从零开始学iOS7开发系列教程-事务管理软件开发实战-Chapter15

欢迎继续我们的学习。

开发环境：
Xcode 5.1 DP 2 +iOS 7.1 beta2

在之前的内容中，其实我们已经算是完成了一个简单的小to-do list应用。我们可以添加、编辑和删除待办事项，并保存在沙盒文件中。
而从这一章开始，我们将学习如何创建多个代办事务清单。这也是为什么这个应用的名称是Checklists而不是Checklist。

为了实现这一功能，我们需要按以下步骤来完成：

1.创建一个新的界面，用于显示所有的代办事务清单
2.创建一个新的界面，可以让用户添加/编辑代办事务清单
3.当触碰代办事务清单列表中的某一项时，会显示该清单所属的所有代办事项
4.将所有的代办事务清单保存在文件中，然后在需要的时候从该文件中加载。

这里我们需要两个新的界面，也就是说需要两个新的视图控制器：用于显示用户代办事务清单列表的AllListsViewController，以及可以让用户添加和编辑待办事项清单列表的ListDetailViewController

当前应用的首界面是ChecklistsViewController。该文件是Xcode作为Single View Application 模板的一部分创建的，其名称则是根据我们新建项目时所选的Class Prefix命名的。
不过考虑到我们即将做出的调整，这里的Checklists显然有点不合时宜了。因为它用来显示一个单一的待办事项清单。

因此，首先我们将重新命名ChecklistsViewController为ChecklistViewController。

还记得怎么做吗？在Xcode中切换到ChecklistsViewController.h，光标放到@interface这一行语句的ChecklistsViewController上，然后使用菜单中的Refactor工具来更名。

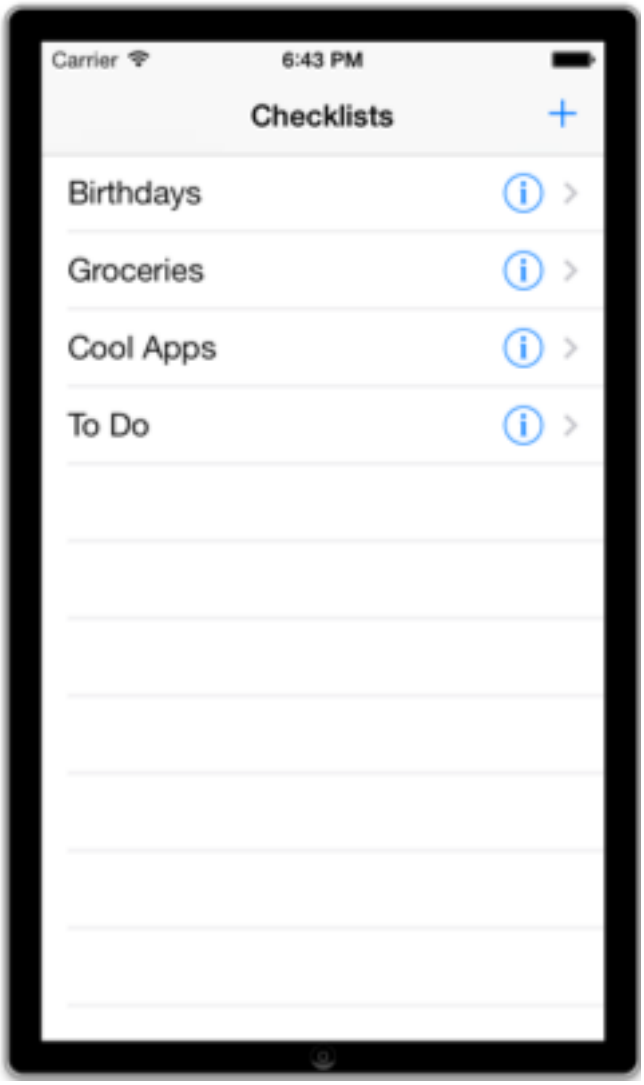
更名后记得clean一下，然后再编译运行一下，看看一切是否正常。

如果没有问题，那么就可以继续下面的工作了。

显示所有代办事项清单的列表界面

首先我们要添加一个AllListsViewController视图控制器，而且这个界面将成为该应用的首界面。

当我们完成后该界面显示如下：



当然，这个界面的显示内容和之前的界面类似。它是一个用于显示Checklist对象（而非ChecklistItem对象）的表视图控制器。
从现在起，我们用AllLists来代指这个界面，用Checklist代指显示单一代办事务清单的界面。

