iOS 5 ARC 完全指南

书籍：《iOS 5 By Tutorials》

翻译：Kevin

联系：support@gungyi.com

网站：http://www.gungyi.com

鸣谢：GungYi 移动应用开发

CocoaChina 社区

时间：2012-02-09

目录

ARC 完全指南

iOS 5 最显著的变化就是增加了 Automatic Reference Counting（自动引用计数）。ARC 是新 LLVM 3.0 编译器的特性，完全消除了手动内存管理的烦琐。在你的项目中使用 ARC 是非常简单的，所有的编程都和以前一样，除了你不再调用 retain, release, autorelease。启用 ARC 之后，编译器会自动在适当的地方插入适当的 retain, release, autorelease 语句。你不再需要担心内存管理，因为编译器为你处理了一切。注意 ARC 是编译器特性，而不是 iOS 运行时特性（除了 weak 指针系统），它也不是其它语言中的垃圾收集器。因此 ARC 和手动内存管理性能是一样的，有些时候还能更加快速，因为编译器还可以执行某些优化。

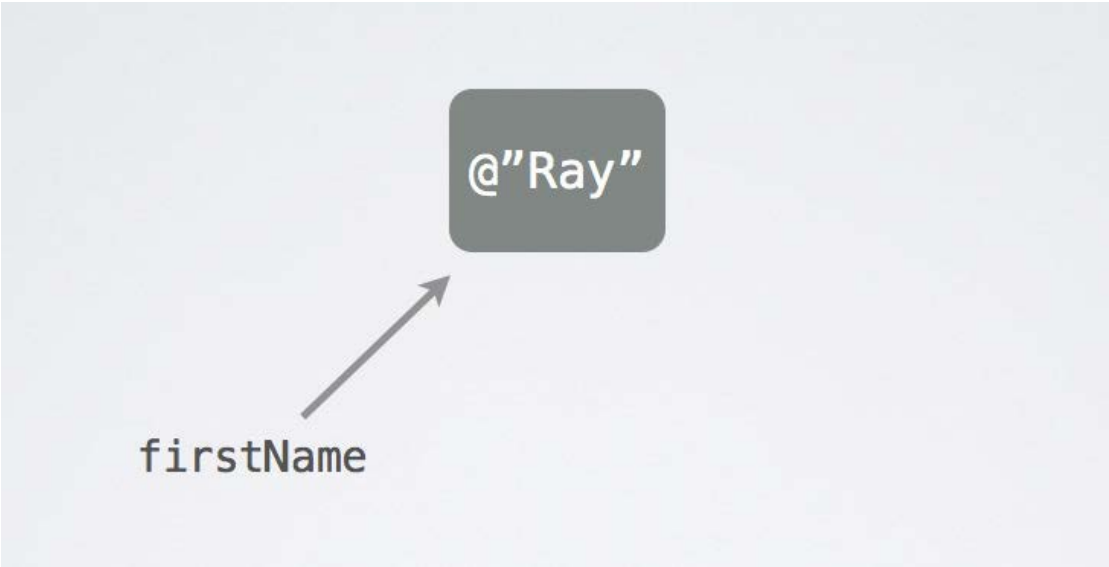
指针保持对象的生命

ARC 的规则非常简单：**只要还有一个变量指向对象，对象就会保持在内存中。**当指针指向新值，或者指针不再存在时，相关联的对象就会自动释放。这条规则对于实例变量、synthesize 属性、本地变量都是适用的。

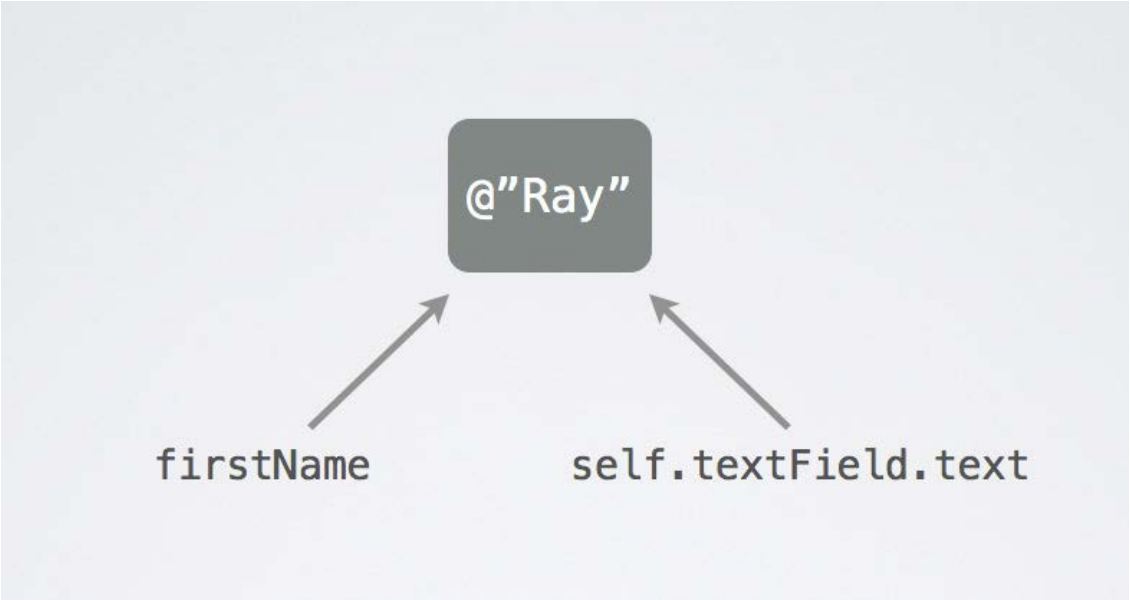
我们可以按“所有权”（ownership）来考虑 ARC 对象：

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

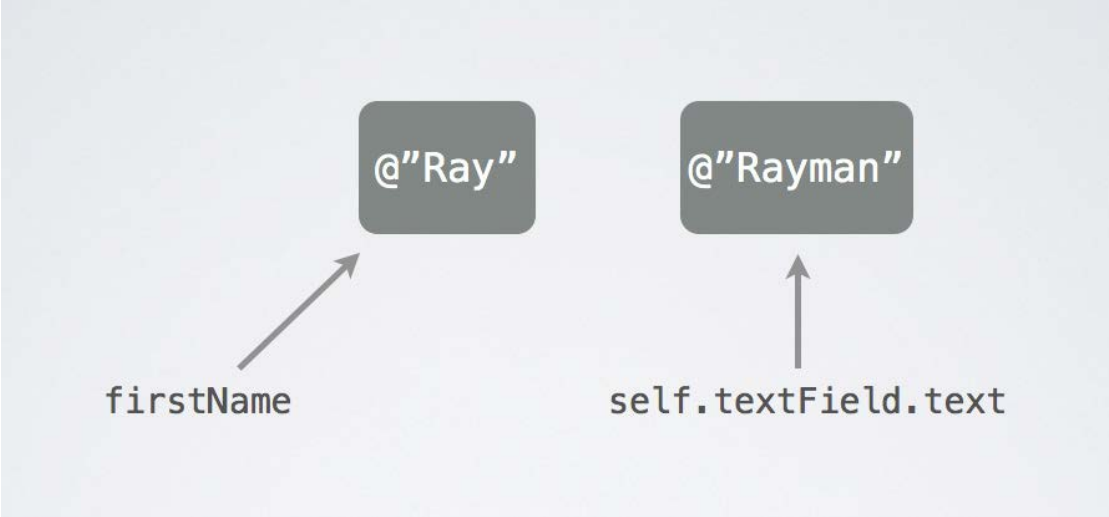
firstName 变量成为 NSString 对象的指针，也就是拥有者，该对象保存了文本输入框的内容。



一个对象可以有多个拥有者，在上面例子中，UITextField 的 text 属性同样也是这个 String 对象的拥有者，也就是有两个指针指向同一个对象：



随后用户改变了输入框的文本，此时 text 属性就指向了新的 String 对象。但原来的 String 对象仍然还有一个所有者（firstName 变量），因此会继续保留在内存中。

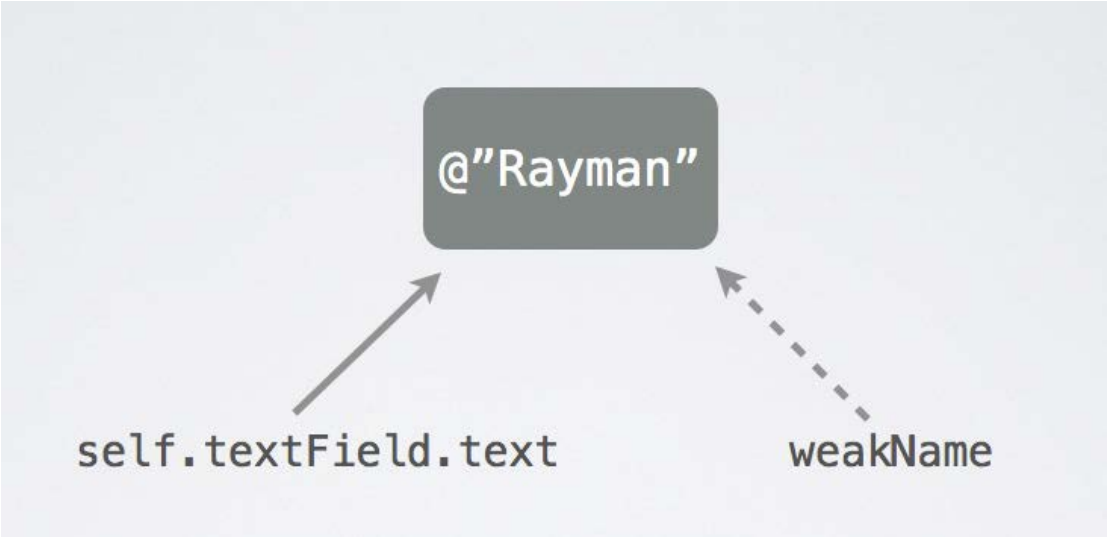


只有当 firstName 获得新值，或者超出作用域（本地变量方法返回时、实例变量对象释放时），String 对象不再拥有任何所有者，retain 计数降为 0，这时对象会被释放。

