目录：
基础

底层：

31、RunLoop是什么?
32、假设有一个字符串aabcad，请写一段程序，去掉字符串中不相邻的重复字符串，即上述字符串处理之后的输出结果为：aabcd
@autoclosure（自动闭包）
34、iOS app启动如何优化?
35、swift面试题：
36、怎样防止反编译?
37、UITableView性能优化，超实用

线程：
37、 不要阻塞主线程
38、谈谈你对多线程开发的理解？ios中有几种实现多线程的方法?
39、进程和线程的区别？同步异步的区别？并行和并发的区别?
40、ViewController生命周期
41、iOS 中的多线程
42、内存管理的几条原则时什么？按照默认法则.那些关键字生成的对象需要手动释放？在和property结合的时候怎样有效的避免内存泄露?
谁申请，谁释放
43、dispatch_barrier_async的作用是什么?
44、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）
http：
46、http与https的区别?
47、服务器能否知道APNS推送后有没有到达客户端的方法?

app：
48、1.什么方式可以看到上架App的头文件?
2.阅读过哪些框架的源码?能说说它的架构方式吗
49、iOS iAP内购审核可能失败的问题
50、IAP内购中虚拟货币导致审核无法通过的问题?

iOS内推QQ群：656315826

基础：
1、如何令自己所写的对象具有拷贝功能?
如果想让自己的类具备copy方法，并返回不可边类型，必须遵循nscopying协议，并且实现
– (id)copyWithZone:(NSZone *)zone
如果让自己的类具备mutableCopy方法，并且放回可变类型，必须遵守NSMutableCopying，并实现– (id)mutableCopyWithZone:(nullable NSZone *)zone
注意：再此说的copy对应不可边类型和mutableCopy对应不可边类型方法，都是遵从系统规则而已。如果你想实现自己的规则，也是可以的。

2、 释放时，调用clearDeallocating函数。clearDeallocating函数首先根据对象地址获取所有weak指针地址的数组，然后遍历这个数组把其中的数据设为nil，最后把这个entry从weak表中删除，最后清理对象的记录。

追问的问题一：

1.实现weak后，为什么对象释放后会自动为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?。

追问的问题二：

2.当weak引用指向的对象被释放时，又是如何去处理weak指针的呢?

1、调用objc_release

2、因为对象的引用计数为0，所以执行dealloc

3、在dealloc中，调用了_objc_rootDealloc函数

4、在_objc_rootDealloc中，调用了object_dispose函数

5、调用objc_destructInstance

6、最后调用objc_clear_deallocating,详细过程如下：

a. 从weak表中获取废弃对象的地址为键值的记录

b. 将包含在记录中的所有附有 weak修饰符变量的地址，赋值为 nil

c. 将weak表中该记录删除

d. 从引用计数表中删除废弃对象的地址为键值的记录

3、题目：Swift mutating关键字的使用?
答案：在Swift中，包含三种类型(type): structure,enumeration,class
其中structure和enumeration是值类型(value type),class是引用类型(reference type)
但是与Objective-C不同的是，structure和enumeration也可以拥有方法(method)，其中方法可以为实例方法(instance method)，也可以为类方法(type method)，实例方法是和类型的一个实例绑定的。

在swift官方教程中有这样一句话：

“Structures and enumerations are value types. By default, the properties of a value type cannot be modified from within its instance methods.”

大致意思就是说，虽然结构体和枚举可以定义自己的方法，但是默认情况下，实例方法中是不可以修改值类型的属性。

```
//// 1. 在结构体的实例方法里面修改属性
struct Persion {
    var name = ""
    mutating func modify(name:String) {
        self.name = name
    }
}


/// 2. 在协议里面， 如何继承的结构体或枚举类型，想要改遍属性值， 必须添加 mutating
protocol Persionprotocol {
    var name : String {get}
    mutating func modify(name:String)
}
```

```
struct Persion : Persionprotocol {
    var name = ""
    mutating func modify(name:String) {
        self.name = name
    }
}


/// 3. 在枚举中直接修改self属性
enum Switch {
    case On, Off

    mutating func operatorTion() {
        switch self {
        case .On:
            self = .Off
        default:
            self = .On
        }
    }
}

var a = Switch.On
a.operatorTion()

print(a)
```

4、UIView和 CALayer是什么关系?

UIView 显示在屏幕上归功于 CALayer，通过调用 drawRect 方法来渲染自身的内容，调节 CALayer 属性可以调整 UIView 的外观，UIView 继承自 UIResponder，比起 CALayer 可以响应用户事件，Xcode6 之后可以方便的通过视图调试功能查看图层之间的关系。

UIView 是 iOS 系统中界面元素的基础，所有的界面元素都继承自它。它内部是由 Core Animation 来实现的，它真正的绘图部分，是由一个叫 CALayer(Core Animation Layer)的类来管理。UIView 本身，更像是一个 CALayer 的管理器，访问它的跟绘图和坐标有关的属性，如 frame，bounds 等，实际上内部都是访问它所在 CALayer 的相关属性。

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，如：

```
- (class) layerClass {
// 使某个 UIView的子类使用 GL来进行绘制
return ([CAEAGLLayer class]);
}
```

UIView 的 CALayer 类似 UIView 的子 View 树形结构，也可以向它的 layer 上添加子 layer，来完成某些特殊的显示。例如下面的代码会在目标 View 上敷上一层黑色的透明薄膜。

```
grayCover = [[CALayer alloc]init];
grayCover.backgroudColor = [[UIColor
blackColor]colorWithAlphaComponent:0.2].CGColor;
[self.layer addSubLayer:grayCover];
```

补充部分：这部分有深度了，大致了解一下吧，UIView 的 layer 树形在系统内部被系统维护着三份 copy
1.逻辑树，就是代码里可以操纵的，例如更改 layer 的属性等等就在这一份
2.动画树，这是一个中间层，系统正是在这一层上更改属性，进行各种渲染操作。
3.显示树，这棵树的内容是当前正被显示在屏幕上的内容。这三棵树的逻辑结构都是一样的，区别只有各自的属性

5、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
这个题目主要是考察关于 objc 中对 self 和 super 的理解：
self 是类的隐藏参数，指向当前调用方法的这个类的实例。而 super 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，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调用[self class] 时，会转化成 objc_msgSend 函数id objc_msgSend(id self, SELop, ...)。

调用[super class] 时 ，会转化成objc_msgSendSuper函数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;
};
```

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，告诉程序从父类中开始找方法，找到方法后，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 objc Runtime 开源代码对- (Class)class 方法的实现

```
-(Class)class {
return object_getClass(self);
}
```

6、@synthesize 和 @dynamic 分别有什么作用?
@property有两个对应的词，一个是@synthesize（合成实例变量），一个是@dynamic。
如果@synthesize和@dynamic都没有写，那么默认的就是 @synthesize var = _var;
// 在类的实现代码里通过 @synthesize 语法可以来指定实例变量的名字。

7、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，只有在消实发送出来之后，才确定被调用的代码。通过动态类型和动态绑定技术，您的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。当您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，您不必在Objective-C 代码中做任何工作，就可以自动获取动态绑定的好处。您在每次发送消息时，

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。

8、、Category（类别）、 Extension（扩展）和继承的区别

区别：
1. 分类有名字，类扩展没有分类名字，是一种特殊的分类。
2. 分类只能扩展方法（属性仅仅是声明，并没真正实现），类扩展可以扩展属性、成员变量和方法。
3. 继承可以增加，修改或者删除方法，并且可以增加属性。

9、为什么代理要用weak？代理的delegate和dataSource有什么区别？block和代理的区别?
通过weak打破循环引用。
delegate是一个类委托另一个类实现某个方法，协议里面的方法主要是与操作相关的。
datasource一个类通过datasource将数据发送给需要接受委托的类，协议里面的方法主要是跟内容有关的。
代理和block的区别：代理和block的共同特性是回调机制。不同的是代理的方法比较多，block代码比较集中；代理的运行成本要低于block的运行成本，block的出站需要从栈内存拷贝到堆内存。公共接口比较多时，用代理解耦；简单回调和异步线程中使用block。

10、id和NSObject＊的区别

id是一个 objc_object 结构体指针，定义是

typedef struct objc_object *id

id可以理解为指向对象的指针。所有oc的对象 id都可以指向，编译器不会做类型检查，id调用任何存在的方法都不会在编译阶段报错，当然如果这个id指向的对象没有这个方法，该崩溃还是会崩溃的。

NSObject *指向的必须是NSObject的子类，调用的也只能是NSObjec里面的方法否则就要做强制类型转换。

不是所有的OC对象都是NSObject的子类，还有一些继承自NSProxy。NSObject *可指向的类型是id的子集。

11、如何令自己所写的对象具有拷贝功能?

如果想让自己的类具备copy方法，并返回不可边类型，必须遵循nscopying协议，并且实现
– (id)copyWithZone:(NSZone *)zone

如果让自己的类具备mutableCopy方法，并且放回可变类型，必须遵守NSMutableCopying，并实现– (id)mutableCopyWithZone:(nullable NSZone *)zone

注意：再此说的copy对应不可边类型和mutableCopy对应不可边类型方法，都是遵从系统规则而已。如果你想实现自己的规则，也是可以的。

12、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 。

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?

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.
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：

