atomic

设置成员变量的@property属性时，默认为atomic，提供多线程安全。

在多线程环境下，原子操作是必要的，否则有可能引起错误的结果。加了atomic，setter函数会变成下面这样：

```
{lock}
    if (property != newValue) {
        [property release];
        property = [newValue retain];
    }
{unlock}
```

nonatomic

禁止多线程，变量保护，提高性能。

ios提供了copy和mutablecopy方法，顾名思义，copy就是复制了一个imutable的对象，而mutablecopy就是复制了一个mutable的对象。以下将举几个例子来说明。

1、系统的非容器类对象：这里指的是NSString、NSNumber等等一类的对象。

```
NSString *string = @"origion";
NSString *stringCopy = [string copy];
NSMutableString *stringMCopy = [string mutableCopy];
[stringMCopy appendString:@"!!"];
```

查看内存可以发现，string和stringCopy指向的是同一块内存区域(又叫apple弱引用weak reference)，此时stringCopy的引用计数和string的一样都为2。而stringMCopy则是我们所说的真正意义上的复制，系统为其分配了新内存，但指针所指向的字符串还是和string所指的一样。

再看下面的例子：

```
NSMutableString *string = [NSMutableString stringWithString:
@"origion"];
NSString *stringCopy = [string copy];
 NSMutableString *mStringCopy = [string copy];
NSMutableString *stringMCopy = [string mutableCopy];
[mStringCopy appendString:@"mm"];//error
[string appendString:@" origion!"];
[stringMCopy appendString:@"!!"];
```

以上四个NSString对象所分配的内存都是不一样的。但是对于mStringCopy其实是个imutable对象，所以上述会报错。
对于系统的非容器类对象，我们可以认为，如果对一不可变对象复制，copy是指针复制（浅拷贝）和mutableCopy就是对象复制（深拷贝）。如果是对可变对象复制，都是深拷贝，但是copy返回的对象是不可变的。

2、系统的容器类对象：指NSArray，NSDictionary等。对于容器类本身，上面讨论的结论也是适用的，需要探讨的是复制后容器内对象的变化。

```
//copy返回不可变对象，mutablecopy返回可变对象
NSArray *array1 = [NSArray
arrayWithObjects:@"a",@"b",@"c",nil];
NSArray *arrayCopy1 = [array1 copy];
//arrayCopy1是和array同一个NSArray对象（指向相同的对象），包括array里面的元素也是指向相同的指针
NSLog(@"array1 retain count: %d",[array1 retainCount]);
NSLog(@"array1 retain count: %d",[arrayCopy1 retainCount]);
NSMutableArray *mArrayCopy1 = [array1 mutableCopy];
 //mArrayCopy1是array1的可变副本，指向的对象和array1不同，但是其中的元素和array1中的元素指向的是同一个对象。mArrayCopy1还可以修改自己的对象
[mArrayCopy1 addObject:@"de"];
[mArrayCopy1 removeObjectAtIndex:0];
```

array1和arrayCopy1是指针复制，而mArrayCopy1是对象复制，mArrayCopy1还可以改变期内的元素：删除或添加。但是注意的是，容器内的元素内容都是指针复制。
下面用另一个例子来测试一下。

```
NSArray *mArray1 = [NSArray arrayWithObjects:[NSMutableString stringWithString:@"a"],@"b",@"c",nil];
NSArray *mArrayCopy2 = [mArray1 copy];
NSLog(@"mArray1 retain count: %d",[mArray1 retainCount]);
NSMutableArray *mArrayMCopy1 = [mArray1 mutableCopy];
NSLog(@"mArray1 retain count: %d",[mArray1 retainCount]);
 //mArrayCopy2,mArrayMCopy1和mArray1指向的都是不一样的对象，但是其中的元素都是一样的对象—同一个指针
 //一下做测试
 NSMutableString *testString = [mArray1 objectAtIndex:0];
 //testString = @"1a1";//这样会改变testString的指针，其实是将@“1a1”临时对象赋给了testString
[testString appendString:@" tail"];//这样以上三个数组的首元素都被改变了
```

由此可见，对于容器而言，其元素对象始终是指针复制。如果需要元素对象也是对象复制，就需要实现深拷贝。

当一个视图控制器被创建，并在屏幕上显示的时候。 代码的执行顺序
1、 alloc 创建对象，分配空间
2、init (initWithNibName) 初始化对象，初始化数据
3、loadView 从nib载入视图 ，通常这一步不需要去干涉。除非你没有使用xib文件创建视图
4、viewDidLoad 载入完成，可以进行自定义数据以及动态创建其他控件
5、viewWillAppear 视图将出现在屏幕之前，马上这个视图就会被展现在屏幕上了
6、viewDidAppear 视图已在屏幕上渲染完成

当一个视图被移除屏幕并且销毁的时候的执行顺序，这个顺序差不多和上面的相反
1、viewWillDisappear 视图将被从屏幕上移除之前执行
2、viewDidDisappear 视图已经被从屏幕上移除，用户看不到这个视图了
3、dealloc 视图被销毁，此处需要对你在init和viewDidLoad中创建的对象进行释放

关于viewDidUnload： 在发生内存警告的时候如果本视图不是当前屏幕上正在显示的视图的话， viewDidUnload将会被执行，本视图的所有子视图将被销毁，以释放内存，此时开发者需要手动对viewLoad、viewDidLoad中创建的对象释放内存。 因为当这个视图再次显示在屏幕上的时候，viewLoad、viewDidLoad 再次被调用，以便再次构造

视图。

当我们创建一个UIViewController类的对象时，通常系统会生成几个默认的方法，这些方法大多与视图的调用有关，但是在视图调用时，这些方法的调用顺序如何，需要整理下。
通常上述方法包括如下几种，这些方法都是UIViewController类的方法：

- (void)viewDidLoad;

- (void)viewDidUnload;

- (void)viewWillAppear:(BOOL)animated;

- (void)viewDidAppear:(BOOL)animated;

- (void)viewWillDisappear:(BOOL)animated;

- (void)viewDidDisappear:(BOOL)animated;

下面介绍下APP在运行时的调用顺序。

1）- (void)viewDidLoad;

一个APP在载入时会先通过调用loadView方法或者载入IB中创建的初始界面的方法，将视图载入到内存中。然后会调用viewDidLoad方法来进行进一步的设置。通常，我们对于各种初始数据的载入，初始设定等很多内容，都会在这个方法中实现，所以这个方法是一个很常用，很重要的方法。

但是要注意，这个方法只会在APP刚开始加载的时候调用一次，以后都不会再调用它了，所以只能用来做初始设置。

2) - (void)viewDidUnload;

在内存足够的情况下，软件的视图通常会一直保存在内存中，但是如果内存不够，一些没有正在显示的viewcontroller就会收到内存不够的警告，然后就会释放自己拥有的视图，以达到释放内存的目的。但是系统只会释放内存，并不会释放对象的所有权，所以通常我们需要在这里将不需要在内存中保留的对象释放所有权，也就是将其指针置为nil。

这个方法通常并不会在视图变换的时候被调用，而只会在系统退出或者收到内存警告的时候才会被调用。但是由于我们需要保证在收到内存警告的时候能够对其作出反应，所以这个方法通常我们都需要去实现。

另外，即使在设备上按了Home键之后，系统也不一定会调用这个方法，因为IOS4之后，系统允许将APP在后台挂起，并将其继续滞留在内存中，因此，viewcontroller

并不会调用这个方法来清除内存。

3）- (void)viewWillAppear:(BOOL)animated;

系统在载入所有数据后，将会在屏幕上显示视图，这时会先调用这个方法。通常我们会利用这个方法，对即将显示的视图做进一步的设置。例如，我们可以利用这个方法来设置设备不同方向时该如何显示。

另外一方面，当APP有多个视图时，在视图间切换时，并不会再次载入viewDidLoad方法，所以如果在调入视图时，需要对数据做更新，就只能在这个方法内实现了。所以这个方法也非常常用。

4) - (void)viewDidAppear:(BOOL)animated;

有时候，由于一些特殊的原因，我们不能在viewWillApper方法里，对视图进行更新。那么可以重写这个方法，在这里对正在显示的视图进行进一步的设置。

5) - (void)viewWillDisappear:(BOOL)animated;

在视图变换时，当前视图在即将被移除、或者被覆盖时，会调用这个方法进行一些善后的处理和设置。

由于在IOS4之后，系统允许将APP在后台挂起，所以在按了Home键之后，系统并不会调用这个方法，因为就这个APP本身而言，APP显示的view，仍是挂起时候的view，所以并不会调用这个方法。

6) - (void)viewDidDisappear:(BOOL)animated;

