



iOS 5 By Tutorials

第 2 章  
初级 ARC

版本 1.0

翻译时间：2012-11-26

DevDiv 热心网友自发组织翻译

## 写在前面

iOS 5 By Tutorials 是由 iOS Tutorial Team 所编写，在这里希望大家尊重原创，尊重知识版权，此次翻译是 DevDiv 热情网友感觉此书还行，便自发组织翻译，并无偿分享给广大 iOS 开发者。内容仅供交流学习使用，切勿商用或者其它一切用途。

如果你觉得内容可以的话，请主动到官网进行购买：

<http://www.raywenderlich.com/>

**严重警告：限下载后 24 小时内删除**

读者查看下面的帖子可以持续关注章节翻译更新情况

**[iOS 5 By Tutorials 翻译各章节汇总](#)**

## 目录

写在前面	2
目录	3
第 1 章	介绍 4
第 2 章	ARC 基础 (Beginning ARC) 5
2.0.	工作原理 (How It Works) 5
2.1.	指针让对象存在 (Pointers Keep Objects Alive) 6
2.2.	应用 (The App) 10
2.3.	自动转化 (Automatic Conversion) 15
2.4.	AppDelegate.m 22
2.5.	Main.m 23
2.6.	SoundEffect.m 24
2.7.	SVProgressHUD.m 25
2.8.	实战 (Doing It For Real) 26
2.9.	移植问题 (Migration Woes) 28
2.10.	手工转化 (Converting By Hand) 31
2.11.	Dealloc 32
2.12.	SoundEffect Getter 方法 32
2.13.	请释放我, 让我走 33
2.14.	属性 34
2.15.	免费桥接 (Toll-Free Bridging) 38
2.16.	委托和弱属性 44
2.17.	Unsafe_unretained 53
2.18.	在 iOS4 上使用 ARC 54
2.19.	下一步去哪? (Where To Go From Here?) 55

## 第 1 章 介绍

### iOS 5 By Tutorials 中文翻译\_第一章\_介绍

DevDiv 热心网友自发组织翻译

## 第 2 章 ARC 基础 (Beginning ARC)

by Matthijs Hollemans

IOS5 中最具颠覆性的变化当属自动引用计数 (Automatic Reference Counting) 的引入，缩写为 ARC。ARC 是新的 LLVM 3.0 编译器具备的特性之一，这项技术完全摒弃了让所有 IOS 开发者由爱生恨的手动内存管理。

在你的工程中使用 ARC 非常简单。你还像往常一样编程，只是不再调用 `retain`，`release` 和 `autorelease` 了。这基本上就是 ARC 的全部。

如果开启了自动引用计数，编译器就会在你程序中的恰当位置，插入 `retain`，`release` 和 `autorelease`。你不必再为这个操心，因为编译器会替你搞定。这让我震惊。实际上，ARC 的用法简单到你可以不再学习本教程了。;-)

但如果你对 ARC 还心存疑虑 -- 可能你你不信任它总是能不出错，或者你觉得比起自己做内存管理，ARC 会比较慢 -- 那么请继续。本教程剩下的部分会驱散这些迷雾，并告诉你如何处理在项目中使用时出现的一些不太直观的后果。

而且，我们会带给你如何将从未使用过 ARC 的程序转换为使用此技术的亲身体验。你可以使用同样的技术来将你现有的 ios 工程转换为使用 ARC，从而避免了成堆的内存困扰！

### 2.0. 工作原理 (How It Works)

大概你已经熟知了手动内存管理，它基本上以下面的方式工作：

- 1.如果你需要持有一个对象的话，你就需要保留(`retain`)它，除非它已经被你保留。
- 2.如果你不再使用一个对象的话，你就需要释放(`release`)它，除非它已经被你释放(使用 `autorelease`)。

作为初学者时，你大概会有一段艰难的时间，那时这个概念一直萦绕在你脑海；但是那段时间过后，这会成为你的第二天性；而现在，你总是能够正确的以 `release` 来平衡你的 `retain`，除非是你忘了。

手动管理内存的原则并不难，但却很容易犯错。这些小错误可能导致可怕的后果。要么因为你将对象释放了太多次，你的变量指向了不再有效的数据，从而导致程序崩溃；要么因为你没有充分释放对象，让他们永远存在下去，而导致内存耗尽。

xcode 的静态分析器对查找这类问题很有帮助，但 ARC 更进了一步。它通过自动为你

插入适当的 `retain` 和 `release`，从而完全避免了内存管理问题！

认识到 ARC 是 objective-c 编译器的一个特性是很重要的，因此与 ARC 相关的一切都发生在构建你的程序时。ARC 不是一个运行时特性（有一小部分例外，就是弱指针系统），它也不是你从其他语言了解的垃圾回收。

ARC 所作的只是在编译代码时向其中插入 `retain` 和 `release`，就在你会自己添加他们的地方 -- 或者至少是你应该添加他们的地方。这就使 ARC 和手动管理代码一样快速，有时候甚至更快一些，因为它私下可以执行某些优化。

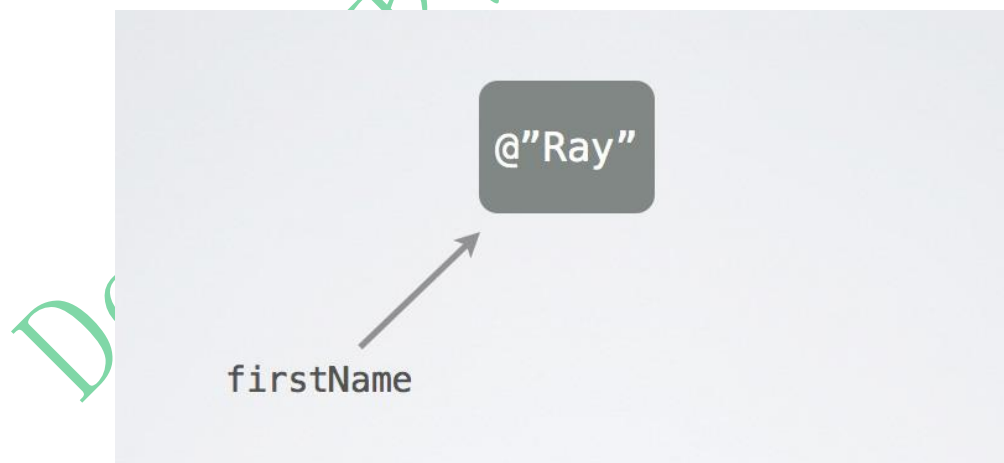
## 2.1. 指针让对象存在 (Pointers Keep Objects Alive)

你要学到的 ARC 的新规则相当简单。使用手动内存管理，你必须保留一个对象来让它存在。这不再是必要的，你要做的只是创建一个对象的指针。当指针得到一个新值或者不再存在，相关联的对象会被释放。这对于所有变量都适用：成员变量，同步属性，甚至是局部变量。

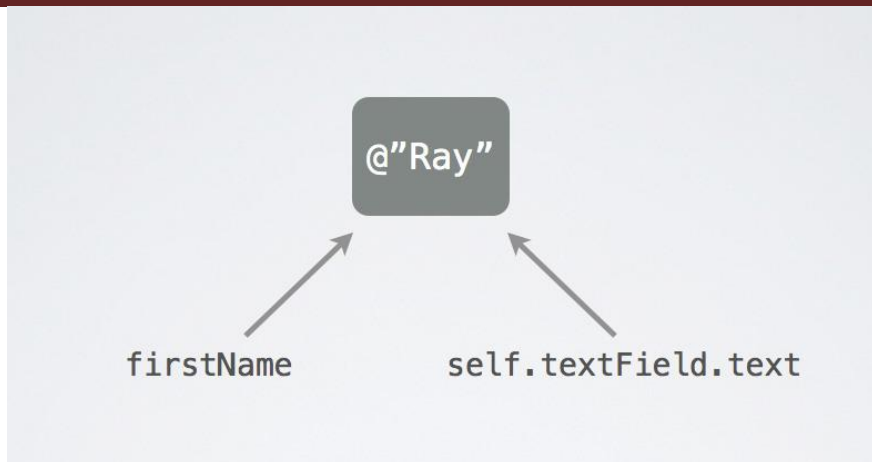
从所有权方面来考虑这点是有道理的。当你做下面的事情时，

```
NSString *firstName = self.textField.text;
```

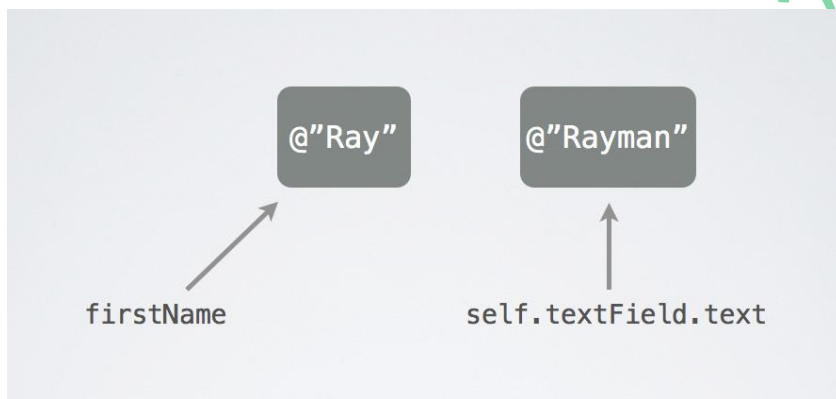
`firstName` 变量成为了指向文本框内容中的 `NSString` 对象的指针。`firstName` 变量现在是这个 `string` 对象的所有者。



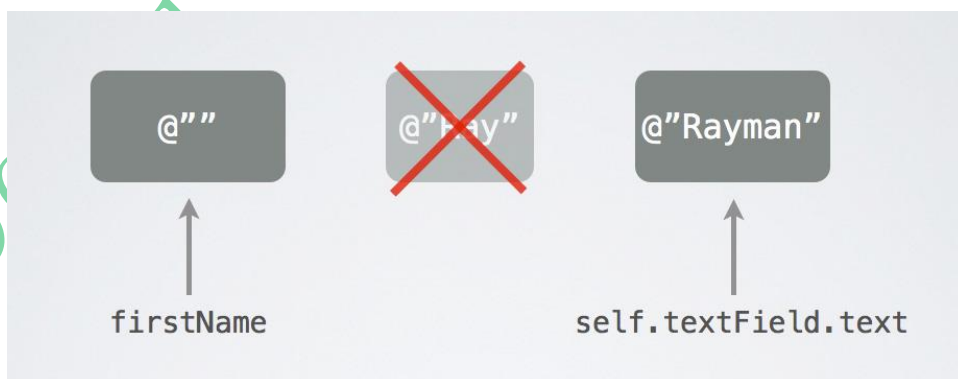
一个对象可以有多个所有者。在用户改变 `UITextField` 的内容之前，它的 `text` 属性同样是这个 `string` 对象的所有者。这样就有两个指针保持同一个对象的存在：



过了一会，用户又向文本框中输入了新的文字，文本框的 `text` 属性现在指向了一个新的 `string` 对象。原来的 `string` 对象仍然有一个所有者(`firstName` 变量)，因此存在于内存中。



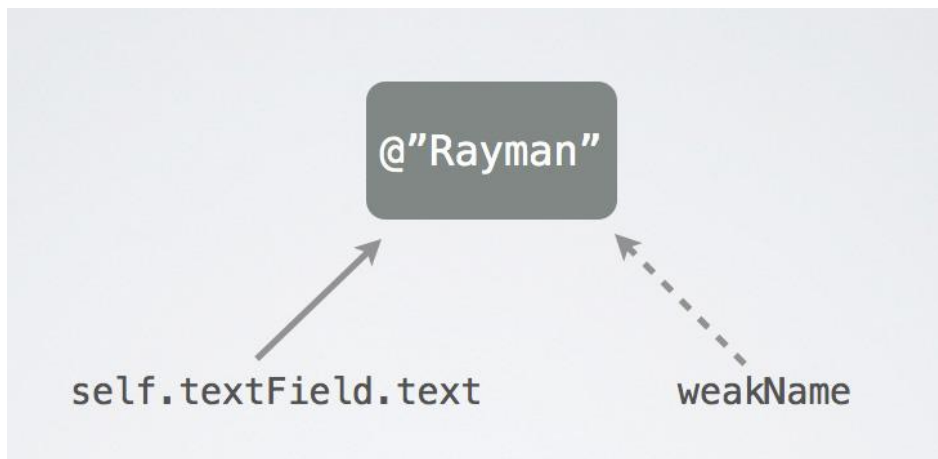
只有当 `firstName` 也得到一个新值，或者超出存在范围 -- 可能因为它是局部变量而方法结束了，或者因为它是成员变量而它所属的对象已经被释放 --- 所有权才会过期。`string` 对象不再有任何所有者，它的保留计数降至 0 从而使该对象被释放。



我们将诸如 `firstName` 和 `textField` 这类指针称为“强” (“strong”) 指针，因为他们保持了对对象的存在。默认情况下，成员变量和局部变量是强指针。

同样也存在"弱"指针("weak" pointer)。弱指针变量仍然何以指向对象，但是不再成为所有者：

```
__weak NSString *weakName = self.textField.text;
```



`weakName` 指针变量指向和 `textField.text` 属性相同的 `string` 对象，但不再是所有者。如果文本框的内容改变，该 `string` 对象不再有所有者，所以被释放了：



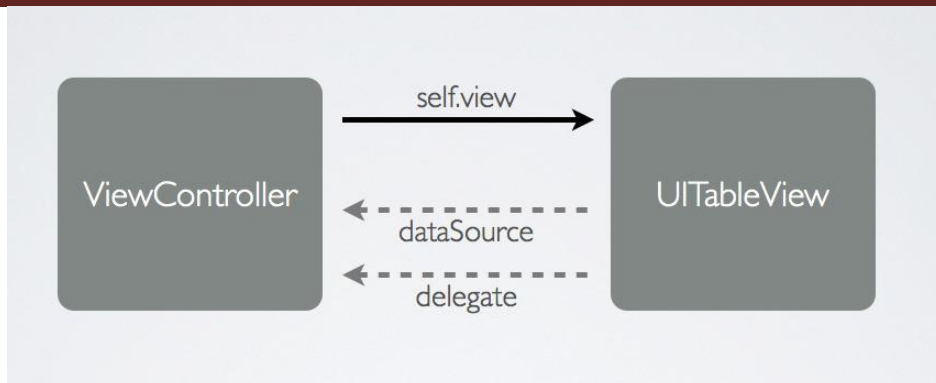
当这发生(对象被释放)时，`weakName` 的值被自动置为 `nil`。即所谓"归零"("zeroing")弱指针。

注意这是特别便利的特性，因为它防止了弱指针指向已被释放的内存。这类事情过去曾导致了大量的 BUG -- 你可能听说过"野指针" 或者 "僵尸" -- 但是感谢这些归零弱指针，那些事不再会发生！

你大概不会频繁使用弱指针。他们在两个对象是父子关系时最有用。父母会对孩子拥有强指针 --- 因此"拥有"孩子 -- 但是为了防止所有权循环，孩子仅对父母拥有弱指针。

委托模式是一个例子。你的视图控制器可能拥有一个 `UITableView` 的强指针。表格视图的数据源和委托指针指回视图控制器，但是是弱指针。我们今后会详细讨论这这一点。





注意下面的代码并不是很有用：

```
__weak NSString *str = [[NSString alloc] initWithFormat:...];  
NSLog(@"%@", str); // will output "(null)"
```

string 对象没有所有者(因为 str 是弱指针)，所以对象会在创建后立刻被释放。Xcode 在你做这件事的时候会给出一个警告因为这可能并非是你所希望发生的事情（"Warning: assigning retained object to weak variable; object will be released after assignment"）。

你可以使用\_\_strong 关键字将变量标识为强指针：

```
__strong NSString *firstName = self.textField.text;
```

但因为变量默认就是强指针，这就有点多余了。

属性也可以是强属性或弱属性，其写法如下：

```
@property (nonatomic, strong) NSString *firstName;  
@property (nonatomic, weak) id <MyDelegate> delegate;
```

ARC 很了不起，它真的会删掉大量凌乱的代码。你不必再考虑何时 retain 何时 release，只须关心对象间的相互关系。你需要问自己的问题是：谁拥有什么？

举例来说，以前，你不可能像这样写代码：

```
id obj = [array objectAtIndex:0];  
[array removeObjectAtIndex:0];  
NSLog(@"%@", obj);
```

在手动内存管理下，将对象移出数组，obj 变量的内容就不再有效。对象一旦不再是数组的成员就会被释放。用 NSLog()打印这个对象会让程序崩溃。而在 ARC 中，上面的代码行之有效。因为我们将对象放到了 obj 变量中，它是一个强指针，数组就不再是此对象的唯

一所有者了。即使将对象从数组中移除，对象依然存在，因为 `obj` 仍旧指向它。

自动引用计数也有一些限制。作为起步，ARC 只对 `objective-c` 有效。如果你的程序使用 `core foundation` 或者 `malloc()` 和 `free()`，那么你仍然对其内存管理负有责任。在本教程的后面你能看到例子。而且，某些语言规则会更加严格以确保 ARC 总能正确工作。这些仅仅是小的牺牲，你获得的好处将比你放弃的要得多得多！

仅仅是因为 ARC 为你在恰当的位置处理 `retain` 和 `release`，并不意味着你可以完全忘掉内存管理。因为强指针使对象保持存在，还是会有些情况下需要你手动将这些指针置为 `nil`，否则程序将用光其内存。如果你保留所有你曾经创建的对象，那么 ARC 也就永远不能释放他们。所以，不论你何时创建对象，你都还是需要考虑谁拥有它，以及此对象应该存在多久。

