Broadview
www.broadview.com.cn

王志刚作品系列

·适用于iOS5·

iPhone UIKit 详解

王志刚 王中元 朱蕾 编著

在iPhone应用程序开发中用得最多、也最重要的应该是UIKit框架了，
本书将会就如下问题，给出答案：

◎UIKit到底是何种框架，包含了哪些功能?
◎我想使用UISlider，到底该如何使用?
◎UIBarButton该如何初始化?
◎我想让iPhone程序画面进行全屏显示，该如何实现?
……

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

UIKit框架是iPhone应用程序开发中最基本的框架，也是用得最多、最重要的框架。本书就是一本UIKit开发大全，包括UIKit框架中各种类、控件使用技巧的相关介绍。本书每个章节都配有详细的应用实例，方便读者对UIKit中各种类、控件的理解，也可直接应用于自己的iPhone应用程序中。本书可作为开发iPhone应用程序的工具书。

图书在版编目（CIP）数据

iPhone UIKit详解 /王志刚，王中元，朱蕾编著. —北京：电子工业出版社，2012.7
ISBN 978-7-121-17100-0

Ⅰ.①i… Ⅱ.①王… ②王… ③朱… Ⅲ.①移动电话机—应用程序—程序设计 Ⅳ.①TN929.53

中国版本图书馆CIP数据核字（2012）第101854号

策划编辑：孙学瑛
责任编辑：付　睿
特约编辑：赵树刚
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1／16　　印张：30.25　　字数：630千字
印　　次：2012年7月第1次印刷
印　　数：3000册　　定价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

在iPhone应用程序开发中用得最多、也最重要的应该是UIKit框架（Framework）了，UIKit框架中不仅包含构建iPhone应用程序画面的各种控件，以及与iPhone特色的画面布局控制、跳转控制相关的各种类，甚至还包含控制摄像头、加速度传感器、接近传感器等iPhone特色传感器的API。

本书缘起

对于UIKit框架中种类繁多的各种类，要让iPhone程序员们一一记住它们是不现实的。即便是经验丰富的iPhone程序员，在开发过程中也会为了某个画面功能的实现，需要查开发文档或者“求助”于互联网。开发文档不仅解说得比较粗略，而且是英文的，去互联网中查找资料就更加费时费力了。笔者以前时常奢望如果手头能有一本类似于UIKit开发大全那样的中文参考书该有多好。

笔者编写本书的目标正在于此，不仅可以让每一位iPhone程序员在开发iPhone应用程序时可以实时参考，而且对于初学者，相信本书也能加深其对UIKit框架的理解。

本书内容

本书内容共分13章，主要内容分为5个部分。

第1部分（第0章、第1章）为引导部分，介绍UIKit框架的基本概念，以及如何在不使用Xcode的“所见即所得”界面编辑功能下进行iPhone编程。

第2部分（第2～4章）为iPhone应用程序画面基础部分。这一部分介绍构成iPhone应用程序画面的基本类。包括构成画面的UIView及各种常用UI控件（皆为UIView子类），以及与画面控制相关的UIViewController及其子类。

第3部分（第5～7章）介绍iPhone特色画面组成控件，此部分介绍图形、动画、文字显示、屏幕效果等所有iPhone特色效果的实现方式。

第4部分（第9章、第10章）介绍UIKit框架中的事件与动作控制类，此部分介绍各种事件处理方法及与用户交互相关的API。

第5部分（第11章、第12章）为UIKit框架中的其他功能，主要介绍设置/获取应用程序及设备信息的功能，以及复制/粘贴功能，还介绍与摄像头及视频相关的各种API及使用方法。

本书读者对象

本书适合具有一定Objective-C 2.0语言基础的读者使用，作为iPhone应用程序开发的参考书，或者作为学习iPhone软件开发的进阶参考资料，尤其可加深关于UIKit框架部分的理解。

武汉大学计算机学院王中元副教授以及朱蕾负责编写了本书的部分内容。另外，江友华、罗伟、黄建峰、朱至濂参加了本书部分章节的审校及编写工作。在此特别感谢我父母在本书编写过程中给予的大力支持。

0.1 关于本书

0.1.1 本书的目标

UIKit框架（Framework）不仅是iPhone SDK最基础的部分，也是所有iPhone应用程序的基础，本书专门介绍使用UIKit框架进行iPhone软件开发的基础知识。因此，书中关于Objective-C 2.0语言基础以及Xcode使用方法的内容都省略了。

全书所有的章节都围绕UIKit框架展开，尽量讲解得具体、细致。如果在开发iPhone应用程序过程中碰到了如下疑问，相信你会从本书中找到满意的答案。

- UIKit到底是何种框架？包含了哪些功能？
- 我想使用UISlider，到底该如何使用？
- UIBarButton应如何初始化？
- 我想让iPhone程序画面进行全屏显示，该如何实现？
- viewDidLoad方法的调用时机是什么时候？
- 我大致知道这种类的用途，但是我想详细了解一下它的功能。
- API文档中有这种属性的介绍，但是到哪儿能找到具体的使用实例呢？

0.1.2 本书的特征

本书没有介绍与Objective-C 2.0 语言相关的知识，在阅读本书前需要有一定的

Objective-C 2.0 语言基础。下面是笔者撰写的一本关于Objective-C 2.0 语言基础的书籍名称及网站地址：《软件创富密码：iPhone应用程序开发之深入浅出Objective-C 2.0》，http://www.iacademe.net/item?ISBN=978-7-121-13469-2。有兴趣的朋友可以先从这本书开始学习一些关于Objective-C 2.0 语言的编程基础。当然还可以参考其他关于Objective-C 2.0语言的书籍或者网上的相关专题介绍。

对于那些有一定iPhone应用程序开发经验的朋友来说，可以放心地阅读本书。它不仅可以帮助你巩固iPhone应用程序开发的基础、提供开发技能，还可以作为iPhone应用程序开发时的工具书来使用。UIKit框架中的类种类繁多，不可能一一记住，以笔者的经验，开发时如果有一本如工具书一样的参考书将会事半功倍。另外，本书以不使用Xcode的“所见即所得”界面编辑功能为前提（在Xcode 4 以前版本中即不使用Interface Builder）进行解说。关于不使用界面编辑功能的开发方式在第1章中将有详细介绍。

0.1.3 本书的章节组成

本书中包含的主要章节的基本介绍已经归纳于表0-1中。

表0-1 章节主要内容

分　类	章　节	简　介
第1部分 引导部分	第0章 本书导读 第1章 UIKit概要	介绍UIKit框架的基本概念以及如何在不使用Xcode的“所见即所得”界面编辑功能下进行iPhone编程
第2部分 画面基础	第2章 UIView概要 第3章 UIViewController与画面控制 第4章 常用UI控件	注意这里所说的基础，并非“入门”的意思。这一部分介绍构成iPhone应用程序画面的基本类。包括构成画面的UIView及各种常用UI控件（皆为UIView子类），以及与画面控制相关的UIViewController及其子类

续 表

分 类	章 节	简 介
第3部分 iPhone特色画面组成控件	第5章 图形与动画 第6章 文本与Web显示 第7章 表格视图（UITableView） 第8章 全屏显示与画面旋转	iPhone之所以能带来如此大的冲击，笔者认为是iPhone卓越的用户体验，而给用户这种体验的正是富有iPhone特色的画面组成控件。此部分介绍了图形、动画、文字显示、屏幕效果等所有iPhone特色效果的实现方式
第4部分 事件与动作控制	第9章 传感器API 第10章 用户交互相关API	iPhone等智能终端的特点就是通过触摸屏来进行操作，相应地也带来了各种特色的用户交互模式。此部分介绍各种事件处理方法及与用户交互相关的API
第5部分 其他功能	第11章 应用程序及设备相关API	设置/获取应用程序及设备信息的功能，以及复制/粘贴功能
	第12章 视频相关API	介绍与摄像头及视频相关的各种API及使用方法

0.2 关于实例代码

0.2.1 实例代码下载

本书中各章的实例代码都可以从如下网站进行下载：http://www.softchallenger.com。其中，以zip压缩文件形式供读者下载。所有的这些实例代码都遵循MIT License。读者可以在自己的程序中自由使用、复制这些代码，而且不论应用程序是私用或者是商用。但是其中的图片并不遵循MIT License，除了供读者运行实例程序外，禁止用于其他地方。

0.2.2 命名规则

为了节省篇幅，本书各章节中并没有完整附上所有实例代码，如经常省略接口定义（即@interface）部分的代码。因此当阅读时看到没有定义的变量时，默认已经在@interface或者其他地方定义过了，此时可参考下载的实例代码。

另外，本书在定义的实例变量名后附加下画线，如instanceVariableName_。附加下画线的目的是区分各方法中的本地变量。而且当书中没有看到调用实例变量的release方法进行内存释放时，默认在dealloc方法中已经进行了释放处理。

最后，本书中所有用于演示功能的类名都以UIKitPrj开头，如UIKitPrjAlpha.m。

0.2.3 初始化处理代码

本书实例代码中，多数是在UIViewController的viewDidLoad方法中完成对画面各组成元素的初始化。实际开发中，关于此种初始化处理需要进行仔细考虑，不一定适合放在viewDidLoad方法中。因为通常viewDidLoad方法会被多次调用（可参考本书3.8节中的相关介绍），而且在实现了viewDidLoad方法后，往往需要考虑实现viewDidUnload方法，这两个方法通常是一对集合。

但是，如果考虑上述因素后，结果会将初始化代码变得更复杂，最终影响笔者对主要内容的解说。因此实例代码中基本忽略了对这些因素的考虑，提供了一个最简单的初始化处理过程。

0.2.4 实例代码组织

本书实例代码中，基本上是一个画面用于验证一个功能，因此很多时候是不需要导航条及工具条的。但是如果为每一个这样的画面都创建一个工程的话，这个工程的数量就会大大增加，代码量也会大大增加。读者可以看到，在下载的实例代码中，我们使用了UINavigationController，一个应用程序工程中包含了多个实例。因此，往往会出现执行本书中列出的代码时不显示导航条与工具条，而执行下载的实例代码时会显示导航条与工具条的情况（如图0-1所示）。

有导航条/工具条的画面　　无导航条/工具条的画面

图0-1 导航条与工具条的显示差异

读者不要在意这种差异，如果想显示导航条与工具条时，可以在UIViewController的viewWillAppear:animated:方法中追加如下的代码即可。

```
[self.navigationController setNavigationBarHidden:NO animated:YES];
[self.navigationController setToolBarHidden:NO animated:YES];
```

相反，如果想隐藏导航条与工具条时，可以执行如下代码。

```
[self.navigationController setNavigationBarHidden:YES animated:YES];
[self.navigationController setToolBarHidden:YES animated:YES];
```

另外，一个应用程序工程中所有的实例都罗列于由UITableViewController创建的表格中，如图0-2所示。单元格中显示了各功能演示实例的类名称（类名称开头的“UIKitPrj”被省略了），触摸（在模拟器中“单击”）单元后会进入各功能演示画面中。关于如图0-2所示画面的实现原理（各工程中的类名称都一样为RootViewController）及相关代码，读者在学习了本书第7章后就会明白。在这之前读者可先忽略对这部分代码的研读。

