UI笔记

告别OC阶段，UI无疑能让我们激动。
新建一个iPhone工程，不再像之前一样。

1、新建UI工程

xcode5.1

1、create a new xcode peoject
2、iOS/application/empty application/next
3、product Name :填写英文工程名，不能有其他符号，空格，加减什么的
4、Class prefix: 每个类，在前面加上前缀，用以识别，防止重复类名出现。
5、Devices：可选iPhone，iPad，universal

xcode6

1、create a new xcode peoject
2、iOS/application/Single View application/next
3、product Name :填写英文工程名，不能有其他符号，空格，加减什么的
4、 language:objective-c
5、Devices：可选iPhone，iPad，universal
创建出来的UI工程，有一对AppDelegate.h和AppDelegate.m是一个代理的实现。在.m里的很多方法，都是通过实现<UIApplicationDelegate>代理而生成的方法。.h里只有一个属性
`@property (strong, nonatomic) UIWindow *window;//`这个属性是arc下的声明，用了strong，strong相当于retain
`@property (retain, nonatomic) UIWindow *window;//` 这个属性是mrc下的声明，用了retain。

2、UIWindow（画板）

window 相当于一个画板，要想展现自己想要的图像或者图形，需要把画的东

西画在window这个画板上。
window继承于UIView，在ui中，所有的能看得到的东西，都是继承于UIView。
在iOS中，通常用UIWindow来表示窗口，每个app都要把要展现的东西都写在UIWindow上。通常，一个app只创建一个UIWindow对象。

创建UIWindow对象

```
//2015年04月28日11:01:39  北京·蓝欧

    // 创建window对象，初始化时设置window的位置和大小与屏幕相同。
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];

    // 给window设置背景颜色
    self.window.backgroundColor = [UIColor whiteColor];

    // 让我们的window显示
    [self.window makeKeyAndVisible];
```

解释：

1、initWithFrame:[[UIScreen mainScreen] bounds]，初始化window，使这个window跟屏幕一样大小。
2、backgroundColor，设置背景色
3、makeKeyAndVisible，把window设置成可显示的。

3、UIView（视图）

任何在屏幕上显示的图形，都是矩形，只是都做了处理。因为，所有在iOS里能看到的，都是UIView或UIview的子类。

创建view

```
    // 创建View
    // 1.创建对象
    UIView *view1 = [[UIView
alloc]initWithFrame:CGRectMake(100, 122, 100, 100)];

    // 2.设置view1属性（颜色就是属性之一）
    // 设置背景颜色
    view1.backgroundColor = [UIColor yellowColor];

    // 3.添加视图
    // view1 的父视图是window
    // window 的子视图是view1
    // 添加视图时，系统对视图进行retain。我们把这个retain叫做持有。
（window持有）
    [self.window addSubview:view1];

    // 4.释放
    [view1 release];
    view1 = nil;
```

解释：

1、CGRectMake，是一个返回值结构体CGRect，有4个参数的方法，用来确定我们所创建的视图的位置和大小。
2、CGRect是一个结构体，有两个成员变量CGPoint、CGSize，这两个成员变量都是结构体，每个成员变量里都分别有两个成员变量，都是CGFloat类型的。
3、CGPoint的两个成员变量是用来确定我们创建视图的位置。
4、CGSize的两个成员变量是用来确定我们的创建的视图的大小，一个为宽，一个为高。
5、起始位置：从屏幕的最左上角为（0,0）点，往下为正Y轴，右为正X轴。
6、addSubview，把我们创建的视图，添加到window中，这样，才能使我们的视图可见。

4、frame

frame其实也是一个结构体，是结构体CGRect的一个变量。

```
    // frame ,其实就是一个结构体，有两个成员变量，均为结构体，两个成员
变量里分别有两个成员变量都是cgfloat类型的。
    // 根据父视图来得。
    UIView *view4 = [[UIView alloc]init];
    view4.frame = CGRectMake(220, 20, 100, 50);

    view4.backgroundColor = [UIColor greenColor];
    [self.window addSubview:view4];

    [view4 release];
    view4 = nil;
```

5、bounds

在每添加一个view的时候，每个view分别以自己的起点为（0，0）点，创建一个坐标系，这个bounds就是一个结构体，用来确定view的新坐标。

```
//    view1.bounds
    // bounds也是结构体，与frame一样。只是坐标是从自己的原点开始。
    // 当视图创建之后，会产生自己的一套坐标系。给自己的子视图用。起点也是左上角开始。
      NSLog(@"bounds:%@",NSStringFromCGRect(view1.bounds));//bounds:{{0, 0}, {200, 200}}。
```

frame与bounds的区别：

frame与bounds都是结构体，都是用来确定位置和大小的；

frame的位置是从屏幕的最左上角开始算的；

bounds是从新添加的view的最左上角的点开始算。

```
    // view1 frame
    // frame 以父视图坐标系为参照，添加视图。
    // frame 其实就是一个结构体，有两个成员变量，也是结构体，分别 有两个成员变量都是cgfloat类型的。
    NSLog(@"frame:%@",NSStringFromCGRect(view1.frame));//frame:{{50, 50}, {200, 200}}

    // view1 bounds
//    view1.bounds
    // bounds也是结构体，与frame一样。只是坐标是从自己的原点开始。
    // 当视图创建之后，会产生自己的一套坐标系。给自己的子视图用。起点也是左上角开始。
   NSLog(@"bounds:%@",NSStringFromCGRect(view1.bounds));//bounds:{{0, 0}, {200, 200}}。
    // 创建一个subView1
    UIView *subView1 = [[UIView alloc]initWithFrame:CGRectMake(20, 20, 100, 100)];
    subView1.backgroundColor = [UIColor yellowColor];

//    [self.window addSubview:subView1];// 用的是window的坐标系

    [view1 addSubview:subView1];// 用的是view1的坐标系


    // center
    // center是以父视图坐标系为参照，取得的中心点。
        NSLog(@"%@",NSStringFromCGPoint(subView1.center));//{70, 70}
```

center用的是父视图的坐标系。

6、添加子视图

insertSubview:atIndex:

把一个视图插入到指定位置，跟数组一样，位置都从0开始算。若插入的位置大于等于最顶层视图，那么这个视图就插在最顶层。

insertSubview:aboveSubview:

插入一个视图在最顶层。

insertSubview:belowSubview:

插入一个视图在最底层

```
    // 插入到指定位置，这个位置从0开始。
    // 如果插入的位置大于最顶层，则就一直在最顶层。
    [self.window insertSubview:greenView atIndex:3];
    [self.window insertSubview:greenView atIndex:0];
    [self.window insertSubview:greenView atIndex:1];
    [self.window insertSubview:greenView atIndex:2];

    [self.window insertSubview:greenView
belowSubview:redView];
          [self.window insertSubview:greenView
aboveSubview:blueView];
```

7、视图层次管理

bringSubviewToFront:

把指定的子视图拿到最顶层

sendSubviewToBack:

把指定的子视图移动到最后面。

exchangeSubviewAtIndex:withSubviewAtIndex:

把一个位置的视图与另一个位置的视图交换位置。

removeFromSuperview

把某个视图自己从父视图中移除。谁要移除，谁就去调这个方法。

```
    // 视图层次管理
    // 把某个子视图带到最前面
    [self.window bringSubviewToFront:redView];
    // 把某个子视图送到最后面
    [self.window sendSubviewToBack:greenView];

    // 交换两个视图的位置
    [self.window exchangeSubviewAtIndex:0 withSubviewAtIndex:
1];

    // 从父视图把自己移出，谁要移除来，谁就去调这个方法
    [ greenView removeFromSuperview];
```

8、视图属性

hidden

隐藏视图，为BOOL值，YES为隐藏，NO为不隐藏。默认为NO。

alpha

用来控制一个视图的透明度。alpha，是一个CGFloat类型的变量，值在0~1之间的小数，值越大，越不透明。

superview

用来获取父视图，若是无父视图，得到的是null。

subviews

获取所有自己的子视图。返回值是一个NSArray类型的数组，里边元素是每个视图。

tag

用来标记一个视图。有点像往一个视图上贴标签。被标记的视图，可以用viewWithTag: 取出。
AppDelegate.m

```objc
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]] autorelease];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    UIView *view1 = [[UIView
alloc]initWithFrame:CGRectMake(100, 100, 100, 100)];
    view1.backgroundColor = [UIColor blueColor];
    [self.window addSubview:view1];

    UIView *view2 = [[UIView
alloc]initWithFrame:CGRectMake(200, 100, 100, 100)];
    view2.backgroundColor = [UIColor greenColor];
//    [self.window addSubview:view2];

    // hidden
    view1.hidden = NO;

    if (view1.isHidden) {
        NSLog(@"该视图被隐藏了..");

    }else{
        NSLog(@"该视图没有隐藏..");
    }

    view1.hidden = NO;
//
    view2.hidden = YES;

    // alpha透明度，不常用。会让自己的子视图都一起透明
    view1.alpha = 0.2;
    view2.alpha = 0.2;

    // 设置颜色和透明度最常用的、
    // RGB不会让子视图透明
    view1.backgroundColor = [UIColor colorWithRed:0.6 green:
0.1 blue:0.1 alpha:0.5];

    // superview
    // 获取父视图
    UIView *superView = view1.superview;
    NSLog(@"%@",[superView class]);// UIWindow



    // subview
    // 获取子视图
    NSArray *subViews = self.window.subviews;
    NSLog(@"%@",subViews);// ("<UIView: 0x7fc409445610; frame
= (100 100; 100 100); layer = <CALayer: 0x7fc4094458e0>>")

    // tag 通常赋值，赋100以上
    // 一个标签
    // 跨方法，跨文件取视图
```

9、UILabel

UILabel是用来显示文本的控件，是在app中出镜率最高的控件。UILabel同样是UIView的子类。，主要是自己扩展了UIView的显示文字功能。
创建一个UILabel，遵循以下几个步骤来完成：
1、alloc开辟空间，然后initWithFrame来初始化大小；
2、设置UILabel 的相关属性，也就是我们想要的属性
3、把label添加到父视图（UIWindow）上，才可以显示出来；
4、释放label
AppDelegate.m

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

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

    self.window.backgroundColor = [UIColor colorWithRed:0.4
green:0.9 blue:0.1 alpha:1];
    [self.window makeKeyAndVisible];

    UILabel *lable = [[UILabel
alloc]initWithFrame:CGRectMake(100, 100, 200, 50)];
    lable.backgroundColor = [UIColor colorWithRed:1 green:1
blue:1 alpha:0.7];
    [lable release];
    lable = nil;

    return YES;
    }
```

UILabel属性

text

要显示的文本内容label.text = @”呵呵”;

textColor

用来修改要显示的文字的颜色label.textColor = [UIColor redColor];

textAlignment

要显示的文字的对其方式（水平方向）

label.textAlignment = NSTextAlignmentLeft;

font

设置文字的字体和字号label.font = [UIFont fontWithName:@“Helvetica-Bold” size：20]

numberOfLines

在label若是文字太长，就会自动省略后面的字，numberOfLines是为了展示省略的文字。给多少行，就显示多少行，显示完毕为止。

lineBreakMode

断行的模式，根据什么换行
label.lineBreakMode = NSLineBreakByWordWrapping;根据单词为单位换行

shadowColor

设置文字的阴影颜色

shadowOffset

设置文字的阴影大小。label.shadowOffset = CGSizeMake(2,1);//阴影向x正方向偏移2，向y正方向偏移1

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

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

    self.window.backgroundColor = [UIColor colorWithRed:0.4
green:0.9 blue:0.1 alpha:1];
    [self.window makeKeyAndVisible];

    UILabel *lable = [[UILabel
alloc]initWithFrame:CGRectMake(100, 100, 200, 50)];
    lable.backgroundColor = [UIColor colorWithRed:1 green:1
blue:1 alpha:0.7];
    // 设置文字
    lable.textColor = [UIColor grayColor];
    // 设置文字颜色
    lable.text = @"用户登录
111111111111111111111111111111111111111111111111111";
    // 设置对齐方式
    lable.textAlignment = NSTextAlignmentCenter;

    // 行数
    lable.numberOfLines = 0;

    // 字体
    // 设置字体，设置size才有效
//    lable.font = [UIFont fontWithName:@"Papyrus" size:17];
    // 不用知道字体也可以设置大小..

    // 打印所有字体
    NSLog(@"%@",[UIFont familyNames]);
    //阴影效果
//    lable.shadowColor = [UIColor redColor];
//
//    lable.shadowOffset = CGSizeMake(1, 1);

    [self.window addSubview:lable];

    [lable release];
    lable = nil;

    return YES;
}
```

——大海
2015年04月29日21:33:09 北京

10、UITextField输入框

UITextField是控制文字的输入和显示的控件。
所谓输入，在iOS里就是，点击输入框的时候，会弹出键盘，并能把键盘收回。
相比于UILabel，UITextField不仅能显示文字，更能输入文字。
创建UITextField遵循以下几个步骤:
1、alloc开辟空间，initWithFrame初始化与屏幕的大小。
2、设置UITextField的相关属性;
3、把UITextField添加到父视图，得以显示出来;
4、释放UITextField对象。
UITextField继承于UIControl，后者继承于UIview。所以，UITextField对象拥有UIView和UIControl的所有方法和属性。

UITextField文本显示

text

属性text表示要显示的内容textField.text = @“lanoukeji”;此时，在输入框中，有lanoukeji字样。这个text会获取输入框所有的文字。

textColor

要显示的文本的颜色

textAlignment

输入框文本的对齐方式（水平方向）。一般输入框，我们设置对齐方式为左对齐。

font

输入框中，字体的样式和大小

placeholder

在输入框中没有文字时，placeholder提供了占位符，用以提示
textField.placeholder = @“请输入用户名”;

```
    // 文本显示
    // UITextField

    UITextField *textField1 = [[UITextField
alloc]initWithFrame:CGRectMake(50, 50, 200, 50)];

    [self.window addSubview:textField1];
    // 属性
    textField1.backgroundColor = [UIColor whiteColor];
    // 用户输入框中得值，text就得到什么值（都是字符串）
    // 文本框里的值是什么，text就得到什么。
    textField1.text = @"hello MOTO";
    // 颜色
    textField1.textColor = [UIColor grayColor];
    // 对齐方式，一般用左对齐
    textField1.textAlignment = NSTextAlignmentLeft;
    // 设置字体
    textField1.font = [UIFont fontWithName:@"Helvetica-Bold"
size:17];
    // 不用设置字体也可以改变大小
    textField1.font = [UIFont systemFontOfSize:27];

    //占位字符串。
    textField1.placeholder = @"输入用户名";
```

输入控制

enabled

是否允许输入框输入东西。返回值是BOOl类型。默认是YES，允许输入。

clearsOnBeginEditing

是否开始输入的时候，清空输入框的文字内容。返回值是BOOL类型的。默认值是NO不清空。

secureTextEntry

是否输入的文字用圆点来代替显示，返回值是BOOL值，YES是用圆点来代替显示。默认值是NO，不用圆点显示。
secureTextEntry一般用在密码类型的输入框，屏蔽真实输入内容。
textField.secureTextEntry = YES;//密码模式

keyboardType

控制点击输入框时，弹出来的键盘是什么类型的键盘（枚举值）。
textField.keyboardType = UIKeyboardTypeNumberPad;弹出来的是数字键盘

returnKeyType

弹出来的键盘最右下角的return键的类型。也就是说，键盘的return键的字是可以改变的。

inputView

自定义输入视图（默认是键盘）。也就是说，在点击输入框的时候，自定义弹出来一个能输入的视图（键盘就是）。

inputAccessoryView

在弹出键盘的上方，一起弹出一个辅助视图，由我们自己定义辅助视图实现的功能。（默认是nil，也就是弹出来的视图上什么都没有）
textField.inputAccessoryView = myAccessoryView;

```
    // 输入控制
    // enabled 是否允许输入
    textField1.enabled = YES;

    // 开始输入是否清空输入框
    textField1.clearsOnBeginEditing = YES;

    // 密码格式
    textField1.secureTextEntry = NO;
    // 键盘样式
    textField1.keyboardType = UIKeyboardTypeDefault;

    // 修改return键样式
    textField1.returnKeyType = UIReturnKeyGoogle;

    // inputView 可以自己写一个键盘，只用给高度就行了。
    // 点击文本框，弹出一个东西代替系统键盘

    UIView *tempView = [[UIView
alloc]initWithFrame:CGRectMake(0, 0, 0, 30)];
    tempView.backgroundColor = [UIColor redColor];
//
//    textField1.inputView = tempView;

    // inputAccessoryView
    textField1.inputAccessoryView = tempView;
```

外观控制

borderStyle

输入框的边框样式，是一个枚举值。
textField.borderStyle = UITextBorderStyleRoundedRect;这里是我们用的最多的，也就是将矩形处理成圆角矩形。

clearButtonMode

清除按钮模式，也是一个枚举值。其实说来，是用来清空输入框的内容的一个我们想要的按钮。也就是按一下按钮，输入框的东西就会被清空了。

leftView

添加一个视图，用来布局在输入框的左边。通常用来添加一些提示图标，如用户名输入框左边就有一张小图片来提示用户这个输入框是要输入什么的。

leftViewMode

leftView要与leftViewMode配套使用才能生效。

rightView

在输入框的右边，添加一个视图，用来提示用户。与leftView一样。

rightViewMode

rightView与rightViewMode一样使用才能生效。

```
    // 外观控制

//    textField1.borderStyle = UITextBorderStyleLine;
//    textField1.borderStyle = UITextBorderStyleBezel;
    textField1.borderStyle = UITextBorderStyleRoundedRect;//
圆角矩形

    // 清除按钮
    textField1.clearButtonMode = UITextFieldViewModeAlways;//
快速清空文本框
//    textField1.clearButtonMode =
UITextFieldViewModeWhileEditing;
//    textField1.clearButtonMode =
UITextFieldViewModeUnlessEditing;

    // 左视图
    UIView *leftView1 = [[UIView
alloc]initWithFrame:CGRectMake(0, 0, 30, 40)];
    leftView1.backgroundColor = [UIColor blueColor];
    textField1.leftView = leftView1;
    textField1.leftViewMode = UITextFieldViewModeAlways;

    textField1.rightView = leftView1;
    textField1.rightViewMode = UITextFieldViewModeAlways;
```

11、UIButton

按钮UIButton在app的出镜率也相当高。作用是响应用户点击的控件。
创建UIButton，遵循一下几个步骤：
1、一般用便利构造器来创建一个button对象。
2、设置按钮的属性
3、为按钮添加点击事件
4、把按钮添加到父视图，得以显示

5、无需释放。
创建UIButton对象：

```
        // 便利构造器创建一个button
    UIButton *button1 = [UIButton
buttonWithType:UIButtonTypeSystem];
    button1.frame = CGRectMake(100, 100, 100, 50);
    button1.backgroundColor = [UIColor clearColor];

    // 设置button文字
    [button1 setTitle:@"点我撒"
forState:UIControlStateNormal];
    [button1 setTitle:@"点我撒"
forState:UIControlStateHighlighted];
    [button1 setTitle:@"点我撒"
forState:UIControlStateSelected];

    // 设置图片
    // 必须使用镂空图片
    [button1 setImage:[UIImage imageNamed:@"user.png"]
forState:UIControlStateNormal];
    //设置背景图片（不用镂空的）
    // 用来代替按钮
    [button1 setBackgroundImage:[UIImage imageNamed:@"1.png"]
forState:(UIControlStateNormal)];

    // 添加点击事件
    [button1 addTarget:self action:@selector(buttonAction:)
forControlEvents:UIControlEventTouchUpInside];

    [self.window addSubview:button1];
```

添加事件

addTarget:action:forControlEvents:

第一个参数：指定谁来执行这个事件。
第二个参数：指定谁来实现事件的方法，用@selector来寻找该类下的方法。
第三个参数：

removeTarget:action:forControlEvents：

移除按钮的点击事件。
第一个参数：指定谁来执行这个事件。
第二个参数：指定谁来实现事件的方法，用@selector来寻找该类下的方法。
第三个参数：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.window.backgroundColor = [UIColor colorWithRed:
197.0/255 green:200.0/255 blue:114.0/255 alpha:1];
    [self.window makeKeyAndVisible];

    // 便利构造器创建一个button
    UIButton *button1 = [UIButton
buttonWithType:UIButtonTypeSystem];
    button1.frame = CGRectMake(100, 100, 100, 50);
    button1.backgroundColor = [UIColor clearColor];
// 添加点击事件
    [button1  addTarget:self  action:@selector(buttonAction:)
forControlEvents:UIControlEventTouchUpInside];
    return YES;
    }


// 点击button响应方法
// 点击button的时候，会把button1当作参数传进来给sender
- (void)buttonAction:(UIButton *)sender{
   /*
    self.window.backgroundColor = [UIColor colorWithRed:
arc4random() % 256 / 255.0 green: arc4random() % 256 / 255.0
blue: arc4random() % 256 / 255.0 alpha:1];
//    NSLog(@"疼...");
    // 获取点击事件文字
    NSString *str = [sender
titleForState:UIControlStateNormal];

    NSLog(@"%@",str);

    // 移除点击事件
//    [sender removeTarget:self
action:@selector(buttonAction:)
forControlEvents:UIControlEventTouchUpInside];
    */
    NSLog(@"%ld",sender.tag);

    }
```

解释：

1、self指的是指定本类来实现这个点击事件

2、buttonAction方法来实现点击事件想要做的事。

外观控制

setTitle:forState:

设定指定状态下的button的标题，就是button里显示什么字。
[loginButton setTitle:@“登录”forState:UIControlStateNormal];

titleForState

获取指定状态下的button的标题，也就是button里的字。
NSString *normalTitle = [loginButton titleForState:UIControlStateNormal];

setTitleColor:forState:

设定某个状态下的button的标题颜色，就是button里的字的颜色。
[loginButton setTitleColor:[UIColor redColor] forState:UIControlStateNormal];

titleColorForState:

获取某个状态下的button的标题的颜色，就是button里的字的颜色。
UIColor *normalTitleColor = [loginButton titleColorForState:UIControlStateNormal];

setTitleShadowColor:forState:

设定指定状态下的button的标题的阴影颜色。
[loginButton setTitleShadowColor:[UIColor redColor] forState:UIControlStateNormal];

titleShadowColorForState:

获取指定状态下的标题阴影颜色
UIColor *normalTitleShadowColor = [loginButton titleColorForState:UIControlStateNormal];

setImage:forState:

用图片来代替按钮，这样代替的图片，必须是镂空的图片才有效果。
[loginButton setImage:[UIImage imageNamed:@“login.png”] forState:UIControlStateNormal];

imageForState

获取这个代替这个按钮的图片。
UIImage *normalImage = [loginButton imageForState:UIControlStateNormal];

setBackgroundImage:forState:

用一张图片设置成某个按钮的背景，这个图片就不同是镂空的了。通常我们代替某个按钮，都是按照这个做法来做的。
[loginButton setBackgroundImage: [UIImage imageNamed:@“login2.png”] forState:UIControlStateNormal];

backgroundImageForState:

获取某个button的背景图

12、delegate

用弹出键盘和收回键盘举例子
AppDelegate.h声明协议

```
#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder
<UIApplicationDelegate,UITextFieldDelegate>

@property (retain, nonatomic) UIWindow *window;
@end
```

AppDelegate.m

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]] autorelease];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    UITextField *t1 = [[UITextField
alloc]initWithFrame:CGRectMake(50, 50, 200, 40)];
    t1.borderStyle = UITextBorderStyleRoundedRect;
    t1.backgroundColor = [UIColor yellowColor];
    [self.window addSubview:t1];

    // 设置代理，才能执行代理
    t1.delegate = self;

    [t1 release];
    t1 = nil;

    return YES;
}
// 点击键盘上的return键
- (BOOL)textFieldShouldReturn:(UITextField *)textField{

    // 收键盘
    // 解除第一响应者，键盘回收
    [textField resignFirstResponder];

    NSLog(@"return ");

    return YES;
}
```

`t1.delegate = self;`这一步很重要，雇主要指定代理让谁来做。
`[textField resignFirstResponder];`解除第一响应者，把键盘回收。

基本上一个View，都是通过代理来完成的。

13、iOS的启动流程

ios程序的入口，还是从main.m开始。
main.m

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

NSStringFromClass([AppDelegate　class])是指定去AppDelegate这个类中执行该实现的功能。

UIApplicationMain（）方法主要有个功能：

1、创建应用程序的UIApplication对象；

2、创建引用程序代理实例；

3、建立事件循环（死循环），不断检测程序的运行状态，是否触摸，晃动。

iOS执行过程：

启动程序/前台（活跃状态）/将要结束（活跃状态）/进入后台（不活跃状态）/将要进入前台/前台（活跃状态）。

这就是一个死循环，ios没有提供退出程序的机制，只有强制结束程序。

以下是各个状态下执行的对应方法。

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    NSLog(@"---------完成启动..");

    return YES;
}
// 将要解除活跃状态
- (void)applicationWillResignActive:(UIApplication
*)application {
    NSLog(@"---------将要解除活跃状态..");
}
// 进入后台
- (void)applicationDidEnterBackground:(UIApplication
*)application {
    NSLog(@"--------进入后台..");
}

- (void)applicationWillEnterForeground:(UIApplication
*)application {
    NSLog(@"----------将要进入前台..");
}

- (void)applicationDidBecomeActive:(UIApplication
*)application {
    NSLog(@"--------已经变为活跃状态..");
}
// 将要结束
- (void)applicationWillTerminate:(UIApplication *)application
{
    NSLog(@"-------将要结束..");
}
```

——大海

2015年04月29日23:07:07 北京·蓝欧

14、自定义视图

自定义视图的出现，是为了把众多的view放在一个view里，创建的时候，只要创建这个view就好。这样能用尽量少的代码，进行更多更快的布局。

自定义视图，遵循以下几个步骤：

1、创建一个类，继承于UIView，重新定义初始化方法，

- (instancetype)initWithFrame:(CGRect)frame

这个方法里，[self setupView]调用一个我们自定义的私有方法，来实现几个UILabel+UITextField等组合的布局。

2、在appdelegate中创建这个用于布局的类。

LTView.h

声明两个属性分别为UILabel和UITextField类型的属性