毫无疑问，ARC 是 `objective-c` 的未来。Apple 鼓励开发者从手动内存管理中调头，使用 ARC 开始编写他们的新应用。ARC 能保证更简洁的代码和更强壮的程序。有了 ARC，内存相关的崩溃就成为了历史。

但因为我们正在进入从手动到自动内存管理的过渡阶段，你经常会遇到和 ARC 不兼容的代码，不管是在你的代码中还是第三方库。幸运的是你可以将 ARC 和非 ARC 代码组合到同一个工程中，我会告诉你几种方法。

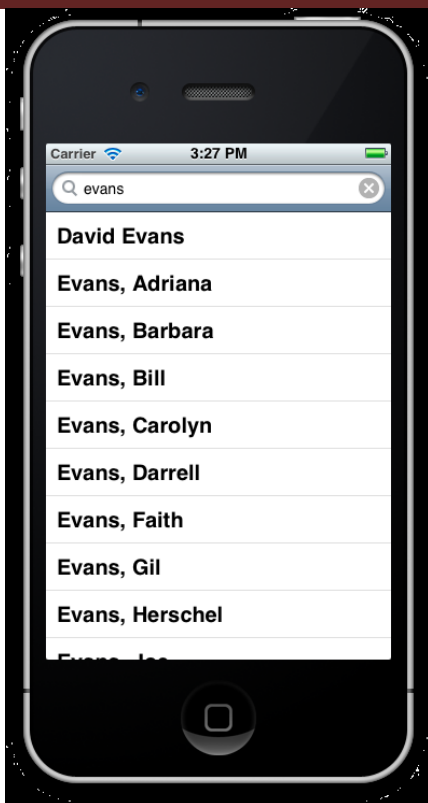
ARC 甚至能和 C++ 很好的组合。除了有一些限制之外，ARC 也可以运行于 IOS4 之上，这将只会有助于加速对于 ARC 的接受。

一个聪明的程序员会尝试尽可能自动的完成它的工作，而那就是 ARC 说提供的：自动完成之前你必须手工去做的那些不体面的编程的活。对我来说，切换到 ARC 是想都不用想的。

## 2.2. 应用 (The App)

为了演示实际如何使用自动引用计数，在此我准备了一个简单的程序，它将从手动内存管理转换到 ARC。Artists 程序包含了一个带有表格视图和搜索条的单个屏幕。当你在搜索条中输入一些东西时，程序调用 MusicBrainz API 来搜索符合名字的音乐家。

程序看上去是这样的：



用他们的话说，MusicBrainz 是"一个开放的音乐百科全书，它搜集音乐数据并对公众开放。"他们有一个免费的 XML 服务，你可以在你的程序中用到它。想了解更多关于 MusicBrainz 的信息，可以访问他们的网站 <http://musicbrainz.org>。

你可以在本章的文件夹下找到 Artists 程序的起始代码。该工程包含以下源码文件：

- **AppDelegate.h/.m:** 应用程序委托。没什么特别，每个程序都有一个。它加载视图控制器并将其安放到窗口中。
- **MainViewController.h/.m/.xib:** 程序的视图控制器。它有表格视图和搜索条，并且完成大部分的工作。
- **SoundEffect.h/.m:** 一个用来播放音效的简单类。程序将在 MusicBrainz 搜索完成时发出微弱的蜂鸣声。
- **main.m:** 程序的入口点。

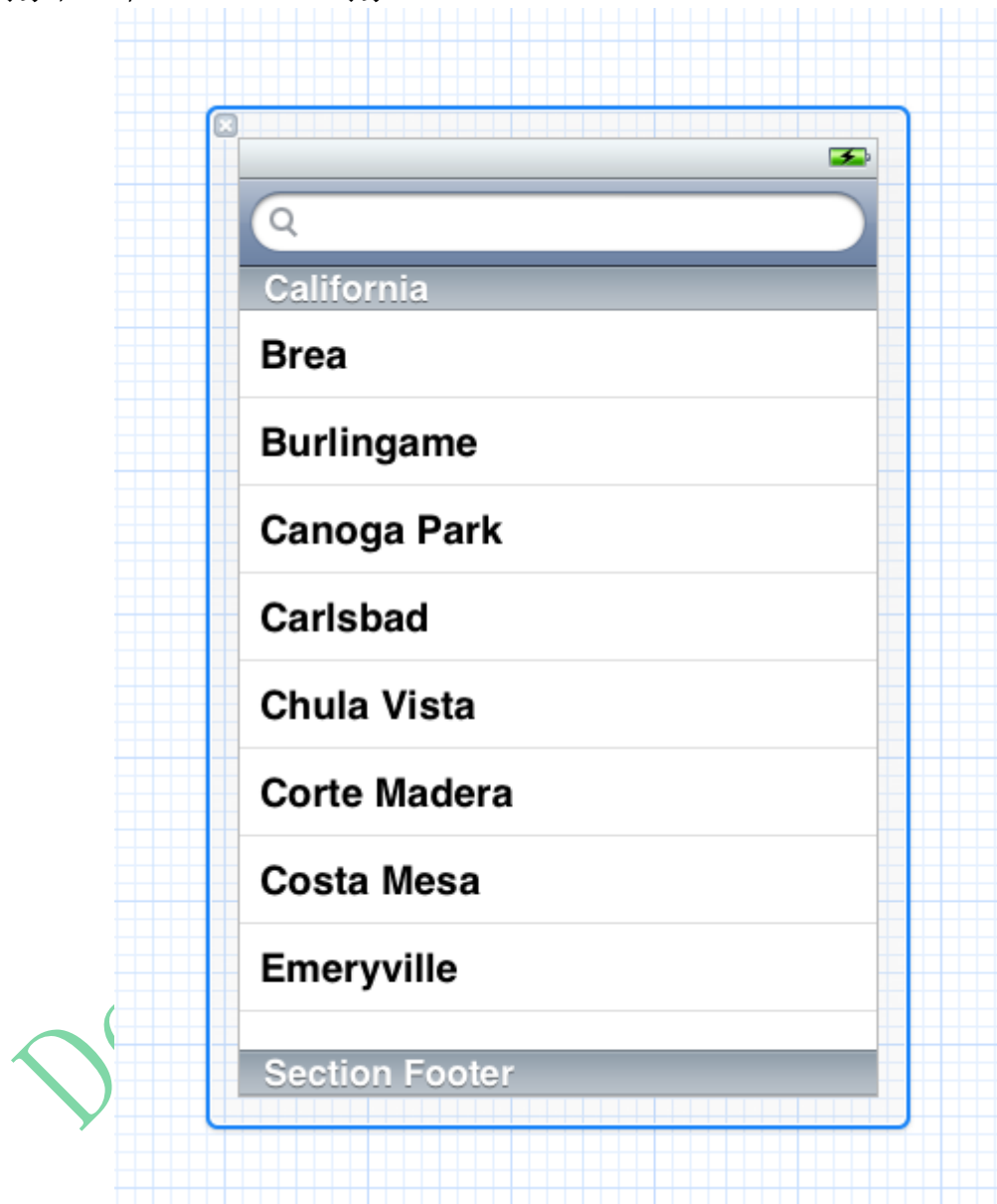
不仅如此，程序还使用了第三方库。你的程序大概使用了一些他们自己的外部组件，学习如何让这些库同 ARC 和谐共存是件好事。

- **AFHTTPRequestOperation.h/.m:** AFNetworking 库的一部分，它使对 web service 的请求更易于执行。因为我们只需要这一个类，所以没有包含完整的库，你可以在这里找到完整的包：

<https://github.com/gowalla/AFNetworking>

- **SVProgressHUD.h/.m/.bundle**: 一个会在搜索时显示于屏幕上进度指示器。你以前可能没见过, bundle 文件。这是一个特殊类型的文件夹, 它包含了 **SVProgressHUD** 要用到的图片文件。要查看这些文件, 可以右键点击 .bundle 文件, 选择“查看包内容” (Show Package Contents) 菜单选项。更多关于这个组件的信息, 参见 <https://github.com/samvermette/SVProgressHUD>

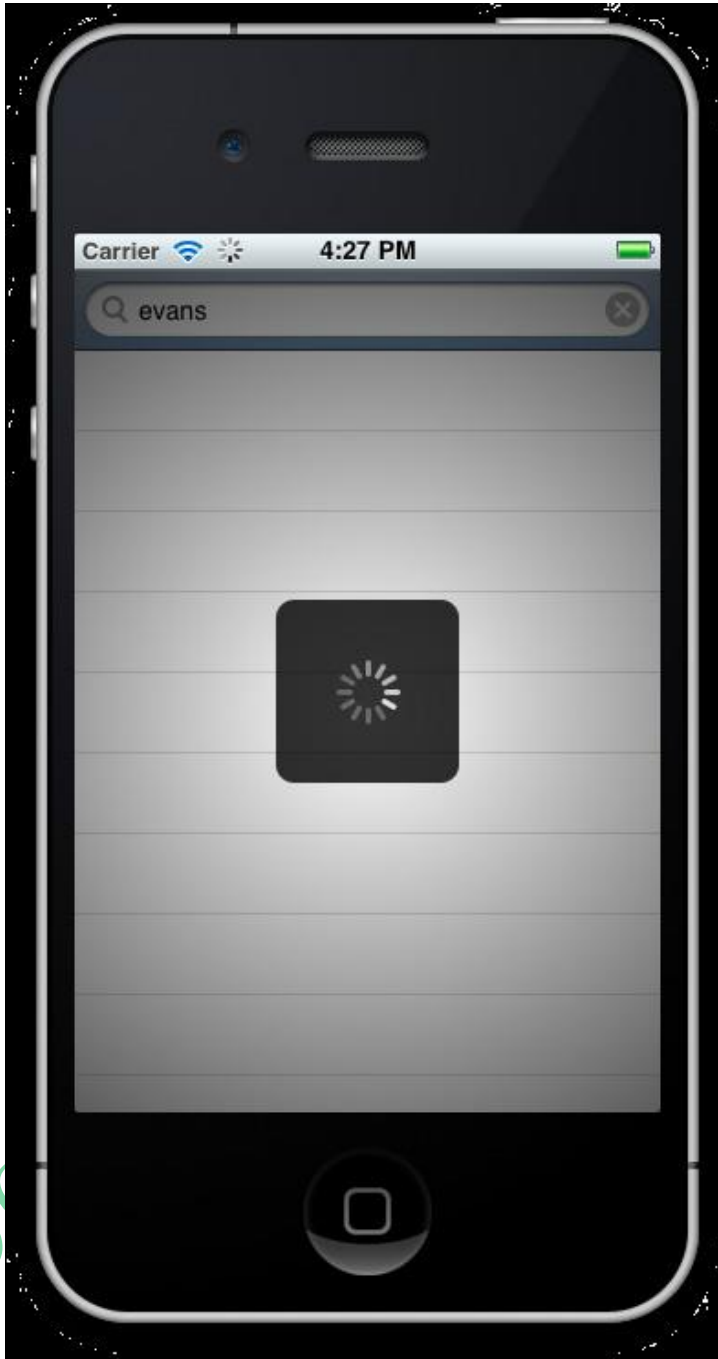
让我们快速浏览一下视图控制器的代码, 这样你能对程序如何运行有一个好的概念。**MainViewController** 是 **UIViewController** 的一个子类。它的 nib 文件包含了一个 **UITableView** 对象和一个 **UISearchBar** 对象:



表格视图显示了 **searchResults** 数组的内容。最初的时候这个指针为 **nil**。当用户执行了一个搜索时, 我们将用 **MusicBrainz** 服务器的响应数据来填充这个数组。如果没有搜索结果, 数组就是空的 (但不是 **nil**) 并且表格显示: "(Nothing found)". 这些都发生在 **UITableViewDataSource** 的常用方法 **numberOfRowsInSection** 和 **cellForRowAtIndexPath** 中:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)theSearchBar
{
    [SVProgressHUD showInView:self.view status:nil
networkIndicator:YES posY:-1 maskType:SVProgressHUDMaskTypeGradient];
}
```

首先，我们创建一个新的 HUD，将其显示在表格视图和搜索条之上，在网络请求完成前阻止任何的用户输入：



然后我们创建 http 请求的 URL。我们使用 MusicBrainz API 来搜索艺术家。

```
NSString *urlString = [NSString stringWithFormat: @"http://musicbrainz.org/ws/2/artist?query=artist:%@&limit=20", [self
escape:searchBar.text]];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:[NSURL URLWithString:urlString]];
```

使用 `escape:` 方法对搜索文本进行了 URL 编码，确保我们得到的是有效的 URL。空格和其他特殊字符被转换成了类似 `%20` 的形式：

```
NSDictionary *headers = [NSDictionary dictionaryWithObject: [self userAgent] forKey:@"User-Agent"];
[request setAllHTTPHeaderFields:headers];
```

我们向 HTTP 请求添加了一个 User-Agent 报头。MusicBrainz API 需要这个。所有的请求应该“有一个适当的 User-Agent 报头来标识发起请求的程序和该程序的版本。”好好遵守你使用的 API 的规定总会是个好注意，所以我们构造了一个 User-Agent 报头如下：

```
com.yourcompany.Artists/1.0 (unknown, iPhone OS 5.0, iPhone Simulator, Scale/1.000000)
```

（我从 AFNetworking 库的其他部分找到了这个公式并将其放到视图控制器的 `userAgent` 方法中。）

MusicBrainz API 还有一些其他的限制。客户端程序每秒对 web service 的请求不能多于一次，否则他们会冒 IP 被封的风险。这对我们的应用来说算不了什么大事 -- 用户不大可能做那么多次搜索 -- 所以我们对此不做什么特别的预防措施。

一旦我们构造了 `NSMutableURLRequest` 对象，就将它给 `AFHTTPRequestOperation` 去执行：

```
AFHTTPRequestOperation *operation = [AFHTTPRequestOperation operationWithRequest:request completion:^(NSURLRequest
*request, NSHTTPURLResponse *response, NSData *data, NSError *error)
{
    // ...
}];
[queue addOperation:operation];
```

`AFHTTPRequestOperation` 是 `NSOperation` 的一个子类，这就是说我们可以把它加到一个 `NSOperationQueue`（在 `queue` 变量中），这样请求就能被异步的处理。因为有 HUD，程序在请求发生时忽略任何的用户输入。

我们为 `AFHTTPRequestOperation` 提供一个代码块(block)，它在请求完成时被调用。在代码块中，首先我们会检查请求是否成功(HTTP 状态码 200)。对于本程序我们并不是特别关心请求为什么失败；如果失败了我们只要告诉 HUD 显示一个特别的"error"动画然后消失。注意请求完成的代码块不一定会在主线程执行，所以，我们需要在 `dispatch_async()`中调用 `SVProgressHUD`。

```
if (response.statusCode == 200 && data != nil)
{
    ...
}
else // something went wrong
```



```
{
dispatch_async(dispatch_get_main_queue(), ^
{
[SVProgressHUD dismissWithError:@"Error"];
}); }
```

现在是最有意思的部分。如果请求成功，我们会分配 `searchResults` 数组并解析响应数据。因为响应数据是 XML，所以我们用 `NSXMLParser` 来解析。

```
self.searchResults = [NSMutableArray arrayWithCapacity:10];
NSXMLParser *parser = [[NSXMLParser alloc] initWithData:data];
[parser setDelegate:self]; [parser parse];
[parser release];
[self.searchResults sortUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
```

你可以在 `NSXMLParserDelegate` 的方法中查看 XML 解析的逻辑，但事实上我们只查找名为 "sort-name" 的元素。其中包含了艺术家的名字。我们将这些名字作为 `NSString` 对象保存到 `searchResults` 数组中。当 XML 解析完成时，我们将结果按照字母表排序，然后在主线程中更新屏幕：