我们可以重写这个方法，对已经消失，或者被覆盖，或者已经隐藏了的视图做一些其他操作。

上述方法的流程图可以简单用如下表示：

运行APP —> 载入视图 —> 调用viewDidLoad方法 —> 调用viewWillAppear方法 —> 调用viewDidAppear方法 —> 正常运行

A

|

|

|

| 载入新的
View |

|

|

V

释放对象所有权 <— 调用viewDidUnload <— 收到内存警告 <— 调用
viewDidDisappear <— 调用viewWillDisappear <— APP需要调用另一个view

IOS程序启动执行顺序

http://www.yifeiyang.net/iphone-developer-advanced-3-iphone-application-startup-process/

IOS 开发 loadView 和 viewDidLoad 的区别

iPhone开发必不可少的要用到这两个方法。 他们都可以用来在视图载入的时候，初始化一些内容。 但是他们有什么区别呢?

viewDidLoad 此方法只有当view从nib文件初始化的时候才被调用。

loadView 此方法在控制器的view为nil的时候被调用。 此方法用于以编程的方式创建view的时候用到。 如:

```
1.
2.  - ( void ) loadView {
3.      UIView *view = [ [ UIView alloc] initWithFrame:[ UIScreen
4.  mainScreen] .applicationFrame] ;
5.      [ view setBackgroundColor:_color] ;
```

```
6.      self.view = view;

7.      [ view release] ;

8.  }

9.
```

你在控制器中实现了loadView方法，那么你可能会在应用运行的某个时候被内存管理控制调用。 如果设备内存不足的时候， view 控制器会收到didReceiveMemoryWarning的消息。 默认的实现是检查当前控制器的view是否在使用。如果它的view不在当前正在使用的view hierarchy里面，且你的控制器实现了loadView方法，那么这个view将被release, loadView方法将被再次调用来创建一个新的view。

Don't read self.view in -loadView. Only *set* it, don't *get* it.

The self.view property accessor *calls* -loadView if the view isn't currently loaded. There's your infinite recursion.

The usual way to build the view programmatically in -loadView, as demonstrated in Apple's pre-Interface-Builder examples, is more like this:

```
UIView   * view  =   [[ UIView alloc ] init ...];
```

```
...
[ view addSubview : whatever ];
[ view addSubview : whatever2 ];
...
self . view  = view ;
[ view release ];
```

And I don't blame you for not using IB. I've stuck with this method for all of Instapaper and find myself much more comfortable with it than dealing with IB's complexities, interface quirks, and unexpected behind-the-scenes behavior.

	iphone5/s	iPhone 6	iPhone 6 Plus
尺寸	4”	4.7”	5.5”
Viewport’s device-width (in CSS pixels)	320	375	414
Viewport’s device-width (Android设备同分辨率参考)	360	360	400
Device Pixel Ratio 像素比	2	2	3 (近似值)
Rendered Pixels 渲染像素 (默认 viewport size * dpr)	640x1136	750×1334	1242×2208
Physical pixels 物理像素（手机显示像素）	640x1136	750×1334	1080×1920
PPI 每英寸所拥有的像素数	326	326	401
Status Bar 状态栏高 (px)	40	40	60
Title Bar 导航条高 (px)	88	88	132
Tab Bar 底栏高 (px)	98	98	147
桌面 icon (px)	120	120	180
图片资源后缀名	@2x	@2x	@3x

一.UIApplicationMain的执行步骤

1.创建一个UIApplication对象,一个程序对应一个UIApplication对象(单例),UIApplication对象是程序的象征

2.接下来会根据第4个参数创建一个UIApplication的delegate对象
3.开启一个消息循环(不断地监听地一些系统事件)
4.监听到相应的事件后,就会给代理发送相应的消息
* 当程序启动完毕,就会发送
application:didFinishLaunchingWithOptions:消息
*当程序进入后台,就会发送applicationDidEnterBackground:消息
....

二.UIApplication
1.设置图标数字
app.applicationIconBadgeNumber =10;

2.设置显示联网状态
app.networkActivityIndicatorVisible = YES;

3.打开一个URL (打电话,打开网址,发短信,发邮件)
NSURL *url = [NSURL URLWithString:@"http://
www.baidu.com"];
[app openURL:url];

4.隐藏状态栏

application.statusBarHidden = NO;

三.常见文件
1.*-Info.plist 文件
1>项目中最主要的文件,描述了软件名称.软件版本,软件唯一标识
2>常见设置
*软件名称 --Bundle display name -- CFBundleDisplayName
*软件版本号 -- Bundle Version === CFBunleVersion
*软件的唯一标识-- Bundle identifier -- CFBundleIdentifier
1.0

2.*-Prefix.pch

1> 该文件中的内容会被项目中其他所有文件所共享

2>用来存全局性的一些宏定义,#import语句

3>控件日志输出

```
/*
如果软件处于调试状态,系统会默认定义一个叫做DEBUG的宏;
如果软件处于发布打包状态,系统就不会定义DEBUG这个宏
*/
#ifdef DEBUG
//调试状态:将MJLog替换成NSLog
#define MJLog(...) NSLog(__VA_ARGS__)
#else
//发布状态:将MJLog替换成空
#define MJLog(...) //fsdfsdfdsfds
#endif
```

3. Default.png 320 X 480

1>程序在启动过程中会全屏显示叫做Default.png 的图片

2>可以有多个版本

Default.png 320 X 480

Default@2x.png 640 X 960

Default-568h@2x.png 640 X 1136

4.Icon.png

1>软件图标

2>苹果官方文档搜索"app icon" --> App Icons On iPad and Iphone

四\程序的完整启动过程(控件器的View的显示过程)

1.点击程序图标

2.执行main函数

3.执行UIApplicationMain函数

4.创建UIApplication对象.UIApplication的delegate对象

5.开启事件循环监听系统事件

6.程序加载完毕后调用delegate对象的application:didFinishLaunchingWithOptions:方法

1>创建窗口对象

self.window = [UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

2>创建控件器对象

self.viewController = [[MJViewController alloc] initWithNibName:@"MJViewController" bundle:nil]

3> 设置窗口的根控件器

self.window.rootViewController = self.viewController;

4>让窗口成为主窗口,并且可见

[self.window makeKeyAndVisible];

UIApplication\AppDelegate\UIWindow\UIViewController

6.自定义容器

1.加载ViewController.xib文件

2.创建Objects下面的所有对象

UIView *myview = [[UIView alloc]init];

myView.backgroundColor=[UIColor dfsfd];

myview.frame = CGRectMake(0,20,320,460);

....

UIButton *btn ...

[myview addSubview:btn];

3.传入ViewController对象作为xib文件的owner

vc.view = myview;

2.CALayer是一个bitmap图象的容器类，当UIView调用自身的drawRect时，CALayer才会创 建这个bitmap图象类。

3.具体占内存的其实是一个bitmap图象类，CALayer只占48bytes, UIView只占 96bytes。而一个iPad的全屏UIView的bitmap类会占到12M的大小!

4.在iOS6时，当系统发出 MemoryWarning时，系统会自动回收bitmap类。但是不回收UIView和CALayer类。这样即回收了大部分内存，又能在需要bitmap类时，通过调用UIView的drawRect: 方法重建。

所以苹果才会说现在会卸载那 些真正占内存的独立内容，卸载UIView对象已经并不重要了。

另外，在每次视图显示和隐藏的过 程中还会调用一些方法

- (void)viewWillAppear:(BOOL)animated

- (void)viewDidAppear: (BOOL)animated

- (void)viewWillDisappear:(BOOL)animated

- (void) viewDidDisappear:(BOOL)animated

写入在每次显示视图和隐藏视图时的代码块。

重要的是： 视图控制对象和视图是两个独立的对象，Controller要借助View才能和用户进行交互，可以把视图看成 一种可再生资源，可以创建，卸载（从前），可以显示，隐藏。

UIView的layoutSubviews和drawRect

博客分类： iosios-ui

UIView的setNeedsDisplay和setNeedsLayout方法。首先两个方法都是异步执行的。setNeedsDisplay会调用自动调用drawRect方法，这样可以拿到UIGraphicsGetCurrentContext，就可以画画了。而setNeedsLayout会默认调用layoutSubViews，就可以处理子视图中的一些数据。

