Table of Contents

Intro 1.1

前言 1.2

介绍 1.3

controller 1.4

- AlertController 1.4.1
- NavigationController 1.4.2
- PageViewController 1.4.3
- TabBarController 1.4.4

view 1.5

- ActivityIndicatorView 1.5.1
- CollectionView 1.5.2
- ImageView 1.5.3
- PickerView 1.5.4
- ProgressView 1.5.5
- ScrollView 1.5.6
- SearchBar 1.5.7
- TableView 1.5.8
- textview 1.5.9
- webview 1.5.10

layout 1.6

- stackview 1.6.1
- ImageView 1.6.2
- ImageView 1.6.3

todoapp 1.7

storage 1.8

- sqlite 1.8.1
- coredata 1.8.2
- plist 1.8.3
- archive 1.8.4
- sqlite 1.8.5

pod 1.9

http 1.10

- readyhttpserver 1.10.1
- selfsignedhttps 1.10.2

介绍全书

%Swift iOS开发小书 %刘传君

前言

在使用Swift语言多次完成iPhone App开发的历程中，我感受到了很多美好的事物。当使用Swift的对象字面量来编写复杂数据结构时，完全可以体会到类似使用脚本的强大表达能力。我使用UIKit可以做成炫酷的、快速用户响应的界面。

然而，作为一个程序员，我本人对Xcode的StoryBoard并不接受，即使官方文档和很多同行都推荐使用这样方式。使用它编写App，显著提高了我的学习成本，因为编写的教程都是长篇的操作流程。这让我觉得我被变成了一个操作员了。

所以这本书是讲解UIKit的，会涉及到AppDelegate、View、ViewController等架构类和具体的UIButton等视图类、以及布局类。但是同时，也是表达我的一个主张。我希望细节都在代码中，无需到操作和IDE内部去寻求；并且这也意味着任何App的修改，我可以使用Xcode，也可以使用我偏爱的编辑器，比如Sublime Text。不仅仅我这么想，我在使用Google研究iOS开发时，常常发现有些人会在提问或者文章的标题中加上一个programmatically，表示一点点的抗议和自我表达。

作为读者，这样的主张下著述的资料对你是有好处的：

1. 你可以最小步骤的把本书的代码跑起来。使用Xcode创建一个Single View App，然后把代码贴到AppDelegate.swift文件内，然后Run就是了
2. 你不必看这些巨大的啰嗦的操作步骤，然后费力的在IDE内寻找对应的菜单、按钮、执行拖放等类似Photoshop操作员的工作
3. 因为无需截图、也基本不太需要操作步骤，因此文章篇幅会缩小，这样可以降低你的阅读压力

编程类的书，核心在代码。你可以拿到本书，首先把代码跑起来，这是重要一步，说明我的代码是正确的。然后才去看代码，如果代码看得懂，后面的文字就不必看了。这也是一个快速学习的方法，也是一种识别学习资料质量的方式。

因为官方主张使用Storyboard，所以我的想法和官方是背道而驰的，因此做起来其实颇有难度。第一次尝试是在2016年1月时，但是几个月后就不得不放弃。等到快一年了再重回此主题，我发现如有神助，因此发奋写作完成本书并分享给你。

刘传君 2017年03月17日

献给

三月里在三圣乡盛放的那朵小花儿

作者

刘传君。创建过产品，创过业。好读书，求甚解。可以通过1000copy#gmail.com联系到我

出品

Vue.js小书 http://www.ituring.com.cn/book/1956

最小的App

如何通过纯粹的代码来完成一个App？可以的。并且只要阅读一页文字就可以把代码运行起来。

首先做些操作性的工作，具体过程是这样的：

1. 打开Xcode
2. 创建一个Single View App

Xcode会进入主要编辑界面。此时，我们打开AppDelegate.swift文件然后覆盖源代码为如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = UIViewController()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
}
```

运行后发现手机（或者仿真器）满屏的蓝色背景就说明成功了。

一起来看看代码。我们需要导入UIKit：

```
import UIKit
```

框架UIKit用来构造和管理用户界面、响应用户交互和系统事件。既然需要创建App，引入UIKit是第一件需要做的工作。

接着查看AppDelegate类。它实现协议UIApplicationDelegate，且必须被标记为@UIApplicationMain，两者配合就设置了App的程序入口点。AppDelegate实现了此协议中规定的函数：

```
 func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
```

这样，UIKit就会在完成App启动调用此函数，从而App可以再次实现自己的初始化逻辑。

在本案例中，我们在启动函数中创建了window实例，它是UIWindow的实例。代码随即把window属性的rootViewController设置为UIViewController的实例。然而UIViewController是什么？此类可以提供管理一组视图的基础架构，目前为止，你可以仅仅知道UIViewController其内的view属性就是一个视图，此视图就是一个可见的矩形区域，并且可以对它设置属性、监听它的事件即可。为了可视化的看到我们的工作成果，我们设置UIViewController.view的背景色为蓝色。这样，我们完成了最基本的一个UI界面。

UIWindow和UIView是构建UI的关键构造块，因此有必要澄清下概念。每个应用都有一个window，这个window并没有任何外观形象，每个view都是被window包含在内的，事件处理也是首先路由到相应的window对象然后被导向到对应的视图。View定义了一个屏幕上的矩形区域，并且管理此区域的内容和用户交互。window通过一个rootViewController的属性view开始关联，然后在此view内可以通过addSubView加入更多的子视图来完成UI的构建。

随后我们会接触更多的UIView类型。实际上按钮（UIButton）、标签（UILabel）等可以看到和交互的控件都是视图的子类,UIView的使用是非常广泛的。

可能很多人会觉得AppDelegate这个名字过于冗长。为此我们得了解一个叫做Delegate的设计模式。Delegate是一个简单而强大的设计模式，可以让其中一个对象委托一部分功能给另外一个对象。做法就是发起对象持有委托对象的实例，并在适当的时间发起消息给委托对象，告诉委托对象将要发生的事件或者已经发生的事件。委托对象可以更新用户界面或者更新自己的状态作为响应。在此案例中，关于委托牵涉到了两个类包括UIApplication、AppDelegate，以及一个协议为UIApplicationDelegate，UIApplication委托事件给AppDelegate，这里的委托事件就是：

```
func application(_ application: UIApplication, didFinishLa
unchingWithOptions launchOptions: [UIApplicationLaunchOpti
onsKey: Any]?) -> Bool
```

以便让在UIApplicationDelegate的对象可以完成自己的初始化代码。UIApplication和UIApplicationDelegate之间可以建立一个协议，此协议规定两者之间可以委托的具体内容，此协议就是UIApplicationDelegate规定的，可以通过手册了解此协议的更多资料。

这样，我们了解到了建造一个App涉及到的最少的类：UIApplication、UIApplicationDelegate、UIWindow、UIViewController，以及UIKit大量使用的委托设计模式，未来的文字将会以此为垫脚石。

View Controller

我们已经了解了类UIViewController，它被称为视图控制器，被用来装入和释放视图、管理视图交互、并且和其他视图控制器一起协作完成整体的App界面。为了术语一致，后文会直接使用它的英文名：View Controller。

View Controller管理一个视图层级体系。比如在一个视图控制器内加入一个按钮视图和一个标签视图，分别名为button、label，那么层次系统如下：

```
    ViewController:UIViewController
        view:UIView
            button:UIButton
            label:UILabel
```

本文引入一个微小的，但是可以和用户交互的案例，来演示View Controller的视图管理能力。基于前文提到的脚手架，代码如下：

1. 界面包括一个按钮和一个标签，标签初始值为0
2. 当点击按钮时，标签的数字会被累加1

代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
```

```
unds)
            let page = Page1()
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page1: UIViewController {
        var count = 0
        var label : UILabel!
        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.backgroundColor = .white
            label   = UILabel()
            label.frame = CGRect(x: 100, y: 100, width: 20
, height: 50)
            label.text =  "0"
            view.addSubview(label)
            let button   = UIButton(type: .system)
            button.frame = CGRect(x: 120, y: 100, width: 2
0, height: 50)
            button.setTitle("+",for: .normal)
            button.addTarget(self, action: #selector(Page1
.buttonAction(_:)), for: .touchUpInside)
            view.addSubview(button)
        }
        func buttonAction(_ sender:UIButton!){
            self.count +=  1
            label.text =  self.count.description
        }
    }
```

编译运行后会看到界面上的按钮和标签，点击按钮标签的值加1，说明App满足我们的最初需求。

代码解释下：

1. UIViewController类内属性view可以把其他view加入其内
2. 按钮的类为UIButton ,可以通过属性frame设置位置和大小，可以通过UIViewController.view对象的方法addSubview把按钮加入到UIViewController内
3. 标签的类为UILabel，可以通过属性frame设置位置和大小，可以通过UIViewController.view对象的方法addSubview把按钮加入到UIViewController内
4. button可以添加事件，通过方法：

```
   button.addTarget(self, action: #selector(Page1.butto
nAction(_:)), for: UIControlEvents.touchUpInside)
```

View Controller内部根据管理的对象的不同，分为两种：

1. Content View Controller ，内容型
2. Container View Controller，容器型

两者的不同之处在于：Container View Controller不但可以管理层次化视图，还可以管理其他的View Controller，从中取得部分视图来嵌入自己的整体界面。Container View Controller需要设置嵌入视图的位置和大小，而原本所在的View Controller管理视图的内容和事件。

Container View Controller存在的目的是为了更好的做View Controller之间的导航，比如UINavigationController, UITabBarController就是典型的Container View Controller。

如下案例演示一个View Controller作为Container View Controller嵌入另外一个View Controller到自己的用户界面内：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
```

```
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page1()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page1: UIViewController{
        var c : UIButton!
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UIButton()
            c.setTitle("Page1",for: .normal)
            c.frame = CGRect(x: 10, y: 50, width: 200, hei
ght: 50)
            view.addSubview(c)
            c.backgroundColor = .blue
            let page2 = Page2()
            self.addChildViewController(page2)
            page2.view.frame = CGRect(x: 10, y: 100, width
: 200, height: 50)
            view.addSubview(page2.view)
        }
    }
    class Page2: UIViewController{
        var c : UIButton!
        var count = 0
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UIButton()
            c.backgroundColor = .red
            c.frame = CGRect(x: 0, y: 0, width: 200, heigh
t: 50)
```

```
            c.setTitle("0",for: .normal)
            c.addTarget(self, action: #selector(buttonActi
on(_:)), for: .touchUpInside)
            view.addSubview(c)
        }
        func buttonAction(_ sender:UIButton!){
            self.count +=  1
            c.setTitle(self.count.description,for: .normal
)
        }
    }
```

本案例涉及到2个View Controller类，分别为Page1、Page2。Page2中有一个按钮，点击时会给按钮的标题的数字加1。而Page1作为Root View Controller，嵌入到Window内，占据全部屏幕大小，它内嵌一个按钮和Page2——也就是另外一个View Controller。为了让Page2的视图可以显示在Page1内，需要把Page2.view属性作为子视图加入到Page1.view内，并且通过：

```
    self.addChildViewController(page2)
```

把Page2作为Page1的Child View Controller，这样就可以重用Page2的已有的逻辑。

作为好奇的程序员，不妨试试删除这一行代码，你会发现在点击Page2的按钮时，数字不会增加，因此说明当此代码行被删除，因此两个View Controller并没有任何关系时，它的操作代码不能被执行到。

一般很少直接使用UIViewController，更多的是直接使用特定目的的View Controller，也就是UIViewController的子类。常用的特定目的的类有这些：

1. AlertController，警示框控制器
2. NavigationController，导航控制器

3. PageViewController，分页控制器
4. TabBarController，标签页控制器

后文会继续介绍。

AlertController

UIAlertController用来向用户显示一个警告信息。

演示此类的App代码可以在装入后立刻显示一个警告信息，代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page:UIViewController{
        override func viewDidAppear(_ animated: Bool) {
            Alert()
        }
        func Alert(){
            let alert = UIAlertController(title: "title",
message: "message", preferredStyle:.alert)
            alert.addAction(UIAlertAction(title: "OK", sty
le: .default, handler: { (action: UIAlertAction!) in
                print("OK")
            }))
            alert.addAction(UIAlertAction(title: "Cancel",
 style: .default, handler:nil))
```

```
            self.present(alert,animated: true,completion:
nil)
        }
        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.backgroundColor = UIColor.black
        }
    }
```

UIAlertController可以通过

```
     UIAlertController(title: "title", message: "message",
 preferredStyle:.alert)
```

指定警告信息的标题、正文和样式。可以通过类似如下的代码：

```
       alert.addAction(UIAlertAction(title: "OK", style: .
default, handler: { (action: UIAlertAction!) in
                print("OK")
            }))
```

方法创建一个操作，此操作会以按钮形式出现在警告信息框内。点击操作按钮后要执行的代码以匿名函数的形式通过handle参数添加。

UIAlertController是UIViewController的子类因此可以使用：

```
    present(_:animated:completion:)
```

来调用。刚刚提到的UIAlertController的样式preferredStyle，除了.alert外，还有.actionSheet，后者可以在屏幕下方显示一个类似菜单的选择器，每个选项都可以单独触控并执行代码。可以使用如下的代码来了解它的创建和使用的方法：

```
    class Page:UIViewController{
        override func viewDidAppear(_ animated: Bool) {
            ActionSheet()
        }
        func ActionSheet() {
            let sheet: UIAlertController = UIAlertControll
er(title:nil, message:nil, preferredStyle:UIAlertControlle
rStyle.actionSheet)
            sheet.addAction(UIAlertAction(title:"Do someth
ing 1", style:UIAlertActionStyle.default, handler:{ action
 in
                print ("Do something 1")
            }))
            sheet.addAction(UIAlertAction(title:"Do someth
ing 2", style:UIAlertActionStyle.default, handler:{ action
 in
                print ("Do something 2")
            }))
            sheet.addAction(UIAlertAction(title:"Cancel",
style:UIAlertActionStyle.cancel, handler:nil))
            self.present(sheet, animated:true, completion:
nil)
        }
    }
```

和.alert的使用方法是非常类似的。

NavigationController

控制器NavigationController常用于用来做层次化UI导航。类名为UINavigationController。

NavigationController可以通过属性包含多个ViewController、一个UINavigationBar、一个可选的UIToolbar。

以一个共三层的ViewController的案例，来展示NavigationController的使用：

1. 共三层层次化UI
2. 每一级页面内有一个按钮，可以继续导航到下一级页面
3. 每一个页面的导航条的左侧按钮可以返还到上一级

代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        var nav : Nav?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            nav = Nav()
            self.window!.rootViewController = nav
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Nav: UINavigationController {
```

```
        var count = 0
        var label : UILabel!
        override func viewDidLoad() {
            super.viewDidLoad()
            self.view.backgroundColor = .white
            self.pushViewController(Level1(), animated: tr
ue)
        }
    }

    class Level1: UIViewController {
        var count = 0
        var label : UILabel!
        override func viewDidLoad() {
            super.viewDidLoad()
            self.title = "Level 1"
            self.view.backgroundColor = .white
            let button   = UIButton(type: .system)
            button.frame = CGRect(x: 120, y: 100, width: 1
00, height: 50)
            button.setTitle("Dive Into 2",for: .normal)
            button.addTarget(self, action: #selector(Level
1.buttonAction(_:)), for: .touchUpInside)
            view.addSubview(button)
        }
        func buttonAction(_ sender:UIButton!){
            let appDelegate = UIApplication.shared.delegat
e as! AppDelegate
            appDelegate.nav?.pushViewController(Level2(),
animated: true)
        }
    }

    class Level2: UIViewController {
        var count = 0
        var label : UILabel!
```

