从零开始学iOS7开发系列教程-事务管理软件开发实战-Chapter9

欢迎继续我们的学习。

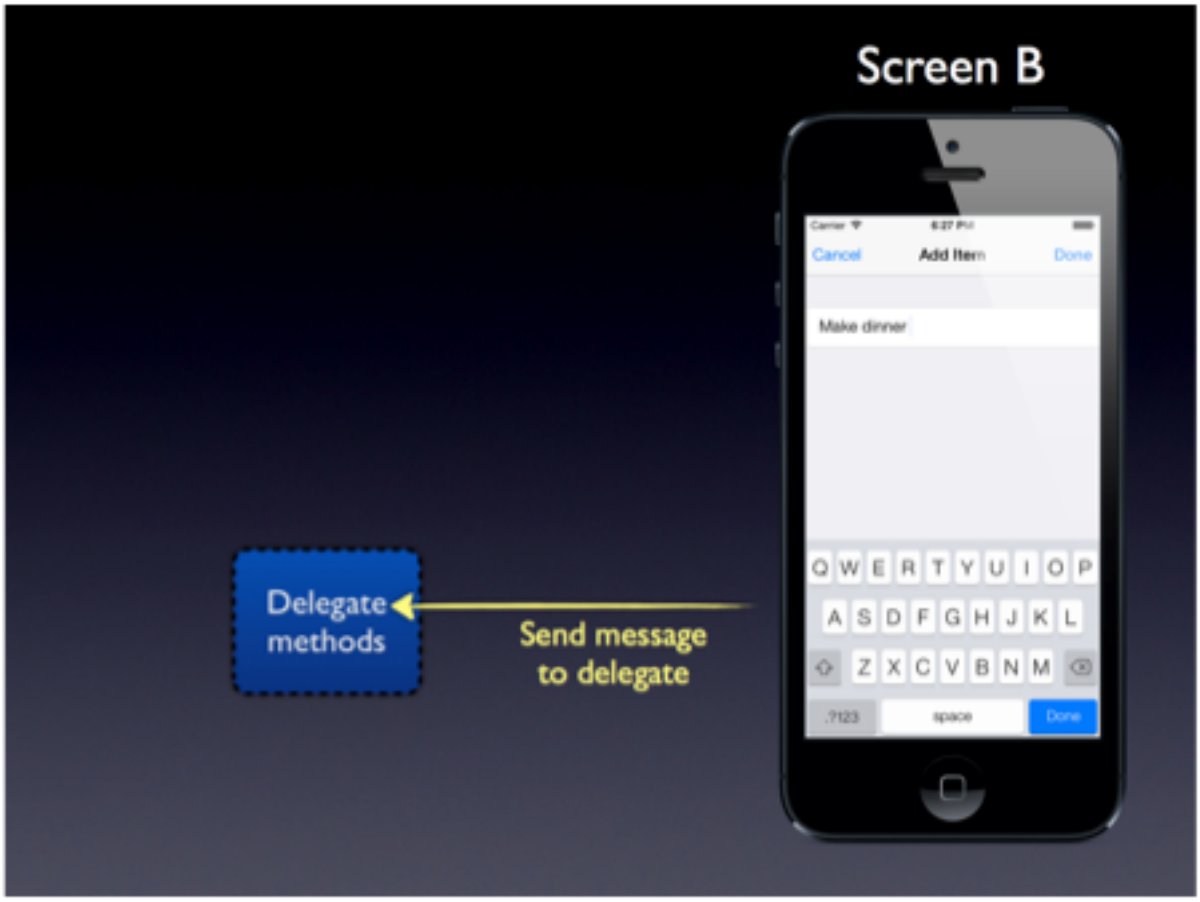
在上一章的学习中，我们已经给用户提供了交互的方式，让他们得以在文本框中输入自己的文本内容。同时程序还会自动检测所输入的内容，保证这个代办事务不会是空的。
在这一章的学习中，我们要开始学习如何把用户所输入的文本添加到一个新的ChecklistItem对象中，并添加到数据模型数组中。为了实现这一点，我们需要创建一个自己的代理。