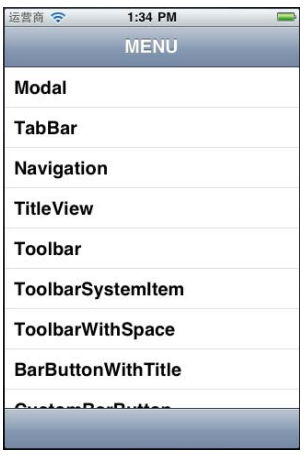
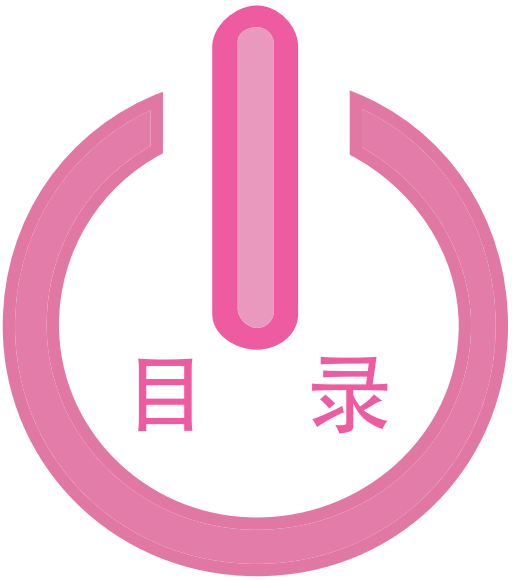


图0-2 启动画面



UIView Controller与画面控制

UIViewController是画面控制的中心类。本章将介绍如何使用UIViewController对由各种UIView组成的画面进行有效管理，以及画面间平滑跳转的方法。

UIViewController中有导航条、标签条、工具条等多种功能界面。正如UIView是组成画面的重要元素，UIViewController也是组成应用程序的重要元素。掌握这两种最基本的类，是成为合格iPhone应用程序员的必要条件。

3.1 UIViewController与画面的关系

3.1.1 UIViewController概要

UIViewController在UIKit中主要功能是用于控制画面的切换，其中的view属性（UIView类型）管理整个画面的外观。

在开发iPhone应用程序时UIViewController是否必不可少呢？显然不是这样，在第1章中的Hello World程序中显然就没有使用UIViewController，这说明不使用UIViewController也能编写出iPhone应用程序。第1章的Hello World程序只有一个画面，我们再考虑拥有两个画面的应用程序的情况。例如第一个画面与Hello World程序的画面类似，画面中显示“Hello World!”，当触摸画面下方的按钮后切换到另外一显示“您好!”的画面。其实我们仍然可以通过改变画面中的控件属性来实现画面外观的切换，例如此时可采取将不需要的控件都隐藏起来的方法。此时仍然不需要UIViewController。

但是，像上述这样实现画面外观的切换，同一个画面中实际上包含别的画面的

控件，代码看起来将非常凌乱。如果能将不同外观的画面进行整体的切换显然更合理，UIViewController正是用于实现这种画面切换方式的，其切换示意图如图3-1所示。

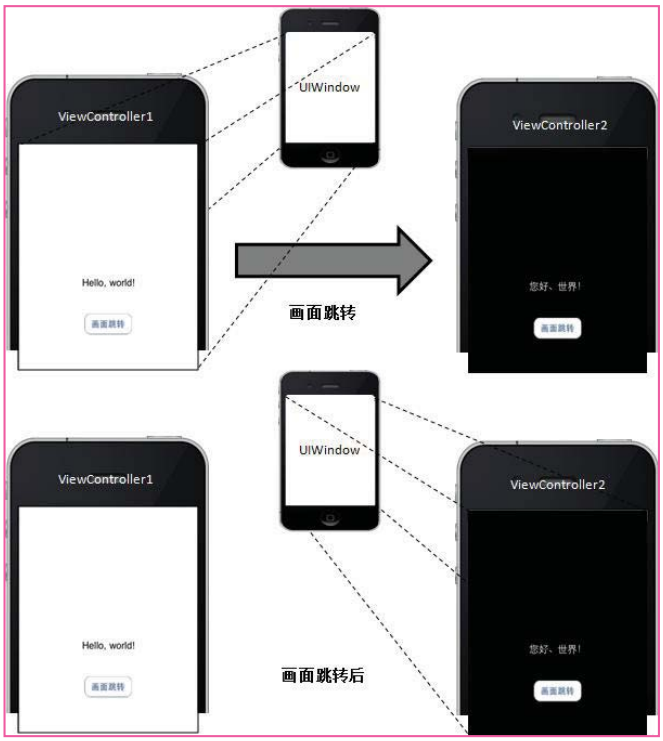


图3-1 UIViewController画面管理示意图

图3-1中分别由两个UIViewController管理两个画面，画面内容分别为“Hello,World!”及“您好，世界！”，当触摸“画面跳转”按钮后跳转到另一画面中。这是一个最简单的画面跳转实例，事先必须将两个画面内容（UIView及其子元素）追加到UIWindow对象中，但显示时只显示其中一个画面。具体实例代码下一小节中有详细介绍。

3.1.2 UIViewController的切换

本节我们实际使用UIViewController来创建画面。以第1章“Hello World!”实例为基础，我们再创建一个新的画面上面显示中文“您好、世界！”，并在这两个画面间实现自由切换。

不过大家需要注意的是，这里实现的画面跳转（或称切换）方法并非最佳

的跳转方法，目的不过是让读者了解UIViewController管理画面的方式，理解以UIViewController为单位切换画面的样子，如图3-1所示。

本实例中有两个画面，第一个画面中显示“Hello,World!”，背景为白色，下方布置有一个“画面跳转”按钮。第二个画面中显示“您好、世界！”，背景为黑色，下方也布置有一个“画面跳转”按钮。当触摸“画面跳转”按钮时，可在两个画面间进行显示切换。两个画面的代码如下。

```
[ViewController1.h]
#import <UIKit/UIKit.h>

@interface ViewController1 : UIViewController
@end

[ViewController1.m]
#import "ViewController1.h"

@implementation ViewController1

- (void)viewDidLoad {
  [super viewDidLoad];

  // 追加“Hello, world!”标签
  // 背景为白色、文字为黑色
   UILabel* label = [[[UILabel alloc] initWithFrame:self.view.
bounds] autorelease];
  label.text = @"Hello, world!";
  label.textAlignment = UITextAlignmentCenter;
  label.backgroundColor = [UIColor whiteColor];
  label.textColor = [UIColor blackColor];
  label.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIVie
wAutoresizingFlexibleHeight;
  [self.view addSubview:label];
```

```
        // 追加按钮
        // 单击按钮后跳转到其他画面
        UIButton* button = [UIButton buttonWithType:UIButtonTypeRoundedRect];
        [button setTitle:@"画面跳转" forState:UIControlStateNormal];
        [button sizeToFit];
        CGPoint newPoint = self.view.center;
        newPoint.y += 50;
        button.center = newPoint;
        button.autoresizingMask =
          UIViewAutoresizingFlexibleTopMargin | UIViewAutoresizingFlexib
leBottomMargin;
        [button addTarget:self
                   action:@selector(buttonDidPush)
         forControlEvents:UIControlEventTouchUpInside];
        [self.view addSubview:button];
    }

    - (void)buttonDidPush {
        // 自己移向背面
        // 结果是ViewController2显示在前
        [self.view.window sendSubviewToBack:self.view];
    }

    @end

    [ViewController2.h]
    #import <UIKit/UIKit.h>

    @interface ViewController2 : UIViewController
    @end

    [ViewController2.m]
```

```
    #import "ViewController2.h"

    @implementation ViewController2

    - (void)viewDidLoad {
      [super viewDidLoad];

      // 追加“您好、世界!”标签
      // 背景为黑色、文字为白色
       UILabel* label = [[[UILabel alloc] initWithFrame:self.view.
bounds] autorelease];
      label.text = @"您好、世界!";
      label.textAlignment = UITextAlignmentCenter;
      label.backgroundColor = [UIColor blackColor];
      label.textColor = [UIColor whiteColor];
      label.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIVie
wAutoresizingFlexibleHeight;
      [self.view addSubview:label];

      // 追加按钮
      // 单击按钮后画面跳转
      UIButton* button = [UIButton buttonWithType:UIButtonTypeRoundedRect];
      [button setTitle:@"画面跳转" forState:UIControlStateNormal];
      [button sizeToFit];
      CGPoint newPoint = self.view.center;
      newPoint.y += 50;
      button.center = newPoint;
      button.autoresizingMask =
        UIViewAutoresizingFlexibleTopMargin | UIViewAutoresizingFlexib
leBottomMargin;
      [button addTarget:self
                 action:@selector(buttonDidPush)
```

```
    forControlEvents:UIControlEventTouchUpInside];
  [self.view addSubview:button];
}

- (void)buttonDidPush {
  // 自己移向背面
  // 结果是ViewController1显示在前
  [self.view.window sendSubviewToBack:self.view];
}

@end
```

ViewController1与ViewController2都是继承**UIViewController**的子类。为了追加画面控件（UIView及其子类）需要重写（Overwrite）**viewDidLoad方法**，其中分别创建标签与按钮，并通过addSubview:方法追加到画面（self.view）中。viewDidLoad方法在UIViewController拥有的UIView（画面基础UIView）被导入后调用。此UIView可通过UIViewController的**view属性**参照，控件追加到画面中的代码如下所示。

```
[self.view addSubview:控件实例];
```

另一个重要的知识是，触摸按钮后实现画面切换。首先调用addTarget:action:forControlEvents:方法设置按钮被触摸时的响应方法**buttonDidPush**，关于按钮等控件的响应事件的设置第4.2节（UIButton）将进行介绍。buttonDidPush方法中只有如下一行代码，实现画面的切换。

```
[self.view.window sendSubviewToBack:self.view];
```

作为UIWindow的子元素，UIView可以通过其**Window属性**参照UIWindow（通常应用程序只有唯一的UIWindow）。这行代码的作用是“将画面自己隐藏到背面去”，因为原来UIWindow中包含有两个画面，当前画面隐藏到背面后，另一个画面就会显示在前面。

至于将两个画面（UIView）追加到UIWindow中的处理在HelloWorldAppDelegate类中实现，以下是HelloWorldAppDelegate的详细代码，黑体字部分表示向UIWindow中追加两个画面。

```
[HelloWorldAppDelegate.h]
#import <UIKit/UIKit.h>
```

```
@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
  UIWindow*window _ ;
  UIViewController*viewController1 _ ;
  UIViewController*viewController2 _ ;
}

@property (nonatomic, retain) UIWindow*window;

@end


[HelloWorldAppDelegate.m]
#import "HelloWorldAppDelegate.h"
#import "ViewController1.h"
#import "ViewController2.h"

@implementation HelloWorldAppDelegate

@synthesize window = window _ ;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

  // 初始化Window
  CGRect bounds = [[UIScreen mainScreen] bounds];
  window _ = [[UIWindow alloc] initWithFrame:bounds];

  // 创建ViewController1与ViewController2
  // 并将其画面《view》追加到Window中
  viewController1 _ = [[ViewController1 alloc] init];
  viewController2 _ = [[ViewController2 alloc] init];
  [window _ addSubview:viewController1 _ .view];
```

```
    [window _ addSubview:viewController2 _ .view];

    // ViewController1放在前面显示
    [window _ bringSubviewToFront:viewController1 _ .view];

    [window _ makeKeyAndVisible];
}


- (void)dealloc {
    [viewController1 _ release];
    [viewController2 _ release];
    [window _ release];
    [super dealloc];
}

@end
```

在应用程序初始化applicationDidFinishLaunching:方法中，首先创建两个画面的实例，然后分别追加（使用addSubview:方法）到UIWindow中，最后调用UIWindow的bringSubviewToFront:方法将第一个画面显示在前面。

