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：

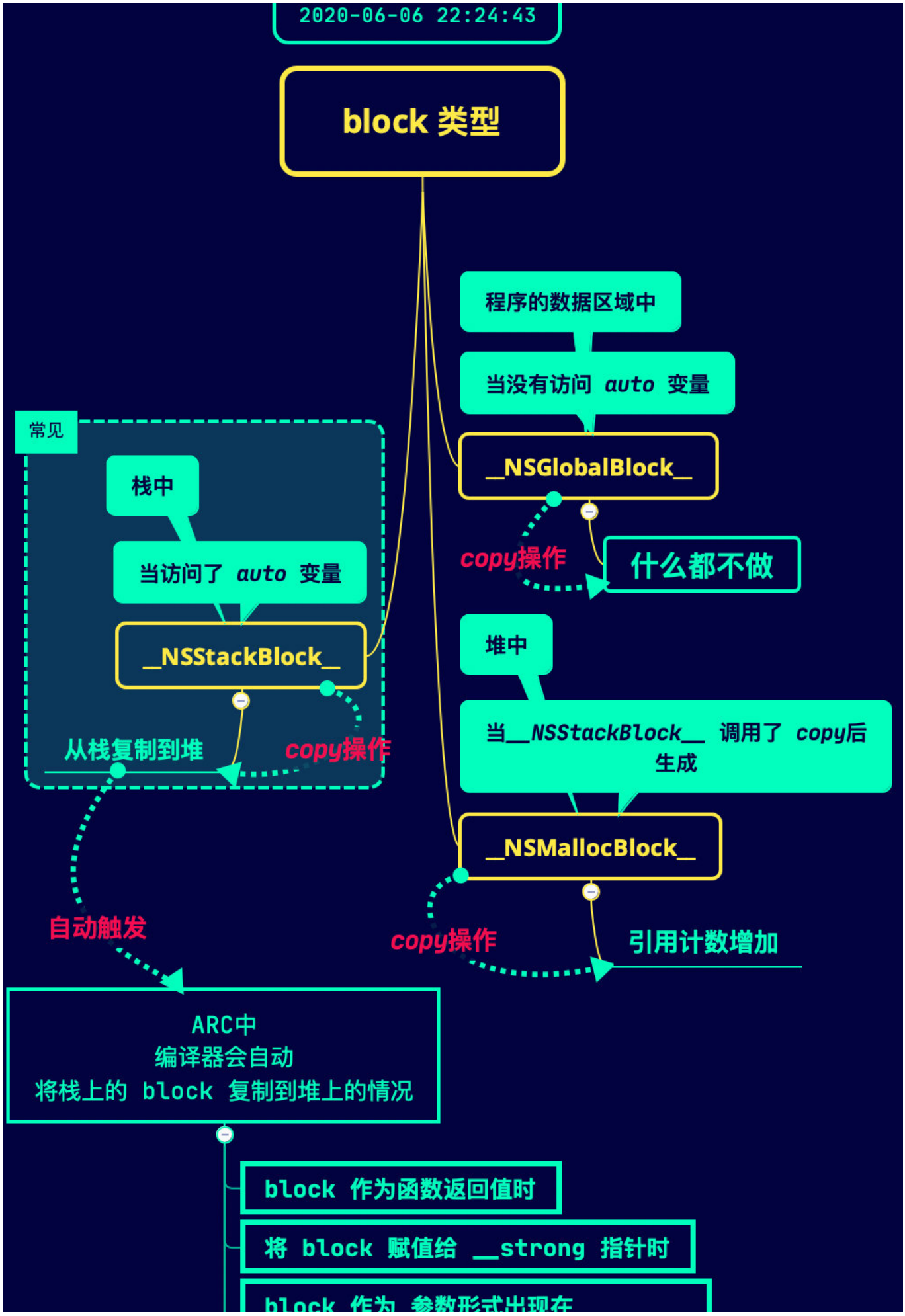
1. NSString、NSArray、NSDictionary 等等经常使用copy关键字，是因为他们有对应的可变类型：NSMutableString、NSMutableArray、NSMutableDictionary；
2. block 也经常使用 copy 关键字，具体原因见官方文档：***Objects Use Properties to Keep Track of Blocks***：

   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：

   在 ARC 环境下，编译器会根据情况自动将栈上的 block 复制到堆上，比如以下情况：

- block 作为函数返回值时
- 将 block 赋值给 __strong 指针时（property 的 copy 属性对应的是这一条）
- block 作为 Cocoa API 中方法名含有 using Block 的方法参数时
- block 作为 GCD API 的方法参数时


2020-06-06 22:24:43
block 类型
程序的数据区域中
当没有访问 auto 变量
NSGlobalBlock
copy操作
什么都不做
常见
栈中
当访问了 auto 变量
NSStackBlock
从栈复制到堆
copy操作
堆中
当__NSStackBlock__ 调用了 copy后生成
NSMallocBlock
自动触发
copy操作
引用计数增加
ARC中
编译器会自动
将栈上的 block 复制到堆上的情况
block 作为函数返回值时
将 block 赋值给 __strong 指针时
block 作为 参数形式出现在

