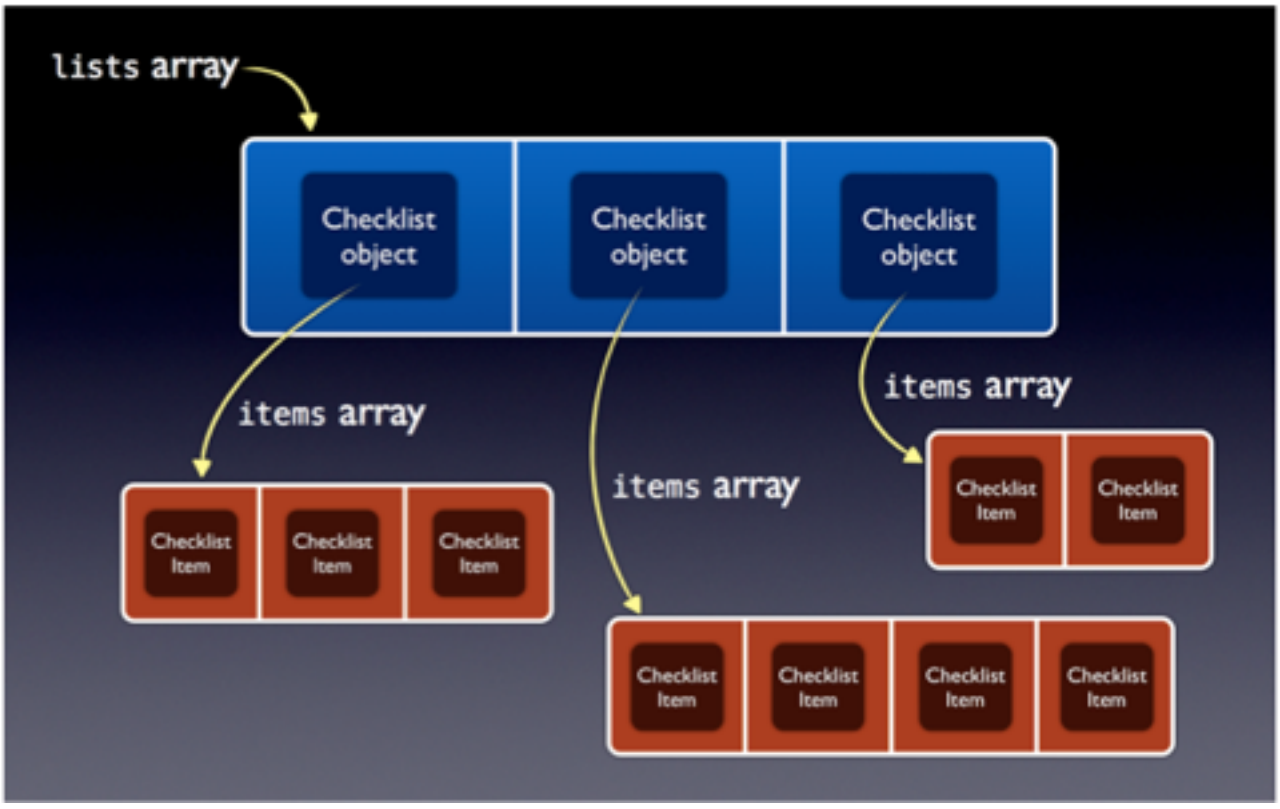
从零开始学iOS7开发系列教程-事务管理软件开发实战-Chapter17

欢迎继续我们的学习。

开发环境：
Xcode 5.1 DP 2 +iOS 7.1 beta2

到目前为止，我们的项目一切进展顺利。遗憾的是checklists中并没有包含任何的to-do 待办事项。待办事项清单和真实的checklists实际上还是分离的。

因此我们需要调整数据模型到类似下面的样子：



在这个数据模型中仍然有包含了Checklist对象的_lists数组，只是现在每个checklist都有自己的数组，这个数组中包含了对应的ChecklistItem对象。

为此，在Xcode中切换到Checklist.h，添加一个属性变量声明：
@property(nonatomic,strong)NSMutableArray *items;

接下来在Checklist.m中添加一个init初始化方法：

-(id)init{

```
    if((self =[super init])){

        self.items = [[NSMutableArray alloc]initWithCapacity:20];
    }
    return self;

}
```

现在Checklist对象将包含一个ChecklistItem对象的数组，初始状态下，该数组是空的。

之前我们曾经修改过AllListsViewController.m中的prepareForSegue方法。当用户触碰主界面上的某一行时，应用会通过segue切换到ChecklistViewController界面，同时和该行相关的Checklist对象会作为参数传递过去。

