

索引

1. [风格纠错题](#)
 1. [优化部分](#)
 2. [硬伤部分](#)
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？](#)
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？](#)
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](#)
7. [@protocol 和 category 中如何使用 @property](#)
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？](#)
10. [weak 属性需要在 dealloc 中置 nil 么？](#)
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？](#)
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. \[对非集合类对象的 copy 操作\]\(#\)
 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？](#)
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？](#)

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？
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？
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
```

22. 22--55题，请看下篇。

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(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

- 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](#)。)

- 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：

<code>int -> NSInteger</code>
<code>unsigned -> NSUInteger</code>
<code>float -> CGFloat</code>
<code>动画时间 -> NSTimeInterval</code>

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](#)）

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：

```
//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：

```
// .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：

```

// .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）》](#)。）

- 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. `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

```
@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改动的修饰符。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)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。
9. `-(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。
11. ~~deLogin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，`Log In` 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. 自己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 IBOulet 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;
}
```

iOS Developer Library

Developer

Programming with Objective-C

Table of Contents

- Introduction
- Defining Classes
- Working with Objects
- Encapsulating Data
- Customizing Existing Classes
- Working with Protocols
- Values and Collections
- Working with Blocks
 - Block Syntax
 - Blocks Take Arguments and Return Values
 - Blocks Can Capture Values from the Enclosing Scope
 - You Can Pass Blocks as Arguments to Methods or Functions
 - Use Type Definitions to Simplify Block Syntax
 - Objects Use Properties to Keep Track of Blocks

Objects Use Properties to Keep Track of Blocks

The syntax to define a property to keep track of a block is similar to a block variable:

```
@interface XYZObject : NSObject
@property (copy) void (^blockProperty)(void);
@end
```

Note: You should specify `copy` as the property attribute, because a block needs to be copied to keep track of its captured state outside of the original scope. This isn't something you need to worry about when using Automatic Reference Counting, as it will happen automatically, but it's best practice for the property attribute to show the resultant behavior. For more information, see [Blocks Programming Topics](#).

微博@iOS程序猿袁

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关于《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 0x7fc1bc
30460
```

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：

```
- (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)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];
    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
//

```



```
}
```

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：



微博@iOS程序猿 袁

克强总理这样评价你的代码风格：

[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 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”是找茬--时政--人民网](#)

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上 ... 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;  
}
```

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：

```
- (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。

```
@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”：

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

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：

```
@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自旋锁，自旋锁相关代码如下：

```

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：

```
- (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 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：

```
[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：



▼ 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

微博@iOS程序猿袁

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.

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:

```
8  
9 #import <Foundation/Foundation.h>  
10  
11 @interface TestClass : NSObject  
12 {  
13     NSObject *_object;  
14 }  
15  
16 @property (nonatomic, retain) NSObject *object;  
17 @property (nonatomic, retain) NSObject *_object;  
18  
19 @end  
20
```

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：

```
@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，但是如果我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看得懂。

举例说明：应用场景：

```
//  
// .m文件  
// http://weibo.com/luohanchenyilong/ (微博@iOS程序猿袁)  
// 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程序猿袁";  
    }  
    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";           ! Use of undeclared identifier '_title'
24     }
25     return self;
26 }
27
28 - (NSString *)title {
29     return _title;                           ! Use of undeclared identifier '_title'
30 }
31
32 - (void)setTitle:(NSString *)title {
33     _title = [title copy];                  ! Use of undeclared identifier '_title'; did you mean 'title'?
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?](#)

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，

```
//  
//  main.m  
//  http://weibo.com/luohanchenyilong/  
//  https://github.com/ChenYilong  
//  Copyright (c) 2015年 微博@ios程序猿袁. 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）

```
微博@iOS程序员袁  
04192  
04193 #ifndef _REWRITER_TYPEDEF_CYLTest  
04194 #define _REWRITER_TYPEDEF_CYLTest  
04195     typedef struct objc_object CYLTest;  
04196     typedef struct {} _objc_exc_CYLTest;  
04197 #endif  
04198  
04199 struct CYLTest_IMPL {  
04200     struct NSObject_IMPL NSObject_IVARS;  
04201 };  
04202  
04203  
04204 /* @end */  
04205  
04206  
04207 int main(int argc, char * argv[]) {  
04208     /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;  
04209         CYLTest *test = ((CYLTest *)(*(id, SEL))(void *)objc_msgSend)((id)((CYLTest *)(*(id, SEL))  
04210             (void *)objc_msgSend)((id)objc_getClass("CYLTest"), sel_registerName("alloc")),  
04211             sel_registerName("init"));  
04212             ((id (*)(id, SEL, SEL))(void *)objc_msgSend)((id)test, sel_registerName("performSelector:"  
04213                 ), (sel_registerName("iOSinit")));  
04214             return 0;  
04215     }  
04216     static struct IMAGE_INFO { unsigned version; unsigned flag; } _OBJC_IMAGE_INFO = { 0, 2 };  
04217 }
```

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。

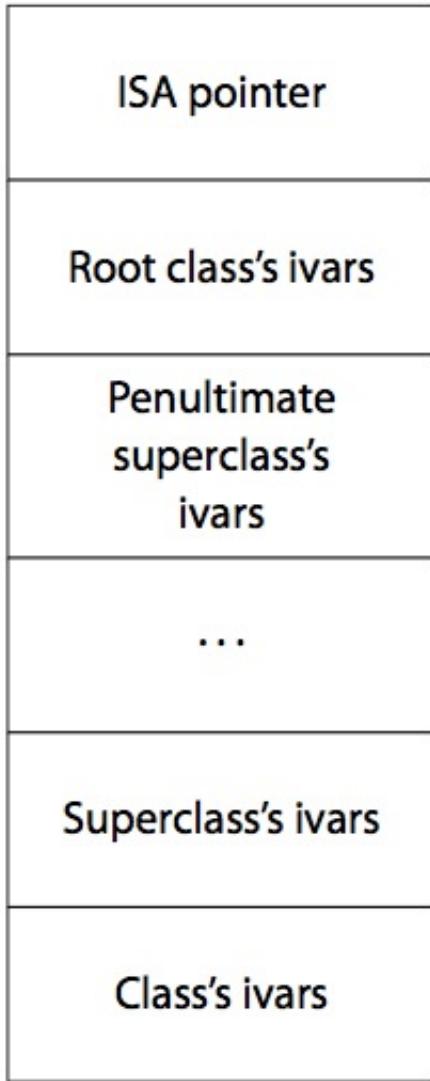
19. 一个objc对象如何进行内存布局？（考虑有父类的情况）

- 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

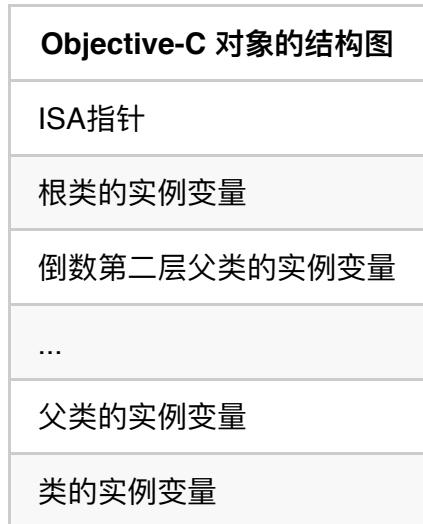
1. 对象方法列表（对象能够接收的消息列表，保存在它所对应的类对象中）
2. 成员变量的列表，
3. 属性列表，

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：



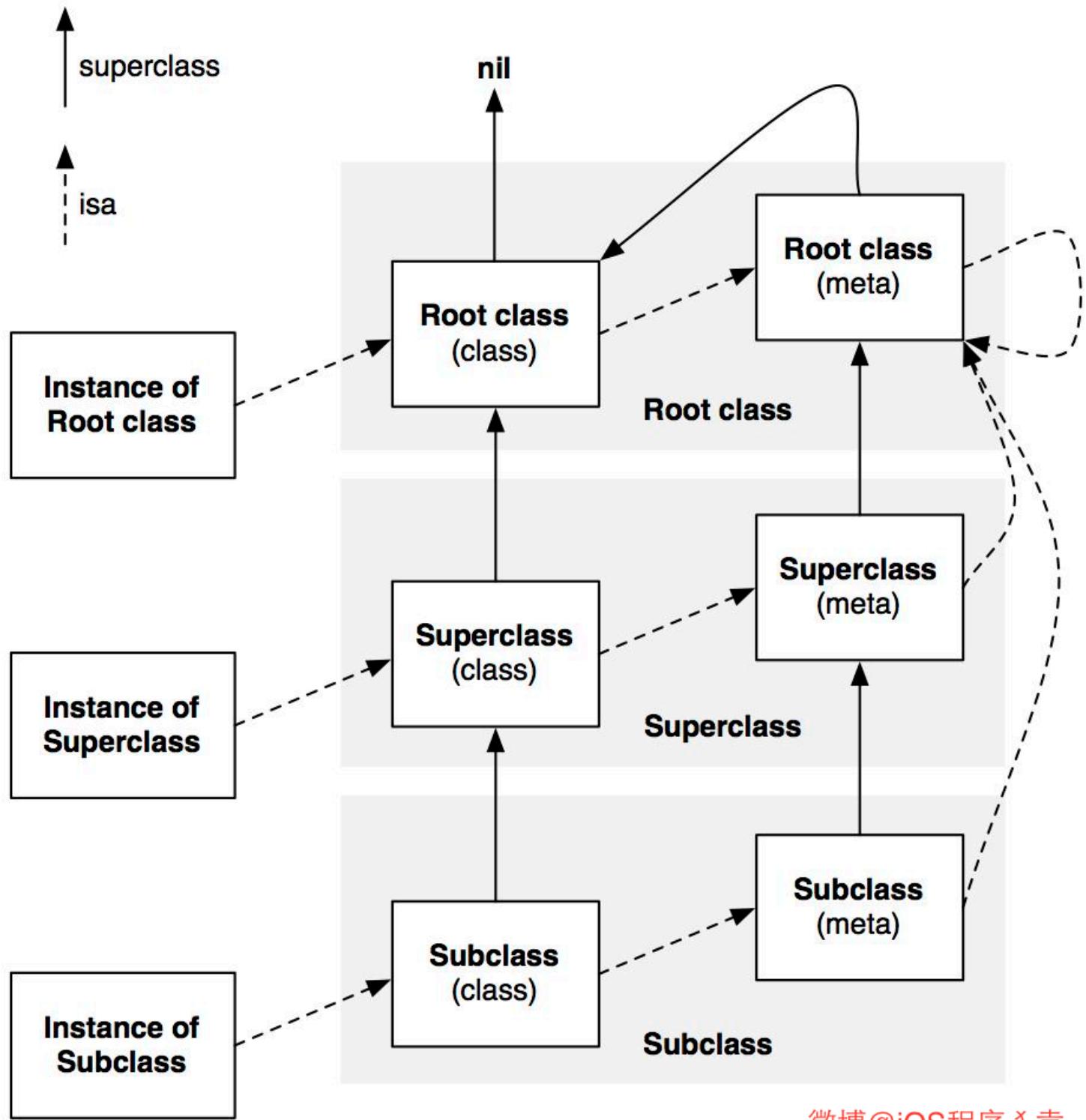
翻译过来就是



- 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？

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？

```
@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程序猿袁. 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：

```
//  
//  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程序猿袁. All rights reserved.  
  
//  
  
#import "Person.h"  
  
@implementation Person  
  
- (instancetype)init  
{  
    self = [super init];  
    if (self) {  
        self.lastName = @"";  
        //NSLog(@"%s类名与方法名: %s (在第%d行), 描述: %@", __PRETTY_FUNCTION__, __LINE__,  
        // NSStringFromClass([self class]));  
        //NSLog(@"%s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：

- 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: Demo21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_0jk35cwn1d3326x0061qym2
80000gn_T_main_a5cecc_mi_0, NSStringFromClass(((Class (*) (id, SEL))(void *)objc_msgSend)((id)self, sel_registerName("class"))));

 NSLog((NSString *)&__NSConstantStringImpl__var_folders_gm_0jk35cwn1d3326x0061qym2
80000gn_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:

```
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。

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程序猿袁)
// 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程序猿袁)  
// 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)

[《招聘一个靠谱的 iOS》](#) –参考答案（下）

说明：面试题来源是[微博@我就叫Sunny怎么了](#)的这篇博文：[《招聘一个靠谱的 iOS》](#)，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程序猿袁](#)整理，未经出题者校对，如有纰漏，请向[微博@iOS程序猿袁](#)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？](#)
2. [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？](#)
4. [28. runloop和线程有什么关系？](#)
5. [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？](#)
7. [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？](#)
9. [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）](#)
11. [35. BAD_ACCESS在什么情况下出现？](#)
12. [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？](#)
14. [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？](#)
16. [40. GCD的队列（dispatchqueue）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）](#)
18. [42. dispatchbarrierasync的作用是什么？](#)
19. [43. 苹果为什么要废弃dispatchgetcurrent_queue？](#)
20. [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");
}
```

21. [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?](#)
22. [46. 如何手动触发一个value的KVO](#)
23. [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?](#)
24. [48. KVC的keyPath中的集合运算符如何使用?](#)
25. [49. KVC和KVO的keyPath一定是属性么?](#)
26. [50. 如何关闭默认的KVO的默认实现, 并进入自定义的KVO实现?](#)
27. [51. apple用什么方式实现对一个对象的KVO?](#)
28. [52. IBOutlet连出来的视图属性为什么可以被设置成weak?](#)
29. [53. IB中User Defined Runtime Attributes如何使用?](#)
30. [54. 如何调试BAD_ACCESS错误](#)
31. [55. lldb \(gdb\) 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。

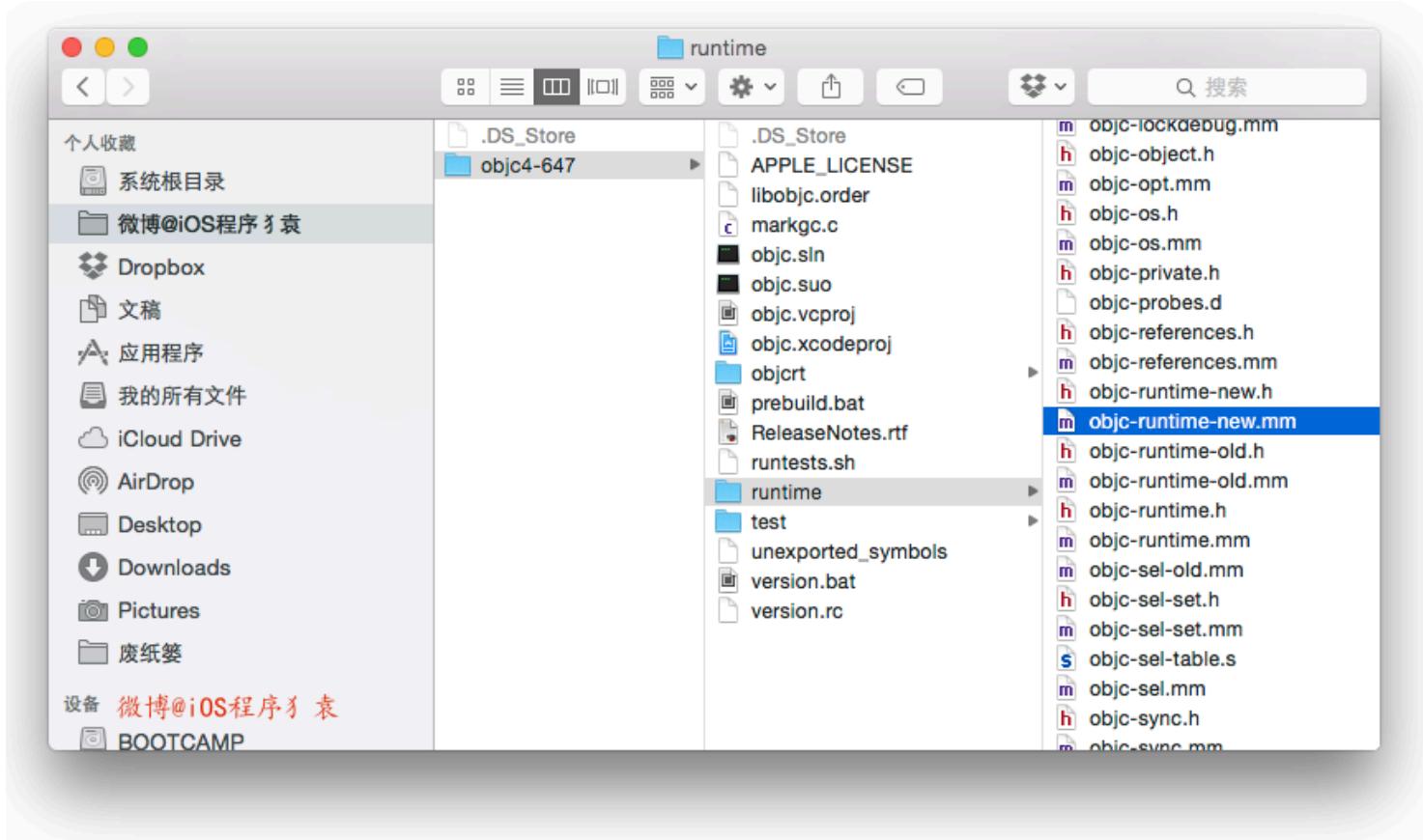
我们可以这样创建一个 `_objc_msgForward` 对象:

```
IMP msgForwardIMP = _objc_msgForward;
```

在[上篇](#)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](#) 下载一个最新版本，找到 `objc-runtime-new.mm`，进入之后搜索 `_objc_msgForward`。



里面有对 `_objc_msgForward` 的功能解释：

```
objc | < > | objc-runtime-new.mm > No Selection  
微博@iOS程序猿袁  
Find ⇧ Find _objc_msgForward 9 × < > Done  
4972 /*****  
4973 * lookUpImpOrForward.  
4974 * The standard IMP lookup.  
4975 * initialize==NO tries to avoid +initialize (but sometimes fails)  
4976 * cache==NO skips optimistic unlocked lookup (but uses cache elsewhere)  
4977 * Most callers should use initialize==YES and cache==YES.  
4978 * inst is an instance of cls or a subclass thereof, or nil if none is known.  
4979 * If cls is an un-initialized metaclass then a non-nil inst is faster.  
4980 * May return _objc_msgForward_impcache. IMPs destined for external use  
4981 * must be converted to _objc_msgForward or _objc_msgForward_stret.  
4982 * If you don't want forwarding at all, use lookUpImpOrNil() instead.  
4983 *****/
```

```
*****  
* lookUpImpOrNil.  
* 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：

```

// 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程序猿袁(https://weibo.com/luohanchenyilong/). All rights reserved.
// 同时, 这也是 objc_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：

```
(void)instrumentObjcMessageSends(YES);
```

或者断点暂停程序运行，并在 gdb 中输入下面的命令：

```
call (void)instrumentObjcMessageSends(YES)
```

以第二种为例，操作如下所示：

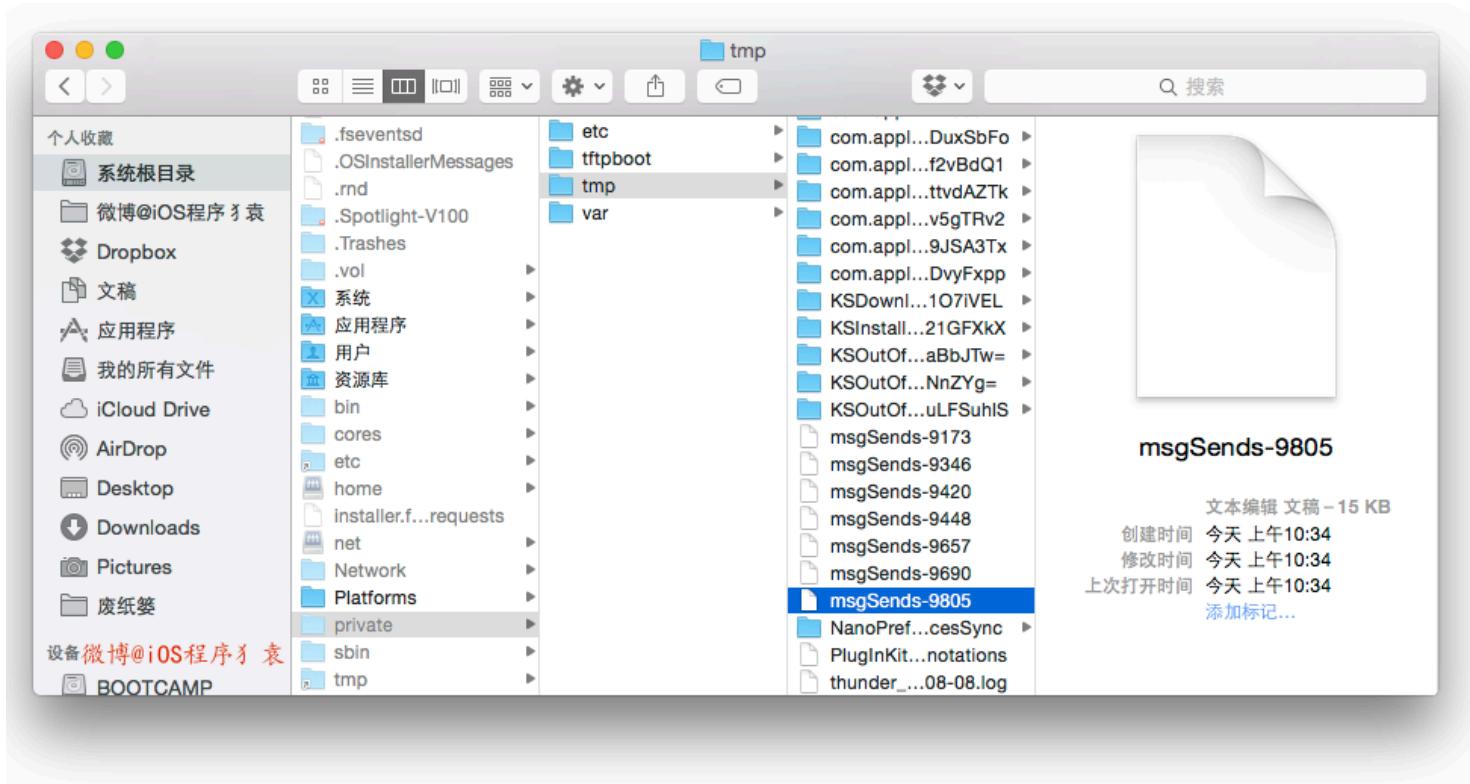
```
1 //  
2 // main.m  
3 // CYLObjcMsgForwardTest  
4 //  
5 // Created by CHENYI LONG on 15/8/9.  
6 // Copyright (c) 2015年 微博@iOS程序猿袁. All rights reserved.  
7 //  
8  
9 #import <UIKit/UIKit.h>  
10 #import "AppDelegate.h"  
11 #import "CYLTest.h"  
12  
13 int main(int argc, char * argv[]) {  
14     @autoreleasepool {  
15         CYLTest *test = [[CYLTest alloc] init]; Thread 1: breakpoint 1.1  
16         [test performSelector:@selector(iOS程序猿袁)]; ! Undeclared selector 'iOS程序猿袁' 2  
17         return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate  
18             class]));  
19     }  
}
```

(lldb) call (void)instrumentObjcMessageSends(YES)

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：

```
open /private/tmp
```



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?](#)），向一个对象发送一条错误的消息：

```
//  
//  main.m  
//  CYLObjcMsgForwardTest  
//  
//  Created by http://weibo.com/luohanchenyilong/.  
//  Copyright (c) 2015年 微博@iOS程序猿袁. 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程序猿袁)];  
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));  
    }  
}
```

The screenshot shows the Xcode IDE with the file `main.m` open. The code is as follows:

```
//  
// main.m  
// CYLObjcMsgForwardTest  
//  
// Created by CHENYI LONG on 15/8/9.  
// Copyright (c) 2015年 微博@iOS程序猿袁. 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程序猿袁)]; // Thread 1: signal SIGABRT  
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));  
    }  
}
```

The line `[test performSelector:@selector(iOS程序猿袁)];` is highlighted in green, indicating it's the source of the error. A yellow warning icon is next to the line number 16. A tooltip for the error says "Thread 1: signal SIGABRT".

Below the editor, the debugger window shows:

```
(lldb) call (void)instrumentobjcMessageSends(YES)  
  
2015-08-09 11:40:01.280 CYLObjcMsgForwardTest[10541:1312227] -[CYLTest iOS程序猿]: unrecognized selector sent to instance 0x7fb92415ce0  
2015-08-09 11:40:01.342 CYLObjcMsgForwardTest[10541:1312227] *** Terminating app due to uncaught exception  
'NSInvalidArgumentException', reason: '-[CYLTest iOS程序猿]: unrecognized selector sent to instance 0x7fb92415ce0'  
*** First throw call stack:  
(  
    0 CoreFoundation 0x000000010387ec65 __exceptionPreprocess + 165  
    1 libobjc.A.dylib 0x0000000101c17bb7 objc_exception_throw + 45  
    2 CoreFoundation 0x00000001038860ad -[NSObject(NSObject) doesNotRecognizeSelector:] + 205  
    3 CoreFoundation 0x00000001037dc13c ___forwarding___ + 988  
    4 CoreFoundation 0x00000001037dbcd8 _CF_forwarding_prep_0 + 120  
    5 CYLObjcMsgForwardTest 0x00000001016e6e15 main + 101  
    6 libdyld.dylib 0x0000000104536145 start + 1  
    7 ??? 0x0000000000000001 0x0 + 1  
)  
libc++abi.dylib: terminating with uncaught exception of type NSException  
(lldb)
```

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

微博@iOS程序猿袁

```
+ 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
+ OS_object NSObject initialize
+ OS_xpc_object NSObject initialize
+ OS_xpc_serializer NSObject initialize
- OS_xpc_serializer OS_xpc_object _xref_dispose
- OS_object OS_object _xref_dispose
- OS_xpc_serializer OS_xpc_object _dispose
- OS_xpc_serializer NSObject dealloc
- OS_xpc_serializer OS_xpc_object _xref_dispose
- OS_object OS_object _xref_dispose
- OS_xpc_serializer OS_xpc_object _dispose
+ OS_dispatch_object NSObject initialize
+ OS_dispatch_mach_msg NSObject initialize
- OS_dispatch_mach_msg OS_dispatch_object _xref_dispose
- OS_object OS_object _xref_dispose
- OS_dispatch_mach_msg OS_dispatch_object _dispose
- OS_dispatch_mach_msg NSObject dealloc
```

```
+ 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》](#)，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 `NSInvocation` 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 `nil`。
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：

--	<code>_objc_msgForward</code> 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：

```
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：



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 \(Github 链接\)](#) 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》](#)详细记录了实现原理，有兴趣可以看下。

同时[RAC\(ReactiveCocoa\)](#)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](#)中的《runtime如何实现weak属性》有论述。（注：[上篇](#)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”：

```
// 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_destroyWeak` 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;  
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。

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

```
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。

```
NSRunLoop *runloop = [NSRunLoop currentRunLoop];
```

参考链接：[《Objective-C之run loop详解》](#)。

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:

```
//  
// http://weibo.com/luohanchenyilong/ (微博@ios程序猿袁)  
// 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程序猿袁)  
// 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》](#)
2. 摘自博文[CFRunLoop](#), 原作者是微博@我就叫Sunny怎么了

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？

编译时根据代码上下文，插入 retain/release

ARC相对于MRC，不是在编译时添加retain/releaseautorelease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/releaseautorelease，这样做性能更好，也是为什么不能在ARC环境下手动retain/releaseautorelease，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】

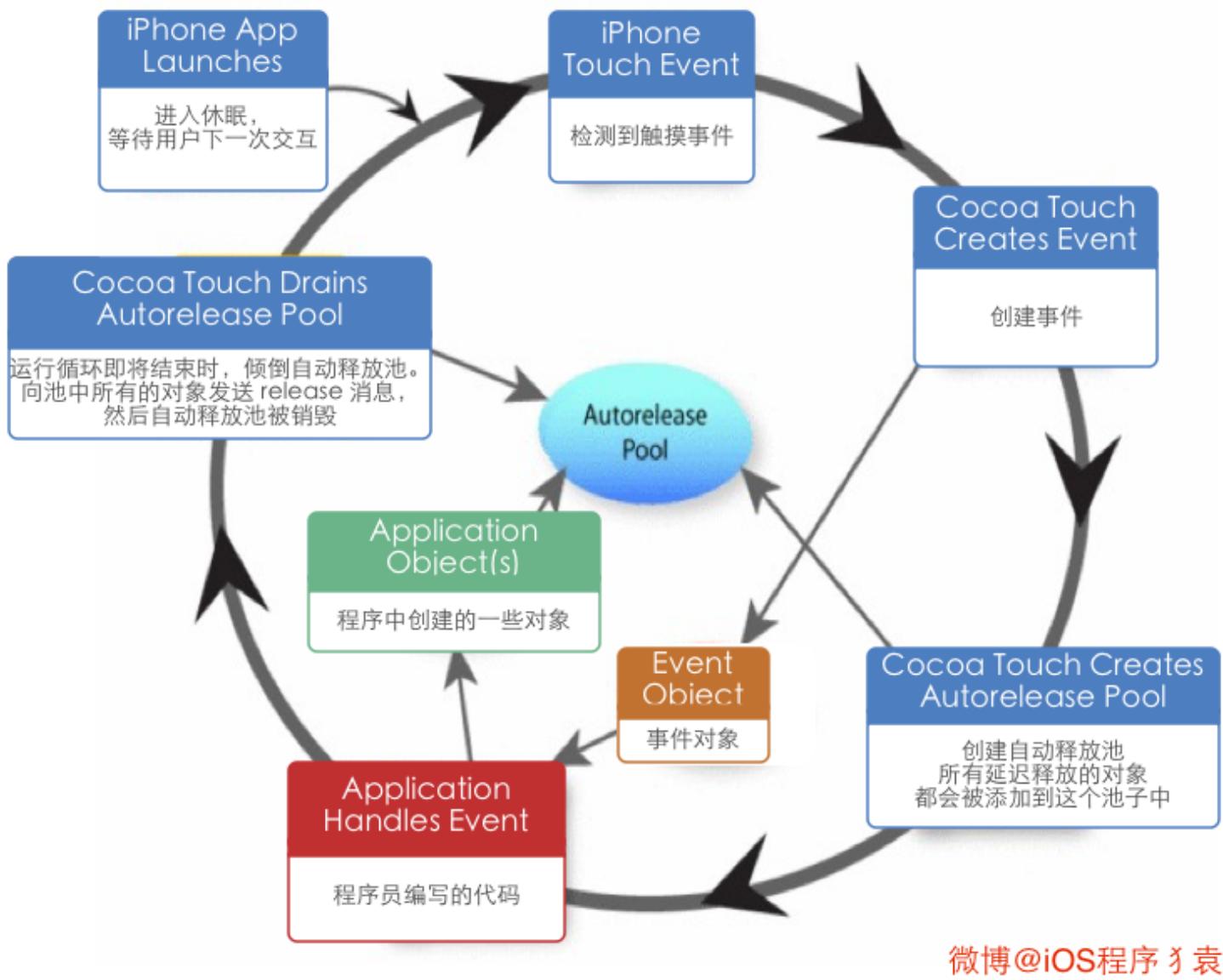
34. 不手动指定autoreleasepool的前提下，一个autorelease对象在什么时刻释放？ (比如在一个vc的viewDidLoad中创建)

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：



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 viewWillAppear 方法执行前就被销毁了。

参考链接：[《黑幕背后的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

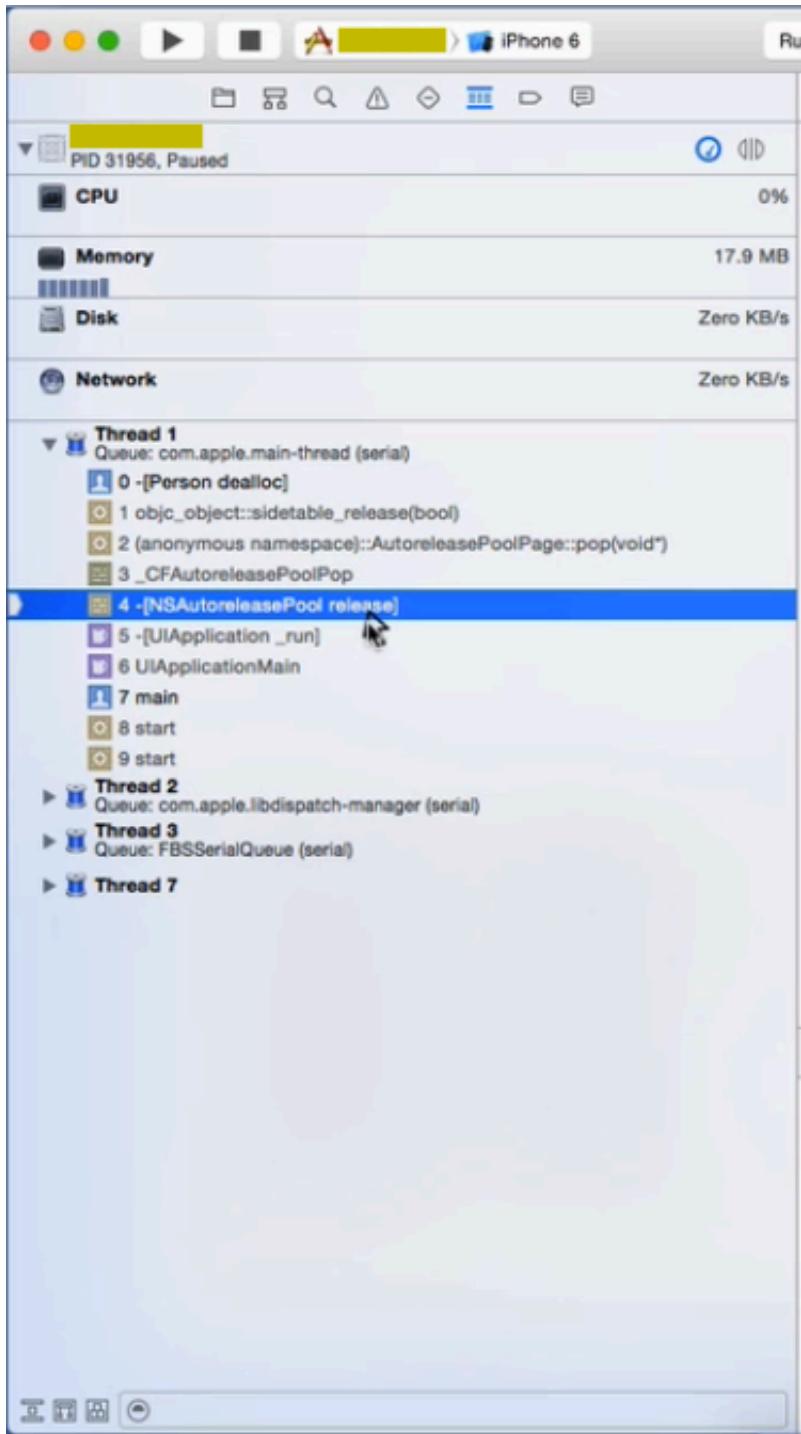
36. 苹果是如何实现autoreleasepool的？

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：我们都应该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：



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，就会发生循环引用。

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](#)。

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：

```
__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](#) 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：

我们都应该知道：**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。进而再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：

```
__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();
```

```
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。

```
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
```

```

58 NSString *a = [NSMutableString stringWithString:@"Tom"];
59 NSLog(@"\n 定以前: -----\n"
60     " 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
61 void (^foo)(void) = ^{
62     a.string = @"Jerry";
63     NSLog(@"\n block内部: -----\n"
64     " 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
65     a = [NSMutableString stringWithString:@"William"];                      // Variable is not assignable (missing __block type specifier)
66 };
67 foo();
68 NSLog(@"\n 定以后: -----\n"
69     " 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

```

StackHeepDemo

2016-05-18 10:49:03.382 StackHeepDemo[2429:1092004]
定以前:
a指向的堆中地址: 0x12f59e900; a在栈中的指针地址: 0x16fdc1fa8
2016-05-18 10:49:03.382 StackHeepDemo[2429:1092004]
block内部:
a指向的堆中地址: 0x12f59e900; a在栈中的指针地址: 0x12f59e960
2016-05-18 10:49:03.382 StackHeepDemo[2429:1092004]
定以后:
a指向的堆中地址: 0x12f59e900; a在栈中的指针地址: 0x16fdc1fa8

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：

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

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

```
[ [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：

```
_weak __typeof__(self) weakSelf = self;
dispatch_group_async(_operationsGroup, _operationsQueue, ^{
{
__typeof__(self) strongSelf = weakSelf;
[strongSelf doSomething];
[strongSelf doSomethingElse];
} );
```

类似的：

```
_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](#)。

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。

```
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？

```

// 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：

```

// 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 `didChangeValueForKey:`。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，`observeValueForKeyPath: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。

```
// .m文件
// Created by https://github.com/ChenYilong
// 微博@ios程序猿袁(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：

```
- (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](#)

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](#)

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》](#)
2. [KVO for manually implemented properties](#)

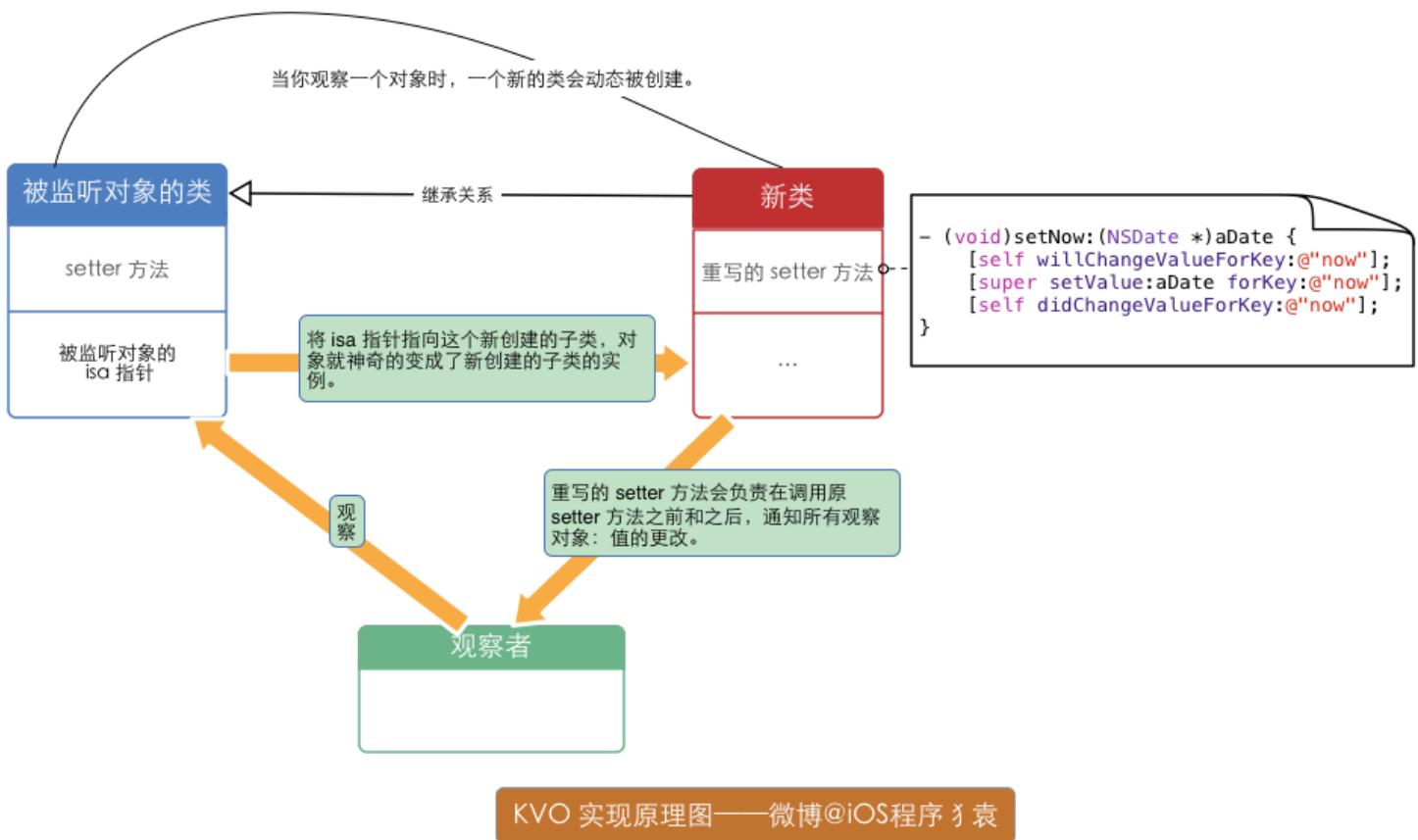
51. apple用什么方式实现对一个对象的KVO？

[Apple 的文档](#)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：



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 `didChangeValueForKey:`。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：

```
- (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 `wilChangeValueForKey:`，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：

```
- (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](#)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](#)。

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。

```
- (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");
}
```

```

5 // Created by 微博@iOS程序猿 on 16/4/6.
6 // Copyright © 2016年 ElonChan. All rights reserved.
7 //
8
9 #import "ViewController.h"
10
11 @interface ViewController : UIViewController
12 @property (nonatomic, strong) NSDate *now;
13 @end
14
15 @implementation ViewController
16
17 - (void)viewDidLoad {
18     [super viewDidLoad];
19     [self addObserver:self forKeyPath:@"now" options:NSKeyValueObservingOptionNew context:nil];
20     NSLog(@"1");
21     [self willChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。
22     NSLog(@"2");
23     [self didChangeValueForKey:@"now"]; // “手动触发self.now的KVO”，必写。
24     NSLog(@"4");
25 }
26
27 - (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary<NSString *, id> *)change context:(void *)context {
28     NSLog(@"3");
29 }

```

KVO实现原理

2016-04-06 15:11:17.727 KVO实现原理[57795:6834661] 1
2016-04-06 15:11:17.728 KVO实现原理[57795:6834661] 2
2016-04-06 15:11:17.728 KVO实现原理[57795:6834661] 3
2016-04-06 15:11:17.728 KVO实现原理[57795:6834661] 4

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](#) 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. IBOulet连出来的视图属性为什么可以被设置成weak?

参考链接: [Should IBOulet 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。

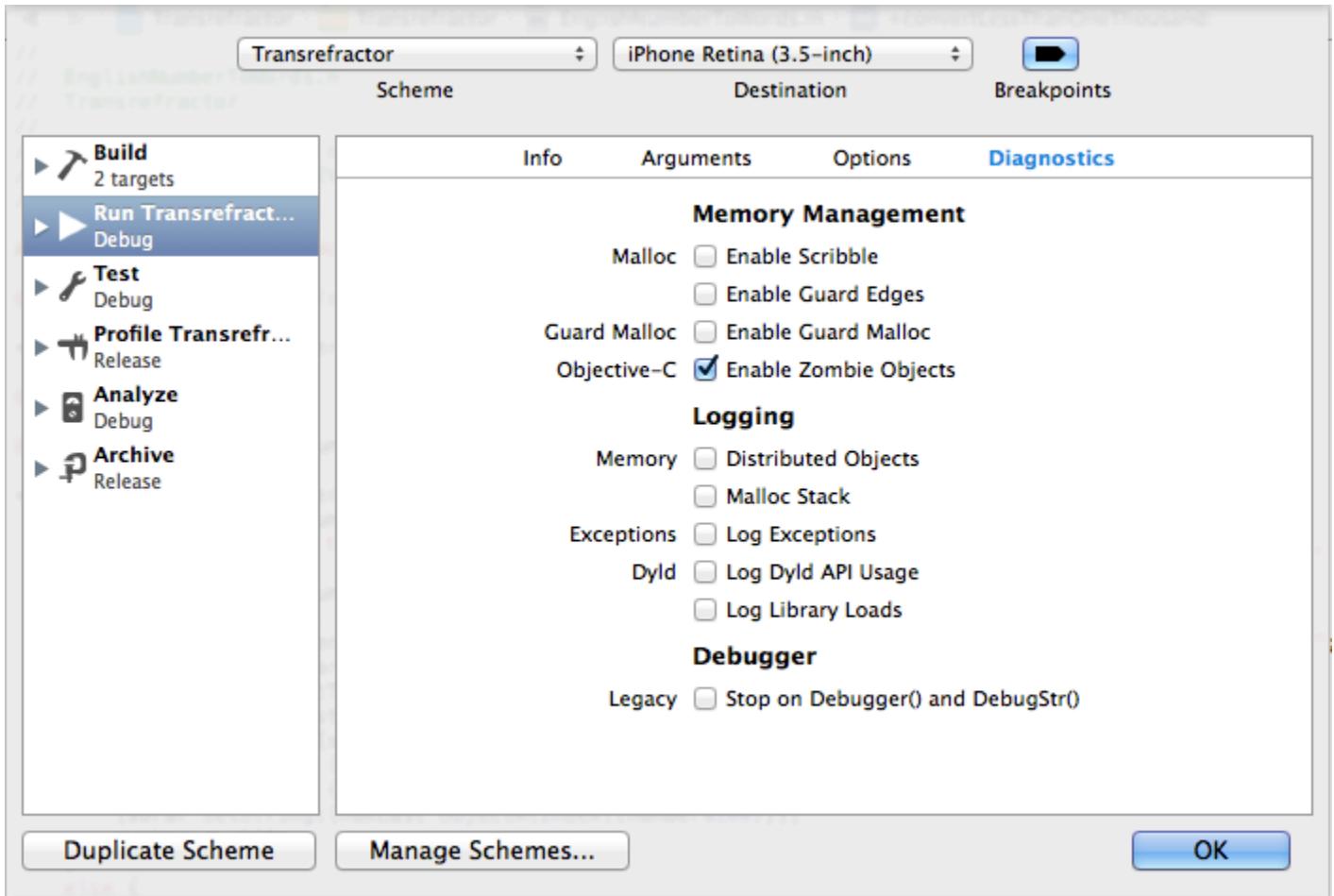
不过这个回答漏了个重要知识点, 使用storyboard (xib不行) 创建的vc, 会有一个叫`_topLevelObjectsToKeepAliveFromStoryboard`的私有数组强引用所有top level的对象, 所以这时即便outlet声明成weak也没关系

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EXCECBADACCESS前访问的最后一个object
2. 通过 Zombie



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 ?

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](#)；
2. 苹果官方文档：[iOS Debugging Magic](#)。

Posted by [微博@iOS程序猿袁](#)

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](#)

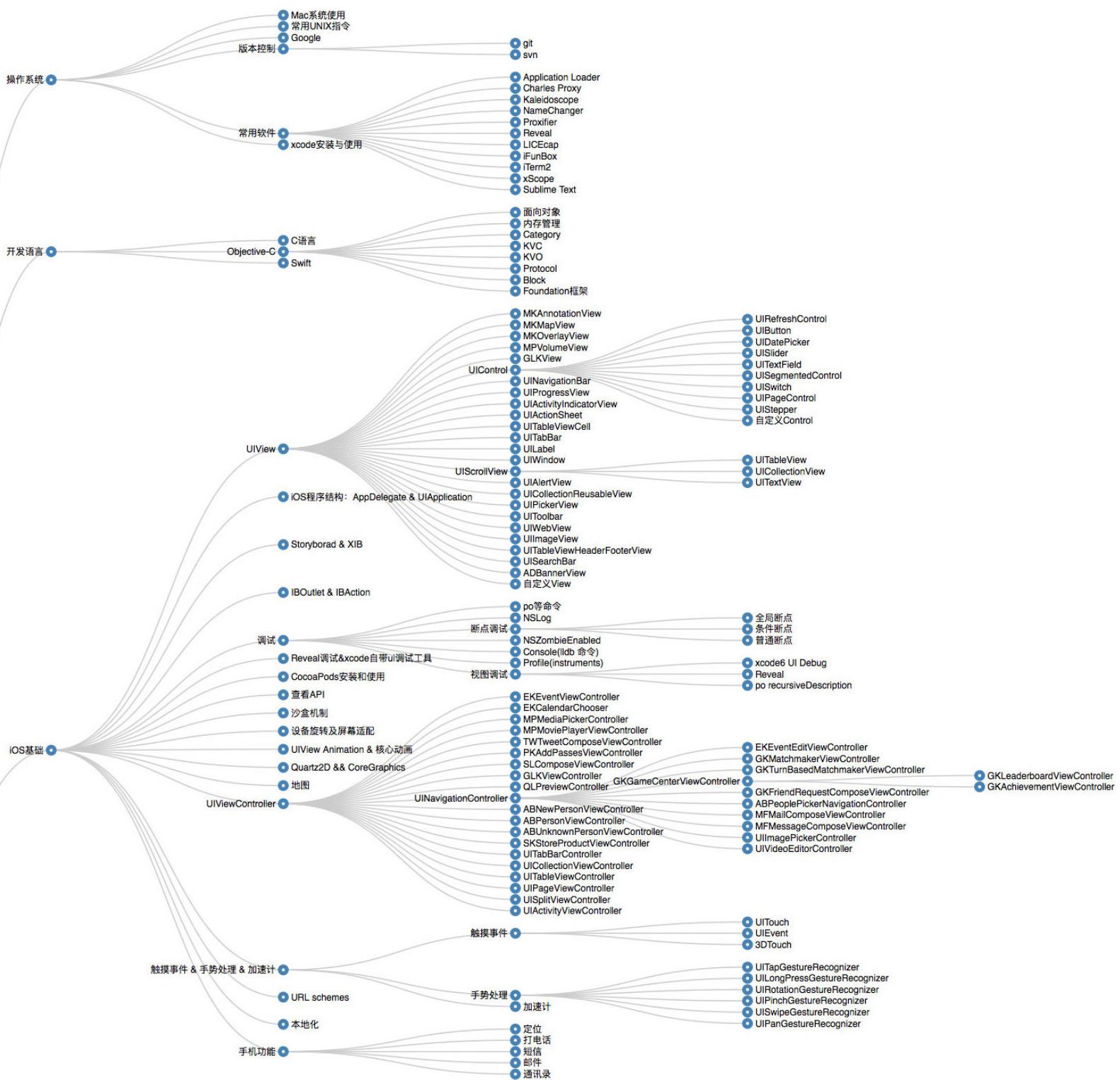
目录

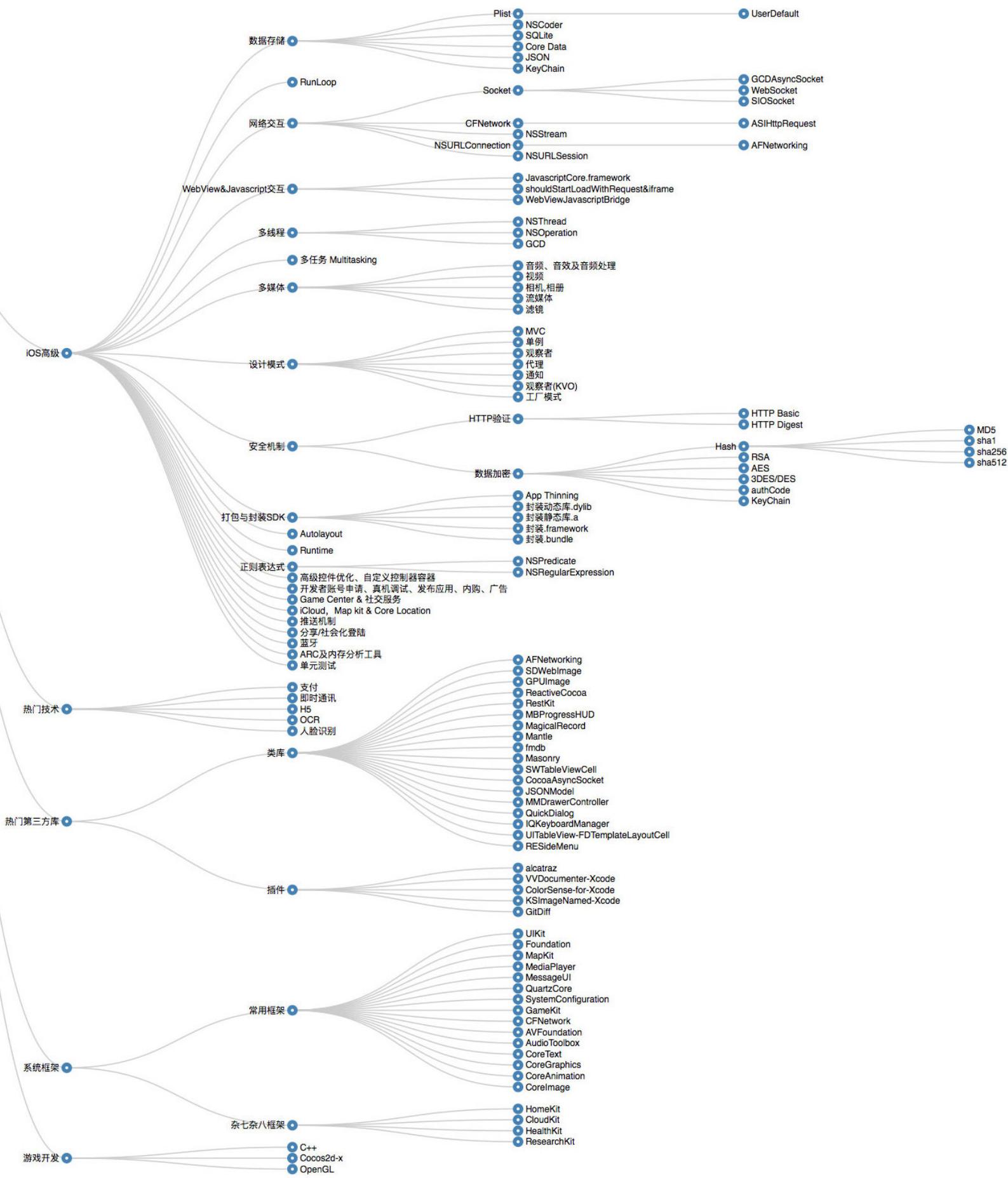
(1) 序言	2
(2) 版本控制	4
(3) <i>UNIX</i> 常用命令	8
(4) <i>C</i> 语言	9
(5) <i>Objective-C</i>	15
(6) 内存管理	35
(7) <i>KVO-KVC</i>	45
(8) <i>Block</i>	47
(9) <i>Swift</i>	59
(10) <i>UI</i>	62
(11) 报错警告调试	85
(12) 第三方框架及其管理	90
(13) 绘图与动画	95
(14) 数据存储	101
(15) <i>RunLoop</i>	110
(16) 网络	117
(17) <i>WebView</i> 与 <i>JS</i> 交互	128
(18) 多线程	137
(19) 多媒体	147
(20) 设计模式	148
(21) 安全机制	158
(22) <i>Runtime</i>	159
(23) 性能优化	178
(24) 通知与推送	184
——补充	186

(1) 序言

培训机构量产iOS程序员，导致了现在iOS就业的浮躁和困难。但是技术好的人仍然不愁工作，而一些想进入行业捞一笔就走的人，势必在今年这种艰难就业形式下，被迫淘汰，转行。

这张图是github上一位大牛所制作。建议找工作的同学，把这张图打印出来，自己对着看，有哪些知识点遗忘的，赶紧去复习，每天过一遍，保证你面试的时候胸有成竹。





(2) 版本控制

面试过程中，可能会问及一些关于版本控制的问题，理解下SVN和Git的原理，记住常用命令即可。

SVN

- SVN 是集中式源代码管理工具

概念：

- 1> Repository 代码仓库，保存代码的仓库
- 2> Server 服务器，保存所有版本的代码仓库
- 3> Client 客户端，只保存当前用户的代码仓库
- 4> 用户名&密码 访问代码仓库需要使用自己的"用户名和密码"，从而可以区分出不同的人对代码做的修改

操作：

- 1> checkout 将服务器上最新的代码仓库下载到本地，"只需要做一次"
- 2> update 从服务器上将其他人所做的修改下载到本地，"每天上班必须要做的事情"
- 3> commit 将工作提交到服务器，"每天下班之前至少做一次"

- SVN服务器安装(略)
- SVN常用命令

切换工作目录

```
$ cd 工作目录
```

checkout服务器上的代码仓库

```
$ svn co http://xxx/svn/xxxx --username=manager --password=manager
```

提示：checkout(co)之后，命令行会记录用户名和密码，后续操作不用再另行指定

查看本地代码库状态

```
$ svn st
```

错误提示："is not a working copy"，必须在svn的工作目录下才能正确使用svn的命令

查看svn日志

```
$ svn log
```

查看某一个文件的日志

```
$ svn log filename
```

查看某一个文件某个版本的日志

```
$ svn log filename@1
```

创建文件

```
$ touch main.c
```

打开并编写文件内容

```
$ open main.c
```

查看工作目录状态

```
$ svn st
```

将文件添加到本地版本库中

```
$ svn add main.c/main.*
```

将文件提交到服务器的版本库中

```
$ svn ci -m "备注信息"
```

注意：一定要养成写注释的良好习惯

删除文件

```
$ svn rm Person.h
```

提交删除

```
$ svn ci -m "删除了文件"
```

注意：不要使用文件管理器直接删除文件

撤销修改

```
$ svn revert Person.m
```

恢复到之前的某个版本

```
$ svn update -r 5
```

冲突解决

(p) postpone	对比
(mc) mine-conflict	使用我的
(tc) theirs-conflict	使用对方的

svn st 显示的文件状态

' '	没有修改
'A'	被添加到本地代码仓库
'C'	冲突
'D'	被删除
'I'	被忽略
'M'	被修改
'R'	被替换
'X'	外部定义创建的版本目录
'?'	文件没有被添加到本地版本库内
'!'	文件丢失或者不完整（不是通过svn命令删除的文件）
'~'	受控文件被其他文件阻隔

Git

- git是一款开源的分布式版本控制工具

```
$ git help  
查看git所有命令的帮助  
$ git help 子命令
```

要退出帮助信息，按"q"
翻看下页，按"空格"
翻看上页，按"CTRL+B"
要搜索相关文字，按"/"然后输入"相关文字"

创建代码仓库

```
$ git init
```

配置用户名和邮箱

```
$ git config user.name manager  
$ git config user.email manager@gmail.com
```

以上两个命令会将用户信息保存在当前代码仓库中

如果要一次性配置完成可以使用一下命令

```
$ git config --global user.name manager  
$ git config --global user.email manager@gmail.com
```

以上两个命令会将用户信息保存在用户目录下的 .gitconfig 文件中

查看当前所有配置

```
$ git config -l
```

创建代码，开始开发

```
$ touch main.c  
$ open main.c
```

将代码添加到代码库

查看当前代码库状态

```
$ git status
```

将文件添加到代码库

```
$ git add main.c
```

将修改提交到代码库

```
$ git commit -m "添加了main.c"
```

在此一定要使用 -m 参数指定修改的备注信息

否则会进入 vim 编辑器，如果对vim不熟悉，会是很糟糕的事情

将当前文件夹下的所有新建或修改的文件一次性添加到代码库

```
$ git add .
```

添加多个文件

```
$ touch Person.h Person.m
```

```
$ git add .
$ git commit -m "添加了Person类"
$ open Person.h
$ git add .
$ git commit -m "增加Person类属性"
```

注意 使用git时，每一次修改都需要添加再提交，这一点是与svn不一样的

查看所有版本库日志

```
$ git log
```

查看指定文件的版本库日志

```
$ git log 文件名
```

回到当前版本，放弃所有没有提交的修改

```
$ git reset --hard HEAD
```

回到上一个版本

```
$ git reset --hard HEAD^
```

回到之前第3个修订版本

```
$ git reset --hard HEAD~3
```

回到指定版本号的版本

```
$ git reset --hard e695b67
```

查看分支引用记录

```
$ git reflog
```

为什么要用源代码管理工具

- 能追踪一个项目从诞生一直到定案的过程
- 记录一个项目的所有内容变化
- 方便地查阅特定版本的修订情况

最常用的版本控制工具是什么，能大概讲讲原理么？

- 最常用的版本控制工具有SourceTree（GIT）和CornerStone（SVN）；
- 原理提到svn是集中式代码管理，解释下具体意思，git也这样回答就行了。
- 集中式代码管理（SVN）的核心是服务器，所有开发者在开始新一天的工作之前必须从服务器获取代码，然后开发，最后解决冲突，提交。所有的版本信息都放在服务器上。如果脱离了服务器，开发者基本上可以说是无法工作的。
- 分布式的版本控制系统，在Git中并不存在主库这样的概念，每一份出的库都可以独立使用，任何两个库之间的不一致之处都可以进行合并。

(3) UNIX常用命令

做开发说用不到命令行，那肯定是不可能的。所以记住几个常用的命令还是很有用。

1. cd 改变工作目录

2. pwd 输出当前工作目录的绝对路径

在UNIX中要执行什么命令，一定要知道自己当前所在的工作目录

3. ls 查看文件

\$ ls 显示文件

\$ ls -a 显示所有文件

\$ ls -l 列表显示文件

\$ ls -la 列表显示所有文件

4. touch 用于更改文件访问和修改时间的标准UNIX程序，也被用于创建新文件

```
$ touch test.txt
```

注意：touch不修改test.txt内容，只更改它的访问、修改时间，如果test.txt不存在，它会被创建

1. cat 连续查看文件内容

2. more 分页查看文件内容

提示：

1> 命令和参数之间需要添加空格

2> 如果要使用当前目录中的文件名，输入到一半时，按TAB键能够补全

(4) C语言

C语言，开发的基础功底，iOS很多高级应用都要和C语言打交道，所以，C语言在iOS开发中的重要性，你懂的。里面的一些问题可能并不是C语言问题，但是属于计算机的一些原理性的知识点，所以我就不再另外写一篇文章了，直接写在这里。

当你写下面的代码时会发生什么事？

- `least = MIN(*p++, b);`
- 结果是：`((p++) <= (b) ? (p++) : (*p++))` 这个表达式会产生副作用，指针p会作三次++自增操作。

用预处理指令#define声明一个常数，用以表明1年中有多少秒（忽略闰年问题）

```
define SECONDS_PER_YEAR (60 60 24 * 365)UL(UL无符号长整形)
```

写一个"标准"宏MIN，这个宏输入两个参数并返回较小的一个。

```
define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

写一个标准宏Max，并给出以下代码的输出

```
int array[5] = {1, 2, 3, 4, 5};  
int *p = &array[0];  
int max = Max(*p++, 1);  
printf("%d %d", max, *p);
```

参考答案： 1, 2

```
#define Max(X, Y) ((X) > (Y) ? (X) : (Y))
```

当看到宏时，就会想到宏定义所带来的副作用。对于++、-，在宏当中使用是最容易产生副作用的，因此要慎用。

分析：

p指针指向了数组array的首地址，也就是第一个元素对应的地址，其值为1。

宏定义时一定要注意每个地方要加上圆括号

*p++相当于*p，p++，所以Max(*p++, 1)相当于：

`(*p++) > (1) ? (*p++) : (1)`

=>

`(1) > (1) ? (*p++) : (1)`

=>

第一个*p++的结果是，p所指向的值变成了2，但是1 > 1为值，所以最终max的值就是1。

而后面的(*p++)也就不会执行，因此p所指向的地址对应的值就是2，而不是3。

扩展：如果上面的*p++改成*(++p)如何？

```
(*++p) > (1) ? (*++p) : (1)
=>
(2) > (1) ? (*++p) : (1)
=>
max = *++p;
=>
*p = 3, max = 3;
```

define定义的宏和const定义的常量有什么区别？

λ #define定义宏的指令，程序在预处理阶段将用#define所定义的内容只是进行了替换。因此程序运行时，常量表中并没有用#define所定义的宏，系统并不为它分配内存，而且在编译时不会检查数据类型，出错的概率要大一些。

λ const定义的常量，在程序运行时是存放在常量表中，系统会为它分配内存，而且在编译时会进行类型检查。

#define定义表达式时要注意“边缘效应”，例如如下定义：

```
#define N 2 + 3 // 我们预想的N值是5，我们这样使用N
int a = N / 2; // 我们预想的a的值是2.5，可实际上a的值是3.5
```

关键字volatile有什么含意？并给出三个不同的例子

- 优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是volatile变量的几个例子：

- 并行设备的硬件寄存器（如：状态寄存器）
- 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)
- 多线程应用中被几个任务共享的变量

完成字符串拷贝可以使用sprintf、strcpy、以及memcpy函数，请问这些函数有什么区别？你喜欢哪一个？为什么？

这些函数的区别在于实现功能以及操作对象不同。

- strcpy：函数操作的对象是字符串，完成从源字符串到目的字符串的拷贝功能。
- sprintf：这个函数主要用来实现（字符串或基本数据类型）向字符串的转换功能。如果源对象是字符串，并且指定%s格式符，也可实现字符串拷贝功能。
- memcpy：函数顾名思义就是内存拷贝，实现将一个内存块的内容复制到另一个内存块这一功能。内存块由其首地址以及长度确定。因此，memcpy 的操作对象适用于任意数据类型，只要能给出对象的起始地址和内存长度信息、并且对象具有可操作性即可。鉴于memcpy 函数等长拷贝的特点以及数据类型代表的物理意义，memcpy函数通常限于同种类型数据或对象之间的拷贝，其中当然也包括字符串拷贝以及基本数据类型的拷贝。

- 对于字符串拷贝来说，用上述三个函数都可以实现，但是其实现的效率和使用的方便程度不同：
 - `strcpy` 无疑是最合适的选择：效率高且调用方便。
 - `snprintf` 要额外指定格式符并且进行格式转化，麻烦且效率不高。
 - `memcpy` 虽然高效，但是需要额外提供拷贝的内存长度这一参数，易错且使用不便；并且如果长度指定过大的话（最优长度是源字符串长度 + 1），还会带来性能的下降。其实 `strcpy` 函数一般是在内部调用 `memcpy` 函数或者用汇编直接实现的，以达到高效的目的。因此，使用 `memcpy` 和 `strcpy` 拷贝字符串在性能上应该没有什么大的差别。
 - 对于非字符串类型的数据的复制来说，`strcpy` 和 `snprintf` 一般就无能为力了，可是对 `memcpy` 却没有什么影响。但是，对于基本数据类型来说，尽管可以用 `memcpy` 进行拷贝，由于有赋值运算符可以方便且高效地进行同种或兼容类型的数据之间的拷贝，所以这种情况下 `memcpy` 几乎不被使用。`memcpy` 的长处是用来实现（通常是内部实现居多）对结构或者数组的拷贝，其目的是或者高效，或者使用方便，甚或两者兼有。

sprintf, strcpy, memcpy 使用上有什么要注意的地方

- `strcpy` 是一个字符串拷贝的函数，它的函数原型为 `strcpy(char dst, const char src);`
- 将 `src` 开始的一段字符串拷贝到 `dst` 开始的内存中去，结束的标志符号为 '`\0`'，由于拷贝的长度不是由我们自己控制的，所以这个字符串拷贝很容易出错。
- 具备字符串拷贝功能的函数有 `memcpy`，这是一个内存拷贝函数，它的函数原型为 `memcpy(char dst, const char src, unsigned int len);` 将长度为 `len` 的一段内存，从 `src` 拷贝到 `dst` 中去，这个函数的长度可控。但是会有内存读写错误。（比如 `len` 的长度大于要拷贝的空间或目的空间）
- `sprintf` 是格式化函数。将一段数据通过特定的格式，格式化到一个字符串缓冲区中去。`sprintf` 格式化的函数的长度不可控，有可能格式化后的字符串会超出缓冲区的大小，造成溢出。

static关键字的作用

- 隐藏。编译多个文件时，所有未加 `static` 前缀的全局变量和函数都全局可见。
 - 保持变量内容的持久。全局变量和 `static` 变量都存储在静态存储区，程序开始运行就初始化，只初始化一次。`static` 控制了变量的作用范围。
 - 默认初始化为 0。在静态数据区，内存中的所有字节都是 `0x00`，全局变量和 `static` 变量都是默认初始化为 0。

static关键字区别：

- static全局变量与普通的全局变量有什么区别：static全局变量只初始化一次，防止在其他文件单元中被引用；
- static局部变量和普通局部变量有什么区别：static局部变量只被初始化一次，下一次依据上一次结果值；
- static函数与普通函数有什么区别：static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝

关键字const

- const int a;int const a; 作用是一样：a 是一个常整型数
- const int a;int const a; a 是一个指向常整型数的指针(整型数是不可修改的，但指针可以)
- int * const a;a 是一个指向整型数的常指针(指针指向的整型数是可以修改的，但指针是不可修改的)
- int const * const a;a 是一个指向常整型数的常指针(指针指向的整型数是不可修改的，同时指针也是不可修改的)

堆栈

- 管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生内存泄露 (memory leak)。
- 申请大小：
 - 栈：在Windows下，栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在Windows下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。
 - 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。
- 碎片问题：
对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出
- 分配方式：
堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloc函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

- 分配效率：
栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制是很复杂的

数组和指针的区别

- 数组可以申请在栈区和数据区；指针可以指向任意类型的内存块
sizeof作用于数组时，得到的是数组所占的内存大小；作用于指针时，得到的都是4个字节的大小
- 数组名表示数组首地址，是常量指针，不可修改指向。比如不可以将++作用于数组名上；普通指针的值可以改变，比如可将++作用于指针上
- 用字符串初始化字符数组是将字符串的内容拷贝到字符数组中；用字符串初始化字符指针是将字符串的首地址赋给指针，也就是指针指向了该字符串

引用和指针的区别

- 指针指向一块内存，内容存储所指内存的地址。
- 引用是某块内存的别名。
- 引用使用时不需要解引用(*)而指针需要
- 引用只在定义时被初始化，之后不可变，指针可变。
- 引用没有const
- 引用不能为空
- sizeof引用得到的是所指向变量（对象）的大小，sizeof指针是指针本身的大小。
- 指针和引用的自增(++)运算意义不一样：引用++为引用对象自己++，指针++是指向对象后面的内存
- 程序需要为指针分配内存区域，引用不需要。

用变量a给出下面的定义

- 一个整型数 (An integer)
- 一个指向整型数的指针 (A pointer to an integer)
- 一个指向指针的指针，它指向的指针是指向一个整型数 (A pointer to a pointer to an integer) r
- 一个有10个整型数的数组 (An array of 10 integers)
- 一个有10个指针的数组，该指针是指向一个整型数的。 (An array of 10 pointers to integers)
- 一个指向有10个整型数数组的指针 (A pointer to an array of 10 integers)
- 一个指向函数的指针，该函数有一个整型参数并返回一个整型数 (A pointer to a function that takes an integer as an argument and returns an integer)

- 一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数 (An array of ten pointers to functions that take an integer argument and return an integer)
- 答案是：
- int a; // An integer
- int *a; // A pointer to an integer
- int **a; // A pointer to a pointer to an integer
- int a[10]; // An array of 10 integers
- int *a[10]; // An array of 10 pointers to integers
- int (*a)[10]; // A pointer to an array of 10 integers
- int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
- int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

请写出以下代码输出

```
int a[5] = {1, 2, 3, 4, 5};
int *ptr = (int *)(&a + 1);
printf("%d, %d", *(a + 1), *(ptr + 1));
```

参考答案：2，随机值

这种类型题好像挺常见的。考的就是C语言上的指针的理解和数组的理解。

分析：

a代表有5个元素的数组的首地址，a[5]的元素分别是1, 2, 3, 4, 5。接下来，a + 1表示数据首地址加1，那么就是a[1]，也就是对应于值为2。但是，这里是&a + 1，因为a代表的是整个数组，它的空间大小为5 * sizeof(int)，因此&a + 1就是a+5。a是个常量指针，指向当前数组的首地址，指针+1就是移动sizeof(int)个字节。

因此，ptr是指向int *类型的指针，而ptr指向的就是a + 5，那么ptr + 1也相当于a + 6，所以最后的*(ptr + 1)就是一个随机值了。而*(ptr - 1)就相当于a + 4，对应的值就是5。

简述内存分区情况

- 代码区：存放函数二进制代码
- 数据区：系统运行时申请内存并初始化，系统退出时由系统释放，存放全局变量、静态变量、常量
- 堆区：通过malloc等函数或new等操作符动态申请得到，需程序员手动申请和释放
- 栈区：函数模块内申请，函数结束时由系统自动释放，存放局部变量、函数参数

用NSLog函数输出一个浮点类型，结果四舍五入，并保留一位小数

```
float money = 1.011;
NSLog(@"%@", money);
```

(5) Objective-C

面试笔试都是必考语法知识的。请认真复习和深入研究OC。

方法和选择器有何不同? (Difference between method and selector?)