其中， block 的 property 设置为 copy， 对应的是这一条：将 block 赋值给 __strong 指针时。

换句话说：

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，不需要写还依然写。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

```
// .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 0x7fcd1b
```

第2条原因，如下：

> 该属性使用了互斥锁（atomic 的底层实现，老版本是自旋锁，iOS10开始是互斥锁--spinlock底层实现改变了。），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，则不使用互斥锁（atomic 的底层实现，老版本是自旋锁，iOS10开始是互斥锁--spinlock底层实现改变了。）。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 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)se
+ (instancetype)userWithName:(NSString *)name age:(NSUInteger)age sex:(CYLSex)se

@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];
    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:

```
// .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

```
// .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:

```
- (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：

```
- (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？”

```
- (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：

克强总理这样评价你的代码风格：

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;
}
```

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 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：

```
- (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：

```
@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 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，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
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：

```
/**
 * 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 。

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 发送消息会产生崩溃。

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”：

```
// 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。

```
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。

```
// 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;
    }];
}
```

也就是有两个步骤：

1. 在setter方法中做如下设置：

    ```
    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：

```
//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

```
// .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

```
// .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

```
// 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

```
    #import "CYLNSObject+RunAtDealloc.h"
```

然后就可以使用了：

```
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

参考博文： ***Fun With the Objective-C Runtime: Run Code at Deallocation of Any Object***

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，则不使用互斥锁（atomic 的底层实现，老版本是自旋锁，iOS10开始是互斥锁--spinlock底层实现改变了。）。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：

   ```
       @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(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：

```
@property(nonatomic, readwrite, copy, null_resettable) NSString *initBy;
- (NSString *)initBy __attribute__((objc_method_family(none)));
```

1. 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互斥锁（atomic 的底层实现，老版本是自旋锁，iOS10开始是互斥锁--spinlock底层实现改变了。），互斥锁（atomic 的底层实现，老版本是自旋锁，iOS10开始是互斥锁--spinlock底层实现改变了。）相关代码如下：

```
static inline void reallySetProperty(id self, SEL _cmd, id newValue, ptrdiff_t off
set, 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 atomi
c, 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：

```
- (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

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***
2. ***Variable property attributes or Modifiers in iOS***

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：

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

然后进行下面的操作：

```
    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：

```
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：

```
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：

```
[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：

```
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：

```
NSMutableArray *array = [NSMutableArray arrayWithObjects:[NSMutableString stringWi
thString:@"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：

```
[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：

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
```

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. 3 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,

4 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：

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
3. 使用了 @dynamic 时
4. 在 @protocol 中定义的所有属性

5. 在 category 中定义的所有属性
6. 重写（overridden）的属性

    当你在子类中重写（overridden）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:

```
@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：

```
//
// .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：

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?***

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:

```
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。

3、 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。

4、 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:

```
// 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Obje
  #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
  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，

```
//
//  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

```
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）

我们可以看到大概是这样的：

```
((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,
  3. 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：

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:

20. 一个objc对象的isa的指针指向什么？有什么作用?

`isa` 顾名思义 `is a` 表示对象所属的类。

`isa` 指向他的类对象，从而可以找到对象上的方法。

同一个类的不同对象，他们的 isa 指针是一样的。

21. 下面的代码输出什么?

```
    @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：

```
//
//  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 rig
hts 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__,
        NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", __PRETTY_FUNCTION__, __LINE__,
    }
    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

}

@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：

```
//
//  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 rig
hts 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_
        //NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", __PRETTY_FUNCTION__, __LINE_
    }
    return self;
}

- (void)setLastName:(NSString*)lastName
{
    NSLog(@"🔴类名与方法名：%s（在第%d行），描述：%@", __PRETTY_FUNCTION__, __LINE__, @"根
    _lastName = @"炎黄";
}

@end
```

那么打印结果将会是这样的：

```
🔴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：

```
    $ clang -rewrite-objc test.m
```

将这道题目中给出的代码被转化为:

```
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_gm_0jk35cwn1d3326x0061qym

    NSLog((NSString *)&__NSConstantStringImpl__var_folders_gm_0jk35cwn1d3326x0061qym
```

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:

```
    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:

```
    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:

```
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(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的博文刨根问底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。

参考 NSObject 上面的方法：

```
- (IMP)methodForSelector:(SEL)aSelector;
+ (IMP)instanceMethodForSelector:(SEL)aSelector;
```

参考： Apple Documentation-Objective-C Runtime-NSObject-methodForSelector:

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。~~

  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 一节中有一个MRC环境下的例子：

```
// 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)***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。

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
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

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
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@iOS程序犭袁

@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

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?