```
dispatch_async(dispatch_get_main_queue(), ^
{
[self.soundEffect play];
[self.tableView reloadData];
[SVProgressHUD dismiss];
});
```

这就是本程序的工作原理。他使用了手动内存管理，并没有使用任何 `IOS5` 的特性。现在就让我们将它转化为 ARC 吧。

## 2.3. 自动转化 (Automatic Conversion)

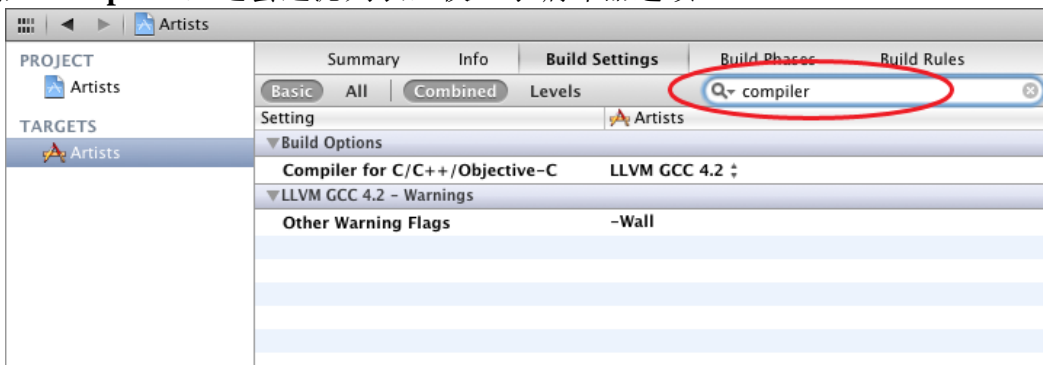
现在，我们要将 `Artist` 程序转化到 ARC，这基本上表示我们会干掉所有对于 `retain`, `release` 和 `autorelease` 的调用，但是还是有些需要特别注意的地方：

- 1.xcode 有一个自动转化工具，能够转换你的代码。
- 2.你可以手工转化这些文件。
- 3.你可以对你不希望转化的源文件禁用 ARC。这对于你不想修改的第三方库很有用。

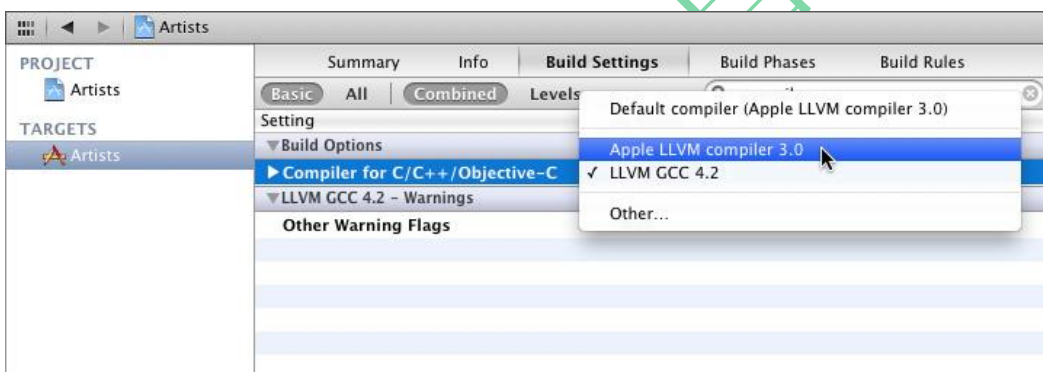
我们会对 `Artist` 程序使用所有这些选项，这样能让你了解它们都是怎么运用的。在本部分，我们将使用 xcode 的自动转化工具，来转化除了 `MainViewController` 和 `AFHTTPRequestOperation` 之外的源文件。

在进行转化之前，因为工具会覆盖原始文件，所以你应该拷贝一份工程代码作为备份。尽管 xcode 提供了源代码快照功能，但是作为预防措施，我还是创建了一个备份。

ARC 是新的 LLVM 3.0 编译器的一个特性。你的现有工程很可能使用的是早先的 gcc 4.2 或者 LLVM-GCC 编译器，所以你应该首先将工程切换到新编译器，看看在非 ARC 模式下是否存在错误。转到 **Project Settings** 屏幕，选中 **Artists target**，在 **Build Setting** 的搜索框中输入 "**compiler**"，这会过滤列表，仅显示编译器选项：

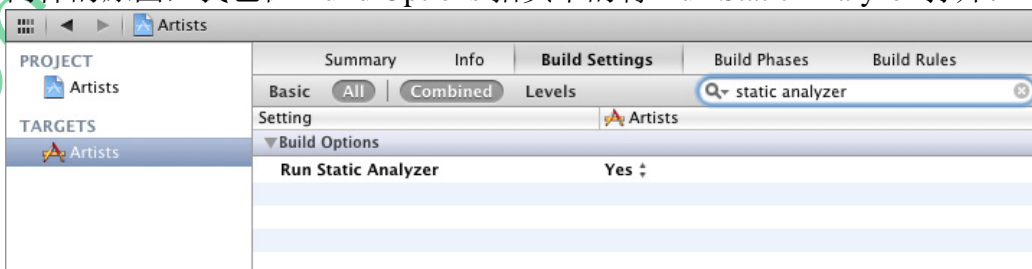


修改 "Compiler for C/C++/Objective-C" 选项，选择 Apple LLVM compiler 3.0:



在 warning 抬头部分，还要将 "Other Warning Flags" 设置为 -Wall。编译器现在会检查所有可能导致问题的状况。默认情况下，大多数这类警告都被关掉了，但是我发现总是将他们当作致命错误是很有用的。也就是说，编译器发出一个警告后，我必须先搞定它才能继续。你是否也想对你的工程采取同样的措施，完全取决于你，但是在转化到 ARC 的过程中，我建议你好对待编译器提示的任何事情。

出于同样的原因，我也在 Build Options 抬头下的将 Run Static Analyzer 打开：



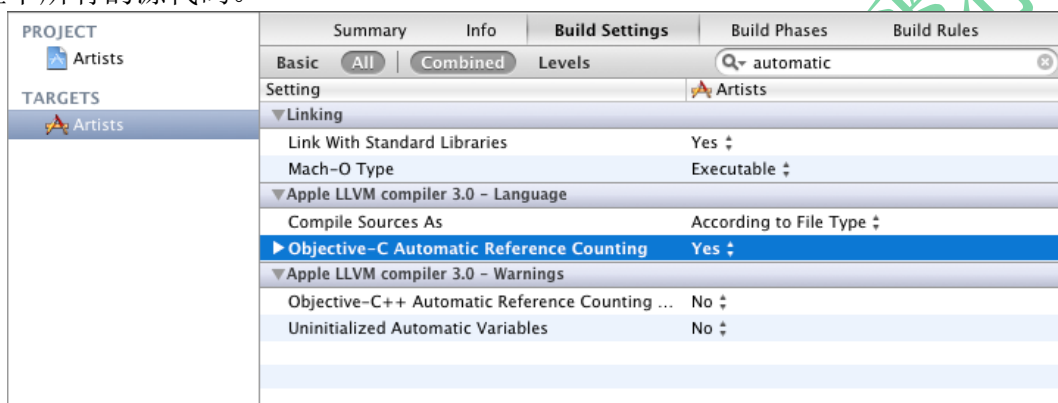
Xcode 现在会在每次构建程序时运行分析器了。这会让构建过程慢一点，但对于这么小的程序来说基本察觉不到。



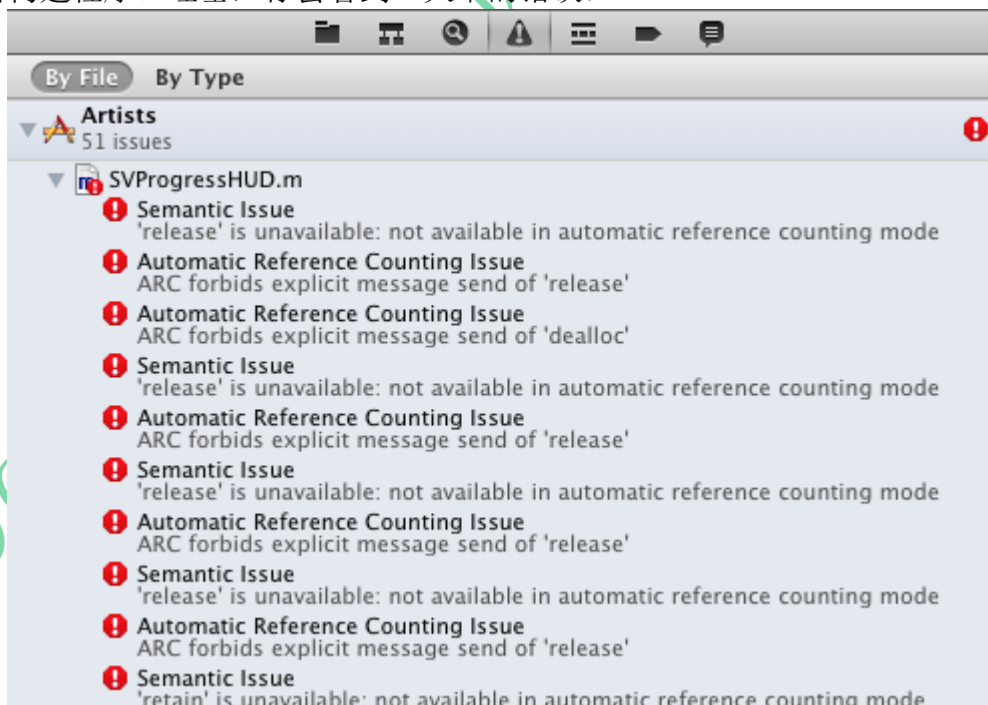
现在让我们构建这个程序，看看它是否在新编译器下存在问题。首先使用 **Product -> Clean** 菜单项(或者 **Shift-Cmd-K**)清理一下工程。然后按下 **Cmd-B** 来构建程序。Xcode 应该不会给出错误或者警告，情况不错。如果你在转化你自己的程序，并且你得到了警告，那么现在就解决他们吧。

让我们玩一把，将编译器切换到 **ARC** 模式然后再次构建程序。我们会得到数不清的错误消息，但是研究一下这些错误到底是什么。

还是在 **Build Settings** 屏幕中，切换到**All**来查看所有可用的设置项(和 **Basic** 不一样，**Basic** 仅仅显示常用设置项)。搜索"automatic"并设置"Objective-C Automatic Reference Counting"选项为 **YES**。这是一个工程范围的标志，它告诉 xcode 想使用 **ARC** 编译器编译工程中所有的源代码。

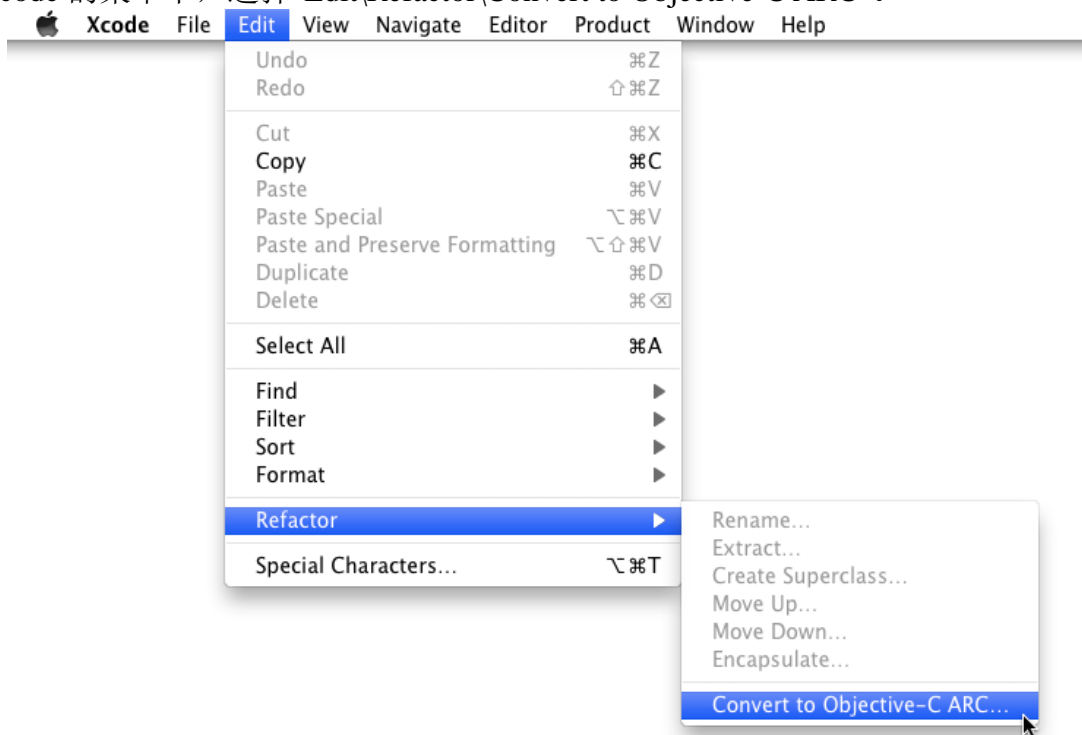


重新构建程序。哇塞，你会看到一大堆的错误：

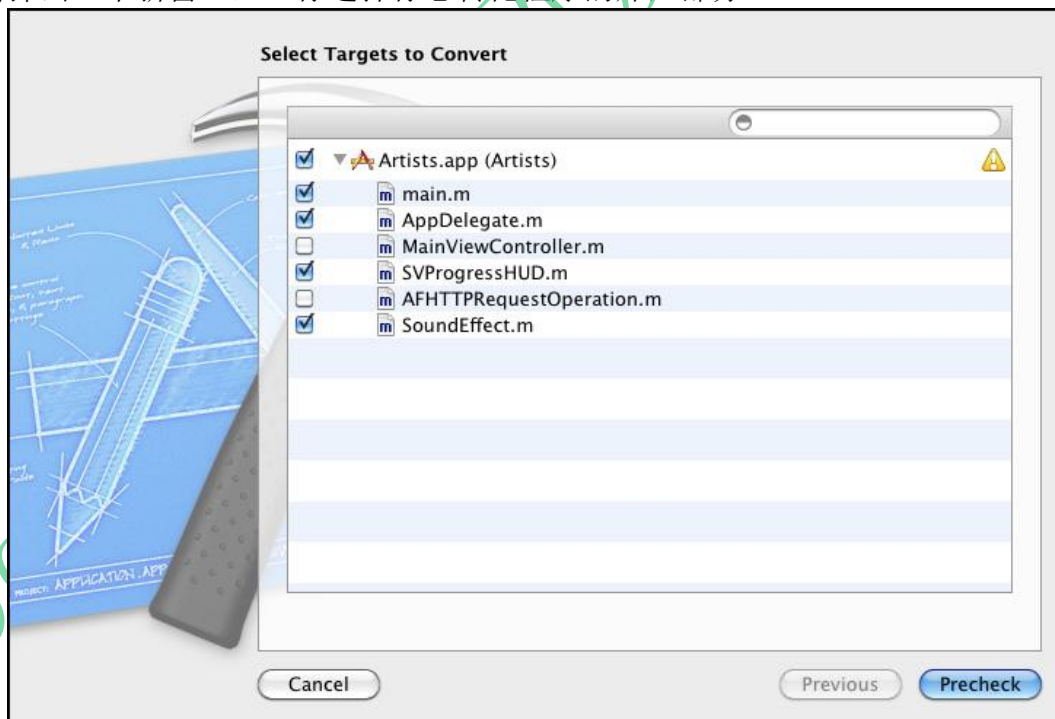


很清楚你需要做些迁移的工作！大多数这类错误很明显，它们显示你不能继续使用 **retain**, **release** 和 **autorelease** 了。我们可以手工改掉所有的错误，但是用自动化工具要简单得多。这个工具将以 **ARC** 模式编译程序，并重写它在源码中碰到的每个错误，直到它完全无错的编译。

从 xcode 的菜单中, 选择"Edit\Refactor\Convert to Objective-C ARC".



这会弹出一个新窗口, 让你选择你想转化程序的那些部分:

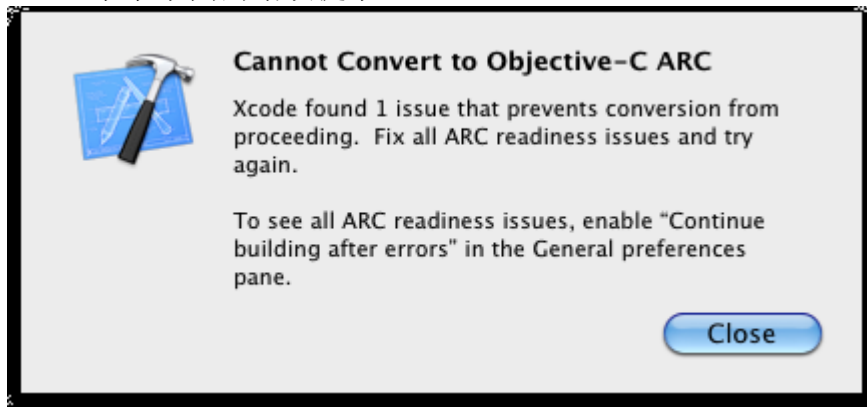


- main.m
- AppDelegate.m
- SVProgressHUD.m
- SoundEffect.m

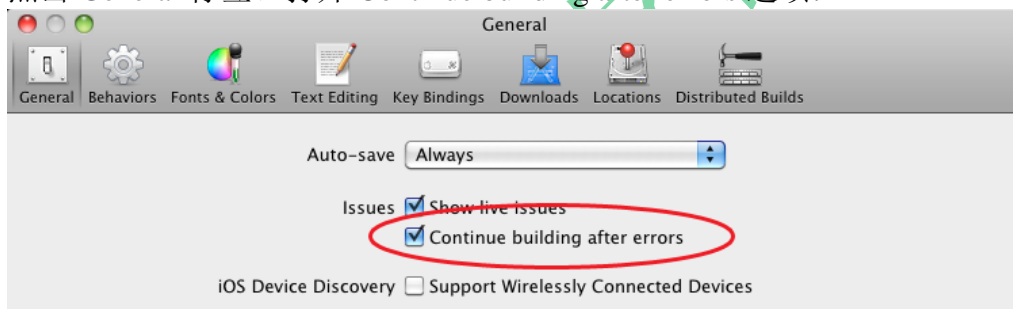
对话框上显示了一个小的警告图标, 指出工程已经使用了 ARC。这是因为我们之前在

Build Setting 中打开了 Objective-c 的自动引用计数，所以转化工具把它当作了一个 ARC 工程。你可以无视这个警告，他不会影响到此次转化。

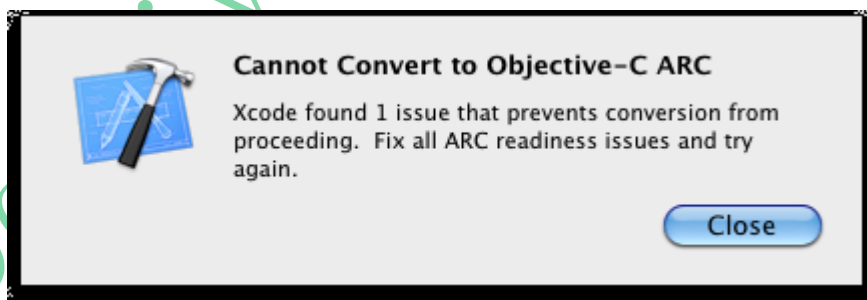
按下 Precheck 按钮开始转化。此工具会先检查你的代码是否处于足够好的状态，以执行到 ARC 的转化。我们确实曾使工程在新的 LLVM 3.0 编译器下构建成功，但是显然还不够好。xcode 显示了下面的错误提示：



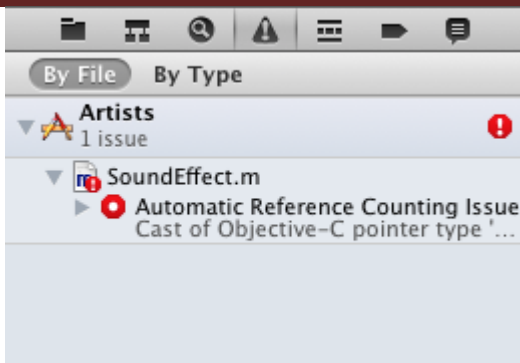
它提示"ARC 准备就绪问题(ARC readiness issues)"，并要求我们打开"Continue building after errors"选项。我们先做后面一件事。打开 Xcode 的 preferences 窗口（在 xcode 下的菜单项中），点击 General 标签。打开 Continue building after errors 选项：



让我们再试一次。选择 Edit\Refactor\Convert to Objective-C ARC，并且选中除 MainViewController.m 和 AFHTTPRequestOperation.m 之外的所有源文件，按下 Precheck 开始。



不幸的是，我们再次得到了一条错误消息。和以前不同，这次编译器能够识别转化前需要修复的所有问题。幸运的是，只有一个错误：

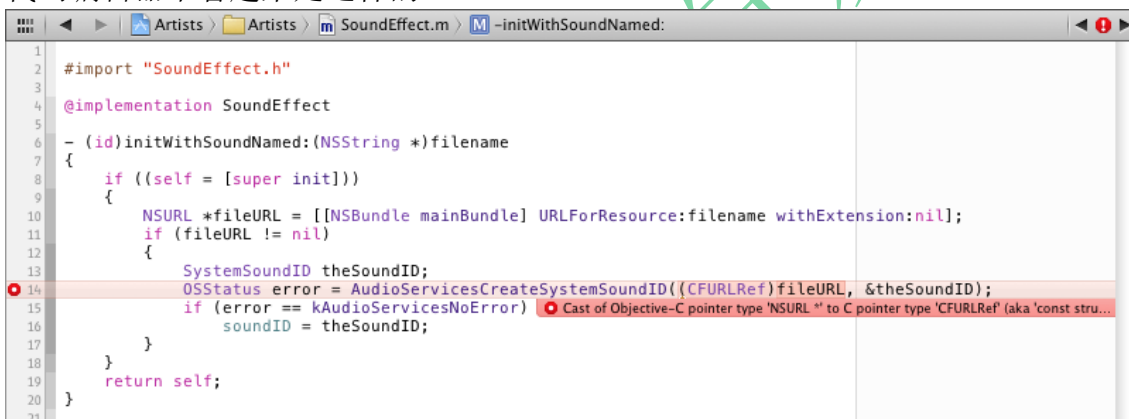


(你可能在这个列表中显示了更多的错误。有时转化工具所提示的并不真的是"ARC readiness" 问题。)

问题的完整描述是:

Cast of Objective-C pointer type 'NSURL \*' to C pointer type 'CFURLRef' (aka 'const struct \_\_CFURL \*') requires a bridged cast

它在代码编辑器中看起来是这样的:



我们会在后面详细讨论这个问题，但现在源代码尝试将一个 NSURL 对象转换成一个 CFURLRef 对象。AudioServicesCreateSystemSoundID()函数接受一个 CFURLRef 参数，此参数描述了声音文件的位置，但是我们使用了 NSURL 代替了它。CFURLRef 和 NSURL 是"免费桥接(toll-free bridged)"的，这让 NSURL 和 CFURLRef 可以互换使用。

一般说来，iOS 中基于 C 的 API 会使用 core foundation 对象(CF 就代表它)，而基于 Objective-C 的 API 使用由 NSObject 扩展而来的"真正的"对象。有时候你需要在这二者之间进行转换，而这是免费桥接技术所允许的。

但是，当你使用 ARC 的时候，编译器需要知道它应该对这些免费桥接对象做什么。如果你用 NSURL 代替一个 CFURLRef，那在一天结束的时候谁来释放它的内存呢？为了解决这个难题，Apple 引入了一组关键字：\_\_bridge, \_\_bridge\_transfer 和 \_\_bridge\_retained。我们在本教程的后面会深入探讨这个问题。

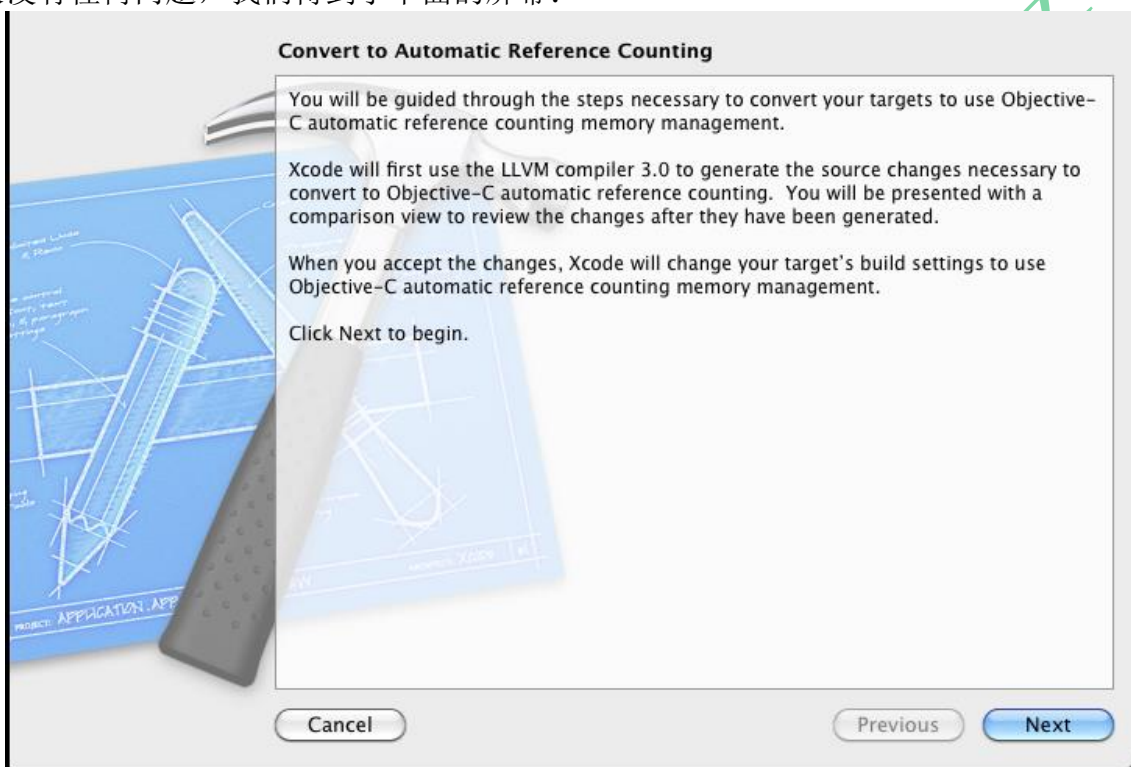
现在，我们需要将源代码改成这样:

OSStatus error = AudioServicesCreateSystemSoundID((\_\_bridge CFURLRef) fileURL, &theSoundID);

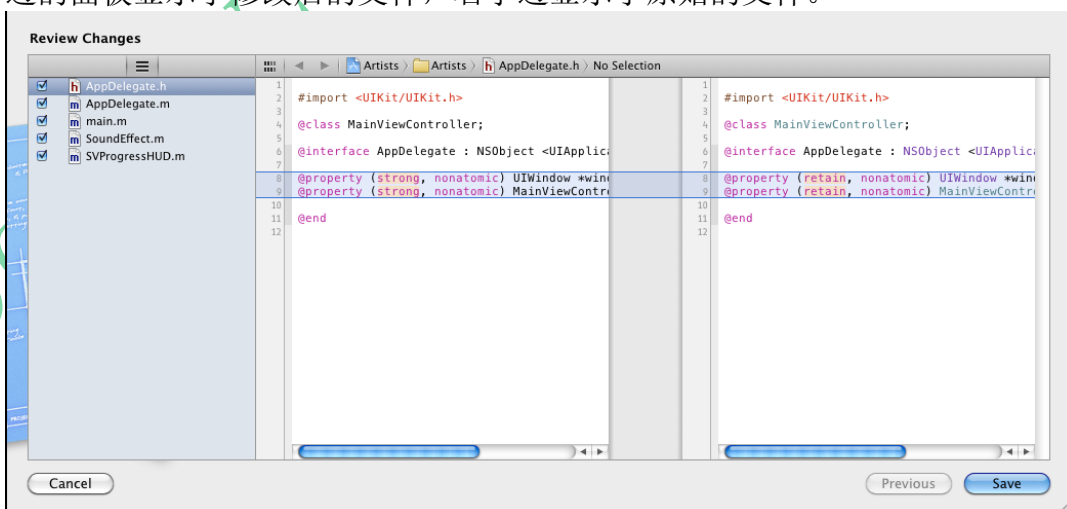
我们在 CFURLRef 转换中加入了\_\_bridge 关键字。

"预检查(pre-check)"可能不仅仅给你这一个错误。你可以安全的忽视其他错误，上面对 SoundEffect.m 的修改是我们唯一需要处理的地方。转换工具有时对"ARC readiness issue"有点不太确定。

让我们再次运行转化工具 - Edit\Refactor\Convert to Objective-C ARC。这一次 pre-check 运行结果没有任何问题，我们得到了下面的屏幕：



点击 Next 继续。几秒钟之后，Xcode 会显示所有被修改的文件，而且显示了在哪被修改。左手边的面板显示了修改后的文件，右手边显示了原始的文件。



对这些修改都浏览一遍总是个好主意，这样可以避免 xcode 没有帮倒忙。让我们浏览一遍转化工具建议修改的地方吧。



## AppDelegate.h