Don't panic，到目前为止，我们其实已经多次接触过了代理：表视图有一个代理，可以对用户对行数据的触屏产生回应。text field文本域有一个代理，可以用来验证文本的长度。在上一个系列的教程中，我们还使用了一个代理对象来监听alert view提示对话框。别忘了我们这款应用本身也有一个名为ChecklistsAppDelegate的代理。在iOS的开发中，代理几乎无处不在。

代理模式通常用于解决类似下面的问题：
我们通过界面A打开了界面B，不过在应用进行的过程中，界面B有时候也想和界面A主动联络，比如当它关闭自身的时候。一个比较好的解决方法是，让A成为B的代理，这样B就可以在需要的时候给A发送消息了。

代理模式的一个好处是，界面B实际上不需要了解界面A的任何事情，它只需要知道A是自己的代理就可以了。比如UITableView并不在乎你的视图控制器会是怎样的，只要它能在表视图请求的时候提供正确的cell就好了。在这种模式下，界面B独立于界面A，但还是可以跟它说上话，术语党给了个可怕的名词叫做loose coupling（松耦合）。松耦合被认为是一种很好的面向对象设计实践。

还是打个不恰当的比方吧，爸妈生了你，他们知道关于你的一切，但大多数情况下你对他们了解的并不多。不过在学校里面的时候你偶尔会缺钱，这时就会想起父母这个财务代理了。不过拿到钱之后，你还是很快忘了他们，逍遥自在去了。
当然，这样是非常不孝顺的做法，做子女的也还是要多了解下自己的父母，不要只是要钱的时候才想起他们。



比如在上图中，界面B只知道界面A有这些代理方法，但是对界面A的其它一无所知。
换句话说，你只知道爸妈关键时刻能给钱你花，至于其它的你就根本不在乎了。哎，孽子啊！

在这里，我们将使用代理模式让AddItemViewController向ChecklistsViewController发送通知，但同时AddItemViewContyroller又无需知道ChecklistsViewController的其它事情。

在Xcode中打开AddItemViewController.h, 在#import和@interface之间输入以下代码：

```
@class AddItemViewController;
@class ChecklistItem;

@protocol AddItemViewControllerDelegate <NSObject>
```

```
-(void)addItemViewControllerDidCancel:(AddItemViewController*)controller;

-(void)addItemViewController:(AddItemViewController*)controller
        didFinishAddingItem:(ChecklistItem*)item;

@end
```

通过以上代码，我们定义了AddItemViewControllerDelegate协议。
需要记住的是，在@protocol和@end之间的语句块都属于方法声明。

术语解释：Protocols(协议）

在现实生活中，我们经常会碰到协议，大多数时候它和合同属于同义词。比如毕业时签署就业协议，离婚了要签离婚协议等等。
在计算机的世界里面也会经常遇到协议，比如网络通讯协议，TCP/IP协议神马的。
不过在iOS开发里，此协议非彼协议，它只是一堆方法的名称。协议中不会定义实例变量，也不会实现所声明的任何方法。
它只是向世人宣布：任何号称遵守该协议的对象都必须实现方法X,Y和Z，等等。

协议是一种半强制性的东东，你可以不遵守协议，但一旦你声明要遵守某个协议，就必须做到一些事情。
打个不恰当的比方，佛教有五戒，不杀生、不偷盗、不邪淫、不妄语、不饮酒。你可以不信佛教，但如果你向世人宣布自己是个佛教徒，那么就必须守持五戒，而不能为所欲为。

这里我们定义了AddItemViewControllerDelegate协议，其中列出了两个方法，addItemViewControllerDidCancel:和addItemViewController:didFinishAddingItem:。
注意，代理方法的名字通常会很长。