```
        override func viewDidLoad() {
            super.viewDidLoad()
            self.title = "Level 2"
            self.view.backgroundColor = .white
            let button   = UIButton(type: .system)
            button.frame = CGRect(x: 120, y: 100, width: 1
00, height: 50)
            button.setTitle("Dive Into 3",for: .normal)
            button.addTarget(self, action: #selector(Level
2.buttonAction(_:)), for: .touchUpInside)
            view.addSubview(button)
        }
        func buttonAction(_ sender:UIButton!){
            let appDelegate = UIApplication.shared.delegat
e as! AppDelegate
            appDelegate.nav?.pushViewController(Level3(),
animated: true)
        }
    }
    class Level3: UIViewController {
        var count = 0
        var label : UILabel!
        override func viewDidLoad() {
            super.viewDidLoad()
            self.title = "Level 3"
            self.view.backgroundColor = .white
            let button   = UIButton(type: .system)
            button.frame = CGRect(x: 120, y: 100, width: 1
00, height: 50)
            button.setTitle("End",for: .normal)
            button.addTarget(self, action: #selector(Level
3.buttonAction(_:)), for: .touchUpInside)
            view.addSubview(button)
        }
        func buttonAction(_ sender:UIButton!){
```

```
        }
    }
```

解释下：

1. 把Nav作为UINavigationController的子类，实例化此类并设置到window.rootViewController上
2. 每一层都是继承于UIViewController，类名分别为Level1、Level2、Level3
3. Nav类视图加载完成后，把第一层Level1压入导航视图
4. 无需任何用户代码，UINavigationController本身默认提供页面的顶部条状区域，它被称为导航条(UINavigationBar)
5. 点击此导航条的左侧按钮可以返还到上一级页面；导航条中间显示当前ViewController的title属性值

PageViewController

类PageViewController也是一个特殊的UIViewController，它可以管理多个页面，每个页面都是一个UIViewController，可程序控制或者用户手势来切换页面。

如下代码，展示了PageViewController的使用：

```
import UIKit
class VCBase: UIViewController
{

    var label : UILabel?
    override func viewDidLoad()
    {
        super.viewDidLoad()
        label = UILabel(frame: CGRect(x: 0, y: 0, widt
h: view.frame.width, height: 200))
        label!.textAlignment = .center
        view.addSubview(label!)
        view.backgroundColor = UIColor.white
    }
}
class Page1: VCBase
{
    override func viewDidLoad()
    {
        super.viewDidLoad()
        label!.text = "#1"
    }
}
class Page2: VCBase
{
    override func viewDidLoad()
```

```
        {
            super.viewDidLoad()
            label!.text = "#2"
        }
    }
    class PageViewController :UIPageViewController,UIPageV
iewControllerDataSource{
        var vcs :[UIViewController]
        required init(){
            vcs = [Page1(),Page2()]
            super.init(transitionStyle: .scroll, navigatio
nOrientation: .horizontal, options: nil)
        }
        required  init?(coder: NSCoder){
            fatalError()
        }
        override func viewDidLoad() {
            self.dataSource = self
            setViewControllers( [vcs[0]], direction: .forw
ard, animated: false, completion: nil)
        }
        func pageViewController(_ pageViewController: UIPa
geViewController, viewControllerBefore viewController: UIV
iewController) -> UIViewController?
        {
            let v = (viewController)
            let i = vcs.index(of: v)! - 1
            if i < 0 {
                return nil
            }
            return vcs[i]
        }
        func pageViewController(_ pageViewController: UIPa
geViewController, viewControllerAfter viewController: UIVi
ewController) -> UIViewController?
        {
```

```
            let v = (viewController)
            let i = vcs.index(of: v)! + 1
            if i > vcs.count - 1 {
                return nil
            }
            return vcs[i]
        }
        func presentationCount(for pageViewController: UIP
ageViewController) -> Int
        {
            return vcs.count
        }
        func presentationIndex(for pageViewController: UIP
ageViewController) -> Int
        {
            return 0
        }
    }
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
    {
        var window: UIWindow?
        func application(_ application: UIApplication, wid
getdidFinishLaunchingWithOptions launchOptions: [UIApplica
tionLaunchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = PageViewCont
roller()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
```

前面的两个类都是UIViewController的子类，每个初始化一个标签，内容不同以示区别。类PageViewController是UIPageViewController的子类，实现了UIPageViewControllerDataSource并在装入时设置数据源为自身。通过：

```
     setViewControllers( [vcs[0]], direction: .forward, an
imated: false, completion: nil)
```

设置UIPageViewController的当前的ViewController。这里设置的是Page1的实例，因此App启动后显示的初始页面就是Page1。

协议UIPageViewControllerDataSource的内容如下：

```
    public protocol UIPageViewControllerDataSource : NSObj
ectProtocol {
        public func pageViewController(_ pageViewControlle
r: UIPageViewController, viewControllerBefore viewControll
er: UIViewController) -> UIViewController?
        public func pageViewController(_ pageViewControlle
r: UIPageViewController, viewControllerAfter viewControlle
r: UIViewController) -> UIViewController?
        optional public func presentationCount(for pageVie
wController: UIPageViewController) -> Int // The number of
 items reflected in the page indicator.
        optional public func presentationIndex(for pageVie
wController: UIPageViewController) -> Int // The selected
item reflected in the page indicator.
    }
```

其中前两个函数会有UIKit传入当前视图，希望App返回此视图的前一个视图或者是后一个视图。后两个函数分别返回总页面数量和初始页面对应的选择项索引，这两个函数是可选实现的，如果实现了，就会在页面底部显示页面指示器。

TabBarController

类UITabBarController是一个特殊的UIViewController，它可以包含多个UIViewController，并且在页面底部显示一个Tabbar作为UIViewController的切换显示开关。

如下案例，展示了包含两个UIViewController的UITabBarController，可以通过底部的Tabbar来切换显示：

```
    import UIKit
    class Page1: UIViewController {
        override func viewDidLoad() {
            super.viewDidLoad()
            let cc = UILabel(frame: CGRect(x: 10,y: 50,wid
th: 200,height: 50))
            cc.text = "Page #1"
            cc.textColor = UIColor.black
            self.view.addSubview(cc)
        }
    }
    class Page2: UIViewController {
        override func viewDidLoad() {
            super.viewDidLoad()
            let cc = UILabel(frame: CGRect(x: 10,y: 50,wid
th: 200,height: 50))
            cc.text = "Page #2"
            cc.textColor = UIColor.black
            self.view.addSubview(cc)
        }
    }
    class Tabbar: UITabBarController, UITabBarControllerDe
legate {
        override func viewDidLoad() {
            super.viewDidLoad()
```

```
            delegate = self
        }

        override func viewWillAppear(_ animated: Bool) {
            viewControllers = [Page1(),Page2()]
            let r = UIImage.imageWithColor(UIColor.black)
            viewControllers![0].tabBarItem = UITabBarItem(
title: "Page 1",
                              image:r,
                tag:0)
            viewControllers![1].tabBarItem = UITabBarItem(
title: "Page 2",
                image: r,
                 tag:1)

        }

        //Delegate methods
        func tabBarController(_ tabBarController: UITabBar
Controller, didSelect viewController: UIViewController) {
            print("did select viewController: \(viewContro
ller.tabBarItem.tag)")
        }
    }
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Tabbar()
            self.window!.rootViewController!.view.backgrou
ndColor = UIColor.white
```

```
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    // same code
    extension UIImage {
        class func imageWithColor(_ color: UIColor) -> UII
mage {
            let rect = CGRect(x: 0.0, y: 0.0, width: 10.0,
height: 10.0 )
            UIGraphicsBeginImageContext(rect.size)
            let context = UIGraphicsGetCurrentContext()

            context?.setFillColor(color.cgColor)
            context?.fill(rect)

            let image = UIGraphicsGetImageFromCurrentImage
Context()
            UIGraphicsEndImageContext()

            return image!
        }
    }
    //let r = UIImage.imageWithColor(UIColor.redColor()).i
mageWithRenderingMode(.AlwaysOriginal)
    //let b = UIImage.imageWithColor(UIColor.blueColor()).
imageWithRenderingMode(.AlwaysOriginal)
```

可以点击屏幕显示的下方按钮切换视图控制器。

本案例中，共有三个需要关注的类：

1. Page1，Page2。都是UIViewController的子类，内含一个UILabel区别显示为Page #1，Page #2

2. Tabbar 是UITabBarController的子类，实现协议UITabBarControllerDelegate，并在viewDidLoad时设置委托到自身；属性viewControllers是一个数组，用于装入多个UIViewController的实例，当执行：

   ```
   func viewWillAppear(_ animated: Bool)
   ```

初始化此数组，创建需要选择的UIViewController实例进来。

Tabbar就是一个UIViewController的子类，因此可以把它设置到self.window!.rootViewController上，从而成为window的rootViewController。

View

视图类UIView定义了一个矩形区域，并且管理此用户界面内的交互。视图可以管理子视图，全部子视图和自身视图一起，构成一个层次化的视图树。

我们完成一个案例，代码可以创建两个50见方的矩形，它们并排在一起，并且还有第三个视图，它嵌入在第二个视图内。三个视图的背景色分别为红、黄、蓝。当点击黄色矩形视图时，蓝色矩形会在蓝色和绿色之间切换。代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController{
        var a : UIView!
        var b : UIView!
        var c : UIView!
        override func viewDidLoad() {
            super.viewDidLoad()
            a = UIView()
```

```
            a.frame = CGRect(x: 0, y: 50, width: 50, heigh
t: 50)
            view.addSubview(a)
            a.backgroundColor = .red
            b = UIView()
            b.frame = CGRect(x: 50, y: 50, width: 50, heig
ht: 50)
            view.addSubview(b)
            b.backgroundColor = .yellow
            c = UIView()
            c.frame = CGRect(x: 0, y: 0, width: 25, height
: 25)
            b.addSubview(c)
            c.backgroundColor = .blue
            let gesture = UITapGestureRecognizer(target: s
elf, action: #selector(action(_:)))
            b.addGestureRecognizer(gesture)
        }
        func action(_ sender:UIButton!){
            c.backgroundColor = (c.backgroundColor == .gre
en) ?  .blue :  .green
        }
    }
```

根视图为UIViewController.view，接下来构建出来的视图层次树为：

```
view
-- a
-- b
----c
```

每个视图都可以通过方法addSubview来加入子视图，每个子视图的位置和大小可以通过属性.frame设置。UITapGestureRecognizer可以为UIView添加手势操作。

代码演示了UIView的层次树的构造，外观和事件处理。然而UIView本身在现实代码中使用并不多，它作为基础类，定义了视图的共同行为，实际上常用的是它的子类，比如UIButton、UILabel等，它的常用子类会在下文讲解。

ActivityIndicatorView

UIActivityIndicatorView就像一朵旋转的菊花，它常常用来指示有任务正在进行中。

代码展示了它的使用：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page :UIViewController{
        var v : UIActivityIndicatorView?
        override func viewDidLoad() {
            v = UIActivityIndicatorView()
            v!.frame = CGRect(x: 100, y: 100, width: 100,
height: 100)
            view.addSubview(v!)
            self.view.backgroundColor = UIColor.black

        }
        override func viewDidAppear(_ animated: Bool) {
            v!.startAnimating()
```

```
        }
        override func viewDidDisappear(_ animated: Bool) {
            v!.stopAnimating()
        }
    }
```

CollectionView

类UICollectionView管理一个有序的数据项目集合，并使用定制的布局展示它们。

假设我们希望以流式布局方式来显示一个多部分的集合，第一个部分是计算机语言，第二个部分是操作系统，像是这样：

```
     [["Java","Swift","JavaScript","PHP","Python"],["OS X"
,"Windows","Linux"]]
```

那么，使用UICollectionView是非常恰当的，示例代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController, UICollectionViewDelegate
FlowLayout, UICollectionViewDataSource {
        var items = [["Java","Swift","JavaScript","PHP","P
ython"],["OS X","Windows","Linux"]]
        var collectionView: UICollectionView!
```

```
        override func viewDidLoad() {
            super.viewDidLoad()
            let layout: UICollectionViewFlowLayout = UICol
lectionViewFlowLayout()
            layout.sectionInset = UIEdgeInsets(top: 20, le
ft: 20, bottom: 20, right: 20)
            layout.itemSize = CGSize(width: 80, height: 60
)
            var frame = view.frame
            frame.origin.y += 30
            collectionView = UICollectionView(frame: frame
, collectionViewLayout: layout)
            collectionView.dataSource = self
            collectionView.register(UICollectionViewCell.s
elf, forCellWithReuseIdentifier: "Cell")
            collectionView.backgroundColor = UIColor.white
            self.view.addSubview(collectionView)
        }
        func numberOfSections(in collectionView: UICollect
ionView) -> Int{
            return items.count
        }
        func collectionView(_ collectionView: UICollection
View, numberOfItemsInSection section: Int) -> Int {
            return items[section].count
        }

        func collectionView(_ collectionView: UICollection
View, cellForItemAt indexPath: IndexPath) -> UICollectionV
iewCell {
            let cell = collectionView.dequeueReusableCell(
withReuseIdentifier: "Cell", for: indexPath)
            cell.backgroundColor = .blue
            let l = UILabel()
            l.text = items[indexPath.section][indexPath.ro
w]
```

```
            l.frame =  CGRect(x: 0.0, y: 0.0, width: 100.0
,height: 20.0 )
            cell.addSubview(l)
            return cell
        }
    }
```

你可以实际执行它，查看显示的效果。此类也需要实现一个数据源协议：UICollectionViewDataSource，通过此协议实现，可以告知UICollectionView需要展示的数据源更会有多少部分，每个部分有多少项目，以及每个项目的Cell实例的内容。本文中的Cell实例内有一个UILabel的子视图，其文本内容就是要展示的项目的文本内容。

在函数

```
 func viewDidLoad()
```

中，使用了UICollectionViewFlowLayout作为UICollectionView的布局管理类，它可以通过属性sectionInset指定每个Cell的四方位置的空隙，通过属性itemSize指定每个Cell的大小。

和UICollectionView关联的一个重要类是UICollectionViewLayout，它可以管理所有Cell的大小和位置。刚刚看到的UICollectionViewFlowLayout是UICollectionViewLayout的一个子类。

ImageView

UIImageView可以显示一张图、或者一组动图。

如下代码显示一张图：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    extension UIImage {
        class func imageWithColor(_ color: UIColor) -> UII
mage {
            let rect = CGRect(x: 0.0, y: 0.0, width: 10.0,
height: 10.0 )
            UIGraphicsBeginImageContext(rect.size)
            let context = UIGraphicsGetCurrentContext()
            context?.setFillColor(color.cgColor)
            context?.fill(rect)
            let image = UIGraphicsGetImageFromCurrentImage
Context()
            UIGraphicsEndImageContext()
            return image!
```

```
        }
    }
    class Page: UIViewController {
        var bgImage: UIImageView!
        override func viewDidLoad() {
            super.viewDidLoad()
            bgImage = UIImageView()
            bgImage!.frame = CGRect(x: 0,y: 0,width: 100,h
eight: 100)
            bgImage.image = UIImage.imageWithColor(UIColor
.red)
            self.view.addSubview(bgImage!)
        }
    }
```

需要做的就是实例化UIImageView后，把属性.image赋值为UIImage的实例，后者可以用来装入图片。这里再次使用UIImage的扩展函数：