不过当前ChecklistViewController仍然从其自身的_items数组中获取ChecklistItem对象。现在我们需要让它从checklist对象中获取items数组。

切换到ChecklistViewController.m，对以下方法做出修改：

```
-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    return [self.checklist.items count];
}

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

    UITableViewCell *cell =[tableView dequeueReusableCellWithIdentifier:@"ChecklistItem"];

    ChecklistItem *item = self.checklist.items[indexPath.row];


    [self configureTextForCell:cell withChecklistItem:item];
    [self configureCheckmarkForCell:cell withChecklistItem:item];

    return cell;
}

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

    UITableViewCell *cell =[tableView cellForRowAtIndexPath:indexPath];
    ChecklistItem *item = self.checklist.items[indexPath.row];
    [item toggleChecked];


    [self configureCheckmarkForCell:cell withChecklistItem:item];
    [self saveChecklistItems];

    [tableView deselectRowAtIndexPath:indexPath animated:YES];
```

```
}



-(void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath{

    [self.checklist.items removeObjectAtIndex:indexPath.row];
    [self saveChecklistItems];

    NSArray *indexPaths = @[indexPath];

    [tableView deleteRowsAtIndexPaths:indexPaths
withRowAnimation:UITableViewRowAnimationAutomatic];

}

#pragma mark itemDetailviewController delegate

-(void)itemDetailViewControllerDidCancel:(ItemDetailViewController *)controller{

    [self dismissViewControllerAnimated:YES completion:nil];

}

-(void)itemDetailViewController:(ItemDetailViewController *)controller didFinishAddingItem:
(ChecklistItem *)item{

    NSInteger newRowIndex = [self.checklist.items count];
    [self.checklist.items addObject:item];


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

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

    [self dismissViewControllerAnimated:YES completion:nil];
}


-(void)itemDetailViewcontroller:(ItemDetailViewController *)controller didFinishEditingItem:
(ChecklistItem *)item{

    NSInteger index = [self.checklist.items indexOfObject:item];
```

```
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index inSection:0];

    UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:indexPath];

    [self configureTextForCell:cell withChecklistItem:item];
    [self saveChecklistItems];
    [self dismissViewControllerAnimated:YES completion:nil];

}



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

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

        //1
        UINavigationController *navigationController = segue.destinationViewController;

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

        //3
        controller.delegate = self;
    }else if([segue.identifier isEqualToString:@"EditItem"]){

        UINavigationController *navigationController = segue.destinationViewController;

        ItemDetailViewController *controller = (ItemDetailViewController*)
navigationController.topViewController;

        controller.delegate = self;

        NSIndexPath * indexPath = [self.tableView indexPathForCell:sender];

        controller.itemToEdit = self.checklist.items[indexPath.row];

    }
}
```

注意在上面的代码中，我们用self.checklist.items(紫色代码）替代了之前的_items。

接下来从ChecklistViewController.m中删除以下方法：

• (NSString *)documentsDirectory • (NSString *)dataFilePath
• (void)saveChecklistItems
• (void)loadChecklistItems

• (id)initWithCoder:(NSCoder *)aDecoder
这看起来很可怕，一下子删掉这么多方法代码，真的不会出问题吗?
之前我们从文件中加载和保存checlist items，不过现在这个工作已经不再是视图控制器负责的了。最好是让Checklist对象可以自行完成这些工作。加载和保存数据模型对象实际上是数据模型自己的事情，而不是控制器该做的事情。

不过在完成这个大的变动之前，首先让我们测试下当前的状况。
首先会看到Xcode提示有4个error，因为我们在不同的地方仍然在调用saveChecklistItem方法。显然现在应该删除这些代码行。

在ChecklistViewController.m中找到并删除调用saveChecklistItems的代码:
[self saveChecklistItems];

接下来从@implementation部分删掉_items实例变量。

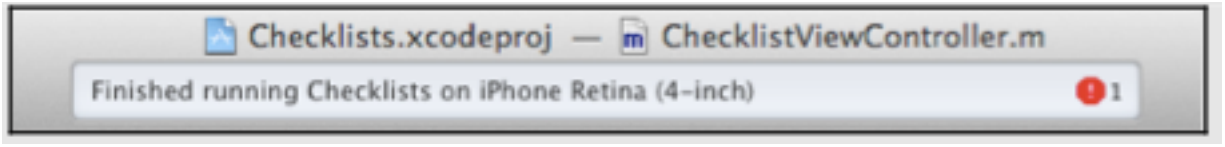
之前是:
@implementation ChecklistViewController {
NSMutableArray *_items;
}
现在更改为:

@implementation ChecklistViewController

因为这里不再有任何实例变量，所以{}花括号也就没有存在的必要了。
需要注意的是，不同于其它语句，@implementation语句行是不会用分号来结束的!