综上两个方法都是异步执行的，layoutSubviews方便数据计算，drawRect方便视图重绘。

先大概看下ios layout机制相关的这几个方法：

- (CGSize)sizeThatFits:(CGSize)size

- (void)sizeToFit

———————-

- (void)layoutSubviews

- (void)layoutIfNeeded

- (void)setNeedsLayout

———————–

- (void)setNeedsDisplay

- (void)drawRect

一、

layoutSubviews在以下情况下会被调用：

1、init初始化不会触发layoutSubviews。

2、addSubview会触发layoutSubviews。

3、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。

4、滚动一个UIScrollView会触发layoutSubviews。

5、旋转Screen会触发父UIView上的layoutSubviews事件。

6、改变一个UIView大小的时候也会触发父UIView上的layoutSubviews事件。

7、直接调用setLayoutSubviews。

8、直接调用setNeedsLayout。

在苹果的官方文档中强调:You should override this method only if the autoresizing behaviors of the subviews do not offer the behavior you

want.

layoutSubviews, 当我们在某个类的内部调整子视图位置时，需要调用。反过来的意思就是说：如果你想要在外部设置subviews的位置，就不要重写。

刷新子对象布局

-layoutSubviews方法：这个方法，默认没有做任何事情，需要子类进行重写

-setNeedsLayout方法： 标记为需要重新布局，异步调用layoutIfNeeded刷新布局，不立即刷新，但layoutSubviews一定会被调用

-layoutIfNeeded方法：如果，有需要刷新的标记，立即调用layoutSubviews进行布局（如果没有标记，不会调用layoutSubviews）

如果要立即刷新，要先调用[view setNeedsLayout]，把标记设为需要布局，然后马上调用[view layoutIfNeeded]，实现布局

在视图第一次显示之前，标记总是“需要刷新”的，可以直接调用[view layoutIfNeeded]

二、

drawRect在以下情况下会被调用：

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。drawRect 掉用是在Controller->loadView, Controller->viewDidLoad 两方法之后掉用的.所以不用担心在 控制器中,这些View的drawRect就开始画了.这样可以在控制器中设置一些值给View(如果这些View draw的时候需要用到某些变量 值).

2、该方法在调用sizeToFit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。

sizeToFit会自动调用sizeThatFits方法；

sizeToFit不应该在子类中被重写，应该重写sizeThatFits

sizeThatFits传入的参数是receiver当前的size，返回一个适合的size
sizeToFit可以被手动直接调用
sizeToFit和sizeThatFits方法都没有递归，对subviews也不负责，只负责自己
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。
-setNeedsDisplay方法：标记为需要重绘，异步调用drawRect
-setNeedsDisplayInRect:(CGRect)invalidRect方法：标记为需要局部重绘
以上1,2推荐；而3,4不提倡

drawRect方法使用注意点：
1、 若使用UIView绘图，只能在drawRect：方法中获取相应的contextRef并绘图。如果在其他方法中获取将获取到一个invalidate 的ref并且不能用于画图。drawRect：方法不能手动显示调用，必须通过调用setNeedsDisplay 或 者 setNeedsDisplayInRect，让系统自动调该方法。
2、若使用calayer绘图，只能在drawInContext: 中（类似鱼drawRect）绘制，或者在delegate中的相应方法绘制。同样也是调用setNeedDisplay等间接调用以上方法
3、若要实时画图，不能使用gestureRecognizer，只能使用touchbegan等方法来掉用setNeedsDisplay实时刷新屏幕

三、
layoutSubviews对subviews重新布局
layoutSubviews方法调用先于drawRect
setNeedsLayout在receiver标上一个需要被重新布局的标记，在系统runloop的下一个周期自动调用layoutSubviews

layoutIfNeeded方法如其名，UIKit会判断该receiver是否需要layout.根据Apple官方文档,layoutIfNeeded方法应该是这样的

layoutIfNeeded遍历的不是superview链，应该是subviews链

drawRect是对receiver的重绘，能获得context

setNeedDisplay在receiver标上一个需要被重新绘图的标记，在下一个draw周期自动重绘，iphone device的刷新频率是60hz，也就是1/60秒后重绘

```
@interface MyClass()//注意(),即定义一个空类别
{
    私有变量;
}
- (void)PrivateMethod;//在类别中定义私有方法
@end
```

object-c 为了让java的开发者习惯 使用.的操作，所以可以将接口类中的变量 使用@property来声明属性。但是在.h中声明的属性，必须在.m中使用@synthesize或者@dynamic来实现（传言，在最近出的ios6中这不已经省了），否则属性不可用。

熟悉object－c语法的都知道@synthesize实际的意义就是 自动生成属性的setter和getter方法。

@dynamic 就是要告诉编译器，代码中用@dynamic修饰的属性，其getter和setter方法会在程序运行的时候或者用其他方式动态绑定，以便让编译器通过编译。其主要的作用就是用在NSManagerObject对象的属性声明上，由于此类对象的属性一般是从Core Data的属性中生成的，core data 框架会在程序运行的时候为此类属性生成getter和setter方法。

文一

我从苹果文档中得知，一般的应用在进入后台的时候可以获取一定时间来运行相关任务，也就是说可以在后台运行一小段时间。

还有三种类型的可以运行在后以，

1.音乐

2.location

3.voip

文二

在IOS后台执行是本文要介绍的内容，大多数应用程序进入后台状态不久后转入暂停状态。在这种状态下，应用程序不执行任何代码，并有可能在任意时候从

内存中删除。应用程序提供特定的服务，用户可以请求后台执行时间，以提供这些服务。

判断是否支持多线程

```
UIDevice* device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
backgroundSupported = device.multitaskingSupported;
```

声明你需要的后台任务

Info.plist中添加UIBackgroundModes键值，它包含一个或多个string的值，包括

audio:在后台提供声音播放功能，包括音频流和播放视频时的声音

location：在后台可以保持用户的位置信息

voip：在后台使用VOIP功能

前面的每个value让系统知道你的应用程序应该在适当的时候被唤醒。例如，一个应用程序，开始播放音乐，然后移动到后台仍然需要执行时间，以填补音频输出缓冲区。添加audio键用来告诉系统框架，需要继续播放音频，并且可以在合适的时间间隔下回调应用程序；如果应用程序不包括此项，任何音频播放在移到后台后将停止运行。

除了添加键值的方法，IOS还提供了两种途径使应用程序在后台工作：

Task completion—应用程序可以向系统申请额外的时间去完成给定的任务

Local notifications—应用程序可以预先安排时间执行local notifications 传递

文三
如何让程序后台播放音乐

http://developer.apple.com/library/ios/#qa/qa1668/_index.html

文四

如果你的应用程序需要后台运行，可以使用以下方法：

1。应用程序可以请求一个有限的时间内完成一些重要任务。

2。应用程序可以声明为支持特定服务需要定期后台执行时间。

3。应用程序可以使用本地生成用户在指定的时间的警报，应用程序正在运行与否的通知。

原文地址：http://blog.csdn.net/diyagoanyhacker/article/details/7071055

作者：禚来强

声明：此文一部分来自网络，一部分来自官方文档（翻译），还有一部分是作者的总结

文五
后台运行被第一次提到

http://developer.apple.com/library/ios/#releasenotes/General/WhatsNewIniPhoneOS/Articles/iPhoneOS4.html#//apple_ref/doc/uid/TP40009559-SW1

文六

后台运行官方文档

http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html#//apple_ref/doc/uid/TP40007072-CH4-SW3

在IOS后台执行是本文要介绍的内容，大多数应用程序进入后台状态不久后转入暂停状态。在这种状态下，应用程序不执行任何代码，并有可能在任意时候从内存中删除。应用程序提供特定的服务，用户可以请求后台执行时间，以提供这些服务。

判断是否支持多线程

```
UIDevice* device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
backgroundSupported = device.multitaskingSupported;
```

声明你需要的后台任务

Info.plist中添加UIBackgroundModes键值，它包含一个或多个string的值，包括

audio:在后台提供声音播放功能，包括音频流和播放视频时的声音

location：在后台可以保持用户的位置信息

voip：在后台使用VOIP功能

前面的每个value让系统知道你的应用程序应该在适当的时候被唤醒。例如，一个应用程序，开始播放音乐，然后移动到后台仍然需要执行时间，以填补音频输出缓冲区。添加audio键用来告诉系统框架，需要继续播放音频，并且可以在合适的时间间隔下回调应用程序；如果应用程序不包括此项，任何音频播放在移到后台后将停止运行。

除了添加键值的方法，IOS还提供了两种途径使应用程序在后台工作：

Task completion—应用程序可以向系统申请额外的时间去完成给定的任务

Local notifications—应用程序可以预先安排时间执行local notifications 传递

实现长时间的后台任务

应用程序可以请求在后台运行以实现特殊的服务。这些应用程序并不连续的运行，但是会被系统框架在合适的时间唤醒，以实现这些服务

1、 追踪用户位置：略

2、在后台播放音频：

添加UIBackgroundModes中audio值，注册后台音频应用。这个值使得应用程序可以在后台使用可听的背景，如音乐播放或者音频流应用。对于支持音频和视频功能的应用程序也可以添加该值以保证可以继续持续的运行流。