```
 CYLArrayCopyDmo (
)
 CYLArrayCopyDmo(
    1,
    2,
    3,
    4
)
```

为了理解这种做法，首先要知道，两种情况：

对非集合类对象的 copy 与 mutableCopy 操作；
对集合类对象的 copy 与 mutableCopy 操作。
1. 对非集合类对象的copy操作：

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

```
[immutableObject copy] // 浅复制
[immutableObject mutableCopy] //深复制
[mutableObject copy] //深复制
[mutableObject mutableCopy] //深复制
```

比如以下代码：

NSMutableString *string = [NSMutableString stringWithString:@"origin"];//copy
NSString *stringCopy = [string copy];
查看内存，会发现 string、stringCopy 内存地址都不一样，说明此时都是做内容拷贝、深拷贝。即使你进行如下操作：

[string appendString:@"origion!"]
stringCopy 的值也不会因此改变，但是如果不使用 copy，stringCopy 的值就会被改变。 集合类对象以此类推。 所以，

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

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

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

NSMutableArray *array = [NSMutableArray arrayWithObjects:[NSMutableString 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。

14、UIView和 CALayer是什么关系?
UIView 显示在屏幕上归功于 CALayer，通过调用 drawRect 方法来渲染自身的内容，调节 CALayer 属性可以调整 UIView 的外观，UIView 继承自 UIResponder，比起CALayer 可以响应用户事件，Xcode6 之后可以方便的通过视图调试功能查看图层之间的关系。UIView 是 iOS 系统中界面元素的基础，所有的界面元素都继承自它。它内部是由Core Animation 来实现的，它真正的绘图部分，是由一个叫 CALayer(Core Animation Layer)的类来管理。UIView 本身，更像是一个 CALayer 的管理器，访问它的跟绘图和坐标有关的属性，如 frame，bounds 等，实际上内部都是访问它所在 CALayer 的相关属性。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，如：

```
– (class) layerClass {
// 使某个 UIView的子类使用 GL来进行绘制
return ([CAEAGLLayer class]);
}
```

UIView 的 CALayer 类似 UIView 的子 View 树形结构，也可以向它的 layer 上添加子layer，来完成某些特殊的显示。例如下面的代码会在目标 View 上敷上一层黑色的透明薄膜。

```
grayCover = [[CALayer alloc]init];
grayCover.backgroudColor = [[UIColor
blackColor]colorWithAlphaComponent:0.2].CGColor;
[self.layer addSubLayer:grayCover];
```

补充部分：这部分有深度了，大致了解一下吧，UIView 的 layer 树形在系统内部被系统维护着三份 copy1.逻辑树，就是代码里可以操纵的，例如更改 layer 的属性等等就在这一份2.动画树，这是一个中间层，系统正是在这一层上更改属性，进行各种渲染操作。3.显示树，这棵树的内容是当前正被显示在屏幕上的内容。这三棵树的逻辑结构都是一样的，区别只有各自的属性

15、static有什么作用?

答案:

1)函数体内 static 变量的作用范围为该函数体,不同于 auto 变量,该变量的内存只被分配一次,因此其值在下次调用时仍维持上次的值;
2)在模块内的 static 全局变量可以被模块内所用函数访问,但不能被模块外其它函数访问;
3)在模块内的 static 函数只可被这一模块内的其它函数调用,这个函数的使用范围被限制在声明它的模块内;
4)在类中的 static 成员变量属于整个类所拥有,对类的所有对象只有一份拷贝;
5)在类中的 static 成员函数属于整个类所拥有,这个函数不接收 this 指针,因而只能访问类的static 成员变量。

底层
16、main()之前的过程有哪些?
1）dyld 开始将程序二进制文件初始化
2）交由ImageLoader 读取 image，其中包含了我们的类，方法等各种符号（Class、Protocol 、Selector、 IMP）
3）由于runtime 向dyld 绑定了回调，当image加载到内存后，dyld会通知runtime进行处理
4）runtime 接手后调用map_images做解析和处理
5）接下来load_images 中调用call_load_methods方法，遍历所有加载进来的Class，按继承层次依次调用Class的+load和其他Category的+load方法
6）至此 所有的信息都被加载到内存中
7）最后dyld调用真正的main函数

注意：dyld会缓存上一次把信息加载内存的缓存，所以第二次比第一次启动快一点

17、KVO基本原理?
答:
1.KVO是基于runtime机制实现的

2.当某个类的属性对象第一次被观察时，系统就会在运行期动态地创建该类的一个派生类，在这个派生类中重写基类中任何被观察属性的setter 方法。派生类在被重写的setter方法内实现真正的通知机制

3.如果原类为Person，那么生成的派生类名为NSKVONotifying_Person

4.每个类对象中都有一个isa指针指向当前类，当一个类对象的第一次被观察，那么系统会偷偷将isa指针指向动态生成的派生类，从而在给被监控属性赋值时执行的是派

生类的setter方法

5.键值观察通知依赖于NSObject 的两个方法: willChangeValueForKey: 和 didChangevlueForKey:；在一个被观察属性发生改变之前，willChangeValueForKey:一定会被调用，这就 会记录旧的值。而当改变发生后，didChangeValueForKey:会被调用，继而 observeValueForKey:ofObject:change:context: 也会被调用。

18、： Swift 下的如何使用 KVC?

答:

```
/// 1. 定义一个类
class Animal  {
    var name = "Animal"
}

/// 2. 我们想使用kvc的方式改变实例的名字的时候，发现并没有setValue这个方法
let anim = Animal()
//anim.setValue  发现并没有setValue这个方法

/// 3. 怎么像oc一样可以使用kvc的方式来访问控制
// 步骤1:  继承NSObject
class Animal1 : NSObject {
    var name = "Animal1"
}

   // 步骤2：调用, 运行后发现报错 原因：swift4.0之前，这种方式是可以的，4.0的时候，减少隐式 @objc 自动推断， 也就是说，如果你要调用，必须显式声明，在方法前添加@objc, 或者在类前添加@objcMembers(这种情况会给所有的实例变量和方法都隐式的添加了@objc)
  Animal1().setValue("Dog", forKey: "name")

  /// 4. 除了这种可以实现KVC, 还有其它方法么，答案肯定是有的， 如下
   let anim = Animal()
   let key = \Animal.name
   anim[keyPath:key] = "Dog"
   print("\(anim.name)")
```

19、：Swift有哪些模式匹配?

模式匹配总结：
/// 通配符模式（Wildcard Pattern）
/// 如果你在 Swift 编码中使用了 _ 通配符，就可以表示你使用了通配符模式
/// 例如，下面这段代码在闭区间 1...10 中迭代，每次迭代都忽略该区间的当前值：
```
for _ in 0...10 {
    //print("hello")
}
```

```
// 标识符模式（Identifier Pattern）
// 定义变量或者常量时候，可以认为变量名和常量名就是一个标识符模式，用于接收和匹配一个特定类型的值：
let i = 1 // i 就是一个标识符模式
```

```
// 值绑定模式（Value-Binding Pattern）
// 值绑定在 if 语句和 switch 语句中用的较多。
// 比如 if let 和 case let, 还有可能是 case var。 let 和 var 分别对应绑定为常量和变量。
var str: String? = "nil"

if let v = str {
    // 使用v这个常量做什么事（use v to do something）
    //print("hello")
}
```

```
// 元组模式（Tuple Pattern）
// 顾名思义，元组模式就是匹配元组中元素的模式：
let tuple = ("world", 21)
switch tuple {
case ("hello", let num):
    print("第一个元素匹配，第二个元素可以直接使用\(num)")
case (_, 0...10):
    print("元组的第一个元素可以是任何的字符串，第二个元素在0~10之间")
case ("world", _):
```

```
    print("只要元组的第一个元素匹配， num可以是任何数")
default:
    print("default")
}


// 但是在 for-in 语句中，由于每次循环都需要明确匹配到具体的值，所以以下代码是错误的：
//let pts = [(0, 0), (0, 1), (0, 2), (1, 2)]
//for (x, 0) in pts {
//
//}

