您的第一个 iPhone 应用程序

介绍

本教程向您演示如何创建一个简单的 iPhone 应用程序。本文不打算对 iPhone 目前可用的特性作全面介绍，而是介绍一些技术，让您对基础开发过程有初步了解。

如果您刚开始使用 Cocoa Touch 进行 iPhone 开发，则您需要阅读本文档。阅读之前，您需要大致了解计算机编程基础，尤其是要了解 Objective-C。如未曾用过该语言，则至少应通读 *学习 Objective-C：入门教材*。

本文档不是为了创建一个优雅漂亮的应用程序，而是为了向您描述：

- 如何使用 Xcode 创建并管理一个工程
- 基础设计模式和 iPhone 开发的基本技术
- Interface Builder 使用入门
- 如何让应用程序响应来自标准用户接口控件的用户输入

另外，我们还在教程中指出其他一些文档。只有阅读这些文档，您才能充分理解 iPhone 开发的工具和技术。

重要：为了学习本教程，您需要安装 iPhone SDK 和开发者工具，它们位于 iPhone 开发中心。
文档描述的工具包含在 iPhone SDK v3.0 里面—请检查一下 Xcode 版本，它不能低于 3.1.3。

文档的组织方式

本文档分为如下章节：

教程概述和设计模式

本章概述您将要创建的应用程序以及将会使用的设计模式。

教程概述

在学习过程中，您将创建一个很简单的应用程序。它含有一个文本字段，一个标签和一个按键。您可以把名字输入到文本字段中，再按下按键，这时标签的文本就会变成 “Hello, <Name>!”:

尽管这是个很简单的应用程序，但它介绍了 基本的设计模式、工具、以及利用 Cocoa Touch 进行 iPhone 开发的基础技术。 Cocoa Touch 包括 UIKit 和 Foundation 这两个框架。当在 iPhne OS 上开发事件驱动的图形化应用程序时，您需要使用它们提供的工具和基本结构。同时，Cocoa Touch 还包含其他几个框架，它们提供一些基本的服务，可用于访问设备的特色内容，例如访问用户的联系人。如需要进一步了解 Cocoa Touch 及其在哪些方面适应于 iPhone OS，请阅读 *iPhone OS 技术概览*。 另外，我们将在“设计模式”一节中描述您将使用的主要的设计模式。

虽然本教程不太顾及用户界面，但应用程序的表现形式是其获得成功的关键。您应该阅读 *iPhone 人机接口指南* 并且研究基于本文档的样例代码(HelloWorld)，这样您才能明白如何改善用户接口，以使其成为一个成熟的应用程序。

同时，您也将了解视图控制器如何工作以及它如何同 iPhone 应用程序的的架构相适应。

设计模式

请务必阅读 *Cocoa 基础指南* 的设计模式这一章。您将使用主要的模式如下：

- 委托
- 模型 视图 控制器
- 目标-动作

下面对这些模式作简单介绍并且指出应用程序在什么地方会使用它们。

委托

委托模式是一个对象周期性地向被指定为其委托的另一个对象发送消息，向其请求输入或者通知某件事情正在发生。该模式可替换类继承来对可复用对象的功能进行扩展。

在本文将要创建的应用程序中，应用程序对象会向其委托发送消息，通知它主要的启动例程已经完成并且定制的配置可开始执行。为了建立并管理视图，委托会创建一个控制器实例。另外，当用户点击 Return 按键后，文本字段也会通知它的委托（即所创建的控制器对象）。

委托方法通常会集中在一起形成一份 协议。 一份协议基本上就是一个方法的列表。如果一个类遵循某个协议，则它要保证实现协议所要求的方法（有些方法可选择实现与否）。委托协议规定了一个对象可以发送给委托的所有消息。如果需要进一步了解协议及其在 Objective-C 中的作用，请查看 *Objective-C 编程语言*的协议。

模型-视图-控制器

模式-视图-控制器 (或者“MVC”)设计模式把应用程序中的对象设定为三种角色。

模型对象表示数据。例如，在一款游戏中，SpaceShips 和 Rockets 是模型对象，在一个用于生产的应用中，ToDo 项和 Contacts 是模型对象，在一个绘画应用中，Circles 或 Squares 是模型对象。

本文将创建的应用程序用到的数据非常简单-仅仅是一个字符串-并且该字符串只在一个方法中使用，因此，严格说，我们甚至没有必要在程序中使用模型对象，但是程序所用到的设计原理却非常重要。在其他的应用程序中，模型对象将会更加复杂并且可以在多个地方进行访问。

视图对象知道如何显示数据（模型），并且它们有可能会允许用户对数据进行编辑。

在本文将要创建的应用程序中，您需要一个主视图来包含其它几个视图— 一个文本字段，它用于捕获用户输入信息；第二个文本字段，它用于显示文本，而文本内容则是基于用户的输入；另外还需要一个按键，用户利用它来告知我们第二个文本字段应该被更新。

控制器对象位于模型和视图之间。

在本文将要创建的应用程序中，控制器对象将会从输入文本字段中取得数据，并把数据存放在一个字符串中，然后再把第二个文本字段的内容更新成恰当的值，更新操作则由按键发送出来的动作触发。

目标-动作

目标-动作机制允许一个控件对象(诸如按键或滑动条) 向另外一个对象发送一条消息（即动作），以之作为对某个用户事件（例如一个点击事件或者一个敲击事件）的响应。接收到消息的对象则可以对消息进行解释，并将其作为一个特定于应用程序的指令进行处理。

在本文将要创建的应用程序中，当按键被敲击时，它会通知控制器根据用户的输入更新模型和视图。

创建工程

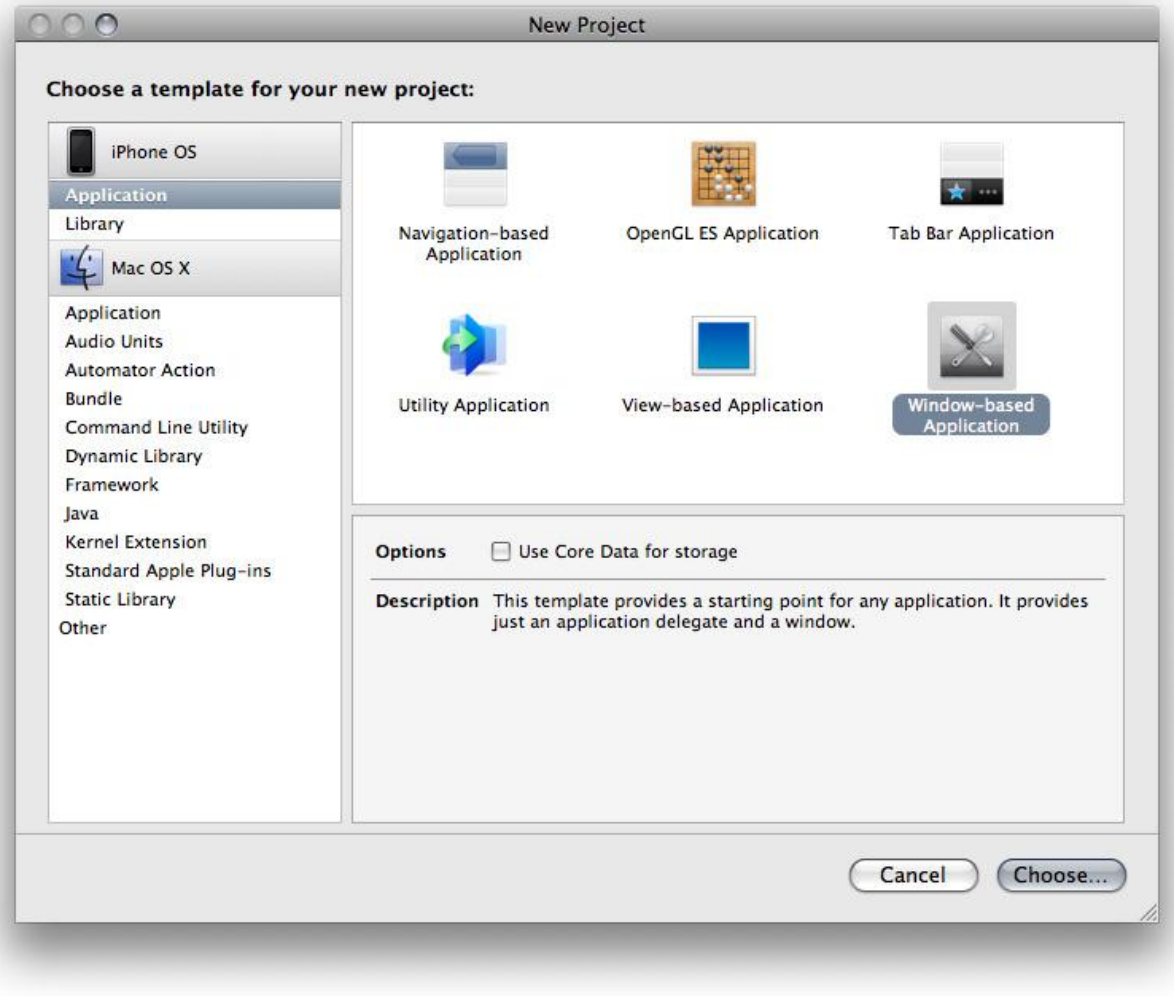
在本章，您将使用 Xcode 创建前面所说的工程，同时还将查明应用程序的启动过程。

Xcode

我们主要利用 Xcode 来创建 iPhone 应用程序，它是苹果的 IDE(集成开发环境)。您也可以利用它来创建各种不同类型的工程，包括 Cocoa 以及命令行工具。

请注意： 本书,我们约定>> 表示一个段落的开始（有时,该段落包含其后的无序列表），而段落内容是教程里需要您执行的操作。

代码列表不显示 Xcode 的模板文件里的注释。
>> 启动 Xcode(缺省情况下，Xcode 位于 `/Developer/Applictions` 里面)，然后请选择 File > New Project，这样就可以创建一个新工程。您应该会看到一个新的窗口,它和下图相似：



请注意：如果您没有看到 “Use Core Data for storage”这一选项，则请您务必您安装 iPhone OS SDK 3.0 版本—您应该安装 Xcode3.1.3 或者更高的版本。

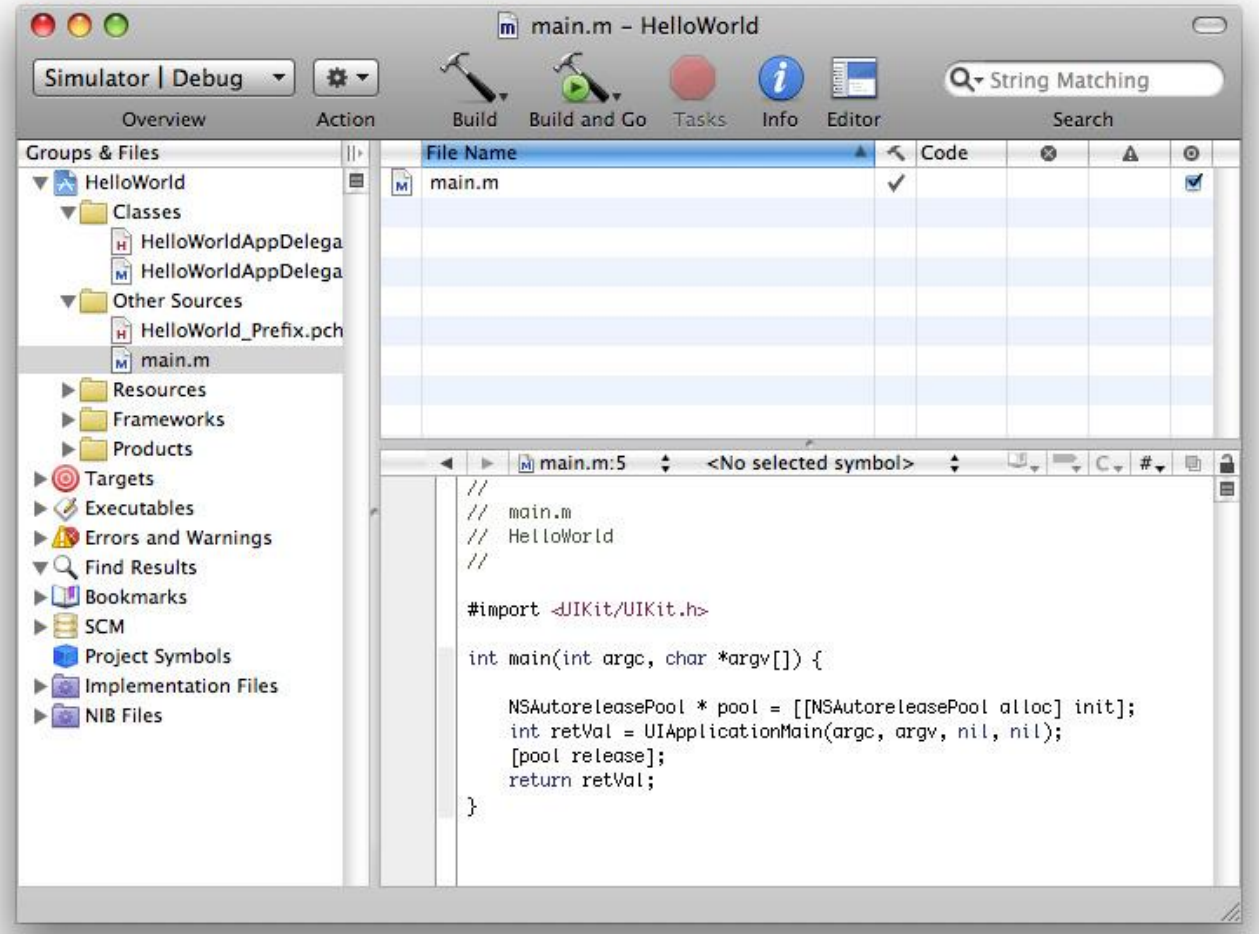
>> 请选中 Window-Based Application 并点击 Choose 按键。 (请*不要*勾选“Use Core Data for storage”，本例不使用 Core Data 机制。)

