

Q & A — ObjectC

分类和扩展有什么区别？可以分别用来做什么？分类有哪些局限性？分类的结构体里面有哪些成员？

- 分类 (Category) 在不知道源码的情况下，为类添加扩展的方法、属性、协议，通过分类可以将庞大一个类的方法进行划分,从而便于代码的日后的维护、更新以及提高代码的阅读性
- 类扩展 (Extension) 是 Category 的一个特例，有时候也被称为匿名分类，它的作用是为一个类添加一些私有的成员变量和方法，使用类扩展的方法必须在 @implementation 中实现，否则编译会报错。实际上在.m 里面的 @interface 就是扩展。

区别：

1. 分类是在运行时，通过 runtime 把分类信息添加到类中的，而扩展是在编译时就添加了
2. 分类声明的属性，只会生成 getter/setter 方法的声明，不会自动生成成员变量 ivar 和 getter/setter 的实现，而扩展会。
3. 分类不可用为类添加实例变量，而扩展可以
4. 分类可以添加方法的实现，而扩展不可以，只能声明

局限性：

1. 分类无法直接添加成员变量，需要通过关联类来实现
2. 分类方法和原类方法重名时候，原类方法被覆盖
3. 多个分类方法重名时候，会调用最后编译的那个分类里的方法实现

成员：

1. instanceProperties、classProperties
2. instanceMethod、classMethod
3. protocols
4. name、class

```
struct category_t {  
    const char *name;      //名字  
    classref_t cls;        //类的引用  
    struct method_list_t *instanceMethods; //实例  
方法列表  
    struct method_list_t *classMethods; //类方法列
```

表

```
    struct protocol_list_t *protocols; // 协议列表
    struct property_list_t
*instanceProperties; // 实例属性列表
    // 此属性不一定真正的存在
    struct property_list_t *_classProperties; // 
类属性列表
};
```

讲一下 **atomic** 的实现机制；为什么不能保证绝对的线程安全（最好可以结合场景来说）

机制：

对 set/get 方法，进行加锁设置，使用 os_unfair_lock

为什么不能绝对安全：

1. 只是对 set/get 方法加锁，但是对方法没有加锁，如：
NSMutableArray 的方法调用
2. 重写 set/get 方法后，就无锁了

被 **weak** 修饰的对象在被释放的时候会发生什么？
是如何实现的？知道 **sideTable** 么？里面的结构可以画出来么

发生什么

被 weak 修饰的对象在被释放的时候，会把 weak 指针自动置位 nil

实现：

runTime 会把对 weak 修饰的对象放到一个全局的哈希表中，用 weak 修饰的对象的内存地址为 key，weak 指针为值，在对象进行销毁时，用通过自身地址去哈希表中查找到所有指向此对象的 weak 指针，并把所有的 weak 指针置位 nil

SideTable：

存放引用计数和 weak 指针

```
struct SideTable {
    spinlock_t slock;           // 操作 SideTable 时用
到的锁
```

```
    RefcountMap refcnts;           //引用计数器的值的散列表
    weak_table_t weak_table;     //存放weak指针的哈希表
};


```

关联对象有什么应用，系统如何管理关联对象？其被释放的时候需要手动将所有的关联对象的指针置空么？

应用

给分类添加实例变量

管理：

通过，AssociationManager 进行管理

AssociationsHashMap: key: 对象的内存地址 value:

ObjectAssociationMap

ObjectAssociationMap: key: 关联对象的key value:

ObjcAssociation

ObjcAssociation: 值 和 内存管理策略

手动置空

不需要，对象在 dealloc，会自动处理。

KVO的底层实现？如何取消系统默认的KVO并手动触发（给KVO的触发设定条件：改变的值符合某个条件时再触发KVO）？

实现：

类的属性被添加监听后，会在运行时，动态创建一个类的子类，并把实例对象的isa指向这个子类。在子类中，重写了set方法的实现：

willChangeValueForKey;

set;

didChangeValueForKey;

手动出发：

重写 automaticallyNotifiesObserversForKey

重写 set 方法，并调用 willChangeValueForKey 和

didChangeValueForKey

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString *)key {
    if ([key isEqualToString:@"age"]) {
        return NO;
    }
    return [super
automaticallyNotifiesObserversForKey:key];
}

- (void)setAge:(NSInteger)age {
    if (age > 18 ) {
        [self willChangeValueForKey:@"age"];
        _age = age;
        [self didChangeValueForKey:@"age"];
    }else {
        _age = age;
    }
}
```

**AutoreleasePool所使用的数据结构是什么?
AutoreleasePoolPage结构体了解么?**

AutoreleasePool是由多个AutoreleasePoolPage以双向链表的形式连接起来的

class_ro_t和class_rw_t的区别?

Class_ro_t: 存放最一开始的类信息, 即编译时候就确定的内容
Class_rw_t: 存放添加分类信息之后的内容

iOS中内省的几个方法? class方法和objc_getClass方法有什么区别?

方法:

- isMemberOfClass //对象是否是某个类型的对象
- isKindOfClass //对象是否是某个类型或某个类型子类的对象
- isSubclassOfClass //某个类对象是否是另一个类型的子类
- isAncestorOfClass //某个类对象是否是另一个类型的父类
- respondsToSelector //是否能响应某个方法
- conformsToProtocol //是否遵循某个协议

区别：

1. 实例 (Class) 返回类对象，类 (class) 返回还是类对象
2. 而 objc_getClass (类对象) 返回元类，objc_getClass (实例对象) 返回类对象

[self class] 和 [super class]

其实 super 是一个 Magic Keyword，它本质是一个编译器指示符，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。

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。

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，

在运行时创建类的方法 objc_allocateClassPair 的方法名尾部为什么是 pair（成对的意思）？

会生成，类对象和元类对象

```
Class objc_allocateClassPair(Class superclass,
const char *name,
size_t extraBytes){
    ...省略了部分代码
    //生成一个类对象
    cls = alloc_class_for_subclass(superclass,
extraBytes);
    //生成一个类对象元类对象
    meta = alloc_class_for_subclass(superclass,
extraBytes);

objc_initializeClassPair_internal(superclass,
name, cls, meta);
return cls;
}
```

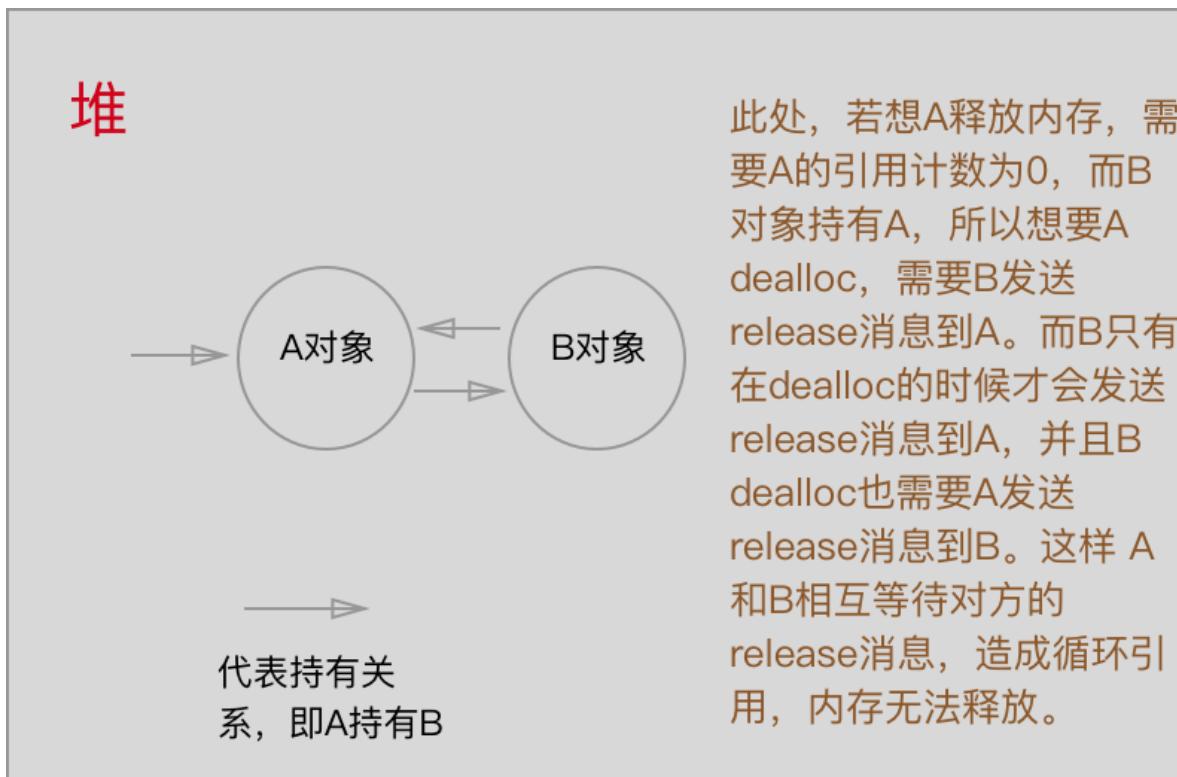
一个 int 变量被 __block 修饰与否的区别？

被修饰后，会被包装成一个对象

```
struct __Block_byref_age_0 {
    void *_isa;
```

```
__Block_byref_age_0 *__forwarding; //指向自己
int __flags;
int __size;
int age;//包装的具体的值
};
// age = 20;会被编译成下面这样
(age.__forwarding->age) = 20;
```

循环引用的产生



为什么在 block 外部使用 `_weak` 修饰的同时需要在内部使用 `_strong` 修饰

```
__weak typeof(self) weakSelf = self;
void (^block) (void) = ^{
    __strong typeof(weakSelf) strongSelf =
    weakSelf;
    if (strongSelf) {
    }
};
```

为了避免在 block 的执行过程中，突然出现 self 被释放的尴尬情况：

在 block 之前定义对 self 的一个弱引用 weakSelf, 因为是弱引用，所以 self 被释放时 weakSelf 会变为 nil;

在 block 中引用该弱引用，考虑到多线程情况，通过强引用 strongSelf 来引用该弱引用，这时如果 self 不为 nil 就会 retain self, 以防止在 block 内部使用过程中 self 被释放。内部的 strongSelf 仅仅是个局部变量，存在栈中，会在 block 执行结束后回收，不会再造成循环引用

在 block 块中使用该强引用 strongSelf, 注意对 strongSelf 进行 nil 检测，因为多线程在弱引用 weakSelf 对强引用 strongSelf 赋值时，弱引用 weakSelf 可能已经为 nil 了

强引用 strongSelf 在 block 作用域结束之后，自动释放。

block 不需要使用 weakSelf 情况

当 block 本身不被 self 持有，而被别的对象持有，同时不产生循环引用的时候，就不需要 weakSelf. 最常见的代码就是 UIView 的动画代码

- UIView 的某个负责动画的对象持有了 block
- block 持有了 self

因为 self 并没有持有 block, 所以就不存在循环引用，因此就不需要使用 weakSelf;