```
#import <UIKit/UIKit.h>

@interface LTView3 : UIView

@property(nonatomic,retain)UILabel *aLabel;
@property(nonatomic,retain)UITextField *aTextField;

@end
```

LTView.m

重写初始化方法，在初始化方法内，对两个属性进行初始化，完成内部的布局

```
#import "LTView.h"

@implementation LTView

- (instancetype)initWithFrame:(CGRect)frame{
    self = [super initWithFrame:frame];
    if (self) {

        [self p_setupView];

    }
    return self;
}
//  私有方法
// 把初始化方法中的视图布局搬到私有方法中
- (void)p_setupView{
    self.aLabel = [[[UILabel
alloc]initWithFrame:CGRectMake(0, 0, 60,
CGRectGetHeight(self.frame))] autorelease];
    self.aLabel.backgroundColor = [UIColor cyanColor];
    [self addSubview:_aLabel];

    self.aTextField = [[[UITextField
alloc]initWithFrame:CGRectMake(CGRectGetMaxX(self.aLabel.fram
e)+20,
CGRectGetMinY(self.aLabel.frame),CGRectGetWidth(self.frame)-
CGRectGetWidth(self.aLabel.frame)-20 ,
CGRectGetHeight(self.aLabel.frame))] autorelease];
    self.aTextField.backgroundColor = [UIColor yellowColor];
    [self addSubview:_aTextField];
}
- (void)dealloc{
    [self.aTextField release];
    [self.aLabel release];
    [super dealloc];
}
@end
```

AppDelegate.m

这样创建一个UILabel和一个UITextField这两个视图的布局，只需要创建一个LTView类的对象即可。

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    // 创建LTView
    LTView *ltv1 = [[LTView
alloc]initWithFrame:CGRectMake(50, 50, 300, 50)];
    ltv1.backgroundColor = [UIColor blueColor];
    [self.window addSubview:ltv1];
    // 释放
    [ltv1 release];
    ltv1 = nil;
    return YES;
}
- (void)dealloc
{
    [self.window release];
    [super dealloc];
}
```

15、视图控制器Controller

MVC

mvc的出现，是为了将视图（view）与数据（model）两者之间进行解耦合（就是尽量让代码之间相互关联尽量减少）。M层位model，属于数据类，V层为界面，直接与用户交互；C层为控制器，用于对model和view之间的交互进行控制。

mvc是一种设计模式，目前在市场上的软件开发中，广泛使用mvc模式，对软件进行分层，便于维护（就是哪里有问题就去哪个模块找原因），便于逻辑控制。

视图控制器的功能：

1、接管分担APPdelegate的工作；

2、显示视图

3、响应事件（包括button点击事件、各种输入框，屏幕事件等等）

4、控制model层与视图层的交互

使用controller的几个步骤：

1、创建***ViewController类

2、在Appdelegate类中，创建***ViewController的对象

3、设置window的根视图控制器（这个根视图控制器就把window全部覆盖了。也就是我们不再直接把以后的子视图放在window上了，而是放在这个视图控制

器自动生成的一个view上，这个view把window覆盖）。
4、创建一个我们自定义的类MyView，继承于UIView，重写初始化方法，添加子视图布局。
5、***Viewcontroller类中声明MyView的属性（在延展里声明）
6、dealloc调MyView的属性
7、重写loadview方法，在这个方法中，创建Myview对象，并替换掉控制器自动生成的view。
ViewController.h
创建视图控制器，继承于UIViewController

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@end
```

ViewController.m
在延展中声明MyView的属性。把视图控制器生成view替换成MyView的属性的属性。

```
#import "ViewController.h"
#import "MyView.h"

// 延展
@interface ViewController ()
// 要替换的view
@property(nonatomic ,retain)MyView *mV;

@end

@implementation ViewController

// 加载视图
- (void)loadView{
    // 创建要替换的视图
    self.mV = [[[MyView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]] autorelease];
    // 替换视图
    self.view = _mV;

}
// 加载完视图
- (void)viewDidLoad {
    [super viewDidLoad];

}
```

MyView.h
创建MYView类，继承于UIView，这个类用来定义我们自己的视图。不再受试图控制器生成的view的约束

```objc
#import <UIKit/UIKit.h>

@interface MyView : UIView

@end
```

MyView.m

```objc
#import "MyView.h"

@implementation MyView

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}

- (void)p_setupView{
    // 设置背景色
    self.backgroundColor = [UIColor redColor];

    UILabel *label = [[UILabel
alloc]initWithFrame:CGRectMake(100, 100, 100, 100)];
    label.backgroundColor = [UIColor yellowColor];

    [self addSubview:label];
    [label release];
    label = nil;

}
@end
```

AppDelegate.h

代理类

```objc
#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (retain, nonatomic) UIWindow *window;
@end
```

AppDelegate.m

设置根视图控制器，创建ViewController类的对象，用controller的view代替window。

```
#import "AppDelegate.h"
#import "ViewController.h"
@interface AppDelegate ()

@end

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.window.backgroundColor = [UIColor whiteColor];
    // 创建ViewController
    ViewController *rootVC = [[ViewController alloc]init];
    self.window.rootViewController = rootVC;
    // 设置根视图控制器
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)dealloc
{
    [self.window release];
    [super dealloc];
}
```

16、屏幕旋转

视图控制器，本身可以检测到屏幕是否有在旋转，如果有屏幕旋转这个事件，处理这个旋转，需要重写一下几个方法即可。

- (NSUInteger)supportedInterfaceOrientations

这个方法是用来设置设备支持旋转的方向（这里说的旋转方式不是手机的旋转方向，而是屏幕里屏幕的旋转方向，因为屏幕内容与手机旋转方向是相反的）

```
// 设备支持旋转方向
- (NSUInteger)supportedInterfaceOrientations{
    return UIInterfaceOrientationMaskAll;
}
```

return回去的，是屏幕内容的旋转方向。

willRotateToInterfaceOrientation:duration:

将要旋转前要作的事，目前苹果已经弃用

didRotateFromInterfaceOrientation:

完成旋转后要做的事，目前苹果已经弃用。
当检测到旋转后，应重启另一套布局，用来适应屏幕的宽高等。重要标志是bounds的值是否改变。如果改变，则会调用一个方法layoutSubviews。
在这个方法里，可以对布局进行修改
RootViewController.m
视图控制器，检测到屏幕是否有旋转，如果旋转，则进行对应的处理。
这里的旋转可以旋转几个方向，可以设置某些方向的旋转无效。return响应的方向即可。

```
#import "RootViewController.h"
#import "MyView.h"
@interface RootViewController ()

@property(nonatomic,retain)MyView *mv;

@end

@implementation RootViewController

- (void)willRotateToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation duration:
(NSTimeInterval)duration{
    NSLog(@"将要旋转..");
}

- (void)didRotateFromInterfaceOrientation:
(UIInterfaceOrientation)fromInterfaceOrientation{
    NSLog(@"旋转完成..");
}

// 设备支持旋转方向
- (NSUInteger)supportedInterfaceOrientations{
    return UIInterfaceOrientationMaskAll;
}
- (void)loadView{
    self.mv = [[[MyView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]] autorelease];
    self.view = _mv;
}
- (void)dealloc
{
    [_mv release];
    [super dealloc];
}
- (void)viewDidLoad {
    [super viewDidLoad];
}

// 接收到内存警告
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // 根视图已经加载过并且根视图没有显示在屏幕上
    if([self isViewLoaded]== YES && self.view.window == nil){
        // 将根视图销毁
        self.view = nil;
    }
}
@end
```

MyView.m

```
#import "MyView.h"

@interface MyView ()

@property(nonatomic,retain)UIView *v;

@end
@implementation MyView
- (void)dealloc
{
    [_v release];
    [super dealloc];
}

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}
- (void)p_setupView{

    self.backgroundColor = [UIColor greenColor];
    self.v = [[UIView alloc]initWithFrame:CGRectMake(50, 50,
230, 300)];
    self.v.backgroundColor = [UIColor grayColor];
    [self addSubview:_v];

}
// 当bounds发生改变的时候，自动调用此方法。
-(void)layoutSubviews{
    NSLog(@"bounds 发生改变");
    // 获取当前设备方向
    if (([UIApplication
sharedApplication].statusBarOrientation ==
UIInterfaceOrientationPortraitUpsideDown) || ([UIApplication
sharedApplication].statusBarOrientation ==
UIInterfaceOrientationPortrait)) {
        self.v.frame = CGRectMake(50, 50, 230, 300);
    }else{
        self.v.frame = CGRectMake(100, 50, 300, 230);
    }
}
@end
```

旋转后对布局的选择，适应屏幕

不努力，对不起的是自己。

——大海
2015年04月30日22:11:01 北京·蓝欧

17、事件处理

在操作手机时，经常会有各种操作，在iOS中，就会用事件来处理这些操作（应该叫响应）。
UIEvent，事件，就是硬件捕捉一个用户的操作设备的对象。
iOS事件分为三类：触摸事件，晃动事件，远程控制事件
触摸事件：用户通过触摸屏幕触摸设备操作对象，输入数据。
实现触摸，就是iOS的UIView继承于UIResponder，支持多点触摸。所以，只要继承于UIview的所有视图，都可以响应触摸方法。

触摸处理，其实就是实现几个方法即可。

开始触摸：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
```

在开始触摸屏幕时，就会执行这个方法，因此，想要在触摸屏幕时，实现一些功能，或者提示，都可以在这个方法里添加。
NSSet集合里放着对象，就是UITouch对象。对象里包含这个触摸事件的情况，包括所在的window的frame，所在的视图的frame，所在window的位置，所在视图的位置等等。
event包含了这次触摸事件的事件。包括触摸事件的类型（开始触摸？触摸移动？）以及触摸所在的哪个window，哪个view和所在window和view的位置。

触摸移动：

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
```

在屏幕上滑动时，就是会执行这个方法。
touches和event与开始触摸方法一样的意思。

触摸结束：

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
```

触摸事件结束所掉方法。

取消触摸：

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

取消触摸所调方法。

一个例子：
让一个视图，随着触摸移动而移动：
NewView.m
这个视图是我们自己定义的视图。也就是视图控制器控制的子视图。由于AppDelegate和ViewController与前面的写法一样，就不写出来了。

```
//
//  NewView.m
//  UILesson4_Touch
//
//  Created by lanou3g on 15/5/1.
//  Copyright (c) 2015年 H.Z. All rights reserved.
//

#import "NewView.h"
#import "MyView.h"

@interface NewView ()
// 保存起始点
@property(nonatomic,assign)CGPoint point;
// 保存开始frame的点
@property(nonatomic,assign)CGPoint framepoint;

@end

@implementation NewView

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
    }
    return self;
}
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent
*)event{

    UITouch *touch =  [touches anyObject];
    // 保存手指第一次触碰位置
    self.point = [touch locationInView:self.superview];
    // frame的第一次位置
    self.framepoint = self.frame.origin;
    NSLog(@"%@",NSStringFromCGPoint(self.point));
}
-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent
*)event{
    UITouch *touch =  [touches anyObject];
    CGPoint movepoint = [touch
locationInView:self.superview];
    CGFloat detailX = movepoint.x - self.point.x;
    CGFloat detailY = movepoint.y - self.point.y;

    self.frame = CGRectMake(self.framepoint.x+detailX,
self.framepoint.y+detailY, CGRectGetWidth(self.frame),
CGRectGetHeight(self.frame));
    self.backgroundColor = [UIColor colorWithRed:arc4random()
%256/255.0 green:arc4random()%256/255.0 blue:arc4random()
%256/255.0 alpha:1];
    NSLog(@"%@",NSStringFromCGPoint(self.point));
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent
*)event{
}
```

18、响应者链

响应者： UIResponder类是响应者类，所有继承UIResponder 的类，都是响应者。iOS中，所有能响应事件的对象，都是响应者。
响应者链：就是多个响应者组成的链。

检测触碰视图：

当接手一个事件时，
过程如下：
查询UIApplication是否有响应者，若有，往window找有没有子响应者/找window/找window上的view（viewcontroller）/找view的子视图。最终确认触摸的位置，完成查询。

响应触碰事件

与检测正好相反，优先考虑在子视图中响应事件。也就是说，如果子视图不处理事件，就会往查询的反方向传递事件，让后者处理。一直到最底的UIApplication，如果都没有响应这个事件，就会把事件抛弃。
响应过程如下：
触摸的view/Rootview/controller/window/delegate/UIApplication。

阻断响应者链

响应者链可以被打断，阻断自己的子视图不再检索。

```
userInteractionEnabled = NO;
```

默认为YES，当设置为YES时，事件可以正常传递给子视图。如果设置为NO时，就是告诉系统，我不再在响应者链中，此时，系统就会将其移出响应者链，则会导致其子视图的所有响应中断。不再对子视图进行检索。

ps：响应者链是能够响应事件的对象组成的链，事件在该链上传递，最终结果事件或被处理或被抛弃。所以一个对象在不在响应者链里是其能否响应事件的首要前提条件！
测试：分别在
AppDlegate.m/window.m/ViewController.m/rootView.m/TouchView.m中用

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
```

打印一句话测试即可。

—大海
2015年05月03日22:06:45 北京·蓝欧

19、target/action设计模式

在写程序的时候，用了MVC将程序分层。那么就得在写程序的时候把对应的代码写在对应的层里。
view层，只需要布局，仅仅需要布局。对于逻辑性的代码（比如按钮的点击事件响应等），都要写在controller里，view层中不允许出现。
target/action模式，就是用来解决这个问题的。也就是所谓的解耦合。程序里就应该做到“高聚合，低耦合”。
高聚合的意思，就是将属于自己的东西，都写在自己的层次里，尽量少的写在其他层次中。
低耦合的意思，是不同的层次之间，尽量少的关联，但不能不关联。
以下是target/action实现解耦合的流程：（只写关键代码）
AppDelegate.m

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[[UIWindow alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.window.backgroundColor = [UIColor whiteColor];
    RootViewController *root = [[RootViewController
alloc]init];
    self.window.rootViewController = root;

    [self.window makeKeyAndVisible];
    [root release];
    root = nil;
    return YES;
}
```

代理，将根视图设置为RootViewController的对象，不再使用自己的window。

RootViewController.m

```
#import "RootViewController.h"
#import "RootView.h"
#import "ButtonView.h"
#import "ColorView.h"
@interface RootViewController ()
@property(nonatomic,retain)RootView *rv;
@end

@implementation RootViewController

-(void)loadView{
    self.rv = [[[RootView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.view = _rv;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    // 告诉buttonView target 是谁，action是谁。
    [self.rv.bv myAddTarget:self
action:@selector(bvAction:)];
}

- (void)bvAction:(ButtonView *)sender{
    NSLog(@"123456");
    NSLog(@"%@",sender);
}
```

1、@property(nonatomic,retain)RootView *rv;将试图设置为试图控制器所对应的视图（也就是将RootView代替controller自己本身的视图self.view = _rv;）。

2、[self.rv.bv myAddTarget:self action:@selector(bvAction:)];

（1）rv为controller被替代的根视图

（2）bv是我们自己定义的一个视图（ButtonView）

（3）myAddTarget:self action:@selector(bvAction:)是我们在ButtonView中定义的一个方法

（4）参数self是controller本身，将controller本身传过去

（5）参数@selector(bvAction:)是用方法选择器将bvAvtion方法传过去。

RootView.h

```
#import <UIKit/UIKit.h>
@class ButtonView;
@class ColorView;
@interface RootView : UIView

// bv 属性将要在controller中使用
@property(nonatomic,retain)ButtonView *bv;
@end
```

把视图ButtonView作为RootView的一个属性。

RootView.m

```
#import "RootView.h"
#import "ButtonView.h"
@implementation RootView
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}

- (void)p_setupView{
    self.backgroundColor = [UIColor yellowColor];

    self.bv = [[[ButtonView
alloc]initWithFrame:CGRectMake(100, 100, 100,
100)]autorelease];
    self.bv.backgroundColor = [UIColor blueColor];
    [self addSubview:_bv];
}
@end
```

根视图，就是把那些视图都加在这个rootView上。

ButtonView.h

```
#import <UIKit/UIKit.h>
@interface ButtonView : UIView{
    // 目标实例变量
    id _target;
    // 事件实例变量
    // 方法选择器类型
    SEL _action;
}
// 添加目标和事件
- (void)myAddTarget:(id)target action:(SEL)action;
@end
```

1、我们自己定义的视图，定义了两个实例变量，target是id类型的，也就是能代表所有的类型，用来接收一个目标，这个目标是为了执行action中的方法的。action是用来接收传进来的我自己要实现的方法（这里是点击事件bvButtonAction）。

2、- (void)myAddTarget:(id)target action:(SEL)action;
声明一个方法用来把controller中的要执行action方法的对象，执行的方法action传进来。

ButtonView.m

```
#import "ButtonView.h"

@implementation ButtonView
// 添加目标和事件
- (void)myAddTarget:(id)target action:(SEL)action{
    _target = target;
    _action = action;

}
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent
*)event{
    NSLog(@"开始点击..");
    // 让target执行action方法
    [_target performSelector:_action withObject:self];
}
@end
```

在我们真正 要布局的地方重写初始化方法，把对应的视图建出来。
注意，我们想实现的点击事件，不在这里，这里没有点击事件，因为点击事件是逻辑性的东西，就把它写在了控制器ViewController里。
具体流程如下：
1、从controller开始，重写loadView方法，把rootView初始化，并代替自己的视图（self.view），然后去到rootView中，在初始化rootView的时候，把ButtonView也初始化了，创建了一个ButtonView视图bv。
2、当把视图都创建完毕后，完成加载，会回到controller中，调用viewDidload方法。此时我们就要重写这个方法，在viewDidload方法里，我们写上
[self.rv.bv myAddTarget:self action:@selector(bvAction:)];
这个调用方法，也就是把Buttonview中声明的myAddTarget: action:方法放到这里来调用。就会把控制器自己和在控制器里定义的ButtonView的点击响应事件方法传回去给ButtonView。
3、在ButtonView中，就会完成两个实力变量_target 和_action赋值。
4、此时，viewDidload方法执行完毕，真正完成视图的加载。
5、点击视图时，触发touchesBegan方法，在这个方法里，
[_target performSelector:_action withObject:self];
这个才是我们真正让_target和 _action完成功能的语句。这个意思就是让_target
（这就是之前传进来了controler ）去执行_action （也就是之前传进来的bvAction方法）self就是ButtonView自己，设置这个withObject的参数，就是为了controller中定义bvAction方法的时候，参数sender得到ButtonView对象。

20、delegate代理设计模式

代理设计模式，功能跟addTarget/action比较像但是也有一些不同：

同上小节代码，加载完毕各种视图后。
去到视图：
TouchView.h

```
#import <UIKit/UIKit.h>
@class TouchView;// 这里引入，是为了协议里方法的参数类型。

// 协议
@protocol TouchViewDelegate

- (void)changeColor:(TouchView *)aView;

@end

@interface TouchView : UIView

@property(nonatomic,assign)id<TouchViewDelegate>delegate;

@end
```

在这里，touchView相当于雇主，写一个协议protocol，在协议里声明了一个方法- (void)changeColor:(TouchView *)aView;这个方法就是让接协议的类去实现。
声明一个id类型属性delegate，遵循我的协议<TouchViewDelegate>，这里把这个属性成为代理。
TouchView.m

```
#import "TouchView.h"

@implementation TouchView

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent
*)event{
    NSLog(@"开始点击》》");
    [_delegate changeColor:self];
}

@end
```

点击视图，触发点击事件，让代理去调上面声明的方法changeColor，参数传self（view就是TouchView本身）。此时，就会到controller（遵循协议的类）中去寻找changeColor实现方法。如下

RootViewController.m

```
#import "RootViewController.h"
#import "RootView.h"
#import "TouchView.h"
// 在延展遵循协议，外部看不到
@interface RootViewController ()<TouchViewDelegate>

@property(nonatomic,retain)RootView *rv;
@end

@implementation RootViewController
// 实现代理方法
- (void)changeColor:(TouchView *)aView{
    aView.backgroundColor = [UIColor
colorWithRed:arc4random() % 256/255.0 green:arc4random() %
256/255.0 blue:arc4random() % 256/255.0 alpha:1];
}
-(void)loadView{
    self.rv = [[[RootView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.view = _rv;
}
- (void)viewDidLoad {
    [super viewDidLoad];
    // 设置代理。
    self.rv.tv.delegate = self;
}
```

因为点击事件是逻辑性的东西，应该写在控制器中，因此用RootViewController遵循touchView声明的协议，实现协议中声明的
`- (void)changeColor:(TouchView *)aView`方法，完成一次代理过程。
但是，仅仅这样还不能让让代理模式实现，因为就算`controller`遵循了协议，并没有告诉雇主，`controller`是遵循协议的类，也就是雇主还没有承认`controller`就是代理人。因此在加载完毕后调`viewDidload`方法的时候，就应该把自己告诉给雇主，我遵循了你的协议。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 设置代理。
    self.rv.tv.delegate = self;
}
```

代理模式到此结束。

21、手势识别器

UIImageView

这个类是iOS专门用来显示图片的类，几乎所有的图片，都是用这个类显示的。
初始化方法：initWithImage:

```
    UIImageView *img = [[UIImageView alloc]initWithImage:
[UIImage imageNamed:@"/Users/lanou3g/Desktop/UICode/
UILesson5_GestureRecognizer/UILesson5_GestureRecognizer/
2.png"]];
    img.frame = CGRectMake(100, 100, 200, 200);
```

通过图片名来初始化一个UIImageView类。

注意:
ImageView的默认是NO，是关闭交互。要将
userInteractionEnabled（响应者链是否交互）设置为YES
```
self.img.userInteractionEnabled = YES;
```

手势

手势是加在某个视图上的，否则手势无效。

轻拍手势：

```
    // 轻拍手势
    // 创建轻拍手势识别器
   UITapGestureRecognizer  *tap  =  [[UITapGestureRecognizer
alloc]initWithTarget:self action:@selector(tapAction:)];


// 轻拍手势响应方法
- (void)tapAction:(UITapGestureRecognizer *)sender{
    NSLog(@"我是轻拍手势..");
    // 换图片
   // 获取手势图片
    UIImageView *temp = (UIImageView *)sender.view;
    temp.image = [UIImage imageNamed:@"3.png"];
   }
```

长按手势：

```
    // 长按手势
    UILongPressGestureRecognizer *longPress =
[[UILongPressGestureRecognizer alloc]initWithTarget:self
action:@selector(longPressAction:)];
    longPress.minimumPressDuration = 0.5;
    [self.img addGestureRecognizer:longPress];


// 长按响应方法
- (void)longPressAction:(UILongPressGestureRecognizer
*)sender{
    NSLog(@"长按手势");

    if (sender.state == UIGestureRecognizerStateBegan) {
        NSLog(@"长按开始..");
        [UIView animateWithDuration:1 animations:^{
            // 修改view的frame
            CGRect temp1 = sender.view.frame;
            temp1.size.width +=50;
            temp1.size.height +=50;
            // 修改完成附回
            sender.view.frame = temp1;

            sender.view.alpha = 0.5;
        }];
    }else if (sender.state == UIGestureRecognizerStateEnded){
        NSLog(@"长按结束..");
        [UIView animateWithDuration:1 animations:^{
            // 修改view的frame
            CGRect temp1 = sender.view.frame;
            temp1.size.width -=50;
            temp1.size.height -=50;
            // 修改完成附回
            sender.view.frame = temp1;

            sender.view.alpha = 1;
        }];
    }
}
```

旋转手势：

```
    // 旋转
    UIRotationGestureRecognizer *rotation =
[[UIRotationGestureRecognizer alloc]initWithTarget:self
action:@selector(rotationAction:)];
    [self.img addGestureRecognizer:rotation];
// 旋转手势响应方法
- (void)rotationAction:(UIRotationGestureRecognizer *)sender{
    NSLog(@"我是旋转手势..");
    sender.view.transform =
CGAffineTransformRotate(sender.view.transform,
sender.rotation);
    sender.rotation = 0;
    }
```

捏合手势：

```
    // 捏合
    UIPinchGestureRecognizer *pinch =
[[UIPinchGestureRecognizer alloc]initWithTarget:self
action:@selector(pinchAction:)];
    [self.img addGestureRecognizer:pinch];

// 捏合手势响应方法
- (void)pinchAction:(UIPinchGestureRecognizer *)sender{
    NSLog(@"我是捏合手势");
    sender.view.transform =
CGAffineTransformScale(sender.view.transform, sender.scale,
sender.scale);
    sender.scale = 1;
   }
```

平移手势：

```
    // 平移
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer
alloc]initWithTarget:self action:@selector(panAction:)];
    [self.img addGestureRecognizer:pan];

// 平移手势响应方法
- (void)panAction:(UIPanGestureRecognizer *)sender{
    NSLog(@"我是平移手势..");
    CGPoint point = [sender translationInView:sender.view];
    sender.view.transform =
CGAffineTransformTranslate(sender.view.transform, point.x,
point.y);
    // 将偏移量归零
    [sender setTranslation:CGPointZero inView:sender.view];
    }
```

轻扫手势：

边缘触发：

——大海
2015年05月04日22:09:46 北京·蓝欧

22、UISegmentedControl分段控件

分段控件提供了一栏按钮，但是每次只能激活一个按钮，每一个按钮对应不同的屏幕显示的东西（这里的不同，应该理解为数据的不同，view是相同的，如

筛选出不同的信息，但是view是一样的（布局样式是一样的））。

RootView.m