实例运行后的效果如图3-2所示。

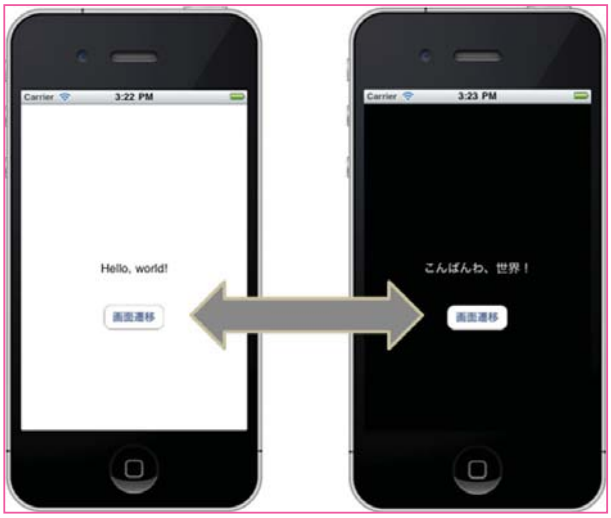


图3-2 画面切换实例

3.2 画面跳转

3.2.1 使用UITabBarController实现并列画面跳转

前一节介绍了由UIViewController实现的画面切换。实际上并非真的实现了两个画面间的跳转，而是同时启动两个画面，控制其中哪一个画面显示在前台，哪一个画面显示在后台而已。这种画面跳转方式有一个很大的缺点，即当画面数量增加时，画面跳转的实现代码将越来越复杂，而且各个画面间不可避免地有相互依赖关系。

实际上，在UIKit中提供了专用的管理画面跳转的类，这就是**UITabBarController**类以及**UINavigationController**类。本节首先介绍使用UITabBarController类如何实现画面间的跳转功能。

下面我们以上一节实例代码为基础，将其改造成使用UITabBarController类来实现画面间的跳转。首先将**HelloWorldAppDelegate**改名为**MultiViewAppDelegate**，并进行如下修改（代码中粗体字部分）。

[MultiViewAppDelegate.h]

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

@interface MultiViewAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    UIViewController *rootController;
}

@property (nonatomic, retain)  UIWindow *window;

@end
```

[MultiViewAppDelegate.m]

```
#import "MultiViewAppDelegate.h"
#import "ViewController1.h"
```

```
    #import "ViewController2.h"

    @implementation MultiViewAppDelegate

    @synthesize window;


    #pragma mark -
    #pragma mark Application lifecycle

    - (BOOL)application:(UIApplication *)application didFinishLaunchin
gWithOptions:(NSDictionary *)launchOptions {
      // 初始化window实例变量
      CGRect frame = [[UIScreen mainScreen] bounds];
      self.window = [[UIWindow alloc] initWithFrame:frame];

      // 创建母体Controller实例
      rootController = [[UITabBarController alloc] init];

      // 创建画面1与画面2的Controller实例
      ViewController1 *tab1 = [[[ViewController1 alloc] init]
autorelease];
      ViewController2 *tab2 = [[[ViewController2 alloc] init]
autorelease];

      // 将画面1、画面2的Controller实例以数组的形式追加到母体Controller中
      NSArray *tabs = [NSArray arrayWithObjects:tab1,tab2,nil];
      [(UITabBarController)rootController setViewControllers:tabs
animated:NO];

      // 将母体Controller的view追加到Window中
      [self.window addSubView: rootController.view];
        [self.window makeKeyAndVisible];
```

```
    return YES;
}

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

@end
```

要修改的地方并不多。首先删除前例中单独创建两个画面的实例变量ViewController1以及ViewController2，取而代之的是UITabBarController类型的实例变量**rootController**。上例中将两个画面的view直接追加到Window中，本例中只需要将UITabBarController类型实例变量的view追加到Window中。而将两个画面的UIViewController对象以数组的形式追加到UITabBarController类型的实例变量中，此时调用了**setViewControllers: animated:方法**，此方法的第一个参数即为UIViewController对象数组。

接着，我们开始修改两个画面的实例代码。其实施方法如下。

- 按钮去掉，删除与按钮相关的代码。
- 重写（或称覆盖）init方法，其中追加与标签相关的代码。

两个画面中的init方法的代码如下。

```
[ViewController1.m]
- (id)init {
  if ((self = [super init])) {
    // 设置tabBar的相关属性
    self.title = @"Hello";
    UIImage*icon = [UIImage imageNamed:@"ball1.png"];
    self.tabBarItem =
        [[[UITabBarItem alloc] initWithTitle:@"Hello" image:icon
tag:0] autorelease];
  }
  return self;
```

```
    }
    [ViewController2.m]
    - (id)init {
      if ((self = [super init])) {
        // 设置tabBar的相关属性
        self.title = @"您好";
        UIImage*icon = [UIImage imageNamed:@"ball2.png"];
        self.tabBarItem =
          [[[UITabBarItem alloc] initWithTitle:@"您好" image:icon tag:0]
autorelease];
      }
      return self;
    }
```

重写的init方法中也没有追加多少内容。设置title属性后，导入图标用的图片，并将其设置到UIViewController的tabBarItem属性中。调用**initWithTitle:image:tag:方法**进行UITabBarItem类的初始化。第一个参数为标签条中显示的标题；第二个参数为指定显示的图标图片；第三个参数为标签的序号，此序号通常用于程序内检索。

执行这个经过改造的工程后，将显示如图3-3所示的结果。

图3-3 标签实现的画面跳转

3.2.2 使用UINavigationController实现多层画面跳转

iPhone4手机的自带应用程序中，既有使用UITabBarController来进行画面切换控制的，也有使用UINavigationController来实现多画面间的跳转的。例如iPod音乐播放界面就采用了UITabBarController来进行画面切换控制，而iPhone手机设置程序则采用了UINavigationController来实现多层次画面间的跳转，图3-4、图3-5是这两个程序部分画面的跳转示意图。

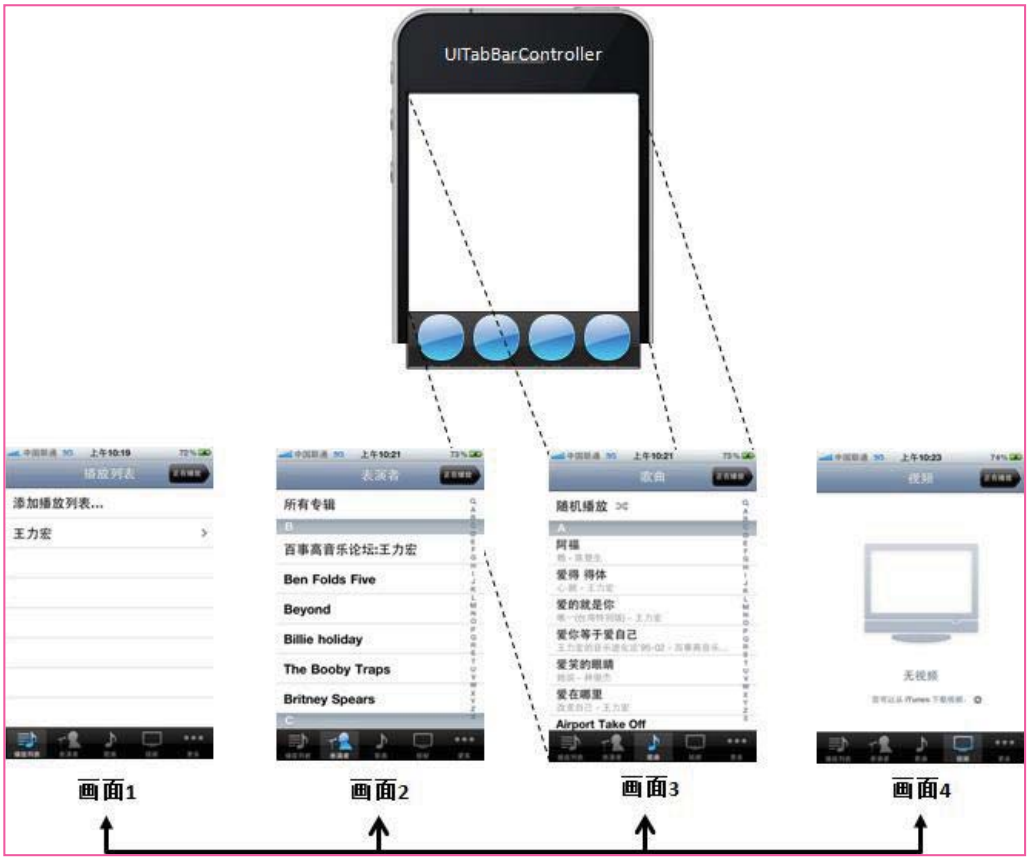


图3-4 iPod音乐播放程序画面跳转示意图

与UITabBarController实现画面并行切换形式不同，UINavigationController是实现画面多层次跳转，也是其最大的特征。如图3-5 所示，画面1-1可以跳转至其下一层的画面1-1-1以及画面1-1-2中。另外UINavigationController可以自动地记忆跳转所经过的路径，按照这些记录的路径信息，可以依次返回到上层画面中（即支持返回按钮）。

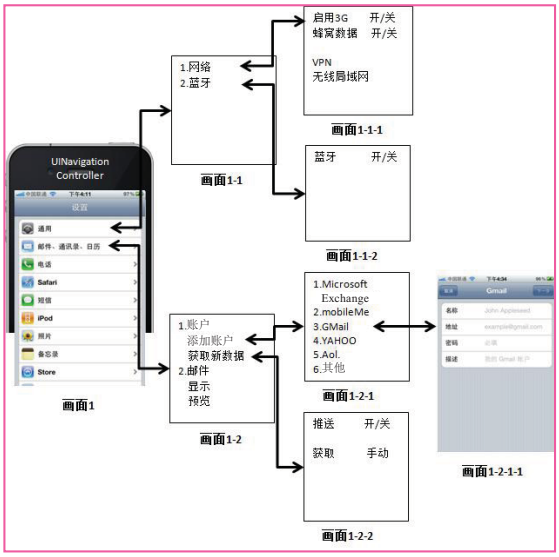


图3-5 iPhone设置程序部分画面跳转示意图

下面我们将上一节采用UITabBarController实现的画面切换程序，再一次改造成采用UINavigationController来实现画面的跳转程序。其中两个下一层的画面保持不变，在这之前我们还将追加一个主画面，主画面非常简单，只有一个iPhone表格视图组成，两个下层画面的名称依次显示在表格中，当单击任何一个名称时将会跳转到对应的下层画面中，整个应用程序的跳转示意图如图3-6所示。

图3-6 改造目标程序的跳转示意图

要实现上述程序，首先要进行如下两处修改。

- 在HelloWorldAppDelegate.m中将基准ViewController由UITableViewController替换为UINavigationController。
- 新追加主画面的TopMenuController类。

需要注意一点，这里我们对画面1以及画面2的实现代码其实是没有进行任何修改的。只是留下了些原实例中追加的关于UITabBarController的注释，请务必忽略。

下面我们看看HelloWorldAppDelegate.m的修改代码，见代码中的黑体字部分。

```
〖HelloWorldAppDelegate.m〗
#import "HelloWorldAppDelegate.h"
#import "TopMenuController.h"

@implementation HelloWorldAppDelegate

@synthesize window = window_;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // 初始化Window实例变量
    CGRect bounds = [[UIScreen mainScreen] bounds];
    window_ = [[UIWindow alloc] initWithFrame:bounds];

    // 创建基准的Controller对象
     TopMenuController*topMenu = [[[TopMenuController alloc] init]
autorelease];
    rootController_ = [[UINavigationController alloc] initWithRootVi
ewController:topMenu];

    // 将主画面的view追加到Window中
    [window_ addSubview:rootController_.view];

    [window_ makeKeyAndVisible];
```

```
}

- (void)dealloc {
  [rootController _ release];
  [window _ release];
  [super dealloc];
}

@end
```

大家可以看到，其实只修改了其中三行代码。首先需要导入TopMenuController类，并创建其实例。接着初始化UINavigationController，需要向**initWithRootViewController:方法**中传入根画面的Controller，这里将创建好的TopMenuController实例传入。然后将UINavigationController的view属性追加到UIWindow中（使用addSubView:方法）。

下面是新追加的主菜单画面TopMenuController的代码。这里将TopMenuController以UITableViewController子类形式来创建。

〖TopMenuController.h〗

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

@interface TopMenuController : UITableViewController
{
 @private
  NSMutableArray* items _ ;
}

@end
```