完成上述步骤后，屏幕会出现一张表格。请在上面选择工程的存储位置。

>> 请选择一个合适的位置（例如您可以放在桌面也可以放在一个定制的工程目录），然后为工程添加一个名称—`HelloWorld`—再点击保存按键。

请注意：在后续章节中,我们假定您将工程命名为 `HelloWorld`，因此应用程序的委托类就叫做 `HelloWorldAppDelegate`。如果使用其他名称，则应用程序委托类的名称将为 *YourProjectName*`AppDelegate`。

完成上述步骤后，您将看到如下的新工程窗口：



如果以前未曾用过 Xcode，则请花点时间来研究下该应用。请阅读 *Xcode 工作空间指南*，它可以帮助您理解工程窗口的组织方式以及如何执行诸如编辑和保存文件这样的基本任务。现在，您可以链编并运行程序，这样就能看到模拟器的外观。

>> 请选择 Build > Build and Go (Run)或者点击工具栏中的 Build and Go 按键。

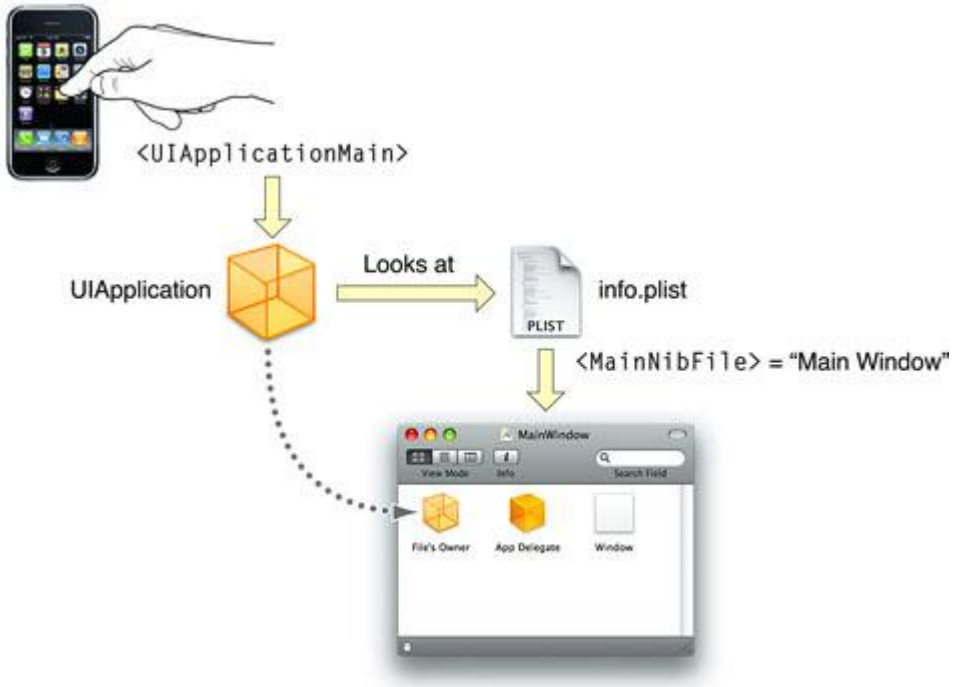
iPhone 模拟器应该会自动启动。当您的应用程序启动后，您只看到一个白色屏幕。如希望了解白色的屏幕从何而来，则您需先了解应用程序如何启动。

>> 退出模拟器。

应用程序引导

您创建的模板工程已设置了基本的应用程序环境。它创建一个应用程序对象，将应用程序和窗口服务器连接起来，建立一个运行循环以及其他等等。大部分的工作通过 `UIApplicationMain` 函数完成，请看图 2-1。

图 2-1 应用程序引导



main.m 文件中的 main 函数会调用 UIApplicationMain 函数:

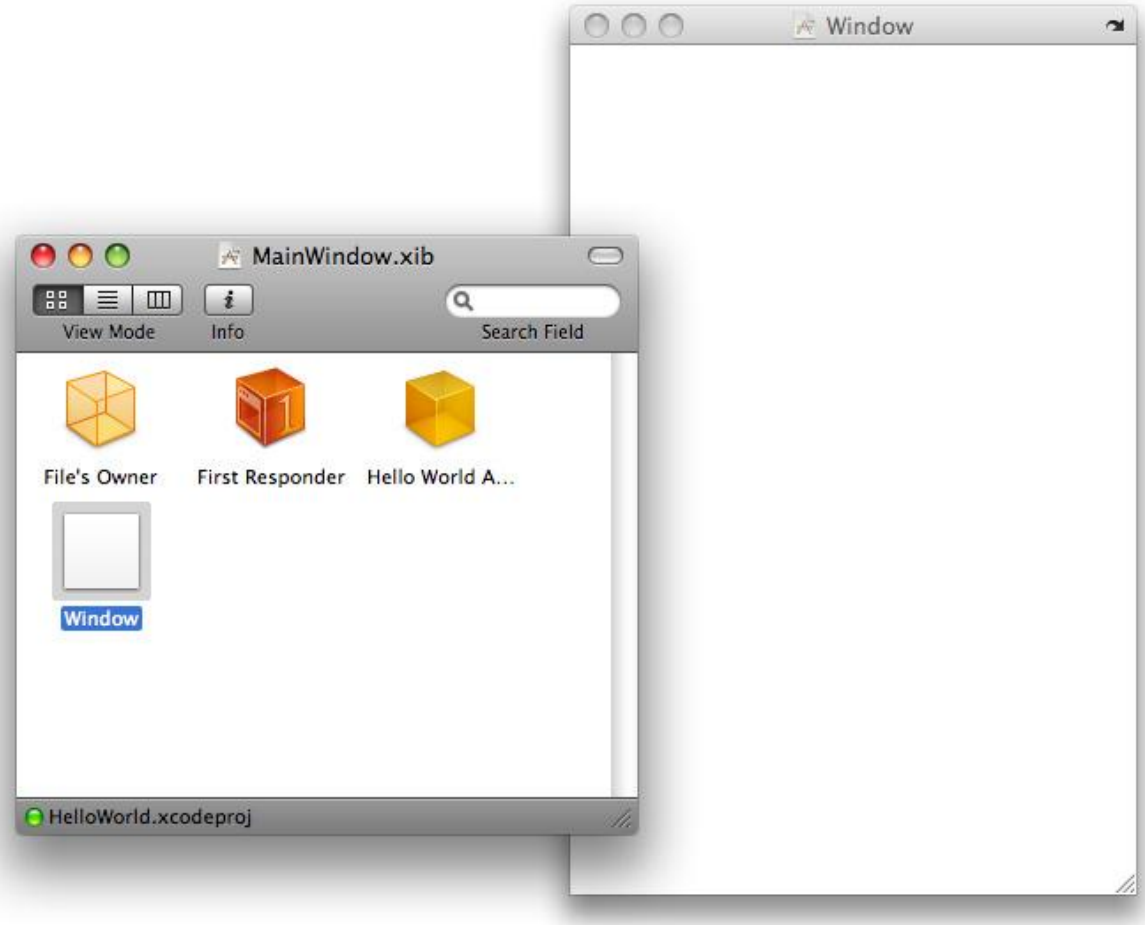
```
int retVal = UIApplicationMain(argc, argv, nil, nil);
```

该函数将会创建一个 UIApplicaion 类的实例。同时它会搜索应用程序的 Info.plist 属性列表文件。Info.plist 文件是一部字典，它包含诸如应用程序名称、图标这样的信息。它也可以包含应用程序对象应该加载的 **nib 文件**的名称，该名称由 NSMainNibFile 键指定。Nib 文件含有一份用户接口元素及其他对象的档案—您将在后续章节进一步了解 Nib 文件的知识。本工程的 Info.plist 文件具有下面的内容：

```
<key>NSMainNibFile</key>
<string>MainWindow</string>
```

这表明应用程序启动时将会加载 MainWindow nib 文件。

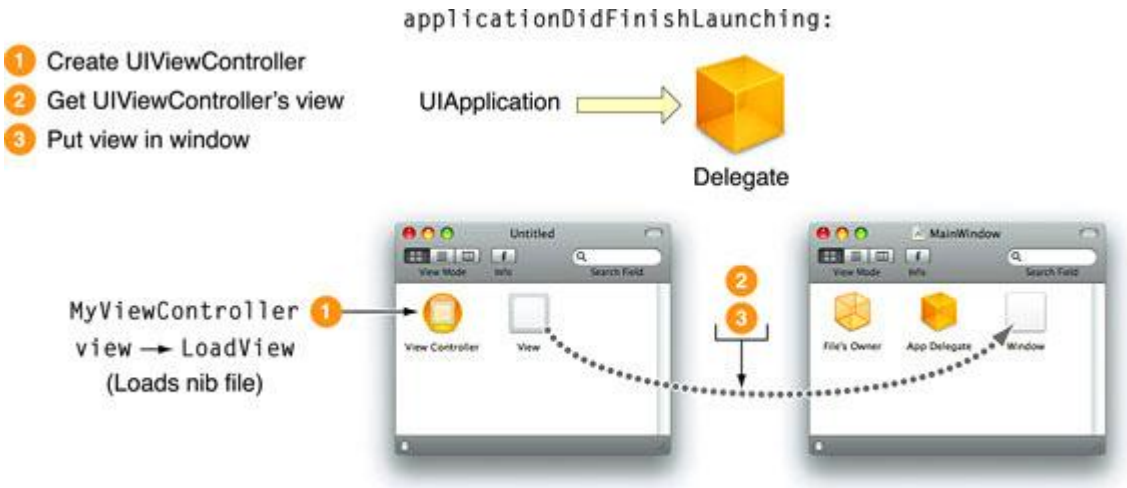
>> 如希望查看 nib 文件，请双击工程窗口 Resource group 中的 MainWindow.xib 文件。（虽然该文件的扩展名为"xib"，但是我们习惯称之为"nib 文件"）。Interface Builder 将会启动并打开该文件。



Interface Builder 文档包含四个对象：

- 一个 **文件拥有者**代理对象。实际上，文件拥有者对象是 `UIApplication` 实例—我们将在“文件拥有者”一节讨论该对象。
- 一个 **第一响应者**代理对象。 本教材并未使用第一响应者，但是您可以阅读 i*iPhone 应用程序的编程指南*中的事件处理以了解更多信息。
- 一个 `HelloWorldAppDelegate` 的实例，它会被设置成应用程序的**委托**。我们将在下一节讨论委托。
- 一个窗口。它被设置为白色背景、启动时可见。应用程序启动时，您看到的窗口就是它。

应用程序完成启动后，您可以执行附加定制。下图描述一种通用模式－您将在下一章使用它：



应用程序对象在完成启动后会向委托发送 `applicationDidFinishLaunching:`消息。通常情况下，委托不是自己配置用户接口，而是创建一个**视图控制器**对象（一种特定的控制器，它负责管理一个视图—遵循"模型-视图-控制器"描述的模型-视图-控制器设计模式）。然后委托向视图控制器请求视图（这个视图由视图控制器根据要求创建），并将其添加成窗口的子视图。