关于Xcode的错误提示

当Xcode检测到代码中的明显错误时，会在顶部显示一个警告或错误图标。



当我们修复了这个错误后，有时候错误提示不会立即消失。所以最好的方式是command+Bl来重新编译一下。如果仍然有错误，就说明没有修改成功。如果错误消失了，当然皆大欢喜。

接下来让我们给不同的Checklist对象添加一些伪数据，这样就可以知道新的数据模型是否生效了。在AllListsViewController的initWithCoder方法中，我们已经在_lists数组中放入了一些伪Checklist对象，现在是时候添加一些新东西了。

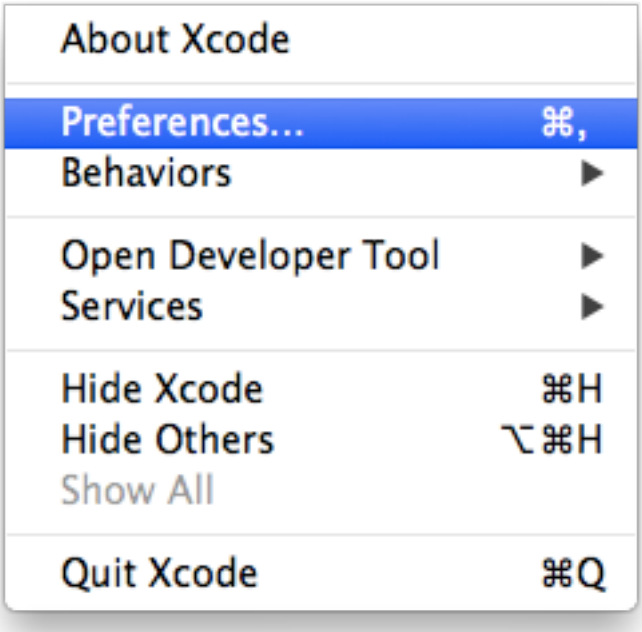
在xcode中切换到AllListsViewController.m，在文件顶部添加一行代码:

#import “ChecklistItem.h"