[TopMenuController.m]

```
#import "TopMenuController.h"

@implementation TopMenuController
```

```
- (void)dealloc {
  [items _ release];
  [super dealloc];
}

- (id)init {
  if ( (self = [super initWithStyle:UITableViewStylePlain]) ) {
    self.title = @"主菜单";
    //-------------------------------<1>
    // 创建显示用数组
    items _ = [[NSMutableArray alloc] initWithObjects:
                  @"ViewController1",
                  @"ViewController2",
                  nil ];
  }
  return self;
}

#pragma mark ----- UITableViewDataSource Methods -----

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

  return [items _ count];
}

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

  // 检查单元是否已经创建
  UITableViewCell* cell = [tableView dequeueReusableCellWithIdenti
```

```
fier:@"simple-cell"];
        if ( !cell ) {
            // 没有创建的单元新创建
            cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
reuseIdentifier:@"simple-cell"] autorelease];
        }
        // 设置单元中显示的文本字符串
        cell.textLabel.text = [items _ objectAtIndex:indexPath.row];
        //------<2>
        return cell;
    }

    #pragma mark ----- UITableViewDelegate Methods ---------<3>

    - (void)tableView:(UITableView*)tableView
    didSelectRowAtIndexPath:(NSIndexPath*)indexPath {

        Class class = NSClassFromString( [items _ objectAtIndex:indexPath.row] );
        id viewController = [[[class alloc] init] autorelease];
        if ( viewController ) {
            [self.navigationController pushViewController:viewController
animated:YES];
        }
    }

    @end
```

与表相关的解说将放在第7章中，代码中有不理解的地方，请参考第7章的相关介绍。首先看看init方法中的内容。在titile属性中设置画面的标题（<1>）。使用UINavigationController时，此处设置的标题将在画面的最上方中心位置显示（见图3-7）。接着还创建了**NSArray数组**，NSArray数组中的元素将显示在表中。具体处理在**tableView:cellForRowAtIndexPath:方法**中，将NSArray中的元素设置到表的单元中。

图3-7　UINavigationController的标题

其次，我们再确认一下**tableView:didSelectRowAtIndexPath:方法**的处理内容。此方法将在表单元被触摸（单击）时调用，参数 indexPath 中保存了具体被触摸（单击）的行信息。

本例中，表单元中直接放置了跳转对象画面的类名，tableView:didSelectRowAtIndexPath:方法中首先创建被触摸行类的实例。然后跳转到对应画面中。调用**UINavigationController的pushViewController: animated:方法**实现画面跳转；向此方法的第一个参数中传入画面（UIViewController）的实例后，然后就会自动跳转到对应层次的画面中。另外如果将animated参数设置为YES，则下一画面将以动画的形式显示出来。此时UINavigationController实例可以通过UIViewController的navigationController属性获取。只要是UINavigationController管理下的UIViewController，随时都可以通过其**navigationController属性**获取UINavigationController实例本身。

这样就算完成了整个层次的画面跳转应用程序。跳转到下一层画面后，将自动显示如图3-8所示的返回按钮。

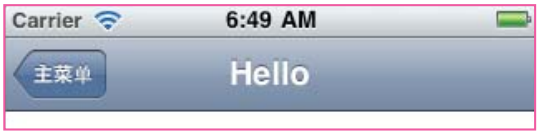


图3-8　UINavigationController的返回按钮

3.2.3　跳转到任意画面

上一小节中通过 **pushViewController: animated:方法**能实现画面的跳转，而且能在导航条上自动追加返回上一画面的返回按钮。这种“返回到前一画面”的功能正确的表述应该为“返回到上一级”画面，调用 **popViewControllerAnimation:方法**也能实现同样的功能。其他的如UINavigationController类还提供了直接返回到主画面的**popToRootViewControllerAnimation:方法**，以及返回到任意一级画面的**popToViewController: animated:方法**，以下是这三种方法的调用代码。

```
    // 返回到上一级画面
    [self.navigationController popViewControllerAnimation:Yes];
    // 返回根画面
    [self.navigationController popToRootViewControllerAnimation:Yes];

    // 返回任意指定画面
    [self.navigationController popToViewController:viewController
animated:Yes];
```

另外，从iPhone OS 3.0以后，可以通过调用setViewControllers:animated:方法将画面的跳转历史路径（堆栈）完全替换。替换历史路径的示意图如图3-9所示。

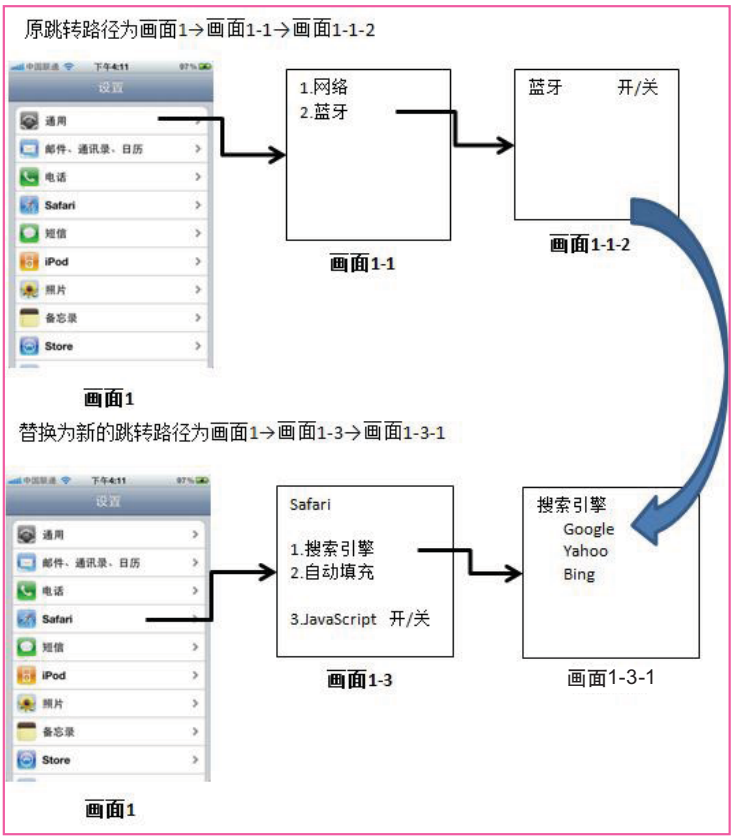


图3-9 替换历史路径

实现图3-9所示的替换历史路径的代码如下。

```
    // 将跳转历史路径替换为画面1> 画面1-3> 画面1-3-1
    id  scene1 = [[[Scene alloc] init] autorelease];
```

```
    // 创建画面1的实例
    id  scene13 = [[[Scene3 alloc] init] autorelease];
    // 创建画面1-3的实例
    id  scene131 = [[[Scene31 alloc] init] autorelease];
    // 创建画面1-3-1的实例
    NSArray *history = [NSArray arrayWithObjects:scene1,scene13,scene13
1,nil];
    [self.navigationController setViewControllers:history
animated:Yes];
```

popToViewController: animated:方法的第一个参数中必须传入UIViewController的实例，不是新创建的实例，而是实际跳转过程中原画面的实例。此时，可以通过UINavigationController的**viewControllers属性**来参照。viewControllers属性中保存的正是NSArray形式的跳转画面实例集合。

调用setViewControllers:animated方法进行跳转画面堆栈替换时，也可以viewControllers属性中保存的实例集合为基础，进行部分UIViewController实例的替换。

3.2.4 模态（modal）画面的显示方法

PC桌面软件中经常可以看到如“文件读取对话框”等模态对话框的画面类型。这些画面就显示在主画面的上方，当对话框中的操作结束，关闭对话框画面后将显示原来的画面，属于一种临时画面。iPhone应用程序中也能实现这种模态画面，例如iPhone通信录管理程序中，追加新的通信录时也使用了这种模态画面。

模态画面没有什么特别的地方，与其他画面一样也是由UIViewController的子类实现的画面，只是调用的方式不同而已。以下是模态画面显示的调用方式以及显示后关闭画面的实例代码。

```
    // ModalDialog为UIViewController的子类
    id dialog = [[[ModalDialog alloc] init] autorelease];
    [self presentModalViewController:dialog animated:YES];

    // 关闭模态UIViewController
    [self dismissModalViewControllerAnimationed:YES];
```

如上述代码所示，将UIViewController子类的实例作为**presentModalViewController:animated:方法**的第一个参数进行调用后，就能实现以模态方式显示画面。关闭时调用**dismissModalViewControllerAnimationed:方法**。模态画面调用后的示意图如图3-10所示。

图3-10 模态画面显示示意图

从iPhone OS 3.0开始，追加了设置模态画面显示／隐藏时动画效果的modalTranstionStyle属性，可设置三种不同的值，分别如下。

- UIModalTransitionStyleCoverVertical：画面从下向上徐徐弹出，关闭时向下隐藏（默认方式）。
- UIModalTransitionStyleFlipHorizontal：从前一个画面的后方，以水平旋转的方式显示后一画面。
- UIModalTransitionStyleCrossDissolve：前一画面逐渐消失的同时，后一画面逐渐显示。

以下是图3-10所示模态画面的代码，仅供参考。大家可以看到，其与普通的UIViewController子类没有任何区别。

```
// 以模态形式显示的画面
// 内容与普通画面一样
@interface ModalDialog : UIViewController
@end
@implementation ModalDialog
```

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

    // 追加1个标签
     UILabel* label = [[[UILabel alloc] initWithFrame:self.view.
bounds] autorelease];
    label.backgroundColor = [UIColor blackColor];
    label.textColor = [UIColor whiteColor];
    label.textAlignment = UITextAlignmentCenter;
    label.text = @"您好。我是模态画面。";
    [self.view addSubview:label];

    // 追加关闭按钮
    UIButton* goodbyeButton = [UIButton buttonWithType:UIButtonTypeR
oundedRect];
    [goodbyeButton setTitle:@"Good-bye" forState:UIControlStateNormal];
    [goodbyeButton sizeToFit];
    CGPoint newPoint = self.view.center;
    newPoint.y += 80;
    goodbyeButton.center = newPoint;
    [goodbyeButton addTarget:self
                      action:@selector(goodbyeDidPush)
            forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:goodbyeButton];
}

- (void)goodbyeDidPush {
    // 关闭模态对话框
    [self dismissModalViewControllerAnimated:YES];
}

@end
```

3.3 UITabBarController的使用技巧

3.3.1 UITabBar的参照

管理标签条的UITabBar类的实例，可以通过UITabBarController的**tabBar属性**取得。UITabBar的**items属性**中以NSArray形式保存管理的标签条项目，当前选中的标签（显示的画面）的索引保存在**selectedItem属性**中。

另外，UITabBarController管理下的各UIViewController的**tabBarItem属性**（注意UITabBarItem实例中UITabBarItem为UIBarItem的子类，与UIView间没有继承关系）也会自动追加到UITabBar的items属性中。

3.3.2 系统图标的使用

可以使用UIKit中提供的系统标签项目作为标签条的项目。针对特定的功能（或画面），系统标签项目提供与此功能相匹配的图标图片以及标题的组合。这样不仅能简单地设置标签项目，而且因为使用了这些用户已经习惯的标签项目，将方便用户更直观地操作，应用程序的界面也将更加人性化。

通过在UITabBarItem初始化时调用**initWithTabBarSystemItem:tag:方法**来设置系统标签项目。实例代码如下。