如果我们宣布ChecklistsViewController遵守该协议，那么它就必须实现这两个方法。一个小的技巧是，我们可以用协议名来引用ChecklistsViewController，使用下面的语法即可：

```
id <AddItemViewControllerDelegate> delegate;
```

这里的delegate变量此时就代表到实现该协议方法的对象的引用。我们可以向delegate变量的对象发送消息，而不用管它到底是个怎样的对象。

按照惯例，代理方法需要将其所有者作为第一个（或唯一的）参数，所以这里我们在两个方法中都将AddItemViewController对象作为参数。这不是必须的，但却是一种很好的方式。例如，一个table view delegate对象可能同时是两个table view的代理。在这种情况下，就必须区分开两个表视图。这就是为什么表视图代理方法中有一个到发送通知的UITableView对象的引用。

如果你有过其它编程语言的开发经验，可能会发现协议和某种叫"接口"的东西很类似。是的，只可惜Objective-C的设计者已经将interface这个术语提前占用了，就只好用@protocol来代替。

注意@class ChecklistItem;这行代码，它的作用是通知代理协议ChecklistItem对象的存在。此前我们使用#import让某个对象了解关于其它对象的信息，那么这两者之间有什么区别呢?

1.@class ChecklistItem只是通知编译器：当你看到ChecklistItem的时候，需要知道它是一个我将要用到的对象名称。但编译器并不知道这个对象能做什么，有哪些属性，只知道它是一个对象。
2.#import "ChecklistItem.h"实际上在编译的时候把ChecklistItem.h文件的内容添加到了当前的文件。此时编译器知道ChecklistItem的所有信息，比如有哪些方法，有哪些属性，等等。如果我们想在当前的类中访问对象的属性或者调用它的方法，就需要用到#import。

这里我们在协议中用@class ，是因为协议方法中有一个ChecklistItem类型的对象参数，但因为这里只是方法声明，我们不需要了解ChecklistItem的其它信息。

术语解释结束了，刚才这一大堆东西似乎还是有点不好消化的，如果你没太看懂，不要纠结于此，先继续前进，等后面回过头来再看。此外，在后面的内容中对有些知识点会不断重复解释的，所以不要害怕。

回到Xcode，还是在AddItemViewController.h中，在@interface和@end之间添加一个属性声明：

```
@property(nonatomic,weak) id <AddItemViewControllerDelegate> delegate;
```

接下来切换到AddItemViewController.m，替换cancel和done方法中的现有代码如下：

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

    [self.delegate addItemViewControllerDidCancel:self];
}

- (IBAction)done:(id)sender {

    ChecklistItem *item = [[ChecklistItem alloc]init];
    item.text = self.textField.text;
    item.checked = NO;

    [self.delegate addItemViewController:self didFinishAddingItem:item];

}
```

当然，不幸的是你的代码刚输完就看到一堆错误：

```
51 - (IBAction)cancel
52 {
53   [self.delegate addItemViewControllerDidCancel:self];
54 }
55
56 - (IBAction)done
57 {
58   ChecklistItem *item = [[ChecklistItem alloc] init];
59   item.text = self.textField.text;
60   item.checked = NO;
61
62   [self.delegate addItemViewController:self didFinishAddingItem:item];
63 }
```

这是因为在AddItemViewController.m中编译器只知道ChecklistItem是个对象，但不知道它究竟是神马。我们仍然要用传统方式来导入其定义。

在AddItemViewController.m的顶部添加一行代码：

#import "ChecklistItem.h"

现在错误消息去无踪~

接下来看看cancel和done 两个动作方法的变化。当用户触碰cancel按钮的时候，会发送addItemViewControllerDidCancel消息到代理对象。同样的是done按钮，只不过发送了另一条消息，并且将一个新的ChecklistItem对象作为参数传递。

如果此时编译运行应用，显然cancel和done按钮是不能正常工作的。因为我们还没用告诉Add Item界面，它的代理对象究竟是谁。这就意味着self.delegate属性是nil,因此消息只有天知道发送到哪儿了。

小提示：

如果你有过其它编程语言的开发经验，可能对这种"空指针"感到很恐怖，特别是C++。在Objective-C中，向nil发送消息是完全有效的，这样做并不会让你的应用直接崩溃。我们也无需添加一个if(self.delegate !=nil)检查。需要注意的是不要给已经释放的对象发送消息。

好了，继续前进。

现在我们需要让ChecklistsViewController成为AddItemViewController的代理。

在Xcode中切换到ChecklistsViewController.h，然后更改其代码为：

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

@interface ChecklistsViewController : UITableViewController<AddItemViewControllerDelegate>

- (IBAction)addItem:(id)sender;


@end
```