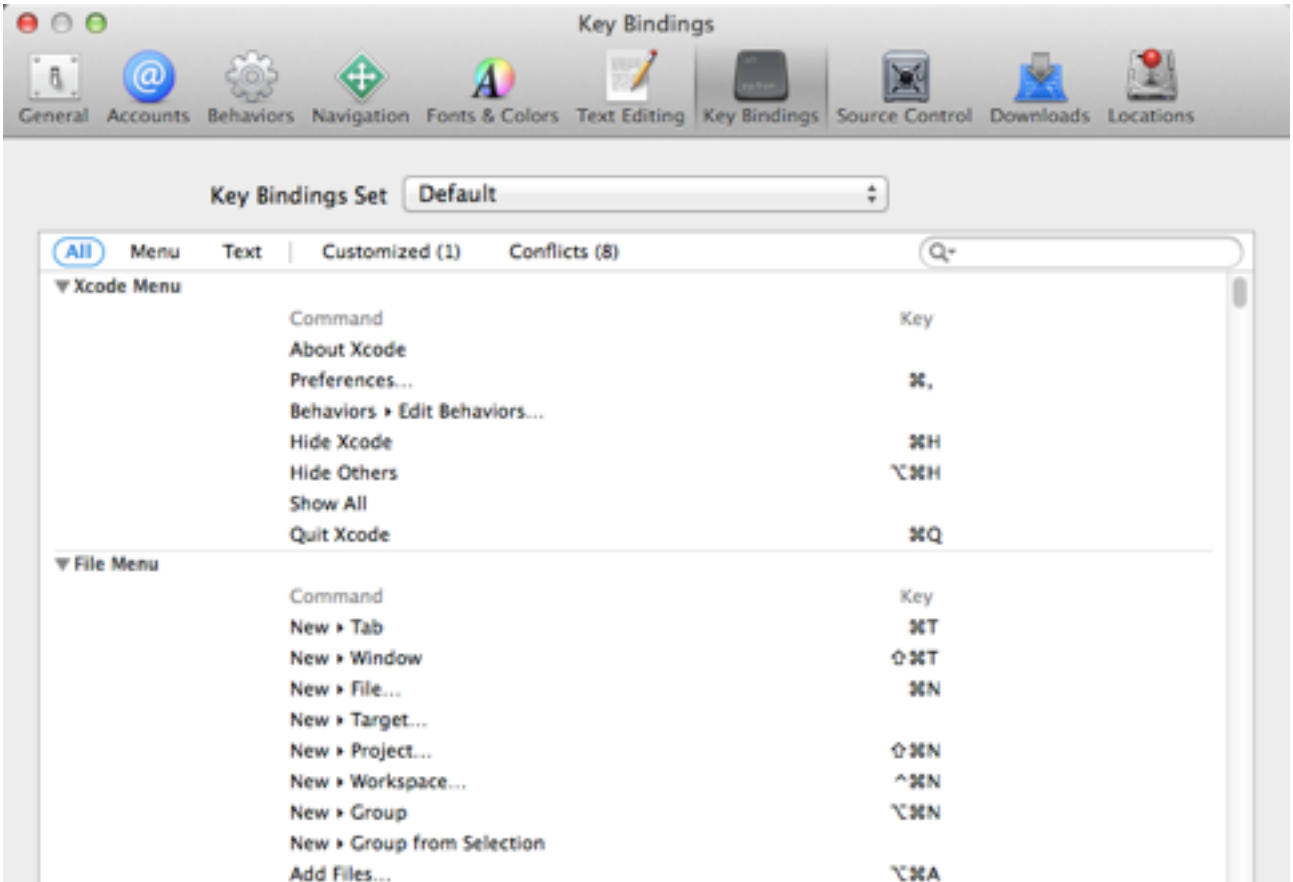
这里介绍一个Xcode的小技巧，也就是Xcode的快捷键。

当我们在Xcode的.m文件时，经常会发现文件中有很多代码。
假如我们想一键回到文件的顶部（比如刚才的操作），可以按command+向上的键；反过来，如果我们想一键回到文件的底部，可以按command+向下的键。

Xcode的快捷键当然不止这一两个，那么我们如何了解所有的快捷键用法呢？
在Xcode的菜单栏上选择preferences...



然后切换到下图的Key Bindings选项卡，就可以看到Xcode的所有快捷键了。
如果你想高效coding，了解一些快捷键是很有用的，虽然开始用有点恼火，用久了之后就会发现自己的coding效率大增。



好了，继续刚才的任务。

还是在AllListsViewController.m中，更改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];


        for(Checklist *list in _lists){
            ChecklistItem *item = [[ChecklistItem alloc]init];
            item.text = [NSString stringWithFormat:@"Item for : %@",list.name];
            [list.items addObject:item];
        }
    }
    return self;

}
```

在以上代码中，只有黄色高亮显示的部分是新的。
和之前不同，这里我们又将接触到一个新的关键词-for。
for语句是编程中一个非常特殊但又非常重要的语法结构。

编程语言的语法结构

在我们学习英语或其它非母语的时候，语法是非常基础而又极其重要的东西。如果不懂语法，别人就会说我们学的是chenglish。在学习计算机编程语言的时候，语法同样非常重要。

这里让我们来复习下之前已经接触过的语法结构。
现代编程语言至少提供了以下的基本语法结构：

1.可以将数值保存在变量中

有的变量很简单（术语党称之为基本数据类型,primitive type），比如int（整型变量，不是整形变量！），BOOL（布尔变量），等等。
有的变量很高大上，它们可以用来存储对象（比如UIButton,ChecklistItem)。
还有一些变量可以保存对象的集合（NSMutableArray)

2.可以从变量中读取数值，并使用它们进行基本数学运算或比较

基本数学运算当然就是加减乘除之类的了，比较就是（大于等于，不等于，等等）

3.可以做出条件判断

比如之前我们学过if判断语句，当然还有一个switch判断语句我们还没有接触过。

4.可以将复杂的功能整合到类似方法和函数的组合单元中。

我们可以调用这些方法或函数，并获得一个返回值，以便在后续的计算中使用

5.可以将函数（方法）和数据（变量）整合到对象（类）中。

6.可以重复执行一系列语句的组合。

这就是for语句的作用了。当然还有其它方式可以来重复执行语句的组合，比如while和do - while
重复计算这种脑残行为其实是计算机最擅长的行为。

可以认为所有的计算设备都是强迫症的重度患者吗？~

有了以上的基本语法结构，我们可以构建任何复杂的事物。在此之前我们已经接触了大部分的内容，不过重复计算（或者用术语党的说法叫“循环”）还是个新事物。
如果你能掌握以上所有的概念，那么就在成为一个软件开发者的道路上前进了一大步。

接下来让我们来看看所谓的loop循环语句究竟是怎样工作的。

