

1. 风格纠错题

```
typedef enum{
    UserSex_Man,
    UserSex_Woman
}UserSex;

@interface UserModel :NSObject

@property(nonatomic, strong) NSString *name;
@property(assign, nonatomic) int age;
@property(nonatomic, assign) UserSex sex;

-(id)initUserModelWithUserName: (NSString*)name withAge:(int)age;

-(void)doLogIn;

@end
```

修改完的代码：

修改方法有很多种，现给出一种做示例：

```
// .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(NSUInteger, 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](#) 一文：

```
//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](#) 。)

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：

```
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》](#) 。

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：

~~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](#)）~~

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` 这种多余的关键词，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](#) ）

- ```
//错误，不要使用"and"来连接参数
- (int)runModalForDirectory:(NSString *)path andFile:(NSString *)name andT
//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
```

- Objective-C 有 designated 和 secondary 初始化方法的概念。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：

<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... 5/49

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：

```
// .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(NSUInteger, 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:(
- (instancetype)initWithName:(NSString *)name age:(NSUInteger)age;
+ (instancetype)userWithName:(NSString *)name age:(NSUInteger)age 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）》](#)。）

- 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

```
@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. `initWithUserName` 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

```
@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](#)》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
2. 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
5. `UserModel :NSObject` 应为 `UserModel : NSObject`，也就是：右侧少了一个空格。
6. `@interface` 与 `@property` 属性声明中间应当间隔一行。
7. 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。
8. `-(id)initWithUserName:(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。
9. `-(id)initWithUserName:(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)initWithUserName:(NSString*)name withAge:(int)age;` 方法中 `(NSString*)name` 应为 `(NSString *)name`，少了空格。
11. `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](#))

## 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 NSInteger 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](#):

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?

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

- (instancetype)initWithUserId:(NSString *)userId {
 self = [super init];
 if (!self) {
```



```
 return nil;
 }
 _userId = [userId copy];
 return self;
}
```



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

```
// .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;
```

```
// .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：

```
-[__NSArrayI removeObjectAtIndex:]: unrecognized selector sent to instance
```

第2条原因，如下：

该属性使用了自旋锁，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))。

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:

```
- (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：

```
// .h文件
// http://weibo.com/luohanchenyilong/
// https://github.com/ChenYilong
// 修改完的代码

typedef 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:

```
- (id)copyWithZone:(NSZone *)zone {
 CYLUser *copy = [[self class] allocWithZone:zone]
 initWithName:_name
 age:_age
 sex:_sex];
```







## 李克强痛斥某些办事机构：办个事儿咋就这么难？ - 中国政府网

[www.gov.cn](http://www.gov.cn) › 新闻 › 滚动 ▾

2015年5月6日 - 他费解地发问：老百姓办个事儿咋就这么难？政府给老百姓办事为啥要设这么多道“障碍”？“你妈是你妈”，这怎么证明呢？简直是笑话！“我看到有家 ...

## 逾六成受访者：证明“你妈是你妈”是找碴--时政--人民网

[politics.people.com.cn](http://politics.people.com.cn) › 时政 ▾

2015年5月16日 - 近日**李克强**总理两度开腔痛斥“奇葩证明”：继5月6日在国务院常务会议 ... 像“你妈是你妈”的证明，“京报调查”结果就显示逾六成人认为这是找茬，这 ...

微博@iOS程序猿袁

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：

```
[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

```
[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](#)）

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：

```
-(void)setSpeed:(int)_speed{
 if(_speed < 0) speed = 0;
```



```
if(_speed > 300) speed = 300;
_speed = speed;
}
```

```
@property = getter + setter;
```

例如下面这个类：

```
@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：

```
@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:

```
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:

```
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：

```
/// Defines a property attribute
typedef struct {
 const char *name; /**< The name of the attribute */
 const char *value; /**< The value of the attribute (usually e
} 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](#)，C就代表Copy，N代表nonatomic，V就代表对应的实例变量。

**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。

```
@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
3. `ivar_list`：成员变量列表
4. `method_list`：方法列表
5. `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







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。

```
obj1 = 0;
objc_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 `objc_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。

```
// 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](#)）

```
- (void)setObject:(NSObject *)object
{
 objc_setAssociatedObject(self, "object", object, OBJC_ASSOCIATION_ASSIGN);
 [object cyl_runAtDealloc:^(
 _object = nil;
)];
}
```



```
// 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

#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
```



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:

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

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

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：

```
@property(n nonatomic, strong, getter=p_initBy, setter=setP_initBy:)NSString
```