- selector是一个方法的名字， method是一个组合体，包含了名字和实现.

Core Foundation的内存管理

- 凡是带有Create、Copy、Retain等字眼的函数，创建出来的对象，都需要在最后做一次release
- 比如CFRunLoopObserverCreate release函数：CFRelease(对象);

malloc和New的区别

- new 是c++中的操作符， malloc是c 中的一个函数
- new 不止是分配内存，而且会调用类的构造函数，同理delete会调用类的析构函数，而malloc则只分配内存，不会进行初始化类成员的工作，同样free也不会调用析构函数
- 内存泄漏对于malloc或者new都可以检查出来的，区别在于new可以指明是那个文件的那一行，而malloc没有这些信息。
- new 和 malloc效率比较
- new可以认为是malloc加构造函数的执行。
- new出来的指针是直接带类型信息的。

你是否接触过OC中的反射机制？简单聊一下概念和使用

- class反射
- 通过类名的字符串形式实例化对象

```
Class class = NSClassFromString(@"student");
Student *stu = [[class alloc] init];
```
- 将类名变为字符串

```
Class class = [Student class];
NSString *className = NSStringFromClass(class);
```
- SEL的反射
- 通过方法的字符串形式实例化方法

```
SEL selector = NSSelectorFromString(@"setName");
[stu performSelector:selector withObject:@"Mike"];
```



iOS资源群: 190892815

iOS逆向资源: (定期分享)

<http://weibo.com/Edstick>

- 将方法变成字符串
NSStringFromSelector(@selector*(setName:))

什么是SEL?如何声明一个SEL?通过那些方法能够,调用SEL包装起来的方法?

- SEL就是对方法的一种包装。包装的SEL类型数据它对应相应的方法地址，找到方法地址就可以调用方法。在内存中每个类的方法都存储在类对象中，每个方法都有一个与之对应的SEL类型的数据，根据一个SEL数据就可以找到对应的方法地址，进而调用方法。
- SEL s1 = @selector(test1); // 将test1方法包装成SEL对象
- SEL s2 = NSSelectorFromString(@"test1"); // 将一个字符串方法转换成为SEL对象
- 调用方法有两种方式：
- 1.直接通过方法名来调用 [person text]
- 2.间接的通过SEL数据来调用 SEL aaa=@selector(text); [person performSelector:aaa];

协议中<NSObject>是什么意思?子类继承了父类,那么子类会遵守父类中遵守的协议吗?协议中能够定义成员变量?如何约束一个对象类型的变量要存储的地址是遵守一个协议对象?

- 遵守NSObject协议
- 会
- 能，但是只在头文件中声明，编译器是不会自动生成实例变量的。需要自己处理getter和setter方法
- id<xxx>

NS/CF/CG/CA/UI这些前缀分别是什么含义

- 函数归属于属于cocoa Fundation框架
- 函数归属于属于core Fundation框架
- 函数归属于属于CoreGraphics.frameworks框架
- 函数归属于属于CoreAnimation.frameworks框架
- 函数归属于属于UIKit框架

面向对象都有哪些特征以及你对这些特征的理解。

- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。
- 封装：封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据

和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。

- 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。
- 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

我们说的Objective-C是动态运行时语言是什么意思？(When we call objective c is runtime language what does it mean?)

- 主要是将数据类型的确定由编译时,推迟到了运行时。这个问题其实浅涉及到两个概念,运行时和多态。
- 简单来说,运行时机制使我们直到运行时才去决定一个对象的类别,以及调用该类别对象指定方法。
- 多态:不同对象以自己的方式响应相同的消息的能力叫做多态。
- 意思就是假设生物类(life)都拥有一个相同的方法-eat;那人类属于生物,猪也属于生物,都继承了life后,实现各自的eat,但是调用是我们只需调用各自的eat方法。也就是不同的对象以自己的方式响应了相同的消息(响应了eat这个选择器)。因此也可以说,运行时机制是多态的基础.

readwrite, readonly, assign, retain, copy, nonatomic属性的作用?

- readwrite 是可读可写特性;需要生成getter方法和setter方法;
- readonly 是只读特性 只会生成getter方法 不会生成setter方法 ,不希望属性在类外改变;
- assign 是赋值特性,setter方法将传入参数赋值给实例变量;仅设置变量时; assign用于简单数据类型,如NSInteger,double,bool;
- retain 表示持有特性,setter方法将传入参数先保留,再赋值,传入参数的引用计数retaincount会+1;
- copy 表示赋值特性,setter方法将传入对象复制一份;需要完全一份新的变量时;
- nonatomic 非原子操作,决定编译器生成的setter getter是否是原子操作;
- atomic表示多线程安全,一般使用 nonatomic。

简述NotificationCenter、KVC、KVO、Delegate?并说明它们之间的区别?(重点)

- KVO (Key-Value- Observing) : 一对多, 观察者模式,键值观察机制, 它提供了观察某一属性变化的方法, 极大简化了代码。
- KVC(Key-Value-Coding): 是键值编码, 一个对象在调用setValue的时候,
 - 检查是否存在相应key的set方法, 存在就调用set方法。
 - set方法不存在, 就查找_key的成员变量是否存在, 存在就直接赋值。
 - 如果_key没找到, 就查找相同名称的key, 存在就赋值。
 - 如果没有就调用valueForUndefinedkey和setValue: forUndefinedKey。
- Delegate: 通常发送者和接收者的关系是直接的一对一的关系。
 - 代理的目的是改变或传递控制链。允许一个类在某些特定时刻通知到其他类, 而不需要获取到那些类的指针。
 - 可以减少框架复杂度。消息的发送者(sender)告知接收者(receiver)某个事件将要发生, delegate同意然后发送者响应事件, delegate机制使得接收者可以改变发送者的行为。
- Notification: 观察者模式, 通常发送者和接收者的关系是间接的多对多关系。消息的发送者告知接收者事件已经发生或者将要发送, 仅此而已, 接收者并不能反过来影响发送者的行为。
- 区别
 - 效率肯定是delegate比NSNotification高。
 - delegate方法比notification更加直接, 需要关注返回值, 所以delegate方法往往包含should这个很传神的词。相反的, notification最大的特色就是不关心结果。所以notification往往用did这个词汇。
 - 两个模块之间联系不是很紧密, 就用notification传值, 例如多线程之间传值用notificaiton。
 - delegate只是一种较为简单的回调, 且主要用在一个模块中, 例如底层功能完成了, 需要把一些值传到上层去, 就事先把上层的函数通过delegate传到底层, 然后在底层call这个delegate, 它们都在一个模块中, 完成一个功能, 例如说 NavController 从 B 界面到A 点返回按钮(调用popViewController方法)可以用delegate比较好。

懒加载(What is lazy loading ?)

- 就是懒加载,只在用到的时候才去初始化。也可以理解成延时加载。我觉得最好也最简单的一个例子就是tableView中图片的加载显示了,一个延时加载,避免内存过高,一个异步加载,避免线程堵塞提高用户体验

OC有多继承吗?没有的话可以用什么方法替代?

- 多继承即一个子类可以有多个父类,它继承了多个父类的特性。
- Object-c的类没有多继承,只支持单继承,如果要实现多继承的话,可以通过类别和协议的方式来实现。
- protocol (协议) 可以实现多个接口,通过实现多个接口可以完成多继承;
- Category (类别) 一般使用分类,用Category去重写类的方法,仅对本Category有效,不会影响到其他类与原有类的关系。

分别描述类别(categories)和延展(extensions)是什么?以及两者的区别?继承和类别在实现中有何区别?为什么Category只能为对象添加方法,却不能添加成员变量?

- 类别: 在没有原类.m文件的基础上,给该类添加方法;
- 延展:一种特殊形式的类别,主要在一个类的.m文件里声明和实现延展的作用,就是给某类添加私有方法或是私有变量。
- 两个的区别:
 - 延展可以添加属性并且它添加的方法是必须要实现的。延展可以认为是一个私有的类目。
 - 类别可以在不知道,不改变原来代码的情况下往里面添加新的方法,只能添加,不能删除修改。
 - 并且如果类别和原来类中的方法产生名称冲突,则类别将覆盖原来的方法,因为类别具有更高的优先级。
 - 继承可以增加,修改删除方法,添加属性。
- Category只能为对象添加方法,却不能添加成员变量的原因:如果可以添加成员变量,添加的成员变量没有办法初始化

Objective-C有私有方法么?私有变量呢?如多没有的话,有没有什么代替的方法?

- objective-c类里面的方法只有两种,静态方法和实例方法.但是可以通过把方法的声明和定义都放在.m文件中来实现一个表面上的私有方法。有私有变量,可以通过@private来修饰,或者把声明放到.m文件中。在Objective-C中,所有实例变量默认都是私有的,所有实例方法默认都是公有的

include与#import的区别? #import与@class的区别?

- import指令是Object-C针对#include的改进版本, #import确保引用的文件只会被引用一次,这样你就不会陷入递归包含的问题中。
- import与@class二者的区别在于:
 - import会链入该头文件的全部信息,包括实例变量和方法等;而@class只是告诉编译器,其后面声明的名称是类的名称,至于这些类是如何定义的,暂时不用考虑。
 - 在头文件中一般使用@class来声明这个名称是类的名称,不需要知道其内部的实体变量和方法.

- 而在实现类里面，因为会用到这个引用类的内部的实体变量和方法，所以需要使用#import来包含这个被引用类的头文件。
- 在编译效率方面，如果你有100个头文件都#import了同一个头文件，或者这些文件是依次引用的，如A->B, B->C, C->D这样的引用关系。当最开始的那个头文件有变化的话，后面所有引用它的类都需要重新编译，如果你的类有很多的话，这将耗费大量的时间。而是用@class则不会。
- 如果有循环依赖关系，如:A->B, B->A这样的相互依赖关系，如果使用#import来相互包含，那么就会出现编译错误，如果使用@class在两个类的头文件中相互声明，则不会有编译错误出现。

浅复制(拷贝)和深复制的区别? (Difference between shallow copy and deep copy?)

- 浅复制(copy): 只复制指向对象的指针，而不复制引用对象本身。
- 深复制(mutableCopy): 复制引用对象本身。深复制就好理解了,内存中存在了两份独立对象本身,当修改A时,A_copy不变。

类变量的@protected,@private,@public,@package声明各有什么含义?

变量的作用域不同。

- @protected 该类和子类中访问，是默认的;
- @private 只能在本类中访问;
- @public 任何地方都能访问;
- @package 本包内使用，跨包不可以

Objective-C与C、C++之间的联系和区别?

- Objective-C和C++都是C的面向对象的超集。
- Object与C++的区别主要点：Objective-C是完全动态的，支持在运行时动态类型决议(dynamic typing)，动态绑定(dynamic binding)以及动态装载(dynamic loading)；而C++是部分动态的，编译时静态绑定，通过嵌入类(多重继承)和虚函数(虚表)来模拟实现。
- Objective-C 在语言层次上支持动态消息转发，其消息发送语法为 [object function]；而且C++ 为 object->function()。两者的语义也不同，在 Objective-C 里是说发送消息到一个对象上，至于这个对象能不能响应消息以及是响应还是转发消息都不会 crash；而在 C++ 里是说对象进行了某个操作，如果对象没有这个操作的话，要么编译会报错(静态绑定)，要么程序会 crash 掉的(动态绑定)。

目标-动作机制

- 目标是动作消息的接收者。一个控件，或者更为常见的是它的单元，以插座变量的形式保有其动作消息的目标。

- 动作是控件发送给目标的消息，或者从目标的角度看，它是目标为了响应动作而实现的方法。程序需要某些机制来进行事件和指令的翻译。这个机制就是目标-动作机制。

Objective-C优点和缺点

- 优点:1.Cateogies 2.Posing 3.动态识别 4.指标计算 5.弹性讯息传递 6.不是一个过度复杂的C衍生语言 7.Objective-C与C++可混合编程
- 缺点:1.不支持命名空间 2.不支持运算符重载 3.不支持多重继承 4.使用动态运行时类型,所有的方法都是函数调用,所以很多编译时优化方法都用不到。(如内联函数等),性能低劣。

C语言的函数调用和oc的消息机制有什么区别？

- 对于C语言，函数的调用在编译的时候会决定调用哪个函数。编译完成之后直接顺序执行。
- OC的函数调用成为消息发送。属于动态调用过程。在编译的时候并不能决定真正调用哪个函数（事实证明，在编译阶段，OC可以调用任何函数，即使这个函数并未实现，只要申明过就不会报错。而C语言在编译阶段就会报错）。只有在真正运行的时候才会根据函数的名称找到对应的函数来调用。

什么是谓词

谓词就是通过NSPredicate给定的逻辑条件作为约束条件，完成对数据的筛选。

//定义谓词对象，谓词对象中包含了过滤条件

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"age<%d",30];
```

//使用谓词条件过滤数组中的元素，过滤之后返回查询的结果

```
NSArray *array = [persons filteredArrayUsingPredicate:predicate];
```

//可以使用&&进行多条件过滤

```
predicate = [NSPredicate predicateWithFormat:@"name='l' && age>40"];
```

```
array = [persons filteredArrayUsingPredicate:predicate];
```

//包含语句的使用

```
predicate = [NSPredicate predicateWithFormat:@"self.name IN {'1','2','4'} || self.age IN{30,40}"];
```

//指定字符开头和指定字符结尾，是否包含指定字符

//name以a开头的

```
predicate = [NSPredicate predicateWithFormat:@"name BEGINSWITH 'a'"];  
//name以ba结尾的  
  
predicate = [NSPredicate predicateWithFormat:@"name ENDSWITH 'ba'"];  
//name中包含字符a的  
  
predicate = [NSPredicate predicateWithFormat:@"name CONTAINS 'a'"];  
//like进行匹配多个字符  
  
//name中只要有s字符就满足条件  
  
predicate = [NSPredicate predicateWithFormat:@"name like 's'"];  
//?代表一个字符，下面的查询条件是：name中第二个字符是s的  
  
predicate = [NSPredicate predicateWithFormat:@"name like '?s'"];
```

C与OC混用

处理.m可以识别c和oc，.mm可以识别c c++ oc 但是cpp只能用c/c++

atomic和nonatomic的区别

- atomic提供多线程安全，防止读写未完成的时候被另外一个线程读写，造成数据错误。
- nonatomic在自己管理内存的环境中，解析的访问器保留并自动释放返回值，若指定了nonatomic，那么访问器只是简单的返回这个值。

常见的oc数据类型哪些，和c的基本类型有啥区别

- 常见的：NSInteger CGFloat NSString NSNumber NSArray NSDate
- NSInteger根据32或者64位系统决定本身是int还是long
- CGFloat根据32或者64位系统决定本身是float还是double
- NSString NSNumber NSArray NSDate都是指针类型的对象，在堆中分配内存，c语言中的char int 等都是在栈中分配空间

id和nil代表什么

- id类型的指针可以指向任何OC对象
- nil代表空值（空指针的值，0）

nil和NULL的区别？

- 从oc的官方语法上看，nil表示对象的指针 即对象的引用为空
- null表示指向基础数据类型变量 即c语言变量的指针为空
- 在非arc中 两个空可以互换，但是在arc中 普通指针和对象引用被严格限制，不能互换

nil、Nil、NULL和NSNull区别

- nil和C语言的NULL相同，在objc/objc.h中定义。nil表示Objective-C对象的值为空。在C语言中，指针的空值用NULL表示。在Objective-C中，nil对象调用任何方法表示什么也不执行，也不会崩溃。
- Nil:那么对于我们Objective-C开发来说，Nil也就代表((void *)0)。但是它是用于代表空类的。比如：Class myClass = Nil;
- NULL: 在C语言中，NULL是无类型的，只是一个宏，它代表空。这就是在C/C++中的空指针。对于我们Objective-C开发来说，NULL就表示((void*)0).
- NSNull:NSNull是继承于NSObject的类型。它是很特殊的类，它表示是空，什么也不存储，但是它却是对象，只是一个占位对象。使用场景就不一样了，比如说服务端接口中让我们在值为空时，传空。NSDictionary *parameters = @{@"arg1": @"value1",@"arg2" : arg2.isEmpty ? [NSNull null] : arg2};
- NULL、nil、Nil这三者对于Objective-C中值是一样的，都是(void *)0，那么为什么要区分呢？又与NSNull之间有什么区别：
 - NULL是宏，是对于C语言指针而使用的，表示空指针
 - nil是宏，是对于Objective-C中的对象而使用的，表示对象为空
 - Nil是宏，是对于Objective-C中的类而使用的，表示类指向空
 - NSNull是类类型，是用于表示空的占位对象，与JS或者服务端的null类似的含意

向一个nil对象发送消息会发生什么？

- 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。
- 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)
- 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。

-如果方法返回值为结构体，正如在《Mac OS X ABI 函数调用指南》，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。其他的结构体数据类型将不是用0填充的。

- 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。**self.和self->的区别**
- self.是调用get或者set方法
- self是当前本身，是一个指向当前对象的指针
- self->是直接访问成员变量

类方法和实例方法的本质区别和联系

类方法	实例方法
属于类对象	属于实例对象
只能类对象调用	实例对象调用
self是类对象	self是实例对象
类方法可以调用其他类方法	实例方法可以调用实例方法
类方法不能访问成员变量	实例方法可以访问成员变量
类方法不能直接调用对象方法	实例方法可以调用类方法

_block/weak修饰符区别

- _block在arc和mrc环境下都能用，可以修饰对象，也能修饰基本数据类型
- _weak只能在arc环境下使用，只能修饰对象(NSString)，不能修饰基本数据类型(int)
- _block对象可以在block中重新赋值，_weak不行。

写一个NSString类的实现

```
NSString *str = [[NSString alloc] initWithCString:nullTerminatedCString encoding:encoding];
```

为什么标准头文件都有类似以下的结构？

```
ifndef __INCvxWorksh  
define __INCvxWorksh  
ifdef __cplusplus  
extern "C" {  
endif  
ifdef __cplusplus  
}  
endif  
endif
```

显然，头文件中的编译宏“#ifndef INCvxWorksh、#define INCvxWorksh、#endif”的作用是防止该头文件被重复引用

init和initWithobject区别（语法）？

- 后者给属性赋值

@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：

```
@property = ivar (实例变量) + getter (取方法) + setter (存方法)
```

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

ivar、getter、setter如何生成并添加到类中：

这是编译器自动合成的，通过@synthesize关键字指定，若不指定，默认为

@synthesize propertyName = _propertyName; 若手动实现了getter/setter方法，则不会自动合成。

现在编译器已经默认为我们添加@synthesize propertyName = _propertyName; 因此不再需要手动添加了，除非你真的要改成员变量名。

生成getter方法时，会判断当前属性名是否有_，比如声明属性为@property (nonatomic, copy) NSString * _name; 那么所生成的成员变量名就会变成__name，如果我们要手动生成getter方法，就要判断是否以_开头了。

不过，命名都要有规范，是不允许声明属性是使用_开头的，不规范的命名，在使用runtime时，会带来很多的不方便的。

这个写法会出什么问题：@property (copy) NSMutableArray *array;

- 没有指明为nonatomic，因此就是atomic原子操作，会影响性能。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。在我们的应用程序中，几乎都是使用nonatomic来声明的，因为使用atomic并不能保证绝对的线程安全，对于要绝对保证线程安全的操作，还需要使用更高级的方式来处理，比如 NSSpinLock、@synchronized等
- 因为使用的是copy，所得到的实际是NSArray类型，它是不可变的，若在使用中使用了增、删、改操作，则会crash

@protocol和category中如何使用 @property

- 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
- 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：
 - objc_setAssociatedObject
 - objc_getAssociatedObject

@property中有哪些属性关键字？

1. 原子性 (atomic, nonatomic)
2. 读写 (readwrite, readonly)
3. 内存管理 (assign, strong, weak, unsafe_unretained, copy)
4. getter、setter

isa指针问题

- isa: 是一个Class类型的指针. 每个实例对象有个isa的指针,他指向对象的类, 而Class里也有个isa的指针, 指向meteClass(元类)。元类保存了类方法的列表。当类方法被调用时, 先会从本身查找类方法的实现, 如果没有, 元类会向他父类查找该方法。同时注意的是: 元类 (meteClass) 也是类, 它也是对象。元类也有isa指针,它的isa指针最终指向的是一个根元类(root meteClass).根元类的isa指针指向本身, 这样形成了一个封闭的内循环。

如何访问并修改一个类的私有属性?

- 一种是通过KVC获取
- 通过runtime访问并修改私有属性

如何为 Class 定义一个对外只读对内可读写的属性?

在头文件中将属性定义为readonly,在.m文件中将属性重新定义为readwrite

Objective-C 中, meta-class 指的是什么?

meta-class 是 Class 对象的类,为这个Class类存储类方法,当一个类发送消息时,就去这个类对应的meta-class中查找那个消息,每个Class都有不同的meta-class,所有的meta-class都使用基类的meta-class(假如类继承NSObject,那么他所对应的meta-class也是NSObject)作为他们的类

Objective-C 的class是如何实现的? Selector是如何被转化为 C 语言的函数调用的?

- 当一个类被正确的编译过后, 在这个编译成功的类里面, 存在一个变量用于保存这个类的信息。我们可以通过[NSSelectorFromString]或[obj class]。这样的机制允许我们在程序执行的过程当中, 可以Class来得到对象的类, 也可以在程序执行的阶段动态的生成一个在编译阶段无法确定的一个对象。 (isa指针)
- @selector()基本可以等同C语言的中函数指针,只不过C语言中, 可以把函数名直接赋给一个函数指针, 而Objective-C的类不能直接应用函数指针, 这样只能做一个@selector语法来取.

```
@interface foo  
  
-(int)add:int val;  
@end  
  
SEL class_func ; //定义一个类方法指针  
class_func = @selector(add:int);
```

- @selector是查找当前类的方法, 而[object @selector(方法名:方法参数..)] ;是取object对应类的相应方法.
- 查找类方法时, 除了方法名,方法参数也查询条件之一.

- 可以用字符串来找方法 SEL 变量名 = NSSelectorFromString(方法名字的字符串);
- 可以运行中用SEL变量反向查出方法名字字符串。 NSString *变量名 = NSStringFromSelector(SEL参数);
- 取到selector的值以后，执行seletor。 SEL变量的执行.用performSeleccor方法来执行. [对象 performSelector:SEL变量 withObject:参数1 withObject:参数2];

对于语句 **NSString *obj = [[NSData alloc] init];**， 编译时和运行时 obj分别是什么类型？

- 编译时是NSString类型，运行时是NSData类型。

@synthesize和@dynamic分别有什么作用？

- @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@synthesize var = _var;
- @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。
- @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。

NSString 的时候用copy和strong的区别？

oc中NSString为不可变字符串的时候，用copy和strong都是只分配一次内存，但是如果用copy的时候，需要先判断字符串是否是不可变字符串，如果是不可变字符串，就不再分配空间，如果是可变字符串才分配空间。如果程序中用到NSString的地方特别多，每一次都要先进行判断就会耗费性能，影响用户体验，用strong就不会再进行判断，所以，不可变字符串可以直接用strong。

NSArray、NSSet、NSDictionary与NSMutableArray、 NSMutableSet、NSMutableDictionary的特性和作用（遇到copy修饰产生的变化）

- 特性：
- NSArray表示不可变数组，是有序元素集，只能存储对象类型，可通过索引直接访问元素，而且元素类型可以不一样，但是不能进行增、删、改操作； NSMutableArray是可变数组，能进行增、删、改操作。通过索引查询值很快，但是插入、删除等效率很低。

- NSSet表示不可变集合，具有确定性、互异性、无序性的特点，只能访问而不能修改集合；NSMutableSet表示可变集合，可以对集合进行增、删、改操作。集合通过值查询很快，插入、删除操作极快。
- NSDictionary表示不可变字典，具有无序性的特点，每个key对应的值是唯一的，可通过key直接获取值；NSMutableDictionary表示可变字典，能对字典进行增、删、改操作。通过key查询值、插入、删除值都很快。
- 作用：
- 数组用于处理一组有序的数据集，比如常用的列表的dataSource要求有序，可通过索引直接访问，效率高。
- 集合要求具有确定性、互异性、无序性，在iOS开发中是比较少使用到的，笔者也不清楚如何说明其作用
- 字典是键值对数据集，操作字典效率极高，时间复杂度为常量，但是值是无序的。在ios中，常见的JSON转字典，字典转模型就是其中一种应用。

请把字符串2015-04-10格式化日期转为NSDate类型

```
NSString *timeStr = @"2015-04-10";
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
formatter.dateFormat = @"yyyy-MM-dd";
formatter.timeZone = [NSTimeZone defaultTimeZone];
NSDate *date = [formatter dateFromString:timeStr];
// 2015-04-09 16:00:00 +0000
NSLog(@"%@", date);
```

在一个对象的方法里：self.name=@object;和name=@object有什么不同

- 这是老生常谈的话题了，实质上就是问setter方法赋值与成员变量赋值有什么不同。通过点语法self.name实质上就是[self setName:@object];。而name这里是成员变量，直接赋值。
一般来说，在对象的方法里成员变量和方法都是可以访问的，我们通常会重写Setter方法来执行某些额外的工作。比如说，外部传一个模型过来，那么我会直接重写Setter方法，当模型传过来时，也就是意味着数据发生了变化，那么视图也需要更新显示，则在赋值新模型的同时也去刷新UI。这样也不用再额外提供其他方法了。

怎样使用performSelector传入3个以上参数，其中一个为结构体

```
- (id)performSelector:(SEL)aSelector;
- (id)performSelector:(SEL)aSelector withObject:(id)object;
- (id)performSelector:(SEL)aSelector withObject:(id)object1
withObject:(id)object2;
```

因为系统提供的performSelector的api中，并没有提供三个参数。因此，我们只能传数组或者字典，但是数组或者字典只有存入对象类型，而结构体并不是对象类型，那么怎么办呢？没有办法，我们只能通过对象放入结构作为属性来传过去了：

```
ypedef struct HYBStruct {
```

```

int a;
int b;
} *my_struct;

@interface HYBObject : NSObject

@property (nonatomic, assign) my_struct arg3;
@property (nonatomic, copy) NSString *arg1;
@property (nonatomic, copy) NSString *arg2;

@end

@implementation HYBObject

// 在堆上分配的内存，我们要手动释放掉
- (void)dealloc {
    free(self.arg3);
}

@end

```

测试：

```

my_struct str = (my_struct)(malloc(sizeof(my_struct)));
str->a = 1;
str->b = 2;
HYBObject *obj = [[HYBObject alloc] init];
obj.arg1 = @"arg1";
obj.arg2 = @"arg2";
obj.arg3 = str;

[self performSelector:@selector(call:) withObject:obj];

// 在回调时得到正确的数据的
- (void)call:(HYBObject *)obj {
    NSLog(@"%@", obj.arg3->a, obj.arg3->b);
}

```

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？

实际上，编译器在编译时会转换成objc_msgSend，大概会像这样：

```
((void (*)(id, SEL))(void)objc_msgSend)((id)obj,
sel_registerName("foo"));
```

也就是说，[obj foo];在objc动态编译时，会被转换为：objc_msgSend(obj, @selector(foo));这样的形式，但是需要根据具体的参数类型及返回值类型进行相应的类型转换。

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

// 输出
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。

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？

- 两者都可以。

什么时候使用NSMutableArray，什么时候使用NSArray？

- 当数组在程序运行时，需要不断变化的，使用NSMutableArray，当数组在初始化后，便不再改变的，使用NSArray。需要指出的是，使用NSArray只表明的是该数组在运行时不发生改变，即不能往NSAarry的数组里新增和删除元素，但不表明其数组内的元素的内容不能发生改变。NSArray是线程安全的，NSMutableArray不是线程安全的，多线程使用到NSMutableArray需要注意。

类NSObject的那些方法经常被使用？

- NSObject是Objetive-C的基类，其由NSObject类及一系列协议构成。
- 其中类方法alloc、class、description 对象方法init、dealloc、-performSelector:withObject:afterDelay:等经常被使用

什么是简便构造方法？

- 简便构造方法一般由CocoaTouch框架提供，如NSNumber的 + numberWithBool: + numberWithChar: + numberWithDouble: + numberWithFloat: + numberWithInt:
- Foundation下大部分类均有简便构造方法，我们可以通过简便构造方法，获得系统给我们创建好的对象，并且不需要手动释放。

什么是构造方法，使用构造方法有什么注意点。

什么是构造方法：构造方法是对象初始化并一个实例的方法。

构造方法有什么用：一般在构造方法里 对类进行一些初始化操作

注意点：方法开头必须以init开头，接下来名称要大写 例如 initWithName , initLayout

创建一个对象需要经过那三个步骤？

- 开辟内存空间
- 初始化参数
- 返回内存地址值

Get方法的作用是什么？

Get方法的作用：为调用者返回对象内部的成员变量

Set方法的作用是什么？ Set方法的好处？

- Set方法的作用：为外界提供一个设置成员变量值的方法。
- Set方法的好处：
 - 不让数据暴露在外，保证了数据的安全性
 - 对设置的数据进行过滤

结构体当中能定义oc对象吗？

不能，因为结构体当中只能是类型的声明不能进行分配空间

点语法本质是什么,写一个点语法的例子,并写上注释

- 点语法的本质是方法的调用，而不是访问成员变量，当使用点语法时，编译器会自动展开成相应的方法。切记点语法的本质是转换成相应的set和get方法，如果没有set和get方法，则不能使用点语法。
- 例如有一个Person类 通过@property 定义了name和age属性,再提供了一个run方法。
- Person *person = [Person new];
- person.name=@”itcast”; //调用了person的setName方法
- int age = person.age; // 调用了person的age方法
- person.run //调用了person的run方法

id类型是什么,instancetype是什么,有什么区别?

- id类型：万能指针，能作为参数，方法的返回类型。

- `instancetype`: 只能作为方法的范围类型，并且返回的类型是当前定义类的类类型。

成员变量名的命名以下划线开头的好处?

- 与get方法的方法名区分开来；
- 可以和一些其他的局部变量区分开来，下划线开头的变量，通常都是类的成员变量。

这段代码有什么问题吗:

```
@implementation Person
- (void)setAge:(int)newAge {
self.age = newAge; }
@end
会死循环,会重复调用自己!self.age 改为_age即可;
并且书写不规范:setter方法中的newAge应该为age
```

截取字符串”20 | http://www.baidu.com”中,”|”字符前面和后面的数据,分别输出它们。

```
NSString * str = @"20 | http://www.baidu.com";
NSArray *array = [str componentsSeparatedByString:@" | "]; //这是分别
输出的截取后的字符串
for (int i = 0; i<[array count]; ++i) { NSLog(@"%@",i,[array
objectAtIndex:i]); }
```

写一个完整的代理,包括声明,实现

```
//创建
@protocol MyDelagate
@required
-(void)eat:(NSString *)foodName;
@optional
-(void)run;
@end

//声明
@interface person: NSObject< MyDelagate>

//实现
@implementation person
-(void)eat:(NSString *)foodName;
{ NSLog(@"吃:%@!",foodName);}
-(void)run
```

```
{ NSLog(@"run!"); }
@end
```

isKindOfClass、isMemberOfClass、selector作用分别是什么

- isKindOfClass,作用是,某个对象属于某个类型或者继承自某类型
- isMemberOfClass:某个对象确切属于某个类型
- selector:通过方法名,获取在内存中的函数的入口地址

请分别写出SEL、id、@的意思？

- SEL是“selector”的一个类型,表示一个方法的名字-----就是一个方法的入口地址
- id是一个指向任何一个继承了Object(或者NSObject)类的对象。需要注意的是id是一个指针,所以在使用id 的时候不需要加*。
- @:OC中的指令符

unsigned int 和int 有什么区别。假设int长度为65535， 请写出 unsigned int与 int的取值范围

int:基本整型，当字节数为2时 取值范围为-32768~32767，当字节数为4时 取值范围负的2的31次方 到 2的31次方减1

unsigned int：无符号基本整型，当字节数为2时 取值范围为0~65535，当字节数为4时 取值范围为0到2的32次方减1

Foundation对象与Core Foundation对象有什么区别

- Foundation对象是OC的，Core Foundation对象是C对象
- 数据类型之间的转换
 - ARC:**bridge_retained** (持有对象所有权,F->CF) 、 **bridge_transfer** (释放对象所有权CF->F)
 - 非ARC: **__bridge**

编写一个函数，实现递归删除指定路径下的所有文件。

```
+ (void)deleteFiles:(NSString *)path{
    // 1.判断文件还是目录
    NSFileManager * fileManger = [NSFileManager defaultManager];
    BOOL isDir = NO;
```

```
BOOL isExist = [fileManger fileExistsAtPath:path
isDirectory:&isDir];
if (isExist) {
    // 2. 判断是不是目录
    if (isDir) {
        NSArray * dirArray = [fileManger
contentsOfDirectoryAtPath:path error:nil];
        NSString * subPath = nil;
        for (NSString * str in dirArray) {
            subPath = [path
stringByAppendingPathComponent:str];
            BOOL issubDir = NO;
            [fileManger fileExistsAtPath:subPath
isDirectory:&issubDir];
            [self deleteFiles:subPath];
        }
    }
} else{
    NSLog(@"%@",path);
    [manager removeItemAtPath:filePath error:nil];
}
}else{
    NSLog(@"你打印的是目录或者不存在");
}
}
```

(6) 内存管理

ARC处理原理

ARC是Objective-C编译器的特性，而不是运行时特性或者垃圾回收机制，ARC所做的只不过是在代码编译时为你自动在合适的位置插入release或autorelease，只要没有强指针指向对象，对象就会被释放。

- 前端编译器

前端编译器会为“拥有的”每一个对象插入相应的release语句。如果对象的所有权修饰符是`__strong`，那么它就是被拥有的。如果在某个方法内创建了一个对象，前端编译器会在方法末尾自动插入release语句以销毁它。而类拥有的对象（实例变量/属性）会在dealloc方法内被释放。事实上，你并不需要写dealloc方法或调用父类的dealloc方法，ARC会自动帮你完成一切。此外，由编译器生成的代码甚至会比你自己写的release语句的性能还要好，因为编辑器可以作出一些假设。在ARC中，没有类可以覆盖release方法，也没有调用它的必要。ARC会通过直接使用objc_release来优化调用过程。而对于retain也是同样的方法。ARC会调用objc_retain来取代保留消息。

- ARC优化器

虽然前端编译器听起来很厉害的样子，但代码中有时仍会出现几个对retain和release的重复调用。ARC优化器负责移除多余的retain和release语句，确保生成的代码运行速度高于手动引用计数的代码。

下面关于Objective-C内存管理的描述错误的是

- A 当使用ARC来管理内存时，代码中不可以出现autorelease
- B autoreleasepool 在 drain 的时候会释放在其中分配的对象
- C 当使用ARC来管理内存时，在线程中大量分配对象而不用autoreleasepool则可能会造成内存泄露
- D 在使用ARC的项目中不能使用NSZone

- 参考答案：A
- 理由：ARC只是在大多时候编译自动为我们添加上内存管理的代码，只是我们的源代码看不到而已，但是在编译时，编译器会添加上相关内存管理代码。对于自动释放池，在drain时会将自动释放池中的所有对象的引用计数减一，若引用计数为0，则会自动释放掉其内存。如果在线程中需要大量分配内存，我们理应添加上自动释放池，以防内存泄露。比如在for循环中要分配大量的内存处理数据，那么我们应该在for循环内添加自动释放池，在每个循环后就将内存释放掉，防止内存泄露。在ARC项目中，自然不能手动使用NSZone，也不能调用父类的dealloc。

MRC文件在ARC工程混合编译时，需要在文件的Compiler Flags上添加什么参数

- A -shared B -fno-objc-arc C -fobjc-arc D -dynamic

参考答案：B

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？
- 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
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。
- weak与assign的不同？
- 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。
- assign 可以用非 OC 对象,而 weak 必须用于 OC 对象

调用对象的release 方法会销毁对象吗？

不会, 调用对象的release 方法只是将对象的引用计数器-1, 当对象的引用计数器为0的时候会调用了对象的dealloc 方法才能进行释放对象的内存。

自动释放池常见面试代码

```
for (int i = 0; i < someLargeNumber; ++i)
{
    NSString *string = @"Abc";
    string = [string lowercaseString];
    string = [string stringByAppendingString:@"xyz"];
    NSLog(@"%@", string);
}
```

问：以上代码存在什么样的问题？如果循环的次数非常大时，应该如何修改？

存在问题：问题处在每执行一次循环，就会有一个string加到当前runloop中的自动释放池中，只有当自动释放池被release的时候，自动释放池中的标示了autorelease的这些数据所占用的内存空间才能被释放掉。假设，当someLargeNumber大到一定程度时，内存空间将被耗尽而没有被释放掉，所以就出现了内存溢出的现象。

解决办法1：如果i比较大，可以用@autoreleasepool {}解决，放在for循环外，循环结束后，销毁创建的对象，解决占据栈区内存的问题

解决方法2：如果i玩命大，一次循环都会造成自动释放池被填满，自动释放池放在for循环内，每次循环都将上一次创建的对象release

修改之后：

```

for(int i = 0; i<1000;i++){
NSAutoreleasePool * pool1 = [[NSAutoreleasePool alloc] init];
NSString *string = @"Abc";
string = [string lowercaseString];
string = [string stringByAppendingString:@"xyz"];
 NSLog(@"%@",string);
//释放池
[pool1 drain]; }

```

objective-C对象的内存布局是怎样的？

- 由于Objective-C中没有多继承，因此其内存布局还是很简单的，就是：最前面有个isa指针，然后父类的实例变量存放在子类的成员变量之前

看下面的程序,第一个NSLog会输出什么?这时str的retainCount是多少?第二个和第三个呢?为什么?

```

NSMutableArray* ary = [[NSMutableArray array] retain];
NSString *str = [NSString stringWithFormat:@"test"];
[str retain];
[ary addObject:str];
 NSLog(@"%@",str,[str retainCount]);
[str retain];
[str release];
[str release];
 NSLog(@"%@",str,[str retainCount]);
[ary removeAllObjects];
 NSLog(@"%@",str,[str retainCount]);
str的retainCount创建+1, retain+1, 加入数组自动+1 3
retain+1, release-1, release-1 2
数组删除所有对象, 所有数组内的对象自动-1 1

```

回答person的retainCount值,并解释为什么

Person * per = [[Person alloc] init]; 此时person 的retainCount的值是1
self.person = per;
在self.person 时,如果是assign, person的 retainCount的值不变,仍为1 若是,retain person的retainCount的值加1,变为2
若是,copy person的retainCount值不变,仍为1

什么时候需要在程序中创建内存池?

- 用户自己创建的数据线程，则需要创建该线程的内存池

如果不创建内存池，是否有内存池提供给我们？

- 界面线程维护着自己的内存池，用户自己创建的数据线程，则需要创建该线程的内存池

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。

- `objc_autoreleasepoolPush`
- `objc_autoreleasepoolPop`
- `objc_autorelease`

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

objc使用什么机制管理对象内存？

- 通过引用计数器(retainCount)的机制来决定对象是否需要释放。每次runloop完成一个循环的时候，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。

为什么要进行内存管理？

- 因为移动设备的内存极其有限，当一个程序所占内存达到一定值时，系统会发出内存警告。当程序达到更大的值时，程序会闪退，影响用户体验。为了保证程序的运行流畅，必须进行内存管理。

内存管理的范围？

- 管理所有继承自NSObject的对象，对基本数据类型无效。是因为对象和其他数据类型在系统中存储的空间不一样，其他局部变量主要存储在栈区（因为基本数据类型占用的存储空间是固定的，一般存放于栈区），而对象存储于堆中，当代码块结束时，这个代码块所涉及到的所有局部变量会自动弹栈清空，指向对象的指针也会被回收，这时对象就没有指针指向，但依然存在于堆内存中，造成内存泄露。

objc使用什么机制管理对象内存(或者内存管理方式有哪些)? (重点)

- MRC(manual retain-release)手动内存管理
- ARC(automatic reference counting)自动引用计数
- Garbage collection (垃圾回收)。但是iOS不支持垃圾回收，ARC作为LLVM3.0编译器的一项特性，在iOS5.0 (Xcode4) 版本后推出的。
- ARC的判断准则，只要没有强指针指向对象，对象就会被释放。

iOS是如何管理内存的?

- 这个问题的话上一个问题也提到过,讲下block的内存管理,ARC下的黄金法则就行。
- 这里说下swift里的内存管理:
delegate照样weak修饰,闭包前面用[weak self],swift里的新东西,unowned,举例,如果self在闭包被调用的时候可能为空,则用weak,反之亦然,如果为空时使用了unowned,程序会崩溃,类似访问了悬挂指针,在oc中类似于unsafe_unretained,类似assign修饰了oc对象,对象被销毁后,被unowned修饰的对象不会为空,但是unowned访问速度更快,因为weak需要unwarp后才能使用

内存管理的原则

- 只要还有人在使用这个对象,那么这个对象就不会被回收
- 只有你想使用这个对象,那么就应该让这个对象的引用计数器加1
- 当你不想使用这个对象时,应该让对象的引用计数器减1
- 谁创建,就由谁来release
 - 如果你通过alloc, new, copy 来创建一个对象,当你不想用这个对象的时候就必须调用release 或者autorelease 让引用计数器减1
 - 不是你创建的就不用你负责 release
- 谁retain 谁release
 - 只要你调用了retain ,无论这个对象如何生成,都需要调用release
- 总结:
有加就应该有减,曾让某个计数器加1,就应该让其在最后减1

内存管理研究的对象:

- 野指针:指针变量没有进行初始化或指向的空间已经被释放。
 - 使用野指针调用对象方法,会报异常,程序崩溃。
 - 通常再调用完release方法后,把保存对象指针的地址清空,赋值为nil,找oc中没有空指针异常,所以[nil retain]调用方法不会有异常。
- 内存泄露
 - 如 Person * person = [Person new];(对象提前赋值nil或者清空)在栈区的person已经被释放,而堆区new产生的对象还没有释放,就会造成内存泄露
 - 在MRC手动引用计数器模式下,造成内存泄露的情况
 - 没有配对释放,不符合内存管理原则
 - 对象提前赋值nil或者清空,导致release不起作用。
- 僵尸对象 : 堆中已经被释放的对象(retainCount = 0)
- 空指针 : 指针赋值为空,nil

如何判断对象已经被销毁

- 重写dealloc方法，对象销毁时，会调用，重写时一定要[super dealloc]
retainCount = 0，使用retain能否复活对象
- 已经被释放的对象无法复活

对象与对象之间存在的关系

1. 继承关系
2. 组合关系（是一种强烈的包含关系）
3. 依赖关系(对象作为方法参数传递)

对象的组合关系中，确保成员变量不被提前释放？

- 重写set方法，在set方法中，retain该对象。

成员变量的对象，在哪里配对释放？

- dealloc中释放

对象组合关系中，内存泄露有哪几种情况？

- set方法没有retain对象
- 没有release旧对象
- 没有判断向set方法中传入的是否为同一个对象

正确重写set方法

- 判断是否为同一对象
- release旧对象
- retain新对象

分别描述内存管理要点、autorelease、release、
NSAutoreleasePool?并说明autorelease是什么时候被release的?
简述什么时候由你负责释放对象,什么时候不由你释放?
[NSAutoreleasePool release]和[NSAutoreleasePool drain]有什么区别?

- 内存管理要点:Objective-C 使用引用计数机制(retainCount)来管理内存。
- 内存每被引用一次,该内存的引用计数+1,每被释放一次引用计数-1。

- 当引用计数 = 0 的时候,调用该对象的 dealloc 方法,来彻底从内存中删除该对象。
- alloc,allocWithZone,new(带初始化)时:该对象引用计数 +1;
- retain:手动为该对象引用计数 +1;
- copy:对象引用计数 +1;//注意copy的OC数据类型是否有mutable,如有为深拷贝,新对象计数为1,如果没有,为浅拷贝,计数+1
- mutableCopy:生成一个新对象,新对象引用计数为 1;
- release:手动为该对象引用计数 -1;
- autorelease:把该对象放入自动释放池,当自动释放池释放时,其内的对象引用计数 -1。
- NSAutoreleasePool: NSAutoreleasePool是通过接收对象向它发送的autorelease消息,记录该对象的release消息,当自动释放池被销毁时,会自动向池中的对象发送release消息。
- autorelease 是在自动释放池被销毁,向池中的对象发送release
只能释放自己拥有的对象。
- 区别是:在引用计数环境下(在不使用ARC情况下),两者基本一样,在GC(垃圾回收制)环境下,release 是一个no-op(无效操作),所以无论是不是GC都使用drain
- 面试中内存管理,release和autorelease的含义?这里尤其要强调下 autorelease,它引申出自动释放池,也能引申出Run loop!

自动释放池是什么,如何工作 ?

- 什么是自动释放池: 用来存储多个对象类型的指针变量
- 自动释放池对池内对象的作用: 存入池内的对象, 当自动释放池被销毁时, 会对池内对象全部做一次release操作
- 对象如何加入池中: 调用对象的autorelease方法
- 自动释放池能嵌套使用吗: 能
- 自动释放池何时被销毁: 简单的看, autorelease的"}"执行完以后。而实际情况是 Autorelease对象是在当前的runloop迭代结束时释放的, 而它能够释放的原因是系统在每个runloop迭代中都加入了自动释放池Push和Pop
- 多次调用对象的autorelease方法会导致: 野指针异常
- 自动释放池的作用: 将对象与自动释放池建立关系, 池子内调用autorelease, 在自动释放池销毁时销毁对象, 延迟release销毁时间

自动释放池什么时候释放?

- 通过Observer监听RunLoop的状态, 一旦监听到RunLoop即将进入睡眠等待状态, 就释放自动释放池 (kCFRunLoopBeforeWaiting)

iPhone OS有没有垃圾回收?autorelease 和垃圾回收制(gc)有什么关系?

- iOS 中没有垃圾回收。autorelease只是延迟释放,gc是每隔一段时间询问程序,看是否有无指针指向的对象,若有,就将它回收。他们两者没有什么关系。

ARC问题

1. 什么是arc机制: 自动引用计数.
2. 系统判断对象是否销毁的依据: 指向对象的强指针是否被销毁
3. arc的本质: 对retainCount计算, 创建+1 清空指针 - 1 或者到达autoreleasepool的大括号 -1
4. arc目的: 不需要程序员关心retain和release操作.
5. 如何解决arc机制下类的相互引用: .h文件中使用@class关键字声明一个类, 两端不能都用强指针, 一端用strong一端用weak

ARC通过什么方式帮助开发者管理内存?

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, 手动去做未必优化得好, 因此直接交给编译器来优化, 相信苹果吧!

开发项目时你是怎么检查内存泄露

- 静态分析 analyze
- instruments工具里面有个leak 可以动态分析

如果在block中多次使用 weakSelf的话, 可以在block中先使用strongSelf, 防止block执行时weakSelf被意外释放

对于非ARC, 将 weak 改用为 block 即可

麻烦你设计个简单的图片内存缓存器 (移除策略是一定要说的)

- 内存缓存是个通用话题, 每个平台都会涉及到。cache算法会影响到整个app的表现。候选人最好能谈下自己都了解哪些cache策略及各自的特点。
- 常见的有FIFO,LRU,LFU等等。由于NSCache的缓存策略不透明, 一些app开发者会选择自己做一套cache机制, 其实并不难。
- FIFO : 新访问的数据插入FIFO队列尾部, 数据在FIFO队列中顺序移动; 淘汰FIFO队列头部的数据;

- LRU：新数据插入到链表头部；每当缓存数据命中，则将数据移到链表头部；当链表满的时候，将链表尾部的数据丢弃；
- LFU：新加入数据插入到队列尾部（因为引用计数为1）；队列中的数据被访问后，引用计数增加，队列重新排序；当需要淘汰数据时，将已经排序的列表最后的数据块删除；

常见的出现内存循环引用的场景有哪些？

- 定时器（NSTimer）：NSTimer经常会被作为某个类的成员变量，而NSTimer初始化时要指定self为target，容易造成循环引用（self->timer->self）。另外，若timer一直处于validate的状态，则其引用计数将始终大于0，因此在不再使用定时器以后，应该先调用invalidate方法
- block的使用：block在copy时都会对block内部用到的对象进行强引用(ARC)或者retainCount增1(非ARC)。在ARC与非ARC环境下对block使用不当都会引起循环引用问题，一般表现为，某个类将block作为自己的属性变量，然后该类在block的方法体里面又使用了该类本身，简单说就是self.someBlock = Type var{[self dosomething];或者self.otherVar = XXX;或者_otherVar = ...};出现循环的原因是：self->block->self或者self->block->_ivar（成员变量）
- 代理（delegate）：在委托问题上出现循环引用问题已经是老生常谈了，规避该问题的杀手锏也是简单到哭，一字诀：声明delegate时请用assign(MRC)或者weak(ARC)，千万别手贱玩一下retain或者strong，毕竟这基本逃不掉循环引用了！

对象添加到通知中心中，当通知中心发通知时，这个对象却已经被释放了，可能会出现什么问题？

- 其实这种只是考查对通知的简单应用。通知是多对多的关系，主要使用场景是跨模块传值。当某对象加入到通知中心后，若在对象被销毁前不将该对象从通知中心中移除，当发送通知时，就会造成崩溃。这是很常见的。所以，在添加到通知中心后，一定要在释放前移除。

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？

- 对于基本数据类型默认关键字是：atomic,readwrite,assign
- 对于普通的Objective-C对象：atomic,readwrite,strong

写一个便利构造器

```
+ (id)Person {
    Person *person = [Person alloc] init];
    return [person autorelease]; 备注：ARC时不用 autorelease
}
```

写出下面程序段的输出结果

```
NSDictionary *dict = [NSDictionary dictionaryWithObject:@"a string"
                                             value forKey:@"akey"];
NSLog(@"%@", [dict objectForKey:@"akey"]);
[dict release];
```

打印输出 a string value, 然后崩溃—原因：便利构造器创建的对象，之后的release，会造成过度释放

请写出以下代码的执行结果

```
NSString * name = [ [ NSString alloc] init ];  
name = @"Habb";  
[ name release];
```

打印输出结果是： Habb，在[name release]前后打印均有输出结果 —会造成内存泄露
---原先指向的区域变成了野指针，之后的释放，不能释放之前创建的区域

写出方法获取ios内存使用情况？

iOS是如何管理内存的？

我相信很多人的回答是内存管理的黄金法则，其实如果我是面试官，我想要的答案不是这样的。我希望的回答是工作中如何处理内存管理的。

参考答案：

- Block内存管理：由于使用block很容易造成循环引用，因此一定要小心内存管理问题。最好在基类controller下重写dealloc，加一句打印日志，表示类可以得到释放。如果出现无打印信息，说明这个类一直得不到释放，表明很有可能是使用block的地方出现循环引用了。对于block中需要引用外部controller的属性或者成员变量时，一定要使用弱引用，特别是成员变量像_testId这样的，很多人都没有使用弱引用，导致内存得不到释放。
- 对于普通所创建的对象，因为现在都是ARC项目，所以记住内存管理的黄金法则就可以解决。

很多内置的类，如tableview的delegate的属性是assign不是retain？

- tableview的代理一般都是它所属的控制器，控制器会对它内部的view进行一次retain操作，而tableview对代理控制器也进行一次retain操作，就会出现循环引用问题。

(7) KVO-KVC

KVC的底层实现?

当一个对象调用setValue方法时，方法内部会做以下操作：

- ①检查是否存在相应key的set方法，如果存在，就调用set方法
- ②如果set方法不存在，就会查找与key相同名称并且带下划线的成员属性，如果有，则直接给成员属性赋值
- ③如果没有找到_key，就会查找相同名称的属性key，如果有就直接赋值
- ④如果还没找到，则调用valueForUndefinedKey:和setValue:forUndefinedKey:方法。

这些方法的默认实现都是抛出异常，我们可以根据需要重写它们。

KVO的底层实现?

- kvo基于runtime机制实现。
- 使用了isa 混写 (isa-swizzling)，当一个对象(假设是person对象，person的类是MYPerson)的属性值(假设person的age)发生改变时，系统会自动生成一个类，继承自MYPerson : NSKVNNotifying_MYPerson，在这个类的setAge方法里面，调用 [super setAge:age] [self willChangeValueForKey:@"age"] 和 [self didChangeValueForKey:@"age"]
,而这两个方法内部会主动调用监听者内部的 - (void)observeValueForKeyPath 这个方法。
- 想要看到NSKVONotifying_MYPerson很简单，在self.person.age = 20；这里打断点，在调试区域就能看到 _person->NSObject->isa=(Class)NSKVONotifying_MYPerson.同时我们在 self.person = [[MYPerson alloc] init];后面打断点，看到_person->NSObject->isa=(Class)MYPerson,由此可见，在添加监听者之后，person类型已经由MYPerson被改变成NSKVONotifying_MYPerson

什么是KVO和KVC?

答：KVC:键 – 值编码 使用字符串直接访问对象的属性。

KVO:键值观察机制，它提供了观察某一属性变化的方法

KVO的缺陷?

KVO是一个对象能够观察另外一个对象的属性的值，并且能够发现值的变化。前面两种模式更加适合一个controller与任何其他的对象进行通信，而KVO更加适合任何类型的对象侦听

另外一个任意对象的改变（这里也可以是controller，但一般不是controller）。这是一个对象与另外一个对象保持同步的一种方法，即当另外一种对象的状态发生改变时，观察对象马上作出反应。它只能用来对属性作出反应，而不会用来对方法或者动作作出反应。

优点：

- 1.能够提供一种简单的方法实现两个对象间的同步。例如：model和view之间同步；
- 2.能够对非我们创建的对象，即内部对象的状态改变作出响应，而且不需要改变内部对象（SKD对象）的实现；
- 3.能够提供观察的属性的最新值以及先前值；
- 4.用key paths来观察属性，因此也可以观察嵌套对象；
- 5.完成了对观察对象的抽象，因为不需要额外的代码来允许观察值能够被观察

缺点：

- 1.我们观察的属性必须使用strings来定义。因此在编译器不会出现警告以及检查；
- 2.对属性重构将导致我们的观察代码不再可用；
- 3.复杂的“IF”语句要求对象正在观察多个值,这是因为所有的观察代码通过一个方法来指向；
- 4.当释放观察者时需要移除观察者。

(8) Block

Block底层原理实现

- 首先我们来看四个函数

```
void test1()
{
    int a = 10;

    void (^block)() = ^{
        NSLog(@"a is %d", a);
    };

    a = 20;

    block(); // 10
}

void test2()
{
    __block int a = 10;

    void (^block)() = ^{
        NSLog(@"a is %d", a);
    };

    a = 20;

    block(); // 20
}

void test3()
{
    static int a = 10;

    void (^block)() = ^{
        NSLog(@"a is %d", a);
    };

    a = 20;

    block(); // 20
}

int a = 10;
void test4()
{
    void (^block)() = ^{
        NSLog(@"a is %d", a);
    };
}
```

```

    a = 20;

    block(); //20
}

```

- 造成这样的原因是：传值和传址。为什么说会有传值和传址，把.m编译成c++代码。得到.cpp文件，我们来到文件的最后，看到如下代码

```

struct __test1_block_impl_0 {
    struct __block_impl impl;
    struct __test1_block_desc_0* Desc;
    int a;
    __test1_block_impl_0(void *fp, struct __test1_block_desc_0*
Desc, int _a, int flag=0): a(_a){
        impl.isa = &_NSConcreteStackBlock;
        impl.Flags = flags;
        impl.FuncPtr = fp;
        Desc = desc;
    }
};

```

```

static void __test1_block_func_0(struct __test1_block_im_0
*cself)
{
    int a = __cself->a;
    NSLog(a); //这里就是打印a的值，代码太长，而且没意义，我就不敲出来了。
}

```

```

void test1()
{
    int a = 10;
    void (*block)() = (void (*)())&__test1_block_impl_0((void
*))__test1_block_func_0, &__test1_block_desc_0_DATA, a);

    a = 20;
    ((void (*)(__block_impl *))((__block_ipml *)block)->FuncPtr)
    ((__block_impl *)block);
}

```

```

int main(int argc, const char * argv[])
{
    /* @autoreleasepool */ { __AtAutoreleasePool
    __autoreleasepool;
        test1();
    }
    return 0;
}

```

```
static struct IMAGE_INFO { unsigned version; unsigned flag; }
_OBJC_IMAGE_INFO = { 0, 2 };
```

- 我们看到void test1()中， void (*block)() 右边最后面， 把a传进去了， 也就是把10这个值传进去了。
- 而且对void (block)()简化分析， void (block)() = &__test1_block_impl_0();所以block就是指向结构体的指针。
- 10传入block后， 代码最上面创建的__test1_block_impl_0结构体中， a = 10；
- 对void test1()中最下面的函数进行简化分析， 得到(block)->FuncPtr(block)， 我们在回到刚才test1_block_impl_0这个结构体中， **impl.FuncPtr = fp**;而fp又是传入结构体的第一个参数， 而在void (*block)()中， 传入结构体的第一个参数为test1_block_func_0， 也就是说(block)->FuncPtr(block) => __test1_block_func_0(block);
- 上一步相当于调用**test1_block_func_0 ()** 这个函数， 我们来看这个函数， 有这样一段代码： int a = cself->a;访问block中的a值， 传递给a； 所以是10.这种就是传值！！！

=====

- 我们再来看test2();添加了__block会发送什么变化呢

```
void test2()
{
    __attribute__((__blocks__(byref))) __Block_byref_a_0 a =
{ (void*)0, (__Block_byref_a_0 *)&a, 0, sizeof(__Block_byref_a_0), 10 };

    void(*block)() = (void (*)())&__test2_block_impl_0((void *)
) __test2_block_func_0, &__test2_block_desc_0_DATA,
( __Block_byref_a_0 *)&a, 570425344);

    (a.__forwarding->a) = 20;

    ((void (*)(__block_impl *))((__block_ipml *)block)->FuncPtr)
( __block_impl *block);
}
```

```
int main(int argc, const char * argv[])
{
    /* @autoreleasepool */ { __AtAutoreleasePool
__autoreleasepool;
    test2();
}
```

```

    return 0;
}
static struct IMAGE_INFO { unsigned version; unsigned flag; }
_OBJC_IMAGE_INFO = { 0, 2 };

```

- 代码虽然很多看着很复杂，但是我们只需要看我们想要知道的，睁大你的眼睛，看到void(*block)()这个函数的最后面，有个&a,天啊，这里传的是a的地址。从test2到test4，都是传址，所以a的值发生改变，block打印出来的是a的最终值。
- 总结：只有普通局部变量是传值，其他情况都是传址。

block的定义

```

// 无参无返回
void(^block)();
// 无参有返回
int(^block1)();
// 有参有返回
int(^block1)(int number);

```

也可以直接打入inline来自动生成block格式

```

<#returnType#>(^<#blockName#>)(<#parameterTypes#>) =
^(<#parameters#>) {
    <#statements#>
};

```

block的内存管理

- 无论当前环境是ARC还是MRC,只要block没有访问外部变量,block始终在全局区
- MRC情况下
 - block如果访问外部变量,block在栈里
 - 不能对block使用retain,否则不能保存在堆里
 - 只有使用copy,才能放到堆里
- ARC情况下
 - block如果访问外部变量,block在堆里
 - block可以使用copy和strong,并且block是一个对象

block的循环引用

- 如果要在block中直接使用外部强指针会发生错误,使用以下代码在block外部实现可以解决

```
_weak typeof(self) weakSelf = self;
```

- 但是如果在block内部使用延时操作还使用弱指针的话会取不到该弱指针,需要在block内部再将弱指针强引用一下
`_strong typeof(self) strongSelf = weakSelf;`

描述一个你遇到过的retain cycle例子。

```
block中的循环引用：一个viewController
@property (nonatomic, strong) HttpRequestHandler * handler;
@property (nonatomic, strong) NSData * data;
_handler = [HttpRequestHandler sharedManager];
[downloadData:^(id responseData){
_data = responseData;
}];
self 拥有 _handler, _handler 拥有 block, block 拥有 self (因为使用了 self 的
_data 属性, block 会 copy 一份 self)
解决方法：
_weak typeof(self) weakSelf = self
[downloadData:^(id responseData){
weakSelf.data = responseData;
```

block中的weak self，是任何时候都需要加的么？

- 不是什么任何时候都需要添加的，不过任何时候都添加似乎总是好的。只要出现像 self->block->self.property/self->_ivar 这样的结构链时，才会出现循环引用问题。好好分析一下，就可以推断出是否会有循环引用问题。

通过block来传值

- 在控制器间传值可以使用代理或者block, 使用block相对来说简洁
- 在前一个控制器的touchesBegan:方法内实现如下代码

```
ModalViewController *modalVc = [[ModalViewController alloc]
init];

modalVc.valueBlcok = ^(NSString *str){
NSLog(@"ViewController拿到%@", str);
};

[self presentViewController:modalVc animated:YES completion:nil];
```

- 在 ModalViewController 控制器的.h 文件中声明一个 block 属性 @property (nonatomic ,strong) void(^valueBlcok)(NSString *str);

- 并在.m文件中实现方法

```

-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:
(UIEvent *)event
{
    // 传值:调用block
    if (_valueBlock) {
        _valueBlock(@"123");
    }
}

```

- 这样在ModalViewController回到上一个控制器的时候,上一个控制器的label就能显示ModalViewController传过来的字符串

block作为一个参数使用

- 新建一个类,在.h文件中声明一个方法- (void)calculator:(int(^)(int result))block;
- 并在.m文件中实现该方法

```

-(void)calculator:(int (^)(int))block
{
    self.result = block(self.result);
}

```

- 在其他类中调用该方法

```

CalculatorManager *mgr = [[CalculatorManager alloc] init];
[mgr calculator:^(int result){
    result += 5;
    return result;
}];

```

block作为返回值使用

- 在masonry框架中我们可以看到如下用法
make.top.equalTo(superview.mas_top).with.offset(padding.top); 这个方法实现就是将block作为返回值来使用
- 来分析一下这段代码: 其实可以将这段代码看成
make.top, make.equalTo, make.with, make.offset, 所以可以得出一个结论是make.top返回了一个make, 才能实现make.top.equalTo

- 那来模仿一下这种功能的实现
- 新建一个类,在.h文件中声明一个方法- (CalculatorManager *(^)(int a))add;
- 在.m文件中实现方法

```
- (CalculatorManager * (^)(int a))add
{
    return ^(int a){
        _result += a;
        return self;
    };
}
```

- 这样就可以在别的类中实现上面代码的用法

```
mgr.add(1).add(2).add(3);
```

block的变量传递

- 如果block访问的外部变量是局部变量,那么就是值传递,外界改了,不会影响里面
- 如果block访问的外部变量是__block或者static修饰,或者是全局变量,那么就是指针传递,block里面的值和外界同一个变量,外界改变,里面也会改变
- 验证一下是不是这样
- 通过Clang来将main.m文件编译为C++
- 在终端输入如下命令clang -rewrite-objc main.m

```
void(*block)() = ((void (*)())&__main_block_impl_0((void
*)__main_block_func_0,           &__main_block_desc_0_DATA,
(__Block_byref_a_0 *)&a, 570425344));
void(*block)() = ((void (*)())&__main_block_impl_0((void
*)__main_block_func_0,           &__main_block_desc_0_DATA, a));
```

- 可以看到在编译后的代码最后可以发现被__block修饰过得变量使用的是&a,而局部变量是a

block的注意点

- 在block内部使用外部指针且会造成循环引用情况下,需要用__weak修饰外部指针
`__weak typeof(self) weakSelf = self;`
- 在block内部如果调用了延时函数还使用弱指针会取不到该指针,因为已经被销毁了,需要在block内部再将弱指针重新强引用一下

```
__strong typeof(self) strongSelf = weakSelf;
```

- 如果需要在block内部改变外部变量的话,需要在用__block修饰外部变量

使用block有什么好处? 使用NSTimer写出一个使用block显示(在UILabel上)秒表的代码。

说到block的好处, 最直接的就是代码紧凑, 传值、回调都很方便, 省去了写代理的很多代码。

对于这里根本没有必要使用block来刷新UILabel显示, 因为都是直接赋值。当然, 笔者觉得这是在考验应聘者如何将NSTimer写成一个通用用的Block版本。

NSTimer封装成Block版: <http://www.henishuo.com/nstimer-block/>
使用起来像这样:

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:1.0
                                                    repeats:YES
                                              callback:^() {
    weakSelf.secondsLabel.text = ...
}
[[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSRunLoopCommonModes];
```

block跟函数很像:

- 可以保存代码
- 有返回值
- 有形参
- 调用方式一样

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:

```
[UIView animateWithDuration:duration animations:^{
    [self.superview layoutIfNeeded];
}];
[[NSOperationQueue mainQueue]
addOperationWithBlock:^{
    self.someProperty = xyz;
}];
[[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成员变量的系统api，如GCD、NSNotificationCenter就要小心一点。比如GCD内部如果引用了self，而且GCD的其他参数是成员变量，则要考虑到循环引用：

```
__weak __typeof(self) weakSelf = self;  
dispatch_group_async(_operationsGroup, _operationsQueue, ^{  
    __typeof__(self) strongSelf = weakSelf;  
    [strongSelf doSomething];  
    [strongSelf doSomethingElse];  
});
```

类似的：

```
__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。
```

谈谈对Block 的理解?并写出一个使用Block执行UIView动画?

- Block是可以获取其他函数局部变量的匿名函数，其不但方便开发，并且可以大幅提高应用的执行效率(多核心CPU可直接处理Block指令)