```
[UIView animateWithDuration:0.2 animations:^{
    self.alpha = 1;
}];
```

当动画结束时，UIView 会结束持有这个 block，如果没有别的对象持有 block 的话，block 就会释放掉，从而 block 就会释放对于 self 的持有，整个内存引用关系被解除

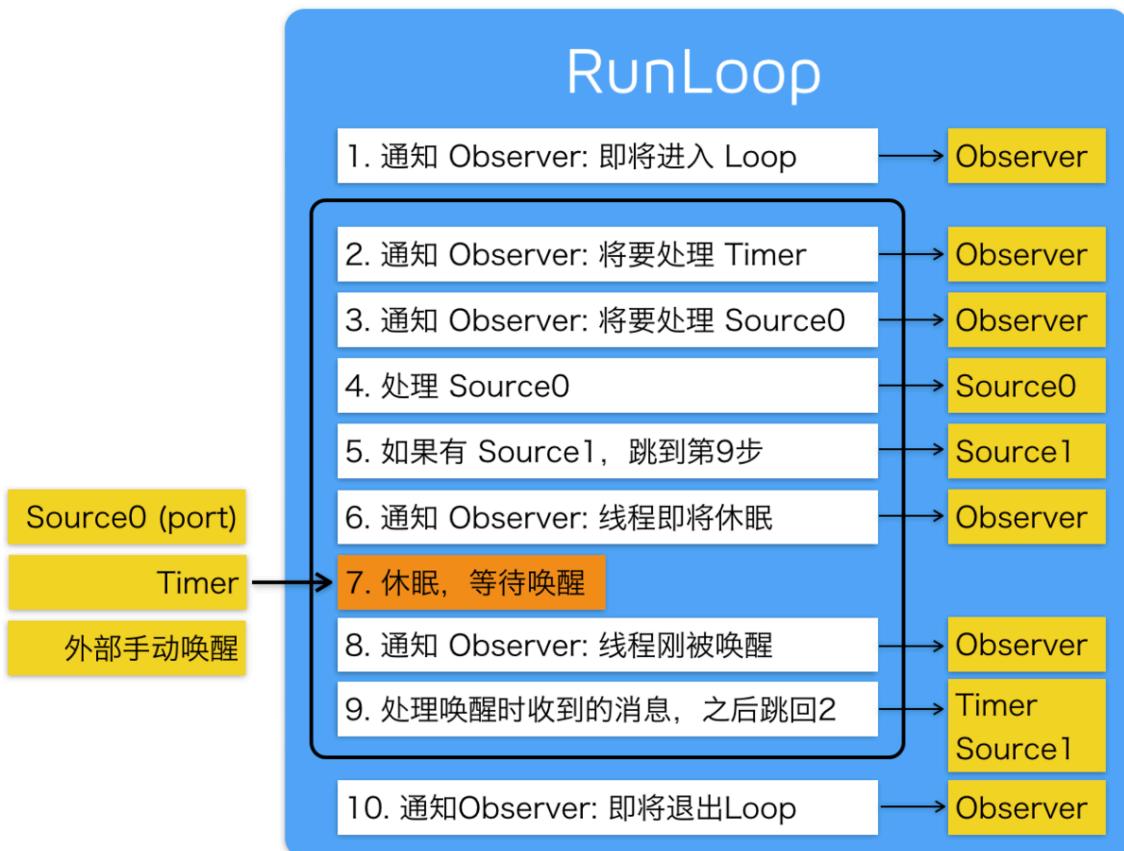
RunLoop 的作用是什么？它的内部工作机制了解么？（最好结合线程和内存管理来说）

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

作用：

1. 保持程序的持续运行，在 iOS 线程中，会在 main 方法给主线程创建一个 RunLoop，保证主线程不被销毁
2. 处理 APP 中的各种事件（如 touch, timer, performSelector 等）
3. 界面更新

4. 手势识别
5. AutoreleasePool
 1. 系统在主线程 RunLoop 注册了 2 个 observer
 2. 第一个 observe 监听即将进入 RunLoop，调用 `_objc_autoreleasePoolPush()` 创建自动释放池
 3. 第二个 observe 监听两个事件，进入休眠之前和即将退出 RunLoop
 4. 在进入休眠之前的回调里，会先释放自动释放池，然后在创建一个自动释放池
 5. 在即将退出的回调里，会释放自动释放池
6. 线程保活
7. 监测卡顿



哪些场景可以触发离屏渲染? (知道多少说多少)

1. 添加遮罩 mask
2. 添加阴影 shadow
3. 设置圆角并且设置 masksToBounds 为 true
4. 设置 allowsGroupOpacity 为 true 并且 layer.opacity 小于 1.0 和有子 layer 或者背景不为空
5. 开启光栅化 shouldRasterize=true

光栅化

将 layer 及其子 layer 矢量图转化为 bitmap (位图) , 并存储在缓存中, 下次存储的时候就从缓存中读取, 一般用于变化不大的 layer

光栅化概念: 将图转化为一个个栅格组成的图象。

光栅化特点: 每个元素对应帧缓冲区中的一像素

光栅化会导致离屏渲染, 影响图像性能, 那么光栅化是否有助于优化性能, 就取决于光栅化创建的位图缓存是否被有效复用, 而减少渲染的频度

光栅化就是通过把视图的内容渲染成纹理并缓存, 等到下次调用的时候直接去缓存的取出纹理, 但是更新内容时候, 会启用离屏渲染, 所以更新的代价比较大, 只能用于静态内容; 而且如果光栅化的元素 100ms 没有被使用, 也将被移除, 故而不常用元素的光栅化并不会优化显示。

AppDelegate 如何瘦身?

利用分类, 进行划分, 或者抽取 manager 管理

反射是什么? 可以举出几个应用场景么?

iOS 中的反射就说 runtime

1. 获取私有变量
2. 解析字典
3. KVO
4. MethodSwizzling

有哪些场景是 NSOperation 比 GCD 更容易实现的

- 可以取消操作: 在运行任务前, 可以在 NSOperation 对象调用 cancel 方法, 标明此任务不需要执行。但是 GCD 队列是无法取消的, 因为它遵循“安排好之后就不管了 (fire and forget) ”的原则。
- 可以指定操作间的依赖关系: 例如从服务器下载并处理文件的动作可以用操作来表示。而在处理其他文件之前必须先下载“清单文件”。而后续的下载工作, 都要依赖于先下载的清单文件这一操作。
- 监控 NSOperation 对象的属性: 可以通过 KVO 来监听 NSOperation 的属性: 可以通过 isCancelled 属性来判断任务是否已取消; 通过 isFinished 属性来判断任务是否已经完成。
- 可以指定操作的优先级: 操作的优先级表示此操作与队列中其他操作之间的优先关系, 我们可以指定它。

App启动优化策略？最好结合启动流程来说

iOS的启动流程：

1. 根据 info.plist 里的设置加载闪屏，建立沙箱，对权限进行检查等
2. 加载可执行文件
3. 加载动态链接库，进行 rebase 指针调整和 bind 符号绑定
4. Objc 运行时的初始处理，包括 Objc 相关类的注册、category 注册、selector 唯一性检查等
5. 初始化，包括了执行 +load() 方法、attribute((constructor)) 修饰的函数的调用、创建 C++ 静态全局变量。
6. 执行 main 函数
7. Application 初始化，到 applicationDidFinishLaunchingWithOptions 执行完
8. 初始化帧渲染，到 viewDidAppear 执行完，用户可见可操作。

启动优化：

1. 减少动态库的加载
2. 去除掉无用的类和 C++ 全局变量的数量
3. 尽量让 load 方法中的内容放到首屏渲染之后再去执行，或者使用 initialize 替换
4. 去除在首屏展现之前非必要的功能
5. 检查首屏展现之前主线程的耗时方法，将没必要的耗时方法滞后或者延迟执行

App无痕埋点的思路了解么？你认为理想的无痕埋点系统应该具备哪些特点？

App 无痕埋点的思路是利用 AOP 来拦截用户的操作并进行标记记录然后进行上传

我认为理想的无痕埋点系统应该具备以下特点：

1. 不侵入业务代码
2. 统计尽可能多的事件
3. 自动生成唯一标识
4. 要能统计到控件在但不同状态意义不同的情况
5. 需要某些机制能够提供业务数据
6. 在合适的时机去上传数据

你知道有哪些情况会导致 app 崩溃，分别可以用什么方法拦截并化解？

1. unrecognized selector sent to instance 方法找不到
2. 数组越界，插入空值
3. [NSDictionary initWithObjects:forKeys:]使用此方法初始化字典时，objects 和 keys 的数量不一致时
4. NSMutableDictionary, setObject:forKey:或者 removeObjectForKey:时，key 为 nil
5. setValue:forUndefinedKey:, 使用 KVC 对对象进行存取值时传入错误的 key 或者对不可变字典进行赋值
6. NSUserDefaults 存储时 key 为 nil
7. 对字符串操作时，传递的下标超出范围，判断是否存在前缀，后缀子串时，子串为空
8. 使用 C 字符串初始化字符串时，传入 null
9. 对可变集合或字符串使用 copy 修饰并进行修改操作
10. 在空间未添加到父元素上之前，就使用 autoLayout 进行布局
11. KVO 在对象销毁时，没有移除 KVO 或者多次移除 KVO
12. 野指针访问
13. 死锁
14. 除 0

利用 runtime 进行方法拦截，然后做逻辑处理

你知道有哪些情况会导致 app 卡顿，分别可以用什么方法来避免

1. 主线程中进化 IO 或其他耗时操作，解决：把耗时操作放到子线程中操作
2. GCD 并发队列短时间内创建大量任务，解决：使用线程池
3. 文本计算，解决：把计算放在子线程中避免阻塞主线程
4. 大量图像的绘制，解决：在子线程中对图片进行解码之后再展示
5. 高清图片的展示，解法：可在子线程中进行下采样处理之后再展示

App 网络层有哪些优化策略？

1. 优化 DNS 解析和缓存
2. 对传输的数据进行压缩，减少传输的数据
3. 使用缓存手段减少请求的发起次数
4. 使用策略来减少请求的发起次数，比如在上一个请求未着地之前，不进行新的请求
5. 避免网络抖动，提供重发机制

@dynamic 与 @synthesize 的区别

- `@dynamic`告诉编译器，不自动生成getter/setter方法，以及实例变量
- `@synthesize`编译器期间，让编译器自动生成getter/setter方法。当有自定义的存或取方法时，自定义会屏蔽自动生成该方法

成员变量与属性

成员变量

1. 成员变量的默认修饰是`@protected`。
2. 成员变量不会自动生成set和get方法，需要自己手动实现。
3. 成员变量不能用点语法调用，因为没有set和get方法，只能使用`->`调用。

属性

1. 属性的默认修饰是`@protected`。
2. 属性会自动生成set和get方法。
3. 属性用点语法调用，点语法实际上调用的是set和get方法

访问权限

`@public`: 共有变量

`@private`: 私有变量 .m中定义变量的默认类型

`@protect`: 子类访问 .h 中定义变量的默认类型

`@package`: 包内，本框架内使用

TCP为什么要三次握手，四次挥手？

三次握手：建立连接

1. 客户端向服务端发起请求链接，首先发送SYN报文，`SYN=1, seq=x`,并且客户端进入`SYN_SENT`状态
2. 服务端收到请求链接，服务端向客户端进行回复，并发送响应报文，`SYN=1, seq=y, ACK=1, ack=x+1`,并且服务端进入到`SYN_RCVD`状态
3. 客户端收到确认报文后，向服务端发送确认报文，`ACK=1, ack=y+1`,此时客户端进入到`ESTABLISHED`，服务端收到用户端发送过来的确认报文后，也进入到`ESTABLISHED`状态，此时链接创建成功

四次挥手：断开链接

1. 客户端向服务端发起关闭链接，并停止发送数据
2. 服务端收到关闭链接的请求时，向客户端发送回应，我知道了，然后停止接收数据

3. 当服务端发送数据结束之后，向客户端发起关闭链接，并停止发送数据
4. 客户端收到关闭链接的请求时，向服务端发送回应，我知道了，然后停止接收数据

为什么需要四次挥手：

因为 TCP 是全双工通信的，在接收到客户端的关闭请求时，还可能在向客户端发送着数据，因此不能再回应关闭链接的请求时，同时发送关闭链接的请求

对称加密和非对称加密的区别？分别有哪些算法的实现？

对称加密，加密的加密和解密使用同一密钥

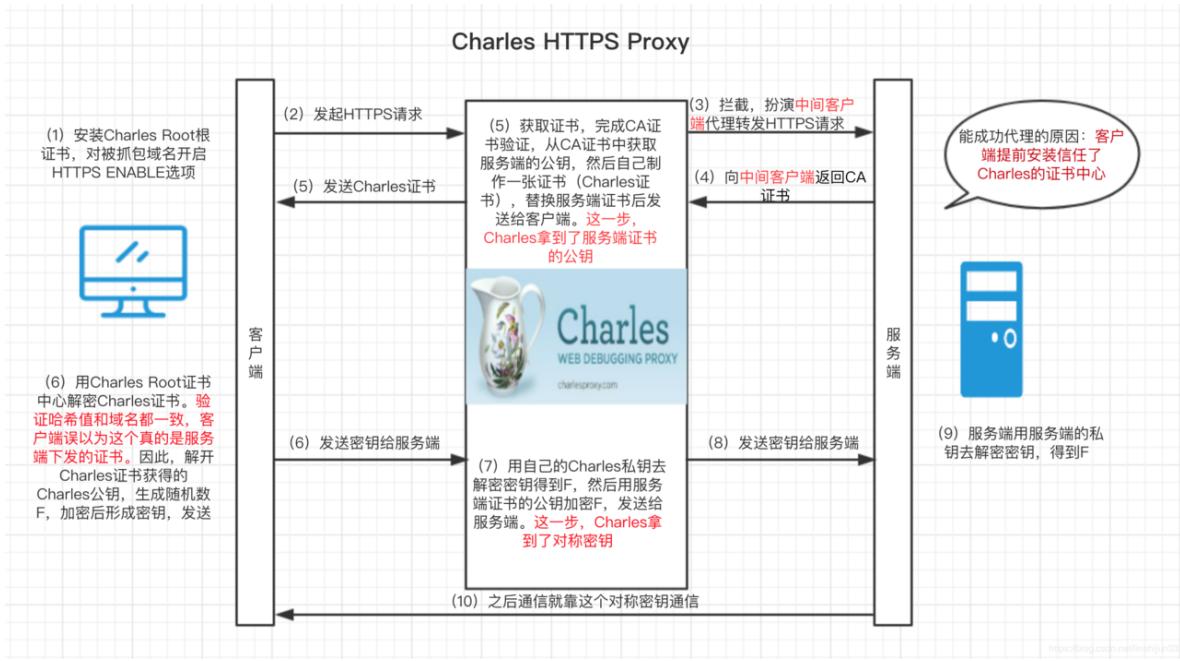
非对称加密，使用一对密钥用于加密和解密，分别为公开密钥和私有密钥。公开密钥所有人都可以获得，通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密

对称加密常用的算法实现有 AES, ChaCha20, DES, 不过 DES 被认为是不安全的；非对称加密用的算法实现有 RSA, ECC

HTTPS是如何实现验证身份和验证完整性的？

使用数字证书和 CA 来验证身份，首先服务端先向 CA 机构去申请证书，CA 审核之后会给出一个数字证书，里面包含公钥、签名、有效期，用户信息等各种信息，在客户端发送请求时，服务端会把数字证书发给客户端，然后客户端会通过信任链来验证数字证书是否是有效的，来验证服务端的身份。

什么是中间人攻击？如何避免



避免：

客户端可以预埋证书在本地，然后进行证书的比较是否是匹配的

了解编译的过程么？分为哪几个步骤？

1. 预编译：主要处理以“#”开始的预编译指令
2. 编译
 1. 词法分析：将字符序列分割成一系列的记号。
 2. 语法分析：根据产生的记号进行语法分析生成语法树。
 3. 语义分析：分析语法树的语义，进行类型的匹配、转换、标识等。
 4. 中间代码生成：源码级优化器将语法树转换成中间代码，然后进行源码级优化，比如把 $1+2$ 优化为 3 。中间代码使得编译器被分为前端和后端，不同的平台可以利用不同的编译器后端将中间代码转换为机器代码，实现跨平台。