```
    class func imageWithColor(_ color: UIColor) -> UIImage
```

现场生产一张指定颜色的图片。

如果想要显示一组动图，可以把Page类改成如下代码，即可看到它的使用方法：

```
    class Page: UIViewController {
        var bgImage: UIImageView!
        override func viewDidLoad() {
            super.viewDidLoad()
            bgImage = UIImageView()
            bgImage!.frame = CGRect(x: 0,y: 0,width: 100,h
eight: 100)
            bgImage.image = UIImage.imageWithColor(UIColor
.red)
```

```
            self.view.addSubview(bgImage!)
        }
    }
    class Page100: UIViewController {
        var bgImage: UIImageView!
        override func viewDidLoad() {
            super.viewDidLoad()
            bgImage = UIImageView()
            bgImage.animationImages = [UIImage.imageWithCo
lor(UIColor.red),
                                       UIImage.imageWithCo
lor(UIColor.yellow),
                                       UIImage.imageWithCo
lor(UIColor.black),
                                       UIImage.imageWithCo
lor(UIColor.blue)]
            bgImage!.frame = CGRect(x: 0,y: 0,width: 100,h
eight: 100)
            self.view.addSubview(bgImage!)
            bgImage.animationDuration = 1
            bgImage.startAnimating()
        }
        override func viewDidDisappear(_ animated: Bool) {
            bgImage.stopAnimating()
        }
    }
```

UIImageView可以响应事件，如下代码演示轻按事件：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
```

```
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        var v: UIImageView!
        override func viewDidLoad() {
            super.viewDidLoad()
            let image1 = UIImage.imageWithColor(UIColor.re
d)
            let image2 = UIImage.imageWithColor(UIColor.bl
ue)
            v = UIImageView(image: image1,highlightedImage
:image2)
            v!.frame = CGRect(x: 0,y: 0,width: 100,height:
 100)
            self.view.addSubview(v!)
            let tapGestureRecognizer = UITapGestureRecogni
zer(target:self, action:#selector(imageTapped(_:)))
            v.isUserInteractionEnabled = true
            v.addGestureRecognizer(tapGestureRecognizer)
        }
        func imageTapped(_ img: AnyObject)
        {
            v.isHighlighted = !v!.isHighlighted
        }
    }
```

通过UITapGestureRecognizer手势识别类，可以为UIImageView添加触控事件，UIImageView本身支持正常图片和高光图片的切换，当轻按事件发生时，可以通过设置属性isHighlighted来切换图片的显示。利用这些特性，可以制作图片按钮。

PickerView

UIPickerView提供了类似皮包的机械密码锁一样风格的控件。它可以提供一组或者多组值的显示和选择。

如下代码展示了此控件的能力：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    // picker view
    class Page: UIViewController, UIPickerViewDelegate, UI
PickerViewDataSource {
        var p: UIPickerView!
        let values = [["1", "2", "3"],["one", "two", "thre
e"]]
        func numberOfComponents(in pickerView: UIPickerVie
w) -> Int{
            return 2
        }
        func pickerView(_ pickerView: UIPickerView, number
OfRowsInComponent component: Int) -> Int{
```

```
            return values[component].count
        }
        func pickerView(_ pickerView: UIPickerView, titleF
orRow row: Int, forComponent component: Int) -> String? {
            return values[component][row]
        }
        func pickerView(_ pickerView: UIPickerView, didSel
ectRow row: Int, inComponent component: Int){
            print(values[component][row])
        }
        func pickerView(_ pickerView: UIPickerView, widthF
orComponent component: Int) -> CGFloat {
            return 100
        }
        override func viewDidLoad() {
            super.viewDidLoad()
            view.backgroundColor = UIColor.white
            // 这个frame太小的话，会看不到文字，以至于误以为代码错
误。
            p = UIPickerView()
            p.frame =  CGRect(x: 10,y: 50,width: 320,heigh
t: 200)
            p.dataSource = self
            p.delegate = self
            view.addSubview(p)
        }
    }
```

本案例提供了2组数据集合供控件用来显示内容和做出选择。为了提供数据给控件，需要实现UIPickerViewDataSource协议，为了监听事件，需要实现UIPickerViewDelegate协议。

ProgressView

进度条类UIProgressView用来显示进度。

如下案例显示一个进度条：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController{
        var timer : Timer!
        var pv: UIProgressView!
        var pv1: UIProgressView!
        override func viewDidLoad() {
            super.viewDidLoad()
            view.backgroundColor = UIColor.white
            timer = Timer.scheduledTimer(timeInterval: 0.1
, target: self, selector: #selector(Page.update), userInfo
: nil, repeats: true)
            pv = UIProgressView()
            pv.frame = CGRect(x: 100, y: 100, width: 100,
height: 10)
```

```
            view.addSubview(pv)
            pv.progress = 0.51
            pv.progressViewStyle = .bar

        }
        func update() {
            pv.progress = pv.progress +  0.01
            if pv.progress >= 1 {
                pv.progress = 0
            }
        }
    }
```

特别注意的是，进度值范围从0到1，而不是从0到100，如果需要一个刻度的进展，那么加0.01。

ScrollView

UIScrollView支持显示尺寸比控件更大的内容。允许用户通过手势来滑动、放大缩小内容。

如下代码展示了UIScrollView的功能：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        var scroll: UIScrollView!
        override func viewDidLoad() {
            super.viewDidLoad()
            view.backgroundColor = UIColor.white
            self.scroll = UIScrollView()
            scroll.backgroundColor = UIColor.red
            self.scroll.contentSize = CGSize(width: 100, h
eight: 300)
            scroll.frame = CGRect(x: 100, y: 100, width: 1
00, height: 100)
            for i in 0  ..< 3 {
```

```
                let b = UILabel()
                b.backgroundColor = UIColor.blue
                b.frame = CGRect(x: 0, y: CGFloat(100 * i)
, width: 100, height: 100)
                b.backgroundColor = UIColor.green
                b.text = "Drag Me ! "
                scroll.addSubview(b)
            }
            view.addSubview(scroll)
        }
    }
```

你可以运行后，拖动查看本来因为控件比内容小而不得不隐藏的内容。

这里的控件高度是100，内容是三个高度为100的、从上到下排列的标签，总高度为300。这三个标签都是UIScrollView的内容，但是总高度比UIScrollView高，因此UIScrollView允许用户通过手势比如滑动去查看更多的被隐藏的内容。你可以看到本案例运行后，在横向（宽度方向）并不能拖动，因为内容宽度、控件宽度是一致的，因此并不必要。

SearchBar

UISearchBar实现了一个用于填写文本以便搜索的控件。它由一个文本框、一个搜索按钮、一个书签按钮、一个取消按钮构成。它并不执行真正的搜索，而是通过实现UISearchBarDelegate协议，在协议函数内执行真正的搜索。

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController,UISearchBarDelegate{
        var c : UISearchBar!
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UISearchBar()
            c.frame = CGRect(x: 100, y: 100, width: 200, h
eight: 50)
            view.addSubview(c)
            c.delegate = self
        }
        func searchBarSearchButtonClicked(_ searchBar: UIS
earchBar) {
```

```
            print ("search : \(searchBar.text)")
        }
    }
```

默认情况下，只是显示搜索按钮。如果想要显示书签按钮和取消按钮，需要这样

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController,UISearchBarDelegate{
        var c : UISearchBar!
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UISearchBar()
            c.frame = CGRect(x: 100, y: 100, width: 200, h
eight: 50)
            view.addSubview(c)
            c.delegate = self
            c.showsCancelButton = true
            c.showsBookmarkButton = true
        }
        func searchBarSearchButtonClicked(_ searchBar: UIS
```

```
earchBar) {
            print ("search : \(searchBar.text)")
        }
        func searchBarBookmarkButtonClicked(_ searchBar: U
ISearchBar) {
            print ("bookmark : \(searchBar.text)")
        }
        func searchBarCancelButtonClicked(_ searchBar: UIS
earchBar) {
            print ("cancel : \(searchBar.text)")
        }
    }
```

在启动后设置属性：

```
    c.showsCancelButton = true
    c.showsBookmarkButton = true
```

相应的按钮事件处理也在UISearchBarDelegate指定的函数中，分别是：

```
    func searchBarBookmarkButtonClicked(_ searchBar: UISea
rchBar)
```

和 func searchBarCancelButtonClicked(_ searchBar: UISearchBar)

TableView

类UITableView表示一个列表视图，此视图可以显示列表，并且列表可以分为多个区间（section）。

显示列表

假设一个案例：

1. 显示计算机语言清单（["java","swift","js"]和操作系统的清单["Windows","OS X","Linux"]
2. 这个清单在一个UITableView上做分区显示，分为两个区间

那么代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        var a : Table!
```

```
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
            a.frame = CGRect(x: 0,y: 50,width: 300,height:
 500)
            self.view.addSubview(a)
        }
    }


    class Table : UITableView,UITableViewDataSource,UITabl
eViewDelegate{
        let sect = ["Lang","OS"]
        let lang = ["java","swift","js"]
        let os = ["Windows","OS X","Linux"]
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
            self.dataSource = self
            self.delegate = self
        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func numberOfSections(in: UITableView) -> Int {
            return 2
        }
        func tableView(_ tableView: UITableView, viewForHe
aderInSection section: Int) -> UIView? {
            let rect = CGRect(x: 0, y: 0, width: tableView
.frame.size.width, height: 44)
            let footerView = UILabel(frame:rect)
            footerView.text = sect[section]
            return footerView
        }
        func tableView(_ tableView: UITableView, heightFor
```

```
HeaderInSection section: Int) -> CGFloat {
            return 44
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return section == 0 ?lang.count:os.count
        }
        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let arr = indexPath.section == 0 ? lang:os
            let a = UITableViewCell(style: .default, reuse
Identifier: nil)
            a.textLabel?.text = String(describing:arr[inde
xPath.row])
            return a
        }
    }
```

代码创建了三个类，其中的AppDelegate和之前的并没有什么不同，Page继承于UIViewController，也和之前的代码类似，只是在载入时把Table作为子视图加入进来。要特别介绍的是Table。

Table继承于UITableView，并实现UITableViewDataSource,UITableViewDelegate，在配合代码：

```
        self.dataSource = self
        self.delegate = self
```

就指明了UITableView的数据源对象为Table，委托对象也是Table。前者指明此对象是UITableView的数据提供者，后者指明此对象是UITableView的外观和行为的提供者。

具体数据提供的方法就是在此类内实现方法：

```
        func numberOfSections(in: UITableView) -> Int {
            return 2
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return section == 0 ?lang.count:os.count
        }
        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let arr = indexPath.section == 0 ? lang:os
            let a = UITableViewCell(style: .default, reuse
Identifier: nil)
            a.textLabel?.text = String(describing:arr[inde
xPath.row])
            return a
        }
```

第一个方法告诉TableView此列表共有两个区间要去显示。第二个方法告诉TableView此列表每个区间的行数，第三个方法告诉TableView指定的区间和行数的内容是什么。

作为UITableView的外观和行为的提供者，具体体现是实现了这些方法：

```
        func tableView(_ tableView: UITableView, viewForHe
aderInSection section: Int) -> UIView? {
            let rect = CGRect(x: 0, y: 0, width: tableView
.frame.size.width, height: 44)
            let footerView = UILabel(frame:rect)
            footerView.text = sect[section]
            return footerView
        }
        func tableView(_ tableView: UITableView, heightFor
HeaderInSection section: Int) -> CGFloat {
            return 44
        }
```

第一个方法为指定的区间创建一个头视图，第二个方法指示指定区间的行高。

协议UITableViewDataSource,UITableViewDelegate还有很多可以实现的方法，具体参考iOS的开发者参考资料。

添加删除修改

类UITableView不但可以显示内容，还可以配合很多操作。这些内容的操作的基本流程就是修改数据源，然后通知TableView重新装入。代码入下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        var a : Table!
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
```

```
            a.frame = CGRect(x: 0,y: 50,width: 300,height:
 500)
            self.view.addSubview(a)
        }
    }
    class Table : UITableView,UITableViewDataSource,UITabl
eViewDelegate{
        var sect = NSMutableArray.init(array: ["Lang","OS"
])
        var lang = NSMutableArray.init(array: ["java","swi
ft","js"])
        var os = NSMutableArray.init(array:["Windows","OS
X","Linux"])
        var t = Timer()
        var count  = 0
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
            self.dataSource = self
            self.delegate = self
            t.invalidate()
            t = Timer.scheduledTimer(timeInterval: 1, targ
et: self, selector: #selector(self.update), userInfo: nil,
 repeats: true);
        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func update() {
            if count == 0 {
                os[0] = "Win"

            }else if count == 1 {
                os.add("FreeBSD")
            }else if count == 2 {
                lang.removeObject(at: 0)
```

```
            }
            count += 1
            if count >= 3 {
                 t.invalidate()
            }
            self.reloadData()
        }
        func numberOfSections(in: UITableView) -> Int {
            return sect.count
        }
        func tableView(_ tableView: UITableView, viewForHe
aderInSection section: Int) -> UIView? {
            let rect = CGRect(x: 0, y: 0, width: tableView
.frame.size.width, height: 44)
            let footerView = UILabel(frame:rect)
            footerView.text = String(describing: sect[sect
ion])
            return footerView
        }
        func tableView(_ tableView: UITableView, heightFor
HeaderInSection section: Int) -> CGFloat {
            return 44
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return section == 0 ?lang.count:os.count
        }
        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let arr = indexPath.section == 0 ? lang  as NS
Array :os  as NSArray
            let a = UITableViewCell(style: .default, reuse
Identifier: nil)
            a.textLabel?.text = String(describing:arr[inde
xPath.row])
            return a
```