```
[UIView transitionWithView:self.view duration:0.2  
options:UIViewAnimationOptionTransitionFlipFromLeft  
  
animations:^{ [[blueViewController view] removeFromSuperview];  
[[self view] insertSubview:yellowViewController.view atIndex:0]; }  
completion:NULL];
```

写出上面代码的Block的定义。

- `typedef void(^animations) (void);`
- `typedef void(^completion) (BOOL finished);`

什么是block

- 对于闭包(block),有很多定义，其中闭包就是获取其它函数局部变量的匿名函数，这个定义即接近本质又较好理解。
- 对于刚接触Block的同学，会觉得有些绕，因为我们习惯写这样的程序`main(){ funA();} funA(){funB();} funB(){.....};`就是函数main调用函数A，函数A调用函数B...函数们依次顺序执行，但现实中不全是这样的，例如项目经理M，手下有3个程序员A、B、C，当他给程序员A安排实现功能F1时，他并不等着A完成之后，再去安排B去实现F2，而是安排给A功能F1，B功能F2，C功能F3，然后可能去写技术文档，而当A遇到问题时，他会来找项目经理M，当B做完时，会通知M，这就是一个异步执行的例子。
- 在这种情形下，Block便可大显身手，因为在项目经理M，给A安排工作时，同时会告诉A若果遇到困难，如何能找到他报告问题(例如打他手机号)，这就是项目经理M给A的一个回调接口，要回掉的操作，比如接到电话，百度查询后，返回网页内容给A，这就是一个Block，在M交待工作时，已经定义好，并且取得了F1的任务号(局部变量)，却是在当A遇到问题时，才调用执行，跨函数在项目经理M查询百度，获得结果后回调该block。

block 实现原理

- Objective-C是对C语言的扩展，block的实现是基于指针和函数指针。
- 从计算语言的发展，最早的goto，高级语言的指针，到面向对象语言的block，从机器的思维，一步步接近人的思维，以方便开发人员更为高效、直接的描述出现实的逻辑(需求)。
- 使用实例:cocoaTouch框架下动画效果的Block的调用

使用`typedef`声明block

```
typedef void(^didFinishBlock) (NSObject *ob);
```

这就声明了一个`didFinishBlock`类型的block，

然后便可用

```
@property (nonatomic,copy) didFinishBlock finishBlock;
```

声明一个block对象，注意对象属性设置为copy，接到block参数时，便会自动复制一份。

`_block`是一种特殊类型，

使用该关键字声明的局部变量，可以被block所改变，并且其在原函数中的值会被改变。

关于block

答：面试时，面试官会先问一些，是否了解block，是否使用过block，这些问题相当于开场白，往往是下面一系列问题的开始，所以一定要如实根据自己的情况回答。

1). 使用block和使用delegate完成委托模式有什么优点？

首先要了解什么是委托模式，委托模式在iOS中大量应用，其在设计模式中是适配器模式中的对象适配器，Objective-C中使用id类型指向一切对象，使委托模式更为简洁。了解委托模式的细节：

iOS设计模式—委托模式

使用block实现委托模式，其优点是回调的block代码块定义在委托对象函数内部，使代码更为紧凑；

适配对象不再需要实现具体某个protocol，代码更为简洁。

2). 多线程与block

GCD与Block

使用 dispatch_async 系列方法，可以以指定的方式执行block

GCD编程实例

dispatch_async的完整定义

```
void dispatch_async(  
    dispatch_queue_t queue,  
    dispatch_block_t block);
```

功能：在指定的队列里提交一个异步执行的block，不阻塞当前线程

通过queue来控制block执行的线程。主线程执行前文定义的 finishBlock 对象

```
dispatch_async(dispatch_get_main_queue(), ^(void){ finishBlock();});
```

解释以下代码的内存泄漏原因

@implementation HJTestViewController

```
- (UITableViewCell *)tableView:(UITableView *)tableView  
cellForRowAtIndexPath:(NSIndexPath *)indexPath {  
    HJTestCell *cell = [tableView  
dequeueReusableCellWithIdentifier:@"TestCell"  
forIndexPath:indexPath];
```

```
[cell setTouchBlock:^(HJTableViewCell *cell) {
    [self reloadData];
}];
return cell;
}
```

原因：

```
[cell setTouchBlock:^(HJTableViewCell *cell) {
    [self reloadData];
}];
```

产生内存泄露的原因是因为循环引用

在给cell设置的TouchBlock中，使用了`_strong`修饰的`self`，由于Block的原理，当`touchBlock`从栈复制到堆中时，`self`会一同复制到堆中，`retain`一次，被`touchBlock`持有，而`touchBlock`又是被`cell`持有的，`cell`又被`tableView`持有，`tableView`又被`self`持有，因此形成了循环引用：`self`间接持有`touchBlock`，`touchBlock`持有`self`

一旦产生了循环引用，由于两个object都被强引用，所以`retainCount`始终不能为0，无法释放，产生内存泄漏

解决办法：

使用`weakSelf`解除`touchBlock`对`self`的强引用

```
__weak __typeof__(self) weakSelf = self;
[cell setTouchBlock:^(HJTableViewCell *cell) {
    [weakSelf reloadData];
}];
```

(9) Swift

网上有很多Swift的语法题，但是Swift现在语法还未稳定，所以在这里暂时不贴出语法题，可以自行搜索。

Swift和Objective-C的联系

- Swift与Objective-C共用同一套运行时环境

我们编写程序，让程序运行起来，被机器执行的代码并非全部是由我们自己来编写的。需要同时运行很多预先写好的支持性的代码，才能让我们自己的代码运行起来。程序并非单独存在的，运行时处在一定的环境当中。我总联想到很多小蚂蚁在泥土上面爬，而我自己写的程序只是其中的一只。

Swift跟Objective-C编译出的程序代码运行在同一套运行环境上面。Swift的类型可以桥接到Objective-C的类型，反之亦然。Swift编写的代码可以调用Objective-C编写的代码，反之也一样。

Objective-C之前积累下来的大量类库，实现不用改写Swift就可以直接调用。

- 同一个工程，可以同时使用Swift和Objective-C

Objective-C在一端，Swift在另一端，两端经中间文件进行桥接。桥接文件包含Objective-C的头文件，编译时自动转成Swift可以识别的形式。Swift就可以使用Objective-C的类和它的函数。

在Swift的类中，加上@objc（类名）的字样，Objective-C也可以使用Swift编写的类。但Swift跟C++的相互调用，需要Objective-C来封装。

- Swift骨子里大多与Objective-C一样

Objective-C出现过的绝大多数概念，比如引用计数、ARC、属性、协议、接口、初始化、扩展类、命名参数、匿名函数等，在Swift中继续有效（可能只是换了个术语）。我自己将Swift看成是Objective-C的一块大大的语法糖，其他人可能有不同感受。

Swift大多数概念与Objective-C一样，也有些概念在Objective-C找不到对应，比如泛型。Swift中将那种操作写一次就可以作用多个类型的语法叫做Generics（泛型）。

Swift比Objective-C有什么优势？

- Swift 容易阅读

不再需要行尾的分号，以及 if/else 语句中围绕条件表达式的括弧。另外就是方法的调用不再互相嵌套成中括号[[[]]]。Swift 中的方法和函数的调用使用行业内标准的在一对括弧内使用逗号分隔的参数列表。这样做的结果就是一种带有简化了句法和语法的更加干净有表现力的语言。

- Swift 更易于维护

Swift 丢掉了对着两个文件的要求。Xcode编译器可以自动计算出以来并执行增量构建。如此，将头文件 同实现文件相分离。把 Objective-C 头文件(.h) 和实现文件 (.m) 合并成了一个代码文件 (.swift)

- Swift 更加安全
Swift代码中的可选类型使得一个nil可选值的可能性变得非常的明确, 这意味它能在你写下一段糟糕的代码时会生成一个编译器错误. 这就建立了一种短程反馈的循环, 可以让程序员带着目标去写代码. 问题在代码被写就时就可以被修复, 这大大节省了你要在修复有关来自 Objective-C 指针逻辑的bug时需要耗费的时间和金钱.
- Swift 代码更少
Swift 减少了重复性语句和字符串操作所需要的代码量。在 Objective-C 中, 使用文本字符串将两块信息组合起来的操作非常繁琐。Swift 采用当代编程语言的特性, 比如使用“+”操作符将两个字符串加到一起
Swift中的类型系统减少了代码语句的复杂性--作为编译器可以理解的类型。比如, Objective-C要求程序员记住特殊字符标记 (%s, %d, %@) 并且提供了一个用逗号分隔的变量来代替每个标记。Swift支持字符串插入, 这就消除了需要记住的标记和允许程序员直接插入变量到面向用户的字符串中
- Swift 速度更快
删除遗留下来的C语言约定大大提升了引擎盖之下Swift的性能, 进行计算密集型任务的性能上, Swift已经逼近C++的表现, 将近是OC运行速度的1.4倍.
- 总结: 使用 Swift, 程序员只要维护原来一半量的代码文件, 手动的代码同步工作为零, 标点输入出错的概率也远远低于以前 -- 这样就能腾出更多的时间写高质量的代码。通过使用可选类型 -- 一种针对返回或不返回值的编译时安全机制, 而返回值是同步操作、网络失效时无效的用户输入以及数据验证错误发生时普遍会遇到的问题。ARC 在 Swift 中对过程式 C 风格的代码, 还有苹果公司 Cocoa 框架使用的面向对象代码都进行了统一。

Swift的内存管理是怎样的?

- Swift 使用自动引用计数 (Automatic Reference Counting, ARC) 来简化内存管理, 这种内存管理方式相比GC而言, 对程序员的要求较高, 并且ARC比GC更容易引起编程错误, 但却比GC快。尤其在性能很重要的场合。

Swift支持面向过程编程吗?

- 它采用了 Objective-C 的命名参数以及动态对象模型, 可以无缝对接到现有的 Cocoa 框架, 并且可以兼容 Objective-C 代码, 支持面向过程编程和面向对象编程。

举例说明Swift里面有哪些是 Objective-C中没有的?

- Swift引入了在Objective-C中没有的一些高级数据类型, 例如tuples (元组), 可以使你创建和传递一组数值。
- Swift还引入了可选项类型 (Optionals), 用于处理变量值不存在的情况。可选项的意思有两种: 一是变量是存在的, 例如等于X, 二是变量值根本不存在。Optionals类似于Objective-C中指向nil的指针, 但是适用于所有的数据类型, 而非仅仅局限于类, Optionals相比于Objective-C中nil指针更加安全和简明, 并且也是Swift诸多最强大功能的核心。

Swift 是一门安全语言吗？

- Swift是一门类型安全的语言，Optionals就是代表。Swift能帮助你在类型安全的环境下工作，如果你的代码中需要使用String类型，Swift的安全机制能阻止你错误的将Int值传递过来，这使你在开发阶段就能及时发现并修正问题。

为什么要在变量类型后面加个问号？

- 用来标记这个变量的值是可选的，一般用“！”和“？”定义可选变量的区别：用“！”定的可选变量必须保证转换能够成功，否则报错，但定义的变量可以直接使用，不会封装在option里；而用“？”号定的可选变量即使转换不成功本身也不会出错，变量值为nil，如果转换成功，要使用该变量进行计算时变量名后需要加“！”

什么是泛型，它们又解决了什么问题？

- 泛型是用来使代码能安全工作。在Swift中，泛型可以在函数数据类型和普通数据类型中使用，例如类、结构体或枚举。
- 泛型解决了代码复用的问题。有一种常见的情况，你有一个方法，需要一个类型的参数，你为了适应另一种类型的参数还得重新再写一遍这个方法。
比如，在下面的代码中，第二个方法是第一个方法的“克隆体”：

```
func areIntEqual(x: Int, _ y: Int) -> Bool {  
    return x == y  
}  
func areStringsEqual(x: String, _ y: String) -> Bool {  
    return x == y  
}  
areStringsEqual("ray", "ray") // true  
areIntEqual(1, 1) // true
```

一个Objective-C开发者可能会采用NSObject来解决问题：

```
- import Foundation  
- func areTheyEqual(x: NSObject, _ y: NSObject) -> Bool {  
-     return x == y  
- }  
- areTheyEqual("ray", "ray") // true  
- areTheyEqual(1, 1) // true
```

这段代码能达到目的，但是编译的时候并不安全。它允许一个字符串和一个整型数据进行比较：

```
-     areTheyEqual(1, "ray")  
//程序可能不会崩溃，但是允许一个字符串和一个整型数据进行比较可能不会得到想要的结果。
```

采用泛型的话，你可以将上面两个方法合并为一个，并同时还保证了数据类型安全。这是实现代码：

```
- func areTheyEqual<T: Equatable>(x: T, _ y: T) -> Bool {
-     return x == y
- }
- areTheyEqual("ray", "ray")
- areTheyEqual(1, 1)
```

(10) UI

viewcontroller的一些方法的说明

viewDidLoad,viewWillDisappear, viewDidAppear方法的顺序和作用?

viewWillAppear:视图即将可见时调用。默认情况下不执行任何操作

viewDidAppear:视图已完全过渡到屏幕上时调用

viewWillDisappear:视图被驳回时调用，覆盖或以其他方式隐藏。默认情况下不执行任何操作

viewDidDisappear:视图被驳回后调用，覆盖或以其他方式隐藏。默认情况下不执行任何操作
loadView; .这是当他们没有正在使用nib视图页面，子类将会创建自己的自定义视图层。绝不能直接调用。

viewDidLoad:在视图加载后被调用，如果是在代码中创建的视图加载器，他将会在loadView方法后被调用，如果是从nib视图页面输出，他将会在视图设置好后被调用。

「initWithNibName: bundle:」载入nib档案来初始化「loadView」载入视图「viewDidLoad」
在载入视图至内存后会呼叫的方法「viewDidUnload」在视图从内存中释放后会呼叫的方法
(当内存过低，释放一些不需要的视图时调用)

「viewWillAppear」当收到视图在视窗将可见时的通知会呼叫的方法

「viewDidAppear」当收到视图在视窗已可见时的通知会呼叫的方法

「viewWillDisappear」当收到视图将去除、被覆盖或隐藏于视窗时的通知会呼叫的方法

「viewDidDisappear」当收到视图已去除、被覆盖或隐藏于视窗时的通知会呼叫的方法

「didReceiveMemoryWarning」收到系统传来的内存警告通知后会执行的方法

「shouldAutorotateToInterfaceOrientation」是否支持不同方向的旋转视图

「willAnimateRotationToInterfaceOrientation」在进行旋转视图前的会执行的方法（用于调整旋转视图之用）

代码的执行顺序

- 1、 alloc 创建对象，分配空间
 - 2、 init (initWithNibName) 初始化对象，初始化数据
 - 3、 loadView 从nib载入视图，通常这一步不需要去干涉。除非你没有使用xib文件创建视图
 - 4、 viewDidLoad 载入完成，可以进行自定义数据以及动态创建其他控件
 - 5、 viewWillAppear 视图将出现在屏幕之前，马上这个视图就会被展现在屏幕上
 - 6、 viewDidAppear 视图已在屏幕上渲染完成当一个视图被移除屏幕并且销毁的时候的执行顺序，这个顺序差不多和上面的相反
-
- 1、 viewWillDisappear 视图将被从屏幕上移除之前执行
 - 2、 viewDidDisappear 视图已经被从屏幕上移除，用户看不到这个视图了
 - 3、 dealloc 视图被销毁，此处需要对你在init和viewDidLoad中创建的对象进行释放

什么是key window?

- 一个窗口当前能接受键盘和非触摸事件时，便被认为是主窗口。而触摸事件则被投递到触摸发生的窗口，没有相应坐标值的事件被投递到主窗口。同一时刻只有一个窗口是主窗口。

谈一谈你是怎么封装view的

- 先添加所需子控件
- 再接收模型数据根据模型数据设置子控件数据和位置
- 简而言之，自己的事情自己做，把不需要暴露出去的封装起来

简单说一下APP的启动过程,从main文件开始说起

程序启动分为两类:1.有storyboard 2.没有storyboard
有storyboard情况下:

1.main函数
2.UIApplicationMain
* 创建UIApplication对象
* 创建UIApplication的delegate对象
3.根据Info.plist获得Main.storyboard的文件名,加载Main.storyboard(有storyboard)
* 创建UIWindow
* 创建和设置UIWindow的rootViewController
* 显示窗口

没有storyboard情况下:

```
1.main函数  
2.UIApplicationMain  
* 创建UIApplication对象  
* 创建UIApplication的delegate对象  
3.delegate对象开始处理(监听)系统事件(没有storyboard)  
* 程序启动完毕的时候，就会调用代理的  
application:didFinishLaunchingWithOptions:方法  
* 在application:didFinishLaunchingWithOptions:中创建UIWindow  
* 创建和设置UIWindow的rootViewController  
* 显示窗口
```

怎么解决缓存池满的问题(cell)

iOS中不存在缓存池满的情况，因为通常我们在ios中开发，对象都是在需要的时候才会创建，有种常用的说话叫做懒加载，还有在UITableView中一般只会创建刚开始出现在屏幕中的cell，之后都是从缓存池里取，不会在创建新对象。缓存池里最多也就一两个对象，缓存池满的这种情况一般在开发java中比较常见，java中一般把最近最少使用的对象先释放。

UIButton与UITableView的层级结构

- 继承结构，属于内部的子控件结构
- UIButton为：UIButton > UIControl > UIView > UIResponder > NSObject
- UITableView为：UITableView > UIScrollView > UIView > UIResponder > NSObject

设置scroll view的contentsize能在ViewDidLoad里设置么,为什么

- 一般情况下可以设置在viewDidLoad中，但在autolayout下，系统会在viewDidAppear之前根据subview的constraint重新计算scrollview的contentsize。这就是为什么，在viewDidLoad里面手动设置了contentsize没用。因为在后面，会再重新计算一次，前面手动设置的值会被覆盖掉。
- 解决办法就是：
 - 去除autolayout选项，自己手动设置contentsize
 - 如果要使用autolayout，要么自己设置完subview的constraint，然后让系统自动根据constraint计算出contentsize。要么就在viewDidAppear里面自己手动设置contentsize。

简述你对UIView、UIWindow和CALayer的理解

- UIView: 属于UIKit.framework框架,负责渲染矩形区域的内容,为矩形区域添加动画,响应区域的触摸事件,布局和管理一个或多个子视图
- UIWindow: 属于UIKit.framework框架,是一种特殊的UIView,通常在一个程序中只会有一个UIWindow,但可以手动创建多个UIWindow,同时加到程序里面。 UIWindow在程序中主要起到三个作用:
 - 作为容器,包含app所要显示的所有视图
 - 传递触摸消息到程序中view和其他对象
 - 与UIViewController协同工作,方便完成设备方向旋转的支持
- CALayer: 属于QuartzCore.framework,是用来绘制内容的,对内容进行动画处理依赖与UIView来进行显示,不能处理用户事件。
- UIView和CALayer是相互依赖的,UIView依赖CALayer提供内容,CALayer依赖UIView的容器显示绘制内容。
- (补充)UIViewController: 每个视图控制器都有一个自带的视图,并且负责这个视图相关的一切事务。方便管理视图中的子视图,负责model与view的通信;检测设备旋转以及内存警告;是所有视图控制类的积累,定义了控制器的基本功能。

frame和bounds有什么不同? (Difference between frame and bounds?)

- frame指的是: 该view在父view坐标系统中的位置和大小 (参照点是父亲的坐标系统)
- bounds指的是: 该view在本身坐标系统中的位置和大小 (参照点是本身坐标系统)

关于页面间传值的问题?

属性传值: A页面设置属性 NSString *paramString, 在跳转B页面的时候初始化 paramString。

```
//A页面.h文件
@property (nonatomic, copy) NSString *paramString;
//A页面.m文件
NextViewController *nextVC = [[NextViewController alloc] init];
```

```
nextVC.paramString = @"参数传质";
[self presentViewController:nextVC animated:YES completion:nil];
```

委托delegate传值：在B页面定义delegate，并且设置delegate属性，在A页面实现delegate协议

通知notification传值：在B页面中发送通知，在A页面注册观察者并且在不用的时候移除观察者。

```
//B页面发送通知
[[NSNotificationCenter defaultCenter]
postNotificationName:@"ChangeNameNotification" object:self
userInfo:@{@[@"name":self.nameTextField.text}}];
[self dismissViewControllerAnimated:YES completion:nil];
//A页面注册观察者
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(ChangeNameNotification:)
name:@"ChangeNameNotification" object:nil];
}
//观察到通知时候的处理方法
-(void)ChangeNameNotification:(NSNotification*)notification{
NSDictionary *nameDictionary = [notification userInfo];
self.nameLabel.text = [nameDictionary objectForKey:@"name"];
}
//通知不使用的时候移除观察者
[[NSNotificationCenter defaultCenter] removeObserver:self];
```

block传值：在B页面定义一个block类型的变量，在B页面跳转A的时候调用这个block。在A页面跳转到B页面的时候对B页面的block赋值。

```
//B页面定义block，并设置block类型的变量
typedef void (^ablock)(NSString *str);
@property (nonatomic, copy) ablock block;
//B页面跳转到A页面调用这个block
self.block(self.nameTextField.text);
[self dismissViewControllerAnimated:YES completion:nil];
//A页面跳转到B页面的时候对B页面的block赋值，这样在B页面跳转的时候就会回调这个block函数
[self presentViewController:second animated:YES completion:nil];
second.block = ^(NSString *str){
    self.nameLabel.text = str;
};
```

kvo传值：在A页面设置B页面的变量second，并且对这个变量进行观察

```
- (void)addObserver:(NSObject * _Nonnull)anObserver forKeyPath:
(NSString * _Nonnull)keyPath options:
(NSKeyValueObservingOptions)options context:(void *
_Nullable)context
```

并在A页面实现

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

在B页面对变量keyPath进行设置，在A页面都会观察的到。

```
@property (nonatomic, strong) SecondViewController *second;
```

//在A视图跳转到B视图的地方添加如下代码

```
self.second = [[SecondViewController alloc]
initWithNibName:@"SecondViewController" bundle:nil];
[self.second addObserver:self forKeyPath:@"userName"
options:NSKeyValueObservingOptionNew context:nil];
[self presentViewController:self.second animated:YES
completion:nil];
```

//实现这个观察对象的方法

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context  
//在B页面对userName进行设置，在A页面都可以听到
```

单例模式传值：通过全局的方式保存

对于通知代理面试常问，代理和通知分别在什么情况下使用？区别？各自优点？

关于视图的生命周期的问题

响应者链条? (What is responder chain?)

- 事件响应链。包括点击事件，画面刷新事件等。在视图栈内从上至下，或者从下之上传播. 可以说点事件的分发，传递以及处理。具体可以去看下touch事件这块。

- 首先解释响应者链的概念
 - UIResponder类，是UIKIT中一个用于处理事件响应的基类。窗口上的所有事件触发，都由该类响应（即事件处理入口）。所以，窗口上的View及控制器都是派生于该类的，例如UIView、UIViewController等。
 - 调用UIResponder类提供的方法或属性，我们就可以捕捉到窗口上的所有响应事件，并进行处理。
 - 响应者链条是由多个响应者对象连接起来的链条，其中响应者对象是能处理事件的对象，所有的View和ViewController都是响应者对象，利用响应者链条能让多个控件处理同一个触摸事件.
- 事件传递机制

如果当前view不能处理当前事件，那么事件将会沿着响应者链(Responder Chain)进行传递，知道遇到能处理该事件的响应者(Responsder Object)。

imageimageimageimageimageimageimageimageimageimageimage

- 接收事件的initial view如果不能处理该事件并且她不是顶层的view，则事件会往它的父View进行传递。
- initial view的父View获取事件后如果仍不能处理，则继续往上传递，循环这个过程。如果顶层的view还是不能处理这个事件的话，则会将事件传递给它们的ViewController，
- 如果ViewController也不能处理，则传递给Window(UIWindow)，此时Window不能处理的话就将事件传递UIApplication，最后如果连Application也不能处理，则废弃该事件

ViewController的loadView,viewDidLoad,viewDidUnload分别是在什么时候调用的？在自定义ViewController的时候这几个函数里面应该做什么工作？

- viewDidLoad在view从nib文件初始化时调用，
- loadView在controller的view为nil时调用。
- 此方法在编程实现view时调用，view控制器默认会注册memory warning notification,当viewcontroller的任何view没有用的时候，viewDidUnload会被调用，在这里实现将retain的view release,如果是retain的IBOutlet view属性则不要在这里release,IBOutlet会负责release。

UITableView的重用机制?(或者如何在一个view上显示多个tableView,tableView要求不同的数据源以及不同的样式 (要求自定义cell), 如何组织各个tableView的delegate和dataSource?请说说实现思路?)

- 查看UITableView头文件,会找到NSMutableArray *visibleCells*,和NSMutableArray *reusableTableCells*两个结构。
- *visibleCells*内保存当前显示的cells,*reusableTableCells*保存可重用的cells。
- TableView显示之初,*reusableTableCells*为空,那么 [tableView dequeueReusableCellWithIdentifier:CellIdentifier]返回nil。
- 开始的cell都是通过 [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] 来创建,而且cellForRowAtIndexPath只是调用最大显示cell数的次数。比如:有100条数据,iPhone一屏最多显示10个cell。
- 程序最开始显示TableView的情况是:
 - 用[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]创建10次cell,并给cell指定同样的重用标识(当然,可以为不同显示类型的cell指定不同的标识)。并且10个cell全部都加入到 *visibleCells*数组,*reusableTableCells*为空。
 - 向下拖动tableView,当cell1完全移出屏幕,并且cell11(它也是alloc出来的,原因同上)完全显示出来的时候。cell11加入到*visibleCells*,cell1移出*visibleCells*,cell1加入到*reusableTableCells*。
 - 接着向下拖动tableView,因为*reusableTableCells*中已经有值,所以,当需要显示新的cell, *cellForRowAtIndexPath*再次被调用的时候,[tableView dequeueReusableCellWithIdentifier:CellIdentifier],返回cell1。cell1加入到*visibleCells*,cell1 移出*reusableTableCells*;cell2移出 *visibleCells*,cell2加入到*reusableTableCells*。之后再需要显示的Cell就可以正常重用了.
 - 注意: 配置Cell的时候一定要注意, 对取出的重用的cell做重新赋值, 不要遗留老数据。

在一个tableView中需要自定义多种样式的cell(两种或三种),通常你如何实现,说说思路即可?

- 同上 !

UITableView的性能优化？滑动的时候有种卡的感觉是为什么？怎么解决？

- 在使用第三方应用时，却经常遇到性能上的问题，普遍表现在滚动时比较卡，特别是cell中包含图片的情况时。
- 实际上针对性地优化一下就可以解决tableView滑动的时候卡顿的问题：
 - 使用不透明视图。不透明的视图可以提高渲染的速度。可以将cell及其子视图的opaque属性设为YES（默认值）。
 - 不要重复创建不必要的cell。UITableView只需要一屏幕的UITableViewCell对象即可。因此在cell不可见时，可以将其缓存起来，而在需要时继续使用它即可。注意：cell被重用时，需要调用setNeedsDisplayInRect:或setNeedsDisplay方法重绘cell。
 - 减少动画效果的使用，最好不要使用insertRowsAtIndexPaths:withRowAnimation:方法，而是直接调用reloadData方法。
 - 减少视图的数目。Cell包含了textLabel、detailTextLabel和imageView等view，而你还可以自定义一些视图放在它的contentView里，创建它会消耗较多资源，并且也影响渲染的性能。
 - cell包含图片，且数目较多，使用自定义的cell速度会比使用默认的要快。继承UITableViewCell，重写drawRect方法：

```
- (void)drawRect:(CGRect)rect { if (image) { [image drawAtPoint:imagePoint]; self.image = nil; } else { [placeHolder drawAtPoint:imagePoint]; } [text drawInRect:textRect withFont:font lineBreakMode:UILineBreakModeTailTruncation]; }
```

不过这样一来，你会发现选中一行后，这个cell就变蓝了，其中的内容就被挡住了。最简单的方法就是将cell的selectionStyle属性设为UITableViewCellSelectionStyleNone，这样就不会被高亮了。
-不需要与用户交互时，使用CALayer，将内容绘制到layer上，然后对cell的contentView.layer调用addSublayer:方法。这个例子中，layer并不会显著影响性能，但如果layer透明，或者有圆角、变形等效果，就会影响到绘制速度了。解决办法可参见后面的预渲染图像。
 - 不要做多余的绘制工作。在实现drawRect:的时候，它的rect参数就是需要绘制的区域，这个区域之外的不需要进行绘制。
 - 预渲染图像。你会发现即使做到了上述几点，当新的图像出现时，仍然会有短暂的停顿现象。解决的办法就是在图形上下文中画，导出成UIImage对象，然后再绘制到屏幕。（头像圆角，或者其他变形的时候，用图形上下文能提高性能。）异步绘制

- 不要阻塞主线程。tableview在更新数据时，整个界面卡住不动，完全不响应用户请求。常见的是网络请求，等待时间长待数秒。
- 解决方案：使用多线程，让子线程去执行这些函数或方法。
- 注意：当下载线程数超过2时，会显著影响主线程的性能。所以在不需要响应用户请求时，下载线程数可以增加到5，不建议再加了，以加快下载速度。如果用户正在交互，应把线程数量控制在2个以内。
- 提前计算并缓存好高度，因为heightforrowatindexpath调用非常频繁
- 选择正确的数据结构：学会选择对业务场景最合适的数据结构是写出高效代码的基础。比如，数组：有序的一组值。使用索引来查询很快，使用值查询很慢，插入/删除很慢。字典：存储键值对，用键来查找比较快。集合：无序的一组值，用值来查找很快，插入/删除很快。
- gzip/zip压缩：当从服务端下载相关附件时，可以通过gzip/zip压缩后再下载，使得内存更小，下载速度也更快。

tableview的cell里如何嵌套collection view？

思路同网易新闻类似，用自定义的继承自UITableViewCell的类，在initWithFrame的构造方法中，初始化自定义的继承自UICollectionView的类

下拉和上拉的原理？

- 以tableView的上拉刷新为例：
 - 为了进行无缝阅读，通过tableView的代理方法，willDisplayCell判断是否是最后一行，
 - 如果是最后一行，在显示最后一行的同时，判断当前是否存在上拉刷新
 - 如果当前没有上拉刷新，就进行加载数据，启动小菊花转啊转。
- 以tableView的下拉刷新为例：
 - 判断当前的上拉刷新视图是否动画
 - 如果没有动画，就不是上拉刷新
 - 然后下拉刷新加载数据
 - 加载完毕数据关闭刷新



iOS资源群：190892815

iOS逆向资源：（定期分享）

<http://weibo.com/Edstick>

如何实现cell的动态的行高？

- 如果希望每条数据显示自身的行高，必须设置两个属性，1.预估行高，2.自定义行高
- 设置预估行高 `tableView.estimatedRowHeight = 200`
- 设置定义行高 `tableView.rowHeight = UITableViewAutomaticDimension`
- 如果要让自定义行高有效，必须让容器视图有一个自下而上的约束

谈谈webView

- iOS开发中webView和native code的配合上的一些经验和技巧。
- webView与运维成本低，更新几乎不依赖App的版本；但在交互和性能上与跟native code有很大差距。
- native code与之对应。
- HTML5确实给web带入了一个新时代。这个时代是什么， web app。也就是说，只有脱离native的前提，在浏览器的环境下，HTML5的意义才能显现，而我们讨论iOS App的时候，HTML5显然没什么意义。
- 不管是用webView还是native code，两个原则：
 - 用户体验不打折
 - 运维成本低
- 为什么不提开发成本。因为做web开发和iOS开发根本就是两回事。当然，web开发发展了这么多年，对于某些功能实现是要比native app快。但多数情况，同一个功能，对于iOS开发者和web开发者，用各自擅长的方式开发成本都最低，所以说某个功能开发成本低，往往是一个伪命题。
- 刚刚说了，webView的优势在于更新不依赖版本，那么在一款App中，只有会频繁更新的界面考虑webView才有意义。那么哪些界面会频繁更新，这就要因App而异了。
- 首页。首页资源可谓必争之地，内容一天一换是正常现象，一天几换也不稀奇。而如果仅仅是内容的更换，非要上个webView就显得有些激进了。而事实上首页的变化千奇百怪，逢年过节变个脸，特殊情况挂个公告，偶尔还要特批强推一把某个业务，等等。此前，我在设计App首页的时候，把首页配置设计的非常复杂。App端要处理n种情况，n各参数，server端要记住n种规则，直到一天，我崩溃了，把首页完全换成webView，才豁然开朗。
 - 活动页。做互联网都知道，活动，是一个最常见的运营手段。特点是，周期短，功能少，但基本不能复用。这些特点都标识了活动不适合做native，要用

webview实现。即使有人告诉你说，我的活动是一个长期活动而且形式不变，也不要相信他。因为在第二期，第三期，第四期他会分别加上一些非常诡异，却有很合理的小变更，而这些变更是你在那个版本根本无法实现的。

- 试水的新功能。这种界面，往往设计不成熟，需要在运行过程中不断收集用户反馈，更新升级，甚至决定去留。所以，只有webview才能hold住如此不稳定的功能。切记在一个功能还没有确定之前，不要大张旗鼓单位开发native code，要知道，你写的这些代码，三天后就要改一遍，而且要发布上线。
- 富文本内容。这个不用多说了吧，按照HTML的常用标签做一个webtext可不是小工程。而且富文本的变化太多了，一点无法匹配，都会导致整个界面巨丑。
- OK，上边说了我认为最该使用webview的4个界面，分别带有不同的特点，但webview的交互是个短板，因此webview在一个App中，只能作为界面，不允许在界面中出现动作。而一个webview的界面如何跟native code结合起来呢，我的答案是，超链接。在webview上点击超链接，会调用webview delegate的shouldload方法，自这里拦截请求，进行处理。由于webview的链接都是URL，因此我建议，把整个App的界面都用URL管理起来。
- 长相问题，webview很难长成native的view。方案：长不成也要装成。在一些情况下，禁用webview滚动，使用滚动框架（iScroll不错）去实现。webview上下留出200pixel的空白背景，y从-200开始。否则大家知道，webview上下会有阴影的背景，不藏起来会很丑。等等，还有很多其他的方法去伪装webview，是要视情景而用。
- cell中嵌套webview，在oc中调用js获取web的高度，
CGFloat height =
[[self.webView
stringByEvaluatingJavaScriptFromString:@"document.body.offsetHeight"]
floatValue];在通过webViewDidFinishLoad里面更新行高。

awakeFromNib与viewDidLoad区别

- awakeFromNib：当.nib文件被加载的时候，会发送一个awakeFromNib消息到.nib文件中的每个对象，每个对象都可以定义自己的awakeFromNib函数来响应这个消息，执行必要操作。也就是说 通过.nib文件创建view对象执行awakeFromNib
- viewDidLoad：当view对象被加载到内存就会执行viewDidLoad，不管是通过nib还是代码形式，创建对象就会执行viewDidLoad

layoutSubviews何时调用？

- 初始化init方法时不会触发
- 滚动uiscrollview触发
- 旋转屏幕触发
- 改变view的值触发，前提是frame改变了
- 改变uiview的大小触发

viewcontroller的didreceivememorywaring在什么时候调用 默认操作是什么

- 应用程序收到来自系统的内存警告时，调用didreceivememorywaring方法
- 默认做法：控制器上的view不再窗口上显示时，调用viewWillUnload，直接销毁view，并调用viewDidUnload

UIWindow和UIView和 CALayer 的联系和区别？

- UIView是视图的基类，UIViewController是视图控制器的基类，UIResponder是表示一个可以在屏幕上响应触摸事件的对象；
- UIWindow是UIView的子类，UIWindow的主要作用：一是提供一个区域来显示UIView，二是将事件（event）的分发给UIView，一个应用基本上只有一个UIWindow。图层不会直接渲染到屏幕上，UIView是iOS系统中界面元素的基础，所有的界面元素都是继承自它。它本身完全是由CoreAnimation来实现的。它真正的绘图部分，是由一个CALayer类来管理。UIView本身更像是一个CALayer的管理器。一个UIView上可以有n个CALayer，每个layer显示一种东西，增强UIView的展现能力。
- 都可以显示屏幕效果
- 如果需要用户交互就要用UIView，其可接收触摸事件（继承UIResponder），而CALayer不能接收触摸事件
- 如果没有用户交互可选用CALayer，因为其所在库较小，占用的资源较少

UIScrollView

- contentSize 内容视图的尺寸
- contentOffset 内容视图当前位置相对滚动视图frame的偏移量
- contentInset 内容视图相对滚动视图frame的展示原点

如何实现瀑布流,流水布局

- 使用UICollectionView
- 使用自定义的FlowLayout
- 需要在layoutAttributesForElementsInRect中设置自定义的布局(item的frame)
- 在prepareLayout中计算布局
- 遍历数据内容，根据索引取出对应的attributes(使用layoutAttributesForCellWithIndexPath)，根据九宫格算法设置布局
- 细节1：实时布局，重写shouldInvalidateLayoutForBoundsChange(bounds改变重新布局，scrollview的contentoffset>bounds)
- 细节2：计算设置itemsize(保证内容显示完整，uicollectionview的content size是根据itemize计算的)，根据列最大高度/对应列数量求出，最大高度累加得到
- 细节3：追加item到最短列，避免底部参差不齐。

UIImage有哪几种加载方式

- 二进制 imageWithData

- Bundle imageNamed
- 本地路径 imageWithContentOfFile

描述九宫格算法

- NSInteger col = x;//定义列数
- NSInteger index = self.shopsView.subviews.count;//获取下标
- CGFloat margin = (self.shopsView.frame.size.width - col*viewW) / (col - 1); //定义间隔
- CGFloat viewX = (index % col) * (viewW + margin);
- CGFloat viewY = (index / col) * (viewH + 10);

实现图片轮播图

- ScrollView只需要设置三个ImageView即可，并且默认显示中间的ImageView
- 根据ScrollView的移动情况，迅速变化三个ImageView中图片数据
- ImageView更新完毕后，偷偷把ScrollView拉回到中间的ImageView位置，这样视觉效果上就实现了无限循环的效果

应用的生命周期

- -(BOOL)application:(UIApplication)application willFinishLaunchingWithOptions:(NSDictionary)launchOptions 告诉代理进程启动但还没进入状态保存
- -(BOOL)application:(UIApplication)application didFinishLaunchingWithOptions:(NSDictionary)launchOptions 告诉代理启动基本完成程序准备开始运行
- -(void)applicationWillResignActive:(UIApplication *)application 当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话了
- -(void)applicationDidBecomeActive:(UIApplication *)application 当应用程序入活动状态执行，这个刚好跟上面那个方法相反
- -(void)applicationDidEnterBackground:(UIApplication *)application 当程序被推送到后台的时候调用。所以要设置后台继续运行，则在这个函数里面设置即可
- -(void)applicationWillEnterForeground:(UIApplication *)application 当程序从后台将要重新回到前台时候调用，这个刚好跟上面的那个方法相反。
- -(void)applicationWillTerminate:(UIApplication *)application 当程序将要退出是被调用，通常是用来保存数据和一些退出前的清理工作。

load initialize方法的区别

- +(void)load;
 - 当类对象被引入项目时, runtime 会向每一个类对象发送 load 消息
 - load 方法会在每一个类甚至分类被引入时仅调用一次, 调用的顺序：父类优先于子类, 子类优先于分类
 - load 方法不会被类自动继承
- +(void)initialize;
 - 也是在第一次使用这个类的时候会调用这个方法

UIScrollView 大概是如何实现的，它是如何捕捉、响应手势的？

- 我对UIScrollView的理解是frame就是他的contentSize,bounds就是他的可视范围,通过改变bounds从而达到让用户误以为在滚动,以下是一个简单的UIScrollView实现
- 第二个问题个人理解是解决手势冲突,对自己添加的手势进行捕获和响应

// 让UIScrollView遵守UIGestureRecognizerDelegate协议,实现这个方法,在这里方法里对添加的手势进行处理就可以解决冲突

- (BOOL)gestureRecognizer:(UIGestureRecognizer)gestureRecognizer
shouldRecognizeSimultaneouslyWithGestureRecognizer:
(UIGestureRecognizer)otherGestureRecognizer

+[UIView animateWithDuration:animations:completion:] 内部大概是如何实现的？

animateWithDuration:这就等于创建一个定时器
animations:这是创建定时器需要实现的SEL
completion:是定时器结束以后的一个回调block

什么时候会发生「隐式动画」？

- 当改变CALayer的一个可做动画的属性,它并不能立刻在屏幕上体现出来.相反,它是从先前的值平滑过渡到新的值。这一切都是默认的行为,你不需要做额外的操作,这就是隐式动画

如何把一张大图缩小为1/4大小的缩略图？

imgData = UIImageJPEGRepresentation(image, 0.6f)

当TableView的Cell改变时，如何让这些改变以动画的形式呈现？

```
[tableView deselectRowAtIndexPath:indexPath animated:TRUE];  
// 重点是这2句代码实现的功能  
[tableView beginUpdates];  
[tableView endUpdates];
```

为什么当 Core Animation 完成时，layer 又会恢复到原先的状态？

- 因为这些产生的动画只是假象,并没有对layer进行改变.那么为什么会这样呢,这里要讲一下图层树里的呈现树.呈现树实际上是模型图层的复制,但是它的属性值表示了当前外观效果,动画的过程实际上只是修改了呈现树,并没有对图层的属性进行改变,所以在动画结束以后图层会恢复到原先状态

设计一个进度条。

1. 自定义一个UIView的子类

```
//提供一个成员属性，接收下载进度值  
@property (nonatomic, assign) CGFloat progress;
```

2. 重写成员属性progress的setter

```
//每次改变成员属性progress的值，就会调用它的setter  
-(void)setProgress:(CGFloat)progress  
{  
    _progress = progress;  
    //当下载进度改变时，手动调用重绘方法  
    [self setNeedsDisplay];  
}
```

3. 重写

```
-(void)drawRect:(CGRect)rect (核心)  
-(void)drawRect:(CGRect)rect  
{  
    //设置圆弧的半径  
    CGFloat radius = rect.size.width * 0.5;  
    //设置圆弧的圆心  
    CGPoint center = CGPointMake(radius, radius);  
    //设置圆弧的开始的角度（弧度制）  
    CGFloat startAngle = - M_PI_2;  
    //设置圆弧的终止角度  
    CGFloat endAngle = - M_PI_2 + 2 * M_PI * self.progress;  
    //使用UIBezierPath类绘制圆弧  
    UIBezierPath *path = [UIBezierPath  
        bezierPathWithArcCenter:center  
        radius:radius - 5 startAngle:startAngle endAngle:endAngle  
        clockwise:YES];  
    //将绘制的圆弧渲染到图层上（即显示出来）  
    [path stroke];  
}
```

如何播放 GIF 图片，有什么优化方案么？

- UIImageView用来显示图片，使用UIImageView中的动画数组来实现图片的动画效果
- 用UIWebView来显示动态图片
- 第三方显示框架

- 通过UIImageView显示动画效果，实际上是把动态的图拆成了一组静态的图，放到数组中，播放的时候依次从数组中取出。如果播放的图片比较少占得内存比较小或者比较常用（比如工具条上一直显示的动态小图标），可以选择用imageNamed: 方式获取图片，但是通过这种方式加到内存中，使用结束，不会自己释放，多次播放动画会造成内存溢出问题。因此，对于大图或经常更换的图，在取图片的时候可以选择imageWithContentsOfFile:方式获取图片，优化内存。
- 使用UIWebView显示图片需要注意显示图片的尺寸与UIWebView尺寸的设置，如果只是为了显示动态图片，可以禁止UIWebView滚动。在显示动态图片的时候，即使是动图的背景处为透明，默认显示出来是白色背景，这个时候需要手动设置UIWebView的透明才能达到显示动图背景透明的效果。

有哪几种方式可以对图片进行缩放，使用 CoreGraphics 缩放时有什么注意事项？

- UIImageView整体拉伸
- UIImage局部拉伸
- UIImage修改大小
- images.xcassets：多亏了Xcode中Asset Catalog的slice和dice，我们不需要代码也能拉伸图片。首先在Xcode中选中图片，然后点击右下角的Show Slicing：
- 图形上下文等比例缩放

XIB与Storyboards的优缺点？

- XIB：在编译前就提供了可视化界面，可以直接拖控件，也可以直接给控件添加约束，更直观一些，而且类文件中就少了创建控件的代码，确实简化不少，通常每个xib对应一个类。
- Storyboard：在编译前提供了可视化界面，可拖控件，可加约束，在开发时比较直观，而且一个storyboard可以有很多的界面，每个界面对应一个类文件，通过storyboard，可以直观地看出整个App的结构。
- XIB：需求变动时，需要修改XIB很大，有时候甚至需要重新添加约束，导致开发周期变长。XIB载入相比纯代码自然要慢一些。对于比较复杂逻辑控制不同状态下显示不同内容时，使用XIB是比较困难的。当多人团队或者多团队开发时，如果XIB文件被发动，极易导致冲突，而且解决冲突相对要困难很多。
- Storyboard：需求变动时，需要修改Storyboard上对应的界面的约束，与XIB一样可能要重新添加约束，或者添加约束会造成大量的冲突，尤其是多团队开发。对于复杂逻辑控制不同显示内容时，比较困难。当多人团队或者多团队开发时，大家会同时修改一个Storyboard，导致大量冲突，解决起来相当困难。

控制器View的加载过程?

当程序访问了控制器的View属性时会先判断控制器的View是否存在，如果存在就直接返回已经存在的view；
如果不存在，就会先调用loadView这个方法；如果控制器的loadView方法实现了，就会按照loadView方法加载自定义的View；
如果控制器的loadView方法没有实现就会判断storyboard是否存在；
如果storyboard存在就会按照storyboard加载控制器的view；如果storyboard不存在，就会创建一个空视图返回。

应用程序的启动流程?

1. 执行Main
2. 执行UIApplicationMain函数。
3. 创建UIApplication对象，并设置UIApplicationMain对象的代理。
 UIApplication的第三个参数就是UIApplication的名称，如果指定为nil，它会默认为UIApplication。
 UIApplication的第四个参数为UIApplication的代理。
4. 开启一个主运行循环。保证应用程序不退出。
5. 加载info.plist。加载配置文件。判断一下info.plist文件当中有没有Main storyboard file base name里面有指定storyboard文件，如果有就去加载info.plist文件，如果没有，那么应用程序加载完毕。

事件传递与响应的完整过程?

在产生一个事件时，系统会将该事件加入到一个由UIApplication管理的事件队列中，
UIApplication会从事件队列中取出最前面的事件，将它传递给先发送事件给应用程序的
主窗口。
主窗口会调用hitTest方法寻找最适合的视图控件，找到后就会调用视图控件的touches
方法来做具体的事情。
当调用touches方法，它的默认做法，就会将事件顺着响应者链条往上传递，
传递给上一个响应者，接着就会调用上一个响应者的touches方法

下列回调机制的理解不正确的是

- A target-action：当两个对象之间有比较紧密的关系时，如视图控制器与其下的某个视图。
- B delegate：当某个对象收到多个事件，并要求同一个对象来处理所有事件时。委托机制必须依赖于某个协议定义的方法来发送消息。
- C NSNotification：当需要多个对象或两个无关对象处理同一个事件时。
- D Block：适用于回调只发生一次的简单任务。
- 参考答案：B

给UIImageView添加圆角

- 最直接的方法就是使用如下属性设置：

```
imgView.layer.cornerRadius = 10;  
// 这一行代码是很消耗性能的  
imgView.clipsToBounds = YES;
```

- 好处是使用简单，操作方便。坏处是离屏渲染（off-screen-rendering）需要消耗性能。对于图片比较多的视图上，不建议使用这种方法来设置圆角。通常来说，计算机系统中CPU、GPU、显示器是协同工作的。CPU计算好显示内容提交到GPU，GPU渲染完成后将渲染结果放入帧缓冲区。
- 简单来说，离屏渲染，导致本该GPU干的活，结果交给了CPU来干，而CPU又不擅长GPU干的活，于是拖慢了UI层的FPS（数据帧率），并且离屏需要创建新的缓冲区和上下文切换，因此消耗较大的性能。
- 给UIImage添加生成圆角图片的扩展API：

```
- (UIImage *)hyb_imageWithCornerRadius:(CGFloat)radius {  
    CGRect rect = (CGRect){0.f, 0.f, self.size};  
  
    UIGraphicsBeginImageContextWithOptions(self.size, NO,  
    UIScreen.mainScreen.scale);  
    CGContextAddPath(UIGraphicsGetCurrentContext(),  
                     [UIBezierPath bezierPathWithRoundedRect:rect  
cornerRadius:radius].CGPath);  
    CGContextClip(UIGraphicsGetCurrentContext());  
  
    [self drawInRect:rect];  
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();  
  
    UIGraphicsEndImageContext();  
  
    return image;  
}
```

然后调用时就直接传一个圆角来处理：

```
imgView.image = [[UIImage imageNamed:@"test"]  
hyb_imageWithCornerRadius:4];
```

这么做就是on-screen-rendering了，通过模拟器->debug->Color Off-screen-rendering看到没有离屏渲染了！（黄色的小圆角没有显示了，说明这个不是离屏渲染了）

- 在画之前先通过UIBezierPath添加裁剪，但是这种不实用

```
- (void)drawRect:(CGRect)rect {  
    CGRect bounds = self.bounds;  
    [[UIBezierPath bezierPathWithRoundedRect:rect cornerRadius:8.0]  
addClip];  
    [self.image drawInRect:bounds];  
}
```

- 通过mask遮罩实现

一个view已经初始化完毕，view上面添加了n个button（可能使用循环创建），除用view的tag之外，还可以采用什么办法来找到自己想要的button来修改Button的值

这个问题有很多种方式，而且不同的使用场景也不一样的。比如说：

- 第一种：如果是点击某个按钮后，才会刷新它的值，其它不用修改，那么不用引用任何按钮，直接在回调时，就已经将接收响应的按钮给传过来了，直接通过它修改即可。
- 第二种：点击某个按钮后，所有与之同类型的按钮都要修改值，那么可以通过在创建按钮时将按钮存入到数组中，在需要的时候遍历查找。

使用drawRect有什么影响？

- drawRect方法依赖Core Graphics框架来进行自定义的绘制，但这种方法主要的缺点就是它处理touch事件的方式：每次按钮被点击后，都会用setNeedsDisplay进行强制重绘；而且不止一次，每次单点事件触发两次执行。这样的话从性能的角度来说，对CPU和内存来说都是欠佳的。特别是如果在我们的界面上有多个这样的UIButton实例。

viewWillLayoutSubviews你总是知道的。

controller layout触发的时候，开发者有机会去重新layout自己的各个subview。

横竖屏切换的时候，系统会响应一些函数，其中 viewWillLayoutSubviews 和 viewDidLayoutSubviews。

```
- (void)viewWillLayoutSubviews {
    [self _shouldRotateToOrientation:(UIInterfaceOrientation)
    [UIApplication sharedApplication].statusBarOrientation];
}

-(void)_shouldRotateToOrientation:
(UIInterfaceOrientation)orientation {

    if (orientation == UIDeviceOrientationPortrait || orientation
    ==
        UIDeviceOrientationPortraitUpsideDown)

    {
        // 竖屏
    }

    else { // 横屏 } }
```

通过上述一个函数就知道横竖屏切换的接口了。

注意：viewWillLayoutSubviews只能用在ViewController里面，在view里面没有响应。

一个tableView是否可以关联两个不同的数据源？

- 当然可以关联多个不同的数据源，但是不能同时使用多个数据源而已。比如，一个列表有两个筛选功能，一个是筛选城市，一个是筛选时间，那么这两个就是两个数据源了。当筛选城市时，就会使用城市数据源；当筛选时间时，就会使用时间数据源。

如何自动计算cell的高度？

- 实现原理：通过数据模型的id作为key，以确保唯一，如何才能保证复用cell时不会出现混乱。在配置完数据后，通过更新约束，得到最后一个控件的frame，就只可以判断cell实际需要的高度，并且缓存下来，下次再获取时，判断是否存在，若存在则直接返回。因此，只会计算一遍

UITableView是如何计算内容高度的？为什么初始化时配置数据时，获取行高的代理方法会调用数据条数次？

- UITableView是继承于UIScrollView的，因此也有contentSize。要得到tableview的contentsize，就需要得到所有cell的高度，从而计算出总高度，才能得到contentsize。因此，在reloadData时，就会调用该代理方法数据条数次。

一个tableView是否可以关联两个不同的数据源？你会怎么处理？

答：首先我们从代码来看，数据源如何关联上的，其实是在数据源关联的代理方法里实现的。

因此我们并不关心如何去关联他，他怎么关联上，方法只是让我返回根据自己的需要去设置如相关的数据源。

因此，我觉得可以设置多个数据源啊，但是有个问题是，你这是想干嘛呢？想让列表如何显示，不同的数据源分区块显示？

给出委托方法的实例，并且说出UITableView的Data Source方法

CocoaTouch框架中用到了大量委托，其中UITableViewDelegate就是委托机制的典型应用，是一个典型的使用委托来实现适配器模式，其中UITableViewDelegate协议是目标，tableview是适配器，实现UITableViewDelegate协议，并将自身设置为talbeview的delegate的对象，是被适配器，一般情况下该对象是UITableViewController。

```
UITableView的Data Source方法有-
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section;
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath;
```

cocoa touch框架

答：iPhone OS 应用程序的基础 Cocoa Touch 框架重用了许多 Mac 系统的成熟模式，但是它更多地专注于触摸的接口和优化。

UIKit 为您提供了在 iPhone OS 上实现图形，事件驱动程序的基本工具，其建立在和 Mac OS X 中一样的 Foundation 框架上，包括文件处理，网络，字符串操作等。

Cocoa Touch 具有和 iPhone 用户接口一致的特殊设计。有了 UIKit，您可以使用 iPhone OS 上的独特的图形接口控件，按钮，以及全屏视图的功能，您还可以使用加速仪和多点触摸手势来控制您的应用。

各色俱全的框架 除了UIKit 外，Cocoa Touch 包含了创建世界一流 iPhone 应用程序需要的所有框架，从三维图形，到专业音效，甚至提供设备访问 API 以控制摄像头，或通过 GPS 获知当前位置。

Cocoa Touch 既包含只需要几行代码就可以完成全部任务的强大的 Objective-C 框架，也在需要时提供基础的 C 语言 API 来直接访问系统。这些框架包括：

Core Animation：通过 Core Animation，您就可以通过一个基于组合独立图层的简单的编程模型来创建丰富的用户体验。

Core Audio：Core Audio 是播放，处理和录制音频的专业技术，能够轻松为您的应用程序添加强大的音频功能。

Core Data：提供了一个面向对象的数据管理解决方案，它易于使用和理解，甚至可处理任何应用或大或小的数据模型。

功能列表：框架分类

下面是 Cocoa Touch 中一小部分可用的框架：

音频和视频：Core Audio , OpenAL , Media Library , AV Foundation

数据管理：Core Data , SQLite

图形和动画：Core Animation , OpenGL ES , Quartz 2D

网络：Bonjour , WebKit , BSD Sockets

用户应用：Address Book , Core Location , Map Kit , Store Kit

xib文件的构成为哪3个图标？都具有什么功能。

File's Owner 是所有 nib 文件中的每个图标，它表示从磁盘加载 nib 文件的对象；

First Responder 就是用户当前正在与之交互的对象；

View 显示用户界面；完成用户交互；是 UIView 类或其子类。

简述应用程序按Home键进入后台时的生命周期,和从后台回到前台时的生命周期? 应用程序:

```
-[AppDelegate application:willFinishLaunchingWithOptions:]  
-[AppDelegate application:didFinishLaunchingWithOptions:]  
-[AppDelegate applicationDidBecomeActive:]
```

退到后台:

```
-[AppDelegate applicationWillResignActive:]  
-[AppDelegate applicationDidEnterBackground:]
```

回到前台:

```
-[AppDelegate applicationWillEnterForeground:]  
-[AppDelegate applicationDidBecomeActive:]
```

ViewController之间,

加载页面:

```
-[mainViewController viewDidLoad] -[mainViewController  
viewWillAppear:] -[mainViewController viewWillLayoutSubviews] -  
[mainViewController viewDidLayoutSubviews] -[mainViewController  
viewDidAppear:]
```

退出当前页面:

```
-[mainViewController viewWillDisappear:]  
-[mainViewController viewDidDisappear:]
```

返回之前页面:

```
-[mainViewController viewWillAppear:]  
-[mainViewController viewWillLayoutSubviews]  
-[mainViewController viewDidLayoutSubviews]  
-[mainViewController viewDidAppear:]
```

是否使用Core Text或者Core Image等? 如果使用过, 请谈谈你使用Core Text或者Core Image的体验。

CoreText

- 随意修改文本的样式
- 图文混排(纯C语言)
- 国外:Niumb

Core Image(滤镜处理)

- * 能调节图片的各种属性(对比度, 色温, 色差等)

分析一下使用手机获取验证码注册账号的实现逻辑(给了一个示例图), 发送到手机的验证码超过60秒钟后重新发送

- 定义一个label属性，赋值为60秒，再定义一个count 设置一个timer 每次减少一秒 把 count-- 再把count的值拼接到label上 当count == 0 的时候 再显示重新发送

你做iphone开发时候,有哪些传值方式,view和view之间是如何传值的?

block, target-action ,代理,属性

有哪几种手势通知方法、写清楚方法名?

```
- (void)touchesBegan:(NSSet)touches withEvent:(UIEvent*)event;
- (void)touchesMoved:(NSSet)touches withEvent:(UIEvent*)event;
- (void)touchesEnded:(NSSet)touches withEvent:(UIEvent*)event;
- (void)touchesCancelled:(NSSet)touches withEvent:(UIEvent*)event;
```

(11) 报错警告调试

你在实际开发中，有哪些手机架构与性能调试经验

- 刚接手公司的旧项目时，模块特别多，而且几乎所有的代码都写在控制器里面，比如 UI控件代码、网络请求代码、数据存储代码
- 接下来采取MVC模式进行封装、重构
 - 自定义UI控件封装内部的业务逻辑
 - 封装网络请求工具类(降低耦合)
 - 封装数据存储工具类

BAD_ACCESS在什么情况下出现?

这种问题是经常遇到的，在开发时经常会出现BAD_ACCESS。原因是访问了野指针，比如访问已经释放对象的成员变量或者发消息、死循环等。

如何调试BAD_ACCESS错误?

出现BAD_ACCESS错误，通常是访问了野指针，比如访问了已经释放了的对象。快速定位问题的步骤有：

1. 重写对象的respondsToSelector方法，先找到出现EXCEBADACCESS前访问的最后一个object
2. 设置Enable Zombie Objects
3. 设置全局断点快速定位问题代码所在行，接收所有的异常

4. Xcode 7 已经集成了 BAD_ACCESS 捕获功能: Address Sanitizer, 与步骤2一样设置
5. analyze 也行 (不一定管用)

什么时候会报 unrecognized selector 异常?

- 当调用对象 (子类, 各级父类) 中不含有对应方法的时候, 并且依旧没有给出“消息转发”的具体方案的时候, 程序在运行时会 crash 并抛出 unrecognized selector 异常
- objective-c 中的每个方法在运行时会被转为消息发送 objc_msgSend(reciver, selector)
- 例如 [person say] 就会被转化为 objc_msgSend(person, @selector(say))
- 运行时会根据对象(reciever) 的 isa 指针找到该对象所对应的类, 然后会依次在对应的类, 父类, 爷爷类, 根类中找对应的方法

下面只讲述对象方法的解析过程:

- 第一步: +(BOOL)resolveInstanceMethod:(SEL)sel 实现方法, 指定是否动态添加方法。若返回 NO, 则进入下一步, 若返回 YES, 则通过 class_addMethod 函数动态地添加方法, 消息得到处理, 此流程完毕。
- 第二步: 在第一步返回的是 NO 时, 就会进入 - (id)forwardingTargetForSelector:(SEL)aSelector 方法, 这是运行时给我们的第二次机会, 用于指定哪个对象响应这个 selector。不能指定为 self。若返回 nil, 表示没有响应者, 则会进入第三步。若返回某个对象, 则会调用该对象的方法。
- 第三步: 若第二步返回的是 nil, 则我们首先要通过 - (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector 指定方法签名, 若返回 nil, 则表示不处理。若返回方法签名, 则会进入下一步。
- 第四步: 当第三步返回方法方法签名后, 就会调用 - (void)forwardInvocation:(NSInvocation *)anInvocation 方法, 我们可以通过 anInvocation 对象做很多处理, 比如修改实现方法, 修改响应对象等
- 第五步: 若没有实现 - (void)forwardInvocation:(NSInvocation *)anInvocation 方法, 那么会进入 - (void)doesNotRecognizeSelector:(SEL)aSelector 方法。若我们没有实现这个方法, 那么就会 crash, 然后提示打不到响应的方法。到此, 动态解析的流程就结束了。

有哪些常见的 Crash 场景?

- 访问了僵尸对象
- 访问了不存在的方法
- 数组越界
- 在定时器下一次回调前将定时器释放, 会 Crash

lldb (gdb) 常用的调试命令?

- p 输出基本类型 // p (int)[[[self view] subviews] count]
- po 用于输出 Objective-C 对象 // po [self view]

- `expr` 可以在调试时动态执行指定表达式，并将结果打印出来。常用于在调试过程中修改变量的值。`//源代码中 a = 1 ; expr a=2` 输出结果：`(int) $0 = 2`

如果一个函数10次中有7次正确，3次错误，问题可能出现在哪里？

这样的问题通过应聘者的分析，可以知道应聘者的功底如何。很多人的回答会是很简单的，没有从多方面去分析。这样的问题也是很有意义的，在项目开发中所产生的bug，有的时候会出现这样的情况，而代码量比较大且业务比较复杂时，通过其他工具并不能分析出来是什么bug，但是我们却可以根据出现的频率推测。笔者把这个问题当作测试部反馈过来的bug描述问题来分析一下。

参考答案：

从问题描述可知，bug不会必现的，因此无法直接定位出错之处。从以下角度出现来分析可能出错之处：

1. 因出错并不是崩溃，因此没有错误日志可看。第一步就是分析函数中的所有分支，是否在语法上存在可能缺少条件的问题。所以，检查所有的分支，确保每个分支执行的结果是正确的
2. 检测函数的参数，保证必传参数不能为空，若为空应该抛出异常。因此，用断言检测参数的正确性是很重要的。
3. 检测函数中每个分支所调用的函数返回结果是正确的，其实就是一个递归的过程（步骤1、2）

你一般是如何调试Bug的？

这个问题看起来很笼统，但又一针见血。通过应聘者的回答，可很直观地看出这个应聘者处理bug的能力，以及其解决问题的思维。

参考答案：

Bug分为测试中的Bug和线上的Bug：

- 线上Bug：项目使用了友盟统计，因此会有崩溃日志，通过解析dysm可以直接定位到大部分bug崩溃之处。解决线上bug需要从主干拉一个新的分支，解决bug并测试通过后，再合并到主干，然后上线。若是多团队开发，可以将fix bug分支与其他团队最近要上线的分支集成，然后集成测试再上线。
- 测试Bug：根据测试所反馈的bug描述，若语义不清晰，则直接找到提bug人，操作给开发人员看，最好是可以通过复现。解决bug时，若能根据描述直接定位bug出错之处，则好处理；若无法直观定位，则根据bug类型分几种处理方式，比如崩溃的bug可以通过instruments来检测、数据显示错误的bug，则需要阅读代码一步步查看逻辑哪里写错。对于开发中出现的崩溃或者数据显示不正常，那就需要根据经验或者相关工具来检测可能出错之处。当然，团队内沟通解决是最好的。

获取一台设备唯一标识的方法有哪些？

- 现在常用的是用UUID + keychain结合来实现这个需求。
- UUID是Universally Unique Identifier的缩写，中文意思是通用唯一识别码。它是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识

资讯的指定。这样，每个人都可以建立不与其它人冲突的UUID。在此情况下，就不需考虑数据库建立时的名称重复问题。苹果公司建议使用UUID为应用生成唯一标识字符串。

```
//获取一个UUID
- (NSString*)uuid {
    CFUUIDRef uuid = CFUUIDCreate( nil );
    CFStringRef uuidString = CFUUIDCreateString( nil, uuid );
    NSString * result = (NSString
*)CFBridgingRelease(CFStringCreateCopy( NULL, uuidString));
    CFRelease(uuid);
    CFRelease(uuidString);
    return result;
}
```

- 现在我们获取到了一个UUID，虽然这个标识是唯一的，但是这样还是无法保证每一次的唯一性，因为当你每次调用这个方法或者把应用卸载了，UUID会重新生成一个不同的。这个时候keychain就起到了作用。
- 所以整个逻辑是这样的：先从keychain取UUID，如果能取到，就用这个比对，如果取不到就重新生成一个保存起来。keychain独立在App之外，是和系统统一等级的，所以你不用担心它挂掉。
- keychain是苹果公司Mac OS中的密码管理系统。它在Mac OS 8.6中被导入，并且包括在了所有后续的Mac OS版本中，包括Mac OS X。一个钥匙串可以包含多种类型的数据：密码（包括网站，FTP服务器，SSH帐户，网络共享，无线网络，群组软件，加密磁盘镜像等），私钥，电子证书和加密笔记等。iOS端同样有个keychain帮助我们管理这些敏感信息。
- 使用过keychain保存过账号密码的童鞋应该对这个工具非常了解，在这里不做过多解释。使用keychain需要导入Security.framework和KeychainItemWrapper.h/.m，KeychainItemWrapper.h/.m搜一下可以下载下来，拖入工程中。保存UUID代码如下：

```
- (void)saveUuidWithKeyChain {
    KeychainItemWrapper *keychainItem = [[KeychainItemWrapper
alloc]

initWithIdentifier:@"UUID" accessGroup:@"com.xxx.www"];
    NSString *strUUID = [keychainItem objectForKey:
(id)kSecValueData];
    if (strUUID == nil || [strUUID isEqualToString:@""])
    {
        [keychainItem setObject:[self uuid] forKey:
(id)kSecValueData];
    }
}
```

注：这个方法中accessGroup:这个参数如果一些App设置相同的话，是可以共享的。

- 从keychain获取UUID的方法如下：

```
- (NSString *)getKeychain {
    KeychainItemWrapper *keychainItem = [[KeychainItemWrapper alloc]

initWithIdentifier:@"UUID" accessGroup:@"com.xxx.www"];
    NSString *strUUID = [keychainItem objectForKey:(id)kSecValueData];
    return strUUID;
}
```

- 至此，基本上唯一标识的几个方法算是写完了，大家可以测试一下，卸载应用再重新装，从keychain读取的UUID还是和之前一样。
- 但这里有个不确定因素，就是手机系统恢复出厂设置或者抹掉所有数据的话，这个方法也可能不起作用了，因为它是依靠钥匙串在生存，钥匙串挂掉的话它也就失效了。

你一般是怎么用 Instruments 的？

- 这个问题也就是考察下你经验如何了，Instruments里面工具很多，也没必要逐一说明，挑几个常用的说下就好
- 参考答案：
- Time Profiler：性能分析
- Zombies：检查是否访问了僵尸对象，但是这个工具只能从上往下检查，不智能
- Allocations：用来检查内存，写算法的那批人也用这个来检查
- Leaks：检查内存，看是否有内存泄露

你一般是如何调试 Bug 的？

- 查看异常报告
- 配置相关环境，重现bug
- 代码检查
- 用测试案例来捕获bug
- 可以请同事一同来审查问题，有些时候当局者迷，旁观者清。

如何对iOS设备进行性能测试？

Profile-> Instruments ->Time Profiler 进行性能测试！

测试iOS版的App注意事项分享以下几点：

1.app使用过程中，接听电话。可以测试不同的通话时间的长短，对于通话结束后，原先打开的app的响应，比如是否停留在原先界面，继续操作时的相应速度等。

2.app使用过程中，有推送消息时，对app的使用影响

3.设备在充电时，app的响应以及操作流畅度

4.设备在不同电量时(低于10%，50%，95%)，app的响应以及操作流畅度

5.意外断电时，app数据丢失情况

6.网络环境变化时，app的应对情况如何：是否有适当提示？从有网络环境到无网络环境时，app的反馈如何？从无网络环境回到有网络环境时，是否能自动加载数据，多久才能开始加载数据

7.多点触摸的情况

8.跟其他app之间互相切换时的响应

9.进程关闭再重新打开的反馈

10.IOS系统语言环境变化时

(12) 第三方框架及其管理

使用过CocoaPods吗？它是什么？CocoaPods的原理？

- CocoaPod是一个第三方库的管理工具，用来管理项目中的第三方框架。
- 在终端中进入（cd命令）你项目所在目录，然后在当前目录下，利用vim创建 Podfile，运行：\$ vim Podfile
- 然后在Podfile文件中输入以下文字：
 - platform :ios, '9.3'
 - pod "AFNetworking", "~> 2.0"
 - 然后保存退出。vim环境下，保存退出命令是:wq
- 这时候，你会发现你的项目目录中，出现一个名字为Podfile的文件，而且文件内容就是你刚刚输入的内容。
这时候，你就可以利用CocoPods下载AFNetworking类库了，运行以下命令：\$ pod install

用cocoapods管理第三方框架的时候我想改版本，怎么办到？

- 可以直接或者终端打开Podfile，修改Podfile文件中第三方框架的版本

集成三方框架有哪些方法

- cocoapods
- framework
- 直接下载源码拖进项目用

SDWebImage的原理实现机制，如何解决TableView卡的问题？

SDWebImage内部实现过程（建议画图记住）

image image

- 入口 setImageWithURL:placeholderImage:options: 会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。
- 进入 SDWebImageManager-downloadWithURL:delegate:options:userInfo:，交给 SDImageCache 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:。
- 先从内存图片缓存查找是否有图片，如果内存中已经有图片缓存， SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo: 到 SDWebImageManager。
- SDWebImageManagerDelegate 回调 webImageManager:didFinishWithImage: 到 UIImageView+WebCache 等前端展示图片。
- 如果内存缓存中没有，生成 NSInvocationOperation 添加到队列开始从硬盘查找图片是否已经缓存。
- 根据 URLKey 在硬盘缓存目录下尝试读取图片文件。这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调 notifyDelegate:。
- 如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中（如果空闲内存过小，会先清空内存缓存）。SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo:。进而回调展示图片。
- 如果从硬盘缓存目录读取不到图片，说明所有缓存都不存在该图片，需要下载图片，回调 imageCache:didNotFindImageForKey:userInfo:。
- 共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。
- 图片下载由 NSURLConnection 来做，实现相关 delegate 来判断图片下载中、下载完成和下载失败。
- connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。
- connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。
- 图片解码处理在一个 NSOperationQueue 完成，不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，最好也在里面完成，效率会好很多。
- 在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成， imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。
- imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。
- 通知所有的 downloadDelegates 下载完成，回调给需要的地方展示图片。
- 将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。写文件到硬盘也在以单独 NSInvocationOperation 完成，避免拖慢主线程。

- SDImageCache 在初始化的时候会注册一些消息通知，在内存警告或退到后台的时候清理内存图片缓存，应用结束的时候清理过期图片。
- SDWI 也提供了 UIButton+WebCache 和 MKAnnotationView+WebCache，方便使用。
- SDWebImagePrefetcher 可以预先下载图片，方便后续使用。
- 如何解决tableView卡顿问题，前面也提了很多方案。通过设置最大并发数，设置当前页的cell，而不是把所有cell一次性设置完，以及数据图片的三级缓存，直接保存在内存中和沙盒缓存中进行读取。降低网络请求的次数，不仅节约用户流量，也会保证tableView滑动的流畅性

SDWebImage怎样实现图片的缓存机制的？

- 图片的缓存，内存缓存，沙盒缓存，操作缓存，以tableViewController为例：
- 每次cell需要显示，都需要重新调用 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath { } 方法
- 每次调用tableView显示行的数据源方法时，如果需要从网络加载图片，就需要将加载图片这样的耗时操作放在子线程上执行，从网络上下载的图片可以以键值对的形式保存在定义的可变字典中，将每张图片的唯一的路径作为键，将从网络下载下来的图片作为值，保存在内存缓存中，这样每次滑动tableView cell重用时就直接判断内存缓存中有没有需要的图片，如果有就不需要再次下载，在没有出现内存警告或者程序员手动清理内存缓存时，就直接从内存缓存中获取图片。
- 为了每次退出程序，再次进入程序时，不浪费用户的流量，需要将第一次进入程序时加载的图片保存在本地沙盒缓存文件中，在沙盒中保存的图片数据没有被改变之前，下次开启程序就直接从沙盒的缓存文件中读取需要显示的图片，并将沙盒缓存文件夹(Cache)中保存的图片保存到内存缓存中，这样用户每次滑动tableView cell重用时直接从内存缓存中读取而不是从沙盒中读取，节约时间。

你用过哪些第三方框架？

- 使用过一些第三方框架，例如AFN, SDWebimage, FMDB, MBProgressHUD, Masonry。

使用 AFNetworking 做过断点续传吗？

- 断点续传的主要思路：
 - 检查服务器文件信息
 - 检查本地文件
 - 如果比服务器文件小，断点续传，利用 HTTP 请求头的 Range 实现断点续传
 - 如果比服务器文件大，重新下载
 - 如果和服务器文件一样，下载完成

使用第三方框架的时候有没有仔细研读过它们的源码 (AFNetworking)？AFN底层原理分析。

- AFNetworking主要是对NSURLSession和NSURLCollection（iOS9.0废弃）的封装，其中主要有以下类：

- AFHTTPRequestOperationManager :内部封装的是 NSURLConnection,负责发送网络请求,使用最多的一个类. (3.0废弃)
- AFHTTPSessionManager :内部封装是 NSURLSession ,负责发送网络请求,使用最多的一个类.
- AFHTTPRequestOperationManager 和 AFHTTPSessionManager :定义的 API(方法名称)是一模一样,没有任何区别.
- AFNetworkReachabilityManager :实时监测网络状态的工具类.当前的网络环境发生改变之后,这个工具类就可以检测到.
- AFSecurityPolicy :网络安全的工具类,主要是针对 HTTPS 服务.
- AFURLRequestSerialization :序列化工具类,基类.上传的数据转换成JSON格式 (AFJSONRequestSerializer).使用不多.
- AFURLResponseSerialization :反序列化工具类;基类.使用比较多:
- AFJSONResponseSerializer; JSON解析器,默认的解析器.
- AFHTTPResponseSerializer; 万能解析器; JSON和XML之外的数据类型,直接返回二进制数据.对服务器返回的数据不做任何处理.
- AFXMLParserResponseSerializer; XML解析器;

AFN默认超时时长是多少啊?