 5. 目标代码生成：此后的过程属于编译器后端，代码生成器将中间代码转换成目标代码（汇编代码），其后目标代码优化器对目标代码进行优化，比如调整寻址方式、使用位移代替乘法、删除多余指令、调整指令顺序等
3. 汇编：汇编器将汇编代码转变成机器指令
4. 静态链接：链接器将各个已经编译成机器指令的目标文件链接起来，经过重定位过后输出一个可执行文件
5. 装载：装载可执行文件、装载其依赖的共享对象
6. 动态链接：动态链接器将可执行文件和共享对象中需要重定位的位置进行修正
7. 最后，进程的控制权转交给程序入口，程序终于运行起来了

静态链接了解么？静态库和动态库的区别

静态链接是指将多个目标文件合并为一个可执行文件

静态库：链接时完整地拷贝至可执行文件中，被多次使用就有多份冗余拷贝

动态库：链接时不复制，程序运行时由系统动态加载到内存，供程序调用，系统只加载一次，多个程序共用，节省内存

内存的几大区域，各自的职能分别是什么？

代码段：编译之后的代码

数据段：字符串常量，全局变量，静态变量

栈：存放函数的参数值，局部变量。系统自动分配并释放

堆：创建出来的对象会放在堆区。人为释放或系统释放

栈和静态区是操作系统自己管理的，对程序员来说相对透明，所以，一般我们只需要关注堆的内存分配

static 和 const 有什么区别？

const 是指声明一个常量

static 修饰全局变量时，表示此全局变量只在当前文件可见 static 修饰局部变量时，表示每次调用的初始值为上一次调用的值，调用结束后存储空间不释放

static 的作用：

- static 是静态修饰符，由他修饰的变量会保存在全局数据区
- 普通的局部变量或者全局变量，都是有系统自动分配内存的，并且当变量离开作用域的时候释放掉
- 而使用 static 修饰的变量，则会在程序运行期都不会释放，只有当程序结束的时候才会释放
- 因此对于那些需要反复使用的变量，我们通常使用 static 来修饰，避免重复创建导致不必要的内存开销

什么时候会出现死锁？如何避免？

死锁是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。发生死锁的四个必要条件：

1. 互斥条件：一个资源每次只能被一个线程使用。

2. 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：线程已获得的资源，在未使用完之前，不能强行剥夺。
4. 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

只要上面四个条件有一个条件不被满足就能避免死锁

说一说你对线程安全的理解？

对于某个资源如果只有读操作，则这个资源无需同步就是线程安全的，若有多个线程进行读写操作，则需要线程同步来保证线程安全。

列举你知道的线程同步策略？如何保证线程同步： 关于锁的使用

1. OSSpinLock 自旋锁，已不再安全，除了这个锁之外，下面写的锁，在等待时，都会进入线程休眠状态，而非忙等
2. os_unfair_lock atomic 就是使用此锁来保证原子性的
3. pthread_mutex_t 互斥锁，并且支持递归实现和条件实现
4. NSLock,NSRecursiveLock,基本的互斥锁，NSRecursiveLock 支持递归调用，都是对 pthread_mutex_t 的封装
5. NSCondition,NSConditionLock，条件锁，也都是对 pthread_mutex_t 的封装
6. dispatch_semaphore_t 信号量
7. @synchronized 也是 pthread_mutex_t 的封装

有哪几种锁？各自的原理？它们之间的区别是什么？最好可以结合使用场景来说

自旋锁：自旋锁在无法进行加锁时，会不断的进行尝试，一般用于临界区的执行时间较短的场景，不过 iOS 的自旋锁 OSSpinLock 不再安全，主要原因发生在低优先级线程拿到锁时，高优先级线程进入忙等 (busy-wait) 状态消耗大量 CPU 时间，从而导致低优先级线程拿不到 CPU 时间，也就无法完成任务并释放锁。这种问题被称为优先级反转。

互斥锁：对于某一资源同时只允许有一个访问，无论读写，平常使用的 NSLock 就属于互斥锁

读写锁：对于某一资源同时只允许有一个写访问或者多个读访问，iOS 中 pthread_rwlock 就是读写锁

条件锁：在满足某个条件的时候进行加锁或者解锁，iOS 中可使用

NSConditionLock 来实现

递归锁：可以被一个线程多次获得，而不会引起死锁。它记录了成功获得锁的次数，每一次成功的获得锁，必须有一个配套的释放锁和其对应，这样才不会引起死锁。只有当所有的锁被释放之后，其他线程才可以获得锁，iOS 可使用 NSRecursiveLock 来实现

链表和数组的区别是什么？插入和查询的时间复杂度分别是多少？

链表和数组都是一个有序的集合，数组需要连续的内存空间，而链表不需要，链表的插入删除的时间复杂度是 $O(1)$ ，数组是 $O(n)$ ，根据下标查询的时间复杂度数组是 $O(1)$ ，链表是 $O(n)$ ，根据值查询的时间复杂度，链表和数组都是 $O(n)$

哈希表

定义：

哈希表（hash table，也叫散列表），是根据键（key）直接访问访问在内存储存位置的数据结构。

哈希表本质是一个数组，数组中的每一个元素成为一个箱子，箱子中存放的是键值对。根据下标 index 从数组中取 value。关键是如何获取 index，这就需要一个固定的函数（哈希函数），将 key 转换成 index。不论哈希函数设计如何完美，都可能出现不同的 key 经过 hash 处理后得到相同的 hash 值，这时候就需要处理哈希冲突。

优缺点：

优点：哈希表可以提供快速的操作。

缺点：哈希表通常是基于数组的，数组创建后难于扩展。也没有一种简便的方法可以以任何一种顺序（例如从小到大）遍历表中的数据项。

综上，如果不需要有序遍历数据，并且可以提前预测数据量的大小。那么哈希表在速度和易用性方面是无与伦比的。

哈希查找步骤：

- 使用哈希函数将被查找的键映射（转换）为数组的索引，理想情况下（hash 函数设计合理）不同的键映射的数组下标也不同，所有的查找时间复杂度为 $O(1)$ 。但是实际情况下不是这样的，所以哈希查找的第二步就是处理哈希冲突。
- 处理哈希碰撞冲突。处理方法有很多，比如拉链法、线性探测法。

哈希表存储过程：

- 使用 hash 函数根据 key 得到哈希值 h
- 如果箱子的个数为 n，那么值应该存放在底 $(h \% n)$ 个箱子中。 $h \% n$ 的

值范围为[0, n-1]。

- 如果该箱子非空（已经存放了一个值）即不同的key得到了相同的h产生了哈希冲突，此时需要使用拉链法或者开放定址线性探测法解决冲突。

常用哈希函数：

哈希查找第一步就是使用哈希函数将键映射成索引。这种映射函数就是哈希函数。如果我们有一个保存0-M数组，那么我们就需要一个能够将任意键转换为该数组范围内的索引（0~M-1）的哈希函数。哈希函数需要易于计算并且能够均匀分布所有键。比如举个简单的例子，使用手机号码后三位就比前三位作为key更好，因为前三位手机号码的重复率很高。再比如使用身份证号码出生年月位数要比使用前几位数要更好。在实际中，我们的键并不都是数字，有可能是字符串，还有可能是几个值的组合等，所以我们需要实现自己的哈希函数。

- 直接寻址法
- 数字分析法
- 平方取中法
- 折叠法
- 随机数法
- 除留余数法

负载因子 = 总键值对数/数组的个数：

负载因子是哈希表的一个重要属性，用来衡量哈希表的空/满程度，一定程度也可以提现查询的效率。负载因子越大，意味着哈希表越满，越容易导致冲突，性能也就越低。所以当负载因子大于某个常数（一般是0.75）时，哈希表将自动扩容。哈希表扩容时，一般会创建两倍于原来的数组长度。因此即使key的哈希值没有变化，对数组个数取余的结果会随着数组个数的扩容发生变化，因此键值对的位置都有可能发生改变，这个过程也成为重哈希（rehash）。

哈希冲突的解决方法：

- 拉链法

简单来说就是数组 + 链表。将键通过hash函数映射为大小为M的数组的下标索引，数组的每一个元素指向一个链表，链表中的每一个结点存储着hash出来的索引值为结点下标的键值对。链法的实现方式（新插入的键值对放在链表头部）带来了两个好处：

1. 头插法可以节省插入耗时。如果插到尾部，则需要时间复杂度为O(n)的操作找到链表尾部，或者需要额外的内存地址来保存尾部链表的位置。
2. 头插法可以节省查找耗时。对于一个数据系统来说，最新插入的数据往往可能频繁的被查询。

- 开放定址线性探测发

使用两个大小为N的数组（一个存放keys，另一个存放values）。使用

数组中的空位解决碰撞，当碰撞发生时（即一个键的 hash 值对应数组的下标被另外两个键占用）直接将下标索引加一（`index += 1`），这样会出现三种结果

NSDictionary

解释版本一：是使用 NSMapTable 实现的，采用拉链法解决哈希冲突

解释版本二：是对 `CFDictionary` 的封装，解决哈希冲突使用的是开放定址线性探测法

Apple 方案选择

- `@synchronized` 使用的是拉链法。拉链法多用于存储的数据是通用类型，能够被反复利用，就像 `@synchronized` 存储的是锁是一种无关业务的实现结构，程序运行时多个对象使用同一个锁的概率相当高，有效的节省了内存。
- `weak` 对象 `associatedObject` 采用的是开放定址线性探测法。开放定址线性探测法用于存储的数据是临时的，用完尽快释放，就像 `associatedObject, weak`。

双向链表与单向链表或是数组的优点

为什么不选择单向链表：单链表的节点只知道它后面的节点（只有指向后一节点的指针），而不知道前面的。所以如果想移动其中一个节点的话，其前后的节点不好做衔接。

为什么不选择数组：数组中元素在内存的排列是连续的，对于寻址操作非常便利；但是对于插入，删除操作很不方便，需要整体移动，移动的元素个数越多，代价越大。而链表恰恰相反，因为其节点的关联仅仅是靠指针，所以对于插入和删除操作会很便利，而寻址操作却比较费时。由于在 LRU 策略中会有非常多的移动，插入和删除节点的操作，所以使用双向链表是比较有优势的

链表删除某个点

单链表

遍历，并记录前节点和当前节点，`persiousNode?.next = currentNode?.next`

双链表

遍历，记录当前节点，`currentNode.prev.next = currentNode.next, cur.nextNode.prevNode = cur.prevNode;`

在类的头文件中尽量少引用其他头文件

有时，类 A 需要将类 B 的实例变量作为它公共 API 的属性。这个时候，我们不应该引入类 B 的头文件，而应该使用向前声明（forward declaring）使用 class 关键字，并且在 A 的实现文件引用 B 的头文件。

这样做有什么优点呢：

- 不在 A 的头文件中引入 B 的头文件，就不会一并引入 B 的全部内容，这样就减少了编译时间。
- 可以避免循环引用：因为如果两个类在自己的头文件中都引入了对方的头文件，那么就会导致其中一个类无法被正确编译。

但是个别的时候，必须在头文件中引入其他类的头文件，主要有两种情况：

- 该类继承于某个类，则应该引入父类的头文件。
- 该类遵从某个协议，则应该引入该协议的头文件。而且最好将协议单独放在一个头文件中。

如何使用不可变对象

在头文件中，设置对象属性为 readonly，在实现文件中设置为 readwrite。这样一来，在外部就只能读取该数据，而不能修改它，使得这个类的实例所持有的数据更加安全

.h 文件