```
for (Checklist *list in _lists) {
...
}
```

上面语句的意思是，对于_lists数组中的每一个Checklist对象，执行花括号之间的语句。

在第一次执行循环的过程中，list变量会保持一个到娱乐这个checklist的引用，因为它是我们在数组中创建并添加的第一个Checklist对象。
接着在循环语句中执行了下面的任务：

```
ChecklistItem *item = [[ChecklistItem alloc] init];
```

```
item.text = [NSString stringWithFormat:
@"Item for %@", list.name];
[list.items addObject:item];
```

这几行语句相信大家都不陌生了。我们创建并初始化了一个ChecklistItem对象。然后将其text属性设置为"Item for 娱乐"，因为这里的%@占位符将使用Checklist对象的name属性来替代（而第一个就是"娱乐"）。
最后我们将这个新的ChecklistItem添加到"娱乐"checklist对象中，或者说添加到它的items数组中。

此时我们就完成了循环的第一轮。
接下来for语句会再次查找_lists数组，然后会发现还有3个Checklist对象。
然后它就会将下一个Checklist对象，也就是"工作"赋予list变量，接着重复上述的步骤。
此时文本内容将是"Item for 工作"。最后会将这个新的ChecklistItem添加到"工作"checklist对象的items数组中。

接下来循环会再次查找_lists数组，。。。

直到程序发现_lists数组中没有新的对象了，就会停止循环语句。

在开发的过程中使用循环可以节省大量的时间和精力。

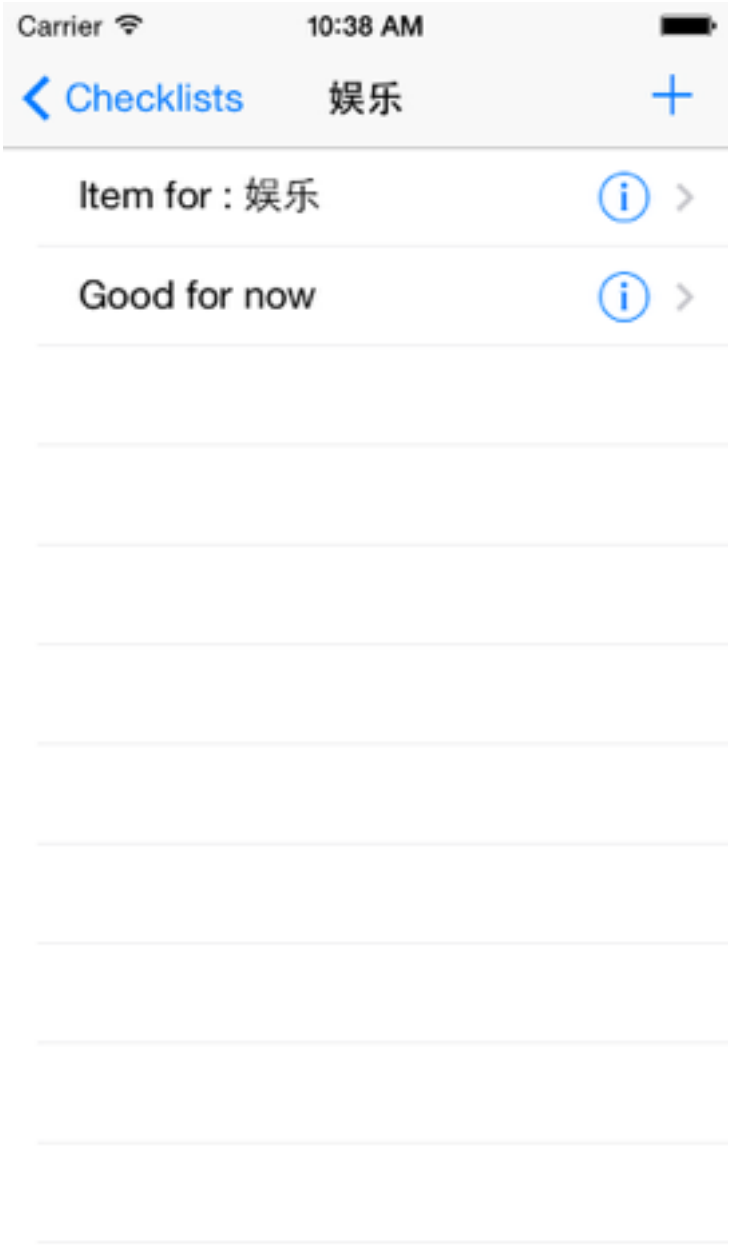
对于刚才的代码，我们也可以用下面的方式来书写：