小结

在本章，您创建了一个新工程并学习了应用程序的启动过程。下一章，您将定义并创建一个视图控制器实例。

添加一个视图控制器

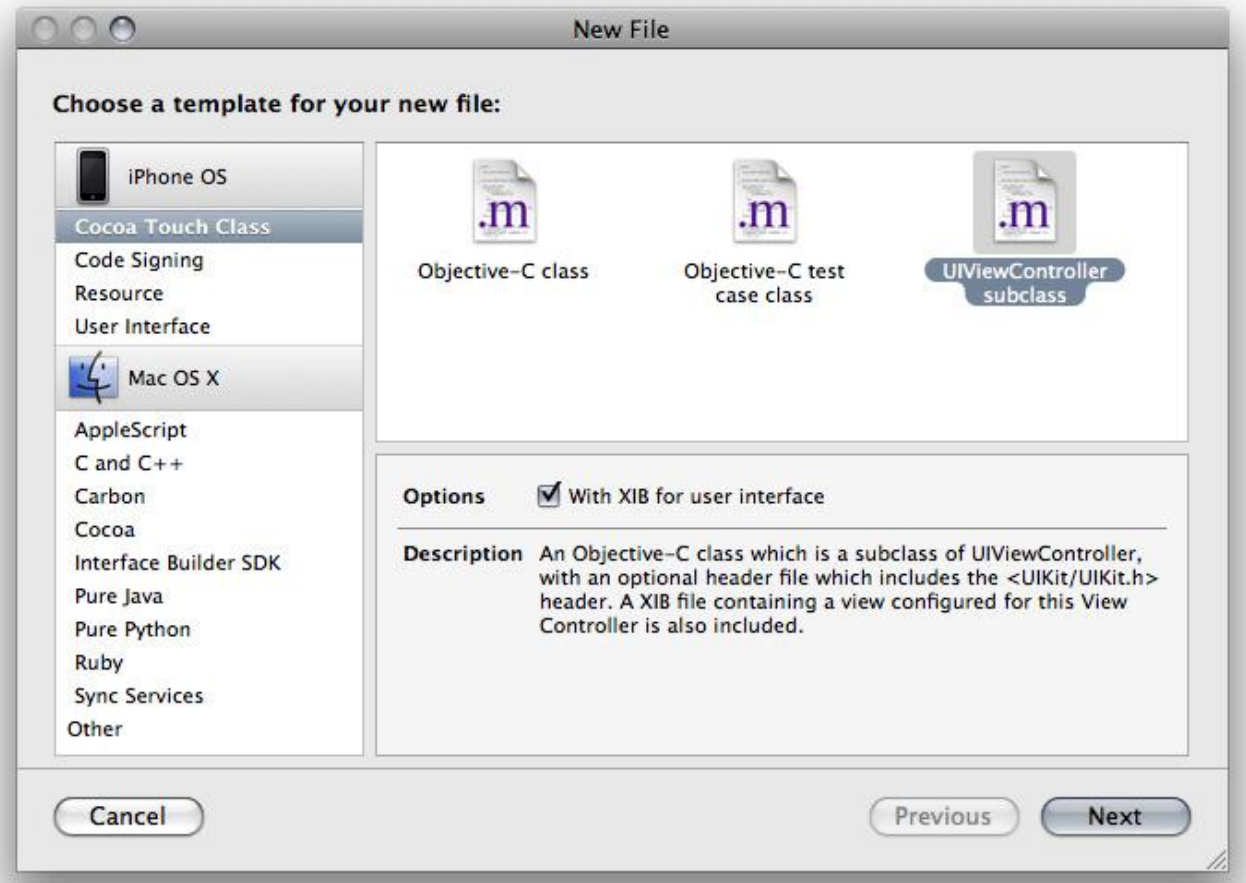
在本文档的示例程序中，您需要使用两个类。一个是 Xcode 的应用程序模板提供的应用程序委托，程序在 nib 文件中创建了一个该类的实例。另一个是需要您实现视图控制器类，您将创建该类的一个实例。

添加一个视图控制器类

在大部分 iPhone 应用程序中，视图控制器起着核心作用。正如其名称所示，它负责管理一个视图。在 iPhone 上，它们也帮助进行导航和内存管理。虽然本节例子程序不使用后两种功能，但对此有所了解很重要。UIKit 提供一个特别的类 — 即 `UIViewController` 类—它封装了视图控制器应该具有的大部分缺省行为。您应从它派生子类，在子类中定制应用程序的行为。

>> 请选中 Xcode 项目管理器里的工程（即 `HelloWorld` 项目，位于 Groups and Files 列表的顶部）或者选中 Classes 文件夹 — 新文件会被加入到当前选择的位置。

>> 请选择 File > New File。在 New File 窗口中，请选择 Cocoa Touch Classes，然后选择 `UIViewController` subclass。同时，请勾选 Options 区域中标题为 With XIB for user interface 的选择框。

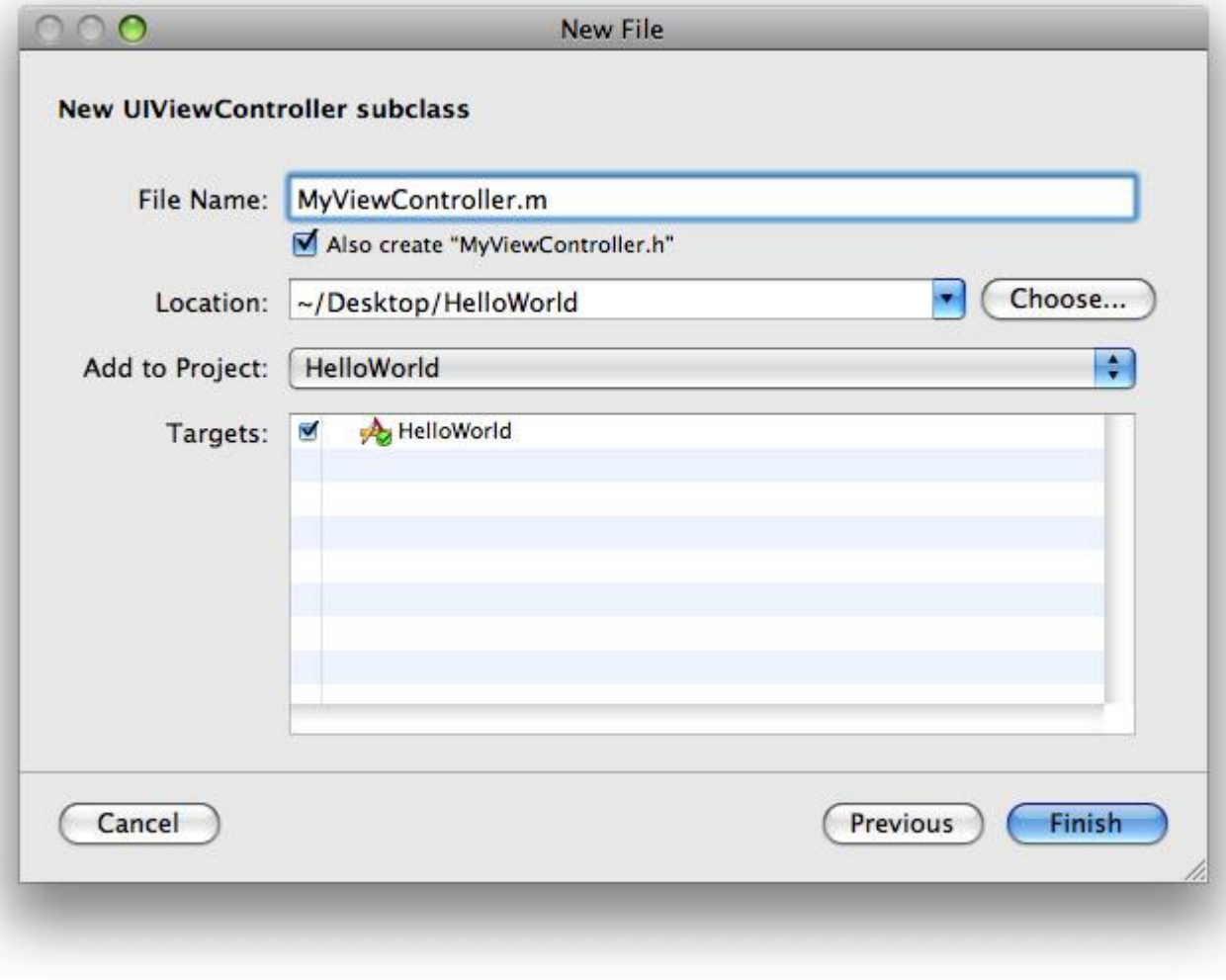


请注意： 如果您没有看到“With XIB for user interface”选项，请确保您安装 iPhone OS SDK 3.0 版本-您应该安装 Xcode3.1.3 或者更高的版本。

选中 “With XIB for user interface”表明 Xcode 在创建视图控制器的同时，会为其创建一份 nib 文件，并将该文件添加到工程。（我们将在下一章详细讨论 Nib 文件。)

>> 请点击 Next，在其后出现的屏幕中为文件起个名字，例如 `MyViewController`。（类名称习惯以一个大写字母开头）。请务必创建 `.m` 和 `.h` 文件，并将二者都添加到工程，如下所示：

图 3-1 MyViewController



>> 请点击 Finish，文件会被添加到工程。

看一下新建的源文件，您会发现 Xcode 已经为您提供了各种方法的存根实现。目前我们无需追究这些方法的具体含义。接下来，我们将创建一个控制器类的实例。

添加一个视图控制器属性

您需确保在应用程序生存期间，视图控制器始终存在。将视图控制器作为应用程序委托的一个实例变量是解决该问题的一个明智的方法。（如希望了解其中因由，请参考 *Cocoa 内存管理编程指南*。）

添加到应用程序委托的实例变量是 `MyViewController` 类的实例。如果您声明了变量但未告知编译器 `MyViewController` 类的相关信息，编译器就会报告错误。通过导入头文件可以解决该问题，但在 Cocoa 中，通常您应该使用一个前向声明（**forward declaration**）—它向编译器承诺 `MyViewController` 类将在其他地方定义，因此编译器现在无需耗时来对其执行检查。（如两个类需相互引用，则前向声明可以避免环状包含，即两个头文件互相包含。）然后，请将 `MyViewController` 类的头文件导入到应用程序委托的实现文件。

>> 请在应用程序委托头文件(`HelloWorldAppDelegate.h`)的接口声明前面-即 `HelloWorldAppDelegate` 声明前面-添加前向声明：

```
@class MyViewController;
```

>> 请在头文件大括号之间添加下面的代码，这是为了向应用程序委托添加一个实例变量：

```
MyViewController *myViewController;
```

>> 请在大括号之后 `@end` 之前添加下面的属性声明：

```
@property (nonatomic, retain) MyViewController *myViewController;
```

*Objective-C 编程语言*的声明属性有关于属性的描述，您可以参考。基本上，上述声明指定：`HelloWorldAppDelegate` 实例含有一个属性,您可以使用 getter 和 setter 方法—即 `myViewController` 和 `setMyViewController:`方法—来访问该属性，同时，委托实例还会保持该属性（我们将在后续章节详细讨论保持）。
为确保正确，请确认 `HelloWorldAppDelegate` 类的接口文件(即 `HelloWorldAppDelegate.h` 文件)如下所示（不显示注释）：

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

@class MyViewController;

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    MyViewController *myViewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) MyViewController *myViewController;

@end
```

现在您可以开始创建视图控制器的实例。

创建视图控制器实例

您已经把视图控制器属性添加到应用程序的委托，现在需要实际创建一个视图控制器实例，并将其设置为属性的值。

>> 请在应用程序委托类实现文件（即 `HelloWorldAppDelegate.m` 文件）中的 `applicationDidFinishLaunching:`方法开头添加如下代码，这些代码用于创建一个 `MyViewController` 实例：