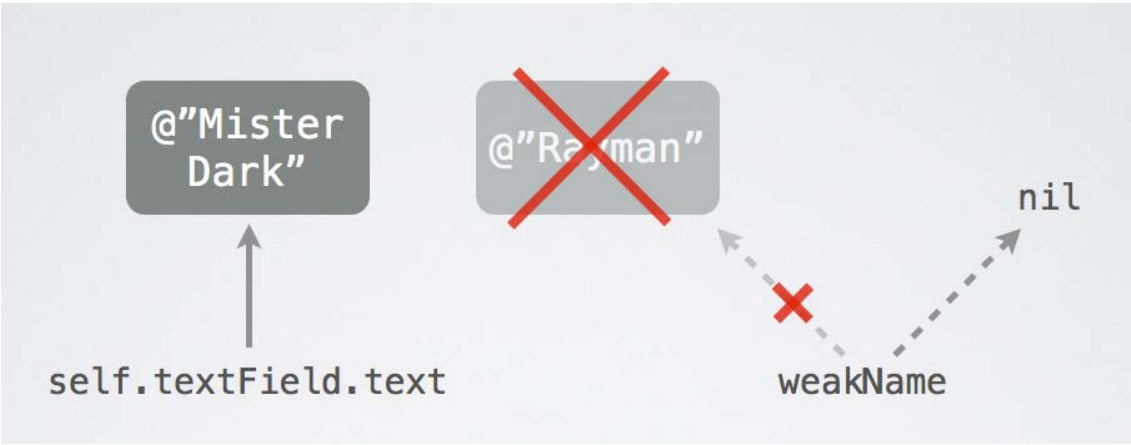
我们称 firstName 和 textField.text 指针为“strong”，因为它们能够保持对象的生命。默认所有实例变量和本地变量都是 strong 类型的指针。

另外还有一种“weak”指针，weak 变量仍然指向一个对象，但不是对象的拥有者：

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

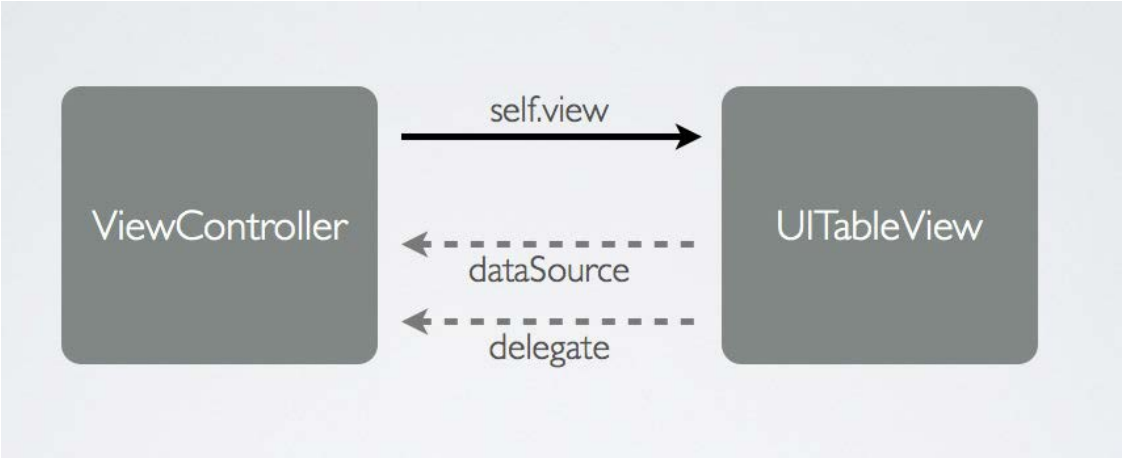


weakName 变量和 textField.text 属性都指向一个 String 对象，但 weakName 不是拥有者。如果文本框的内容发生变化，则原先的 String 对象就没有拥有者，会被释放，此时 weakName 会自动变成 nil，称为“zeroing” weak pointer：



weak 变量自动变为 nil 是非常方便的，这样阻止了 weak 指针继续指向已释放对象。“摇摆指针”和“zombies”会导致非常难于寻找的 Bug。zeroing weak pointer 消除了类似的问题。

weak 指针主要用于“父-子”关系，父亲拥有一个儿子的 strong 指针，因此是儿子的所有者；但为了阻止所有权回环，儿子需要使用 weak 指针指向父亲。典型例子是 delegate 模式，你的 View Controller 通过 strong 指针拥有一个 UITableView，Table view 的 data source 和 delegate 都是 weak 指针，指向你的 View Controller。



注意下面代码是有问题的：

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

String 对象没有拥有者（weak 指针），在创建之后就会被立即释放。Xcode 会给出警告（"Warning: assigning retained object to weak variable; object will be released after assignment"）

变量默认就是__strong 类型的，因此一般我们对于 strong 变量不加__strong 修饰，以下两者是等价的：

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

属性可以是 strong 或 weak，写法如下：

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

有了 ARC，我们的代码可以清晰很多，你不再需要考虑什么时候 retain 或 release 对象。唯一需要考虑的是对象之间的关联，也就是哪个对象拥有哪个对象？

以下代码在 ARC 之前是不可能的，在手动内存管理中，从 Array 中移除一个对象会使对象不可用，对象不属于 Array 时会立即被释放。随后 NSLog()打印该对象就会导致应用崩溃。

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

在 ARC 中这段代码是完全合法的，因为 obj 变量是一个 strong 指针，它成为了对象的拥有者，从 Array 中移除该对象也不会导致对象被释放。

ARC 也有一些限制。首先 ARC 只能工作于 Objective-C 对象，如果应用使用了 Core Foundation 或 malloc()/free()，此时需要你来管理内存。此外 ARC 还有其它一些更为严格的语言规则，以确保 ARC 能够正常地工作。不过总的来说，ARC 无疑利大于弊！

虽然 ARC 管理了 retain 和 release，但并不表示你完全不需要处理内存管理的问题。因为 strong 指针会保持对象的生命，某些情况下你仍然需要手动设置这些指针为 nil，否则可能导致应用内存不足。无论

何时你创建一个新对象时，都需要考虑谁拥有该对象，以及这个对象需要存活多久。

毫无疑问，ARC 是 Objective-C 的未来！Apple 鼓励开发者将手动内存管理迁移至 ARC，同时新项目也推荐使用 ARC。ARC 可以产生更简洁的代码，和更健壮的应用。有了 ARC，内存相关的崩溃已经成为过去！

不过现在正处于手动内存向自动内存管理的过渡期，你仍然会经常遇到与 ARC 尚不兼容的代码（你自己的代码或第三方库）。幸运的是，你可以在同一个项目中组合使用 ARC 和非 ARC 代码，同时 Xcode 还提供自动迁移工具。

ARC 还能很好地结合 C++使用，这对游戏开发是非常有帮助的。对于 iOS 4，ARC 有一点点限制（weak pointer system），但也没太大关系。

Xcode 的 ARC 自动迁移

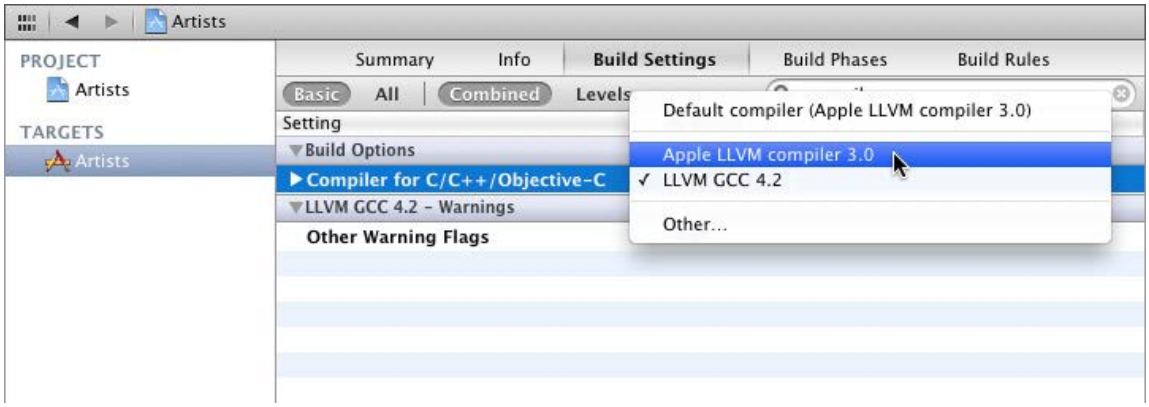
要启用一个项目的 ARC，你有以下几种选择：

1. Xcode 带了一个自动转换工具，可以迁移源代码至 ARC
2. 你可以手动转换源文件
3. 你可以在 Xcode 中禁用某些文件使用 ARC，这点对于第三方库非常有用。

Xcode 的自动迁移工具

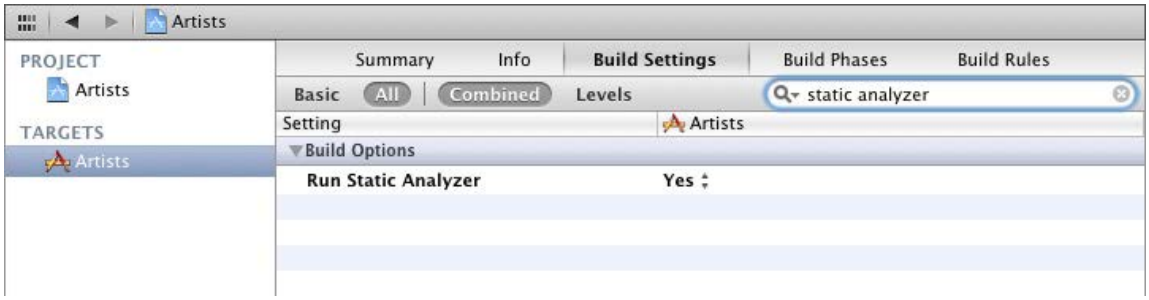
ARC 是 LLVM 3.0 编译器的特性，而现有工程可能使用老的 GCC 4.2 或 LLVM-GCC 编译器，因此首先需要设置使用 LLVM 3.0 编译器：