```
    // self为UITabBarController中设置的UIViewController的子类实例
    self.tarBarItem = [[[UITabBarItem alloc] initWithTabBarSystemItem:U
ITabBarSystemItemFeatured tag:0] autorelease];
```

如上述代码中所示，通过在**initWithTabBarSystemItem:tag:方法**的第一个参数中指定**UITabBarSystemItem**类型的常量来设置各种系统标签项目。不需要tag的话可将其设置成0。这样就将创建的UITabBarItem实例设置到UITabBarController中追加的各画面的tabBarItem属性中了。上述方法第一个参数中可指定的常量以及实际显示的图标列表如表3-1所示。

表3-1　系统标签项目列表

UITabBarSystemItem	图标显示效果	
UITabBarSystemItemMore	当标签条中的图片数超过5个时会显示此图标，触摸此图标会显示其他未显示图标	
UITabBarSystemItemFavorites	显示用户自己的个人收藏内容	
UITabBarSystemItemFeatured	显示应用程序的推荐内容	
UITabBarSystemItemTopRated	显示用户评价高的内容	
UITabBarSystemItemRecents	显示用户最近访问的内容	
UITabBarSystemItemContacts	显示通讯录画面	
UITabBarSystemItemHistory	显示用户操作的历史记录	
UITabBarSystemItemBookmarks	显示应用程序的书签画面	
UITabBarSystemItemSearch	显示搜索画面	
UITabBarSystemItemDownloads	显示下载中/下载完成的内容	
UITabBarSystemItemMostRecent	显示最新内容	
UITabBarSystemItemMostViewed	显示观看最多，最有人气的内容	

3.3.3 自定义图标的使用

上一小节介绍了如何向标签条中设置系统标签项目的方法。正如第3.2.1节中介绍的一样，程序员还可以设置自定义图标。这时在初始化UITabBarItem时需要使用initWithTitle:image:tag:方法。其中第一个参数中设置标签条中显示的标题，image中指定使用的图标图片（UIImage）。以下是具体的实例代码。

```
    // 导入图标图片
    UIImage* icon = [UIImage imageNamed:@"ball1.png"];

    // 指定标题与图标图片后设置UITabBarItem
    self.tabBarItem =
            [[[UITabBarItem alloc] initWithTitle:@"Hello" image:icon
tag:0] autorelease];
```

有一点需要特别注意，向image传递的图标图片有些特殊要求。此图标图片中需要同时存在透明色以及非透明色。在标签条中显示的图标，单色的非透明色将被完全覆盖，如图3-11所示。正确的表述是，将按照图标图片的透明度（A值）来创建显示的图标。

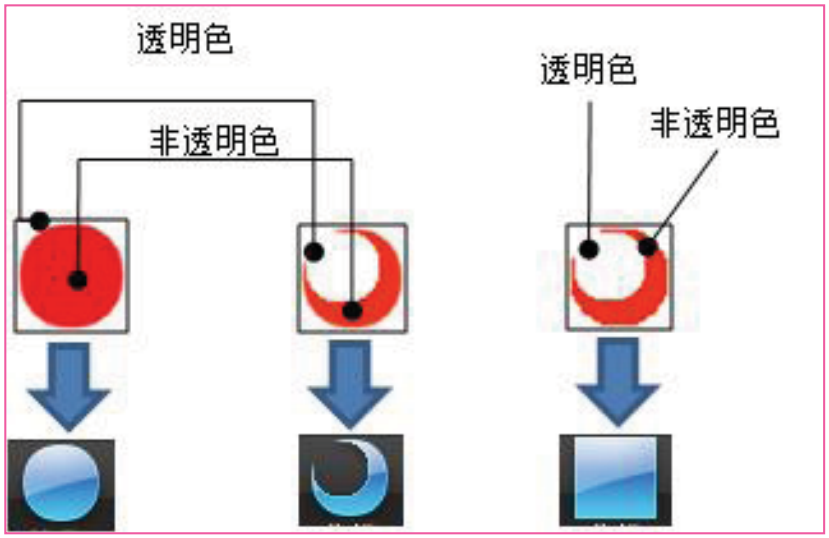


图3-11 标签条的图标图片处理示意图

表3-2是标签条图片在不同状态下的显示示例。

表3-2 标签图标的色彩配置

显示场合	图标显示效果
显示于标签条中，且画面处于活性的场合	蓝色光照效果 
显示于标签条中，且画面处于非活性的场合	灰色暗色效果 
画面超过5个，标签条中显示不下的场合	黑色轮廓 

3.3.4 向标签条中追加6个以上的画面

大家可能已经注意到了，在标签条上最多只能显示5个图标。如果需要设置6个以上的画面的情况下，会出现什么情况呢？实际上表3－1中的UITabBarSystemItemMore图标正是为出现此种情况而准备的。当向UITabBarController中设置10个画面时，将显示如图3-12所示的画面。

图3-12　向UITabBarController中设置6个以上画面时

此时，标签条的第5个图标将自动变成“更多”，当单击“更多”图标后，将在新画面中显示剩下的没有显示出来的图标。当单击画面右上方的“编辑”按钮后，还可以通过拖曳来改变图标的显示顺序，如图3-13所示。

图3-13 改变图标的显示顺序

UIKit提供了如上所述编辑功能，如果不想使用此编辑功能，可以通过将UITabBarController的**customizableViewControllers属性**设置为nil来屏蔽此编辑功能。

上述customizableViewControllers属性中原本保存着匹配图标顺序的各画面实例组成的数组。默认设置了所有的画面。例如，通过编辑customizableViewControllers属性，可实现复杂的带条件的画面显示定制顺序。

```
// 设置了scene1~scene7的7个画面
// 此时,事先将标签条中显示的三个画面固定下来
// 但是,第4个画面(scene4)将来有可能被其他任意画面取代
self. CustomizableViewControllers =
    [NSArray arrayWithObjects:scene4, scene5, scene6, scene6,nil];
```

3.3.5 标签条图标上的标记

此节介绍关于标签条项目的最后一项知识。iPhone手机中经常会显示如图3-14所示的效果，就是通常被称为“标示（badge）”的图形。可以通过设置

UITabBarItem的**badgeValue属性**来简单地实现标示效果。其代码如下。

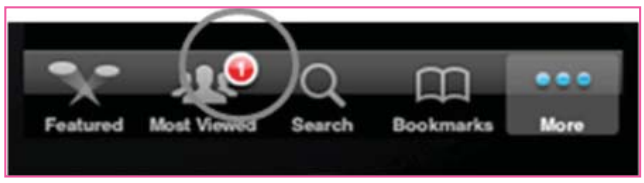


图3-14 标签条图标上的标示

```
    // 首先创建标签条图标的实例
    UITabBarItem*baritem =    [[UITabBarItem alloc] initWithTabBarSyste
mItem:UITabBarSystemItemMostViewed tag:0];
    // 设置badgeValue中显示的数字
    baritem.badgeValue = @"1";

    self.tabBarItem = baritem;
    [baritem release];
```

3.4 UINavigationController的使用技巧

3.4.1 导航条的4个区域

使用UINavigationController后，会显示如图3-15所示的导航条，关于导航条的知识已经在前面的章节中进行了介绍。

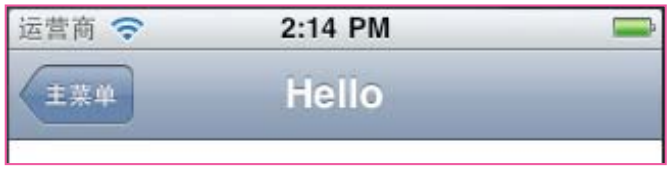


图3-15 导航条

但是实际上此导航条由如图3－16所示的4个部分组成，且由**navigationItem属性**（UINavigationItem实例）进行管理。在各个部分中可以分别配置自己设置的按钮以及文字。

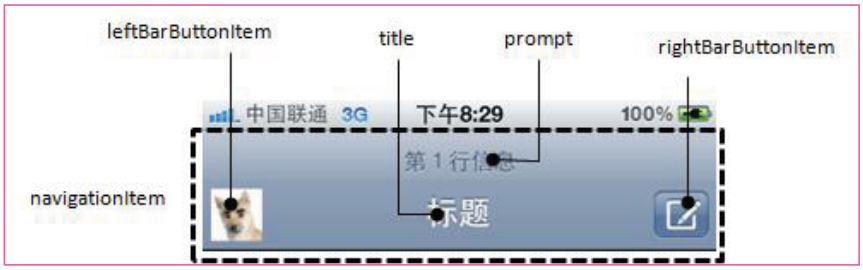


图3-16 导航条的四个部分

设置各个项目具体的实例代码如下。

```
        // 第1行信息的追加
        self.navigationItem.prompt = @"第1行信息";
        // 设置标题
        self.navigationItem.title = @"标题";

        // 在右侧追加按钮
        UIBarButtonItem*rightItem = [[[UIBarButtonItem alloc] initWithBar
ButtonSystemItem:UIBarButtonSystemItemCompose
                                        target:nil
                                        action:nil ] autorelease];
        self.navigationItem.rightBarButtonItem = rightItem;


        // 在左侧追加UIImageView
        UIImage*image = [UIImage imageNamed:@"face.jpg"];
        UIImageView*imageView = [[[UIImageView alloc] initWithImage:image]
autorelease];
        UIBarButtonItem*icon =
            [[[UIBarButtonItem alloc] initWithCustomView:imageView]
autorelease];
        self.navigationItem.leftBarButtonItem = icon;
```

另外，在navigationItem的**leftBarButtonItem属性**中设置了任意按钮后，默认的返回到前一画面的返回按钮将不被显示。如果想再次显示返回按钮时，可以将leftBarButtonItem属性设置为nil。

导航条左侧默认显示的原返回按钮可以通过UINavigationController的setHidesBackButton:animated:方法进行ON/OFF的切换。

UINavigationController的leftBarButtonItem属性与rightBarButtonItem属性中可以设置各种各样的按钮。关于按钮请参照第3.6节中的详细介绍。

3.4.2 导航条的定制

导航条中还有titleView属性，可以在此titleView属性中设置任意UIView的子类，且设置的UIView子类没有任何限制。巧妙地使用设置的子类，可对导航条进行各种各样的定制。下面的实例中我们试着在titleView属性中追加UISlider控件。完成的画面如图3-17所示。

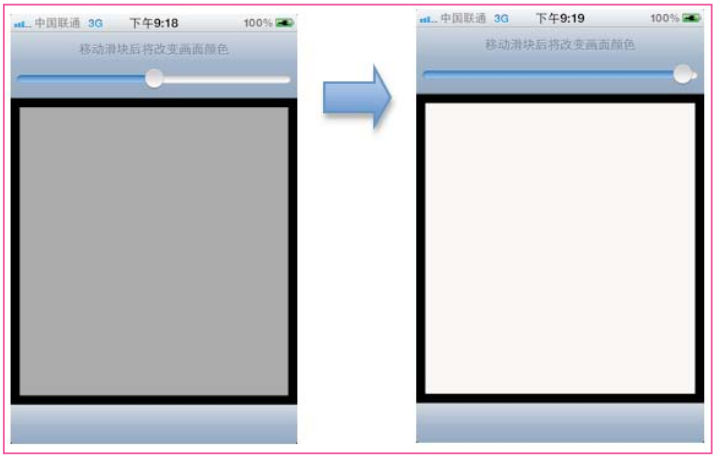


图3-17 向导航条中追加UISlider控件

移动显示在导航条中的滑块后，可以相应地改变画面的颜色。以下是具体的实现代码。

```
〖interface〗
@interface UIKitPrjTitleView : UIViewController
{
 @private
  UILabel* label_;
  UISlider* slider_;
}
@end
```

```
【implementation】
// 声明私有方法
@interface UIKitPrjTitleView ()
- (void)sliderDidChange;
@end

@implementation UIKitPrjTitleView

// finalize
- (void)dealloc {
  [slider_ release];
  [label_ release];
  [super dealloc];
}

- (void)viewDidLoad {
  [super viewDidLoad];
  // 顶部信息设置
  self.navigationItem.prompt = @"移动滑块后将改变画面颜色";
  // 创建UISlider实例,滑块变化时调用sliderDidChange:方法
  slider_ = [[UISlider alloc] init];
  slider_.frame = self.navigationController.navigationBar.bounds;
  slider_.minimumValue = 0.0;
  slider_.maximumValue = 1.0;
  slider_.value = slider_.maximumValue / 2.0;
  [slider_ addTarget:self
              action:@selector(sliderDidChange)
    forControlEvents:UIControlEventValueChanged];
  self.navigationItem.titleView = slider_;
  // 创建标签,并根据滑块的值改变标签的颜色
  label_ = [[UILabel alloc] init];
```