```
@property (strong, nonatomic) UIWindow *window;  
@property (strong, nonatomic) MainViewController *viewController;
```

应用程序委托有两个属性，窗口和主视图控制器。本程序中并没有使用 MainWindow.xib 文件，所以这两个对象是由 AppDelegate 的 application:didFinishLaunchingWithOptions: 所创建，并保存到属性中，以此简化内存管理。

属性的声名原来是：

```
@property (retain, nonatomic)
```

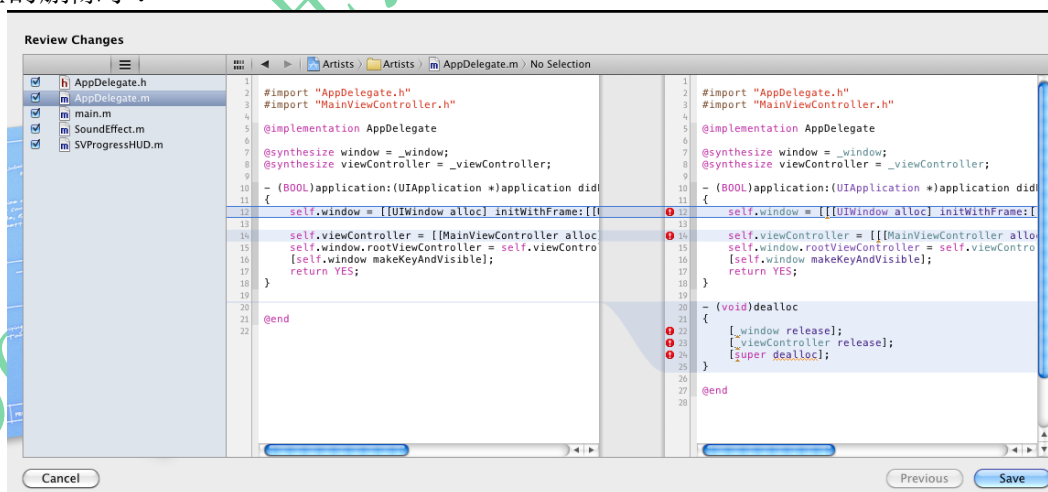
被改为：

```
@property (strong, nonatomic)
```

strong 关键字代表你的意图。它告诉 ARC 属性背后被同步的 ivar 拥有对对象的强引用。也就是说，window 属性包含了一个指向 UIWindow 对象的指针，同时它也成为了 UIWindow 对象的所有者。只要 window 属性还保存着 UIWindow 对象的值，它（UIWindow 对象）就仍然存在于内存中。这个机制同样适用于 viewController 属性和 MainViewController 对象。

## 2.4. AppDelegate.m

在 AppDelegate.m 中，创建窗口和视图控制器对象的代码行被修改了，并且 dealloc 方法被完全的删除了：



没错，不必再调用 `autorelease` 了。创建视图控制器的代码行也一样。

```
self.viewController = [[[MainViewController alloc] initWithNibName: @"MainViewController" bundle:nil] autorelease];
```

现在变成:

```
self.viewController = [[MainViewController alloc] initWithNibName: @"MainViewController" bundle:nil];
```

在 ARC 以前，如果属性被声名为 `retain`，下面的代码会导致内存泄漏：

```
self.someProperty = [[SomeClass alloc] init];
```

`init` 方法返回一个被保留的对象，而将其赋给这个属性会再次保留它。这就是你为什么要用 `autorelease`，为了平衡 `init` 方法中的 `retain`。但是用 ARC 的话，上面的代码是没问题的。编译器足够聪明，它知道这里不应该做两次 `retain`。

我喜爱 ARC 的特性之一是，在大多数情况下，你都不必写 `dealloc` 方法。当一个对象被释放时，它的实例变量和同步属性自动被释放，你不再需要像这样写：

```
- (void)dealloc  
{  
    [_window release];  
    [_viewController release];  
    [super dealloc];  
}
```

因为 `objective-c` 现在会自动处理这些。事实上，甚至没有可能再像上面这样写了。在 ARC 中，你被禁止使用 `release` 和 `[super dealloc]`。你仍然可以实现 `dealloc` -- 后面有例子 -- 但是不再需要你手动释放成员变量了。

转化工具没有做的事情是将 `AppDelegate` 作为 `UIResponder` 的子类，而不是 `NSObject`。当你使用 `xcode` 的模版创建新程序时，`AppDelegate` 类现在是一个 `UIResponder` 的子类了。让他的父类保持为 `NSObject` 没有什么不好，但是如果你想将他改成 `UIResponder` 也可以：

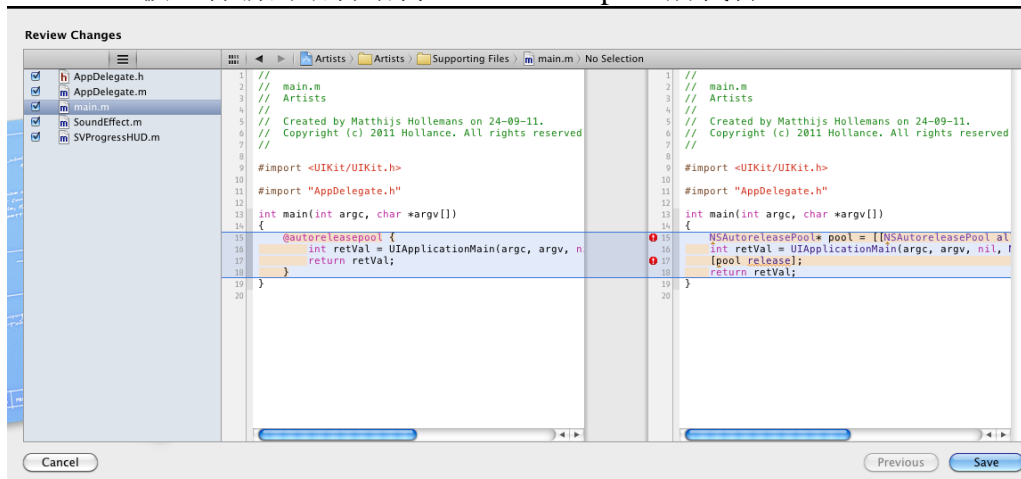
```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

## 2.5. Main.m

在手动内存管理的程序中，`[autorelease]`方法和“自动释放池”紧密协同工作，自动释放池由 `NSAutoreleasePool` 对象表示。即使是 `main.m` 中也有一个(自动释放池)，如果你直接使用线程的话，你还需要为每个线程创建一个 `NSAutoreleasePool`。有时开发者还会将 `NSAutoreleasePool` 放到做大量处理的循环中，以确保单个循环所创建的自动释放对象不会

占用太多内存，并不时的被删除。

ARC 中仍旧使用自动释放，即使你不再直接为对象调用[autorelease]方法了。只要你从除了 alloc, init, copy, mutableCopy 或 new 这些方法之外的方法返回一个对象，ARC 编译器会为你自动释放他们。这些对象仍然在自动释放池中被干掉。最大的不同是 NSAutoreleasePool 被一种新的语言结构 @autoreleasepool 所代替。



转化工具将我们的 main()函数:

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
int retVal = UIApplicationMain(argc, argv, nil,
NSStringFromClass([AppDelegate class]));
[pool release];
return retVal;
```

转化为:

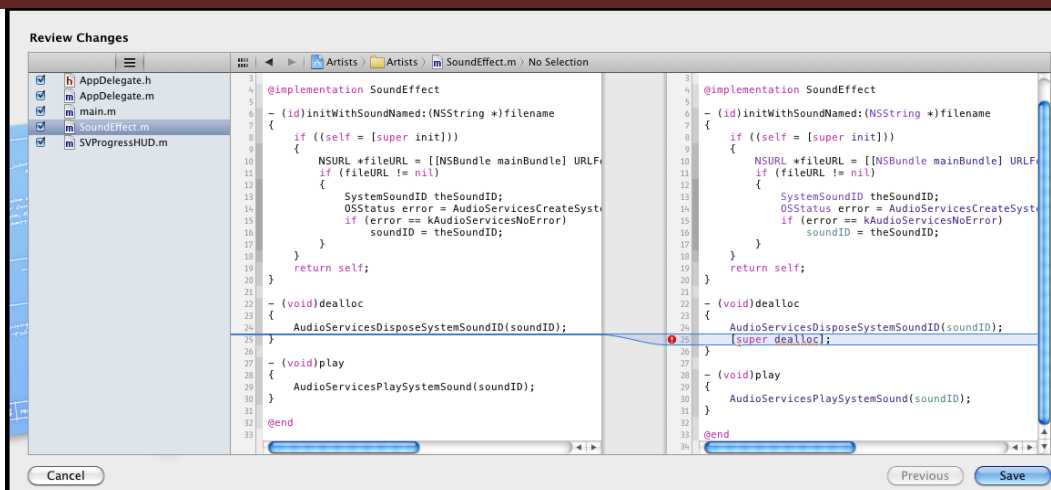
```
@autoreleasepool {
int retVal = UIApplicationMain(argc, argv, nil,
NSStringFromClass([AppDelegate class]));
return retVal; }
```

现在的语法不仅更易于阅读，其底层的实现也有很大的变化，从而使新的自动释放池比老的要快很多。你基本上可以不用担心 ARC 中的 autorelease，除非你在以前的代码中使用了 NSAutoreleasePool，你就需要代之以 @autoreleasepool 语句块。转化工具会自动替你完成，就像它在这里做的那样。

## 2.6. SoundEffect.m

除了对[super dealloc]的调用被删除之外，这个文件变化不大。你不再允许在 dealloc 方法中调用 super 了。

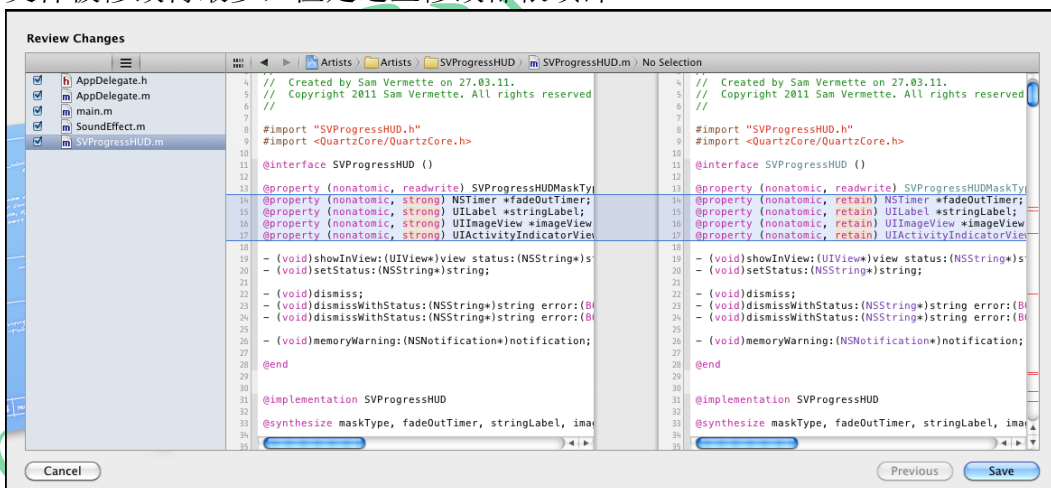




注意这里 dealloc 方法还是必要的。大多数情况下，你可以忘掉 dealloc，编译器会帮你搞定。但是有时，你还是需要手动释放资源。这个类就是这种情况。当 SoundEffect 对象被析构时，我们仍然需要调用 AudioServicesDisposeSystemSoundID() 来清理声音对象，dealloc 就是调用它的绝佳位置。

## 2.7. SVProgressHUD.m

这个文件被修改得最多，但是这些修改都很琐碎。



在 SVProgressHUD.m 的顶部，你会发现被称为“类扩展”的@interface SVProgressHUD()，它有一些属性声明。如果你对类扩展不熟悉的话，可以将他们看做范畴(categories)，除了它们有些特别的能力。类扩展的声明看起来像范畴，但是在()括号中没有名字。类扩展可以有属性和成员变量，这是范畴所不能做的，但你只能在你的.m 文件中使用它们。（也就是说，你不能在别人的类中使用它）

类扩展很酷的地方在于，它允许你为类添加私有属性和方法名。如果你不希望将某些属性或方法暴露在公共的@interface 中的话，你可以用类扩展。这就是 SVProgressHUD 的作者

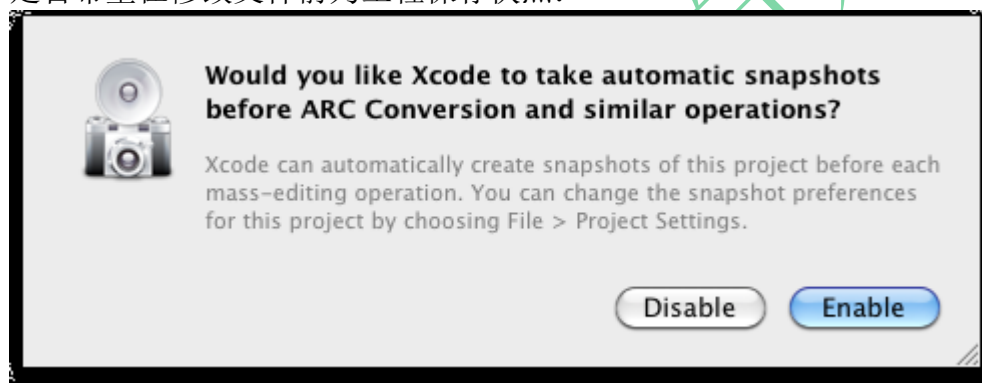
干的事情。

```
@interface SVProgressHUD ()
...
@property (nonatomic, strong) NSTimer *fadeOutTimer;
@property (nonatomic, strong) UILabel *stringLabel;
@property (nonatomic, strong) UIImageView *imageView;
@property (nonatomic, strong) UIActivityIndicatorView *spinnerView;
...
@end
```

就像我们之前看到的，retain 属性会变成 strong 属性。如果你滚动预览窗口，你会发现所有其他的修改只是简单的将 retain 和 release 语句去掉了。

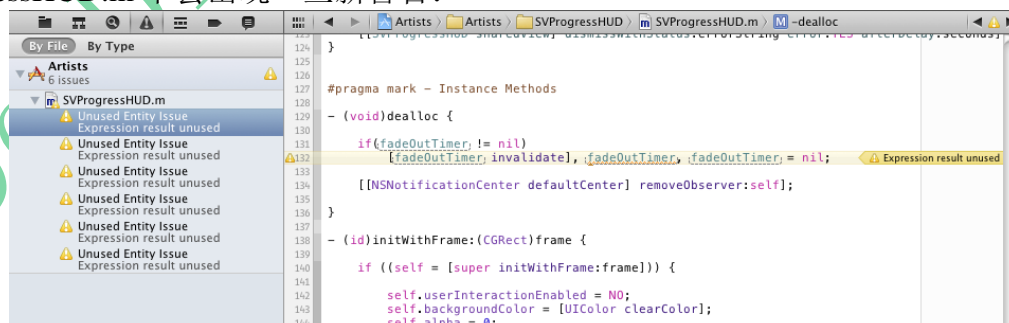
## 2.8. 实战 (Doing It For Real)

当你对转化工具所作的修改感到满意后，按下 Save 按钮来让转化执行。Xcode 会首先询问你，是否希望在修改文件前为工程保存快照：



在此，如果你需要恢复之前的代码，你应该按下 Enable，你可以在 Project 下的 Organizer 窗口中找到快照。

在 ARC 转化工具完成后，按下 Cmd+B 构建程序。构建应该会成功，但是 SVProgressHUD.m 中会出现一些新警告：



注意到这个类仍然使用了 dealloc 方法，这是为了停止计时器，以及从 NotificationCenter 反注册通知。显然，这不是 ARC 会为你做的事。

警告所对应的代码可能曾经是这个样子：

```
if(fadeOutTimer != nil)
[fadeOutTimer invalidate], [fadeOutTimer release], fadeOutTimer = nil;
```

现在改为了：

```
if(fadeOutTimer != nil)
[fadeOutTimer invalidate], fadeOutTimer, fadeOutTimer = nil;
```

转化工具确实删除了对[release]的调用，但是将变量留在了原来的位置。只有变量名的语句没有任何作用，因此 xcode 发出警告，这发生在自动转化工具不能预知的情况下。

如果你对逗号感到困惑，那么你要知道，在 `objectiv-c` 中使用逗号将多个表达式放到一行中是合法的。上面的小技巧是一个常见的惯用法，用来释放对象并将相应的成员变量置为 `nil`。因为都在同一行中，所以 `if` 不需要用大括号了。

为了去掉警告，你可以将此行以及类似的代码改为：

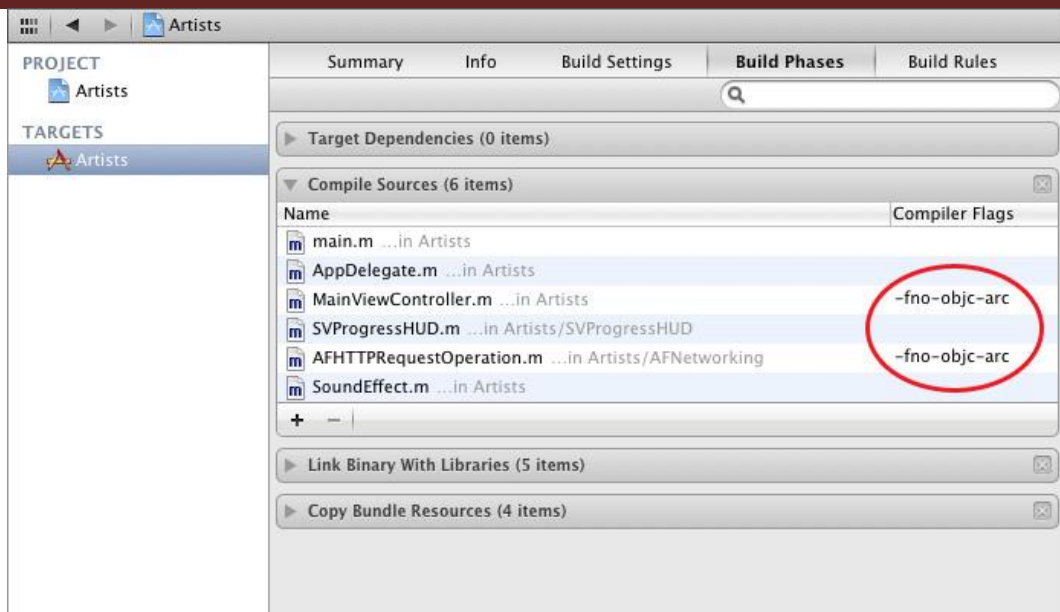
```
if(fadeOutTimer != nil)
[fadeOutTimer invalidate], fadeOutTimer = nil;
```

技术上来说，我们不需要在 `dealloc` 中调用 `fadeOutTimer = nil;`，因为对象在被删除时，将自动释放任何的成员变量。但是，在其他的记时器被停止的方法中，你应该将 `fadeOutTimer` 设为 `nil`。否则，`SVProgressHUD` 将一直等待这个已经停止的记时器。

再次构建程序，现在应该没有警告了。转化完成！

稍等...我们在转化时略过了 `MainViewController` 和 `AFHTTPRequestOperation`。那么如何让他们编译通过呢？当我们尝试用 `ARC` 对工程进行构建时，这些文件当然会产生大量的错误。

答案很简单：转化工具对这两个源文件禁用了 `ARC`。你可以在 `Project Settins` 屏幕的 `Build Phases` 标签上看到：



之前当我们将 Build Settings 内的 Objective-C Automatic Reference Counting 设置为 YES 时，已经在程序范围内启用了 ARC。但是通过使用 `-fno-objc-arc` 标志，你可以让编译器对特定的文件不使用 ARC。xcode 会以关闭 ARC 的方式对这些文件进行编译。

这是因为，Apple 的那些家伙过分的期待开发者一次性的将整个工程转换到 ARC（译者：所以为你自动加上了 `-fno-objc-arc` 标志）。提示：一个简单的方法是使用转化工具转化那些你希望移植的文件，让它自动为剩下的文件加上 `-fno-objc-arc` 标志。你也可以手动去加，但要是你有很多不想使用 ARC 的文件的话，这样做会相当烦人。

## 2.9. 移植问题(Migration Woes)

我们的转化进行得相当顺利。我们只要对 `SoundEffect.m` 的语句做一点小小的修改（插入一个 `__bridge`），转化工具替我们完成剩下的工作。

但是，LLVM 3.0 编译器比起以前的编译器，有点不那么仁慈，它可能在检查过程中查出其他的问题。你可能需要在转化工具接手前编辑更多的代码。

下面是可能遇到的问题时的参考手册，以及关于如何解决他们的一些提示。

### "Cast ... requires a bridged cast"

这是我们之前碰到过的。如果编译器不能自己确定如何转换，他会期待你插入一个 `__bridge` 修饰符。另外还有两种 `bridge` 类型，`__bridge_transfer` 和 `__bridge_retained`，你要使用哪一个完全取决于你想做什么。关于 `bridge` 类型的更多信息参见"免费桥接(Toll-Free Bridging)"部分。

### "Receiver type 'X' for instance message is a forward declaration"

如果你有一个类，假设 `MyView` 是一个 `UIView` 的子类，如果你要调用它的方法或者使用它的某个属性，那么你必须 `#import` 这个类的定义。一般来说你必须先做这件事才能让编译通过，但也有例外。

比如，你在 `.h` 文件中使用了前置声明，说明了 `MyView` 是一个类：

```
@class MyView;
```

此后在你的 `.m` 文件中，你调用：

```
[myView setNeedsDisplay];
```

在之前，你大概能编译通过并正常运行，即使没有使用 `#import` 声明。而在使用 ARC 时，你总是需要显式的添加一个 `import`：

```
#import "MyView.h"
```

### "Switch case is in protected scope"

下面的代码会产生这个错误：

```
switch (X)
{
case Y:
    NSString *s = ...;
    break;
}
```

这种写法已经被禁止了。如果你在一个 `case` 中声明了一个新的指针变量，你就必须将整个 `case` 包含在一对花括号中：

```
switch (X)
{
case Y:
{
    NSString *s = ...;
    break;
}
}
```

现在变量的作用域很明确，这正是 ARC 需要知道的，这样它就能在正确的时刻释放对象。

### "A name is referenced outside the NSAutoreleasePool scope that it was declared in"

你可能会有一些代码会创建自己的自动释放池：

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
// ... do calculations ...
NSArray* sortedResults =
```

```
[[filteredResults sortedArrayUsingSelector:@selector(compare:)]
retain];
[pool release];
return [sortedResults autorelease];
```

转化工具需要将其改为下面的代码：

```
@autoreleasepool
{
// ... do calculations ...
NSArray* sortedResults = [filteredResults sortedArrayUsingSelector:@ selector(compare:)];
}
return sortedResults;
```

但这不再是有效的代码。变量 `sortedResults` 是在 `@ autoreleasepool` 范围内声明的，因此在此范围外不能被访问了。为了解决这个问题，你需要将变量声明的代码移动到 `NSAutoreleasePool` 创建的上面对：

```
NSArray* sortedResults;
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
```

...

现在转化工具就能正确的重写你的代码了。

### "ARC forbids Objective-C objects in structs or unions"

ARC 的限制之一是，你不再能够将 Objective-C 对象放到 C 结构中了。下面的代码不再有效：

```
typedef struct
{
UIImage *selectedImage;
UIImage *disabledImage;
}
ButtonImages;
```

你被推荐使用 Objective-c 类来替代这个结构。后面我们会详细讨论这个，那时我会为你告诉你一些其他的方法。

还可能有些其他的预检查错误，但是这些是最常见的。

注意：如果你多次使用自动转化工具，它的行为可能有些古怪。如果你没有转化全部文件，而让有一些文件没有被它检查，那么之后如果你想转化剩下的文件时，转化工具不再会做任何改变了。所以我建议你只运行它一次，不要分批的转化它们。



## 2.10. 手工转化 (Converting By Hand)

我们差不多将整个工程转到了 ARC，除了 MainViewController 和 AFHTTPRequestOperation。在这个部分，我会告诉你如何手工转化 MainViewController。有时你自己动手会更有乐趣，你会为真正发生的事情而感觉良好。

如果你看一眼 MainViewController.h，你会发现这个类声明了两个成员变量：

```
@interface MainViewController : UIViewController <UITableViewDataSource, UITableViewDelegate,
UISearchBarDelegate, NSXMLParserDelegate>
{
    NSOperationQueue *queue;
    NSMutableString *currentStringValue;
}
```

当你思考它时会觉得奇怪，类的公有接口中有成员变量。一般情况下，实例变量应该是类的内部实现的一部分，而不是你想暴露在公共接口中的东西。对类的使用者来说，类的成员变量是什么无关紧要。从数据隐藏的角度来看，我们应该将这些实现细节放到类的 @implementation 部分。我很高兴的说，这对于 LLVM3.0 是可能实现的(不管你用不用 ARC)。

将成员变量块从 MainViewController.h 移动到 MainViewController.m 中，头文件现在变成了这样：

```
#import <UIKit/UIKit.h>
@interface MainViewController : UIViewController <UITableViewDataSource, UITableViewDelegate, UISearchBarDelegate,
NSXMLParserDelegate>
@property (nonatomic, retain) IBOutlet UITableView *tableView;
@property (nonatomic, retain) IBOutlet UISearchBar *searchBar;
@end
```

MainViewController.m 看起来像这样：

```
@implementation MainViewController
{
    NSOperationQueue *queue;
    NSMutableString *currentStringValue;
}
```