```
MyViewController *aViewController = [[MyViewController alloc]
                initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
[self setMyViewController:aViewController];
[aViewController release];
```

虽然只有三行，但其中含意很多。这些代码作用如下：

- 创建并初始化一个视图控制器类的实例。
- 使用存取方法将新建的视图控制器是设置为 `myViewController` 实例变量值。

  请记住，您未单独声明 `setMyViewController:`方法，而是隐式将其作为属性声明的一部分—详情请参考“添加一个视图控制器属性”。
- 依照内存管理规则释放视图控制器。

您先使用 `alloc` 方法创建一个视图控制器，然后用 `initWithNibName:bundle:`方法对其进行初始化。`init` 方法先指定控制器应加载的 nib 文件，然后指定在哪个 bundle 中可找到该文件。bundle 是文件系统某个位置的抽象，该位置存放了应用程序将会用到的代码和资源。相比自行定位文件系统的资源文件，使用 bundle 有很多优势。它为我们提供了方便而简单的 API—bundle 对象仅通过名称就可以定位某个资源—甚至连名称的本地化的工作，它也为您考虑了。如果您需要进一步研究 bundle，请参考*资源编程指南*。

本书约定您应该拥有任何通过 `alloc` 方法创建的对象（请参考内存管理规则了解其他约定)。因此，您还需要：

- 放弃对所创建的对象的拥有权。
- 通常只在初始化函数中调用存取方法来设置实例变量。

上述代码第二行使用存取方法来设置实例变量，第三行调用 `release` 方法以放弃对所创建对象的拥有权。

您也可以使用其他方式来完成这些功能。例如，可以把这三行代码替换成下面两行：

```
MyViewController *aViewController = [[[MyViewController alloc]
                initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]]
autorelease];
[self setMyViewController:aViewController];
```

该版本使用 `autorelease` 来放弃对新建视图控制器的所有权。不过此种方式中，放弃所有权的动作将在未来的某一时刻执行。如果不理解此代码的含义，则请阅读 *Cocoa 内存管理变成指南*中 Autorelease Pools 一章。通常情况下，请尽可能地避免使用 `autorelease` 方法，因为相对于 `release` 方法来说，它是一种资源密集型操作。

您也可以将最后一行替换如下：

```
self.myViewController = aViewController;
```

此处点号就是调用存取方法（即 `setMyViewController:`），这与前述实现调用的方法并无不同。点号确实提供一种更为紧凑的语法—特别是在使用嵌套表达式的时候。将几个属性合在一起使用时，点号语法能带来一些附加好处，但到底选择哪种语法，则很大程度上取决于个人的偏好-请参考 *Objective-C 编程*

*语言*的已声明的属性。如需进一步了解点号语法，请参看 *Objective-C 编程语言*中的对象，类和消息里面的"点号语法"。

建立视图

视图控制器负责管理和配置视图。您并不直接创建窗口的内容视图，而是从视图控制器获取，并将其添加成窗口子视图。

>> 释放视图控制器后，请添加如下的代码：

```
UIView *controllersView = [myViewController view];
[window addSubview:controllersView];
```

您也可以使用一行代码来完成上面代码的功能：

```
[window addSubview:[myViewController view]];
```

但是将代码分为两行有助于强调内存管理的一个规则，它和我们之前看到的相反。由于您并未使用 *Cocoa 内存管理编程指南*中的内存管理规则里所列出的方法来创建控制器视图，所以您并不拥有该视图。因此，把返回的对象传给窗口后，您无需再对其作后续处理（即不用释放这个对象）。
最后一行来自于 IDE 提供的模板：

```
[window makeKeyAndVisible];
```

这行代码会让窗口－现已含有您的视图－显示在屏幕上。之所以在窗口显示之前把视图添加进去，是为了防止用户在实际内容显示前看到短暂的白屏。

内务处理

您还剩几个任务：导入视图控制器头文件，合成存取方法，在 `dealloc` 方法中释放视图控制器（遵循内存管理规则里面的规定）。

>> 请在应用程序委托类的实现文件(即 `HelloWorldAppDelegate.m`)中执行下述操作：

- 请在文件的顶部导入 `MyViewController` 的头文件：

  ```
  #import "MyViewController.h"
  ```

- 请在类的 `@implementation` 代码块中通知编译器为视图控制器合成存取方法：

  ```
  @synthesize myViewController;
  ```

- 请在 `dealloc` 方法起始处释放视图控制器：

```
    [myViewController release];
```

实现源码列表

为确保正确，请确定您的 `HelloWorldAppDelegate` 类的实现(即 `HelloWorldAppDelegate.m` 文件)如下所示：

```
#import "MyViewController.h"
#import "HelloWorldAppDelegate.h"

@implementation HelloWorldAppDelegate

@synthesize window;
@synthesize myViewController;


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

    MyViewController *aViewController = [[MyViewController alloc]
                        initWithNibName:@"MyViewController" bundle:[NSBundle mainBundle]];
    [self setMyViewController:aViewController];
    [aViewController release];

    UIView *controllersView = [myViewController view];
    [window addSubview:controllersView];
    [window makeKeyAndVisible];
}