```
        label _ .frame = CGRectInset( self.view.bounds, 10, 10 );
        label _ .autoresizingMask = UIViewAutoresizingFlexibleWidth | UIV
iewAutoresizingFlexibleHeight;
        label _ .backgroundColor = [UIColor blackColor];
        [self.view addSubview:label _ ];
        // 调用sliderDidChange方法设置滑块初始值
        [self sliderDidChange];
    }
    // 确保显示导航条以及工具条
    - (void)viewWillAppear:(BOOL)animated {
        [super viewWillAppear:animated];
         [self.navigationController setNavigationBarHidden:NO
animated:NO];
        [self.navigationController setToolbarHidden:NO animated:NO];
        [self.navigationItem setHidesBackButton:YES animated:NO];
    }

    #pragma mark ----- Private Methods -----
    // 画面显示时隐藏返回按钮,触摸画面后显示返回按钮
    - (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
        [self.navigationItem setHidesBackButton:NO animated:YES];
    }
    // 实现滑块移动时调用的方法,改变标签的颜色
    - (void)sliderDidChange {
        UIColor* color = [[[UIColor alloc] initWithRed:slider _ .value
    green:slider _ .valueblue:slider _ .value alpha:1.0 ] autorelease];
        label _ .backgroundColor = color;
    }

    @end
```

定制导航条并不是件特别困难的事情。本例中，创建了UISlider实例后，将其设置到UINavigationItem的titleView属性中即可。分别将UISlider的**minimumValue属性**以及**maximumValue属性**设置为0.0与1.0。因为此值后面将作为颜色值进行设置，而颜色RGB值的范围为0.0到1.0之间。另外将滑块的初始值设置为0.5。最后，当滑块移动时（具体事件为**UIControlEventValueChanged**）调用**sliderDidChange方法**，具体在**addTarget:action:forControlEvents:方法**中进行设置。这样将UISlider设置到titleView属性中后，即完成了导航条的定制。

在sliderDidChange方法中，调用UIColor的**initWithRed:green:blue:alpha:方法**创建标签的颜色，除了alpha参数设置为1.0外，其他三个参数都设置为滑块当前的值。将此创建的UIColor实例设置到UILabel实例的**backgroundColor属性**后，即完成了对标签颜色的设置。

3.4.3 导航条的颜色

可以通过改变UINavigationController类的tintColor属性，改变导航条的背景颜色。以下是实例代码。

```
    // 将导航条变成红色
    self.navigationController.navigationBar.tintColor = [UIColor
redColor];
```

此代码的执行结果如图3-18所示（图中导航条的背景实际为红色）。

图3-18 改变导航条的背景色

3.5 工具条

3.5.1 工具条的显示

从iPhone OS 3.0开始，通过调用UIViewController的**setToolbarItems: animated:方法**可以简单地在画面中追加工具条。下面是简单的实例代码。

```
        // 工具条左侧显示的按钮
        UIBarButtonItem* button1 =
          [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemRefresh
                                target:self
                                action:@selector(buttonDidPush)] autorelease];
        // 自动伸缩按钮以及按钮间的空白
        UIBarButtonItem*spacer =
           [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                                target:nil
                                action:nil ] autorelease];
        // 工具条右侧显示的按钮
        UIBarButtonItem* button2 =
           [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemUndo
                                target:self
                                action:@selector(buttonDidPush) ] autorelease];
        // 全部保存到NSArray中
         NSArray*buttons = [NSArray arrayWithObjects:button1, spacer, button2, nil];

        // 将准备好的NSArray作为工具条的项目设置进去
        [self setToolbarItems:buttons animated:YES];
```

首先，如果要显示工具条，必须将UINavigationController的**toolbarHidden属性**设置为**NO**（也可调用**setToolBarHidden:animated:方法**进行设置）。但是这样只会显示一个空的工具条，必须通过调用UIViewController的**setToolbarItems:animated:方法**在工具条中设置各种按钮项目（UIBarButtonItem）。实例代码执行后，会显示如图3-19所示的画面，在画面的下方显示设置的工具条。

图3-19 工具条实例

3.5.2 工具条的自动隐藏

上一小节介绍了工具条的显示方法。但是，一旦工具条显示，只要没有明确隐藏它，跳转到下一画面后，工具条仍然会保持显示状态，如图3-20所示。

图3-20 画面跳转后工具条仍然显示

有的时候并不希望工具条一直显示着。此时就要用到UIViewController的**hidesBottomBarWhenPushed属性**。在UINavigationController中调用pushViewController方法跳转到下一个画面前，将此画面的hidesBottomBarWhenPushed属性设置为YES后，以后的画面将隐藏工具条。设置**toolbarHidden属性**或者调用**setToolbarHidden:animated:方法**也能实现隐藏工具条，但是使用hidesBottomBarWhenPushed属性后，当再次返回到原画面时工具条将自动显示，而上述两种方式将一直隐藏工具条。实现效果如图3-21所示。

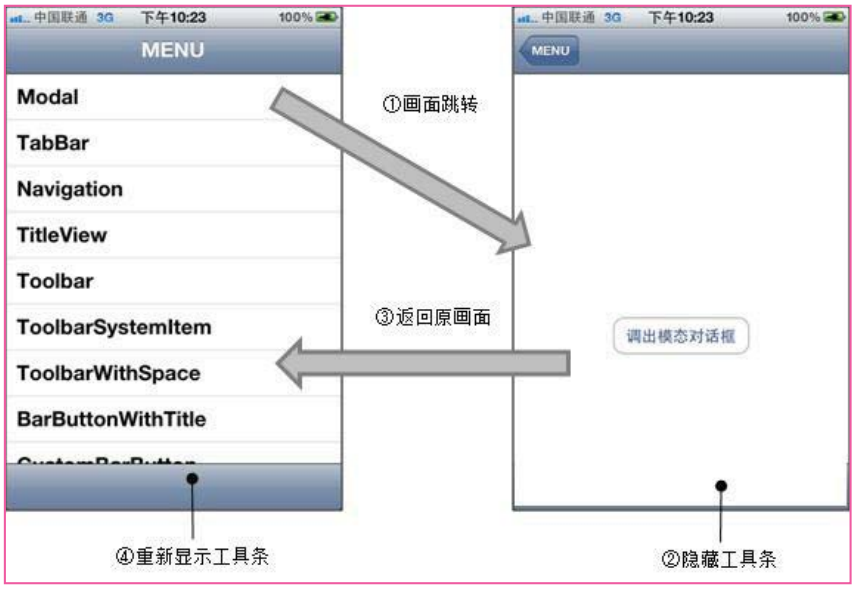


图3-21　使用hidesBottomBarWhenPushed属性后的效果

注意：hidesBottomBarWhenPushed属性仅仅是在画面跳转的时候决定是否隐藏工具条。画面跳转后，如果将UINavigationController的toolbarHidden属性设置成了NO，程序还会以后者的设置优先。

3.5.3　向工具条中追加按钮

关于向工具条中追加按钮的知识，请参照下面第3.6节的介绍。

3.5.4　工具条的颜色

可以通过改变UIToolbar类的tintColor属性，改变导航条的背景颜色。以下是实例代码。

```
// 将工具条变成蓝色
self.navigationController.toolbar.tintColor = [UIColor blueColor];
```

此代码的执行结果如图3-22所示（图中工具条的背景实际为蓝色）。

图3-22 改变工具条的颜色

3.6 按钮项目

3.6.1 系统按钮

可以在导航条、工具条中追加各种各样的UIBarButtonItem（注意UIBarButtonItem为UIBarItem的子类，而与UIView没有继承关系），UIKit中事先提供了各种系统按钮。创建系统按钮时，使用UIBarButtonItem的**initWithBarButtonSystemItem:target:action:方法**。以下是具体的实例代码。

```
UIBarButtonItem* button =
    [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarBut
tonSystemItemUndo
                    target:self
                    action:@selector(buttonDidPush)] autorelease];
```

上述代码中，首先向其第一个参数中传入常量**UIBarButtonSystemItemUndo**，此按钮为刷新画面专用的系统按钮。如图3-23所示。

图3-23 刷新按钮

接着向target中传入self，向action中传入@selector(buttonDidPush)后，当用户触摸此系统按钮时，将调用本类中定义的buttonDidPush方法。使用**initWithBarButtonSystemItem:target:action:方法**，可以追加各种系统按钮，当用户触摸时执行各种对应的处理。表3-3中罗列了全部系统按钮。

表3–3　系统按钮列表

UIBarButtonSystemItem	示　意　图	用　途
UIBarButtonSystemItemDone		保存变化，结束当前模式
UIBarButtonSystemItemCancel		不保存变化，结束当前模式
UIBarButtonSystemItemEdit		进入编辑模式
UIBarButtonSystemItemSave		保存变化，必要时结束当前模式
UIBarButtonSystemItemAdd		追加新项目
UIBarButtonSystemItemCompose		显示创建新项目的视图（画面）
UIBarButtonSystemItemReply		给显示的信息回复，或将显示项目传送到其他地方
UIBarButtonSystemItemAction		打开执行包含应用程序定义动作的操作表
UIBarButtonSystemItemOrganize		将显示项目保存于目录等地方，或者向其他地方送信
UIBarButtonSystemItemBookmarks		显示应用程序书签画面
UIBarButtonSystemItemSearch		显示检索输入框或者检索画面
UIBarButtonSystemItemRefresh		更新内容
UIBarButtonSystemItemStop		中止当前运行中的过程或者任务
UIBarButtonSystemItemCamera		显示包含“摄影”、“选择相册”等动作的操作表

续 表

UIBarButtonSystemItem	示 意 图	用 途
UIBarButtonSystemItemTrash		删除显示项目
UIBarButtonSystemItemPlay		播放/浏览内容
UIBarButtonSystemItemPause		暂定播放
UIBarButtonSystemItemRewind		进入前一内容
UIBarButtonSystemItemForward		进入后一内容
UIBarButtonSystemItemUndo		Undo/取消操作
UIBarButtonSystemItemRedo		Redo／回复取消的操作

3.6.2 工具条按钮间距的调整

表3-3中罗列了所有的系统按钮，实际UIKit中还提供了两个没有出现在表中的常量。分别是**UIBarButtonSystemItemFlexibleSpace**以及 **UIBarButtonSystemItemFixedSpace**。这些也是**UIBarButtonSystemItem**类型常量，但是不是按钮，而是调整按钮间距用的对象。例如，如果没有进行任何处理，依次追加4个按钮后，按钮将显示在工具条左侧，如图3-24所示。

图3-24 左对齐的工具条按钮

如果要让4个按钮等间距地分布在工具条中，在使用UIViewController的**setToolbarItems:方法**追加按钮时，如下述代码一样在4个按钮之间追加UIBarButtonSystemItemFlexibleSpace对象即可。

```
    [self setToolbarItems:[NSArray arrayWithObjects:
                    [self barButtonSystemItem:UIBarButtonSystemItemAction]
    // 追加间距对象UIBarButtonSystemItemFlexibleSpace
                    [self barButtonSystemItem:UIBarButtonSystemItemFlexi
bleSpace]
                    [self barButtonSystemItem:UIBarButtonSystemItemBookmarks]
    // 追加间距对象UIBarButtonSystemItemFlexibleSpace
                    [self barButtonSystemItem:UIBarButtonSystemItemFlexi
bleSpace]
                    [self barButtonSystemItem:UIBarButtonSystemItemReply]
    // 追加间距对象UIBarButtonSystemItemFlexibleSpace
                    [self barButtonSystemItem:UIBarButtonSystemItemFlexi
bleSpace]
                    [self barButtonSystemItem:UIBarButtonSystemItemCompose]
                    nil]];
```

这里为了让代码看起来更整齐，创建了一个新方法barButtonSystemItem:，只需要向此方法中传入系统按钮的常量就可以创建对应的系统按钮了，相关代码如下。