// 需要使用一个另外一个值绑定模式，来达成以上的逻辑：
let pts = [(0, 0), (0, 1), (0, 2), (1, 2)]
for (x, y) in pts {

}


//枚举用例模式（Enumeration Case Pattern）
//枚举用例模式是功能最强大的一个模式，也是整个模式匹配当中，应用面最广的模式，结合枚举中的关联值语法，可以做很多事情。先看一个简单的例子：
enum Direction {
    case up
    case down
    case left
    case right
}

let direction = Direction.up
switch direction {
case .up:
    print("up")
case .down:
    print("down")
case .left:
    print("left")
case .right:
    print("right")
}
```

```
//可选模式
//语句 case let x = y 模式允许你检查 y 是否能匹配 x。
//而 if case let x = y { ... } 严格等同于 switch y { case let x: ... }：当你只想与一条 case 匹配时，这种更紧凑的语法尤其有用。有多个 case 时更适合使用 switch。
//可选模式就是包含可选变量定义模式，在 if case、 for case、 switch-case 会用到。注意 if case let 和 if let的区别：

// 使用可选模式匹配
let a:Int? = nil
if case let x? = a {
    print("\(x)")
}



let arr:[Int?] = [nil, 1, nil, 3]

/// 只匹配非nil的元素
for case let element? in arr {
    //print("\(element)")
}



// 类型转换模式（Type-Casting Pattern）
// 使用 is 和 as 关键字的模式，就叫类型转换模式：
let tp : Any = 0

switch tp {
case let a as Int :
    print("是Int类型")
case let a as String :
    print("是String类型")
default:
    print("其它类型")
}

switch tp {
```

```
case is Int:
    print("是Int类型")
case is String:
    print("是String类型")
default:
    print("其它类型")
}


//表达式模式
//表达式模式只出现在 switch-case 中，Swift的模式匹配是基于~=操作符的，如果表达式的~=值返回true则匹配成功。可以自定义~=运算符，自定义类型的表达式匹配行为:

/// 自定义模式匹配: 注意必需要写两个参数， 不然会报错
struct Affine {
    var a: Int
    var b: Int
}

func ~= (lhs: Affine, rhs: Int) -> Bool {
    return rhs % lhs.a != lhs.b
}

switch 5 {
case Affine(a: 2, b: 0): print("Even number")
case Affine(a: 3, b: 1): print("3x+1")
case Affine(a: 3, b: 2): print("3x+2")
default: print("Other")
}
```

20、objc在向一个对象发送消息时，发生了什么?
答案：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。

21、静态库的原理是什么？你有没有自己写过静态编译库，遇到了哪些问题?
库本质上讲是一种可执行的二进制格式，可以载入内存中执行。是程序代码的集合，

共享代码的一种方式。
静态库是闭源库，不公开源代码，都是编译后的二进制文件，不暴露具体实现。
静态库 一般都是以 .a 或者 .framework 形式存在。
静态库编译的文件比较大，因为整个函数库的数据都会被整合到代码中，这样的好处就是编译后的程序不需要外部的函数库支持，不好的一点就是如果改变静态函数库，就需要程序重新编译。多次使用就有多份冗余拷贝。
使用静态库的好处：模块化分工合作、可重用、避免少量改动导致大量的重复编译链接。
一般公司都会有核心开发团队和普通开发团队，然后公司的核心业务由核心开发团队写成静态库然后让普通开发团队调用，这样就算普通开发团队离职也带不走公司的核心业务代码。一般核心开发团队是不会离职的。
有静态库自然就有动态库了。这里所谓的静态和动态是相对编译期和运行期的。静态库在程序编译时会被链接到代码中，程序运行时将不再需要改静态库，而动态库在编译时不会被链接到代码中，只有程序运行时才会被载入，所以 hook 别人程序或者说做插件都是运用了 runtime 机制，然后动态库注入修改的。
制作 .a 文件时候，要注意 CPU 架构的支持，i386、X86_64、 armv7、armv7s。查看可以通过命令“ lipo -info 静态库名称”。模拟器 .a 文件和真机 .a 文件合并可以通过 "lipo -create 模拟器静态库1名 真机静态库2名 -output 新静态库名称"
一些坑
命名不要太随意，毕竟是被别人拿过去用的要能看懂。
framework中用到了NSClassFromString，但是转换出来的class 一直为nil。解决方法：在主工程的【Other Linker Flags】需要添加参数【-ObjC]即可。
如果Xcode找不到框架的头文件，你可能是忘记将它们声明为public了。 解决方法：进入target的Build Phases页，展开Copy Headers项，把需要public的头文件从Project或Private部分拖拽到Public部分。
尽量不要用 xib 。由于静态框架采用静态链接，linker会剔除所有它认为无用的代码。不幸的是，linker不会检查xib文件，因此如果类是在xib中引用，而没有在O-C代码中引用，linker将从最终的可执行文件中删除类。这是linker的问题，不是框架的问题（当你编译一个静态库时也会发生这个问题）。苹果内置框架不会发生这个问题，因为他们是运行时动态加载的，存在于iOS设备固件中的动态库是不可能被删除的。
有两个解决的办法：
1、 让框架的最终用户关闭linker的优化选项，通过在他们的项目的Other Linker Flags中添加-ObjC和-all_load。
2、 在框架的另一个类中加一个该类的代码引用。例如，假设你有个MyTextField类，被linker剔除了。假设你还有一个MyViewController，它在xib中使用了MyTextField，MyViewController并没有被剔除。你应该这样做：
在MyTextField中：
+(void)forceLinkerLoad_ {}
在MyViewController中：
+(void)initialize {[MyTextField forceLinkerLoad_];}
他们仍然需要添加-ObjC到linker设置，但不需要强制all_load了。

第2种方法需要你多做一点工作，但却让最终用户避免在使用你的框架时关闭linker优化（关闭linker优化会导致object文件膨胀）。

22、runloop是来做什么的？runloop和线程有什么关系？主线程默认开启了runloop么？子线程呢?

runloop:字面意思就是跑圈，其实也就是一个循环跑圈，用来处理线程里面的事件和消息。

runloop和线程的关系：每个线程如果想继续运行，不被释放，就必须有一个runloop来不停的跑圈，以来处理线程里面的各个事件和消息。

主线程默认是开启一个runloop。也就是这个runloop才能保证我们程序正常的运行。子线程是默认没有开始runloop的

8:runloop的mode是用来做什么的？有几种mode?

model:是runloop里面的模式，不同的模式下的runloop处理的事件和消息有一定的差别。

系统默认注册了5个Mode:

（1）kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。

（2）UITrackingRunLoopMode: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。

（3）UIInitializationRunLoopMode: 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用。

（4）GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到。

（5）kCFRunLoopCommonModes: 这是一个占位的 Mode，没有实际作用。

注意iOS 对以上5中model进行了封装

NSDefaultRunLoopMode;

NSRunLoopCommonModes

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

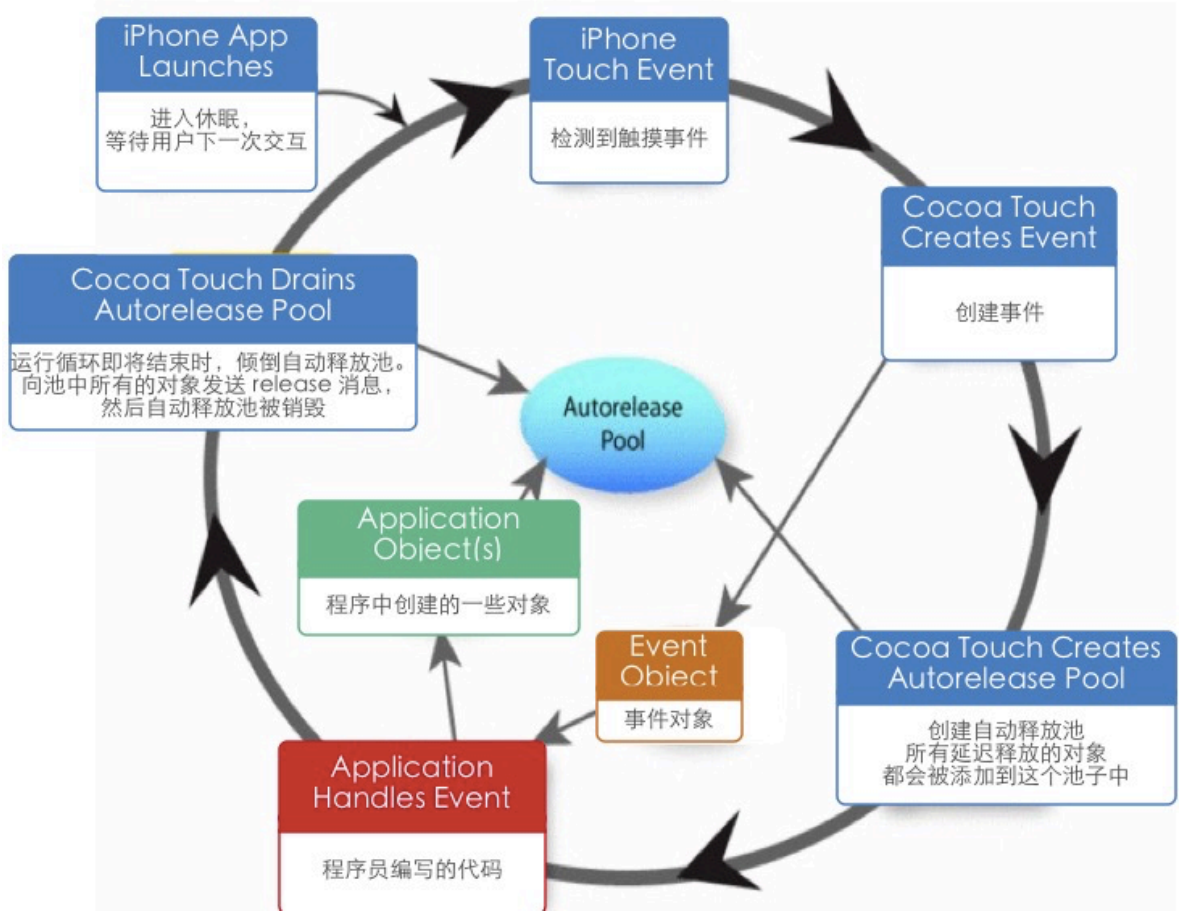
分两种情况：手动干预释放时机、系统自动去释放。

1. 手动干预释放时机--指定autoreleasepool 就是所谓的：当前作用域大括号结束时释放。

2. 系统自动去释放--不手动指定autoreleasepool

Autorelease对象出了作用域之后，会被添加到最近一次创建的自动释放池中，并会在

当前的 runloop 迭代结束时释放。
释放的时机总结起来，可以用下图来表示：



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?

A:运行循环检测到事件并启动后，就会创建自动释放池。

B:子线程的 runloop 默认是不工作，无法主动创建，必须手动创建。(拓展: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)

C: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 这种面默认的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。

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

分两种情况：手动干预释放时机、系统自动去释放。

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。
系统自动去释放--不手动指定autoreleasepool

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

释放的时机总结起来，可以用下图来表示：

autoreleasepool与 runloop 的关系图

下面对这张图进行详细的解释：

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

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

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

在一次完整的运行循环结束之前，会被销毁。

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

子线程的 runloop 默认是不工作，无法主动创建，必须手动创建。

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

但对于 blockOperation 和 invocationOperation 这种默认的Operation ，系统已经帮我

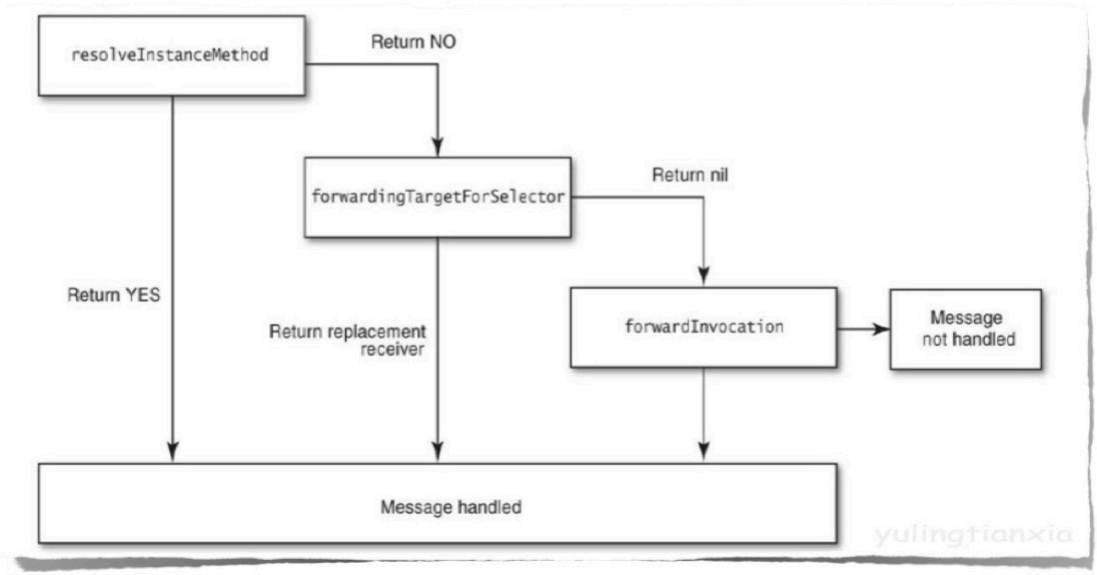
们封装好了，不需要手动创建自动释放池。

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

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

25、OC完整的消息转发机制+代码实现【暴击】。

>消息转发分为两大阶段。第一阶段先征询接收者，所属的类，看其是否能动态添加方法，以处理当前这个“未知的选择子”，这叫做“动态方法解析”。
第二阶段涉及“完整的消息转发机制（full forwarding mechanism）”如果运行期系统已经执行完第一阶段，此时，运行期系统会请求我接收者以其它手段来处理消息。可以细分两小步。1.首先查找有没有replacement receiver进行处理。若无，2.运行期系统把Selector相关信息封装到NSInvocation对象中，再给一次机会，若依旧未处理则让NSObject调用doNotReconizeSelector：
接收者在每一步中均有机会处理消息，步骤越往后，处理消息的代价越大。最好能在第一步就处理完，这样Runtime System可以将方法缓存起来，第三步除了修改目标相比第二步还要创建处理NSInvocation。

> 面试题目：OC完整的消息转发机制+代码实现【暴击】。
>
> 消息转发分为两大阶段。第一阶段先征询接收者，所属的类，看其是否能动态添加方法，以处理当前这个“未知的选择子”，这叫做“动态方法解析”。
> 第二阶段涉及“完整的消息转发机制（full forwarding mechanism）”如果运行期系统已经执行完第一阶段，此时，运行期系统会请求我接收者以其它手段来处理消息。可以细分两小步。1.首先查找有没有replacement receiver进行处理。若无，2.运行期系统把Selector相关信息封装到NSInvocation对象中，再给一次机会，若依旧未处理则让NSObject调用doNotReconizeSelector：
> 接收者在每一步中均有机会处理消息，步骤越往后，处理消息的代价越大。最好能在第一步就处理完，这样Runtime System可以将方法缓存起来，第三步除了修改目标相比第二步还要创建处理NSInvocation。
>
> 
>

26、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scrllView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

```
//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:NSRunLoopCommonMo
des];
```

27、如何手动触发一个value的KVO
所谓的“手动触发”是区别于“自动触发”：

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

想知道如何手动触发，必须知道自动触发 KVO 的原理：

键值观察通知依赖于 NSObject 的两个方法: willChangeValueForKey: 和 didChangevlueForKey: 。在一个被观察属性发生改变之前，willChangeValueForKey: 一定会被调用，这就 会记录旧的值。而当改变发生后，observeValueForKey:ofObject:change:context: 会被调用，继而 didChangeValueForKey: 也会被调用。如果可以手动实现这些调用，就可以实现“手动触发”了。

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

具体做法如下：

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

相关代码已放在仓库里。

```
//  .m文件
//  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 https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOCompliance.html#//apple_ref/doc/uid/20002178-SW3

28、如何对定位和分析项目中影响性能的地方？以及如何进行性能优化?
定位方法：
instruments
　　在iOS上进行性能分析的时候，首先考虑借助instruments这个利器分析出问题出在哪，不要凭空想象，不然你可能把精力花在了1%的问题上，最后发现其实啥都没优化，比如要查看程序哪些部分最耗时，可以使用Time Profiler，要查看内存是否泄漏了，可以使用Leaks等。关于instruments网上有很多资料，作为一个合格iOS开发者，熟悉这个工具还是很有必要的。