```
// 创建segmentcontrol

    // 创建数组
    NSMutableArray *itemsArr = [NSMutableArray array];
    [itemsArr addObject:@"first"];
    [itemsArr addObject:@"second"];
    [itemsArr addObject:@"thied"];
    self.seg = [[[UISegmentedControl
alloc]initWithItems:itemsArr]autorelease];
    // 设置frame
    self.seg.frame = CGRectMake(50, 50, 200, 50);
//    self.seg.backgroundColor = [UIColor greenColor];

    self.seg.tintColor = [UIColor greenColor];

    // 默认选中
    self.seg.selectedSegmentIndex = 0;

    [self addSubview:_seg];
```

1、self.seg = [[UISegmentedControl alloc]initWithItems:itemsArr];为初始化方法，是UISegmentedControl特有的初始化方法。initWithItems：的参数是一个数组。
2、数组中又几个元素，这个segment就会分成几栏button。
3、self.seg.tintColor，给分栏设置颜色
4、self.seg.selectedSegmentIndex：为默认选中哪一个栏，参数是数组的下标。

RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIButton *btn = [UIButton
buttonWithType:UIButtonTypeSystem];
    btn.frame = CGRectMake(50, 150, 50, 50);
    btn.backgroundColor = [UIColor blueColor];
    [btn addTarget:self action:@selector(btnAction:)
forControlEvents:UIControlEventTouchUpInside];

    [self.view addSubview:btn];

    // 添加segment点击事件
    [self.rv.seg addTarget:self action:@selector(segAction:)
forControlEvents:UIControlEventValueChanged];

}

- (void)btnAction:(UIButton *)sender{
    NSLog(@"btn");
    // 设置segment分段上的文字
    [self.rv.seg setTitle:@"第一项" forSegmentAtIndex:0];
    [self.rv.seg setTitle:@"第二项" forSegmentAtIndex:1];
    [self.rv.seg setTitle:@"第三项" forSegmentAtIndex:2];
    [self.rv.seg insertSegmentWithTitle:@"第四项" atIndex:3
animated:YES];



}
// segment 响应方法
- (void)segAction:(UISegmentedControl *)sender{
    NSLog(@"%ld",sender.selectedSegmentIndex);

    if (sender.selectedSegmentIndex == 0) {
        [sender setTintColor:[UIColor redColor]];
    }else if(sender.selectedSegmentIndex == 1){
        [sender setTintColor:[UIColor blueColor]];
    }else{
        [sender setTintColor:[UIColor whiteColor]];
    }
}
```

1、在controller里实现segment的逻辑

2、在这里添加一个button，每点击一次button，就会改变一次segment的标题：setTitle:@"第一项" forSegmentAtIndex:0

说明把第0个分段的标题设置成“第一项”。

3、selectedSegmentAtIndex：0.被选中的分栏的下标为0

4、setTintColor:改变分栏的颜色

5、

addTarget:selfaction:@selector(segAction:)forControlEvents:UIControlEventValueChanged

添加segment点击事件：第一个参数：谁来执行，第二个参数，到谁那里去找segAction方法，然后执行，第三个参数：事件改变的时候才执行。

23、UISlider（进度条）

在iOS中是一个滑块控件，基本用于视频播放进度，控制音量大小，控制播放器的进度条等等。
UISlider继承于UIControl，拖动UISlider时，会提供一系列的值，滑块在不同的位置，获取的值也是不同的。
RootView.m

```
// slider 进度条
    // 创建slider
    self.slider = [[[UISlider
alloc]initWithFrame:CGRectMake(50, 200, 200,
50)]autorelease];
    // 最小值 负数也可以
    self.slider.minimumValue = 0;
    // 最大值
    self.slider.maximumValue = 1;
    // 当前值
    self.slider.value = 1;

    // 划过区域颜色
    self.slider.minimumTrackTintColor = [UIColor greenColor];
    // 未划过区域颜色
    self.slider.maximumTrackTintColor = [UIColor whiteColor];

    // 小圆点变颜色
    self.slider.thumbTintColor = [UIColor redColor];
    // 设置图片后，小圆点才会变颜色
//    [self.slider setThumbImage:[UIImage imageNamed:@"/
Users/lanou3g/Desktop/UICode/UILesson6_Segment/
UILesson6_Segment/2.png"] forState:UIControlStateNormal];
    // 最小值图片
    self.slider.minimumValueImage = [UIImage
imageNamed:@"2.png"];
    // 最大值图片
    self.slider.maximumValueImage = [UIImage
imageNamed:@"2.png"];

    [self addSubview:_slider];
```

1、初始化方法：initWithFrame与普通的视图初始化方法类似
2、minimumValue，设置slider的最小值（一般从0开始，但是也可以是负数）
3、maximumValue，设置slider的最大值
4、minimumTrackTintColor，设置进度条划过后的颜色
5、maximumTrackTintColor，设置进度条未划过的颜色
6、thumbTintColor，进度条上当前位置小圆点的颜色（想要把小圆点的颜色改变，得首先把这个小圆点用图片来代替：调setThumbImage方法）
7、小圆点的颜色和小圆点的图片会相互覆盖
8、minimumValueImage，设置slider最小值的图片
9、maximumValueImage，设置slider最大值的图片
RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 添加slider滑动事件
[self.rv.slider addTarget:self
action:@selector(sliderAction:) forControlEvents:
(UIControlEventValueChanged)];
}
// slider 响应方法
- (void)sliderAction:(UISlider *)sender{
    NSLog(@"slider");
    NSLog(@"%.1f",sender.value);
    // 使用slider的值，改变视图透明度
    self.view.alpha = sender.value;

}
```

添加事件的参数与segment一样的意思
sender.value为slider的当前值。

24、UIImageView

UIImageView相当于一个相框，用来显示图片，里边可以是一张图片，也可以是一组图片。

RootView.m

```
- (void)p_setupView{
    self.backgroundColor = [UIColor yellowColor];

    self.imv = [[UIImageView
alloc]initWithFrame:CGRectMake(100, 100, 200, 200)];
    self.imv.backgroundColor = [UIColor whiteColor];

    // 不要将数据写入view

//    // 准备一个image的数组
//    NSMutableArray *imArr = [NSMutableArray array];
//    // 循环生成image放入数组
//    for(int i = 0; i < 7;i ++){
//        NSString *nameStr = [NSString
stringWithFormat:@"%d.tiff",i];
//        [imArr addObject:[UIImage imageNamed:nameStr]];
//    }
//    // 将image数组 赋值给animationImages
//    self.imv.animationImages = imArr;

    [self addSubview:_imv];
```

RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // 准备一个image的数组
    NSMutableArray *imArr = [NSMutableArray array];
    // 循环生成image放入数组
    for(int i = 0; i < 7;i ++){
        NSString *nameStr = [NSString
stringWithFormat:@"%d.tiff",i];
        [imArr addObject:[UIImage imageNamed:nameStr]];
    }
    // 执行一组所用时间
    self.rv.imv.animationDuration = 1;
    // 循环5此
//    self.rv.imv.animationRepeatCount = 5;
    // 将image数组 赋值给animationImages
    self.rv.imv.animationImages = imArr;
    // 开始动画
    [self.rv.imv startAnimating ];
    // 结束动画,就什么都不显示了。
//    [self.rv.imv stopAnimating];

}
```

1、初始化方法：initWithFrame与普通视图差不多
2、animationImages，设置一组图片（是一个数组）
3、animationDuration，设置执行一组图片用多长时间
4、animationRepeatCount，执行多少次这组图片
5、startAnimating，开始动画
6、stopAnimating，停止动画（停止动画的意思不是让图片停止在某个状态，而是直接不再显示图片）

25、UIControl

UIControl是所有控制类控件的基类，UIButton，UISlider，UISegmentControl等等都继承于该类
核心功能：
添加事件：addTarget: action: forControlEvents:
移除事件：removeTarget: action: forControlEvents:

26、所有类继承关系

1、基类：NSObject
（1）、自定义的类——自定义类的子类...
（2）、NSString——NSMutableString
（3）、NSArray——NSMutableArray
（4）、NSDictionary——NSMutableDictionary

（5）、NSSet——NSMutableSet
（6）、NSDate

（7）、UIGestureRecognizer
a、UILongPressGestureRecognizer
b、UIPanGestureRecognizer
c、UIPinchGestureRecognizer
d、UIRotationGestureRecognizer
e、UISwipeGestureRecognizer
f、UITapGestureRecognizer

（8）、UIResponder
a、UIApplication
b、APPDelegate
c、UIViewController
e、UIView
（a）、UIAlertView
（b）、UILabel
（c）、UIImageView
（d）、UIWindow
（e）、UIControl
UIButton
UITextField
UISlider
UISegmentedControl

——大海
2015年05月05日17:34:06 北京·蓝欧

27、UIScrollView

```
//
//  RootView.m
//  UILesson7_ScrollView
//
//  Created by lanou3g on 15/5/6.
//  Copyright (c) 2015年 H.Z. All rights reserved.
//

#import "RootView.h"

@implementation RootView

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}

-(void)p_setupView{
    self.backgroundColor = [UIColor yellowColor];
    UIImageView *imv = [[UIImageView alloc]initWithImage:
[UIImage imageNamed:@"1.png"]];

    imv.frame = CGRectMake(0, 0, 150, 150);
    // 设置tag值，为了后面缩放的delegate（在controller里）取到imv
    imv.tag = 100;
    [self addSubview:imv];

    self.scrollView = [[UIScrollView
alloc]initWithFrame:CGRectMake(100, 100, 200, 200)];
    self.scrollView.backgroundColor = [UIColor greenColor];

    // scrollView 属性

    // contentsize，能够滑动的决定性因素(能够决定滚动的范围)
    self.scrollView.contentSize = imv.frame.size;

    // contentOffset 偏移
    self.scrollView.contentOffset = CGPointMake(-50, -50);
    // 返回顶部
    self.scrollView.scrollsToTop = YES;

    // pagingEnabled 整页滑动（轮播图，相册）
    self.scrollView.pagingEnabled = NO;

    // 边界是否反弹
    self.scrollView.bounces = YES;
    // 是否能够滚动
//    self.scrollView.scrollEnabled = NO;
    // 是否显示水瓶滚动条
    self.scrollView.showsHorizontalScrollIndicator = NO;
    // 是否显示垂直滚动条
    self.scrollView.showsVerticalScrollIndicator = YES;
```

RootViewController.m

```
#import "RootViewController.h"
#import "RootView.h"
@interface RootViewController ()<UIScrollViewDelegate>
@property(nonatomic,retain)RootView *rv;
@end

@implementation RootViewController

-(void)loadView{
    self.rv = [[[RootView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]]autorelease];
    self.view = _rv;
}
// scrollView 代理方法
// 指定缩放视图
- (UIView *)viewForZoomingInScrollView:(UIScrollView
*)scrollView{
    return [scrollView viewWithTag:100];
}
- (void)viewDidLoad {
    [super viewDidLoad];
    // 设置代理
    self.rv.scrollView.delegate = self;
}
```

代理方法

RootViewController.m

```
#pragma scrollView 的代理方法
-(void)scrollViewDidScroll:(UIScrollView *)scrollView{
    NSLog(@"已经滚动。。");
}

-(void)scrollViewWillBeginDragging:(UIScrollView
*)scrollView{
    NSLog(@"将要开始拖拽..");
}
-(void)scrollViewDidEndDragging:(UIScrollView *)scrollView
willDecelerate:(BOOL)decelerate{
    NSLog(@"已经结束拖拽..");
}
- (void)scrollViewWillBeginDecelerating:(UIScrollView
*)scrollView{
    NSLog(@"将要开始减速..");
}

- (void)scrollViewDidEndDecelerating:(UIScrollView
*)scrollView{
    NSLog(@"结束减速..");
}

// 缩放有关
// 缩放开始
- (UIView *)viewForZoomingInScrollView:(UIScrollView
*)scrollView{

    return [scrollView viewWithTag:100];
}
// 缩放结束
-(void)scrollViewDidEndZooming:(UIScrollView *)scrollView
withView:(UIView *)view atScale:(CGFloat)scale{

}
```

28、UIPageControl

RootView.m

```
#import "RootView.h"

@implementation RootView

- (instancetype)initWithFrame:(CGRect)frame{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}

- (void)p_setupView{
    self.backgroundColor = [UIColor yellowColor];
    // scrollView
    self.scrollView = [[UIScrollView
alloc]initWithFrame:self.bounds];
    self.scrollView.backgroundColor = [UIColor grayColor];
    [self addSubview:_scrollView];


    // pageConrol
    self.pc = [[UIPageControl
alloc]initWithFrame:CGRectMake(CGRectGetMinX(self.frame),
CGRectGetMaxY(self.frame)-50, CGRectGetWidth(self.frame),
50)];
    self.pc.backgroundColor = [UIColor colorWithRed:0 green:0
blue:0 alpha:0];
    // 设置有几页
    self.pc.numberOfPages = 5;
    // d当前页
    self.pc.currentPage = 1;
    [self addSubview:_pc];

}

@end
```

RootViewController.m

```
#import "RootViewController.h"
#import "RootView.h"
@interface RootViewController ()<UIScrollViewDelegate>
@property(nonatomic,strong)RootView *rv;
@property(nonatomic,strong)NSMutableArray *arr;
@end

@implementation RootViewController

-(void)loadView{
    self.rv = [[RootView alloc]initWithFrame:[[UIScreen
mainScreen] bounds]];
    self.view = self.rv;
}

- (void)viewDidLoad {

    // 设置scrollView代理
    self.rv.scrollView.delegate = self;
    [super viewDidLoad];
    // 设置pageControl的响应事件
    [self.rv.pc addTarget:self action:@selector(pageAction:)
forControlEvents:(UIControlEventTouchUpInside)];

    // 处理数据
    self.arr = [NSMutableArray array];
    for (int i = 0 ; i < 7 ; i ++) {
        NSString *str = [NSString
stringWithFormat:@"%d.jpeg",i];
        [self.arr addObject:str];
    }
    // 设置
    self.rv.scrollView.contentSize =
CGSizeMake(CGRectGetWidth(self.rv.frame) * self.arr.count,
CGRectGetHeight(self.rv.frame));
    // 设置不显示水平滑动线
    self.rv.scrollView.showsHorizontalScrollIndicator = NO;
    // 添加图片
    int i = 0;
    for (NSString *name in self.arr) {
        UIImageView *temp = [[UIImageView
alloc]initWithImage:[UIImage imageNamed:name]];
        temp.frame =
CGRectMake(CGRectGetWidth(self.rv.frame)*i,
CGRectGetMinY(self.rv.scrollView.frame),
CGRectGetWidth(self.rv.scrollView. frame),
CGRectGetHeight(self.rv.scrollView.frame));
        [self.rv.scrollView addSubview:temp];
        i ++;
    }

    // 整页滚动
    self.rv.scrollView.pagingEnabled = YES;
    // 设置pageControl的页数
    self.rv.pc.numberOfPages = self.arr.count;
    // 设置pageContol的当前页
    self.rv.pc.currentPage = 0;
```

——大海
2015年05月06日19:51:47 北京 · 蓝欧

29、UINavigationController导航控制器

UINavigationController是iOS中常用的控制器，作用是用来管理多个视图控制器。主要管理有层级的控制器（也就是一层到下一层）。
UINavigationController内部管理视图控制器，主要是因为UINavigationController里是栈结构，把视图控制器入栈和出栈的过程。也就是说，入栈，就是把视图控制器加到UINavigationController中，就可以控制视图在哪一个层次（页面）显示出来。简单来说，就是那么多个页面，你想让哪一个显示出来。
在UINavigationController中，至少有一个controller存在，这个视图就是根视图控制器（RootViewController）。

UINavigationController展示出来的视图，都是栈顶的controller的view。
导航控制器的 view 在 屏幕的上方，44像素高，与屏幕同宽。

navigationController

这是每一个导航控制器都有的属性，也就是说在UINavigationController中，加入多个视图控制器后，每一个视图控制器都可以通过self. navigationController

来获取自己被哪个控制器所有。
AppDelegate.m

```
#import "AppDelegate.h"
#import "RootViewController.h"
@interface AppDelegate ()

@end

@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];

    // 试图控制器
    RootViewController *root = [[RootViewController
alloc]init];

    // 导航控制器
    UINavigationController *rootNC = [[UINavigationController
alloc]initWithRootViewController:root];
    rootNC.navigationBar.barTintColor = [UIColor blackColor];
    // 加图片或者加颜色的时候，（0，0）点就以导航左下角
    // 没有加的话（0，0）点就是在屏幕的左上角
    [rootNC.navigationBar setBackgroundImage:[UIImage
imageNamed:@"2.png"] forBarMetrics:(UIBarMetricsDefault)];

    self.window.rootViewController = rootNC;

    [self.window makeKeyAndVisible];

     [root release];
     root = nil;

    [rootNC release];
    rootNC = nil;

    return YES;
}
```

解释：

1、`UINavigationController *rootNC = [[UINavigationController alloc]initWithRootViewController:root];`
用根视图控制器来初始化导航控制器，将根视图控制器入栈。

2、`rootNC.navigationBar.barTintColor`设置导航控制器本身视图的颜色

3、`[rootNC.navigationBar setBackgroundImage:[UIImage imageNamed:@"2.png"] forBarMetrics:(UIBarMetricsDefault)];`设置导航控制器本身视图的背景图。

4、导航控制器，如果加上颜色或者图片，则（0，0）点就会在导航条左下角；如果本身没有加颜色或者图片，（0，0）点就是在屏幕的左上角。

5、`self.window.rootViewController = rootNC;`把导航控制器给window的根视图控制器。

RootViewController.m

```
#import "RootViewController.h"
#import "FirstViewController.h"
@interface RootViewController ()

@end

@implementation RootViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor yellowColor];

    UIButton *btn = [UIButton buttonWithType:
(UIButtonTypeSystem)];
    btn.frame = CGRectMake(100, 64, 100, 100);
    btn.backgroundColor = [UIColor blueColor];
    [btn addTarget:self action:@selector(btnAction:)
forControlEvents:(UIControlEventTouchUpInside)];
    [self.view addSubview:btn];
    //item
    // title
//    self.navigationItem.title = @"RootView";
    UIView *v = [[UIView alloc]initWithFrame:CGRectMake(0, 0,
200, 44)];
    v.backgroundColor = [UIColor redColor];
    self.navigationItem.titleView = v;
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem
alloc]initWithTitle:@"口可口可"
style:UIBarButtonItemStylePlain target:self
action:@selector(leftAction:)];

    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithTitle:@"口合口合"
style:UIBarButtonItemStylePlain target:self
action:@selector(rightAction:)];

}

- (void)btnAction:(UIButton *)sender{
    NSLog(@"点击了..");

    FirstViewController *firstVC = [[FirstViewController
alloc]init];
    [self.navigationController pushViewController:firstVC
animated:YES];
    [firstVC release];
    firstVC = nil;

}
```

解释:

1、在这个根视图控制器自己的view中，定义了一个Button，点击的时候，去加载一个视图FirstViewController，同时导航控制器把FirstViewController入栈。这样的效果，就是点击button的时候，打开一个view，这个view是FirstViewController的view。此时FirstViewController就是RootViewController的

下一层。

2、self.navigationController，获取所在的导航控制器

3、pushViewController，把视图控制器入栈。（栈顶）

4、self.navigationItem.titleView，在导航控制器的view里添加一个view。

5、self.navigationItem.leftBarButtonItem，导航控制器的view的左边的一个按钮，可以返回上一级（或者返回你想去的控制器的view），当返回的时候，就会出栈，把自己和把下一级的所有控制器都销毁。

6、self.navigationItem.rightBarButtonItem，导航控制器view的右边的按钮，可以编辑当前页或者前进或者去到某个视图（依据addtarget的action里边怎么写）。

FirstViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor greenColor];

    UIButton *btn = [UIButton buttonWithType:
(UIButtonTypeSystem)];
    btn.frame = CGRectMake(100, 100, 100, 100);
    btn.backgroundColor = [UIColor blueColor];
    [btn addTarget:self action:@selector(btnAction:)
forControlEvents:(UIControlEventTouchUpInside)];
    [self.view addSubview:btn];


    // navigation中的controller
//    NSLog(@"%@",self.navigationController.viewControllers);
    // 隐藏自带的back
    // 1、添加一个leftBar
//    self.navigationItem.leftBarButtonItem =
[[UIBarButtonItem alloc]initWithTitle:@"口可口可"
style:UIBarButtonItemStylePlain target:self
action:@selector(leftAction:)];
    // 2、直接隐藏
    self.navigationItem.hidesBackButton = YES;
//    NSLog(@"%@",self.navigationController);
    NSLog(@"%@",self.navigationController.viewControllers);

}

- (void)btnAction:(UIButton *)sender{
    NSLog(@"点击了..first");
    // 出栈，出来的是最上面的ViewController
    // 出栈的控制器会被销毁，数据就没了。
//    [self.navigationController
popViewControllerAnimated:YES];

    // secondViewController
    SecondViewController *secondVC = [[SecondViewController
alloc]init];
    [self.navigationController pushViewController:secondVC
animated:YES];
    [secondVC release];
    secondVC = nil;

}
```

解释：

1、隐藏导航控制器view的back：

（1）我们自己添加一个leftBar

（2）直接隐藏`self.navigationItem.hidesBackButton = YES;`

2、`[self.navigationController popViewControllerAnimated:YES];`把自己从导航控制器的栈中出栈。若是不是栈顶的试图控制器出栈，则会把该视图控制器以上的所有视图控制器都出栈。所有出栈的控制器的数据全部销毁。

3、回到根视图控制器：`[self.navigationController popToRootViewControllerAnimated:YES];`就会把根视图控制器以上的试图控制器全部出栈，销毁。

4、去到指定视图控制器：

```
 NSArray *temp = self.navigationController.viewControllers;
[self.navigationController  popToViewController:temp[1] animated:YES];
```

先获取所在导航控制器里所有的试图控制器，然后去到指定试图控制器，把自己下一集的试图控制器，都出栈销毁。

——大海
2015年05月07日21:08:23 北京·蓝欧

30、UITableView

UITableView是iOS中最重要的视图，没有之一。
基本上每个应用程序都会用到UITableView来布局。
UITableView继承于UIScrollView，所以可以滚动。可以表现为两种风格：UITableViewStylePlain和UITableViewStyleGrouped。
UITableView基本是用来显示数据。

UITableView的使用：

1、在AppDelegate.m中，把UINavigationController创建出来，并给window的rootViewController。
AppDelegate.m

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];

    RootViewController *root = [[RootViewController
alloc]init];

    UINavigationController *rootNC = [[UINavigationController
alloc]initWithRootViewController:root];
    self.window.rootViewController = rootNC;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

2、创建RootViewController，并初始化对应的视图RootView。

RootView.h

```
#import <UIKit/UIKit.h>

@interface RootView : UIView

// tableView继承于Scrollview
@property(nonatomic,retain)UITableView *tv;
@end
```

RootView.m

```
#import "RootView.h"

@implementation RootView

- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self p_setupView];
    }
    return self;
}

- (void)p_setupView{

    self.backgroundColor = [UIColor yellowColor];
    self.tv = [[[UITableView alloc]initWithFrame:self.bounds
style:(UITableViewStyleGrouped)]autorelease];
    // 分割线的样式
    self.tv.separatorStyle =
UITableViewCellSeparatorStyleSingleLine;
    // 分割线的颜色
    self.tv.separatorColor = [UIColor greenColor];

    self.tv.rowHeight = 100;
//    NSLog(@"%f",self.tv.rowHeight);
    [self addSubview:_tv];
}

- (void)dealloc
{
    [_tv release];
    [super dealloc];
}

@end
```

解释:

1、self.tv = [[[UITableView alloc]initWithFrame:self.bounds style:(UITableViewStyleGrouped)]autorelease];

第一个参数，放TableView的视图大小，位置等，一般设置跟屏幕一样大。

第二个参数，tableview的显示样式，这里是UITableViewStyleGrouped（分组形式）。

2、separatorStyle，tableview中，每一条信息之间，都有一条线隔开，这里设置的就是这条线的颜色。

3、rowHeight，设置每一行的行高，如果给0，在屏幕上就不显示建好的tableview（因为每行都重叠在一起了），所以，必须要给一个值。

RootViewController.m

```objc
#import "RootViewController.h"
#import "RootView.h"
#import "NextViewController.h"
// dataSource 的代理
@interface RootViewController
()<UITableViewDataSource,UITableViewDelegate>
@property(nonatomic,retain)RootView *rv;
@end

@implementation RootViewController
// 有多少个分区
// 1.确定有多少个分区
-(NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView{
    return 10;
}

// 每个分区有多少行
// 这个方法走多少次，根据有多少个分区决定
// 2.确定每个分区有多少航
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section{

    if (section == 2) {
        return  10;
    }else{
        return 5;
    }
}
// 创建cell
//这个方法走多少次，是根据各分区行数之和
// 3.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath{
//    UITableViewCell *cell = [[UITableViewCell
alloc]initWithStyle:(UITableViewCellStyleSubtitle)
reuseIdentifier:nil];