构建程序...运行正常。这项技术让你的.h 文件更加简洁，而将成员变量放到他们真正属于的地方。

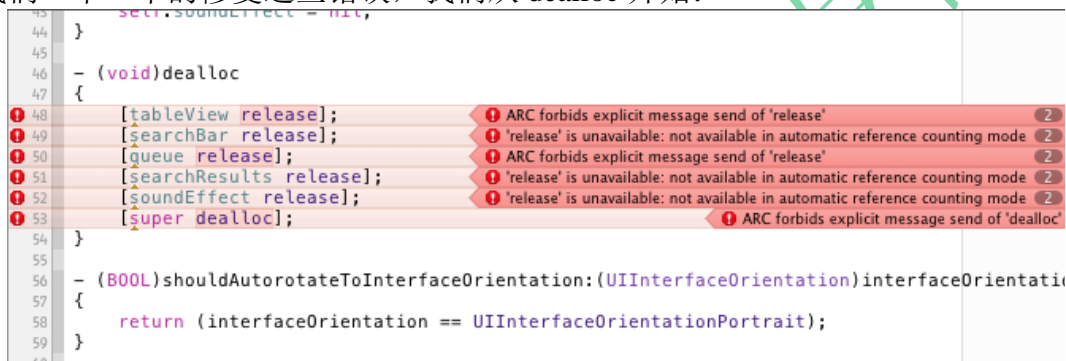
你可以对 SoundEffect 类干同样的事。只要将成员变量部分转移到.m 文件中。因为我们现在不会在 SoundEffect.h 中引用 SystemSoundID，你也可以将对 AudioService.h 的#import 移到 SoundEffect.m 中去。SoundEffect 的头文件中不再暴露任何实现的细节了。干净又漂亮。

注意：你也可以将成员变量放到类扩展中。这在类的实现分布于多个文件中时很有用。你可以将扩展放到一个共享的私有的头文件中，这样所有不同的实现文件对成员变量都有访问权限。

现在是为 MainViewController.m 实现 ARC 的时候了。转到 Build Phases 设置，将-fno-objc-arc 编译器标志从 MainViewController.m 上删除。xcode 可能还不能立即得知的以变化，尝试一次新的构建，你应该会得到一堆的错误，但是如果 xcode 还是说"构建成功"的话，你可以关闭工程然后重新打开它。

## 2.11. Dealloc

让我们一个一个的修复这些错误，我们从 dealloc 开始：



```
44 }
45
46 - (void)dealloc
47 {
48     [tableView release];
49     [searchBar release];
50     [queue release];
51     [searchResults release];
52     [soundEffect release];
53     [super dealloc];
54 }
55
56 - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
57 {
58     return (interfaceOrientation == UIInterfaceOrientationPortrait);
59 }
60 }
```

The screenshot shows the following error messages on the right side of the code editor:

- ARC forbids explicit message send of 'release'
- 'release' is unavailable: not available in automatic reference counting mode
- ARC forbids explicit message send of 'release'
- 'release' is unavailable: not available in automatic reference counting mode
- 'release' is unavailable: not available in automatic reference counting mode
- 'release' is unavailable: not available in automatic reference counting mode
- ARC forbids explicit message send of 'dealloc'

dealloc 中的每一行都产生了一个错误。我们不应该再调用[release]和[super dealloc]了。因为我们不再会在 dealloc 中做任何事情了，我们可以简单的把整个 dealloc 方法删除。

保留 dealloc 方法的唯一情况是，当你需要释放某些不在 ARC 保护伞之下的资源时。比如说对 Core Foundation 对象调用 CFRelease()，对使用 malloc()分配的内存调用 free()，反注册通知，停止记时器，等等。

如果你是某对象的委托，有时就需要显式的断开同它的连接，但是这通常是自动发生的。大多数情况下，委托是弱引用（我们很快就会讨论它），所以如果某个被释放的对象是另一个对象的委托，委托指针会在该委托对象被释放后置为 nil。弱指针会对自身做清理工作。

顺便说一下，在你的 dealloc 方法中，你仍然可以引用成员变量，因为那时他们还没有被释放，直到 dealloc 返回。

## 2.12. SoundEffect Getter 方法

soundEffect 方法调用了 release，所以很容易修复：



```

45 - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
46 {
47     return (interfaceOrientation == UIInterfaceOrientationPortrait);
48 }
49
50 - (SoundEffect *)soundEffect
51 {
52     if (soundEffect == nil) // lazy loading
53     {
54         SoundEffect *theSoundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
55         self.soundEffect = theSoundEffect;
56         [theSoundEffect release];
57     }
58     return soundEffect;
59 }
60
61 #pragma mark - UITableViewDataSource
62
63 - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
64 {
65

```

这个方法是 soundEffect 实际的 getter 方法。它使用了延迟加载技术（lazy loading technique），在音效第一次使用时加载它。在此我使用了手动内存管理下创建对象的一般模式。首先新对象被保存到一个临时变量中，然后它被赋给真正的属性，最后释放局部变量。这就是我的写法，你可能也会这么做：

```

SoundEffect *theSoundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
self.soundEffect = theSoundEffect;
[theSoundEffect release];

```

我们可以仅仅删掉 release 调用，但留下了一个不再有用的局部变量：

```

SoundEffect *theSoundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
self.soundEffect = theSoundEffect;

```

所以，我们可以只写一行代码：

```

self.soundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];

```

在手动内存管理下，这会导致内存泄漏(会多出一个 retain)，但对于 ARC 这没有问题。

## 2. 13. 请释放我，让我走

就像你不能够再调用 release 一样，你也不能再调用 autorelease 了：

```

71 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
72 {
73     static NSString *CellIdentifier = @"Cell";
74
75     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
76     if (cell == nil)
77     {
78         cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
79     }
80     if ([self.searchResults count] == 0)
81     {
82         cell.textLabel.text = @"(Nothing found)";
83     }
84     else
85     {
86         cell.textLabel.text = [self.searchResults objectAtIndex:indexPath.row];
87     }
88     return cell;
89 }

```

解决方法很简单，将这一行：

```

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

```

改为:

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

下一个有错的方法是 `escape:`，但是我们先把它放一放。这是和免费桥接相关的事情，我会用一个特殊的章节来讨论它。

剩下的两个错误是和 `release` 调用相关，在 `searchBarSearchButtonClicked:` 和 `parser:didEndElement:` 中，你可以简单的删除他们。

## 2. 14. 属性

如果你看 `MainViewController.m` 文件的顶部，你会发现他使用了一个类扩展来声明了两个私有属性，`searchResult` 和 `soundEffect`:

```
@interface MainViewController ()
@property (nonatomic, retain) NSMutableArray *searchResults;
@property (nonatomic, retain) SoundEffect *soundEffect;
@end
```

这主要能让手动内存管理更简单一些，这也是开发者使用属性的原因，当你这么做的时候:

```
self.searchResults = [NSMutableArray arrayWithCapacity:10];
```

这个 `setter` 方法会处理旧值的释放（如果有的话），以及新值的保留。开发者使用属性可以减少对何时需要保留和释放的考虑。但现在用 `ARC`，你可以完全不必考虑这一切！

以我的观点，如果使用属性只是为了简化内存管理的话，现在已经不再必要了。你仍然可以继续这么做，但我认为现在用成员变量更好，而只在你需要让其他类从公共接口访问到内部数据时才使用属性。

因此，你要删除类扩展和 `searchResult` 及 `soundEffect` 的 `@synthesize` 声明。添加新的成员变量来替代他们:

```
@implementation MainViewController
{
    NSOperationQueue *queue;
    NSMutableString *currentStringValue;
    NSMutableArray *searchResults;
    SoundEffect *soundEffect;
}
```

当然，这意味着我们不能再使用 `self.searchResults` 和 `self.soundEffect`。修改

viewDidUnload 如下:

```
- (void)viewDidUnload
{
    [super viewDidUnload];
    self.tableView = nil;
    self.searchBar = nil;
    soundEffect = nil;
}
```

我们还是需要将 soundEffect 设为 nil，因为在此我们希望释放 SoundEffect 对象，当 iPhone 收到低内存警告时，我们应该释放尽可能多的内存，SoundEffect 对象此时可以被释放。因为成员变量默认创建强关系，所以将 soundEffect 设为 nil 将去掉 SoundEffect 对象的所有者，soundEffect 将立即被释放。

soundEffect 方法现在变成了:

```
- (SoundEffect *)soundEffect
{
    if (soundEffect == nil) // lazy loading
    {
        soundEffect = [[SoundEffect alloc] initWithSoundNamed:@"Sound.caf"];
    }
    return soundEffect;
}
```

上面的代码是我们尽可能简化的结果。SoundEffect 对象分配后被赋给了 soundEffect 成员变量。这个变量成为了它的所有者，对象将一直存在直到将 soundEffect 设为 nil（在 viewDidUnload 中），或者直到 MainViewController 被释放。

在这个文件的剩余部分中，用 searchResults 替换掉 self.searchResults。再次构建程序，唯一有错的地方应该是在 escape: 方法中。

注意在 searchBarSearchButtonClicked 中，我们还是需要这样写:

```
[self.soundEffect play];
```

即使现在没有名为 soundEffect 的属性了，还是可以这样写。点语法不是仅限于属性使用，虽然大多数情况下都是。如果你觉得这样写不舒服，你可以将此行改写为:

```
[[self soundEffect] play];
```

但不要这样写:

```
[soundEffect play];
```

因为我们对 `SoundEffect` 对象使用了延迟加载，在你调用 `soundEffect` 方法之前，`soundEffect` 变量将一直是 `nil` 的。因此，为了确保我们确实有一个 `SoundEffect` 对象，你应该总是通过 `self` 来访问。如果你觉得这种情况确实需要将 `soundEffect` 声明为属性，那么请便。萝卜白菜各有所爱。:-)

作为最佳实践，如果你定义了一个属性，你应该总是使用属性。唯一需要访问属性背后的成员变量的地方是在 `init` 中，或者当你提供自定义的 `getter` 和 `setter` 方法时。这就是为什么同步声明经常修改成员变量的原因：

```
@synthesize propertyName = _propertyName;
```

这种结构将防止你意在使用 "`self.propertyName`" 的时候，输入 "`propertyName`" 从而误用了其背后的成员变量。

谈到属性，`MainViewController` 在其 `.h` 文件中还有两个 `outlet` 属性：

```
@property (nonatomic, retain) IBOutlet UITableView *tableView;  
@property (nonatomic, retain) IBOutlet UISearchBar *searchBar;
```

对于所有的 `outlet` 属性推荐使用弱关系。这些视图对象已经是视图控制器层次结构的一部分了，不需要在别的地方 `retain` 它。将 `outlet` 声明为弱引用的最大好处是，你不需要再花时间编写你的 `viewDidUnload` 方法了。

现在我们的 `viewDidUnload` 看起来像这样：

```
- (void)viewDidUnload  
{  
    [super viewDidUnload];  
    self.tableView = nil;  
    self.searchBar = nil;  
    soundEffect = nil;  
}
```

你现在可以将它简化为下面的样子：

```
- (void)viewDidUnload  
{  
    [super viewDidUnload];  
    soundEffect = nil;  
}
```

没错，因为 `tableView` 和 `searchBar` 属性是弱引用，他们在所指向的对象被销毁后自动被设为 `nil`。这就是我们所谓的“归零”弱指针。

当 iPhone 收到低内存警告时，视图控制器的主视图被卸载，同时会 **release** 它所有的子视图。那时，`UITableView` 和 `UISearchBar` 对象被释放，并且归零弱指针系统自动设置 `self.tableView` 和 `self.searchBar` 为 `nil`。我们不再需要自己在 `viewDidUnload` 中做这个了。实际上，在 `viewDidUnload` 被调用时，这些属性已经是 `nil` 了。我很快会为你演示这点，但为此我们需要为程序加上第二个页面。

这不意味着你可以彻底忘记 `viewDidUnload`。记住，只要你还保留有对象的指针，它就一直存在。如果你不希望继续这样的话，你可以将他们的指针设为 `nil`。这就是我们对 `soundEffect` 说做的。我们这时还不希望删除 `searchResult` 数组，但是如果愿意，也可以在此将其设为 `nil`。任何你不再需要的数据，如果它是一个弱 `outlet` 属性，你还是在 `viewDidUnload` 中将它置为 `nil`！这对 `didReceiveMemoryWarning` 同样适用。

所以从现在开始，将你的 `outlet` 属性变为弱属性。唯一应该是强属性的 `outlet` 是那些在 `nib` 文件中从 `File's Owner` 到顶层对象的连接。

总结一下，新的属性修饰语如下：

- **strong**. 是 `retain` 的同义词。一个强属性会成为所指向对象的所有者。
- **weak**. 这个属性代表一个弱指针。当所指向的对象被释放时，他会自动被设为 `nil`。记住，对于 `outlet` 使用它。
- **unsafe\_unretained**. 这是原来的 `"assign"` 的同义词。它只在特殊情况下以及你想将目标设为 iOS4 时使用。后面会讲到它。
- **copy**. 这还是和以前一样。这将制作对象的一份拷贝，并创建强关系。
- **assign**. 你能再为对象使用它了，但你还是可以用于基础类型如 `BOOL`, `int` 和 `float`。

在 ARC 以前，我们能这样写：

```
@property (nonatomic, readonly) NSString *result;
```

他会隐式的创建一个 `assign` 属性。这个对于 `readonly` 属性没问题。毕竟，你在谈到只读数据的时候 `retain` 又有什么意义？但是，用 ARC 时，上面的代码将产生错误：

```
"ARC forbids synthesizing a property of an Objective-C object with unspecified ownership or storage attribute"
```

你必须显式的声明你希望此属性为 `strong`, `weak` 或者 `unsafe_unretained`。大多数情况下 `strong` 是恰当的答案：

```
@property (nonatomic, strong, readonly) NSString *result;
```

前面我曾提到，如果你声明了一个属性，你应该总是通过 `self.propertyName` 而不是其背后的实例变量(除非是在 `init` 中或者任何自定义的 `getter` 或 `setter` 方法中)来访问。这对于 `readonly` 属性特别适用。如果你通过他们的实例变量来编辑这个属性，ARC 会被搞糊涂并会产生一个奇怪的 BUG。正确的做法是重新将它在类扩展中定义为 `readwrite` 属性。

在你的.h 中这样写:

```
@interface WeatherPredictor
@property (nonatomic, strong, readonly) NSNumber *temperature;
@end
```

在你的.m 中这样写:

```
@interface WeatherPredictor ()
@property (nonatomic, strong, readwrite) NSNumber *temperature;
@end
```

## 2. 15. 免费桥接 (Toll-Free Bridging)

让我们解决最后一个方法中的问题，这样我们就能重新运行程序了。

```
108 - (NSString *)escape:(NSString *)text
109 {
110     return [(NSString *)CFURLCreateStringByAddingPercentEscapes(
111         NULL,
112         (CFStringRef)text,
113         NULL,
114         (CFStringRef)@"!*'();:@&=+$/%#[]",
115         CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding))
116         autorelease];
117 }
118
119
```

这个方法使用了 `CFURLCreateStringByAddingPercentEscapes()` 函数来对字符串进行 URL 编码。我们用它来确保搜索文本中的空格或其他特殊字符被转换为 HTTP GET 请求所能接受的合法字符。

编译器给出了一些错误:

- Cast of C pointer type 'CFStringRef' to Objective-C pointer type 'NSString \*' requires a bridged cast
- Cast of Objective-C pointer type 'NSString \*' to C pointer type 'CFStringRef' requires a bridged cast
- Semantic Issue: 'autorelease' is unavailable: not available in automatic reference counting mode
- Automatic Reference Counting Issue: ARC forbids explicit message send of 'autorelease'

最后两个错误很简单，是说不能使用 `[autorelease]` 了。让我们先搞定这个。修改后的方法为:

```
- (NSString *)escape:(NSString *)text
{
return (NSString *)CFURLCreateStringByAddingPercentEscapes(
NULL,
(CFStringRef)text,
NULL,
(CFStringRef)@"!*'();:@&=+$/%#[]", CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding));
}
```



```
}
```

另外两个错误和类型转换应该使用“桥接”有关。在这个方法里面有三处转换：

```
• (NSString *)CFURLCreateStringByAddingPercentEscapes(...)  
• (CFStringRef)text  
• (CFStringRef)@"!*()::@&=+$/?%#[]"
```

编译器仅仅抱怨了前两个。第三个是对一个常量对象进行转换，所以并不需要特殊的内存管理。这是一个将被嵌入应用程序可执行文件中的字符串常量。和“真正的”对象不一样，它不会被分配和释放。

如果你希望这样写也可以：

```
CFSTR(@"!*()::@&=+$/?%#[]")
```

CFSTR()宏用一个指定的字符串创建一个 CFStringRef 对象。这个字符串常量是一个标准的 C 字符串，所以不需要以@打头。不需要将 NSString 对象转换为 CFStringRef，我们能直接得到一个 CFStringRef 对象。你要用哪个取决于你的喜好，他们会得到完全相同的结果。

当你在两个世界之间移动对象时，桥接转换(bridged cast)是必要的。一边是 objective-c 的世界，另一边是 Core Foundation。

对于目前大多数的程序来说，对 Core Foundation 的使用并不多，你可以用方便的 objective-c 类做任何你想做的事。但是，某些低级别的 API 比如 Core Graphics 和 Core Text 是基于 Core Foundation 的，他们不太可能有 Objective-c 的版本。幸运的是，iOS 的设计者让这两个不同王国的对象之间的转换很方便。你不需要处理它！

在所有情况下，NSString 和 CFStringRef 可以当作一个东西对待。你可以接受一个 NSString 对象，把它当作一个 CFStringRef 对象，或者将 CFStringRef 对象用作 NSString。这就是免费桥接背后的思想。之前这只需要做一个简单的转换：

```
CFStringRef s1 = [[NSString alloc] initWithFormat:@"Hello, %@!", name];
```

当然，你也需要记住在完成使用后，释放该对象：

```
CFRelease(s1);
```

相反的额，从 Core Foundation 到 Objective-C，也同样简单：

```
CFStringRef s2 = CFStringCreateWithCString(kCFAllocatorDefault, bytes, kCFStringEncodingMacRoman);  
NSString *s3 = (NSString *)s2;  
// release the object when you're done  
[s3 release];
```