```
        }
    }
```

此代码和上一节的代码相比，不同在于：

1. 之前的计算机语言数组和操作系统数据被改成了可变的数组，因为在这个代码中，我们需要修改数据来验证对UITableView的修改
2. 在Table的init函数内，创建一个Timer，它每秒激发一个定时器事件，在不同的激发次数中，分别对数据做修改、添加、删除
3. 调用reload方法，从而让UITableView重新加载数据

默认提供的删除和列表重排

可以自己添加按钮并执行对UITableView的列表的删除和重排。但是也可以使用它自己提供了删除和重排的UI。删除流程是这样的：

1. 用户设置UITableView为编辑模式
2. 系统在当前内容的基础上，加上删除按钮（内容左侧，红色的减号图标），以及重排按钮（内容右侧）
3. 用户可以选择点击删除按钮，系统向左推移内容，显示一个delete按钮
4. 用户点击delete按钮，系统就会调用程序员实现的委托对象的函数：func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath)

重排流程是这样的：

1. 用户设置UITableView为编辑模式
2. 系统在当前内容的基础上，加上删除按钮（内容左侧，红色的减号图标），以及重排按钮（内容右侧）
3. 用户可以选择按住拖动重排按钮，系统可视化这样拖动过程
4. 用户拖动完成，系统就会调用程序员实现的委托对象的函数： func tableView(_ tableView: UITableView, moveRowAt sourceIndexPath:

IndexPath, to destinationIndexPath: IndexPath)

代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        var a : LangTableRowOper1?
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = LangTableRowOper1()
            a!.frame = CGRect(x: 0,y: 200,width: 300,heigh
t: 200)
            self.view.addSubview(a!)
            let b = UIButton()
            b.setTitle("edit", for: UIControlState())
            b.backgroundColor = UIColor.red
            b.addTarget(self, action: #selector(ViewContro
ller.edit(_:)), for: .touchDown)

            let e = UIButton()
```

```
            e.setTitle("Done", for: UIControlState())
            e.backgroundColor = UIColor.blue
            e.addTarget(self, action: #selector(Page.done(
_:)), for: .touchDown)

            let sv = UIStackView()

            sv.backgroundColor = UIColor.gray
            sv.axis = UILayoutConstraintAxis.horizontal
            sv.distribution = .equalCentering;
            sv.alignment = .center;
            sv.spacing = 10;
            sv.frame = CGRect(x: 0,y: 100,width: 300,heigh
t: 50)
            sv.addArrangedSubview(b)

            sv.addArrangedSubview(e)
            sv.translatesAutoresizingMaskIntoConstraints =
 true
            self.view.addSubview(sv)

        }
        func edit( _ b : UIButton!){
            a!.setEditing(true, animated: true)
        }
        func done( _ b : UIButton!){
            a!.setEditing(false, animated: true)
        }
    }
    class Table : UITableView,UITableViewDataSource,UITabl
eViewDelegate{
        var arr = NSMutableArray.init(array: ["java","swif
t","js"])
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
```

```
            self.dataSource = self
            self.delegate = self

        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return arr.count
        }
        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let a = UITableViewCell(style: .default, reuse
Identifier: nil)
            a.textLabel?.text = String(describing: arr[ind
exPath.row])
            return a
        }
        func tableView(_ tableView: UITableView, commit ed
itingStyle: UITableViewCellEditingStyle, forRowAt indexPat
h: IndexPath) {
            if editingStyle ==  .delete{
                arr.removeObject(at: indexPath.row) // htt
p://stackoverflow.com/questions/21870680/invalid-update-in
valid-number-of-rows-in-section-0
                self.deleteRows(at: [indexPath], with: UIT
ableViewRowAnimation.fade)
            }
        }
        func tableView(_ tableView: UITableView, canMoveRo
wAt indexPath: IndexPath) -> Bool
        {
            return true;
        }
        func tableView(_ tableView: UITableView, moveRowAt
```

```
 sourceIndexPath: IndexPath, to destinationIndexPath: Inde
xPath) {

            let s = sourceIndexPath.row
            let d = destinationIndexPath.row
            let temp = arr[s]
            arr.removeObject(at: s)
            arr.insert(temp, at: d)
        }
    }
```

注意，代码中：

```
    func tableView(_ tableView: UITableView, canMoveRowAt
indexPath: IndexPath) -> Bool
        {
            return true;
        }
```

此函数需要实现，从而告诉UITableView那些行是可以拖动重排的。这里全部返还true，表示所有内容都可以重排。

TableView的装饰界面

除了显示section和row之外，TableView可以加入表头表位，节头节尾，帮助程序员更好的组织内容。

我们现在来展示了一个有两个section，每个section有行的界面。通过代码加入了表头表尾、节头节尾。如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
```

```
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        self.window = UIWindow(frame: UIScreen.main.bounds
)
        let page = Page()
        page.view.backgroundColor = .blue
        self.window!.rootViewController = page
        self.window?.makeKeyAndVisible()
        return true
    }
}
class Table : UITableView,UITableViewDataSource{
    let arrs = [["Row 1","Row 2"],["Row 1"]]
    let titles  = ["Section Title 1","Section Title 2"]
    let footers  = ["Section Footer 1","Section Footer 2"]
    let tableheader = "Table Header"
    let tablefooter = "Table Footer"
    convenience init(){
        self.init(frame: CGRect.zero, style:UITableViewSty
le.grouped)
    }
    override init(frame: CGRect, style: UITableViewStyle)
{
        super.init(frame:frame,style:style)
        self.dataSource = self
        self.tableHeaderView = UIView()
        self.tableHeaderView!.frame = CGRect(x: 0, y: 0,wi
dth: 200,height: 20)
        let l = UILabel()
        l.text = tableheader
        l.frame = CGRect(x: 0, y: 0,width: 200,height: 20)
        self.tableHeaderView?.addSubview(l)

        self.tableFooterView = UIView()
        self.tableFooterView!.frame = CGRect(x: 0, y: 0,wi
```

```
dth: 200,height: 20)
        let f = UILabel()
        f.text = tablefooter
        f.frame = CGRect(x: 0, y: 0,width: 200,height: 20)
        self.tableFooterView?.addSubview(f)
    }
    required init?(coder aDecoder: NSCoder) {
        super.init(coder:aDecoder)
    }
    func tableView(_ tableView: UITableView, numberOfRowsI
nSection section: Int) -> Int {
        return arrs[section].count
    }
    func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell{
        let a = UITableViewCell(style: UITableViewCellStyl
e.value1, reuseIdentifier: nil)
        a.textLabel?.text = String(arrs[indexPath.section]
[indexPath.row])
        return a
    }
    func numberOfSections(in tableView: UITableView) -> In
t{
        return arrs.count
    }
    func tableView(_ tableView: UITableView, titleForHeade
rInSection section: Int) -> String?{
        return titles[section]
    }

    func tableView(_ tableView: UITableView, titleForFoote
rInSection section: Int) -> String?{
        return footers[section]
    }
}
```

```
class Page: UIViewController {
    var a : Table!
    override func viewDidLoad() {
        super.viewDidLoad()
        a  = Table()
        a.frame = CGRect(x: 0,y: 30,width: 300,height: 400
)
        self.view.addSubview(a)
    }
}
```

这里比较特别的是，函数

```
    func tableView(_ tableView: UITableView, titleForHeade
rInSection section: Int) -> String?{
        return titles[section]
    }
```

告诉TableView，每个指定section的头标题。 func tableView(_ tableView: UITableView, titleForFooterInSection section: Int) -> String?{ return footers[section] } 告诉TableView，每个指定section的尾标题。

标记

类UITableView支持对每个行做标记和取消标记，标记可以有多种。其中比较常用的是打对号图标。如下代码，演示了如何对每个行打对号和取消打对号：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
```

```
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Table : UITableView,UITableViewDataSource,UITabl
eViewDelegate{
        let arr = ["java","swift","js"]
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
            self.dataSource = self
            self.delegate = self

        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return arr.count
        }

        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let a = UITableViewCell(style: .default, reuse
Identifier: nil)
            a.textLabel?.text = String(arr[indexPath.row])
            return a
        }
```

```
        func tableView(_ tableView: UITableView, didSelect
RowAt indexPath: IndexPath){
            print("did select \(indexPath.row)")
            self.deselectRow(at: indexPath, animated: fals
e)
            if  self.cellForRow(at: indexPath)?.accessoryT
ype !=  .checkmark{
                self.cellForRow(at: indexPath)?.accessoryT
ype = .checkmark
            }else{
                self.cellForRow(at: indexPath)?.accessoryT
ype = .none
            }
        }
    }
    class Page: UIViewController {
        var a : Table!
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
            a.frame = CGRect(x: 0,y: 30,width: 300,height:
 400)
            self.view.addSubview(a)
        }
    }
```

运行起来后，可看到三个行，点击任何一个行都会显示对号标记在行尾，再点击一次就会取消此标记。查询UITableViewCellAccessoryType的官方文档可以得到更多的标记类型。

重用cell创建

在之前的代码中，每次调用到函数：

```
    func tableView(_ tableView: UITableView, cellForRowAt
```

```
indexPath: IndexPath) -> UITableViewCell
```

需要一个UITableViewCell实例时，都是采用临时创建的方式。如果创建的UITableViewCell实例比较多时，可以通过重用已经创建的UITableViewCell实例来提高效率。做法就是：

1. 注册一个UITableViewCell类，指定其的重用标识符
2. 通过重用标识符创建实例

UIKit会在内部对此过程优化。实例代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Table: UITableView,UITableViewDataSource{
        let arr = ["javascript","delphi"]
        let MyIdentifier = "cell"
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
            self.dataSource = self
            self.register(UITableViewCell.self, forCellReu
```

```
seIdentifier: MyIdentifier)
        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return arr.count
        }

        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let a = tableView.dequeueReusableCell(withIden
tifier: MyIdentifier)!
            a.textLabel?.text = String(arr[indexPath.row])
            return a
        }
    }
    class Page: UIViewController {
        var a : Table!
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
            a.frame = CGRect(x: 0,y: 30,width: 300,height:
 400)
            self.view.addSubview(a)
        }
    }
```

复合的Cell

之前的代码，创建的Cell都是简单文字；实际上，每个Cell都可以作为一个容器，装入更多的元素。如下代码展示了一个复合的Cell的创建：

```
    import UIKit
```

```
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Table : UITableView,UITableViewDataSource{
            let arr = ["java","swift","js"]
            override init(frame: CGRect, style: UITableVie
wStyle) {
                super.init(frame:frame,style:style)
                self.dataSource = self
            }
            required init?(coder aDecoder: NSCoder) {
                super.init(coder:aDecoder)
            }
            func tableView(_ tableView: UITableView, numbe
rOfRowsInSection section: Int) -> Int {
                return arr.count
            }

            func tableView(_ tableView: UITableView, cellF
orRowAt indexPath: IndexPath) -> UITableViewCell {
                let a = UITableViewCell(style: .default, r
euseIdentifier: nil)
                a.textLabel?.text = String(arr[indexPath.r
```

```
ow])
                let s = UISwitch()
                s.frame = CGRect(x: 0,y: 0,width: 20,heigh
t: 20)
                s.addTarget(self, action: #selector(Table.
action(_:)), for: .valueChanged)
                s.isOn = true
                a.accessoryView = s
                return a
            }
            func action(_ sender : UISwitch!){
                print(sender.isOn)
            }
        }
    class Page: UIViewController {
        var a : Table!
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
            a.frame = CGRect(x: 0,y: 30,width: 300,height:
 400)
            self.view.addSubview(a)
        }
    }
```

本案例在每个Cell中添加了一个UISwitch按钮，并且可以如同一般的UIView一样的响应此按钮的事件。

默认的Cell风格

可以不必自己定制Cell样式，而是直接使用系统提供的，这样你可以通过设置不同的UITableViewCellAccessoryType、文字、文字1、图片、UITableViewCellStyle而让Cell外观变得丰富多彩：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = Page()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }


    class Page: UIViewController {
        var a : Table!
        override func viewDidLoad() {
            super.viewDidLoad()
            a  = Table()
            a.frame = CGRect(x: 0,y: 30,width: 300,height:
 400)
            self.view.addSubview(a)
        }
    }


    class Row {
        var text : String = ""
        var text2 : String = ""
        var image : UIImage
        var access : UITableViewCellAccessoryType
```

```
        var style :  UITableViewCellStyle
        init( text : String ,text2:String ,image:UIImage,a
ccess:UITableViewCellAccessoryType,style :  UITableViewCel
lStyle){
            self.text = text
            self.text2 = text2
            self.image = image
            self.access = access
            self.style = style
        }
    }
    class Table: UITableView,UITableViewDataSource{
        let arr = [
            Row(
                text:"java",
                text2:"old plain",
                image:UIImage.imageWithColor(UIColor.red),
                access:UITableViewCellAccessoryType.checkm
ark,
                style: UITableViewCellStyle.value1),
            Row(
                text:"ruby",
                text2:"new cool slow",
                image:UIImage.imageWithColor(UIColor.green
),
                access:UITableViewCellAccessoryType.detail
Button,
                style: UITableViewCellStyle.value2),
            Row(
                text:"swift",
                text2:"new cool quick ",
                image:UIImage.imageWithColor(UIColor.blue)
,
                access:UITableViewCellAccessoryType.detail
DisclosureButton,
                style: UITableViewCellStyle.subtitle)
```

```
        ]
        override init(frame: CGRect, style: UITableViewSty
le) {
            super.init(frame:frame,style:style)
            self.dataSource = self

        }
        required init?(coder aDecoder: NSCoder) {
            super.init(coder:aDecoder)
        }
        func tableView(_ tableView: UITableView, numberOfR
owsInSection section: Int) -> Int {
            return arr.count
        }

        func tableView(_ tableView: UITableView, cellForRo
wAt indexPath: IndexPath) -> UITableViewCell {
            let a = UITableViewCell(style: arr[indexPath.r
ow].style, reuseIdentifier: nil)
            a.textLabel?.text = arr[indexPath.row].text
            a.detailTextLabel?.text = arr[indexPath.row].t
ext2
            a.imageView?.image = arr[indexPath.row].image
            a.accessoryType = arr[indexPath.row].access
            return a
        }
    }
    extension UIImage {
        class func imageWithColor(_ color: UIColor) -> UII
mage {
            let rect = CGRect(x: 0.0, y: 0.0, width: 10.0,
height: 10.0 )
            UIGraphicsBeginImageContext(rect.size)
            let context = UIGraphicsGetCurrentContext()

            context?.setFillColor(color.cgColor)
```

```
            context?.fill(rect)

            let image = UIGraphicsGetImageFromCurrentImage
Context()
            UIGraphicsEndImageContext()

            return image!
        }
    }
```

案例显示了三行，每行有不同的风格组合，都是通过设置UITableViewCellAccessoryType、文字、文字1、图片、UITableViewCellStyle来达成的。你可以实际运行此代码，了解不同样式的差异。可以通过官方手册查询UITableViewCellStyle和UITableViewCellAccessoryType的不同选择。这里列出的类为UIImage扩展出来的方法：

```
    class func imageWithColor(_ color: UIColor) -> UIImage
```

是为了不必寻找图片，而可以即席按需创建出可以用于实例的图片，传递不同的颜色值，可以得到不同颜色的小方块图片。我们只是需要展示Cell的显示图片的能力，因此只有通过代码创建出图就好，不必为此单独寻找适合工程使用的图片。

UITableViewController

我们一直使用UITableView，把它加入到一个ViewController内，然后由AppDelegate加载后者。实际上，可以使用UITableViewController直接由AppDelegate加载：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
```

```
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            let page = LangTableViewController()
            page.view.backgroundColor = .blue
            self.window!.rootViewController = page
            self.window?.makeKeyAndVisible()
            return true
        }
    }

    class LangTableViewController : UITableViewController{
        let arr = ["swift","obj-c","ruby"]
        let MyIdentifier = "cell"
        override func viewDidLoad() {
            super.viewDidLoad()
            tableView.register(UITableViewCell.self, forCe
llReuseIdentifier: MyIdentifier)
        }
        override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
            return arr.count
        }
        override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
            let a = tableView.dequeueReusableCell(withIden
tifier: MyIdentifier)
            a!.textLabel?.text = arr[indexPath.row]
            return a!
        }
    }
```

和UITableView的使用的不同之处，在于：

1. 本来需要实现UITableViewDataSource和UITableViewDelegate的方法，现在已经有UITableViewController实现，作为UITableViewController的子类，新类需要的是覆盖父类的方法。
2. 使用UITableViewController会自动把UITableView填满AppDelegate.window视图内。无需程序员指定位置和大小。

TextView

UITextView实现了一个多行的、可滚动内容、可编辑的内容区域。

代码示例如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController,UITextViewDelegate{
        var c : UITextView!
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UITextView()
            c.frame = CGRect(x: 10, y: 50, width: 200, hei
ght: 400)
            view.addSubview(c)
            c.text = "text\nand new lines\nand more lines"
            c.delegate = self
        }
        func textViewDidChange(_ textView: UITextView){
            print ("new value : \(c.text)")
```

```
        }
    }
```

和其他视图一样，使用过程如下：

1. 实例化
2. 设置大小位置
3. 加入父视图

实现协议UITextViewDelegate的话，可以在类内接受事件，比如：

```
    func textViewDidChange(_ textView: UITextView){
        print ("new value : \(c.text)")
    }
```