    // 重用标示符
    static NSString *cell_id = @"cell_1";
    // 先从重用池用找cell
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:cell_id ];
    // 判断是否找到cell
    if (cell == nil) {
        cell = [[UITableViewCell alloc]initWithStyle:
(UITableViewCellStyleSubtitle) reuseIdentifier:cell_id];
    }
    // 给cell赋值
    cell.textLabel.text = [NSString
stringWithFormat:@"section:%ld,row :%ld",
indexPath.section,indexPath.row ];
//    NSLog(@"section:%ld,row :%ld",
indexPath.section,indexPath.row);
    // cell 属性
    // 文本
    cell.backgroundColor = [UIColor colorWithRed:arc4random()
```

解释：

以下13点，都是UITableViewDataSource协议的方法。

datasource意思是数据源，每个tableview都有数据源，负责给cell提供数据，下面13点都是dataSource协议里的方法，UITableViewDataSource跟delegate功能一样，只是名字不一样。

1、(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView;
这个方法，是创建一个tableview过程中，最先执行的一个方法，这里，用来确定这个tableview有多少个分区（section）。返回值就是有多少个分区。

2、(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section，这个方法是执行的第二个方法，意思是，这个tableview里的这个分区里，有多少行。

3、- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath，这个方法是第三个执行的方法，也就是在这个方法里对tableview 进行初始化，创建出来。
第一个参数：当前操作的视图，
第二个参数：一般我们取这个对象里边的row（行）和section（分区）

4 [[UITableViewCell alloc]initWithStyle:(UITableViewCellStyleSubtitle) reuseIdentifier:cell_id],
创建UITableViewCell，UITableView中，每一个单元格，叫做cell，为UITableViewCell类的对象，有4种样式，UITableViewCellStyleSubtitle的样式是其中一种。
reuseIdentifier，是"根据一个重用符号对滑出屏幕的那些cell重用"。
在创建tableview的过程中，如果在当前屏幕中，能容下几个cell就先创建几个cell。当把tableview往上滑时，才会创建后面的cell，滑出屏幕的那些cell会被系统放入"重用池"，里边的信息没有删除，在重用池中，等待被系统重新使用。滑出屏幕的cell，已经被销毁（放进重用池），重新划回去（看到的不再是原来的cell，而是新的cell，可以是从重用池中重用的，也可能是新创建的，总之不再是原来的cell）。所以，在创建一个cell的时候，就会用一个字符串来标记，。重用机制，就是在重用池找有没有标记的cell如果有，拿出来用，如果没有，则直接创建一个新的cell。

5、cell.textLabel.text，给cell赋值

6、cell.imageView.image，给cell添加图片。每个cell其实有三部分组成，就是最左边的图片部分，中间的文本部分，右边的辅助信息部分。这个图片会按照cell的高按比例进行缩放，若cell行高足够高，则显示全部图片。

7、cell.detailTextLabel.text，文本区域的子标题

8、accessoryType，辅助信息（位于cell最右边），用于提示

9、accessoryView，也是辅助信息的另一种形式，是对type的扩展，可以放任意视图

10、selectionStyle，cell被选中的样式（基本失效）

11、-(NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section，用来设置每一个分区的标题。
第一个参数，当前tableview，第二个参数，当前分区

12、-(NSString *)tableView:(UITableView *)tableView titleForFooterInSection:(NSInteger)section，设置分区尾的标题，一般用来声明版权归谁所有等等。

13、- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView，tableview最右边的一排索引，可以点索引去到对应分区。

UITableViewDelegate 协议的方法

1、(CGFloat)tableView:(UITableView *)tableView heightForHeaderInSection:(NSInteger)section，每一个分区头部都可以设置任意view，这个方法就是view的高度。

2、-(UIView *)tableView:(UITableView *)tableView viewForHeaderInSection:(NSInteger)section，这个方法，就是在分区头部创建一个view。这里给的view的高度，要根据上面那个方法的高度来给。
3、- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath，这个方法是精准控制到某个区的某个cell。
4、- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath，这个方法是，点击一个cell，能干什么事，一般用来转到下一个视图。
5、在controller的viewDidload方法中设置代理：
self.rv.tv.dataSource = self;dataSource的设置
self.rv.tv.delegate = self;delegate的设置。
一般在tableview中，这两个协议都同时遵循。

31、界面间传值

界面间传值，分两种，一种情况是push入栈，从上一级视图到下一级视图，这只需要在下一级视图中，定义一个属性，用来把上一级的视图的值接收，在进入下一集视图之前封装好，然后在就可以在本视图中得到上一级传下来的值。
另一种是pop出栈，逆向传值，这样的情况，由于pop出栈的视图数据全部销毁，所以不能通过属性直接传值，此时可以用delegate来实现，声明一个协议，协议里定义一个可以传值的方法，参数即为要传出去的值。然后再在本视图中声明一个属性，遵循这个协议。这个视图就是相当于雇主，设置了一个协议，需要一个代理来实现这个协议的方法。此时就让上一级视图来接收这个协议，实现协议中的方法，这个时候，在下一级视图即将被pop出栈之前，用自己的代理来调用这个协议里的方法，此时的上一级的视图实现的方法的参数就可以赋给上一级视图的接受对象。

第一种（点击到下一级视图）：

RootViewController.m

```objc
#import "RootViewController.h"
#import "FirstViewController.h"
@interface RootViewController ()
@property(nonatomic,retain)UITextField *tf;
@end

@implementation RootViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 布局视图
    [self p_setupView];

}
#pragma mark 布局视图
- (void)p_setupView{
    self.view.backgroundColor = [UIColor yellowColor];

    // textField
    self.tf = [[UITextField
alloc]initWithFrame:CGRectMake(50, 100, 200, 50)];
    self.tf.borderStyle = UITextBorderStyleRoundedRect;
    [self.view addSubview:_tf];

    // rightBarButton
    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithBarButtonSystemItem:
(UIBarButtonSystemItemBookmarks) target:self
action:@selector(rightBarButtonAction:)];

}
#pragma mark 右barButton点击方法
- (void)rightBarButtonAction:(UIBarButtonItem *)sender{
    NSLog(@"右barButton被点击..");
    // 创建firstVC
    FirstViewController *firstVC = [[FirstViewController
alloc]init];

    // 使用属性接收值
    firstVC.textStr = self.tf.text;

    [self.navigationController pushViewController:firstVC
animated:YES];
    // 释放
    [firstVC release];
    firstVC = nil;
}
#pragma mark 响应触摸事件
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent
*)event{
    // 释放第一响应者，手键盘
    [self.tf resignFirstResponder ];
    // 阻断响应者链，也可以手键盘
//    self.view.userInteractionEnabled = NO;
}
```

解释：

1、`@property(nonatomic,retain)UITextField *tf;`声明属性，用来传值。

2、`FirstViewController *firstVC = [[FirstViewController alloc]init];`创建下一级视图，是为了得到下一级视图接收传值对象。

3、`firstVC.textStr = self.tf.text;`在下一级中声明的属性`testStr`就是用来接收上一级传过去值的对象。到这里就可获取到上个视图传到下一个视图的值。

在push入栈之前完成（也就是传到下一级视图之前）。

FirstViewController.h

```
#import <UIKit/UIKit.h>
#import "PassValueDelegate.h"
@interface FirstViewController : UIViewController
// 用来接收上一个页面的值
@property(nonatomic,copy)NSString *textStr;

// 声明代理
@property(nonatomic,assign)id<PassValueDelegate>delegate;

@end
```

`@property(nonatomic,copy)NSString *textStr;;`即为声明的接收传值属性。

`FirstViewController.m`接收值

```
#import "FirstViewController.h"
@interface FirstViewController ()
@property(nonatomic,retain)UITextField *tf;
@end
@implementation FirstViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    [self p_setupView];
    // 赋值，得到上一级传过来的值
    self.tf.text = self.textStr;

}
@end
```

`self.tf.text = self.textStr;`，获取到值

第二种情况：逆向传值

PassValueDelegate.h

声明一个协议

```
#import <Foundation/Foundation.h>

@protocol PassValueDelegate <NSObject>

- (void)passValue:(NSString *)aString;

@end
```

FirstViewController.h

```
#import <UIKit/UIKit.h>
#import "PassValueDelegate.h"
@interface FirstViewController : UIViewController
// 用来接收上一个页面的值
@property(nonatomic,copy)NSString *textStr;

// 声明代理
@property(nonatomic,assign)id<PassValueDelegate>delegate;

@end
```

@property(nonatomic,assign)id<PassValueDelegate>delegate;声明一个属性，遵循协议的属性。

FirstViewController.m

```
#import "FirstViewController.h"

@interface FirstViewController ()
@property(nonatomic,retain)UITextField *tf;
@end

@implementation FirstViewController

#pragma mark 布局视图
- (void)p_setupView{
    self.view.backgroundColor = [UIColor blueColor];
    // textfield
    self.tf = [[UITextField
alloc]initWithFrame:CGRectMake(50, 100, 200, 50)];
    self.tf.borderStyle = UITextBorderStyleRoundedRect;
    [self.view addSubview:_tf];
    // leftBarButton
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem
alloc]initWithBarButtonSystemItem:
(UIBarButtonSystemItemReply) target:self
action:@selector(leftButtonAction:)];
}

- (void)leftButtonAction:(UIBarButtonItem *)sender{
    NSLog(@"点击左边..");
    // 使用代理方法
    [self.delegate passValue:self.tf.text];
    [self.navigationController
popViewControllerAnimated:YES];
}
```

在pop出栈之前，用代理调协议的方法参数就是要传过去的值。

RootViewController.h

```
#import <UIKit/UIKit.h>
#import "PassValueDelegate.h"
@interface RootViewController :
UIViewController<PassValueDelegate>
@end
```

上一级视图（RootViewController），接收协议，遵循协议。

RootViewController.m

```
#import "RootViewController.h"
#import "FirstViewController.h"
@interface RootViewController ()
@property(nonatomic,retain)UITextField *tf;
@end

@implementation RootViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 布局视图
    [self p_setupView];
}
// 实现协议方法
-(void)passValue:(NSString *)aString{
    self.tf.text = aString;
}
#pragma mark 布局视图
- (void)p_setupView{
    self.view.backgroundColor = [UIColor yellowColor];

    // textField
    self.tf = [[UITextField
alloc]initWithFrame:CGRectMake(50, 100, 200, 50)];
    self.tf.borderStyle = UITextBorderStyleRoundedRect;
    [self.view addSubview:_tf];

    // rightBarButton
    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithBarButtonSystemItem:
(UIBarButtonSystemItemBookmarks) target:self
action:@selector(rightBarButtonAction:)];

}
#pragma mark 右barButton点击方法
- (void)rightBarButtonAction:(UIBarButtonItem *)sender{
// 创建firstVC
    FirstViewController *firstVC = [[FirstViewController
alloc]init];
// 设置代理
    firstVC.delegate = self;
     // 释放
    [firstVC release];
    firstVC = nil;
}
```

-(void)passValue:(NSString *)aString，实现协议方法，这个参数就是从下一级传进来的值，self.tf.text = aString;接收值。
但是，我们应该在创建下一级视图的同时，告诉雇主（下一级视图firstVC），你的代理是我。

—大海
2015年05月08日21:50:17 北京·蓝欧

32、UITableView编辑

tableview 的编辑可以分为四个步骤：
1、让tableView处于可编辑状态
2、指定哪些行可以被编辑
3、指定编辑样式（delete还是insert）
4、完成编辑（1、修改数据源，2、修改UI界面）

1、让tableView处于可编辑状态

（1）self.rv.tabelView.editing == YES

BOOL类型的editing，设置为YES，即为可编辑状态，为NO则为不可编辑状态。

（2）setEditing:!self.rv.tabelView.editing animated:YES

第一个参数：BOOL类型的，为YES则可编辑，为NO不可编辑
第二个参数：BOOL类型的，为YES则添加动画，为NO不添加动画

RootViewController.m

```
#pragma mark 响应方法
- (void)leftAction:(UIBarButtonItem *)sender{
    NSLog(@"添加..");
    // 记录添加样式
    _editingStyle = UITableViewCellEditingStyleInsert;
    // 1.让tableView处于编辑状态
    // 加动画
    [self.rv.tabelView setEditing:!self.rv.tabelView.editing
animated:YES];
}
- (void)rightAction:(UIBarButtonItem *)sender{
    NSLog(@"删除..");
    // 记录删除样式
    _editingStyle = UITableViewCellEditingStyleDelete;

//    if (self.rv.tabelView.editing == YES) {
//        [self.rv.tabelView setEditing:NO animated:YES];
//    }else{
//        [self.rv.tabelView setEditing:YES animated:YES];
//    }
    // 让tableView处于编辑状态 和 退出编辑状态
    [self.rv.tabelView setEditing:!self.rv.tabelView.editing
animated:YES];
}
```

解释：

navigation的响应事件（导航条左边为添加，右边为删除）：

1、`setEditing:!self.rv.tabelView.editing`，这里直接取！（取反），是为了确定tableView是否处于编辑状态，如果是编辑状态，则设置成不可编辑状态，如果是飞编辑状态，则设置成可编辑状态。

2、`_editingStyle = UITableViewCellEditingStyleInsert;` 是用来获取编辑样式的，其实应该说是记录样式，如果记录的样式是`insert`，则在后面提交的时候，往数据源添加数据，如果记录的样式是`delete`，那么再后面提交的时候，往把数据源的要删除的数据删除。

2、指定哪些行可以被编辑

```
// 2.指定哪些行可以被编辑
// 询问每一行处于什么状态（走行数那么多次）
- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath{
//    if (indexPath.row == 0) {
//        return YES;
//    }else{
//        return NO;
//    }
    return YES;

}
```

在创建cell 的时候，会每创建一个cell都会走一遍这个方法，目的是为了询问该行是否可编辑（与是否处于编辑状态不一样）。这里直接return YES的意思是告诉系统，我的每行都可以编辑。

3、指定编辑样式（delete还是insert）

```
#pragma mark delegate

// 3.指定编辑样式
- (UITableViewCellEditingStyle )tableView:(UITableView
*)tableView editingStyleForRowAtIndexPath:(NSIndexPath
*)indexPath{

    return _editingStyle;
}
```

解释：
这个方法是delegate里的方法；
这个方法里，直接return _editingStyle，是返回我们上面记录的编辑样式（delete还是insert）

4、完成编辑（1、修改数据源，2、修改UI界面）

```
// 4.完成编辑
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath{
    NSLog(@"编辑完成..");
    // 判断为删除样式
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // 1.修改数据源
        [self.dataArr[indexPath.section]
removeObjectAtIndex:indexPath.row];
        //    [self.dataArr[indexPath.section]
removeObjectAtIndex:indexPath.row+1];
        // 2.修改UI界面
        //    [tableView reloadData];

        //    NSIndexPath *ind = [NSIndexPath
indexPathForRow:indexPath.row +1
inSection:indexPath.section];
        // @[indexPath]告诉你是哪一行（数组）
        [tableView deleteRowsAtIndexPaths:@[indexPath]
withRowAnimation:(UITableViewRowAnimationLeft)];
    }else if(editingStyle ==
UITableViewCellEditingStyleInsert){// 判断为添加
        // 执行添加
        // 1. 修改数据源
//        [self.dataArr[indexPath.section] addObject:@"我是新
来的.."];
        [self.dataArr[indexPath.section] insertObject:@"我是新
来的哦" atIndex:indexPath.row + 1];
        // 2.修改UI
//        [tableView reloadData];
        // +1是为了控制再后一行出现
        NSIndexPath *ind = [NSIndexPath
indexPathForRow:indexPath.row+1 inSection:indexPath.section];
        [tableView insertRowsAtIndexPaths:@[ind]
withRowAnimation:(UITableViewRowAnimationLeft)];
    }
}
```

解释：

1、根据editingStyle判断返回来的编辑样式是insert还是delete，分别对数据源进行不同操作。

2、判断为insert编辑样式，修改数据源，[self.dataArr[indexPath.section] insertObject:@"我是新来的哦" atIndex:indexPath.row + 1]，在数组中改行的后面插入一个对象。indexPath.section作为数组的元素个数，是因为，前面定义的数据源的结构是一个可变数组里边嵌套的两个可变数组，这样self.dataArr[indexPath.section]就是取到外层大数组的的元素下标，这里的section的值就是数组的元素个数。indexPath.row + 1是大数组下标下，取得的小数组，小数组里元素的下标。这样就能把我们想要插入的东西插入到我们想要插入的地方。

3、NSIndexPath *ind = [NSIndexPath indexPathForRow:indexPath.row+1 inSection:indexPath.section];重新给indexPath设置新值，为了后面的多位置添加。
4、修改数据源完成后修改UI界面[tableView insertRowsAtIndexPaths:@[ind] withRowAnimation:(UITableViewRowAnimationLeft)]，ind数组是重新设置的indexPath的值，里边有多个section和 row。也就可以添加多个位置的信息。
5、删除和添加一样，但是要把添加变成删除。[self.dataArr[indexPath.section] removeObjectAtIndex:indexPath.row]。

33、UITableView移动

移动的步骤：

1、让tableView处于编辑状态

与编辑的第一步一样

```
#pragma mark 响应方法
- (void)leftAction:(UIBarButtonItem *)sender{
    NSLog(@"添加..");
    // 记录添加样式
    _editingStyle = UITableViewCellEditingStyleInsert;

    // 1.让tableView处于编辑状态
    // 加动画
    [self.rv.tabelView setEditing:!self.rv.tabelView.editing animated:YES];


}
```

2、指定哪些行可以移动

```
// 2.指定哪些行可以移动(第一步与添加删除一样，处于编辑状态)
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)indexPath{
    return YES;
}
```

这里return YES的意思是所有行都可以移动

3、完成移动

```
// 3.完成移动
- (void)tableView:(UITableView *)tableView
moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath
toIndexPath:(NSIndexPath *)destinationIndexPath{

    // 修改数据源
    // 取原来的
//    NSString *str = self.dataArr[sourceIndexPath.section]
[sourceIndexPath.row];
//
//    self.dataArr[sourceIndexPath.section]
[sourceIndexPath.row] =
self.dataArr[destinationIndexPath.section]
[destinationIndexPath.row];
//    self.dataArr[destinationIndexPath.section]
[destinationIndexPath.row] = str;

    // 思路：删除原来的，插入新的
    // 把原来的东西取出来放在str里
//    NSLog(@"%@",self.dataArr[sourceIndexPath.section]
[sourceIndexPath.row]);
    NSString *str = self.dataArr[sourceIndexPath.section]
[sourceIndexPath.row] ;
    // 删除原来的
    [self.dataArr[sourceIndexPath.section]
removeObjectAtIndex:sourceIndexPath.row];
    // 把str 插入新位置
    [self.dataArr[destinationIndexPath.section]
insertObject:str atIndex:destinationIndexPath.row];

    // 修改UI
    [tableView moveRowAtIndexPath:sourceIndexPath
toIndexPath:destinationIndexPath];
}
```

解释；

1、移动与编辑的区别，就是编辑第三步要给定一个样式，而移动不需要给定样式。

2、修改数据源：要把原来位置的数据，放在一个临时变量里，然后删除原来位置的数据，再在新的目标位置插入这个数据内容。

（1）、把原来位置的数据取出来放在临时变量str里。`NSString *str = self.dataArr[sourceIndexPath.section][sourceIndexPath.row]`

这里sourceIndexPath.section事原来位置的分区，sourceIndexPath.row原来位置分区里的第几行。（因为数据结构是大数组嵌套两个小数组，sourceIndexPath.section来去到小数组，sourceIndexPath.row来去到小数组里的元素）

（2）、删除原来位置值`[self.dataArr[sourceIndexPath.section] removeObjectAtIndex:sourceIndexPath.row]`。

（3）、把原来位置的数据插入新的位置

```
[self.dataArr[destinationIndexPath.section] insertObject:str
atIndex:destinationIndexPath.row]
```
3、修改UI：[tableView moveRowAtIndexPath:sourceIndexPath toIndexPath:destinationIndexPath]。
第一个参数：原来位置
第二个位置：移到的目标位置

4、但是这样的移动，会跨分区移动，会导致系统崩溃，因此，在移动的时候，需要对移动的位置进行监测：

```
#pragma mark TableView 自己的方法

 // 监视移动是否跨区域
- (NSIndexPath *)tableView:(UITableView *)tableView
targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath
*)sourceIndexPath toProposedIndexPath:(NSIndexPath
*)proposedDestinationIndexPath{
    // 判断是否再同一个区域，再同一区域，返回目的分区的行，不再同一区
域，返回原来分区的行
    if (sourceIndexPath.section ==
proposedDestinationIndexPath.section) {
        return proposedDestinationIndexPath;
    }else{
        return sourceIndexPath;
    }
}
```

解释：
1、
第一个参数：当前操作的tableView
第二个参数：原来位置
第三个参数：目标位置
2、判断，如果原来位置的分区等于目标位置的分区。返回新的位置，否则不在同一个分区，返回原来的位置。

注意，因为以上这些方法，都是RootViewController遵循了UITableViewDataSource,UITableViewDelegate这两个协议，协议里的方法，所以，必须要按照代理的方法，设置代理等步骤
设置代理：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.rv.tabelView.dataSource = self;
    self.rv.tabelView.delegate = self;
    [self p_data];
    // 处理数据
    // test
//    self.rv.tabelView.editing = YES;
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem
alloc]initWithTitle:@"添加" style:(UIBarButtonItemStylePlain)
target:self action:@selector(leftAction:)];
    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithTitle:@"删除" style:
(UIBarButtonItemStylePlain) target:self
action:@selector(rightAction:)];

}
```

34、UITableViewController

与UIViewController有区别，就是UITableViewController自身的带的是tableView。
AppDelegate.m

```
#import "AppDelegate.h"
#import "RootTableViewController.h"
@interface AppDelegate ()

@end

@implementation AppDelegate


- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];
    // Override point for customization after application
launch.
    self.window.backgroundColor = [UIColor whiteColor];

    RootTableViewController *rootTVC =
[[RootTableViewController alloc]initWithStyle:
(UITableViewStyleGrouped)];
    UINavigationController *ncRoot = [[UINavigationController
alloc]initWithRootViewController:rootTVC];
    self.window.rootViewController = ncRoot;

    [self.window makeKeyAndVisible];

    [ncRoot release];
    ncRoot = nil;
    [rootTVC release];
    rootTVC = nil;

    return YES;
}
```

解释：

1、与原来的UIViewController一样，创建出来的RootTableViewController的对象rootTVC，style可以用plain和grouped两种。

2、UINavigationController创建出来是用rootTVC作为参数创建的。

RootTableViewController.m

```
#import "RootTableViewController.h"
#define kCELL_ID @"cell_1"
@interface RootTableViewController ()

@end

@implementation RootTableViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 注册,注册要用的哪个cell

    [self.tableView registerClass:[UITableViewCell class]
forCellReuseIdentifier:kCELL_ID];
}
#pragma mark - Table view data source
//#warning 这好像有点问题
- (NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView {

    // Return the number of sections.
    return 2;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {

    // Return the number of rows in the section.
    return 10;
}


- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:kCELL_ID
forIndexPath:indexPath];

    cell.textLabel.text = [NSString
stringWithFormat:@"section:%ld,row:
%ld",indexPath.section,indexPath.row];

    return cell;
}


/*
// Override to support conditional editing of the table view.
- (BOOL)tableView:(UITableView *)tableView
canEditRowAtIndexPath:(NSIndexPath *)indexPath {
    // Return NO if you do not want the specified item to be
editable.
    return YES;
}
*/
/*
// Override to support editing the table view.
```

解释：

1、UITableViewController已经帮我们封装了我们实现代理那些方法，就是上面那些注释起来的方法。

2、`[self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:kCELL_ID]`，是对我们将要用到的cell进行注册。第一个参数，是UITableViewCell类，第二个参数，是我们设的宏定义，用来作为cell的重用字符。

——大海

在.....想你的三百六十五天·韩子

2015年05月11日20:11:00 北京·蓝欧

35、自定义UITableViewCell

自定义cell 的出现，是为了方便管理，简化页面的布局。

自定义的cell就是要把自定义一个tableViewCell类，继承于UITableViewCell，然后再在各个cell上封装各种控件。在自定义cell中，要注意的是，要把控件加到

cell 的contentView上。用contentView来管理显示cell里的控件，不会影响self的view的布局。移动，添加，显示，都在contentView上完成，而不需要在self的view上。

MyTableViewCell.h

```
#import <UIKit/UIKit.h>
@class Person;
@interface MyTableViewCell : UITableViewCell
@property(nonatomic,retain)UIImageView *myImageView;

@property(nonatomic,retain)UILabel *myLabel;
// 返回高度
+ (CGFloat)myCellHeight;
// 使用person对象赋值
- (void)personValue:(Person *)aPerson;


@end
```

MyTableViewCell类继承于UITableViewCell，拥有UITableViewCell里全部的东西。

- (void)personValue:(Person *)aPerson，声明一个方法，参数是person类的对象，是为了给我需要的属性和变量进行赋值（self.myImageView.image，self.myLabel.text）。

声明一个UILabel类的属性myLabel

MyTableViewCell.m

```
#import "MyTableViewCell.h"
#import "Person.h"
@implementation MyTableViewCell
// 重写初始化方法
-(instancetype)initWithStyle:(UITableViewCellStyle)style
reuseIdentifier:(NSString *)reuseIdentifier{
    if (self = [super initWithStyle:style
reuseIdentifier:reuseIdentifier]) {
        // 加载视图
        [self p_setupView];
    }
    return self;
}
// 使用person对象赋值
- (void)personValue:(Person *)aPerson{

    // 设置属性
    self.myImageView.image = [UIImage
imageNamed:aPerson.imageStr];
    self.myLabel.text = aPerson.name;
}
- (void)p_setupView{
    self.myImageView = [[UIImageView
alloc]initWithFrame:CGRectMake(10, 10, 60, 60)];
    self.myImageView.backgroundColor = [UIColor orangeColor];
    // 所有自定义cell控件，都加在contentView上。contentView再cell
之上
    [self.contentView addSubview:_myImageView];
    self.myLabel = [[UILabel
alloc]initWithFrame:CGRectMake(CGRectGetMaxX(self.myImageView
.frame)+10, CGRectGetMinY(self.myImageView.frame),
CGRectGetWidth(self.contentView.frame)-
CGRectGetWidth(self.myImageView.frame)-30,
CGRectGetHeight(self.myImageView.frame))];
    self.myLabel.backgroundColor = [UIColor lightGrayColor];
    [self.contentView addSubview:_myLabel];

}
+ (CGFloat)myCellHeight{
    return 80;
}
// bounds改变的时候自动调用
- (void)layoutSubviews{

//    NSLog(@"%@",NSStringFromCGRect(self.frame));
    // 改变
    self.myLabel.frame =
CGRectMake(self.myLabel.frame.origin.x,
self.myLabel.frame.origin.y,
CGRectGetWidth(self.contentView.frame)-
CGRectGetWidth(self.myImageView.frame)-30,
self.myLabel.frame.size.height);
}

- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    [super setSelected:selected animated:animated];
//    NSLog(@"选中？");
```