注意我们用了#import "AddItemViewController.h"，因为ChecklistsViewController这个界面还是需要了解AddItemViewController界面的信息的，反之则不然。此外我们加了一个尖括号，让ChecklistsViewController遵从AddItemViewControllerDelegate协议。每当在@interface声明部分看到<>尖括号的时候，就意味着这个对象要实现某个特殊的协议。

接下来切换到ChecklistsViewController.m，在@end前添加以下代码用来实现两个协议方法：

```
-(void)addItemViewControllerDidCancel:(AddItemViewController *)controller{

    [self dismissViewControllerAnimated:YES completion:nil];

}

-(void)addItemViewController:(AddItemViewController *)controller didFinishAddingItem:
(ChecklistItem *)item{

    [self dismissViewControllerAnimated:YES completion:nil];
}
```

这两个方法的当前作用是关闭Add Item界面。注意，这里是AddItemViewController自己完成cancel和done方法，只不过我们把责任交给了ChecklistsViewController这个代理对象。

现在让我们来复习下如何在两个对象间设置代理模式。对象A是对象B的代理，而对象B将向对象A发送消息：

1.在对象B的.h中定义一个@protocol 代理
2.在对象B的.h中声明一个代理协议的属性变量
3.让对象B在适当的时候向代理对象发送消息，比如当用户触碰cancel或done按钮时
4.让对象A遵从代理协议，在@interface声明部分添加一个尖括号包含协议
5.通知对象B，对象A现在是它的代理。

这里我们看到还有最后一件事情没有完成：通知AddItemViewController，现在ChecklistsViewController是它的代理。当我们使用storyboard的时候，一个比较合适的地方时在prepareForSegue:sender:方法中。

在Xcode中切换到ChecklistsViewController.m，然后在@end前添加一个方法：

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender{

    if([segue.identifier isEqualToString:@"AddItem"]){

        //1
        UINavigationController *navigationController = segue.destinationViewController;

        //2
        AddItemViewController *controller = (AddItemViewController*)
navigationController.topViewController;

        //3
        controller.delegate = self;
    }
}
```

当界面跳转的时候，UIKit会触发segue的prepareForSegue方法。segue是storyboard中两个视图控制器之间的那个箭头。prepareForSegue允许我们向新的视图控制器传递数据。

接下来让我们仔细看看prepareForSegue方法的代码：

1.新的视图控制器可以在segue.destinationViewController中找到。对于我们这款应用，目标视图控制器不是AddItemViewController，而是包含了它的导航控制器

2.为了获取AddItemViewController对象，我们可以查看导航控制器的topViewController属性，该属性指向导航控制器中的当前活跃界面

3.一旦我们获得了到AddItemViewController对象的引用，就需要将delegate属性设置为self,而self这里其实是ChecklistsViewController.

通过这三个步骤，ChecklistsViewController正式成为AddItemViewController的代理。

再看看下面的代码：