此事件会在内容改变时得到调用。

WebView

UIWebView用来在App内嵌入Web页面。如下代码装入Apple.com官方首页在App内。

```
    import UIKit
    class Page: UIViewController{
        var c : UIWebView!
        override func viewDidLoad() {
            super.viewDidLoad()
            c = UIWebView()
            c.frame = super.view.frame
            view.addSubview(c)
            c.frame.origin.y += 20
            let url = URL(string:"http://apple.com")
            let ro = URLRequest(url:url!)
            c.loadRequest(ro)
        }
    }
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
```

关键代码在于：

```
let url = URL(string:"http://apple.com")
let ro = URLRequest(url:url!)
c.loadRequest(ro)
```

构建URL对象和URLRequest请求对象，然后使用WebView的方法loadRequest发起请求，即可加载页面在WebView内。

AutoLayout

典型的，加入一个子视图的做法就是创建实例，指定位置，然后把它加入到视图里面来。在指定位置的代码处，一般就是设置frame属性即可，就像这样：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController{
        var vw : UIView!
        override func viewDidLoad() {
            super.viewDidLoad()
            vw = UIView()
            vw.backgroundColor = .red
            vw.frame = CGRect(x: 5, y: 5, width: 50, heigh
t: 50)
            view.addSubview(vw)
        }
    }
```

代码把一个边长为50的正方形加入到x: 5, y: 5的位置上，位置相对于它的父视图的左上脚点。如果我们的相对点变化了，比如说希望把这个正方形以自身的右下脚点为惨遭，放到相对于父视图的右下脚点的的-5,-5的位置上的话，会产生些问题：

1. 正方形的左上角的x和y的值就得自己计算
2. 当手机旋转时，屏幕的横向长度和纵向长度就变了，因此x和y的值也得自己重新计算

解决此问题的要点在于，App只是声明自己的位置关系，具体的坐标有UIKit来计算。现在UIKit提供的NSLayoutConstraint类就可以解决此问题。

NSLayoutConstraint

就以此问题为例，可以去掉：

```
    vw.frame = CGRect(x: 5, y: 5, width: 50, height: 50)
```

改成如下的约束：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
```

```
        }
    }
    class Page: UIViewController{
        var vw : UIView!
        override func viewDidLoad() {
            super.viewDidLoad()
            vw = UIView()
            vw.backgroundColor = .red
            view.addSubview(vw)
            vw.translatesAutoresizingMaskIntoConstraints =
 false
            let horizontalConstraint = NSLayoutConstraint(
item: vw, attribute: NSLayoutAttribute.right, relatedBy: N
SLayoutRelation.equal, toItem: view, attribute: NSLayoutAt
tribute.right, multiplier: 1, constant: -5)
            let verticalConstraint = NSLayoutConstraint(it
em: vw, attribute: NSLayoutAttribute.bottom, relatedBy: NS
LayoutRelation.equal, toItem: view, attribute: NSLayoutAtt
ribute.bottom, multiplier: 1, constant: -5)
            let widthConstraint = NSLayoutConstraint(item:
 vw, attribute: NSLayoutAttribute.width, relatedBy: NSLayo
utRelation.equal, toItem: nil, attribute: NSLayoutAttribut
e.notAnAttribute, multiplier: 1, constant: 50)
            let heightConstraint = NSLayoutConstraint(item
: vw, attribute: NSLayoutAttribute.height, relatedBy: NSLa
youtRelation.equal, toItem: nil, attribute: NSLayoutAttrib
ute.notAnAttribute, multiplier: 1, constant: 50)
            view.addConstraints([horizontalConstraint, ver
ticalConstraint, widthConstraint, heightConstraint])

        }
    }
```

首先必须设置此视图的属性：

```
vw.translatesAutoresizingMaskIntoConstraints = false
```

这样AutoLayout才能生效。随后创建的第一约束：

```
let horizontalConstraint = NSLayoutConstraint(item: vw
, attribute: NSLayoutAttribute.right, relatedBy: NSLayoutR
elation.equal, toItem: view, attribute: NSLayoutAttribute.
right, multiplier: 1, constant: -5)
```

看起来有些复杂，但是如果知道一个公式就更容易读懂此代码。NSLayoutConstraint用来创建阅读，其函数参数标签有：

1. item 要定位的视图对象
2. attribute 要定位的视图对象的属性
3. relatedBy 相对关系，也就是公式内的操作符号，最常用的就是等于，但是也可以使用大于小于等操作符号
4. toItem 用来对定位点的视图对象
5. attribute 用来对定位点的视图对象的属性
6. multiplier 系数
7. constant 常数

公式由此标签指定如下：

```
item.attribute relatedBy toItem.attribute*multiplier +
constant
```

标签relatedBy指定的是一个操作符号，此处为“=”，因此此公式简化为：

```
item.attribute = toItem.attribute*multiplier + constan
t
```

再次的把标签带入参数值，就是这样：

```
vw.right = view.right*1 - 5
```

读出来的意思就是：

```
视图vw的右边x值约束为视图view的右边x值乘以1再减去5
```

于是可以类推第二个约束为正方形的下边y值和父视图的下边y值的约束定义：

```
视图vw的下边y值约束为视图view的下边y值乘以1再减去5
```

此案例中，正方形的宽度则只是一个常数值，并不需要相对于任何视图的任何属性，因此创建类NSLayoutConstraint时，公式中：

```
item.attribute relatedBy toItem.attribute*multiplier +
constant
```

的toItem.attribute*multiplier整体是无意义的，而简化为：

```
item.attribute = constant
```

这也就是代码：

```
 let widthConstraint = NSLayoutConstraint(item: vw, at
tribute: NSLayoutAttribute.width, relatedBy: NSLayoutRelat
ion.equal, toItem: nil, attribute: NSLayoutAttribute.notAn
Attribute, multiplier: 1, constant: 50)
```

标签toItem为nil， 标签attribute为NSLayoutAttribute.notAnAttribute的含义。同样的，正方形的高度也是如此分析。

使用自动布局，一个视图的位置的参照物不再仅仅是父视图的左上脚点，而是变得丰富多彩，依据不同的情况有不同的选择：

1. 指定视图的中心点和父视图重合
2. 指定一个视图和它的兄弟视图紧靠
3. 靠左靠右靠上靠下

如此等等。因此之前的指定xy值的做法，就称为自动布局方法的一个特殊案例。为了完整理解此约束的概念，下面的案例演示了一个视图依赖于它的兄弟视图的做法：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController{
        var vw : UIView!
        var vw2 : UIView!
        func pinToBottomRight(_ view1:UIView,_ view2:UIVie
w,_ x: CGFloat,_ y :CGFloat){
            vw.translatesAutoresizingMaskIntoConstraints =
 false
            let horizontalConstraint = NSLayoutConstraint(
item: view1, attribute: NSLayoutAttribute.right, relatedBy
: NSLayoutRelation.equal, toItem: view, attribute: NSLayou
```

```
tAttribute.right, multiplier: 1, constant: x)
            let verticalConstraint = NSLayoutConstraint(it
em: view1, attribute: NSLayoutAttribute.bottom, relatedBy:
 NSLayoutRelation.equal, toItem: view, attribute: NSLayout
Attribute.bottom, multiplier: 1, constant: y)
            let widthConstraint = NSLayoutConstraint(item:
 view1, attribute: NSLayoutAttribute.width, relatedBy: NSL
ayoutRelation.equal, toItem: nil, attribute: NSLayoutAttri
bute.notAnAttribute, multiplier: 1, constant: 50)
            let heightConstraint = NSLayoutConstraint(item
: view1, attribute: NSLayoutAttribute.height, relatedBy: N
SLayoutRelation.equal, toItem: nil, attribute: NSLayoutAtt
ribute.notAnAttribute, multiplier: 1, constant: 50)
            view.addConstraints([horizontalConstraint, ver
ticalConstraint, widthConstraint, heightConstraint])

        }
        func pinToBottomLeft(_ view1:UIView,_ view2:UIView
,_ x: CGFloat,_ y :CGFloat){
            vw2.translatesAutoresizingMaskIntoConstraints
= false
            let horizontalConstraint2 = NSLayoutConstraint
(item: view1, attribute: NSLayoutAttribute.right, relatedB
y: NSLayoutRelation.equal, toItem: view2, attribute: NSLay
outAttribute.left, multiplier: 1, constant: x)
            let verticalConstraint2 = NSLayoutConstraint(i
tem: view1, attribute: NSLayoutAttribute.bottom, relatedBy
: NSLayoutRelation.equal, toItem: view2, attribute: NSLayo
utAttribute.bottom, multiplier: 1, constant: y)
            let widthConstraint2 = NSLayoutConstraint(item
: view1, attribute: NSLayoutAttribute.width, relatedBy: NS
LayoutRelation.equal, toItem: nil, attribute: NSLayoutAttr
ibute.notAnAttribute, multiplier: 1, constant: 50)
            let heightConstraint2 = NSLayoutConstraint(ite
m: view1, attribute: NSLayoutAttribute.height, relatedBy:
NSLayoutRelation.equal, toItem: nil, attribute: NSLayoutAt
```

```
tribute.notAnAttribute, multiplier: 1, constant: 50)
            view.addConstraints([horizontalConstraint2, ve
rticalConstraint2, widthConstraint2, heightConstraint2])

        }
        override func viewDidLoad() {
            super.viewDidLoad()
            vw = UIView()
            vw.backgroundColor = .red
            view.addSubview(vw)
            pinToBottomRight(vw,view,-5,-5)
            vw2 = UIView()
            vw2.backgroundColor = .blue
            view.addSubview(vw2)
            pinToBottomLeft(vw2,vw,-5,0)

        }
    }
```

这里涉及到了三个视图，view作为根视图，vw做一个一个子视图，位置参照view的右下脚点；vw2视图以vw的左下脚点为参照，它的右侧坐标保持在vw左侧坐标的-5的位置。

Layout Anchor

使用NSLayoutConstraint的方式，是UIKit的布局的核心，也是最全面的方案。然而，它看起来过于复杂，难以阅读和书写，因此UIKit布局还有另外的方式可以，这就是Layout Anchor方式，它使用了NSLayoutAnchor。通过提供定位锚点以及和锚点之间的关系的设置来完成布局约束。比如定位是相对于某个视图的底边的话，那么可以使用 bottomAnchor。锚点可以使用topAnchor、bottomAnchor、rightAnchor、leftAnchor、centerXAnchor、centerYAnchor等。同样的布局需求，采用Layout Anchor的代码。具体而言如果想要建立一个约束：

视图1的右边和视图2的左边差距为-5

可以表达为：

```
view1.rightAnchor = view2.leftAnchor - 5
```

解决本案例的需求对应的代码就是这样：

import UIKit @UIApplicationMain class AppDelegate: UIResponder, UIApplicationDelegate { var window: UIWindow? func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool { self.window = UIWindow(frame: UIScreen.main.bounds) self.window!.rootViewController = Page() self.window?.makeKeyAndVisible() return true } } class Page: UIViewController{ var view1 : UIView! var view2 : UIView! override func viewDidLoad() { super.viewDidLoad() view1 = UIView() view1.backgroundColor = .red view.addSubview(view1) view1.layoutMargins = UIEdgeInsets(top:0, left:0, bottom:0, right:0) view.layoutMargins = UIEdgeInsets(top:0, left:0, bottom:0, right:0) view1.translatesAutoresizingMaskIntoConstraints = false let margins = view.layoutMarginsGuide view1.bottomAnchor.constraint(equalTo:margins.bottomAnchor,constant: -20).isActive = true view1.rightAnchor.constraint(equalTo:margins.rightAnchor,constant: -20).isActive = true view1.heightAnchor.constraint(equalToConstant: 100).isActive = true view1.widthAnchor.constraint(equalToConstant: 100).isActive = true

```
    view2 = UIView()
    view2.backgroundColor = .blue
    view.addSubview(view2)
    view2.translatesAutoresizingMaskIntoConstraints = fals
```

```
e
    let margins2 = view1.layoutMarginsGuide
    view2.topAnchor.constraint(equalTo:margins2.topAnchor)
.isActive = true
    view2.rightAnchor.constraint(equalTo:margins2.leftAnch
or,constant: -20).isActive = true
    view2.heightAnchor.constraint(equalToConstant: 100).is
Active = true
    view2.widthAnchor.constraint(equalToConstant: 100).isA
ctive = true

}
```

}

视图的锚点是通过属性layoutMarginsGuide获得的。比如获得底边的锚点：

```
    let margins = view.layoutMarginsGuide
    let anchor = margins.bottomAnchor
```

代码：

```
    view1.layoutMargins = UIEdgeInsets(top:0, left:0, bott
om:0, right:0)
    view.layoutMargins = UIEdgeInsets(top:0, left:0, botto
m:0, right:0)
```

的目的在于清除视图的内衬边距。默认情况下，视图的边是有厚度的，厚度为8个点。一个视图相对于另外一个视图按锚点定位时，相对的点是内衬的位置。也就是说，如果一个视图的顶边锚点等于另外一个视图的顶边

锚点的话，其实是真正的顶边-8的。清除了内衬的厚度后，对齐的就是真正的顶边了。你不妨删除下这两行代码，会看到两个顶边并不能够对齐，差距的长度就是8。

Visual Layout Language

第三种布局方法是采用Visual Layout Language，就是使用一种特定的用于布局的语言来指定布局关系。这个语言，体现在代码中就是一个有语法的字符串。比如一个对象的宽度为100，水平方向贴着屏幕的右侧，间距为5，那么Visual Layout Language的表达就是：

```
    H:[view1(100)]-5-|
```

这里的：

1.H表示水平方向

1. []标识对象名字
2. ()标识对象尺寸，这里就是宽度
3.
    - 标识下一个对象
4. 数字表示间距，这里的5标识间距为5
5. | 标识屏幕边，这里就是最右侧

读下来就是对象view宽度100，横向距离屏幕右侧为5。完成本章需求的案例，采用Visual Layout Language的方法，代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate {
     var window: UIWindow?
     func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
```

```
OptionsKey: Any]?) -> Bool {
        self.window = UIWindow(frame: UIScreen.main.bounds
)
        self.window!.rootViewController = Page()
        self.window?.makeKeyAndVisible()
        return true
    }
   }
   class Page: UIViewController{
    var view1 : UIView!
    var view2 : UIView!
    override func viewDidLoad() {
        super.viewDidLoad()
        view1 = UIView()
        view1.backgroundColor = .red
        view.addSubview(view1)

        view2 = UIView()
        view2.backgroundColor = .blue
        view.addSubview(view2)
        view2.translatesAutoresizingMaskIntoConstraints =
false
        view1.translatesAutoresizingMaskIntoConstraints =
false
        let views = ["view1":view1,"view2":view2]
         let hConstraint=NSLayoutConstraint.constraints(wi
thVisualFormat: "H:[view1(100)]-5-[view2(100)]-5-|", optio
ns: NSLayoutFormatOptions(rawValue: 0), metrics: nil, view
s: views)
        view.addConstraints(hConstraint)
        let vConstraint1=NSLayoutConstraint.constraints(wi
thVisualFormat: "V:[view1(100)]-5-|", options: NSLayoutFor
matOptions(rawValue: 0), metrics: nil, views: views)
        let vConstraint2=NSLayoutConstraint.constraints(wi
thVisualFormat: "V:[view2(100)]-5-|", options: NSLayoutFor
matOptions(rawValue: 0), metrics: nil, views: views)
```

```
        view.addConstraints(vConstraint1+vConstraint2)
    }
   }
```