优化建议：
1.用ARC管理内存
* ARC(Automatic Reference Counting, 自动引用计数)和iOS5一起发布，它避免了最常见的也就是经常是由于我们忘记释放内存所造成的内存泄露。它自动为你管理retain和release的过程，所以你就不必去手动干预了。下面是你会经常用来去创建一个View的代码段：UIView *view = [[UIView alloc] init];
*　　// ...
*　[self.view addSubview:view];
*　　[view release];
* 忘掉代码段结尾的release简直像记得吃饭一样简单。而ARC会自动在底层为你做这些工作。除了帮你避免内存泄露，ARC还可以帮你提高性能，它能保证释放掉不再需要的对象的内存。这都啥年代了，你应该在你的所有项目里使用ARC!
2.在正确的地方使用 reuseIdentifier
* 一个开发中常见的错误就是没有给UITableViewCells， UICollectionViewCells，甚至是UITableViewHeaderFooterViews设置正确的reuseIdentifier。
* 为了性能最优化，table view用 tableView:cellForRowAtIndexPath: 为rows分配cells的时候，它的数据应该重用自UITableViewCell。 一个table view维持一个队列的数据可重用的UITableViewCell对象。不使用reuseIdentifier的话，每显示一行table view就不得不设置全新的cell。这对性能的影响可是相当大的，尤其会使app的滚动体验大打折扣。
* 自iOS6起，除了UICollectionView的cells和补充views，你也应该在header和footer views中使用reuseIdentifiers
3.尽量把views设置为完全不透明
* 如果你有透明的Views你应该设置它们的opaque(不透明)属性为YES。例如一个黑色半透明的可以设置为一个灰色不透明的View替代.原因是这会使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。
* Apple的文档对于为图片设置透明属性的描述是：
* (opaque)这个属性给渲染系统提供了一个如何处理这个view的提示。如果设为YES， 渲染系统就认为这个view是完全不透明的，这使得渲染系统优化一些渲染过程和提高性能。如果设置为NO，渲染系统正常地和其它内容组成这个View。默认值是YES。

* 在相对比较静止的画面中，设置这个属性不会有太大影响。然而当这个view嵌在scroll view里边，或者是一个复杂动画的一部分，不设置这个属性的话会在很大程度上影响app的性能。
* 换种说法，大家可能更好理解：只要一个视图的不透明度小于1,就会导致blending.blending操作在iOS的图形处理器（GPU）中完成的,blending主要指的是混合像素颜色的计算。举个例子,我们把两个图层叠加在一起,如果第一个图层的有透明效果,则最终像素的颜色计算需要将第二个图层也考虑进来。这一过程即为Blending。为什么Blending会导致性能的损失？原因是很直观的,如果一个图层是完全不透明的,则系统直接显示该图层的颜色即可。而如果图层是带透明效果的,则会引入更多的计算,因为需要把下面的图层也包括进来,进行混合后颜色的计算。

4. 避免过于庞大的XIB
* iOS5中加入的Storyboards(分镜)正在快速取代XIB。然而XIB在一些场景中仍然很有用。比如你的app需要适应iOS5之前的设备，或者你有一个自定义的可重用的view,你就不可避免地要用到他们。
* 如果你不得不XIB的话，使他们尽量简单。尝试为每个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。
* 需要注意的是，当你加载一个XIB的时候所有内容都被放在了内存里，包括任何图片。如果有一个不会即刻用到的view，你这就是在浪费宝贵的内存资源了。Storyboards就是另一码事儿了，storyboard仅在需要时实例化一个view controller.
* 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。在OS X中，图片和声音资源被缓存在named cache中以便将来用到时获取。在iOS中，仅图片资源会被存进named caches。取决于你所在的平台，使用NSImage 或UIImage 的imageNamed:方法来获取图片资源。

5. 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。一直使用主线程的风险就是如果你的代码真的block了主线程，你的app会失去反应
* 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。或者使用像 AFNetworking这样的框架来异步地做这些操作。如果你需要做其它类型的需要耗费巨大资源的操作(比如时间敏感的计算或者存储读写)那就用 Grand Central Dispatch，或者 NSOperation 和 NSOperationQueues.你可以使用NSURLConnection异步地做网络操作: + (void)sendAsynchronousRequest: (NSURLRequest *)request queue:(NSOperationQueue *)queue completionHandler:(void (^)(NSURLResponse*, NSData*, NSError*))handler

6. 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，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。

7. 选择正确的Collection
学会选择对业务场景最合适的类或者对象是写出能效高的代码的基础。当处理collections时这句话尤其正确。
Apple有一个 Collections Programming Topics 的文档详尽介绍了可用的classes间的差别和你该在哪些场景中使用它们。这对于任何使用collections的人来说是一个必读的文档。
呵呵，我就知道你因为太长没看…这是一些常见collection的总结：
* 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。
8. 打开gzip压缩
* 大量app依赖于远端资源和第三方API，你可能会开发一个需要从远端下载XML, JSON, HTML或者其它格式的app。
* 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。一个用户现在还在edge网络，下一分钟可能就切换到了3G。不论什么场景，你肯定不想让你的用户等太长时间。
* 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。好消息是，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。像Google App Engine这些云服务提供者也已经支持了压缩输出。

29、串行并行，异步同步的区别?

先来说一个队列和任务：
队列分为串行和并行
任务的执行分为同步和异步
这两两组合就成为了串行队列同步执行，串行队列异步执行，并行队列同步执行，并行队列异步执行
而异步是多线程的代名词，异步在实际引用中会开启新的线程，执行耗时操作。

那我们先来知道一个非常重要的事情：
------- 队列只是负责任务的调度，而不负责任务的执行 ---------

-------- 任务是在线程中执行的 ----------

队列和任务的特点：
队列的特点：先进先出，排在前面的任务最先执行，
串行队列：任务按照顺序被调度，前一个任务不执行完毕，队列不会调度
并行队列：只要有空闲的线程，队列就会调度当前任务，交给线程去执行，不需要考虑前面是都有任务在执行，只要有线程可以利用，队列就会调度任务。
主队列：专门用来在主线程调度任务的队列，所以主队列的任务都要在主线程来执行，主队列会随着程序的启动一起创建，我们只需get即可
全局队列：是系统为了方便程序员开发提供的，其工作表现与并发队列一致，那么全局队列跟并发队列的区别是什么呢?
1.全局队列：无论ARC还是MRC都不需要考录释放，因为系统提供的我们只需要get就可以了
2.并发队列：再MRC下，并发队列创建出来后，需要手动释放dispatch_release()
同步执行：不会开启新的线程，任务按顺序执行
异步执行：会开启新的线程，任务可以并发的执行

30、线程是什么？进程是什么？二者有什么区别和联系?
答案：一个程序至少有一个进程,一个进程至少有一个线程：
进程：一个程序的一次运行，在执行过程中拥有独立的内存单元，而多个线程共享一块内存
线程：线程是指进程内的一个执行单元。
联系：线程是进程的基本组成单位
区别：(1)调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位
(2)并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行
(3)拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源. (4)系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。
举例说明：操作系统有多个软件在运行（QQ、office、音乐等），这些都是一个个进程，而每个进程里又有好多线程（比如QQ，你可以同时聊天，发送文件等）

31、RunLoop是什么?
答案：Run Loop是一让线程能随时处理事件但不退出的机制。RunLoop 实际上是一个对象，这个对象管理了其需要处理的事件和消息，并提供了一个入口函数来执行 Event Loop 的逻辑。线程执行了这个函数后，就会一直处于这个函数内部“接受消息->等待->处理”的循环中，直到这个循环结束（比如传入 quit 的消息），函数返回。让线程在没有处理消息时休眠以避免资源占用、在有消息到来时立刻被唤醒。

OSX/iOS 系统中，提供了两个这样的对象：NSRunLoop 和 CFRunLoopRef。

CFRunLoopRef 是在 CoreFoundation 框架内的，它提供了纯 C 函数的 API，所有这些 API 都是线程安全的。NSRunLoop 是基于 CFRunLoopRef 的封装，提供了面向对象的 API，但是这些 API 不是线程安全的。

线程和 RunLoop 之间是一一对应的，其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 RunLoop，如果你不主动获取，那它一直都不会有。RunLoop 的创建是发生在第一次获取时，RunLoop 的销毁是发生在线程结束时。你只能在一个线程的内部获取其 RunLoop（主线程除外）。

32、假设有一个字符串aabcad，请写一段程序，去掉字符串中不相邻的重复字符串，即上述字符串处理之后的输出结果为：aabcd
答案：NSMutableString * str = [[NSMutableString alloc]initWithFormat;@“aabcad”];

```
for (int i = 0 ,i < str.length – 1 ;i++){
    unsigned char a = [str characterAtIndex:i];
    for (int j = i + 1 ,j < str.length ,j++){
        unsigned char b = [str characterAtIndex:j];
        if (a == b ){
            if (j == i + 1){
                }else{
                [str deleteCharactersInRange:NSMakeRange(j, 1)];
                }
            }
        }
    }
NSLog(@“%@”,str);
```