```
@interface EOCPerson : NSObject  
@property (nonatomic, copy, readonly) NSString *firstName;  
@end
```

.m 文件

```
@interface EOCPerson ()  
@property (nonatomic, copy, readwrite) NSString *firstName;  
@end
```

```
@implementation EOCPerson  
@end
```

缓存时选用 NSCache 而非 NSDictionary

NSCache 优于 NSDictionary 的几点：

- 当系统资源将要耗尽时，`NSCache`具备自动删减缓冲的功能。并且还会先删减“最久未使用”的对象。
- `NSCache`不拷贝键，而是保留键。因为并不是所有的键都遵从拷贝协议（字典的键是必须要支持拷贝协议的，有局限性）。
- `NSCache`是线程安全的：不编写加锁代码的前提下，多个线程可以同时访问 `NSCache`。

“自动释放池快”降低内存峰值 `autoreleasepool` 的应用

在这里，`doSomethingWithInt:`方法可能会创建临时对象。随着循环次数的增加，临时对象的数量也会飙升，而只有在整个for循环结束后，这些临时对象才会得意释放。

```
for (int i = 0; i < 100000; i++) {  
    [self doSomethingWithInt:i];  
}
```

因此，我们需要用自动释放池来降低这种突兀的变化，这样一来，每次循环结束，我们都会将临时对象放在这个池里面，而不是线程的主池里面。

```
NSArray *databaseRecords = /* ... */;  
NSMutableArray *people = [NSMutableArray new];  
for (NSDictionary *record in databaseRecords) {  
    @autoreleasepool {  
        EOCPerson *person = [[EOCPerson  
alloc] initWithRecord:record];  
        [people addObject:person];  
    }  
}
```

`loadView`是干嘛用的？

`loadView`在 `view` 为 `nil` 时调用，早于 `ViewDidLoad`

当用到 nib 文件时，加载 nib 文件

没有用到，默认创建一个空 view，可以自定义一个 view 赋值给 `self.view`，不必调用 `super`

`assign`、`retain`、`copy`、`weak`、`strong`

`assign`:

`assign`一般用来修饰基本的数据类型，包括基础数据类型（`NSInteger`, `CGFloat`）和C数据类型（`int`, `float`, `double`, `char`, 等等），为什么呢？`assign`声明的属性是不会增加引用计数的，也就是说声明的属性释放后，就没有了，即使其他对象用到了它，也无法留住它，只会crash。但是，即使被释放，指针却还在，成为了野指针，如果新的对象被分配到了这个内存地址上，又会crash，所以一般只用来声明基本的数据类型，因为它们会被分配到栈上，而栈会由系统自动处理，不会造成野指针。

retain:

与`assign`相对，我们要解决对象被其他对象引用后释放造成的问题，就要用`retain`来声明。`retain`声明后的对象会更改引用计数，那么每次被引用，引用计数都会+1，释放后就会-1，即使这个对象本身释放了，只要还有对象在引用它，就会持有，不会造成什么问题，只有当引用计数为0时，就被`dealloc`析构函数回收内存了。

copy:

最常见到`copy`声明的应该是`NSString`。`copy`与`retain`的区别在于`retain`的引用是拷贝指针地址，而`copy`是拷贝对象本身，也就是说`retain`是浅复制，`copy`是深复制，如果是浅复制，当修改对象值时，都会被修改，而深复制不会。之所以在`NSString`这类有可变类型的对象上使用，是因为它们有可能和对应的可变类型如`NSMutableString`之间进行赋值操作，为了防止内容被改变，使用`copy`去深复制一份。`copy`工作由`copy`方法执行，此属性只对那些实现了`NSCopying`协议的对象类型有效。

以上三个可以在MRC中使用，但是`weak`和`strong`就只能在ARC中使用，也就是自动引用计数，这时就不能手动去进行`retain`、`release`等操作了，ARC会帮我们完成这些工作。

weak:

`weak`其实类似于`assign`，叫弱引用，也是不增加引用计数。一般只有在防止循环引用时使用，比如父类引用了子类，子类又去引用父类。`IBOutlet`、`Delegate`一般用的就是`weak`，这是因为它们会在类外部被调用，防止循环引用。

strong:

相对的，`strong`就类似与`retain`了，叫强引用，会增加引用计数，类内部使用的属性一般都是`strong`修饰的，现在ARC已经基本替代了MRC，所以我们最常见的就是`strong`了。

nonatomic:

在修饰属性时，我们往往还会加一个 nonatomic，这又是什么呢？它的名字叫非原子访问。对应的有 atomic，是原子性的访问。我们知道，在使用多线程时为了避免在写操作时同时进行写导致问题，经常会对要写的对象进行加锁，也就是同一时刻只允许一个线程去操作它。如果一个属性是由 atomic 修饰的，那么系统就会进行线程保护，防止多个写操作同时进行。这有好处，但也有坏处，那就是消耗系统资源，所以对于 iPhone 这种小型设备，如果不是进行多线程的写操作，就可以使用 nonatomic，取消线程保护，提高性能。

assign 和 weak 区别

| 项目 | weak | assign |
|--------------|---------|----------------|
| 修饰对象 | 对象 | 基本数据类型（也可以是对象） |
| 赋值方式 | 复制引用 | 复制数据 |
| 对对象的引用计数器的影响 | 无影响 | 无影响 |
| 对象销毁后 | 自动为 nil | 不变 |

assign：修饰基本数据类型，修饰对象类型时，不改变其引用计数，会产生悬垂指针，修饰的对象在被释放后，assign 指针仍然指向原对象内存地址，如果使用 assign 指针继续访问原对象的话，就可能会导致内存泄漏或程序异常

问题：@property (nonatomic, copy) NSMutableArray *test;

用 copy 修饰，内部会生成，NSArray，而不是 NSMutableArray，这样可以初始化成功，但是在调用 NSMutableArray 方法时候，会发生崩溃，无法找到方法，因为真实的类型为 NSArray，而这里没有 addObject 等属于 NSMutableArray 的方法

```
2020-02-16 10:42:35.517046+0800
demo[30334:12795094] -[__NSArray0 addObject:]: unrecognized selector sent to instance
0x7fff80617ad0
(lldb) po self.test
<__NSArray0 0x7fff80617ad0>
)
```

Protocol 中使用 @property

在 protocol 中使用 property 时，只会生成 setter 和 getter 方法的声明，而在遵守该协议的类中，要手动添加实例变量，并且需要实现 setter 和 getter 方法

Person.h

```
//设置协议，添加property
#import <Foundation/Foundation.h>

@protocol personDelegate <NSObject>
@property (nonatomic, copy) NSString *name;
@end
```

```
@interface Person : NSObject
@end
```

Student.h

```
//遵守Person的协议，并声明实例变量
#import <Foundation/Foundation.h>
#import "Person.h"
```

```
@interface Student : NSObject <personDelegate>
{
    NSString *_name;
}
@end
```

Student.m

```
//实现setter和getter
//有两种方法：自动实现和手动实现
```

```
//-----自动-----//
@implementation Student
@synthesize name;
@end
```

```
//-----手动-----//
@implementation Student
- (void)setName:(NSString *)name
```

```
{  
    _name = name;  
}  
- (NSString *)name  
{  
    return _name;  
}  
@end
```

深复制 浅复制

浅：拷贝指针地址，对象不变没有增加，这是多了一个指向的地址
深：是拷贝对象本身，生成一个新的，存在两份

对象的内存销毁时间表

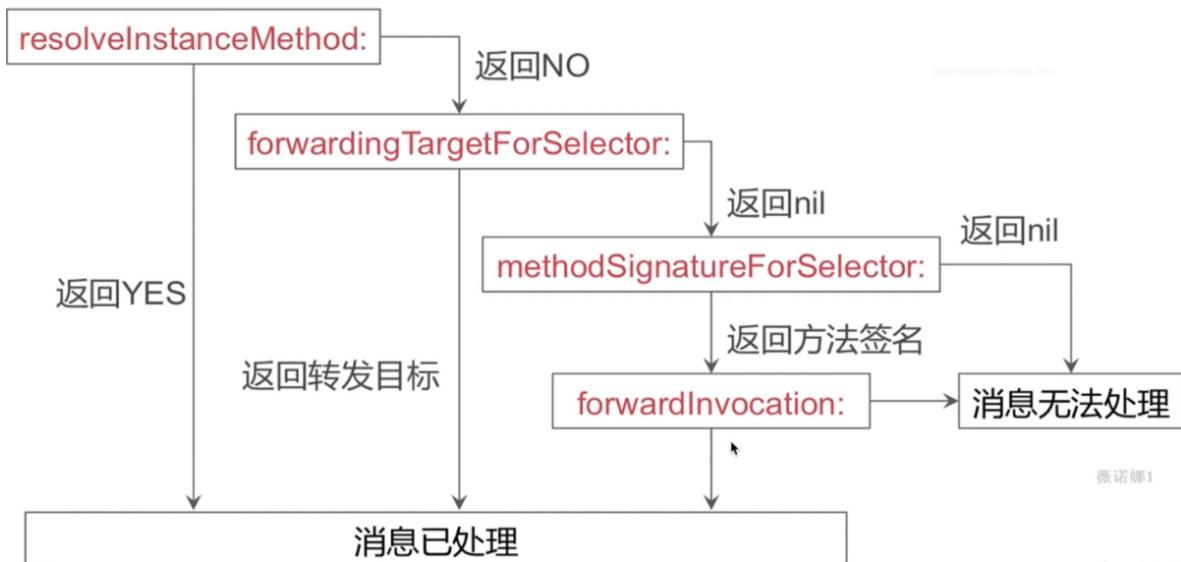
1. 调用 -release：
* 对象正在被销毁，生命周期即将结束。
* 不能再有新的 __weak 弱引用，否则将指向 nil.
* 调用 [self dealloc]
2. 子类 调用 -dealloc
* 继承关系中最底层的子类 在调用 -dealloc
* 如果是 MRC 代码 则会手动释放实例变量们 (iVars)
* 继承关系中每一层的父类 都在调用 -dealloc
3. NSObject 调 -dealloc
* 只做一件事：调用 Objective-C runtime 中的 object_dispose() 方法
4. 调用 object_dispose()
* 为 C++ 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()

objc_msgForward 调用后，发生什么

不进行 objc_msgSend，跳过查找 IMP 的过程，直接进行转发过程

1. resolveInstanceMethod:方法 (或 resolveClassMethod:)。
2. forwardingTargetForSelector:方法
3. methodSignatureForSelector:方法

4. forwardInvocation:方法
5. doesNotRecognizeSelector: 方法



runtime 如何实现 weak 变量的自动置 nil

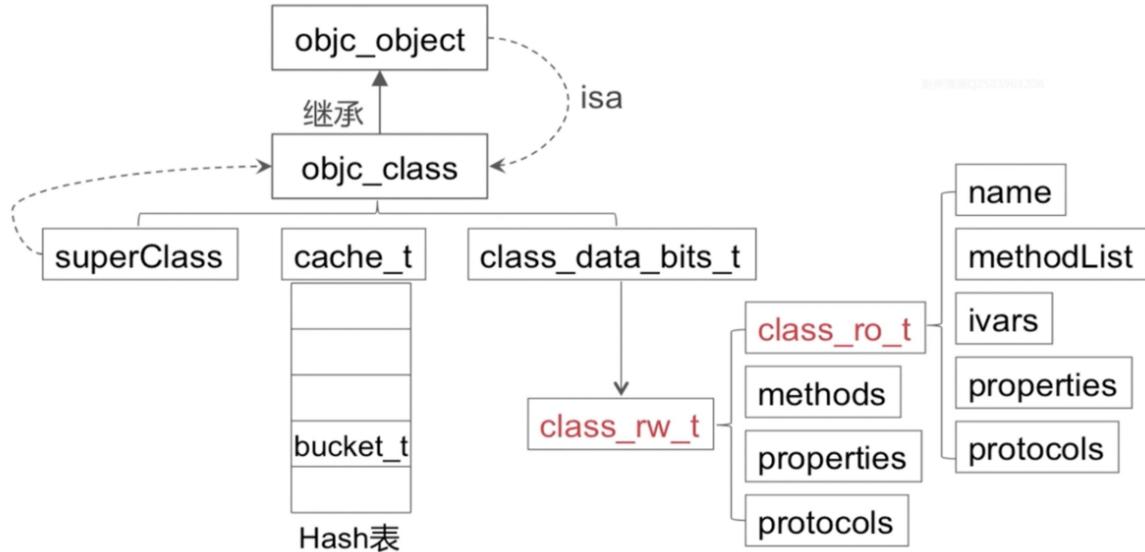
runtime 对注册的类，会进行布局，对于 weak 对象会放入一个 hash 表中。用 weak 指向的对象内存地址作为 key，当此对象的引用计数为 0 的时候会 dealloc，假如 weak 指向的对象内存地址是 a，那么就会以 a 为键，在这个 weak 表中搜索，找到所有以 a 为键的 weak 对象，从而设置为 nil。

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。



从图可见，ivars 是在 class_ro_t 里面的，也就是在编译阶段就确定了，后期，只能添加 method、protocol、property

一个 autorealese 对象在什么时刻释放？

分两种情况，手动干预与系统自动处理：

- 手动干预，就是手动添加 @autoreleasepool { }，变量添加到括弧内部，形成 autorelease 对象
- 自动处理，由 runloop 控制，超过作用域后，会自动添加到最近创建的 autoreleasepool 中，在 runloop 休眠时，进行释放

手动 @autoreleasepool { } 等价于