在Xcode左侧的项目导航部分，右键单击Checklists群组，选择New File，在Cocoa Touch部分选择Objective-C class模板。点击Next,然后将新文件命名为AllListsViewController，将其设置为subclass of UITableViewController，其它部分仍然不勾选。

在我们可以顺利编译运行之前，需要对模板默认创建的文件做一些调整工作。首先我们需要在表视图中放入一些伪数据，从而让它可以顺利跑起来。
我一向的习惯是，每做一次小的更新就编译运行一下，看看一切是否顺利。
如果一切正常，我们就可以继续下一步的工作了。

在Xcode中切换到AllListsViewController.m，删除numberOfSectionsInTableView方法。

更改numberOfRowsInSection方法如下：

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
  // Return the number of rows in the section.
    return 3;
}
```

更改cellForRowAtIndexPath方法如下：

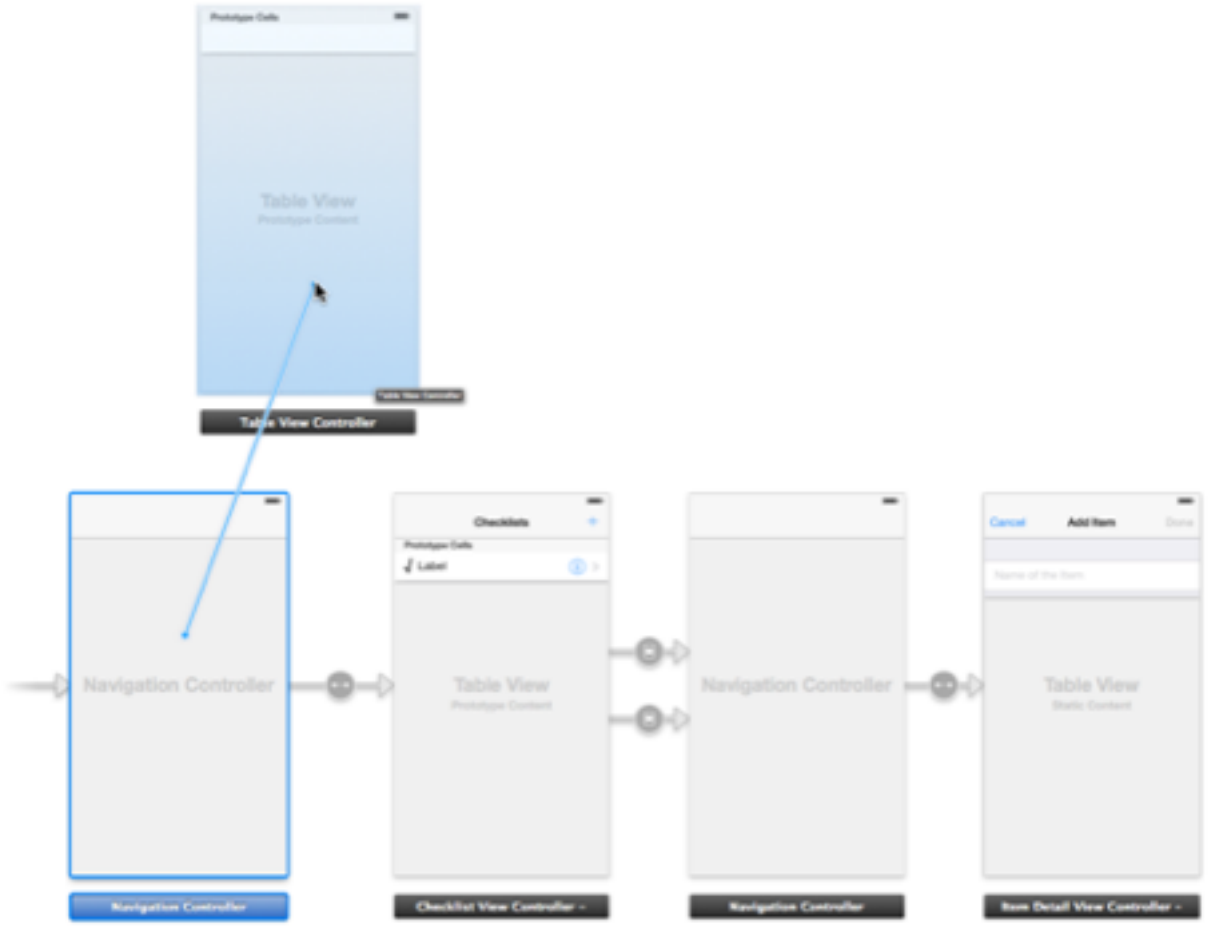
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath
{

    static NSString *CellIdentifier = @"Cell";
   UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

      // Configure the cell...
   if(cell == nil){

      cell = [[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier];
   }
   cell.textLabel.text = [NSString stringWithFormat:@"清单: %ld",(long)indexPath.row];
   return cell;
}
```

