真正作用。

`\_\_block` 关键字修饰后，int类型也从4字节变成了32字节，这是 Foundation 框架 malloc 出来的。这也同样能证实上面的结论。（PS：居然比 NSObject alloc 出来的 16 字节要多一倍）。

理解到这是因为堆栈地址的变更，而非所谓的“写操作生效”，这一点至关重要，要不然你如何解释下面这个现象：

以下代码编译可以通过，并且在block中成功将a的从Tom修改为Jerry。

```Objective-C

NSMutableString \*a = [NSMutableString stringWithString:@"Tom"];

NSLog(@"\n 定以前：------------------------------------\n\

a指向的堆中地址：%p；a在栈中的指针地址：%p", a, &a); //a在栈区

void (^foo)(void) = ^{

a.string = @"Jerry";

NSLog(@"\n block内部：------------------------------------\n\

a指向的堆中地址：%p；a在栈中的指针地址：%p", a, &a); //a在栈区

a = [NSMutableString stringWithString:@"William"];

};

foo();

NSLog(@"\n 定以后：------------------------------------\n\

a指向的堆中地址：%p；a在栈中的指针地址：%p", a, &a); //a在栈区

```

![enter image description here](http://i66.tinypic.com/34euhcy.jpg)

这里的a已经由基本数据类型，变成了对象类型。block会对对象类型的指针进行copy，copy到堆中，但并不会改变该指针所指向的堆中的地址，所以在上面的示例代码中，block体内修改的实际是a指向的堆中的内容。

但如果我们尝试像上面图片中的65行那样做，结果会编译不通过，那是因为此时你在修改的就不是堆中的内容，而是栈中的内容。

上文已经说过：\*\*Block不允许修改外部变量的值\*\*，这里所说的外部变量的值，指的是栈中指针的内存地址。栈区是红灯区，堆区才是绿灯区。

### 39. 使用系统的某些block api（如UIView的block版本写动画时），是否也考虑引用循环问题？

系统的某些block api中，UIView的block版本写动画时不需要考虑，但也有一些api 需要考虑：

所谓“引用循环”是指双向的强引用，所以那些“单向的强引用”（block 强引用 self ）没有问题，比如这些：

```Objective-C

[UIView animateWithDuration:duration animations:^{ [self.superview layoutIfNeeded]; }];

```

```Objective-C

[[NSOperationQueue mainQueue] addOperationWithBlock:^{ self.someProperty = xyz; }];

```

```Objective-C

[[NSNotificationCenter defaultCenter] addObserverForName:@"someNotification"

object:nil

queue:[NSOperationQueue mainQueue]

usingBlock:^(NSNotification \* notification) {

self.someProperty = xyz; }];

```

这些情况不需要考虑“引用循环”。

但如果你使用一些参数中可能含有 ivar 的系统 api ，如 GCD 、NSNotificationCenter就要小心一点：比如GCD 内部如果引用了 self，而且 GCD 的其他参数是 ivar，则要考虑到循环引用：

```Objective-C

\_\_weak \_\_typeof\_\_(self) weakSelf = self;

dispatch\_group\_async(\_operationsGroup, \_operationsQueue, ^

{

\_\_typeof\_\_(self) strongSelf = weakSelf;

[strongSelf doSomething];

[strongSelf doSomethingElse];

} );

```

类似的：

```Objective-C

\_\_weak \_\_typeof\_\_(self) weakSelf = self;

\_observer = [[NSNotificationCenter defaultCenter] addObserverForName:@"testKey"

object:nil

queue:nil

usingBlock:^(NSNotification \*note) {

\_\_typeof\_\_(self) strongSelf = weakSelf;

[strongSelf dismissModalViewControllerAnimated:YES];

}];

```

self --> \_observer --> block --> self 显然这也是一个循环引用。

检测代码中是否存在循环引用问题，可使用 Facebook 开源的一个检测工具 [\*\*\*FBRetainCycleDetector\*\*\*](https://github.com/facebook/FBRetainCycleDetector) 。