现在我们有 ARC，编译器需要知道谁负责释放那些转换的对象。如果你将 NSObject 作为 Core Foundation 对象，那么 ARC 不会负责释放它。但你确实需要告诉 ARC 你的意图，编译器不能自己来推断。同样的，如果你创建了一个 Core Foundation 对象但将其转换为了 NSObject 对象，你就需要告诉 ARC 得到它的所有权，并及时释放它。这就是桥接转换要做的。

让我们先看一个简单的情形。CFURLCreateStringByAddingPercentEscapes() 函数接受一打的参数，其中有两个是 CFStringRef 对象。这是 NSString 在 core foundation 中对等的类。之前，我们只用将这些 NSString 对象转换为 CFStringRef，但要用 ARC，则需要更多的信息。我们已经处理了常量字符串，它不需要桥接转换，因为它是一个不会被释放的特殊对象。但是，text 参数则是另一回事了。

text 变量是传给此方法的 NSString 对象。就像局部变量，方法参数是强指针；对象在进入方法的时候被保留。text 变量的值将在指针被销毁前一直存在。因为它是一个局部变量，这发生在 escape: 方法结束时。

我们希望 ARC 保留对此变量的所有权，但我们希望暂时将其作为 CFStringRef。为了应对这种情况，我们使用 \_\_bridge 区分符 (specifier)。它告诉 ARC 所有权没有任何变化，并应该用一般法则来释放该对象。

我们已经在前面的 SoundEffect.m 中用过 \_\_bridge 了：

```
OSStatus error = AudioServicesCreateSystemSoundID((__bridge CFURLRef)fileURL, &theSoundID);
```

这种情况也做了相同的处理。fileURL 变量包含一个由 ARC 管理的 NSURL 对象。但是，AudioServicesCreateSystemSoundID() 函数希望接受一个 CGURLRef 对象。幸运的是，NSURL 和 CFURLRef 是免费桥接的，因此我们可以将其中的一个转换为另一个。因为我们仍然希望在完成使用后由 ARC 来释放它，我们使用 \_\_bridge 关键字指出 ARC 仍旧管理着它。

修改下列 escape: 方法：

```
- (NSString *)escape:(NSString *)text
{
    return (NSString *)CFURLCreateStringByAddingPercentEscapes(
        NULL,
        (__bridge CFStringRef)text,
        NULL,
        CFSTR("!*();:@&=+$/%#[]"),
        CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding));
}
```

除了一个错误之外，这几乎解决了所有的错误。

大多数情况下，当你将一个 Objective-C 对象转换为 Core Foundation 对象，或者反过来时，你会希望使用 \_\_bridge。但是，在有些情况下，你确实希望给予 ARC 的所有权，或者解



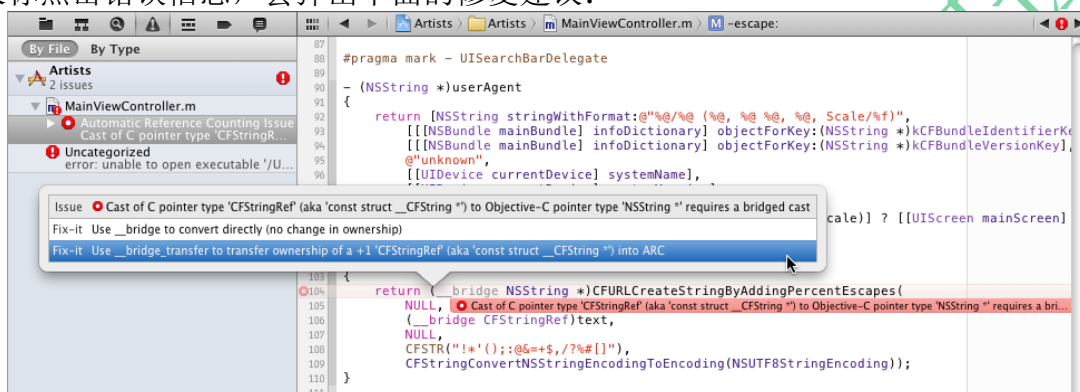
除其所有权。这时，另外有两个桥接转换关键字可供你使用：

- `__bridge_transfer`: 给予 ARC 所有权
- `__bridge_retained`: 解除 ARC 的所有权

在我们的源代码中还有一个错误，就在此行：

```
return (NSString *)CFURLCreateStringByAddingPercentEscapes(
```

如果你点击错误信息，会弹出下面的修复建议：



它给出了两种可能的解决方案：`__bridge` 和 `__bridge_transfer`。这里正确的选择是 `__bridge_transfer`。`CFURLCreateStringByAddingPercentEscapes()` 函数创建了一个新的 `CFStringRef` 对象。当然，我们更愿意使用 `NSString` 所以需要转换。我们想要做的是：

```
CFStringRef result = CFURLCreateStringByAddingPercentEscapes(. . .);  
NSString *s = (NSString *)result;  
return s;
```

因为函数名有 "Create" 前缀，它返回一个被 `retain` 的对象。有人负责在某个时候释放它。如果我们不将此对象作为 `NSString` 返回，那么我们的代码看起来可能像这样：

```
- (void)someMethod  
{  
    CFStringRef result = CFURLCreateStringByAddingPercentEscapes  
    (. . .);  
    // do something with the string // . . .  
    CFRelease(result);  
}
```

请记住 ARC 仅仅能作用于 ARC 上，不能处理 Core Foundation 创建的对象。你自己仍然能够对 Core Foundation 对象调用 `CFRelease()`！

我们想在 `escape` 中要做的是，将新的 `CFStringRef` 对象转换为 `NSString` 对象，然后在我们不再使用此字符串时，ARC 应该自动释放它。但是 ARC 需要被告知这么做。因此，我们

使用 `__bridge_transfer` 修饰符来表示：“hey ARC，CFStringRef 对象现在是一个 NSString 对象，我们希望你去处理它，这样我们就不必自己调用 `CFRelease()` 了。”

此方法现在变成了：

```
- (NSString *)escape:(NSString *)text
{
    return (__bridge_transfer NSString *)
    CFURLCreateStringByAddingPercentEscapes( NULL,
    (__bridge CFStringRef)text,
    NULL,
    CFSTR("!*() ;:@&=+$/%#[]"),
    CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding));
}
```

我们得到了 `CFURLCreateStringByAddingPercentEscapes()` 的结果，它是一个 `CFStringRef` 对象，将其转换为 `NSString` 对象，交给 ARC 来处理。

如果我们只是用 `__bridge` 来代替，那么你应该会有内存泄漏。在你完成使用此对象后，没有人调用 `CFRelease()`，ARC 并不知道它应该释放这个对象。作为结果，对象将永远存在于内存之中。所以，你选择正确的桥接标识符是相当重要的！

为了便于记住应该使用哪一种类型的桥接，有一个名为 `CFBridgingRelease()` 的帮助函数。它做了和 `__bridge_transfer` 转换同样的事情，但是更简洁。escape: 方法的最后版本变成了：

```
- (NSString *)escape:(NSString *)text
{
    return CFBridgingRelease(CFURLCreateStringByAddingPercentEscapes(
    NULL,
    (__bridge CFStringRef)text,
    NULL,
    CFSTR("!*() ;:@&=+$/%#[]"), CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncoding)));
}
```

较之于 `(__bridge_transfer NSString *)`，我们现在改用 `CFBridgingRelease()`，将 `CFURLCreateStringByAddingPercentEscapes()` 放入其中。这个 `CFBridgingRelease()` 函数是一个内联函数，所以调用它和直接转换一样快。它名为 `CFBridgingRelease()` 的原因是，你会在任何原本要使用 `CFRelease()` 来平衡新对象创建的地方使用它。

另一个需要用到桥接转换的常用的框架是 `AddressBook` 框架。例如：

```
- (NSString *)firstName
{
    return CFBridgingRelease(ABRecordCopyCompositeName(...));
}
```

在任何你会调用名字由 `Create`, `Copy`, 或者 `Retain` 组成的函数的地方, 你必须调用 `CFBridgingRelease()` 来将所有权安全的传送给 ARC。

`__bridge_retained` 又如何使用呢? 这会在相反的情况下使用。假设你有一个 `NSString` 对象, 你需要对他使用一些 `Core Foundation` API, 这些 API 希望获得字符串对象的所有权。你不希望 ARC 也释放该对象, 因为这会导致过度释放一次, 这有可能导致程序崩溃。也就是说, 你用 `__bridge_retained` 将对象交给 `Core Foundation`, 这样 ARC 就不再负责释放它。代码如下:

```
NSString *s1 = [[NSString alloc] initWithFormat:@"Hello, %@!", name];
CFStringRef s2 = (__bridge_retained CFStringRef)s1;
// do something with s2 // ...
CFRelease(s2);
```

一旦(`__bridge_retained CFStringRef`)的转换发生, ARC 认为自己不再对释放字符串对象负责。如果你在这里使用 `__bridge`, 程序将有可能崩溃。在 `Core Foundation` 代码完成处理之前, ARC 可能使字符串对象被释放。

这种转换也有一个帮助函数 `CFBridgingRetain()`。你可以根据它的名字知道, 它能让 `core Foundation` 保留对象。上面的代码可以优化为:

```
CFStringRef s2 = CFBridgingRetain(s1);
// ...
CFRelease(s2);
```

现在, 代码的含义变得更清晰了。对 `CFRelease()` 的调用可以恰当的用 `CFBridgingRetain()` 来保持平衡。我觉得你不会经常需要使用这种特别的桥接类型。在我的脑海里, 不记得有哪个经常使用的 API 需要用到它。

你的程序中不太可能大量用到 `Core Foundation` 代码。你要用到的框架大部分是 `Objective-C` 的, 除了 `Core Graphics` (它没有免费桥接类型), 通讯录, 以及偶尔使用的低级别函数。但是如果你用到它, 编译器会在你需要使用桥接转换时提醒你。

注意: 不是所有的 `Objective-c` 和 `Core Foundation` 对象可以免费桥接的。比如, `CGImage` 和 `UIImage` 就不能彼此转换, `CGColor` 和 `UIColor` 也不行。下面的网页列出了可以互相转换的类型: <http://bit.ly/j65Ceo>

`__bridge` 转换不仅仅限于和 `Core Foundation` 的互动。一些 API 接受 `void*` 指针, 这让你可以保存任何对象的引用, 不管是 `Objective-C` 对象, `Core Foundation` 对象, `malloc()` 创建的内存缓冲, 等等。`void*` 标记表示: 这是一个指针, 但它所指向的实际数据类型可能是任何东西。

为了将 `Objective-C` 对象转换为 `void*`, 或者相反, 你会需要做一个 `__bridge` 转换。例

如:

```
MyClass *myObject = [[MyClass alloc] init];
[UIView beginAnimations:nil context:((__bridge void *)myObject)];
```

在动画委托方法中，你要反向转化来得到对象:

```
- (void)animationDidStart:(NSString *)animationID
context:(void *)context
{
    MyClass *myObject = (__bridge MyClass *)context;
    ...
}
```

我们在下一章将看到另一个例子，那里我们会对 Cocos2D 使用 ARC。

总结一下:

- 将所有权从 Core Foundation 转移到 Objective-C 时，需要使用 `CFBridgingRelease()`。
- 将所有权从 Objective-C 转移到 Core Foundation 时，需要使用 `CFBridgingRetain()`。
- 当你想将一种类型临时当作另一种类型使用，而不转移所有权时，需要使用 `__bridge`。

以上就是 `MainViewController` 需要关注的要点。现在所有的错误都应该解决了，你可以构建和运行此程序。在本教程中，我们不会将 `AFHTTPRequestOperation` 转换到 ARC 了。

在不久的将来，你会发现许多你喜欢的第三方库并不是以 ARC 风格编写的。同时维护两个版本的库，一个使用 ARC 另一个不用，这一点都不好玩。所以我建议众多的库维护者只选择一种。新库可以仅仅使用 ARC 编写，但是老库可能很难转换。因此，你仅有一部分代码可能保留支持 ARC 的方式(the-fno-objc-arc 编译器标志)。

幸运的是，ARC 是基于单个文件的，所以将这些库组合到你的 ARC 下 (ARCified) 的工程中不会有问题。因为有时候对大部分文件禁用 ARC 将导致麻烦，我们下一章会谈及将非 ARC 的第三方库加入到你工程中的更聪明的做法。

## 2. 16. 委托和弱属性

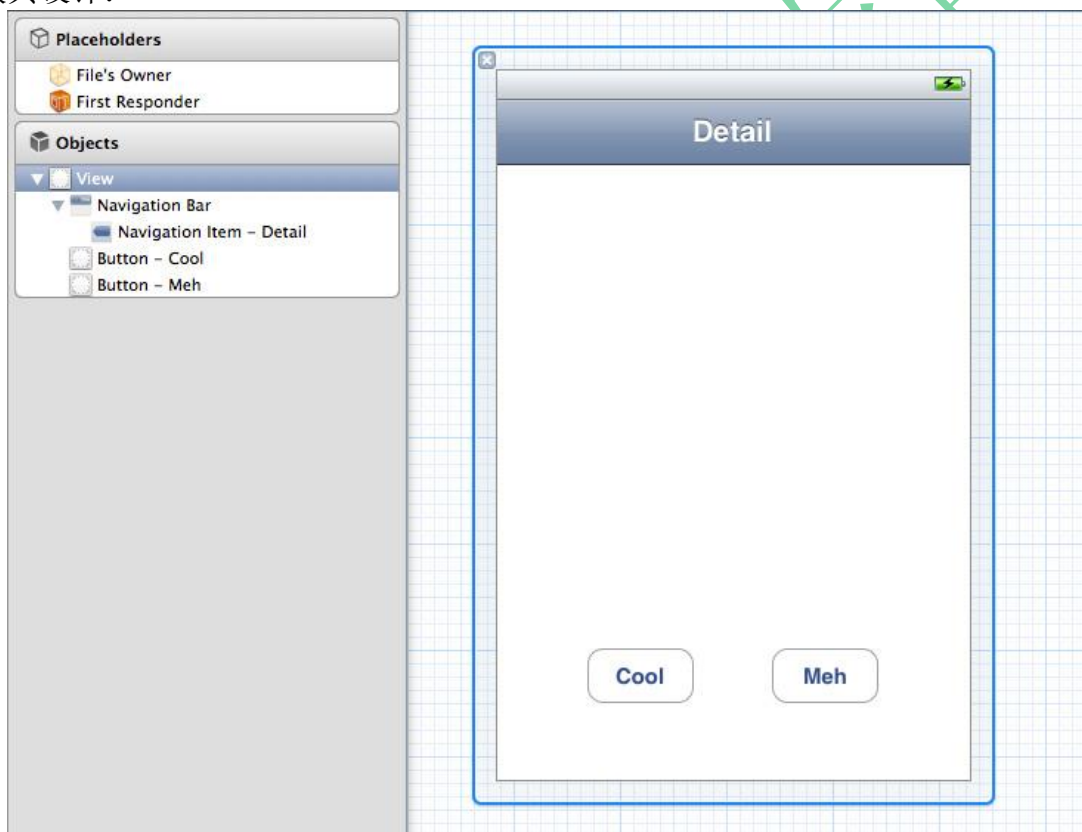
迄今为止，你所看到的程序还是很简单的，仅仅演示了 ARC 的某些方面。为了演示余下的方面，我们首先必须往程序中添加一个新的页面。

为工程添加一个新的 UIViewController 子类，用 XIB 创建用户界面，将其命名为 DetailViewController。

向 DetailViewController.h 中添加两个动作方法：

```
@interface DetailViewController : UIViewController
- (IBAction)coolAction;
- (IBAction)mehAction;
@end
```

我们会将这两个动作连接到 NIB 文件中的两个按钮上。打开 DetailViewController.xib 文件，修改其设计：



它仅有一个导航条和两个按钮。按下 Control 键同时拖动每个按钮到文件所有者(File's Owner)，并连接他们的 Touch Up Inside 事件到相应的动作上。

在 DetailViewController.m 中，添加这两个动作方法的实现。目前，我们使用空实现：

```
- (IBAction)coolAction {
}
- (IBAction)mehAction {
}
```

我们会对主视图控制器做些修改，让你在点击一个搜索结果时调用这个详情页面。在

MainViewController.m 中，修改 didSelectRowAtIndexPath 方法：

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    DetailViewController *controller = [[DetailViewController alloc] initWithNibName:@"DetailViewController" bundle:nil];
    [self presentViewController:controller animated:YES completion:nil];
}
```

这会初始化 DetailViewController 并将其显示在当前视图之上。

添加下列方法：

```
- (NSIndexPath *)tableView:(UITableView *)tableView
willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    if ([searchResults count] == 0)
        return nil;
    else
        return indexPath;
}
```

如果没有搜索结果，我们会在表格中放置一行显示"(Nothing Found)"。我们不希望此时用户点击后打开详情页。

因为 MainViewController 不认识 DetailViewController，所以我们必须加入一个#import。将此加入到 MainViewController.h：