解释：

1、自定义cell要重写原来的初始化方法initWithStyle:reuseIdentifier:，再初始化方法里，对自定义cell重新进行控件的布局。

2、- (void)p_setupView，自定义的私有方法，用来对控件的布局，所有的控件布局都写在这个方法里。

3、[self.contentView addSubview:_myImageView];所有的控件都加在contentView上，contentView是再cell自己的视图之上的。我们后面操作的cell里的控件都是再contentView上操作的。？

4、+ (CGFloat)myCellHeight，我们自己生命的一个类方法，用来返回cell的固定高度。

5、- (void)layoutSubviews，这个方法是再bounds改变的时候，会自动调用，？

6、- (void)personValue:(Person *)aPerson，这个方法传进我们alloc出来的Person的一个对象，这个对象里的数据赋值给我们声明的self.myImageView.image和self.myLabel.text。

```
7、- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    [super setSelected:selected animated:animated];
//    NSLog(@"选中？");

}？？？
```

RootTableViewController.m

```objc
#import "RootTableViewController.h"
#import "MyTableViewCell.h"
#import "Person.h"
#define kCELL_ID @"cell_1"
@interface RootTableViewController ()
// 记录分组名
@property(nonatomic,retain)NSMutableArray *groupNameArr;
// 记录分组
@property(nonatomic,retain)NSMutableDictionary *groupDict;

@end

@implementation RootTableViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    [self.tableView registerClass:[MyTableViewCell class]
forCellReuseIdentifier:kCELL_ID];

    self.tableView.separatorStyle =
UITableViewCellSeparatorStyleNone;
    [self p_setData];
}

#pragma mark 数据处理
- (void)p_setData{
   NSString *path = [[NSBundle
mainBundle]pathForResource:@"stu.plist" ofType:nil];
    NSDictionary *dict = [NSDictionary
dictionaryWithContentsOfFile:path];

    NSLog(@"%@",dict);

    // 创建groupDict
    self.groupDict = [NSMutableDictionary dictionary];

    // 遍历
    for (NSString *key in dict) {

        NSMutableArray *tempArr = [NSMutableArray array];
        // 对应key下创建可变数组
        [self.groupDict setObject:tempArr forKey:key];
        // 遍历数组，得到小字典
        for (NSDictionary * d in dict[key]) {
            Person *p = [[Person alloc]init];
            // kvc赋值
            [p setValuesForKeysWithDictionary:d];
            // 蒋赋好值的person对象存入
            [self.groupDict[key] addObject:p];
        }
    }
    // 打印检测
    for (NSString *key in self.groupDict) {
        for (Person *p in self.groupDict[key]) {
            NSLog(@"name : %@",p.name );
        }
    }

    // 获取分组名
```

解释：

1、再controller中的- (void)viewDidLoad 方法中，是在视图加载完成后，执行[self.tableView registerClass:[MyTableViewCell class] forCellReuseIdentifier:kCELL_ID];，对MyTableViewCell 进行注册，只有注册了，再后面创建cell的时候，才不会出现崩溃。这里对哪个cell进行注册，下面就只能创建哪个cell。

2、self.tableView.separatorStyle = UITableViewCellSeparatorStyleNone;这里是把每个cell的分割线都设置为没有分割线

3、[self p_setData]，执行到这里，就会去执行p_setData方法，这个方法，用来处理数据。

4、我们从.plist文件得到数据源，plist文件，是我们自己创建的专门加入数据的文件，这样添加数据，会比直接再代码里添加好的多，因为plist文件是持久再硬盘中的，不用占用内存，需要的时候，才把数据加载到内存，NSString *path = [[NSBundle mainBundle]pathForResource:@"stu.plist" ofType:nil];

这个语句是获取我们plist文件的路径。

5、我们的数据加载进来后，用字典来装数据NSDictionary *dict = [NSDictionary dictionaryWithContentsOfFile:path]

6、[p setValuesForKeysWithDictionary:d]这里是KVC赋值，这里的KVC赋值，能大大减少我们的赋值工作，详细赋值情况，在OC笔记中的KVC那节课有介绍。

7、- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView，这个是dataSource里的方法，是创建cell前要告诉有几个分区。里边返回的return self.groupNameArr.count;是因为数组元素个数就是分区个数。在上节课有介绍。

8、- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section，这个方法是再cell创建前，告诉系统，有多少行。这个方法是dataSource所必须实现的。这里的返回值：return [self.groupDict[self.groupNameArr[section]] count]; 数组groupNameArr取到的是key，然后用groupDict根据key取到有多少个数组。然后计算个数。

9、创建一个cell，kCELL_ID 是宏定义，作为cell的重用标示符。MyTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:kCELL_ID forIndexPath:indexPath];

10、给我们的Person类赋值，（再这里就提到Model类Person，model类使用来对接数据源和我们用到真是数据的中间类，这个model类负责把数据源的数据，进行相应处理，处理成我们所需要的数据，然后返回给controller。）Person *p = self.groupDict[self.groupNameArr[indexPath.section]][indexPath.row]这里的字典取出来的东西，与上一条一样，groupNameArr[indexPath.section]是取到所有的key，groupDict[key]就是我们字典里的数组，数组取下标[indexPath.row]，就是我们之前从数据源赋值进来的Person对象。

11、- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath

这个方法是返回row的高度，里边的返回是return [MyTableViewCell myCellHeight]，这里是调用之前声明好的类方法，返回cell 的固定高度。这个高度返回给row的高度。

12、大多数数据处理的方式，是用数组套数组，字典套字典，数组套字典，字典套数组。我们这个程序的数据处理方式，是：

（1）.最外层是字典，字典的key是我们再plist里写的L和W，value是数组

（2）.字典里套数组，分别是L和W的value，

（3）． 每个数组里有一个小字典，小字典里的key是imageStr和name，value是

图片名字和姓名。

给你的爱一直很安静，每天那么喧闹，却每次在别人的笑声中，思念你·韩子...

—大海
2015年05月13日22:55:31 北京·蓝欧

36、多种类型的CELL混合使用

很多情况下，都会多个cell混合使用。比如qq聊天，微信聊天等等。
多个cell混合使用，如何创建：
AppDelegate.m

```
#import "AppDelegate.h"
#import "RootTableViewController.h"
@interface AppDelegate ()
@end
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];
    // Override point for customization after application
launch.
    self.window.backgroundColor = [UIColor whiteColor];

    RootTableViewController *rootTVC =
[[RootTableViewController alloc]initWithStyle:
(UITableViewStyleGrouped)];

    UINavigationController *ncRoot = [[UINavigationController
alloc]initWithRootViewController:rootTVC];
    self.window.rootViewController = ncRoot;
    [self.window makeKeyAndVisible];
    return YES;
}
```

解释：
1、与我们普通创建cell一样，在AppDelegate这个类里，创建一个tableViewController。

```
RootTableViewController *rootTVC = [[RootTableViewController
alloc]initWithStyle:(UITableViewStyleGrouped)];
```

2、创建一个UINavigationController，根据rootTVC创建。
3、self.window.rootViewController = ncRoot;，把导航添加到window的根视图。
RootTableViewController.m

```objc
#import "RootTableViewController.h"
#import "Person.h"
#import "BoyCell.h"
#import "GirlCell.h"
#define kCell_Boy @"Boy"
#define kCell_Girl @"Girl"
@interface RootTableViewController ()

@property(nonatomic,strong)NSMutableArray *dataAarr;

@end

@implementation RootTableViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 注册
    [self.tableView registerClass:[BoyCell class]
forCellReuseIdentifier:kCell_Boy];
    [self.tableView registerClass:[GirlCell class]
forCellReuseIdentifier:kCell_Girl];

    // 数据处理
    [self p_setData];
}
#pragma mark 处理数据
- (void)p_setData{
    Person *p1 = [[Person alloc]init];
    p1.name = @"老王";
    p1.sex = @"女";
    Person *p2 = [[Person alloc]init];
    p2.name = @"老李";
    p2.sex = @"女";
    Person *p3 = [[Person alloc]init];
    p3.name = @"老铁";
    p3.sex = @"女";
    Person *p4 = [[Person alloc]init];
    p4.name = @"老朱";
    p4.sex = @"男";
    Person *p5 = [[Person alloc]init];
    p5.name = @"老马";
    p5.sex = @"男";
    // 装入数组
    self.dataAarr = @[p1,p2,p3,p4,p5].mutableCopy;
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
#pragma mark - Table view data source
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {

    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {

    return self.dataAarr.count;
}
// 创建cell

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

解释：

1、要创建多少个cell就要注册多少次cell。这里注册了两个cell。

```
// 注册
    [self.tableView registerClass:[BoyCell class]
forCellReuseIdentifier:kCell_Boy];
    [self.tableView registerClass:[GirlCell class]
forCellReuseIdentifier:kCell_Girl];
```

2、`self.dataAarr = @[p1,p2,p3,p4,p5].mutableCopy;`是我们将可变数组声明成属性，将赋好值的person对象装入数组中，这里`@[]`指的是数组字面量，`@[].mutableCopy`是可变数组字面量必须要加上的。

3、创建cell方法中，用if来判断性别，用于对这两个cell布局不一样。

BoyCell.m

```
#import "BoyCell.h"

@implementation BoyCell
-(instancetype)initWithStyle:(UITableViewCellStyle)style
reuseIdentifier:(NSString *)reuseIdentifier{
    self = [super initWithStyle:style
reuseIdentifier:reuseIdentifier];
    if (self) {
//        [self p_setupView];
    }
    return self;
}
// 布局视图
- (void)p_setupView{
    self.nameLabel = [[UILabel alloc]init];
    _nameLabel.frame = CGRectMake(0, 0, 100, 44);
    _nameLabel.backgroundColor = [UIColor lightGrayColor];
    [self.contentView addSubview:_nameLabel];
}
// 再getter 方法中写懒加载
-(UILabel *)nameLabel{
    [self p_setupView];
    return _nameLabel;
}
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    [super setSelected:selected animated:animated];
}
@end
```

解释：

1、重写BoyCell的初始化方法，这里不把布局放在初始化方法里，是为了后面的懒加载。

2、把布局放在nameLabel的getter方法里，也就是当后面调用nameLabel的getter方法的时候，才会把布局创建好。也就是我们经常用self.nameLabel的时候，就会完成布局。注意的是，这里的布局是根据自己的需要，那些元素，哪些控件需要懒加载才出现的，才把这些控件写在getter方法里布局。

37、自适应高度

像微博之类的通常发的微博不分长短，占屏幕的高度也不一样，这就是根据文

字，内容的长短，高度，来确定一个cell的高度。
这就是自适应高度。
RootTableViewController.m

```
#import "RootTableViewController.h"
#import "RootCell.h"
#define kCell_id @"cell_1"
@interface RootTableViewController ()
@property(nonatomic,strong)NSString *str ;
@end
@implementation RootTableViewController
- (void)viewDidLoad {
    [super viewDidLoad];

    [self.tableView registerClass:[RootCell class]
forCellReuseIdentifier:kCell_id];
}
#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return 3;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    RootCell *cell = [tableView
dequeueReusableCellWithIdentifier:kCell_id forIndexPath:indexPath];
    // 1.获取数据
    self.str = @"Sent when the application is about to move from
active to inactive state. This can occur for certain types of
temporary interruptions (such as an incoming phone call or SMS
message) or when the user quits the application and it begins the
transition to the background state.";
    // 创建myLabel，text赋值
//    NSString *s = [NSString stringWithFormat:@"\ue416😉"];
    cell.myLabel.text = _str;
    // 计算文字高度
    CGFloat h = [self stringHeight:_str];
   // label 的高度，就是文字的高度
    CGRect temp = cell.myLabel.frame;
    temp.size.height = h;
    cell.myLabel.frame = temp;
    return cell;
}
// 3.cell 的高度（固定高度+文字高度）
- (CGFloat )tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath{
    return [RootCell cellHeight]+[self stringHeight:_str];
}

// 2.计算文字的高度
- (CGFloat )stringHeight:(NSString *)aString{
     CGRect r = [_str boundingRectWithSize:CGSizeMake([[UIScreen
mainScreen] bounds].size.width-20, 2000)
options:NSStringDrawingUsesLineFragmentOrigin
attributes:@{NSFontAttributeName:[UIFont systemFontOfSize:17.0f]}
context:nil];
    return r.size.height;
}
@end
```

解释：

1、再viewDidLoad方法里，注册cell
2、获取一段较长的文字用作测试。
3、得到文字后，计算文字的高度

计算文字高度：我们自定义的方法，

```
// 2.计算文字的高度
- (CGFloat )stringHeight:(NSString *)aString{
     CGRect r = [_str boundingRectWithSize:CGSizeMake([[UIScreen
mainScreen] bounds].size.width-20, 2000)
options:NSStringDrawingUsesLineFragmentOrigin
attributes:@{NSFontAttributeName:[UIFont systemFontOfSize:17.0f]}
context:nil];
    return r.size.height;
}
```

方法里，str调的那个方法，是根据宽度，自己的字号大小，来确定得到文字的总高度。
4、获cell的高度，cell的高度为固定高度+文字高度。

```
// 3.cell 的高度（固定高度+文字高度）
- (CGFloat )tableView:(UITableView *)tableView
heightForRowAtIndexPath:(NSIndexPath *)indexPath{
    return [RootCell cellHeight]+[self stringHeight:_str];
}
```

[RootCell cellHeight]是我们再RootCell类里声明的一个类方法，返回一个固定高度，这个固定高度，包括各种边距。
这就完成了我们的自适应高度。
总的来说，对一个cell要根据内容来调整高度，应该按照如下步骤：
1、获取文字内容
2、计算文字高度（调boundingRectWithSize固定方法）
3、计算cell的高度（包括固定高度和给的文字高度）在heightForRowAtIndexPath方法里设置cell的高度。

38、单例模式

单例模式，就是我们定义的一个类，这个类只创建一次对象，也只创建一个对象。这个类的对象，供整个程序使用。
单例模式的三个必要条件：
1、单例类只能有一个对象
2、这个变量必须是自己自行创建的。
3、这个变量必须给整个程序使用。
我们创建一个单例类，如下：
TestHandle.h

```objc
#import <Foundation/Foundation.h>

@interface TestHandle : NSObject

@property(nonatomic,strong)NSString *str;

// 类方法，创建对象
+ (instancetype)shareTest;
@end
```

TestHandle.m

```objc
#import "TestHandle.h"
// 保证生命周期：从程序开始，到程序结束
static TestHandle *testHandle = nil;

@implementation TestHandle
//单例类
#pragma mark 伪单例
// 类方法，创建对象
+ (instancetype)shareTest{
    if(testHandle == nil){
        testHandle = [[TestHandle alloc]init];
        testHandle.str = @"呵呵";
    }
    return testHandle;
}
@end
```

解释：

1、`static TestHandle *testHandle = nil;`用static来修饰，保证生命周期一直到程序结束。并且将其赋值为nil。
2、创建一个类方法，（目的是只能用类名来调，不能用对象来调）
3、判断testHandle是否为nil，如果为nil就创建一个对象，如果不为nil，则返回原来的testHandle给外面。这就保证了只创建一次这个类的对象。

FirstViewController.m

```
@interface FirstViewController ()
@property(nonatomic,strong)UILabel *label;
@end

@implementation FirstViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.label = [[UILabel alloc]initWithFrame:CGRectMake(100, 100,
100, 100)];
    self.label.backgroundColor = [UIColor yellowColor];
    [self.view addSubview:_label];
}
// 视图将要出现
//-(void)viewWillAppear:(BOOL)animated{
//
//}


// 视图已经出现
-(void)viewDidAppear:(BOOL)animated{
    _label.text = [[TestHandle shareTest] str];
}
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
    SecondViewController *sec = [[SecondViewController alloc]init];
    [self.navigationController pushViewController:sec
animated:YES];
}
```

类外使用单例:
`_label.text = [[TestHandle shareTest] str];`这里的`[TestHandle shareTest]`就是我们用类名调类方法，然后根据上面的步骤判断是否为空，是否创建等等。

39、模态viewController

模态viewController是我们在页面中，遇到像编辑，注册，登陆，相册，相机等等的时候，需要弹出一个页面，这个页面应该用模态来展示。
模态viewController不在响应者链中，它只是一个临时弹出来满足我们需求的界面，这个界面弹出来，覆盖在屏幕的最顶层，也就是我们看到的那层。当需求结束，这个界面收回。它不作为响应者链内的响应者。
EditViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // barbutton
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem
alloc]initWithTitle:@"返回" style:(UIBarButtonItemStyleDone)
target:self action:@selector(leftAction:)];
    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithTitle:@"完成" style:
(UIBarButtonItemStyleDone) target:self
action:@selector(rightAction:)];
}
- (void)leftAction:(UIBarButtonItem *)sender{
    NSLog(@"返回");
    // 模态消除
    [self dismissViewControllerAnimated:YES completion:nil];
}
- (void)rightAction:(UIBarButtonItem *)sender{
    NSLog(@"完成..");
    if (_indexPath != nil) {
        [self updatePerson];
    }else{
        // 添加
        [self addNewPerson];
    }
    // 退出模态
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

解释，这里再导航条上添加了两个按钮，用于编辑和返回，并给响应方法，再响应方法中，leftAction: 方法里，用self来调模态消失方法。
`[self dismissViewControllerAnimated:YES completion:nil];`
在rightAction:方法中，调用`[self presentViewController:editNc animated:YES completion:nil];`也是让模态出现。

40、UITabBarController-标签视图控制器

UITabBarController是我们一个app在屏幕下方的一块，与UINavigationController类似，都是用来管理多个视图控制器的控制器。用UITabBarController能很方便的对平级的视图控制器进行切换，这些视图控制器之间没有直接的关系，也没有等级关系是一种平级的关系。

UITabBarController的使用

AppDelegate.m

```
@interface AppDelegate ()<UITabBarControllerDelegate>
@end
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    // tabBarController
    UITabBarController *tabBar = [[UITabBarController
alloc]init];
    self.window.rootViewController = tabBar;
    // viewController
    BlueViewController *blueVc = [[BlueViewController
alloc]init];
//    [blueVc.tabBarItem setImage:[UIImage
imageNamed:@"2.png"]];
    // 系统提供样式.
    blueVc.tabBarItem = [[UITabBarItem
alloc]initWithTabBarSystemItem:(UITabBarSystemItemContacts)
tag:100];
    blueVc.tabBarItem.title = @"蓝色";
    // 角标
    blueVc.tabBarItem.badgeValue = @"+99";

    RedViewController *redVc = [[RedViewController
alloc]init];
    redVc.tabBarItem = [[UITabBarItem
alloc]initWithTabBarSystemItem:(UITabBarSystemItemBookmarks)
tag:101];
    redVc.tabBarItem.title = @"红色";
    redVc.tabBarItem.badgeValue = @"我擦";
    // 设置代理
    tabBar.delegate = self;
    // 控制器数组
    tabBar.viewControllers =
@[blueVc,redVc,greenVc,grayVc,orangeVc,yellowVc];
    // 选中数组中的控制器
    tabBar.selectedIndex = 0;
    tabBar.tabBar.barTintColor = [UIColor purpleColor];
    // 设置背景图片
//    [tabBar.tabBar setBackgroundImage:[UIImage
imageNamed:@"2.png"]];
    [self.window makeKeyAndVisible];
    return YES;
}
- (void)tabBarController:(UITabBarController
*)tabBarController didSelectViewController:(UIViewController
*)viewController{
    NSLog(@"--------
%ld---------",tabBarController.selectedIndex);
    viewController.tabBarItem.badgeValue = nil;
}
```

解释:

1、UITabBarController *tabBar = [[UITabBarController

alloc]init];初始化UITabBarController。

2、self.window.rootViewController = tabBar;把UITabBarController的对象赋给window的根视图。

3、创建两个视图控制器BlueViewController和RedViewController。用来添加到UITabBarController中，让其所有。

4、tabBar.viewControllers = @[blueVc,redVc];将两个视图控制器加载到UITabBarController中，这里用的是数组对其进行赋值。

5、对两个视图控制器在UITabBarController里的样式和tag值，

```
blueVc.tabBarItem = [[UITabBarItem alloc]initWithTabBarSystemItem:(UITabBarSystemItemContacts) tag:100];
```

6、tabBar.delegate = self;必须要设置代理。因为后面有一个方法需要实现。

7、<UITabBarControllerDelegate>这个代理的方法。

8、blueVc.tabBarItem.title = @"蓝色";设置再tabBar中的显示的标题。

9、blueVc.tabBarItem.badgeValue = @"+99";是角标，也就是出现的红色的数字。

10、viewController.tabBarItem.badgeValue = nil;这个语句在delegate的方法中，目的是为了把再选中标签的时候，让红色的数字消失。

11、selectedIndex，选中某个视图控制器

12、在每一个tabBar中，每一个标签（每一个视图控制器），都用自己配套的navigation来管理自己的层级关系。也就是，有多少个视图控制器，要想管理自己视图的层级关系，就要有多少个UINavigationController，每一个navigation都只管理自己的层级关系，跟其他navigation不相干。

13、UIAppearance，一键换肤。可以对tabBar和navigationBar换肤。

41、Block传值

页面间的传值，有层级正向和逆向传值。正向传值一般用属性传值即可，前面第31节有详细介绍。逆向传值有代理传值和我们的block传值。代理传值，我们再第31节已经详细介绍了。这里说的是block如何传值。

FirstViewController.m（这是上级视图控制器）

```
#import "FirstViewController.h"
#import "SecondViewController.h"
@interface FirstViewController ()
@property(nonatomic,strong)UITextField *textField;
@end

@implementation FirstViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]initWithTitle:@"next" style:
(UIBarButtonItemStyleDone) target:self
action:@selector(rightAction:)];
}
- (void)rightAction:(UIBarButtonItem *)sender{

    SecondViewController *secondVc = [[[SecondViewController
alloc]init]autorelease];
    // 给second传值
    secondVc.str = self.textField.text;

    // 执行再second调用的block，赋值给first
    secondVc.mb = ^(NSString *str){
        self.textField.text = str;
    };
    [self.navigationController pushViewController:secondVc
animated:YES];

}
```

1、添加一个navigation导航的右按钮，添加点击事件

2、响应方法中，实现我们再second视图控制器中声明的 block。这个block再这里实现，在second视图中调用，传进来一个参数，那个参数就是我们再second视图想要再first视图显示的数据。这里就是str。

SecondViewController.h（下一级视图控制器）

```
#import <UIKit/UIKit.h>
// block
typedef void (^MyBlock)(NSString *str);

@interface SecondViewController : UIViewController
@property(nonatomic,copy)NSString *str;
// 定义一个block属性
// 定义一个block，我返回值，一个参数
@property(nonatomic,copy)MyBlock mb;

@end
```

`typedef void (^MyBlock)(NSString *str);`给一个block取名为MyBlock，参数是NSString类型的。

`@property(nonatomic,copy)MyBlock mb;`在这里声明一个block变量mb。为的是再其他类中可以用到。

SecondViewController.m

```
#import "SecondViewController.h"
@interface SecondViewController ()
@property(nonatomic,strong)UITextField *textField;
@end

@implementation SecondViewController

- (void)dealloc
{
    // block 的release
    Block_release(_mb);
    [super dealloc];
}
- (void)viewDidLoad {
    [super viewDidLoad];
    self.view.backgroundColor = [UIColor lightGrayColor];
    self.textField = [[[UITextField
alloc]initWithFrame:CGRectMake(100, 100, 200,
50)]autorelease];
    _textField.backgroundColor = [UIColor blueColor];
    self.textField.text = self.str;
    [self.view addSubview:_textField];
    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem
alloc]initWithTitle:@"上一个" style:(UIBarButtonItemStyleDone)
target:self action:@selector(leftAction:)];
}

- (void)leftAction:(UIBarButtonItem *)sender{
    // pop之前，调一个block传值
    self.mb(self.textField.text);

    [self.navigationController
popToRootViewControllerAnimated:YES ];

}
```

1、self.mb(self.textField.text);这里调用block将textField的值作为参数，去到first视图控制器中执行。就能实现把值传给上级视图的需求。
2、再dealloc方法中，释放block，固定方法Block_release(_mb);

—大海
遇见你，是此生最美的风景·韩子...
2015年05月14日23:03:20 北京·蓝欧

42、XML、JSON数据结构解析

所谓解析，就是我们跟后台数据对接的时候，根据固定的格式来提取数据内容的一种方式。
解析的前提，1、提前订好用什么格式。2、提供数据的时候按照格式提供。3、获取数据的时候按照格式获取。

XML解析

xml是一种可扩展标记语言，作用是用来存储数据和传输数据。
功能：数据存储、内容管理、配置文件
xml解析分为两种：

SAX 逐行解析

SAX解析，是逐行解析的，也就是从根节点开始，把所有的标签对应匹配，直到根节点标签也匹配完成，则完成解析。
date.txt（数据源）

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
      <message>
        <receiver>百里与</receiver>
        <sender>张雷</sender>
        <date>2015年5月18日</date>
        <content>今晚如家..</content>
      </message>
      <message>
        <receiver>私钥</receiver>
        <sender>城东林</sender>
        <date>2015年5月18日</date>
        <content>今晚如家东哥..</content>
      </message>
      <message>
        <receiver>鹏神</receiver>
        <sender>金源</sender>
        <date>2015年5月28日</date>
        <content>今晚如家彭老板..</content>
      </message>
</messages>
```