当audio值设置后，当你的应用程序进入后台后，系统的多媒体框架会自动阻止它被挂断，但是，如果应用程序停止播放音频或者视频，系统将挂断应用程序。

当你的应用程序在后台时，你可以执行任意的系统音频框架去初始化后台音频。你的应用程序在后台时应该限制自身，使其执行与工作相关的代码，不能

执行任何与播放内容无关的任务

由于有多个应用程序支持音频，前台的应用程序始终允许播放音频，后台的应用程序也被允许播放一些音频内容，这取决于audio session object的设置。应用程序应该始终设置它们的audio session object，并小心的处理其他类型的音频相关notifications和中断。详见audio session programming guide。

3、实现VOIP应用：

VOIP程序需要稳定的网络去连接和它相关的服务，这样它才能接到来电和其他相关的数据。系统允许VOIP程序被挂断并提供组件去监听它们的sockets，而不是在任意时候都处于唤醒状态。设置VOIP应用程序如下：

A、 添加UIBackgroundModes中的VOIP键值

B、 为VOIP设置一个应用程序socket

C、 在移出后台之前，调用setKeepAliveTimeout:handler:方法去建立一个定期执行的handler，你的应用程序可以运行这个handler来保持服务的连接。

D、 设置你的audio session去处理这种切换

释义：

A、大多数VOIP应用需要设置后台audio 应用去传递音频，因此你应该设置audio 和voip两个键值。

B、为了使应用程序在后台时保持稳定的连接，你必须tag你的主通讯socket专门应用于VOIP，tagging这个socket来告诉系统，它必须在你的应用程序中断时接管这个socket。这个切换本身对于你的应用程序时透明的，当新的数据到达socket的时候，系统会唤醒应用程序，并将socket的控制权返回给应用程序，这样应用程序就可以处理新来的数据。

你只需要tag用于voip服务的socket，这个socket用来接收来电或者其他相关的数据来保持你的VOIP服务的连接。根据收到的信息，这个socket要决定下一步的动作。比如一个来电，你会想弹出一个本地的通知来告知用户；对于其他不是那么关键的数据，你可能会想悄悄的处理这些数据并让系统将应用程序重新中断。

在IOS中，sockets是用流或者更高级的结构，设置一个VOIP的socket，你只需要在通常的设置中添加一个特殊的key来标明这个接口是用于连接VOIP服务的，下表列出了流的接口和设置：

设置流接口用于voip

接口

设置

NSInputStream 和NSOutputStream
对于 Cocoa streams, 使用 setProperty:forKey: 方法添加

NSStreamNetworkServiceType
属性给
stream.
改属性的值设为
NSStreamNetworkServiceTypeVoIP.
NSURLRequest
对于 URL loading system, 使用 setNetworkServiceType:

method of your NSMutableURLRequest object to set the network service type of the request. The service type should be set to NSURLNetworkServiceTypeVoIP.
CFReadStreamRef和CFWriteStreamRef

For Core Foundation streams, use the CFReadStreamSetProperty or CFWriteStreamSetProperty function to add the kCFStreamNetwork-ServiceType property to the stream. The value for this property should be set to kCFStreamNetworkServiceTypeVoIP.
（注意：当设置socket的时候,你需要在你的主信号通道中设置合适的service type key。当设置声道时，不需要设置这个key）

由于，VOIP应用程序需要一直运行以确保收到来电，所以如果程序通过一个非零的exit code退出，系统将自动重启这个应用程序（这种退出方式可以发生在内存压力大时终止程序运行）。尽管如此，中断应用程序会release所有的sockets，包括那个用于连接voip 服务的socket。因此，当程序运行时，它需要一直从头创建socket。

C、 为了防止断连，voip程序需要定期被唤醒去检查它的服务。为了容易实现这个行为，IOS通过使用（UIApplication setKeepAliveTimeout:handler:）方法建立一个特殊的句柄。你可以在applicationDidEnterBackground方法中建立该句柄。一旦建立，系统至少会在超时之前调用该句柄一次，来唤醒你的应用程序。

这个keep-alive handler在后台执行，必须尽快的返回参数，它有最多30秒的时间来执行所需的任务，如果这段时间内句柄没有返回，那么系统将终止应用程序。

当你建立了handler之后，确定应用程序所需的最大超时。系统保证会在最大超时之前调用handler，但是这个时间是不确定的，所以你的handler必须在你申明的超时之前做好执行程序的准备。

D、设置audio session，详见Audio Session Programming Guide.

在后台完成有限长度的任务

在被终止之前的任意时间,应用程序会调用beginBackgroundTaskWithExpirationHandler:方法让系统给出额外的时间来完成一些需要在后台长时间执行的任务。（UIApplication的backgroundTimeRemaining属性包含程序运行的总时间）

可以使用task completion去保证那些比较重要但是需要长时间运行的程序不会由于用户切入后台而突然关闭。比如，你可以用这项功能来将用户的信息保存到disk上或者从网络下载一个重要的文件。有两种方式来初始化这样的任务：

1、将长时间运行的重要任务用beginBackgroundTaskWithExpirationHandler：和endBackgroundTask：包装。这样就在程序突然切入后台的时候保护了这些任务不被中断。

2、当你的应用程序委托applicationDidEnterBackground：方法被调用时再启动任务

中的两个方法必须是一一对应的，endBackgroundTask：方法告诉系统任务已经完成，程序在此时可以被终止。由于应用程序只有有限的时间去完成后台任务，你必须在超时或系统将要终止这个程序之前调用这个方法。为了避免被终止，你也可以在一个任务开始的时候提供一个expiration handler和endBackgroundTask：方法。（可以查看backgroundTimeRemaining属性来确定还剩多少时间）。

一个程序可以同时提供多个任务，每当你启动一个任务的时候，beginBackgroundTaskWithExpirationHandler：方法将返回一个独一无二的handler去识别这个任务。你必须在endBackgroundTask：方法中传递相同的handler来终止该任务。

Listing 4-2 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
UIApplication* app = [UIApplication sharedApplication];
```

```
bgTask = [app beginBackgroundTaskWithExpirationHandler:^{
[app endBackgroundTask:bgTask];
bgTask = UIBackgroundTaskInvalid;
}];
// Start the long-running task and return immediately.
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DE
FAULT,
0), ^{
// Do the work associated with the task.
[app endBackgroundTask:bgTask];
bgTask = UIBackgroundTaskInvalid;
});
}
```

上述例子中，bgTask变量是一个类的成员变量，存储着指向该后台任务标示的指针。

在expriation handler中，可以添加关闭任务所需的代码。尽管如此，加入的代码不能执行太长的时间，当expriation handler被调用的时候，该程序已经非常接近被关闭，所以只有极短的时间来清除状态信息并终止任务。

安排Local Notification的传递

UILocalNotification类提供了一种方法来传递local notifications。和push notifications需要设置remote server不同，local notifications 在程序中安排并在当前的设备上执行。满足如下条件可以使用该能力：

1、一个基于时间的程序，可以在将来特定的时间让程序post 一个alert，比如闹钟

2、一个在后台运行的程序，post 一个local notification去引起用户的注意

为了安排local notification 的传递，需要创建一个UILocalNotification的实例，

并设置它，使用UIApplication类方法来安排它。Local notification对象包含了所要传递的类型（sound，alert，或者badge）和时间何时呈现） 。UIApplication类方法提供选项去确定是立即传递还是在指定的时间传递。

Listing 4-3 Scheduling an alarm notification