Project Settings -> target -> Build Settings，在搜索框中输入 compiler，就可以列出编译器选项设置：



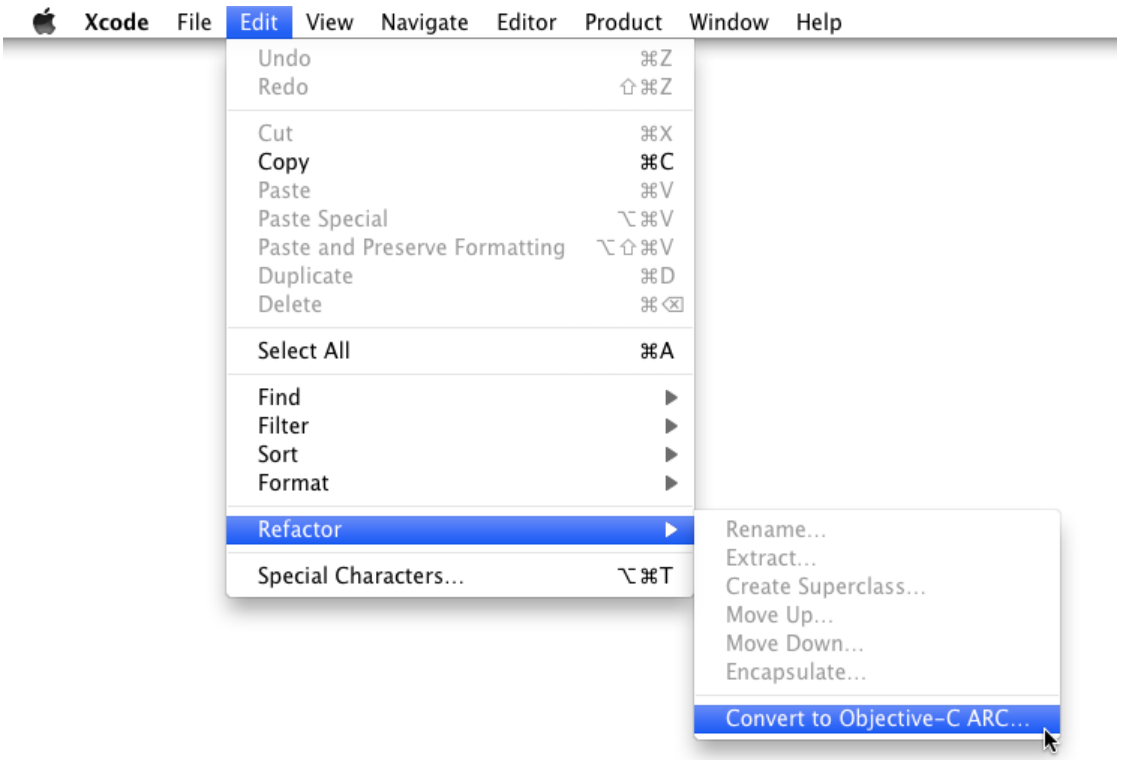
另外最好也选上 Warnings 中的 Other Warning Flags 为 -Wall，这样编译器就会检查所有可能的警告，有助于我们避免潜在的问题。

同样，Build Options 下面的 Run Static Analyzer 选项也最好启用，这样每次 Xcode 编译项目时，都会运行静态代码分析工具来检查我们的代码。

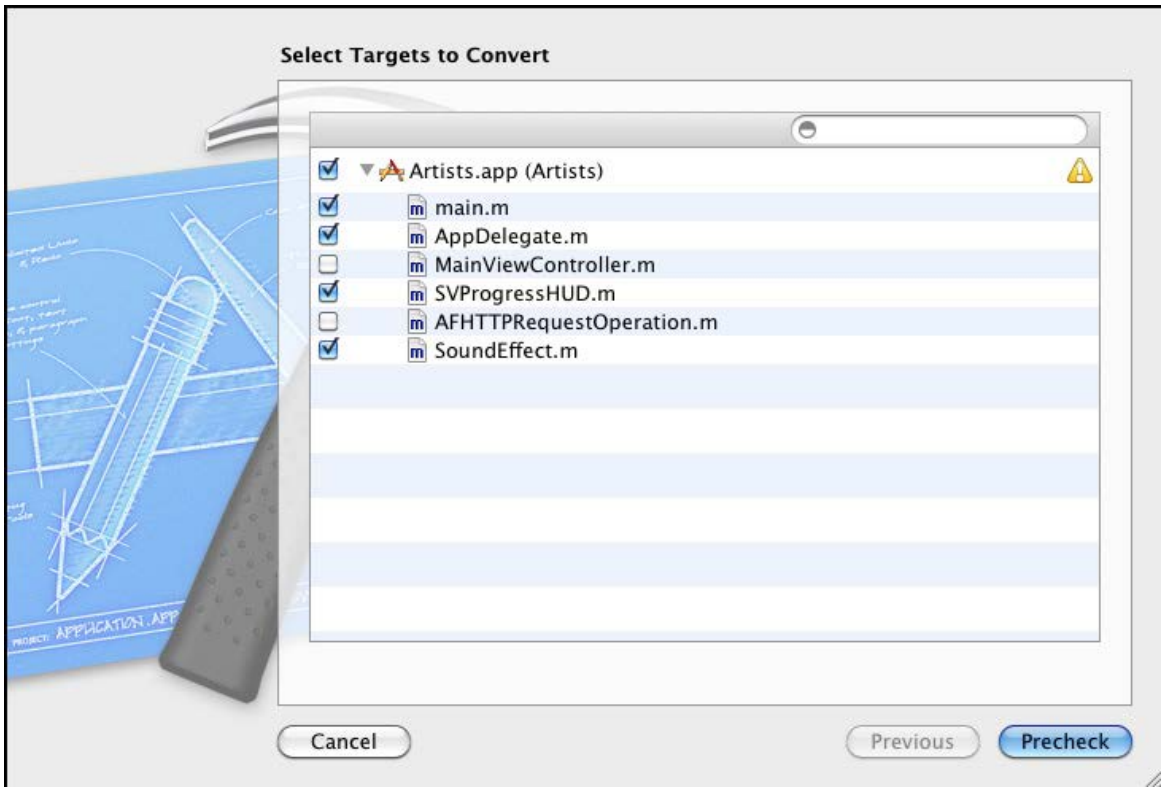


Build Settings 下面，选择“All”，搜索框输入"automatic"，可以设置"Objective-C Automatic Reference Counting"选项为 Yes，不过 Xcode 自动转换工具会自动设置这个选项，这里只是告诉你如何手动设置而已。

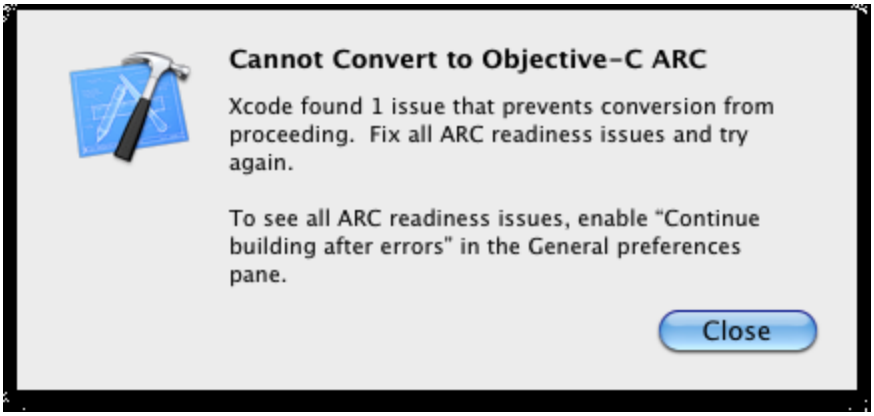
Xcode 的 ARC 自动转换工具：Edit\Refactor\Convert to Objective-C ARC



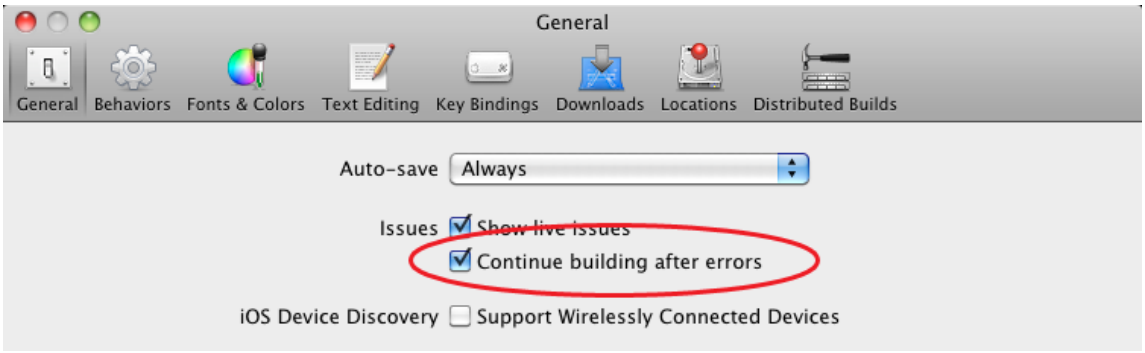
Xcode 会显示一个新窗口，让你选择哪些文件需要转换：



点击Precheck按钮，Xcode可能会提示项目不能转换为ARC，需要你准备好转换：

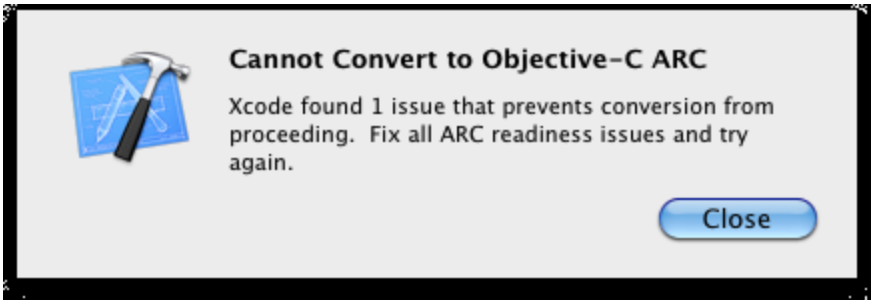


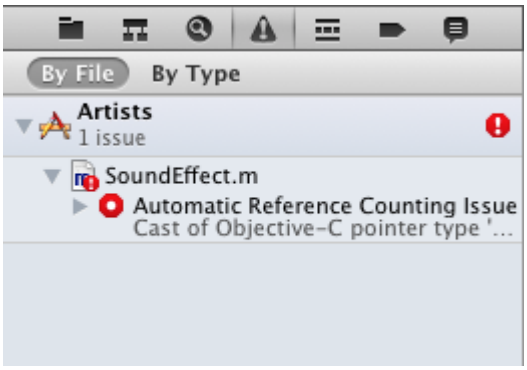
打开 Xcode Preferences 窗口，选择 General 标签，启用 Continue building after errors 选项：