```
    - (UIBarButtonItem*)barButtonSystemItem:(UIBarButtonSystemItem)
systemItem {
      UIBarButtonItem* button =
        [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:systemItem
                                    target:nil
                                    action:nil] autorelease];
      return button;
    }
```

执行后，将显示如图3-25所示的效果。

图3-25　等间距按钮

如上述实例所示，**UIBarButtonSystemItemFlexibleSpace**能自动调节按钮间的间距。

另外，不仅可以调整按钮间的间距，将其配置到左端（传递给setToolbarItems:方法的数组的第一个元素）时，可创建靠右的工具条按钮（见图3-26）。同时配置到左右端（数组的第一项及最后一项）时，将创建居中的工具条按钮（见图3-27）。

图3-26 靠右的工具条按钮

图3-27 居中的工具条按钮

如果不想自动调整按钮间的间距，而是指定固定间距值时，使用**UIBarButtonSystemItemFixedSpace**。通过指定UIBarButtonSystemItemFixedSpace创建UIBarButtonItem实例，然后通过width属性指定宽度。以下是实例代码。

```
    // 指定 UIBarButtonSystemItemFixedSpace 创建UIBarButtonItem实例
    UIBarButtonItem*fixedSpace = [self barButtonSystemItem:UIBarButton
SystemItemFixedSpace];
    // 将宽度固定为35个像素
    fixedSpace.width = 35;
    // 以35个像素取代其中一个按钮
     [self setToolbarItems:[NSArray arrayWithObjects:
     [self barButtonSystemItem:UIBarButtonSystemItemAction],
     [self barButtonSystemItem:UIBarButtonSystemItemFlexibleSpace],
     [self barButtonSystemItem:UIBarButtonSystemItemBookmarks],
     [self barButtonSystemItem:UIBarButtonSystemItemFlexibleSpace],
        fixedSpace,
       [self barButtonSystemItem:UIBarButtonSystemItemFlexibleSpace],
       [self barButtonSystemItem:UIBarButtonSystemItemCompose],
      nil]];
```

代码执行后，显示如图3-28所示的效果。UIBarButtonSystemItemFixedSpace主要用于有特定按钮显示／隐藏间切换需要的场合，通过它当按钮隐藏时不至于破坏工具条的外观。

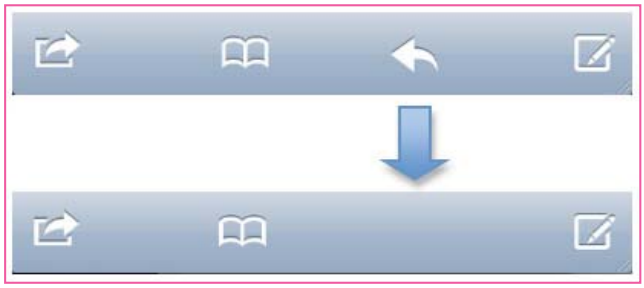


图3-28　固定间隔的使用效果

3.6.3　定制按钮

上一小节介绍了UIKit中提供的系统按钮，导航条以及工具条中还可以追加自定义按钮，如图3-29所示。

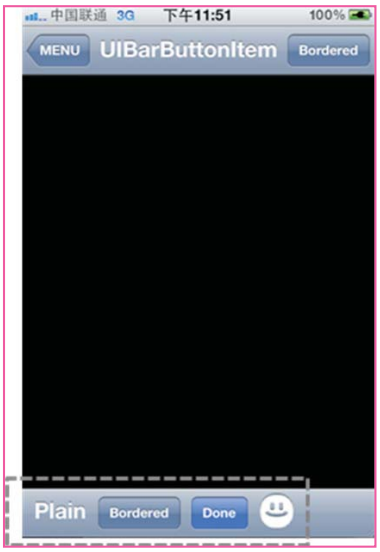


图3-29　自定义按钮

首先，可以使用initWithTitle:style:target:action:方法创建文本按钮，另外还可以使用initWithImage:style:target:action:方法创建图标按钮。图3-30、图3-31分别是文本按钮以及图标按钮。

图3-30　文本按钮

图3-31　图标按钮

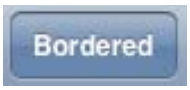
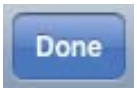
指定图标图片创建按钮时，请注意图片将被变换为单色。原理与标签条图标相同（请参照第3.3.3节）。以下是具体的实例代码。

```
// 创建文本按钮
UIBarButtonItem*button = [[[UIBarButtonItem alloc] initWithTitle:@"Plain"
                    style:style
                    target:nil
                    action:nil] autorelease];
// 创建图标图片按钮
UIImage*image = [UIImage imageNamed:@"smile.png"];
UIBarButtonItem*icon = [[[UIBarButtonItem alloc] initWithImage:image

style:UIBarButtonItemStylePlain
                     target:self
                     action:@selector(buttonDidPush:)] autorelease];
```

上述两种方法中，target与action上面已经介绍过，与事件响应有关。通过在style中指定不同的常量，可以改变按钮的外观。表3-4中罗列了常量值及其显示的外观区别。

表3-4 style中可指定的常量列表

UIBarButtonItemStyle	文本按钮	图标按钮
UIBarButtonItemStylePlain		
UIBarButtonItemStyleBordered		
UIBarButtonItemStyleDone		

其次，UIBarButtonItem中不仅可以使用文本以及图标图片创建，还可以使用任意UIView的子类进行创建（见图3-32）。

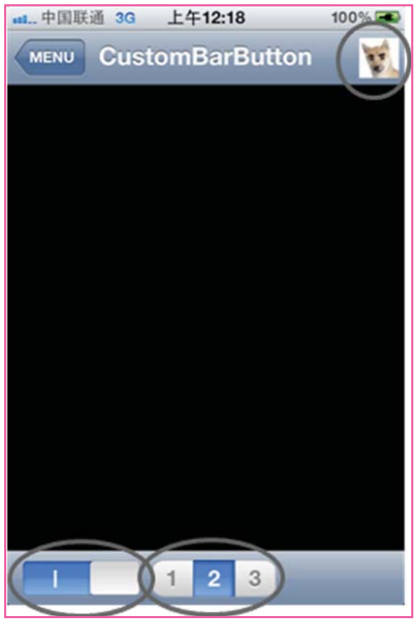


图3-32　定制按钮

有不适合创建定制按钮的类，下列为适合创建定制按钮的代表类。

- UIImageView。
- UISwitch。
- UISegmentedControl。

下面首先分别创建3种类的实例，然后作为参数传递到initWithCustomView:方法中，完成定制按钮的创建。代码如下。

```
        // 在导航条中追加UIImageView
        UIImage* image = [UIImage imageNamed:@"face.jpg"];
        UIImageView*imageView = [[[UIImageView alloc] initWithImage:image]
autorelease];
        UIBarButtonItem* icon =
            [[[UIBarButtonItem alloc] initWithCustomView:imageView]
autorelease];
        self.navigationItem.rightBarButtonItem = icon;

        // 向工具条中追加UISwitch
        UISwitch*theSwitch = [[[UISwitch alloc] init] autorelease];
```

```
        theSwitch.on = YES;
        UIBarButtonItem* switchBarButton =
            [[[UIBarButtonItem alloc] initWithCustomView:theSwitch]
autorelease];

        // 向工具条中追加UISegmentedControl
        NSArray*segments = [NSArray arrayWithObjects:@"1", @"2", @"3", nil];
        UISegmentedControl* segmentedControl =
            [[[UISegmentedControl alloc] initWithItems:segments]
autorelease];
        segmentedControl.selectedSegmentIndex = 1;
        segmentedControl.frame = CGRectMake( 0, 0, 100, 30 );
        UIBarButtonItem*segmentedBarButton =
           [[[UIBarButtonItem alloc] initWithCustomView:segmentedControl]
autorelease];