33、@autoclosure（自动闭包）

1:自动闭包，顾名思义是一种自动创建的闭包，用于包装函数参数的表达式，可以说是一种简便语法.
2:自动闭包不接受任何参数，被调用时会返回被包装在其中的表达式的值。
3:自动闭包的好处之二是让你能够延迟求值,因为代码段不会被执行直到你调用这个闭包，这样你就可以控制代码什么时候执行。
4：含有autoclosure特性的声明同时也具有noescape的特性，及默认是非逃逸闭包，除非传递可选参数escaping.如果传递了该参数，那么将可以在闭包之外进行操作闭包,形式为：请使用@autoclosure(escaping)。

实战操作理解:

```
/// 我们定义有一个方法接受一个闭包，当闭包执行的结果为true的时候进行打印：
func method(result:() -> Bool) {
    if result() {
        print("method")
    }
}

/// 1. 直接挑用方法
method { () -> Bool in
    return true
}

/// 2. 闭包在园括号里
method(result: {return true})

/// 3. 使用尾部闭包方式，闭包体在圆括号之外
method(){return true}

/// 4. 在 Swift 中对闭包的用法可以进行一些简化，在这种情况下我们可以省略掉return，写成：
method(){true}

/// 5:还可以更近一步，因为这个闭包是最后一个参数，所以可以使用尾随闭包(trailing closure) 的方式把大括号拿出来，然后省略括号，变成：
method {true}



//=========================================//
/// 但是不管哪种方式，表达上不太清晰，看起来不舒服。于是@autoclosure就登场了。我们可以改换方法参数，在参数名前面加上@autoclosure关键字：
func method1( result:@autoclosure ()->Bool) {
    if result() {
        print("method1")
    }
}

/// 调用下，我们看一下
method1(result: true)
/// 上面调用是不是舒服多了，直接进行调用了，Swift 将会把 true 这个表达式自动转换为 () -> Bool。这样我们就得到了一个写法简单，表意清楚的式子。
```

/// 1个闭包优势可能不是那么的明显，如果有多个闭包，那么优势就明显了，而@autoclosure是可以修饰任何位置的参数：

```
/// 我们先看看不加@autoclosure，使用时是什么样子
func method3(result1: ()->Bool, result2:  ()->Bool) {
    if result1() && result2() {
        print("method3 ... 0")
        return
    }
    print("method3 ... 1")
}

method3(result1: {5 > 1}, result2: {4 > 3})

/// 我们加上@autoclosure之后对比下：
func method4(result1: @autoclosure ()->Bool, result2:  @autoclosure ()->Bool) {
    if result1() && result2() {
        print("method4 ... 0")
        return
    }
    print("method4 ... 1")
}

method4(result1: 5 > 1, result2: 4 > 3)
```

34、iOS app启动如何优化?

1. 我们可以通过在 Xcode 中 Edit scheme –> Run –> Auguments 将环境变量 DYLD_PRINT_STATISTICS 设为 1,在控制台看到main()函数之前的启动时间。

2. 分解优化目标 分步达到优化目的　　1). 耗时操作异步处理

2). 如果启动流程依赖网络请求回来才能继续,那么需要考虑网络极差情况下的启动速度

3). 如果APP有loading广告页并且对分辨率的要求较高,请尝试做缓存吧

4). 主页面Controller中的viewDidLoad和viewWillAppear方法中尽量少做事情

5). 排查清理项目中未使用到的类库以及Framework

6). 删减合并一些OC类,删减没有用到或者可以不用的静态变量、方法等

7). 轻量化+load方法中的内容,可延迟到+initialize中

1. 实现一个函数，输入是任一整数，输出要返回输入的整数 乘以 2

这道题很多人上来就这样写：

```
func mult(_ num: Int) –> Int {
    return num * 2
```

}

接下来面试官会说，那假如我要实现乘以4 呢？程序员想了一想，又定义了另一个方法：

```
func mult(_ num: Int) -> Int {
    return num * 4
}
```

这时面试官会问，假如我要实现返回 乘以 6, 乘以 8 的操作呢？能不能只定义一次方法呢？正确的写法是利用 Swift 的 柯里化特性：

```
func mult(_ num: Int) -> (Int) -> Int {
  return { val in
    return num * val
  }
}

let multTwo = mult(2), multFour = mult(4), multSix = mult(6), multEight = mult(8)
```

35、swift面试题：

教育类的app, 显示分数的地方，统计平均分的时候需要显示小数位，并且不能四舍五入，同时为了界面美观，不能出现小数点后面都为0的情况。

题目：Float类型，转换成字符串输出，要求保留N位小数, 特殊情况，如果转换之后结果为XX.00(0的个数=保留的小数的个数), 则返回XX

例子： 11.11保留1位 = "11.1" 11.01保留1位 = "11" 11.001保留2位 = "11"

```
extension Float {
    func floatWithDecimal(_ num : Int) -> String {
        let floatNum = Float(Int(self * powf(10, Float(num)))) / powf(10, Float(num))
        return (floatNum * powf(10, Float(num))).truncatingRemainder(dividingBy: powf(10, Float(num)))
            == 0 ? "\(Int(floatNum))" : "\(floatNum)"
    }
}
```

36、怎样防止反编译?

本地数据加密

对NSUserDefaults，sqlite存储文件数据加密，保护帐号和关键信息

URL编码加密

对程序中出现的URL进行编码加密，防止URL被静态分析

网络传输数据加密

对客户端传输数据提供加密方案，有效防止通过网络接口的拦截获取数据

方法体，方法名高级混淆
对应用程序的方法名和方法体进行混淆，保证源码被逆向后无法解析代码
程序结构混排加密
对应用程序逻辑结构进行打乱混排，保证源码可读性降到最低

37、UITableView性能优化，超实用

1. Cell重用
1.1>数据源方法优化
– (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath;
在可见的页面会重复绘制页面，每次刷新显示都会去创建新的Cell，非常耗费性能。
解决方案：首先创建一个静态变量reuseID（代理方法返回Cell会调用很多次，防止重复创建，static保证只会被创建一次，提高性能），然后，从缓存池中取相应identifier的Cell并更新数据，如果没有，才开始alloc新的Cell，并用identifier标识Cell。每个Cell都会注册一个identifier（重用标识符）放入缓存池，当需要调用的时候就直接从缓存池里找对应的id，当不需要时就放入缓存池等待调用。（移出屏幕的Cell才会放入缓存池中，并不会被release）所以在数据源方法中做出如下优化：
// 调用次数太多，static 保证只创建一次reuseID，提高性能
static NSString *reuseID = “reuseCellID”;
// 缓存池中取已经创建的cell
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:reuseID];
1.2>缓存池的实现
当Cell要alloc时，UITableView会在堆中开辟一段内存以供Cell缓存之用。Cell的重用通过identifier标识不同类型的Cell，由此可以推断出，缓存池外层可能是一个可变字典，通过key来取出内部的Cell，而缓存池为存储不同高度、不同类型（包含图片、Label等）的Cell，可以推断出缓存池的字典内部可能是一个可变数组，用来存放不同类型的Cell，缓存池中只会保存已经被移出屏幕的不同类型的Cell。
1.3>缓存池获取可重用Cell两个方法的区别
–(nullable __kindof UITableViewCell *)dequeueReusableCellWithIdentifier:
(NSString *)identifier;
这个方法会查询可重用Cell，如果注册了原型Cell，能够查询到，否则，返回nil；而且需要判断if（cell == nil），才会创建Cell，不推荐
–(__kindof UITableViewCell *)dequeueReusableCellWithIdentifier:(NSString
*)identifier forIndexPath:(NSIndexPath *)indexPath NS_AVAILABLE_IOS(6_0);
使用这个方法之前，必须通过xib（storyboard）或是Class（纯代码）注册可重用Cell，而且这个方法一定会返回一个Cell
注册Cell
– (void)registerNib:(nullable UINib *)nib forCellReuseIdentifier:(NSString