```
Checklist* list; ChecklistItem *item;
list = _lists[0];
item = [[ChecklistItem alloc] init]; item.text = @"Item for 娱乐"; [list.items addObject:item];
list = _lists[1];
item = [[ChecklistItem alloc] init]; item.text = @"Item for 工作"; [list.items addObject:item];
list = _lists[2];
item = [[ChecklistItem alloc] init]; item.text = @"Item for 学习"; [list.items addObject:item];
list = _lists[3];
item = [[ChecklistItem alloc] init]; item.text = @"Item for 家庭"; [list.items addObject:item];
```

好吧，你可能会说这没神马，哥打字速度快，不怕。
不过如果有100个Checklist对象肿么办？如果有1000个，10000个呢？
你打算用拷贝粘贴的方式来完成这段代码吗？还是选择用Loop轻轻松松搞定一切？

实际上，大多数时候我们并不知道可能会有多少个对象，因此有时候甚至不可能手写完成任务。通过使用循环语句，我们无需为此担忧。
循环语句可以很好的处理三个事项，也可以同样高效的处理300000个事项。
另外一个好消息是，循环和数组经常在一起搭配着工作。

编译运行项目，此时你会看到每个checklist都有属于自己的代办事项了。
你可以试着玩一下，添加和删除事项，然后看看每个list是否和其它list完全独立。



接下来我们要把刚才删掉的load/save代码放回来。
这一次我们会让AllListsViewController来主持相关工作。

在Xcode中切换到AllListsViewController.m，然后在initWithCoder方法之前添加以下方法：

```
-(NSString*)documentsDirectory{

    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths firstObject];
    return documentsDirectory;

}

-(NSString*)dataFilePath{
```

```
    return [[self documentsDirectory]stringByAppendingPathComponent:@"Checklists.plist"];
}

-(void)saveChecklists{
    NSMutableData *data = [[NSMutableData alloc]init];

    NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]initForWritingWithMutableData:data];

    [archiver encodeObject:_lists forKey:@"Checklists"];
    [archiver finishEncoding];
    [data writeToFile:[self dataFilePath] atomically:YES];

}

-(void)loadChecklists{

    NSString *path = [self dataFilePath];
    if([[NSFileManager defaultManager]fileExistsAtPath:path]){

        NSData *data =[[NSData alloc]initWithContentsOfFile:path];
        NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver
alloc]initForReadingWithData:data];

        _lists = [unarchiver decodeObjectForKey:@"Checklists"];
        [unarchiver finishDecoding];

    }else{

        _lists = [[NSMutableArray alloc]initWithCapacity:20];
    }

}
```

以上方法的内容和之前我们在ChecklistViewController中所做的几乎完全相同，区别在于我们加载和保存的是_list数组，而不是_items数组。

接下来更改initWithCoder方法：

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

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

        [self loadChecklists];

    }
    return self;

}
```