        [self setToolbarItems:[NSArray arrayWithObjects:
                switchBarButton,
                segmentedBarButton,
                nil]];
```

代码执行后效果如图3－32所示，分别在导航条中追加了UIImageView的按钮，在工具条中追加了UISwitch以及UISegmentedControl按钮。

3.7 UIViewController与相关类间关系概要

3.7.1 UIViewController与UIView/UIWindow的关系

UIViewController中以view属性的形式拥有UIView，此UIView作为UIWindow的subviews追加进来显示在画面中。为反映这种联系，以让各元素间的关系（通过属性）更清楚地呈现在大家面前，特绘制了如图3-33所示的对象关系图。

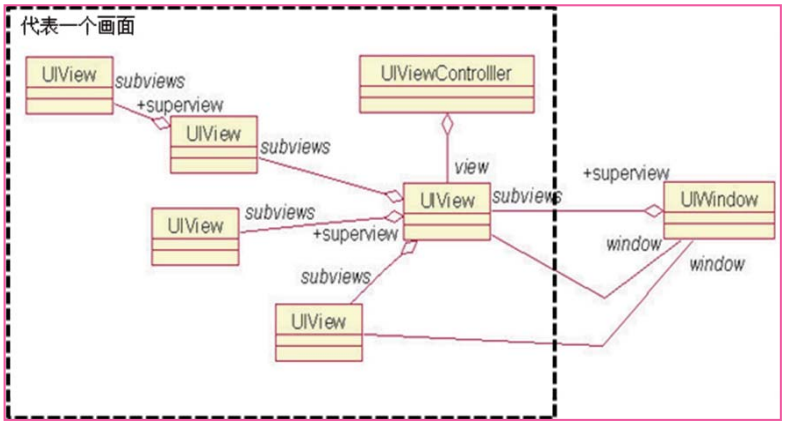


图3-33 UIViewController类与UIView/UIWindow间的关系

图中所有用线关联起来的对象之间均有属性关系。其中用带四方形箭头的线连接的两个对象之间，四方形箭头指向的一方拥有另一方。例如UIViewController与中间的UIView之间，UIViewController以view属性的形式拥有UIView对象，属性名称显示在靠近UIView一边。此图中反映的关系特征如下。

- UIViewController拥有一个UIView（view属性）。
- UIViewController的view中可以追加任意数目的UIView以创建画面。
- UIView中可以包含多个UIView(subviews)。
- UIView可通过superview属性参照父UIView。
- 如果是UIWindow的子元素（subviews），则可通过Window属性参照UIWindow。

3.7.2 UITabBarController与各画面的关系

接着我们将UITabBarController与各画面的关系归纳为如图3-34所示。

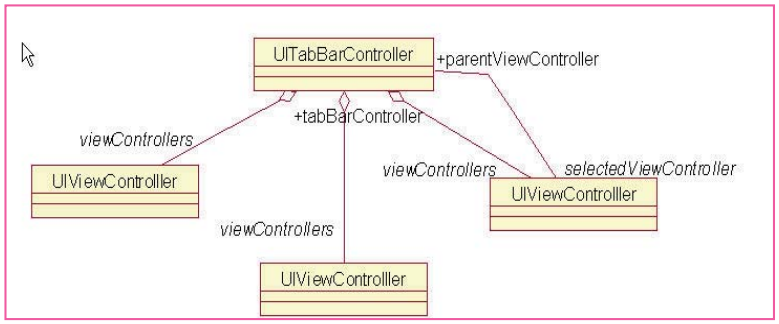


图3-34 UITabBarController与各画面的关系

此图反映的关系特征如下。

- UITabBarController通过viewControllers属性管理多个UIViewController。
- viewControllers的UIViewController中，可通过selectedViewController属性参照当前画面的UIViewController。
- 对于viewControllers中的UIViewController来说，可通过tabBarController属性参照UITabBarController。
- 对于viewControllers中的UIViewController来说，也可通过parentViewController属性参照UITabBarController。

3.7.3 UINavigationController与各画面的关系

这里将UINavigationController与各画面的关系归纳为如图3-35所示。

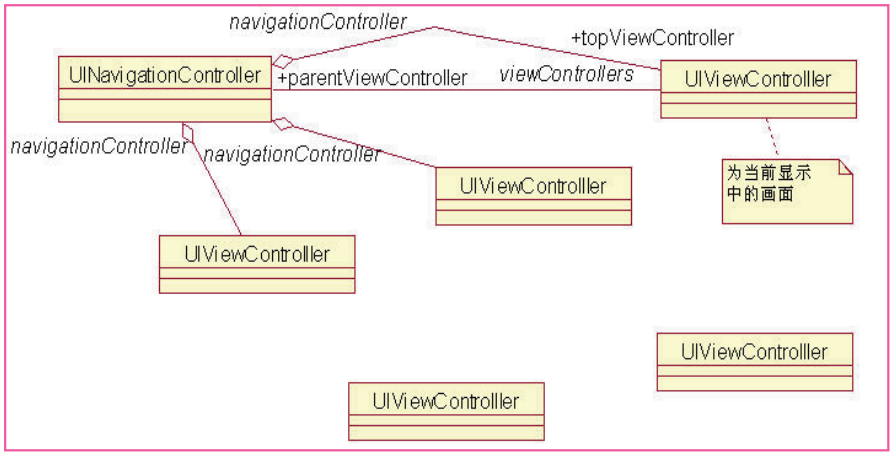


图3-35 UINavigationController与各画面的关系

此图反映的关系特征如下。

- UINavigationController的viewControllers属性中保存了跳转路径中所有画面的UIViewController。
- 跳转路径中所有画面的UIViewController中，可通过topViewController取得当前画面的UIViewController。
- 对于跳转路径中所有画面的UIViewController来说，可以通过其navigationController属性参照UINavigationController。

● 跳转路径中已经退出的画面与UINavigationController没有任何关系。

● 跳转路径中各画面的parentViewController属性参照的不是上一画面而是UINavigationController。

3.7.4 UIViewController与模态画面的关系

这里将UIViewController与模态画面的关系归纳为如图3-36所示。

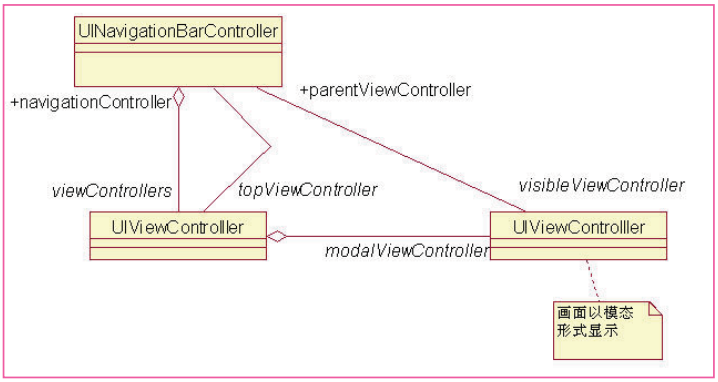


图3-36 UIViewController与模态画面间的关系

此图反映的关系特征如下。模态画面可通过当前画面UIViewController的modalViewController属性进行参照。

● 就算模态画面被显示，UINavigationController的topViewController也不是指向模态画面，而是仍然指向弹出模态画面的原画面的UIViewController。

● 对于UINavigationController的visibleViewController属性来说，如果模态画面被显示，则指向模态画面。

● 模态画面的parentViewController并非指向模态画面的母画面，而是指向UINavigationController。

3.8 UIViewController的状态监视

3.8.1 状态通知方法

关于UIViewController的状态监视相关的方法，可归纳为如表3-5所示。

表3-5 UIViewController的状态监视方法列表

方 法 名	调用时机及注意点
viewDidLoad	UIViewController中的UIView被导入时调用。 画面创建完成后紧接着被调用的方法，并非每次画面显示时都被调用。例如，通过UINavigationController跳转到下一画面后，再次返回到本画面时将不被调用。同时必须注意，并非意味着仅仅被调用一次
viewWillAppear	画面显示时必被调用。 例如，必须显示导航条或者工具条的画面，不是在viewDidLoad中，而是在此方法中进行设置
viewDidAppear	画面显示后被调用
viewWillDisappear	画面隐藏前被调用。 下一画面显示前需要处理的东西，加入到此方法中
viewDidDisappear	画面隐藏后被调用

只通过表3-5还不能确切知道画面跳转时（跳转前/跳转后）到底何种时机调用何种方法，画面跳转时调用具体方法的时机可归纳为如图3-37所示。

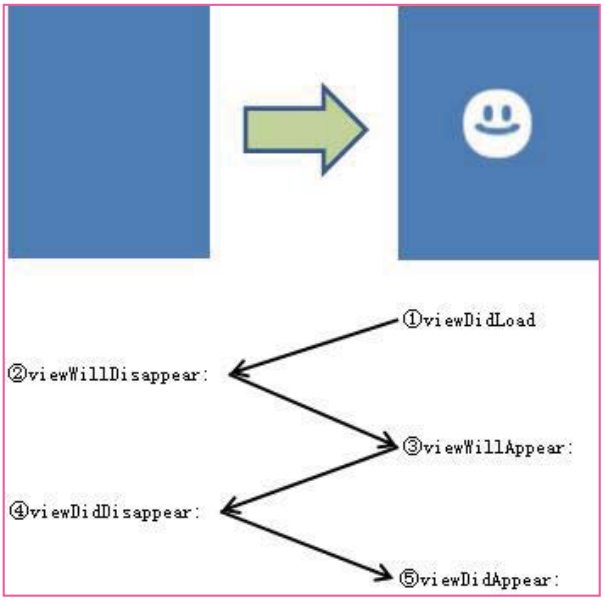


图3-37 画面跳转时各方法的调用时机

3.8.2 基点view的导入方法

UIViewController中拥有画面基点的UIView（即view属性），此UIView是否被完全导入（load）可以通过isViewLoaded方法进行检测。根据上一小节介绍的viewDidLoad方法被调用的时机来看，当然此时isViewLoaded方法将返回YES。那么，这个view实际上被导入的时机到底是什么时候呢？

我们可以通过重写（overwrite）loadView方法来进行一下调查，实例代码如下。

```
-(void)loadView{
  NSLog(@"before loadView:%d",[self isViewLoaded]);
  [super loadView];
  NSLog(@"after loadView:%d",[self isViewLoaded]);
}
```

结果我们可以看到，在“super loadView”之前isViewLoaded方法返回NO，在其之后返回YES。因此我们可以看出，在loadView方法中基点UIView被导入了。我们可以像上述实例代码中一样重写loadView方法，在其中不调用父类的loadView方法，而是创建自定义的基点UIView，以达到重新定制画面的目的。

3.8.3 内存不足时的解决方式

在iPhone应用程序中内存管理非常重要。如果不加注意内存被消耗完的话，应用程序最终将无法运行，会被强制停止。特别需要注意的是对于那些跳转画面层次比较深，跳转画面比较多的应用程序来说，跳转路径中的所有画面默认都将是驻留内存的。作为内存不足时的对策，当出现内存不足时，在关闭画面前可在各个画面中显示警告信息。这些显示警告信息的具体处理代码放在UIViewController的didReceiveMemoryWarning方法中。而且从iOS 3.0开始，此时还将调用后台待机画面的viewDidUnload方法，释放内存的处理就可以放在此方法中。而iOS 3.0以前，只能通过didReceiveMemoryWarning方法追加内存不足的处理。以下是内存不足对策的具体实例代码。

```
[interface]
// 在此类的imageView_实例变量中保存巨大的图像对象
@interface UIKitPrjObserving :UIViewController
```

```
{
@private
    UIImageView* imageView_;
    NSInteger count_;
}
@end

[implementation]
// dealloc中同样也追加了释放imageView_实例变量的处理
-(void)dealloc{
  [imageView_ release];
  [super dealloc];
}

// 画面导入时创建容纳巨大图像的UIImageView实例,并保存在imageView_中
-(void)viewDidLoad{
  [super viewDidLoad];

  UIImage* image = [UIImage imageNamed:@"town.jpg"];
  imageView_ = [[UIImageView alloc] initWithImage:image];
  imageView_.frame = [self.view.bounds];
  imageView_.autoresizingMask =
  UIViewAutoresizingFlexibleWith | UIViewAutoresizingFlexibleHeight;
  [self.view addSubview: imageView_];
}

// 此方法当内存不足时才调用
// 方法中释放imageView_,解决内存不足的问题
// 释放如果不将其赋值为nil,误访问时会出现应用程序强制关闭
-(void)viewDidUnLoad{
  [super viewDidUnLoad];
```

```
    [imageView _ release];
    imageView _ = nil;
}
```

像上述实例一样，在viewDidUnLoad方法追加内存不足时的对策处理，注意此方法只在发生内存不足时才会被调用。

最后，如果后台驻留了多个画面的情况下，关于viewDidUnLoad方法与didReVeiveMemoryWarning方法的调用顺序归纳为如图3-38所示。正如图中所示，最上层画面的viewDidUnLoad方法将不被调用，而且所有画面的didReceiveMemoryWarning方法都将被调用。

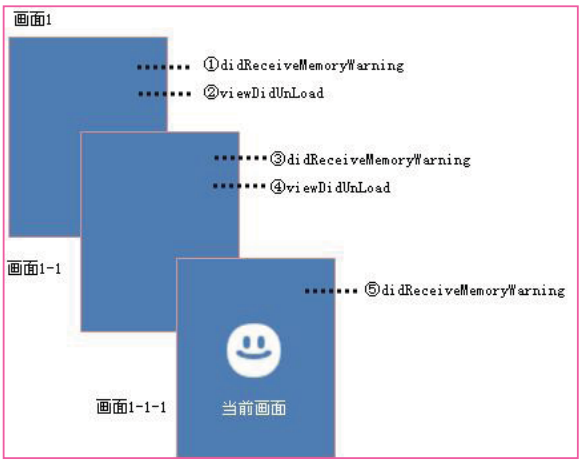


图3-38　内存不足时调用方法的顺序

如果想在iPhone模拟器中再现内存不足的现象，可以使用iPhone模拟器中提供的“模拟内存警告”功能。使用此功能可向应用程序发出内存不足警告。

“模拟内存警告”菜单在iPhone模拟器主菜单的“硬件”之下，如图3-39所示。

图3-39　模拟内存警告的菜单

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

iPhone UIKit详解

本书是一本UIKit开发大全，不仅让每一位iPhone程序员学会UIKit框架的应用，加深对这一框架的理解，更能在开发iPhone应用程序时实时参考。

- **介绍了UIKit框架的基本概念**以及如何在不使用Xcode的“所见即所得”界面编辑功能下进行iPhone编程。
- **iPhone应用程序画面基础**，介绍构成iPhone应用程序画面的基本类。包括构成画面的UIView及各种常用UI控件（皆为UIView子类），以及画面控制相关的UIViewController及其子类。
- **iPhone特色画面组成控件**，介绍了图形、动画、文字显示、屏幕效果等所有iPhone特色效果的实现方式。
- **UIKit框架中的事件与动作控制类**，介绍各种事件处理方法及用户交互相关的API。
- **UIKit框架中的其他重要知识**，其中介绍了设置/获取应用程序及设备信息的功能，以及拷贝/粘贴功能，还介绍了与摄像头及视频相关的各种API及使用方法。

本书适合拥有一定Objective-C语言基础的读者作为iPhone应用程序开发时的参考书，或者作为学习iPhone软件开发的进阶参考资料，尤其可加深关于UIKit框架部分的理解。

本书代码下载链接

http://www.softechallenger.com/download/?ISBN=ISBN 978-7-121-17100-0

上架建议：移动开发

ISBN 978-7-121-17100-0

9 787121 171000 >

定价：79.00元

策划编辑：孙学瑛
责任编辑：付　睿
封面设计：侯士卿