*)identifier NS_AVAILABLE_IOS(5_0);
– (void)registerClass:(nullable Class)cellClass forCellReuseIdentifier:(NSString *)identifier NS_AVAILABLE_IOS(6_0);
好处：如果缓冲区 Cell 不存在，会使用原型 Cell 实例化一个新的 Cell，不需要再判断，同时代码结构更清晰。
2. 定义一种(尽量少)类型的Cell及善用hidden隐藏(显示)subviews
2.1>一种类型的Cell
分析Cell结构，尽可能的将 相同内容的抽取到一种样式Cell中，前面已经提到了Cell的重用机制，这样就能保证UITbaleView要显示多少内容，真正创建出的Cell可能只比屏幕显示的Cell多一点。虽然Cell的'体积'可能会大点，但是因为Cell的数量不会很多，完全可以接受的。好处：
* 减少代码量，减少Nib文件的数量，统一一个Nib文件定义Cell，容易修改、维护
* 基于Cell的重用，真正运行时铺满屏幕所需的Cell数量大致是固定的，设为N个。所以如果如果只有一种Cell，那就是只有N个Cell的实例；但是如果有M种Cell，那么运行时最多可能会是“M x N = MN”个Cell的实例，虽然可能并不会占用太多内存，但是能少点不是更好吗。
2.2>善用hidden隐藏(显示)subviews
只定义一种Cell，那该如何显示不同类型的内容呢？答案就是，把所有不同类型的view都定义好，放在cell里面，通过hidden显示、隐藏，来显示不同类型的内容。毕竟，在用户快速滑动中，只是单纯的显示、隐藏subview比实时创建要快得多。

3. 提前计算并缓存Cell的高度
在iOS中，不设UITableViewCell的预估行高的情况下，会优先调用”tableView:heightForRowAtIndexPath:”方法，获取每个Cell的即将显示的高度，从而确定UITableView的布局，实际就是要获取contentSize（UITableView继承自UIScrollView,只有获取滚动区域，才能实现滚动）,然后才调用”tableView:cellForRowAtIndexPath”,获取每个Cell，进行赋值。如果项目中模块有10000个Cell需要显示，可想而知…
解决方案：我个人认为，可以创建一个frame模型，提前计算每个Cell的高度。参考其中一篇博客的时候，在解决这个问题的时候，可以将计算Cell的高度放入数据模型，但这与MVC设计模式可能稍微有点冲突，这个时候我就想到MVVM这种设计模式，这个时候才能稍微有点MVVM这种设计模式的优点（其实还是很不理解的），可以讲计算Cell高度放入ViewModel（视图模型）中，让Model（数据模型）只负责处理数据。
4.异步绘制（自定义Cell绘制）
遇到比较复杂的界面的时候，如复杂点的图文混排，上面的那种优化行高的方式可能就不能满足要求了，当然了，由于我的开发经验尚短，说实话，还没遇到要将自定义的Cell重新绘制

5.滑动时，按需加载
开发的过程中，自定义Cell的种类千奇百怪，但Cell本来就是用来显示数据的，不说

100%带有图片，也差不多，这个时候就要考虑，下滑的过程中可能会有点卡顿，尤其网络不好的时候，异步加载图片是个程序员都会想到，但是如果给每个循环对象都加上异步加载，开启的线程太多，一样会卡顿，我记得好像线程条数一般3-5条，最多也就6条吧。这个时候利用UIScrollViewDelegate两个代理方法就能很好地解决这个问题。
– (void)scrollViewDidEndDragging:(UIScrollView *)scrollView willDecelerate:(BOOL)decelerate
– (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
6.缓存View
　当Cell中的部分View是非常独立的，并且不便于重用的，而且“体积”非常小，在内存可控的前提下，我们完全可以将这些view缓存起来。当然也是缓存在模型中。

7.避免大量的图片缩放、颜色渐变等，尽量显示“大小刚好合适的图片资源”
8.避免同步的从网络、文件获取数据，Cell内实现的内容来自web，使用异步加载，缓存请求结果
9.渲染
　9.1>减少subviews的个数和层级
　　子控件的层级越深，渲染到屏幕上所需要的计算量就越大；如多用drawRect绘制元素，替代用view显示
　9.2>少用subviews的透明图层
　　对于不透明的View，设置opaque为YES，这样在绘制该View时，就不需要考虑被View覆盖的其他内容（尽量设置Cell的view为opaque，避免GPU对Cell下面的内容也进行绘制）
　9.3>避免CALayer特效（shadowPath）
　　给Cell中View加阴影会引起性能问题，如下面代码会导致滚动时有明显的卡顿：
```
view.layer.shadowColor = color.CGColor;
view.layer.shadowOffset = offset;
view.layer.shadowOpacity = 1;
view.layer.shadowRadius = radius;
```

线程：
37、 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。一直使用主线程的风险就是如果你的代码真的block了主线程，你的app会失去反应
* 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。或者使用像 AFNetworking这样的框架来异步地做这些操作。如果你需要做其它类型的需要耗费巨大资源的操作(比如时间敏感的计算或者存储读写)那就用 Grand Central Dispatch，或者 NSOperation 和 NSOperationQueues.你可以使用NSURLConnection异步地做网络操作：+ (void)sendAsynchronousRequest:

(NSURLRequest *)request queue:(NSOperationQueue *)queue
completionHandler:(void (^)(NSURLResponse*, NSData*, NSError*))handler

6. 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，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。

38、谈谈你对多线程开发的理解？ios中有几种实现多线程的方法?
答案：
好处：
1.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
2.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
3.程序的运行速度可能加快
4·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：
1.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。
2.更多的线程需要更多的内存空间。
3.线程的中止需要考虑其对程序运行的影响。
4.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：
NSObject类方法
NSThread
NSOperation
GCD

39、进程和线程的区别？同步异步的区别？并行和并发的区别?
参考答案：
进程：是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.
线程：是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

同步：阻塞当前线程操作，不能开辟线程。
异步：不阻碍线程继续操作，可以开辟线程来执行任务。
并发：当有多个线程在操作时,如果系统只有一个CPU,则它根本不可能真正同时进行一个以上的线程，它只能把CPU运行时间划分成若干个时间段,再将时间 段分配给各个线程执行，在一个时间段的线程代码运行时，其它线程处于挂起状。.这种方式我们称之为并发(Concurrent)。
并行：当系统有一个以上CPU时,则线程的操作有可能非并发。当一个CPU执行一个线程时，另一个CPU可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。
区别：并发和并行是即相似又有区别的两个概念，并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔内发生。在多道程序环境下，并发性是指在一段时间内宏观上有多个程序在同时运行，但在单处理机系统中，每一时刻却仅能有一道程序执行，故微观上这些程序只能是分时地交替执行。倘若在计算机系统中有多个处理机，则这些可以并发执行的程序便可被分配到多个处理机上，实现并行执行，即利用每个处理机来处理一个可并发执行的程序，这样，多个程序便可以同时执行。

40、ViewController生命周期
按照执行顺序排列：
1. initWithCoder：通过nib文件初始化时触发。
2. awakeFromNib：nib文件被加载的时候，会发生一个awakeFromNib的消息到nib文件中的每个对象。
3. loadView：开始加载视图控制器自带的view。
4. viewDidLoad：视图控制器的view被加载完成。
5. viewWillAppear：视图控制器的view将要显示在window上。
6. updateViewConstraints：视图控制器的view开始更新AutoLayout约束。
7. viewWillLayoutSubviews：视图控制器的view将要更新内容视图的位置。
8. viewDidLayoutSubviews：视图控制器的view已经更新视图的位置。
9. viewDidAppear：视图控制器的view已经展示到window上。
10. viewWillDisappear：视图控制器的view将要从window上消失。
11. viewDidDisappear：视图控制器的view已经从window上消失。

41、iOS 中的多线程
iOS中的多线程，是Cocoa框架下的多线程，通过Cocoa的封装，可以让我们更为方便的使用线程，做过C++的同学可能会对线程有更多的理解，比如线程的创立，信号量、共享变量有认识，Cocoa框架下会方便很多，它对线程做了封装，有些封装，可以让我们创建的对象，本身便拥有线程，也就是线程的对象化抽象，从而减少我们的工程，提供程序的健壮性。
* GCD是(Grand Central Dispatch)的缩写 ，从系统级别提供的一个易用地多线程类

库，具有运行时的特点，能充分利用多核心硬件。GCD的API接口为C语言的函数，函数参数中多数有Block，关于Block的使用参看这里，为我们提供强大的“接口”，对于GCD的使用参见本文
* NSOperation与Queue
NSOperation是一个抽象类，它封装了线程的细节实现，我们可以通过子类化该对象，加上NSQueue来同面向对象的思维，管理多线程程序。具体可参看这里：一个基于NSOperation的多线程网络访问的项目。
* NSThread
NSThread是一个控制线程执行的对象，它不如NSOperation抽象，通过它我们可以方便的得到一个线程，并控制它。但NSThread的线程之间的并发控制，是需要我们自己来控制的，可以通过NSCondition实现。
参看 iOS多线程编程之NSThread的使用