```
if ([segue.identifier isEqualToString:@"AddItem"]) { ...
}
```

这是因为在视图控制器间可能存在多个segue，所以最好为每个segue设置一个标识符。

在Xcode中打开storyboard,然后选中Checklists View Controller和导航控制器之间的segue，在右侧面板中切换到Attributes inspector,然后将Identifier部分输入AddItem

编译运行应用，看看一切是否工作正常。

当我们触碰+按钮的时候，会触发segue到Add Item界面，当我们触碰该界面上cancel或done按钮的时候，AddItemViewController会发送一条消息到其代理，也就是ChecklistsViewController。当然现在代理对象只是简单的关闭Add Item界面，不过很快我就会来点实质的东西。

继续推进之前，再来看一个小东西。

相同和相等

在刚才判断某个segue的标识符时，我们没有用==

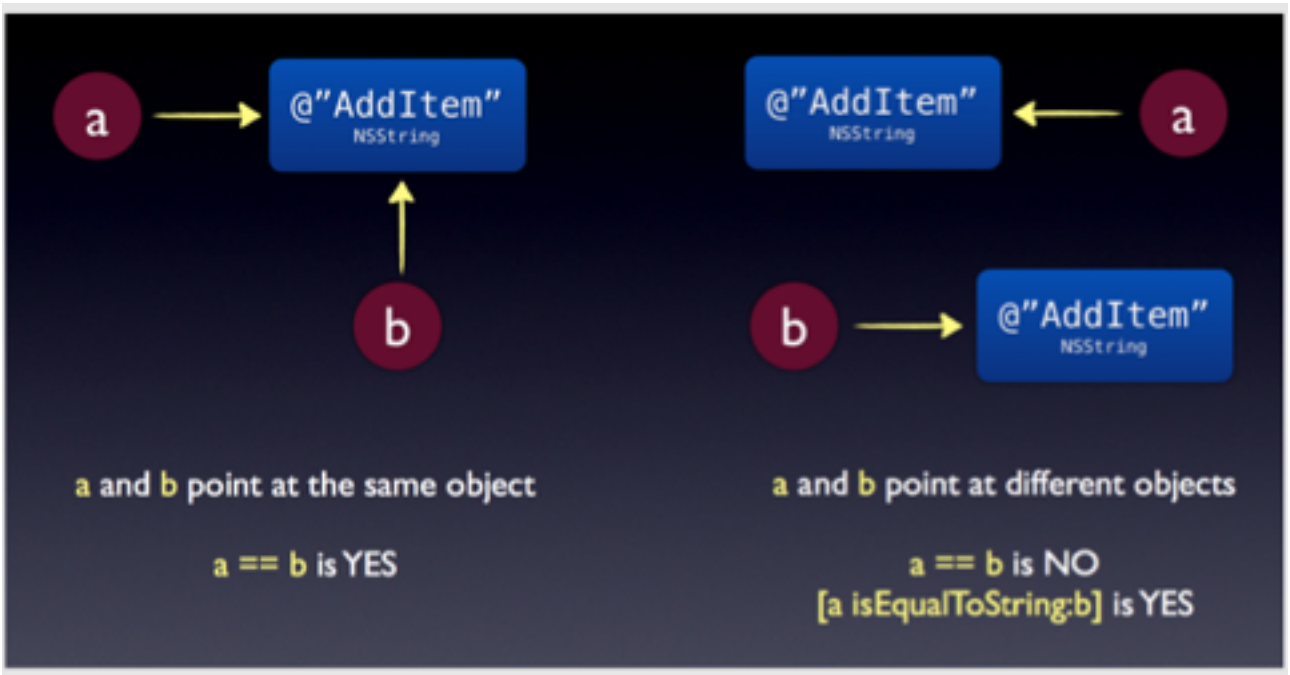
```
if (segue.identifier == @"AddItem") { ...
}
```

这是肿么回事呢？这就涉及到在iOS中如何判断两个对象是相等的。

如果我们用==，实际上是检查两个对象是否是相同的，也就是两个对象其实是同一个对象。在我们这个情况下显然不是这样的，字符串@“AddItem”和segue.identifier是两个独立的字符串对象。

不过两个字符串对象虽然不是“相同”的，但它们的内容却可能是一样的，都是@“AddItem”。为了比较两个对象的值是否相等，我们会使用isEqual方法，比如isEqualToString。

所以，==用来判断两个对象是否是相同的，而isEqual方法则用来判断两个对象的内容是否相等。当然，对于基本数据 类型，用==显然就足够了。



举个例子吧，两个人的名字都叫苍井空，一个是岛国著名的苍老师，而另一个则是苍井家的武道高收益没。它们的显然不是同一个人，因此如果说 cangjingkong1 =2 cangjingkong2，那么结果显然是NO，但[cangjingkong1.name isEqualToString:cangjingkong2.name]则是正确的。

当然，也可能我给你讲了一个关于苍老师的故事，而你听到这个故事就想起自己所了解的苍老师，那么实际上canglaoshi1== canglaoshi2是成立的，是YES

好了，终于走到了这一步，我们可以把新的ChecklistItem放入数据模型和表视图了。否则还那么苦逼的搞个代理模式干吗？！

在Xcode中切换到ChecklistsViewController.m，然后更改didFinishAddingItem这个代理方法的内容如下：