### 40. GCD的队列（`dispatch\_queue\_t`）分哪两种类型？

1. 串行队列Serial Dispatch Queue

2. 并行队列Concurrent Dispatch Queue

### 41. 如何用GCD同步若干个异步调用？（如根据若干个url异步加载多张图片，然后在都下载完成后合成一张整图）

使用Dispatch Group追加block到Global Group Queue,这些block如果全部执行完毕，就会执行Main Dispatch Queue中的结束处理的block。

```Objective-C

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_group\_t group = dispatch\_group\_create();

dispatch\_group\_async(group, queue, ^{ /\*加载图片1 \*/ });

dispatch\_group\_async(group, queue, ^{ /\*加载图片2 \*/ });

dispatch\_group\_async(group, queue, ^{ /\*加载图片3 \*/ });

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

// 合并图片

});

```

### 42. `dispatch\_barrier\_async`的作用是什么？

在并行队列中，为了保持某些任务的顺序，需要等待一些任务完成后才能继续进行，使用 barrier 来等待之前任务完成，避免数据竞争等问题。

`dispatch\_barrier\_async` 函数会等待追加到Concurrent Dispatch Queue并行队列中的操作全部执行完之后，然后再执行 `dispatch\_barrier\_async` 函数追加的处理，等 `dispatch\_barrier\_async` 追加的处理执行结束之后，Concurrent Dispatch Queue才恢复之前的动作继续执行。

打个比方：比如你们公司周末跟团旅游，高速休息站上，司机说：大家都去上厕所，速战速决，上完厕所就上高速。超大的公共厕所，大家同时去，程序猿很快就结束了，但程序媛就可能会慢一些，即使你第一个回来，司机也不会出发，司机要等待所有人都回来后，才能出发。 `dispatch\_barrier\_async` 函数追加的内容就如同 “上完厕所就上高速”这个动作。

（注意：使用 `dispatch\_barrier\_async` ，该函数只能搭配自定义并行队列 `dispatch\_queue\_t` 使用。不能使用： `dispatch\_get\_global\_queue` ，否则 `dispatch\_barrier\_async` 的作用会和 `dispatch\_async` 的作用一模一样。 ）

### 43. 苹果为什么要废弃`dispatch\_get\_current\_queue`？

`dispatch\_get\_current\_queue`容易造成死锁

### 44. 以下代码运行结果如何？

- (void)viewDidLoad

{

[super viewDidLoad];

NSLog(@"1");

dispatch\_sync(dispatch\_get\_main\_queue(), ^{

NSLog(@"2");

});

NSLog(@"3");

}

只输出：1 。发生主线程锁死。

### 45. addObserver:forKeyPath:options:context:各个参数的作用分别是什么，observer中需要实现哪个方法才能获得KVO回调？

```Objective-C

// 添加键值观察

/\*

1 观察者，负责处理监听事件的对象

2 观察的属性

3 观察的选项

4 上下文

\*/

[self.person addObserver:self forKeyPath:@"name" options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld context:@"Person Name"];

```

observer中需要实现一下方法：

```Objective-C

// 所有的 kvo 监听到事件，都会调用此方法

/\*

1. 观察的属性

2. 观察的对象

3. change 属性变化字典（新／旧）

4. 上下文，与监听的时候传递的一致

\*/

- (void)observeValueForKeyPath:(NSString \*)keyPath ofObject:(id)object change:(NSDictionary \*)change context:(void \*)context;

```

### 46. 如何手动触发一个value的KVO

所谓的“手动触发”是区别于“自动触发”：

自动触发是指类似这种场景：在注册 KVO 之前设置一个初始值，注册之后，设置一个不一样的值，就可以触发了。

想知道如何手动触发，必须知道自动触发 KVO 的原理：

键值观察通知依赖于 NSObject 的两个方法: `willChangeValueForKey:` 和 `didChangevlueForKey:` 。在一个被观察属性发生改变之前， `willChangeValueForKey:` 一定会被调用，这就

会记录旧的值。而当改变发生后， `observeValueForKey:ofObject:change:context:` 会被调用，继而 `didChangeValueForKey:` 也会被调用。如果可以手动实现这些调用，就可以实现“手动触发”了。