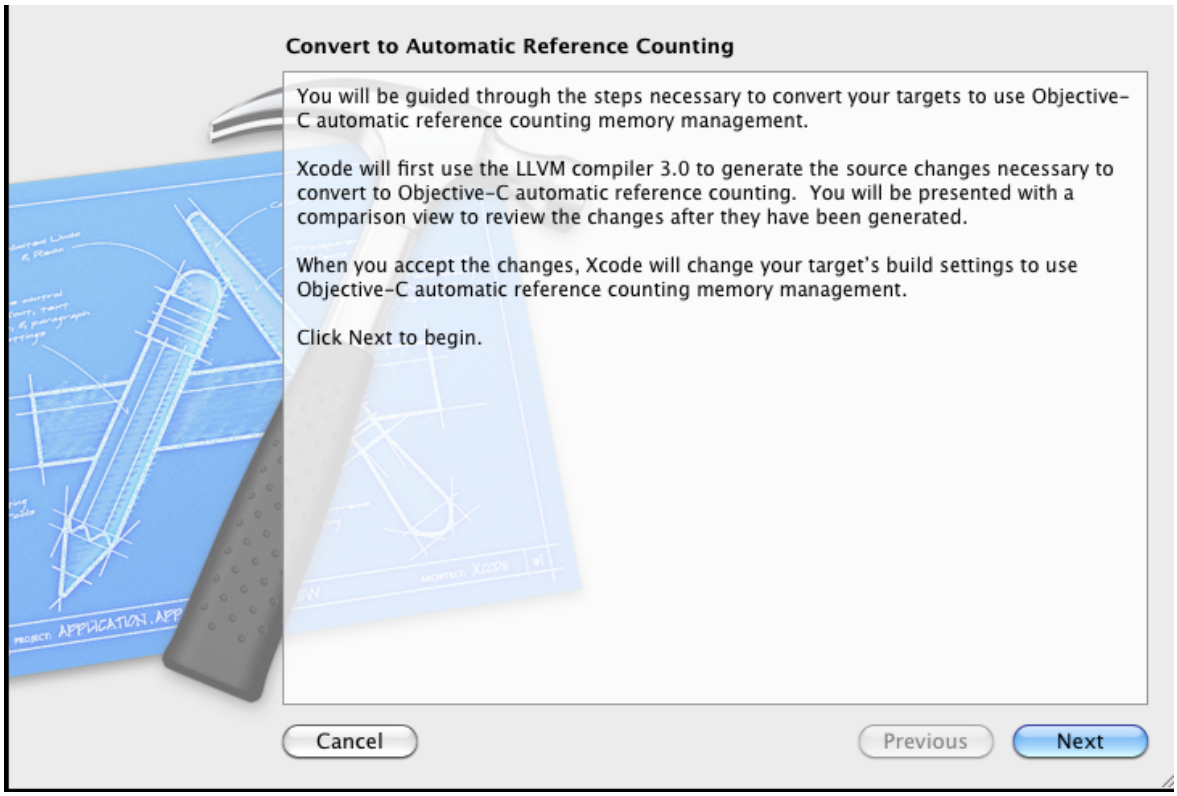
再次执行 Edit\Refactor\Convert to Objective-C ARC，点击 Precheck，Xcode 可能还会提示错误，此时需要你手工修改某些源代码，这些错误是 Xcode 无法自动转换的，主要是"toll-free bridged"类型错误，如转换 NSURL 为 CFURLRef 类型，这些转换前面需要加上__bridge、__bridge_transfer、或 __bridge_retained 关键字

"toll-free bridged"允许 Core Foundation 对象和 Objective-C 对象之间进行转换，但是 ARC 必须知道由谁来负责释放这些对象，因此才增加了__bridge、__bridge_transfer、__bridge_retained 三个关键字。后面会详细说明它们各自的用法。

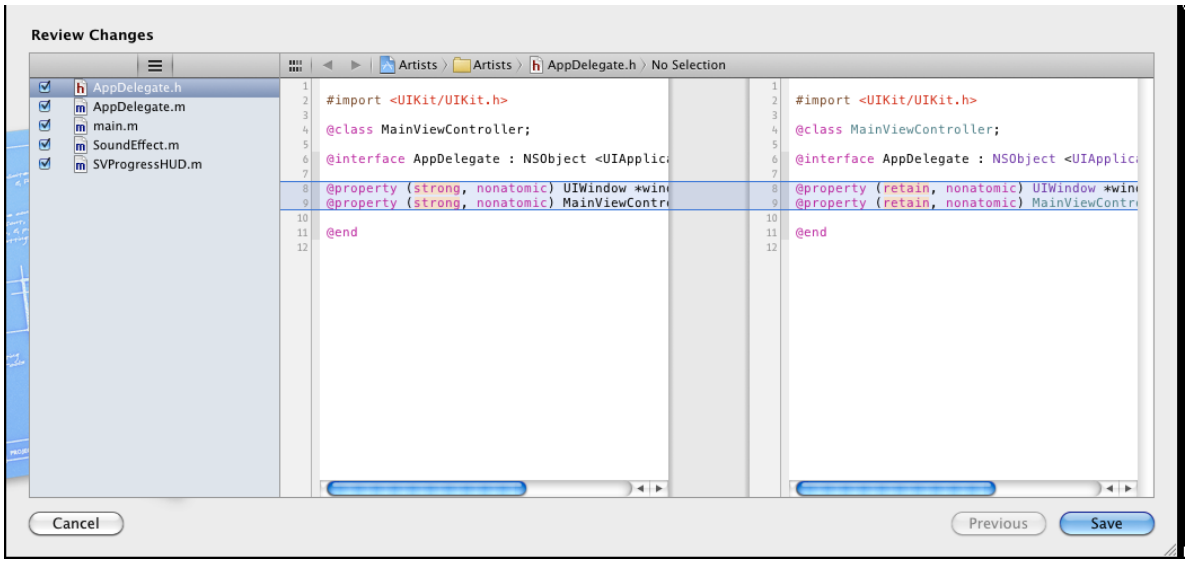




解决了这个问题之后，再次执行 Edit\Refactor\Convert to Objective-C ARC，这时 Xcode 会提示以下窗口：



点击 Continue，几秒钟后，Xcode 会提示所有文件的转换预览，显示源文件的所有改变。左边是修改后的文件，右边是原始文件。在这里你可以一个文件一个文件地查看 Xcode 的修改，以确保 Xcode 没有改错你的源文件：



转换后的其它问题

最后，Xcode 移除所有 retain、release、autorelease 调用之后，可能会导致代码出现其它警告、无效语法等，这些都需要你手工进行修改。如：

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

修改为：

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

这时候逗号语句中间的部分就不需要了，我们可以直接手动修改为：

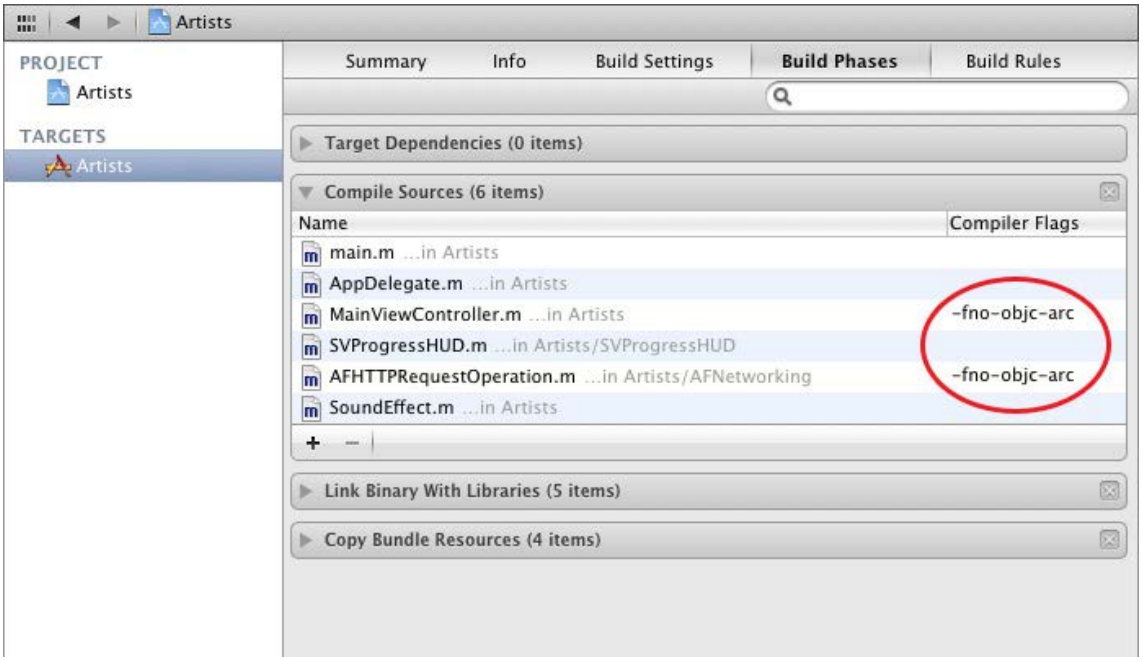
```
if(fadeOutTimer != nil)
    [fadeOutTimer invalidate], fadeOutTimer = nil;
```

注意：Xcode 的自动转换工具最好只使用一次，多次使用可能会出现比较诡异的问题。假如你第一次转换没有转换所有的文件，当你稍后试图再次转换剩余的文件时，Xcode 实际上不会执行任何转换操作。因此最好一次就完成转换，没有转换的文件可以考虑手工进行修改。

禁止某些文件的 ARC

对于某些我们不希望使用 ARC 的文件，例如第三方库源文件，可以在 Project Settings -> Build Phases 中，对这些文件选中 -fno-objc-arc 标志。这样 Xcode 编译项目时，这些文件就不会使用 ARC。

由于开发者很可能不会对所有文件都启用 ARC，Xcode 允许项目结合使用 ARC 和非 ARC 的源文件。你当然可以按上面的设置，对指定文件设置 -fno-objc-arc 标志。但最简单的方法是直接使用 Xcode 的 ARC 转换工具，取消选中那些不希望进行 ARC 转换的源文件，这样 Xcode 会自动对这些文件设置 -fno-objc-arc 标志。如果有许多文件都不使用 ARC，使用这个方法无疑会减少很多时间。



ARC 自动迁移的常见问题

在 ARC 自动迁移过程中，LLVM 3.0 可能会报出许多错误或警告，你需要手工进行改变，Xcode 才能继续迁移

"Cast ... requires a bridged cast"
Toll-Free Bridging 对象转换时，需要使用 __bridge, __bridge_transfer, __bridge_retained 关键字，使用哪个要看具体情况，后面会专门讲解。

"Receiver type 'X' for instance message is a forward declaration"
在.h 头文件中，我们经常使用前向声明 @class MyView，如果在.m 实现文件中，我们没有#import "MyView.h"，Xcode 就会报错，加上就好了。

"Switch case is in protected scope"

```
switch (X)
{
```

```
case Y:
    NSString *s = ...;
    break;
}
```

类似这样的代码就会报上面错误，ARC 不允许这样的代码，指针变量需要定义在 switch 语句外面，或者使用{}定义一个新的块：

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

这样 ARC 才能确定变量的作用域，从而在适当的时候释放该变量。

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

例如以下代码：

```
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
// . . . do calculations . . .
NSArray* sortedResults =
    [[filteredResults
sortedArrayUsingSelector:@selector(compare:)] retain];
[pool release];
return [sortedResults autorelease];
```

会转换为：

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

但是这段代码是非法的，因为 return sortedResults;超出了变量的作用域。解决办法是将 NSArray* sortedResults;变量定义放在 NSAutoreleasePool 之前，这样 Xcode 就能转换出正常的@autoreleasepool 代码。