```
- (void)scheduleAlarmForDate:(NSDate*)theDate
{
UIApplication* app = [UIApplication sharedApplication];
NSArray* oldNotifications = [app scheduledLocalNotifications];
// Clear out the old notification before scheduling a new one.
if ([oldNotifications count] > 0)
[app cancelAllLocalNotifications];
// Create a new notification.
UILocalNotification* alarm = [[[UILocalNotification alloc] init] autorelease];
if (alarm)
{
alarm.fireDate = theDate;
alarm.timeZone = [NSTimeZone defaultTimeZone];
alarm.repeatInterval = 0;
alarm.soundName = @"alarmsound.caf";
alarm.alertBody = @"Time to wake up!";
[app scheduleLocalNotification:alarm];
}
}
```

(可以最多包含128个 local notifications active at any given time, any of which can be configured to repeat at a specified interval.)如果在调用该notification的时候，程序已经处于前台，那么application：didReceiveLocalNotification：方法将取而代之。

小结：关于详解在IOS后台执行的内容介绍完了，希望本文对你有所帮助!

文七

iOS不是真正的多任务系统，在用户按下Home按钮后，所有应用程序都会进入

后台状态，并且大部分都会迅速进入暂停状态，应用程序的所有工作内存都在RAM中，在暂停时它完全不执行。因此，切换回这样的应用程序非常快。但是如果系统需要更多的内存给当前处于活动状态的应用程序，就有可能终结暂停状态的应用程序，它们的内存也将被释放。

一方面，应用程序在进入后台状态时，需要释放一些资源，使自身的暂停快照更小，从而减少从RAM中清除的风险，另一方面，为了避免被终结而丢失用户的数据，需要在用户离开时保存他们的进度信息，这些工作，需要在5秒钟内完成，不然会被系统认定有异常被强制退出。可能通过接收应用程序发送的通知（UIApplicationDidEnterBackgroundNotification）来触发处理，如果在处理代码中加上下面这条语句则必然会导致异常退出：

view plain
[NSThread sleepForTimeInterval:10];
可以通过一种方法来请求更多后台时间来避免此问题。假设接收通知而触发的处理方法是applicationDidEnterBackground:
view plain
-(void)applicationDidEnterBackground{
NSLog(@"%@",NSStringFromSelector(_cmd));

//得到当前应用程序的UIApplication对象
UIApplication *app = [UIApplication sharedApplication];

//一个后台任务标识符
UIBackgroundTaskIdentifier taskID;
taskID = [app beginBackgroundTaskWithExpirationHandler:^{
//如果系统觉得我们还是运行了太久，将执行这个程序块，并停止运行应用程序
[app endBackgroundTask:taskID];
}];
//UIBackgroundTaskInvalid表示系统没有为我们提供额外的时候
if (taskID == UIBackgroundTaskInvalid) {

```
NSLog(@"Failed to start background task!");
return;
}
NSLog(@"Starting background task with %f seconds remaining",
app.backgroundTimeRemaining);
[NSThread sleepForTimeInterval:10];
NSLog(@"Finishing background task with %f seconds
remaining",app.backgroundTimeRemaining);
//告诉系统我们完成了
[app endBackgroundTask:taskID];
}
```

文八：

- 上一篇

  函数属性__attribute__

- 下一篇

  UIBezierPath

UIWebView

```
- (NSString *)stringByEvaluatingJavaScriptFromString:(NSString
*)script;执行脚本
@protocol UIWebViewDelegate <NSObject>

@optional
- (BOOL)webView:(UIWebView *)webView
shouldStartLoadWithRequest:(NSURLRequest *)request
navigationType:(UIWebViewNavigationType)navigationType
{
 NSLog(@"request = %@",request);
    NSString *requesturl =[[request URL] absoluteString];
    NSArray *componets =[requesturl
componentsSeparatedByString:@"#"];
    if ([componets count]==1)
    {
        //Resize the content of webview
        CGRect rect_webview = m_mywebview.frame;
        rect_webview.size.height =
webView.scrollView.contentSize.height+80;
        m_mywebview.frame= rect_webview;
        [[m_mywebview scrollView] setScrollEnabled:NO];

        [m_mywebview setHidden:NO];

        scro_content.contentSize = CGSizeMake(device_size.width,
rect_webview.size.height+rect_webview.origin.y+50);

    }
    else
    {
        自定义协议
    }
}
- (void)webViewDidStartLoad:(UIWebView *)webView;
- (void)webViewDidFinishLoad:(UIWebView *)webView;
- (void)webView:(UIWebView *)webView didFailLoadWithError:
```

```
(NSError *)error;

@end


//脚本
function triggerEvents(action,params)
{
    var url ="#action="+action +"&params="+params;
    document.location = url;
}
```

calayer self.contents = (id)[image CGImage]; [self addSublayer:[_label layer]];

CoreAnimation

dispatch

多线程 runloop

运行时

KVO 优点缺点

KVO的条件：

1）类必须是符合KVC

2）类为属性发出KVO更改通知

3）基于的keys被适当地注册。

有两种机制来确保更改通知被发出。NSObject的自动支持并被默认可用于类所有遵循KVC的属性。

当通知被发出时，手动更改通知提供附加控制，并且需要额外的代码。你可以控制自动通知属性，通过子类化并实现automaticallyNotifiesObserversForKey:方法。

KVC [id setValue:@”” forKey@“”]

[account setName:@"Savings"];

// 使用 setValue:forKey:.

[account setValue:@"Savings" forKey:@"name"];

关于远程通知

wedgit

分享 UIActivity UIActivityViewController

Affine 放射 放射变换

@encode 是用于表示一个类型的字符串，

@defs 用于返回一个Objective-C类的struct结构，这个struct与原Objective-C类具有相同的内存布局。就象你所知的那样，Objective-C类可以理解成由基本的C struct加上额外的方法构成。

@compatibility_alis 是用于给一个类设置一个别名。这样就不用重构以前的类文件就可以用新的名字来替代原有名字。

@dynamic 是相对于 @synthesize的，它们用样用于修饰 @property，用

于生成对应的的getter和setter方法。但是@ dynamic表示这个成员变量的getter和setter方法并不是直接由编译器生成，而是手工生成或者运行时生成。

光有个年龄还不能满足对teacher的描述，我想加个profession实例来存teacher的专业。直观的想法是子类化Teacher，其实也可以用类别。

你需要了解一下 runtime 编程知识，关注一下 objc_setAssociatedObject 和 objc_getAssociatedObject 。

```
1.  //
2.  //  Teacher+Profession.m
3.  //
4.
5.  #import "Teacher+Profession.h"
6.  #import <objc/runtime.h>
7.
8.  const char *ProfessionType = "NSString *";
9.  @implementation Teacher (Profession)
10.
11. -(void)setProf:(NSString*)prof
12. {
13.
        objc_setAssociatedObject(self, ProfessionType, prof,
    OBJC_ASSOCIATION_RETAIN_NONATOMIC);
14. }
15.
16. -(NSString *)prof
```

```
{
    NSString *pro = objc_getAssociatedObject(self, ProfessionType);
    return pro;
}