以上方法的内容和模板默认的代码很相似，区别在于我们在cell中放入了一些文本。
最后一步当然就是在storyboard中添加新的视图控制器
在Xcode中切换到storyboard，然后从右下方的Objects Library中拖出一条Table View Controller到画布上。
按住ctrl键，从最左侧的导航控制器拖出一条线到这个新的table view controller，从弹出菜单中选择Relationship Segue- root view。
这样一来，之前在导航控制器和Checklist View Controller之前的关联就消失了。此时Checklist不再是应用的首界面。

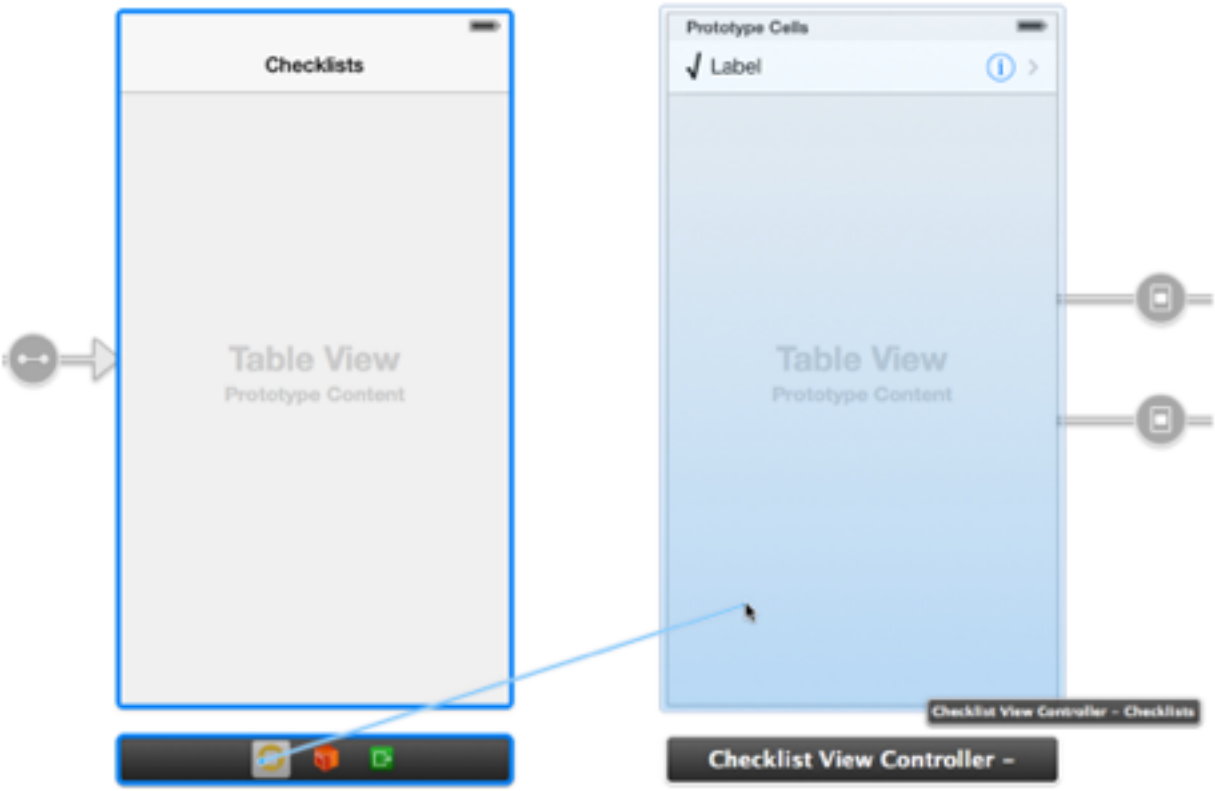


选中这个新的表视图控制器，在Xcode右侧面板中切换到identity inspector，然后将Class属性更改为AllListsViewController。

双击该表视图控制器的导航栏，将标题更改为Checklists
为了让画布井井有条，最好手动调整下画布上的视图控制器的元素，让新的表视图控制器处于其他界面之间。

为了让我们对表视图有更深入的了解，这里暂时不会用prototype cell。当然如果你自己想试试看也没关系，作为练习你甚至可以重写这部分的代码。