```
void *context = objc_autoreleasePoolPush();
// {} 中的代码
objc_autoreleasePoolPop(context);
```

把括弧内的变量进行入栈操作，超过作用域后，进行出栈操作，释放其中的对象，不用 runloop 控制

自动处理

RunLoop 每执行一次循环，就会进行一次 Push 和 Pop 操作，也就是旧池释放和新池创建，旧池释放会给对象发送 release 信号，进行释放。然后再创建新池，创建的对象，在超过作用域后，会自动加入到新创建的 autoreleasepool 中，等 runloop 休眠时或者退出时，进行释放。

什么时候手动创建

- 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。
- 进行大量 for 循环创建临时变量时候，内部创建

@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。

使用 block 时什么情况会发生引用循环，如何解决

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

```
__weak typeof(self) weakSelf = self;
void (^block)(void) = ^{
    __strong typeof(weakSelf) strongSelf =
weakSelf;
};
```

在 block 内如何修改 block 外部变量

```
__block int a = 0;
void (^foo)(void) = ^{
    a = 1;
};
```

block 内部的变量会被 copy 到堆区

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

如何用 GCD 同步若干个异步调用

```

dispatch_queue_t queue =
dispatch_queue_create("com.queue.a",
DISPATCH_QUEUE_CONCURRENT);
dispatch_group_t group =
dispatch_group_create();

// 这种方式，如果内部是异步执行，那么 notify 会离开被执行
// 行，如果是同步，则可以
dispatch_group_async(group, queue, ^{
    dispatch_async(queue, ^{
        });
    });
dispatch_group_async(group, queue, ^{
    dispatch_sync(queue, ^{
        });
    });
});

// 这种方式，不管异步同步，都可以正确的在所有任务完成后，
// 进行通知 notify
dispatch_group_enter(group);
dispatch_async(queue, ^{
    dispatch_group_leave(group);
});

dispatch_group_notify(group, queue, ^{
    });

```

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_async(queue, ^{
    // 1
});

```

```
dispatch_barrier_async(queue, ^{
    // 2
});

dispatch_async(queue, ^{
    // 3
});
```

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");
}
```

输出1，然后崩溃，发生锁死。

在主线程中运用主队列同步，也就是把任务放到了主线程的队列中。同步对于任务是立刻执行的，那么当把任务放进主队列时，它就会立马执行，只有执行完这个任务，viewDidLoad才会继续向下执行。而viewDidLoad和任务都是在主队列上的，由于队列的先进先出原则，任务又需等待viewDidLoad执行完毕后才能继续执行，viewDidLoad和这个任务就形成了相互循环等待，就造成了死锁。

下面代码输出顺序

```
dispatch_queue_t serialQueue =
dispatch_queue_create("test",
DISPATCH_QUEUE_SERIAL);

NSLog(@"1");

dispatch_async(serialQueue, ^{
    NSLog(@"2");
```

```
});  
NSLog(@"%@", @"3");  
dispatch_sync(serialQueue, ^{
    NSLog(@"%@", @"4");
});  
NSLog(@"%@", @"5");
```

打印顺序是13245

原因是：

首先先打印1

接下来将任务2添加至串行队列上，由于任务2是异步，不会阻塞线程，继续向下执行，打印3

然后是任务4,将任务4添加至串行队列上，因为任务4和任务2在同一串行队列，根据队列先进先出原则，任务4必须等任务2执行后才能执行，又因为任务4是同步任务，会阻塞线程，只有执行完任务4才能继续向下执行打印5

所以最终顺序就是13245

这里的任务4在主线程中执行，而任务2在子线程中执行。

记住：

同步任务会阻塞线程，等前面队列内的任务执行完之后，才会继续进行此任务，然后再进行向下进行

异步任务不会阻塞，会把任务添加到执行队列后，继续向下执行，由队列控制任务执行

`dispatch_get_global_queue` 全局队列：子线程

`dispatch_get_main_queue` 主队列：主线程

下面test会执行吗

```
dispatch_async(dispatch_get_global_queue(0, 0),
^{
    [self performSelector:@selector(test:)
    withObject:nil afterDelay:0];
});
```

不会，GCD开启一个新的线程，但是并没有添加RunLoop，而performSelector，是将任务添加到当前线程的runloop中，

而如果将dispatch_get_global_queue改成主队列，由于主队列所在的主线程是默认开启了runloop的，就会去执行(将dispatch_async改成同步，因为同步是在当前线程执行，那么如果当前线程是主线程，test方法也是会去执行的)。

怎么用GCD实现多读单写？

可以多个读者同时读取数据，而在读的时候，不能去写入数据。并且，在写的过程中，不能有其他写者去写。

```
// 读的时候，不能有写
- (id)readDataForKey:(NSString *)key
{
    __block id result;

    dispatch_sync(_concurrentQueue, ^{
        result = [self valueForKey:key];
    });

    return result;
}

// 写的时候，不能有其他任何操作，只能有一个在进行
- (void)writeData:(id)data forKey:(NSString *)key
{
    dispatch_barrier_async(_concurrentQueue, ^{
        [self setValue:data forKey:key];
    });
}
```

同步执行（sync）和异步执行（async）

同步（Sync）：同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行，即会阻塞线程。只能在当前线程中执行任务(是当前线程，不一定是主线程)，不

具备开启新线程的能力。

异步 (Async)：线程会立即返回，无需等待就会继续执行下面的任务，不阻塞当前线程。可以在新的线程中执行任务，具备开启新线程的能力（并不一定开启新线程）。如果不是添加到主队列上，异步会在子线程中执行任务

队列

队列 (Dispatch Queue)：这里的队列指执行任务的等待队列，即用来存放任务的队列。队列是一种特殊的线性表，采用 FIFO（先进先出）的原则，即新任务总是被插入到队列的末尾，而读取任务的时候总是从队列的头部开始读取

串行队列 (Serial Dispatch Queue)：

同一时间内，队列中只能执行一个任务，只有当前的任务执行完成之后，才能执行下一个任务。（只开启一个线程，一个任务执行完毕后，再执行下一个任务）。主队列是主线程上的一个串行队列，是系统自动为我们创建的

并发队列 (Concurrent Dispatch Queue)：

同时允许多个任务并发执行。（可以开启多个线程，并且同时执行任务）。并发队列的并发功能只有在异步 (dispatch_async) 函数下才有效

GCD 比 NSThread 优势？

1. NSThread，每次都会开启一个新的线程，浪费资源
2. GCD，会开启一个线程，并创建队列，通过线程池这个方式来进行任务调度

如何手动触发一个 value 的 KVO

```
[self willChangeValueForKey:@"time"];
self.time = @"3";
[self didChangeValueForKey:@"time"];
```

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，KVO会进行方法查找，然后进行成员变量查找

方法查找：setKey、_setKey

成员变量查找：_key、key、isKey、_isKey

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}

关联类，对象实现弱引用

定义一个简单对象，该对象提供一个被weak修饰的属性：

```
@interface WeakAssociatedObjectWrapper :  
NSObject  
@property (nonatomic, weak) id object;  
@end  
  
@implementation WeakAssociatedObjectWrapper  
@end
```

需要定义weak associated object时，使用该类对象作为中转：

```
@interface UIView <ViewController>  
@property (nonatomic, weak) UIViewController *vc;  
@end  
  
@implementation UIView <ViewController>  
  
- (void)setVc:(UIViewController *)vc {  
    WeakAssociatedObjectWrapper *wrapper =  
    [WeakAssociatedObjectWrapper new];  
    wrapper.object = vc;  
    objc_setAssociatedObject(self,  
    @selector(vc), wrapper,  
    OBJC_ASSOCIATION_RETAIN_NONATOMIC);  
}
```

```
- (UIViewController *)vc {
    WeakAssociatedObjectWrapper *wrapper =
objc_getAssociatedObject(self, _cmd);
    return wrapper.object;
}

@end
```

PerformSelecter

当调用 NSObject 的 performSelecter:afterDelay: 后，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。

当调用 performSelector:onThread: 时，实际上其会创建一个 Timer 加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效。

开启常驻子线程

```
// 开启子线程
- (void)addThread {
    self.thread = [[NSThread alloc]
initWithTarget:self
selector:@selector(threadAction) object:nil];
    [self.thread start];
}
// 添加runloop到子线程
- (void)threadAction {
    // 子线程没有runloop，也就没有自动释放池，需要自己
创建
    @autoreleasepool {
        // 在子线程中执行，所以直接获取，就是在子线程中
创建runloop
        NSRunLoop *runloop = [NSRunLoop
currentRunLoop];
        [runloop addPort:[NSMachPort port]
forMode:NSDefaultRunLoopMode];
        [runloop run];
    }
}
```

```

    }
}

// 子线程执行任务
- (void)executeInThread {
    [self
performSelector:@selector(<#selector#>)
onThread:self.thread withObject:nil
waitForDone:YES];
}

```

Texture 流畅的原因

UI 线程中一旦出现繁重的任务就会导致界面卡顿，这类任务通常分为 3 类：排版，绘制，UI 对象操作。

其中前两类操作可以通过各种方法扔到后台线程执行，而最后一类操作只能在主线程完成，并且有时后面的操作需要依赖前面操作的结果（例如 TextView 创建时可能需要提前计算出文本的大小）。ASDK 所做的，就是尽量将能放入后台的任务放入后台，不能的则尽量推迟（例如视图的创建、属性的调整）。

即在主线程的 RunLoop 中添加一个 Observer，监听了 kCFRunLoopBeforeWaiting 和 kCFRunLoopExit 事件，在收到回调时，遍历所有之前放入队列的待处理的任务，然后一一执行。

请问前后台切换，会发生些什么，系统哪些方法会被调用，viewcontroller 哪些方法会被调用

进入后台：

| 方法 | 作用 |
|-------------------------------|---|
| applicationWillResignActive | 点击 Home 键，app 开始准备进入后台，这个时候会进入该回调，意味着 app 被挂起，进程即将失去活跃。经过不严谨的测试，大约有 10 分钟左右的时间用来处理事务。 |
| applicationDidEnterBackground | 当 applicationWillResignActive 回调方法完全执行完毕后，会进入 applicationDidEnterBackground。 |

进入前台：

| 方法 | 作用 |
|--------------------------------|--|
| applicationWillEnterForeground | 在 app 未被杀死的情况下，点击 icon 再次进入 app，重新回到前台之前会先进入 applicationWillEnterForeground 回调 |
| applicationDidBecomeActive | 当 applicationWillEnterForeground 执行完毕后，会进入 applicationDidBecomeActive 回调，正式回归活跃。 |

前后台切换，主要的坑点在于：VC 中并没函数会调用，尤其注意：VC 相关的 Appear 和 Disappear 函数并不会被调用。想在 VC 中监听切换，只能监听通知，每个在 appdelegate 的生命代理方法都有对应的通知。如果考虑 APP 在后台被 kill 的情况：

进入后台后，如果没有后台运行权限及功能，可能在一段时间后被系统 kill 掉，再次进入 app 后，会重新进入启动流程。

请问对无序的 Array 排序，有什么好的方法，代码越少

```
// Comparator (比较器)
[arr sortUsingComparator:^NSComparisonResult(id _Nonnull obj1, id _Nonnull obj2) {
    // 升序
    return [obj1 compare:obj2];
    // 降序
    // return [obj2 compare:obj1];
}];

// Descriptor (描述器)
NSSortDescriptor *ageDescriptor =
[NSSortDescriptor sortDescriptorWithKey:@"age"
ascending:YES];
NSSortDescriptor *nameDescriptor =
[NSSortDescriptor sortDescriptorWithKey:@"name"]
```