里边对应三组message，根节点为messages，根节点只能有一对。

Model.h（我们的model类，用来把数据源获取的东西存入对象，然后再界面显示）

```
#import <Foundation/Foundation.h>
@interface Model : NSObject
@property(nonatomic,copy)NSString *receiver;
@property(nonatomic,copy)NSString *sender;
@property(nonatomic,copy)NSString *date;
@property(nonatomic,copy)NSString *content;
@end
```

model类里，一般再数据源里，有多少个节点有我们想要的数据，就定义多少个属性，并且这些属性名字跟数据源的标签（节点）名字一样。这样做的目的

是为了在给model类对象赋值的时候，用kvc对其整体赋值，而不用一个 一个的判断数据源节点名字和model对象属性的名字匹配才能赋值。（kvc怎么赋值，去前面OC笔记里看）。

Model.m

```
#import "Model.h"
@implementation Model
// 在找不到key的时候不会崩
- (void)setValue:(id)value forUndefinedKey:(NSString *)key{
}
@end
```

这里定义了一个方法，是专门给kvc准备的，目的就是再kvc赋值的时候，找不到对应的键时，不会出现系统崩溃。

ViewController.m

```objc
#import "ViewController.h"
#import "Model.h"
// 遵循代理
@interface ViewController ()<NSXMLParserDelegate>
// 保存数据
@property(nonatomic,strong)NSMutableArray *dataArr;
// 拼接字符串
@property(nonatomic,strong)NSMutableString *catStr;
@end
@implementation ViewController
// xml的SAX解析
- (IBAction)xmlSAX:(id)sender {
    NSLog(@"xml的SAX解析");
    // 1.获取文件路径
    NSString *filePath = [[NSBundle mainBundle]
pathForResource:@"date.txt" ofType:nil];
    // 2.读取数据
    NSData *data = [NSData dataWithContentsOfFile:filePath];
//    NSLog(@"%@",data);
    // 3.解析data
    NSXMLParser *xmlParser = [[NSXMLParser
alloc]initWithData:data];
    // 4.设置代理
    xmlParser.delegate = self;
    // 5.开始解析
    [xmlParser parse];
}

#pragma mark -代理方法
// 1.开始解析文档
- (void)parserDidStartDocument:(NSXMLParser *)parser{
    NSLog(@"开始解析");
    self.dataArr = [NSMutableArray array];
}
// 2.开始解析标签
- (void)parser:(NSXMLParser *)parser didStartElement:
(NSString *)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName attributes:(NSDictionary
*)attributeDict{
    NSLog(@"开始解析标签");
    // 初始化字符串
    _catStr = [NSMutableString string];
    // 判断是否读到message,然后创建model类
    if ([elementName isEqualToString:@"message"]) {

        // 准备
        Model *model = [[Model alloc]init];
        // 放入数组
        [self.dataArr addObject:model];
    }
}
// 3.获取标签内内容string 参数就是传过来的标签里的内容，但是不一定是整
个内容，需要拼接
- (void)parser:(NSXMLParser *)parser foundCharacters:
(NSString *)string{
    NSLog(@"获取标签内内容");
    // 拼接字符串
```

解释：

1、我们测试，是用storyBoard再view上拖了按钮，按钮的点击事件的响应方法来实现的。

2、步骤：

（1）、获取文件路径：NSString *filePath = [[NSBundle mainBundle] pathForResource:@"date.txt" ofType:nil];第一个参数：文件名，第二个参数，文件类型，写nil即可。

（2）、获取文件里的数据，NSData *data = [NSData dataWithContentsOfFile:filePath];

（3）、开始用xml（SAX方式）解析数据data。NSXMLParser *xmlParser = [[NSXMLParser alloc]initWithData:data];即根据data初始化NSXMLParser。

（4）、设置代理xmlParser.delegate = self;再延展那里要把

（5）、开始解析[xmlParser parse];

（6）、遵循协议<NSXMLParserDelegate>

3、代理方法：

（1）、开始解析文档parserDidStartDocument:方法。参数是当前NSXMLParser类负责解析的对象

（2）、开始解析标签（节点）。_catStr = [NSMutableString string];catStr是我们之前声明的一个属性，用来拼接我们从数据源接收的到每一个标签里的内容。原因是我们再解析数据的时候，对每一个标签获取内容时，不一定一次能把内容获取完，只能用拼接的方式，把内容拼接完整。

（3）、if ([elementName isEqualToString:@"message"])判断当前标签是否与"message"一样，如果一样，则创建model类对象，并把model对象存入数组中。这个数组是我们声明的一个属性，用来存放我们从数据源获取的所有的数据。声明可变数组的语义用strong，因为如果用copy，则拷贝出来的那份数组是不可变数组。也就是，可变的东西，拷贝一份新的，新的那份数不可变的。

（4）、获取标签里的内容。parser:(NSXMLParser *)parser foundCharacters:(NSString *)string，这个string，就是我们标签里传回来的信息，不一定是完整的标签内容，可能只是一部分，因此需要拼接。[_catStr appendString:string];

（5）、结束标签。就是读取标签内容完事后，会遇到结束标签。然后判断每一个标签是什么的时候把拼好的字符串赋给对应model对象的属性。

但是这样的判断太复杂，这里用kvc直接对model对象的属性进行赋值即可。[model setValue:_catStr forKey:elementName];并且每次把_catStr赋完值后，需要对其清空。

（6）、完成赋值即结束解析文档。可以打印完成装载的数组。

（7）、parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError。当解析出现错误时，中断其他方法，跑来这方法执行。

dom解析

用storyBoard拖一个按钮的点击事件的响应方法，来测试dom解析。用dom解析，需要引入第三方GDataXMLNode类。这个类就贴出来了。

ViewController.m

```
// XML DOM解析（GData）
- (IBAction)xmlDom:(id)sender {
    // 1.获取文件路径
    NSString *filePath = [[NSBundle mainBundle]
pathForResource:@"date.txt" ofType:nil];
    // 2.获取数据
   NSData *data = [NSData dataWithContentsOfFile:filePath];

    // 3.创建对象(引入的第三方对象)
    GDataXMLDocument *xmlDocument = [[GDataXMLDocument
alloc]initWithData:data options:0 error:nil];

    // 4.获取根节点
    GDataXMLElement *rootElement = xmlDocument.rootElement;
    // 初始化数组
    self.dataArr = [NSMutableArray array];
    // 5.获取子节点
    for (GDataXMLElement *child in rootElement.children) {
        // 创建model类对象
        Model *model = [[Model alloc]init];
        for (GDataXMLElement *valueElement in child.children)
{
            // kvc
            // stringvalue是节点里的值
            // name是节点名
            [model setValue:valueElement.stringValue
forKey:valueElement.name];
        }
        // 存到数组
        [self.dataArr addObject:model];
    }
   for (Model *model in _dataArr) {
   NSLog(@"receiver:%@,sender:%@,date:%@,content:
%@",model.receiver,model.sender,model.date,model.content);
    }

}
```

解释：

1、获取文件路径

2、获取文件里的数据

3、创建引入的第三方类的对象。`GDataXMLDocument *xmlDocument = [[GDataXMLDocument alloc]initWithData:data options:0 error:nil];`第一个参数，获取的数据
第二个参数，放0 即可
第三个参数，错误信息，放nil即可

4、获取根节点：`GDataXMLElement *rootElement = xmlDocument.rootElement;`

5、获取根节点的子节点，是一个数组。rootElement.children

6、遍历根节点的子节点，获取子节点的子节点。child.children

7、子节点已经为叶子节点（即不再有分支的节点），就可以直接用kvc来往model类对象的属性赋值。`[model setValue:valueElement.stringValue forKey:valueElement.name];`

8、保存到数组`[self.dataArr addObject:model];`

JSON解析

json是一种轻量级的数据交换格式，没有xml那样的标签，因为再网络传输数据的时候，标签也占有流量，很多时候，内容没有标签所占的流量长。所以很多时候，用的是JSON来传输数据。但是JSON格式的数据不能很好的表明数据层次，因此存储数据，不应该用JSON，而是用xml。

json的表现形式如下：
数组用[ ]表示，多个数据用逗号（，）隔开。[person1,person2]
对象用{ }表示，每个对象用“”：“”表示，冒号前是键，冒号后是值。对象和对象之间用逗号（，）隔开。如：{“name”:”老王””age　”:”30”}

json系统解析

ViewController.m

```
// JSON系统解析
- (IBAction)json:(id)sender {
    // 1.获取文件路径
    NSString *filePath = [[NSBundle mainBundle]
pathForResource:@"jsonData.txt" ofType:nil];
    // 2.获取数据
    NSData *data = [NSData dataWithContentsOfFile:filePath];
    // 3.解析
    NSArray *arr = [NSJSONSerialization
JSONObjectWithData:data options:(NSJSONReadingAllowFragments)
error:nil];
//    NSLog(@"%@",arr);
    // 初始化
    self.dataArr = [NSMutableArray array];
    //4.model
    for (NSDictionary *dict in arr) {
        Model *model = [[Model alloc]init];
        // kvc
        [model setValuesForKeysWithDictionary:dict];
        [self.dataArr addObject:model];
    }
    // 打印
    for (Model *model in _dataArr) {
        NSLog(@"receiver:%@,sender:%@,date:%@,content:
%@",model.receiver,model.sender,model.date,model.content);
    }

}
```

解释：
1、获取文件路径。
2、获取文件的数据。
3、解析NSArray　*arr　=　[NSJSONSerialization

JSONObjectWithData:data options:(NSJSONReadingAllowFragments) error:nil];
第一个参数：获取的数据
第二个参数：返回的格式，
（1）NSJSONReadingMutableContainers返回可变数组或字典。
（2）NSJSONReadingMutableLeaves返回的是可变字符串
（3）NSJSONReadingAllowFragments返回的既不是数组也不是字典
4、遍历arr数组，获取里边的字典。这个为什么是字典，是因为我们跟后台数据沟通过后，得到的结果（这里就是我们自己设计的数据源，往回传的时候，就是数组里套字典）
5、在遍历数组arr里用kvc对model对象的属性进行赋值[model setValuesForKeysWithDictionary:dict];用字典赋值
6、把对象装入数组。

JSONKit（json第三方解析）

用json第三方类来解析，JSONKit类

```
// JSON第三方 JSONKit
- (IBAction)jsonKit:(id)sender {
    // 1.获取文件路径
    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"jsonData.txt" ofType:nil];
    // 2.获取数据
    NSData *data = [NSData dataWithContentsOfFile:filePath];

    // 3.解析
    NSArray *arr = [data objectFromJSONData];
//    NSLog(@"%@",arr);
    self.dataArr = [NSMutableArray array];
    //4.model
    for (NSDictionary *dict in arr) {
        Model *model = [[Model alloc]init];
        // kvc
        [model setValuesForKeysWithDictionary:dict];
        [self.dataArr addObject:model];
    }
    // 打印
    for (Model *model in _dataArr) {
        NSLog(@"receiver:%@,sender:%@,date:%@,content:%@",model.receiver,model.sender,model.date,model.content);
    }
}
```

解释：
1、获取文件路径
2、获取文件数据
3、解析NSArray *arr = [data objectFromJSONData];
4、遍历数组arr，创建model对象，用kvc对model类对象的属性进行赋值。
5、添加到数组dataArr中。

孤身一人 彷徨在大都市
即那个像被人丢弃在空啤酒罐

如果非要探究彼此的一切 才叫爱的话，还不如永久长眠
直到世界的尽头，也不愿与你分离
曾在千万个夜晚许下心愿
一去不回的时光，为何却如此耀眼
对憔悴不堪的心落井下石 渺茫的思念，
在这个悲剧的夜 而人们总是追求表面答案
结果错失无可取代的宝物 在这个充斥着欲望的街头
就连夜空繁星也难以照亮我们
在世界结束之前，谁愿给我讲一个 与繁花盛开最贴切的不幸
谁都满怀着期望，却又不相信永远
可是也一定梦想着明天
短暂的时光，与这个悲剧的夜
直到世界的尽头，也不愿与你分离
曾在千万个夜晚许下心愿
一去不回的时光，为何却如此耀眼
对憔悴不堪的心落井下石
渺茫的思念在这个悲剧的夜
在这个悲剧的夜 。。。。。。

—大海 直到世界的尽头
2014年7月6日 20:32:30 于广州市@韩子

—大海
2015年05月18日21:05:02 北京·蓝欧

43、网络编程

网络请求常用的是get和post方式的请求。但是这两者有区别，主要区别是参数怎么传给服务器。
get请求的参数会直接拼接到网址，用问号？隔开参数和域名。这样的请求方式比较快捷，数据量小，最大是255字符，并且最主要的是，地址是可以看见的，也就是说，参数是再地址上可以被用户看见的。
post请求的参数，不会显示再网址上，用户是看不见参数的，并且数据量很大，用NSData来传输，大概为1G左右。

get请求

get请求有同步请求和异步请求。
同步请求的意思，是按顺序请求，一个请求完事了，才能进行下一个请求。
异步请求的意思，是同时可以进行多个请求。
同步请求，只需坐了解，不用于开发。开发用的是异步请求。

get同步请求

我们用storyBoard再view上拖了一个按钮，点击事件的响应方法来测试请求。
ViewController.m

```
// get同步请求
- (IBAction)getT:(id)sender {
    // 1.准备url
    NSString *urlStr = [NSString stringWithFormat:@"http://
172.21.35.70/index.php?userName=%@&password=
%@",self.userName.text,self.passWord.text];

    NSURL *url = [NSURL URLWithString:urlStr];
    // 2.创建请求
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    // 3.设置
    [request setHTTPMethod:@"GET"];

    // 4.创建响应对象
    NSURLResponse *response = nil;
    // 5.创建错误对象
    NSError *error = nil;
    // 6.创建连接
    NSData *data = [NSURLConnection
sendSynchronousRequest:request returningResponse:&response
error:&error];

    // 7.数据处理
    // 返回类型
    NSLog(@"type = %@",response.MIMEType);
    // 返回长度
    NSLog(@"length =%lld",response.expectedContentLength);
    // 8.解析数据
    NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:data options:(NSJSONReadingAllowFragments)
error:nil];
    NSLog(@"%@",dict);
}
```

解释：

1、获取服务器的URL。先拼接一段服务器的网址，并把参数拼接到网址上。然后把拼接好的字符串，赋值给NSURL。

```
NSURL *url = [NSURL URLWithString:urlStr];
```

2、创建请求对象request。`NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];`这个对象是我们用来在网络上请求的必须工具。

3、设置请求方式为GET请求：`[request setHTTPMethod:@"GET"];`

4、创建一个请求出错的对象`NSError *error = nil;`请求出错的话，会怎么处理。

5、创建一个响应对象：`NSURLResponse *response = nil;`用来接受服务器传回来的进行封装，包括文档类型，大小，如果文本资源，则还包括文本的编码名称和文件名。

6、创建连接，请求回来的数据，用NSData接收

```
NSData *data = [NSURLConnection sendSynchronousRequest:request
```

returningResponse:&response error:&error];发送同步连接和反馈请求。
7、请求完毕，对请求回来的数据进行处理。
8、解析请求回来的数据：json解析。街完毕就可以给我们的model类赋值了。

get异步请求

get异步请求，有两种方式，delegate和block。都是固定用法，背下来就好。
这里只写delegate，block再post异步请求写。都是一样的。
这里测试，同样是通过按钮点击事件的响应方法来测试的。
ViewController.m

```
// get异步请求
- (IBAction)getY:(id)sender {
    // 1.准备url
    NSString *urlStr = [NSString stringWithFormat:@"http://
172.21.35.70/index.php?userName=%@&password=
%@",self.userName.text,self.passWord.text];

    NSURL *url = [NSURL URLWithString:urlStr];
    // 2.创建请求对象
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    // 3.建立连接
    NSURLConnection *conn = [NSURLConnection
connectionWithRequest:request delegate:self];
    // 4.启动
    [conn start];
}
#pragma mark -代理方法

// 1。收到响应的时候
- (void)connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response{
    self.recieverData = [NSMutableData data];
}

// 2。接收数据（不是一次性传完，要拼接）
- (void)connection:(NSURLConnection *)connection
didReceiveData:(NSData *)data{

    [_recieverData appendData:data];

}
// 3。数据接收完毕
-(void)connectionDidFinishLoading:(NSURLConnection
*)connection{
    NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:_recieverData options:
(NSJSONReadingAllowFragments) error:nil];
    NSLog(@"%@",dict);
}
// 4。发生错误
-(void)connection:(NSURLConnection *)connection
didFailWithError:(NSError *)error{
}
```

解释：

1、与同步请求一样，获取url

2、创建request请求对象。

3、创建连接，这里跟同步请求不一样。`NSURLConnection *conn = [NSURLConnection connectionWithRequest:request delegate:self];`
用的是代理的方法，就要在延展那个地方遵循协议
`<NSURLConnectionDataDelegate>`

4、启动链接`[conn start];`

5、遵循代理方法。

（1） 收到响应的方法，把可变`NSMutableData`初始化，这个对象用于接收传

回来的数据。
（2） 接收到数据的方法，把传回来的data添加到我们刚初始化的NSMutableData对象中。因为传回来的data不会一次性传完，我们需要再每一次传回来的时候，添加到NSMutableData的对象中，这样才是完整的数据。
（3） 接收数据完毕方法，再这个方法里，直接对传回的数据进行解析。这里用的是json解析。
（4）、请求出错的情况，走这个方法，对错误请求进行处理。

post请求

post请求，也有同步和异步请求。与get方式类似。
post同步请求，也不用于开发，只做了解

post同步请求

ViewController.m

```
// post同步请求
- (IBAction)postT:(id)sender {
    // 1.准备url
    NSString *urlStr = @"http://172.21.35.70/index.php";
    NSURL *url = [NSURL URLWithString:urlStr];
    // 2.准备请求对象
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    [request setHTTPMethod:@"post"];
    // 3.准备参数
    NSString *paramStr = [NSString
stringWithFormat:@"userName=%@&password=
%@",self.userName.text,self.passWord.text];
    // 将参数转为nsdata类型
    NSData *paramData = [paramStr
dataUsingEncoding:NSUTF8StringEncoding];
    // 添加参数
    [request setHTTPBody:paramData];
    // 4.建立链接
    NSData *data = [NSURLConnection
sendSynchronousRequest:request returningResponse:nil
error:nil];
    // 5.解析
    NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:data options:(NSJSONReadingAllowFragments)
error:nil];
    NSLog(@"dict = %@",dict);
}
```

解释：
1、与get请求一样，先获取url
2、创建request请求对象。

3、不同的地方从这里开始，post请求要再这里准备好传过去服务器的参数。因为get是再url里直接给参数传过去，post与之不一样的地方，就是再这里处理参数。这里是将我们要传过去的数据，转化成string类型拼接，然后把这个拼接好的数据，转化成NSData类型对象。`NSData *paramData = [paramStr dataUsingEncoding:NSUTF8StringEncoding];`
4、准备好参数后，直接可以创建连接，用NSData的新对象data来接受传回来的数据
5、直接json解析传回来的数据。

post异步请求

ViewController.m
post异步请求，有delegate和block方式，上面get异步请求写了delegate方式了，这里写的是block方式。

```
// post异步
- (IBAction)postY:(id)sender {
    // 1.准备url
    NSString *urlStr = @"http://172.21.35.70/index.php";
    NSURL *url = [NSURL URLWithString:urlStr];
    // 2.创建请求对象
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    [request setHTTPMethod:@"post"];

    // 3.准备参数 NSOperationQueue多线程队列
    NSString *str = [NSString stringWithFormat:@"userName=
%@&password=%@",self.userName.text,self.passWord.text];
    NSData *param = [str
dataUsingEncoding:NSUTF8StringEncoding];

    [request setHTTPBody:param];
    // 4.建立连接
    [NSURLConnection sendAsynchronousRequest:request queue:
[NSOperationQueue mainQueue]
completionHandler:^(NSURLResponse *response, NSData *data,
NSError *connectionError) {
        NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:data options:(NSJSONReadingAllowFragments)
error:nil];
        NSLog(@"%@",dict);
    }];
}
```

解释：
1、获取url
2、创建request请求对象。
3、准备参数，先把参数拼接成字符串，然后转换成NSData类型对象。
`NSData *param = [str dataUsingEncoding:NSUTF8StringEncoding];`
`[request setHTTPBody:param];`把参数设置成请求主体。
4、建立连接，这里跟post同步请求不一样。第二个参数queue，是NSOperationQueue多线程队列。后面会讲
5、第三个参数就是block，再这个block里对数据进行解析。

—大海
2015年05月19日18:11:20 北京 · 蓝欧

44、异步下载图片

异步图片下载，其实跟上一节课的异步请求几乎一样。只是请求出来的东西不是我们之前各种json格式的字符串，而是图片。

ViewController.m
这里测试是用storyBoard拖出来一个按钮点击事件的响应方法。

```
- (IBAction)buttonAction:(id)sender {
    // 准备url
    NSString *urlStr = @"http://g.hiphotos.baidu.com/image/
pic/item/10dfa9ec8a136327c4c674fc938fa0ec09fac7d2.jpg";
    NSURL *url = [NSURL URLWithString:urlStr];
    // 准备request
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    request.HTTPMethod = @"GET";
    // 创建连接
    [NSURLConnection sendAsynchronousRequest:request queue:
[NSOperationQueue mainQueue]
completionHandler:^(NSURLResponse *response, NSData *data,
NSError *connectionError) {
        // 解析数据
        UIImage *image = [UIImage imageWithData:data];
        self.imageView.image = image;
    }];
   }
```

解释：

1、准备url。

2、创建request请求对象

3、创建连接，这里用的是get异步请求方式。

4、再block中解析数据。`UIImage *image = [UIImage imageWithData:data];`用的是`imageWithData`，来将数据转化成`image`。

因为之后要用到很多次的网络解析，我们为了方便，就把这个网络解析写成一个工具类。到了其他项目直接引入即可.

下面是打包成工具类的过程。

Tool.h

```
#import <Foundation/Foundation.h>
typedef void(^BackDataBlock)(id backData);
@interface Tool : NSObject
+ (void)solveDataWithURLString:(NSString *)urlStr httpMethod:
(NSString *)method httpBody:(NSString *)body backDataBlock:
(BackDataBlock)bdb;
@end
```

创建一个Tool工具类，声明一个类方法。

第一个参数：`urlStr`，请求数据的网址

第二个参数：请求类型：GET？POST?

第三个参数：请求参数（post请求专用）

第四个参数：block块。

原因：请求数据是一个相对于计算机来说，比较长的过程，这个过程就会被放到子线程去完成，而主线程仍然在跑，我们的目的是为了得到请求回来的数据，如果不加block，那么还没有请求数据回来，就已经把data返回去了，这个data就是一个空值。因此，这里要想把请求回来的数据data返回出去，就在参数里加我们自己声明的一个block。参数是id类型的。`typedef void(^BackDataBlock)(id backData);`这样，就能把请求到的数据作为block的参数传出去。在我们需要的地方，就可以直接使用。

Tool.m

```
#import "Tool.h"
@implementation Tool
+ (void)solveDataWithURLString:(NSString *)urlStr httpMethod:
(NSString *)method httpBody:(NSString *)body backDataBlock:
(BackDataBlock)bdb{
    // 准备url
    NSURL *url = [NSURL URLWithString:urlStr];
    // 创建请求对象
    NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:url];
    if ([method isEqualToString:@"POST"]) {
        request.HTTPMethod = @"POST";
        request.HTTPBody = [body
dataUsingEncoding:NSUTF8StringEncoding];
    }
    // 创建链接
    [NSURLConnection sendAsynchronousRequest:request queue:
[NSOperationQueue mainQueue]
completionHandler:^(NSURLResponse *response, NSData *data,
NSError *connectionError) {
        // block传值
        bdb(data);
    }];
}
@end
```

解释：

这个方法里边的代码，与刚才一样。

主要是创建连接这个地方，再系统的block里，调用我们自定义的block，把请求回来的data作为block 的参数，传到外面去。

ViewController.m

```
#import "ViewController.h"
#import "Tool.h"
@interface ViewController ()
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@end
@implementation ViewController
- (IBAction)buttonAction:(id)sender {
    // 自定义方法进行网络请求
    [Tool solveDataWithURLString:@"http://
g.hiphotos.baidu.com/image/pic/item/
10dfa9ec8a136327c4c674fc938fa0ec09fac7d2.jpg"
httpMethod:@"GET" httpBody:nil backDataBlock:^(id backData) {

        UIImage *image = [UIImage imageWithData:backData];
        // 推到主线程执行,不要再子线程更新UI
        // 第一个参数：再主线程里做什么事(执行什么方法)
        // 第二个参数，传进主线程做什么事的那个方法的参数
        // 第三个参数，YES,等主线程完成了，继续执行子线程，NO不用等主
线程完成就执行子线程

        [self
performSelectorOnMainThread:@selector(imageData:)
withObject:image waitUntilDone:YES];
    }];
}
// 要再主线程中执行的方法
- (void)imageData:(id)object{
    self.imageView.image = object;
}
@end
```

解释：

1、点击事件响应方法里，工具类调用类方法，就是我们自定义的类方法。传回来的backData就是请求回来的数据，就可以直接使用了。

2、注意：若是请求回来的数据，是对UI进行更新，就不能把更新UI这一步放在block里，因为block也是再子线程中完成的，若是多个子线程同时再跑，更新UI很可能被滞后。导致页面刷新出bug。因此，要把更新UI推到主线程。

`[self performSelectorOnMainThread:@selector(imageData:) withObject:image waitUntilDone:YES];`（第一个参数，推到主线程，顺带执行的方法，第二个参数，方法需要的参数，第三个参数，YES是主线程方法执行完毕，回来执行子线程，NO是不等主线程方法执行完就自己执行自己线程的其他语句）并实现`imageData:`方法，传过去的参数就是解析好的数据image。对UI进行更新。`self.imageView.image = object;`

45、KVO观察者模式

kvo的意思，是键值观察者，设一种设计模式。

观察者模式的机制是：一个观察者，观察一个对象的某个属性是否发生变化，若果发生变化，就会触发观察者的一个方法。这个方法是一个固定方法。

观察者模式的运用，分为5个步骤：
1、确定观察者是谁（哪个对象）
2、观察哪个对象的什么属性（可以是多个）
3、是谁作为观察者
4、观察到属性改变后，能干什么
5、移除观察