那么“手动触发”的使用场景是什么？一般我们只在希望能控制“回调的调用时机”时才会这么做。

具体做法如下：

如果这个 `value` 是 表示时间的 `self.now` ，那么代码如下：最后两行代码缺一不可。

相关代码已放在仓库里。

```Objective-C

// .m文件

// Created by https://github.com/ChenYilong

// 微博@iOS程序犭袁(http://weibo.com/luohanchenyilong/).

// 手动触发 value 的KVO，最后两行代码缺一不可。

//@property (nonatomic, strong) NSDate \*now;

- (void)viewDidLoad {

[super viewDidLoad];

\_now = [NSDate date];

[self addObserver:self forKeyPath:@"now" options:NSKeyValueObservingOptionNew context:nil];

NSLog(@"1");

[self willChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。

NSLog(@"2");

[self didChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。

NSLog(@"4");

}

```

但是平时我们一般不会这么干，我们都是等系统去“自动触发”。“自动触发”的实现原理：

> 比如调用 `setNow:` 时，系统还会以某种方式在中间插入 `wilChangeValueForKey:` 、 `didChangeValueForKey:` 和 `observeValueForKeyPath:ofObject:change:context:` 的调用。

大家可能以为这是因为 `setNow:` 是合成方法，有时候我们也能看到有人这么写代码:

```Objective-C

- (void)setNow:(NSDate \*)aDate {

[self willChangeValueForKey:@"now"]; // 没有必要

\_now = aDate;

[self didChangeValueForKey:@"now"];// 没有必要

}

```

这完全没有必要，不要这么做，这样的话，KVO代码会被调用两次。KVO在调用存取方法之前总是调用 `willChangeValueForKey:` ，之后总是调用 `didChangeValueForkey:` 。怎么做到的呢?答案是通过 isa 混写（isa-swizzling）。下文《apple用什么方式实现对一个对象的KVO？》会有详述。

参考链接： [Manual Change Notification---Apple 官方文档](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOCompliance.html#//apple\_ref/doc/uid/20002178-SW3)

### 47. 若一个类有实例变量 `NSString \*\_foo` ，调用setValue:forKey:时，可以以foo还是 `\_foo` 作为key？

都可以。

### 48. KVC的keyPath中的集合运算符如何使用？

1. 必须用在集合对象上或普通对象的集合属性上

2. 简单集合运算符有@avg， @count ， @max ， @min ，@sum，

3. 格式 @"@sum.age"或 @"集合属性.@max.age"

### 49. KVC和KVO的keyPath一定是属性么？

KVC 支持实例变量，KVO 只能手动支持[手动设定实例变量的KVO实现监听](https://yq.aliyun.com/articles/30483)

### 50. 如何关闭默认的KVO的默认实现，并进入自定义的KVO实现？

请参考：

1. [《如何自己动手实现 KVO》](http://tech.glowing.com/cn/implement-kvo/)

2. [\*\*KVO for manually implemented properties\*\*]( http://stackoverflow.com/a/10042641/3395008 )

### 51. apple用什么方式实现对一个对象的KVO？

[Apple 的文档](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOImplementation.html)对 KVO 实现的描述：

> Automatic key-value observing is implemented using a technique called isa-swizzling... 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 ...

从[Apple 的文档](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOImplementation.html)可以看出：Apple 并不希望过多暴露 KVO 的实现细节。不过，要是借助 runtime 提供的方法去深入挖掘，所有被掩盖的细节都会原形毕露：

> 当你观察一个对象时，一个新的类会被动态创建。这个类继承自该对象的原本的类，并重写了被观察属性的 setter 方法。重写的 setter 方法会负责在调用原 setter 方法之前和之后，通知所有观察对象：值的更改。最后通过 ` isa 混写（isa-swizzling）` 把这个对象的 isa 指针 ( isa 指针告诉 Runtime 系统这个对象的类是什么 ) 指向这个新创建的子类，对象就神奇的变成了新创建的子类的实例。我画了一张示意图，如下所示：