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

@end
```

测试应用程序

现在您可以测试应用程序。

>> 编译并运行工程(请选择 Build > Build and Run，或者点击 Xcode 工具栏的 Build and Run 按键)。

应用程序应能通过编译，不会报告错误。然后您将在模拟器中再一次看到白色的屏幕。

小结

在本部分，您添加了一个视图控制器类及其伴生 nib 文件。在应用程序委托中，您为控制器实例声明了一个实例变量和存取方法。同时，您还合成了存取方法并且执行了其他几个内务处理任务。最重要的事情是您创建了一个视图控制器实例并将其视图传给窗口。下一章，我们将使用 Interface Builder 来配置 nib 文件，控制器要使用该文件来加载视图。

查看 Nib 文件

Interface Builder 用于创建并配置 nib 文件。本章描述两个重要的概念：插座变量（outlet）以及文件拥有者代理对象。

Interface Builder

您需使用 Interface Builder 创建用户接口。Interface Builder 并不生成源码，而是让您直接操作对象，并将这些对象保存在一份被称为 nib 文件的档案。

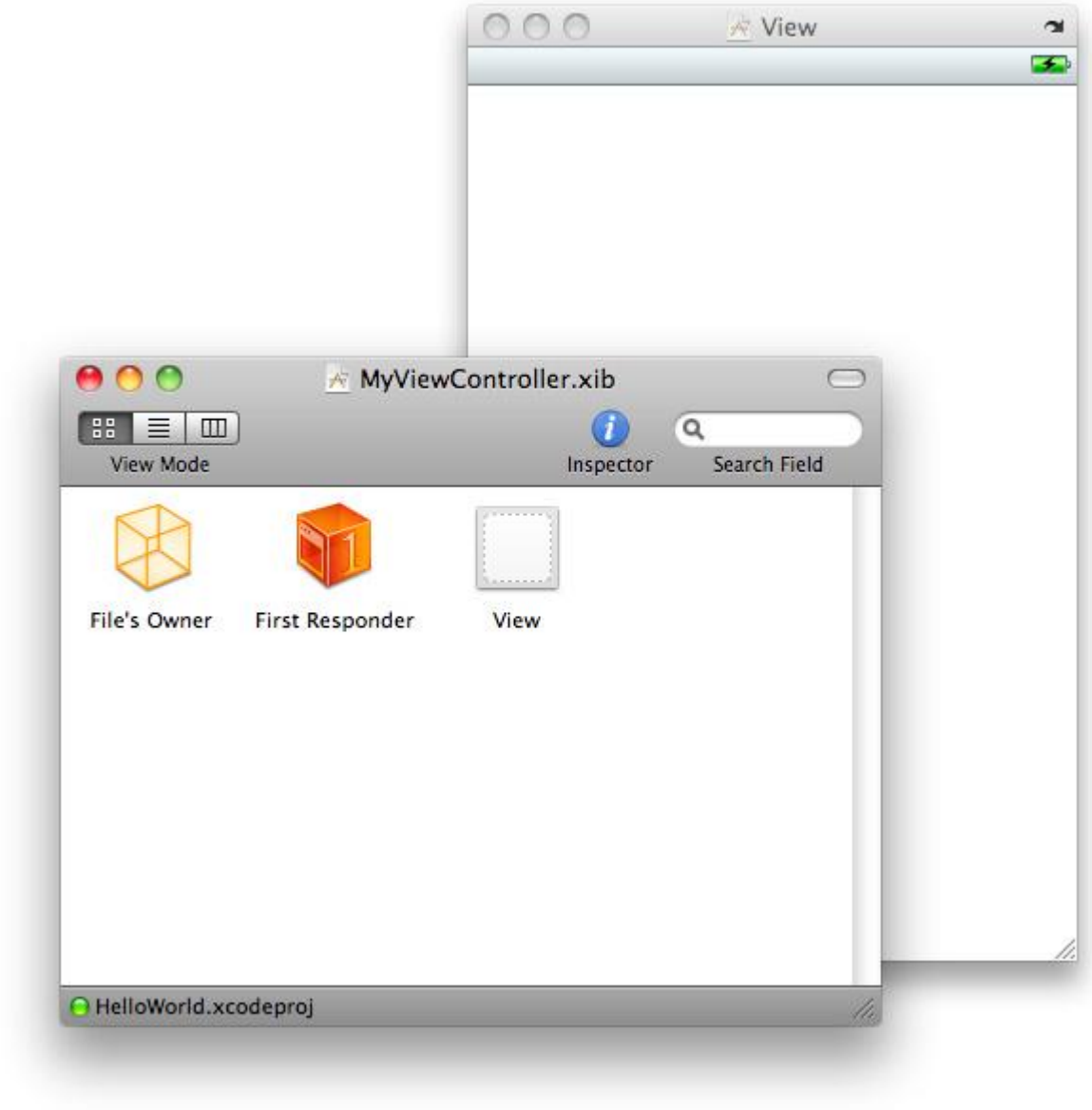
术语：虽然 Interface Builder 文档的扩展名可能是“.xib” ，但历史上，其扩展名是“.nib”（“NextStep Interface Builder”的首字母缩写），因此人们就俗称其为“Nib 文件”。

程序运行时会加载 nib 文件，解档文件中的对象并且将其恢复到被保存至文件那一瞬间的状态-包括对象间的所有关联。如需进一步了解 Interface Builder，请阅读 *Interface Builder 用户指南*。

查看 Nib 文件

>> 请双击 Xcode 中视图控制器的 XIB 文件(即 `MyViewController.xib` 文件)，Interface Builder 会为您打开该文件。

文件包含三个对象，文件拥有者代理，第一响应者代理以及一个视图。视图和 XIB 文件窗口分开显示，以便于您对视图进行编辑。

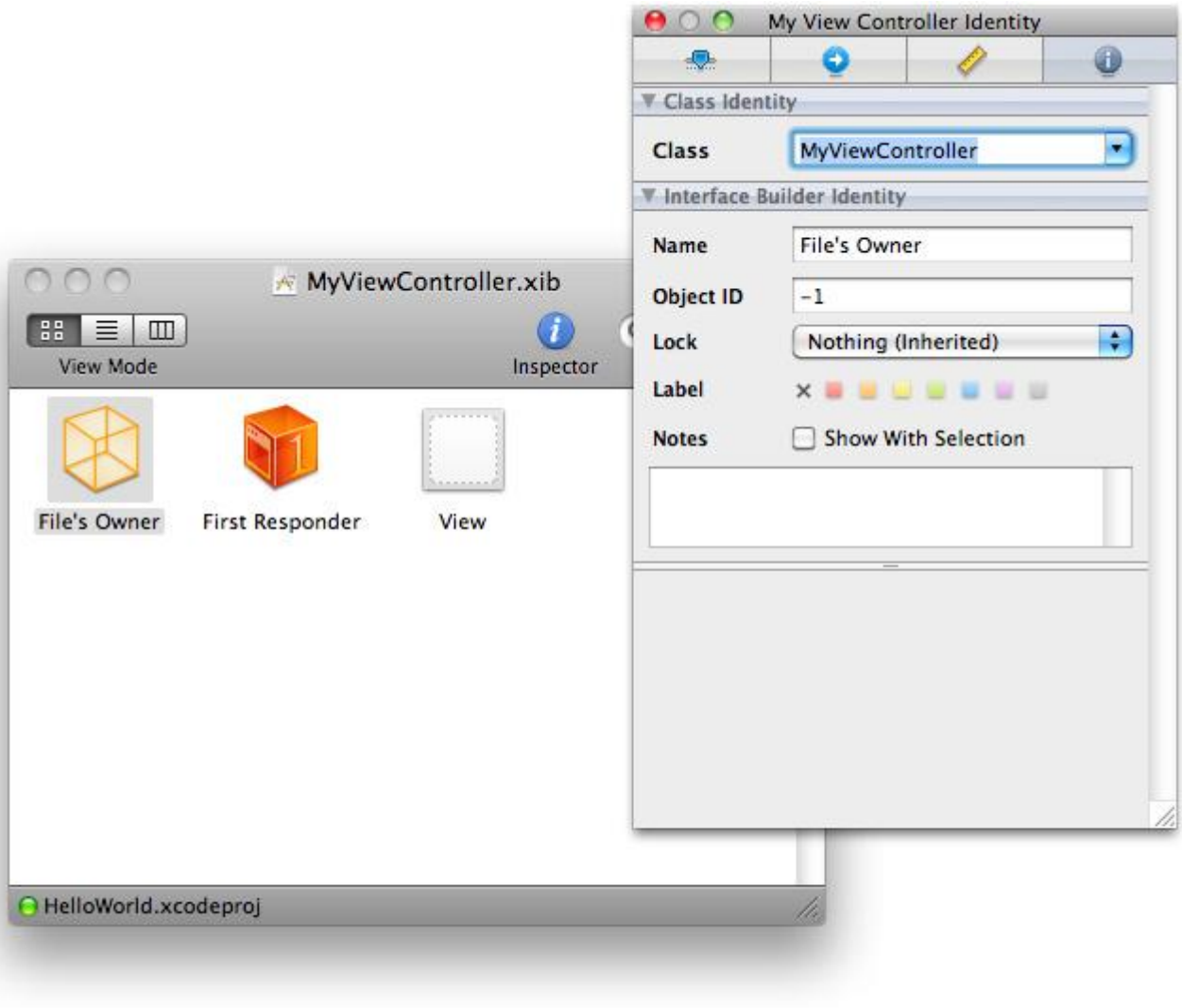


文件拥有者

Interface Builder 文档中的文件拥有者对象和您添加进去的其他对象不同。它不是在加载 nib 文件的时候创建，并且它会被设置为用户接口的拥有者—通常情况下，用户接口拥有者负责加载接口。如果您需更加详尽的资料，请参看*资源编程指南*。在您的应用程序中，文件的拥有者是 `MyViewController` 的实例。

Interface Builder 需要知道文件拥有者是什么类型的对象，这样它才能让您在文件拥有者和其他对象之间建立恰当的关联。您可以利用 Identity Inspector 来告诉 Interface Builder 该对象所属的类。实际上，当 nib 文件伴随着视图控制器类一同被创建出来时，nib 文件中的文件拥有者的类型就已被设置完成。不过现在，了解一下查看器对您有很大好处。

>> 请在 Interface Builder 文档的窗口中选择文件拥有者的图标，然后选择 Tools > Identity Inspector ，这样 Identity inspect 就会显示出来，如下图所示：

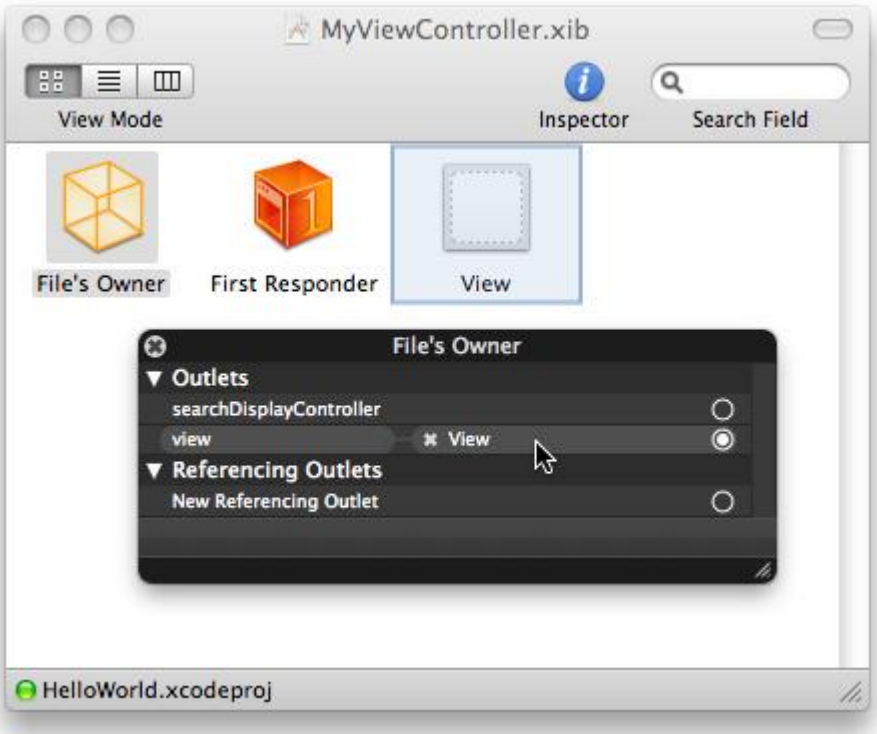


Class Identity 中的 Class 字段的值应该是 `MyViewController`。该值只是向 Interface Builder 承诺文件拥有者是该类的实例，把该字段设置成某个类并不能保证文件拥有者就是所设的类的实例。文件拥有者的类型取决于加载 nib 文件的时您所设置的对象。如果它是其他类的实例，则 nib 文件中建立的关联就无法正确建立。

视图插座变量

您可以使用查看器面板来查看-建立或打断-一个对象的关联。

>> 请在 Interface Builder 文档窗口中按住 Control 键并点击文件拥有者，这样就可以在屏幕中显示一个半透明的面板，面板里显示了文件拥有者的关联。



目前，文件拥有者只关联视图控制器的 `view` 插座变量。一个**插座变量**就是一个属性（通常是一个实例变量），只不过这个属性和 nib 文件中的某个项关联在一起。此处的关联表明当 nib 文件被加载并且 `UIView` 的实例解档之后，视图控制器的 `view` 实例变量会被设定指向 nib 文件中的视图。

加载 Nib 文件

视图控制器在 `loadView` 方法中会自动加载 nib 文件。加载哪个文件呢？ 请回想一下，您在 `initWithNibName:bundle:`方法的第一个参数已指定所要加载的 nib 文件的名称了。(请参看"创建一个试图控制器实例"一节)。通常情况下，在视图控制器整个生存过程中，`loadView` 方法只调用一次，目的是为了创建视图。当您调用视图控制器的 `view` 方法时，如果视图尚未被创建出来，则控制器会自动调用自己的 `loadView` 方法。(如果视图控制器由于接收到内存警告而清除了自己的视图，则在必要的时候，`loadView` 方法会被再次调用以创建视图）。
如果希望编程创建视图控制器的视图，则可以重载 `loadView` 方法，并在您自己的实现中创建视图。

如果您使用 `initWithNibName:bundle:`方法初始化一个视图控制器，但是希望在视图加载之后执行附加配置，则应该重载控制器的 `viewDidLoad` 方法。

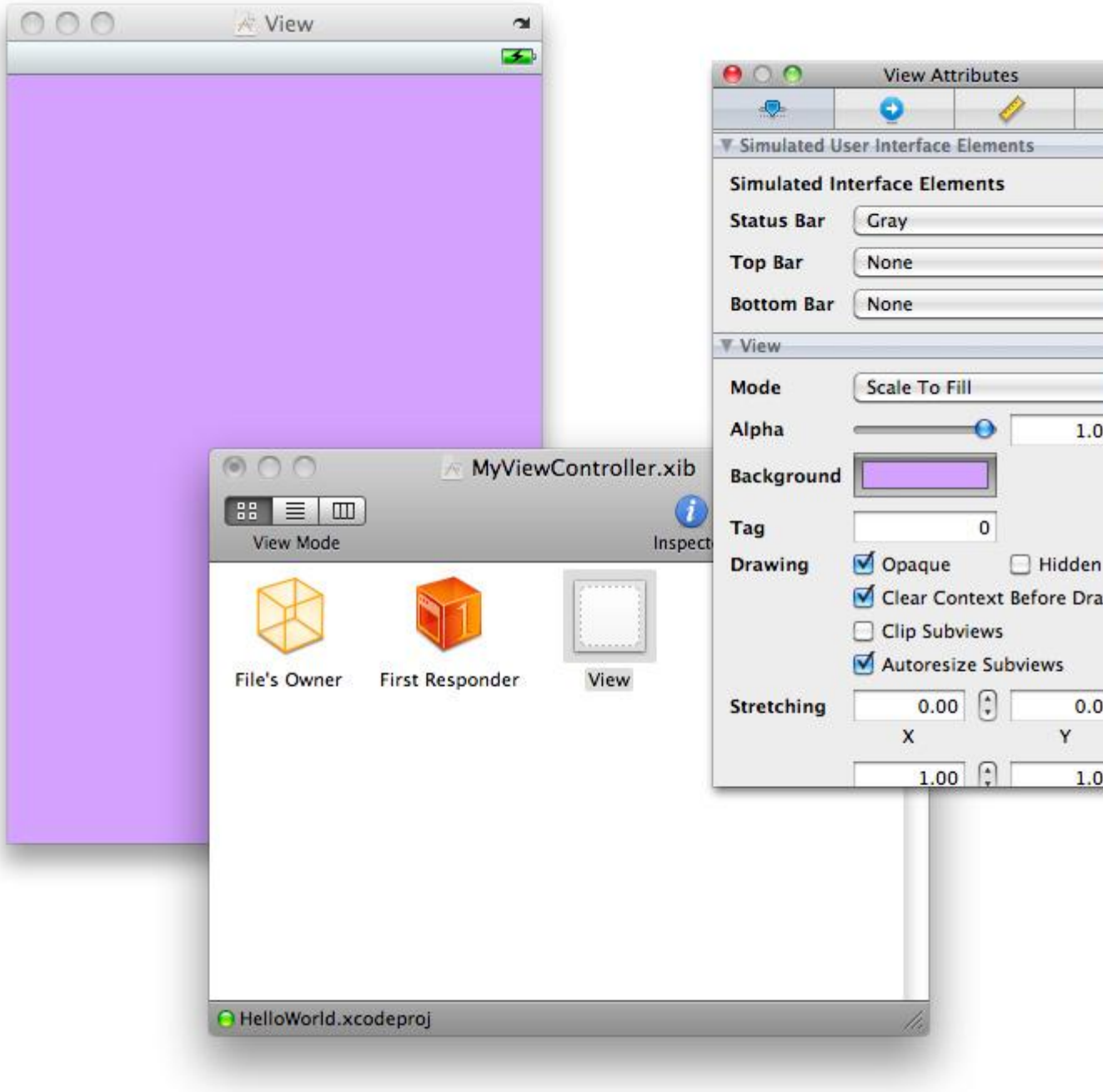
您可以使用一个 `NSBundle` 实例自行加载 nib 文件。如需进一步了解 nib 文件的加载，请参考*资源编程指南*。

测试应用程序

为了确信应用程序可以正确执行，您可以把视图的背景色设置成其他颜色（非白色），并在程序运行后验证新颜色是否显示。

>> 请在 Interface Builder 中选择视图，然后选择 Tools > Attributes Inspector ，这样屏幕就会显示 Attributes inspector。

>> 请点击 Background 选色板上的方框，让颜色面板显示在屏幕上。然后在其中选择一种不同的颜色。



>> 保存 nib 文件

>> 编译并运行工程(请点击工具栏中的 Build and Go 按键)。

您的应用程序应该可以正确编译，而后应该可以再次在模拟器里看到具有恰当颜色的屏幕。

>> 请把视图的背景颜色恢复成白色并保存 nib 文件。

小结

在本部分，您查看了 nib 文件，学习了插座变量以及如何设置视图的背景颜色。您还进一步了解如何加载资源以及视图控制器如何加载 nib 文件。

下一章，您将在视图中添加控件。

配置视图

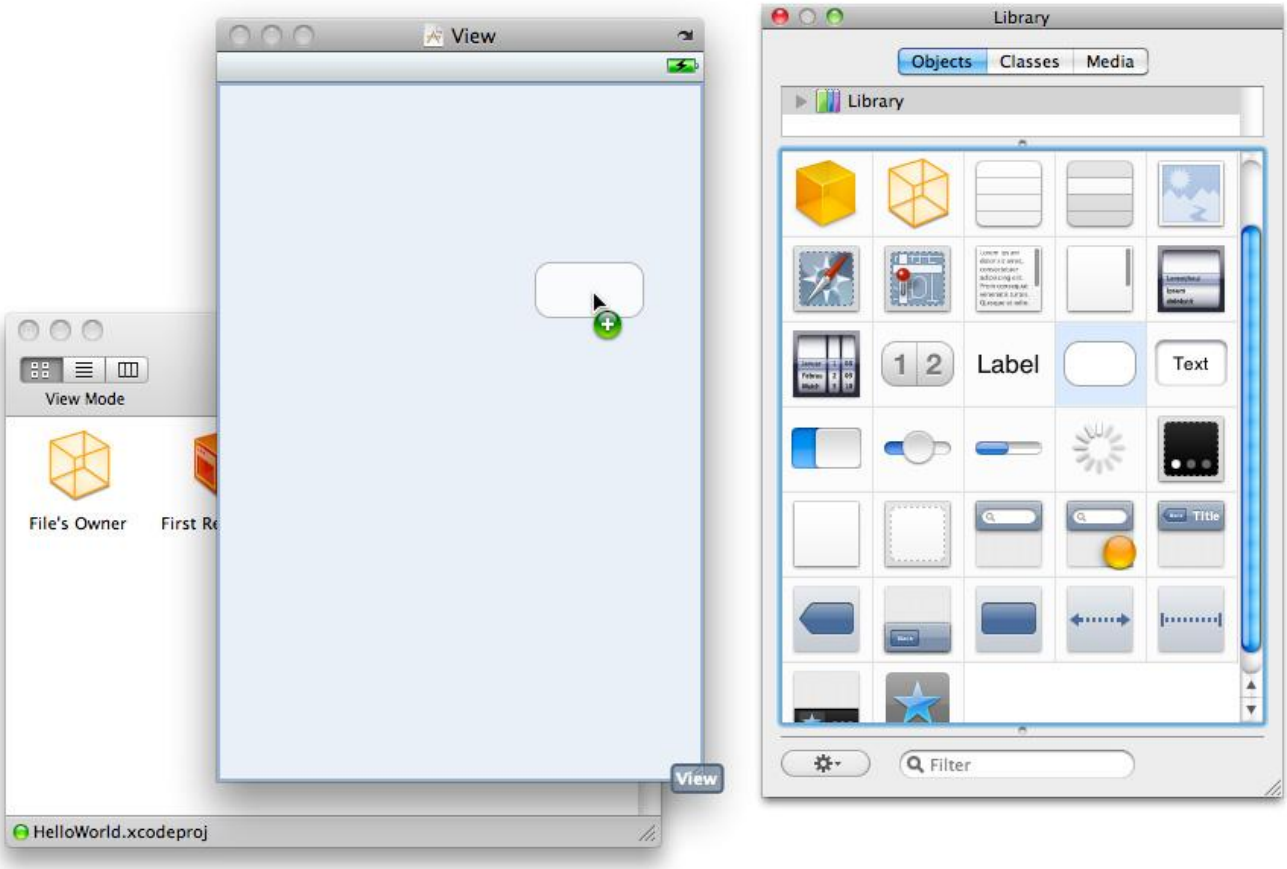
Interface Builder 包含一个对象库，您可以将其中的对象添加至 nib 文件。在这个对象库里，一部分是用户接口元素，例如按键和文本字段； 其它则是控制器对象，例如视图控制器。您的 nib 文件已含有一个视图-现在，您只需要添加按键和文本字段。

添加用户接口元素

只要将用户接口元素从 Interface Builder 库拖过来，就可以将其添加至视图。

>> 请在 Interface Builder 中选择 Tools > Library，让对象库的窗口显示在屏幕上

您可以从对象库中拖动视图项，然后将其放在视图上面，这和您在一个绘画应用里执行的动作相似。

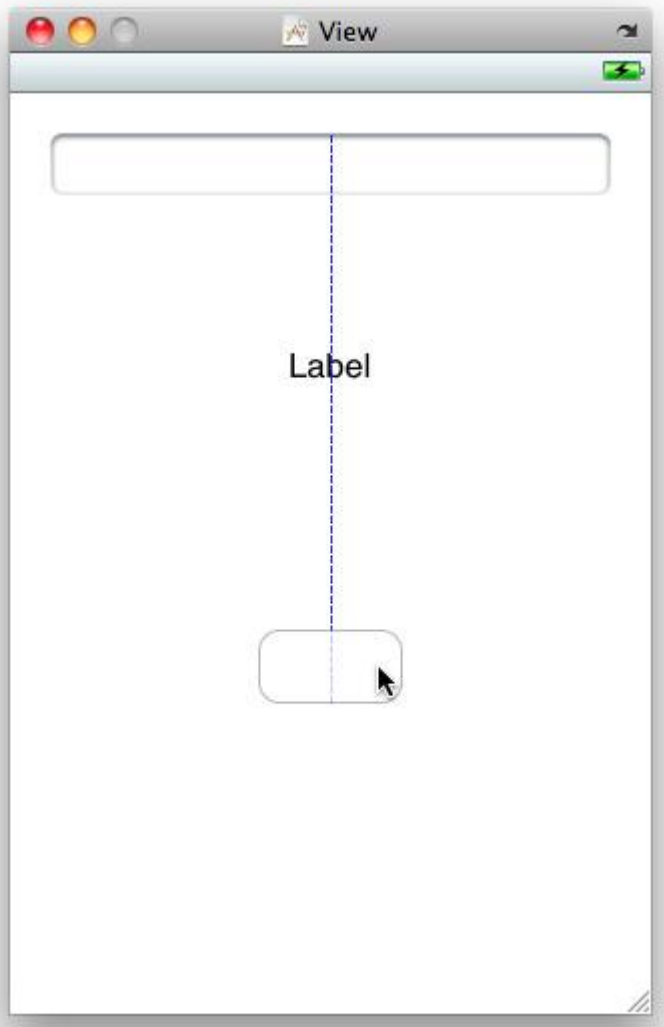


>> 请在视图上添加一个文本字段(`UITextField`)，一个标签 (`UILabel`)以及一个按键(`UIButton`)。

接下来，您可以使用视图项适当位置的尺寸调整点来调整尺寸，可以拖动这些视图项来进行重定位。当在视图内移动视图项时，视图上会出现蓝色的点划线，这是对齐引导线。

>> 请根据下图排布视图中的控件:

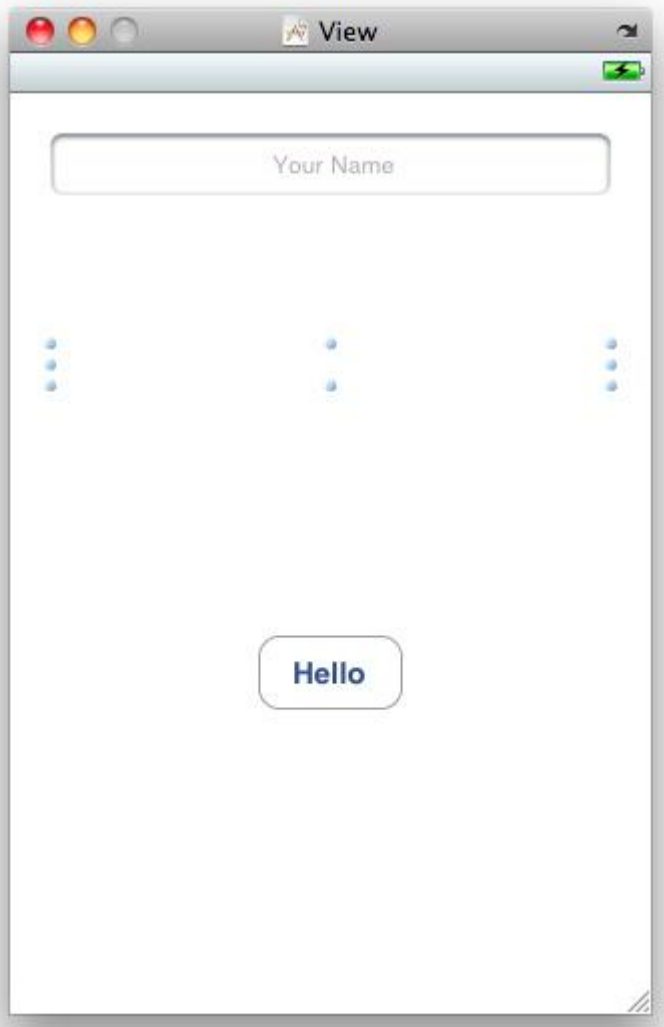
图 5-1 View 中包含数个用户接口元素以及一条蓝色的引导线。



>> 接下来，请执行下述操作：

1. 请在文本字段的属性查看器里输入 `Your Name`，这将作为文本字段的占位字符串。
2. 调整标签的尺寸，将其扩展至与视图等宽。
3. 删除标签中的文本(“Label”)，您可以在标签的属性查看器里删除，也可以直接选中文本（双击可执行选中操作）再按删除键进行删除。
4. 给按键添加一个标题，请在按键中双击鼠标，然后输入 `Hello`。
5. 请使用查看器将文本字段和标签的文本对齐方式都设置为居中对齐。

最后您的视图应该如下图所示：



>> 请勾选标签属性查看器视图区域中的 Clear Context Before Drawing 选项。这是为确保在更新祝贺词的时候，标签会先删除之前的字符串再绘制新字符串。如果不这样做，字符串就会相互重叠。

我们还需要对文本字段执行几处更改-第一处改动显而易见，其它改动向则不然。首先，您可能希望名称首字符大写。第二，您可能希望合理地配置文本字段的关联键盘，以便用户输入名称。同时，键盘上还需要显示一个 Done 按键。

我们执行这些变动的指导原则是：当文本字段被放到视图时，您知道它所应包含的内容。因此，您应该合理地设计文本字段，使得运行时键盘可以恰当地配置自己，从而更好地适应用户的任务。您可以在文本字段的文本输入特征中完成这些设置。

>> 请选择 Interface Builder 中的文本字段，然后选择它的属性查看器。请在属性查看器的文本输入特征区域中执行下述选择：

- 在 Capitalize 弹出菜单中， 请选择 Words
- 在 Keyboard Type 弹出菜单中，请选择 Default
- 在 Keyboard Return Key 弹出菜单中，请选择 Done

>> 请保存文件。

如果在 Xcode 上编译并运行应用程序，当程序运行起来时，您会看到用户接口元素根据您指定的位置摆放。按下视图中的按键，它会变成高亮，在文本字段里点一下，键盘会显示出来。但是目前，我们没有办法让键盘消失。您需要在视图控制器和其它对象之间建立恰当的关联，才能解决该问题，也才能添加其他功能。这些内容将在下部分介绍。

视图控制器接口声明

为了建立视图控制器到用户接口的关联，您需要指定一些插座变量（之前说过插座变量就是实例变量）。同时您还需要一个非常简单的模型对象的声明，一个字符串即可。

>> 请在 Xcode `MyViewController.h` 文件中的 `MyViewController` 类里面添加下面的实例变量：