具体如下：创建一个Person类
Person.h（被观察者）

```
#import <Foundation/Foundation.h>
@interface Person : NSObject

@property(nonatomic,copy)NSString *name;
@end
```

声明一个属性，name作为被观察的属性
ViewController.m（观察者）

```
#import "ViewController.h"
#import "Person.h"
@interface ViewController ()
@property (nonatomic,strong)Person *p;
@end

@implementation ViewController
/*
 观察者模式 KVO 4步:

 1、被观察的是谁
 2、观察哪个属性
 3、谁观察
 4、改变后干什么
 5、移除观察。
 */
- (void)viewDidLoad {
    [super viewDidLoad];
    self.p = [[Person alloc]init];
    _p.name = @"网贴纸";
    // 添加监控 KVO
    [_p addObserver:self forKeyPath:@"name" options:
(NSKeyValueObservingOptionNew) context:nil];
// 观察者要执行的事件
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:
(id)object change:(NSDictionary *)change context:(void
*)context{
        self.view.backgroundColor = [UIColor yellowColor];
        // 移除
        [object removeObserver:self forKeyPath:@"name"];
}
- (IBAction)buttonAction:(id)sender {
    _p.name = @"王铁球";
}
@end
```

解释：

1、这里测试是从storyBoard拖一个点击事件响应方法来做测试。

2、再加载视图过程中，对Person对象的属性name赋值_p.name= @"网贴纸"; 并添加监控（KVO），[_p addObserver:self forKeyPath:@"name" options:(NSKeyValueObservingOptionNew) context:nil];
第一个参数：self，指定观察者为viewController自己。一般观察者都是controller;
第二个参数：要观察的Person对象的属性为name（这里用到了KVC）
第三个参数：观察属性的哪方面？这里观察的书name属性的新值
第四个参数：任意类型的指针（C语言）

3、点击事件响应方法里，把_p.name = @"王铁球";重新赋值

4、检测到观察的属性发生改变，就会调用一个固定方法，再这个方法里，做一些事。
第一个参数：键值路径（详细去看KVC）
第二个参数：被观察的属性
第三个参数：一个字典，里边有观察属性的哪方面，这里是观察name的新值
第四个参数：任意类型指针（C语言）

5、每做完一次相应的处理后，要把观察者移除，为的就是保证在对象被释放后，观察者不再继续观察被释放的对象。

46、UITableView异步下载图片

因为tableView是可以滑动的，并且滑出屏幕的cell会被重新加载。因此，请求图片的时候，属于动态请求，动态更新，这里就是用KVO观察者模式来检测哪些cell移出屏幕。然后更新UI

News.h

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
@interface News : NSObject
@property (nonatomic,strong)NSString *image;
@property (nonatomic,strong)NSString *title;
// 解析后存储图片
@property (nonatomic,strong)UIImage *myImage;
// 判断图片是否在加载
@property (nonatomic,assign)BOOL isLoading;
// 加载图片
- (void)loadImage;

@end
```

这里定义一个news类作为model类，声明三个属性，image用来存放请求到的图片路径，title用来存放请求到的标题，isLoading 用来记录是否图片加载完成，myImage用来存放解析请求到的数据的图片。
声明一个加载图片的实例方法：loadImage。

News.m

```
#import "News.h"
#import "Tool.h"
@implementation News
// kvc找不到key，防崩
- (void)setValue:(id)value forUndefinedKey:(NSString *)key{
}
// 加载图片

- (void)loadImage{
    [Tool solveDataWithURLString:_image httpMethod:@"GET"
httpBody:nil backDataBlock:^(id backData) {
        self.myImage = [UIImage imageWithData:backData];
        // 提示加载完成
        _isLoading = NO;
    }];
    // 提示加载中
    _isLoading = YES;
}
@end
```

实现loadImage方法，引入的是上面设计的网络请求的类Tool，请求到数据返回来的backData直接用来解析成图片myImage。并标记是否加载完成。因为block请求需要一定的时间，主线程把_isLoading = YES;先执行，告诉正在加载。再block里标记成NO，加载完成。

TableViewCell.h

```
#import <UIKit/UIKit.h>
@interface TableViewCell : UITableViewCell
@property (weak, nonatomic) IBOutlet UIImageView *imV;
@property (weak, nonatomic) IBOutlet UILabel *label;
@end
```

在storyBoard拖进两个属性。用来存请求下来的数据，放到cell中显示。

TableViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.dataArr = [NSMutableArray array];
    // 网络请求
    [Tool solveDataWithURLString:@"http://
project.lanou3g.com/teacher/yihuiyun/lanouproject/
activitylist.php" httpMethod:@"GET" httpBody:nil
backDataBlock:^(id backData) {
        // json解析
        NSDictionary *dict = [NSJSONSerialization
JSONObjectWithData:backData options:
(NSJSONReadingAllowFragments) error:nil];
        for (NSDictionary *d in dict[@"events"]) {
            // 创建model对象
            News *new = [[News alloc]init];
            // kvc存值
            [new setValuesForKeysWithDictionary:d];
//            NSLog(@"%@",new.image);
            // 放入容器
            [_dataArr addObject:new];
        }
        // 更新数据
        // 再创建cell的时候，需要确定行数 = _dataArr.count，但是是
异步请求再子线程中请求，主线程的视图加载还是继续进行。所以得到的
_dataArr.count = 0，这就需要刷新数据
        [self.tableView reloadData];
        // 打印检验
        for (News *n in _dataArr) {
            NSLog(@"%@ : %@",n.image,n.title);
        }
    }];
}
- (NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView {
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return _dataArr.count;
}
```

在加载视图的时候，进行网络请求（Tool），再block对数据进行JSON解析，然后创建new的model类对象，将解析到的数据存到model类中。

注意：`[self.tableView reloadData];`这里需要每次请求的数据，要进行更新，为了接下来创建`cell`的时候，可以以之前装入数组的`model`的个数来确定`cell`的行数。`return _dataArr.count;`

创建`cell`

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    TableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"cell"
forIndexPath:indexPath];
    News *new = _dataArr[indexPath.row];
    cell.label.text = new.title;
    // 加载图片
    if ((nil == new.myImage) && (new.isLoading == NO)) {
        // 图片为空
        // 临时占位图
        cell.imV.image = [UIImage imageNamed:@"3.png"];
        // 加载图片
        [new loadImage];
        // 添加观察者
        [new addObserver:self forKeyPath:@"myImage" options:
(NSKeyValueObservingOptionNew) context:(__bridge_retained
void *)(indexPath)];
    }else{
        if (new.myImage == nil) {
            cell.imV.image = [UIImage imageNamed:@"3.png"];
        }else{
           cell.imV.image = new.myImage;
        }
    }
    return cell;
}
```

1、创建cell时，需要注册重用字符，这一步已经在storyBoard做好了。这就可以直接创建cell

2、News *new = _dataArr[indexPath.row];获取每一个model类news的对象。

3、cell.label.text = new.title;给cell赋值。

4、if ((nil == new.myImage) && (new.isLoading == NO))判断图片是否加载，如果没有加载，则用本地的占位图来展示代替显示：cell.imV.image = [UIImage imageNamed:@"3.png"];，此时，就需要马上加载网络请求下来的图片来替代临时占位图：[new loadImage];

5、加载完成图片后，马上对其进行监控，：[new addObserver:self forKeyPath:@"myImage" options:(NSKeyValueObservingOptionNew) context:(__bridge_retained void *)(indexPath)]; 也就是添加观察者。
第一个参数：观察者为tableViewController自己，
第二个参数：键值路径，也就是观察的news对象属性
第三个参数：观察属性的哪方面，这里是观察新值
第四个参数：任意类型指针，这里的指针是C语言的语法指针，需要oc做一些处理，把两者指针桥接起来(__bridge_retained void *)(indexPath)，这里本来是直接传indexPath位置信息过去即可，也就是传哪个分区，哪一行发生改变。为的是后面观察者观察的属性改变时调用的方法里cell的位置

6、else（或者有图片，或者图片正在加载），再其内部再分开判断，if (new.myImage == nil)如果对象里没有图片，先用占位图来显示，如果对象里有图片，直接在cell上显示。

观察者监控的值改变

```
-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:
(id)object change:(NSDictionary *)change context:(void
*)context{
    // 1.拿到新值
    UIImage *tempImage = change[NSKeyValueChangeNewKey];
    // 判断为空
    if ([tempImage isEqual:[NSNull null]]) {
        return ;
    }
    // 2.拿到当前显示cell的indexPath
    NSArray *indexArr = [self.tableView
indexPathsForVisibleRows];
    // 3.拿到刚请求到的cell的indexPath。
    NSIndexPath *indexPath = (__bridge NSIndexPath *)context;

    // 4.判断是否正在显示
    if ([indexArr containsObject:indexPath]) {
        // 拿到cell
        TableViewCell *cell = (TableViewCell *)
[self.tableView cellForRowAtIndexPath:indexPath];
        // 换图
        cell.imV.image = tempImage;
        // 更新
        [self.tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:(UITableViewRowAnimationAutomatic)];
    }
    // 移除观察者
    [object removeObserver:self forKeyPath:@"myImage"];
}
```

1、change[NSKeyValueChangeNewKey];（字典根据key取值）拿到改变的新的图片

2、获取当前再屏幕上显示的cell的位置indexPath。（是个数组）

```
NSArray *indexArr = [self.tableView indexPathsForVisibleRows];
```

3、获取正在请求的cell的位置（刚要滑进屏幕的那个cell）：

(__bridge NSIndexPath *)context，把之前传进来的位置，桥接处理后给到这里的indexPath。

4、判断刚要划进屏幕来的cell，是否正在显示，也就是通过 刚才获取的再屏幕上显示的所有的indexPath（数组）是否包含这个传进来的（就是正在请求的indexPath）。

如果包含（也就是正在显示），根据传进来的indexPath获取cell

5、拿到cell了，对它进行换图cell.imV.image = tempImage;

6、更新tableView：

```
[self.tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:(UITableViewRowAnimationAutomatic)];
```

7、移除观察者：[object removeObserver:self forKeyPath:@"myImage"];

47、数据持久化

数据持久化，就是把内存的数据存到本地文件的过程。

iOS沙盒机制

每个iOS应用程序，都有一个沙盒，沙盒其实是一个文件夹，文件夹名字是一串很长的字符，按照一定算法生成的，每一个app都不能读写其他app的这个文件夹，只能读写自己app的文件夹。起保护作用，让app安全。
沙盒下的文件夹：
Document文件夹：存放app运行中生成的文件或者数据库，这些文件会被备份，但是不能存太大音频或者视频等文件，因为上传会被拒。
Library下有俩文件夹：
    Caches：存放缓存文件，如音频，视频，不会备份到iCloud
    Preferences：存放用户的偏好设置文件
tmp：存放临时文件，程序结束时会会清空。

viewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 沙盒

    // 获取用户名
    NSString *userName = NSUserName();
    NSLog(@"-----%@",userName);
    // 获取主路径
    NSString *rootPath = NSHomeDirectoryForUser(userName);
    NSLog(@"%@",rootPath);
    // 另外一种获取主路径的方式
    NSString *rootPath1 = NSHomeDirectory();
    NSLog(@"=====%@",rootPath1);
    // 每个沙盒有下有三个文件夹
    // document作用：存放程序运行中生成的文件或者数据库，存在这里的文件，会被备份（不能存音频视频等大文件，否则上传会被拒）。
    NSString *documentPath = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)[0];
    NSLog(@"++++%@",documentPath);

    // 获取cache路径
    // 作用，存放缓存文件，例如音频视频图片，（不会备份到icloud）
    NSString *cachePath = NSSearchPathForDirectoriesInDomains(NSCachesDirectory, NSUserDomainMask, YES)[0];
    // 获取tmp
    // 作用：存放到临时文件，程序下次启动不需要这些文件，程序结束要清空。
    NSString *tempPath = NSTemporaryDirectory();
    }
```

解释：

获取document文件夹的路径，`NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)[0];`这个方法，获取的是一个数组，取第0个即可

简单对象写入文件

```
- (void)viewDidLoad {
    [super viewDidLoad];
 NSString *documentPath =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES)[0];
    NSLog(@"-----%@",documentPath);
    // 简单对象写入
    NSString *str1 = @"Hello ,World";
    //文件路径
    NSString *filePath = [documentPath
stringByAppendingString:@"/小白.avi"];
    // 写入
    [str1 writeToFile:filePath atomically:YES
encoding:NSUTF8StringEncoding error:nil];
    // 读
    NSString *str2 = [NSString
stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:nil];
   NSLog(@"====%@",str2);
}
```

思路：

准备要写入的数据——获取要写入的位置document文件夹，并创建文件（得到创建的文件路径）——写入文件——读

数组写入

```
NSString *documentPath =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES)[0];

    // 数组写入
    NSArray *arr1 = @[@"123",@"321"];
    // 文件路径
    NSString *filePath = [documentPath
stringByAppendingPathComponent:@"lost revers.mp3"];
    // 写
    [arr1 writeToFile:filePath atomically:YES];
    // 读
    NSArray *arr2 = [NSArray
arrayWithContentsOfFile:filePath];

    NSLog(@"%@",arr2);
```

字典写入

```
    // 字典
    NSDictionary *dict =
@{@"key1":@"value1",@"key2":@"value2"};

    // 获取file
    NSString *filePath = [documentPath
stringByAppendingPathComponent:@"dict.txt"];
    // 写
    [dict writeToFile:filePath atomically:YES];
    // 读
    NSDictionary *dict2 = [NSDictionary
dictionaryWithContentsOfFile:filePath];
    NSLog(@"%@",dict2);
```

复杂对象写入

思路就是把复杂对象转化成简单对象，然后写入.。这里复杂变为简单，就是用到归档和反归档，归档就是将复杂对象转化成NSData类型，然后写入文件中；反归档就是读取出来数据位NSData类型，转化成复杂对象。

Person.h

```
#import <Foundation/Foundation.h>
@interface Person : NSObject<NSCoding>
@property (nonatomic,copy)NSString *name;
@property (nonatomic,assign)NSInteger age;
@end
```

person.m

```
#import "Person.h"
@implementation Person
// 归档：编码
- (void)encodeWithCoder:(NSCoder *)aCoder{
    // p_name p_age为标记
    [aCoder encodeObject:self.name forKey:@"p_name"];
    [aCoder encodeInteger:self.age forKey:@"p_age"];
}
// 反编码，把上面编码的东西组合起来
- (id)initWithCoder:(NSCoder *)aDecoder{
    if (self = [super init]) {
        self.name = [aDecoder decodeObjectForKey:@"p_name"];
        self.age = [aDecoder decodeIntegerForKey:@"p_age"];
    }
    return self;
}

@end
```

ViewController.m

```
    // 归档，反归档
    Person *p1 = [[Person alloc]init];
    p1.name = @"塞班";
    p1.age = 18;
    NSMutableData *personData = [NSMutableData data];
    // 创建归档工具
    NSKeyedArchiver *arciver = [[NSKeyedArchiver
alloc]initForWritingWithMutableData:personData];
    // 开始进行二进制转换
    [arciver encodeObject:p1 forKey:@"p1"];
    // 完成转换
    [arciver finishEncoding];
    NSLog(@"%@",personData);
    // 路径
    NSString *filePath = [documentPath
stringByAppendingPathComponent:@"huahua.avi"];
    // 写
   [personData writeToFile:filePath atomically:YES];

    NSString *filePath = [documentPath
stringByAppendingPathComponent:@"huahua.avi"];

    // 读
    NSData *data = [NSData dataWithContentsOfFile:filePath];
    // 创建反归档工具
    NSKeyedUnarchiver *unarchver = [[NSKeyedUnarchiver
alloc]initForReadingWithData:data];

    // 将二进制转换成对象
    Person *p2 = [unarchver decodeObjectForKey:@"p1"];
    [unarchver finishDecoding];
       NSLog(@"%@",p2.name);
```

更简单的存取（但是每次只能存取一次）

```
    Person *p3 = [[Person alloc]init];
    p3.name = @"王铁柱";
    p3.age = 10;

    NSString *filePath = [documentPath
stringByAppendingPathComponent:@"铁柱"];
    // 存
    [NSKeyedArchiver archiveRootObject:p3 toFile:filePath];
    // 取

    Person *p4 = [NSKeyedUnarchiver
unarchiveObjectWithFile:filePath];
    NSLog(@"%@",p4.name);
```

NSUserDefaults

可作为导航页，根据一个key（这个key我们自定义）判断是否有值，没有值则把这个key和值存入NSUserDefaults对象中。下一次就不会再走里边的添加语句，。

```
    // NSUserDefaults
    // 单例
    NSUserDefaults *ud = [NSUserDefaults
standardUserDefaults];
    // 存
    [ud setObject:@"05芭比" forKey:@"蓝鸥"];
    // 把所有setObject全部存进去
    [ud synchronize];
    // 取
    NSString *str = [ud objectForKey:@"蓝鸥"];
    NSLog(@"%@",str);
     //可以作为导航页..
    if ([ud objectForKey:@"first"] == nil) {
        [ud setObject:@"YES" forKey:@"first"];
        [ud synchronize];
        NSLog(@"第一次..");
    }else{
        NSLog(@"第二次..");
    }
```

NSFileManager

```
// NSFileManager
    // 创建文件夹
    // 1.获取再哪个文件夹下
    NSString *documentPath1 =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES)[0];
    NSFileManager *fileMgr = [NSFileManager defaultManager];
    // 2.创建文件夹
    NSString *testDirectory = [documentPath1
stringByAppendingPathComponent:@"test/123"];
    // 3.创建目录
    [fileMgr createDirectoryAtPath:testDirectory
withIntermediateDirectories:YES attributes:nil error:nil];
    // 创建文件
    NSString *filePath1 = [testDirectory
stringByAppendingPathComponent:@"test11.txt"];
    NSString *content = @"测试内容";
    [fileMgr createFileAtPath:filePath1 contents:[content
dataUsingEncoding:NSUTF8StringEncoding] attributes:nil];
    // 更改文件名
    NSArray *filesName = [fileMgr
subpathsAtPath:testDirectory];
    NSLog(@"%@",filesName);
    NSString *filePath2 = [testDirectory
stringByAppendingPathComponent:@"testChange.txt"];
    [fileMgr moveItemAtPath:filePath1 toPath:filePath2
error:nil];
    // 移动文件位置

    // 删除文件

    // 删除文件夹
    [fileMgr removeItemAtPath:filePath2 error:nil];
    // 判断一个文件是否再文件夹中
```

千年之后的你会在哪里，身边有怎样风景？·韩子

——大海

2015年05月21日23:01:45 北京·蓝欧

48、断点续传

断点续传，一般在下载或者上传较大文件的时候，中途暂停或者断网，导致下

载中断的，需要记录已经下载下来的文件的大小，然后下次启动下载的时候，从断点位置开始，继续下载。
UIProgressView，一般用来显示下载进度的，不能手工改变它的值。很像slider又不是slider

这里我们用UIProgressView来显示我们下载的一首音乐进度。
ViewController.m

```
#import "ViewController.h"
@interface ViewController ()<NSURLConnectionDataDelegate>{
    // 记录整个文件的总大小
    long long _total;
    // 记录当前下载大小
    long long _current;
    // 记录请求下来的数据
    NSMutableData *_reData;
    // 点击状态
    BOOL _flag;
}
@property (nonatomic,strong)NSFileHandle *fileHandle;// 操作文件用的
@property (nonatomic,strong)NSURLConnection *conn;
@property (weak, nonatomic) IBOutlet UIProgressView *progress;
@end
@implementation ViewController
- (IBAction)buttonAction:(UIButton *)sender {
    if (_flag == NO) {
        // 准备url
        NSURL *url = [NSURL URLWithString:@"http://uploads.mp3songurls.com/2930.mp3"];
        // 创建request
        NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
        // 设置参数
        NSString *range = [NSString stringWithFormat:@"bytes:%lld",_current];
        [request setValue:range forHTTPHeaderField:@"Range"];

        // 创建连接

        self.conn = [NSURLConnection connectionWithRequest:request delegate:self];
        // 启动
        [self.conn start];
        [sender setTitle:@"暂停" forState:(UIControlStateNormal)];
        _flag = YES;
    }else{
        // 连接取消
        [self.conn cancel];
        _flag = NO;
        [sender setTitle:@"下载" forState:(UIControlStateNormal)];

    }
}
@end
```

解释：

1、再延展中声明了四个属性，`long long _total;`为记录要下载的文件的总大小，`long long _current;`用来记录已经下载的文件的大小，`NSMutableData *_reData;`用来记录请求下载的数据，这里用可变数据，是为了每一次请求往里加。`BOOL _flag;`用来记录按钮的点击状态。

2、这个测试是用storyBoard拖一个按钮点击事件响应方法来对下载过程中，点击暂停或开始。
3、再按钮响应方法里，网络请求一个mp3文件与网络请求那一节一样，准备url，request，但是设置参数这个地方
[request setValue:range forHTTPHeaderField:@"Range"];设置本次下载数据 再哪个范围。这个range在request中，后面创建连接的时候，会把这个range的范围给到服务器，获取已经下载好的数据范围，然后再从这点开始下载。
4、创建连接，异步请求
self.conn = [NSURLConnection connectionWithRequest:request delegate:self];

```
#pragma mark - 代理方法
- (void)connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response{
    // 获取请求文件总大小
    _total = response.expectedContentLength;
    // 沙盒路径
    NSString *cache =
NSSearchPathForDirectoriesInDomains(NSCachesDirectory,
NSUserDomainMask, YES)[0];
    NSLog(@"%@",cache);
    // 创建文件
    NSFileManager *fileMgr = [NSFileManager defaultManager];
    NSString *filePath = [cache
stringByAppendingPathComponent:@"music.mp3"];
    NSLog(@"%@",filePath);
    [fileMgr createFileAtPath:filePath contents:nil
attributes:nil];
    // 文件句柄
    self.fileHandle = [NSFileHandle
fileHandleForWritingAtPath:filePath];
}
- (void)connection:(NSURLConnection *)connection
didReceiveData:(NSData *)data{
    // 计算当前
    _current += data.length;
    // 指向文件最末尾处
    [self.fileHandle seekToEndOfFile];
    // 写数据
    [self.fileHandle writeData:data];
    // 回到主线程更新UI
    [self
performSelectorOnMainThread:@selector(updateProgress:)
withObject:nil waitUntilDone:YES];
}
- (void)updateProgress:(id)sender{
    self.progress.progress = (long double)_current/(long
double)_total;
}
- (void)connectionDidFinishLoading:(NSURLConnection
*)connection{
}
```

解释：1、设置句柄，是为了操作文件：self.fileHandle = [NSFileHandle fileHandleForWritingAtPath:filePath];与后面的[self.fileHandle writeData:data];（指向文件末尾） 和[self.fileHandle writeData:data];（再文件末尾写入数据）。这就是为什么data用的书可变的NSMutableData的原因。

49、数据库

数据库是数据持久化的一种方式。
我们用的基本上都是关系型数据库，了解就好，不用细问。
基本上手机用的数据库都是sqlite。
在使用sqlite之前，需要引入libsqlite3.dylib库。
数据库操作，基本上就是对数据库里的表进行：增删改查四种形式：
1、增：

```
insert into Lanou05(sname,ssex,sage)values(' 呵呵 ',' 男 ',' 19 ')"
```

其中：Lanou05为表名，第一个括号里，sname,ssex,sage三个字段名（也就是表的各个列的名字）values后面的括号里，是要插入对应字段的值。其余的是sql语句的固定写法。
2、删：

```
delete from Lanou05 where sid=' 1 '
```

其中：Lanou05为表名，where sid = '1'是删除sid = 1的那一行！是sql语句的固定写法。
3、改：

```
update Lanou05 set ssex = '男' where sid = '1'
```

其中：Lanou05为表名 ，将sid=1的行的ssex设置为男。sql的固定写法。
4、查：
全部查：select * from Lanou05
其中Lanou05为表名，*为所有列，为sql固定写法
部分查：

```
select * from Lanou05 where sid = '1'
```

其中：Lanou05为表名，*代表所有列，取得是sid = 1的所有行的值。

Sid	Sname	Sage
1	呵呵	19
2	哈哈	30

如果全查，两行都被取到，如果部分查，取到sid = 1或者sid = 2的那行数据。
根据你给的sid = 多少，或者是其他条件。

DataBaseHandle.h

```
#import <Foundation/Foundation.h>
@class Student;
@interface DataBaseHandle : NSObject
+ (instancetype)shareDataBase;
// 1.打开数据库
- (void)openDb;
// 2.关闭数据库
- (void)closedDb;
// 3.建表
- (void)createTable;
// 4.插入
- (void)insertStudent:(Student *)aStu;
// 5.修改
- (void)updateWithId:(NSInteger)sid sex:(NSString *)ssex;
// 6.删除
- (void)deleteWithId:(NSInteger)sid;
// 7.查找
// 1.全查
- (NSArray *)selectAll;
// 2.条件查找
- (NSArray *)selectWithSex:(NSString *)sex;
@end
```

解释：

1、定义了一个单例类DataBaseHandle，一般数据库操作，都用单例类来操作。

2、声明了若干自定义方法，用于对数据库进行操作。

打开数据库：

DataBaseHandle.m

```
#import "DataBaseHandle.h"
#import <sqlite3.h>
#import "Student.h"
static DataBaseHandle *dataBase = nil;
@implementation DataBaseHandle
+ (instancetype)shareDataBase{
    if (dataBase == nil) {
        dataBase = [[DataBaseHandle alloc]init];
    }
    return dataBase;
}

// 声明一个数据库对象
static sqlite3 *db = nil;
// 获取document路径
- (NSString *)p_documentPath{
   return
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES)[0];
}
// 1.打开数据库
- (void)openDb{
    // 判断是否打开
    if (nil != db) {
        NSLog(@"数据库已经打开..");
        return ;
    }
    // 如果为空，就去打开数据库
    NSString *dbPath = [[self p_documentPath]
stringByAppendingPathComponent:@"stuDB.sqlite"];
    NSLog(@"----%@",dbPath);
    // 如果有这个数据库直接打开，如果没有，创建一个再打开
    // dbPath.UTF8String,把oc字符串转成C
    int result = sqlite3_open(dbPath.UTF8String, &db);
    if (result == SQLITE_OK) {
        NSLog(@"数据库打开成功..");
    }else{
        NSLog(@"数据库打开失败..");
    }
}
```

解释:

1、初始化单例类。+ (instancetype)shareDataBase

2、我们把数据库文件，写在沙盒里的document路径下，命名为stuDB.sqlite

3、static sqlite3 *db = nil;是我们声明的一个数据库对象。

4、如果数据库已经打开，直接返回，跳出方法。

5、创建数据库本地文件stuDB.sqlite，把路径dbPath与数据库对象db的地址关联，直接打开数据库sqlite3_open(dbPath.UTF8String, &db);

6、result == SQLITE_OK的话，说明打开数据库成功，SQLITE_OK的值为0.