另外也可以用关键字进行特殊说明，来避免编译器报错：

```
@property(n 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：

```
static inline void reallySetProperty(id self, SEL _cmd, id newValue, ptrdi
{
 if (offset == 0) {
 object_setClass(self, newValue);
 return;
 }
}
```





```
 _object = nil;
 }];
}
```





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：

```
NSMutableArray *array = [NSMutableArray arrayWithObjects:[NSMutableString
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：

```
[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](#)

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](#) 所说：

Mac Developer Library

Developer

PDF

Programming with Objective-C

Table of Contents

Introduction

Defining Classes

Working with Objects

Encapsulating Data

Properties Encapsulate an Object's Values

Declare Public Properties for Exposed Data

Use Accessor Methods to Get or Set Property Values

Dot Syntax Is a Concise Alternative to Accessor Method Calls

Most Properties Are Backed by Instance Variables

Access Instance Variables Directly from\_INITIALIZER Methods

You Can Implement Custom Accessor Methods

Properties Are Atomic by Default

Manage the Object Graph through Ownership and Responsibility

Exercises

You Can Customize Synthesized Instance Variable Names

As mentioned earlier, the default behavior for a writeable property is to use an instance variable called `_propertyName`.

If you wish to use a different name for the instance variable, you need to direct the compiler to synthesize the variable using the following syntax in your implementation:

```
@implementation YourClass
@synthesize propertyName = instanceVariableName;
...
@end
```

For example:

```
@synthesize firstName = ivar_firstName;
```

In this case, the property will still be called `firstName`, and be accessible through `firstName` and `setFirstName`: accessor methods or dot syntax, but it will be backed by an instance variable called `ivar_firstName`.

Important: If you use `@synthesize` without specifying an instance variable name, like this:

```
@synthesize firstName;
```

the instance variable will bear the same name as the property.

In this example, the instance variable will also be called `firstName`, without an underscore.

微博@iOS程序猿袁

微博@iOS程序猿袁

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。

```
@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：

```
@implementation CYLPerson
@synthesize firstName = _myFirstName;
@synthesize lastName = _myLastName;
@end
```





```

 }
 return self;
}

- (NSString *)title {
 return _title;
}

- (void)setTitle:(NSString *)title {
 _title = [title copy];
}

@end

```

结果编译器报错：

```

1 //
2 // .m文件
3 // http://weibo.com/luohanchenyilong/（微博@iOS程序猿袁）
4 // https://github.com/ChenYilong
5 // 打开第14行和第17行中任意一行，就可编译成功
6
7 @import Foundation;
8
9 @interface CYLObject : NSObject
10 @property (nonatomic, copy) NSString *title;
11 @end
12
13 @implementation CYLObject {
14 NSString *_title;
15 }
16
17 //@synthesize title = _title;
18
19 - (instancetype)init
20 {
21 self = [super init];
22 if (self) {
23 _title = @"微博@iOS程序猿袁";
24 }
25 return self;
26 }
27
28 - (NSString *)title {
29 return _title;
30 }
31
32 - (void)setTitle:(NSString *)title {
33 _title = [title copy];
34 }
35
36 @end

```

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?](#)

















}
@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：

```
//
// ChenPerson.m
//
// Created by https://github.com/ChenYilong on 15/8/30.
// Copyright (c) 2015年 http://weibo.com/luohanchenyilong/ 微博@iOS程序猿袁
//

#import "ChenPerson.h"

@implementation ChenPerson

@synthesize lastName = _lastName;

- (instancetype)init
{
 self = [super init];
}
```





第一个参数是 `objc_super` 这样一个结构体，其定义如下：

```
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：

```
- (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(object_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](#)的博文[创根问底Objective-C Runtime（1）- Self & 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。
- 在MRC中，对于使用retain或copy策略的需要。





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()`

## 24. objc中的类方法和实例方法有什么本质区别和联系?

1. 类方法是属于类对象的
2. 类方法只能通过类对象调用
3. 类方法中的self是类对象
4. 类方法可以调用其他的类方法
5. 类方法中不能访问成员变量
6. 类方法中不能直接调用对象方法

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
3. 实例方法中的self是实例对象
4. 实例方法中可以访问成员变量
5. 实例方法中直接调用实例方法
6. 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](#)，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](#)

@property部分主要参考 [Apple官方文档：Properties Encapsulate an Object's Values](#)  
runtime部分主要参考[Apple官方文档：Declared Properties](#)

## 26. runtime如何实现weak变量的自动置nil?

## 28. runloop和线程有什么关系?

## 29. runloop的mode作用是什么?

### 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?