```
UITextField *textField;
UILabel *label;
NSString *string;
```

>> 然后您需要为这些实例变量添加 属性声明， 同时还需添加一个 `changeGreeting:`动作方法的声明：

```
@property (nonatomic, retain) IBOutlet UITextField *textField;
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, copy) NSString *string;
- (IBAction)changeGreeting:(id)sender;
```

`IBOutlet` 是一个特殊的关键字，它唯一的作用是通知 Interface Builder 将某个实例变量或者属性当成插座变量。实际上，这个关键字被定义为空白，因此在编译的时候它没有任何作用。

`IBAction` 是一个特殊的关键字，它唯一的作用是告诉 Interface Builder 将某个方法当成目标/动作关联中的动作。它被定义为 `void`。

视图控制器还将作为文本字段的委托，因此它必须采用 `UITextFieldDelegate` 协议。如希望指定一个类采用某种协议，请在接口继承的类的名称后面添加协议名称，并将其放在尖括号(<>) 中。

>> 请在 `UIViewController` 后面添加`<UITextFieldDelegate>`，这是为了让 `UIViewController` 对象采用 `UITextFieldDelegate` 协议。

您的接口文件应如下所示：

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

@interface MyViewController : UIViewController <UITextFieldDelegate> {
    UITextField *textField;
    UILabel *label;
    NSString *string;
}
@property (nonatomic, retain) IBOutlet UITextField *textField;
```

```
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, copy) NSString *string;
- (IBAction)changeGreeting:(id)sender;
@end
```