关闭数据库sqlite3_close(db)

```
// 2.关闭数据库
- (void)closedDb{
    // 关闭数据库
    int result = sqlite3_close(db);
    if (result == SQLITE_OK) {
        NSLog(@"数据库关闭..");
    }else{
        NSLog(@"数据库关闭失败..");
    }
}
```

关闭数据库很重要，因为会占用资源，但是关闭数据库的位置很重要，视情况而定。

建表

```
// 3.建表
- (void)createTable{
    // 准备sql语句
    NSString *sqlStr = @"create table if not exists Lanou05(sid integer primary key autoincrement not null,sname text,ssex text,sage integer )";

    // 执行sql语句
    // 第一个参数：数据库名字
    // 第三个参数，回调函数..
    int result = sqlite3_exec(db, sqlStr.UTF8String, NULL, NULL, NULL);
    if (result == SQLITE_OK) {
        NSLog(@"键表成功..");
    }else{
        NSLog(@"建表失败..");
    }
}
```

建表的sql语句是：

```
create table if not exists Lanou05
(
   sid integer primary key autoincrement not null,
   sname text,
   ssex text,
   sage integer
)
```

其中：

sid，sname，ssex，sage为表的列名，紧跟再列明后面的是这一列的数据类型。

primary key是主键（主键唯一，不能为空）autoincrement是自增长，不需要用户插入这一列的数据，而是再插入数据的时候，sid自动+1.

sqlite3_exec(db, sqlStr.UTF8String, NULL, NULL, NULL);执行sql语句，第一个参数是数据库名字，第二个参数是sql语句，OC的NSString类型的要转为C语言语句，就需要sqlStr.UTF8String，第三个参数书是回调函数。

注意的是，建表前，要把数据库打开。

插入数据（增）

```
// 4.插入
- (void)insertStudent:(Student *)aStu{
    // 准备sql语句
    NSString *sqlStr = [NSString stringWithFormat:@"insert
into
Lanou05(sname,ssex,sage)values('%@','%@','%ld')",aStu.sname,a
Stu.ssex,aStu.sage];
    // 执行sql语句
    int result = sqlite3_exec(db, sqlStr.UTF8String, NULL,
NULL, NULL);
    if (result == SQLITE_OK) {
        NSLog(@"已插入...");
    }else{
        NSLog(@"插入失败..");
    }
}
```

注意：要在表创建成功后，才能操作。

修改数据

```
// 5.修改
- (void)updateWithId:(NSInteger)sid sex:(NSString *)ssex{
    // 准备sql
    NSString *sqlStr = [NSString stringWithFormat:@"update
Lanou05 set ssex = '%@' where sid = '%ld'",ssex,sid];
    // 执行sql
    int result = sqlite3_exec(db, sqlStr.UTF8String, NULL,
NULL, NULL);
    if (result == SQLITE_OK) {
        NSLog(@"修改成功..");
    }else{
        NSLog(@"修改失败..");
    }
}
```

删除数据

```
// 6.删除
- (void)deleteWithId:(NSInteger)sid{
    // 准备sql
    NSString *sqlStr = [NSString stringWithFormat:@"delete
from Lanou05 where sid='%ld'",sid];
    // 执行sql
    int result = sqlite3_exec(db, sqlStr.UTF8String, NULL,
NULL, NULL);
    if (result == SQLITE_OK) {
        NSLog(@"已经删除..");
    }else{
        NSLog(@"删除失败..");
    }

}
```

全部查询

```
// 1.全查
- (NSArray *)selectAll{
    NSMutableArray *stuArr = nil;
    // sql语句
    NSString *sqlStr = @"select * from Lanou05";
    // 创建一个伴随指针
    sqlite3_stmt *stmt = nil;
    // 预执行
    // -1是最大值，有符号，取反加1
    int result = sqlite3_prepare_v2(db, sqlStr.UTF8String,
-1, &stmt, NULL);
    if (result == SQLITE_OK) {
        stuArr = [NSMutableArray array];// 初始化数组
        // 判断是否还有数据
        while (sqlite3_step(stmt) == SQLITE_ROW) {
            // 取数据
            // 获取sid列，第二个参数是columnId
            NSInteger sid = sqlite3_column_int(stmt, 0);
            // 获取sname列，第二个参数是columnId
            NSString *sname = [NSString stringWithUTF8String:
(const char *)sqlite3_column_text(stmt, 1)];
            // 获取ssex列，第二个参数是columnId
            NSString *ssex = [NSString stringWithUTF8String:
(const char *)sqlite3_column_text(stmt, 2)];
            // 获取sage列，第二个参数是columnId
            NSInteger sage = sqlite3_column_int(stmt, 3);
            // 创建model类对象
            Student *s = [[Student alloc]init];
            s.sid = sid;
            s.sname = sname;
            s.ssex = ssex;
            s.sage = sage;
            [stuArr addObject:s];
        }
    }
    // 关闭伴随指针
    sqlite3_finalize(stmt);
    for (Student *s in stuArr) {
        NSLog(@"name:%@,sex:%@,age:
%ld",s.sname,s.ssex,s.sage);
    }
    return stuArr;
}
```

解释:

1、sqlite3_stmt *stmt = nil;创建一个伴随指针，用来指向数据库表的某一行。一般从第0行开始，随着查询进行，一直往下指。

2、预执行：sqlite3_prepare_v2(db, sqlStr.UTF8String, -1, &stmt, NULL)将数据库、sql语句、伴随指针等关联。其中-1为最大值（有符号数值取反+1）、&stmt为伴随指针地址。

3、while (sqlite3_step(stmt) == SQLITE_ROW)伴随指针指向的如果有其他行，往下执行取出里边的数据。

4、NSInteger sid = sqlite3_column_int(stmt, 0);取的是第0列的数据，sid的列id为0.其余依次加1.把取出来的数据，赋值给我们定义的整型对象。

5、NSString *sname = [NSString stringWithUTF8String:(const char *)sqlite3_column_text(stmt, 1)];取的是第1列的数据sname的所在的列id

为1，数据库取出text类型的数据需要转换成OC语言的NSString类型。
6、创建model对象，把数据保存到model。
7、记得要关闭伴随指针：sqlite3_finalize(stmt);

部分查询

```
// 2.条件查找
- (NSArray *)selectWithSex:(NSString *)sex{
    NSMutableArray *stuArr = nil;
    // 准备sql语句
    NSString *sqlStr = @"select *from Lanou05 where ssex
= ?";
    // 伴随指针
    sqlite3_stmt *stmt = nil;
    // 预执行
    int result = sqlite3_prepare_v2(db, sqlStr.UTF8String,
-1, &stmt, NULL);
    if (result == SQLITE_OK) {
        // 初始化数组
        stuArr = [NSMutableArray array];
        // 绑定问号?
        // 1 是第一个问号，如果有两个问号，写两句，第二句就变成2
        sqlite3_bind_text(stmt, 1, sex.UTF8String, -1, NULL);
        while (sqlite3_step(stmt) == SQLITE_ROW) {
            NSInteger sid = sqlite3_column_int(stmt, 0);
            NSString *sname = [NSString stringWithUTF8String:
(const char*)sqlite3_column_text(stmt, 1)];
            NSString *ssex = [NSString stringWithUTF8String:
(const char *)sqlite3_column_text(stmt, 2)];
            NSInteger sage = sqlite3_column_int(stmt, 3);
            Student *s = [[Student alloc]init];
            s.sid = sid;
            s.sname = sname;
            s.ssex = ssex;
            s.sage = sage;
            [stuArr addObject:s];
        }
    }
    for (Student *s in stuArr) {
        NSLog(@"name:%@,sex:%@,age:
%ld",s.sname,s.ssex,s.sage);
    }
    // 关闭伴随指针
    sqlite3_finalize(stmt);
    return stuArr;
}
```

解释：

1、select *from Lanou05 where ssex = ?，这里的问号代表占位，这个问号占位后，后面赋值的时候，需要绑定。
2、sqlite3_bind_text(stmt, 1, sex.UTF8String, -1, NULL);绑定问号。

长颈鹿的脖子那么长。。。

——大海
2015年05月28日23:25:00 北京·蓝欧

50、集合视图UICollectionView

UICollectionView是一种新的数据展现方式，简单来说可以理解成多行的tableview（这是UICollectionView最简单的方式，因为它由很多种自定义的展现方式）。最简单的UICollectionView就是一个GridView，以多列的形式对数据进行展示。标准的UICollectionView包含三个部分，他们都是UIView的子视图：
cell：用于展示内容的主体，对于不同的cell可以指定不同的宽高和不同的内容。
Supplementary Views ：追加视图，类似于tableview里每个section的header和footer，用来标记每一个section。
Decoration Views：用来装饰视图，这是每一个视图的背景（例如书架、相框等）。

简单实现一个UICollectionView

实现一个UICollectionView与tableview基本上没多大区别。都是dataSource和delegate 设计模式的，前者提供数据源，后者提供样式和用户交互响应。

UICollectionViewDataSource里的方法

1、-numberOfSectionsInCollection:这个事section的数量
2、-collectionView:numberOfItemsInSection:这个方法是确定每个section有多少个item
3、-collectionView:cellForItemAtIndexPath:这个方法是确定的某个位置上显示什么样的cell
实现这三个方法，就可以让UICollectionView实现，但是必须要设置代理：
collectionView.dataSource = self
collectionView.delegate = self;

```
#pragma mark -dataSource
-(NSInteger)numberOfSectionsInCollectionView:
(UICollectionView *)collectionView{
    return 2;
}
- (NSInteger)collectionView:(UICollectionView
*)collectionView numberOfItemsInSection:(NSInteger)section{
    return 200;
}
- (UICollectionViewCell *)collectionView:(UICollectionView
*)collectionView cellForItemAtIndexPath:(NSIndexPath
*)indexPath{
    MyCollectionViewCell *cell = [collectionView
dequeueReusableCellWithReuseIdentifier:@"cell"
forIndexPath:indexPath];
    cell.backgroundColor = [UIColor lightGrayColor];
    cell.myImageView.image = [UIImage imageNamed:@"2.png"];
    return cell;
}
```

解释：
1、创建2个section和每个section创建200个item。
2、创建cell，`MyCollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cell" forIndexPath:indexPath];`重用标志为cell。
根据这之前注册的重用标示符：`[collectionView registerNib:[UINib nibWithNibName:@"MyCollectionViewCell" bundle:nil] forCellWithReuseIdentifier:@"cell"];`
这里注册的cell，是因为在MyCollectionViewCell.xib文件中设置了Identifier的值为cell。
3、`cell.myImageView.image = [UIImage imageNamed:@"2.png"];`在cell上展示图片。

UICollectionViewDelegate的方法

UICollectionViewDelegate代理里的方法，是无关数据的view的外形和用户交互的一些方法。

1、-collectionView:shouldHighlightItemAtIndexPath: 是否应该高亮?
2、-collectionView:didHighlightItemAtIndexPath: 如果1回答为是，那么高亮
3、-collectionView:shouldSelectItemAtIndexPath: 无论1结果如何，都询问是否可以被选中?
4、-collectionView:didUnhighlightItemAtIndexPath: 如果1回答为是，那么现在取消高亮
5、-collectionView:didSelectItemAtIndexPath: 如果3回答为是，那么选中cell

```
// 选中事件
-(void)collectionView:(UICollectionView *)collectionView
didSelectItemAtIndexPath:(NSIndexPath *)indexPath{
    NSLog(@"%@",indexPath);
}
```

关于cell

相比于tableview的cell来说collectionView的cell要逊一些，因为，collectionView的cell，没有各种各样的样式，主要是由展示的对象所决定的。collectionView的cell 展示的数据更多偏向于图像。需求将会是各种各样的，需要自定义。

UICollectionViewLayout

这是collectionView的精髓所在，也是collectionView和tableview的最大不同之处。
Layout决定了collectionView如何显示再界面上，再显示之前，要生成合适的UICollectionViewLayout的对象，并将该对象赋值给collectionView的collectionViewLayout属性。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //集合视图
    // FlowLayout
    // 布局视图用的
    UICollectionViewFlowLayout *layout =
[[UICollectionViewFlowLayout alloc]init];
    // 每个item大小
    layout.itemSize = CGSizeMake(100 , 100);
    // 设置行距
    layout.minimumLineSpacing = 20;
    // 设置左右间距
    layout.minimumInteritemSpacing = 20;
    // 设置滚动方向
    layout.scrollDirection =
UICollectionViewScrollDirectionHorizontal;
    // 内边距
    layout.sectionInset = UIEdgeInsetsMake(20, 10, 10, 10);
    // header
    layout.headerReferenceSize = CGSizeMake(100, 100);
    // 创建collectionView
    UICollectionView *collectionView = [[UICollectionView
alloc]initWithFrame:self.view.bounds
collectionViewLayout:layout];
    // 数据源和代理
    collectionView.dataSource = self;
    collectionView.delegate = self;
    // 添加到视图
    // 默认黑色
    collectionView.backgroundColor = [UIColor yellowColor];
    [self.view addSubview:collectionView];
    // 注册cell
//    [collectionView registerClass:[UICollectionViewCell
class] forCellWithReuseIdentifier:@"cell"];

    [collectionView registerNib:[UINib
nibWithNibName:@"MyCollectionViewCell" bundle:nil]
forCellWithReuseIdentifier:@"cell"];
}
```

只需区别cell 的注册即可。

1、确定itemSize，确定每一个item的大小，如果想改变某个cell的大小，可以用方法-collectionView:layout:sizeForItemAtIndexPath:

2、layout.minimumLineSpacing = 20;每行的距离不小于20

3、layout.minimumInteritemSpacing = 20;每个item左右间距不小于20

4、设置滚动方向，这里是左右滚动，默认上下滚动：layout.scrollDirection = UICollectionViewScrollDirectionHorizontal;

设置滚动方向后，会看到我们的行距或者间距的实际距离。

5、layout.sectionInset = UIEdgeInsetsMake(20, 10, 10, 10);设置内

边距，对屏幕四边的距离。
6、设置完毕layout后就可以创建collectionView了。

```
UICollectionView *collectionView = [[UICollectionView alloc] initWithFrame:self.view.bounds collectionViewLayout:layout];
```

My tea's gone cold I'm wondering why I got out of bed at all .The morning rain clouds up my window and I can't see at all. And even if I could it'll all be grey, put your picture on my wall. It reminds me that it's not so bad

——大海
2015年05月29日21:37:32 北京·蓝鸥

51、多线程

程序：就是代码生成的可执行的应用。
进程：跑起来的程序。
线程：程序中独立运行的代码段。（block就是）。
一个进程是由多个线程（一个以上）组成的。进程申请资源和分配资源给线程用，线程用到资源。

每个进程都包含至少一个线程，即主线程。主线程再程序跑动的时候被创建，用于执行main函数里的东西。

主线程负责的是，所有代码的执行，包括UI更新，网络请求，本地存储等等。
主线程的代码都是顺序执行，不能并行执行。

用户可以开辟许多新的线程，这些线程都是主线程的子线程。
注意的是，主线程和子线程都是相互独立执行代码的，互不影响，并发执行。

注意：UI更新，必须放在主线程中去更新。因为开辟的子线程，是不受控制的，什么时候执行完毕我们无法控制。再子线程更新UI出现的bug无法预料。

NSThread

NSThread是相对于后面两个NSOperationQueue、GCD来说，是一个轻量级的多线程。

创建NSThread：

```
    // NSThread
    // 1.
    NSThread *t1 = [[NSThread alloc]initWithTarget:self
selector:@selector(threadAction:) object:nil];
    // 手动开启线程
    [t1 start];
    // 结束子线程
    [t1 cancel];

    // 2.不需要手动开启子线程
                                                    [NSThread
detachNewThreadSelector:@selector(threadAction:)
toTarget:self withObject:@"123456789"];
```

第一种创建方式：需要手动启动[t1 start];也要手动结束[t1 cancel];子线程。
第一个参数：随来执行
第二个参数：去执行什么方法？这个方法里一般是子线程要完成的一些事：（网络请求，或者其他）。
第三个参数：去执行的那个方法，顺便带一个参数过去。
第二种创建方式：不需要手动去开启子线程。参数意义与第一种一样。

注意：

再多线程方法中，需要添加自动释放池。

应用程序再启动时，系统自动为主线程创建了自动释放池。我们手动添加的子线程，需要添加自动释放池。

NSOperation

NSOperation

再mvc中属于M层，用来封装单个任务的相关代码的一个抽象类。这个抽象类不能直接使用，而是用它的子类来执行实际任务。

NSOperation只是一个操作，本身并没有主线程、子线程之分，可以使用再任意线程中。通常与NSOperationQueue一起使用。

NSInvocationOperation

```
    NSInvocationOperation *inOp = [[NSInvocationOperation
alloc]initWithTarget:self
selector:@selector(operationAction:) object:nil];
    [inOp start];
```

NSInvocationOperation封装了target和action，即是谁（self）去执行什么方法(operationAction:)。
第三个参数：去执行operationAction:方法顺带过去的参数。

NSBlockOperation

```
    NSBlockOperation *blockOp = [NSBlockOperation
blockOperationWithBlock:^{
        NSLog(@"犯我德邦者，虽远必诛..");
    }];
    [blockOp start];
```

封装了要执行的代码块。要执行的代码都放在block里执行了。

NSOperationQueue

NSOperationQueue实际上是一个操作队列，用来管理一组NSOperation对象的执行。
NSOperationQueue会根据NSOperation的需要开辟适量的线程，保证任务的并行执行。
NSOperation也可以调节在队列的优先级。

```
// 创建执行队列
    NSOperationQueue *queue = [[NSOperationQueue alloc]init];
    // 当前最大操作执行队列
    queue.maxConcurrentOperationCount = 2;
    // 依赖关系,前者等到后者执行完毕后才会执行
//    [inOp addDependency:blockOp];
    [blockOp addDependency:inOp];
    // 添加操作
    [queue addOperation:inOp];
    [queue addOperation:blockOp];
```

1、queue.maxConcurrentOperationCount = 2;当值设置为1的时候，就是线程同步执行。也就是一个线程执行完毕，才能执行下一个线程。
2、inOp和blockOp都是上面NSOperation子类创建的对象。
3、[queue addOperation:inOp];把任务添加到队列。

NSObject

只需要记住一个方法即可：
- (void)performSelectorInBackground: withObject:
再后台执行某个方法，并顺带给一个参数。

GCD

GCD是苹果公司封装好的一个系统，用来应对多核处理系统。gcd大部分代码都是函数的多线程，性能高，功能强大。
GCD以队列的方式进行工作，先进先出（fifo），GCD根据分发队列的的类型，创建适量线程，完成工作。
cgd分为三种队列：
主队列，全局队列，自定义队列。
1、串行队列：前一个任务完成，才能执行下一个任务（包括主队列和自定义队列）
2、并行队列：各个任务独立执行，互不干扰，也是先进先出（包括了全局队列和自定义队列）。

主队列（串行队列）

```
// 这是一个类型,c语言函数，返回一个主队列
      // 主队列（串）
    dispatch_queue_t mainQueue = dispatch_get_main_queue();
    // 添加任务(三种队列都用这个方法添加)
    dispatch_async(mainQueue, ^{
        NSLog(@"第1个任务，当前线程：%@",[NSThread currentThread]);
    });
    dispatch_async(mainQueue, ^{
        NSLog(@"第2个任务，当前线程：%@",[NSThread currentThread]);
    });
    dispatch_async(mainQueue, ^{
        NSLog(@"第3个任务，当前线程：%@",[NSThread currentThread]);
    });
    dispatch_async(mainQueue, ^{
        NSLog(@"第4个任务，当前线程：%@",[NSThread currentThread]);
    });
    dispatch_async(mainQueue, ^{
        NSLog(@"第5个任务，当前线程：%@",[NSThread currentThread]);
```

解释：
1、dispatch_queue_t mainQueue，是创建一个对象（暂且说是对象），dispatch_queue_t是一个类型，mainQueue是类型的实例名。
2、dispatch_get_main_queue()，获取主队列
3、dispatch_async（mainQueue，^(block)），这个方法是吧任务添加到mainQueue主队列中。block块里是要执行的任务。
4、主队列是串行队列。

自定义队列（串行队列）

```
    // 自定义队列（串）
    // 第二个参数，告诉是串行的。
    dispatch_queue_t myQueue1 = dispatch_queue_create("myQ1",
DISPATCH_QUEUE_SERIAL);
    dispatch_async(myQueue1, ^{
        NSLog(@"第1个任务，当前线程：%@",[NSThread
currentThread]);
    });
    dispatch_async(myQueue1, ^{
        NSLog(@"第2个任务，当前线程：%@",[NSThread
currentThread]);
    });
    dispatch_async(myQueue1, ^{
        NSLog(@"第3个任务，当前线程：%@",[NSThread
currentThread]);
    });
```

解释：

1、dispatch_queue_create是创建自定义队列的方式。第一个参数是队列标示符，第二个参数是告诉系统，这个队列是串行的。
2、dispatch_async，添加任务到myQueue1这个自定义队列中。block块中是执行的任务。

全局队列（并行队列）

```
    // 全局（并行）
    dispatch_queue_t globelQ =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0);
    dispatch_async(globelQ, ^{
        NSLog(@"第1个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(globelQ, ^{
        NSLog(@"第2个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(globelQ, ^{
        NSLog(@"第3个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(globelQ, ^{
        NSLog(@"第5个任务，当前线程为：%@",[NSThread
currentThread]);
    });
```

解释
dispatch_get_global_queue创建一个全局队列。
DISPATCH_QUEUE_PRIORITY_DEFAULT：队列的优先级，default是系统默认。

自定义队列（并行队列）

```
    // 自定义（并行）
    dispatch_queue_t myQueue2 = dispatch_queue_create("Q2",
DISPATCH_QUEUE_CONCURRENT);

    dispatch_async(myQueue2, ^{
         NSLog(@"第1个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(myQueue2, ^{
        NSLog(@"第2个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(myQueue2, ^{
        NSLog(@"第3个任务，当前线程为：%@",[NSThread
currentThread]);
    });
    dispatch_async(myQueue2, ^{
        NSLog(@"第4个任务，当前线程为：%@",[NSThread
currentThread]);
    });
```

解释：
dispatch_queue_create创建自定义队列。第一个参数是队列标示符，第二个参数：DISPATCH_QUEUE_CONCURRENT是告诉系统，创建的是并行的队列。

延迟执行

```
    // 延迟执行
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)
(3 * NSEC_PER_SEC)), dispatch_get_main_queue(), ^{
        NSLog(@"等了3秒..");
    });
```

类似于NSTimer那一节的用法，让程序暂停一段时间再执行。
重复执行：

```
    // 重复执行
    dispatch_apply(10,
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0), ^(size_t t) {
        NSLog(@"%zu",t);
    });
```

给多少数值，就重复执行多少次这个任务。

重写单例handle

```
#import "MyHandle.h"
static MyHandle *handle;
@implementation MyHandle
+ (instancetype)shareHandle{
//    if (nil == handle) {
//        handle = [[MyHandle alloc]init];
//    }
    // 这段代码在整个程序只执行一次
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        handle = [[MyHandle alloc]init];
    });
    return handle;
}
@end
```

解释：

1、dispatch_once，打出这个方法的时候，会自动把static dispatch_once_t onceToken;补上，这段代码，也就是dispatch_once这个方法，只执行一次，整个程序。

dispatch_sync()与dispatch_async()的区别：

两者都是将任务添加到队列中，但是前者再block没有执行完，是不会执行后面的代码，后者是添加到任务后立即返回来执行下边的代码。

Dispatch Group

有时候我们操作一个任务，但是操作这个任务，需要等待其他一些任务完成后才能进行。此时我们就用Dispatch Group来把这些要等待的任务装起来：

```
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, ^{
   // Some asynchronous work
});
```

dispatch_group_async跟dispatch_async一样，会把任务放到queue中执行，不过它比dispatch_async多做了一步操作就是把这个任务和group相关联。

把一些任务放到Dispatch Group后，我们就可以调用dispatch_group_wait来等待这些任务完成。若任务已经全部完成或为空，则直接返回，否则等待所有任务完成后返回。注意：返回后group会清空。

```
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
dispatch_release(group);  (MRC下)
```

线程间通信

主线程进入子线程：

NSThread、NSInvocationOperation、NSBlockOperation、GCD等

子线程返回主线程

```
GCD：dispatch_get_main_queue()
NSObject：-  (void)performSelectorOnMainThread:(SEL)aSelector
withObject:(id)arg waitUntilDone:(BOOL)wait
```
更新UI一般再这个地方执行。

线程互斥

当多个线程同时访问同一资源时，会造成数据出错（脏读）。线程互斥就是限制多个线程同时访问一个资源，NSLock就是将资源加锁。加锁就是再一个线程访问资源的时候，加锁，限制其他线程访问资源，当线程访问资源结束，就会解锁，把资源空出来，让给需要的线程。

52、动画