@end
```

现在就可以通过setProf: 和 prof 来存取 teacher 的 profession 值了。

1）在给程序添加消息转发功能以前，必须覆盖两个方法，即methodSignatureForSelector:和forwardInvocation:。methodSignatureForSelector:的作用在于为另一个类实现的消息创建一个有效的方法签名。forwardInvocation:将选择器转发给一个真正实现了该消息的对象。

设计模式

频率	所属类型	模式名称	模式	简单定义
5	创建型	Singleton	单件	保证一个类只有一个实例，并提供一个访问它的全局访问点。
5	结构型	Composite	组合模式	将对象组合成树形结构以表示部分整体的关系，Composite使得用户对单个对象和组合对象的使用具有一致性。
5	结构型	FAÇADE	外观	为子系统中的一组接口提供一致的界面，facade提供了一高层接口，这个接口使得子系统更容易使用。

5	结构型	Proxy	代理	为其他对象提供一种代理以控制对这个对象的访问
5	行为型	Iterator	迭代器	提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。
5	行为型	Observer	观察者	定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知自动更新。
5	行为型	Template Method	模板方法	定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，Template Method使得子类可以不改变一个算法的结构即可以重定义该算法得某些特定步骤。
4	创建型	Abstract Factory	抽象工厂	提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类。
4	创建型	Factory Method	工厂方法	定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到了子类。
4	结构型	Adapter	适配器	将一类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。
4	结构型	Decorator	装饰	动态地给一个对象增加一些额外的职责，就增加的功能来说，Decorator模式相比生成子类更加灵活。
4	行为型	Command	命令	将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。
4	行为型	State	状态	允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类。
4	行为型	Strategy	策略模式	定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户。
3	创建型	Builder	生成器	将一个复杂对象的构建与他的表示相分离，使得同样的构建过程可以创建不同的表示。
3	结构型	Bridge	桥接	将抽象部分与它的实现部分相分离，使他们可以独立的变化。

3	行为型	China of Responsibility	职责链	使多个对象都有机会处理请求，从而避免请求的送发者和接收者之间的耦合关系
2	创建型	Prototype	原型	用原型实例指定创建对象的种类，并且通过拷贝这些原型来创建新的对象。
2	结构型	Flyweight	享元	享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。
2	行为型	Mediator	中介者	用一个中介对象封装一些列的对象交互。
2	行为型	Visitor	访问者模式	表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作。
1	行为型	Interpreter	解释器	给定一个语言，定义他的文法的一个表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。
1	行为型	Memento	备忘录	在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

Core Foundation 框架Core Foundation框架 (CoreFoundation.framework) 是一组C语言接口，它们为iOS应用程序提供基本数据管理和服务功能。下面列举该框架支持进行管理的数据以及可提供的服务：

群体数据类型 (数组、集合等)程序包字符串管理日期和时间管理原始数据块管理偏好管理URL及数据流操作线程和RunLoop端口和soket通讯Core Foundation框架和Foundation框架紧密相关，它们为相同功能提供接口，但Foundation框架提供Objective-C接口。如果您将Foundation对象和 Core Foundation类型掺杂使用，则可利用两个框架之间的 “toll-free bridging”。所谓的Toll-free bridging是说您可以在某个框架的方法或

函数同时使用Core Foundatio和Foundation 框架中的某些类型。很多数据类型支持这一特性，其中包括群体和字符串数据类型。每个框架的类和类型描述都会对某个对象是否为 toll-free bridged，应和什么对象桥接进行说明。

如需进一步信息，请阅读Core Foundation 框架参考。

自 Xcode4.2 开始导入ARC机制后，为了支持对象间的转型，Apple又增加了许多转型用的关键字。这一讲我们就来了解其用法，以及产生的理由。

引子我们先来看一下ARC无效的时候，我们写id类型转void*类型的写法：

id obj = [[NSObject alloc] init];void *p = obj;反过来，当把void*对象变回id类型时，只是简单地如下来写，

id obj = p;[obj release];但是上面的代码在ARC有效时，就有了下面的错误：

error: implicit conversion of an Objective-C pointer to ’ void *’ is disallowed with ARC void *p = obj; ^ error: implicit conversion of a non-Objective-C pointer type ’ void *’ to ’ id’ is disallowed with ARC id o = p; ^

__bridge为了解决这一问题，我们使用 __bridge 关键字来实现id类型与void*类型的相互转换。看下面的例子。

id obj = [[NSObject alloc] init]; void *p = (__bridge void *)obj; id o = (__bridge id)p;将Objective-C的对象类型用 __bridge 转换为 void* 类型和使用 __unsafe_unretained 关键字修饰的变量是一样的。被代入对象的所有者需要明确对象生命周期的管理，不要出现异常访问的问题。

除过 __bridge 以外，还有两个 __bridge 相关的类型转换关键字：

__bridge_transfer__bridge_retained接下来，我们将看看这两个关键字的区别。

__bridge_retained先来看使用 __bridge_retained 关键字的例子程序：

id obj = [[NSObject alloc] init]; void *p = (__bridge_retained void *)obj;从名字上我们应该能理解其意义：类型被转换时，其对象的所有权也将被变换后变量所持有。如果不是ARC代码，类似下面的实现：

id obj = [[NSObject alloc] init]; void *p = obj;[(id)p retain];可以用一个实际的例子验证，对象所有权是否被持有。

void *p = 0; {   id obj = [[NSObject alloc] init];   p = (__bridge_retained void *)obj;} NSLog(@"class=%@", [(__bridge id)p class]);出了大括号的范围后，p 仍然指向一个有效的实体。说明他拥有该对象的所有权，该对象没有因为出其定义范围而被销毁。

__bridge_transfer相反，当想把本来拥有对象所有权的变量，在类型转换后，让其释放原先所有权的时候，需要使用 __bridge_transfer 关键字。文字有点绕口，我们还是来看一段代码吧。

如果ARC无效的时候，我们可能需要写下面的代码。

// p 变量原先持有对象的所有权id obj = (id)p;[obj retain];[(id)p release];那么ARC有效后，我们可以用下面的代码来替换：

// p 变量原先持有对象的所有权id obj = (__bridge_transfer id)p;可以看出来，__bridge_retained 是编译器替我们做了 retain 操作，而 __bridge_transfer 是替我们做了 release1。

Toll-Free bridged在iOS世界，主要有两种对象：Objective-C 对象和 Core Foundation 对象0。Core Foundation 对象主要是有C语言实现的 Core Foundation Framework 的对象，其中也有对象引用计数的概念，只是不是 Cocoa Framework::Foundation Framework 的 retain/release，而是自身的 CFRetain/CFRelease 接口。

这两种对象间可以互相转换和操作，不使用ARC的时候，单纯的用C原因的类型转换，不需要消耗CPU的资源，所以叫做 Toll-Free bridged。比如NSArray和CFArrayRef, NSString和CFStringRef，他们虽然属于不同的Framework，但是具有相同的对象结构，所以可以用标准C的类型转换。

比如不使用ARC时，我们用下面的代码：

NSString *string = [NSString stringWithFormat:...];CFStringRef cfString = (CFStringRef)string;同样，Core Foundation类型向Objective-C类型转换时，也是简单地用标准C的类型转换即可。

但是在ARC有效的情况下，将出现类似下面的编译错误：

Cast of Objective-C pointer type ‘NSString *’ to C pointer type ‘CFStringRef’ (aka ‘const struct __CFString *’) requires a bridged cast Use __bridge to convert directly (no change in ownership) Use __bridge_retained to make an ARC object available as a +1 ‘CFStringRef’ (aka ‘const struct __CFString *’)错误中已经提示了我们需要怎样做：用 __bridge 或者 __bridge_retained 来转型，其差别就是变更对象的所有权。

正因为Objective-C是ARC管理的对象，而Core Foundation不是ARC管理的对象，所以才要特意这样转换，这与id类型向void*转换是一个概念。也就是说，当这两种类型（有ARC管理，没有ARC管理）在转换时，需要告诉编译器怎样处理对象的所有权。

上面的例子，使用 __bridge/__bridge_retained 后的代码如下：

NSString *string = [NSString stringWithFormat:...];CFStringRef cfString = (__bridge CFStringRef)string;只是单纯地执行了类型转换，没有进行所有权的转移，也就是说，当string对象被释放的时候，cfString也不能被使用了。

NSString *string = [NSString stringWithFormat:...];CFStringRef cfString = (__bridge_retained CFStringRef)string;...CFRelease(cfString); // 由于Core Foundation的对象不属于ARC的管理范畴，所以需要自己release使用 __bridge_retained 可以通过转换目标处（cfString）的 retain 处理，来使所有权转移。即使 string 变量被释放，cfString 还是可以使用具体的对象。只是有一点，由于Core Foundation的对象不属于ARC的管理范畴，所以需要自己release。

实际上，Core Foundation 内部，为了实现Core Foundation对象类型与Objective-C对象类型的相互转换，提供了下面的函数。

CFTypeRef CFBridgingRetain(id X) { return (__bridge_retained CFTypeRef)X;} id CFBridgingRelease(CFTypeRef X) { return (__bridge_transfer id)X;}所以，可以用 CFBridgingRetain 替代 __bridge_retained 关键字：

NSString *string = [NSString stringWithFormat:...];CFStringRef cfString = CFBridgingRetain(string);...CFRelease(cfString); // 由于Core Foundation不在ARC管理范围内，所以需要主动release。__bridge_transfer所有权被转移的同时，被转换变量将失去对象的所有权。当Core Foundation对象类型向Objective-C对象类型转换的时候，会经常用到 __bridge_transfer 关键字。

CFStringRef cfString = CFStringCreate...();NSString *string = (__bridge_transfer NSString *)cfString; // CFRelease(cfString);因为已经用 __bridge_transfer 转移了对象的所有权，所以不需要调用release同样，我们可以使用 CFBridgingRelease() 来代替 __bridge_transfer 关键字。

CFStringRef cfString = CFStringCreate...();NSString *string = CFBridgingRelease(cfString);

总结由上面的学习我们了解到 ARC 中类型转换的用法，那么我们实际使用中按照怎样的原则或者方法来区分使用呢，下面我总结了几点关键要素。

明确被转换类型是否是 ARC 管理的对象Core Foundation 对象类型不在 ARC 管理范畴内Cocoa Framework::Foundation 对象类型（即一般使用到的Objectie-C对象类型）在 ARC 的管理范畴内如果不在 ARC 管理范畴内的对象，那么要清楚 release 的责任应该是谁各种对象的生命周期是怎样的1. 声明 id obj 的时候，其实是缺省的申明了一个 __strong 修饰的变量，所以编译器自动地加入了 retain 的处理，所以说 __bridge_transfer 关键字只为我们做了 release 处理。

研究Core Animation已经有段时间了，关于Core Animation，网上没什么好的介绍。苹果网站上有篇专门的总结性介绍，但是似乎原理性的东西不多，看得人云山雾罩，感觉，写那篇东西的人，其实是假设读的人了解界面动画技术的原理的。今天有点别的事情要使用Linux，忘掉了ssh的密码，没办法重新设ssh，结果怎么也想不起来怎么设ssh远程登陆了，没办法又到网上查了一遍，太浪费时间了，痛感忘记记笔记是多么可怕的事情。鉴于Core Animation的内容实在是非常繁杂，应用的Obj-C语言本身的特性也很多，所以写个备忘录记录一下，懂的人看了后如果发现了错误，还不吝指教。

1.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。它本身完全是由CoreAnimation来实现的（Mac下似乎不是这样）。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，访问它的跟绘图和跟坐标有关的属性，例如frame，bounds等等，实际上内部都是在访问它所包含的CALayer的相关属性。

2.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，例如通过