- AFN的默认超时时长是60s.

举出5个以上你所熟悉的iOS sdk库有哪些和第三方库有哪些?

1> iOS-sdk:

```
Foundation.framework,
CoreGraphics.framework,
UIKit.framework,
MediaPlayer.framework,
CoreAudio.framework
```

2> 第三方库:

1.AFNetworking

2.SDWebImage

3.masonry

4.MBProgressHUD

3> 框架分类:

音频和视频

```
Core Audio
OpenAL
Media Library
AVFoundation
```

数据管理

Core Data
SQLite

图片和动画

Core Animation
OpenGL ES
Quartz 2D

网络

Bonjour
WebKit
BSD Sockets

用户应用

Address Book
Core Location
Map Kit
Store Kit

第三方API是怎么用的？

- 大公司的开放API,github上面别人发布的框架用第三方API,在官方文档都有说明,按照官方文档一步一步做参考官方提供的示例程序。先自己创建一个工程试试,等熟悉了,在使用到项目中

实现过框架或者库以供他人使用么?如果有,请谈一谈构建框架或者库时候的经验;如果没有,请设想和设计框架的public的API,并指出大概需要如何做、需要注意哪些问题,以使人人更容易地使用你的框架。

从以下角度出发来思考和设计公共框架:

- 确保外部调用简单,且保证有详细的头文件注释说明。
- 确保API编码规范,保证风格统一。
- 确保API易扩展,可以考虑预留参数
- 确保没有外部依赖或者依赖要尽可能的少,以保证公共库的纯洁(原则上不能有外部依赖)
- 确保易维护,不存在冗余API

简述下苹果的开发框架?

Foundation

提供OC的基础类(像NSObject)、基本数据类型等

UIKit

创建和管理应用程序的用户界面

QuartzCore

提供动画特效以及通过硬件进行渲染的能力

CoreGraphics

提供2D绘制的基于C的API

SystemConfiguration

检测当前网络是否可用和硬件设备状态

AVFoundation

提供音频录制和回放的底层API，同时也负责管理音频硬件

CFNetwork

访问和配置网络，像HTTP、FTP和Bonjour Services

CoreFoundation

提供抽象的常用数据类型，如Unicode strings、XML、URL等

CoreLocation

使用GPS和WIFI获取位置信息

(13) 绘图与动画

CAAnimation的层级结构

- CAPropertyAnimation是CAAnimation的子类，也是个抽象类，要想创建动画对象，应该使用它的两个子类：CABasicAnimation和CAKeyframeAnimation
- 属性解析：keyPath：通过指定CALayer的一个属性名称为keyPath(NSString类型)，并且对CALayer的这个属性的值进行修改，达到相应的动画效果。比如，指定@”position”为keyPath，就修改CALayer的position属性的值，以达到平移的动画效果
- CABasicAnimation， CAPropertyAnimation的子类
- 属性解析：
- fromValue：keyPath相应属性的初始值
- toValue：keyPath相应属性的结束值
- 随着动画的进行，在长度为duration的持续时间内，keyPath相应属性的值从fromValue渐渐地变为toValue。如果fillMode=kCAFillModeForwards和removedOnCompletion=NO，那么在动画执行完毕后，图层会保持显示动画执行后的状态。但在实质上，图层的属性值还是动画执行前的初始值，并没有真正被改变。比如，

CALayer的position初始值为(0,0), CABasicAnimation的fromValue为(10,10), toValue为(100,100), 虽然动画执行完毕后图层保持在(100,100)这个位置, 实质上图层的position还是为(0,0)

- CAKeyframeAnimation, CPropertyAnimation的子类, 跟CABasicAnimation的区别是: CABasicAnimation只能从一个数值(fromValue)变到另一个数值(toValue), 而CAKeyframeAnimation会使用一个NSArray保存这些数值
- 属性解析:
- values: 就是上述的NSArray对象。里面的元素称为”关键帧”(keyframe)。动画对象会在指定的时间(duration)内, 依次显示values数组中的每一个关键帧
- path: 可以设置一个CGPathRef\CGMutablePathRef, 让层跟着路径移动。path只对CALayer的anchorPoint和position起作用。如果你设置了path, 那么values将被忽略
- keyTimes: 可以为对应的关键帧指定对应的时间点, 其取值范围为0到1.0, keyTimes中的每一个时间值都对应values中的每一帧。当keyTimes没有设置的时候, 各个关键帧的时间是平分的
- CABasicAnimation可看做是最多只有2个关键帧的CAKeyframeAnimation
- CAAnimationGroup, CAAnimation的子类, 可以保存一组动画对象, 将CAAnimationGroup对象加入层后, 组中所有动画对象可以同时并发运行
- 属性解析:
- animations: 用来保存一组动画对象的NSArray
默认情况下, 一组动画对象是同时运行的, 也可以通过设置动画对象的beginTime属性来更改动画的开始时间
- CATransition, CAAnimation的子类, 用于做转场动画, 能够为层提供移出屏幕和移入屏幕的动画效果。iOS比Mac OS X的转场动画效果少一点
- UINavigationController就是通过CATransition实现了将控制器的视图推入屏幕的动画效果
- 属性解析:
- type: 动画过渡类型
- subtype: 动画过渡方向
- startProgress: 动画起点(在整体动画的百分比)
- endProgress: 动画终点(在整体动画的百分比)
- UIView动画
- UIKit直接将动画集成到UIView类中, 当内部的一些属性发生改变时, UIView将为这些改变提供动画支持
执行动画所需要的工作由UIView类自动完成, 但仍要在希望执行动画时通知视图, 为此需要将改变属性的代码放在[UIView beginAnimations:nil context:nil]和[UIView commitAnimations]之间

- Block动画
- 帧动画

PNG PNG

谈谈你对Core Graphic 绘图的了解？

- Core Graphics是基于C的API，可以用于一切绘图操作
- Core Graphics 和Quartz 2D的区别
 - quartz是一个通用的术语，用于描述在IOS和MAC OS X ZHONG 整个媒体层用到的多种技术 包括图形、动画、音频、适配。
 - Quartz 2D 是一组二维绘图和渲染API，Core Graphic会使用到这组API
 - Quartz Core 专指Core Animation用到的动画相关的库、API和类
- Core Graphics是高度集成于UIView和其他UIKit部分的。
- Core Graphics数据结构和函数可以通过前缀CG来识别。
- 系统拥有坐标系，如320 480 硬件有retain屏幕和非retain屏：如320 480、640 960 Core Graphics 使用的是系统的坐标系来绘制图片。在分辨率为640 960手机上绘制图片时，实际上Core Graphics 的坐标是320*480。这个时候每个坐标系上的点，实际上拥有两个像素。
- 视图可以通过子视图、图层或实现drawRect: 方法来表现内容，如果说实现了 drawRect: 方法，那么最好就不要混用其他方法了，如图层和子视图。自定义绘图大部分是由UIKit或者Core Graphics来实现的。
- Core Graphics的优点：快速、高效，减小应用的文件大小。同时可以自由地使用动态的、高质量的图形图像。使用Core Graphics，可以创建直线、路径、渐变、文字与图像等内容，并可以做变形处理
- 2D绘图一般可以拆分成以下几个操作: 线条，路径，文本，图片，渐变
- 由于像素是依赖于目标的，所以2D绘图并不能操作单独的像素，我们可以从上下文（Context）读取它。
绘图就好比在画布上拿着画笔机械的进行画画，通过制定不同的参数来进行不同的绘制。
<http://www.tuicool.com/articles/jIJzMf>
<http://blog.csdn.net/mangosnow/article/details/37054765>

Core Animation(核心动画)?

- CoreAnimation也就是核心动画, 是一组非常强大的动画处理API, 可以使用少量的代码做出绚丽的效果, 是直接作用在CALayer上的, 并非UIView, 并且Core Animation的动画执行过程都是在后台操作, 不会阻塞主线程.
- 所有动画都是作用在CALayer上的, 当把动画添加到Layer上, 是不直接修改它的属性, Core Animation维护了两个平行layer的层次结构, 模型层树可以看到Layer的状态, 表示层树则是动画正在表现的值的近似。
- Core Animation的使用步骤:
 - 使用它需要先添加QuartzCore.framework框架和引入主头文件<QuartzCore/QuartzCore.h>(iOS7.0+ 不需要)
 - 初始化一个CAAnimation对象, 并设置一些动画相关属性
 - 通过调用CALayer的addAnimation:forKey:方法增加CAAnimation对象到CALayer中, 这样就能开始执行动画了
 - 通过调用CALayer的removeAnimationForKey:方法可以停止CALayer中的动画

转场动画?

- CATransition-转场动画, 作为CAAnimation的子类, 用于做转场动画, 能够为层提供移出屏幕和移入屏幕的动画效果。iOS比Mac OS X的转场动画效果少一点。
UINavigationController就是通过CATransition实现了将控制器的视图推入屏幕的动画效果.

Cocoa Touch提供了哪几种Core Animation过渡类型?

- Cocoa Touch 提供了 4 种 Core Animation 过渡类型, 分别为: 交叉淡化、推挤、显示和覆盖。

使用UIView的动画函数, 实现转场动画

- 单视图:
`+ (void)transitionWithView:(UIView*) view duration:(NSTimeInterval)duration options:(UIViewControllerAnimatedOptions)options animations:(void (^)(void))animations completion:(void (^)(BOOL finished))completion;`
- 双视图:
`+ (void)transitionFromView:(UIView)fromView toView:(UIView)toView duration:(NSTimeInterval)duration options:(UIViewControllerAnimatedOptions)options completion:(void (^)(BOOL finished))completion;`

一个动画怎么实现？

- 以转场动画为例：
 - 创建CATransition对象
CATransition *animation = [CATransition animation];
 - 设置运动时间(即动画时间)
animation.duration = DURATION;
 - 设置运动type(类型)
animation.type = type;
if (subtype != nil) { //设置子类 (和type配合使用, 指定运动的方向)
animation.subtype = subtype;}
 - 设置运动速度(动画的运动轨迹, 用于变化起点和终点之间的插值计算,形象点说它决定了动画运行的节奏,比如是均匀变化(相同时间变化量相同)还是先快后慢,先慢后快还是先慢再快再慢)
animation.timingFunction = UIViewAnimationOptionCurveEaseInOut;
 - 将动画添加到view的Layer层
[view.layer addAnimation:animation forKey:@"animation"];
 - 动画类型如下：
typedef enum : NSUInteger {
Push, //推挤
Cube, //立方体
} AnimationType;

说说Core Animation是如何开始和结束动画的

不是很清楚题目的真正要求，是想知道核心动画的哪些知识点。如何开始和结束动画，这核心动画有很多种，每种动画还有很大的区别。

参考答案：

动画的开始和结束都可以通过CAMediaTiming协议来处理，核心动画的基类是遵守了CAMediaTiming协议的，可以指定动画开始时间、动画时长、动画播放速度、动画在完成时的行为（停留在结束处、动画回到开始处、动画完成时移除动画）。

动画有基本类型有哪几种；表视图有哪几种基本样式。

- 动画有两种基本类型：一种为UIView动画,又称隐式动画,动画后frame的数值发生了变化.另一种是CALayer动画,又称显示动画,动画后模型层的数据不会发生变化,图形回到原来的位置。
- UITableViewCellStylePlain：普通样式

- UITableViewCellStyleGrouped:分组样式
- UITableViewCellCellStyleDefault:Default样式：左边一个显示图片的imageView，一个标题textLabel，没有detailTextLabel。
- UITableViewCellCellStyleSubtitle:Subtitle样式：左边一个显示图片的imageView，上边一个主标题textLabel，一个副标题detailTextLabel。主标题字体大且加黑，副标题字体小在主标题下边。
- UITableViewCellCellStyleValue1:Value1样式：左边一个显示图片的imageView，左边一个主标题textLabel，右边一个副标题detailTextLabel，主标题字体比较黑。
- UITableViewCellCellStyleValue2:Value2样式：左边一个主标题textLabel字体偏小，挨着右边一个副标题detailTextLabel，字体大且加黑。

CADisplayLink

- CADisplayLink是一种以屏幕刷新频率触发的时钟机制，每秒钟执行大约60次左右
- CADisplayLink是一个计时器，可以使绘图代码与视图的刷新频率保持同步，而NSTimer无法确保计时器实际被触发的准确时间
- 使用方法：
 定义CADisplayLink并制定触发调用方法
 将显示链接添加到主运行循环队列

Quatrz 2D的绘图功能的三个核心概念是什么并简述其作用。

上下文：主要用于描述图形写入哪里；

路径：是在图层上绘制的内容；

状态：用于保存配置变换的值、填充和轮廓，alpha 值等。

(14) 数据存储

sqlite中插入特殊字符的方法和接收到处理方法。

除其他的都是在特殊字符前面加“/”，而'->"。方法：keyWord = keyWord.replace("/", "//");

什么是NSManagedObject模型？

NSManagedObject是NSObject的子类，Core Date的重要组成部分。是一个通用类，实现了Core Date模型层所需的基本功能，用户可以通过NSManagedObject建立自己的数据模型。

你实现过多线程的Core Data么？

NSPersistentStoreCoordinator, NSManagedObjectContext和NSManagedObject中的哪些需要在线程中创建或者传递？你是用什么样的策略来实现的？

1> CoreData是对SQLite数据库的封装

2> coreData中有三个对象是必须掌握的，

NSManagedObject :只要定义一个类继承于该类就会创建一张与之对应的表，也就是一个继承于该类的类就对应一张表。每一个通过继承该类创建出来的对象，都是该类对应的表中的一条数据

NSManagedObjectContext：用于操作数据库，只要有类它就能对数据库的表进行增删改查

NSPersistentStoreCoordinator：决定数据存储的位置（SQLite/XML/其它文件中）

3> Core data本身并不是一个并发安全的架构所以在多线程中实现Core data会有问题.问题在于

>2.1 CoreData中的NSManagedObjectContext在多线程中不安全

>2.2如果想要多线程访问CoreData的话，最好的方法是一个线程一个
NSManagedObjectContext

>2.3每个NSManagedObjectContext对象实例都可以使用同一个

NSPersistentStoreCoordinator实例，这是因为NSManagedObjectContext会在使用NSPersistentStoreCoordinator前上锁

简单描述下客户端的缓存机制？

- 缓存可以分为：内存数据缓存、数据库缓存、文件缓存
 - 每次想获取数据的时候
 - 先检测内存中有无缓存
 - 再检测本地有无缓存(数据库\文件)
 - 最终发送网络请求
 - 将服务器返回的网络数据进行缓存（内存、数据库、文件），以便下次读取

什么是序列化和反序列化，用来做什么

- 序列化把对象转化为字节序列的过程
- 反序列化把直接序列恢复成对象
- 把对象写到文件或者数据库中，并且读取出来

OC中实现复杂对象的存储

- 遵循NSCoding协议，实现复杂对象的存储，实现该协议后可以对其进行打包或者解包，转化为NSDate

iOS中常用的数据存储方式有哪些？

1. 数据存储有四种方案，NSUserDefaults,KeyChain,File,DB.
2. 其中File有三种方式：plist,Archiver,Stream
3. DB包括core Data和FMDB

说一说你对SQLite的认识

- SQLite是目前主流的嵌入式关系型数据库，其最主要的特点就是轻量级、跨平台，当前很多嵌入式操作系统都将其作为数据库首选。
- 虽然SQLite是一款轻型数据库，但是其功能也绝不亚于很多大型关系数据库。
- 学习数据库就要学习其相关的定义、操作、查询语言，也就是大家日常说得SQL语句。和其他数据库相比，SQLite中的SQL语法并没有太大的差别，因此这里对于SQL语句的内容不会过多赘述，大家可以参考SQLite中其他SQL相关的内容，这里还是重点讲解iOS中如何使用SQLite构建应用程序。先看一下SQLite数据库的几个特点：
 - 基于C语言开发的轻型数据库
 - 在iOS中需要使用C语言语法进行数据库操作、访问（无法使用ObjC直接访问，因为libsqlite3框架基于C语言编写）
 - SQLite中采用的是动态数据类型，即使创建时定义了一种类型，在实际操作时也可以存储其他类型，但是推荐建库时使用合适的类型（特别是应用需要考虑跨平台的情况时）
 - 建立连接后通常不需要关闭连接（尽管可以手动关闭）
- 在iOS中操作SQLite数据库可以分为以下几步（注意先在项目中导入libssqlite3框架）：
 - 打开数据库，利用sqlite3_open()打开数据库会指定一个数据库文件保存路径，如果文件存在则直接打开，否则创建并打开。打开数据库会得到一个sqlite3类型的对象，后面需要借助这个对象进行其他操作。
 - 执行SQL语句，执行SQL语句又包括有返回值的语句和无返回值语句。
 - 对于无返回值的语句（如增加、删除、修改等）直接通过sqlite3_exec()函数执行；

- 对于有返回值的语句则首先通过sqlite3_prepare_v2()进行sql语句评估（语法检测），然后通过sqlite3_step()依次取出查询结果的每一行数据，对于每行数据都可以通过对应的sqlite3_column_类型()方法获得对应列的数据，如此反复循环直到遍历完成。当然，最后需要释放句柄。

说一说你对FMDB的认识

- FMDB是一个处理数据存储的第三方框架，框架是对sqlite的封装，整个框架非常轻量级但又不失灵活性，而且更加面向对象。FMDB有如下几个特性：
 - FMDB既然是对于libsqlite3框架的封装，自然使用起来也是类似的，使用前也要打开一个数据库，这个数据库文件存在则直接打开否则会创建并打开。这里FMDB引入了一个FMDatabase对象来表示数据库，打开数据库和后面的数据库操作全部依赖此对象。
 - 对于数据库的操作跟前面KCDbManager的封装是类似的，在FMDB中FMDatabase类提供了两个方法executeUpdate:和executeQuery:分别用于执行无返回结果的查询和有返回结果的查询。当然这两个方法有很多的重载这里就不详细解释了。唯一需要指出的是，如果调用有格式化参数的sql语句时，格式化符号使用“?”而不是“%@”、等。
 - 我们知道直接使用libsqlite3进行数据库操作其实是线程不安全的，如果遇到多个线程同时操作一个表的时候可能会发生意想不到的结果。为了解决这个问题建议在多线程中使用FMDatabaseQueue对象，相比FMDatabase而言，它是线程安全的。
 - 将事务放到FMDB中去说并不是因为只有FMDB才支持事务，而是因为FMDB将其封装成了几个方法来调用，不用自己写对应的sql而已。其实在使用libsqlite3操作数据库时也是原生支持事务的（因为这里的事务是基于数据库的，FMDB还是使用的SQLite数据库），只要在执行sql语句前加上“begin transaction;”执行完之后执行“commit transaction;”或者“rollback transaction;”进行提交或回滚即可。另外在Core Data中大家也可以发现，所有的增、删、改操作之后必须调用上下文的保存方法，其实本身就提供了事务的支持，只要不调用保存方法，之前所有的操作是不会提交的。在FMDB中FMDatabase有beginTransaction、commit、rollback三个方法进行开启事务、提交事务和回滚事务。

说一说你对Core Data的认识

Core Data使用起来相对直接使用SQLite3的API而言更加的面向对象，操作过程通常分为以下几个步骤：

- 创建管理上下文

创建管理上下文可以细分为：加载模型文件->指定数据存储路径->创建对应数据类型的存储->创建管理对象上下文并指定存储。

经过这几个步骤之后可以得到管理对象上下文NSManagedObjectContext，以后所有的数据操作都由此对象负责。同时如果是第一次创建上下文，Core Data会自动创建存储文件（例如这里使用SQLite3存储），并且根据模型对象创建对应的表结构。

- **查询数据**
对于有条件的查询，在Core Data中是通过谓词来实现的。首先创建一个请求，然后设置请求条件，最后调用上下文执行请求的方法。
- **插入数据**
插入数据需要调用实体描述对象NSEntityDescription返回一个实体对象，然后设置对象属性，最后保存当前上下文即可。这里需要注意，增、删、改操作完最后必须调用管理对象上下文的保存方法，否则操作不会执行。
- **删除数据**
删除数据可以直接调用管理对象上下文的deleteObject方法，删除完保存上下文即可。注意，删除数据前必须先查询到对应对象。
- **修改数据**
修改数据首先也是取出对应的实体对象，然后通过修改对象的属性，最后保存上下文。

OC中有哪些数据存储方式,各有什么区别?

- OC中有四种数据存储方式:
 - NSUserDefaults,用于存储配置信息
 - SQLite,用于存储查询需求较多的数据
 - CoreData,用于规划应用中的对象
- 使用基本对象类型定制的个性化缓存方案.
 - NSUserDefaults:对象中储存了系统中用户的配置信息,开发者可以通过这个实例对象对这些已有的信息进行修改,也可以按照自己的需求创建新的配置项。
 - SQLite擅长处理的数据类型其实与NSUserDefaults差不多,也是基础类型的小数据,只是从组织形式上不同。开发者可以以关系型数据库的方式组织数据,使用SQL DML来管理数据。一般来说应用中的格式化的文本类数据可以存放在数据库中,尤其是类似聊天记录、Timeline等这些具有条件查询和排序需求的数据。
 - CoreData是一个管理方案,它的持久化可以通过SQLite、XML或二进制文件储存。它可以把整个应用中的对象建模并进行自动化的管理。从归档文件还原模型时CoreData并不是一次性把整个模型中的所有数据都载入内存,而是根据运行时状态,把被调用到的对象实例载入内存。框架会自动控制这个过程,从而达到控制内存消耗,避免浪费。无论从设计原理还是使用方法上看,CoreData都比较复杂。因此,如果仅仅是考虑缓存数据这个需求,CoreData绝对不是一个优选方案。
 - CoreData的使用场景在于:整个应用使用CoreData规划,把应用内的数据通过CoreData建模,完全基于CoreData架构应用。
 - 使用基本对象类型定制的个性化缓存方案:从需求出发分析缓存数据有哪些要求:按Key查找,快速读取,写入不影响正常操作,不浪费内存,支持归档。这些都是基本需求,那么再进一步或许还需要固定缓存项数量,支持队列缓存,缓存过期等。

数据存储这一块,面试常问,你常用哪一种数据存储?什么是序列化?sqlite是直接用它还是用封装了它的第三方库?尤其是会问sqlite和core data的区别?

iOS平台怎么做数据的持久化?coredata和sqlite有无必然联系?coredata是一个关系型数据库吗?

- iOS中可以有四种持久化数据的方式: 属性列表、对象归档、 SQLite3和Core Data
- coredata可以使你以图形界面的方式快速的定义app的数据模型,同时在你的代码中容易获取到它。
- coredata提供了基础结构去处理常用的功能,例如保存,恢复,撤销和重做,允许你在app中继续创建新的任务。
- 在使用coredata的时候,你不用安装额外的数据库系统,因为coredata使用内置的sqlite数据库。
- coredata将你app的模型层放入到一组定义在内存中的数据对象。
- coredata会追踪这些对象的改变,同时可以根据需要做相应的改变,例如用户执行撤销命令。
- 当coredata在对你app数据的改变进行保存的时候,core data会把这些数据归档,并永久性保存。
- mac os x中sqlite库,它是一个轻量级功能强大的关系数据引擎,也很容易嵌入到应用程序。可以在多个平台使用,sqlite是一个轻量级的嵌入式sql数据库编程。
- 与coredata框架不同的是,sqlite是使用程序式的,sql的主要的API来直接操作数据表。
- Core Data不是一个关系型数据库,也不是关系型数据库管理系统(RDBMS)。
- 虽然Core Data支持SQLite作为一种存储类型,但它不能使用任意的SQLite数据库。
- Core Data在使用的过程种自己创建这个数据库。Core Data支持对一、对多的关系。

如果后期需要增加数据库中的字段怎么实现,如果不使用CoreData呢?

- 编写SQL语句来操作原来表中的字段
- 增加表字段: ALTER TABLE 表名 ADD COLUMN 字段名 字段类型;
- 删除表字段: ALTER TABLE 表名 DROP COLUMN 字段名;
- 修改表字段: ALTER TABLE 表名 RENAME COLUMN 旧字段名 TO 新字段名;

SQLite数据存储是怎么用?

- 添加SQLite动态库: 导入主头文件: #import <sqlite3.h>
- 利用C语言函数创建\打开数据库, 编写SQL语句

简单描述下客户端的缓存机制?

- 缓存可以分为：内存数据缓存、数据库缓存、文件缓存
- 每次想获取数据的时候
- 先检测内存中有无缓存
- 再检测本地有无缓存(数据库\文件)
- 最终发送网络请求
- 将服务器返回的网络数据进行缓存（内存、数据库、文件）以便下次读取

你实现过多线程的Core Data么？

NSPersistentStoreCoordinator, NSManagedObjectContext和NSManagedObject中的哪些需要在线程中创建或者传递？你是用什么样的策略来实现的？

- CoreData是对SQLite数据库的封装
- CoreData中的NSManagedObjectContext在多线程中不安全
- 如果想要多线程访问CoreData的话，最好的方法是一个线程一个NSManagedObjectContext
- 每个NSManagedObjectContext对象实例都可以使用同一个NSPersistentStoreCoordinator实例，这是因为NSManagedObjectContext会在使用NSPersistentStoreCoordinator前上锁

Core Data数据迁移

博客地址: <http://blog.csdn.net/jasonblog/article/details/17842535>

FMDB的使用和对多张表的处理

博客地址: <http://blog.csdn.net/wscqqlucy/article/details/8464398>

说说数据库的左连接和右连接的区别

- 数据库左连接和右连接的区别：主表不一样通过左连接和右连接，最小条数为3（记录条数较小的记录数），最大条数为12（3×4）
技术博客的地址：<http://www.2cto.com/database/201407/317367.html>

iOS 的沙盒目录结构是怎样的？App Bundle 里面都有什么？

1. 沙盒结构

- Application：存放程序源文件，上架前经过数字签名，上架后不可修改
- Documents：常用目录，iCloud备份目录，存放数据,这里不能存缓存文件,否则上架不被通过
- Library
 - Caches：存放体积大又不需要备份的数据,SDWebImage缓存路径就是这个
 - Preference：设置目录，iCloud会备份设置信息

- tmp: 存放临时文件, 不会被备份, 而且这个文件下的数据有可能随时被清除的可能

2. App Bundle 里面有什么

- Info.plist:此文件包含了应用程序的配置信息.系统依赖此文件以获取应用程序的相关信息
- 可执行文件:此文件包含应用程序的入口和通过静态连接到应用程序target的代码
- 资源文件:图片,声音文件一类的
- 其他:可以嵌入定制的数据资源

你会如何存储用户的一些敏感信息, 如登录的 token

使用keychain来存储,也就是钥匙串,使用keychain需要导入Security框架

自定义一个keychain的类

```
#import <Security/Security.h>
@implementation YCKKeyChain

+(NSMutableDictionary *)getKeychainQuery:(NSString *)service {
return [NSMutableDictionary dictionaryWithObjectsAndKeys:
       (__bridge_transfer id)kSecClassGenericPassword,
       (__bridge_transfer id)kSecClass,
       service, (__bridge_transfer id)kSecAttrService,
       service, (__bridge_transfer id)kSecAttrAccount,
       (__bridge_transfer
id)kSecAttrAccessibleAfterFirstUnlock,      (__bridge_transfer
id)kSecAttrAccessible,
       nil];
}

+(void)save:(NSString *)service data:(id)data {
// 获得搜索字典
NSMutableDictionary *keychainQuery = [self
getKeychainQuery:service];
// 添加新的删除旧的
SecItemDelete((__bridge retained CFDictionaryRef)keychainQuery);
// 添加新的对象到字符串
[keychainQuery setObject:[NSKeyedArchiver
archivedDataWithRootObject:data]      forKey:(__bridge_transfer
id)kSecValueData];
// 查询钥匙串
SecItemAdd((__bridge retained CFDictionaryRef)keychainQuery,
NULL);
}

+(id)load:(NSString *)service {
id ret = nil;
```

```

NSMutableDictionary *keychainQuery = [self
getKeychainQuery:service];
// 配置搜索设置
[keychainQuery setObject:(id)kCFBooleanTrue forKey:
(__bridge_transfer id)kSecReturnData];
[keychainQuery setObject:(__bridge_transfer id)kSecMatchLimitOne
forKey:    (__bridge_transfer id)kSecMatchLimit];
CFDataRef keyData = NULL;
if (SecItemCopyMatching((__bridge_released
CFDictionaryRef)keychainQuery, (CFTypeRef *)&keyData) == noErr) {
@try {
    ret = [NSKeyedUnarchiver unarchiveObjectWithData:
(__bridge_transfer NSData *)keyData];
} @catch (NSEception *e) {
    NSLog(@"Unarchive of %@ failed: %@", service, e);
} @finally {
}
}
return ret;
}

+(void)delete:(NSString *)service {
NSMutableDictionary *keychainQuery = [self
getKeychainQuery:service];
SecItemDelete((__bridge_released CFDictionaryRef)keychainQuery);
}

```

在别的类实现存储,加载,删除敏感信息方法

```

// 用来标识这个钥匙串
static NSString const KEY_IN_KEYCHAIN = @"com.yck.app.allinfo";
// 用来标识密码
static NSString const KEY_PASSWORD = @"com.yck.app.password";

```

```

+(void)savePassWord:(NSString *)password
{
NSMutableDictionary *passwordDict = [NSMutableDictionary
dictionary];
[passwordDict setObject:password forKey:KEY_PASSWORD];
[YCKKeyChain save:KEY_IN_KEYCHAIN data:passwordDict];
}
+(id)readPassWord
{
NSMutableDictionary *passwordDict = (NSMutableDictionary *)
[YCKKeyChain load:KEY_IN_KEYCHAIN];
return [passwordDict objectForKey:KEY_PASSWORD];
}
+(void)deletePassWord
{
}

```

```
[ YCKKeyChain delete:KEY_IN_KEYCHAIN];
}
```

使用 **NSUserDefaults** 时，如何处理布尔的默认值？(比如返回 NO，不知道是真的 NO 还是没有设置过)

```
if([[NSUserDefaults standardUserDefaults] objectForKey:@"ID"] == nil)
{    NSLog(@"没有设置"); }
```

MD5和Base64的区别是什么，各自使用场景是什么？

做过加密相关的功能的，几乎都会使用到MD5和Base64，它们两者在实际开发中是最常用的。

- MD5：是一种不可逆的摘要算法，用于生成摘要，无法逆着破解得到原文。常用的是生成32位摘要，用于验证数据的有效性。比如，在网络请求接口中，通过将所有的参数生成摘要，客户端和服务端采用同样的规则生成摘要，这样可以防篡改。又如，下载文件时，通过生成文件的摘要，用于验证文件是否损坏。
- Base64：属于加密算法，是可逆的，经过encode后，可以decode得到原文。在开发中，有的公司上传图片采用的是将图片转换成base64字符串，再上传。在做加密相关的功能时，通常会将数据进行base64加密/解密。

plist文件是用来做什么的。一般用它来处理一些什么方面的问题。

- plist是iOS系统中特有的文件格式。我们常用的NSUserDefaults偏好设置实质上就是plist文件操作。plist文件是用来持久化存储数据的。
- 我们通常使用它来存储偏好设置，以及那些少量的、数组结构比较复杂的不适合存储数据库的数据。比如，我们要存储全国城市名称和id，那么我们要优先选择plist直接持久化存储，因为更简单。

当存储大块数据是怎么做？

- 你有很多选择，比如：
- 使用NSUserDefaults
- 使用XML, JSON, 或者 plist
- 使用NSCoding存档
- 使用类似SQLite的本地SQL数据库
- 使用 Core Data
- NSUserDefaults的问题是什么？虽然它很nice也很便捷，但是它只适用于小数据，比如一些简单的布尔型的设置选项，再大点你就要考虑其它方式了
- XML这种结构化档案呢？总体来说，你需要读取整个文件到内存里去解析，这样是很不经济的。使用SAX又是一个很麻烦的事情。
- NSCoding？不幸的是，它也需要读写文件，所以也有以上问题。
- 在这种应用场景下，使用SQLite 或者 Core Data比较好。使用这些技术你用特定的查询语句就能只加载你需要的对象。在性能层面来讲，SQLite和Core Data是很相似的。他们的不同在于具体使用方法。Core Data代表一个对象的graph model，但SQLite就是

一个DBMS。Apple在一般情况下建议使用Core Data，但是如果你有理由不使用它，那么就去使用更加底层的SQLite吧。如果你使用SQLite，你可以用FMDB(<https://GitHub.com/ccgus/fmdb>)这个库来简化SQLite的操作，这样你就不用花很多经历了解SQLite的C API了

怎么解决sqlite锁定的问题

1> 设置数据库锁定的处理函数

```
int sqlite3_busy_handler(sqlite3*, int(*)(void*,int), void*);
```

函数可以定义一个回调函数，当出现数据库忙时，sqlite会调用该函数

当回调函数为NULL时，清除busy handle，申请不到锁直接返回

回调函数的第二个函数会被传递为该由此次忙事件调用该函数的次数

回调函数返回非0，数据库会重试当前操作，返回0则当前操作返回SQLITE_BUSY

2> 设定锁定时的等待时间

```
int sqlite3_busy_timeout(sqlite3*, 60);
```

定义一个毫秒数，当未到达该毫秒数时，sqlite会sleep并重试当前操作

如果超过ms毫秒，仍然申请不到需要的锁，当前操作返回sqlite_BUSY

当ms<=0时，清除busy handle，申请不到锁直接返回

(15) Runloop

什么是 Runloop?

- 从字面上讲就是运行循环。
- 它内部就是do-while循环，在这个循环内部不断地处理各种任务。
- 一个线程对应一个RunLoop，主线程的RunLoop默认已经启动，子线程的RunLoop得手动启动（调用run方法）
- RunLoop只能选择一个Mode启动，如果当前Mode中没有任何Source(Sources0、Sources1)、Timer，那么就直接退出RunLoop
- 基本的作用就是保持程序的持续运行，处理app中的各种事件。通过runloop，有事运行，没事就休息，可以节省cpu资源，提高程序性能。

RunLoop对象

iOS中有2套API来访问和使用RunLoop

- Foundation: NSRunLoop
- Core Foundation: CFRunLoopRef

- NSRunLoop和CFRunLoopRef都代表着RunLoop对象
- NSRunLoop是基于CFRunLoopRef的一层OC包装，所以要了解RunLoop内部结构，需要多研究CFRunLoopRef层面的API。

RunLoop与线程

- 每条线程都有唯一的一个与之对应的RunLoop对象
- 主线程的RunLoop已经自动创建好了，子线程的RunLoop需要主动创建
- RunLoop在第一次获取时创建，在线程结束时销毁

获得RunLoop对象

- Foundation

```
[NSRunLoop currentRunLoop]; // 获得当前线程的RunLoop对象
```

```
[NSRunLoop mainRunLoop]; // 获得主线程的RunLoop对象
```

- Core Foundation

```
CFRunLoopGetCurrent(); // 获得当前线程的RunLoop对象
```

```
CFRunLoopGetMain(); // 获得主线程的RunLoop对象
```

RunLoop相关类

Core Foundation中关于RunLoop的5个类

CFRunLoopRef
 CFRunLoopModeRef
 CFRunLoopSourceRef
 CFRunLoopTimerRef
 CFRunLoopObserverRef

CFRunLoopModeRef

CFRunLoopModeRef代表RunLoop的运行模式。

一个RunLoop包含若干个Mode，每个Mode又包含若干个(set)Source/(array)Timer/(array)Observer

每次RunLoop启动时，只能指定其中一个 Mode，这个Mode被称作CurrentMode

如果需要切换Mode，只能退出Loop，再重新指定一个Mode进入

这样做主要是为了分隔开不同组的Source/Timer/Observer，让其互不影响

mode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滑动时会切换到该Mode
- UIInitializationRunLoopMode: run loop启动时, 会切换到该mode
- NSRunLoopCommonModes (kCFRunLoopCommonModes) : Mode集合

苹果公开提供的Mode有两个:

- NSDefaultRunLoopMode (kCFRunLoopDefaultMode)
- NSRunLoopCommonModes (kCFRunLoopCommonModes)

CFRunLoopTimerRef

- CFRunLoopTimerRef是基于时间的触发器
- CFRunLoopTimerRef基本上说的就是NSTimer, 它受RunLoop的Mode影响
- GCD的定时器不受RunLoop的Mode影响

CFRunLoopSourceRef

CFRunLoopSourceRef是事件源 (输入源)

按照官方文档, Source的分类

Port-Based Sources

Custom Input Sources

Cocoa Perform Selector Sources

按照函数调用栈, Source的分类

Source0: 非基于Port的

Source1: 基于Port的, 通过内核和其他线程通信, 接收、分发系统事件

CFRunLoopObserverRef

- CFRunLoopObserverRef是观察者, 能够监听RunLoop的状态改变
- 可以监听的时间点有以下几个
 - kcfRunLoopEntry(即将进入loop)//1
 - kcfRunLoopBeforeTimers(即将处理timer)//2
 - kcfRunLoopBeforeSources(即将处理source)//4
 - kcfRunLoopBeforeWaiting(即将进入休眠)//32
 - kcfRunLoopAfterWaiting(刚从休眠中唤醒)//64
 - kcfRunLoopExit(即将退出loop)//128

- 添加Observer

```

CFRunLoopObserverRef observer =
CFRunLoopObserverCreateWithHandler(CFAllocatorGetDefault(),
kCFRunLoopAllActivities, YES, 0, ^(CFRunLoopObserverRef observer,
CFRunLoopActivity activity) {
    NSLog(@"%@", @"----监听到RunLoop状态发生改变---%zd", activity);
});

// 添加观察者：监听RunLoop的状态
CFRunLoopAddObserver(CFRunLoopGetCurrent(), observer,
kCFRunLoopDefaultMode);

// 释放Observer
CFRelease(observer);

```

RunLoop处理逻辑

- 通知Observer: 即将进入Loop (1)
- 通知Observer: 将要处理Timer (2)
- 通知Observer: 将要处理Source0 (3)
- 处理Source0 (4)
- 如果有Source0, 跳到第9步 (5)
- 通知Observer: 线程即将休眠 (6)
- 休眠, 等待唤醒: (7)
 - Source0(port)。
 - timer启动
 - RunLoop设置的timer已经超时
 - Runloop被外部手动唤醒
- 通知Observer: 线程将被唤醒 (8)
- 处理未处理的时间 (9)
 - 如果用户定义的定时器启动, 处理定时器事件并重启RunLoop。进入步骤2.
 - 如果输入源启动, 传递相应的消息。

- 如果RunLopp被显式唤醒而且时间还没超时，重启RunLoop，进入步骤2.
- 通知Observer：即将退出Loop

RunLoop的应用

- NSTimer
- ImageView显示
- PerformSelector
- 常驻线程
- 自动释放池

runloop定时源和输入源

[image image image image image image image image image image image](http://www.jianshu.com/p/335a9b19adab)

<http://www.jianshu.com/p/335a9b19adab>

- Runloop处理的输入事件有两种不同的来源：输入源（input source）和定时源（timer source）
- 输入源传递异步消息，通常来自于其他线程或者程序。
- 定时源则传递同步消息，在特定时间或者一定的时间间隔发生

NSRunLoop的实现机制,及在多线程中如何使用

- 实现机制：回答runloop的基本作用，处理逻辑，前面都有。
- 程序创建子线程的时候，才需要手动启动runloop。主线程的runloop已经默认启动。
- 在多线程中，你需要判断是否需要runloop。如果需要runloop，那么你要负责配置runloop并启动。你不需要在任何情况下都去启动runloop。比如，你使用线程去处理一个预先定义好的耗时极长的任务时，你就可以无需启动runloop。RunLoop只在你要和线程有交互时才需要

runloop和线程有什么关系？

- 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

```
( argc, * argv[] ) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, , NSStringFromClass([AppDelegate class]));
    }
}
```

重点是UIApplicationMain()函数，这个方法会为mainthread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。

- 对其它线程来说，runloop默认是没有启动的，runloop只在你要和线程有交互时才需要。
- 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runloop。
NSRunLoop *runloop = [NSThread currentThread];

autorelease对象在什么情况下会被释放？

- 分两种情况：手动干预释放和系统自动释放
- 手动干预释放就是指定autoreleasepool，当前作用域大括号结束就立即释放
- 系统自动去释放：不手动指定autoreleasepool，Autorelease对象会在当前的runloop迭代结束时释放
- kCFRunLoopEntry(1)：第一次进入会自动创建一个autorelease
- kCFRunLoopBeforeWaiting(32)：进入休眠状态前会自动销毁一个autorelease，然后重新创建一个新的autorelease
- kCFRunLoopExit(128)：退出runloop时会自动销毁最后一个创建的autorelease

对于runloop的理解不正确的是

- A 每一个线程都有其对应的RunLoop
- B 默认非主线程的RunLoop是没有运行的
- C 在一个单独的线程中没有必要去启用RunLoop
- D 可以将NSTimer添加到runloop中

- 参考答案：C
- 理由：说到RunLoop，它可是多线程的法定。通常来说，一个线程一次只能执行一个任务，执行完任务后就会退出线程。但是，对于主线程是不能退出的，因此我们需要让主线程即时任务执行完毕，也可以继续等待是接收事件而不退出，那么RunLoop就是关键法宝了。但是非主线程通常来说就是为了执行某一任务的，执行完毕就需要归还资源，因此默认是不运行RunLoop的。NSThread提供了一个添加NSTimer的方法，这个方法是在应用正常状态下会回调。

runloop的mode作用是什么？

mode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滑动时会切换到该Mode
- UIInitializationRunLoopMode: run loop启动时，会切换到该mode
- NSRunLoopCommonModes (kCFRunLoopCommonModes) : Mode集合

苹果公开提供的Mode有两个：

- NSDefaultRunLoopMode (kCFRunLoopDefaultMode)
- NSRunLoopCommonModes (kCFRunLoopCommonModes)

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。当我们滚动的时候，也希望不调度，那就应该使用默认模式。但是，如果希望在滚动时，定时器也要回调，那就应该使用common mode。

请写出NSTimer使用时的注意事项（两项即可）

思路和上一题一样，如果想要销毁timer，则必须先将timer置为失效，否则timer就一直占用内存而不会释放。造成逻辑上的内存泄漏。该泄漏不能用xcode及instruments测出来。另外对于要求必须销毁timer的逻辑处理，未将timer置为失效，若每次都创建一次，则之前的不能得到释放，则会同时存在多个timer的实例在内存中。

参考答案：

- 注意timer添加到runloop时应该设置为什么mode
- 注意timer在不需要时，一定要调用invalidate方法使定时器失效，否则得不到释放

UITableViewCell上有个UILabel，显示NSTimer实现的秒表时间，手指滚动cell过程中，label是否刷新，为什么？

和上一题一样的思路，如果要cell滚动过程中定时器正常回调，UI正常刷新，那么要将timer放入到CommonModes下，因为是NSDefaultRunLoopMode，只有在空闲状态下才会回调。

为什么UIScrollView的滚动会导致NSTimer失效？

- 思路和上一题一样，解决办法有2个，一个是更改mode为NSRunLoopCommonModes(无论runloop运行在哪个mode,都能运行),还有种办法是切换到主线程来更新UI界面的刷新

```
//将timer添加到NSDefaultRunLoopMode中
[NSTimer scheduledTimerWithTimeInterval: target:
selector:@selector(timerTick:) userInfo: repeats:];
//然后再添加到NSRunLoopCommonModes里
NSTimer *timer = [NSTimer timerWithTimeInterval: target:
selector:@selector(timerTick:) userInfo: repeats:];
[[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSRunLoopCommonModes];
```

在滑动页面上的列表时，timer会暂定回调，为什么？如何解决？

- 思路和上一题一样

在开发中如何使用RunLoop？什么应用场景？

- 开启一个常驻线程（让一个子线程不进入消亡状态，等待其他线程发来消息，处理其他事件）
- 在子线程中开启一个定时器
- 在子线程中进行一些长期监控
- 可以控制定时器在特定模式下执行
- 可以让某些事件（行为、任务）在特定模式下执行
- 可以添加Observer监听RunLoop的状态，比如监听点击事件的处理（在所有点击事件之前做一些事情）

(16) 网络

http请求方式？

通常，HTTP的请求方式有3种，分别是：POST、GET、HEAD。POST和GET方法是用于数据发送的。

POST：它将要发送的数据单独放在一个流中进行发送，而不是附加在URL地址后面，这样做的好处是这些数据不会出现在URL地址中。

GET：它将要发送的数据直接添加在URL后面，如：

www.sina.com.cn?username=""&password=""，这样的好处是可以直接将数据加在URL后，而不需在用另外的流来发送这些数据，但是缺点也显而易见，它将用户的信息显示出来了。

HEAD：它是请求资源的元数据方法。在具体的应用中，我暂时还没遇到过，也不去对它进行研究，需要是在学习。

Http定义了与服务器交互的不同方法，最基本的方法有？

- URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查，改，增，删4个操作。
- GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。

socket编程简述

它是基于TCP/IP协议，Socket就是一个可以连通网络上不同计算机程序之间的管道，把一堆数据从管道的A端扔进去，则会从管道的B端（也许同时还可以从C、D、E、F……端冒出来）。管道的端口由两个因素来唯一确认，即机器的IP地址和程序所使用的端口号。

Socket可以支持数据的发送和接收，它会定义一种称为套接字的变量，发送数据时首先创建套接字，然后使用该套接字的sendto等方法对准某个IP/端口进行数据发送；接收端也首先创建套接字，然后将该套接字绑定到一个IP/端口上，所有发向此端口的数据会被该套接字的recv等函数读出。如同读出文件中的数据一样。

TCP/IP的socket提供下列三种类型套接字。 流式套接字、数据报式套接字、原始式套接字。

客户端编程步骤：

- 1：加载套接字库，创建套接字(WSAStartup()/socket());
- 2：向服务器发出连接请求(connect());
- 3：和服务器端进行通信(send()/recv());
- 4：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

常用第三方库：1， Asyncsocket库

asihttp代码原理，异步请求的原理，异步请求最大数目，为什么只能这么多？

ASIHTTPRequest是一个简易使用的类库,通过包装CFNetwork API 来简化 和服务器端的通讯. 它编写的语言是Objective-C 能够应用于Mac OS X and iPhone 平台的应用程序.

异步: 请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕这个数量是跟cpu有关的,并发性取决于cpu核数,每个核只能同时处理一个任务.4核cpu理论上可以并发处理4个任务,如果按http来算就是4个请求,但是cpu是抢占式资源,所以一般来说并发量是要根据任务的耗时和cpu的繁忙度来计算4个左右只是个经验值你开10个短耗时的任务和几个长耗时任务的效率是不同的。

JSONKit、SBJson、TouchJSON和原生的区别？

JSONKit、SBJson、TouchJSON（性能从左到右，越右越差,主要就是性能上的差别）

App需要加载超大量的数据，给服务器发送请求，但是服务器卡住了如何解决？

- 1> 设置请求超时
- 2> 给用户提示请求超时
- 3> 根据用户操作再次请求数据

HTTP的通信的 发送请求、接收响应 包含哪些内容？OC中是怎样实现的？

1. 请求：一个请求包含以下内容：

2. 请求行：包含了请求方法、请求资源路径、HTTP协议版本

 GET /XXServer/resources/images/1.jpg HTTP/1.1

3. 请求头：包含了对客户端的环境描述、客户端请求的主机地址等信息

 ○ Host: 192.168.1.105:8080 // 客户端想访问的服务器主机地址

 ○ User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9) Firefox/30.0
 // 客户端的类型，客户端的软件环境

 ○ Accept: text/html, // 客户端所能接收的数据类型

 ○ Accept-Language: zh-cn // 客户端的语言环境

 ○ Accept-Encoding: gzip // 客户端支持的数据压缩格式

 ○ 请求体：客户端发给服务器的具体数据，比如文件数据

 ○ OC中请求NSURLRequest

 ○ 发送给服务器的请求包含：

 ○ 请求行：包含了请求方法、请求资源路径、HTTP协议版本

 ○ 请求头：对客户端的环境描述、客户端请求的主机地址等信息

 ○ 请求体：客户端发给服务器的具体数据

 ○ 默认超时时常：60s

 ○ 响应：

 ○ 一个响应包括：

 ○ 状态行：包含了HTTP协议版本、状态码、状态英文名称 HTTP/1.1 200 OK

 ○ 响应头：包含了对服务器的描述、对返回数据的描述

 ○ Server: Apache-Coyote/1.1 // 服务器的类型

 ○ Content-Type: image/jpeg // 返回数据的类型

 ○ Content-Length: 56811 // 返回数据的长度

 ○ Date: Mon, 23 Jun 2014 12:54:52 GMT // 响应的时间

 ○ 实体内容：服务器返回给客户端的具体数据，比如文件数据

 ○ OC中响应用NSURLRespose：返回给客户端的回应包含：

 ■ 状态行：包含了HTTP协议版本、状态码、状态英文名称

 ■ 响应头：包含了对服务器的描述、对返回数据的描述

 ■ 实体内容：服务器返回给客户端的具体二进制数据

 ■ 常用属性： expectedContentLength (下载时返回文件的长度)

 suggestedFilename (建议保存的文件名)

http 的post与get区别与联系，实践中如何选择它们？

	GET	POST
用途	从服务器上获取数据	向服务器传送数据提交方式

	GET	POST
服务器解析	Request.QueryString获取变量的值	Request.Form获取提交的数据
数据大小	最大1024字节	无限制
安全性	URL中能看到提交的数据	隐藏在请求头中

知道TCP/UDP吗？说说关于UDP/TCP的区别？

- UDP: 是用户数据报协议: 主要用在实时性要求高以及对质量相对较弱的地方,但面对现在高质量的线路不是容易丢包除非是一些拥塞条件下,如流媒体
- TCP: 是传输控制协议:是面连接的,那么运行环境必然要求其可靠性不可丢包有良好的拥塞控制机制如http ftp telnet 等

	TCP	UDP
发送与接收	安全送达	只管发送
建立连接	是 (三次握手)	否 (有数据包, 无需连接)
数据大小	无限制	每个数据报64k
可靠性	可靠	不可靠
速度	慢 (三次握手才能完成连接)	快 (无需连接)
应用	流媒体	qq

什么是三次握手与四次挥手？

- 三次握手实现的过程：
 - 第一次握手：建立连接时，客户端发送同步序列编号到服务器，并进入发送状态，等待服务器确认
 - 第二次：服务器收到同步序列编号，确认并同时自己也发送一个同步序列编号+确认标志，此时服务器进入接收状态
 - 第三次：客户端收到服务器发送的包，并向服务器发送确认标志，随后链接成功。
 - 注意：是在链接成功后在进行数据传输。
- 四次挥手：
 - 第一次：客户端向服务器发送一个带有结束标记的报文。
 - 第二次：服务器收到报文后，向客户端发送一个确认序号，同时通知自己相应的应用程序：对方要求关闭连接
 - 第三次：服务器向客户端发送一个带有结束标记的报文。

- 第四次：客户端收到报文后，向服务器发送一个确认序号。链接关闭。

分析json、xml的区别?json、xml解析方式的底层是如何处理的?

1. Json与xml的区别:

- 可读性方面:基本相同,xml的可读性比较好
- 可扩展性方面:都具有很好的扩展性
- 编码难度方面:相对而言:JSON的编码比较容易
- 解码难度:json的解码难度基本为零,xml需要考虑子节点和父节点
- 数据体积方面:json相对于xml来讲,数据体积小,传递的速度跟快些
- 数据交互方面:json与JavaScript的交互更加方便,更容易解析处理,更好的数据交互
- 数据描述方面:xml对数据描述性比较好
- 传输速度方面:json的速度远远快于xml

2. JSON底层原理:

- 遍历字符串中的字符,最终根据格式规定的特殊字符,比如{}号,[]号,:号等进行区分,{}号是一个字典的开始,[]号是一个数组的开始,:号是字典的键和值的分水岭,最终乃是将json数据转化为字典,字典中值可能是字典,数组,或字符串而已。

3. XML底层原理:

- XML解析常用的解析方法有两种:DOM解析和SAX解析。
 - DOM 采用建立树形结构的方式访问 XML 文档,而 SAX 采用的事件模型。
 - DOM 解析把 XML 文档转化为一个包含其内容的树,并可以对树进行遍历。
 - 使用 DOM 解析器的时候需要处理整个 XML 文档,所以对性能和内存的要求比较高。
 - SAX在解析 XML 文档的时候可以触发一系列的事件,当发现给定的tag的时候,它可以激活一个回调方法,告诉该方法制定的标签已经找到。
 - SAX 对内存的要求通常会比较低,因为它让开发人员自己来决定所要处理的 tag。特别是当开发人员只需要处理文档中所包含的部分数据时,SAX 这种扩展能力得到了更好的体现。
1. (补充)其他解析方式有自定义二进制解析,就是按字节去解析,电话会谈就是如此,还可以是字符串之间用特殊符号连接的数据,将此数据用特殊符号可以分割成所用数据。

http和scoket通信的区别?socket连接相关库,TCP,UDP的连接方法,HTTP的几种常用方式?

1. http和scoket通信的区别:

- http是客户端用http协议进行请求,发送请求时候需要封装http请求头,并绑定请求的数据,服务器一般有web服务器配合(当然也非绝对)。http请求方式为客户端主动发起请求,服务器才能给响应,一次请求完毕后则断开连接,以节省资源。

服务器不能主动给客户端响应(除非采取http长连接技术)。iphone主要使用类是NSURLConnection。

- socket是客户端跟服务器直接使用socket“套接字”进行连接,并没有规定连接后断开,所以客户端和服务器可以保持连接通道,双方都可以主动发送数据。一般在游戏开发或股票开发这种要求即时性很强并且保持发送数据量比较大的场合使用。主要使用类是CFSocketRef。

通信底层原理 (OSI七层模型)

- OSI简介: OSI采用了分层的结构化技术, 共分七层, 物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。
 - 物理层: 主要定义物理设备标准, 如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流 (就是由1、0转化为电流强弱来进行传输, 到达目的地后在转化为1、0, 也就是我们常说的数模转换与模数转换)。这一层的数据叫做比特。
 - 数据链路层: 定义了如何让格式化数据以进行传输, 以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正, 以确保数据的可靠传输。
 - 网络层: 在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加, 而网络层正是管理这种连接的层。
 - 传输层: 定义了一些传输数据的协议和端口号 (WWW端口80等), 如: TCP (传输控制协议, 传输效率低, 可靠性强, 用于传输可靠性要求高, 数据量大的数据), UDP (用户数据报协议, 与TCP特性恰恰相反, 用于传输可靠性要求不高, 数据量小的数据, 如QQ聊天数据就是通过这种方式传输的)。主要是将从下层接收的数据进行分段和传输, 到达目的地址后再进行重组。常常把这一层数据叫做段。
- 会话层: 通过传输层 (端口号: 传输端口与接收端口) 建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求 (设备之间需要互相认识可以是IP也可以是MAC或者是主机名)
- 表示层: 可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如, PC程序与另一台计算机进行通信, 其中一台计算机使用扩展二十一进制交换码 (EBCDIC), 而另一台则使用美国信息交换标准码 (ASCII) 来表示相同的字符。如有必要, 表示层会通过使用一种通格式来实现多种数据格式之间的转换。
- 应用层: 是最靠近用户的OSI层。这一层为用户的应用程序 (例如电子邮件、文件传输和终端仿真) 提供网络服务。

all people seem to need date processing这一句话的意思是所有的人似乎都需要处理数据

Application ->all

Presentation ->people

Session ->seem

Transport ->to

Network ->need

Data ->date

Physical ->processing

设计一套大文件（如上百M的视频）下载方案

- NSURLConnection
- 支持断点下载，自动记录停止下载时断点的位置
- 遵守NSURLSessionDownloadDelegate协议
- 使用NSURLSession下载大文件，被下载文件会被自动写入沙盒的临时文件夹tmp中
- 下载完毕，通常需要将已下载文件移动其他位置（tmp文件夹中的数据被定时删除），通常是cache文件夹中
- 下载步骤：
 - 设置下载任务task的为成员变量

```
@property (nonatomic, strong) NSURLSessionDownloadTask *task;
```

- 获取NSURLSession对象

```
NSURLSession *session = [NSURLSession sessionWithConfiguration:[NSURLSessionConfiguration defaultSessionConfiguration] delegate:self delegateQueue:[[NSOperationQueue alloc] init]];
```

- 初始化下载任务任务

```
self.task = [session downloadTaskWithURL:(此处为下载文件路径 URL)];
```

- 实现代理方法

```
/*每当写入数据到临时文件的时候，就会调用一次该方法，通常在该方法中获取下载进度/
```

```
- (void)URLSession:(NSURLSession )session downloadTask:(NSURLSessionDownloadTask )downloadTask didWriteData:(int64_t)bytesWritten totalBytesWritten:(int64_t)totalBytesWritten totalBytesExpectedToWrite:(int64_t)totalBytesExpectedToWrite {
```

```
// 计算下载进度
```

- ```
CGFloat progress = 1.0 * totalBytesWritten / totalBytesExpectedToWrite;
```

```
}
```

```
/*任务终止时调用的方法，通常用于断点下载/
```

```
- (void)URLSession:(NSURLSession)session downloadTask:(NSURLSessionDownloadTask)downloadTask didResumeAtOffset:
```

```

(int64_t)fileOffset expectedTotalBytes:(int64_t)expectedTotalBytes
{
 //fileOffset: 下载任务中止时的偏移量
}
/*遇到错误的时候调用, error参数只能传递客户端的错误/
-(void)URLSession:(NSURLSession)session task:(NSURLSessionTask)task
didCompleteWithError:(NSError *)error
{ }
/*下载完成的时候调用, 需要将文件剪切到可以长期保存的文件夹中/
-(void)URLSession:(NSURLSession)session downloadTask:
(NSURLSessionDownloadTask)downloadTask
didFinishDownloadingToURL:(NSURL *)location
{
 //生成文件长期保存的路径
 ■ NSString *file =
 [[NSSearchPathForDirectoriesInDomains(NSCachesDirectory,
 NSUserDomainMask, YES) lastObject]
 stringByAppendingPathComponent:downloadTask.response.suggestedFilename];
 ■ //获取文件句柄
 ■ NSFileManager *fileManager = [NSFileManager defaultManager];
 ■ //通过文件句柄, 将文件剪切到文件长期保存的路径
 ■ [fileManager moveItemAtURL:location toURL:[NSURL
 fileURLWithPath:file] error:nil];
}

■ 操作任务状态
/*开始/继续下载任务/
[self.task resume];
/*暂停下载任务/
[self.task suspend];

```

## HTTP协议的特点，关于HTTP请求GET和POST的区别？

HTTP协议的特点：

- HTTP超文本传输协议，是短连接，是客户端主动发送请求，服务器做出响应，服务器响应之后，链接断开。HTTP是一个属于应用层面向对象的协议，HTTP有两类报文：请求报文和响应报文。
- HTTP请求报文：一个HTTP请求报文由请求行、请求头部、空行和请求数据4部分组成。
- HTTP响应报文：由三部分组成：状态行、消息报头、响应正文。

## 即时聊天App不会采用的网络传输方式

- A UDP  
B TCP  
C HTTP

D FTP

参考答案：D

理由：FTP是文件传输协议，是File Transfer Protocol的简称，它的作用是用于控制互联网上文件的双向传输，因此一定不会是即时聊天使用的；UDP是面向无连接的传输层协议，数据传输是不可靠的，它只管发，不管收不收得到；TCP是面向连接的，可靠的传输层协议；HTTP是超文本传输协议，对应于应用层，而HTTP是基于TCP的。

## 在App中混合HTML5开发App如何实现的。在App中使用HTML5的优缺点是什么？

在ios中，通常是用UIWebView来实现，当然在ios8以后可以使用WKWebView来实现。有以下几种实现方法：

通过实现UIWebView的代理方法来拦截，判断scheme是否是约定好的，然后ios调用本地相关API来实现：

```
- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)request navigationType:(UIWebViewNavigationType)navigationType;
```

在ios7以后，可以直接通过JavaScriptCore这个库来实现，通过往JS DOM注入对象，而这个对象对于我们ios的某个类的实例。更详细请阅读：

OC JavaScriptCore与js交互

WKWebView新特性及JS交互

Swift JavaScriptCore与JS交互

可以通过WebViewJavascriptBridge来实现。具体如何使用，请大家去其它博客搜索吧！

优缺点：

ios加入H5响应比原生要慢很多，体验不太好，这是缺点。

ios加入H5可以实现嵌入别的功能入口，可随时更改，不用更新版本就可以上线，这是最大的优点。

## 介绍一下XMPP?有什么优缺点吗？

XMPP(Extensible Messaging and Presence Protocol,前称)是一种以XML为基础的开放式实时通信协议，是经由互联网工程工作小组(IETF)通过的互联网标准。简单的说，XMPP就是一种协议，一种规定。就是说，在网络上传东西，要建立连接，TCP/IP连接，建立后再传东西，而XMPP就是规定你传的东西的格式。XMPP是基于XML的协议。优点开放：

XMPP协议是自由、开放、公开的，并且易于了解。而且在客户端、服务器、组件、源码库等方面，都已经各自有多种实现。标准：

互联网工程工作小组(IETF)已经将Jabber的核心XML流协议以XMPP之名，正式列为认可的实时通信及Presence技术。而XMPP的技术规格已被定义在RFC 3920及RFC 3921。任何IM供应商在遵循XMPP协议下，都可与Google Talk实现连接。证实可用：第一个Jabber(现在XMPP)技术是Jeremie Miller在1998年开发的，现在已经相当稳定；数以百计的开发者为XMPP技术而努力。今日的互联网上有数以万计的XMPP服务器运作着，并有数以百万计的人们使用XMPP实时传讯软件。

### 分散式：

XMPP网络的架构和电子邮件十分相像；XMPP核心协议通信方式是先创建一个stream，XMPP以TCP传递XML数据流，没有中央主服务器。任何人都可以运行自己的XMPP服务器，使个人及组织能够掌控他们的实时传讯体验。

### 安全：

任何XMPP协议的服务器可以独立于公众XMPP网络（例如在企业内部网络中），而使用SASL及TLS等技术的可靠安全性，已自带于核心XMPP技术规格中。

### 可扩展：

XML命名空间的威力可使任何人在核心协议的基础上建造定制化的功能；为了维持通透性，常见的扩展由XMPP标准基金会。弹性佳：

XMPP除了可用在实时通信的应用程序，还能用在网络管理、内容供稿、协同工具、文件共享、游戏、远程系统监控等。多样性：

用XMPP协议来建造及布署实时应用程序及服务的公司及开放源代码计划分布在各种领域；用XMPP技术开发软件，资源及支持的来源是多样的，使得你不会陷于被“绑架”的困境。

### 缺点

#### 数据负载太重：

随着通常超过70%的XMPP协议的服务器的数据流量的存在和近60%的被重复转发，XMPP协议目前拥有一个大型架空中存在的数据提供给多个收件人。新的议定书正在研究，以减轻这一问题。

#### 没有二进制数据：

XMPP协议的方式被编码为一个单一的长的XML文件，因此无法提供修改二进制数据。因此，文件传输协议一样使用外部的HTTP。如果不可避免，XMPP协议还提供了带编码的文件传输的所有数据使用的Base64。至于其他二进制数据加密会话（encrypted conversations）或图形图标（graphic icons）以嵌入式使用相同的方法。

## NSURLConnection的几个常用的代理？

- NSURLConnectionDownloadDelegate：能够实现监听下载进度！但是下载之后，找不到下载好的文件！
- NSURLConnectionDataDelegate 是针对数据下载提供的方法！需要注意的是，需要自己实现监听进度的业务逻辑！
- 利用NSURLConnection 的异步回调进行文件下载：
  - 如果是小文件下载，问题不大！可以直接使用异步回调进行下载
  - 如果使用异步回调的方法进行大文件下载，则会出现内存暴涨的情况！
- 内存暴涨的原因：大文件下载之后，默认是放在内存中的，所以下载的文件越大，越耗费内存。
- 存在的缺点：使用异步回调实现文件，无法监听下载进度！并且对于大文件下载，会造成内存暴涨！
- 基于以上两点，一般，在进行文件下载的时候，使用代理回调监听下载进度！并且在下载文件的时候，手动管理内存！

## NSURLConnection&NSURLSession的区别？

- 虽然 NSURLConnection 在 iOS 9.0 中已经被废弃，但是作为资深的 iOS 程序员，必须要了解 NSURLConnection 的细节，
  - NSURLSession: 用于替代 NSURLConnection
  - 支持后台运行的网络任务
  - 暂停、停止、重启网络任务，不再需要 NSOperation 封装
  - 请求可以使用同样的配置容器
  - 不同的 session 可以使用不同的私有存储
  - block 和代理可以同时起作用
  - 直接从文件系统上传、下载