这里我们删除了之前的测试数据，然后让loadChecklists方法来完成一切。

还有一些扫尾工作没有完成，不过这里再介绍一个Xcode中写代码的小技巧。

用#pragma mark来人工区分代码块

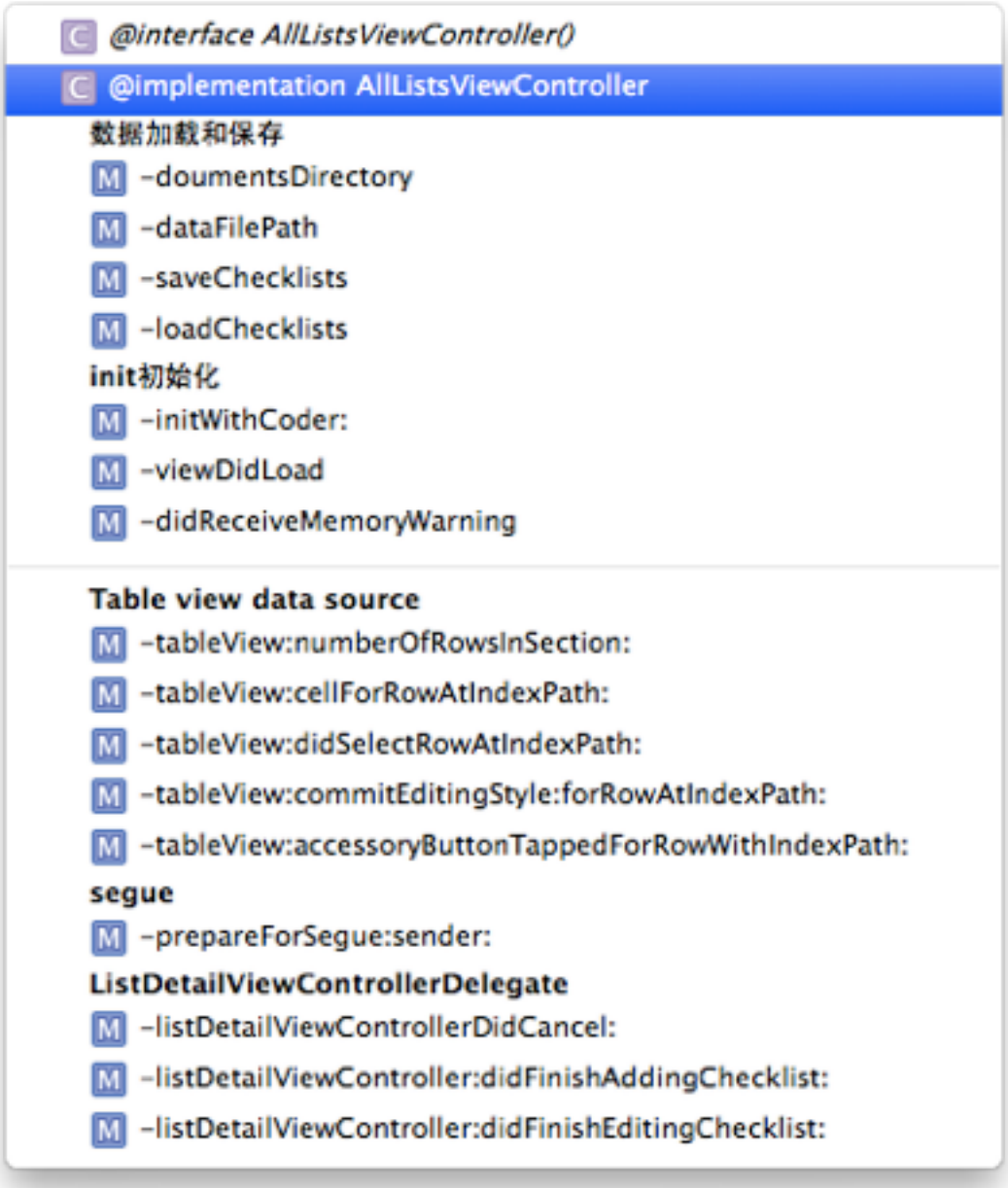
当我们在Xcode中编写代码的时候，如果遇到一些内容很复杂的.m文件，里面可能会有几十个方法。虽然可以用注释的方式来为每个方法做一些解释，但一大堆方法放在一起的时候，就很难理解整个文件的代码结构了。
这时可以借助一个特殊的语句#pragma mark 来帮助我们。

比如我们可以在刚才涉及到数据保存和加载的几个方法之前添加一行语句
#pragma mark 数据加载和保存

然后在表视图的数据源相关代理方法前添加一行
#pragma mark 表视图数据源代理方法

当我们点击Xcode上方的.m旁边的紫色背景的小图标时：

会看到一个下拉清单：



可以看到刚才#pragma mark之后的文本会用黑色加粗的方式显示在列表中。通过这些文本，我们可以很清楚的了解哪些方法属于同一类的方法。

通过适当的添加#pragma mark语句和注释语句，可以让自己或团队所开发的项目更加具有可读性和可维护性，而不是一个人自娱自乐。

此时还有一些工作要做。

在Xcode中切换到Checklist.h，在@interface这一行语句后面添加一个<NScoding>协议声明：
@interface Checklist : NSObject<NSCoding>

接下来做神马呢？

切换到Checklist.m，然后添加以下方法：

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

   if((self = [super init])){

      self.name = [aDecoder decodeObjectForKey:@"Name"];
      self.name = [aDecoder decodeObjectForKey:@"Items"];
   }
   return self;
}