```
1
- (class) layerClass {
2
  return ([CAEAGLLayer class]);
3
}
```

使某个UIView的子类使用GL来进行绘制。

3.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，来完成某些特殊的表示。例如下面的代码

```
1
grayCover = [[CALayer alloc] init];
2
grayCover.backgroundColor = [[[UIColor blackColor]
colorWithAlphaComponent:0.2] CGColor];
3
[self.layer addSubLayer: grayCover];
```

会在目标View上敷上一层黑色的透明薄膜。

4.UIView的layer树形在系统内部，被系统维护着三份copy（这段理解有点吃不准）。

第一份，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。

第二份，动画树，这是一个中间层，系统正在这一层上更改属性，进行各种渲染操作。

第三份，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.动画的运作

UIView的主layer以外（我觉得是这样），对它的subLayer，也就是子layer的属性进行更改，系统将自动进行动画生成，动画持续时间有个缺省时间，个人感觉大概是0.5秒。在动画时间里，系统自动判定哪些属性更改了，自动对更改的属性进行动画插值，生成中间帧然后连续显示产生动画效果。

6.坐标系系统（对position和anchorPoint的关系还是犯晕）

CALayer的坐标系系统和UIView有点不一样，它多了一个叫anchorPoint的属性，它使用CGPoint结构，但是值域是0~1，也就是按照比例来设置。这个点是各种图形变换的坐标原点，同时会更改layer的position的位置，它的缺省值是{0.5, 0.5}，也就是在layer的中央。
某layer.anchorPoint = CGPointMake(0.f, 0.f);
如果这么设置，layer的左上角就会被挪到原来的中间的位置，
加上这样一句就好了
某layer.position = CGPointMake(0.f, 0.f);

7.真实例子的分析

这是iphone上iBook翻页的效果，假设每一页都是一个UIView，我觉得一个页面是贴了俩个Layer，文字Layer显示正面的内容，背面layer用文字layer的快照做affine翻转，贴在文字layer的后面。因为Layer可以设置显示阴影，也许后面的阴影效果没有使用单独的一个layer来显示。至于这个曲面效果，我查了很多资料也没有结果，估计是使用了GL的曲面绘图?

8.最后一个让人恶心的。
layer可以设置圆角显示，例如UIButton的效果，也可以设置阴影显示，但是如果layer树中的某个layer设置了圆角，树中所有layer的阴影效果都将显示不了了。如果既想有圆角又想要阴影，好像只能做两个重叠的UIView，一个的layer显示圆角，一个的layer显示阴影.....

CALayer属于Core Animation部分的内容，比较重要而不太好理解。以下是园子中看到的一篇文章的摘录：

1. UIView是iOS系统中界面元素的基础，所有的界面元素都是继承自它。它

本身完全是由CoreAnimation来实现的。**它真正的绘图部分，是由一个CALayer类来管理。**UIView本身更像是一个CALayer的管理器，访问它的跟绘图和跟坐标有关的属性，例如frame，bounds等，实际上内部都是在访问它所包含的CALayer的相关属性。

2. UIView有个重要属性layer，可以返回它的主CALayer实例。

```
// 要访问层，读取UIView实例的layer属性
CALayer *layer = myView.layer
```

所有从UIView继承来的对象都继承了这个属性。这意味着你可以转换、缩放、旋转，甚至可以在Navigation bars，Tables，Text boxes等其它的View类上增加动画。每个UIView都有一个层，控制着各自的内容最终被显示在屏幕上的方式。

UIView的layerClass方法，可以返回主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。代码示例：

```
- (class)layerClass {
   return ([CAEAGLLayer class]);
}
```

上述代码使得某个UIView的子类使用GL来进行绘制。

3. 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，来完成某些特殊的表示。即CALayer层是可以嵌套的。

示例代码：

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

上述代码会在目标View上敷上一层黑色透明薄膜的效果。

4. UIView的layer树形在系统内部，被维护着三份copy。分别是逻辑树，这里是代码可以操纵的；动画树，是一个中间层，系统就在这一层上更改属性，进行各种渲染操作；显示树，其内容就是当前正被显示在屏幕上得内容。

5. 动画的运作：对UIView的subLayer（非主Layer）属性进行更改，系

统将自动进行动画生成，动画持续时间的缺省值似乎是0.5秒。

6. 坐标系统：**CALayer的坐标系统比UIView多了一个anchorPoint属性，使用CGPoint结构表示，值域是0~1，是个比例值**。这个点是各种图形变换的坐标原点，同时会更改layer的position的位置，它的缺省值是{0.5,0.5}，即在layer的中央。

某layer.anchorPoint = CGPointMake(0.f,0.f);

如果这么设置，只会将layer的左上角被挪到原来的中间位置，必须加上这一句：

某layer.position = CGPointMake(0.f,0.f);

最后：layer可以设置圆角显示（cornerRadius），也可以设置阴影 (shadowColor)。但是如果layer树中某个layer设置了圆角，树种所有layer的阴影效果都将不显示了。因此若是要有圆角又要阴影，变通方法只能做两个重叠的UIView，一个的layer显示圆角，一个layer显示阴影.....

7.渲染：当更新层，改变不能立即显示在屏幕上。当所有的层都准备好时，可以调用setNeedsDisplay方法来重绘显示。

```
[gameLayer setNeedsDisplay];
```

若要重绘部分屏幕区域，请使用setNeedsDisplayInRect:方法，通过在CGRect结构的区域更新：

```
[gameLayer
setNeedsDisplayInRect:CGRectMake(150.0,100.0,50.0,75.0)];
```

如果是用的Core Graphics框架来执行渲染的话，可以直接渲染Core Graphics的内容。用renderInContext:来做这个事。

```
[gameLayer renderInContext:UIGraphicsGetCurrentContext()];
```

8.变换：要在一个层中添加一个3D或仿射变换，可以分别设置层的transform或affineTransform属性。

```
characterView.layer.transform =
CATransform3DMakeScale(-1.0,-1.0,1.0);

CGAffineTransform transform =
CGAffineTransformMakeRotation(45.0);
```

```
backgroundView.layer.affineTransform = transform;
```

9.变形：Quartz Core的渲染能力，使二维图像可以被自由操纵，就好像是三维的。图像可以在一个三维坐标系中以任意角度被旋转，缩放和倾斜。CATransform3D的一套方法提供了一些魔术般的变换效果。

NSCoder　归档

View Code //

// UserEntity.m

// NSCodingDemo

//

// Created by   on 11-5-20.

// Copyright 2011 , Inc. All rights reserved.

//

#import "UserEntity.h"

#import "AddressEntity.h"

#define SAMPLEDATA_KEY_USERNAME (@"userName")

#define SAMPLEDATA_KEY_GENDER (@"gender")

#define SAMPLEDATA_KEY_AGE (@"age")

#define k_FRIENDS (@"frinds")

#define k_ADDRESS (@"address")

#define k_userEntity_addresses (@"addresses")

```
@implementation UserEntity
@synthesize userName = _userName;
@synthesize gender   = _gender;
@synthesize age      = _age;
@synthesize friends  = _friends;
@synthesize address  = _address;
@synthesize addresses = _addresses;

-(void)dealloc
{
    [self.userName release];
    [self.friends release];
    [self.address release];
    [self.addresses release];
    [super dealloc];
}

  - (void)encodeWithCoder:(NSCoder*)coder
  {
      [coder encodeObject:self.userName
forKey:SAMPLEDATA_KEY_USERNAME];
      [coder encodeInt:self.age forKey:SAMPLEDATA_KEY_AGE];
      [coder encodeBool:self.gender
forKey:SAMPLEDATA_KEY_GENDER];
      [coder encodeObject:self.friends forKey:k_FRIENDS];
      [coder encodeObject:self.address forKey:k_ADDRESS];
      [coder encodeObject:self.addresses
```

```
forKey:k_userEntity_addresses];
 }

 - (id)initWithCoder:(NSCoder*)decoder
 {
    if (self = [super init])
    {
        if (decoder == nil)
        {
            return self;
        }
        self.userName = [decoder
decodeObjectForKey:SAMPLEDATA_KEY_USERNAME];
        self.gender = [decoder
decodeBoolForKey:SAMPLEDATA_KEY_GENDER];
        self.age = [decoder decodeIntForKey:SAMPLEDATA_KEY_AGE];
        self.friends = [decoder decodeObjectForKey:k_FRIENDS];
        self.address = [decoder decodeObjectForKey:k_ADDRESS];
        self.addresses = [decoder decodeObjectForKey:
k_userEntity_addresses];
    }
    return self;
 }