```
ascending:YES];
[arr2 sortUsingDescriptors:@[ageDescriptor,
nameDescriptor]];
```

NSHashTable & NSMapTable

NSHashTable 是 **NSSet** 的通用版本，和 NSSet / NSMutableSet 不同的是，**NSHashTable** 具有下面这些特性：

- NSSet / NSMutableSet 持有成员的强引用，通过 hash 和 isEqual: 方法来检测成员的散列值和相等性。
- **NSHashTable** 是可变的，没有不可变的对应版本。
- **NSHashTable** 可以持有成员的弱引用。
- **NSHashTable** 可以在加入成员时进行 copy 操作。
- **NSHashTable** 可以存储任意的指针，通过指针来进行相等性和散列检查。

NSMapTable 是 **NSDictionary** 的通用版本。和 NSDictionary / NSMutableDictionary 不同的是，**NSMapTable** 具有下面这些特性：

- NSDictionary / NSMutableDictionary 对键进行拷贝，对值持有强引用。
- **NSMapTable** 是可变的，没有不可变的对应版本。
- **NSMapTable** 可以持有键和值的弱引用，当键或者值当中的一个被释放时，整个这一项就会被移除掉。
- **NSMapTable** 可以在加入成员时进行 copy 操作。
- **NSMapTable** 可以存储任意的指针，通过指针来进行相等性和散列检查。

NSArray 和 NSSet

NSArray：有序的，使用索引来查询很快，使用值查询很慢，插入/删除很慢

NSSet：无序的，用值来查找很快，插入/删除很快

GET 和 POST 方式的区别

- GET 的请求参数一般以?分割拼接到 URL 后面，POST 请求参数在 Body 里面
- GET 参数长度限制为 2048 个字符，POST 一般是没限制的
- GET 请求由于参数裸露在 URL 中，是不安全的，POST 请求则是相对安全

之所以说是相对安全，是因为，如果 POST 虽然参数非明文，但如果被抓包，GET 和 POST 一样都是不安全的。(HTTPS 该用还是得用)

HTTPS 和 HTTP 的区别

HTTPS 协议 = HTTP 协议 + SSL/TLS 协议

SSL 的全称是 Secure Sockets Layer，即安全套接层协议，是为网络通信提供安全及数据完整性的一种安全协议。TLS 的全称是 Transport Layer Security，即安全传输层协议。
即 HTTPS 是安全的 HTTP。

NSUserDefaults

支持数据类型：NSNumber, NSString, NSData, NSArray, NSDictionary, NSURL

NSCoder 归档 (序列化)

Object 与 (NSData) 二进制数据 之间的转化

对象 转换为 NSData 的过程。一般保存自定义的对象，但是只有遵守 NSCoding 的类才能只用归档。

遵守 NSCoding 协议必须要实现两个 require 方法：

```
(void)encodeWithCoder:(NSCoder *)aCoder //归档会触发  
- (nullableinstancetype)initWithCoder:(NSCoder *)aDecoder //解归档会触发
```

plist 文件保存

支持数据类型：NSString、NSNumber、NSData、NSArray、NSDictionary

UICollectionViewLayout 流水布局

- 调用 - (void)prepareLayout 进行初始化
- 重载 - (CGSize)collectionViewContentSize 返回内容的大小

- 重载 - (NSArray<UICollectionViewLayoutAttributes *> *)layoutAttributesForElementsInRect:(CGRect)rect 方法返回 rect 中所有元素的布局属性，返回的是一个数组
- 重载 - (UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath:(NSIndexPath *)indexPath 方法返回对应的 indexPath 的位置的 cell 的布局属性。
- 重载 - (nullable UICollectionViewLayoutAttributes *)layoutAttributesForSupplementaryViewOfKind:(NSString *)elementKind atIndexPath:(NSIndexPath *)indexPath;方法返回对应 indexPath 的位置的追加视图的布局属性，如果没有就不用重载
- 重载 - (nullable UICollectionViewLayoutAttributes *)layoutAttributesForDecorationViewOfKind:(NSString *)elementKind atIndexPath:(NSIndexPath *)indexPath;方法返回对应 indexPath 的位置的装饰视图的布局属性，如果没有也不需要重载
- 重载 - (BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)newBounds;当边界发生改变时，是否应该刷新。

什么是单例模式

个单例类，在整个程序中只有一个实例，并且提供一个类方法供全局调用，在编译时初始化这个类，然后一直保存在内存中，到程序（APP）退出时由系统自动释放这部分内存。

- UIApplication(应用程序实例类)
- NSNotificationCenter(消息中心类)
- NSFileManager(文件管理类)
- NSUserDefaults(应用程序设置)
- NSURLCache(请求缓存类)
- NSHTTPCookieStorage(应用程序 cookies 池)

重复初始化单例类会怎样？

程序直接崩溃

Terminating app due to uncaught exception

'NSInternalInconsistencyException', reason: 'There can only be one UIApplication instance.'

存储位置？

单例是在存储器的全局区域（数据段），在编译时分配内存，只要程序还在运行就会一直占用内存，在 APP 结束后由系统释放这部分内存内存

沙盒



- Documents: 保存应用运行时生成的需要持久化的数据,iTunes会自动备份该目录。苹果建议将在应用程序中浏览到的文件数据保存在该目录下。
- Library:

- Caches:

一般存储的是缓存文件，例如图片视频等，此目录下的文件不会再应用程序退出时删除。在手机备份的时候，iTunes不会备份该目录。例如音频,视频等文件存放

- Preferences:

保存应用程序的所有偏好设置iOS的Settings(设置)，我们不应该直接在这里创建文件，而是需要通过NSUserDefaults这个类来访问应用程序的偏好设置。 *iTunes会自动备份该文件目录下的内容。比如说:是否允许访问图片,是否允许访问地理位置.....

- tmp: 临时文件目录，在程序重新运行的时候，和开机的时候，会清空tmp文件夹。

Timer

创建方式：

1. scheduledTimerWithTimeInterval
2. timerWithTimeInterval

第一个方法默认创建了一个NSTimer并自动添加到了当前线程的RunLoop中去，第二个需要我们手动添加

释放处理：

1. 使用block的方式
2. 使用NSProxy来初始化一个子类

以 `scheduledTimerWithTimeInterval` 的方式触发的 timer，在滑动页面上的列表时，timer 会暂停，为什么？

自动添加到了当前线程的 Runloop 中去，在 default mode 下

解决：

// 方法1

将 timer 加入到 `NSRunLoopCommonModes` 中

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

// 方法2

放到另一个线程中，然后开启另一个线程的 runloop，这样可以保证与主线程互不干扰，而现在主线程正在处理页面滑动

```
dispatch_async(dispatch_get_global_queue(0, 0),  
^ {  
    timer = [NSTimer  
scheduledTimerWithTimeInterval:1 target:self  
selector:@selector(repeat:) userInfo:nil  
repeats:true];  
    [[NSRunLoop currentRunLoop] run];  
});
```

performSelector 传递多个参数

```
[self performSelector:@selector(mutilParams:  
withObject: @{@"key": @"value"} ]);  
  
- (void)mutilParams:(NSDictionary *)dict {  
}
```

[class new] 与 [[class alloc] init] 区别

对于后者，有分配和初始化的过程，alloc 从应用程序的虚拟地址空间上为该对象分配足够的内存，并且将新对象的引用计数加1、将对象的成员变量初始为零，init 会做真正的初使化工作，为对象的实例变量赋予合理有用的值。

一般不推荐使用[class new]，而推荐使用[[class alloc] init]

发现[class new]默认调用 alloc 与 init 方法，那么我们无法使用自定义的初始化方法，多了更多的局限性。那么[[class alloc] init] 会更方便，当然[[class alloc] init] 的设计也是由于历史的原因

Super

它只是一个"编译器指示符",告诉编译器在父类中搜索方法的实现

self = [super init] 为什么这么写?

```
- (instancetype)init
{
    self = [super init]; // call the designated
initializer
    if (self) {
        // Custom initialization
    }
    return self;
}
```

- 优先调用[super init] 是为了使超类完成它们自己的初始化工作
- 如果初始化一个对象失败，就会返回 nil，当返回 nil 的时候 self = [super init] 测试的主体就不会再继续执行。如果不这样做，你可能会操作一个不可用的对象，它的行为是不可预测的，最终可能会导致你的程序崩溃。

self.name 与 _name 的区别

self.name 是方法调用
_name 是直接操作 ivar

#include、#import、@class

#include

1. 在 C 语言中，我们使用 #include 来引入头文件。使用 #include “xx.h” 来引入自定义的头文件，使用 #include 来引入库中的头文件。
2. #include 不是不能防止重复引用头文件，而是操作起来比较复杂。
3. #include：为了防止重复引用可采用：

```
#ifndef ViewController_h  
  
#define ViewController_h  
  
#endif
```

#import

1. #import 是 #include 的升级版，可以防止重复引入头文件这种现象的发生。
2. import 在引入头的时候，就是完全将头文件拷贝到现在的文件中。所以也有效率上的问题。
3. #import 最大的问题在于，需要避免出现头文件递归引入的现象。
(如：A 引入 B，B 引入 A，那么 A、B 的头文件会互相不停的拷贝)

@class

1. @class 用来告诉编译器，有这样一个类，使书写代码时，不报错。
但是 @class 只是使导入的类名在引用时不受影响，不能创建该类的对象，因为创建对象时也需要访问其内部方法。
2. 因为 #import 引入头文件有效率问题，所以，当还没有调用类中方法，仅仅是定义类变量的时候，使用 @class 来提醒编译器。而在真正需要调用类方法的时候，进行 #import。
3. 如果 A 是 B 的父类，那么这是在 B.h 中就必须要使用 #import 来引入 A 的头，因为需要知道 A 类中有哪些变量和方法，以免 B 类中重复定义。

== 和 isEqual

==

对于基本类型，== 运算符比较的是值；

对于对象类型，== 运算符比较的是对象的地址（即是否为同一对象）

isEqual

先进行指针比较，同一个内存地址，则 YES。不同则进行对象比较，判断对象是否相同

isEqualToString

只判断字符串是否相等

```
NSString *a = @"a";
NSString *b = @"a";
if (a == b) {
    NSLog(@"yes");
} else {
    NSLog(@"no");
}
```

Result: Yes

hash 如何重写

1. 使用对象内存地址作为 hash 值

```
- (NSUInteger)hash {
    return [super hash];
}
```

[super hash]返回的就是该对象的内存地址

联想到前面对 hash 值唯一性的要求，使用对象的内存地址作为 hash 值不是很好

2. 对关键属性的 hash 值进行位或运算作为 hash 值

```
- (NSUInteger)hash {
    return [self.name hash] ^ [self.birthday
hash];
}
```

Objc 中向一个 nil 对象发送消息会怎样

OC 中向 nil 发消息，程序是不会崩溃的

空指针

1. 没有存储任何内存地址的指针就称为空指针 (NULL 指针)
2. 空指针就是被赋值为 0 的指针，在没有被具体初始化之前，其值为

0。

野指针

不是NULL指针，是指向"垃圾"内存（不可用内存）的指针。野指针是非常危险的。

Cell的复用方法区别：

- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier;
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier forIndexPath:(NSIndexPath *)indexPath;

方法一：

在没有初始化的时候，需要手动初始化

```
TableViewCell *cell = [tableView  
dequeueReusableCellWithIdentifier:@"reuseIdentifier"];  
if (!cell) {  
    cell = [[TableViewCell alloc]  
initWithStyle:UITableViewCellStyleDefault  
reuseIdentifier:@"reuseIdentifier"];  
}
```

方法二：

一定会返回cell，系统自动生成

```
TableViewCell *cell = [tableView  
dequeueReusableCellWithIdentifier:@"reuseIdentifier" forIndexPath:indexPath];
```

NSUserDefaults 以什么格式存储在手机中？

plist文件格式，支持 NSArray、NSDictionary、NSString、NSNumber、BOOL

NSUserDefaults will store the user preferences into a file into the

Block 方式添加 View

```
[self.view addSubview:^{
    UIView *view = [[UIView alloc] init];
    view;
}];
```

SizeToFit

让 Label 根据内容自适应大小，如，label 设置文本之后，调用一下 sizeToFit 方法，然后再去拿 label.frame.size，就可以得到文本的实际宽度了。

UIView、UIControl、UIButton

UIView

- UIControl
 - UIButton、UISlider、UISwitch

convertRect toView

```
CGRect newRect = [self.blueView
convertRect:CGRectMake(50, 50, 100, 100)
toView:self.greenView];
CGRect newRect = [self.blueView
convertRect:self.blueView.bounds toView:nil];
```

- 把 blueView 上的一个 view 为 CGRectMake(50, 50, 100, 100) 对试图转换到 greenView 上，算出正方形在绿色控件中的位置
- toView 为 nil 时候，就代表传的是 window

CoreAnimation

CABaseAnimation

基本动画，时常、开始结束状态等

CAKeyframeAnimation

关键路路径动画，设置不同的时间点

CAAnimationGroup

复杂动画都可以分解成多个简单动画

CATransition

转场动画 (渐入入渐出)- UIViewController

URL 构成

__kindof

```
@property (nonatomic, strong) NSArray<UIView *>  
*viewCollection;
```

如果用泛型，这样声明，那么数组内只能是 UIView 类型，对于子类，如 UIWebView、UIButton，编译器就会出警告

```
@property (nonatomic, strong) NSArray<__kindof  
UIView *> *viewCollection;
```

用这种结构声明，这个数组就可以包含 UIView 以及 UIView 的子类型，例如 UIWebView 或 UIButton

typeof

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

一元运算可以根据 () 内的变量自动识别并返回其数据类型

类方法，如何调用类方法

```
+ (NSMutableDictionary *)mediatorCache {  
    static NSMutableDictionary *dict;  
    static dispatch_once_t onceToken;  
    dispatch_once(&onceToken, ^{  
        dict = [NSMutableDictionary dictionary];  
    });  
    return dict;  
}
```

```
+ (void)doSomething {
    1. [[[self class] mediatorCache]
setObject:processBlock forKey:scheme];
    2. [[self mediatorCache]
setObject:processBlock forKey:scheme];
}
```

1和2均可以使用，但是合理的方式应该为1，
1，的意思是，先调用 class 获取到类，然后从类中查找类方法

怎么检查出无用代码？

通过 ObjC 的 runtime 源码