```
#import "DetailViewController.h"
```

(我们将其加入到.h 文件而不是.m 文件，原因很快就会知道的。)

如果你现在运行程序，点击某一行会进入详情页，但是你还不能关闭它。我们连接到"Cool"和"Meh"按钮上的动作仍然是空的，点击他们没有效果。





为此，我们给详情页设置一个委托。这是你对这类情况的一般处理方法。如果 A 调用了 B，并且 B 需要告诉 A 一些事情 -- 比如说，他需要关闭 -- 这时，你可以让 A 作为 B 的委托。我肯定你之前见过这种模式，因为它遍布于 iOS 的 API 之中。

修改 DetailViewController.h 如下：

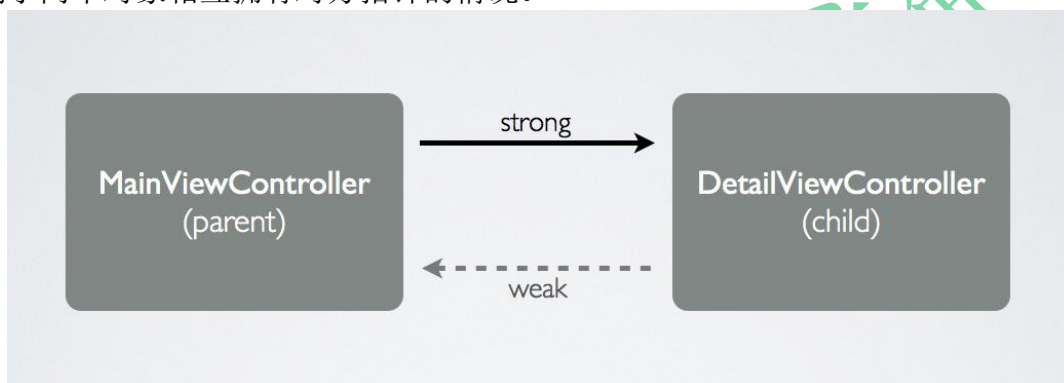
```
#import <UIKit/UIKit.h>
@class DetailViewController;
@protocol DetailViewControllerDelegate <NSObject>
- (void)detailViewController:(DetailViewController *)controller
didPickButtonWithIndex:(NSInteger)buttonIndex; @end

@interface DetailViewController : UIViewController
@property (nonatomic, weak) id <DetailViewControllerDelegate> delegate;
- (IBAction)coolAction;
- (IBAction)mehAction;
@end
```

我们已经添加了一个委托协议，它带有一个方法，同时添加了一个遵守该协议的属性。注意该属性被声明为"weak"。保持委托指针为弱指针是必须的，这样可以防止所有权循环（ownership cycles）。

你应该熟悉保留循环(retain cycle)的概念，两个对象互相保留，这会导致他们都得不到释放。这是内存泄漏的常见形式。在实现垃圾回收（GC）处理内存管理的系统中，垃圾搜集器可以识别这种循环，并会释放他们。但是 ARC 不是垃圾回收，仍然需要你自己处理所有权循环。弱指针是打断这类循环的重要工具。

MainViewController 创建了 DetailViewController，并将其展现到屏幕上。它为此对象创建了一个强引用。DetailViewController 也对委托有一个引用。它并不是真的关心委托是哪个对象，但是大多数时间会是显示它的视图控制器，也就是 MainViewController。所以这里我们碰到了两个对象相互拥有对方指针的情况。



如果这些指针都是强指针的话，就会形成所有权循环。最好能防止这种循环。父母(MainViewController)通过强指针拥有孩子(DetailViewController)。如果孩子需要指向父母的引用，不管是用委托还是别的，它应该使用一个弱指针。

因此，规则是，委托应该被声明为弱指针。大多数情况下你的属性和实例变量应该是强类型，但是这是个例外。

在 DetailViewController.m 中，同步委托属性：

```
@synthesize delegate;
```

修改动作方法：

```
- (IBAction)coolAction
{
    [self.delegate detailViewController:self didPickButtonWithIndex:0];
}
- (IBAction)mehAction
{
    [self.delegate detailViewController:self didPickButtonWithIndex:1];
}
```

在 MainViewController.h 中，将 DetailViewControllerDelegate 添加到@interface 这行：

```
@interface MainViewController : UIViewController <UITableViewDataSource, UITableViewDelegate, UISearchBarDelegate,
```

```
NSXMLParserDelegate, DetailViewControllerDelegate>
```

（这就是为什么我们要在.h 而不是.m 文件中添加#import 声明的原因。）

在 MainViewController.m 中，修改 didSelectRowAtIndexPath 来设置委托属性：

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    DetailViewController *controller = [[DetailViewController alloc] initWithNibName:@"DetailViewController" bundle:nil];
    controller.delegate = self;
    [self presentViewController:controller animated:YES
    completion:nil];
}
```

最后，在底部添加如下代码：

```
#pragma mark - DetailViewControllerDelegate
- (void)detailViewController:(DetailViewController *)controller
didPickButtonWithIndex:(NSInteger)buttonIndex
{
    NSLog(@"Picked button %d", buttonIndex);
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

这里我们只是简单的退出了视图控制器。运行程序测试一下。现在你可以点击 Cool 或者 Meh 按钮来关闭详情页。

只是为了确定 DetailViewController 被释放了，我们为其添加一个 dealloc 方法，向 DEBUG 面板输出一些东西：

```
- (void)dealloc
{
    NSLog(@"dealloc DetailViewController");
}
```

在这种情况下，你可以侥幸去设置委托为强属性（如果你不信我可以试试）。一旦 MainViewController 调用 dismissViewControllerAnimated:，它就失去了对 DetailViewController 的强引用。那时，不再有任何指针指向该对象，所以它就被释放了。

遵循下列被推荐的模式仍然是不错的主意：

- 父母指向孩子: 强指针
- 孩子指向父母: 弱指针

孩子不应该帮助父母保持存在。我们在本教程的第二部分谈论 block 的时候，将看到因为所有权导致问题的例子。

详情页不是很令人激动，但我们可以让它有趣一点，可以将被选择的艺术家的名字显示在导航条上。为 `DetailViewController.h` 添加如下代码：

```
@property (nonatomic, strong) NSString *artistName;
@property (nonatomic, weak) IBOutlet UINavigationController *navigationBar;
```

`artistName` 属性将包含被选择的艺术家的名字。之前你大概会将这个属性设为 `retain`(或者 `copy`)，所以现在它变成了 `strong`。

`navigationBar` 属性是一个 `outlet`。像以前一样，`nib` 文件中不是顶层对象的 `outlet` 应该设为 `weak`，他们会在低内存时被自动释放。

同步 `DetailViewController.m` 中的属性：

```
@synthesize artistName;
@synthesize navigationBar;
```

修改 `viewDidLoad` 方法：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationBar.topItem.title = self.artistName;
}
```

别忘了从 `nib` 文件连到导航条的 `outlet`：

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    NSString *artistName = [searchResults objectAtIndex:indexPath.row];
    DetailViewController *controller = [[DetailViewController alloc] initWithNibName:@"DetailViewController" bundle:nil];
    controller.delegate = self;
    controller.artistName = artistName;
    [self presentViewController:controller animated:YES
    completion:nil];
}
```

运行程序，你将看到艺术家的名字显示在导航条上：



对于 `NSString` 以及 `NSArray` 类对象，开发者经常会使用 `copy` 属性。这是为了在设置完该属性后不可以再修改对象值。即使 `NSString` 在创建时是不可变类型，实际传给属性的值可以是一个随后可以修改的 `NSMutableString`。

对于 ARC 你仍然可以使用 `copy` 修饰符。如果你仍然有点偏执的希望你的属性是真正的不可修改，那么可以将 `artistName` 属性的声明修改为：

```
@property (nonatomic, copy) NSString *artistName;
```

通过添加 `copy` 修饰符，这样如果我们像这样赋值：

```
controller.artistName = artistName;
```

程序首先从局部变量创建一份字符串对象的拷贝，然后将拷贝保存到属性中。除此之外，这个修饰符同 `strong` 引用工作方式相同。

让我们看看，当我们在 dealloc 中输出 artistName 和 navigationBar 的值时，发生了什么：

```
- (void)dealloc
{
    NSLog(@"dealloc DetailViewController");
    NSLog(@"artistName %@", self.artistName);
    NSLog(@"navigationBar %@", self.navigationBar);
}
```

如果你运行程序，并关闭详情页，你会看到这两个属性都还有值：

```
Artists[833:207] dealloc DetailViewController Artists[833:207] artistName 'Evans, Bill'
Artists[833:207] navigationBar <UINavigationController: 0x686d970; frame = (0 0; 320 44); autoresize = W+BM; layer = <CALayer:
0x686d9e0>>
```

但是，析构一旦完成，这些对象将被释放和析构（因为没人再持有它们）。也就是说，artistName 字符串对象将被释放，UINavigationController 对象将作为视图结构的一部分被释放。navigationBar 属性本身是弱属性，因此不包括在内存管理之中。

现在，我们有第二个页面了，我们可以从 MainViewController.m 中，测试 viewDidUnload 方法。为此，我们向该方法加入一些 NSLog() 声明：

```
- (void)viewDidUnload
{
    [super viewDidUnload];
    NSLog(@"tableView %@", self.tableView); NSLog(@"searchBar %@", self.searchBar);
    soundEffect = nil;
}
```

运行程序并打开详情页。然后从模拟器的 Hardware 菜单中，选择 Simulate Memory Warning。在 Debug 面板，你应该能看到：

```
Artists[880:207] Received memory warning.
Artists[880:207] tableView (null)
Artists[880:207] searchBar (null)
```

因为 tableView 和 searchBar 是弱属性，这些对象仅为视图结构所有。一旦主视图被卸载，它将释放所有的子视图。因为我们没有用强指针来维持 UITableView 和 UISearchBar，这些对象在 viewDidUnload 被调用之前已经释放。

只是为了看看不同点，我们将在 MainViewController.h 中使用强引用：

```
@property (nonatomic, strong) IBOutlet UITableView *tableView;
@property (nonatomic, strong) IBOutlet UISearchBar *searchBar;
```



再次运行程序，并重新引发模拟器内存警告。现在 Debug 面板将显示对象仍然存在：

```
Artists[912:207] Received memory warning.  
Artists[912:207] tableView <UITableView: 0xa816400; . . .> Artists[912:207] searchBar <UISearchBar: 0x8821360; . . .>
```

将这些 outlet 作为强引用并不一定是错的，但是你要是这么做了，就有责任在 `viewDidUnload` 中将这属性设置为 `nil`。

## 2.17. Unsafe\_unretained

我们几乎将 ARC 的基础内容都讲完了 - 在此，我还要提到一件你应该知道的事情。

除了 `strong` 和 `weak`，还有一个新的修饰符，`unsafe_unretained`。一般情况下你不会用到它。编译器不会为被声明为 `unsafe_unretained` 的变量或属性自动添加 `retain` 或 `release`。

这个新修饰符采用“unsafe”来组成它的名字，这是因为它可以指向一个不再存在的对象。如果你试着使用那种指针，你的程序极有可能崩溃。这是需要用 `NSZombieEnabled` 调试工具来找出的问题。技术上讲，如果你不使用任何 `unsafe_unretained` 属性或变量，你绝不会再向已经被释放的对象发送消息。

大多数时间里，你都希望用 `strong`，有时也用 `weak`，几乎不会用 `unsafe_unretained`。`unsafe_unretained` 仍然还存在的原因，是为了兼容 iOS4，在 iOS4 中弱指针系统不可用，另外的原因是为了一些其他的技巧。

让我们看看 `unsafe_unretained` 如何工作：

```
@property (nonatomic, unsafe_unretained)  
IBOutlet UITableView *tableView;  
@property (nonatomic, unsafe_unretained)  
IBOutlet UISearchBar *searchBar;
```

运行程序并模拟低内存警告。

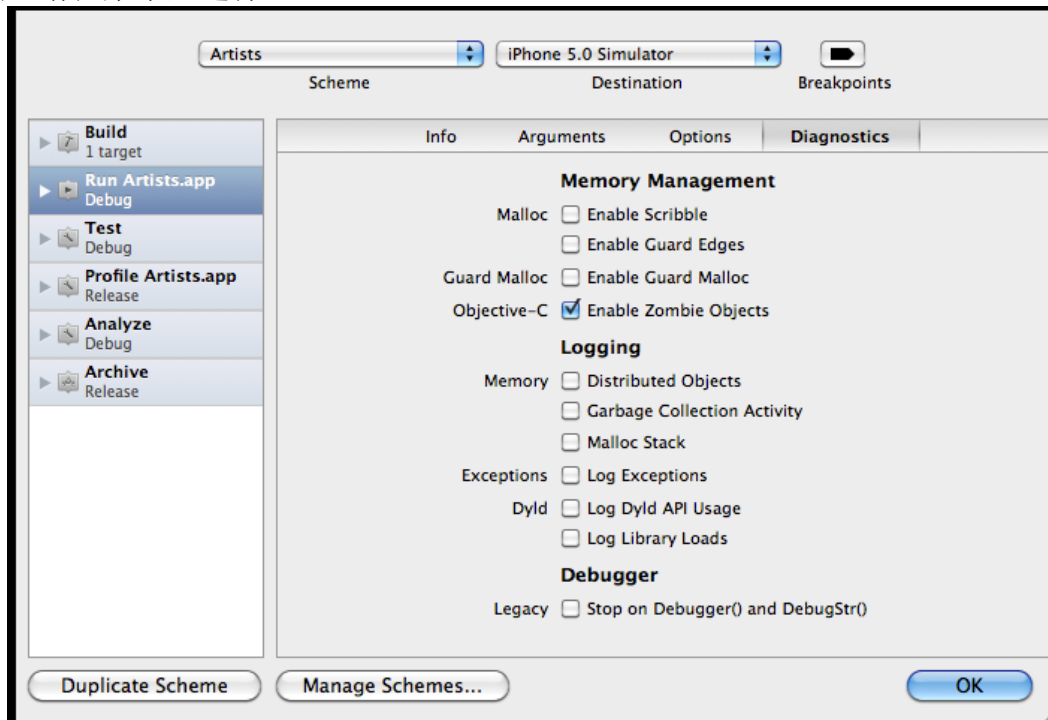
```
Artists[982:207] Received memory warning.  
Artists[982:207] *** -[UITableView retain]: message sent to deallocated instance 0x7033200
```

哎呀，程序崩溃了。`unsafe_unretained` 指针对于所指向的对象，并没有所有权。这意味着 `UITableView` 不被这个指针维持其存在，并会在 `viewDidUnload` 被调用前释放掉（它唯一的所有者是主视图）。如果这是一个弱指针，那么它的值会被置为 `nil`。记住，这是“归零”弱指针的一个很酷的特性。我们之前在 `NSLog()` 中见过它输出了“(null)”。

但是，不像一个真正的弱指针，一个 `unsafe_unretained` 指针在相关对象“死去”后并未置为 `nil`。它仍保持着原来的值。当你尝试向此对象发送一条消息时 -- 这发生在你 `NSLog()` 它

的时候 -- 你在向一个不再存在的对象发送消息。有时，这碰巧能成功，如果原来对象的内存被另一个对象所覆盖，但是通常你的程序会挂掉...这就是我们在此所见的。这也表明为什么它被称为"不安全(unsafe)"。

注意：我在这个方案（scheme）的 Diagnostics 标签中打开了 zombie 选项。要查看这个设置面板，请在菜单里选择 Product->Edit Scheme...



如果没打开这个设置，程序可能不会崩溃，或者可能在之后的某个地方崩溃。如果你想试试的话，祝你好运吧！那些 BUG 相当狡猾。

顺便提一下，这里大概是将属性还原为弱属性的好地方：

```
@property (nonatomic, weak) IBOutlet UITableView *tableView;  
@property (nonatomic, weak) IBOutlet UISearchBar *searchBar;
```

对于开启了 ARC 的程序，Enable Zombie Objects 设置(也就是 NSZombieEnabled)不再特别的有用了，所以你可以关掉它...除非在你使用 unsafe\_unretained 指针的时候！

既然 unsafe\_unretained 这么危险，那么为什么当初还要使用它呢？最大的原因是因为 iOS4。

## 2. 18. 在 iOS4 上使用 ARC

因为 ARC 基本上是 LLVM 3.0 编译器而非 iOS5 的新特征，所以你也可以将它用在 iOS4.0 及以上版本上。ARC 唯一需要 iOS5 的部分是弱指针。这意味着如果你想在 iOS4 上部署 ARC，你就不能使用弱属性或者 `__weak` 变量。

你不需要做什么特别的事情来让 ARC 工程在 iOS4 上跑起来。如果你选择一个 iOS4 的版本作为你的部署目标（Deployment Target），那么编译器将自动向工程中插入一个兼容库，来让 ARC 工程可以在 iOS4 上工作。就是这样，只要将 iOS4.x 选为部署目标就搞定了。

如果你在代码中用到了弱引用，编译器将给出下列错误：

```
"Error: the current deployment target does not support automated __weak references"
```

在 iOS4 上你不能使用 `weak` 或者 `__weak`，所以将那些弱属性替换为 `unsafe_unretained` 并将 `__weak` 变量替换为 `__unsafe_unretained`。记住，这些变量在对象被析构的时候不会被置为 `nil`，所以如果你不小心的话，你的变量可能会指向不再存在的对象。一定要用 `NSZombieEnabled` 来测试你的程序。

## 2.19. 下一步去哪？（Where To Go From Here?）

恭喜你，你已经学完了 ARC 的基础部分，并准备好了在你的新应用中使用他们 - 而且你知道如何将原有工程转移到 ARC！

如果你想学习更多关于 ARC 的知识，敬请关注下一章，我们将会覆盖以下内容：

- **在 ARC 上使用语句块。** 语句块的法则有点变动。你需要特别小心避免所有权循环，他是仅有的连 ARC 也无法自动处理的内存问题。
- **如何在 ARC 中实现单例（singleton）。** 你不能再重写 `retain` 和 `release` 方法来确保单例类仅有一个实例了，那如何在 ARC 上实现单例呢？
- **更多关于 autorelease 的内容。** 包含 `autorelease` 和 `autorelease pool` 的所有内容。
- **使用 ARC 和 cocos2D 编写游戏。** 我同样会解释 ARC 如何适配 Object-C++，如果你的游戏使用了 Box2D 物理引擎的话，你需要了解这个。
- **静态库。** 如何让你自己的静态库区分开工程中的 ARC 和非 ARC 的部分。



点击这里访问: [DevDiv.com](http://DevDiv.com) 移动开发论坛