![enter image description here](http://i62.tinypic.com/sy57ur.jpg)

KVO 确实有点黑魔法：

> Apple 使用了 ` isa 混写（isa-swizzling）`来实现 KVO 。

下面做下详细解释：

键值观察通知依赖于 NSObject 的两个方法: `willChangeValueForKey:` 和 `didChangevlueForKey:` 。在一个被观察属性发生改变之前， `willChangeValueForKey:` 一定会被调用，这就会记录旧的值。而当改变发生后， `observeValueForKey:ofObject:change:context:` 会被调用，继而 `didChangeValueForKey:` 也会被调用。可以手动实现这些调用，但很少有人这么做。一般我们只在希望能控制回调的调用时机时才会这么做。大部分情况下，改变通知会自动调用。

比如调用 `setNow:` 时，系统还会以某种方式在中间插入 `wilChangeValueForKey:` 、 `didChangeValueForKey:` 和 `observeValueForKeyPath:ofObject:change:context:` 的调用。大家可能以为这是因为 `setNow:` 是合成方法，有时候我们也能看到有人这么写代码:

```Objective-C

- (void)setNow:(NSDate \*)aDate {

[self willChangeValueForKey:@"now"]; // 没有必要

\_now = aDate;

[self didChangeValueForKey:@"now"];// 没有必要

}

```

这完全没有必要，不要这么做，这样的话，KVO代码会被调用两次。KVO在调用存取方法之前总是调用 `willChangeValueForKey:` ，之后总是调用 `didChangeValueForkey:` 。怎么做到的呢?答案是通过 isa 混写（isa-swizzling）。第一次对一个对象调用 `addObserver:forKeyPath:options:context:` 时，框架会创建这个类的新的 KVO 子类，并将被观察对象转换为新子类的对象。在这个 KVO 特殊子类中， Cocoa 创建观察属性的 setter ，大致工作原理如下:

```Objective-C

- (void)setNow:(NSDate \*)aDate {

[self willChangeValueForKey:@"now"];

[super setValue:aDate forKey:@"now"];

[self didChangeValueForKey:@"now"];

}

```

这种继承和方法注入是在运行时而不是编译时实现的。这就是正确命名如此重要的原因。只有在使用KVC命名约定时，KVO才能做到这一点。

KVO 在实现中通过 ` isa 混写（isa-swizzling）` 把这个对象的 isa 指针 ( isa 指针告诉 Runtime 系统这个对象的类是什么 ) 指向这个新创建的子类，对象就神奇的变成了新创建的子类的实例。这在[Apple 的文档](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOImplementation.html)可以得到印证：

> Automatic key-value observing is implemented using a technique called isa-swizzling... 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 ...

然而 KVO 在实现中使用了 ` isa 混写（ isa-swizzling）` ，这个的确不是很容易发现：Apple 还重写、覆盖了 `-class` 方法并返回原来的类。 企图欺骗我们：这个类没有变，就是原本那个类。。。

但是，假设“被监听的对象”的类对象是 `MYClass` ，有时候我们能看到对 `NSKVONotifying\_MYClass` 的引用而不是对 `MYClass` 的引用。借此我们得以知道 Apple 使用了 ` isa 混写（isa-swizzling）`。具体探究过程可参考[ 这篇博文 ](https://www.mikeash.com/pyblog/friday-qa-2009-01-23.html)。

那么 `wilChangeValueForKey:` 、 `didChangeValueForKey:` 和 `observeValueForKeyPath:ofObject:change:context:` 这三个方法的执行顺序是怎样的呢？

`wilChangeValueForKey:` 、 `didChangeValueForKey:` 很好理解，`observeValueForKeyPath:ofObject:change:context:` 的执行时机是什么时候呢？

先看一个例子：

代码已放在仓库里。

```Objective-C

- (void)viewDidLoad {

[super viewDidLoad];

[self addObserver:self forKeyPath:@"now" options:NSKeyValueObservingOptionNew context:nil];

NSLog(@"1");

[self willChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。

NSLog(@"2");

[self didChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。

NSLog(@"4");

}

- (void)observeValueForKeyPath:(NSString \*)keyPath ofObject:(id)object change:(NSDictionary<NSString \*,id> \*)change context:(void \*)context {

NSLog(@"3");

}