"ARC forbids Objective-C objects in structs or unions"
使用 ARC 之后，C Struct 中不能使用 Objective-C 对象，下面代码就是非法的：

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

解决办法是定义一个 Objective-C 类，不使用 C Struct。

属性 property

对于.h 头文件，Xcode 主要是将属性定义由 retain 变为 strong，这些属性是类对外的接口，因此定义在.h 文件中：

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

变为

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

在 ARC 之前，开发者经常会在.m 实现文件中使用 class extension 来定义 private property，如下：

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

这样做主要是简化实例对象的手动内存管理，让 property 的 setter 方法自动管理原来对象的释放，以及新对象的 retain。但是有了 ARC，这样的代码就不再需要了。一般来说，仅仅为了简化内存管理，是不再需要使用 property 的，虽然你仍然可以这样做，但直接使用实例变量是更好的选择。只有那些属于 public 接口的实例变量，才应该定义为 property。

我们可以直接在.m 类实现中定义 private 实例变量：

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

我们在使用时，虽然没有定义 property，也可以直接

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

如果你觉得这很别扭，也可以使用

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

如果你还是觉得应该定义 property，那就定义一个吧，反正也没什么害处。

作为 property 的最佳实践，如果你定义某个东西为 property，则你应该在任何地方都按属性来使用它。唯一例外的是 init 方法、自定义的 getter 和 setter 方法。因此很多时候我们会这样写 synthesize 语句：

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

实际上_propertyName 实例变量甚至可以不定义，编译器会自动为 property 定义 "_*" 的实例变量

IBOutlet

在 ARC 中,所有*outlet*属性都推荐使用 weak,这些 view 对象已经属于 View Controller 的 view hierarchy，不需要再次定义为 strong（ARC 中效果等同于 retain）。唯一应该使用 strong 的 outlet 是 File's Owner，连接到 nib 的顶层对象。

将 outlet 定义为 weak 的优点是简化了 viewDidUnload 方法的实现：

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

现在可以简化为：

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

因为 tableView 和 searchBar 这两个 property 定义为 weak，当它们指向的对象被释放时，这两个变量会自动设置为 nil。

当 iOS App 接收到低内存警告时,View Controller 的 main view 会被 unload，同时会释放所有 subview。这时 UITableView 和 UISearchBar 对象会自动释放，zeroing weak pointer system 就会自动设置 self.tableView 和 self.searchBar 为 nil。因此不需要在 viewDidUnload 中再次设置为 nil，实际上当 viewDidUnload 被调用时，这两个属性已经是 nil 了。

这并不意味着你可以不需要 viewDidUnload，只要你保持一个对象的指针，对象就会存活。当你不需要某个对象时，可以手动设置指针为 nil。如上面示例代码中的 soundEffect = nil; **viewDidUnload()方法里面需要设置所有非 outlet 变量为 nil，同样还有 didReceiveMemoryWarning()方法。**

property 的修饰符总结如下：

- **strong**：等同于"retain"，属性成为对象的拥有者
- **weak**:属性是 weak pointer,当对象释放时会自动设置为 nil,记住 Outlet 应该使用 Weak
- **unsafe_unretained**：等同于之前的"assign"，只有 iOS 4 才应该使用
- **copy**：和之前的 copy 一样，复制一个对象并创建 strong 关联
- **assign**：对象不能使用 assign，但原始类型（BOOL、int、float）仍然可以使用

readonly property

在 ARC 之前，我们可以如下定义一个 readonly property：

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

这会隐式地创建一个 assign property，这种用法对于 readonly 值来说是适当的。毕竟你何必对只读数据进行 retain 呢？但上面在 ARC 中会报错：

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

你必须显式地使用 strong, weak 或 unsafe_unretained，多数情况下使用 strong 是正确的选择：

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

对于 readonly property，我们应该总是使用 self.propertyName 来访问实例变量（除了 init 和自定义的 getter 和 setter 方法）。否则直接修改实例变量会混淆 ARC 并导致奇怪的 Bug。正确的方法是使用 class extension 重新定义 property 为 readwrite：

.h 文件：

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

.m 文件：

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

autorelease、release、retain 调用

对于.m 文件，Xcode 会移除所有的 autorelease、retain、release 调用

```
self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]] autorelease];
```

修改为

```
self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
mainScreen] bounds]];

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

修改为

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

dealloc 方法

另外启用 ARC 之后，dealloc 方法在大部分时候都不再需要了，因为你不能调用实例对象的 release 方法，也不能调用[super dealloc]。假如原先的 dealloc 方法只是释放这些对象，Xcode 就会把 dealloc 方法完全移除。你不再需要手动释放任何实例变量。

如果你的 dealloc 方法处理了其它资源（非内存）的释放，如定时器、Core Foundation 对象,则你仍然需要在 dealloc 方法中进行手动释放,如 CFRelease(), free()等。这时 Xcode 会保留 dealloc 方法，但是移除所有的 release 和[super dealloc]调用。如下：

```
- (void)dealloc
{
    AudioServicesDisposeSystemSoundID(soundID);
}
```

AutoreleasePool

ARC 仍然保留了 AutoreleasePool，但是采用了新的 Block 语法，于是我们的 main 函数会如下修改：

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

会修改变：

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

Toll-Free Bridging

当你在 Objective-C 和 Core Foundation 对象之间进行转换时，就需要使用 Bridge cast。

今天的多数应用很少需要使用 Core Foundation，大多数工作都可以直接使用 Objective-C 类来完成。但是某些底层 API，如 Core Graphics 和 Core Text，都基于 Core Foundation，而且不太可能会有 Objective-C 的版本。幸运的是，iOS 的设计使得这两种类型的对象非常容易转换。

例如 NSString 和 CFStringRef 就可以同等对待，在任何地方都可以互换使用，背后的设计就是 toll-free bridging。在 ARC 之前，只需要使用一个简单的强制类型转换即可：

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

当然，alloc 分配了 NSString 对象，你需要在使用完之后进行释放，注意是释放转换后的 CFStringRef 对象：

```
CFRelease(s1);
```

反过来，从 Core Foundation 到 Objective-C 的方向也类似：

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

现在我们使用了 ARC，情况变得不一样！以下代码在手动内存管理中是完全合法的，但在 ARC 中却存在问题：

```
- (NSString *)escape:(NSString *)text
{
    return [(NSString *)CFURLCreateStringByAddingPercentEscapes(
            NULL,
            (CFStringRef)text,
            NULL,
            (CFStringRef)@"!*'();:@&=+$,/?%#[]", // 这里不需要 bridging
casts，因为这是一个常量，不需要释放!
            CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncodin
g))
           autorelease];
}
```

首先需要移除 autorelease 调用。然后编译器还会报两个类型转换错误：

- 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

错误分别来自以下两行代码：

```
(NSString *)CFURLCreateStringByAddingPercentEscapes(...)
(CFStringRef)text
```

编译器必须知道由谁来负责释放转换后的对象，如果你把一个 NSObject 当作 Core Foundation 对象来使用，则 ARC 将不再负责释放该对象。但你必须明确地告诉 ARC 你的这个意图，编译器没办法自己做主。同样如果你创建一个 Core Foundation 对象并把它转换为 NSObject 对象，你也必须告诉 ARC 占据对象的所有权，并在适当的时候释放该对象。这就是所谓的 bridging casts。

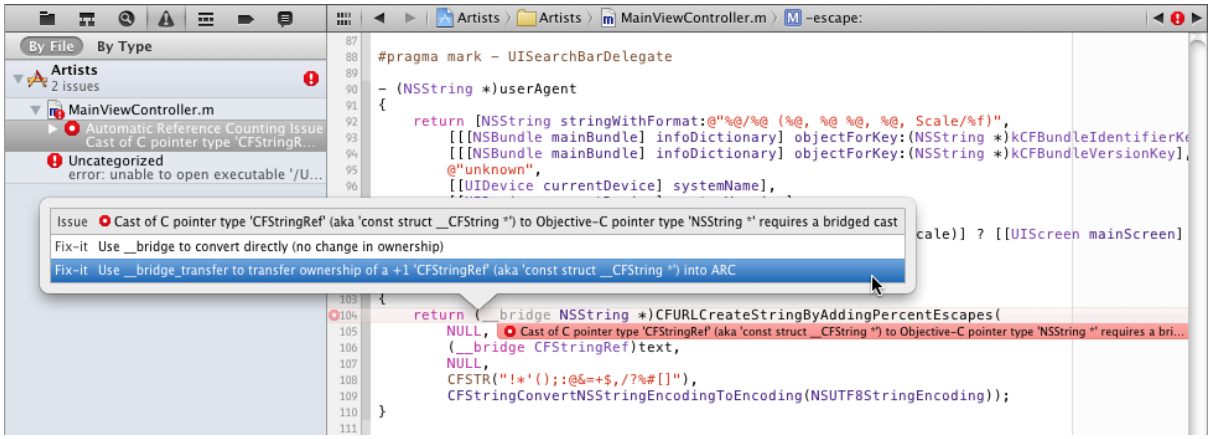
CFURLCreateStringByAddingPercentEscapes()函数的参数需要两个 CFStringRef 对象，其中常量 NSString 可以直接转换，因为不需要进行对象释放；但是 text 参数不一样，它是传递进来的一个 NSString 对象。而函数参数和局部变量一样，都是 strong 指针，这种对象在函数入口处会被 retain，并且对象会持续存在直到指针被销毁（这里也就是函数返回时）。

对于 text 参数，我们希望 ARC 保持这个变量的所有权，同时又希望临时将它当作 CFStringRef 对象来使用。这种情况下可以使用__bridge 说明符，它告诉 ARC 不要更改对象的所有权，按普通规则释放该对象即可。

多数情况下，Objective-C 对象和 Core Foundation 对象之间互相转换时，我们都应该使用__bridge。但是有时候我们确实需要给予 ARC 某个对象的所有权，或者解除 ARC 对某个对象的所有权。这种情况下我们就需要使用另外两种 bridging casts：

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

在上面代码中，"`return (NSString *)CFURLCreateStringByAddingPercentEscapes`"，编译器弹出的修复提示有两个：



两个解决办法：__bridge 和 __bridge_transfer，正确的选择应该是 __bridge_transfer。

因为 CFURLCreateStringByAddingPercentEscapes() 函数创建了一个新的 CFStringRef 对象，当然我们要的是 NSString 对象，因此我们使用了强制转换。实际上我们真正想要做的是：

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

从 CFURLCreateStringByAddingPercentEscapes 函数的 create 可以看出，函数会返回一个 retain 过的对象。某个人需要负责在适当的时候释放该对象，如果我们不把这个对象返回为 NSString，则通常我们需要自己调用：

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

不过 ARC 只能作用于 Objective-C 对象，不能释放 Core Foundation 对象。因此这里你仍然需要调用 CFRelease()来释放该对象。

这里我们的真实意图是:转换新创建的 CFStringRef 对象为 NSString 对象，并且当我们不再需要使用这个 NSString 对象时，ARC 能够适当地释放它。

因此我们使用 __bridge_transfer 告诉 ARC:"嘿！ARC，这个 CFStringRef 对象现在是一个 NSString 对象了，我希望你来销毁它，我这里就不调用 CFRelease()来释放它了"。

如果我们使用 __bridge，就会导致内存泄漏。ARC 并不知道自己应该在使用完对象之后释放该对象，也没有人调用 CFRelease()。结果这个对象就会永远保留在内存中。因此选择正确的 bridge 说明符是至关重要的。

为了代码更加可读和容易理解，iOS 还提供了一个辅助函数：CFBridgingRelease()。函数所做事情和 __bridge_transfer 强制转换完全一样，但更加简洁和清晰。CFBridgingRelease() 函数定义为内联函数，因此不会导致额外的开销。函数之所以命名为 CFBridgingRelease()，是因为一般你会在需要使用 CFRelease()释放对象的地方，调用 CFBridgingRelease()来传递对象的所有权。

因此最后我们的代码如下：