-(void)encodeWithCoder:(NSCoder *)aCoder{

   [aCoder encodeObject:self.name forKey:@"Name"];
   [aCoder encodeObject:self.items forKey:@"Items"];
}
```

因为name和items属性都属于对象（NSString和NSMutableArray），所以可以用decodeObjectForKey:和encodeObject:forKey:来加载和保存。

重要说明！！！
千万不要把之前的init初始化方法从Checklist.m中删除了！
该对象需要init和initWithCoder两个方法才能正常工作。当用户添加一个新的checklist到应用中时会调用常规的init方法。而initWithCoder方法则用于当应用启动时加载已有的checklists。
如果没有常规的init方法，那么当用户尝试添加新的to-do 事项到checklist的时候程序就会崩溃。因为此时self.items数组没有被初始化，仍然是nil。

好了，在编译运行应用之前，记得从模拟器的沙盒地址中删除.plist文件。
如果不删除的话，应用很可能崩溃，因为之前的文件内部结构和新的数据模型不再匹配了。

奇怪的程序崩溃现象

在我第一次写这篇教程的时候，忘了在编译运行前删除.plist文件。程序似乎仍然可以正常工作，但是当我添加了一个新的checklist之后，整个世界崩溃了：

*** Terminating app due to uncaught exception 'NSRangeException', reason: '*** -[NSMutableIndexSet addIndexesInRange:]: Range {2147483647, 1} exceeds maximum index value of NSNotFound - 1'

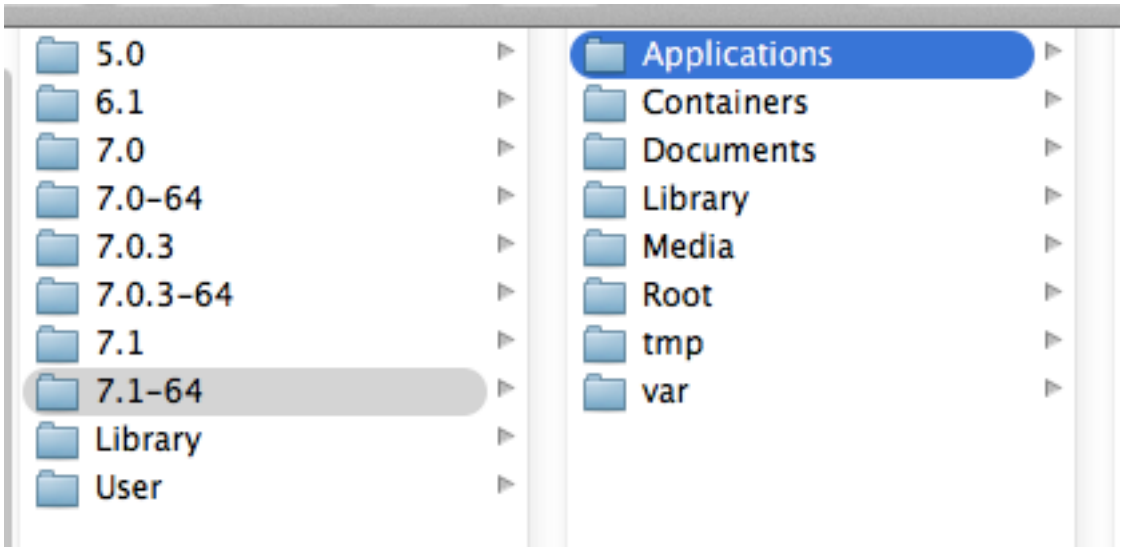
导致问题的代码行是：

```
[self.tableView insertRowsAtIndexPaths:indexPaths
withRowAnimation:UITableViewRowAnimationAutomatic];
```

这是个非常奇怪的错误信息，我想了很久也找不到原因。最终还是想到了那个旧的plist文件。当我从应用的沙盒中删除它之后，整个世界清静了。
为了确保问题是由这个文件引起的，我把文件备份又放回去，果然再次崩溃。

错误的原因在于，尽管旧的plist文件的内部格式不再符合新的数据模型，程序还是加载了它，从而让表视图处于尴尬的境地。此后对表视图的操作就会导致应用崩溃。

在学习iOS开发的过程中，我们经常会遇到此类的bug。也就是说程序崩溃并非由代码直接引起的，而是看你由之前做错的事情引起的。
在后续的教程中，我们会专门来学习一下debug的技巧。
不过有一个小小的技巧是，为了避免受到之前的错误影响，在重新编译之前，我们最好找到沙盒所在的Applications文件夹，然后把所有的沙盒文件夹全部删除。



好了，现在大胆的编译运行应用，然后尝试添加一个checklist和对应的to-do事项。

关闭应用，再次运行。

Oops,你会发现界面中没有内容。

这是肿么办？辛苦这大半天就是这个结果？不甘心啊！

预知后事如何，且听下回分解。

今日福利，仙境般的布达拉宫

看起来还是美女比较养眼