方法NSLayoutConstraint.constraints中需要关心两个参数，一个是withVisualFormat指示的就是Visual Layout Language字符串，另外一个就是最后一个参数views，由：

```
    let views = ["view1":view1,"view2":view2]
```

构建，标识的是在Visual Layout Language使用的对象标识和代码中对象实例的对照关系。具体的Visual Layout Language可以参照官方手册。

StackView

有很多布局可以使用横向布局和纵向布局来组合完成，此种情况下，可以使用UIStackView来简化创建布局。

假设我们想要横向布局3个标签，标签之间有边距的话，可以这样：

```
  import UIKit
  @UIApplicationMain
  class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        self.window = UIWindow(frame: UIScreen.main.bounds
)
        self.window!.rootViewController = Page()
        self.window?.makeKeyAndVisible()
        return true
    }
  }
  class Page: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let label1 = UILabel()
        label1.backgroundColor = UIColor.blue
        label1.text  = "Hallo"
        let label2 = UILabel()
        label2.backgroundColor = UIColor.yellow
        label2.text  = "Hi"
        let label3 = UILabel()
        label3.backgroundColor = UIColor.red
        label3.text  = "Hello"
        let stackView   = UIStackView()
```

```
        stackView.axis  = .horizontal
        stackView.spacing   = 6
        stackView.addArrangedSubview(label1)
        stackView.addArrangedSubview(label2)
        stackView.addArrangedSubview(label3)
        stackView.translatesAutoresizingMaskIntoConstraint
s = false;
        self.view.addSubview(stackView)
    }
  }
```

创建标签实例并初始化后，就可以直接丢进UIStackView容器内，此容器实例通过“轴向”属性

```
    .axis  = .horizontal
```

表示为横向布局，加入的视图一个个的水平的排过去。这个属性还可以取值为vertical表示垂直方向排布。而通过属性：

```
    .spacing   = 6
```

说明，各个视图之间的距离为6。比较特别的是，如果想要让UIStackView管理加入控件的布局，必须使用addArrangedSubview，而不是addSubview方法。使用UIStackView布局，你不需要指定视图的位置，也常常不需要指定大小，因为UIKit会帮你自动计算位置和大小。

嵌套使用

只要StackView的横向和纵向布局可以嵌套使用，可以解决很多布局的问题。比如把三个横向的StackView一起放到一个纵向的容器内。假设我们想要做一个案例，它有一个纵向布局，布局内有一个横向布局和一个标签，横向布局中有两个标签。层次结构看起来是这样的：

```
    verticalStackView
    --horizontalStackView
    ----label
    ----label
    --label
```

代码如下：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            self.window!.rootViewController = Page()
            self.window?.makeKeyAndVisible()
            return true
        }
    }
    class Page: UIViewController {
        override func viewDidLoad() {
            super.viewDidLoad()
            var verticalStackView: UIStackView!
            verticalStackView = UIStackView()
            verticalStackView.axis = .vertical
            verticalStackView.distribution = .fillEqually
            verticalStackView.spacing = 10
            let blue = UILabel()
            blue.backgroundColor = .blue
            let green = UILabel()
            green.backgroundColor = .green
```

```
            let horizontalStackView = UIStackView()
            horizontalStackView.axis = .horizontal
            horizontalStackView.distribution = .fillEquall
y
            horizontalStackView.spacing = 10
            horizontalStackView.addArrangedSubview(blue)
            horizontalStackView.addArrangedSubview(green)
            let red = UILabel()
            red.backgroundColor = .red
            red.heightAnchor.constraint(equalToConstant: 3
4).isActive = true
            verticalStackView.addArrangedSubview(horizonta
lStackView)
            verticalStackView.addArrangedSubview(red)
            verticalStackView.frame = view.bounds
            view.addSubview(verticalStackView)
        }
    }
```

就是说，一个StackView内是可以组合另外一个StackView的，这样就可以构成常用的并且比较复杂的布局了。另外属性distribution这里使用了一个新值：

```
    verticalStackView.distribution = .fillEqually
```

指示在StackView内的视图均等的填充全部的空间。此属性还可以取值如下：

1. Fill 让其中一个子视图占据最大空间，其他视图为自然大小。
2. FillEqually 调整每个视图，等量占完全部空间。
3. FillProportionally 每个视图等比的拉伸以便填满空间。比如一个视图长度100，另外一个视图长度200，等需要等比拉伸时，可能一个长度变成150，另外一个变成300
4. EqualSpacing调整视图间的间隙而不是拉伸视图的方法来填满空间

5. EqualCentering会努力让每个子视图的中心点之间是等空间的

Todo App

现在我们来完成一个虽然微小但是比较完整的App，一个管理Todo的App。用户可以使用此App查看Todo列表、删除Todo、添加新的Todo。不妨先把如下代码跑起来，然后尝试一些操作来熟悉此App的功能：

1. 进入App后，会看到Todo列表
2. 点击右上按钮可以进入添加Todo界面，在此界面中，可以录入Todo内容，点击Save按钮；系统会添加此新Todo到列表中，并退回到列表界面，界面上回出现新的Todo项目
3. 在列表上向左手势滑动，然后点击在右侧出现的红色Delete，即可删除此Todo项目

代码如下：

```
    import UIKit
    class TodoListPage : UITableViewController{
        let ReuseId = "cell"
        override func viewDidLoad() {
            super.viewDidLoad()
            tableView.register(UITableViewCell.self, forCe
llReuseIdentifier: ReuseId)
            navigationItem.rightBarButtonItem = UIBarButto
nItem(barButtonSystemItem: .add, target: self, action: #se
lector(addTapped))
            title = "Todo App"
        }
        override func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
            return AppDelegate.count()
        }
        override func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

```
            let a = tableView.dequeueReusableCell(withIden
tifier: ReuseId)
            a!.textLabel?.text = AppDelegate.langAt(indexP
ath.row)
            return a!
        }
        func addTapped (_ sender:UIButton){
            self.navigationController!.pushViewController(
TodoAddPage(), animated: true)
        }
        override func tableView(_ tableView: UITableView,
commit editingStyle: UITableViewCellEditingStyle, forRowAt
 indexPath: IndexPath) {
            if editingStyle ==  .delete{
                AppDelegate.removeTodo(indexPath.row)
            }
        }
        override func tableView(_ tableView: UITableView,
canMoveRowAt indexPath: IndexPath) -> Bool
        {
            return true;
        }
    }
    class TodoAddPage: UIViewController {
        var edit : UITextView!
        override func viewDidLoad() {
            super.viewDidLoad()
            view.backgroundColor = UIColor.gray
            navigationItem.title = "Add Todo Item"
            edit = UITextView()
            edit.becomeFirstResponder()
            edit.backgroundColor = UIColor.blue
            let btn = UIButton()
            btn.backgroundColor = .black
            btn.setTitle("Save", for: UIControlState())
            btn.addTarget(self, action: #selector(save(_:)
```

```
), for: .touchDown)
            let stackView   = UIStackView()
            stackView.axis  = .vertical
            stackView.distribution = .fill
            stackView.spacing   = 5
            stackView.addArrangedSubview(edit)
            stackView.addArrangedSubview(btn)
            stackView.frame = CGRect(x:0,y:80,width:view.b
ounds.width,height:230)
            self.view.addSubview(stackView)
        }
        func save(_ sender: UIButton!){
            AppDelegate.addTodo(edit.text)
            self.navigationController!.popViewController(a
nimated: true)
        }
    }
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        static var Delegate:AppDelegate{
            return UIApplication.shared.delegate as! AppDe
legate
        }
        let arr : NSMutableArray = NSMutableArray(array: [
"Item 1","Item 2","Item 3"])
        var nav :UINavigationController?
        var p : TodoListPage!
        var window: UIWindow?
        class func langAt (_ index : Int)-> String{
            let appDelegate = UIApplication.shared.delegat
e as! AppDelegate
            return appDelegate.arr[index] as! String
        }
        class func count ()-> Int{
            let appDelegate = UIApplication.shared.delegat
```

```
e as! AppDelegate
            return appDelegate.arr.count
        }
        class func addTodo (_ item : String){
            let appDelegate = UIApplication.shared.delegat
e as! AppDelegate
            appDelegate.arr.add(item)
            appDelegate.p.tableView.reloadData()
        }
        class func removeTodo (_ index : Int){
            let appDelegate = UIApplication.shared.delegat
e as! AppDelegate
            appDelegate.arr.removeObject(at: index)
            appDelegate.p.tableView.reloadData()
        }
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            self.window = UIWindow(frame: UIScreen.main.bo
unds)
            p = TodoListPage()
            nav = UINavigationController(rootViewControlle
r: p)
            self.window!.rootViewController = nav
            self.window?.makeKeyAndVisible()
            return true
        }
    }
```

部分需要说明的技术细节是这样的：

1. 列表使用了UITableViewController
2. 需要删除时，交互界面采用的是UITableViewController内置的删除方式，就是通过手势划过的方式来引发删除动作
3. 添加新的Todo的界面，使用了UIViewController，其中的输入视图和

保存按钮使用了StackView布局来管理它们的位置和大小
4. 使用了UINavigationController做导航

整个UI体系的视图层次图是这样的：

```
--AppDelegate:UIApplicationDelegate
----UINavigationController
------TodoListPage : UITableViewController
------TodoAddPage: UIViewController
--------edit : UITextView!
--------btn  : UIButton
```

这一次，数据存储使用了NSMutableArray，一个可修改的数组，并且此数组放置于AppDelegate类内，为了方便别的类访问此数据源，AppDelegate提供了三个方法，分别是提供了数组的大小、添加一个项目到数据、删除一个指定位置的项目，也就是这些函数：

```
class func count ()-> Int
class func addTodo (_ item : String)
class func removeTodo (_ index : Int)
```

函数前的class表明它们是类函数，这样就可以使用类名来引用此类函数了，就像这样：

```
AppDelegate.count()
```

相应的，TodoListPage作为一个UITableViewController的子类，它访问的数据源也就可以方便的从AppDelegate中获得了。

类UINavigationController可以在顶部bar上加入右侧按钮，这里使用了系统提供的.add按钮，也就是一个标题为“+”的按钮，在TodoListPage类内使用：

```
    navigationItem.rightBarButtonItem = UIBarButtonItem(ba
rButtonSystemItem: .add, target: self, action: #selector(a
ddTapped))
```

即可添加，在函数addTapped中使用了UINavigationController.pushViewController函数，把TodoAddPage推入到UINavigationController内，也就是说，页面TodoAddPage会称为当前可见的ViewController；在TodoAddPage内当点击Save按钮时，又会使用到UINavigationController.popViewController，弹出当前ViewController，从而回到前一个ViewController。pushViewController函数和popViewController函数常常是成对使用的。

TodoAddPage页面内，使用了StackView来做布局:

```
            stackView.axis  = .vertical
            stackView.distribution = .fill
```

布局方案是纵向布局，分布选项为.fill填充方案，从而让一个子视图占据最大空间，其他子视图占据自然尺寸。

持久化

所谓的持久化，就是将数据保存到永久存储中，使得App重启后可以继续访问。在iOS开发中，有很多数据持久化的方案，接下来我将尝试着介绍一下5种方案：

1. plist文件（属性列表）
2. preference（偏好设置）
3. NSKeyedArchiver（归档）
4. SQLite 3
5. CoreData

存储方法最终都会把数据存储在沙盒中。iOS程序默认情况下只能访问程序自己的目录，这个目录被称为“沙盒”。

既然沙盒就是一个文件夹，其中最常用的就是Documents目录，适合存储重要数据。之后的sqlite和coredata案例都会使用此目录来做数据库存储。可以使用FileManager类来访问此目录，管理其中的文件。如下代码：

```
    let urls = FileManager.default.urls(for: .documentDire
ctory, in: .userDomainMask)
    let docURL = urls[urls.count-1]
    let url = docURL.appendingPathComponent(dbname)
```

可以获得Documents的URL，并使用appendingPathComponent，创建一个在此目录下的文件URL。

其中的第二个参数，通常取值为

```
 userDomainMask - 用户目录
```

完整的取值列表为：

```
userDomainMask - 用户目录
localDomainMask - 本机
networkDomainMask - 局域网内
systemDomainMask - 指向/System
allDomainsMask - 以上全部
```

访问sqlite

可以使用sqlite作为关系数据库来存储iOS本地数据。这意味着，通过sql语言方便的做数据的增删改查。

sqlite本身提供了C语言的API，使用Swift访问需要

1. 创建一个桥接文件，引入C API，访问SQLite数据库需要使用SQLite官方提供的C语言风格的API，所以需要添加桥接文件
2. 引入sqlite的动态链接库

首先我们让Swift到Sqlite的道路打通。

访问sqlite的C语言库

创建一个Single View App之后，需要一些令人厌倦的操作的过程

1、点击项目名称 - 查看General页面 - Linked Frameworks and Libraries - “+” - 搜索“libsqlite3.dylib” - 然后点击Add

1. 新建一个头文件，命名为SQLite-Bridge.h。具体过程为：

```
 右击项目名称 - New File… - Header File - 命名为“SQLite-Bridge.h”,
```

并在这个头文件中加一行代码

```
#import "sqlite3.h"
```

注意，此处包含的头文件，是sqlite3.h，而不是sqlite.h。

1. 把此头文件设置为桥接文件。首先定位到修改点，操作过程为：

```
 点击项目名称 - Build Settings - 点击All - 点击Combined -
单击“Objective-C Bridging Header”（可以通过搜索“bridging”
快速定位此配置项） - 双击后面后弹出添加文件名，把刚刚创建的头文
件名称写进去- 然后回车
```