>> 保存 `MyViewController.h` 文件，以便让 Interface Builder 注意到上述改动。

本节末，您就可以对项目进行测试。请在实现文件（即 `MyViewController.m`）中实现一个空的 `changeGreeting:`方法。

>> 请在`@implementation MyViewController` 一行后面添加：

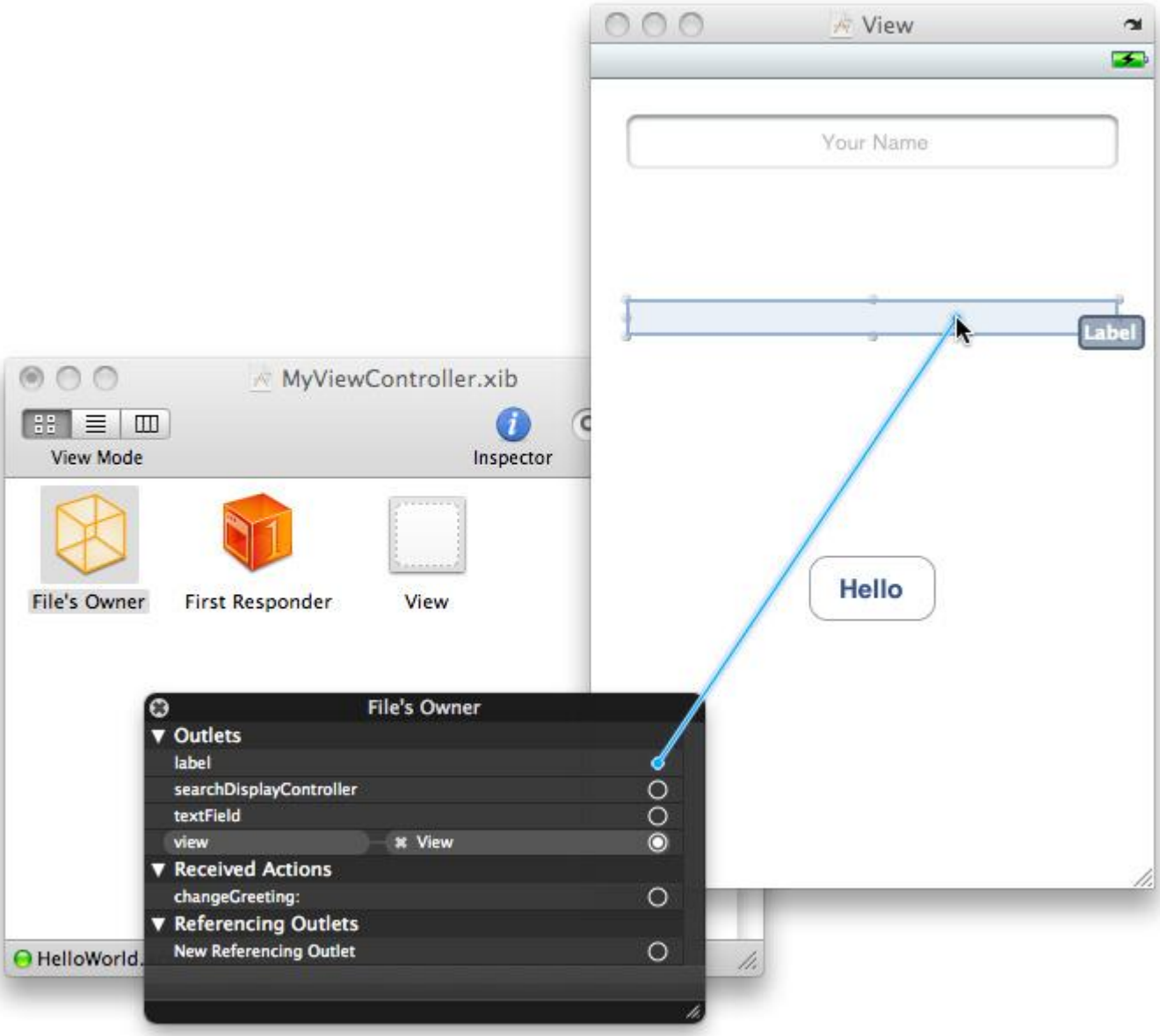
```
-(IBAction)changeGreeting:(id)sender {


}
```

>> 保存文件。

制定关联

您已经定义了视图控制器的插座变量和动作，现在开始在 nib 文件中建立关联。

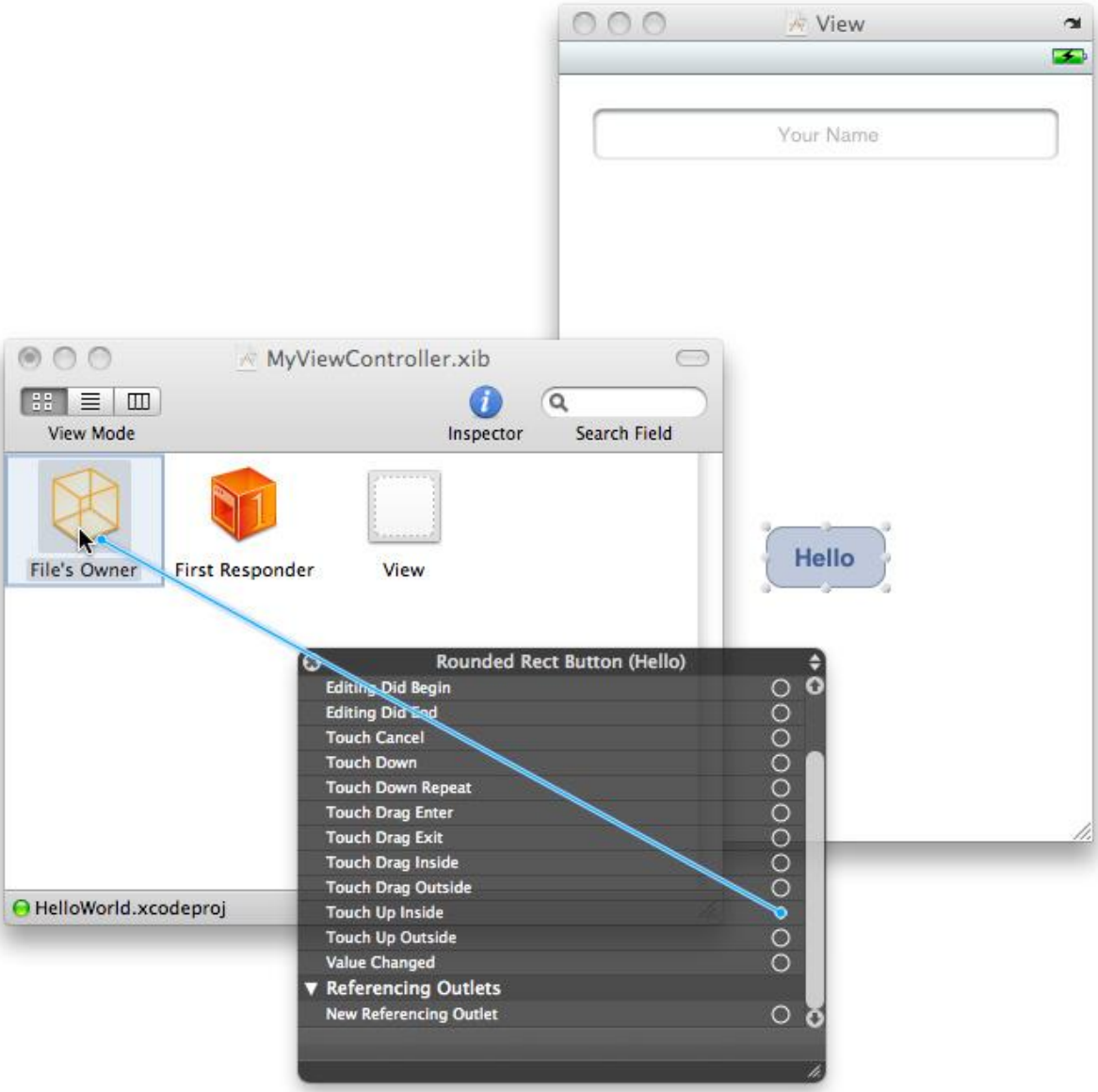
>> 关联标签和文本字段的插座变更。按下 Contrl 键并点击文件拥有者，一个半透明的面板就会显示出来，面板里面显示了所有可用的插座变量和动作。请将光标从列表右边的圆圈拖动到所要关联的视图项上面，这样就可以建立二者之间的关联。

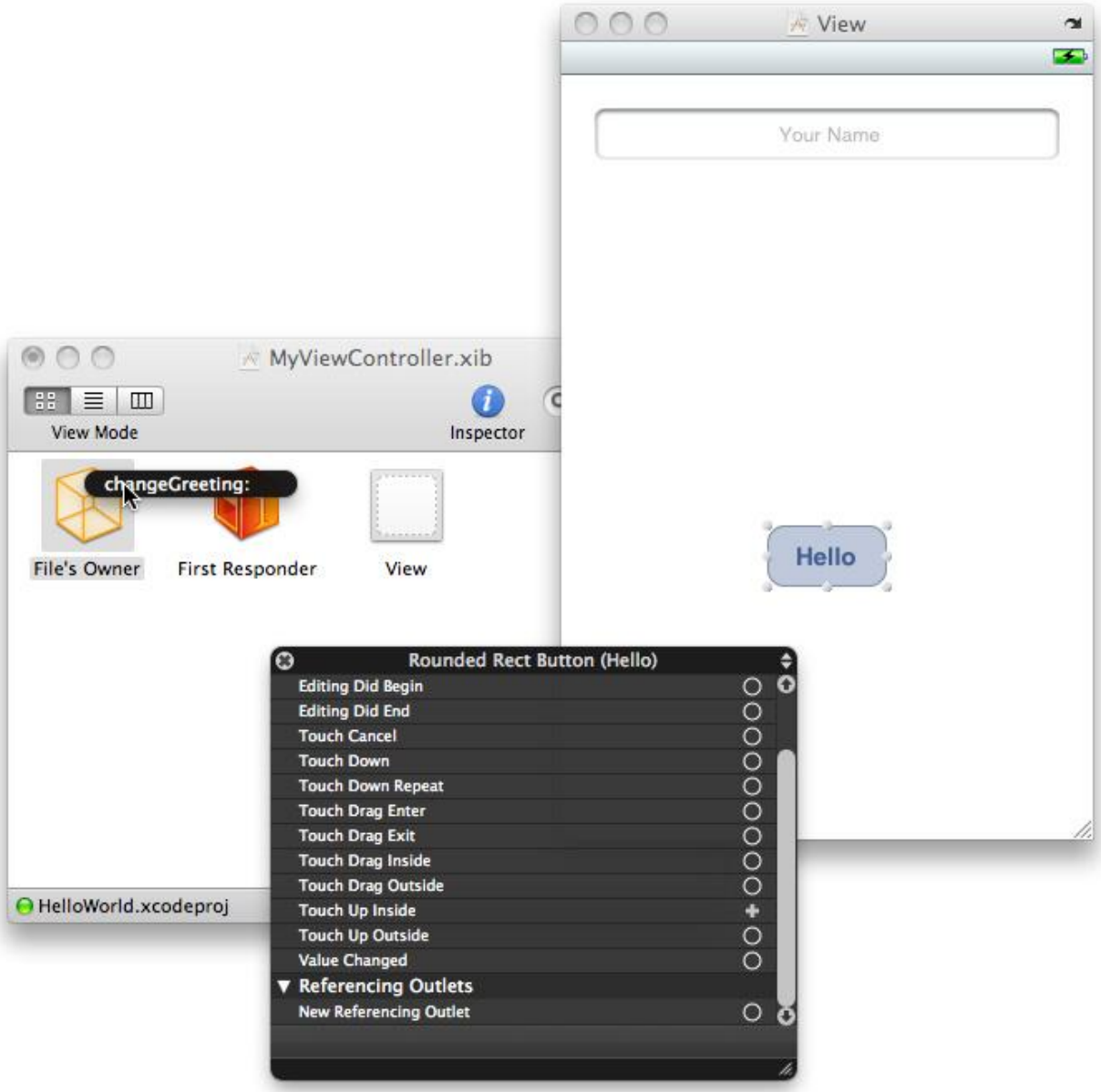


您可以通过面板右下角的尺寸调整角来调整面板尺寸，让它一次所能显示的插座变量和动作减少一些。如果面板没有足够的空间来显示所有的插座变量和动作，则其上会出现滚动条，您可以使用它在列表视图里导航。

Interface Builder 不允许您将插座变量和一个错误类型的元素关联在一起。举个例子，因为标签属性被声明为 `UILabel` 的实例，因此它不能和文本字段建立关联。

>> 设定按键的动作方法。请按下 Control 键并点击视图中的按键，在随后出现的查看器中，请将光标从事件列表 Touch Up 项的开口圆圈拖动到文件拥有者的图标。待文件拥有者上面出现一个半透明面板后，请选择其中的 `changeGreeting:`方法（您可能需要在查看器中滚动才能看到 Touch Up 事件）。


View
Your Name
MyViewController.xib
View Mode
Inspector
Search Field
File's Owner
First Responder
View
Hello
HelloWorld.xcodeproj
Rounded Rect Button (Hello)
Editing Did Begin
Editing Did End
Touch Cancel
Touch Down
Touch Down Repeat
Touch Drag Enter
Touch Drag Exit
Touch Drag Inside
Touch Drag Outside
Touch Up Inside
Touch Up Outside
Value Changed
Referencing Outlets
New Referencing Outlet



上述关联的意思是：当运行程序时，如果您的手指从按键离开，按键就会发送一条 `changeGreeting:` 消息给文件拥有者对象。（如果需要所有控件事件的定义， 请参看 `UIControl`。）

当用户点击键盘中的 Return 按键，文本字段会向委托发送一条消息。您可以利用该回调函数来消除键盘。（请参看“文本字段委托”）。

>> Se 将文本字段的委托设置成文件拥有者（即视图控制器）。请按下 Control 按键，并将光标从文本字段向文件拥有者拖动，在弹出来的半透明面板中选择 `delegate` 。

测试

现在可以对应用程序进行测试。

>> 链编并运行项目。

在编译时会出现几个编译器警告，那是因为您没有实现属性的存取函数 — 下一章我们将会修正这些警告。尽管如此，按钮还是可以正常工作（被点击时高亮）。同时，当您触摸文本字段的时候，键盘会显示出来，您可以利用它来输入文本。但是，我们没有办法去除键盘。要去除键盘，您需要实现相关的委托方法。这将在下一章介绍。.