从All Lists View Controller中删除空白的prototype cell
按住Ctrl键，从All Lists View Controller底部的icon拖出一条线到右侧的Checklist View Controller，选择push



这样我们就从All Lists界面添加了一个到Checklist界面的push切换。注意这个新的segue没有附着到任何一个按钮或table view cell上面。我们不能直接使用界面元素的交互来触发这个segue。这就意味着后面需要用纯手写代码的方式来实现。

点击这个新的segue，在Xcode右侧面板中切换到Attributes inspector，然后将identifier属性更改为ShowChecklist。

这里我们选择的的Style是Push，因为当执行该segue的时候，我们会将Checklist View Controller push到导航堆栈上。

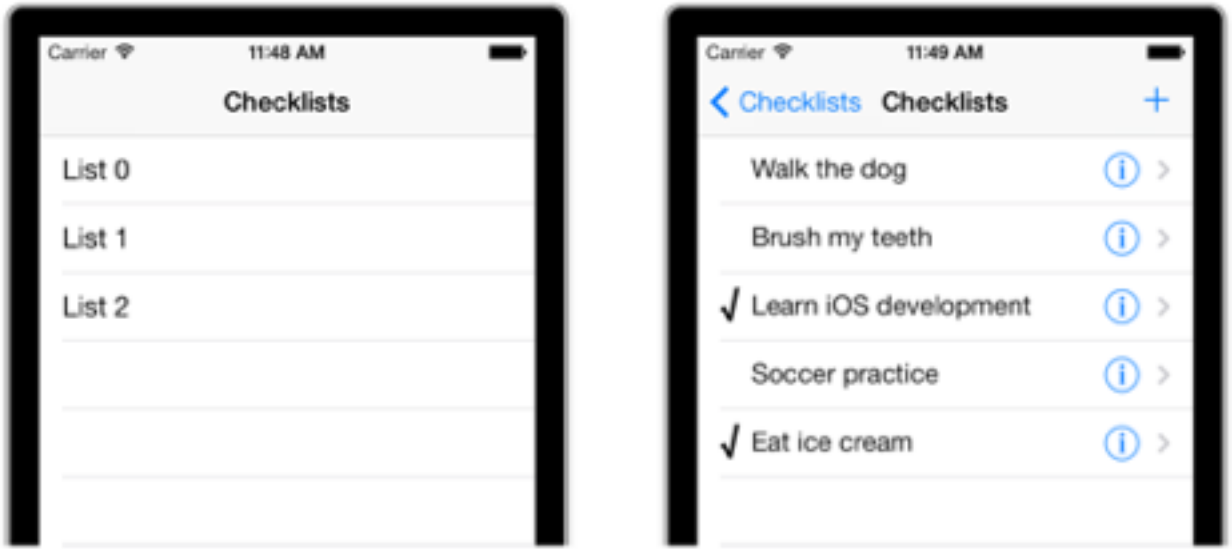
切换到AllListsViewController.m，添加一个didSelectRowAtIndexPath方法：