## XML是什么？XML与HTML的区别？

- XML的简单使其易于在任何应用程序中读写数据，这使XML很快成为数据交换的唯一公共语言，虽然不同的应用软件也支持其它的数据交换格式，但不久之后他们都将支持XML，那就意味着程序可以更容易的与Windows,Mac OS,Linux以及其他平台下产生的信息结合，然后可以很容易加载XML数据到程序中并分析他，并以XML格式输出结果。
- XML去掉了之前令许多开发人员头疼的SGML（标准通用标记语言）的随意语法。在 XML中，采用了如下的语法：
  - 任何的起始标签都必须有一个结束标签。
  - 可以采用另一种简化语法，可以在一个标签中同时表示起始和结束标签。这种语法是在大于符号之前紧跟一个斜线 (/)，例如<tag/ >。XML解析器会将其翻译成<tag></tag>。
  - 标签必须按合适的顺序进行嵌套，所以结束标签必须按镜像顺序匹配起始标签，例如this is a sample string。这好比是将起始和结束标签看作是数学中的左右括号：在没有关闭所有的内部括号之前，是不能关闭外面的括号的。
  - 所有的特性都必须有值。
  - 所有的特性都必须在值的周围加上双引号。
- XML与HTML的设计区别是：XML的核心是数据，其重点是数据的内容。而HTML被设计用来显示数据，其重点是数据的显示。
- XML和HTML语法区别：HTML的标记不是所有的都需要成对出现，XML则要求所有的标记必须成对出现；HTML标记不区分大小写，XML则大小敏感,即区分大小写。

## 网络图片处理问题中怎么解决一个相同的网络地址重复请求的问题？

利用字典（图片地址为key， 下载操作为value）

## sip是什么？

- 1> SIP (Session Initiation Protocol) , 会话发起协议
- 2> SIP是建立VOIP连接的 IETF 标准，IETF是全球互联网最具权威的技术标准化组织
- 3> 所谓VOIP，就是网络电话，直接用互联网打电话，不用耗手机话费

## TCP/IP四层模型

- TCP/IP是一组协议的代名词，它还包括许多协议，组成了TCP/IP协议簇。TCP/IP协议簇分为四层，IP位于协议簇的第二层(对应OSI的第三层)，TCP位于协议簇的第三层(对应OSI的第四层)。
- 应用层：应用程序间沟通的层，如简单电子邮件传输（SMTP）、文件传输协议（FTP）、网络远程访问协议（Telnet）等。
- 传输层：在此层中，它提供了节点间的数据传送服务，如传输控制协议（TCP）、用户数据报协议（UDP）等，TCP和UDP给数据包加入传输数据并把它传输到下一层中，这一层负责传送数据，并且确定数据已被送达并接收。
- 互连网络层：负责提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机（但不检查是否被正确接收），如网际协议（IP）。
- 网络接口层：对实际的网络媒体的管理，定义如何使用实际网络（如Ethernet、Serial Line等）来传送数据。

## (17) WebView与JS交互

### iOS中调用HTML

#### 1. 加载网页

```
NSURL *url = [[NSBundle mainBundle] URLForResource:@"index"
withExtension:@"html"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
[self.webView loadRequest:request];
```

#### 2. 删除

```
NSString *str1 = @"var word =
document.getElementById('word');");
NSString *str2 = @"word.remove();";

[webView stringByEvaluatingJavaScriptFromString:str1];
[webView stringByEvaluatingJavaScriptFromString:str2];
```

#### 3. 更改

```
NSString *str3 = @"var change =
document.getElementsByClassName('change')[0];"
"change.innerHTML = '好你的哦！';";
[webView stringByEvaluatingJavaScriptFromString:str3];
```

#### 4. 插入

```
NSString *str4 = @"var img = document.createElement('img');"
 "img.src = 'img_01.jpg';"
 "img.width = '160';"
 "img.height = '80';"
 "document.body.appendChild(img);";
[webView stringByEvaluatingJavaScriptFromString:str4];
```

#### 5. 改变标题

```
NSString *str1 = @"var h1 =
document.getElementsByTagName('h1')[0];"
 "h1.innerHTML='简书啊啊啊啊啊';";
[webView stringByEvaluatingJavaScriptFromString:str1];
```

#### 6. 删除尾部

```
NSString *str2
=@"document.getElementById('footer').remove();";
[webView stringByEvaluatingJavaScriptFromString:str2];
```

#### 7. 拿出所有的网页内容

```
NSString *str3 = @"document.body.outerHTML";
NSString *html = [webView
stringByEvaluatingJavaScriptFromString:str3];
NSLog(@"%@", html);
```

## 在HTML中调用OC

```
-(BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:
(NSURLRequest *)request navigationType:
(UIWebViewNavigationType)navigationType{
 NSString *str = request.URL.absoluteString;
 NSRange range = [str rangeOfString:@"ZJY://"];
 if (range.location != NSNotFound) {
 NSString *method = [str substringFromIndex:range.location
+ range.length];
 SEL sel = NSSelectorFromString(method);
 [self performSelector:sel];
 }
 return YES;
}

// 访问相册
- (void)getImage{
 UIImagePickerController *pickerImg = [[UIImagePickerController
alloc] init];
 pickerImg.sourceType =
 UIImagePickerControllerSourceTypePhotoLibrary;
```

```
[self presentViewController:pickerImg animated:YES
completion:nil];
}
```

## JavaScriptCore 使用

- JavaScriptCore是webkit的一个重要组成部分，主要是对JS进行解析和提供执行环境。iOS7后苹果在iPhone平台推出，极大的方便了我们对js的操作。我们可以脱离webview直接运行我们的js。iOS7以前我们对JS的操作只有webview里面一个函数stringByEvaluatingJavaScriptFromString，JS对OC的回调都是基于URL的拦截进行的操作。大家用得比较多的是WebViewJavascriptBridge和EasyJSWebView这两个开源库，很多混合都采用的这种方式。

```
#import "JSContext.h"
#import "JSValue.h"
#import "JSManagedValue.h"
#import "JSVirtualMachine.h"
#import "JSExport.h"
```

- JSContext:JS执行的环境，同时也通过JSVirtualMachine管理着所有对象的生命周期，每个JSValue都和JSContext相关联并且强引用context。
- JSValue:JS对象在JSVirtualMachine中的一个强引用，其实就是Hybird对象。我们对JS的操作都是通过它。并且每个JSValue都是强引用一个context。同时，OC和JS对象之间的转换也是通过它，相应的类型转换如下：

| Objective-C type | JavaScript type        |
|------------------|------------------------|
| nil              | undefined              |
| NSNull           | null                   |
| NSString         | string                 |
| NSNumber         | number, boolean        |
| NSDictionary     | Object object          |
| NSArray          | Array object           |
| NSDate           | Date object            |
| NSBlock (1)      | Function object (1)    |
| id (2)           | Wrapper object (2)     |
| Class (3)        | Constructor object (3) |

- JSManagedValue:JS和OC对象的内存管理辅助对象。由于JS内存管理是垃圾回收，并且JS中的对象都是强引用，而OC是引用计数。如果双方相互引用，势必会造成循环引用，而导致内存泄露。我们可以用JSManagedValue保存JSValue来避免。

- JSVirtualMachine:JS运行的虚拟机，有独立的堆空间和垃圾回收机制。
- JSExport:一个协议，如果JS对象想直接调用OC对象里面的方法和属性，那么这个OC对象只要实现这个JSExport协议就可以了。
- Objective-C -> JavaScript

```

self.context = [[JSContext alloc] init];

NSString *js = @"function add(a,b) {return a+b}";

[self.context evaluateScript:js];

JSValue *n = [self.context[@"add"] callWithArguments:@[@2,
@3]];

NSLog(@"%@", @(n.toInt32)); //---5

```

- JavaScript -> Objective-C.JS调用OC有两个方法：block和JSExport protocol。
- block(JS function):

```

self.context = [[JSContext alloc] init];

self.context[@"add"] = ^(NSInteger a, NSInteger b) {
 NSLog(@"%@", @(a + b));
};

[self.context evaluateScript:@"add(2,3)"];

```

我们定义一个block，然后保存到context里面，其实就是转换成了JS的function。然后我们直接执行这个function，调用的就是我们的block里面的内容了。

- JSExport protocol:

```

//定义一个JSExport protocol
@protocol JSExportTest <JSExport>

- (NSInteger)add:(NSInteger)a b:(NSInteger)b;

@property (nonatomic, assign) NSInteger sum;

@end

//建一个对象去实现这个协议:

@interface JSProtocolObj : NSObject<JSExportTest>
@end

@implementation JSProtocolObj
@synthesize sum = _sum;
//实现协议方法

```

```

- (NSInteger)add:(NSInteger)a b:(NSInteger)b
{
 return a+b;
}
//重写setter方法方便打印信息,
- (void)setSum:(NSInteger)sum
{
 NSLog(@"%@", @(sum));
 _sum = sum;
}

@end

//在VC中进行测试
@interface ViewController () <JSExportTest>

@property (nonatomic, strong) JSProtocolObj *obj;
@property (nonatomic, strong) JSContext *context;

@end

@implementation ViewController

- (void)viewDidLoad {
 [super viewDidLoad];
 //创建context
 self.context = [[JSContext alloc] init];
 //设置异常处理
 self.context.exceptionHandler = ^(JSContext *context, JSValue
*exception) {
 [JSContext currentContext].exception = exception;
 NSLog(@"exception:%@", exception);
 };
 //将obj添加到context中
 self.context[@"OCObj"] = self.obj;
 //JS里面调用Obj方法，并将结果赋值给Obj的sum属性
 [self.context evaluateScript:@"OCObj.sum = OCObj.addB(2,3)"];
}

}

```

demo很简单，还是定义了一个两个数相加的方法，还有一个保存结果的变量。在JS中进行调用这个对象的方法，并将结果赋值sum。唯一要注意的是OC的函数命名和JS函数命名规则问题。协议中定义的是add: b:，但是JS里面方法名字是addB(a,b)。可以通过JSExportAs这个宏转换成JS的函数名字。

- 内存管理:现在来说说内存管理的注意点，OC使用的ARC，JS使用的是垃圾回收机制，并且所有的引用都是强引用，不过JS的循环引用，垃圾回收会帮它们打破。

JavaScriptCore里面提供的API，正常情况下，OC和JS对象之间内存管理都无需我们去关心。不过还是有几个注意点需要我们去留意下。

1、不要在block里面直接使用context，或者使用外部的JSValue对象。

//错误代码：

```
self.context[@"block"] = ^{
 JSValue *value = [JSValue valueWithObject:@"aaa"
inContext:self.context];
};
```

这个代码，不用自己看了，编译器都会提示你的。这个block里面使用self，很容易就看出来了。

//一个比较隐蔽的

```
JSValue *value = [JSValue valueWithObject:@"ssss"
inContext:self.context];

self.context[@"log"] = ^{
 NSLog(@"%@", value);
};
```

这个是block里面使用了外部的value，value对context和它管理的JS对象都是强引用。这个value被block所捕获，这边同样也会内存泄露，context是销毁不掉的。

//正确的做法，str对象是JS那边传递过来。

```
self.context[@"log"] = ^(NSString *str){
 NSLog(@"%@", str);
};
```

2、OC对象不要用属性直接保存JSValue对象，因为这样太容易循环引用了。

看个demo，把上面的示例改下：

```
//定义一个JSExport protocol
@protocol JSExportTest <JSExport>
//用来保存JS的对象
@property (nonatomic, strong) JSValue *jsValue;
```

```
@end
```

//建一个对象去实现这个协议：

```
@interface JSProtocolObj : NSObject<JSExportTest>
@end
```

```
@implementation JSProtocolObj
```

```
@synthesize jsValue = _jsValue;
```

```
@end
```

```

//在VC中进行测试
@interface ViewController () <JSExportTest>

@property (nonatomic, strong) JSProtocolObj *obj;
@property (nonatomic, strong) JSContext *context;

@end

@implementation ViewController

- (void)viewDidLoad {
 [super viewDidLoad];
 //创建context
 self.context = [[JSContext alloc] init];
 //设置异常处理
 self.context.exceptionHandler = ^ (JSContext *context, JSValue
*exception) {
 [JSContext currentContext].exception = exception;
 NSLog(@"exception:%@", exception);
 };
 //加载JS代码到context中
 [self.context evaluateScript:
 @"function callback (){};

 function setObj(obj) {
 this.obj = obj;
 obj.jsValue=callback;
 }"];
 //调用JS方法
 [self.context[@"setObj"] callWithArguments:@[self.obj]];
}

```

上面的例子很简单，调用JS方法，进行赋值，JS对象保留了传进来的obj，最后，JS将自己的回调callback赋值给了obj，方便obj下次回调给JS；由于JS那边保存了obj，而且obj这边也保留了JS的回调。这样就形成了循环引用。

怎么解决这个问题？我们只需要打破obj对JSValue对象的引用即可。当然，不是我们oc中的weak。而是之前说的内存管理辅助对象JSManagedValue。

JSManagedValue 本身就是我们需要的弱引用。用官方的话来说叫garbage collection weak reference。但是它帮助我们持有JSValue对象必须同时满足一下两个条件（不翻译了，翻译了怪怪的！）：

The JSManagedValue's JavaScript value is reachable from JavaScript  
The owner of the managed reference is reachable in Objective-C.  
Manually adding or removing the managed reference in the  
JSVirtualMachine determines reachability.

意思很简单，`JSManagedValue` 帮助我们保存`JSValue`，那里面保存的JS对象必须在JS中存在，同时 `JSManagedValue` 的owner在OC中也存在。我们可以通过它提供的两个方法``` + (`JSManagedValue`)managedValueWithValue:(`JSValue`)value;  
(`JSManagedValue`)managedValueWithValue:(`JSValue`)value andOwner:(`id`)owner 创建`JSManagedValue`对象。通过`JSVirtualMachine`的方法-  
(void)addManagedReference:(`id`)object withOwner:(`id`)owner 来建立这个弱引用关系。通过- (void)removeManagedReference:(`id`)object withOwner:(`id`)owner``` 来手动移除他们之间的联系。

把刚刚的代码改下：

```
//定义一个JSExport protocol
@protocol JSExportTest <JSExport>
//用来保存JS的对象
@property (nonatomic, strong) JSValue *jsValue;

@end

//建一个对象去实现这个协议：

@interface JSProtocolObj : NSObject<JSExportTest>
//添加一个JSManagedValue用来保存JSvalue
@property (nonatomic, strong) JSManagedValue *managedValue;

@end

@implementation JSProtocolObj

@synthesize jsValue = _jsValue;
//重写setter方法
- (void)setJsValue:(JSValue *)jsValue
{
 _managedValue = [JSManagedValue
managedValueWithValue:jsValue];

 [[[JSContext currentContext] virtualMachine]
addManagedReference:_managedValue
 withOwner:self];
}

@end

//在vc中进行测试
@interface ViewController () <JSExportTest>

@property (nonatomic, strong) JSProtocolObj *obj;
```

```

@property (nonatomic, strong) JSContext *context;

@end

@implementation ViewController

- (void)viewDidLoad {
 [super viewDidLoad];
 //创建context
 self.context = [[JSContext alloc] init];
 //设置异常处理
 self.context.exceptionHandler = ^(JSContext *context, JSValue
*exception) {
 [JSContext currentContext].exception = exception;
 NSLog(@"exception:%@",exception);
 };
 //加载JS代码到context中
 [self.context evaluateScript:
 @"function callback (){};

 function setObj(obj) {
 this.obj = obj;
 obj.jsValue=callback;
 }"];
 //调用JS方法
 [self.context[@"setObj"] callWithArguments:@[self.obj]];
}

```

注：以上代码只是为了突出用 `JSManagedValue` 来保存 `JSValue`，所以重写了 `setter` 方法。实际不会写这么搓的姿势。。。应该根据回调方法传进来参数，进行保存 `JSValue`。

### 3、不要在不同的 `JSVirtualMachine` 之间进行传递JS对象。

一个 `JSVirtualMachine` 可以运行多个 `context`，由于都是在同一个堆内存和同一个垃圾回收下，所以相互之间传值是没问题的。但是如果在不同的 `JSVirtualMachine` 传值，垃圾回收就不知道他们之间的关系了，可能会引起异常。

- 线程:JavaScriptCore 线程是安全的，每个 `context` 运行的时候通过 `lock` 关联的 `JSVirtualMachine`。如果要进行并发操作，可以创建多个 `JSVirtualMachine` 实例进行操作。
- 与 `UIWebView` 的操作

通过上面的 demo，应该差不多了解 OC 如何和 JS 进行通信。下面我们看看如何对 `UIWebView` 进行操作，我们不再通过 URL 拦截，我们直接取 `UIWebView` 的 `context`，然后进行对 JS 操作。

在UIWebView的finish的回调中进行获取

```
- (void)webViewDidFinishLoad:(UIWebView *)webView
{
 self.context = [webView
valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext
"];
}
```

上面用了私有属性，可能会被苹果给拒了。这边要注意的是每个页面加载完都是一个新的context，但是都是同一个JSVirtualMachine。如果JS调用OC方法进行操作UI的时候，请注意线程是不是主线程。

## (18) 多线程

**你们项目中为什么多线程用GCD而不用NSOperation呢？你有没有发现国外的大牛他们多线程都是用NSOperation？你能告诉我他们这样做的理由吗？**

关系：

①:先搞清两者的关系,NSOperationQueue用GCD构建封装的，是GCD的高级抽象！

②:GCD仅仅支持FIFO队列，而NSOperationQueue中的队列可以被重新设置优先级，从而实现不同操作的执行顺序调整。GCD不支持异步操作之间的依赖关系设置。如果某个操作的依赖另一个操作的数据（生产者-消费者模型是其中之一），使用NSOperationQueue能够按照正确的顺序执行操作。GCD则没有内建的依赖关系支持。

③:NSOperationQueue支持KVO，意味着我们可以观察任务的执行状态。

了解以上不同，我们可以从以下角度来回答

**性能:**①:GCD更接近底层，而NSOperationQueue则更高级抽象，所以GCD在追求性能的底层操作来说，是速度最快的。这取决于使用Instruments进行代码性能分析，如有必要的话

②:从异步操作之间的事务性，顺序行，依赖关系。GCD需要自己写更多的代码来实现，而NSOperationQueue已经内建了这些支持

③:如果异步操作的过程需要更多的被交互和UI呈现出来，NSOperationQueue会是一个更好的选择。底层代码中，任务之间不太互相依赖，而需要更高的并发能力，GCD则更有优势

最后的一句话:别忘了高德纳的教诲：“在大概97%的时间里，我们应该忘记微小的性能提升。过早优化是万恶之源。”只有Instruments显示有真正的性能提升时才有必要用低级的GCD。

## 详解GCD死锁 unix上进程怎么通信?

- UNIX主要支持三种通信方式：
- 基本通信：主要用来协调进程间的同步和互斥
  - 锁文件通信:通信的双方通过查找特定目录下特定类型的文件(称锁文件)来完成进程间对临界资源访问时的互斥；例如进程p1访问一个临界资源，首先查看是否有一个特定类型文件，若有，则等待一段时间再查找锁文件。
    - 记录锁文件
- 管道通信：适应大批量的数据传递
- IPC：适应大批量的数据传递

列举几种进程的同步机制、进程的通信途径、死锁及死锁的处理方法。

- 进程的同步机制原子操作 信号量机制 自旋锁 管程，会合，分布式系统
  - 进程之间通信的途径：共享存储系统消息传递系统管道：以文件系统为基础
  - 进程死锁的原因：资源竞争及进程推进顺序非法
  - 死锁的4个必要条件：互斥、请求保持、不可剥夺、环路
  - 死锁的处理：鸵鸟策略、预防策略、避免策略、检测与解除死锁

## 线程与进程的区别和联系？

- 线程是进程的基本单位。
- 进程和线程都是由操作系统所产生的程序运行的基本单元,系统利用该基本单元实现系统对应用的并发性。
- 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。
- 进程有独立的地址空间,一个进程崩溃后,在保护模式下 不会对其它进程产生影响。
- 线程只是一个进程中的不同执行路径。
- 线程有自己的堆栈和局部变量,但线程之间没有单独的地址空间,一个线程死掉就等于整个进程死掉,所以多进程的程序要比多线程的程序健壮,但在进程切换时,耗费资源较大,效率要差一些。

- 但对于一些要求同时进行并且又要共享某些变量的并发操作,只能用线程,不能用进程。

## iOS线程间怎么通信?

- performSelector:onThread:withObject:waitUntilDone:
- NSMachPort  
(基本机制: A线程(父线程)创建NSMachPort对象, 并加入A线程的runloop。当创建B线程(辅助线程)时, 将创建的NSMachPort对象传递到主体入口点, B线程(辅助线程)就可以使用相同的端口对象将消息传回A线程(父线程)。)

## iOS多线程的底层实现?

- 首先搞清楚什么是线程、什么是多线程
  - Mach是第一个以多线程方式处理任务的系统, 因此多线程的底层实现机制是基于Mach的线程
  - 开发中很少用Mach级的线程, 因为Mach级的线程没有提供多线程的基本特征, 线程之间是独立的
- 开发中实现多线程的方案
  - C语言的POSIX接口: #include <pthread.h>
  - OC的NSThread
  - C语言的GCD接口(性能最好, 代码更精简)
  - OC的NSOperation和NSOperationQueue(基于GCD)

## 谈谈多线程安全问题的几种解决方案?何为线程同步,如何实现的?分线程回调主线程方法是什么,有什么作用?

- 解决方案: 使用锁: 锁是线程编程同步工具的基础。锁可以让你很容易保护代码中一大块区域以便你可以确保代码的正确性。
  - 使用POSIX互斥锁;
  - 使用NSLock类;
  - 使用@synchronized指令等。
- 回到主线程的方法: dispatch\_async(dispatch\_get\_main\_queue(), ^{});
- 作用: 主线程是显示UI界面,子线程多数是进行数据处理

## 使用atomic一定是线程安全的吗?

不是的。

atomic原子操作, 系统会为setter方法加锁。 具体使用 @synchronized(self){// code }

nonatomic不会为setter方法加锁。

atomic: 线程安全, 需要消耗大量系统资源来为属性加锁

nonatomic: 非线程安全, 适合内存较小的移动设备

使用atomic并不能保证绝对的线程安全, 对于要绝对保证线程安全的操作, 还需要使用更高级的方式来处理, 比如NSSpinLock、@syncronized等

# 谈谈你对多线程开发的理解(多线程的好处，多线程的作用)? ios中有几种实现多线程的方法?

- 好处:
  - 使用线程可以把占据时间长的程序中的任务放到后台去处理
  - 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
  - 程序的运行效率可能提高
  - 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。
- 缺点:
  - 如果有大量的线程,会影响性能,因为操作系统需要在它们之间切换。
  - 更多的线程需要更多的内存空间。
  - 线程的中止需要考虑其对程序运行的影响。
- 通常块模型数据是在多个线程间共享的，需要防止线程死锁情况的发生。
- 实现多线程的方法:
  1. NSObject的类方法 // -(void)performSelectorInBackground/OnMainThread:(SEL)aSelector withObject:(id)arg
  2. NSThread
  3. NSOperation
  4. GCD

## OC中异步使用的哪种事件模型,iOS中异步实现机制

- 异步非阻塞 I/O (AIO)

## 详细谈谈GCD

1. 推出的时间 iOS4 目的是用来取代NSThread (ios2.0推出) 的，是 C语言框架，它能够自动利用更多CPU的核数，并且会自动管理线程的生命周期。
  - CGD的两个核心概念：任务，队列
  - 任务：记为在block中执行的代码。
  - 队列：用来存放任务的。
  - 注意事项：队列 != 线程。队列中存放的任务最后都要由线程来执行!。队列的原则:先进先出,后进后出(FIFO/ First In First Out)
2. 队列又分为四种种：1 串行队列 2 并发队列 3 主队列 4 全局队列
  - 串行队列：任务一个接一个的执行。
  - 并发队列：队列中的任务并发执行。
  - 主队列：跟主线程相关的队列，主队列里面的内容都会在主线程中执行（我们一般在主线程中刷新UI）。
  - 全局队列：一个特殊的并发队列。
3. 并发队列与全局队列的区别：
  - 并发队列有名称,可以跟踪错误。全局队列没有

- 在ARC中两个队列不需要考虑释放内存,但是在MRC中并发队列是创建出来的需要release操作,而全局队列只有一个不需要。
- 一般在开发过程中我们使用全局队列。

## 1. 执行任务的两个函数

- '同步'执行任务:dispatch\_sync(<#dispatch\_queue\_t queue#>, <#^(void)block#>)
- '异步'执行任务:dispatch\_async(dispatch\_queue\_t queue, <#^(void)block#>)

## 2. "同步"和"异步"的区别:

- "同步": 只能在'当前'线程中执行任务,不具备开启新线程的能力.
- "异步": 可以在'新'的线程中执行任务,具备开启新线程的能力.

## 3. 各个队列的执行效果:

- 串行队列同步执行,既在当前线程中顺序执行
- 串行队列异步执行,开辟一条新的线程,在该线程中顺序执行
- 并行队列同步执行,不开辟线程,在当前线程中顺序执行
- 并行队列异步执行,开辟多个新的线程,并且线程会重用,无序执行
- 主队列异步执行,不开辟新的线程,顺序执行
- 主队列同步执行,会造成死锁('主线程'和'主队列'相互等待,卡住主线程)

## 4. 线程间通讯: 经典案例: 子线程进行耗时操作(例如下载更新等) 主线程进行UI刷新。

- 经典用法(子线程下载(耗时操作),主线程刷新UI):

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
 // 执行耗时的异步操作...
 dispatch_async(dispatch_get_main_queue(), ^{
 // 回到主线程, 执行UI刷新操作
 });
});
```

## 1. 延迟操作

调用 NSObject 方法:[self performSelector:@selector(run) withObject:nil afterDelay:2.0];

// 2秒后再调用self的run方法

GCD函数实现延时执行:dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)

(2.0 \* NSEC\_PER\_SEC)), dispatch\_get\_main\_queue(), ^{

// 2秒后执行这里的代码... 在哪个线程执行, 跟队列类型有关

## 1. 队列组的使用:

- 项目需求:首先:分别异步执行两个耗时操作;其次:等两次耗时操作都执行完毕后,再回到主线程执行操作.使用队列组(dispatch\_group\_t)快速,高效的实现上述需求.

```
dispatch_group_t group = dispatch_group_create(); // 队列组
```

```

dispatch_queue_t queue = dispatch_get_global_queue(0, 0); // 全局并发队列
dispatch_group_async(group, queue, ^{
 // 异步执行操作1
 // longTime1
});
dispatch_group_async(group, queue, ^{
 // 异步执行操作2
 // longTime2
});
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
 // 在主线程刷新数据
 // reload Data
});

```

## GCD内部怎么实现的

- iOS和OS X的核心是XNU内核，GCD是基于XNU内核实现的
- GCD的API全部在libdispatch库中
- GCD的底层实现主要有Dispatch Queue和Dispatch Source
  - Dispatch Queue：管理block(操作)
  - Dispatch Source：处理事件(MACH端口发送,MACH端口接收,检测与进程相关事件等10种事件)

## GCD的queue、main queue中执行的代码一定是在main thread么？

- 对于queue中所执行的代码不一定在main thread中。如果queue是在主线程中创建的，那么所执行的代码就是在主线程中执行。如果是在子线程中创建的，那么就不会在main thread中执行。
- 对于main queue就是在主线程中的，因此一定会在主线程中执行。获取main queue就可以了，不需要我们创建，获取方式通过调用方法dispatch\_get\_main\_queue来获取。

## 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。

```

dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,);
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
});

```

```
});
```

## 有a、b、c、d 4个异步请求，如何判断a、b、c、d都完成执行？如果需要a、b、c、d顺序执行，该如何实现？

1. 对于这四个异步请求，要判断都执行完成最简单的方式就是通过GCD的group来实现：

```
2. dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
3. dispatch_group_t group = dispatch_group_create();
4. dispatch_group_async(group, queue, ^{
 /*任务a */ });
5. dispatch_group_async(group, queue, ^{
 /*任务b */ });
6. dispatch_group_async(group, queue, ^{
 /*任务c */ });
7. dispatch_group_async(group, queue, ^{
 /*任务d */ });
8.
9. dispatch_group_notify(group, dispatch_get_main_queue(), ^{
10. // 在a、b、c、d异步执行完成后，会回调这里
11. });

```

当然，我们还可以使用非常老套的方法来处理，通过四个变量来标识a、b、c、d四个任务是否完成，然后在runloop中让其等待，当完成时才退出run loop。但是这样做会让后面的代码得不到执行，直到Run loop执行完毕。

要求顺序执行，那么可以将任务放到串行队列中，自然就是按顺序来异步执行了

## 发送10个网络请求，然后再接收到所有回应之后执行后续操作，如何实现？

从题目分析可知，10个请求要全部完成后，才执行某一功能。比如，下载10图片后合成一张大图，就需要异步全部下载完成后，才能合并成大图。

做法：通过dispatch\_group\_t来实现，将每个请求放入到Group中，将合并成大图的操作放在dispatch\_group\_notify中实现。

```
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
});

```

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

- dispatch\_get\_current\_queue容易造成死锁。详情点击该API查看官方注释。

## 如果让你来实现 `dispatch_once`, 你会怎么做?

- [http://www.dreamingwish.com/article/gcd-guide-dispatch-once-2.html \(超级详细解析\)](http://www.dreamingwish.com/article/gcd-guide-dispatch-once-2.html)
- 个人觉得说出实现的思路即可，无锁的线程同步编程，每一处的线程竞争都考虑到并妥善处理
- 线程A执行Block时，任何其它线程都需要等待。
- 线程A执行完Block应该立即标记任务完成状态，然后遍历信号量链来唤醒所有等待线程。
- 线程A遍历信号量链来signal时，任何其他新进入函数的线程都应该直接返回而无需等待。
- 线程A遍历信号量链来signal时，若有其它等待线程B仍在更新或试图更新信号量链，应该保证此线程B能正确完成其任务：a.直接返回 b.等待在信号量上并很快又被唤醒。
- 线程B构造信号量时，应该考虑线程A随时可能改变状态（“等待”、“完成”、“遍历信号量链”）。
- 线程B构造信号量时，应该考虑到另一个线程C也可能正在更新或试图更新信号量链，应该保证B、C都能正常完成其任务：a.增加链节并等待在信号量上 b.发现线程A已经标记“完成”然后直接销毁信号量并退出函数。

## 关于`NSOperation`:

- `NSOperation`: 抽象类,不能直接使用,需要使用其子类.(类似的类还有核心动画)
- 两个常用子类: `NSInvocationOperation`(调用) 和 `NSBlockOperation`(块);
- 两者没有本质区别,后者使用 Block 的形式组织代码,使用相对方便.
- `NSInvocationOperation`在调用start方法后, 不会开启新的线程只会在当前线程中执行。
- `NSBlockOperation` 在调用start方法后, 如果封装的操作数>1会开辟多条线程执行 =1 只会在当前线程执行.
- `NSOperationQueue` 创建的操作队列默认为全局队列, 队列中的操作执行顺序是无序的, 如果需要让它有序执行需要添加依赖关系。
- // 操作op3依赖于操作op2 [op3 addDependency:op2];
- // 操作op2依赖于操作op1 [op2 addDependency:op1];
- 同时可以设置最大并发数
- `NSOperationQueue` `NSOperation`支持 取消暂停的操作 但是正在进行的操作并不能取消, 这一旦取消不可恢复。
  - `NSOperationQueue`支持KVO, 可以监测operation是否正在执行 (`isExecuted`) 、是否结束 (`isFinished`) , 是否取消 (`isCancelled`)

## `NSOperation queue`?

- 存放`NSOperation`的集合类。不能说队列, 不是严格的先进先出

## NSOperation与GCD的区别

- GCD
  - 1. GCD是iOS4.0推出的，主要针对多核cpu做了优化，是纯c语言的技术。
  - 2. GCD是将任务（block）添加到队列（串行、并行、全局、主队列），并且以同步/异步的方式执行任务的函数。
  - 3. GCD提供了一些NSOperation不具备的功能
    - 一次性执行
    - 延迟执行
    - 调度组
    - GCD 是严格的队列，先进先出 FIFO；
- NSOperation
  - 1. NSOperation是iOS2.0推出的，iOS4.0以后又重写了NSOperation
  - 2. NSOperation是将操作（异步的任务）添加到队列（并发队列），就会执行指定的函数
  - 3. NSOperation提供的方便操作
    - 最大并发数
    - 队列的暂停和继续
    - 取消所有的操作
    - 指定操作之间的依赖关系依赖关系，可以让异步任务同步执行.
    - 将KVO用于NSOperation中，监听一个operation是否完成。
    - 能够设置NSOperation的优先级，能够使同一个并行队列中的任务区分先后地执行
    - 对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度

## GCD与NSThread的区别

- NSThread 通过 @selector 指定要执行的方法，代码分散
- GCD 通过 block 指定要执行的代码，代码集中，所有的代码写在一起的，让代码更加简单，易于阅读和维护
- 使用 GCD 不需要管理线程的创建/销毁/复用的过程！程序员不用关心线程的生命周期
- 如果要开多个线程 NSThread 必须实例化多个线程对象
- NSThread 靠 NSObject 的分类方法实现的线程间通讯，

## 为什么要取消/恢复队列呢？

- 一般在内存警告后取消队列中的操作。
- 为了保证scrollView在滚动的时候流畅 通常在滚动开始时，暂停队列中的所有操作，滚动结束后，恢复操作。

## Object C中创建线程的方法是什么？如果在主线程中执行代码，方法是什么？如果想延时执行代码、方法又是什么？

线程创建有三种方法：使用NSThread创建、使用GCD的dispatch、使用子类化的NSOperation，然后将其加入NSOperationQueue；

NSThread创建线程的三种方法：

```
NSThread *thread = [[NSThread alloc] initWithTarget:self
selector:@selector(run:) object:@“nil”];
[NSThread detachNewThreadSelector:@selector(run:) toTarget:self
withObject:@“我是分离出来的子线程”];
[self performSelectorInBackground:@selector(run:) withObject:@“我是
后台线程”];
```

在主线程执行代码，就调用performSelectorOnMainThread方法。

如果想延时执行代码可以调用

performSelector:onThread:withObject:waitUntilDone:方法；

GCD：

利用异步函数dispatch\_async()创建子线程。

在主线程执行代码， dispatch\_async(dispatch\_get\_main\_queue(), ^{});

延迟执行代码（延迟·可以控制代码在哪个线程执行）：

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(2.0 *
NSEC_PER_SEC)),
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^{});
```

NSOperationQueue：

使用NSOperation的子类封装操作，再将操作添加到NSOperationQueue创建的队列中，实现多线程。

在主线程执行代码，只要将封装代码的NSOperation对象添加到主队列就可以了。

## 下面关于线程管理错误的是

- A. GCD所用的开销要比NSThread大
- B. 可以在子线程中修改UI元素
- C. NSOperationQueue是比NSThread更高层的封装
- D. GCD可以根据不同优先级分配线程

- 参考答案：B
- 理由：首先，UI元素的更新必须在主线程。GCD与Block配合使用，block需要自动捕获上下文变量信息等，因此需要更多的资源，故比NSThread开销要大一些。  
NSOperationQueue与NSOperation配合使用，比NSThread更易于操作线程。GCD提供了多个优先级，我们可以根据设置优先级，让其自动为我们分配线程。

## (19) 多媒体

**iPhone OS主要提供了几种播放音频的方法?**

SystemSound Services

AVAudioPlayer 类

Audio Queue Services

OpenAL

**使用AVAudioPlayer类调用哪个框架、使用步骤?**

AVFoundation.framework

步骤：配置 AVAudioPlayer 对象；

实现 AVAudioPlayer 类的委托方法；

控制 AVAudioPlayer 类的对象；

监控音量水平；

回放进度和拖拽播放。

## (20) 设计模式

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

三者优缺点：

delegate的优势：

- 1.非常严格的语法。所有将听到的事件必须是在delegate协议中有清晰的定义。
- 2.如果delegate中的一个方法没有实现那么就会出现编译警告/错误
- 3.协议必须在controller的作用域范围内定义
- 4.在一个应用中的控制流程是可跟踪的并且是可识别的；
- 5.在一个控制器中可以定义多个不同的协议，每个协议有不同的delegates
- 6.没有第三方对象要求保持/监视通信过程。
- 7.能够接收调用的协议方法的返回值。这意味着delegate能够提供反馈信息给controller
- 8.经常被用在存在父子关系的对象之间通信，例如控制器和控制器的view（自己加的理解）

缺点：

- 1.需要定义很多代码：1.协议定义；2.controller的delegate属性；3.在delegate本身中实现delegate方法定义
- 2.在释放代理对象时，需要小心的将delegate改为nil。一旦设定失败，那么调用释放对象的方法将会出现内存crash
- 3.在一个controller中有多个delegate对象，并且delegate是遵守同一个协议，但还是很难告诉多个对象同一个事件，不过有可能。
- 4.经常用在一一对的通信。（不知道是缺点还是优点，只能算是特点）（自己加的理解）

notification的优势：

- 1.不需要编写多少代码，实现比较简单
- 2.对于一个发出的通知，多个对象能够做出反应，即一对多的方式实现简单
- 3.controller能够传递context对象（dictionary），context对象携带了关于发送通知的自定义的信息

缺点：

- 1.在编译期不会检查通知是否能够被观察者正确的处理；

2. 在释放注册的对象时，需要在通知中心取消注册；
3. 在调试的时候应用的工作以及控制过程难跟踪；
4. 需要第三方对象来管理controller与观察者对象之间的联系；
5. controller和观察者需要提前知道通知名称、UserInfo dictionary keys。如果这些没有在工作区间定义，那么会出现不同步的情况；
6. 通知发出后，controller不能从观察者获得任何的反馈信息（相比较delegate）。

KVO的优势：

1. 能够提供一种简单的方法实现两个对象间的同步。例如：model和view之间同步；
2. 能够对非我们创建的对象，即内部对象的状态改变作出响应，而且不需要改变内部对象（SKD对象）的实现；
3. 能够提供观察的属性的最新值以及先前值；
4. 用key paths来观察属性，因此也可以观察嵌套对象；
5. 完成了对观察对象的抽象，因为不需要额外的代码来允许观察值能够被观察
6. 可以一对多。

缺点：

1. 我们观察的属性必须使用strings来定义。因此在编译器不会出现警告以及检查；
2. 对属性重构将导致我们的观察代码不再可用；
3. 复杂的“IF”语句要求对象正在观察多个值。这是因为所有的观察代码通过一个方法来指向；
4. 当释放观察者时不需要移除观察者。

1. 效率肯定是delegate比NSNotification高。

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

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

should这个很传神的词。也就是好比你做我的delegate，我会问你我想关闭窗口你愿意吗？你需要给我一个答案，我根据你的答案来决定如何做下一步。相反的，notification最大的特色就是不关心接受者的态度，

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

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

受者的反应。

## 2、KVO和NSNotification的区别：

和delegate一样，KVO和NSNotification的作用也是类与类之间的通信，与delegate不同的是

1) 这两个都是负责发出通知，剩下的事情就不管了，所以没有返回值；2) delegate只是一对一，而这两个可以一对多。这两者也有各自的特点。

总结：

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

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

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

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

## 什么是设计模式

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

## 设计模式的分类

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

## 类工厂方法是什么？

- 类工厂方法的实现是为了向客户提供方便，它们将分配和初始化合在一个步骤中，返回被创建的对象，并进行自动释放处理。
- 这些方法的形式是+ (type)className...（其中 className 不包括任何前缀）。
- 工厂方法可能不仅仅为了方便使用。它们不但可以将分配和初始化合在一起，还可以为初始化过程提供对象的分配信息。
- 类工厂方法的另一个目的是使类（比如 NSWorkspace）提供单例。虽然 init... 方法可以确认一个类在每次程序运行过程只存在一个实例，但它需要首先分配一个“生的”实例，然后还必须释放该实例。工厂方法则可以避免为可能没有用的对象盲目分配内存。

## 单例是什么？

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

### 1. 单例模式的要点：

显然单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

### 1. 单例模式的优点：

- 实例控制：Singleton 会阻止其他对象实例化其自己的 Singleton 对象的副本，从而确保所有对象都访问唯一实例。
- 灵活性：因为类控制了实例化过程，所以类可以更加灵活修改实例化过程

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

image image image image image image image image image image

## 简要描述观察者模式，并运用此模式编写一段代码？

- 观察者模式（Observer）是指一个或多个对象对另一个对象进行观察，当被观察对象发生变化时，观察者可以直接或间接地得到通知，从而能自动地更新观察者的数据，或者进行一些操作。
- 具体到iOS的开发中，实现观察者模式常用的方式有KVO和Notification两种。
- 两者不同在于，KVO是被观察者主动向观察者发送消息；Notification是被观察者向NotificationCenter发送消息，再由NotificationCenter post通知到每个注册的观察者。

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

- MVC就是Model-View-Controller的缩写,M指的是业务模型,V指的是用户页面,C指的是控制器。MVC是架构模式,是讲M和V的代码分离,从而使同一个程序可以使用不同的表现形式。
- 单例,代理,观察者,工厂模式等
- 单例模式:上面有详细说明
- 代理模式:代理模式给某一个对象提供一个代理对象,并由代理对象控制对源对象的引用.比如一个工厂生产了产品,并不想直接卖给用户,而是搞了很多代理商,用户可以直接找代理商买东西,代理商从工厂进货.常见的如QQ的自动回复就属于代理拦截,代理模式在iphone中得到广泛应用.

## MVC优点不正确的是

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

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

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

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

优点：

层次更加分明显清晰

代码简洁优雅

减少VC的复杂性

代码和界面完全分离

方便测试

缺点：

MVVM需要使用数据绑定机制，对于os x 开发，可以直接使用Coocoa Binding，对于ios，没有太好的数据绑定方法，可以使用kvo，但如果需要绑定的属性太多的话，需要编写大量的selector代码。

ReactiveCocoa提供了一种很方便优雅的绑定机制。

## ReactiveCocoa

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

## 哪些途径可以让 ViewController 瘦下来？

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

## 你在你的项目中用到了哪些设计模式？

项目中使用了很多的设计模式，我相信面试官最好听到的不仅仅是设计模式的名字，更想听到的是这些设计模式在项目中如何应用。因此，笔者认为这个问题隐式地说明了应该回答设计模式及其在项目中的应用。

参考答案：

- 单例设计模式：在项目中，单例是必不可少的。比如UIApplication、NSUserDefaults就是苹果提供的单例。在项目中经常会将用户数据管理封装成一个单例类，因此用户的信息需要全局使用。
  - MVC设计模式：现在绝大部分项目都是基于MVC设计模式的，现在有一部分开发者采用MVVM、MVP等模式。
  - 通知(NSNotification)模式：通知在开发中是必不可少的，对于跨模块的类交互，需要使用通知；对于多对多的关系，使用通知更好实现。
  - 工厂设计模式：在我的项目中使用了大量的工厂设计模式，特别是生成控件的API，都已经封装成一套，全部是扩展的类方法，可简化很多的代码。
  - KVC/KVO设计模式：有的时候需要监听某个类的属性值的变化而做出相应的改变，这时候会使用KVC/KVO设计模式。在项目中，我需要监听model中的某个属性值的变化，当变化时，需要更新UI显示，这时候使用KVC/KVO设计模式就很方便了。
- 就说这么多吧，还有很多的设计模式，不过其它并不是那么常用。

## 如何实现单例，单例会有什么弊端？

单例在项目中的是必不可少的，它可以使我们全局都可共享我们的数据。这只是简单的问题，大家根据自己的情况回答。

参考答案：

- 首先，单例写法有好几种，通常的写法是基于线程安全的写法，结合dispatch\_once来使用，保证单例对象只会被创建一次。如果不小心销毁了单例，再调用单例生成方法是不会再创建的。
- 其次，由于单例是约定俗成的，因此在实际开发中通常不会去重写内存管理方法。单例确实给我们带来的便利，但是它也会有代价的。单例一旦创建，整个App使用过程都不会释放，这会占用内存，因此不可滥用单例。

## 你在你的项目中用到了哪些设计模式？

- 设计模式有很多，面试官肯定不想听你把项目里的设计模式名字报给他，他想听得肯定是你是怎么去用这些设计模式的
- 参考答案：
- MVC：这个设计模型大部分应用应该都在用，介绍下MVC就好
- 单例：单例在项目中用的还是蛮多的，像登录界面，对一些第三方框架二次封装等等
- KVC/KVO：这个用的应该也很多，KVC用来替换掉系统的tabbar，用KVO来监听偏移量来完成下拉刷新，改变导航条背景颜色这些
- 工厂方法：这个用的更多了，设置一些自定义View肯定要用到这个设计模式
- 如何实现单例，单例会有什么弊端？
- 这个问题还是蛮简单的，说下单例是怎么写的，单例的缺点就好

- 参考答案:

// OC版

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

 static dispatch_once_t onceToken;
 dispatch_once(&onceToken, ^{
 sharedInstance = [[self alloc] init];
 });

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

- 单例的缺点也就是会一直占着这块内存,不会被释放

## 单例书写

- 伪单例

1、获取单例对象的方法

- (DataHandle \*)sharedDataHandle; // 创建单例对象的方法。类方法 命名规则: shared + 类名

2、方法的实现

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

static DataHandle \*handle = nil;

// 伪单例 和 完整的单例。 以及线程的安全。

// 一般使用伪单例就足够了 每次都用 sharedDataHandle 创建对象。

- (DataHandle \*)sharedDataHandle

{

// 添加同步锁, 一次只能一个线程访问。如果有多个线程访问, 等待。一个访问结束后下一个。

@synchronized(self){ if (nil == handle) {

handle = [[DataHadle alloc] init];

}

}

return handle;

}

- 完整单例

## 完整的单例

完整的单例要求比较高，不仅要求我们通过方法获取的对象是单例，如果有对该对象进行 copy mutableCopy copyWithZone 等操作时，也是同一份对象。这就要求我们必须重写这些方法，在这些方法内部做一些操作。

完整的单例要做到四个方面：

为单例对象实现一个静态实例，然后设置成nil，

构造方法检查静态实例是否为nil，是则新建并返回一个实例，

重写allocWithZone方法，用来保证其他人直接使用alloc和init试图获得一个新实例的时候不会产生一个新实例，

适当实现copyWithZone, ,retain,retainCount,release和autorelease 等方法

### 1、获取单例对象的方法

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

```
@synchronized(self){
 if (nil == handle) {

 handle = [[super allocWithZone:nil] init]; // 避免死循环
 // 如果 在单例类里面重写了 allocWithZone 方法 ， 在创建单例对象
 // 时 使用 [[DataHandle alloc] init] 创建，会死循环。
 }
}
return handle;
```

### 3、重写 allocWithZone copy mutableCopy copyWithZone

防止外界拷贝造成多个实例，保证实例的唯一性。

注意：如果自己重写了 allocWithZone 就不要再调用自身的 alloc 方法，否则会出现死循环。

- (instancetype)allocWithZone:(struct \_NSZone \*)zone  
{  
 return [DataHandle sharedDataHandle];  
}
- (id)copy  
{  
 return self;  
}
- (id)mutableCopy  
{  
 return self;  
}

- (id)copyWithZone:(struct \_NSZone \*)zone  
{  
return self;  
}  
4、重写 alloc retain release autorelease retainCount
- (instancetype)alloc  
{  
return [DataHandle sharedDataHandle];  
}

// 因为只有一个实例，一直不释放，所以不增加引用计数。无意义。

- (instancetype)retain  
{  
return self;  
}
- (oneway void)release  
{  
// nothing  
}
- (instancetype)autorelease  
{  
return self;  
}
- (NSUInteger)retainCount  
{  
return NSUIntegerMax; // 返回整形最大值。 }

## (21) 安全机制

### 苹果的安全机制有哪些

- 没经过用户同意，你不能随便获取用户信息。
- 所有的程序都在沙盒里运行，B程序不能进入A程序的运行范围。
- 如果跟钱有关，比如说支付宝，这些底层的实现都是保密的，只提供接口供开发者调用，这样的话安全性得到保障。
- 如果要防止代码被反编译，可以将自己的代码中的.m文件封装成静态库（.a文件）或者是framework文件，只提供给其它人.h文件。这样就保证了个人代码的安全性。
- 网络登录的话跟用户名跟密码相关要发送POST请求，如果是GET请求的话密码会直接在URL中显示。然后同时要对帐号密码采用加密技术，加一句：我们公司用的是MD5，但是现在MD5有一个专门的网站来破解，为了防止这个，可以采用加盐技术。

### iOS 的签名机制大概是什么样的？

- 假设，我们有一个APP需要发布，为了防止中途篡改APP内容，保证APP的完整性，以及APP是由指定的私钥发的。首先，先将APP内容通过摘要算法，得到摘要，再用私钥对摘要进行加密得到密文，将源文本、密文、和私钥对应的公钥一并发布即可。
- 那么如何验证呢？  
验证方首先查看公钥是否是私钥方的，然后用公钥对密文进行解密得到摘要，将APP用同样的摘要算法得到摘要，两个摘要进行比对，如果相等那么一切正常。这个过程只要有一步出问题就视为无效。

### 客户端安全性处理方式？

- 1> 网络数据传输(敏感数据[账号\密码\消费数据\银行卡账号]，不能明文发送)
- 2> 协议的问题(自定义协议，游戏代练)
- 3> 本地文件存储(游戏的存档)
- 4> 源代码(混淆)

### 如何进行数据加密？

常见的加密算法:MD5\SHA\DES\3DES\RC2和RC4\RSA\IDEA\DSA\AES

加密算法的选择:一般公司都会有一套自己的加密方案，按照公司接口文档的规定去加密

MD5的特点:

- (1) 输入两个不同的明文不会得到相同的输出值
- (2) 根据输出值，不能得到原始的明文，即其过程不可逆

现在的MD5已不再是绝对安全，对此，可以对MD5稍作改进，以增加解密的难度

加盐 (Salt) : 在明文的固定位置插入随机串，然后再进行MD5

先加密，后乱序: 先对明文进行MD5，然后对加密得到的MD5串的字符进行乱序

总之宗旨就是: 黑客就算攻破了数据库，也无法解密出正确的明文

## (22) Runtime

### Runtime是什么

Runtime 又叫运行时，是一套底层的 C 语言 API，其为 iOS 内部的核心之一，我们平时编写的 OC 代码，底层都是基于它来实现的。比如：

```
[receiver message];
// 底层运行时会被编译器转化为:
objc_msgSend(receiver, selector)
// 如果其还有参数比如:
[receiver message:(id)arg...];
// 底层运行时会被编译器转化为:
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

### 为什么需要Runtime

- Objective-C 是一门动态语言，它会将一些工作放在代码运行时才处理而并非编译时。也就是说，有很多类和成员变量在我们编译的时候是不知道的，而在运行时，我们所编写的代码会转换成完整的确定的代码运行。
- 因此，编译器是不够的，我们还需要一个运行时系统(Runtime system)来处理编译后的代码。
- Runtime 基本是用 C 和汇编编写的，由此可见苹果为了动态系统的高效而做出的努力。苹果和 GNU 各自维护一个开源的 Runtime 版本，这两个版本之间都在努力保持一致。

### Runtime 的作用

OC 在三种层面上与 Runtime 系统进行交互：

#### 1. 通过 Objective-C 源代码

只需要编写 OC 代码，Runtime 系统自动在幕后搞定一切，调用方法，编译器会将 OC 代码转换成运行时代码，在运行时确定数据结构和函数。

#### 2. 通过 Foundation 框架的 NSObject 类定义的方法

Cocoa 程序中绝大部分类都是 NSObject 类的子类，所以都继承了 NSObject 的行为。（NSProxy 类是个例外，它是个抽象超类）

一些情况下，NSObject 类仅仅定义了完成某件事情的模板，并没有提供所需要的代码。例如 -description 方法，该方法返回类内容的字符串表示，该方法主要用来调试程序。NSObject 类并不知道子类的内容，所以它只是返回类的名字和对象的地址，NSObject 的子类可以重新实现。

还有一些NSObject的方法可以从Runtime系统中获取信息，允许对象进行自我检查。例如：

-class方法返回对象的类；  
-isKindOfClass: 和 -isMemberOfClass: 方法检查对象是否存在与指定的类的继承体系中(是否是其子类或者父类或者当前类的成员变量)；  
-respondsToSelector: 检查对象能否响应指定的消息；  
-conformsToProtocol: 检查对象是否实现了指定协议类的方法；  
-methodForSelector: 返回指定方法实现的地址。

### 3. 通过对 Runtime 库函数的直接调用

Runtime 系统是具有公共接口的动态共享库。头文件存放于/usr/include/objc 目录下，这意味着我们使用时只需要引入objc/Runtime.h头文件即可。

许多函数可以让你使用纯 c 代码来实现 Objc 中同样的功能。除非是写一些 Objc 与其他语言的桥接或是底层的 debug 工作，你在写 Objc 代码时一般不会用到这些 c 语言函数。

## Runtime的相关术语

### 1. SEL

它是selector在 Objc 中的表示(Swift 中是 Selector 类)。selector 是方法选择器，其实作用就和名字一样，日常生活中，我们通过人名辨别谁是谁，注意 Objc 在相同的类中不会有命名相同的两个方法。selector 对方法名进行包装，以便找到对应的方法实现。它的数据结构是：

```
typedef struct objc_selector *SEL;
```

我们可以看出它是个映射到方法的 C 字符串，你可以通过 Objc 编译器命令

@selector() 或者 Runtime 系统的 sel\_registerName 函数来获取一个 SEL 类型的方法选择器。

注意：不同类中相同名字的方法所对应的 selector 是相同的，由于变量的类型不同，所以不会导致它们调用方法实现混乱。

### 2.id

id 是一个参数类型，它是指向某个类的实例的指针。定义如下：

```
typedef struct objc_object *id;
struct objc_object { Class isa; };
```

以上定义，看到 objc\_object 结构体包含一个 isa 指针，根据 isa 指针就可以找到对象所属的类。

注意：isa 指针在代码运行时并不总指向实例对象所属的类型，所以不能依靠它来确定类型，要想确定类型还是需要用对象的 -class 方法。PS:KVO 的实现机理就是将被观察对象的 isa 指针指向一个中间类而不是真实类型。

### 3.Class

```
typedef struct objc_class *Class;
```

Class 其实是指向 objc\_class 结构体的指针。objc\_class 的数据结构如下：

```
struct objc_class {
 Class isa _OBJC_ISA_AVAILABILITY;
```

```

#if !__OBJC2__
 Class super_class
OBJC2_UNAVAILABLE;
 const char *name
OBJC2_UNAVAILABLE;
 long version
OBJC2_UNAVAILABLE;
 long info
OBJC2_UNAVAILABLE;
 long instance_size
OBJC2_UNAVAILABLE;
 struct objc_ivar_list *ivars
OBJC2_UNAVAILABLE;
 struct objc_method_list **methodLists
OBJC2_UNAVAILABLE;
 struct objc_cache *cache
OBJC2_UNAVAILABLE;
 struct objc_protocol_list *protocols
OBJC2_UNAVAILABLE;
#endif
} OBJC2_UNAVAILABLE;

```

从 `objc_class` 可以看到，一个运行时类中关联了它的父类指针、类名、成员变量、方法、缓存以及附属的协议。

其中 `objc_ivar_list` 和 `objc_method_list` 分别是成员变量列表和方法列表：

```

// 成员变量列表
struct objc_ivar_list {
 int ivar_count
OBJC2_UNAVAILABLE;
#ifndef __LP64__
 int space
OBJC2_UNAVAILABLE;
#endif
 /* variable length structure */
 struct objc_ivar ivar_list[1]
OBJC2_UNAVAILABLE;
}
OBJC2_UNAVAILABLE;

// 方法列表
struct objc_method_list {
 struct objc_method_list *obsolete
OBJC2_UNAVAILABLE;

 int method_count
OBJC2_UNAVAILABLE;
#ifndef __LP64__
 int space
OBJC2_UNAVAILABLE;

```

```
#endif
/* variable length structure */
struct objc_method method_list[1]
OBJC2_UNAVAILABLE;
}
```

由此可见，我们可以动态修改 \*methodList 的值来添加成员方法，这也是 Category 实现的原理，同样解释了 Category 不能添加属性的原因。

objc\_ivar\_list 结构体用来存储成员变量的列表，而 objc\_ivar 则是存储了单个成员变量的信息；同理，objc\_method\_list 结构体存储着方法数组的列表，而单个方法的信息则由 objc\_method 结构体存储。

值得注意的是，objc\_class 中也有一个 isa 指针，这说明 Objc 类本身也是一个对象。为了处理类和对象的关系，Runtime 库创建了一种叫做 Meta Class(元类) 的东西，类对象所属的类就叫做元类。Meta Class 表达了类对象本身所具备的元数据。

我们所熟悉的类方法，就源自于 Meta Class。我们可以理解为类方法就是类对象的实例方法。每个类仅有一个类对象，而每个类对象仅有一个与之相关的元类。

当你发出一个类似 [NSObject alloc](类方法) 的消息时，实际上，这个消息被发送给了一个类对象(Class Object)，这个类对象必须是一个元类的实例，而这个元类同时也是一个根元类(Root Meta Class)的实例。所有元类的 isa 指针最终都指向根元类。

所以当 [NSObject alloc] 这条消息发送给类对象的时候，运行时代码 objc\_msgSend() 会去它元类中查找能够响应消息的方法实现，如果找到了，就会对这个类对象执行方法调用。

最后 objc\_class 中还有一个 objc\_cache，缓存，它的作用很重要，后面会提到。

#### 4.Method

Method 代表类中某个方法的类型

```
typedef struct objc_method *Method;
struct objc_method {
 SEL method_name
OBJC2_UNAVAILABLE;
 char *method_types
OBJC2_UNAVAILABLE;
 IMP method_imp
OBJC2_UNAVAILABLE;
}
```

objc\_method 存储了方法名，方法类型和方法实现：

方法名类型为 SEL

方法类型 method\_types 是个 char 指针，存储方法的参数类型和返回值类型

`method_implementation` 指向了方法的实现，本质是一个函数指针

`Ivar`

`Ivar` 是表示成员变量的类型。

```
typedef struct objc_ivar *Ivar;
struct objc_ivar {
 char *ivar_name
OBJC2_UNAVAILABLE;
 char *ivar_type
OBJC2_UNAVAILABLE;
 int ivar_offset
OBJC2_UNAVAILABLE;
#ifndef __LP64__
 int space
OBJC2_UNAVAILABLE;
#endif
}
```

其中 `ivar_offset` 是基地址偏移字节

## 5. IMP

`IMP` 在 `objc.h` 中的定义是：

```
typedef id (*IMP)(id, SEL, ...);
```

它就是一个函数指针，这是由编译器生成的。当你发起一个 ObjC 消息之后，最终它会执行的那段代码，就是由这个函数指针指定的。而 `IMP` 这个函数指针就指向了这个方法的实现。

如果得到了执行某个实例某个方法的入口，我们就可以绕开消息传递阶段，直接执行方法，这在后面 `Cache` 中会提到。

你会发现 `IMP` 指向的方法与 `objc_msgSend` 函数类型相同，参数都包含 `id` 和 `SEL` 类型。每个方法名都对应一个 `SEL` 类型的方法选择器，而每个实例对象中的 `SEL` 对应的方法实现肯定是唯一的，通过一组 `id` 和 `SEL` 参数就能确定唯一的方法实现地址。

而一个确定的方法也只有唯一的一组 `id` 和 `SEL` 参数。

## 6. Cache

`Cache` 定义如下：

```
typedef struct objc_cache *Cache
struct objc_cache {
 unsigned int mask /* total = mask + 1 */
OBJC2_UNAVAILABLE;
 unsigned int occupied
OBJC2_UNAVAILABLE;
 Method buckets[1]
OBJC2_UNAVAILABLE;
};
```

Cache 为方法调用的性能进行优化，每当实例对象接收到一个消息时，它不会直接在 `isa` 指针指向的类的方法列表中遍历查找能够响应的方法，因为每次都要查找效率太低了，而是优先在 Cache 中查找。

Runtime 系统会把被调用的方法存到 Cache 中，如果一个方法被调用，那么它有可能今后还会被调用，下次查找的时候就会效率更高。就像计算机组成原理中 CPU 绕过主存先访问 Cache 一样。

7.Property  
typedef struct objc\_property \*Property;  
typedef struct objc\_property \*objc\_property\_t;//这个更常用  
可以通过`class_copyPropertyList` 和 `protocol_copyPropertyList` 方法获取类和协议中的属性：

```
objc_property_t *class_copyPropertyList(Class cls, unsigned int *outCount)
objc_property_t *protocol_copyPropertyList(Protocol *proto, unsigned int *outCount)
```

注意：

返回的是属性列表，列表中每个元素都是一个 `objc_property_t` 指针

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
/** 姓名 */
@property (strong, nonatomic) NSString *name;
/** age */
@property (assign, nonatomic) int age;
/** weight */
@property (assign, nonatomic) double weight;
@end
```

以上是一个 Person 类，有3个属性。让我们用上述方法获取类的运行时属性。

```
unsigned int outCount = 0;

objc_property_t *properties = class_copyPropertyList([Person class], &outCount);

NSLog(@"%@", outCount);

for (NSInteger i = 0; i < outCount; i++) {
 NSString *name = @(property_getName(properties[i]));
 NSString *attributes =
 @(property_getAttributes(properties[i]));
 NSLog(@"%@", name, attributes);
}
```

打印结果如下：

```
test[2321:451525] 3
test[2321:451525] name-----T@ "NSString", &, N, V_name
```

```
test[2321:451525] age-----Ti,N,V_age
test[2321:451525] weight-----Td,N,V_weight
property_getName 用来查找属性的名称，返回 c 字符串。
property_getAttributes 函数挖掘属性的真实名称和 @encode 类型，返回 c 字符串。

objc_property_t class_getProperty(Class cls, const char *name)
objc_property_t protocol_getProperty(Protocol *proto, const char
 *name, BOOL isRequiredProperty, BOOL isInstanceProperty)
class_getProperty 和 protocol_getProperty 通过给出属性名在类和协议中获得
属性的引用。
```

## Runtime与消息

- 一些 Runtime 术语讲完了，接下来就要说到消息了。体会苹果官方文档中的 messages aren't bound to method implementations until Runtime。消息直到运行时才会与方法实现进行绑定。
- 这里要清楚一点，objc\_msgSend 方法看不清来好像返回了数据，其实objc\_msgSend 从不返回数据，而是你的方法在运行时实现被调用后才会返回数据。下面详细叙述消息发送的步骤：
  - 首先检测这个 selector 是不是要忽略。比如 Mac OS X 开发，有了垃圾回收就不理会 retain, release 这些函数。
  - 检测这个 selector 的 target 是不是 nil，Objc 允许我们对一个 nil 对象执行任何方法不会 Crash，因为运行时会被忽略掉。
  - 如果上面两步都通过了，那么就开始查找这个类的实现 IMP，先从 cache 里查找，如果找到了就运行对应的函数去执行相应的代码。
  - 如果 cache 找不到就找类的方法列表中是否有对应的方法。
  - 如果类的方法列表中找不到就到父类的方法列表中查找，一直找到 NSObject 类为止。如果还找不到，就要开始进入动态方法解析了，后面会提到。
- 在消息的传递中，编译器会根据情况在 objc\_msgSend , objc\_msgSend\_stret , objc\_msgSendSuper , objc\_msgSendSuper\_stret 这四个方法中选择一个调用。如果消息是传递给父类，那么会调用名字带有 Super 的函数，如果消息返回值是数据结构而不是简单值时，会调用名字带有 stret 的函数。

## 方法中的隐藏参数

疑问：

我们经常用到关键字 self ，但是 self 是如何获取当前方法的对象呢？

其实，这也是 Runtime 系统的作用， self 实在方法运行时被动态传入的。

当 `objc_msgSend` 找到方法对应实现时，它将直接调用该方法实现，并将消息中所有参数都传递给方法实现，同时，它还将传递两个隐藏参数：

接受消息的对象 (`self` 所指向的内容，当前方法的对象指针)

方法选择器 (`_cmd` 指向的内容，当前方法的 `SEL` 指针)

因为在源代码方法的定义中，我们并没有发现这两个参数的声明。它们时在代码被编译时被插入方法实现中的。尽管这些参数没有被明确声明，在源代码中我们仍然可以引用它们。

这两个参数中，`self` 更实用。它是在方法实现中访问消息接收者对象的实例变量的途径。

这时我们可能会想到另一个关键字 `super`，实际上 `super` 关键字接收到消息时，编译器会创建一个 `objc_super` 结构体：

```
struct objc_super { id receiver; Class class; };
```

这个结构体指明了消息应该被传递给特定的父类。`receiver` 仍然是 `self` 本身，当我们想通过 `[super class]` 获取父类时，编译器其实是将指向 `self` 的 `id` 指针和 `class` 的 `SEL` 传递给了 `objc_msgSendSuper` 函数。只有在 `NSObject` 类中才能找到 `class` 方法，然后 `class` 方法底层被转换为 `object_getClass()`，接着底层编译器将代码转换为 `objc_msgSend(objc_super->receiver, @selector(class))`，传入的第一个参数是指向 `self` 的 `id` 指针，与调用 `[self class]` 相同，所以我们得到的永远都是 `self` 的类型。因此你会发现：

```
// 这句话并不能获取父类的类型，只能获取当前类的类型名
NSLog(@"%@", NSStringFromClass([super class]));
```

获取方法地址

`NSObject` 类中有一个实例方法：`methodForSelector`，你可以用它来获取某个方法选择器对应的 `IMP`，举个例子：

```
void (*setter)(id, SEL, BOOL);
int i;

setter = (void (*)(id, SEL, BOOL))[target
 methodForSelector:@selector(setFilled:)];
for (i = 0 ; i < 1000 ; i++)
 setter(targetList[i], @selector(setFilled:), YES);
```

当方法被当做函数调用时，两个隐藏参数也必须明确给出，上面的例子调用了1000次函数，你也可以尝试给 `target` 发送1000次 `setFilled:` 消息会花多久。

虽然可以更高效的调用方法，但是这种做法很少用，除非时需要持续大量重复调用某个方法的情况，才会选择使用以免消息发送泛滥。

注意：

`methodForSelector:` 方法是由 `Runtime` 系统提供的，而不是 `objc` 自身的特性  
动态方法解析

你可以动态提供一个方法实现。如果我们使用关键字 `@dynamic` 在类的实现文件中修饰一个属性，表明我们会为这个属性动态提供存取方法，编译器不会再默认为我们生成这个属性的 `setter` 和 `getter` 方法了，需要我们自己提供。

```
@dynamic propertyName;
```

这时，我们可以通过分别重载 `resolveInstanceMethod:` 和 `resolveClassMethod:` 方法添加实例方法实现和类方法实现。

当 `Runtime` 系统在 `Cache` 和类的方法列表(包括父类)中找不到要执行的方法时，`Runtime` 会调用 `resolveInstanceMethod:` 或 `resolveClassMethod:` 来给我们一次动态添加方法实现的机会。我们需要用 `class_addMethod` 函数完成向特定类添加特定方法实现的操作：

```
void dynamicMethodIMP(id self, SEL _cmd) {
 // implementation
}
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
 if (aSEL == @selector(resolveThisMethodDynamically)) {
 class_addMethod([self class], aSEL, (IMP)
dynamicMethodIMP, "v@:");
 return YES;
 }
 return [super resolveInstanceMethod:aSEL];
}
@end
```

上面的例子为 `resolveThisMethodDynamically` 方法添加了实现内容，就是 `dynamicMethodIMP` 方法中的代码。其中 `"v@:"` 表示返回值和参数，这个符号表示的含义见：[Type Encoding](#)

注意：

动态方法解析会在消息转发机制侵入前执行，动态方法解析器将会首先给予提供该方法选择器对应的 `IMP` 的机会。如果你想让该方法选择器被传送到转发机制，就让 `resolveInstanceMethod:` 方法返回 `NO`。

## 消息转发

### 1. 重定向

消息转发机制执行前，`Runtime` 系统允许我们替换消息的接收者为其他对象。通过 `- (id)forwardingTargetForSelector:(SEL)aSelector` 方法。

```
- (id)forwardingTargetForSelector:(SEL)aSelector
{
 if(aSelector == @selector(mysteriousMethod:)){
```

```
 return alternateObject;
 }
 return [super forwardingTargetForSelector:aSelector];
}
```