42、内存管理的几条原则时什么？按照默认法则.那些关键字生成的对象需要手动释放？在和property结合的时候怎样有效的避免内存泄露?
谁申请，谁释放
遵循Cocoa Touch的使用原则；
内存管理主要要避免“过早释放”和“内存泄漏”，对于“过早释放”需要注意@property设置特性时，一定要用对特性关键字，对于“内存泄漏”，一定要申请了要负责释放，要细心。
关键字alloc 或new 生成的对象需要手动释放；
设置正确的property属性，对于retain需要在合适的地方释放，

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。

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。 ）

44、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。

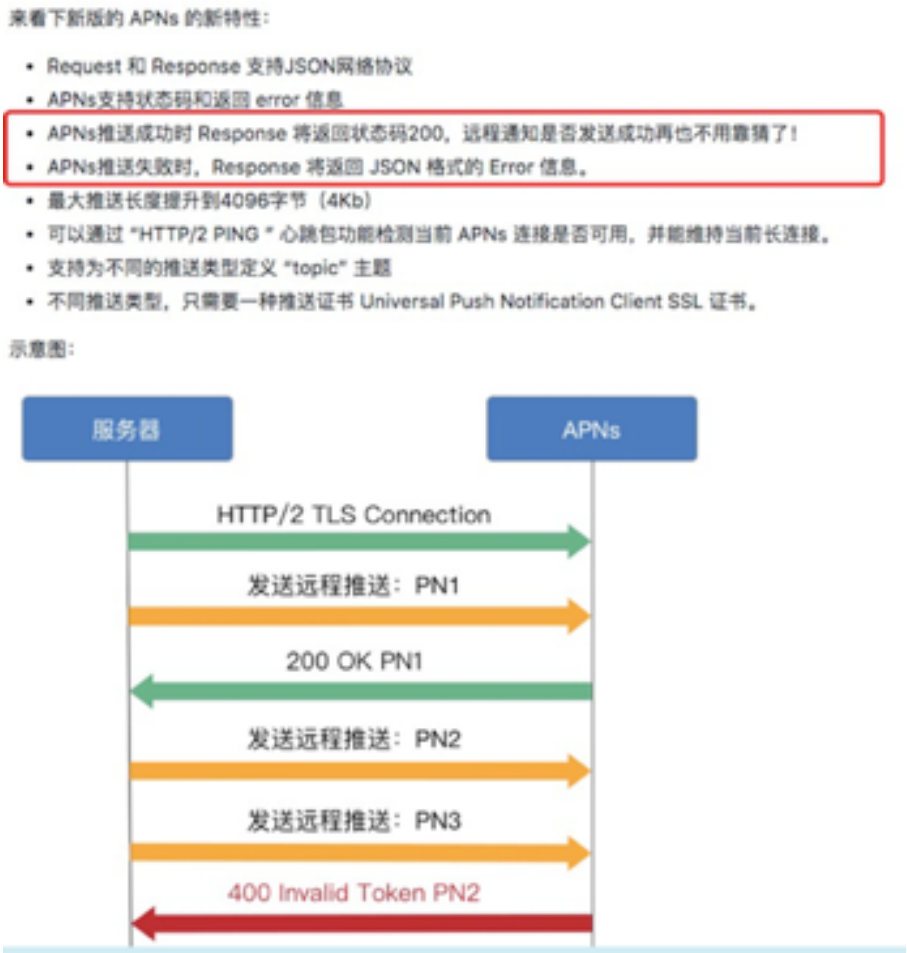
```
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, queue, ^{ /*加载图片1 */ });
dispatch_group_async(group, queue, ^{ /*加载图片2 */ });
dispatch_group_async(group, queue, ^{ /*加载图片3 */ });
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
        // 合并图片
});
```

http：
46、http与https的区别?

A:安全性上的区别:HTTPS是HTTP协议的安全加强版，通过在HTTP上建立加密层，对传输数据进行加密。主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性
B:表现形式：HTTPS站点会在地址栏上显示一把绿色小锁，表明这是加密过的安全网站，如果采用了全球认证的顶级EV SSL证书的话，其地址栏会以绿色高亮显示，方便用户辨认。
C:SEO：在2015年之前百度是无法收录HTTPS页面的，不过自从2015年5月份百度搜索全站HTTPS加密后，就已经可以收录HTTPS了。谷歌则是从2014年起便开始收录HTTPS页面，并且HTTPS页面权重比HTTP页面更高。从SEO的角度来说，HTTPS和HTTP区别不大，甚至HTTPS效果更好。
D:技术层面：如果要说HTTPS和HTTP的区别，最关键的还是在技术层面。比如HTTP标准端口是80，而HTTPS标准端口是443；HTTP无需证书，HTTPS需要CA机构颁发的SSL证书；HTTP工作于应用层，HTTPS工作于传输层。

47、服务器能否知道APNS推送后有没有到达客户端的方法?
APNS是苹果提供的远程推送的服务，APP开发此功能之后，用户允许推送之后，服务端可以向安装了此app的用户推送信息。但是APNS推送无法保证100%到达。

目前关于APNS苹果更新了新的策略，即 APNS/HTTP2.

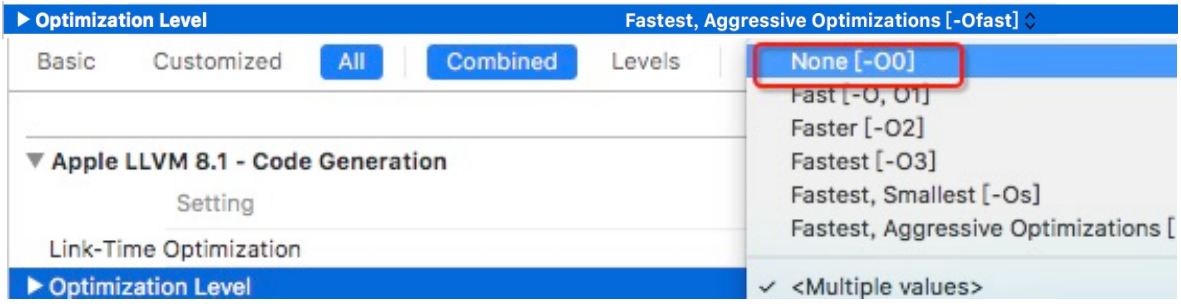


如果服务器像APNS服务器推送信息之后，服务器能够接收到APNS是否真的成功像客户端推送成功了某个信息。这样在一定程度上提高了APNS的成功概率。

app：

48、1.什么方式可以看到上架App的头文件?

2.阅读过哪些框架的源码?能说说它的架构方式吗



49、iOS iAP内购审核可能失败的问题

Guideline 3.1.1 - Business - Payments - In-App Purchase
We noticed that your app uses in-app purchase products to purchase credits or currencies that are not consumed within the app, which is not appropriate for the App Store.

苹果提交iAP内购审核的时候，可能出现上面的问题。出现这个问题有可能的原因是因为你的app中在iAP内购中购买的商品，能够通过其他的渠道或者方式购买。此处在AppStore是不允许的。比如，你在安卓充值100元人民币，那么如果商品一样能够使用在iOS设备上，苹果不会允许你上线的。当然这里指的是虚拟类商品。

另外就是在审核的时候不能以任何方式，通过活动或者兑换码的形式，能够获取到iAP内购中能够获取到的商品。

App上架后，如何修改app上显示的公司名称?
1.先修改开发者账号中填写的公司名称。
2.再提交更新版本。
如何修改开发者账号中的公司名称：
登陆到Apple developer上面，在people里面的开发者列表中找到agent，让agent的这个人直接拨打苹果开发部咨询电话，修改开发者账号上的公司名或者用你注册的账号的邮箱直接写邮件：“我需要更改公司名称”到chinadev@asia.apple.com，让苹果开发部客服来处理。

50、IAP内购中虚拟货币导致审核无法通过的问题?

有的时候需要在app中使用虚拟货币，在我们的app中可以使用虚拟货币进行购买道具等，比如直播中的礼物，游戏中的道具等。

苹果对于虚拟货币是需要提成的，提成的额度为30%。所以对于这块的审核比较严格。首先你们的购买的道具在ios端和安卓端是需要分开的。如果大家玩游戏的就会发现游戏的数据在两端是分开的。
用户在安卓手机上购买的道具在ios上不能使用。因为这样也间接的影响了苹果的收入。

另外就是在审核期间不能有可以兑换在appStore可购买的商品，的任意活动或者兑换码，这个也是苹果不允许的。因为这个也会影响苹果的收入。

另外就是可能有人会在苹果审核之后隐藏ipa支付，此处提醒下，苹果会扫描你的app代码中是否有支付宝，微信等关于支付的字段。使用开关加h5的方式可以通过审核，但是此处也有风险，风险就是一旦被发现，可能的结果就是苹果直接封掉账号。app无法使用。