2. 编译。如果编译通过，那么Swift到sqlite的链接过程就算对了。

一个sqlite访问的封装和使用案例

为了方便让Swift App使用，做一个封装是必要的。封装代码可以让开发者只要使用此查询执行函数、而不必关心sqlite的C API的底层细节，就可以和数据库完成交互。如下案例，不仅包括封装于类Sqlite的代码，也包括使用此代码的使用方法。案例中创建了一个todo的表，并且在其中插入一条数据，然后做一个查询，打印查询出来的结果：

```
    import UIKit
    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window : UIWindow?
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            window = UIWindow()
            window!.rootViewController = UIViewController(
)
            window!.rootViewController!.view.backgroundCol
or = .blue
            window!.makeKeyAndVisible()
            let dbname = "mysqlite"
            let db = Sqlite(dbname)
            if db.createDatabase(){
                print("创建数据库 OK")
```

```
            }
            print(db.ExecuteQuery( "CREATE TABLE IF NOT EX
ISTS 'todo' ('id' integer NOT NULL PRIMARY KEY AUTOINCREME
NT, 'item' TEXT);"))
            print(db.ExecuteQuery( "insert into todo(item)
values('1');"))
            print(db.SelectQuery( "select * from todo;"))
            return true
        }
    }
    class Sqlite: NSObject
    {
        var dbname :String!
        var DBpath :String!
        init(_ dbname:String){
            super.init()
            self.dbname = dbname
            self.DBpath = self.databasePath()
        }
        func createDatabase()->Bool
        {
            var success:Bool=false
            print(DBpath)
            if (FileManager.default.fileExists(atPath: DBp
ath))
            {
                success = true
            }
            else
            {
                let pathfrom:String=(Bundle.main.resourceP
ath! as NSString).appendingPathComponent(dbname)
                do {
                    try FileManager.default.copyItem(atPat
h: pathfrom, toPath: DBpath)
                    success = true
```

```
                } catch _ {
                    success = false
                }
            }
            return success
        }
        func databasePath() -> String
        {
            var path:Array=NSSearchPathForDirectoriesInDom
ains(FileManager.SearchPathDirectory.documentDirectory, Fi
leManager.SearchPathDomainMask.userDomainMask, true)
            let directory=path[0]
            let DBpath=(directory as NSString).appendingPa
thComponent(dbname)
            return DBpath as String
        }
        func ExecuteQuery(_ str:String) -> Bool
        {
            var result:Bool=false
            var db: OpaquePointer? = nil
            var stmt:OpaquePointer? = nil
            let strExec=str.cString(using: String.Encoding
.utf8)
            if (sqlite3_open(DBpath, &db)==SQLITE_OK)
            {
                if (sqlite3_prepare_v2(db, strExec! , -1,
&stmt, nil) == SQLITE_OK)
                {
                    if (sqlite3_step(stmt) == SQLITE_DONE)
                    {
                        result=true
                    }
                }
                sqlite3_finalize(stmt)
            }
            sqlite3_close(db)
```

```
            return result
        }
        func SelectQuery(_ str:String) -> Array<Dictionary
<String,String>>
        {
            var result:Array<Dictionary<String,String>>=[]
            var db: OpaquePointer? = nil
            var stmt:OpaquePointer? = nil
            let strExec=str.cString(using: String.Encoding
.utf8)
            if ( sqlite3_open(DBpath,&db) == SQLITE_OK)
            {
                if (sqlite3_prepare_v2(db, strExec! , -1,
&stmt, nil) == SQLITE_OK)
                {
                    while (sqlite3_step(stmt) == SQLITE_RO
W)
                    {
                        var i:Int32=0
                        let icount:Int32=sqlite3_column_co
unt(stmt)
                        var dict=Dictionary<String, String
>()
                        while i < icount
                        {
                            let strF=sqlite3_column_name(s
tmt, i)
                            let strV = sqlite3_column_text
(stmt, i)
                            let rFiled:String=String(cStri
ng: strF!)
                            let rValue:String=String(cStri
ng: strV!)
                            dict[rFiled] = rValue
                            i += 1
                        }
```

```
                            result.insert(dict, at: result.cou
nt)
                        }
                        sqlite3_finalize(stmt)
                    }
                    sqlite3_close(db)
                }
                return result
            }
        }
```

作为程序员，可以直接使用此类做数据库的访问，如果对sqlite内部C API的使用感兴趣，则可以进一步的阅读此类的实现部分。

Core Data

Core Data是官方对Sqlite访问的封装框架。使用它的好处是：

1. 不需要自己引入Sqlite动态库和创建桥接文件
2. 不需要使用SQL语言即可访问Sqlite

使用它的首要需求是引入它，像是这样：

```
    import CoreData
```

依然假设我们的问题是存储todo项目，字段为id、item，希望创建数据库、创建表、插入数据、查询数据一气呵成，代码如下：

```
    import UIKit
    import CoreData

    @UIApplicationMain
    class AppDelegate: UIResponder, UIApplicationDelegate
{
        var window: UIWindow?
        var context :NSManagedObjectContext!
        func application(_ application: UIApplication, did
FinishLaunchingWithOptions launchOptions: [UIApplicationLa
unchOptionsKey: Any]?) -> Bool {
            let dbname = "foo.bar"
            // section : model
            let model = NSManagedObjectModel()
            let entity = NSEntityDescription()
            entity.name = "todo"
            var properties = Array<NSAttributeDescription>
()
            let idAttr = NSAttributeDescription()
            idAttr.name = "id"
```

```
            idAttr.attributeType = .integer64AttributeType
            idAttr.isOptional = false
            idAttr.isIndexed = true
            properties.append(idAttr)
            let itemAttr = NSAttributeDescription()
            itemAttr.name = "item"
            itemAttr.attributeType = .stringAttributeType
            itemAttr.isOptional = false
            itemAttr.isIndexed = true
            properties.append(itemAttr)
            entity.properties = properties
            model.entities = [entity]

            // section : coordinator
            let coordinator = NSPersistentStoreCoordinator
(managedObjectModel: model)
            let urls = FileManager.default.urls(for: .docu
mentDirectory, in: .userDomainMask)
            let docURL = urls[urls.count-1]
            let url = docURL.appendingPathComponent(dbname
)

            do {
                try FileManager.default.removeItem(at: url
)
            } catch _ {
            }
            do {
                try coordinator.addPersistentStore(ofType:
 NSSQLiteStoreType, configurationName: nil, at: url, optio
ns: nil)
            } catch {
                print("caught: \(error)")
                abort()
            }
            //section : context
```

```
            context = NSManagedObjectContext(concurrencyTy
pe: .mainQueueConcurrencyType)
            context.persistentStoreCoordinator = coordinat
or
            // section : app code

            let todos = [
                (item: "Item 1",id:1),
                (item: "Item 2" ,id:2),
                (item: "Item 3",id:3)
            ]
            for todo in todos {
                let a =  NSEntityDescription.insertNewObje
ct(forEntityName: "todo", into: context)
                a.setValue(todo.item, forKey:"item")
                a.setValue(todo.id, forKey:"id")
            }
            do {
                try context.save()
            } catch _ {
            }
            let fetchRequest = NSFetchRequest<NSFetchReque
stResult>(entityName:"todo")
            //        fetchRequest.predicate = NSPredicate
(format: "notificationId = 13")

            do {
                let list = try context.fetch(fetchRequest)
 as? [NSManagedObject]
                print(list!.count)
                for zoo in list! {
                    print(zoo.value(forKey: "item")!,zoo.v
alue(forKey: "id")!)
                }
            } catch let error as NSError {
                // failure
```

```
                print("Fetch failed: \(error.localizedDesc
ription)")
            }
            window?.rootViewController  = UIViewController
()
            return true
        }
        func applicationWillTerminate(_ application: UIApp
lication) {
            if context.hasChanges {
                do {
                    try context.save()
                } catch {
                    print("caught: \(error)")
                    abort()
                }
            }
        }
    }
```

完成要求的一系列的动作的过程中，首先我们来看创建数据库的模型，也就是标注为：

```
    // section : model
```

开始的代码，它使用了如下的类：

1. NSManagedObjectModel
2. NSEntityDescription
3. NSAttributeDescription

既然是对Sqlite的封装，那么这些类显然分别对应于数据库、表、字段的。所以尽管代码看起来有些冗长，意图倒是比较明显的。它们等同于创建这样的表：

```
    表名: todo
    表字段清单:
        名字  类型 可选 索引
        id   int64 否  是
        item string否  是
```

此处的代码只是创建了一个内存中的模型，真正的创建使用了NSPersistentStoreCoordinator，在标注为：

```
    // section : coordinator
```

开始的代码中进行。NSPersistentStoreCoordinator用来把模型和真实的存储关联起来。NSPersistentStoreCoordinator的物理存储使用了Sqlite（NSSQLiteStoreType），但是也可以使用两位两种存储，分别为

// public let NSBinaryStoreType: String // public let NSInMemoryStoreType: String

代码中对FileManager的引用的目的就是在文件存储中关联一个文件，如果存在就会删除它。接下来的NSManagedObjectContext可以调用它的save()方法，把当前上下文的数据存储起来，也可以使用fetch方法，通过NSFetchRequest指定表名和条件来查询数据。如果需要插入数据，则需要使用NSEntityDescription的insertNewObject()方法。

这只是一个案例，真正的应用有可能正在编辑数据时，App被退出，那么数据将不能被保存。如果需要在退出前保存的话，需要在：

```
    func applicationWillTerminate(_ application: UIApplica
tion)
```

检测变化，并调用NSManagedObjectContext.save()来保存数据。

plist

plist是一个XML的子集，使用UTF-8编码的文本文件，它的根节点是一个字典，内容由多个主键和值构成。常常用来存储App的配置信息。

可以直接使用NSDictionary类直接存储它的内容到plist内，比如这样：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = Page()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
}
class Page: UIViewController {
    let filename = "/profile.plist"
    let data:[String:String] = ["key1" : "value1", "key2":
"value2", "key3":"value3"]
    override func viewDidLoad() {
        super.viewDidLoad()
        bar()
    }
    func bar(){
        do {
            let documentDirectory = NSSearchPathForDirecto
riesInDomains(.documentDirectory, .userDomainMask, true)[0
```

```
] as String
            let path = "\(documentDirectory)/\(filename)"
            print(path)
            let dict = NSDictionary(dictionary: data)
            let isWritten = dict.write(toFile: path, atomi
cally: true)
            let dict1 = NSDictionary(contentsOfFile: path)
            print(dict1)
            print("file created: \(isWritten)")
            let text = try String(contentsOfFile: path, en
coding: String.Encoding.utf8)
            print(text)
        }catch {print("\(error)")}

    }
}
```

类NSDictionary的方法write可以指定文件名，然后写入到指定文件。初始化方法 NSDictionary(contentsOfFile:)可以读取plist到词典对象内。

因为plist是UTF-8编码的文本文件，所以，可以使用String打开此文件，输出内容如下（去掉主题无关的文件头后）：

```
<plist version="1.0">
<dict>
    <key>key1</key>
    <string>value1</string>
    <key>key2</key>
    <string>value2</string>
    <key>key3</key>
    <string>value3</string>
</dict>
</plist>
```

作为键值对的值，不仅仅可是是字符串，还可以是数字、日期、数组和词典。如下案例展示了plist存储的更多可能性：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = Page()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
}
class Page: UIViewController {
    let filename = "/profile.plist"
    let data:[String:Any] = ["key1" : "value1", "key2":["k
ey11":"value11"], "key3":[1,"2",NSDate(),3.1]]
    override func viewDidLoad() {
        super.viewDidLoad()
        bar()
    }
    func bar(){
        do {
            let documentDirectory = NSSearchPathForDirecto
riesInDomains(.documentDirectory, .userDomainMask, true)[0
] as String
            let path = "\(documentDirectory)/\(filename)"
            print(path)
            let dict = NSDictionary(dictionary: data)
            let isWritten = dict.write(toFile: path, atomi
cally: true)
```

```
            let dict1 = NSDictionary(contentsOfFile: path)
            print(dict1)
            print("file created: \(isWritten)")
            let text = try String(contentsOfFile: path, en
coding: String.Encoding.utf8)
            print(text)
        }catch {print("\(error)")}
    }
}
```

Archive

想把对象序列化到文件，可以首先令此对象符合NSCoding协议，然后使用归档类完成序列化。

如下案例，可以把对象User序列化到文件内：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = Page()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
}
class User : NSObject,NSCoding{
    var id : Int32
    var name : String
    init(_ id: Int32 ,_ name : String) {
        self.id = id
        self.name = name
    }
    required convenience init?(coder decoder: NSCoder) {
        guard let id = decoder.decodeInt32(forKey: "id") a
s? Int32,
            let name = decoder.decodeObject(forKey:"name")
 as? String
```

```
            else { return nil }

        self.init(id,name)
    }
    func encode(with coder: NSCoder) {
        coder.encode(self.id, forKey: "id")
        coder.encode(self.name, forKey: "name")
    }
    override public var description: String { return "{id:
\(id),name:\(name)}" }
}
class Page: UIViewController {
    let filename = "/todo.archive"
    override func viewDidLoad() {
        super.viewDidLoad()
        bar()
    }
    func bar(){
        do {
            let documentDirectory = NSSearchPathForDirecto
riesInDomains(.documentDirectory, .userDomainMask, true)[0
] as String
            let path = "\(documentDirectory)\(filename)"
            let u = User(1,"user")
            NSKeyedArchiver.archiveRootObject(u, toFile: p
ath)
            let user1 = NSKeyedUnarchiver.unarchiveObject(
withFile: path) as? User
            print(user1)
        }catch {print("\(error)")}
    }
}
```

运行后输出的结果:

```
Optional({id:1,name:user})
```

协议NSCoding需要实现两个方法：

```
public protocol NSCoding {
    public func encode(with aCoder: NSCoder)
    public init?(coder aDecoder: NSCoder) // NS_DESIGNATED
_INITIALIZER
}
```

方法encode(with:)是从NSCoder内解码数据，方法init?(coder:)是编码数据到NSCoder内。大体上函数内的实现就是一个个的用类似的代码把当前对象的成员做编码和解码。

然后这个对象就可以传递给函数：

```
    NSKeyedArchiver.archiveRootObject
```

完成归档，然后使用：

```
    NSKeyedUnarchiver.unarchiveObject(withFile: path) as?
User
```

完成反归档。

更强大的是，如果对象内有成员指向到其他对象，归档类可以把这些指向对象一起归档，比如Man类通过成员wife指向一个Woman类，那么可以连同此成员对象一起做归档。方法如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = Page()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
}
class Man : NSObject,NSCoding{
    var id : Int32
    var name : String
    var wife : Woman
    init(_ id: Int32 ,_ name : String,_ wife : Woman) {
        self.id = id
        self.name = name
        self.wife = wife
    }
    required convenience init?(coder decoder: NSCoder) {
        guard let id = decoder.decodeInt32(forKey: "id") a
s? Int32,
            let name = decoder.decodeObject(forKey:"name")
 as? String,
            let todo = decoder.decodeObject(forKey:"wife")
 as? Woman
            else { return nil }

        self.init(id,name,todo)
    }
    func encode(with coder: NSCoder) {
        coder.encode(self.id, forKey: "id")
        coder.encode(self.name, forKey: "name")
        coder.encode(wife,forKey:"wife")
```

```
    }
    override public var description: String { return "Man:
{id:\(id),name:\(name)},\(wife)" }
}

class Woman : NSObject,NSCoding{
    var id : Int32
    var name : String
    init(_ id: Int32 ,_ name : String) {
        self.id = id
        self.name = name
    }
    required convenience init?(coder decoder: NSCoder) {
        guard let id = decoder.decodeInt32(forKey: "id") a
s? Int32,
            let name = decoder.decodeObject(forKey:"name")
 as? String
            else { return nil }

        self.init(id,name)
    }
    func encode(with coder: NSCoder) {
        coder.encode(self.id, forKey: "id")
        coder.encode(self.name, forKey: "name")
    }
    override public var description: String { return "wife
:{id:\(id),name:\(name)}" }
}

class Page: UIViewController {
    let filename = "/man.archive"
    override func viewDidLoad() {
        super.viewDidLoad()
        bar()
    }
```

```
    func bar(){
        do {
            let documentDirectory = NSSearchPathForDirecto
riesInDomains(.documentDirectory, .userDomainMask, true)[0
] as String
            let path = "\(documentDirectory)\(filename)"
            let u = Man(1,"Man1",Woman(1,"Wife1"))
            NSKeyedArchiver.archiveRootObject(u, toFile: p
ath)
            let user1 = NSKeyedUnarchiver.unarchiveObject(
withFile: path) as? Man
            print(user1)
        }catch {print("\(error)")}
    }
}
```

要点在于，在NSCoder协议需要的两个函数中，通过NSCoder对象，归档和反归档此成员对象，并且此成员对象也必须指向NSCoder协议即可。

如果成员对象是一个对象的数组，做法是类似的。如下案例有两个类，Department和Task。Department通过成员Tasks包含一个Task的数组。把Task数组整体作为一个对象来处理，使用NSCoder.encode函数即可做归档，使用NSCoder.decodeObject即可反归档：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        window = UIWindow()
        window!.rootViewController = Page()
        window!.rootViewController!.view.backgroundColor =
 .blue
```

```
        window!.makeKeyAndVisible()
        return true
    }
}
class Department: NSObject, NSCoding {
    var name = ""
    var manager = ""

    var tasks: [Task]?