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

    [self performSegueWithIdentifier:@"ShowChecklist" sender:nil];
}
```

记住当用户触碰表视图中的某一行时就会触发这个代理方法。之前，触碰某一行会自动执行segue，因为我们将segue和prototype cell关联在一起。但这里我们不使用prototype cell，因此需要手写代码来执行segue。当然你也看到了，没有想象中那么难，只需要调用一个performSegueWithIdentifier方法就足以实现了。

编译运行应用，效果是类似下面的：



当我们触碰某一行的时候，会看到熟悉的ChecklistViewController界面出现在眼前。我摸可以触碰左上角的Checklists按钮返回首界面。这就是导航控制器的真正威力所在！

注意，如果你的应用没有顺利进行，那么可以用FileMerge工具来对比下代码。很可能出错的一点是：
在AllListsViewController.m中，cellForRowAtIndexPath方法中重用cell的一行代码是：

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
```

而不是：

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

之所以容易出现上面的错误，是因为Xcode的自动完成功能有时候会让你选择错误的方法而不自知！

接下来我们需要将Checklist View Controller的功能复制到新的All Lists界面中。我们将在顶部添加一个+按钮，让用户可以创建新的checklist，让用户可以用swipe-to-delte的方式删除checklist，让用户可以用细节显示按钮来编辑checklist的名称。当然，我们还需要将Checklis 对数组保存到Checklists.plist文件中。
因为这些工作和之前大同小异，所以接下来的工作会稍微轻松点，主要是复习之前的内容

首先当然还是创建一个代表checklist的数据模型。

在Xcode中基于Objective-C class模板创建一个新的文件，设置subclass of NSObject，然后将其命名为Checklist。

更改Checklist.h的代码如下：

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

@interface Checklist : NSObject

@property(nonatomic,copy)NSString *name;

@end
```

这里我们为Checklist对象添加了一个名为name的字符串属性。

接下来我们将为AllListsViewController添加一个数组，从而可以保存新的Checklist对象。

切换到AllListsViewController.m中，在文件顶部添加一行代码：’
#import “Checklist.h"

然后在@implementation部分添加一个新的实例变量，用于保存Checklist对象：

```
@implementation AllListsViewController{

   NSMutableArray *_lists;
}
```

为了方便起见，我们将先使用测试数据来填充列表，具体是在initWithCoder方法中完成。
记住当应用从storyboard中加载视图控制器的时候，UIKit将会自动触发这一方法

在AllListsViewController.m中，添加一个initWithCoder方法如下;

```
-(id)initWithCoder:(NSCoder *)aDecoder{

   if((self =[super initWithCoder:aDecoder])){

      _lists = [[NSMutableArray alloc]initWithCapacity:20];
      Checklist *list;

      list = [[Checklist alloc]init];
      list.name = @"娱乐";
```

```
        [_lists addObject:list];

        list = [[Checklist alloc]init];
        list.name = @"工作";
        [_lists addObject:list];

        list = [[Checklist alloc]init];
        list.name = @"学习";
        [_lists addObject:list];

        list = [[Checklist alloc]init];
        list.name = @"家庭";
        [_lists addObject:list];

    }
    return self;