```
-(void)addItemViewController:(AddItemViewController *)controller didFinishAddingItem:
(ChecklistItem *)item{

    NSInteger newRowIndex = [_items count];
    [_items addObject:item];

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:newRowIndex inSection:0];

    NSArray *indexPaths = @[indexPath];
    [self.tableView insertRowsAtIndexPaths:indexPaths
withRowAnimation:UITableViewRowAnimationAutomatic];


    [self dismissViewControllerAnimated:YES completion:nil];
}
```

以上方法的内容和之前在addItem方法中所做的相当类似。实际上我只是把addItem动作方法的代码拷贝粘贴到这个代理方法中。区别在于我们不再自己创建ChecklistItem对象，而是由在AddItemViewController中添加。我们只需要把新的对象添加到_items数组中就可以了。和之前一样，我们通知表视图有一个新的行，然后关闭Add Items 界面。

接下来删除ChecklistsViewController.h和.m中的addItem动作方法。

为了保险起见，在storyboard中检测+按钮是否已经不再和addItem动作方法关联在一起。

编译运行项目，现在我们可以添加自己的事项了！

好吧，目的是实现了！你可能会问，为毛我们不直接在AddItemViewController中增加一个到ChecklistViewController的属性变量？搞这种协议代理神马的太麻烦了，哥到现在脑子里面还是一盆浆糊有木有？！！！

```
// in the .h file
@interface AddItemViewController
@property (nonatomic, weak) ChecklistsViewController *checklistsViewController;
@end
// in the .m file
#import "ChecklistsViewController.h" @implementation AddItemViewController
- (IBAction)done {
// directly call a method from ChecklistsViewController [self.checklistsViewController
addItemWithText:self.textField.text];
}
@end
```

好吧，这样做看起来能省点代码，但却把两个对象紧密耦合在一起了。作为一个通用的设计原则，如果界面A打开了界面B，然后通常我们不需要界面B过多了解触发它的界面A。

如果我们让AddItemViewController获得一个到ChecklistsViewController的直接饮用，那么我们就无法从应用的其它地方来打开Add Item界面了。虽然在这个应用中不会碰到这种麻烦事，但显然在其它应用中会遇到从多个不同的地方打开某一个界面的情况。使用代理模式可以有效的保证界面B和界面A之间的独立性。

此外，代理协议可以被任何想使用的界面所使用。
这就是iOS的开发模式，当然，还有另外一种方式也是可以的，就是所谓的unwind segue，在下一下系列的教程中我们将会碰到。

好了，漫长的一章终于要终结了。在这一章的学习中我们接触到了iOS开发的另一个重要概念代理和协议。我猜你有很多问题，先慢慢消化下吧。

还是福利时间。
虽然是冬天，还是要多看看绿色的，养眼。

养眼的不只是绿色，还有MM