```
#define RW_INITIALIZED (1<<29)
bool isInitialized() {
    return getMeta()->data()->flags &
RW_INITIALIZED;
}
```

isInitialized 的结果会保存到元类的 class_rw_t 结构体的 flags 信息里，
flags 的 1<<29 位记录的就是这个类是否初始化了的信息。而 flags 的
其他位记录的信息，你可以参看 objc runtime 的源码，如下

```
// 类的方法列表已修复
#define RW_METHODIZED          (1<<30)

// 类已经初始化了
#define RW_INITIALIZED         (1<<29)

// 类在初始化过程中
#define RW_INITIALIZING        (1<<28)

// class_rw_t->ro 是 class_ro_t 的堆副本
#define RW_COPIED_RO           (1<<27)

// 类分配了内存，但没有注册
#define RW_CONSTRUCTING        (1<<26)
```

```
// 类分配了内存也注册了  
#define RW_CONSTRUCTED (1<<25)  
  
// GC : class 有不安全的 finalize 方法  
#define RW_FINALIZE_ON_MAIN_THREAD (1<<24)  
  
// 类的 +load 被调用了  
#define RW_LOADED (1<<23)
```

NSString *testObject = [[NSData alloc] init]; testObject 在编译时和运行时分别是什么类型的对象？

首先，声明 NSString *testObject 是告诉编译器，testObject 是一个指向某个 NSString 对象的指针。因为不管指向的是什么类型的对象，一个指针所占的内存空间都是固定的，所以这里声明成任何类型的对象，最终生成的可执行代码都是没有区别的。这里限定了 NSString 只不过是告诉编译器，请把 testObject 当做一个 NSString 来检查，如果后面调用了非 NSString 的方法，会产生警告。

接着，你创建了一个 NSData 对象，然后把这个对象所在的内存地址保存在 testObject 里。那么运行时，testObject 指向的内存空间就是一个 NSData 对象。你可以把 testObject 当做一个 NSData 对象来用。

内存中的栈和堆的区别是什么？

管理方式

对于栈来讲，是由编译器自动管理的，无需我们手动控制，对于堆来讲，释放工作有程序猿控制，这样就容易产生 **memory Leak**

申请大小

栈是向低地址扩展的数据结构，是一块连续的内存区域。这句话的意思是栈顶上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 **1M**)，总之是编译器确定的一个常数)，如果申请的空间超过了栈的剩余空间时候，就 **overflow**。因此，能获得栈的空间较小。

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。堆的大笑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快从栈中弹出

分配方式:

堆都是动态分配的，没有静态分配的堆。栈有两种分配方式：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。动态分配是有 **alloc** 函数进行分配的，但是栈的动态分配和堆是不同的，他的动态分配由编译器进行释放，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，计算机会在底层堆栈提供支持，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，他的机制是很复杂的。

#define 和 const 定义的变量

宏

- 只是在预处理器里进行文本替换，没有类型，不做任何类型检查
- 编译器可以对相同的字符串进行优化。只保存一份到 **.rodata** 段。甚至有相同后缀的字符串也可以优化
- 如果是整形、浮点型会有多份拷贝，但这些数写在指令中。
- 占的只是代码段而已，大量用宏会导致二进制文件变大。

常量

- 共享一块内存空间，就算项目中 **N** 处用到，也不会分配 **N** 块内存空间
- 可以根据 **const** 修饰的位置设定能否修改
- 在编译阶段会执行类型检查

```
#define MIN_V(A,B) (A < B ? A : B)
```

MD5 和 Base64 的区别是什么，各自场景是什么？

MD5:

全称为 **message digest algorithm 5(信息摘要算法)**，可以进行加密，但是不能解密，属于单向加密，通常用于文件校验

Base64

把任意序列的 8 为字节描述为一种不易为人识别的形式，通常用于邮件、**http** 加密。登陆的用户名和密码字段通过它加密，可以进行加密和解密。

哪些类不适合使用单例模式？即使他们在周期中只会出现一次？

- 单例对象一旦建立，对象指针是保存在静态区的，单例对象在堆中分配的内存空间，会在应用程序终止后才会被释放。
- 单例类无法继承，因此很难进行类的扩展。
- 单例不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。

如何添加一个自定义字体到工程中？

添加 **ttf** 文件。

通过 [**UIFont fontWithName:@"字体名" size:20.0**]; 使用

Configuration 中， debug 和 release 的区别是什么？

- **Debug** 是调试版本，包括的程序信息更多，只有 **DEBUG** 版的程序才能设置断点、单步执行、使用 **TRACE/ASSERT** 等调试输出语句。
- **RELEASE** 不包含任何调试信息，所以体积小、运行速度快。

关闭默认的隐式动画效果

```
[CATransaction begin];
[CATransaction setDisableActions:YES];
self.myview.layer.position = CGPointMake(10,
10);
[CATransaction commit];
```

显式和隐式动画的区别

1、隐式动画一直存在，如需关闭需设置；显式动画是不存在，如需显式要开启（创建）。

2、显式动画是指用户自己通过 **beginAnimations:context:** 和

commitAnimations 创建的动画。

隐式动画是指通过 UIView 的 animateWithDuration:animations: 方法创建的动画。

3、隐式动画是系统框架自动完成的。Core Animation 在每个 runloop 周期中自动开始一次新的事务，即使你不显式的用 [CATransaction begin] 开始一次事务，任何在一次 runloop 循环中属性的改变都会被集中起来，然后做一次 0.25 秒的动画。在 iOS4 中，苹果对 UIView 添加了一种基于 block 的动画方法：+animateWithDuration:animations:。这样写对做一堆的属性动画在语法上会更加简单，但实质上它们都是在做同样的事情。CATransaction 的 +begin 和 +commit 方法在 +animateWithDuration:animations: 内部自动调用，这样 block 中所有属性的改变都会被事务所包含。

4、显式动画，Core Animation 提供的显式动画类型，既可以直接对退出属性做动画，也可以覆盖默认的图层行为。我们经常使用的 CABasicAnimation, CAKeyframeAnimation, CATransitionAnimation, CAAnimationGroup 等都是显式动画类型，这些 CAAnimation 类型可以直接提交到 CALayer 上。

无论是隐式动画还是显式动画，提交到 layer 后，经过一系列处理，最后都经过上文描述的绘制过程最终被渲染出来。

Controller 生命周期

init —> loadView —> viewDidLoad —> viewWillAppear —>
viewWillLayoutSubviews —> viewDidLayoutSubviews —>
>viewDidAppear —> viewWillDisappear —> viewDidDisappear
init —> loadView —> viewDidLoad —> viewWillAppear —>
viewWillLayoutSubviews —> viewDidLayoutSubviews —>
>viewDidAppear —> viewWillDisappear —> viewDidDisappear —> 接
收到内存警告 —> viewWillUnload —> viewDidUnload

A -> B 没有修改 B 的 View 属性

[A viewWillAppear:]

[B viewDidLoad]

[B viewWillAppear:]

[B viewWillLayoutSubviews]

[B viewDidLayoutSubviews]

[A viewDidAppear:]

[B viewWillAppear:]

A -> B 修改了 B 的 View 属性

```
[B viewDidLoad]  
[A viewWillDisappear:]  
[B viewWillAppear:]  
[B viewWillLayoutSubviews]  
[B viewDidLayoutSubviews]  
[A viewDidDisappear:]  
[B viewWillAppear:]
```

修改了 View 属性，所以要先进行 `loadView`, `viewDidLoad` 才能对属性进行调整，所以在没有 Push 之前，已经执行了 `viewDidLoad` 方法了。

View 生命周期

```
initWithFrame -->willMoveToSuperview: -->didMoveToSuperview-->willMoveToWindow:-->didMoveToWindow --->layoutSubviews
```

单纯将视图从父视图上移除，只会触发 `UIView` 的 `removeFromSuperview` 函数。不会触发其他的生命周期函数。

- `willMoveToWindow`
- `didMoveToWindow`
- `removeFromSuperview`

Foundation & Core Foundation

```
#import <CoreFoundation/CFData.h>  
#import <Foundation/NSDate.h>
```

Foundation

- OC 对象
- 系统进行 ARC 内存管理

Core Foundation

- C 对象
- ARC 下，仍需手动管理

多态

- 不同的对象对同一个消息的不同响应，子类从写父类的方法来实现不同处理
- OC不支持多继承，但可以利用 protocol 实现多个接口完成类似功能

重写一个类的方式用继承好还是分类好？

最好是，用分类，新添加方法，但要注意，不用重写原有方法，应为分类重写后，会覆盖类的方法，导致，其他地方用类原来的方法都会变成分类的实现方式

继承的话，还要处理很多其他内容

OC 动态性

动态类型：运行时确定对象类型

动态绑定：运行时确定方法调用

动态加载：运行时加载需要的资源或可执行的代码

NSCopying & NSMutableCopying

```
- (id)mutableCopyWithZone:(NSZone *)zone {
    FYRuntime *copy = [[[self class]
allocWithZone:zone] init];
    // 浅拷贝
    copy.name = self.name;
    // 深拷贝
    copy.name = [self.name mutableCopy];

    return copy;
}

- (id)copyWithZone:(NSZone *)zone {
    FYRuntime *copy = [[[self class]
allocWithZone:zone] init];
    // 浅拷贝
    copy.name = self.name;
    // 深拷贝
    copy.name = [self.name copy];

    return copy;
}
```

父类实现 **NSCopying** 协议，子类继承，也实现 **NSCopying**，怎么操作？

子类重写 `copyWithZone` 方法，调用 `[super copyWithZone]` 初始化，并且拷贝父类中的实例变量，然后再对子类的变量进行处理

```
- (id)copyWithZone:(NSZone *)zone {
    Child *child = [super copyWithZone:zone]; // 子类调用 [super copyWithZone:zone]
    child.j = self.j;

    return child;
}
```

UIWindow

- UIWindow 是一种特殊的 UIView，继承于 UIView，通常在一个 app 中至少会有一个 UIWindow
- iOS 程序启动完毕后，创建的第一个视图控件就是 UIWindow，接着创建控制器的 View，最后将控制器的 View 添加到 UIWindow 上，于是控制器的 View 就显示在屏幕上。
- 一个 iOS 程序之所以能显示在屏幕上，完全是因为它有 UIWindow，也就是说，没有 UIWindow 就看不到任何 UI 界面。
- 状态栏和键盘都是特殊的 UIWindow

```
UIKIT_EXTERN const UIWindowLevel
UIWindowLevelNormal;
UIKIT_EXTERN const UIWindowLevel
UIWindowLevelAlert;
UIKIT_EXTERN const UIWindowLevel
UIWindowLevelStatusBar;
```

线程安全下的 Set / Get

```
- (NSString *)time {
    NSString *s = nil;
    @synchronized (self) {
        s = [[_time retain] autorelease];
    }
}
```

```
    return s;
}

- (void) setTime:(NSString *)time {
    @synchronized (self) {
        [time retain];
        [_time release];
        _time = time;
    }
}
```

UIImageView 初始化图片的方式？

- 通过 imageNamed 方式，会先去内存中查找，然后再去资源中查找，找到后，加载入内存，然后返回图片
- 通过 contentsOfFile 方式，直接从磁盘中查找，不会进行缓存

一个上线的项目，某个方法会出问题，在不破坏方法的前提下，怎么做？

利用 runtime、Method Swizzle 进行方法交换，

targetContentOffset

collectionView 吸附效果，最终的停止位置

dSYM 符号表

当编译器将源码转换为机器码时，会生成一个调试符号表，表内是二进制结构到原始源码的映射关系。调试符号表保存在 dSYM (debug symbol 调试符号表) 文件内。调试模式下符号表会保存在编译的二进制内，发布模式则将符号表保存在 dSYM 文件内用于减少包的体积。

当崩溃发生时，会在设备存储一份未符号化的崩溃日志

获取崩溃日志后，通过 dSYM 对追溯链中的每个地址进行符号化，转换为函数信息，产生的结果就是符号化后的崩溃日志。

traitCollectionDidChange

ViewController 和 View 都遵守 UITraitEnvironment 协议，自动旋转屏幕时候触发这个

```
// 横竖屏适配
- (void)traitCollectionDidChange:
(UITraitCollection *)previousTraitCollection
{
    [super
traitCollectionDidChange:previousTraitCollection
];
    // 转至竖屏
    if (self.traitCollection.verticalSizeClass
== UIUserInterfaceSizeClassRegular) { }
    // 转至横屏
    if (self.traitCollection.verticalSizeClass
== UIUserInterfaceSizeClassCompact) { }
}
```

强制旋转屏幕