小结

您在视图控制器类的接口中添加了实例变量和属性声明，以及一个动作方法的声明。您还在类实现中添加了动作方法的存根实现，并且对 nib 文件进行配置。

实现视图控制器

实现视图控制器要做三件事。您需要对实例变量作一些处理，包括内存管理方面的处理；需要实现 `changeGreeting:`方法；需要确保用户点击 Done 按键的时候，键盘会消失。

视图控制器中的属性

首先要让编译器合成存取方法。

>> 请在 `MyViewController.m` 文件的`@implementation MyViewController` 后面添加下列代码：

```
@synthesize textField;
@synthesize label;
@synthesize string;
```

添加这些代码之后，根据接口文件中给定的属性规格，编译器会为其合成相应的存取方法。举个例子，接口文件中 `string` 属性的声明是`@property (nonatomic, copy) NSString *string;`)，因此编译器会为其生成两个存取方法，即`- (NSString *)string` 方法和
`- (void)setString:(NSString *)newString` 方法。`setString:` 方法会生成一个传入字符串的副本，这有助于确保**封装**（传入的字符串有可能会发生改变，因此您希望让控制器保留一份自己的副本）。如需进一步了解封装，请参考 *Objective-C 面向对象编程指南*中的“抽象的机制”。
所有属性的声明都指定自身属于视图控制器，因此您必须在 `dealloc` 方法中放弃对这些属性的所有权（`retain` 以及 `copy` 都隐含控制器对属性的拥有权，请参考 *Cocoa 内存管理编程指南*的内存管理规则）。
>> 请更新 `MyViewController.m` 文件中的 `dealloc` 方法，让它在调用超类的实现之前先释放实例变量：

```
- (void)dealloc {
    [textField release];
    [label release];
    [string release];
```

```
    [super dealloc];
}
```

changeGreeting: 方法

点击视图中的按键，它会向视图控制器发送 `changeGreeting:` 消息。之后，视图控制器会取得文本字段的内容，并据此更新标签的内容。

>> 请按如下方式实现 `MyViewController.m` 文件中 `changeGreeting:` 方法：

```
- (IBAction)changeGreeting:(id)sender {

    self.string = textField.text;

    NSString *nameString = string;
    if ([nameString length] == 0) {
        nameString = @"World";
    }
    NSString *greeting = [[NSString alloc] initWithFormat:@"Hello, %@!", nameString];
    label.text = greeting;
    [greeting release];
}
```

对这个方法我们有下面几点说明。

- `self.string = textField.text;`

  这行代码先从文本字段中获取文本，然后将结果设置给控制器的 `string`。

  在本例中，实际上，您不会在其他地方使用 string，但是理解该变量的角色具有重要意义。它是视图控制器所管理的一个很简单的模型对象。通常情况下，应用程序应该在其自有的模型对象中保存应用程序数据信息—应用程序数据不应该*存放*在用户接口元素中。

- `@"World"`是一个字符串常量，它通过一个 `NSString` 的实例来表示。

- 和 `printf` 函数相似，`initWithFormat:`方法符串按照格式化字符串所指定的格式创建一个新字符串。`%@`表明此处应该使用一个字符串对象来代替。如需进一步了解字符串，请参考 *Cocoa 字符串编程指南*一文。

文本字段的委托

链编并运行应用程序。点击按键后，标签显示“Hello, World!”。但是选择文本字段进行输入，您会发现您没有办法表示已完成输入，也没有办法消除键盘。

在 iPhone 应用程序中，当一个允许文本输入的元素变成第一响应者时，键盘就会自动显示出来，而当该元素不再处于第一响应者状态，键盘就会消失。（如果进一步了解第一响应者，请阅读 in *iPhone 应用程序编程指南*中的事件处理）。您不能直接向键盘发送消息，但是可以切换文本输入元素的第一响应者状态，利用该操作的附加效果来显示或消除键盘。
在应用程序中，当用户点击文本字段时，该控件就会变成第一响应者，因此键盘就会显示出来。而当用户点击键盘中的 Done 按键时，您希望键盘消失。

`UITextFieldDelegate` 协议包含一个 `textFieldShouldReturn:`方法，一旦用户点击 Return 按键，文本字段就会调用该方法（和按键的标题无关）。由于您将视图控制器设置成文本字段的委托(参考“制定关联”一节)，因此您可以实现该方法，在方法中向文本字段发送 `resignFirstResponder` 消息—这个消息的附加效果会让键盘消失。
>> 请按照下述代码实现 `MyViewController.m` 文件中的 `textFieldShouldReturn:`方法：

```
- (BOOL)textFieldShouldReturn:(UITextField *)theTextField {
    if (theTextField == textField) {
        [textField resignFirstResponder];
    }
    return YES;
}
```

这个应用程序并非真的需要包含 `theTextField == textField` 检查，因为它只包含一个文本字段。不过这种模式值得注意。因为有些时候，您的对象可能会被设置成多个相同类型的对象的委托，并且您可能还需要区分这些对象。

>> 链编并运行应用程序；它应该具有您所预期的行为。（当您输入完名称之后，点击 Done 按键，键盘会消失，然后再点击一下标题为 Hello 的按键，标签就会显示“Hello, <Your Name>!”。）

如果应用程序不具有您所预期的行为，则您需要排除问题（请参考 span class="content_text">“排除问题”一章）

小结

您已完成视图控制器的实现，第一个 iPhone 应用程序到此也就完成了。祝贺您。

请花点时间思考一下，视图控制器如何同整个应用程序架构相适应。您以后写得大部分 iPhone 应用程序可能都会用到视图控制器。

休息一下，然后开始考虑接下来您应该做什么。

排除疑难

本部分描述一些您会可能碰到的常见问题的解决方法。

代码和编译器警告

如果程序不能正确工作，请首先把您的代码和“代码列表”里完整的代码进行比较。
您的代码应该可以通过编译，没有任何警告。Objective-C 是一种非常灵活的编程语言，有些时候您从编译器得到的大都是警告。通常情况下，您应该把这些警告当成错误对待。

检查 Nib 文件中的关联

作为一个开发人员，当程序出现问题的时候，您本能地会去检查代码中的错误。但是在 Cocoa 中，还可能有另外一个地方出错。许多应用程序的配置都被“编码”在 nib 文件当中，如果没有建立正确的关联，应用程序也可能没有预期行为。

如果您点击按键而标签文本不更新，则有可能是您没有将按键的动作和视图控制器关联起来，或者由于您将视图控制器的插座变量关联到了文本字段或者标签。

如果点击 Done 而键盘不消失，则可能是没有关联文本字段委托（请参看“制定关联”）。如果您已关联了委托，则该现象有可能是一个更加难以察觉的错误引起的（请参考“委托方法名称”)。

委托方法的名称

和委托相关的一个常见的错误是写错委托方法的名称。即使您已经正确设置了委托对象，如果委托没有实现名称正确的方法，则委托方法就不会被调用。通常情况下，最好从文档中复制粘贴方法声明。

下一步做什么？

在学习 iPhone 开发的道路上，下一步应该做什么呢？本章为您提供一些建议。

用户接口

在学习本教程的过程中，您创建了一个很简单的 iPhone 应用程序。但对于 Cocoa Touch 开发环境提供的丰富的特性而言，您仅是触及皮毛。现在是进一步探索的时候了，就从这个应用程序开始吧。正如之前所说，用户界面是一个 iPhone 程序成功的关键。因此，请尝试改善程序的用户界面。您可以为控件添加图像和颜色，可以为应用程序添加背景图像和图标。请检查一下 Interface Builder 的查看器，看看还可以如何配置接口元素。

编程创建用户接口元素

本教程中，您使用 Interface Builder 创建用户接口。通过 Interface Builder，您可以快速轻松地对用户接口部件进行装配。但有时候，您可能希望- 或者需要 -在代码中创建用户接口。（举个例子，在创建一个定制的表视图单元时，您通常需要通过编程来创建排布子视图）。

第一步，请打开 `MyViewController` nib 文件，移除视图中的文本字段。

如希望在代码中创建整个的视图层，则需要重载 `loadView` 方法。而本例中，您希望在加载 nib 文件后执行附加配置（添加另外一个视图），因此需重载 `viewDidLoad` 方法。（不论您使用什么方法来加载主视图-使用 nib 文件或是重载 `loadView` 方法-您都可以重载 `viewDidLoad` 方法，它是一个很常见的重载点。）

请按照下述代码实现 `MyViewController.m` 文件中的 `viewDidLoad` 方法：

```
- (void)viewDidLoad {

    CGRect frame = CGRectMake(20.0, 68.0, 280.0, 31.0);
    UITextField *aTextField = [[UITextField alloc] initWithFrame:frame];
    self.textField = aTextField;
    [aTextField release];

    textField.textAlignment = UITextAlignmentCenter;
    textField.borderStyle = UITextBorderStyleRoundedRect;

    textField.autocapitalizationType = UITextAutocapitalizationTypeWords;
    textField.keyboardType = UIKeyboardTypeDefault;
    textField.returnKeyType = UIReturnKeyDone;
    textField.delegate = self;
    [self.view addSubview:textField];
}
```

请注意，和使用 Interface Builder 创建配置文本字段相比，编程的方法需要相当多的代码。

链编并运行应用程序。确保应用程序的行为和之前一样。

把程序安装到设备

如果您的计算机通过 30 针的 USB 线连接了合适的设备（iPhone 或者 iPod Touch），并且您拥有来自 iPhone 开发者计划 的合法证书，则请将工程的活动 SDK 设置为"iPhone OS" （不要设置为"iPhone OS

Simulator"），链编并运行工程。假如代码编译成功，Xcode 就会自动把应用程序上传到设备上。详情请参看 *iPhone 开发指南*。

额外的功能

接下来您还可以尝试扩展程序的功能。可以尝试的方向很多：

- 以前您把视图作为画布，在其上放置预制的用户接口控件，现在您可以尝试写一个定制的视图，让视图可以绘制自身的内容并响应触碰事件。请看一下诸如 MoveMe 或者 Metronome 这样的例子，以从中获取灵感。

- 本文档中，您使用 Interface Builder 布局应用程序用户接口，但是实际上，许多应用程序使用表视图来排布接口。利用表视图，您很容易就能创建一个尺寸大于屏幕边界的接口-它允许您通过滚动来显示额外的接口。请首先研究一下如何使用表视图来创建一个简单的列表。您可以参看几个样例代码工程-包括 TableViewSuite-，然后再实现自己的工程。

- 导航控制器和标签栏控制器提供的架构可用于创建 drill-down 风格的接口，并且它还允许用户选择应用程序中的不同视图。导航控制器经常和表视图结合，但这两种控制器都需要与视图控制器结合在一起使用。请参考一些使用导航控制器的应用程序样例-例如 SimpleDrillDown，然后对这些样例进行扩展以创建您自己的应用程序。

- 对应用程序进行本地化经常可以扩大其潜在的市场。国际化就是让应用程序本地化的过程。如需进一步了解国际化，请阅读*国际化编程*。

- 要想在 iPhone 上提供良好的用户体验，性能是关键。您应该学会使用 Mac OS X 提供的各种性能工具—特别是 Instruments—对应用程序进行优化，从而使其对资源的需求降到最低。

最重要的事情是尝试新的想法并动手实验-目前苹果提供很多的样例代码，您可以参考以获得灵感，文档也很多，它们可以帮助您理解一些概念和编程接口。