如果此方法返回 `nil` 或者 `self`, 则会计入消息转发机制(`forwardInvocation:`), 否则将向返回的对象重新发送消息。

## 2. 转发

当动态方法解析不做处理返回 `NO` 时, 则会触发消息转发机制。这时 `forwardInvocation:` 方法会被执行, 我们可以重写这个方法来自定义我们的转发逻辑:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
 if ([someOtherObject respondsToSelector:
 [anInvocation selector]])
 [anInvocation invokeWithTarget:someOtherObject];
 else
 [super forwardInvocation:anInvocation];
}
```

唯一参数是个 `NSInvocation` 类型的对象, 该对象封装了原始的消息和消息的参数。我们可以实现 `forwardInvocation:` 方法来对不能处理的消息做一些处理。也可以将消息转发给其他对象处理, 而不抛出错误。

注意: 参数 `anInvocation` 是从哪来的?

在 `forwardInvocation:` 消息发送前, `Runtime` 系统会向对象发送 `methodSignatureForSelector:` 消息, 并取到返回的方法签名用于生成 `NSInvocation` 对象。所以重写 `forwardInvocation:` 的同时也要重写 `methodSignatureForSelector:` 方法, 否则会抛异常。

当一个对象由于没有相应的方法实现而无法相应某消息时, 运行时系统将通过 `forwardInvocation:` 消息通知该对象。每个对象都继承了 `forwardInvocation:` 方法。但是, `NSObject` 中的方法实现只是简单的调用了 `doesNotRecognizeSelector:`。通过实现自己的 `forwardInvocation:` 方法, 我们可以将消息转发给其他对象。

`forwardInvocation:` 方法就是一个不能识别消息的分发中心, 将这些不能识别的消息转发给不同的接收对象, 或者转发给同一个对象, 再或者将消息翻译成另外的消息, 亦或者简单的“吃掉”某些消息, 因此没有响应也不会报错。这一切都取决于方法的具体实现。

注意:

`forwardInvocation:` 方法只有在消息接收对象中无法正常响应消息时才会被调用。所以, 如果我们向往一个对象将一个消息转发给其他对象时, 要确保这个对象不能有该消息的所对应的方法。否则, `forwardInvocation:` 将不可能被调用。

转发和多继承

转发和继承相似，可用于为 Objc 编程添加一些多继承的效果。就像下图那样，一个对象把消息转发出去，就好像它把另一个对象中的方法接过来或者“继承”过来一样。

这使得在不同继承体系分支下的两个类可以实现“继承”对方的方法，在上图中 `Warrior` 和 `Diplomat` 没有继承关系，但是 `Warrior` 将 `negotiate` 消息转发给了 `Diplomat` 后，就好似 `Diplomat` 是 `Warrior` 的超类一样。

消息转发弥补了 Objc 不支持多继承的性质，也避免了因为多继承导致单个类变得臃肿复杂。

## 转发与继承

虽然转发可以实现继承的功能，但是 `NSObject` 还是必须表面上很严谨，像 `respondsToSelector:` 和 `isKindOfClass:` 这类方法只会考虑继承体系，不会考虑转发链。

`Warrior` 对象被问到是否能响应 `negotiate` 消息：

```
if ([aWarrior respondsToSelector:@selector(negotiate)])
 ...
```

回答当然是 NO，尽管它能接受 `negotiate` 消息而不报错，因为它靠转发消息给 `Diplomat` 类响应消息。

如果你就是想要让别人以为 `Warrior` 继承到了 `Diplomat` 的 `negotiate` 方法，你得重新实现 `respondsToSelector:` 和 `isKindOfClass:` 来加入你的转发算法：

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
 if ([super respondsToSelector:aSelector])
 return YES;
 else {
 /* Here, test whether the aSelector message can *
 * be forwarded to another object and whether that *
 * object can respond to it. Return YES if it can. */
 }
 return NO;
}
```

除了 `respondsToSelector:` 和 `isKindOfClass:` 之外，  
`instancesRespondToSelector:` 中也应该写一份转发算法。如果使用了协议，  
`conformsToProtocol:` 同样也要加入到这一行列中。

如果一个对象想要转发它接受的任何远程消息，它得给出一个方法标签来返回准确的方法描述 `methodSignatureForSelector:`，这个方法会最终响应被转发的消息。从而生成一

个确定的 `NSInvocation` 对象描述消息和消息参数。这个方法最终响应被转发的消息。它需要像下面这样实现：

```
- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector
{
 NSMethodSignature* signature = [super
methodSignatureForSelector:selector];
 if (!signature) {
 signature = [surrogate
methodSignatureForSelector:selector];
 }
 return signature;
}
```

## 健壮的实例变量(Non Fragile ivars)

在 `Runtime` 的现行版本中，最大的特点就是健壮的实例变量了。当一个类被编译时，实例变量的内存布局就形成了，它表明访问类的实例变量的位置。实例变量一次根据自己所占空间而产生位移：

上图左是 `NSObject` 类的实例变量布局。右边是我们写的类的布局。这样子有一个很大的缺陷，就是缺乏拓展性。哪天苹果更新了 `NSObject` 类的话，就会出现问题：

我们自定义的类的区域和父类的区域重叠了。只有苹果将父类改为以前的布局才能拯救我们，但这样导致它们不能再拓展它们的框架了，因为成员变量布局被固定住了。在脆弱的实例变量(`Fragile ivar`)环境下，需要我们重新编译继承自 `Apple` 的类来恢复兼容。如果是健壮的实例变量的话，如下图：

在健壮的实例变量下，编译器生成的实例变量布局跟以前一样，但是当 `Runtime` 系统检测到与父类有部分重叠时它会调整你新添加的实例变量的位移，那样你再子类中新添加的成员变量就被保护起来了。

注意：

在健壮的实例变量下，不要使用 `siof(SomeClass)`，而是用 `class_getInstanceSize([SomeClass class])` 代替；也不要使用 `offsetof(SomeClass, SomeIvar)`，而要使用 `ivar_getOffset(class_getInstanceVariable([SomeClass class], "SomeIvar"))` 来代替。

总结

我们让自己的类继承自 `NSObject` 不仅仅是因为基类有很多复杂的内存分配问题，更是因为这使得我们可以享受到 `Runtime` 系统带来的便利。

虽然平时我们很少会考虑一句简单的调用方法，发送消息底层所做的复杂的操作，但深入理解 Runtime 系统的细节使得我们可以利用消息机制写出功能更强大的代码。

## runtime实现的机制是什么,怎么用，一般用于干嘛. 你还能记得你所使用的相关的头文件或者某些方法的名称吗?

- 需要导入<objc/message.h><objc/runtime.h>
- runtime，运行时机制，它是一套C语言库
- 实际上我们编写的所有OC代码，最终都是转成了runtime库的东西，比如类转成了 runtime库里面的结构体等数据类型，方法转成了runtime库里面的C语言函数，平时调方法都是转成了objc\_msgSend函数（所以说OC有个消息发送机制）
- 因此，可以说runtime是OC的底层实现，是OC的幕后执行者
- 有了runtime库，能做什么事情呢？ runtime库里面包含了跟类、成员变量、方法相关的API，比如获取类里面的所有成员变量，为类动态添加成员变量，动态改变类的方法实现，为类动态添加新的方法等
- 因此，有了runtime，想怎么改就怎么改

## Objective-C 如何对已有的方法，添加自己的功能代码以实现类似记录日志这样的功能？

这题目主要考察的是runtime如何交换方法。先在分类中添加一个方法,注意不能重写系统方法,会覆盖

```
+ (NSString *)myLog
{
 // 这里写打印行号,什么方法,哪个类调用等等
}

// 加载分类到内存的时候调用
+(void)load
{
 // 获取imageWithName方法地址
 Method description = class_getClassMethod(self,
@selector(description));

 // 获取imageWithName方法地址
 Method myLog = class_getClassMethod(self, @selector(myLog));

 // 交换方法地址, 相当于交换实现方式
 method_exchangeImplementations(description, myLog);
}
```

## 如何让 Category 支持属性?

使用runtime可以实现

头文件

```
@interface NSObject (test)
@property (nonatomic, copy) NSString *name;
@end
```

.m文件

```
@implementation NSObject (test)
// 定义关联的key
static const char *key = "name";
-(NSString *)name
{
 // 根据关联的key, 获取关联的值。
 return objc_getAssociatedObject(self, key);
}
-(void)setName:(NSString *)name
{
 // 第一个参数: 给哪个对象添加关联
 // 第二个参数: 关联的key, 通过这个key获取
 // 第三个参数: 关联的value
 // 第四个参数: 关联的策略
 objc_setAssociatedObject(self, key, name,
 OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
```

## Toll-Free Bridging 是什么? 什么情况下会使用?

- Toll-Free Bridging用于在Foundation对象与Core Foundation对象之间交换数据,俗称桥接
- 在ARC环境下,Foundation对象转成 Core Foundation对象
- 使用\_\_bridge桥接以后ARC会自动管理2个对象

- 使用`_bridge_retained`桥接需要手动释放Core Foundation对象
- 在ARC环境下, Core Foundation对象转成 Foundation对象
- 使用`_bridge`桥接,如果Core Foundation对象被释放,Foundation对象也同时不能使用了,需要手动管理Core Foundation对象
- 使用`_bridge_transfer`桥接,系统会自动管理2个对象

## **performSelector:withObject:afterDelay: 内部大概是怎么实现的,有什么注意事项么?**

- 创建一个定时器,时间结束后系统会使用runtime通过方法名称(Selector本质就是方法名称)去方法列表中找到对应的方法实现并调用方法
- 注意事项
  - 调用`performSelector:withObject:afterDelay:`方法时,先判断希望调用的方法是否存在`respondsToSelector:`
  - 这个方法是异步方法,必须在主线程调用,在子线程调用永远不会调用到想调用的方法

## **什么是 Method Swizzle (黑魔法) , 什么情况下会使用?**

- 在没有一个类的实现源码的情况下, 想改变其中一个方法的实现, 除了继承它重写、和借助类别重名方法暴力抢先之外, 还有更加灵活的方法Method Swizzle。
- Method swizzling指的是改变一个已存在的选择器对应的实现的过程。OC中方法的调用能够在运行时通过改变——通过改变类的调度表 (dispatch table) 中选择器到最终函数间的映射关系。
- 在OC中调用一个方法, 其实是向一个对象发送消息, 查找消息的唯一依据是selector的名字。利用OC的动态特性, 可以实现在运行时偷换selector对应的方法实现。
- 每个类都有一个方法列表, 存放着selector的名字和方法实现的映射关系。IMP有点类似函数指针, 指向具体的Method实现。
- 我们可以利用`method_exchangeImplementations`来交换2个方法中的IMP,
- 我们可以利用`class_replaceMethod`来修改类,
- 我们可以利用`method_setImplementation`来直接设置某个方法的IMP,
- 归根结底, 都是偷换了selector的IMP

## **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;

解释如下:

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。

## 为什么其他语言里叫函数调用， objective c里则是给对象发消息 (或者谈下对runtime的理解)

先来看看怎么理解发送消息的含义：

[receiver message]会被编译器转化为：

`objc_msgSend(receiver, selector)`

如果消息含有参数，则为：

`objc_msgSend(receiver, selector, arg1, arg2, ...)`

如果消息的接收者能够找到对应的selector，那么就相当于直接执行了接收者这个对象的特定方法；否则，消息要么被转发，或是临时向接收者动态添加这个selector对应的实现内容，要么就干脆玩完崩溃掉。

现在可以看出[receiver message]真的不是一个简简单单的方法调用。因为这只是在编译阶段确定了要向接收者发送message这条消息，而receive将要如何响应这条消息，那就要看运行时发生的情况来决定了。

OC 的 Runtime 铸就了它动态语言的特性，Objc Runtime使得C具有了面向对象能力，在程序运行时创建，检查，修改类、对象和它们的方法。可以使用runtime的一系列方法实现。

顺便附上oc中一个类的数据结构 /usr/include/objc/runtime.h

```
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;
OC中一个类的对象实例的数据结构 (/usr/include/objc/objc.h)：
typedef struct objc_class *Class; // Represents an instance
of a class. struct objc_object {
 Class isa
OBJC_ISA_AVAILABILITY;
}; // A pointer to an instance of a class.
typedef struct objc_object *id;
```

向object发送消息时，Runtime库会根据object的isa指针找到这个实例object所属于的类，然后在类的方法列表以及父类方法列表寻找对应的方法运行。id是一个objc\_object结构类型的指针，这个类型的对象能够转换成任何一种对象。

然后再来看看消息发送的函数：objc\_msgSend函数

在引言中已经对objc\_msgSend进行了一点介绍，看起来像是objc\_msgSend返回了数据，其实objc\_msgSend从不返回数据而是你的方法被调用后返回了数据。下面详细叙述下消息发送步骤：

检测这个 selector 是不是要忽略的。比如 Mac OS X 开发，有了垃圾回收就不理会retain, release 这些函数了。

检测这个 target 是不是 nil 对象。ObjC 的特性是允许对一个 nil 对象执行任何一个方法不会 Crash，因为会被忽略掉。

如果上面两个都过了，那就开始查找这个类的 IMP，先从 cache 里面找，完了找得到就跳到对应的函数去执行。

如果 cache 找不到就找一下方法分发表。

如果分发表找不到就到超类的分发表去找，一直找，直到找到NSObject类为止。

如果还找不到就要开始进入动态方法解析了，后面会提到。

后面还有：

动态方法解析resolveThisMethodDynamically

消息转发forwardingTargetForSelector

## runtime如何实现weak属性？

- 通过关联属性来实现：
- // 声明一个weak属性，这里假设delegate，其实weak关键字可以不使用，
- // 因为我们重写了getter/setter方法
- @property (nonatomic, weak) id delegate;
- 
- - (id)delegate {

```
• return objc_getAssociatedObject(self, @"__delegate_key");
• }
•
• // 指定使用OBJC_ASSOCIATION_ASSIGN, 官方注释是:
• // Specifies a weak reference to the associated object.
• // 也就是说对于对象类型, 就是weak了
• - (void)setDelegate:(id)delegate {
• objc_setAssociatedObject(self, @"__delegate_key",
delegate, OBJC_ASSOCIATION_ASSIGN);
• }
• 通过objc_storeWeak函数来实现, 不过这种方式几乎没有遇到有人这么使用过,
因为这里不细说了。
```

## runtime如何通过selector找到对应的IMP地址?

- 每个selector都与对应的IMP是一一对应的关系, 通过selector就可以直接找到对应的IMP:

## 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。

IMP msgForward = \_objc\_msgForward;

如果手动调用objcmsgForward, 将跳过查找IMP的过程, 而是直接触发“消息转发”, 进入如下流程:

- 第一步: + (BOOL)resolveInstanceMethod:(SEL)sel实现方法, 指定是否动态添加方法。若返回NO, 则进入下一步, 若返回YES, 则通过class\_addMethod函数动态地添加方法, 消息得到处理, 此流程完毕。
- 第二步: 在第一步返回的是NO时, 就会进入-(id)forwardingTargetForSelector:(SEL)aSelector方法, 这是运行时给我们的第二次机会, 用于指定哪个对象响应这个selector。不能指定为self。若返回nil, 表示没有响应者, 则会进入第三步。若返回某个对象, 则会调用该对象的方法。
- 第三步: 若第二步返回的是nil, 则我们首先要通过- (NSMethodSignature \*)methodSignatureForSelector:(SEL)aSelector指定方法签名, 若返回nil, 则表示不处理。若返回方法签名, 则会进入下一步。
- 第四步: 当第三步返回方法方法签名后, 就会调用-(void)forwardInvocation:(NSInvocation \*)anInvocation方法, 我们可以通过anInvocation对象做很多处理, 比如修改实现方法, 修改响应对象等

- 第五步：若没有实现- (void)forwardInvocation:(NSInvocation \*)anInvocation方法，那么会进入- (void)doesNotRecognizeSelector:(SEL)aSelector方法。若我们没有实现这个方法，那么就会crash，然后提示打不到响应的方法。到此，动态解析的流程就结束了。

## 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。

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）

## 动态绑定

- 在运行时确定要调用的方法，动态绑定将调用方法的确定也推迟到运行时。在编译时，方法的调用并不和代码绑定在一起，只有在消息发送出来之后，才确定被调用的代码。通过动态类型和动态绑定技术，代码每次执行都可以得到不同的结果。运行时因子负责确定消息的接收者和被调用的方法。运行时的消息分发机制为动态绑定提供支持。当向一个动态类型确定了的对象发送消息时，运行环境系统会通过接收者的isa指针定位对象的类，并以此为起点确定被调用的方法，方法和消息是动态绑定的。而且，不必在Objective-C 代码中做任何工作，就可以自动获取动态绑定的好处。在每次发送消息时，特别是当消息的接收者是动态类型已经确定的对象时，动态绑定就会例行而透明地发生

## (23) 性能优化

### 入门级

1. 用ARC管理内存
  2. 在正确的地方使用 reuseIdentifier
  3. 尽量把views设置为透明
  4. 避免过于庞大的XIB
  5. 不要阻塞主线程
- 
1. 在ImageViews中调整图片大小。如果要在UIImageView中显示一个来自bundle的图片，你应保证图片的大小和UIImageView的大小相同。在运行中缩放图片是很耗费资源的，特别是UIImageView嵌套在UIScrollView中的情况下。如果图片是从远端服务加载的你不能控制图片大小，比如在下载前调整到合适大小的话，你可以在下载完成后，最好是用background thread，缩放一次，然后在UIImageView中使用缩放后的图片。
  2. 选择正确的Collection。
    - Arrays: 有序的一组值。使用index来lookup很快，使用value lookup很慢，插入/删除很慢。
    - Dictionaries: 存储键值对。用键来查找比较快。
    - Sets: 无序的一组值。用值来查找很快，插入/删除很快。
  3. 打开gzip压缩。app可能大量依赖于服务器资源，问题是我们的目标是移动设备，因此你就不能指望网络状况有多好。减小文档的一个方式就是在服务端和你的app中打开gzip。这对于文字这种能有更高压缩率的数据来说会有更显著的效用。iOS已经在NSURLConnection中默认支持了gzip压缩，当然AFNetworking这些基于它的框架亦然。

### 中级

1. 重用和延迟加载(lazy load) Views
  - 更多的view意味着更多的渲染，也就是更多的CPU和内存消耗，对于那种嵌套了很多view在UIScrollView里边的app更是如此。

- 这里我们用到的技巧就是模仿UITableView和UICollectionView的操作: 不要一次创建所有的Subview, 而是当需要时才创建, 当它们完成了使命, 把他们放进一个可重用的队列中。这样的话你就只需要在滚动发生时创建你的views, 避免了不划算的内存分配。

## 2. Cache, Cache, 还是Cache!

- 一个极好的原则就是, 缓存所需要的, 也就是那些不大可能改变但是需要经常读取的东西。
- 我们能缓存些什么呢? 一些选项是, 远端服务器的响应, 图片, 甚至计算结果, 比如UITableView的行高。
- NSCache和NSDictionary类似, 不同的是系统回收内存的时候它会自动删掉它的内容。

## 3. 权衡渲染方法.性能能还是要bundle保持合适的大小。

## 4. 处理内存警告.移除对缓存, 图片object和其他一些可以重创建的objects的strong references.

## 5. 重用大开销对象

## 6. 一些objects的初始化很慢, 比如NSDateFormatter和NSCalendar。然而, 你又不可避免地需要使用它们, 比如从JSON或者XML中解析数据。想要避免使用这个对象的瓶颈你就需要重用他们, 可以通过添加属性到你的class里或者创建静态变量来实现。

## 7. 避免反复处理数据.在服务器端和客户端使用相同的数据结构很重要。

## 8. 选择正确的数据格式.解析JSON会比XML更快一些, JSON也通常更小更便于传输。从iOS5起有了官方内建的JSON deserialization 就更加方便使用了。但是XML也有XML的好处, 比如使用SAX 来解析XML就像解析本地文件一样, 你不需要像解析json一样等到整个文档下载完成才开始解析。当你处理很大的数据的时候就会极大地减低内存消耗和增加性能。

## 9. 正确设定背景图片

- 全屏背景图, 在view中添加一个UIImageView作为一个子View
- 只是某个小的view的背景图, 你就需要用UIColor的colorWithPatternImage来做了, 它会更快地渲染也不会花费很多内存:

10. 减少使用Web特性。想要更高的性能你就要调整下你的HTML了。第一件要做的事就是尽可能移除不必要的javascript，避免使用过大的框架。能只用原生js就更好了。尽可能异步加载例如用户行为统计script这种不影响页面表达的javascript。注意你使用的图片，保证图片的符合你使用的大小。
11. Shadow Path 。Core Animation不得不先在后台得出你的图形并加好阴影然后才渲染，这开销是很大的。使用shadowPath的话就避免了这个问题。使用shadow path的话iOS就不必每次都计算如何渲染，它使用一个预先计算好的路径。但问题是自己计算path的话可能在某些View中比较困难，且每当view的frame变化的时候你都需要去update shadow path.

## 12. 优化Table View

- 正确使用reuseIdentifier来重用cells
- 尽量使所有的view opaque，包括cell自身
- 避免渐变，图片缩放，后台选人
- 缓存行高
- 如果cell内现实的内容来自web，使用异步加载，缓存请求结果
- 使用shadowPath来画阴影
- 减少subviews的数量
- 尽量不适用cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果
- 使用正确的数据结构来存储数据
- 使用rowHeight, sectionFooterHeight 和 sectionHeaderHeight来设定固定的高，不要请求delegate

## 13. 选择正确的数据存储选项

- NSUserDefaults的问题是什么？虽然它很nice也很便捷，但是它只适用于小数据，比如一些简单的布尔型的设置选项，再大点你就要考虑其它方式了
- XML这种结构化档案呢？总体来说，你需要读取整个文件到内存里去解析，这样是很不经济的。使用SAX又是一个很麻烦的事情。
- NSCoder？不幸的是，它也需要读写文件，所以也有以上问题。
- 在这种应用场景下，使用SQLite 或者 Core Data比较好。使用这些技术你用特定的查询语句就能只加载你需要的对象。
- 在性能层面来讲，SQLite和Core Data是很相似的。他们的不同在于具体使用方法。
- Core Data代表一个对象的graph model，但SQLite就是一个DBMS。
- Apple在一般情况下建议使用Core Data，但是如果你有理由不使用它，那么就去使用更加底层的SQLite吧。

- 如果你使用SQLite，你可以用FMDB这个库来简化SQLite的操作，这样你就不用花很多经历了解SQLite的C API了。

# 高级

1. 加速启动时间。快速打开app是很重要的，特别是用户第一次打开它时，对app来讲，第一印象太太太重要了。你能做的就是使它尽可能做更多的异步任务，比如加载远端或者数据库数据，解析数据。避免过于庞大的XIB，因为他们是在主线程上加载的。所以尽量使用没有这个问题的Storyboards吧！一定要把设备从Xcode断开来测试启动速度
  2. 使用Autorelease Pool。NSAutoreleasePool负责释放block中的autoreleased objects。一般情况下它会自动被UIKit调用。但是有些状况下你也需要手动去创建它。假如你创建很多临时对象，你会发现内存一直在减少直到这些对象被release的时候。这是因为只有当UIKit用光了autorelease pool的时候memory才会被释放。消息是你可以在你自己的@autoreleasepool里创建临时的对象来避免这个行为。
  3. 选择是否缓存图片。常见的从bundle中加载图片的方式有两种，一个是指定 imageNamed，二是用imageWithContentsOfFile，第一种比较常见一点。
  4. 避免日期格式转换。如果你要用NSDateFormatter来处理很多日期格式，应该小心以待。就像先前提到的，任何时候重用NSDateFormatters都是一个好的实践。如果你可以控制你所处理的日期格式，尽量选择Unix时间戳。你可以方便地从时间戳转换到NSDate:
    - (NSDate\*)dateFromUnixTimestamp:(NSTimeInterval)timestamp {  
    return [NSDate dateWithTimeIntervalSince1970:timestamp];  
}

这样会比用C来解析日期字符串还快！需要注意的是，许多web API会以微妙的形式返回时间戳，因为这种格式在javascript中更方便使用。记住用dateFromUnixTimestamp之前除以1000就好了。

## 平时你是如何对代码进行性能优化的？

- 利用性能分析工具检测，包括静态 Analyze 工具，以及运行时 Profile 工具，通过 Xcode 工具栏中 Product->Profile 可以启动，启动后界面如下：

image image image image image image image image image image

- 比如测试程序启动运行时间，当点击Time Profiler应用程序开始运行后.就能获取到整个应用程序运行消耗时间分布和百分比.为了保证数据分析在统一使用场景真实需要注意一定要使用真机,因为此时模拟器是运行在Mac上，而Mac上的CPU往往比iOS设

备要快。

- 为了防止一个应用占用过多的系统资源，开发iOS的苹果工程师们设计了一个“看门狗”的机制。在不同的场景下，“看门狗”会监测应用的性能。如果超出了该场景所规定的运行时间，“看门狗”就会强制终结这个应用的进程。开发者们在crashlog里面，会看到诸如0x8badf00d这样的错误代码。

## 优化Table View

- 正确使用reuseIdentifier来重用cells
- 尽量使所有的view opaque，包括cell自身
- 如果cell内现实的内容来自web，使用异步加载，缓存请求结果
- 减少subviews的数量
- 尽量不适用cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果
- 使用rowHeight, sectionFooterHeight和 sectionHeaderHeight来设定固定的高，不要请求delegate

## UIImage加载图片性能问题

- imageNamed初始化
- imageWithContentsOfFile初始化
- imageNamed默认加载图片成功后会内存中缓存图片,这个方法用一个指定的名字在系统缓存中查找并返回一个图片对象.如果缓存中没有找到相应的图片对象,则从指定地方加载图片然后缓存对象，并返回这个图片对象.
- imageWithContentsOfFile则仅只加载图片,不缓存.
- 加载一张大图并且使用一次，用imageWithContentsOfFile是最好,这样CPU不需要做缓存节约时间.
- 使用场景需要编程时，应该根据实际应用场景加以区分，UIImage虽小，但使用元素较多问题会有所凸显.
  - 不要在viewWillAppear 中做费时的操作：viewWillAppear: 在view显示之前被调用，出于效率考虑，方法中不要处理复杂费时操作；在该方法设置 view 的显示属性之类的简单事情，比如背景色，字体等。否则，会明显感觉到 view 有卡顿或者延迟。
  - 在正确的地方使用reuseIdentifier: table view用tableView:cellForRowAtIndexPath:为rows分配cells的时候，它的数据应该重用自UITableViewCell。
  - 尽量把views设置为透明：如果你有透明的Views你应该设置它们的opaque属性为YES。系统用一个最优的方式渲染这些views。这个简单的属性在IB或者代码里都可以设定。
  - 避免过于庞大的XIB：尽量简单的为每个Controller配置一个单独的XIB，尽可能把一个View Controller的view层次结构分散到单独的XIB中去，当你加载一个引用了图片或者声音资源的nib时，nib加载代码会把图片和声音文件写进内存。

- 不要阻塞主线程：永远不要使主线程承担过多。因为UIKit在主线程上做所有工作，渲染，管理触摸反应，回应输入等都需要在它上面完成，大部分阻碍主进程的情形是你的app在做一些牵涉到读写外部资源的I/O操作，比如存储或者网络。

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
 // 选择一个子线程来执行耗时操作
 dispatch_async(dispatch_get_main_queue(), ^{
 // 返回主线程更新UI
 });
});
```

- 在Image Views中调整图片大小

如果要在UIImageView中显示一个来自bundle的图片，你应保证图片的大小和UIImageView的大小相同。在运行中缩放图片是很耗费资源的.

## 讲讲你用Instrument优化动画性能的经历吧（别问我什么是Instrument）

Apple的instrument为开发者提供了各种template去优化app性能和定位问题。很多公司都在赶feature，并没有充足的时间来做优化，导致不少开发者对instrument不怎么熟悉。但这里面其实涵盖了非常完整的计算机基础理论知识体系，memory，disk，network，thread，cpu，gpu等等，顺藤摸瓜去学习，是一笔巨大的知识财富。动画性能只是其中一个template，重点还是理解上面问题当中CPU GPU如何配合工作的知识。

## facebook启动时间优化

- 1.瘦身请求依赖
- 2.UDP启动请求先行缓存
- 3.队列串行化处理启动响应

## (24) 通知与推送

### 本地通知和远程推送通知对基本概念和用法?

image image image image image image image image image image

本地通知和远程推送通知都可以向不在前台运行的应用发送消息,这种消息既可能是即将发生的事件,也可能是服务器的新数据.不管是本地通知还是远程通知,他们在程序界面的显示效果相同,都可能显示为一段警告信息或应用程序图标上的徽章.

本地通知和远程推送通知的基本目的都是让应用程序能够通知用户某些事情,而且不需要应用程序在前台运行.二者的区别在于本地通知由本应用负责调用,只能从当前设备上的iOS发出,而远程通知由远程服务器上的程序发送到APNS,再由APNS把消息推送至设备上的程序

### iOS允许最近本地通知数最大是多少 (c)

A. 16    B. 32    C. 64    D. 128

### 把程序自己关掉和程序进入后台,远程推送的区别

- 关掉后不执行任何代码,不能处理事件
- 应用程序进入后台状态不久后转入挂起状态。在这种状态下,应用程序不执行任何代码,并有可能在任意时候从内存中删除。只有当用户再次运行此应用,应用才会从挂起状态唤醒,代码得以继续执行
- 或者进入后台时开启多任务状态,保留在内存中,这样就可以执行系统允许的动作
- 远程推送是由远程服务器上的程序发送到APNS,再由APNS把消息推送至设备上的程序,当应用程序收到推送的消息会自动调用特定的方法执行事先写好的代码

### 本地通知和远程推送通知对基本概念和用法?

- 本地通知和远程推送通知都可以向不在前台运行的应用发送消息,这种消息既可能是即将发生的事件,也可能是服务器的新数据.不管是本地通知还是远程通知,他们在程序界面的显示效果相同,都可能显示为一段警告信息或应用程序图标上的徽章.
- 本地通知和远程推送通知的基本目的都是让应用程序能够通知用户某些事情,而且不需要应用程序在前台运行.二者的区别在于本地通知由本应用负责调用,只能从当前设备上的iOS发出,而远程通知由远程服务器上的程序发送到APNS,再由APNS把消息推送至设备上的程序

### Push Notification 是如何工作的?

- 推送通知分为两种,一个是本地推送,一个是远程推送
- 本地推送:不需要联网也可以推送,是开发人员在APP内设定特定的时间来提醒用户干什么

- 远程推送:需要联网,用户的设备会于苹果APNS服务器形成一个长连接,用户设备会发送uuid和Bundle identifier给苹果服务器,苹果服务器会加密生成一个deviceToken给用户设备,然后设备会将deviceToken发送给APP的服务器,服务器会将deviceToken存进他们的数据库.这时候如果有人发送消息给我,服务器端就会去查询我的deviceToken,然后将deviceToken和要发送的信息发送给苹果服务器,苹果服务器通过deviceToken找到我的设备并将消息推送到我的设备上,这里还有个情况是如果APP在线,那么APP服务器会于APP产生一个长连接,这时候APP服务器会直接通过deviceToken将消息推送到设备上

## 为什么 **NotificationCenter** 要 **removeObserver**? 如何实现自动**remove**?

- 如果不移除的话,万一注册通知的类被销毁以后又发了通知,程序会崩溃.因为向野指针发送了消息
- 实现自动remove:通过自释放机制,通过动态属性将remove转移给第三者,解除耦合,达到自动实现remove

## 是否可以把比较耗时的操作放在**NSNotificationCenter**中

- 如果在异步线程发的通知, 那么可以执行比较耗时的操作;
- 如果在主线程发的通知, 那么就不可以执行比较耗时的操作

## ——补充

### UML

- 统一建模语言（UML，Unified Modeling Language）是面向对象软件的标准化建模语言。UML因其简单、统一的特点，而且能表达软件设计中的动态和静态信息，目前已成为可视化建模语言的工业标准。在软件无线电系统的开发过程中，统一建模语言可以在整个设计周期中使用，帮助设计者缩短设计时间，减少改进的成本，使软硬件分割最优。
- 用例图 静态图 行为图 交互图 实现图

如果设计一个交易平台，流程如下：A用户在平台P发现B用户有东西出售，P平台的业务逻辑是需要A先付款到P得中间账户，等到A确认收货后，P向B付款，你能尝试某种表示方式来让程序员，产品设计，美术设计明白这个流程么？

从1-n个五序列的数字中排序的算法有哪些，简单代码实现一个并写出时间复杂度

有一个6克和21克的砝码，怎样称量三次将420克的糖分成270克和150克

一桌子人，每个人额头上都贴有标签，绿色和红色，绿色至少1个，大家都能看到别人头上的标签，看不到自己头上的标签，然后开始闭眼游戏，大家觉得自己头上是绿色标签就拍一下手掌，第一次闭眼没人拍掌，第二次，第三次也同样没有人拍掌，第四次有一个人拍掌了，请问绿色标签有几张

是否使用过coreImage和coreText？如果使用过，说说你的体验

- coreImage是IOS5中新加的一个Objective-c的框架，提供了强大高效的图像处理功能，用来对基于像素的图像进行操作与分析。

写一个单项链表逆序

什么是OpenGL？具体使用

开启一个其他线程来计算 $1+1$  并且把结果用主线程显示在label上更新UI

## ffmpeg框架

### 静态链接库

**init和initWithobject区别（语法）？**

**什么是OOP？**

**为NSString扩展一个方法，方法能判断字符串是否是Url地址（即判断字符串是否以“http://”），放回BOOL值类型**

**写一个iphone程序，有2屏，可以通过滑动切换，第二屏有一个webview，读取本地的html文件，Html文件中会加载一个本地xml文件，获取xml文件中的数据内容并显示。（可选：html中加载的js文件）**

**如何避免json解析出现内存泄露，内存泄露后怎么解决**

**什么情况下会发生内存泄漏和内存溢出？**

**有方法查看当前系统内存使用情况吗？**

- 静态分析：通过静态分析我们可以最初步的了解到代码的一些不规范的地方或者是存在的内存泄漏，这是我们第一步对内存泄漏的检测。当然有一些警告并不是我们关心的可以略过。
- 通过instruments来检查内存泄漏  
这个方法能粗略的定位我们在哪里发生了内存泄漏。方法是完成一个循环操作，如果内存增长为0就证明我们程序在该次循环操作中不存在内存泄漏，如果内存增长不为0那证明有可能存在内存泄漏，当然具体问题需要具体分析。
- 代码测试内存泄漏  
在做这项工作之前我们要注意一下，在dealloc的方法中我们是否已经释放了该对象所拥有的所有对象。观察对象的生成和销毁是否配对。准确的说就是init（创建对象的方法）和dealloc是否会被成对触发（简单说来就是走一次创建对象就有走一次dealloc该对象）。

下面是自己遇到的一些比较隐秘的造成内存泄漏的情况：

- 1.两个对象互相拥有：也就是说对象a里面retain/addSubview了b对象，b对象同时也retain/addSubView了a对象。注意：delegate不要用retain属性，要用assign属性也会导致互相拥有。
- 2.有时候需要用removeFromSuperView来释放：具体说明，也许我的a对象拥有一个b对象，b对象add到了c对象上，而在我们的设计中b对象的生命周期应该和a对象相同；这时候只一句[b release]/self.b = nil是不能把b对象释放掉的（一般情况下release会

使其retainCount- 1,[super dealloc]会再次将所有subView的retainCount- 1,而b并不是a的subView，所有最后的一次- 1没有了）；所以我们需要在之前加上[b removeFromSuperview]。

写一个贪吃蛇的算法

请写一个类似于**NSMutableArray**的类，可以添加，删除，以及如何以最快的速度查找某个元素？

给一个数字，判断从左读和从右读是否是一致的？例如12321和4444

给一个数组[1、3、2、4、8]排序后[1、2、3、4、8]？

怎么用下面的类？北京品科艺科技有限公司

- NSUserDefaults
- NSManagedObjectContext
- NSPredict怎么从xib文件加载成UIView?

OAuth2.0授权的过程，是否用过1.0？

id是编译时还是运行时

蓝牙

CFNetwork基于啥？

逻辑运算跟位运算的区别？

三维动画的旋转的原理？底层怎么实现？点与点用矩阵变换实现的。

NSFileManager/NSFileHandle为什么不用文件加载而用数据库？

UIKit基于什么？

默写二分查找算法？

设计一个忽略大小写比较两个字符串的算法？

怎么过滤一段既有字符串又有数字的，让他只剩数字？



iOS资源群：190892815

iOS逆向资源：（定期分享）

<http://weibo.com/Edstick>

## 支付功能。微信支付 支付宝支付

### 友盟分享 报错

假设某一个生物每B年繁殖一次，一次繁殖N个只，寿命是M年，编写程序起初有X只生物，求T年之后生物的总数是多少？

=有个不相同的数，从中随机取N个数，要求N个数互不相同。

```
// (已经有随机函数int getRand (int min, int max) 表示获取从最小值到最大值的随机整数)
{
 return min + (max - min) *rand() / RAND_MAX;
}
```

### 什么是消息推送？

在iPhone中怎么写入C++程序，详细写下来

远程推送；本地消息和原地推送的区别

苹果怎么实现安全机制

- 设备控制和保护：支持用户从一系列密码设计策略中根据安全需求来进行选择，包括超时设定、密码长度以及密码更新周期等
- 数据保护。256位AES硬件加密算法。远程信息擦除，本地信息擦除
- 安全网络通信，VPN SSL WAP/WAP2认证方式接入wifi
- 安全的iOS平台。运行时保护-沙盒机制，应用之间不能相互访问，系统资源与用户程序隔离。强制前面。安全认证框架  
极光推送原理（第三方框架），怎么用的？遇到什么问题？怎么解决的？  
微信的附近功能怎么实现的？  
怎么实现第三方登陆

### autolayout? sizeclass

### HealthKit是什么？

2014年6月2日召开的年度开发者大会上，苹果发布了一款新的移动应用平台，可以收集和分析用户的健康数据，这是苹果计划为其计算和移动软件推出的一系列新功能的一部分该移动应用平台被命名为“Healthkit”，苹果高管告诉开发者，它可以整合iPhone或iPad上其它健康应用收集的数据，如血压和体重等。

HealthKit框架提供了一个结构，应用可以使用它来分享健康和健身数据。HealthKit管理从不同来源获得的数据，并根据用户的偏好设置，自动将不同来源的所有数据合并起来。应用还可以获取每个来源的原始数据，然后执行自己的数据合并。

HealthKit也可以直接与健康和健身设备一起工作。在iOS8.0中，系统可以自动将兼容的低功耗蓝牙心率仪的数据直接保存在

HealthKit存储中。如果有M7运动协处理器，系统还可以自动导入计步数据。其他的设备和数据源必须要有配套的应用才可以获取数据并保存在HealthKit中。

HealthKit另外提供了一个应用来帮助管理用户的健康数据。健康应用为用户展示HealthKit的数据。用户可以使用健康应用来查看、添加、删除或者管理其全部的健康和健身数据。用户还可以编辑每种数据类型的分享权限。

HealthKit和健康应用在iPad上都不可用。HealthKit框架不能用于应用扩展。

## HomeKit是什么？

HomeKit，是苹果2014年发布的智能家居平台。HomeKit库是用来沟通和控制家庭自动化配件的，这些家庭自动化配件都支持苹果的HomeKit Accessory Protocol。HomeKit应用程序可让用户发现兼容配件并配置它们。用户可以创建一些action来控制智能配件（例如恒温或者光线强弱），对其进行分组，并且可以通过Siri触发。HomeKit对象被存储在用户iOS设备的数据库中，并且通过iCloud还可以同步到其他iOS设备。HomeKit支持远程访问智能配件，并支持多个用户设备和多个用户。HomeKit还对用户的安全和隐私做了处理。

## iCloud是什么包含了哪些技术与服务？

iCloud是苹果公司所提供的云端服务，2011年6月6日苹果公司执行长乔布斯（Steve Jobs）抱病主持全球开发者大会（WWDC），正式发表云端服务iCloud, iOS 5 以及 OS X Lion.其中iCloud的功能是存储内容，包括购买的音乐、应用、电子书等推送到所有设备,iCloud是一系列服务的技术封装,使用者可以免费储存5GB资料。可以备份存放照片、音乐、通讯录、短信、文档等内容，在你需要的时候以无线方式将他们推送到你所有的设备上。自行执行。轻松自如、运作流畅，它就是这么管用。可与亲朋好友共享体验的完整平台。它不仅安全，而且可提供丰富的社交体验，从任何计算机或设备均可随时方便地进行访问。iCloud上所提供的应用程序包括办公生产率、开发工具、媒体和窗口小部件等。随着时间的推移，还将通过易于使用的市场空间和应用程序开发工具箱提供范围更广的应用程序。

服务:应用软件、电子书与备份、Documents in the Cloud、Photo Stream（照片流）、iTunes Match、Mobile Me

CoreData：中多线程中处理大量数据同步时的操作

CoreData:是CoCocoa中处理数据绑定数据的关键特性，提供完整的对象持久化存储方案。如果你使用sqlite3厌倦了敲打sql语句，CoreData正解决了你这烦恼。sqlite3是CoreData处理的数据类型之一，当你将CoreData和sqlite的结合起来使用的话，你将能开发出强大的数据库应用。

## CoreData与多线程操作:

为了在查询数据的时候不让界面停滞，使用多线程是不可避免，一般我们会用thread，串行线程或者并发线程。coredata与多线程交互的时候，每个线程都必须拥有一个manager context对象，一般有两种方式：

- 1.每一个线程使用私有的manager context，共享一个 persistent store coordinator
- 2.每个线程使用私有的manager context和私有的persistent store coordinator

对于这两种方式，我们比较推荐使用第一种方式，因为使用第二种方式的会消耗我们更多的内存，所以推荐使用第一种。注意：CoreData里面还带有一个通知

NSManagedObjectContextDidSaveNotification，主要监NSManagedObjectContext的数据是否改变，并合并数据改变到相应context

## 请解释一下Handoff是什么，并简述它是如何实现iOS、Mac/网页应用互通的开发指南

Handoff英译是用手推开某人，在计算机领域是CDMA术语，表示切换的意思。OSX 10.10 Yosemite新增了一个酷炫的功能“HandOff”，打开这个功能之后，用户可以在Mac上对iPad和iPhone进行操作，比如能够编写iPhone上未完成的邮件，并且可以在Mac上打开iPhone的热点等等，Mac的Hand Off功能只能识别Mac周围的iPhone手机。

Handoff的核心思想就是：用户在一个应用里所做的任何操作都包含着一个activity，一个activity可以和一个特定用户的多台设备关联起来。用行话来说，抽象出这种activity的类叫做NSUserActivity，大部分时间我们都会和这个类打交道。需要一提的是，所有的设备都必须靠近（靠近是指两台设备的蓝牙能够彼此连接），这样Handoff才能正常工作。而且还有两个先决条件得满足：第一个条件是得有一个能正常使用的iCloud账号，而且用户应该在每台准备使用Handoff的设备上登陆这个iCloud账号。事实上，当在不同的设备上切换时，为了保证正在进行的activity不被中断而且被关联到同一个用户，应该尽可能地在所有设备上使用同一个iCloud账号。第二个条件是当两个或两个以上不同的应用想要在同一个用户activity进行Handoff的操作时需要具备的，在这种情况下，所有涉及到的应用必须使用Xcode里相同的团队标识（TeamID）签名。

当编写一个支持Handoff的应用时，需要关注以下三个交互事件：

- 1.为将在另一台设备上继续做的事创建一个新的用户activity。
- 2.当需要时，用新的数据更新已有的用户activity。
- 3.把一个用户activity传递到另一台设备。

需要注意的是Handoff相关的测试只能在真实设备上进行，所以你得有至少两台运行着iOS8.0或以上系统的设备。不管是多台iPhone，多台iPad或者同时拥有iPhone和iPad都可以。

## 1. Runloop和线程的关系

1.一一对应，主线程的runloop已经创建，子线程的必须手动创建

2.runloop在第一次获取时创建，在线程结束时销毁

//在 runloop中有多个运行模式，但是只能选择一种模式运行，mode 中至少要有一个 timer 或者是 source

Mode:

系统默认注册5个 Mode:

kCFRunLoopDefaultMode:App默认 mode，通常主线程在这个 mode 下运行

UITrackingRunLoopMode:界面跟踪 mode，用于 ScrollView 追踪触摸滑动，保证滑动时不受其他 mode 影响

kCFRunLoopCommonModes:占位用的 mode，不是一个真正的 mode

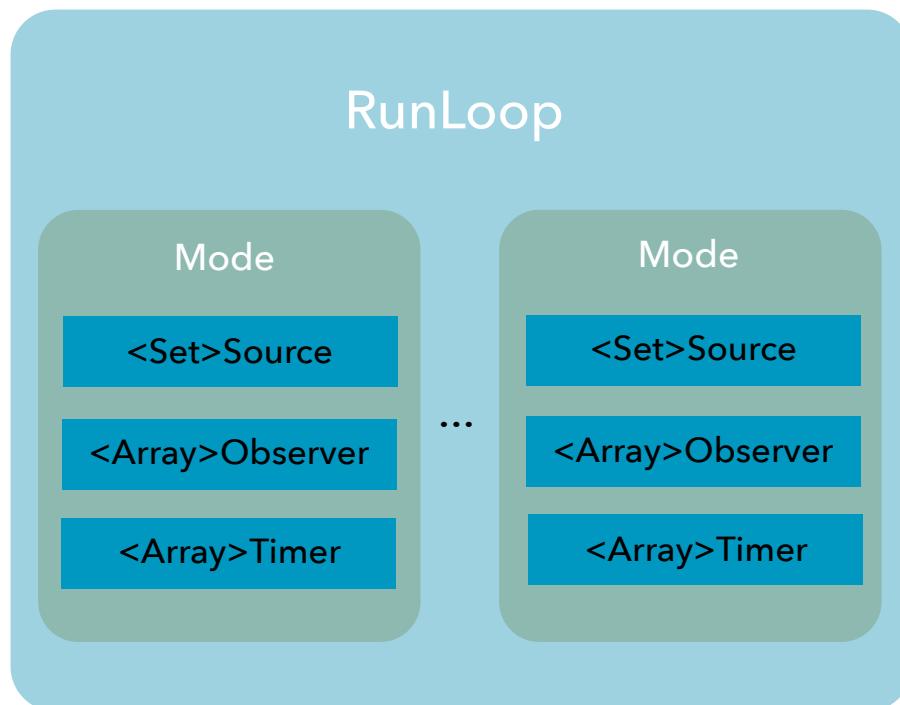
NSRunLoopCommonModes 相当于 NSDefaultRunLoopMode + UITrackingRunLoopMode

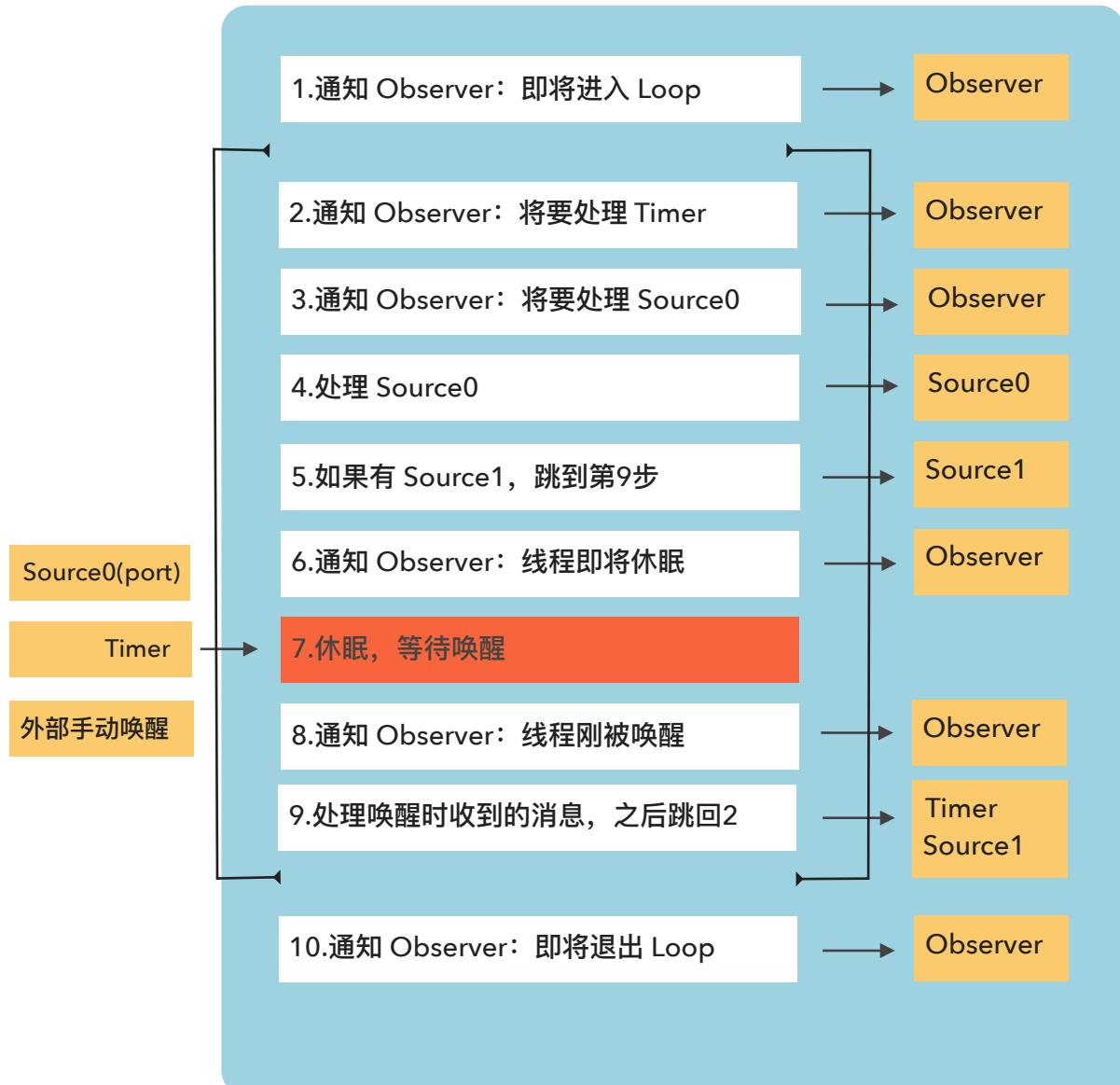
UIInitializationRunLoopMode:刚启动 App 时进入的第一个 mode，启动完成之后不再使用

GSEventReceiveRunLoopMode:接受系统事件的内部 mode，通常用不到

runloop 学习链接:

<http://blog.csdn.net/hherima/article/details/51746125>





## 2. 自动释放池什么时候释放？

//第一次创建：启动 runloop时候  
//最后一次销毁：runloop 退出的时候  
//其他时候的创建和销毁：当 runloop 即将睡眠时销毁之前的释放池，重新创建一个新的

## 3. 什么情况下使用 weak 关键字，和 assign 的区别？

1、ARC 中，有可能出现循环引用的地方使用，比如：delegate 属性

2、自定义 IBOutlet 控件属性一般也是使用 weak

区别：weak 表明一种非持有关系，必须用于 OC 对象；assign 用于基本数据类型

## 4. 怎么用 copy 关键字？

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；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、block也使用 copy

## 5. @property (copy) NSMutableArray \*array; 这写法会出什么问题？

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；

## 6. 如何让自己的类用 copy 修饰符？即让自己写的对象具备拷贝功能

具体步骤：

1、需声明该类遵从 NSCopying 或 NSMutableCopying 协议

2、实现 NSCopying 协议。该协议只有一个方法：

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

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

本质：@property = ivar + getter + setter; (实例变量+getter方法+setter方法)

在编译期自动生成getter、setter，还自动向类中添加适当类型的实例变量，也可以用 @synthesize 语法来指定实例变量的名字

## 8.多线程

1、NSThread线程的生命周期：线程任务执行完毕之后被释放

```
24 - (void)createChildThread1 {
25 //1. 创建线程
26 NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@selector
27 (run:) object:@"abc"];
28 //2. 启动线程
29 [thread start];
30 }
31
32 - (void)createChildThread2 {
33 [NSThread detachNewThreadSelector:@selector(run:) toTarget:self
34 withObject:@"分离子线程"];
35 }
36
37 - (void)createChildThread3 {
38 [self performSelectorInBackground:@selector(run:) withObject:@"开启后台线程"];
39 }
```

图3-1：NSThread开启子线程的3种方式

2、线程安全（加互斥锁）：

格式：@synchronized (self){ //需要锁定的代码 }

注意：1.锁必须全局唯一；2.加锁的位置；3.加锁的前提条件；4.加锁结果：线程同步

优点：防止多线程抢夺资源造成的数据安全问题

缺点：耗费 CPU 性能

使用前提：多线程使用同一块资源

线程同步：多条线程在同一条线上按顺序的执行任务

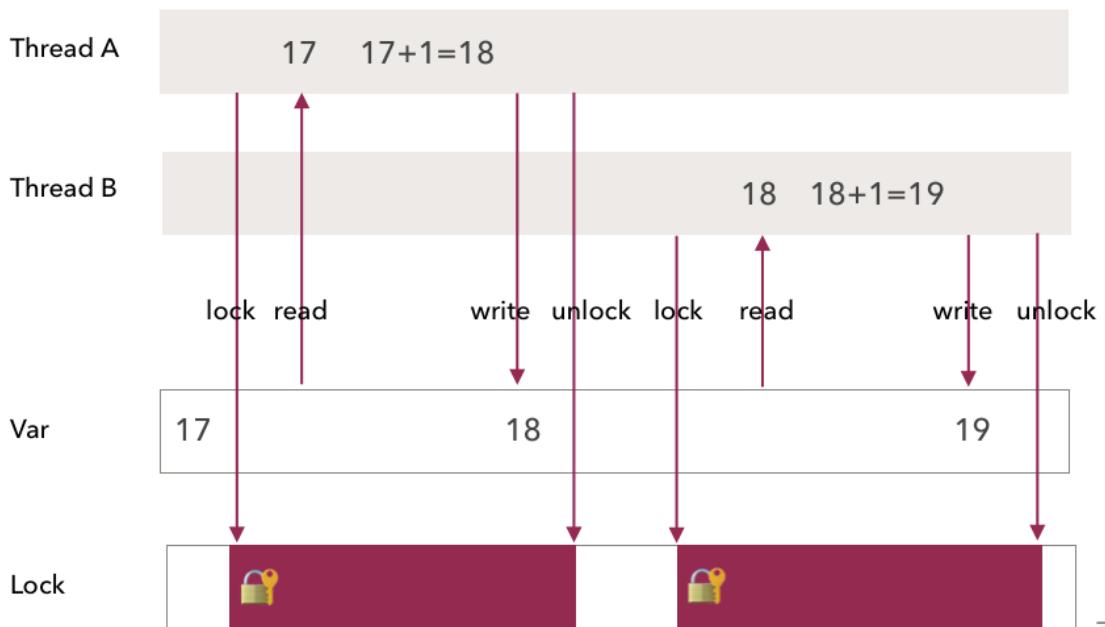


图3-2：互斥锁(官方文档解释)

上图3-2的理解：线程A(Thread A) 和 线程B(Thread B) 同时访问资源变量 Var，为了防止抢夺资源，Thread A 在读取资源变量 Var 之前先加一把锁，然后读取 Var 的数据并在 Thread A 中完成对数据的操作( $17+1=18$ )，然后把数据写入 Var 中，最后开锁 unlock。在 Thread A 对 Var 操作的过程中，Thread B 是无权访问 Var 的，只有 Thread A unlock 之后，Thread B 才能访问资源变量 Var。

**atomic**: 原子属性，为 setter 方法加锁（默认就是 atomic），线程安全，耗资源

**nonatomic**: 非原子属性，不会为 setter 方法加锁，非线程安全

### 3、线程间通信

<1>.NSThread 实现

```
- (void)performSelectorOnMainThread:(SEL)aSelector withObject:(nullable id)arg
waitUntilDone:(BOOL)wait;
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr withObject:(nullable
id)arg waitUntilDone:(BOOL)wait;
```

<2>.GCD 实现

子线程和主线程相互切换

<3>.NSOperationQueue 实现

### 4、GCD（任务、队列）

<1>同步和异步的区别

同步：只能在当前线程中执行任务，不能开启新线程

异步：可在新的线程中执行任务，能开启新线程

<2>队列

并发队列：可多个任务并发（同时）执行（会开启多个线程同时执行任务）；只在异步函数（`dispatch_async`）下有效

串行队列：一个任务执行完毕后，再执行下一个任务

主队列：主队列中的任务都在主线程中执行

```
39 //异步函数+并发队列：会开启多条新线程，队列中的任务是并发执行
40 - (void)asyncConcurrent {
41 //1.创建队列
42 dispatch_queue_t queue = dispatch_queue_create("com.iloveu.download",
43 DISPATCH_QUEUE_CONCURRENT);
44 //2.封装操作---添加任务到队列中
45 dispatch_async(queue, ^{
46 NSLog(@"download1---%@", [NSThread currentThread]);
47 });
48 dispatch_async(queue, ^{
49 NSLog(@"download2---%@", [NSThread currentThread]);
50 });
51 }
52 //异步函数+串行队列：会开线程，开一条线程，队列中的任务是串行执行
53 - (void)asyncSerial { ...}
54
55 //同步函数+并发队列：不会开线程，任务是串行执行
56 - (void)syncConcurrent { ...}
57
58 //同步函数+串行队列：不会开线程，任务是串行执行
59 - (void)syncSerial {
60 dispatch_queue_t queue = dispatch_queue_create("com.iloveu.download",
61 DISPATCH_QUEUE_SERIAL);
62 dispatch_sync(queue, ^{
63 NSLog(@"download1---%@", [NSThread currentThread]);
64 });
65 dispatch_sync(queue, ^{
66 NSLog(@"download2---%@", [NSThread currentThread]);
67 });
68 }
```

```
87 //异步函数+主队列：不会开线程，所有任务都在主线程中执行
88 - (void)asyncMain {
89 dispatch_queue_t queue = dispatch_get_main_queue();
90 dispatch_async(queue, ^{
91 NSLog(@"download1---%@", [NSThread currentThread]);
92 });
93 dispatch_async(queue, ^{
94 NSLog(@"download2---%@", [NSThread currentThread]);
95 });
96 }
97
98 //同步函数+主队列：死锁
99 - (void)syncMain {
100 dispatch_queue_t queue = dispatch_get_main_queue();
101
102 NSLog(@"start---");
103
104 //同步函数：立刻执行，当前任务未执行完，后续任务也不会执行
105 dispatch_sync(queue, ^{
106 NSLog(@"download1---%@", [NSThread currentThread]);
107 });
108 dispatch_sync(queue, ^{
109 NSLog(@"download2---%@", [NSThread currentThread]);
110 });
111
112 NSLog(@"end---");
113 }
```

主队列特点：如果主队列发现当前主线程有任务在执行，那么主队列会暂停调用队列中的任务，直到主线程空闲为止。

上图：方法-syncMain如果在主线程中调用，则会发生死锁，即102行之后的不会打印；如果在子线程中调用，则不会发生死锁，NSLog 都能打印。

RunLoop 的应用：

- NSTimer 在子线程开启一个定时器；控制定时器在特定模式下执行
- imageView 的显示
- performSelector
- 常驻线程（让一个子线程不进入消亡状态，等待其他线程发来消息，处理其他事件）
- 自动释放池

## 9. @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，使用属性的目的，是希望遵守该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 方法声明，如果真的需要给 category 增加属性的实现，需要借助于运行时的两个函数：

`objc_setAssociatedObject`

`objc_getAssociatedObject`

## 10. @property中有哪些属性关键字?

- 1、原子性 — nonatomic 特质
- 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>

## 11. 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，runtime 内部已经帮我们实现了

## 12. @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。

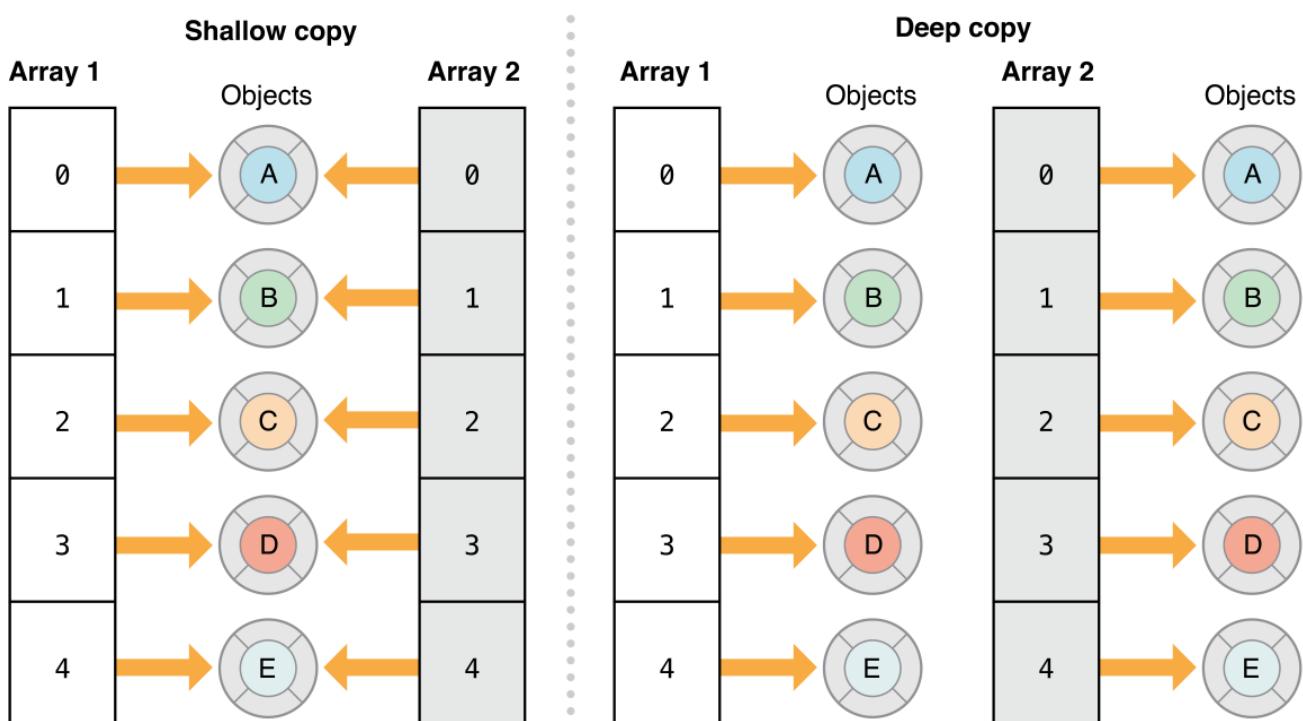
### 13. ARC下，不显式指定任何属性关键字时，默认的关键字都有哪些？

- 1、基本数据类型：atomic、readwrite、assign
- 2、普通 OC 对象：atomic、readwrite、strong

### 14. 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、如果使用 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。

浅复制、深复制->浅复制就是指针拷贝、深复制就是内容拷贝，如图：左浅复制、右深复制



不管是集合类对象，还是非集合类对象，接收到copy和mutableCopy消息时，都遵循以下准则：

- copy返回immutable对象；所以，如果对copy返回值使用mutable对象接口就会crash；
- mutableCopy返回mutable对象；

非集合类对象：

在非集合类对象中：对 immutable(不可变)对象进行 copy 操作，是指针复制， mutableCopy 操作时内容复制；对 mutable(可变)对象进行 copy 和 mutableCopy 都是内容复制。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

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

## 15. @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？

@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。

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

- 1、同时重写了 setter 和 getter 时，系统就不会生成 ivar，使用 @synthesize foo = \_foo；关联 @property 与 ivar
- 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、重载的属性，当在子类中重载了父类中的属性，必须使用 @synthesize 来手动合成 ivar

## 17. 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一只是在运行时不会有任何作用。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)`，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。

## 18. 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？

[`obj foo`]；在objc动态编译时，每个方法在运行时会被动态转为消息发送，即为：  
`objc_msgSend(obj, @selector(foo));`

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

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。

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。

## 20. 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:

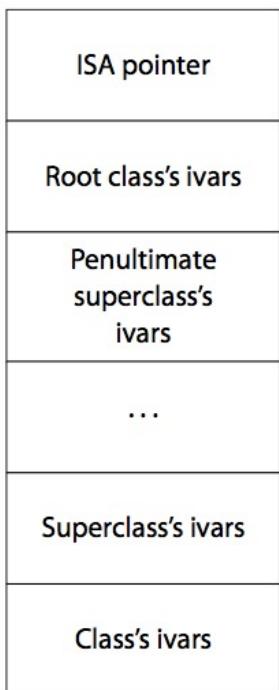
1 对象方法列表 (对象能够接收的消息列表, 保存在它所对应的类对象中)

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:

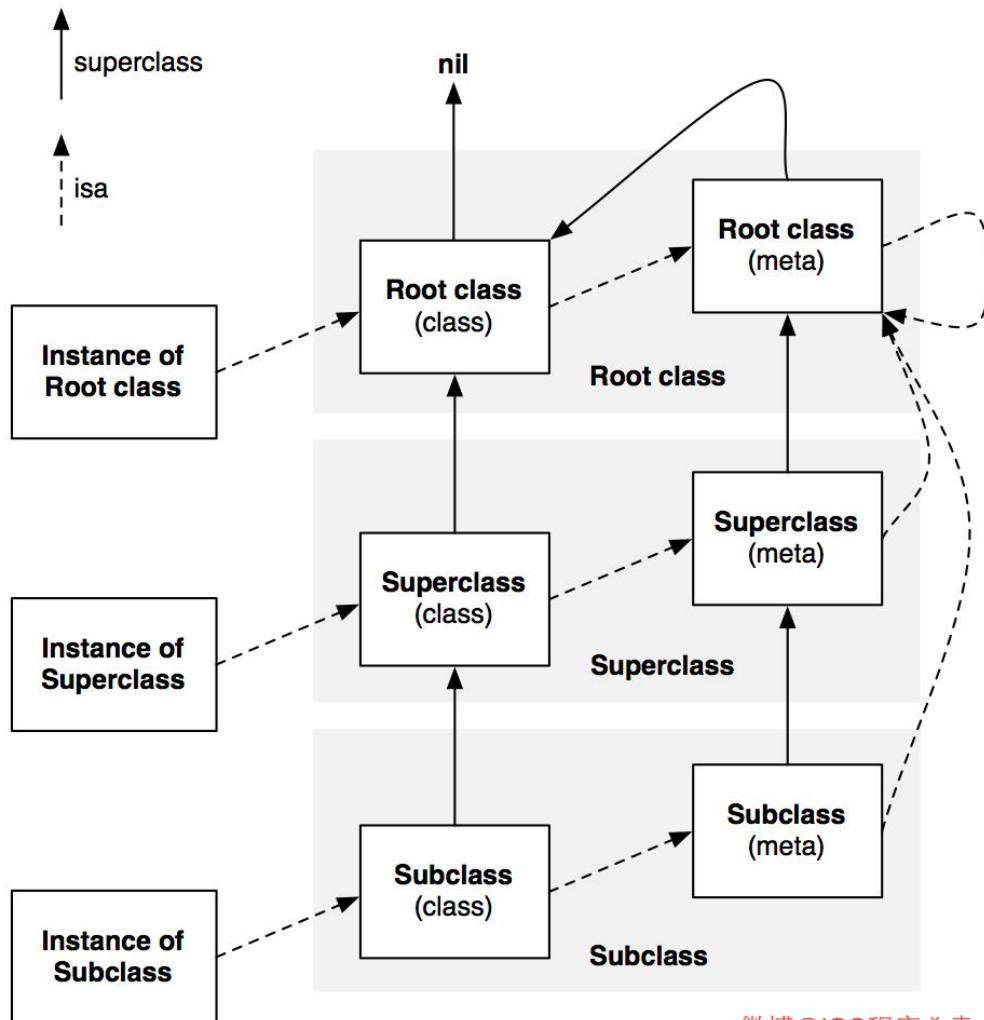


翻译过来就是:



- 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：



## 21. 一个objc对象的isa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

## 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名称、方法实现、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.

### 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。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。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。（看不懂，移步原文有详细介绍）

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

无论在MRC下还是ARC下均不需要。

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

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` 调用 `-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()`

## 25.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)方法

## 26. `_objc_msgForward`函数是什么，直接调用它将会发生什么？

`_objc_msgForward`是一个函数指针（和 `IMP` 的类型一样），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。

`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`。

`_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 `NSInvocation` 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 `nil`。
- 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

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”，

如果用不好会直接导致程序Crash，但是如果用得好，做很多事情，比如JSPatch、RAC(ReactiveCocoa)

## 27. 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。

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。

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）

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。

## 28. 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。

## 29. runloop和线程有什么关系？

实际上，run loop和线程是紧密相连的，可以说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

```
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。  
`NSRunLoop *runloop = [NSRunLoop currentRunLoop];`

## 30. 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)

### 31. 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：只有在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。

### 32. 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：

```
1 function loop() {
2 initialize();
3 do {
4 var msg = get_next_msg();
5 process_msg(msg);
6 } while (msg != nil);
7 }
8 }
```

或使用伪代码来展示下：

```
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;
}
```

### 33. 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。

### 34. ARC通过什么方式帮助开发者管理内存？

编译时根据代码上下文，插入 `retain/release`

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】

### 35. BAD\_ACCESS在什么情况下出现？

访问了野指针，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. 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，就会发生循环引用。

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置空 `xxx = nil`。

### 37. 在block内如何修改block外部变量？

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。block 内部的变量会被 copy 到堆区，进而可以在block内部也可以修改外部变量的值。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。

## 38. 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.

```
1objc_autoreleasepoolPush
2objc_autoreleasepoolPop
3objc_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:

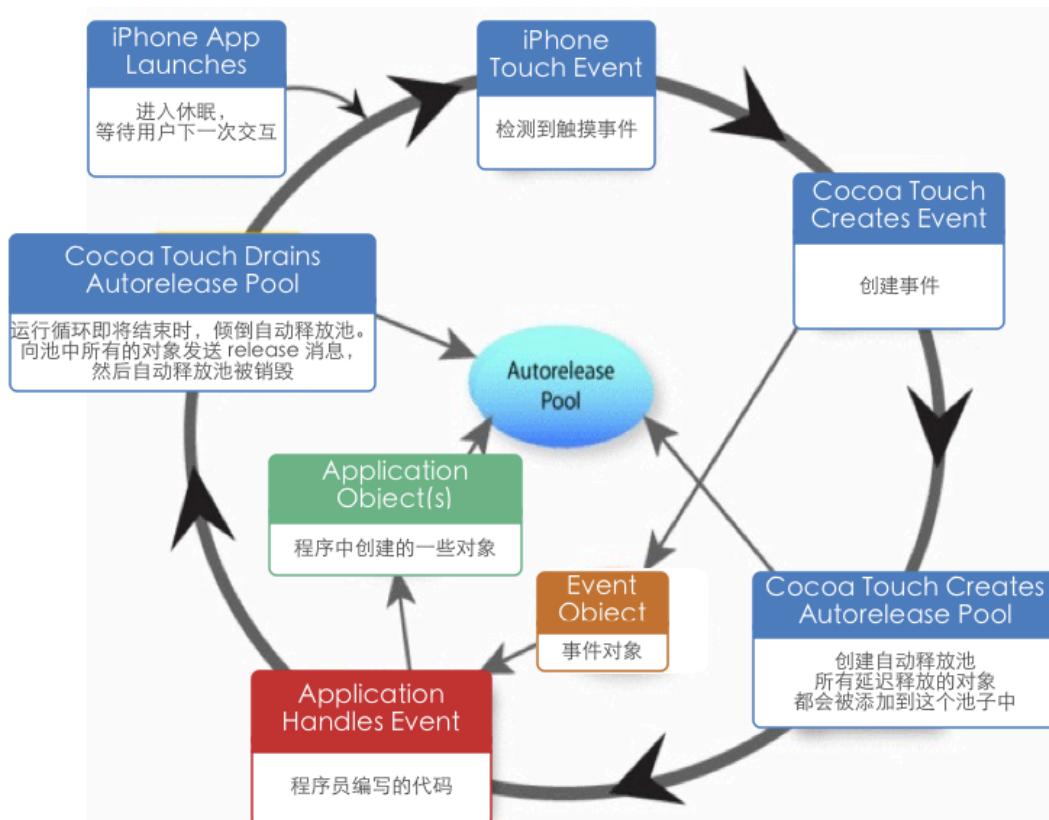
## 39. 不手动指定autoreleasepool的前提下, 一个autorelease对象在什么时刻释放? (比如在一个vc的viewDidLoad中创建)

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:



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。

**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。

## 40. 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？

UIView的block版本写动画时不需要考虑，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，比如这些不用考虑：

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

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

[[NSNotificationCenter defaultCenter] addObserverForName:@"someNotification"
 object:nil
 queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification * notification) {
 self.someProperty = xyz;
}];
```

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:

```
_weak __typeof__(self) weakSelf = self;
dispatch_group_async(_operationsGroup, _operationsQueue, ^
{
 __typeof__(self) strongSelf = weakSelf;
 [strongSelf doSomething];
 [strongSelf doSomethingElse];
});
```

类似的:

```
_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。

## 41.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

## 42.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。

```
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
});
```

实际代码如下图：

```
299 dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
300 dispatch_group_t group = dispatch_group_create();
301 //下图片1,放线程组中
302 dispatch_group_async(group, queue, ^{
303 NSURL *url = [NSURL URLWithString:@""];
304 NSData *imgData1 = [NSData dataWithContentsOfURL:url];
305 self.img1 = [UIImage imageWithData:imgData1];
306 });
307
308 //下载图片2
309 dispatch_group_async(group, queue, ^{
310 NSURL *url = [NSURL URLWithString:@""];
311 NSData *imgData2 = [NSData dataWithContentsOfURL:url];
312 self.img2 = [UIImage imageWithData:imgData2];
313 });
314
315 //合并图片
316 dispatch_group_notify(group, queue, ^{
317 NSLog(@"%@", [NSThread currentThread]);
318 UIGraphicsBeginImageContext(CGSizeMake(200, 200));
319 [self.img1 drawInRect:CGRectMake(0, 0, 200, 100)];
320 self.img1 = nil;
321 [self.img2 drawInRect:CGRectMake(0, 100, 200, 100)];
322 self.img2 = nil;
323 UIImage *img = UIGraphicsGetImageFromCurrentImageContext();
324 UIGraphicsEndImageContext();
325
326 dispatch_async(dispatch_get_main_queue(), ^{
327 NSLog(@"%@", [NSThread currentThread]);
328 //self.imageView.image = img;
329 });
330 });

```

### 43. `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。

(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。)

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

dispatch\_get\_current\_queue容易造成死锁

#### 45. 以下代码运行结果如何?