@end
View Code //
//  AddressEntity.m
```

```
//  NSCodingDemo
//
//  Created by   on 11-5-20.
//  Copyright 2011  , Inc. All rights reserved.
//

#import "AddressEntity.h"
#define k_Id (@"id")
#define k_AddresEntity_address (@"addrss")


@implementation AddressEntity
@synthesize Id, address;
-(void)dealloc
{
    [self.address release];
    [super dealloc];
}
- (NSString *)description
{

    return [NSString stringWithFormat:@"id :%d address: %@" , self.Id,
self.address];
}

- (void)encodeWithCoder:(NSCoder*)coder
{
```

```
    [coder encodeObject:self.address forKey:k_AddresEntity_address];
    [coder encodeInt:self.Id forKey:k_Id];

}

- (id)initWithCoder:(NSCoder*)decoder
{
    if (self = [super init])
    {
        if (decoder == nil)
        {
            return self;
        }
        self.address = [decoder
decodeObjectForKey:k_AddresEntity_address];
        self.Id = [decoder decodeIntForKey:k_Id];

    }
    return self;
}
@end
View Code -(IBAction)write:(id)sender
{
    //get document path
    NSArray *paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
```

```
    NSString *documentsDirectory = [paths objectAtIndex:0];

    NSString *filePath = [documentsDirectory
stringByAppendingPathComponent:k_FILE_NAME];

    NSMutableData *data = [[[NSMutableData alloc] init] autorelease];
    NSKeyedArchiver *archiver = [[[NSKeyedArchiver alloc]
initForWritingWithMutableData:data] autorelease];
    UserEntity *userEntity = [[[UserEntity alloc] init] autorelease];
    userEntity.age = 18;
    userEntity.gender = NO;
    userEntity.userName = @"add";

    NSArray *friends = [[NSArray alloc] initWithObjects:@"ac", @"2",
@"3", @"4", nil];
    userEntity.friends = friends;

    AddressEntity *address = [[[AddressEntity alloc] init] autorelease];
    address.Id = 1;
    address.address = @"china";
    userEntity.address = address;

    AddressEntity *address2 = [[[AddressEntity alloc] init] autorelease];
    address2.Id = 2;
    address2.address = @"china2";

    AddressEntity *address3 = [[[AddressEntity alloc] init] autorelease];
```

```
    address3.Id = 3;
    address3.address = @"china3";

    NSArray *addresses = [[NSArray alloc] initWithObjects:address2,
address3 , nil];

    userEntity.addresses = addresses;

    [archiver encodeObject:userEntity forKey:@"KEY"];
    [archiver finishEncoding];
    BOOL success = [data writeToFile:filePath atomically:YES];



}

-(IBAction)read:(id)sender
{
    NSArray *paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];

    NSString *filePath = [documentsDirectory
stringByAppendingPathComponent:k_FILE_NAME];

    NSData *data = [[NSData alloc] initWithContentsOfFile:filePath];
```

```
    NSKeyedUnarchiver *unArchiver = [[NSKeyedUnarchiver alloc]
initForReadingWithData:data];
    UserEntity *userEntity = [unArchiver decodeObjectForKey:@"KEY"];
    NSLog(@"name is %@ , friend %@ address :%@ , addresses is :
%@", userEntity.userName , userEntity.friends , userEntity.address,
userEntity.addresses);

}
```

5什么是键-值,键路径是什么

模型的性质是通过一个简单的键（通常是个字符串）来指定的。视图和控制器通过键来查找相应的属性值。在一个给定的实体中，同一个属性的所有值具有相同的数据类型。键-值编码技术用于进行这样的查找—它是一种间接访问对象属性的机制。

键路径是一个由用点作分隔符的键组成的字符串，用于指定一个连接在一起的对象性质序列。第一个键的

性质是由先前的性质决定的，接下来每个键的值也是相对于其前面的性质。键路径使您可以以独立于模型

实现的方式指定相关对象的性质。通过键路径，您可以指定对象图中的一个任意深度的路径，使其指向相

关对象的特定属性。

26What are KVO and KVC?

答案：kvc:键 - 值编码是一种间接访问对象的属性使用字符串来标识属

性，而不是通过调用存取方法，直接或通过实例变量访问的机制。
很多情况下可以简化程序代码。apple文档其实给了一个很好的例子。
kvo:键值观察机制，他提供了观察某一属性变化的方法，极大的简化了代码。
具体用看到嗯哼用到过的一个地方是对于按钮点击变化状态的的监控。

6目标-动作机制

目标是动作消息的接收者。一个控件，或者更为常见的是它的单元，以插座变量（参见"插座变量"部分）
的形式保有其动作消息的目标。
动作是控件发送给目标的消息，或者从目标的角度看，它是目标为了响应动作而实现的方法。
程序需要某些机制来进行事件和指令的翻译。这个机制就是目标-动作机制。

11动态绑定
一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。

12**obj-c**的优缺点

objc优点：

1) Cateogies

2) Posing

3) 动态识别

4) 指标计算

5）弹性讯息传递

6) 不是一个过度复杂的 C 衍生语言

7) Objective-C 与 C++ 可混合编程

缺点:

1) 不支援命名空间

2) 不支持运算符重载

3）不支持多重继承

4）使用动态运行时类型，所有的方法都是函数调用，所以很多编译时优化方法都用不到。（如内联函数等），性能低劣。

strcpy是一个字符串拷贝的函数，它的函数原型为strcpy(char *dst, c*****t char *src);

将 src开始的一段字符串拷贝到dst开始的内存中去，结束的标志符号为'\0'，由于拷贝的长度不是由我们自己控制的，所以这个字符串拷贝很

容易出错。具备字符串拷贝功能的函数有memcpy，这是一个内存拷贝函数，它的函数原型为memcpy(char *dst, c*****t char* src, unsigned int len);
将长度为len的一段内存，从src拷贝到dst中去，这个函数的长度可控。但是会有内存叠加的问题。
sprintf是格式化函数。将一段数据通过特定的格式，格式化到一个字符串缓冲区中去。sprintf格式化的函数的长度不可控，有可能格式化后的字符串会超出缓冲区的大小，造成溢出。

浅复制和深复制的区别?
答案：浅层复制：只复制指向对象的指针，而不复制引用对象本身。
深层复制：复制引用对象本身。

类别　　类扩展 （）括号中没有名字的。可以添加成员、方法

26What are KVO and KVC?
答案：kvc:键 - 值编码是一种间接访问对象的属性使用字符串来标识属性，而不是通过调用存取方法，直接或通过实例变量访问的机制。
很多情况下可以简化程序代码。apple文档其实给了一个很好的例子。
kvo:键值观察机制，他提供了观察某一属性变化的方法，极大的简化了代码。
具体用看到嗯哼用到过的一个地方是对于按钮点击变化状态的的监控。

比如我自定义的一个button

[cpp]

```
[self addObserver:self forKeyPath:@"highlighted" options:0 context:nil];
#pragma mark KVO
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary *)change context:(void *)context
{
    if ([keyPath isEqualToString:@"highlighted"] ) {
        [self setNeedsDisplay];
    }
}
```

对于系统是根据keypath去取的到相应的值发生改变，理论上来说是和kvc机制的道理是一样的。

对于kvc机制如何通过key寻找到value：

“当通过KVC调用对象时，比如：[self valueForKey:@”someKey”]时，程序会自动试图通过几种不同的方式解析这个调用。首先查找对象是否带有someKey 这个方法，如果没找到，会继续查找对象是否带有someKey这个实例变量（iVar），如果还没有找到，程序会继续试图调用 -(id) valueForUndefinedKey:这个方法。如果这个方法还是没有被实现的话，程序会抛出一个NSUndefinedKeyException异常错误。

(cocoachina.com注：Key-Value Coding查找方法的时候，不仅仅会查找someKey这个方法，还会查找getsomeKey这个方法，前面加一个get，或者_someKey以及_getsomeKey这几种形式。同时，查找实例变量的时候也会不仅仅查找someKey这个变量，也会查找_someKey这个变量是否存在。）

设计valueForUndefinedKey:方法的主要目的是当你使用-(id)valueForKey方法从对象中请求值时，对象能够在错误发生前，有最后的机会响应这个请求。这样做有很多好处，下面的两个例子说明了这样做的好处。“

来至cocoa，这个说法应该挺有道理。
因为我们知道button却是存在一个highlighted实例变量.因此为何上面我们只是add一个相关的keypath就行了，