    func encode(with aCoder: NSCoder) {
        aCoder.encode(name, forKey: "name")
        aCoder.encode(manager, forKey: "manager")
        aCoder.encode(tasks, forKey: "taskArray")
    }

    required convenience init?(coder aDecoder: NSCoder){
        self.init()

        name = aDecoder.decodeObject(forKey: "name") as! S
tring
        manager = aDecoder.decodeObject(forKey: "manager")
 as! String
        tasks = aDecoder.decodeObject(forKey: "taskArray")
 as? [Task]
    }
    override public var description: String { return "dept
:{name:\(name),tasks:\(tasks)" }
    override init() {
        super.init()
        name = "D1"
        manager = "TJ"
        tasks = []
        tasks?.append(Task("A1","N1"))
        tasks?.append(Task("A2","N2"))
        tasks?.append(Task("A3","N3"))
```

```
    }
}

class Task: NSObject, NSCoding {
    var title = ""
    var notes = ""
    override public var description: String { return "{tit
le:\(title)}" }
    func encode(with aCoder: NSCoder)  {
        // Methods
        aCoder.encode(title, forKey: "title")
        aCoder.encode(notes, forKey: "notes")
    }

    required convenience init?(coder aDecoder: NSCoder) {
        // Methods

        let title = aDecoder.decodeObject(forKey: "title")
 as! String
        let notes = aDecoder.decodeObject(forKey: "notes")
 as! String
        self.init(title,notes)
    }

    init(_ title:String, _ notes : String) {
        self.title = title
        self.notes = notes
        super.init()
    }
}

class Page: UIViewController {
    let filename = "/man.archive"
    override func viewDidLoad() {
        super.viewDidLoad()
        bar()
```

```
    }
    func bar(){
        do {
            let documentDirectory = NSSearchPathForDirecto
riesInDomains(.documentDirectory, .userDomainMask, true)[0
] as String
            let path = "\(documentDirectory)\(filename)"
            let u = Department()
            NSKeyedArchiver.archiveRootObject(u, toFile: p
ath)
            let user1 = NSKeyedUnarchiver.unarchiveObject(
withFile: path) as? Department
            print(user1)
        }catch {print("\(error)")}
    }
}
```

在控制台上期望输出的是：

```
Optional(dept:{name:D1,tasks:Optional([{title:A1}, {title:
A2}, {title:A3}]))
```

CocoaPods

CocoaPods是一个iOS项目的依赖管理器，使用它可以让导入第三方库和处理依赖关系变得简单。

出于测试和验证的目的，这里会创建一个Swift工程，并采用CocoaPods导入第三方HTTP库alamofire。

准备CocoaPods

CocoaPods需要系统内已经安装了ruby，如果没有安装，请首先安装它。可以使用如下命令：

```
sudo gem install cocoapods
```

安装此工具。随即使用：

```
pod setup --verbode
```

做配置。命令执行完毕，cocoapods就是可用的了。

创建一个演示工程

步骤如下：

1. 打开xcode
2. 点击“Create a new Xcode project”
3. 选择Single View App
4. 填写product name为poddemo；填写语言为Swift
5. 设置目录

完成创建后，退出xcode

初始化

打开Terminal，导航到工程目录，执行命令：

```
pod init
```

此命令会在目录内创建Podfile文件。接下来使用xcode打开Podfile文件：

```
open -a Podfile
```

加入alamofire文件的依赖，修改后的Podfile为：

```
target 'poddemo' do
  # Comment the next line if you're not using Swift and do
n't want to use dynamic frameworks
  use_frameworks!

  # Pods for poddemo
  pod 'Alamofire', '~> 4.4'
end
```

退出xcode，在terminal内执行命令：

```
pod install
```

安装Alamofire，并且创建一个扩展名为xcworkspace的文件。

使用Alamofire访问HTTP

再次使用xcoce打开xcworkspace文件。编辑AppDelegate.swift为：

```
import UIKit
import Alamofire
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        foo()
        window = UIWindow()
        window!.rootViewController = UIViewController()
        window!.rootViewController!.view.backgroundColor =
 .blue
        window!.makeKeyAndVisible()
        return true
    }
    func foo(){
        Alamofire.request("https://httpbin.org/get").respo
nseJSON { response in
            print(response.request)
            print(response.response)
            print(response.data)
            print(response.result)
            if let JSON = response.result.value {
                print("JSON: \(JSON)")
            }
        }
    }
}
```

编译运行，期望的输出为：

```
Optional(https://httpbin.org/get)
```

```
Optional(<NSHTTPURLResponse: 0x600000222840> { URL: https:
//httpbin.org/get } { status code: 200, headers {
    "Access-Control-Allow-Credentials" = true;
    "Access-Control-Allow-Origin" = "*";
    Connection = "keep-alive";
    "Content-Length" = 359;
    "Content-Type" = "application/json";
    Date = "Wed, 26 Apr 2017 01:15:59 GMT";
    Server = "gunicorn/19.7.1";
    Via = "1.1 vegur";
} })
Optional(359 bytes)
SUCCESS
JSON: {
    args =     {
    };
    headers =     {
        Accept = "*/*";
        "Accept-Encoding" = "gzip;q=1.0, compress;q=0.5";
        "Accept-Language" = "en;q=1.0";
        Connection = close;
        Host = "httpbin.org";
        "User-Agent" = "poddemo/1.0 (home.poddemo; build:1
; iOS 10.2.0) Alamofire/4.4.0";
    };
    origin = "221.237.156.243";
    url = "https://httpbin.org/get";
}
```

如果输出是对的，就说明通过CocoaPods导入的第三方库已经成功。

http

访问https

访问http或者https，使用URLSession和它的一组相关类。

为了测试方便，省下自己编写https服务器的麻烦，可以使用一个网络服务叫做https://httpbin.org/ip,当访问此URL时，它会返回一个json，格式为：

```
{
    origin = "221.237.156.243";
}
```

访问https代码如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,URLS
essionDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        foo()
        return true
    }
    func foo(){
        let task = URLSession.shared.dataTask(with: URL(st
ring: "https://httpbin.org/ip")!) { (data, response, error
) in
            if error != nil {
                print(error)
            } else {
```

```
                    if let usableData = data {
                        do {
                            let json = try JSONSerialization.j
sonObject(with: usableData, options:[])
                            print("json: \(json)")
                        }
                        catch {
                            print("Error: \(error)")
                        }
                    }
                }
            }
            task.resume()
        }
    }
```

通过URLSession.shared创建一个单例的URLSession，并发起一个数据任务到指定的URL，然后就可以在函数的完成代码块内接受响应，并通过

```
    JSONSerialization.jsonObject
```

解析返回的json。

每个应用可以创建若干个会话（URLSession），每个URLSession可以关联一组任务（NSURLSessionTask）

NSURLSessionTask可以是如下类型：

1. Data Task。使用NSData对象发送和接收数据。此类型任务针对与服务器简短的、经常交互的请求。能够一次返回一小段数据，或者通过完成处理一次返回
2. Upload Task。以文件形式发送数据，它可以在后台上传
3. Download Task。以文件形式获得数据，它可以在后台下载

类似的，可以使用Download Task，通过https下载一个服务器文件到本地。如下案例，下载一个https://httpbin.org/提供的图片:

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,URLS
essionDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        baz()
        return true
    }
    func baz(){
        let request = URLRequest(url:  URL(string: "https:
//httpbin.org/image/png")!)
        let downloadTask =  URLSession.shared.downloadTask
(with: request){ (location, response, error) in
            if let httpResponse = response as? HTTPURLResp
onse {
                print("status code: \(httpResponse.statusC
ode)")
            }
            let file:String = NSHomeDirectory() + "/Docume
nts/1.png"
            let fileManager = FileManager.default
            if fileManager.fileExists(atPath: file) {
                try! fileManager.removeItem(atPath: file)
            }
            try! fileManager.moveItem(atPath:  location!.p
ath, toPath: file)
            print("location:\(file)")
        }
        downloadTask.resume()
    }
```

```
}
```

代码执行后，会通过URLSession.shared.downloadTask函数发起一个下载任务，下载文件到临时目录。随后属于FileManager移动此文件到指定的目录和名字，可以通过此文件名打开验证图片是否正确。

如果想要查看下载进度，就需要指定下载委托，实现它指定的下载进度函数作为事件：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,URLS
essionDownloadDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        foo()
        return true
    }
    private lazy var session:URLSession = {
        return URLSession(configuration: .default, delegat
e: self,
                                          delegateQueue: nil
)
    }()
    func foo(){
        let url = URL(string: "https://httpbin.org/bytes/1
000000000")
        let request = URLRequest(url: url!)
        let downloadTask = session.downloadTask(with: requ
est)
        downloadTask.resume()
    }
```

```
    func urlSession(_ session: URLSession, downloadTask: U
RLSessionDownloadTask,
                    didFinishDownloadingTo location: URL)
{
        print("downloaded to file:\(location)")
    }
    func urlSession(_ session: URLSession, downloadTask: U
RLSessionDownloadTask,
                    didWriteData bytesWritten: Int64, tota
lBytesWritten: Int64,
                    totalBytesExpectedToWrite: Int64) {
        print("downloadeing : %\(Int(floor((CGFloat)(total
BytesWritten)/(CGFloat)(totalBytesExpectedToWrite)*100)))"
)
    }
}
```

这个案例自己创建了URLSession实例，指定配置类型为.default，指定委托为当前类，并且要求当前类AppDelegate实现协议URLSessionDownloadDelegate，在协议指定的函数：

```
    func urlSession(_ session: URLSession, downloadTask: U
RLSessionDownloadTask,
                    didWriteData bytesWritten: Int64, tota
lBytesWritten: Int64,
                    totalBytesExpectedToWrite: Int64)
```

会得到进度的通知。这里再次使用https://httpbin.org/服务，URL为/bytes/1000000000，它可以生产一个随机的指定大小的文件，这里的大小为1000000000。目的是为了稍微大点的文件，以便看到进度事件被调用。

这次也看到了URLSession的Default配置类型，完整的配置类型为：

1. Shared : 单例类型。用于最简单的URL访问。不允许传递委托和配置

对象，因此会有不少的限制
2. Default：类似单例类型，但是可以定制委托和配置对象。且所有的缓存、凭证等等都与会话绑定存储在硬盘内
3. Ephemeral：所有的缓存、凭证等等都与会话绑定存储在内存中
4. Background：类似于Default类型，除了一个独立的进程处理所有的数据传输，这意味着即使App进入后台工作或者是崩溃了也可以继续下载任务。

除了URLSessionDownloadDelegate，还有更多可以使用的委托，采用它们可以处理会话级、任务级事件。协议包括：

```
URLSessionDelegate          会话委托。处理会话级事件
URLSessionTaskDelegate      会话任务委托。处理任务级事件
URLSessionDataDelegate      数据委托。处理数据任务级事件
URLSessionDownloadDelegate 下载委托。处理下载任务级事件
URLSessionStreamDelegate    流委托。处理流任务级事件
```

比如：NSURLSessionDelegate协议内的方法URLSession:didReceiveChallenge:completionHandler:可以用来定制认证。在访问https服务器时就需要处理此事件，以便可以接受服务器使用自签名证书。

创建https和http服务器

有时候，需要创建一个https服务器，以便测试使用。使用node.js完成此项工作还是比较简单的。另外，为了可以创建https，需要openssl工具栈。

首先创建证书：

```
openssl genrsa -out key.pem
openssl req -new -key key.pem -out csr.pem
openssl x509 -req -days 9999 -in csr.pem -signkey key.pem
-out cert.pem
rm csr.pem
```

然后可以看到生成的两个文件：

```
$ls *.pem
cert.pem    key.pem
```

创建node.js文件

编辑js源代码文件（server.js)

```
var https = require('https');
var fs = require('fs');
var options = {
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')
};
var a = https.createServer(options, function (req, res) {
  res.writeHead(200);
```

```
  res.end(JSON.stringify({foo:"bar"}))
}).listen(8000);
```

尝试

运行服务器:

```
node server.js
```

然后访问:

```
https://localhost:8000
```

可以看到json文件的话，就算成功。

如果是http服务器

这样情况，不需要证书，代码改成下面的就可以了。

```
const http = require('http')
const port = 8000
const requestHandler = (request, response) => {
  console.log(request.url)
  response.end(JSON.stringify({foo:"bar"}))
}
const server = http.createServer(requestHandler)
server.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err)
  }
  console.log(`server is listening on ${port}`)
})
```

访问自签名证书https服务器

访问HTTPS服务器时，可以使用自签名证书的、在本地的https服务器，对于调试应用是非常方便的。

Apple已经在iOS 9版本要求必须使用https。如果https服务器是CA签署的证书，那么一路绿灯，如果是自签名证书，就需要做两个额外的工作：

1. 在info.plist内加入一个NSAppTransportSecurity|NSAllowsArbitraryLoads关键字，指明可以任意加载内容

   ```
   <key>NSAppTransportSecurity</key>
    <dict>
        <key>NSAllowsArbitraryLoads</key>
        <true/>
    </dict>
   ```

2. 通过URLSessionDelegate，指明信任服务器证书

代码如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,URLS
essionDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        foo()
        return true
    }
    lazy var ss : URLSession  = {
```

```
        let config = URLSessionConfiguration.default
        let session = Foundation.URLSession(configuration:
 config, delegate: self, delegateQueue: OperationQueue.mai
n)
        return session

    }()
    func foo(){
        let urlString = URL(string: "https://localhost:800
0")
        if let url = urlString {
            let task = ss.dataTask(with: url) { (data, res
ponse, error) in
                if error != nil {
                    print(error)
                } else {
                    if let usableData = data {
                        //                        print(us
ableData) //JSONSerialization
                        do {
                            let json = try JSONSerializati
on.jsonObject(with: usableData, options:[])
                            print("json: \(json)")
                        }
                        catch {
                            print("Error: \(error)")
                        }
                    }
                }
            }
            task.resume()
        }
    }
    public func urlSession(_ session: URLSession, didRecei
ve challenge: URLAuthenticationChallenge, completionHandle
```

```
r: @escaping (URLSession.AuthChallengeDisposition, URLCred
ential?) -> Swift.Void)
    {
        completionHandler(.useCredential, URLCredential(tr
ust: challenge.protectionSpace.serverTrust!))
    }
}
```

node.js服务器代码采用“创建https和http服务器”一节的代码。执行后，输出应该是：

```
json: {
    foo = bar;
}
```

访问http

访问http也是使用URLSession来完成的。因为iOS 9之后的版本默认情况下限定只能访问https，因此依然需要设置NSAllowsArbitraryLoads属性为YES，以便可以访问http。和https访问不同的是，不再需要实现代码URLSessionDelegate，因为不存在证书的challenge的问题。代码如下：

```
import UIKit
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,URLS
essionDelegate {
    var window: UIWindow?
    func application(_ application: UIApplication, didFini
shLaunchingWithOptions launchOptions: [UIApplicationLaunch
OptionsKey: Any]?) -> Bool {
        foo()
        return true
    }
```

```
    func foo(){
        let task = URLSession.shared.dataTask(with: URL(st
ring: "http://localhost:8000")!) { (data, response, error)
 in
            if error != nil {
                print(error)
            } else {
                if let usableData = data {
                    do {
                        let json = try JSONSerialization.j
sonObject(with: usableData, options:[])
                        print("json: \(json)")
                    }
                    catch {
                        print("Error: \(error)")
                    }
                }
            }
        }
        task.resume()
    }
}
```