```
- (NSString *)escape:(NSString *)text
{
    return CFBridgingRelease(CFURLCreateStringByAddingPercentEscapes(
            NULL,
            (__bridge CFStringRef)text,
            NULL,
            CFSTR("!*'();:@&=+$,/?%#[]"),
            CFStringConvertNSStringEncodingToEncoding(NSUTF8StringEncodin
g)));
}
```

另一个常见的需要 CFBridgingRelease 的情况是 AddressBook framework：

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

只要你调用命名为 Create, Copy, Retain 的 Core Foundation 函数，你都需要使用 CFBridgingRelease() 安全地将值传递给 ARC。

__bridge_retained 则正好相反，假设你有一个 NSString 对象，并且要将它传递给某个 Core Foundation API，该函数希望接收这个 string 对象的所有权。这时候你就不希望 ARC 也去释放该对象，否则就会对同一对象释放两次，而且必将导致应用崩溃！换句话说，使用 __bridge_retained 将对象的所有权给予 Core Foundation，而 ARC 不再负责释放该对象。

如下面例子所示：

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

一旦（__bridge_retained CFStringRef）转换完成，ARC 就不再负责释放该对象。如果你在这里使用 __bridge，应用就很可能会崩溃。ARC 可能在 Core Foundation 正在使用该对象时，释放掉它。

同样__bridge_retained 也有一个辅助函数：CFBridgingRetain()。从名字就可以看出，这个函数会让 Core Foundation 执行 retain，实际如下：

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

现在你应该明白了，上面例子的 CFRelease()是和 CFBridgingRetain()对应的。你应该很少需要使用__bridge_retained 或 CFBridgingRetain()。

__bridge 转换不仅仅局限于 Core Foundation 对象，某些 API 使用 void *指针作为参数，允许你传递任何东西的引用：Objective-C 对象、Core Foundation 对象、malloc()内存缓冲区等等。void *表示这是一个指针，但实际的数据类型可以是任何东西！

要将 Objective-C 对象和 void *互相转换，你也需要使用__bridge 转换，如下：

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

在 animation delegate 方法中，你再将对象强制转回来：

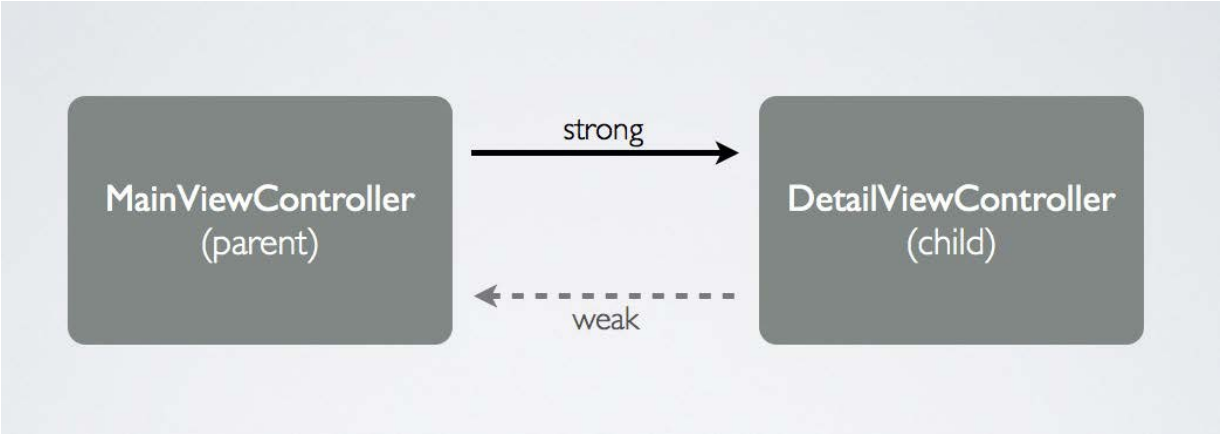
```
- (void)animationDidStart:(NSString *)animationID
context:(void *)context
{
    MyClass *myObject = (__bridge MyClass *)context;
    . . .
}
```

总结：

- 使用 CFBridgingRelease()，从 Core Foundation 传递所有权给 Objective-C；
- 使用 CFBridgingRetain()，从 Objective-C 传递所有权给 Core Foundation；
- 使用__brideg，表示临时使用某种类型，不改变对象的所有权。

Delegate 和 Weak Property

使用 Delegate 模式时，通常我们会使用 weak property 来引用 delegate，这样可以避免所有权回环。



你可能已经熟悉 retain 回环的概念，两个对象互相 retain 时，会导致两个对象都无法被释放，这也是内存泄漏的常见原因之一。垃圾收集器（GC）可以识别并处理 retain 回环的情况，但 ARC 不是 GC，没办法处理所有权回环，因此需要你使用 weak 指针来避免。

```
DetailViewController *controller = [[DetailViewController alloc]
initWithNibName:@"DetailViewController" bundle:nil];
controller.delegate = self;
[self presentViewController:controller animated:YES
completion:nil];
```

在上面代码中，MainViewController 创建一个 DetailViewController，并调用 presentViewController 将 view 呈现出来，从而拥有了一个 strong 指针指向创建的 DetailViewController 对象。反过来，DetailViewController 也通过 delegate 拥有了一个指向 MainViewController 的 weak 指针。

当 MainViewController 调用 dismissViewControllerAnimated:时，就会自动失去 DetailViewController 的 strong 引用，这时候 DetailViewController 对象就会被自动释放。

如果这两个指针都是 strong 类型，就会出现所有权回环。导致对象无法在适当的时候被释放。

unsafe_unretained

除了 strong 和 weak，还有另外一个 unsafe_unretained 关键字，一般你不会使用到它。声明为 unsafe_unretained 的变量或 property，编译器不会为其自动添加 retain 和 release。unsafe_unretained 只是为了兼容 iOS 4，因为 iOS 4 没有 weak pointer system。

"unsafe"表示这种类型的指针可能指向不存在的对象。使用这样的指针很可能导致应用崩溃，启用 NSZombieEnabled 调试工具可以查找这种类型的错误。

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

运行应用并模拟内存不足警告时，可以看到以下输出：

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

应用崩溃了！unsafe_unretained 指针并没有对象的所有权。意味着 UITableView 不会被这个指针保持生命，在 viewDidUnload 被调用之前 UITableView 就已经被释放（UITableView 唯一的拥有者是 main view）。如果这是一个真正的 weak 指针，则它的值会被自动设置为 nil，这也正是"zeroing" weak pointer 的优点。

unsafe_unretained 指针和 weak 指针不一样的是，当相关联的对象释放时，指针不会被设置为 nil，因此它实际上指向不存在的对象。有时候你的应用看上去没有问题，但更大的可能是应用会崩溃。

注意：你需要选择 Product->Edit Scheme->Diagnostics，并启用 zombies 调试选项，才能立即看到应用的崩溃。

iOS 4 中使用 ARC

ARC 主要是 LLVM 3.0 编译器（而不是 iOS 5）的新特性，因此你也可以在 iOS 4.0 之后的系统中使用 ARC，不过需要注意的是，**weak 指针需要 iOS 5 才能使用**。如果你要在 iOS 4 中部署 ARC 应用，你就不能使用 weak property 和__weak 变量。

实际上，要让 ARC 正确运行于 iOS 4 平台，你不需要做任何额外的事情。只要你选择 iOS 4 作为应用的 Deployment Target，编译器就会自动插入一个兼容库到你的工程，使得 ARC 功能在 iOS 4 平台中可用。

这时如果你使用了 weak 引用，编译器会给出如下错误：

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

既然 iOS 4 不能使用 __weak 和 weak，你就得分别替换为 __unsafe_unretained 和 unsafe_unretained，记住这些变量在对象被释放时不会自动设置为 nil，因此如果你不够小心，你的变量就可能会指向不存在的对象。启用 NSZombieEnabled 来调试这种类型的错误吧！

ARC 高级指南

Blocks 与 ARC

Block 和 ARC 可以很好地工作，实际上 ARC 使得 Block 比以前更容易使用。Block 在堆栈中初始创建，如果你需要在超出当前作用域之后，仍然保持 Block 继续存在，就必须使用 [copy] 或 Block_copy() 将 Block 复制到堆。现在 ARC 自动处理了这些复制工作，不过和以前相比，Block 的使用也有一些地方变得不同了。

假如我们有一个 AnimatedView，每秒钟会重新绘制自己几次。我们使用 Block 来提供具体的绘制操作。

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

typedef void (^AnimatedViewBlock)(CGContextRef context, CGRect rect, CFTimeInterval totalTime, CFTimeInterval deltaTime);

@interface AnimatedView : UIView
@property (nonatomic, copy) AnimatedViewBlock block;
@end
```