```
10 - (void)viewDidLoad
11 {
12 [super viewDidLoad];
13 NSLog(@"1");
14 dispatch_sync(dispatch_get_main_queue(), ^{
15 NSLog(@"2");
16 });
17 NSLog(@"3");
18 }
```

只输出：1。发生主线程锁死。

#### 46. 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？

```
// 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：

```
// 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;
```

## 48. 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。

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 `didChangeValueForKey:`。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，`observeValueForKeyPath: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。

“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。

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。

```
//@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。

## 49. 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。

## 50. 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`

## 51. KVC和KVO的keyPath一定是属性么?

KVC 支持实例变量；

如果将一个对象设定成属性，这个属性是自动支持KVO的；如果这个对象是一个实例变量，那么，这个KVO是需要我们自己来实现的。手动支持

手动设定实例变量的KVO实现监听，如下图：

```
// 手动设定KVO
- (void)setAge:(NSString *)age
{
 [self willChangeValueForKey:@"age"];
 _age = age;
 [self didChangeValueForKey:@"age"];
}
- (NSString *)age
{
 return _age;
}
+ (BOOL)automaticallyNotifiesObserversForKey:
 (NSString *)key
{
 // 如果监测到键值为age，则指定为非自动监听对象
 if ([key isEqualToString:@"age"])
 {
 return NO;
 }
 return [super
 automaticallyNotifiesObserversForKey:key];
}
@end
```

```
#import <Foundation/Foundation.h>
@interface Student : NSObject
{
 NSString *_age;
}
- (void)setAge:(NSString *)age;
- (NSString *)age;
@property (nonatomic, strong) NSString *age;
@end
```

## 52. 如何关闭默认的KVO的默认实现，并进入自定义的KVO实现？

利用 Runtime 动态创建类、实现 KVO

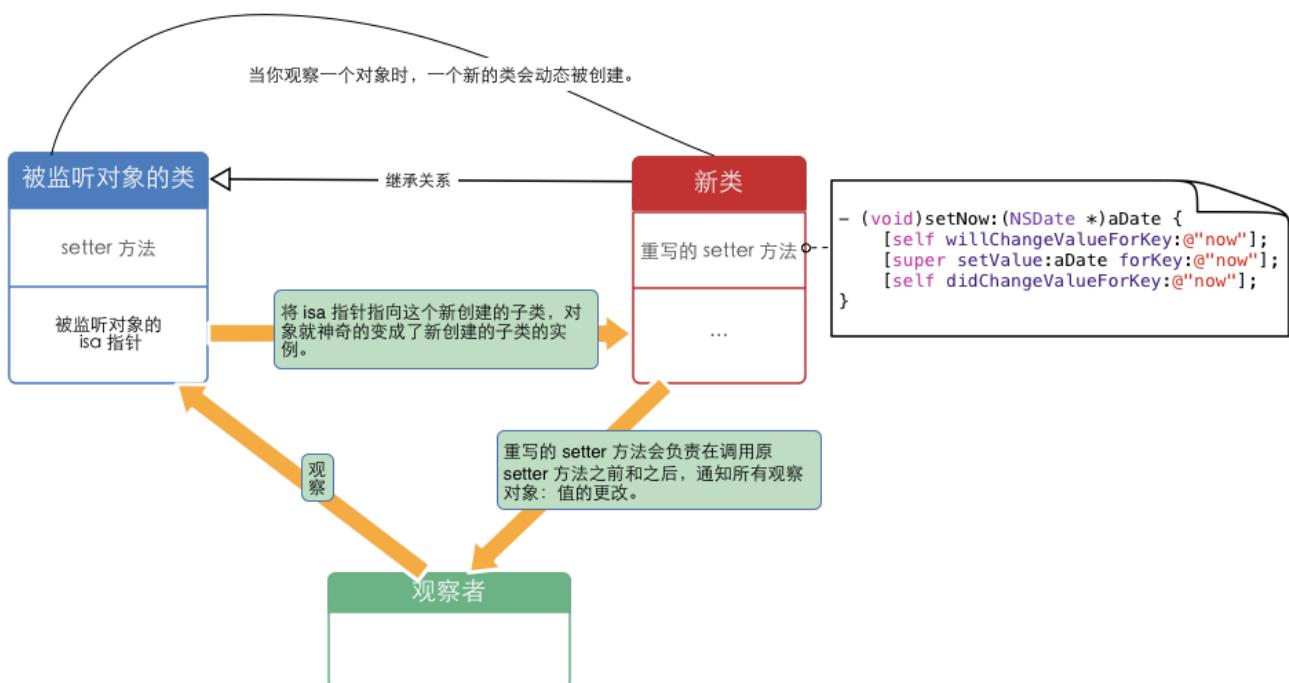
链接: <http://tech.glowing.com/cn/implement-kvo/>

## 53. apple用什么方式实现对一个对象的KVO?

KVO 实现:

在使用KVC命名约定时，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。

实现原理如下图：



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

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 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：

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

```
[self willChangeValueForKey:@"now"];
[super setValue:aDate forKey:@"now"];
[self didChangeValueForKey:@"now"];
}
```

## 54. IBOutlet连出来的视图属性为什么可以被设置成weak?

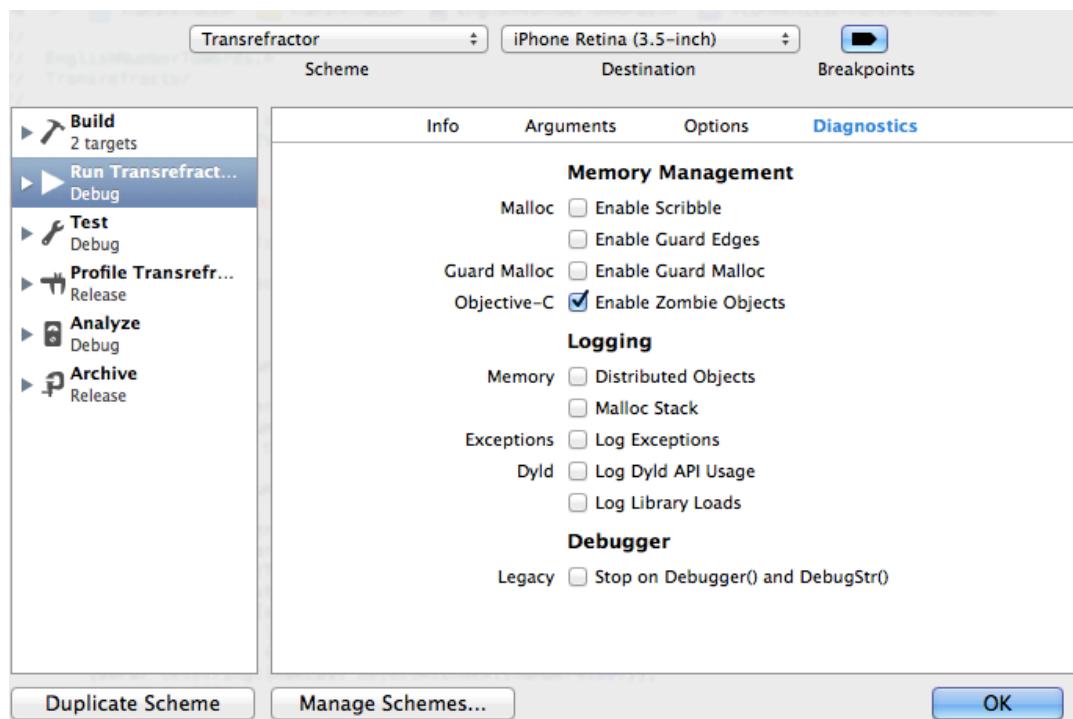
因为有外链那么视图在xib或者storyboard中肯定存在，视图已经对它有一个强引用了。不过这个回答漏了个重要知识，使用storyboard (xib不行) 创建的vc，会有一个叫 \_topLevelObjectsToKeepAliveFromStoryboard的私有数组强引用所有top level的对象，所以这时即便outlet声明成weak也没关系。

## 55. IB中User Defined Runtime Attributes如何使用？（应该知道在哪儿吧）

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。

## 56. 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



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Build Settings 中勾选 Enable Address Sanitizer

## 57. 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

-----华丽而优美的分割线-----

## 58. iOS 容易引起“循环引用”的几种场景

最简单的理解：对象 A 持有对象 B，对象 B 又持有对象 A，就会造成循环引用

### 1、parent-child相互持有、委托模式

### 2、block：

隐式（间接）循环引用。ObjectA 持有ObjectB，ObjectB持有block。那么在block中调用 ObjectA的self会造成间接循环引用；

即使在你的block代码中没有显式地出现”self”，也会出现循环引用！只要你在block里用到了 self所拥有的东西！

显式循环引用，编译器会报警告，在block引用self的时候最好使用weak-strong dance技术。

### 3、NSTimer：

NSTimer会持有对象，所以：在删除对象之前，需要[timer invalidate]。

```
1 @interface FtKeepAlive : NSObject
2 {
3 NSTimer* _keepAliveTimer; // 发送心跳timer
4 }
5 //实现文件
6
7 _keepAliveTimer = [NSTimer scheduledTimerWithTimeInterval:_expired target:
8 self selector:@selector(keepLiveStart) userInfo:nil repeats:YES];
```

如上代码，类持有\_keepAliveTimer，\_keepAliveTimer又持有self，造成循环引用。

### 4、比如把self加入array中。也会造成循环引用

### 5、使用类别添加属性

有一个类A，给A动态添加属性p。如果p中再引用类A，容易造成循环引用

## 59. 类方法load和initialize的区别

iOS会在运行期提前并且自动调用这两个方法。

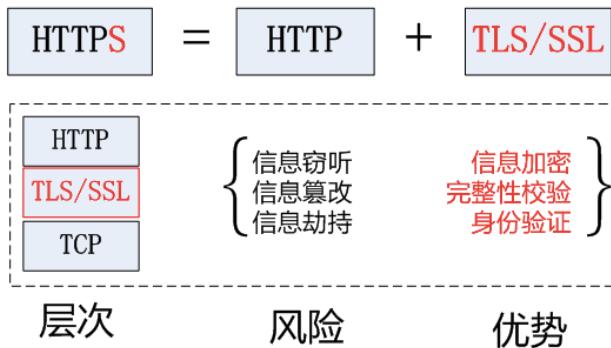
区别：load是只要类所在文件被引用就会被调用，而initialize是在类或者其子类的第一个方法被调用前调用（runtime对+(void)load的调用并不视为类的第一个方法）。所以如果类没有被引用进项目，就不会有load调用；但即使类文件被引用，但是没有使用，那么initialize也不会被调用。相同点在于：方法只会被调用一次。（其实这是相对runtime来说的）。

方法调用的顺序：父类(Superclass)的方法优先于子类(Subclass)的方法，类中的方法优先于类别(Category)中的方法。

|                  | + (void)load  | + (void)initialize |
|------------------|---------------|--------------------|
| 执行时机             | 在程序运行后立即执行    | 在类的方法第一次被调时执行      |
| 若自身未定义，是否沿用父类的方法 | 否             | 是                  |
| 类别中的定义           | 全都执行，但后于类中的方法 | 覆盖类中的方法，只执行一个      |

当类对象被引入项目时，`runtime` 会向每一个类对象发送 `load` 消息。`load` 方法还是非常的神奇的，因为它会在每一个类甚至分类被引入时仅调用一次，调用的顺序是父类优先于子类，子类优先于分类。而且 `load` 方法不会被类自动继承，每一个类中的 `load` 方法都不需要像 `viewDidLoad` 方法一样调用父类的方法。

## 60. HTTP 和 HTTPS



HTTP是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准(TCP)，用于从WWW服务器传输超文本到本地浏览器的传输协议。HTTP是采用明文形式进行数据传输，极易被不法份子窃取和篡改。

HTTPS是在HTTP上建立SSL加密层，并对传输数据进行加密，是HTTP协议的安全版。HTTPS主要作用是：

- (1) 对数据进行加密，并建立一个信息安全通道，来保证传输过程中的数据安全；
- (2) 对网站服务器进行真实身份认证。

区别：

- 1、HTTPS是加密传输协议，HTTP是明文传输协议；
- 2、HTTPS需要用到SSL证书，而HTTP不用；
- 3、HTTPS比HTTP更加安全，对搜索引擎更友好
- 4、HTTPS标准端口443，HTTP标准端口80；
- 5、HTTPS基于传输层，HTTP基于应用层；
- 6、HTTPS在浏览器显示绿色安全锁，HTTP没有显示；

如下图 HTTP 和 HTTPS 的网络分层：



## 61. 常见的Exception Type

### 1> EXC\_BAD\_ACCESS

此类型的Exception是我们最长碰到的Crash，通常用于访问了不改访问的内存导致。一般EXC\_BAD\_ACCESS后面的"()"还会带有补充信息。

**SIGSEGV:** 通常由于重复释放对象导致，这种类型在切换了ARC以后应该已经很少见到了

**SIGABRT:** 收到Abort信号退出，通常Foundation库中的容器为了保护状态正常会做一些检测，例如插入nil到数组中等会遇到此类错误

**SEGV:** (Segmentation Violation)，代表无效内存地址，比如空指针，未初始化指针，栈溢出等

**SIGBUS:** 总线错误，与 SIGSEGV 不同的是，SIGSEGV 访问的是无效地址，而 SIGBUS 访问的是有效地址，但总线访问异常(如地址对齐问题)

**SIGILL:** 尝试执行非法的指令，可能不被识别或者没有权限

### 2> EXC\_BAD\_INSTRUCTION

此类异常通常由于线程执行非法指令导致

### 3> EXC\_ARITHMETIC

除零错误会抛出此类异常

处理此类异常方式：符号化Crash日志。比较常用的有Crashlytics、Flurry、友盟等。

## 62. Category 和 Extension

Category的方法不一定非要在@implementation中实现，也可以在其他位置实现，但是当调用Category的方法时，依据继承树没有找到该方法的实现，程序则会崩溃。

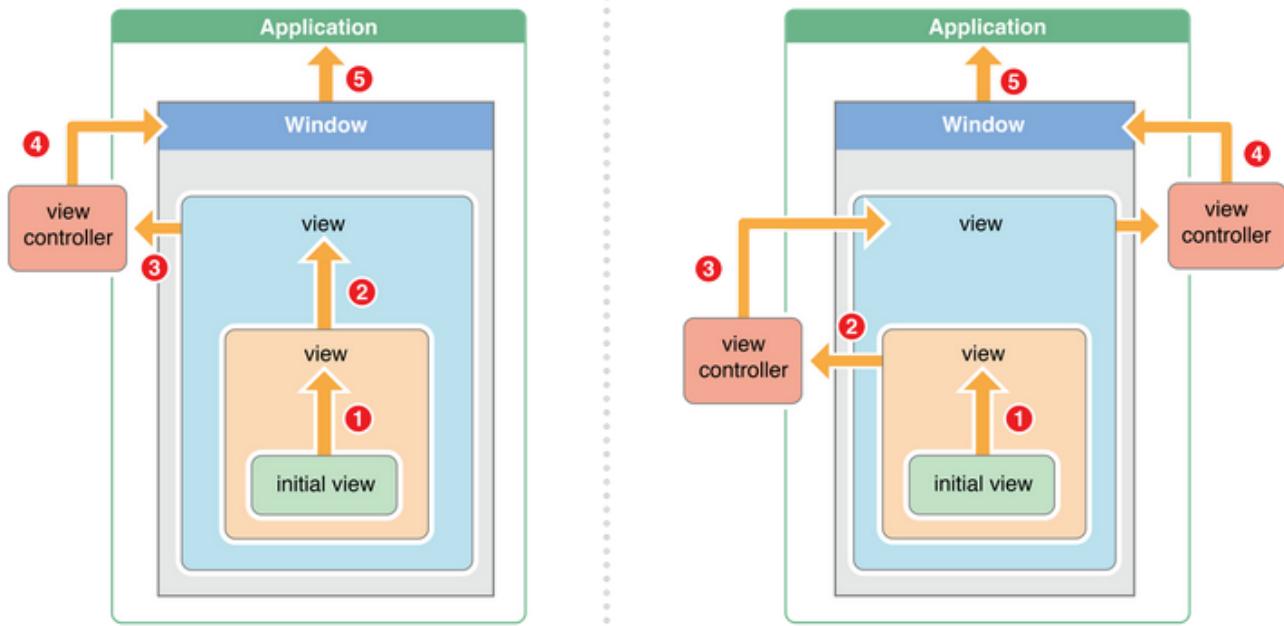
Category理论上不能添加变量，但是可以使用 @dynamic 来弥补这种不足。

**Category为原始类添加方法，必须要小心不要去重写已经存在的方法**

Extension中的方法必须在@implementation中实现，否则编译会报错。

**Extension管理类的私有方法**

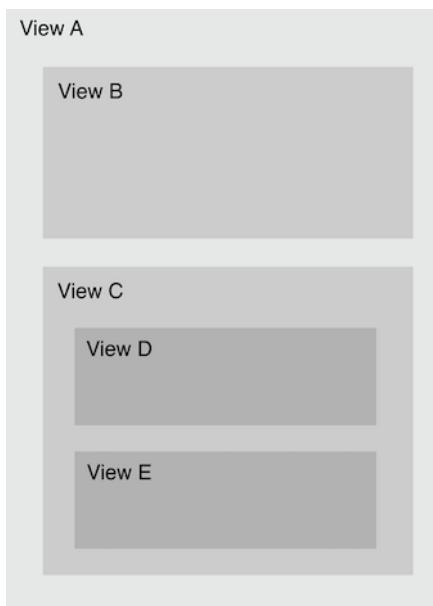
## 63. 响应者链 (responder chain)



如上图，响应者链有以下特点：

- 响应者链通常是由 `initial view` 开始；
- `UIView` 的 `nextResponder` 它的 `superview`; 如果 `UIView` 已经是其所在的 `UIViewController` 的 `top view`, 那么 `UIView` 的 `nextResponder` 就是 `UIViewController`;
- `UIViewController` 如果有 `Super ViewController`, 那么它的 `nextResponder` 为其 `Super ViewController` 最表层的 `View`; 如果没有, 那么它的 `nextResponder` 就是 `UIWindow`;
- `UIWindow` 的 `contentView` 指向 `UIApplication`, 将其作为 `nextResponder`;
- `UIApplication` 是一个响应者链的终点, 它的 `nextResponder` 指向 `nil`, 整个 responder chain 结束。

Hit-Test View 与 Hit-Testing



假设用户触摸了上图的 View E 区域，那么 iOS 将会按下面的顺序反复检测 subview 来寻找 Hit-Test View

- 1 触摸区域在视图 A 内，所以检测视图 A 的 subview B 和 C；
  - 2 触摸区域不在视图 B 内，但是在视图 C 内，所以检查视图 C 的 subview D 和 E；
  - 3 触摸区域不在视图 D 内，在视图 E 中；
- 视图 E 在整个视图体系中是 lowest view，所以视图 E 就是 Hit-Test View。

事件的链有两条：事件的响应链；Hit-Testing 时事件的传递链。

- **响应链：**由离用户最近的view向系统传递。 initial view -> super view -> ....-> view controller -> window -> Application -> AppDelegate
- **Hit-Testing 链：**由系统向离用户最近的view传递。 UIKit -> active app's event queue -> window -> root view ->.....->lowest view

## 64. UITableView 的优化

### 1 重用 cell

### 2 缓存行高

- 若 cell 定高，删除代理中的方法 tableView:heightForRowAtIndexPath: 方法，设置 self.tableView.rowHeight = 88；
- 若 cell 不定高，不要设置estimatedHeightForRow(因为 estimatedHeightForRow 不能和 heightForRow 里面的 layoutIfNeeded 同时存在，这两者同时存在会出现“窜动”的bug)，解决方法是在请求到数据的时候提前计算好行高，用个字典缓存好高度；

### 3 加载网络数据，下载图片，使用异步加载，并缓存，从网络捞回来图片后先根据需要显示的图片大小切成合适大小的图，每次只显示处理过大小的图片，当查看大图时在显示大图，如果服务器能直接返回预处理好的小图和图片的大小更好。图片数量多时，必要的时候要准备好预览图和高清图，需要时再加载高清图，图片的‘懒加载’方法，即延迟加载，当滚动速度很快时避免频繁请求服务器数据。

### 4 使用局部刷新

- 尽量不要使用 reloadData，刷新某一分组、某一行使用对应的方法做局部刷新

### 5 渲染，尽量少用或不用透明图层

- 将cell的opaque值设为Yes，背景色和子 view 不要使用clearColor，尽量不要使用阴影渐变、透明度也不要设置为0

### 6 少用addSubview给Cell动态添加子View，初始化时直接设置好，通过hidden控制显示隐藏，布局也在初始化时直接布局好，避免 cell 的重新布局

### 7 如果 cell 内显示的内容来自 web，使用异步加载，缓存结果请求。

- 8 按需加载cell，cell滚动很快时，只加载范围内的cell，如果目标行与当前行相差超过指定行数，只在目标滚动范围的前后制定n行加载。滚动很快时，只加载目标范围内的cell，这样按需加载，极大地提高了流畅性。方法如下：

```
//按需加载 - 如果目标行与当前行相差超过指定行数，只在目标滚动范围的前后指定3行加载。
- (void)scrollViewWillEndDragging:(UIScrollView *)scrollView withVelocity:(CGPoint)velocity
 targetContentOffset:(inout CGPoint *)targetContentOffset{
 NSIndexPath *ip = [self indexPathForRowAtPoint:CGPointMake(0, targetContentOffset->y)];
 NSIndexPath *cip = [[self indexPathsForVisibleRows] firstObject];
 NSInteger skipCount = 8;
 if (labs(cip.row-ip.row) > skipCount) {
 NSArray *temp = [self indexPathsForRowsInRect:CGRectMake(0, targetContentOffset->y, self.width, self.height)];
 NSMutableArray *arr = [NSMutableArray arrayWithArray:temp];
 if (velocity.y<0) {
 NSIndexPath *indexPath = [temp lastObject];
 if (indexPath.row+33) {
 [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-3 inSection:0]];
 [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-2 inSection:0]];
 [arr addObject:[NSIndexPath indexPathForRow:indexPath.row-1 inSection:0]];
 }
 }
 [needLoadArr addObjectFromArray:arr];
 }
}
```

记得在tableView:cellForRowAtIndexPath:方法中加入判断：

```
if (needLoadArr.count>0&&[needLoadArr indexOfObject:indexPath]==NSNotFound) {
 [cell clear];
 return;
}
```

滚动很快时，只加载目标范围内的Cell，这样按需加载，极大的提高流畅度。

- 9 遇到复杂界面，像朋友圈涉及图文混排的，需要异步绘制，继承UITableViewCell，给自定义的Cell添加draw方法，在方法中利用 GCD 异步绘制；或者直接重写drawRect方但如果在重写drawRect方法就不需要用GCD异步线程了，因为drawRect本来就是异步绘制

绘制 cell 不建议使用 UIView，建议使用 CALayer。 UIView 的绘制是建立在 CoreGraphic 上的，使用的是 CPU。CALayer 使用的是 Core Animation，CPU、GPU 通吃，由系统决定使用哪个。View的绘制使用的是自下向上的一层一层的绘制，然后渲染 Layer处理的是Texture，利用GPU的Texture Cache和独立的浮点数计算单元加速纹理的处理。GPU 不喜欢 透明，所以所有的绘图一定要弄成不透明，对于圆角和阴影这些的可以截个伪透明的小图然后绘制上去。在layer的回调里一定也只做绘图，不做计算！

cell被重用时，它内部绘制的内容并不会被自动清除，因此需要调用setNeedsDisplay或setNeedsDisplayInRect:方法。

## 65. 离屏渲染 (Offscreen-Renderd)

下面的情况或操作会引发离屏渲染：

- 为图层设置遮罩 (`layer.mask`)
- 将图层的`layer.masksToBounds / view.clipsToBounds`属性设置为`true`
- 将图层`layer.allowsGroupOpacity`属性设置为`YES`和`layer.opacity`小于`1.0`
- 为图层设置阴影 (`layer.shadow *`)。
- 为图层设置`layer.shouldRasterize=true` (光栅化)
- 具有`layer.cornerRadius, layer.edgeAntialiasingMask, layer.allowsEdgeAntialiasing`的图层 (圆角、抗锯齿)
- 文本 (任何种类, 包括`UILabel, CATextLayer, Core Text`等)。
- 使用`CGContext`在`drawRect` :方法中绘制大部分情况下会导致离屏渲染, 甚至仅仅是一个空的实现

### 优化方案

圆角优化 使用`CAShapeLayer`和`UIBezierPath`设置圆角；直接覆盖一张中间为圆形透明的图片（推荐使用）

shadow优化 使用`ShadowPath`指定`layer`阴影效果路径，优化性能

使用异步进行`layer`渲染 (Facebook开源的异步绘制框架`AsyncDisplayKit`)

设置`layer`的`opaque`值为`YES`, 减少复杂图层合成

尽量使用不包含透明 (alpha) 通道的图片资源

尽量设置`layer`的大小值为整形值

### Core Animation工具检测离屏渲染

对于离屏渲染的检测，苹果为我们提供了一个测试工具Core Animation。可以在Xcode->Open Developer Tools->Instruments中找到，如下图：

对于 `Misaligned images` 会有两种颜色：一种是洋红色，另一种是黄色。

[blog.cocoabit.com](http://blog.cocoabit.com)



洋红色是因为像素没对齐，比如上面的 label，一般情况下因为像素没对齐，需要抗锯齿，图像会出现模糊的现象。

解决办法：在设置 view 的 frame 时，在高分屏避免出现 21.3, 6.7这样的小数，尤其是 x, y坐标，用 ceil 或 floor 或 round 取整。每 0.5 个点对应一个 pixel, 0.3,0.7这样的就难为 iPhone 了，低分屏不要出现小数。

黄色是因为显示的图片实际大小与显示大小不同，对图片进行了拉伸，测试显示使用 image view 显示实际大小的图也会变黄。

减少洋红色和黄色可以提升滚动的流畅性

## 65. UIView 和 CALayer

UIView是iOS中所有的界面元素都继承自它，它本身完全是由CoreAnimation来实现的。每一个 UIView内部都默认关联着一个layer，真正的绘图部分，是由一个叫CALayer (Core Animation Layer) 的类来管理；

UIView有个layer属性，可以返回它的主CALayer实例，UIView有一个layerClass方法，返回主 layer所使用的类，UIView的子类，可以通过重载这个方法，来让UIView使用不同的CALayer来显示；

UIView的layer树形在系统内部，被维护着三份copy (presentLayer Tree、modelLayer Tree、render Tree)，修改动画的属性，其实是 Layer 的 presentLayer 的属性值；

动画的运作：对UIView的subLayer (非主Layer) 属性进行更改，系统将自动进行动画生成。

### 区别

1 UIView继承自UIResponder，能接收并响应事件， 负责显示内容的管理； 而CALayer继承自NSObject，不能响应事件，负责显示内容的绘制；

2 UIView侧重于展示内容，而CALayer则侧重于图形和界面的绘制；

3 当View展示的时候，View是layer的CALayerDelegate，View展示的内容是由CALayer进行display的；

4 view内容展示依赖CALayer对内容的绘制，UIView的frame也是由内部的CALayer进行绘制；

5 对UIView的属性修改，不会引起动画效果，但是对于CALayer的属性修改，是支持默认动画效果的，在view执行动画的时候，view是layer的代理，layer通过actionForLayer: forKey向对应的代理view请求动画action；

6 每个 UIView 内部都有一个 CALayer 在背后提供内容的绘制和显示，并且 UIView 的尺寸样式都由内部的 Layer 所提供，layer 比 view 多了个 anchorPoint；

7 一个CALayer的frame是由其anchorPoint, position, bounds, transform共同决定的，而一个UIView的的frame只是简单地返回CALayer的frame，同样UIView的center和bounds也只是简单返回CALayer的Position和Bounds对应属性。

## 66. TCP 和 UDP

**TCP：传输控制协议，提供的是面向连接、可靠的字节流服务。**当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，一个TCP连接必须要经过“**三次握手**”才能建立起来，之后才能传输数据。TCP提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

**UDP：用户数据报协议，是面向数据报的运输层协议。**它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去！UDP适用于一次只传送少量数据、对可靠性要求不高的应用环境。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快

**tcp协议和udp协议的区别**

**tcp-面向连接 udp-面向非连接**

**tcp-传输可靠 udp-不可靠**

**tcp-传输大量数据 udp-少量数据**

**tcp-速度慢 udp-快**

TCP连接的三次握手

第一次握手：客户端发送syn包( $syn=j$ )到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN ( $ack=j+1$ )，同时自己也发送一个SYN包 ( $syn=k$ )，即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK ( $ack=k+1$ )，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“**四次握手**”（过程就不细写了，就是服务器和客户端交互，最终确定断开）

## 67. socket 和 http

socket连接和http连接的区别

http是基于socket之上的。socket是一套完成tcp, udp协议的接口。

HTTP协议：简单对象访问协议，对应于应用层，HTTP协议是基于TCP连接的

tcp协议： 对应于传输层

ip协议： 对应于网络层

TCP/IP是传输层协议，主要解决数据如何在网络中传输；而HTTP是应用层协议，主要解决如何包装数据。

Socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口，通过Socket，我们才能使用TCP/IP协议。

http连接：短连接，即客户端向服务器端发送一次请求，服务端响应后连接，请求结束后，会主动释放连接即会断掉

socket连接：长连接，理论上客户端和服务端一旦建立连接将不会主动断掉；但由于各种因素可能会断开，比如：服务端或客户端主机down了，网络故障，或两者之间长时间没有数据传输，网络防火墙可能会断开该连接以释放网络资源。所以当一个socket连接中没有数据的传输，为了维持连接需要发送心跳消息；

**socket建立网络连接的步骤**

建立socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket，另一个运行于服务器端，称为ServerSocket。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

1 服务器监听：服务端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

2 客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务端套接字的地址和端口号，然后就向服务端套接字提出连接请求。

3 连接确认：当服务端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

**socket编程中（荐研究GCDAsyncSocket）的断线重连机制**

断线重连：使得IM软件能够长在线，或者短时间内掉线，最好可以做到用户无感知。目的是让IM软件维持在线的状态。

**实现方法**

IM客户端始终尽可能的保持连接跟服务器的连接，客户端维护已登录状态，以便断线重连。从逻辑层次上来说，断线重连的逻辑是基于登录逻辑的，首次登录成功后，都有可能断线重连

断线重连，实质分两步：一、使客户端断线；二、让客户端重连服务器。一般来说这两步是一个有前后顺序，完整的过程。

一、使客户端断线，即让客户端处于“未连接”状态。以下情况将触发这个事件：

1. 网络切换，如从WiFi切换到4G，网络事件。
2. 网络连接失败、网络不可用。
3. 心跳失败、心跳超时，统称心跳失败。
4. IM软件后台运行即将结束。

## 二、让客户端重连服务器，客户端根据以下几种情况实现重连服务器。

1. iOS系统“网络可用”的通知
2. IM软件切换到前台，用户触发事件。
3. 网络切换，如从WiFi切换到4G，网络事件。
4. 心跳失败的事件。
5. 客户端重新启动事件。

### 断线重连的场景总结为：

1. 重新启动（自动登录）

需要提前加载用户缓存，保证用户到达主界面后能看到历史信息。

2. 网络错误，网络切换

网络连接失败有很多种，不同的场景，客户端要使用不同的逻辑处理。

3. 心跳失败

心跳超时，失败统称心跳失败。这个案例说明当前客户端—服务器连接已经损坏，或者当前用户身份有变化。心跳失败后首先将客户端离线，然后进行断线重连操作，避免心跳失败和网络错误事件一并发生，造成两次登录。

4. 网络可达或者切换到前台

为了避免重复登录，当IM软件处于“登录成功”、“连接中”或者“已注销”的几个状态的时候，客户端忽略“网络可达或者切换到前台”的事件。

### 客户端心跳

IM基本的底层逻辑中有“心跳”概念，即客户端定时向Server发一个信令包，表示客户端还“活着”。注意，是客户端发起的。那么心跳终止了会发生什么事情呢？分两种情况：Server主动断开socket，客户端主动断开socket。

#### 1. Server主动断开socket

Server只是接收客户端发起的心跳。假如，Server长时间没有收到客户端的心跳，**Server认为客户端已经“死了”，主动断开这个连接。此时客户端可能就是假在线了。**

#### 2. 客户端断开socket

客户端对待心跳，要比Server麻烦一些。客户端要关注两个值：

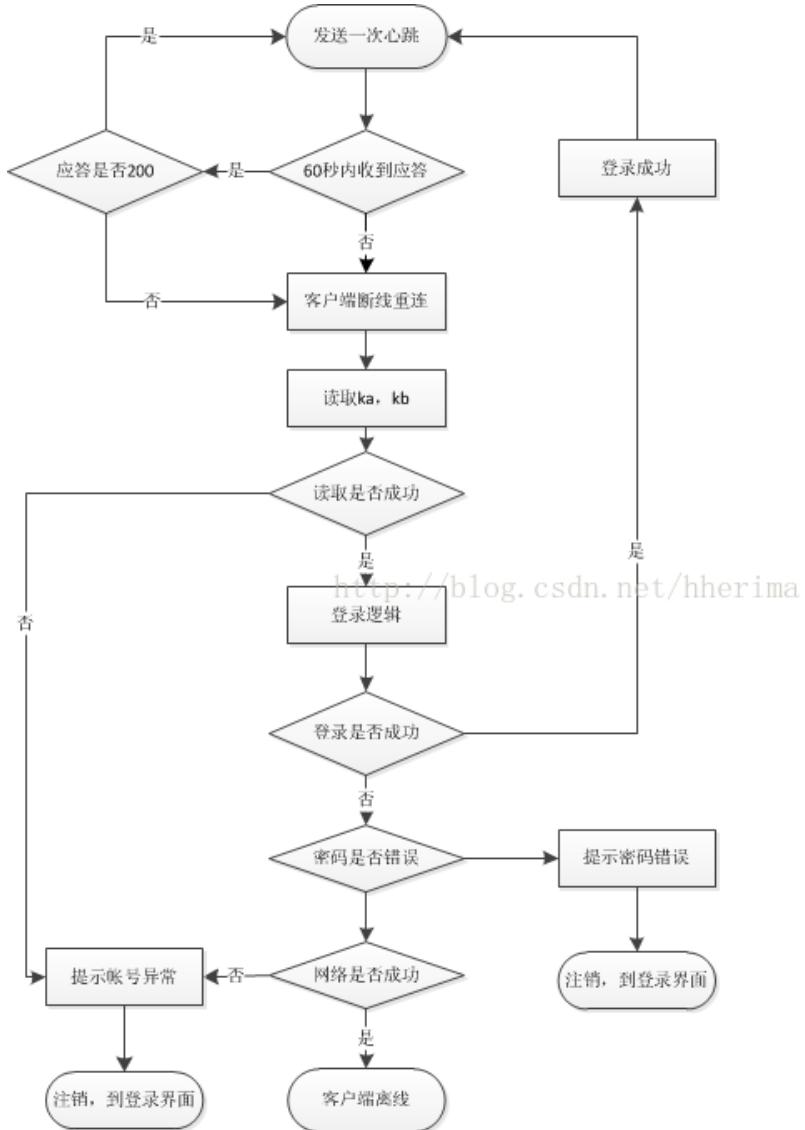
- 心跳间隔值，即客户端多长时间发一次心跳？
- 心跳的超时时间。客户端发送一次心跳，**如果长时间得不到Server应答，代表网络糟糕。客户端需要断开socket，主动离线。**

很明显，第二点就是客户端主动断开的情况，一般情况下，超时时间为60秒。

网上也有争论：到底是否需要心跳，微信是没有心跳的，qq和飞信有心跳。

也有专家说心跳包已经影响到移动网络，因为心跳是定时频繁发送。

下面是“心跳失败”引起的断线重连的流程图



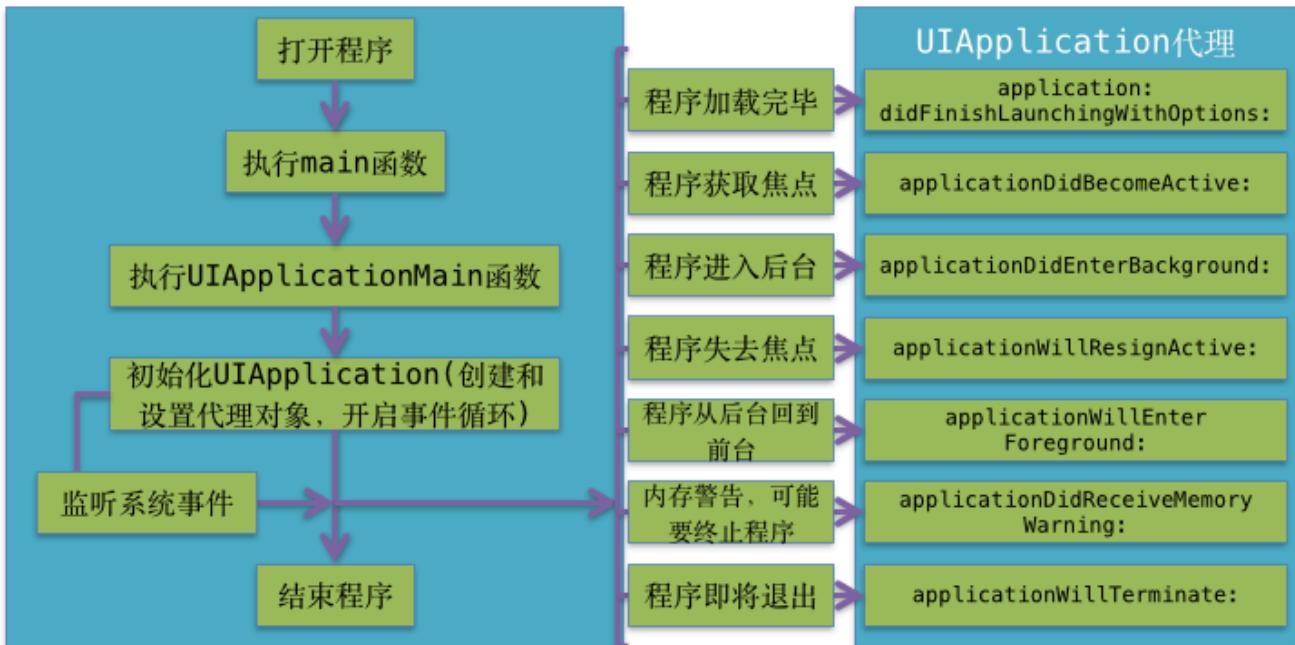
## 68. 远程推送 APNS

远程推送的基本过程

- 1 客户端的app需要将用户的UDID和app的bundleID发送给apns服务器，进行注册， apns将加密后的device Token返回给app；
- 2 app获得device Token后，上传到公司服务器；
- 3 当需要推送通知时，公司服务器会将推送内容和device Token一起发给apns服务器；
- 4 apns再将推送内容推送到相应的客户端app上。

## 69. iOS应用程序生命周期

ios程序启动原理（过程）：如下



appdelegate 执行顺序：如下

```
17 #pragma mark - app 第一次启动
18 - (BOOL)application:(UIApplication *)application willFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
19 //进程启动但还没进入状态保存
20 return YES;
21 }
22 - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
23 //启动基本完成程序准备开始运行
24 return YES;
25 }
26 - (void)applicationDidBecomeActive:(UIApplication *)application {
27 //当应用程序进入活动状态执行
28 }
29
30 #pragma mark - 按 home 键
31 - (void)applicationWillResignActive:(UIApplication *)application {
32 //当应用程序将要入非活动状态执行, 在此期间, 应用程序不接收消息或事件, 比如来电
33 }
34 - (void)applicationDidEnterBackground:(UIApplication *)application {
35 //当程序被推送到后台的时候调用。所以要设置后台继续运行, 则在这个函数里面设置即可
36 [application beginBackgroundTaskWithExpirationHandler:^{
37 NSLog(@"begin Background Task With Expiration Handler");
38 }];
39 }
40
41 #pragma mark - 双击home键, 再打开程序
42 - (void)applicationWillEnterForeground:(UIApplication *)application {
43 //当程序从后台将要重新回到前台时候调用
44 }
45 - (void)applicationDidBecomeActive:(UIApplication *)application {
46 //当应用程序进入活动状态执行
47 }
48
49 - (void)applicationWillTerminate:(UIApplication *)application {
50 //当程序将要退出时被调用, 通常用来保存数据和一些退出前的清理工作。需要设置UIApplicationExitsOnSuspend的键值
```

## 1、应用程序的状态

状态如下：

Not running 未运行 程序没启动

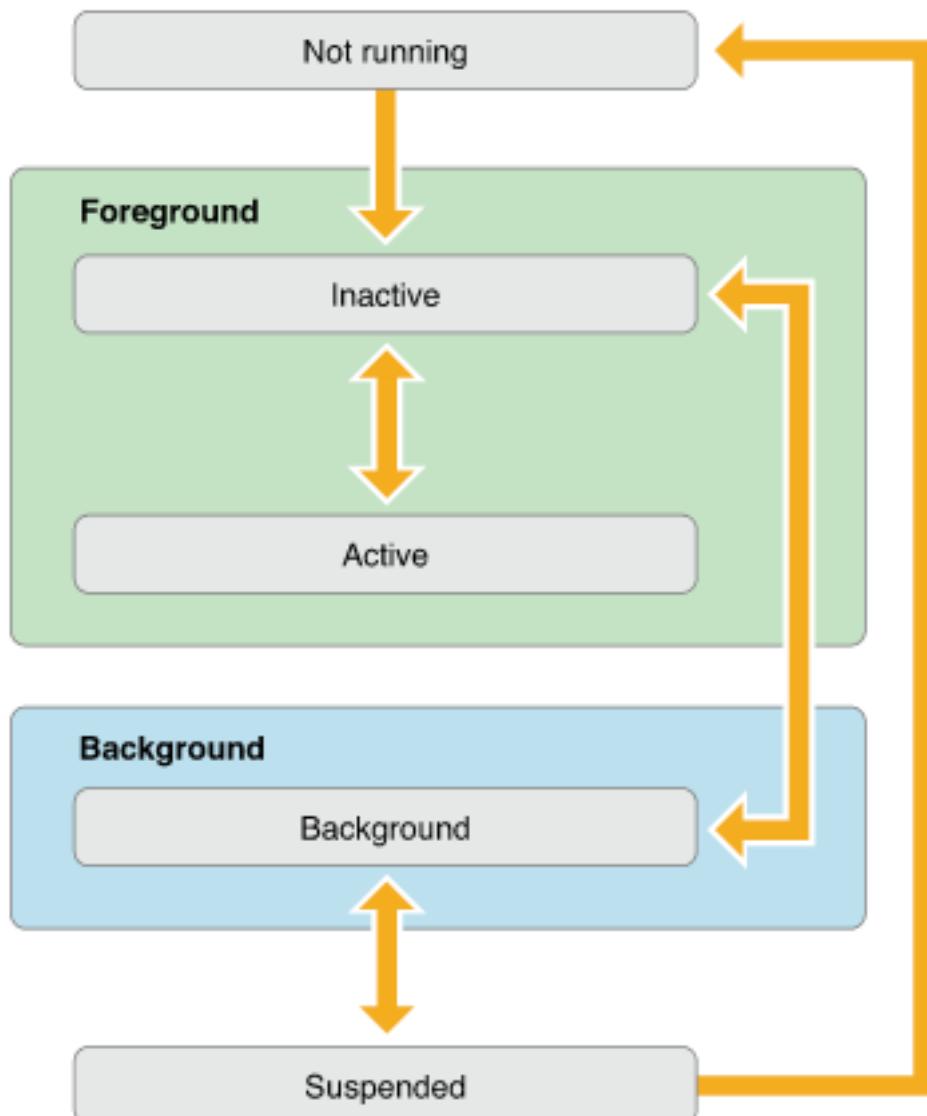
Inactive 未激活 程序在前台运行，不过没有接收到事件。在没有事件处理情况下程序通常停留在这个状态

Active 激活 程序在前台运行而且接收到了事件。这也是前台的一个正常的模式

Background 后台 程序在后台而且能执行代码，大多数程序进入这个状态后会在在这个状态上停留一会。时间到之后会进入挂起状态(Suspended)。有的程序经过特殊的请求后可以长期处于Background状态

Suspended 挂起 程序在后台不能执行代码。系统会自动把程序变成这个状态而且不会发出通知。当挂起时，程序还是停留在内存中的，当系统内存低时，系统就把挂起的程序清除掉，为前台程序提供更多的内存。

下图是程序状态变化图：



## UIApplicationMain

```
UIApplicationMain(int argc, char *argv[], NSString *principalClassName, NSString *delegateClassName);
/*
 argc: 系统或者用户传入的参数个数
 argv: 系统或者用户传入的实际参数
 argc、argv: 直接传递给UIApplicationMain进行相关处理即可
 1.principalClassName: 指定应用程序类名（app的象征），该类必须是UIApplication(或子类)。如果为nil，则系统默认UIApplication类
 2.delegateClassName: 创建并设置UIApplication对象的代理，指定应用程序的代理类，该类必须遵守UIApplicationDelegate协议
 UIApplication函数会根据principalClassName创建UIApplication对象，根据delegateClassName创建一个delegate对象，并将该delegate对象赋值给UIApplication对象中的delegate属性
 3.接着会建立应用的Main Runloop（事件循环），处理与用户交互产生的事件(首先会在程序完毕后调用delegate对象的application:didFinishLaunchingWithOptions:方法)
 程序正常退出时UIApplicationMain函数才返回
*/
```

## 70. view视图生命周期

视图创建：调用viewDidLoad方法

视图即将可见：调用viewWillAppear方法

视图已经可见：调用viewDidAppear方法

视图即将不可见：调用viewWillDisappear方法

视图已经不可见：调用viewDidDisappear方法

系统低内存：调用didReceiveMemoryWarning方法和viewDidUnload方法

注意：

1 viewDidLoad方法在应用运行的时候只会调用一次，其他方法会被调用多次。

2 低内存情况下，iOS会调用didReceiveMemoryWarning和viewDidUnload方法，但是iOS6以后就不在使用viewDidUnload方法，仅支持didReceiveMemoryWarning方法，该方法主要用于释放内存（视图控制器中的一些成员变量和视图的释放）

## 71. autorelease pool

**autorelease的原理是什么？**

autorelease实际上只是把对release的调用延迟了，对于每一个autorelease，系统只是把该Object放入了当前的Autorelease pool中，当该pool被释放时，该pool中的所有Object会被调用Release。

**autorelease的对象何时释放？**

对于autorelease pool本身，会在如下两个条件发生时候被释放

- 1、手动释放Autorelease pool
- 2、Runloop结束后自动释放

对于autorelease pool内部的对象在引用计数 == 0的时候释放。release和autorelease pool的drain都会触发引用计数减一。

**NSAutoreleasePool是什么？**

NSAutoreleasePool实际上是个对象引用计数自动处理器，在官方文档中被称为是一个类。

NSAutoreleasePool可以同时有多个，它的组织是个栈，总是存在一个栈顶pool，也就是当前pool，每创建一个pool，就往栈里压一个，改变当前pool为新建的pool，然后，每次给pool发送drain消息，就弹出栈顶的pool，改当前pool为栈里的下一个 pool。

NSAutoreleasePool 中还提到，每一个线程都会维护自己的 autoreleasepool 堆栈，每一个 autoreleasepool 只对应一个线程。

**NSAutoreleasePool和autoreleasepool的区别**

当你使用ARC，就必须将NSAutoreleasePool的地方换成 @autoreleasepool

两者的作用时间不一样。AutoReleasePool对象的写法作用于运行时，@autoreleasepool作用于编译阶段。如果要启用ARC的话，在编译阶段就需要告诉编译器启用自动引用计数管理，而不能在运行时动态添加。

autoreleased 对象是被添加到了当前最近的 autoreleasepool 中的，只有当这个 autoreleasepool 自身 drain 的时候，autoreleasepool 中的 autoreleased 对象才会被 release 。这个对象的引用计数 -1 ，变成了 0 该 autoreleased 对象最终被释放

向一个对象发送-autorelease消息，就是将这个对象加入到当前AutoreleasePoolPage的栈顶next指针指向的位置。

自动释放池是NSAutoreleasePool的实例，其中包含了收到autorelease消息的对象。当一个自动释放池自身被销毁 (dealloc) 时，它会给池中每一个对象发送一个release消息（如果你给一个对象多次发送autorelease消息，那么当自动释放池销毁时，这个对象也会收到同样数目的release消息）

**autorelease作用：**

- 1 对象执行autorelease方法时会将对象添加到自动释放池中
- 2 当自动释放池销毁时自动释放池中所有对象作release操作

- 3 对象执行autorelease方法后自身引用计数器不会改变，而且会返回对象本身
- 4 autorelease实际上只是把对象release的调用延迟了，对于对象的autorelease系统只是把当前对象放入了当前对应的autorelease pool中，当该pool被释放时 ([pool drain])，该pool中的所有对象会被调用Release，从而释放使用的内存。这个可以说是autorelease的优点，因为无需我们再关注他的引用计数，直接交给系统来做！

### @autoreleasepool

自己开启一个线程，你需要创建自己的自动释放池块。如果你的应用或者线程长时间存活，并可能产生大量的自动释放的对象；你应该手动创建自动释放池块；否则，自动释放的对象会积累并占用你的内存。当使用ARC来管理内存时，在线程中使用 `for` 大量分配对象而不用`autoreleasepool`则可能会造成内存泄露

如果你创建的线程没有调用 Cacoa，你不需要使用自动释放池块。

在ARC环境下，使用`autoreleasepool`可以释放池上下文，但是如下代码，`autoreleasepool`有必要吗？

```
for (int i = 0; i < 1000000; i++) {
 @autoreleasepool {
 NSString *str = @"ABC";
 NSString *string = [str lowercaseString];
 string = [string stringByAppendingString:@"xyz"];
 NSLog(@"%@", string);
 }
}
```

有必要，在遇到需要大量创建对象的地方使用`autoreleasepool`可以加快对象释放的速度。

## 72. view layout

相关方法

- (CGSize)sizeThatFits:(CGSize)size  
- (void)sizeToFit

---

- (void)layoutSubviews  
- (void)layoutIfNeeded  
- (void)setNeedsLayout

---

- (void)setNeedsDisplay  
- (void)drawRect

`layoutSubviews`在以下情况下会被调用：

- 1、`init`初始化不会触发`layoutSubviews`，但是用`initWithFrame`初始化时，当`rect`的值不为`CGRectZero`时，也会触发
- 2、`addSubview`会触发`layoutSubviews`
- 3、设置`view`的`frame`会触发`layoutSubviews`，前提是`frame`的值设置前后发生了变化

- 4、滚动一个UIScrollView会触发layoutSubviews
- 5、旋转Screen会触发父UIView上的layoutSubviews事件
- 6、改变一个UIView大小的时候也会触发父UIView上的layoutSubviews事件

layoutSubviews，当我们在某个类的内部调整子视图位置时，需要调用。  
反过来的意思就是说：如果你想要在外部设置subviews的位置，就不要重写。

### 刷新子对象布局

- layoutSubviews：默认没有做任何事情，需要子类进行重写
- setNeedsLayout：标记为需要重新布局，异步调用layoutIfNeeded刷新布局，不立即刷新，但layoutSubviews一定会被调用
- layoutIfNeeded：如果有需要刷新的标记，立即调用layoutSubviews进行布局（如果没有标记，不会调用layoutSubviews）

注：setNeedsLayout方法并不会立即刷新，立即刷新需要调用layoutIfNeeded方法。

如果要立即刷新，要先调用[view setNeedsLayout]，把标记设为需要布局，然后马上调用[view layoutIfNeeded]，实现布局

setNeedsLayout 在receiver标上一个需要被重新布局的标记，在系统RunLoop的下一个周期自动调用layoutSubviews

layoutIfNeeded UIKit会判断该receiver是否需要layout，遍历的不是superview链，应该是subviews链

在视图第一次显示之前，标记总是“需要刷新”的，可以直接调用[view layoutIfNeeded]

### 重绘

- drawRect:(CGRect)rect：重写此方法，执行重绘任务
- setNeedsDisplay：标记为需要重绘，异步调用drawRect
- setNeedsDisplayInRect:(CGRect)invalidRect：标记为需要局部重绘

sizeToFit会自动调用sizeThatFits方法；

sizeToFit不应该在子类中被重写，应该重写sizeThatFits

sizeThatFits传入的参数是receiver当前的size，返回一个适合的size

sizeToFit可以被手动直接调用

sizeToFit和sizeThatFits方法都没有递归，对subviews也不负责，只负责自己

---

layoutSubviews对subviews重新布局  
layoutSubviews方法调用先于drawRect

drawRect是对receiver的重绘，能获得context

setNeedDisplay在receiver标上一个需要被重新绘图的标记，在下一个draw周期自动重绘，  
iphone device的刷新频率是60hz，也就是1/60秒后重绘

setNeedsDisplay 该方法在调用时，会自动调用drawRect方法。drawRect方法主要用来画图

总结

当需要刷新布局时，用setNeedsLayout方法；当需要重新绘画时，调用setNeedsDisplay方法

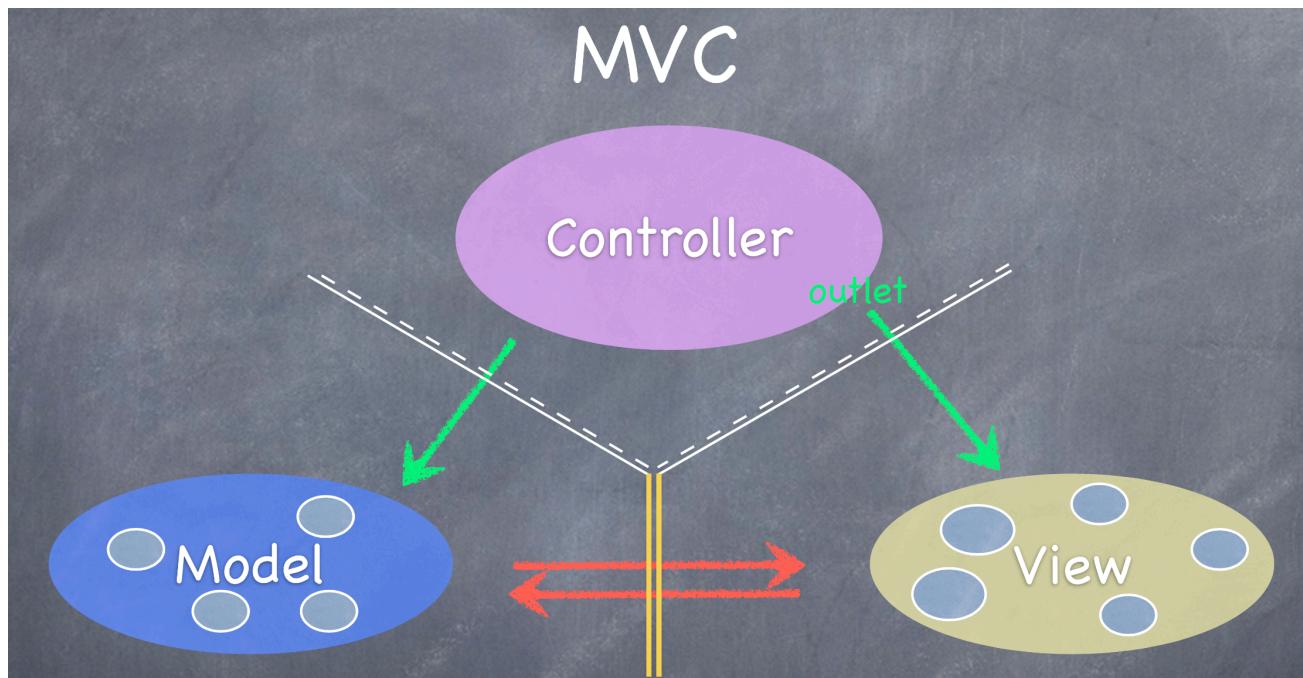
## 73. 应用程序的架构 MVC、MVVM

- 界面
- 数据
- 事件
- 业务

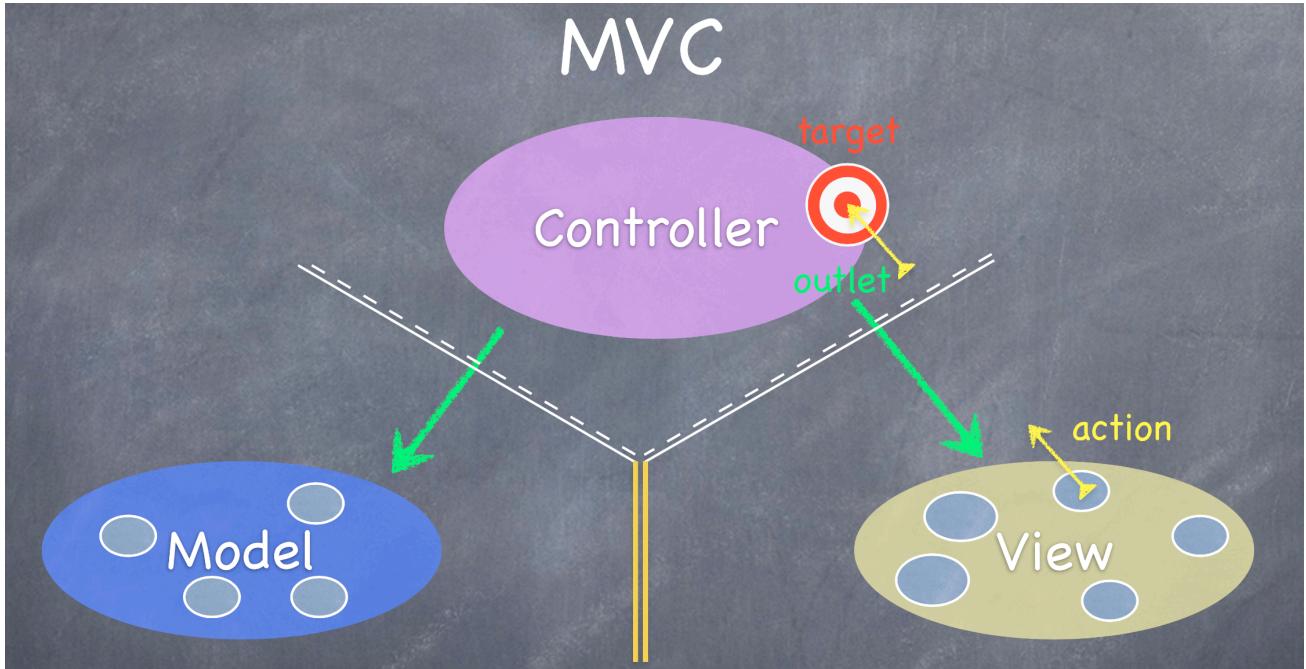
MVC

iOS应用程序都遵循Model-View-Controller的架构，Model负责存储数据和处理业务逻辑，View负责显示数据和与用户交互，Controller是两者的中介，协调Model和View相互协作。它们的通讯规则如下：

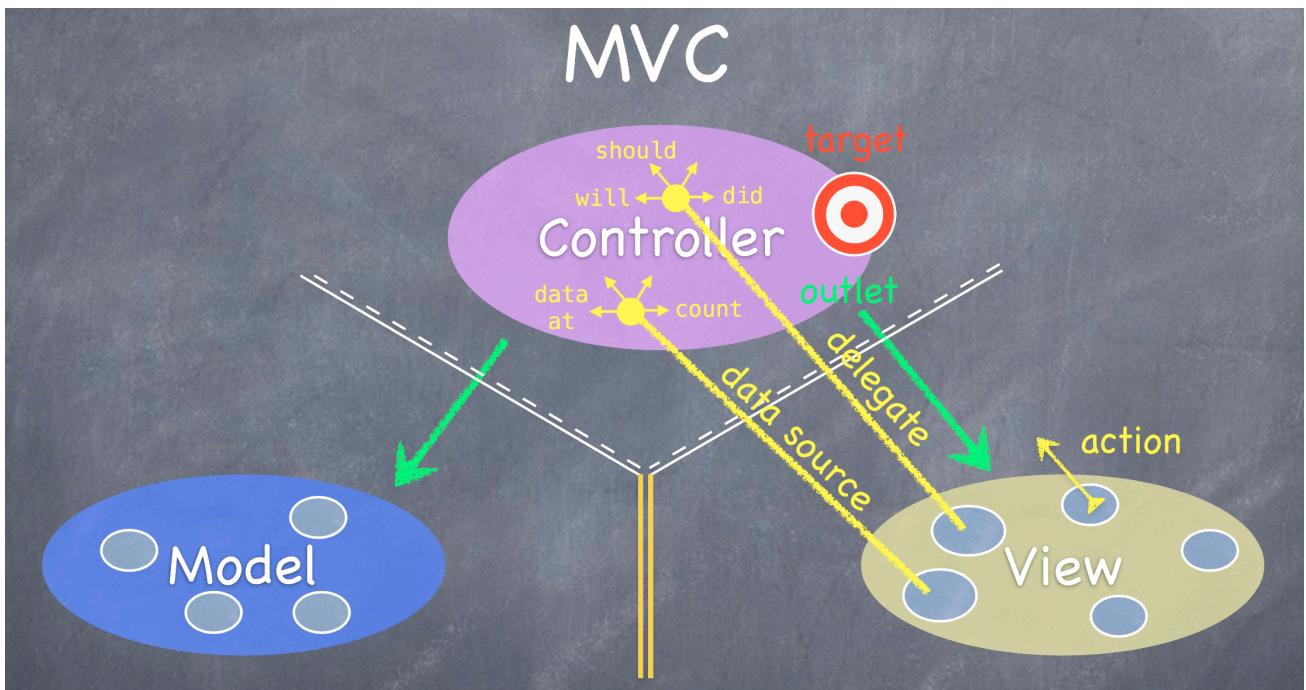
1、Controller能够访问Model和View，Model和View不能互相访问



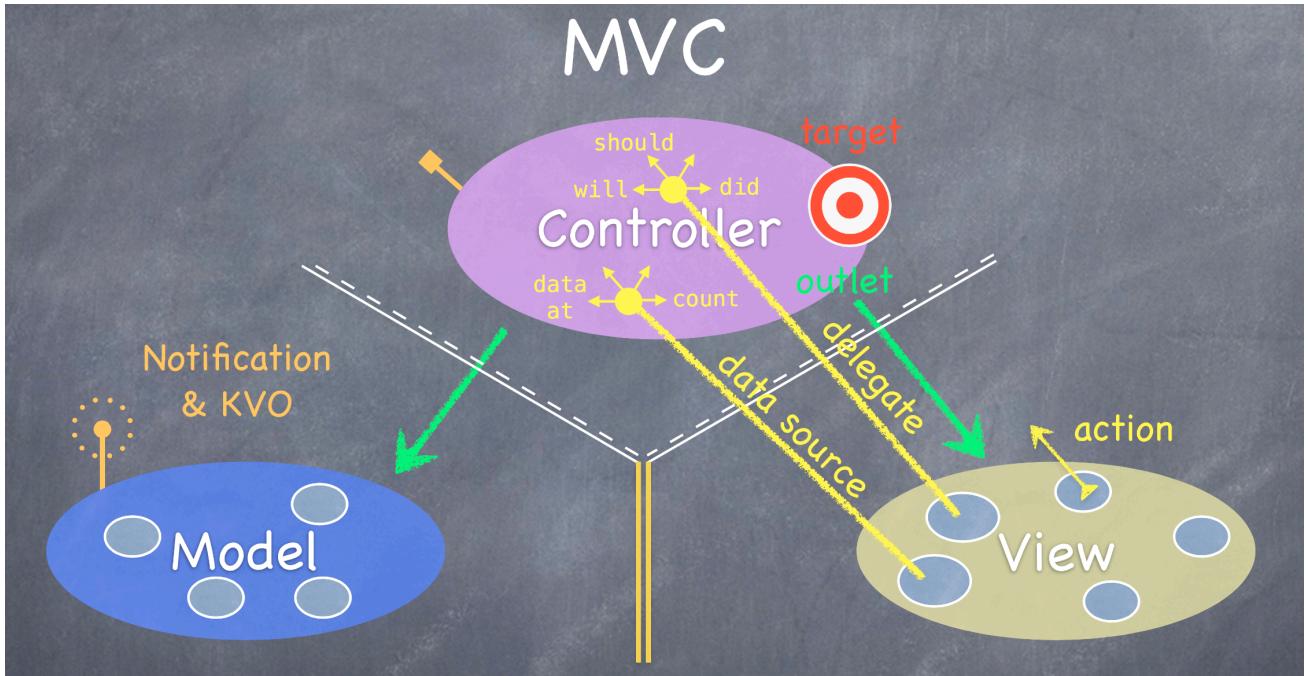
2、当View与用户交互产生事件时，使用target-action方式来处理



3、当View需要处理一些特殊UI逻辑或获取数据源时，通过delegate或data source方式交给Controller来处理



4、Model不能直接与Controller通信，当Model有数据更新时，可以通过Notification或KVO (Key Value Observing)来通知Controller更新View



## MVVM

MVVM就是在MVC的基础上分离出业务处理的逻辑到viewModel层，即：

model层，API请求的原始数据、数据持久化

view层，视图展示，由viewController来控制

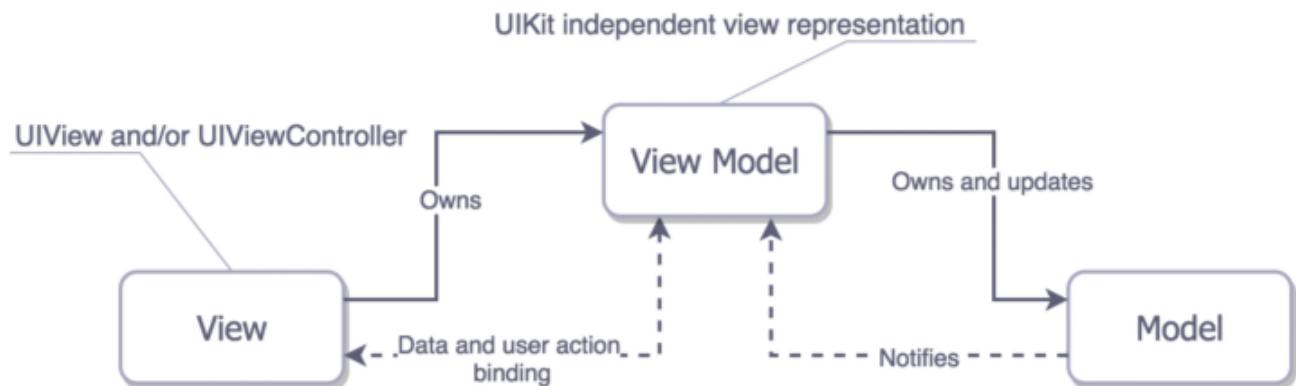
viewModel层，负责网络请求、业务处理和数据转化

简单来说，就是API请求完数据，解析成model，之后在viewModel中转化成能够直接被视图层使用的数据，交付给前端。

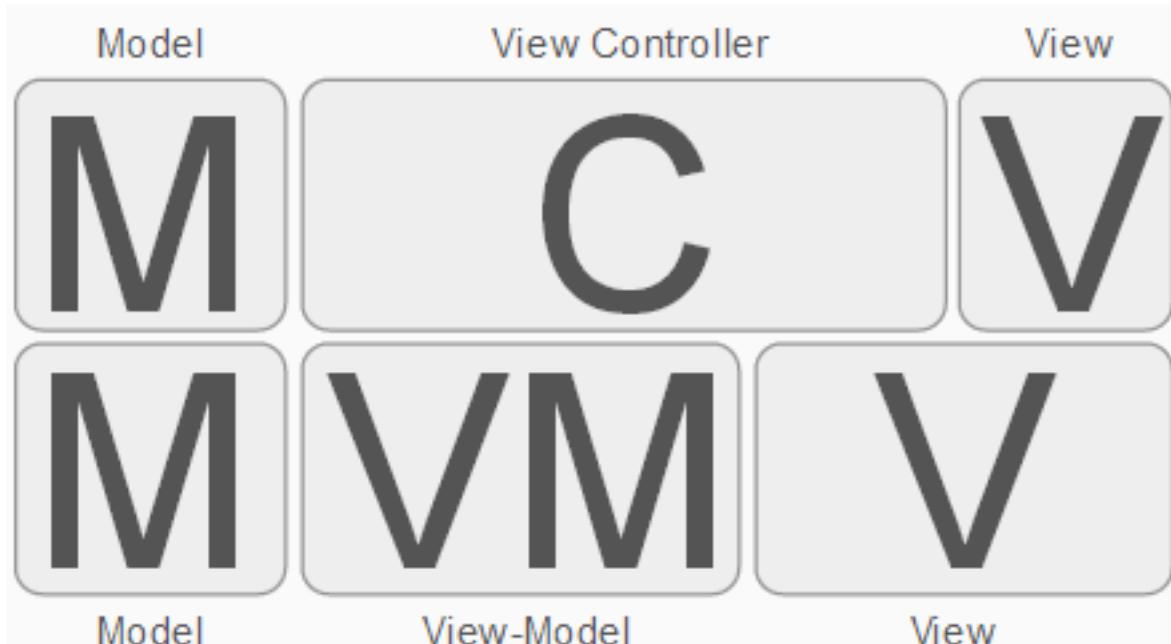
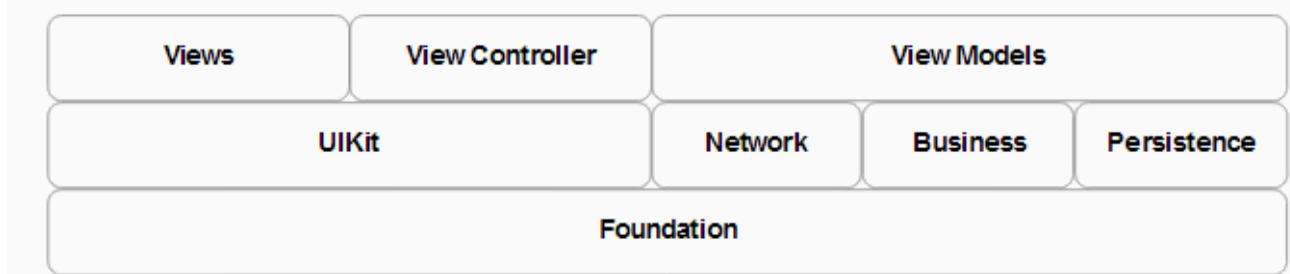
经过viewModel转化之后的数据由viewModel保存，与数据相关的处理都将在viewModel中处理。

viewModel返回给view层

view层是由viewController控制的。view层只做展示，不做业务处理。view层的数据由viewModel提供。



## iOS MVVM Application Layer Architecture



链接有 MVVM 使用详细介绍，里面也附有 Demo  
<http://www.cocoachina.com/ios/20170313/18870.html>  
<http://www.cocoachina.com/ios/20170213/18659.html>  
<http://www.cocoachina.com/ios/20150526/11930.html>

## 74. FMDB、SQLite

FMDatabaseQueue 用于在多线程中执行多个查询或更新，它是线程安全的。

FMDatabase 这个类是线程不安全的，如果在多个线程中同时使用一个 FMDatabase 实例，会造成数据混乱等问题。

1、增(创)表

格式：

(1)create table 表名 (字段名 1 字段类型 1, 字段名 2 字段类型 2, ...) ;  
(2)create table if not exists 表名 (字段名 1 字段类型 1, 字段名 2 字段类型 2, ...) ;

例如：