```

![enter image description here](http://i66.tinypic.com/ncm7th.jpg)

如果单单从下面这个例子的打印上，

顺序似乎是 `wilChangeValueForKey:` 、 `observeValueForKeyPath:ofObject:change:context:` 、 `didChangeValueForKey:` 。

其实不然，这里有一个 `observeValueForKeyPath:ofObject:change:context:` , 和 `didChangeValueForKey:` 到底谁先调用的问题：如果 `observeValueForKeyPath:ofObject:change:context:` 是在 `didChangeValueForKey:` 内部触发的操作呢？ 那么顺序就是： `wilChangeValueForKey:` 、 `didChangeValueForKey:` 和 `observeValueForKeyPath:ofObject:change:context:`

不信你把 `didChangeValueForKey:` 注视掉，看下 `observeValueForKeyPath:ofObject:change:context:` 会不会执行。

了解到这一点很重要，正如 [46. 如何手动触发一个value的KVO](https://github.com/ChenYilong/iOSInterviewQuestions/blob/master/01《招聘一个靠谱的iOS》面试题参考答案/《招聘一个靠谱的iOS》面试题参考答案（下）.md#46-如何手动触发一个value的kvo) 所说的：

“手动触发”的使用场景是什么？一般我们只在希望能控制“回调的调用时机”时才会这么做。

而“回调的调用时机”就是在你调用 `didChangeValueForKey:` 方法时。

### 52. IBOutlet连出来的视图属性为什么可以被设置成weak?

参考链接：[ \*\*\*Should IBOutlets be strong or weak under ARC?\*\*\* ](http://stackoverflow.com/questions/7678469/should-iboutlets-be-strong-or-weak-under-arc)

文章告诉我们：

> 因为既然有外链那么视图在xib或者storyboard中肯定存在，视图已经对它有一个强引用了。

不过这个回答漏了个重要知识，使用storyboard（xib不行）创建的vc，会有一个叫\_topLevelObjectsToKeepAliveFromStoryboard的私有数组强引用所有top level的对象，所以这时即便outlet声明成weak也没关系

### 53. IB中User Defined Runtime Attributes如何使用？

它能够通过KVC的方式配置一些你在interface builder 中不能配置的属性。当你希望在IB中作尽可能多得事情，这个特性能够帮助你编写更加轻量级的viewcontroller

### 54. 如何调试BAD\_ACCESS错误

1. 重写object的respondsToSelector方法，现实出现EXEC\_BAD\_ACCESS前访问的最后一个object

2. 通过 Zombie

![enter image description here](http://i.stack.imgur.com/ZAdi0.png)

3. 设置全局断点快速定位问题代码所在行

4. Xcode 7 已经集成了BAD\_ACCESS捕获功能：\*\*Address Sanitizer\*\*。

用法如下：在配置中勾选✅Enable Address Sanitizer

![enter image description here](https://developer.apple.com/library/prerelease/ios/documentation/DeveloperTools/Conceptual/WhatsNewXcode/Art/xc7-asan\_2x.png)

### 55. lldb（gdb）常用的调试命令？

- breakpoint 设置断点定位到某一个函数

- n 断点指针下一步

- po打印对象

更多 lldb（gdb） 调试命令可查看

1. [ \*\*\*The LLDB Debugger\*\*\* ](http://lldb.llvm.org/lldb-gdb.html)；

2. 苹果官方文档：[ \*\*\*iOS Debugging Magic\*\*\* ](https://developer.apple.com/library/ios/technotes/tn2239/\_index.html)。

----------

Posted by [微博@iOS程序犭袁](http://weibo.com/luohanchenyilong/)

原创文章，版权声明：自由转载-非商用-非衍生-保持署名 | [Creative Commons BY-NC-ND 3.0](http://creativecommons.org/licenses/by-nc-nd/3.0/deed.zh)