```
[[UIDevice currentDevice] setValue:[NSNumber
numberWithInteger:UIDeviceOrientation]
forKey:@"orientation"];
```

Formate 使用

%02ld:%02ld 00:00

contentInsetAdjustmentBehavior & automaticallyAdjustsScrollViewInsets

iOS11 前有效： automaticallyAdjustsScrollViewInsets，
自动调节 ScrollView (以及其子类) 的 inset，防止被导航栏或者 Tab 栏
挡住，但是，布局仍然是从屏幕最顶部到屏幕最底部

iOS11 之后用 contentInsetAdjustmentBehavior 替代上面的

UIScrollViewContentInsetAdjustmentAutomatic
UIScrollViewContentInsetAdjustmentScrollableAxes
UIScrollViewContentInsetAdjustmentNever
UIScrollViewContentInsetAdjustmentAlways,

addChildViewController

```
[self addChildViewController:vc];
[self.view addSubview:vc.view];
[vc didMoveToParentViewController:self];
```

layoutSubviews 触发条件

- init 初始化不会触发 layoutSubviews (initWithFrame 会触发);
- addSubview 会触发 layoutSubviews;
- 设置 view 的 Frame 会触发 layoutSubviews, 当然前提是 frame 的值设置前后发生了变化;
- 滚动一个 UIScrollView 会触发 layoutSubviews;
- 旋转 Screen 会触发父 UIView 上的 layoutSubviews;
- 改变一个 UIView 大小的时候也会触发父 UIView 上的 layoutSubviews;

总结：只要改变 view 的 frame，就会触发 layoutSubviews，另外调用 addSubview 后也会触发 layoutSubviews 方法。

常用设计模式汇总

MVC 模式
代理模式
观察者模式
单例模式
策略模式
简单工厂

切回到主线程的 N 种方式

GCD 方式

异步回到主线程

原来线程和主线程并行执行，即：系统会来回切换这 2 个线程，看起来像 2 个线程同时在执行

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSLog(@"正在执行子线程中的代码");
    dispatch_async(dispatch_get_main_queue(), ^{
        NSLog(@"开始执行 main 线程中的代码");
    });
});
```

```
    NSLog(@"继续执行子线程中的代码");
});
```

同步回到主线程

原来线程和主线程 串行执行，即：中断正在执行的子线程，先回到主线程把需要执行的代码执行完后，再重新回到子线程继续执行。

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSLog(@"正在执行子线程中的代码");
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"开始执行main线程中的代码");
    });
    NSLog(@"继续执行子线程中的代码");
});
```

NSOperationQueue

```
[[NSOperationQueue mainQueue]
addOperationWithBlock:^{
    //需要在主线程执行的代码
}];
```

performSelectorOnMainThread

```
[self
performSelectorOnMainThread:@selector(<#selector #>) withObject:nil waitUntilDone:NO];
withObject: 用来传递需要传会主线程的参数
waitUntilDone:NO 等价于 GCD 方式的异步回到主线程， YES 等价于
GCD 方式中的同步回到主线程
```

RunLoop 与 GCD

dispatch_async(dispatch_get_main_queue(), block) 产生的任务，会触发 runloop 执行， libDispatch 会向主线程的 RunLoop 发送消息，RunLoop 会被唤醒，并从消息中取得这个 block，并在回调

但这个逻辑仅限于 dispatch 到主线程， dispatch 到其他线程仍然是由 libDispatch 处理的

nonatomic 崩溃

```
- (void)nonatomic{
    for (NSInteger i = 0; i < 10000; i++) {
        dispatch_async(dispatch_get_global_queue(0, 0),
        ^{
            self.name = [NSString
stringWithFormat:@"name:%ld", i];
        });
    }
}
```

崩溃，崩溃原因是在子线程 EXC_BAD_ACCESS，对象释放了

1. 在 MRC 模式下，属性 name 的 set 方法如下：

```
-(void)setName:(NSString *)name{
    if (_name != name) {
        [_name release];
        [name retain];
        _name = name;
    }
}
```

2. 虽然在 ARC 模式下不用写其 set 方法，但是在底层还是会走到这里

3. 因为是多线程，且没有加锁保护，所以在一个线程走到[_name release]后，可能在另一个线程又一次去释放，这时候造成崩溃。

4. 把 name 属性的 nonatomic 改成 atomic 就不会崩溃了，因为 atomic 加锁了，是安全的

重构的基本原则

使代码更容易理解，更容易找 BUG，高内聚，低耦合

三次法则：当代码重复出现三次，就必须要重构了；

剥离业务，独立业务

工具类

为什么要离屏渲染

指 GPU 在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作

在 VSync(垂直脉冲) 信号作用下，视频控制器每隔 16.67ms 就会去帧缓冲区(当前屏幕缓冲区) 读取渲染后的数据；但是有些效果被认为不能直接呈现于屏幕前，而需要在别的地方做额外的处理，进行预合成

有些视图渲染后的纹理需要被多次复用，但屏幕内的渲染缓冲区是实时更新的，所以需要通过开辟屏幕外的渲染缓冲区，将视图的内容渲染成纹理并缓存，然后再需要的时候在调入屏幕缓冲区，可以避免多次渲染的开销

重写 drawRect 为何会导致内存大量上涨？

实际上底层的操作是对于 CALayer 来进行的，它有一个属性 contents 也被称为寄宿图，UIView 检测到 -drawRect: 方法被调用了，它就会为视图分配一个寄宿图，图层就创建了一个绘制上下文，这个上下文需要的内存可从这个公式得出：图层宽 * 图层高 * 4 字节，宽高的单位均为像素。

1. 如果在 UIView 初始化时没有设置 rect 大小，将直接导致 drawRect 不被自动调用。
2. 该方法在调用 sizeThatFits 后被调用，所以可以先调用 sizeToFit 计算出 size。然后系统自动调用 drawRect: 方法。
3. 通过设置 contentMode 属性值为 UIViewContentModeRedraw。那么将在每次设置或更改 frame 的时候自动调用 drawRect:。
4. 直接调用 setNeedsDisplay，或者 setNeedsDisplayInRect: 触发 drawRect:，但是有个前提条件是 rect 不能为 0

控制动画暂停

CABasicAnimation

设置 layer 的属性，进行暂停

```
self.view.layer.speed  
self.view.layer.timeOffset
```

UIViewControllerAnimated

```
animator.pauseAnimation()  
animator.fractionComplete = translation.x/100  
animator.continueAnimation
```

CADisplayLink

一个定时器（但并不继承自 NSTimer），它能够让你的应用以屏幕刷新的帧率来同步更新 UI，所以我们可以通过它 1s 内调用指定 target 的指定方法的次数来获得屏幕刷新的帧率，从而来判断 CPU 的运行状态。

iOS设备的屏幕刷新频率(FPS)是60Hz，因此CADisplayLink的selector默认调用周期是每秒60次，这个周期可以通过frameInterval属性设置，CADisplayLink的selector每秒调用次数=60/ frameInterval。比如当frameInterval设为2，每秒调用就变成30次。因此，CADisplayLink周期的设置方式略显不便

iOS设备的屏幕刷新频率是固定的，CADisplayLink在正常情况下会在每次刷新结束都被调用，精确度相当高。

NSTimer的精确度就显得低了点，比如NSTimer的触发时间到的时候，runloop如果在忙于别的调用，触发时间就会推迟到下一个runloop周期

CADisplayLink也不一定准时，如果CPU忙于其它计算，就没法保证以60HZ执行屏幕的绘制动作，导致跳过若干次调用回调方法的机会，跳过次数取决CPU的忙碌程度。

主线程（Main Thread）与主队列（Main Queue）

主队列是系统自动为我们创建的一个串行队列，因此不用我们手动创建。在每个应用程序只有一个主队列，专门负责调度主线程里的任务，不允许开辟新的线程。也就是说，在主队列里的任务，即使是异步的，最终也只能在主线程中运行。因此，开头的第一段代码是可以保证在主线程中运行的。

主线程空闲时才会调度队列中的任务在主线程执行，如果当前主线程正在有任务执行，那么无论主队列中当前被添加了什么任务，都不会被调度

主线程是可以执行主队列之外其他队列的任务的。即使[NSThread mainThread]判断当前线程是主线程，也不能保证当前执行的任务是主队列的任务

```
dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    NSLog(@"isMainThread: %@", @(NSThread.isMainThread));
});
```

alloc

方法的内部. 就是调用了allocWithZone:方法

new -> alloc init

alloc -> allocWithZone

绝对单例

这样创建的，即使通过 [[SingleObjc alloc] init]; 也会只存在一份。

```
@implementation SingleObjc

static SingleObjc *manager;

+ (instancetype)shared {
    if (manager == nil) {
        static dispatch_once_t once;
        dispatch_once(&once, ^{
            manager = [[SingleObjc alloc] init];
        });
    }
    return manager;
}

+ (instancetype)allocWithZone:(struct _NSZone *)
*zone {
    if (manager == nil) {
        static dispatch_once_t ononetoken;
        dispatch_once(&onetoken, ^{
            manager = [super
allocWithZone:zone];
        });
    }
    return manager;
}
@end
```

forwardingTargetForSelector 和 forwardInvocation 区别？

```
-(void)forwardInvocation:(NSInvocation *)
*anInvocation
{
    if (anInvocation.selector ==
```

```

@selector(testMethod))
{
    TestModelHelper1 *h1 =
    [[TestModelHelper1 alloc] init];
    TestModelHelper2 *h2 =
    [[TestModelHelper2 alloc] init];
    [anInvocation invokeWithTarget:h1];
    [anInvocation invokeWithTarget:h2];
}
}

```

- forwardingTargetForSelector 仅支持一个对象的返回，也就是说消息只能被转发给一个对象
- forwardingTargetForSelector 无法处理消息的内容，比如参数和返回值
- forwardInvocation 可以将消息同时转发给任意多个对象，也可以处理返回值和参数
- 返回 forwardInvocation 方法中，最后一个 invokeWithTarget 的对象对应方法的返回值，即最后一个处理消息转发的对象对应方法的返回值。

HTTP 和 HTTPS 区别

1. https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。
2. http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。
3. http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。
4. http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全

HTTP 实现长链接

Connection:keep-alive 和 Keep-Alive: timeout=60

子视图在父视图外，进行响应

- (BOOL)pointInside:(CGPoint)point withEvent:(UIEvent *)event {

```
    if (self.isHidden == NO) {
        UIBezierPath *circle = [UIBezierPath
bezierPathWithArcCenter:self.plusBtn.center
radius:plusBtnWH * 0.5 startAngle:0 endAngle:2 *
M_PI clockwise:YES];
        UIBezierPath *tabbar = [UIBezierPath
bezierPathWithRect:self.bounds];
        if ([circle containsPoint:point] ||
[tabbar containsPoint:point]) {
            return YES;
        }
        return NO;
    }else {
        return [super pointInside:point
withEvent:event];
    }
}
```

JPG VS PNG

JPG在图片压缩方面有巨大优势，但采用有损压缩，图片质量有损失。一般截屏用PNG格式不但比JPG质量高且文件还更小，防锯齿PNG非常有优势。

View 和 Layer

- 首先 UIView 可以响应事件，Layer 不可以
- UIView 主要是对显示内容的管理而 CALayer 主要侧重显示内容的绘制
- 一个 Layer 的 frame 是由它的 anchorPoint, position, bounds 和 transform 共同决定的，而一个 View 的 frame 只是简单的返回 Layer 的 frame，同样 View 的 center 和 bounds 也是返回 Layer 的一些属性。

KVO 崩溃

- KVO 添加次数和移除次数不匹配：
 - 移除了未注册的观察者，导致崩溃。
 - 重复移除多次，移除次数多于添加次数，导致崩溃。
 - 重复添加多次，虽然不会崩溃，但是发生改变时，也同时会被观察多次。

- 被观察者提前被释放，被观察者在 dealloc 时仍然注册着 KVO，导致崩溃。
- 添加了观察者，但未实现 observeValueForKeyPath:ofObject:change:context: 方法，导致崩溃。
- 添加或者移除时 keypath == nil，导致崩溃。

NSURLConnection 和 URLSession

本质上是 NSURLConnection 的机制需要创建此对象的线程的 RunLoop 去监听网络事件，然后执行注册的回调，所以在网络回来前，需要保持 RunLoop 的运行状态。NSURLSession 应该是内部做了这个事情，所以发请求不需要所在线程“常驻”，而不是因为我们可以设置 delegateQueue。从代理方法的调用栈看，NSURLSession 的代理任务都是在 com.apple.NSURLConnection-work 这个 queue 被塞进 delegateQueue 的，如果不设置 delegateQueue 的话，代理任务就在 work 队列执行，设置之后再派发到对应的队列。具体在 work 队列之前做了什么，还没看到比较具体的资料，不过基于底层都是 CFNetworking，这个又是 Source0，可能做的和常驻线程相似。

addObject 做了什么

实际上，做了 retain 和 release 操作

```

-(void)addObject:(id)object{
    _objs[_count] = [object retain];
    _count++;
    //这里必须 retain一次。因为如果外部调用了该方法，然后
    又调用了 release，那么这里存的就是一个野指针。
}

-(void)removeObjectAtIndex:(NSUInteger)index{
    id obj = _objs[index];
    [obj release];
    //因为添加元素的时候有 retain一次，这里必须 release
    _objs[index] = nil;
}

```

常用的数据结构

- 堆

- 栈
- 队列
- 链表
- 树
- 数组