```
create table t_student (id integer, name text, age inetger, score real)
```

2、删表

格式：

(1)drop table 表名 ;  
(2)drop table if exists 表名 ;

例如：

```
drop table t_student;
```

3、插入数据(insert)

格式：

```
insert into 表名 (字段1, 字段2, ...) values (字段1的值, 字段2的值, ...);
```

例如：

```
insert into t_student (name, age) values ('mj', 10);
```

【备注】数据库中的字符串内容应该用单引号 ' 括住

4、更新数据(update)

格式：

```
update 表名 set 字段1= 字段1的值, 字段2= 字段2的值, ...;
```

例如：

```
update t_student set name = 'jack', age = 20;
```

【备注】上面的示例会将 t\_student 表中所有记录的 name 都改为 jack, age 都改为 20。 6、

5、删除数据(delete)

格式： delete from 表名 ;

例如：

```
delete from t_student;
```

## 75. 简述内存分区情况

- 代码区：存放函数二进制代码
- 数据区：系统运行时申请内存并初始化，系统退出时由系统释放，存放全局变量、静态变量、常量
- 堆区：通过 `malloc` 等函数或 `new` 等动态申请到的，需手动申请和释放
- 栈区：函数模块内申请，函数结束时由系统自动释放，存放局部变量、函数参数

## 76. 各属性作用

- `readonly`: 可读可写，需要生成 `getter`、`setter` 方法
- `readonly`: 只读，只生成 `getter`，不会生成 `setter`，不希望属性在类外部被更改
- `assign`: 赋值，`setter` 方法将传入的参数赋值给实例变量；仅设置变量时，`assign` 用于基本数据类型
- `retain`: 表示持有特性，`setter` 方法将传入的参数先保留，在赋值，传入参数的引用计数会+1
- `copy`: 表示赋值特性，`setter` 方法将传入的对象复制一份
- `nonatomic`: 非原子操作，决定编译器生成的 `setter`、`getter` 是否是原子操作
- `atomic`: 表示多线程安全，一般使用 `nonatomic`

## 77. 简述 Notification、KVC、KVO、delegate? 区别？

- KVO: 一对多，观察者模式，键值观察机制，提供了观察某一属性变化的方法，极大简化了代码
- KVC: 键值编码，一个对象在调用 `setValue`
  - 检查是否存在相应 `key` 的 `set` 方法，存在就调用 `set` 方法
  - `set` 方法不存在，就查找 `_key` 的成员变量是否存在，存在就直接赋值
  - 如果 `_key` 没找到，就查找相同名称的 `key`，存在就赋值
  - 如果都没有找到，就调用 `valueForUndefinedKey` 和 `setValue:forUndefinedKey`
- delegate: 发送者和接收者的关系是直接、一对一的关系
- Notification: 观察者模式，发送者和接收者的关系是间接、多对多的关系

区别：

- `delegate` 的效率比 `Notification` 高
- `delegate` 比 `Notification` 更直接，需要关注返回值，常带有 `should` 关键词；  
`Notification` 不关心结果，常带有 `did` 关键词
- 两个模块之间的联系不是很紧密，就用 `Notification` 传值，比如多线程之间的传值
- `KVO` 容易实现两个对象的同步，比如 `Model` 和 `View` 的同步

## 78. id 和 nil 代表什么

- `id`类型的指针可以指向任何 `OC` 对象
- `nil` 代表空值（空指针的值，`0`）

## 79.nil、Nil、NULL、NSNull

OC 中

- nil、Nil、NULL 都表示 `(void *)0`
- NSNull 继承于 `NSObject`, 很特殊的类, 表示是空, 什么也不存储, 但它却是对象, 只是一个占位对象。使用场景: 比如服务端接口让传空, `NSDictionary *params = @{@"arg1" : @"value1", @"arg2" : arg2.isEmpty ? [NSNull null] : arg2};`

区别

- `NULL` 是宏, 是对于 C 语言指针而使用的, 表示空指针
- `nil` 是宏, 是对于 OC 中的对象而使用的, 表示对象为空
- `Nil` 是宏, 是对于 OC 中的类而使用的, 表示类指向空
- `NSNull` 是类类型, 是用于表示空的占位对象, 于 JS 或服务端的 `null` 类似的含义

## 80. 向一个 nil 对象发送消息会发生什么?

- 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
- 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)`
- 如果方法返回值为指针类型, 其指针大小为小于或等于 `sizeof(void *)`, `float`、`double`、`long double`、`long long` 的整型标量, 发送给 `nil` 的消息将返回`0`
- 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也都是`0`

## 81. self. 和 self-> 的区别

- `self.` 是调用 `getter` 或 `setter` 方法
- `self` 是当前本身, 是一个指向当前对象的指针
- `self->` 是直接访问成员变量

## 82. 如何访问并修改一个类的私有属性

- 通过 KVC
- 通过 runtime 访问并修改

## 83. 如何为 class 定义一个对外只读、对内可读写的属性

在头文件 `.h` 中将属性定义为 `readonly`, 在 `.m` 中将原属性重新定义为 `readwrite`

## 84. OC 中，meta-class 指的是什么？

meta-class 是 class 对像的类，为这个 class 类存储类方法，当一个类发送消息时，就去这个类对应的 meta-class 中查找那个消息，每个 class 都有不同的 meta-class，所有的 meta-class 都使用基类的 meta-class(假如类继承自 NSObject，那么他所对应的 meta-class 也是 NSObject)作为他们的类。

## 85. NSString 用 copy 和 strong 的区别

NSString 为不可变字符串，用 copy 和 strong 都是只分配一次内存，但是如果用 copy，需要先判断字符串是不是不可变字符串，如果是不可变字符串，就不再分配空间，如果是可变字符串才分配空间。如果程序中用到的 NSString 特别多，每一次都要先判断就会耗费性能，用 strong 就不会再判断了，所以不可变字符串可以直接用 strong。

## 86. 创建一个对象的步骤

- 开辟内存空间
- 初始化参数
- 返回内存地址值

## 87. setter、getter

- setter 方法：为外界提供一个设置成员变量的方法，好处 — 不让数据暴露在外，保证数据安全性；对设置的数据过滤
- getter 方法：为调用者返回对象内部的成员变量
- 点语法的本质是对 setter 或 getter 方法的调用

## 88. id、instancetype 是什么？区别？

- id：万能指针，能作为参数、方法的返回类型
- instancetype：只能作为方法的返回类型，并且返回的类型是当前定义类的类类型

## 89. 内存管理

- ARC 所做的是在代码编译期自动在合适的位置插入 release 或 autorelease，只要没有强指针指向对象，对象就会被释放。ARC 中不能手动使用 NSZone，也不能调用父类的 dealloc

调用对象的 release 方法会销毁对象吗？

- 不会，调用 release 只是将对象的引用计数器 -1，当对象的引用计数器 =0 时候会调用对象的 dealloc 方法才能释放对象的内存

## objc 使用什么机制管理对象内存?

- 通过引用计数器 (retainCount) 的机制来决定对象是否需要释放。每次 RunLoop 完成一个循环的时候，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，就被释放了。
- ARC 的判断准则：只要没有强指针指向对象，对象就会被释放

## 内存管理的范围?

- 管理所有继承自 NSObject 的对象，对基本数据类型无效。是因为对象和其他基本数据类型在系统中存储的空间不一样，其他局部变量主要存储在栈区（因为基本数据类型占用的存储空间是固定的，一般存放于栈区），而对象存储于堆中，当代码块结束时，这个代码块中的所有局部变量会自动弹栈清空，指向对象的指针也会被回收，这时对象就没有指针指向了，但依然存在于堆中，造成内存泄漏。

## 内存管理研究的对象：

- 野指针：指针变量没有进行初始化或指向的空间已经被释放。
  - 使用野指针调用对象方法，会报异常，程序崩溃
  - 在调用完 release 后，把保存对象指针的地址清空，赋值为 nil，找 oc 中没有空指针异常 所以 [nil retain] 调用方法不会有异常
- 内存泄漏
  - 如 Person \*p = [Person new]; (对象提前赋值 nil 或清空) 在栈区的 p 已经被释放，而 堆区 new 产生的对象还没有释放，就会造成内存泄漏
  - MRC 造成内存泄漏的情况：1、没有配对释放，不符合内存管理原则；2、对象提前赋值 nil 或 清空，导致 release 不起作用
- 僵尸对象：堆中已经被释放的对象 (retainCount=0)
- 空指针：指针赋值为空， nil

△ alloc、allocWithZone、new 时：该对象引用计数 +1；  
△ retain：手动为该对象引用计数 +1；  
△ copy：对象引用计数 +1； //注意：copy 的 oc 数据类型是否有 mutable，如有为深拷贝，新 对象引用计数为 1；如果没有，为浅拷贝，引用计数 +1  
△ mutableCopy：生成一个新对象，新对象引用计数 +1；  
△ release：手动为该对象引用计数 -1；  
△ autorelease：把该对象放入自动释放池，当自动释放池释放时，向池中的对象发送 release 消 息，其内的对象引用计数 -1，只能释放自己拥有的对象；

- △ `NSAutoreleasePool`: `NSAutoreleasePool` 是通过接收对象向它发送的 `autorelease` 消息，记录该对象的 `release` 消息，当自动释放池被销毁时，会自动向池中对象发送 `release` 消息。
- △ 对象如何加入池中：调用对象的 `autorelease` 方法
- △ 多次调用对象的 `autorelease` 方法会导致：野指针异常

## 90. KVC 的底层实现？

当一个对象调用 `setValue` 方法时，方法内部会做以下操作：

- 1、检查是否存在对应 `key` 的 `set` 方法，如果存在，就调用 `set` 方法
  - 2、如果 `set` 方法不存在，就会查找与 `key` 相同名称并且带下划线的成员变量 `_key`，如果有，则直接给成员变量赋值
  - 3、如果没有找到 `_key`，就会查找相同名称的属性 `key`，如果有就直接赋值
  - 4、如果还没找到，则调用 `valueForUndefinedKey:` 和 `setValue:forUndefinedKey:`
- 这些方法的默认实现都是抛出异常，可以根据需要重写他们

## 91. block 的内存管理

- 无论 ARC 还是 MRC，只要 `block` 没有访问外部变量，`block` 始终在全局区
- MRC 下：
  - `block` 如果访问外部变量，`block` 在栈区
  - 不能对 `block` 使用 `retain`，否则不能保存在堆区
  - 只有使用 `copy`，才能放到堆区
- ARC 下
  - `block` 如果访问外部变量，`block` 在堆区
  - `block` 是一个对象，可以使用 `copy` 或 `strong` 修饰，最好是使用 `copy`

## 92. App 的启动过程，从 `main` 说起

App 启动分两类：1、有 `stroyboard`；2、无 `storyboard`

有 `stroyboard` 情况下：

- 1 `main` 函数
- 2 `UIApplicationMain`
  - 创建 `UIApplication` 对象
  - 创建 `UIApplication` 的 `delegate` 对象
- 3 根据 `info.plist` 获得 `Main.storyboard` 的文件名，加载 `Main.storyboard`
  - 创建 `UIWindow`
  - 创建和设置 `UIWindow` 的 `rootViewController`
  - 显示窗口 `window`

没有 storyboard 情况下：

- 1 main 函数
- 2 UIApplicationMain
  - 创建 UIApplication 对象
  - 创建 UIApplication 的 delegate 对象
- 3 delegate 对象开始处理（监听）系统事件
  - 程序启动完毕时，调用 didFinishLaunching 方法
  - didFinishLaunching 方法中创建 UIWindow 设置 rootViewController 并显示窗口 window

## 93. tableview 的 cell 里面如何嵌套 collectionview

用自定义的继承自 UITableViewCell 的类，在 initWithFrame 方法里面初始化自定义的继承自 UICollectionView 的类

## 94. awakeFromNib 和 viewDidLoad 的区别

- awakeFromNib：当 .nib 文件被加载的时候，会发送一个 awakeFromNib 消息到 .nib 文件中每个对象，每个对象都可以定义自己的 awakeFromNib 来响应这个消息。即通过.nib 文件创建的 view 对象会执行awakeFromNib
- viewDidLoad：当 view 被加载到内存就会执行，不管是通过 nib 还是代码形式

## 95. 常见的 Crash 场景

- 访问僵尸对象
- 访问了不存在的方法
- 数组越界
- 在定时器下一次回调前将定时器释放，会 crash

## 96. AFN 断点续传

- 检查服务端文件信息
- 检查本地文件
- 如果本地文件比服务端文件小，断点续传，利用 HTTP 请求头的 Range 实现断点续传
- 如果比服务端文件大，重新下载
- 如果和服务端文件一样，下载完成

afn 默认超时时间是60s

## 97. 客户端的缓存机制

缓存分为：内存数据缓存、数据库缓存、文件缓存

- 每次想获取数据的时候，先检测内存中有无缓存
- 在检测本地有无缓存（数据库/文件）
- 最终发送网络请求
- 将服务端返回的数据进行缓存（内存、数据库、文件）

## 98. 数据存储方式

4种数据持久化：属性列表（plist）、对象归档、sqlite、Core Data

NSUserDefaults 用于存储配置信息

如何存储用户的一些敏感信息，如登录的 token？

使用 keychain 来存储，即钥匙串，使用 keychain 需要导入 Security 框架

使用 NSUserDefaults 时，如何处理布尔的默认值？（比如返回 NO，不知道是真的 NO，还是没有设置过的 NO）

```
if ([[NSUserDefaults standardUserDefaults] objectForKey:@"key"] == nil) {
 NSLog(@"没有设置过");
}
```

## 99. App 需要加载超大量的数据，给服务器发送请求，但服务器卡住了，如何解决？

- 设置请求超时
- 给用户提示请求超时
- 根据用户操作再次请求数据

## 100. 网络图片处理问题中怎么解决一个相同的网络地址重复请求的问题？

- 利用字典（图片地址为 key，下载操作为 value）

## 101. 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_async` 追加 block 到 global queue 中，这些 block 全部执行完毕，就会执行 main dispatch queue 中的结束处理的 block。

代码如下：

```
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
dispatch_group_t group = dispatch_group_create();
//下图片1,放线程组中
dispatch_group_async(group, queue, ^{
 NSURL *url = [NSURL URLWithString:@""];
 NSData *imgData1 = [NSData dataWithContentsOfURL:url];
 self.img1 = [UIImage imageWithData:imgData1];
});

//下载图片2
dispatch_group_async(group, queue, ^{
 NSURL *url = [NSURL URLWithString:@""];
 NSData *imgData2 = [NSData dataWithContentsOfURL:url];
 self.img2 = [UIImage imageWithData:imgData2];
});

//合并图片
dispatch_group_notify(group, queue, ^{
 NSLog(@"combie---%@", [NSThread currentThread]);
 UIGraphicsBeginImageContext(CGSizeMake(200, 200));
 [self.img1 drawInRect:CGRectMake(0, 0, 200, 100)];
 self.img1 = nil;
 [self.img2 drawInRect:CGRectMake(0, 100, 200, 100)];
 self.img2 = nil;
 UIImage *img = UIGraphicsGetImageFromCurrentImageContext();
 UIGraphicsEndImageContext();

 dispatch_async(dispatch_get_main_queue(), ^{
 NSLog(@"UI---%@", [NSThread currentThread]);
 //self.imageView.image = img;
 });
});
});
```

## 102. NSOperation、GCD、NSThread 的区别

NSOperation 与 GCD 的区别:

GCD:

- GCD 是 iOS4.0 推出的，纯 C
- GCD 是将任务 (block) 添加到队列（串行、并行、全局、主队列），并且以同步/异步的方式执行任务的函数
- GCD 提供 NSOperation 不具备的功能：
  - 一次性执行
  - 延迟执行
  - 调度组
  - GCD 是严格的队列，先进先出 FIFO

NSOperation:

- NSOperation 是 iOS2.0 推出的，iOS4.0 以后又重写的
- NSOperation 是将操作（异步任务）添加到队列（并发队列），就会执行指定的函数
- NSOperation 提供的方便操作：
  - 最大并发数
  - 队列的暂停和继续
  - 取消所有的操作
  - 指定操作之间的依赖关系，可以让异步任务同步执行
  - 可以利用 KVO 监听一个 operation 是否完成
  - 可以设置 operation 的优先级，能使同一个并行队列中的任务区分先后地执行
  - 对 NSOperation 继承，在这之上添加成员变量和成员方法，提高代码的复用度

GCD 与 NSThread 的区别:

- NSThread 使用 @selector 指定要执行的方法，代码分散
- GCD 通过 block 指定要执行的方法，代码集中
- GCD 不用管理线程的生命周期（创建、销毁、复用）
- 如果要开多个线程 NSThread 必须实例化多个线程对象
- NSThread 通过 performSelector 方法实现线程间通信

为什么要取消/恢复队列?

- 一般内存警告后取消队列中的操作
- 为了保证 scrollView 在滚动的时候流畅，通常在滚动开始时，暂停队列中的所有操作，滚动结束后，恢复操作

\_bridge : 用于 Foundation 和 Core Foundation 之间数据的桥接

### 103. 是否可以把比较耗时的操作放在 **NSNotificationCenter** 中

- 如果在异步线程发的通知，可以执行耗时的操作
- 如果在主线程发的通知，则不可以执行耗时的操作

## 备注：

本文 3-7、9-57 题均出自<https://github.com/ChenYilong/iOSInterviewQuestions>但是不是全部的拷贝，只是整理自我感觉重要的、关键点上的内容，如果看不懂，请移步原文详细阅读。

本文剩下的题目均是参考了网络各个大v的资源，外加自己学习总结，每个知识点均是查阅不下于 20 个相关内容的网页（可以说一个知识点基本要花费一天的时间吧），进行提炼整理，力尽准确、精要。

每个题目中感觉重要的都是用 其他色 做标注了

## 利用预渲染加速iOS设备的图像显示

```
1. static const CGSize imageSize = {100, 100};
2.
3. - (void)awakeFromNib {
4. if (!self.image) {
5. self.image = [UIImage imageNamed:@"random.jpg"];
6. if (NULL != UIGraphicsBeginImageContextWithOptions)
7.
8. UIGraphicsBeginImageContextWithOptions(imageSize, YES, 0
);
9. else
10. UIGraphicsBeginImageContext(imageSize);
11. [image drawInRect:imageRect];
12. self.image = UIGraphicsGetImageFromCurrentImageContext();
13. }
14. }
```

这里需要判断一下UIGraphicsBeginImageContextWithOptions是否为NULL，因为它是iOS 4.0才加入的。

由于JPEG图像是不透明的，所以第二个参数就设为YES。

第三个参数是缩放比例，iPhone 4是2.0，其他是1.0。虽然这里可以用[UIScreen mainScreen].scale来获取，但实际上设为0后，系统就会自动设置正确的比例了。

## TableView中实现平滑滚动延迟加载图片

利用CFRunLoopMode的特性，可以将图片的加载放到NSDefaultRunLoopMode的mode里，这样在滚动UITrackingRunLoopMode这个mode时不会被加载而影响到。主线程繁忙的时候performSelector:withObject:afterDelay:会延后执行，所以在发生触摸或是视图还在滚动时这个方法不会运行；

```
UIImage *downloadedImage = ...;
[self.avatarImageView performSelector:@selectorsetImage:)
 withObject:downloadedImage
 afterDelay:0
 inModes:@[NSDefaultRunLoopMode]];
```

注意：使用masonry进行label的多行显示设置时，主要是如下两个参数的设置

1、@property(nonatomic)CGFloat preferredMaxLayoutWidth  
2、- (void)setContentHuggingPriority:(UILayoutPriority)priority forAxis:(UILayoutConstraintAxis)axis

## iOS 7 Programming: *Pushing the Limits* Chapter 24 - Deep Objective-C

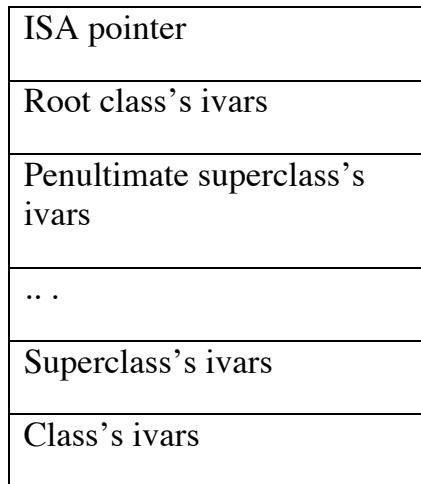


Figure 24-1 Layout of an Objective-C object

The Class structure contains a metaclass pointer (more on that in a moment), a superclass pointer, and data about the class. The data of particular interest are the name, ivars, methods, properties, and protocols. Don't worry too much about the internal structure of Class. There are public functions to access all the information you need.

**The Objective-C runtime is open source, so you can see exactly how it's implemented. Go to the Apple Open Source site ([www.opensource.apple.com](http://www.opensource.apple.com)), and look for the package objc in the Mac code. It isn't included in the iOS packages, but the Mac code is identical or very similar. These particular structures are defined in objc.h and objc-runtime-new.h. You will notice two definitions of many things in these files because of the switch from Objective-C 1.0 to Objective-C 2.0. Look for things marked "new" when there is a conflict.**

Class is itself much like an object. You can send messages to a Class instance—for example, when you call [Foo alloc]—so you need a place to store the list of class methods. They are stored in the metaclass, which is the isa pointer for a Class. It's extremely rare to need to access metaclasses, so I don't dwell on them here; see the "Further Reading" section at the end of this chapter for links to more information. Also see the section "How Message Passing Really Works," later in this chapter, for more information on message passing.

The superclass pointer creates the hierarchy of classes, and the list of ivars, methods, properties, and protocols defines what the class can do. An important point here is that the methods, properties, and protocols are all stored in the writable section of the class definition. They can be changed at runtime, and that's exactly how categories are implemented. Ivars are stored in the read-only section and cannot be modified (because that could impact existing instances). That's why categories cannot add ivars.

In the definition of `objc_object`, shown at the beginning of this section, notice that the `isa` pointer is not `const`. That is not an oversight. The class of an object can be changed at runtime. The superclass pointer of `Class` is also not `const`. The hierarchy can be modified. This topic is covered in more detail in the “ISA Swizzling” section later in this chapter.

**In iOS 7, the `objc_object` structure is more complicated. The `isa` pointer is deprecated, and in 64-bit iOS, it is not actually a pointer. You should not rely on the internal implementation details of this structure. See “[objc explain]: Non-pointer isa” in the “Further Reading” section for more details. ISA swizzling, as discussed in this chapter, is still supported in iOS 7.**

Now that you’ve seen the data structures underlying Objective-C objects, you next look at the kinds of functions you can use to inspect and manipulate them. These functions are written in C, and they use naming conventions somewhat similar to Core Foundation (see Chapter 10). All the functions shown here are public and are documented in the *Objective-C Runtime Reference*. The following is the simplest example:

```
#import <objc/objc-runtime.h>
...
const char *name = class_getName([NSObject class]);
printf("%s\n", name);
```

Runtime methods begin with the name of the thing they act upon, which is almost always also their first parameter. Because this example includes `get` rather than `copy`, you don’t own the memory that is returned to you and should not call `free`. The next example prints a list of the selectors that `NSObject` responds to. The call to `class_copyMethodList` returns a copied buffer that you must dispose of by using `free`.

### **PrintObjectMethods.m (Runtime)**

```
void PrintObjectMethods() { unsigned int count = 0; Method *methods =
class_copyMethodList([NSObject class],
&count);
for (unsigned int i = 0; i < count; ++i) {
 SEL sel = method_getName(methods[i]);
 const char *name = sel_getName(sel);
 printf("%s\n", name);
}
free(methods);
}
```

Because there is no reference counting (automatic or otherwise) in the runtime, there is no equivalent to retain or release. If you fetch a value with a function that includes the word copy, you should call free on it. If you use a function that does not include the word copy, you must not call free on it.

## Working with Methods and Properties

The Objective-C runtime defines several important types:

- Class—Defines an Objective-C class, as described in the previous section, “Understanding Classes and Objects.”
- Ivar—Defines an instance variable of an object, including its type and name.
- Protocol—Defines a formal protocol.
- objc\_property\_t—Defines a property. Its unusual name is probably to avoid colliding with user types defined in Objective-C 1.0 before properties existed.
- Method—Defines an object method or a class method. This provides the name of the method (its *selector*), the number and types of parameters it takes and its return type (collectively its *signature*), and a function pointer to its code (its *implementation*).
- SEL—Defines a selector. A selector is a unique identifier for the name of a method.
- IMP—Defines a method implementation. It’s just a pointer to a function that takes an object, a selector, and a variable list of other parameters (varargs), and returns an object: `typedef id (*IMP)(id, SEL, ...);`
- Now you use this knowledge to build your own simplistic *message dispatcher*. A message dispatcher maps selectors to function pointers and calls the referenced function. The heart of the Objective-C runtime is the message dispatcher `objc_msgSend`, which you learn much more about in the “How Message Passing Really Works” section. The sample `myMsgSend` is how `objc_msgSend` might be implemented if it needed to handle only the simplest cases. The following code is written in C just to prove that the Objective-C runtime is really just C. I added comments to demonstrate the equivalent Objective-C. **MyMsgSend.c (Runtime)**

```
static const void *myMsgSend(id receiver, const char *name) {
```

  - SEL selector = sel\_registerName(name);
  - IMP methodIMP =
  - class\_getMethodImplementation(object\_getClass(receiver),

```

 selector);
▪ return methodIMP(receiver, selector);
▪ } void RunMyMsgSend() {
▪ // NSObject *object = [[NSObject alloc] init];
▪ Class class = (Class)objc_getClass("NSObject");
▪ id object = class_createInstance(class, 0);
▪ myMsgSend(object, "init");
▪ // id description = [object description];
▪ id description = (id)myMsgSend(object, "description");
▪

// const char *cstr = [description UTF8String];
const char *cstr = myMsgSend(description, "UTF8String");
printf("%s\n", cstr);
}

```

You can use this same technique in Objective-C using methodForSelector: to avoid the complex message dispatch of objc\_msgSend. This approach makes sense only if you're going to call the same method thousands of times on an iPhone. On a Mac, you won't see much improvement unless you're calling the same method millions of times. Apple has highly optimized objc\_msgSend. But for very simple methods called many times, you may be able to improve performance 5% to 10% this way.

The following example demonstrates how to bypass objc\_msgSend and shows the performance impact: **FastCall.m (Runtime)**

```

const NSUInteger kTotalCount = 10000000;
typedef void (*voidIMP)(id, SEL, ...);
void FastCall() {
 NSMutableString *string = [NSMutableString string];
 NSTimeInterval totalTime = 0;
 NSDate *start = nil;
 NSUInteger count = 0;
 // With objc_msgSend
 start = [NSDate date];
 for (count = 0; count < kTotalCount; ++count) {
 [string setString:@"stuff"];
 }
 totalTime = -[start timeIntervalSinceNow];
 printf("w/ objc_msgSend = %f\n", totalTime);
 // Skip objc_msgSend.
 start = [NSDate date];
 SEL selector = @selector(setString:);
 voidIMP

```

```

setStringMethod = (voidIMP)[string methodForSelector:selector];
for (count = 0; count < kTotalCount; ++count) {
 setStringMethod(string, selector, @"stuff");
}
totalTime = -[start timeIntervalSinceNow];
printf("w/o objc_msgSend = %f\n", totalTime);
}

```

Be careful with this technique. If you use it incorrectly, it can actually be slower than using normal message dispatch. Because IMP returns an id, ARC will retain and later release the return value, even though this specific method returns nothing (see <http://openradar.appspot.com/10002493> for details). That overhead is more expensive than just using the normal messaging system. In some cases, the extra retain can cause a crash. That's why you have to add the extra voidIMP type. When you declare that the setStringMethod function pointer returns void, the compiler skips the retain.

The important take-away is that you need to do testing on anything you do to improve performance. Don't assume that bypassing the message dispatcher is going to be faster. In most cases, you can get much better and more reliable performance improvements by simply rewriting your code as a function rather than a method. And in the vast majority of cases, objc\_msgSend is the least of your performance overhead.

## Method Signatures and Invocations

NSInvocation is a traditional implementation of the Command pattern. It bundles a target, a selector, a method signature, and all the parameters into an object that can be stored and invoked at a later time. When the invocation is invoked, it sends the message, and the Objective-C runtime finds the correct method implementation to execute.

A *method implementation* (IMP) is a function pointer to a C function with the following signature: id function(id self, SEL \_cmd, ...)

Every method implementation takes two parameters: self and \_cmd. The first parameter is the self pointer that you're familiar with. The second parameter, \_cmd, is the selector that was sent to this object. This reserved symbol in the language is accessed exactly like self.

**Although the IMP typedef suggests that every Objective-C method returns an id, obviously many Objective-C methods return other types such as integers or floating-point numbers, and many Objective-C methods return nothing at all. The actual return type is defined by the message signature, discussed later in this section, not the IMP typedef.**

NSInvocation includes a target and a selector. A target is the object to send the message

to, and the selector is the message to send. A selector is roughly the name of a method. I say “roughly” because selectors don’t have to map exactly to methods. A selector is just a name, like `initWithBytes:length:encoding:`. A selector isn’t bound to any particular class or any particular return value or parameter types. It isn’t even specifically a class or instance selector. You can think of a selector as a string. So `-[NSString length]` and `-[NSData length]` have the same selector, even though they map to different methods’ implementations.

`NSInvocation` also includes a method signature (`NSMethodSignature`), which encapsulates the return type and the parameter types of a method. An `NSMethodSignature` does not include the name of a method, only the return value and the parameters. Here is how you can create one by hand:

```
NSMethodSignature *sig =
 [NSMethodSignature signatureWithObjCTypes:@"@:@:*"];
```

This code is the signature for `-[NSString initWithUTF8String:]`. The first character (@) indicates that the return value is an id. To the message passing system, all Objective-C objects are the same. It can’t tell the difference between an `NSString` and an `NSArray`. The next two characters (@:) indicate that this method takes an id and a SEL. As discussed previously, every Objective-C method takes these as its first two parameters. They’re implicitly passed as `self` and `_cmd`. Finally, the last character (\*) indicates that the first “real” parameter is a character string (`char*`).

**If you do work with type encoding directly, you can use `@encode(type)` to get the string that represents that type rather than hard-coding the letter. For example, `@encode(id)` is the string “@”.**

You should seldom call `signatureWithObjCTypes:`. I do it here only to show that it’s possible to build a method signature by hand. The way you generally get a method signature is to ask a class or instance for it. Before you do that, you need to consider whether the method is an instance method or a class method. The method `-init` is an instance method and is marked with a leading hyphen (-). The method `+alloc` is a class method and is marked with a leading plus (+). You can request instance method signatures from instances and class method signatures from classes by using `methodSignatureForSelector:`. If you want the instance method signature from a class, you use `instanceMethodSignatureForSelector:`. The following example demonstrates this for `+alloc` and `-init`:

```
SEL initSEL = @selector(init);
SEL allocSEL = @selector(alloc);
NSMethodSignature *initSig, *allocSig;
// Instance method signature from instance
initSig = [@"String" methodSignatureForSelector:initSEL];
// Instance method signature from class
```

```
initSig = [NSString instanceMethodSignatureForSelector:initSEL];
// Class method signature from class
allocSig = [NSString methodSignatureForSelector:allocSEL]; If you compare initSig and allocSig, you will discover that they are the same. They each take no additional parameters (besides self and _cmd) and return an id. This is all that matters to the message signature.
```

Now that you have a selector and a signature, you can combine them with a target and parameter values to construct an NSInvocation. An NSInvocation bundles everything needed to pass a message. Here is how you create an invocation of the message [set addObject:stuff] and invoke it:

```
NSMutableSet *set = [NSMutableSet set];
NSString *stuff = @"Stuff";
SEL selector = @selector(addObject:);
NSMethodSignature *sig = [set methodSignatureForSelector:selector];
NSInvocation *invocation =
 [NSInvocation invocationWithMethodSignature:sig];
[invocation setTarget:set]; [invocation setSelector:selector]; // Place the first argument at index 2. [invocation setArgument:&stuff atIndex:2]; [invocation invoke];
```

Note that the first argument is placed at index 2. As discussed previously, index 0 is the target (self), and index 1 is the selector (\_cmd). NSInvocation sets these automatically. Also note that you must pass a pointer to the argument, not the argument itself.

Invocations are extremely flexible, but they're not fast. Creating an invocation is hundreds of times slower than passing a message. Invoking an invocation is efficient, however, and invocations can be reused. They can be dispatched to different targets using invokeWithTarget: or setTarget:. You can also change their parameters between uses. Much of the cost of creating an invocation is in methodSignatureForSelector:, so caching this result can significantly improve performance.

Invocations do not retain their object arguments by default, nor do they make a copy of C string arguments. To store the invocation for later use, you call retainArguments on it. This method retains all object arguments and copies all C string arguments. When the invocation is released, it releases the objects and frees its copies of the C strings. Invocations do not provide any handling for pointers other than Objective-C objects and C strings. If you're passing raw pointers to an invocation, you're responsible for managing the memory yourself.

**If you use an invocation to create an NSTimer, such as by using timerWithTimeInterval: invocation:repeats:, the timer automatically calls retainArguments on the invocation.**

Invocations are a key part of the Objective-C message dispatching system, discussed later in this chapter. This integration with the message dispatching system makes them central to creating trampolines.

## Using Trampolines

A *trampoline* “bounces” a message from one object to another. This technique allows a proxy object to move messages to another thread, cache results, coalesce duplicate messages, or perform any other intermediary processing you’d like. Trampolines generally use `forwardInvocation:` to handle arbitrary messages. If an object does not respond to a selector, before Objective-C throws an error, it creates an `NSInvocation` and passes it to the object’s `forwardInvocation:`. You can use this trampoline to forward the message in any way that you’d like.

In this example, you create a trampoline called `RNObserverManager`. Any message sent to the trampoline is forwarded to registered observers that respond to that selector. This provides functionality similar to `NSNotification`, but is easier to use and faster if there are many observers.

The public interface for `RNObserverManager` is as follows: **RNObserverManager.h (ObserverTrampoline)**

```
#import <objc/runtime.h>
@interface RNObserverManager: NSObject
- (id)initWithProtocol:(Protocol *)protocol
 observers:(NSSet *)observers;
- (void)addObserver:(id)observer;
- (void)removeObserver:(id)observer;
@end
```

You initialize this trampoline with a protocol and an initial set of observers. You can then add or remove observers. Any method defined in the protocol is forwarded to all the current observers if they implement it.

Here is the skeleton implementation for `RNObserverManager`, without the trampoline piece. Everything should be fairly obvious.

**RNObserverManager.m (ObserverTrampoline)**

```
@interface RNObserverManager()
@property (nonatomic, readonly, strong)
 NSMutableSet *observers;
@property (nonatomic, readonly, strong) Protocol *protocol;
@end
```

```

@implementation RNObservableManager
- (id)initWithProtocol:(Protocol *)protocol
 observers:(NSSet *)observers {
 if ((self = [super init])) {
 _protocol = protocol;
 _observers = [NSMutableSet setWithSet:observers];
 }
 return self;
}

- (void)addObserver:(id)observer {
 NSAssert([observer conformsToProtocol:self.protocol],
 @"Observer must conform to protocol.");
 [self.observers addObject:observer];
}

- (void)removeObserver:(id)observer {
 [self.observers removeObject:observer];
}
@end

```

Now you override `methodSignatureForSelector:`. The Objective-C message dispatcher uses this method to construct an `NSInvocation` for unknown selectors. You override it to return method signatures for methods defined in `protocol`, using `protocol_getMethodDescription`. You need to get the method signature from the `protocol` rather than from the `observers` because the method may be optional, and the `observers` might not implement it.

```

- (NSMethodSignature *)methodSignatureForSelector:(SEL)sel
{
 // Check the trampoline itself
 NSMethodSignature *
 result = [super methodSignatureForSelector:sel];
 if (result) {
 return result;
 }
 // Look for a required method
 struct objc_method_description desc =
 protocol_getMethodDescription(self.protocol,
 if (desc.name == NULL) {
 sel, YES, YES);

 // Couldn't find it. Maybe it's optional
 desc = protocol_getMethodDescription(self.protocol, sel, NO, YES);

```

```

}

if (desc.name == NULL) {
 // Couldn't find it. Raise NSInvalidArgumentException
 [self doesNotRecognizeSelector: sel];
return nil; }

return [NSMethodSignature
 signatureWithObjCTypes:desc.types];
} Finally, you override forwardInvocation: to forward the invocation to the observers
that respond to the

```

selector:

```

- (void)forwardInvocation:(NSInvocation *)invocation {
 SEL selector = [invocation selector];
 for (id responder in self.observers) {
 if ([responder respondsToSelector:selector]) {
 [invocation setTarget:responder];
 [invocation invoke];
 }
 }
}

```

To use this trampoline, you create an instance, set the observers, and then send messages to it as the following code shows. Variables that hold a trampoline should generally be of type id so that you can send any message to it without generating a compiler warning.

```

@protocol MyProtocol <NSObject>
- (void)doSomething;
@end
...
id observerManager = [[RNObserverManager alloc]
 initWithProtocol:@protocol(MyProtocol)
 observers:observers];
[observerManager doSomething];

```

Passing a message to this trampoline is similar to posting a notification. You can use this technique to solve a variety of problems. For example, you can create a proxy trampoline that forwards all messages to the main thread as shown here:

### RNMainThreadTrampoline.h (ObserverTrampoline)

```

@interface RNMainThreadTrampoline : NSObject
@property (nonatomic, readwrite, strong) id target;
- (id)initWithTarget:(id)aTarget;
@end

```

### RNMainThreadTrampoline.m (ObserverTrampoline)

```
@implementation RNMainThreadTrampoline
- (id)initWithTarget:(id)aTarget {
 if ((self = [super init])) {
 _target = aTarget;
 }
 return self;
}
- (NSMethodSignature *)methodSignatureForSelector:(SEL)sel
{
 return [self.target methodSignatureForSelector:sel];
}

- (void)forwardInvocation:(NSInvocation *)invocation {
 [invocation setTarget:self.target];
 [invocation retainArguments];
 [invocation performSelectorOnMainThread:@selector(invoker)
} @end

withObject:nil
waitForDone:NO];
forwardInvocation: can transparently coalesce duplicate messages, add logging, forward
messages to other machines, and perform a wide variety of other functions.
```

## How Message Passing Really Works

As demonstrated in the “Working with Methods and Properties” section, earlier in this chapter, calling a method in Objective-C eventually translates into calling a method implementation function pointer and passing it an object pointer, a selector, and a set of function parameters. Like the sample myMsgSend, every Objective-C message expression is converted into a call to objc\_msgSend (or a closely related function; I get to that in “The Flavors of objc\_msgSend,” later in this chapter). However, objc\_msgSend is much more powerful than myMsgSend. Here is how it works:

- 1. Check whether the receiver is nil. If so, call the nil-handler. This is really obscure, undocumented, unsupported, and difficult to make useful. The default is to do nothing, and I don’t go into more detail here. See the “Further Reading” section for more information.**
- 2. In a garbage-collected environment (which iOS doesn’t support, but I include for completeness), check for one of the short-circuited selectors (retain, release, autorelease, retainCount), and if it matches, return self. Yes, that means**

**retainCount returns self in a garbage-collected environment. You shouldn't have been calling it anyway.**

- 3. Check the class's cache to see if it's already worked out this method implementation. If so, call it.**
- 4. Compare the requested selector to the selectors defined in the class. If the selector is found, call its method implementation.**
- 5. Compare the requested selector to the selectors defined in the superclass, and then its superclass, and so on. If the selector is found, call its method implementation.**
- 6. Call `resolveInstanceMethod:` (or `resolveClassMethod:`). If it returns YES, start over. The object is promising that the selector will resolve this time, generally because it has called `class_addMethod`.**
- 7. Call `forwardingTargetForSelector:`. If it returns non-nil, send the message to the returned object. Don't return self here. That would be an infinite loop.**
- 8. Call `methodSignatureForSelector:`, and if it returns non-nil, create an `NSInvocation` and pass it to `forwardInvocation:`.**
- 9. Call `doesNotRecognizeSelector:`. The default implementation of this just throws an exception.**

Some changes have been made to how this process works in the 64-bit version of iOS 7. At the time of writing, these changes are not well documented. See *Hamster Emporium* in the “Further Reading” section at the end of this chapter for more information.

## Dynamic Implementations

The first interesting thing you can do with message dispatch is provide an implementation at runtime by using `resolveInstanceMethod:` and `resolveClassMethod:`. These methods are a common way @ dynamic synthesis is handled. When you declare a property to be @dynamic, you're promising the compiler that an implementation will be available at runtime, even though the compiler can't find one now. Using @ dynamic prevents it from automatically synthesizing an ivar.

Here's an example of how to use this implementation to dynamically create getters and

setters for properties stored in an NSMutableDictionary:

### **Person.h (Person)**

```
@interface Person : NSObject
@property (copy) NSString *givenName;
@property (copy) NSString *surname;
@end
```

### **Person.m (Person)**

```
@interface Person ()
@property (strong) NSMutableDictionary *properties;
@end
@implementation Person
@dynamic givenName, surname;

- (id)init {
 if ((self = [super init])) {
 _properties = [[NSMutableDictionary alloc] init];
 }
 return self;
}
static id propertyIMP(id self, SEL _cmd) {
 return [[self properties] valueForKey:
}

NSStringFromSelector(_cmd)];
static void setPropertyIMP(id self, SEL _cmd, id aValue) {
 id value = [aValue copy];
 NSMutableString *key =
 [NSStringFromSelector(_cmd) mutableCopy];
 // Delete "set" and ":" and lowercase first letter
 [key deleteCharactersInRange:NSMakeRange(0, 3)];
 [key deleteCharactersInRange:
 NSMakeRange([key length] - 1, 1)];
 NSString *firstChar = [key substringToIndex:1];
 [key replaceCharactersInRange:NSMakeRange(0, 1)
 withString:[firstChar lowercaseString]];
 [[self properties] setValue:value forKey:key];
}
+ (BOOL)resolveInstanceMethod:(SEL)aSEL {
 if ([NSStringFromSelector(aSEL) hasPrefix:@"set"]) {
 class_addMethod([self class], aSEL,
 (IMP)setPropertyIMP, "v@:@");
```

```

} else {

 class_addMethod([self class], aSEL,
 (IMP)propertyIMP, "@@:");

}

return YES; }

@end

```

### **main.m (Person)**

```

int main(int argc, char *argv[]) {
 @autoreleasepool {
 Person *person = [[Person alloc] init];
 [person setGivenName:@"Bob"];
 [person setSurname:@"Jones"];
 NSLog(@"%@", [person givenName], [person surname]);
 }
}

```

In this example, you use `propertyIMP` as the generic getter and `setPropertyIMP` as the generic setter. Note how these functions make use of the selector to determine the name of the property. Also note that `resolveInstanceMethod:` assumes that any unrecognized selector is a property setter or getter. In many cases, this is okay. You still get compiler warnings if you pass unknown methods like this:

```
[person addObject:@"Bob"];
```

But if you pass unknown methods this way, you get a slightly surprising result:

```

NSArray *persons = [NSArray arrayWithObject:person];
id object = [persons objectAtIndex:0];
[object addObject:@"Bob"];

```

You get no compiler warning because you can send any message to `id`. And you don't get a runtime error either. You just retrieve the key `addObject:` (including the colon) from the properties dictionary and do nothing with it. This kind of bug can be difficult to track down, and you may want to add additional checking in `resolveInstanceMethod:` to guard against it. But the approach is extremely powerful. Although dynamic getters and setters are the most common use of `resolveInstanceMethod:`, you can also use it to dynamically load code in environments that allow dynamic loading. iOS doesn't allow this approach, but on the Mac you can use `resolveInstanceMethod:` to avoid loading entire libraries until the first time one of the library's classes is accessed. This approach can be useful for large but rarely used classes.

## Fast Forwarding

The runtime gives you one more fast option before falling back to the standard forwarding system. You can implement `forwardingTargetForSelector:` and return another object to pass the message to. This option is particularly useful for proxy objects or objects that add functionality to another object. The `CacheProxy` example demonstrates an object that caches the getters and setters for another object.

### **CacheProxy.h (Person)**

```
@interface CacheProxy : NSProxy
- (id)initWithObject:(id)anObject
@end

properties:(NSArray *)properties;
@interface CacheProxy ()
@property (readonly, strong) id object;
@property (readonly, strong)
@end
```

`NSMutableDictionary *valueForProperty;`

`CacheProxy` is a subclass of `NSProxy` rather than `NSObject`. `NSProxy` is a very thin root class designed for classes that forward most of their methods, particularly classes that forward their methods to objects hosted on another machine or on another thread. It's not a subclass of `NSObject`, but it does conform to the `<NSObject>` protocol. The `NSObject` class implements dozens of methods that might be very hard to proxy. For example, methods that require the local run loop, such as `performSelector:withObject:afterDelay:`, might not make sense for a proxied object. `NSProxy` avoids most of these methods.

To implement a subclass of `NSProxy`, you must override `methodSignatureForSelector:` and `forwardInvocation:`. These throw exceptions if they're called otherwise.

First, you need to create the getter and setter implementations, as in the `Person` example. In this case, if the value is not found in the local cache dictionary, you forward the request to the proxied object, as shown here:

### **CacheProxy.m (Person)**

```
@implementation CacheProxy
// setFoo: => foo
static NSString *propertyNameForSetter(SEL selector) {
 NSMutableString *name =
```

```

[NSStringFromSelector(selector) mutableCopy];
[name deleteCharactersInRange:NSMakeRange(0, 3)];
[name deleteCharactersInRange:
 NSMakeRange([name length] - 1, 1)];
NSString *firstChar = [name substringToIndex:1];
[name replaceCharactersInRange:NSMakeRange(0, 1)
 withString:[firstChar lowercaseString]];
// foo => setFoo:
static SEL setterForPropertyName(NSString *property) {
 NSMutableString *name = [property mutableCopy];
 NSString *firstChar = [name substringToIndex:1];
 [name replaceCharactersInRange:NSMakeRange(0, 1)
 withString:[firstChar uppercaseString]];
 [name insertString:@"set" atIndex:0];
 [name appendString:@":"];
 return NSSelectorFromString(name);
}
// Getter implementation
static id propertyIMP(id self, SEL _cmd) {
 NSString *propertyName = NSStringFromSelector(_cmd);
 id value = [[self valueForProperty] valueForKey:propertyName];
 if (value == [NSNull null]) {
 return nil;
 }
 if (value) {
 return value;
 }
 value = [[self object] valueForKey:propertyName];
 [[self valueForProperty] setValue:value
 forKey:propertyName];
 return value;
}
// Setter implementation
static void setPropertyIMP(id self, SEL _cmd, id aValue) {
 id value = [aValue copy];
 NSString *propertyName = propertyNameForSetter(_cmd);
 [[self valueForProperty] setValue:(value != nil ? value :
 [NSNull null])
 forKey:propertyName];
}

```

```

[[self object] setValue:value forKey:propertyName];
}

Note the use of [NSNull null] to manage nil values. You cannot store nil in an
NSDictionary. In the next block of code, you synthesize accessors for the properties
requested. All other methods are forwarded to the proxied object.

- (id)initWithObject:(id)anObject
 properties:(NSArray *)properties {
 _object = anObject;
 _valueForProperty = [[NSMutableDictionary alloc] init];
 for (NSString *property in properties) {
 // Synthesize a getter
 class_addMethod([self class],
 NSSelectorFromString(property),
 (IMP)propertyIMP,
 "@:@");
 // Synthesize a setter
 class_addMethod([self class],
 }
 return self;
}
setterForPropertyName(property),
(IMP)setPropertyIMP,
"v@:@");

```

The next block of code overrides methods that are implemented by NSProxy. Because NSProxy has default implementations for these methods, they aren't automatically forwarded by forwardingTargetForSelector:

```

- (NSString *)description {
 return [NSString stringWithFormat:@"%@ (%@)",
}

[super description], self.object];
- (BOOL)isEqual:(id)anObject {
 return [self.object isEqual:anObject];
}
- (NSUInteger)hash {
 return [self.object hash];
}
- (BOOL)respondsToSelector:(SEL)aSelector {
 return [self.object respondsToSelector:aSelector];
}

```

```
- (BOOL)isKindOfClass:(Class)aClass {
 return [self.object isKindOfClass:aClass];
}
```

Finally, you implement the forwarding methods. Each of them simply passes unknown messages to the proxied object. See the “Method Signatures and Invocations” section, earlier in this chapter, for more details.

Whenever an unknown selector is sent to CacheProxy, `objc_msgSend` calls `forwardingTargetForSelector:`. If it returns an object, then `objc_msgSend` tries to send the selector to that object. This process is called *fast forwarding*. In this example, CacheProxy sends all unknown selectors to the proxied object. If the proxied object doesn’t appear to respond to that selector, then `objc_msgSend` falls back to normal forwarding by calling `methodSignatureForSelector:` and `forwardInvocation:`. This topic is covered in the next section, “Normal Forwarding.” CacheProxy forwards these requests to the proxied object as well. The rest of the CacheProxy methods are as follows:

```
- (id)forwardingTargetForSelector:(SEL)selector {
 return self.object;
}
- (NSMethodSignature *)methodSignatureForSelector:(SEL)sel
{
 return [self.object methodSignatureForSelector:sel];
}
- (void)forwardInvocation:(NSInvocation *)anInvocation {
 [anInvocation setTarget:self.object];
 [anInvocation invoke];
}
@end
```

## Normal Forwarding

After trying everything described in the previous sections, the runtime tries the slowest of the forwarding options: `forwardInvocation:`. This option can be tens to hundreds of times slower than the mechanisms covered in the previous sections, but it’s also the most flexible. You are passed an `NSInvocation`, which bundles the target, the selector, the method signature, and the arguments. You may then do whatever you want with it. The most common thing to do is to change the target and invoke it, as demonstrated in the CacheProxy example.

If you implement `forwardInvocation:`, you also must implement `methodSignatureForSelector:`. That’s how the runtime determines the method signature for the `NSInvocation` it passes to you. Often `methodSignatureForSelector:` is implemented by asking the object you’re forwarding to.

There is a special limitation of `forwardInvocation:`. It doesn't support *vararg methods*. These methods, such as `arrayWithObjects:`, take a variable number of arguments. The runtime has no way to automatically construct an `NSInvocation` for this kind of method because it has no way to know how many parameters will be passed. Although many *vararg* methods terminate their parameter list with a `nil`, that is not required or universal (`stringWithFormat:` does not), so determining the length of the parameter list is implementation-dependent. The other forwarding methods, such as `Fast Forwarding`, do support *vararg* methods.

Even though `forwardInvocation:` returns nothing itself, the runtime system returns the result of the `NSInvocation` to the original caller. It does so by calling `getReturnValue:` on the `NSInvocation` after `forwardInvocation:` returns. Generally, you call `invoke`, and the `NSInvocation` stores the return value of the called method, but that isn't required. You could call `setReturnValue:` yourself and return. This capability can be handy for caching expensive calls.

## Forwarding Failure

Okay, so you've made it through the entire message resolution chain and haven't found a suitable method. What happens now? Technically, `forwardInvocation:` is the last link in the chain. If it does nothing, then nothing happens. You can use it to swallow certain methods if you want to. But the default implementation of `forwardInvocation:` does do something. It calls `doesNotRecognizeSelector:`. The default implementation of that method just raises an `NSInvalidArgumentException`, but you could override this behavior. Doing so is not particularly useful because this method is required to raise `NSInvalidArgumentException` (either directly or by calling `super`), but it's legal.

You can also call `doesNotRecognizeSelector:` yourself in some situations. For example, if you do not want anyone to call your `init`, you could override it like this:

```
- (id)init {
 [self doesNotRecognizeSelector:_cmd];
}
```

This makes calling `init` a runtime error. Personally, I often do it this way instead:

```
- (id)init {
 NSAssert(NO, @"Use -initWithOptions:");
 return nil; }
```

That way, it crashes when I'm developing, but not in the field. Which form you prefer is somewhat a matter of taste.

You should, of course, call `doesNotRecognizeSelector:` in methods such as

`forwardInvocation`: when the method is unknown. Don't just return unless you specifically mean to swallow the error. That can lead to very challenging bugs.

## The Flavors of `objc_msgSend`

In this chapter, I've referred generally to `objc_msgSend`, but there are several related functions: `objc_msgSend_fpret`, `objc_msgSend_stret`, `objc_msgSendSuper`, and `objc_msgSendSuper_stret`. The `SendSuper` form is obvious. It sends the message to the superclass. The `stret` forms handle most cases when you return a struct. This is for processor-specific reasons related to how arguments are passed and returned in registers versus on the stack. I don't go into all the details here, but if you're interested in this kind of low-level detail, you should read *Hamster Emporium* (see "Further Reading"). Similarly, the `fpret` form handles the case when you return a floating-point value on an Intel processor. It isn't used on the ARM-based processors that iOS runs on, but it is used when you compile for the simulator. There is no `objc_msgSendSuper_fpret` because the floating-point return matters only when the object you're messaging is nil (on an Intel processor), and that's not possible when you message super.

The point of all this is not, obviously, to address the processor-specific intricacies of message passing. If you're interested in that, read *Hamster Emporium*. The point is that not all message passing is handled by `objc_msgSend`, and you cannot use `objc_msgSend` to handle any arbitrary method call. In particular, you cannot return a "large" struct with `objc_msgSend` on any processor, and you cannot safely return a floating point with `objc_msgSend` on Intel processors (such as when compiling for the simulator). This generally translates into: Be careful when you try to bypass the compiler by calling `objc_msgSend` by hand.

## Method Swizzling

In Objective-C, *swizzling* refers to transparently swapping one thing for another. Generally, it means replacing methods at runtime. Using method swizzling, you can modify the behavior of objects that you do not have the code for, including system objects. In practice, swizzling is fairly straightforward, but it can be a little confusing to read. For this example, you add logging every time you add an observer to `NSNotificationCenter`.

**Since iOS 4.0, Apple has rejected some applications from the App Store for using this technique.**

First, you add a category on `NSObject` to simplify swizzling: **RNSwizzle.h (MethodSwizzle)**

```
@interface NSObject (RNSwizzle)
```

```
+ (IMP)swizzleSelector:(SEL)origSelector
@end
```

```
withIMP:(IMP)newIMP;
```

### RNSwizzle.m (MethodSwizzle)

```
@implementation NSObject (RNSwizzle)
+ (IMP)swizzleSelector:(SEL)origSelector
 withIMP:(IMP)newIMP {
 Class class = [self class];
 Method origMethod = class_getInstanceMethod(class,
 origSelector);
 IMP origIMP = method_getImplementation(origMethod);
 if(!class_addMethod(self, origSelector, newIMP,
 method_getTypeEncoding(origMethod))) {
 method_setImplementation(origMethod, newIMP);
 }
 return origIMP;
}
@end
```

Now, look at this code in more detail. You pass a selector and a function pointer (IMP) to this method. What you want to do is to swap the current implementation of that method with the new implementation and return a pointer to the old implementation so you can call it later. You have to consider three cases: The class may implement this method directly, the method may be implemented by one of the superclass hierarchy, or the method may not be implemented at all. The call to `class_getInstanceMethod` returns an IMP if either the class or one of its superclasses implements the method; otherwise, it returns NULL.

If the method was not implemented at all, or if it's implemented by a superclass, you need to add the method with `class_addMethod`. This process is identical to overriding the method normally. If `class_addMethod` fails, you know the class directly implemented the method you're swizzling. You instead need to replace the old implementation with the new implementation using `method_setImplementation`.

When you're done, you return the original IMP, and it's your caller's problem to make use of it. You do that in a category on the target class, `NSNotificationCenter`, as shown in the following code:

### NSNotificationCenter+RNSwizzle.h (MethodSwizzle)

```
@interfaceNSNotificationCenter (RNSwizzle)
+ (void)swizzleAddObserver;
```

```
@end
```

### NSNotificationCenter+RNSwizzle.m (MethodSwizzle)

```
@implementation NSNotificationCenter (RNSwizzle)
typedef void (*voidIMP)(id, SEL, ...);
static voidIMP sOrigAddObserver = NULL;
static void MYAddObserver(id self, SEL _cmd, id observer,
 SEL selector,
 NSString *name,
 id object) {
 NSLog(@"Adding observer: %@", observer);
 // Call the old implementation
 NSAssert(sOrigAddObserver,
 @"Original addObserver: method not found.");
 if (sOrigAddObserver) {
 sOrigAddObserver(self, _cmd, observer, selector, name,
 object);
 }
}

+ (void)swizzleAddObserver {
 NSAssert(!sOrigAddObserver,
 @"Only call swizzleAddObserver once.");
 SEL sel = @selector(addObserver:selector:name:object:);
 sOrigAddObserver = (void *)[self swizzleSelector:sel
} @end
```

withIMP:(IMP)MYAddObserver];

You call `swizzleSelector:withIMP:`, passing a function pointer to your new implementation. Notice that this is a function, not a method, but as covered in “How Message Passing Really Works” earlier in this chapter, a method implementation is just a function that accepts an object pointer and a selector. Notice also the `voidIMP` type. See the section “Working with Methods and Properties,” earlier in this chapter, for how this interacts with ARC. Without that, ARC tries to retain the nonexistent return value, causing a crash.

You then save the original implementation in a static variable, `sOrigAddObserver`. In the new implementation, you add the functionality you want and then call the original function directly.

Finally, you need to actually perform the swizzle somewhere near the beginning of your program:

```
[NSNotificationCenter swizzleAddObserver];
Some people suggest doing the swizzle in a +load method in the category. That makes it
```

much more transparent, which is why I don't recommend that approach. Method swizzling can lead to very surprising behaviors. Using `+load` means that just linking the category implementation causes it to be applied. I've personally encountered this situation when bringing old code into a new project. One of the debugging assistants from the old project had this kind of auto-load trick. It wasn't being compiled in the old project; it just happened to be in the sources directory. When I used "add folder" in Xcode, even though I didn't make any other changes to the project, the debug code started running. Suddenly, the new project had massive debug files showing up on customer machines, and it was very difficult to figure out where they were coming from. So my experience is that using `+load` for this task can be dangerous. However, it's very convenient and automatically ensures that it's called only once. Use your best judgment here.

Method swizzling is a powerful technique and can lead to bugs that are hard to track down. It allows you to modify the behaviors of Apple-provided frameworks, but that can make your code much more dependent on implementation details. It always makes the code more difficult to understand. I typically do not recommend using method swizzling for production code except as a last resort, but it's extremely useful for debugging, performance profiling, and exploring Apple's frameworks.

**There are several other method swizzling techniques. The most common is to use `method_exchangeImplementations` to swap one implementation for another. That approach modifies the selector, which can sometimes break things. It also creates an awkward pseudo-recursive call in the source code that is very misleading to the reader. This is why I recommend using the function pointer approach detailed here. For more information on swizzling techniques, see the "Further Reading" section.**

## ISA Swizzling

As discussed in the "Understanding Classes and Objects" section, earlier in this chapter, an object's ISA pointer defines its class. And, as discussed in "How Message Passing Really Works" (also earlier in this chapter), message dispatch is determined at runtime by consulting the list of methods defined on that class. So far, you've learned ways of modifying the list of methods, but it's also possible to modify an object's class (ISA swizzling). The next example demonstrates ISA swizzling to achieve the same `NSNotificationCenter` logging you did in the previous section, "Method Swizzling."

First, you create a normal subclass of `NSNotificationCenter`, which you use to replace the default `NSNotificationCenter`:

### **MYNotificationCenter.h (ISASwizzle)**

```
@interface MYNotificationCenter : NSNotificationCenter
```

```

// You MUST NOT define any ivars or synthesized properties here.
@end
@implementation MYNotificationCenter
- (void)addObserver:(id)observer selector:(SEL)aSelector
 name:(NSString *)aName object:(id)anObject
{
 NSLog(@"Adding observer: %@", observer);
 [super addObserver:observer selector:aSelector name:aName
} @end

object:anObject];

```

There's nothing really special about this subclass. You could +alloc it normally and use it, but you want to replace the default NSNotificationCenter with your class.

Next, you create a category on NSObject to simplify changing the class:

### **NSObject+SetClass.h (ISASwizzle)**

```

@interface NSObject (SetClass)
- (void)setClass:(Class)aClass;
@end

```

### **NSObject+SetClass.m (ISASwizzle)**

```

@implementation NSObject (SetClass)
- (void)setClass:(Class)aClass {
 NSAssert(
 class_getInstanceSize([self class]) ==
 class_getInstanceSize(aClass),
 @"Classes must be the same size to swizzle.");
 object_setClass(self, aClass);
}

```

@end Now, you can change the class of the default NSNotificationCenter:

```

id nc = [NSNotificationCenter defaultCenter];
[nc setClass:[MYNotificationCenter class]];

```

The most important point to note here is that the size of MYNotificationCenter must be the same as the size of NSNotificationCenter. In other words, you can't declare any ivars or synthesized properties (synthesized properties are just ivars in disguise). Remember, the object you're swizzling has already been allocated. If you added ivars, they would point to offsets beyond the end of that allocated memory. This has a pretty good chance of overwriting the isa pointer of some other object that just happens to be after this object in memory. In all likelihood, when you finally do crash, the other (innocent) object will appear to be the problem. Tracking down this bug is incredibly difficult, which is why I take the trouble of building a category to wrap object\_setClass. I believe it's worth the

effort to include the NSAssert ensuring the two classes are the same size.

After you perform the swizzle, the impacted object is identical to a normally created subclass. This means that it's very low risk for classes that are designed to be subclassed. As discussed in Chapter 22, key-value observing (KVO) is implemented with ISA swizzling. This way, the system frameworks can inject notification code into your classes, just as you can inject code into the system frameworks.

## Method Swizzling Versus ISA Swizzling

Both method and ISA swizzling are powerful techniques that can cause a lot of problems if used incorrectly. In my experience, ISA swizzling is a better technique and should be used when possible because it impacts only the specific objects you target, rather than all instances of the class. However, sometimes your goal is to impact every instance of the class, so method swizzling is the only option. The following list defines the differences between method swizzling and ISA swizzling:

### **Method Swizzling**

- Impacts every instance of the class.
- Is highly transparent. All objects retain their class.
- Requires unusual implementations of override methods.

### **ISA Swizzling**

- Impacts only the targeted instance.
- Changes object class (although this can be hidden by overriding class).
- Allows override methods to be written with standard subclass techniques.