注意 Block 使用了 copy 关键字，使用 strong 会直接导致应用崩溃，因为 Block 在堆栈中创建，必须 copy 到 property（堆）。

这个类只有一个 block 属性，block 有四个参数：context, rect, totalTime 和 deltaTime。其中 context 和 rect 是绘制用途，两个时间参数用来确定动画的步骤

```
#import <QuartzCore/QuartzCore.h>
#import "AnimatedView.h"

@implementation AnimatedView
{
    NSTimer *timer;
```

```
    CFTimeInterval startTime;
    CFTimeInterval lastTime;
}

@synthesize block;

- (id)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        timer = [NSTimer scheduledTimerWithTimeInterval:0.1
                        target:self
                        selector:@selector(handleTimer:)
                        userInfo:nil
                        repeats:YES];
        startTime = lastTime = CACurrentMediaTime();
    }
    return self;
}

- (void)dealloc
{
    NSLog(@"dealloc AnimatedView");
    [timer invalidate];
}

- (void)handleTimer:(NSTimer*)timer
{
    [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
{
    CFTimeInterval now = CACurrentMediaTime();
    CFTimeInterval totalTime = now - startTime;
    CFTimeInterval deltaTime = now - lastTime;
    lastTime = now;

    if (self.block != nil)
        self.block(UIGraphicsGetCurrentContext(), rect, totalTime,
deltaTime);
}
@end
```

这个实现看上去没有问题，但实际上我们已经产生了所有权回环。在 initWithCoder 中我们创建了一个定时器，并且在 dealloc 中停止了定时器。不过假如你运行 App，你会发现 AnimatedView 的 dealloc 方法永远不会被调用到。

NSTimer 显然保持了一个 strong 引用，指向自己的 target，这里也就是 AnimatedView 对象。而 AnimatedView 同时又有一个 strong 引用指向 NSTimer。除非我们显式地释放这些对象，否则永远不会被释放。

你可能会觉得以下代码可以解决问题：

```
@implementation AnimatedView
{
    __weak NSTimer *timer;
    CFTimeInterval startTime;
    CFTimeInterval lastTime;
}
```

但是简单地将 timer 定义为 __weak 并不能解决所有权回环，AnimatedView 不再是 NSTimer 对象的拥有者,但是 NSTimer 仍然被其它对象所拥有(run loop)，而且由于 timer 仍然拥有 AnimatedView 的 strong 引用，它还是会一直保持 AnimatedView 存在，除非 invalidate 定时器，否则 AnimatedView 对象永远无法被释放。

这里我们可以定义一个 stopAnimation 方法来打破 retain 回环，修改 dealloc 和 stopAnimation 方法如下：

```
- (void)stopAnimation
{
    [timer invalidate], timer = nil;
}
- (void)dealloc
{
    NSLog(@"dealloc AnimatedView");
}
```

在 AnimatedView 被释放之前，使用这个类的用户（例如 DetailViewController）应该先调用 stopAnimaTion 来停止定时器。

下面我们看看 DetailViewController.m 中如何提供一个 Block 来处理 Drawing，代码如下：

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

    UIFont *font = [UIFont boldSystemFontOfSize:24.0f];
    CGSize textSize = [self.artistName sizeWithFont:font];
```

```
    self.animatedView.block = ^(CGContextRef context, CGRect
rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
    {
        NSLog(@"totalTime %f, deltaTime %f", totalTime, deltaTime);
        CGPoint textPoint = CGPointMake((rect.size.width
- textSize.width)/2, (rect.size.height - textSize.height)/2);
        [self.artistName drawAtPoint:textPoint withFont:font];
    };
}
```

在 Block 外面，我们创建了一个 UIFont 对象，并计算了 text 绘制所需的大小。在 Block 中，我们使用 textSize 定位文本到矩形的中央并进行绘制。NSLog 则用来显示这个 Block 会定时被调用。

运行 App，看上去好像没有任何问题。但是如果你查看 Debug 输出，你会发现最终 AnimatedView 和 DetailViewController 都无法被释放！

如果你以前使用过 Block，你应该已经知道 Block 会捕获（capture）自己使用到的所有变量。如果这些变量是指针，Block 会对指向的每个对象执行一次 retain。在上面例子中，Block 会 retain self，也就是 DetailViewController（因为我们在 Block 中使用了 self.artistName）。现在即使关闭了 DetailViewController，也不会释放 DetailViewController 对象，因为 Block 保持了它的引用。

另外定时器也会继续运行，因为我们没有调用 stopAnimation 来中止定时器（DetailViewController 的 dealloc 没有被调用）。

简单的解决办法是不要在 Block 中使用 self，这意味着你不能在 Block 中使用任何 property、实例变量(因为实例变量会使用 self->ivar)、方法(method)。而只能使用局部变量，先把要使用的对象保存在局部变量中，然后在 Block 中使用这个局部变量：

```
NSString *text = self.artistName;
self.animatedView.block = ^(CGContextRef context, CGRect rect,
CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    CGPoint textPoint = CGPointMake((rect.size.width - textSize.width)/2,
(rect.size.height - textSize.height)/2);
    [text drawAtPoint:textPoint withFont:font];
};
```

这样就可以了，Block 只使用了局部变量，没有引用 self，因此 Block 不会捕获 DetailViewController 对象，后者在 View 关闭时能够自动被释放。

但很多时候你不能避免在 Block 中使用 self，在 ARC 以前，你可以使用以下技巧：

```
__block DetailViewController *blockSelf = self;
self.animatedView.block = ^(CGContextRef context, CGRect
rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    . . .
    [blockSelf.artistName drawAtPoint:textPoint withFont:font];
};
```

__block 关键字表示 Block 不 retain 这个变量，因此可以使用 blockSelf.artistName 来访问 artistName 属性，而 Block 也不会捕获 self 对象。

不过 ARC 中不能使用这个方法，因为变量默认是 strong 引用，即使标记为 __block 也仍然是 strong 类型的引用。这时候__block 的唯一功能是允许你修改已捕获的变量（没有__block 则变量是只读的）。

ARC 的解决办法是使用 __weak 变量：

```
__weak DetailViewController *weakSelf = self;
self.animatedView.block = ^(CGContextRef context, CGRect
rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    DetailViewController *strongSelf = weakSelf;
    if (strongSelf != nil)
    {
        CGPoint textPoint = CGPointMake((rect.size.width
- textSize.width)/2, (rect.size.height - textSize.height)/2);
        [strongSelf.artistName drawAtPoint:textPoint withFont:font];
    }
};
```

weakSelf 变量引用了 self,但不会进行 retain。我们让 Block 捕获 weakSelf 而不是 self，因此不存在所有权回环。但是我们在 Block 中不能直接使用 weakSelf，因为这是一个 weak 指针，当 DetailViewController 释放时它会自动变成 nil。虽然向 nil 发送 message 是合法的，我们在 Block 中仍然检查了对象是否存在。**这里还有一个技巧，我们临时把 weakSelf 转换为 strong 类型的引用 strongSelf，这样我们在使用 strongSelf 的时候，可以确保 DetailViewController 不会被其它人释放掉！**

对于简单的应用，这人技巧可能没太大必要，我们可以直接使用 weakSelf。因为 AnimatedView 属于 controller 的 view hierarchy,DetailViewController 不可能在 AnimatedView 之前被释放。

但是假如 Block 被异步使用，则创建 strong 引用来保持对象存活就是必要的。此外，如果我们直接使用 artistName 实例变量，可能会编写如下代码：

```
__weak DetailViewController *weakSelf = self;
```

```
self.animatedView.block = ^(CGContextRef context, CGRect
rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
{
    [weakSelf->artistName drawAtPoint:textPoint withFont:font];
};
```

在 DetailViewController 被释放之前，这段代码都能正常工作，一旦 DetailViewController 被释放，nil->artistName 肯定会导致应用崩溃！

因此，如果你需要在 ARC 环境中使用 Block，并且想避免捕获 self，推荐采用如下代码模式：

```
__weak id weakSelf = self;
block = ^()
{
    id strongSelf = weakSelf;
    if (strongSelf != nil)
    {
        // do stuff with strongSelf
    }
};
```

如果目标机器是 iOS 4，就不能使用__weak，可以如下修改：

```
__block __unsafe_unretained id unsafeSelf = self;
```

注意在这种情况下，你无法确定 unsafeSelf 是否仍然指向合法的对象。你需要采取额外的步骤来保护，否则就会面临 zombies！

整体来说，编写 ARC 的 Block 代码和之前是一样的，除了你不再调用 retain 和 release。但是仍然可能会引入微妙的 bug，因为没有显式的 retain 就意味着 Block 不会再捕获某个对象，因此当你使用的时候对象可能已经不存在。

假设有个 DelayedOperation 类,它等待"delay"秒,然后执行 block。在 block 中你会 autorelease 释放掉 DelayedOperation 实例。由于 block 捕获了 "operation"实例，从而保持了对象的生命，下面代码模式在 ARC 之前是可以正常工作的：

```
DelayedOperation *operation = [[DelayedOperation alloc] initWithDelay:5
block:^
{
    NSLog(@"Performing operation");
    // do stuff
    [operation autorelease];
}];
```

但在 ARC 中，你不允许调用 autorelease，因此代码变成如下：

```
DelayedOperation *operation = [[DelayedOperation alloc] initWithDelay:5
block:^
```

```
{
    NSLog(@"Performing operation");
    // do stuff
}];
```

猜猜会发生什么？block 永远不会被执行到。DelayedOperation 实例在创建后立即就会被释放，因为没有人保持对象的生命。解决办法是让 block 捕获 operation 实例对象，并在完成任务之后设置为 nil，从而释放该 operation 对象：

```
__block DelayedOperation *operation = [[DelayedOperation
alloc] initWithDelay:5 block:^
{
    NSLog(@"Performing operation");
    // do stuff

    operation = nil;
}];
```

现在 block 会保持对象存活，注意"opoeration"变量必须声明为__block，因为我们要在 block 里面修改它为 nil。

Singleton 与 ARC

如果你的应用使用了 Singleton，你的实现可能包含以下方法：

```
+ (id)allocWithZone:(NSZone *)zone
{
    return [[self sharedInstance] retain];
}
- (id)copyWithZone:(NSZone *)zone
{
    return self;
}
- (id)retain
{
    return self;
}
- (NSUInteger)retainCount
{
    return NSUIntegerMax;
}
- (oneway void)release
{
    // empty
}
- (id)autorelease
{
```

```
    return self;
}
```

这是典型的 singleton 实现模式，retain 和 release 都覆盖掉，使其不能创建多个实例对象。毕竟 Singleton 就是为了只创建一个全局对象。

在 ARC 环境下，上面代码无法工作。你不能调用 retain 和 release，也不允许你覆盖这些方法。

按我的看法，无论如何，上面这个都不是非常有用的模式。你确定需要只有一个实例的对象吗？我倾向于使用一个称为"Interesting Instance Pattern"的 Singleton 模式。这也是 Apple 在它们的 API 中采用的模式。通常你通过一个 sharedInstance 或 defaultInstance 类方法来访问这个 preferred 实例，但只要你愿意，你也可以创建你自己的实例。

对于 iOS API 来说，这些类能够让你创建自己的实例实际上是一个特性，例如 NSNotificationCenter

下面我们使用 GradientFactory 类来演示实现 Singleton 的首选方式，GradientFactory.h 如下：