}
```

如果你足够细心，会看到在AllListsViewController.m中已经有一个initWithStyle方法了。
还记得表视图控制器的三个init方法吗？其中initWithCoder用于从storyboard中加载视图控制器，initWithNib用于从nib文件中加载视图控制器，而initWithStyle则用于手动创建视图控制器。
因为我们这里将使用storyboard来加载视图控制器，因此用initWithCoder方法就好了。
从.m中删掉initWithStyle方法。

还是在AllListsViewController.m中，更改numberOfRowsInSection方法如下：

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
  // Return the number of rows in the section.
    return [_lists count];
}
```

最后更改cellForRowAtIndexPath方法，用于为row行数据添加所需的cell

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

     static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

        // Configure the cell...
    if(cell == nil){

        cell = [[UITableViewCell alloc]initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier];
```

```
    }

    Checklist *checklist = _lists[indexPath.row];
    cell.textLabel.text = checklist.name;
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}
```

编译运行应用，可以看到类似下面的画面：

好了，在继续之前，让我们来补充点理论知识。

创建table view cell的多种方式

到目前为止，我们已经多次接触到cellForRowAtIndexPath方法了。而这里的cellForRowAtIndexPath方法比之前ChecklistViewController中的方法显然做了更多的事情。在那里我们只用了区区一行代码就获取了一个新的table view cell。
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"ChecklistItem"];

而这里我们用了一大堆代码才实现相同的功能：
```
static NSString *CellIdentifier = @"Cell";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
}
```

当然，相同的是两个方法中都调用了dequeueReusableCellWithIdentifier方法。只是在ChecklistViewController中，storyboard里面提供了一个prototype cell，而这里则没有。
如果表视图无法找到可重用的cell，dequeueReusableCellWithIdentifier方法就会返回nil,此时我们需要手动创建自己的cell。

在iOS开发中，目前提供了四种方式来创建table view cell：

1.使用prototype cell
这是目前为止最方便最快捷的创建table view cell的方式。之前在ChecklistViewController中我们就是用这个方法来实现的。

2.使用static cell
之前在Add/Edit Item界面中我们用的就是static cell
当然这种方法其实也很方便，只不过它的限制很大，因为只有当我们实现一键知道有哪些cell时才能用static cell。
用它的好处是我们无需提供任何数据源的代理方法（包括cellForRowAtIndexPath）

3.手动创建static cell
也就是我们这里所采用的方式
这种方式是iOS5之前最常用的方式。这种方式最麻烦，但灵活度也最大。
如果你不幸需要维护之前石器时代的iOS代码，那么或许会经常看到类似的代码

4.使用nib文件

nib（也可以说XIB）文件类似于storyboard，不过其中仅包含了一个单一的视图控制器，此时可以使用定制的UITableViewCell对象。
这种方法和使用prototype cell类似，只不过我们将在storyboard之外来完成。

当我们手动创建一个cell时，需要指定一个特定的cell style，这种默认的样式中将提供一个预先设置好的布局，其中包含了标签和一个image视图。对于All Lists View Controller，我们使用了Default样式。不过在随后的教程中我们将改用Subtitle，在这种样式下cell具有一个比主标签小一些的次级标签。

使用标准的cell 样式意味着我们无需自己设计cell的布局。对很多应用来说，标准布局已经足够了。prototype cell和static cell也可以使用这些标准的cell 样式。prototype或static cell的默认样式是Custom，也就是说我们需要使用自己的标签，但可以通过Interface builder将其更改为默认样式中的一种。

最后，一个小小的警告。
我经常看到有些XD在写代码的时候，会为每一行创建一个新的cell，而不是重用已有的cell。千万别这么做！尽量使用dequeueReusableCellWithIdentifier询问表视图是否有可重用的cell。为每一行创建一个新的cell会拖垮你应用的速度，因为创建一个新的对象比起重用已有的对象总是要慢的多。对于移动设备来说，资源是很宝贵的。为了保证最佳性能和流畅度，最好重用现有的cell!

好了，今天的学习就到此结束吧。
今天凌晨的比赛，恒大0：3输给了拜仁。
有句话说的好，世俱杯让世界认识恒大，拜仁让恒大认识世界。

好吧，还是看看美女调节下心情吧