```
@interface GradientFactory : NSObject

+ (id)sharedInstance;

- (CGGradientRef)newGradientWithColor1:(UIColor *)color1
    color2:(UIColor *)color2 color3:(UIColor *)color3
    midpoint:(CGFloat)midpoint;
@end
```

这个类有一个 sharedInstance 类方法，用来访问 preferred 实例，同时还有一个 newGradient 方法返回一个 CGGradientRef 对象。你仍然可以 [[alloc] init] 你自己的 GradientFactory 实例，不过按照惯例你不应该这样做。

GradientFactory.m 中如下实现 sharedInstance 方法：

```
+ (id)sharedInstance
{
    static GradientFactory *sharedInstance;
    if (sharedInstance == nil)
    {
        sharedInstance = [[GradientFactory alloc] init];
    }
    return sharedInstance;
}
```

这就是你实现 Singleton 需要做的全部，sharedInstance 方法使用 static local 变量来跟踪实例是否已经存在，如果不存在就创建一个。注意我们不需要显式地设置变量为 nil：

```
static GradientFactory *sharedInstance = nil;
```

在 ARC 中，所有指针变量默认都是 nil，在 ARC 之前，只有实例变量才会默认为 nil。如果你编写下面代码：

```
- (void)myMethod
{
    int someNumber;
    NSLog(@"Number: %d", someNumber);
    NSString *someString;
    NSLog(@"String: %p", someString);
}
```

编译器会报怨："Variable is uninitialized when used here"，而输出则是随机数值：

```
Woot[2186:207] Number: 67
Woot[2186:207] String: 0x4babb5
```

但在 ARC 中，输出则如下：

```
Artists[2227:207] Number: 10120117
Artists[2227:207] String: 0x0
```

int 仍然是随机值（这样使用编译器也会警告），但 someString 的初始值已经是 nil，这样的优点是指针永远不会指向非法对象。

继续实现 GradientFactory 的下面方法：

```
- (CGGradientRef)newGradientWithColor1:(UIColor *)color1
    color2:(UIColor *)color2 color3:(UIColor *)color3
    midpoint:(CGFloat)midpoint;
{
    NSArray *colors = [NSArray
arrayWithObjects:(id)color1.CGColor, (id)color2.CGColor,
(id)color3.CGColor, nil];

    const CGFloat locations[3] = { 0.0f, midpoint, 1.0f };

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGGradientRef gradient =
CGGradientCreateWithColors(colorSpace, (__bridge CFArrayRef)colors,
locations);
    CGColorSpaceRelease(colorSpace);

    return gradient;
}
```

CGGradientCreateWithColors()函数的参数是:一个 CGColorSpaceRef 对象、CGColorRef 对象的数组、CGFloats 数组。

其中 NSArray *colors 使用 ARC 的 __bridge，这样才能传递给 CGGradientCreateWithColors()函数使用。这里我们让 ARC 继续保持 Array 对象的所有权，并负责释放对象。不过 NSArray 只能存储 Objective-C 对象，因此这里需要将 Core Foundation 的 CGColorRef 转换为 id,因为所有 Core Foundation 对象类型都与 NSObject 能够 toll-free bridged。但是不能转换为 UIColor，因为 CGColor 和 UIColor 并不是 toll-free bridged。混合使用这两种不同类型的 framework 体系架构时，确实够乱的。

另外 Array 中的 CGColorRef 对象转换为 id 时并不需要使用 __bridge，因为编译器知道这里的规则,只要数组还存在,就会自动 retain 这些 color 对象。

最后返回一个新创建的 CGGradientRef 对象，注意调用方负责释放这个 gradient 对象，它是一个 Core Foundation 对象，因此 ARC 不会进行处理。调用方需要调用 CGGradientRelease()来释放这个对象，否则应用就会产生内存泄漏。

根据 Cocoa 命名规则，方法名中带有 alloc, init, new, copy, mutableCopy 的方法会传递返回对象的所有权给调用方，我们的 newGradient 方法也一样。老实说，如果你的所有代码都基于 ARC，这个命名惯例已经变得完全不再重要。编译器能够为你处理一切对象释放的工作,不过通过名字让其它人知道你的方法会创建对象并传递所有权，要求调用方释放对象，仍然是一种很好的做法。

假设你有一个方法在非 ARC 源文件中，并命名为 newWidget，它返回一个 autoreleased 而不是 retained 字符串。如果你在 ARC 环境中使用这个方法,ARC 最后会 release 这个返回的对象，从而多次 release 导致应用崩溃。因此最好重命名你的方法为 createWidget 或 makeWidget，让 ARC 知道不需要 release 该对象（或者如果你不能修改方法名，可以使用 NS_RETURNS_NOT_RETAINED 或 NS_RETURNS_RETAINED annotation 告诉编译器这个方法是非标准的）。

混合使用 Core Foundation 和 Objective-C 对象时，你必须非常小心。试着找出下面代码的 bug：

```
CGColorRef cgColor1 = [[UIColor alloc] initWithRed:1 green:0
blue:0 alpha:1].CGColor;
CGColorRef cgColor2 = [[UIColor alloc] initWithRed:0 green:1
blue:0 alpha:1].CGColor;
CGColorRef cgColor3 = [[UIColor alloc] initWithRed:0 green:0
blue:1 alpha:1].CGColor;
NSArray *colors = [NSArray arrayWithObjects:(__bridge
id)cgColor1, (__bridge id)cgColor2, (__bridge id)cgColor3, nil];
```

如果你这样写代码，应用肯定会崩溃，因为你创建了三个 retain 而不是 autorelease 的 UIColor 对象，只要没有 strong 指针引用它们，这些对象就会被释放。由于 CGColorRef 不是 Objective-C 对象，cgColor1、cgColor2 、cgColor3 变量都不是 strong 指针，创建的 UIColor 都会立即被释放，而这些指针现在都指向垃圾内存。下面这样做则是正确的：

```
CGColorRef cgColor1 = [UIColor colorWithRed:1 green:0 blue:0
alpha:1].CGColor;
```

现在我们分配 UIColor 对象并返回一个 autorelease 的对象，对象在 autorelease pool flush 之前都会存在。如果你不希望操心这些恶心的问题，那就尽量不要使用 Core Foundation 框架吧。

下面是这个 Singleton 类 GradientFactory 的使用示例，DetailViewController 的 viewDidLoad 方法如下：

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

    UIFont *font = [UIFont boldSystemFontOfSize:24.0f];
    CGSize textSize = [self.artistName sizeWithFont:font];

    float components[9];
    NSUInteger length = [self.artistName length];
    NSString* lowercase = [self.artistName lowercaseString];
    for (int t = 0; t < 9; ++t)
    {
        unichar c = [lowercase characterAtIndex:t % length];
        components[t] = ((c * (10 - t)) & 0xFF) / 255.0f;
    }

    UIColor *color1 = [UIColor
colorWithRed:components[0] green:components[3] blue:components[6]
alpha:1.0f];
    UIColor *color2 = [UIColor
colorWithRed:components[1] green:components[4] blue:components[7]
alpha:1.0f];
    UIColor *color3 = [UIColor
colorWithRed:components[2] green:components[5] blue:components[8]
alpha:1.0f];

    __weak DetailViewController *weakSelf = self;
```

```
    self.animatedView.block = ^(CGContextRef context, CGRect
rect, CFTimeInterval totalTime, CFTimeInterval deltaTime)
    {
        DetailViewController *strongSelf = weakSelf;
        if (strongSelf != nil)
        {
            CGPoint startPoint = CGPointMake(0.0, 0.0);
            CGPoint endPoint = CGPointMake(0.0, rect.size.height);
            CGFloat midpoint = 0.5f + (sinf(totalTime))/2.0f;

            CGGradientRef gradient = [[GradientFactory sharedInstance]
                newGradientWithColor1:color1 color2:color2
color3:color3
                midpoint:midpoint];

            CGContextDrawLinearGradient(context, gradient,
                startPoint, endPoint,
                kCGGradientDrawsBeforeStartLocation |
                kCGGradientDrawsAfterEndLocation);

            CGGradientRelease(gradient); // Core Foundation 内存仍然需
要手动内存管理

            CGPoint textPoint = CGPointMake(
                (rect.size.width - textSize.width)/2,
                (rect.size.height - textSize.height)/2);
            [strongSelf.artistName
drawAtPoint:textPoint withFont:font];
        }
    };
}
```

如果 Singleton 需要使用在多线程环境中，上面简单的 sharedInstance 方法就不能满足要求了。健壮的实现方法如下：

```
+ (id)sharedInstance
{
    static GradientFactory *sharedInstance;
    static dispatch_once_t done;
    dispatch_once(&done, ^{
        sharedInstance = [[GradientFactory alloc] init];
    });
    return sharedInstance;
}
```

即使多个线程同时执行这个 block，Grand Central Dispatch 库的 dispatch_once()函数也能确保 alloc 和 init 只会被执行一次。

Autorelease 和 AutoreleasePool

ARC 仍然使用了 autorelease 和 autorelease pool，不过现在这个特性是语言结构而不是以前的类（现在是@autoreleasepool）。

在 ARC 中，方法名如果以 alloc, init, new, copy, mutableCopy 开头，就是返回 retain 的对象，其它方法全部返回 autorelease 的对象。这条规则实际上与手动内存管理是一样的。因为 ARC 代码需要与非 ARC 协同工作。

retain 的对象在没有变量指向它时就会立即释放，但 autorelease 对象则是在 autorelease pool 排干（drain）时才会被释放。在 ARC 之前，你需要调用 NSAutoreleasePool 对象的 [drain] 或 [release] 方法，现在则是直接在 @autoreleasepool 块退出时进行 drain：

```
@autoreleasepool
{
    NSString *s = [NSString stringWithFormat:. . .];
} // the string object is deallocated here
```

但是如果你像下面这样编写代码，则即使 NSString 对象在@autoreleasepool 块中创建，stringWithFormat:方法也确实返回了一个 autorelease 对象，但变量 s 是 strong 类型的，只要 s 没有退出作用域，string 对象就会一直存在：

```
NSString *s;
@autoreleasepool
{
    s = [NSString stringWithFormat:. . .];
}
// the string object is still alive here
```

使用 __autoreleasing 可以使 autorelease pool 释放掉该对象，它告诉编译器这个变量指向的对象可以被 autorelease，此时变量不是 strong 指针，string 对象会在@autoreleasepool 块的末尾被释放。不过注意变量在对象释放后，仍然会继续指向一个死掉的对象。如果你继续使用它，应用就会崩溃。代码如下：

```
__autoreleasing NSString *s;
@autoreleasepool
{
    s = [NSString stringWithFormat:. . .];
}
// the string object is deallocated here
NSLog(@"%@", s); // crash!
```

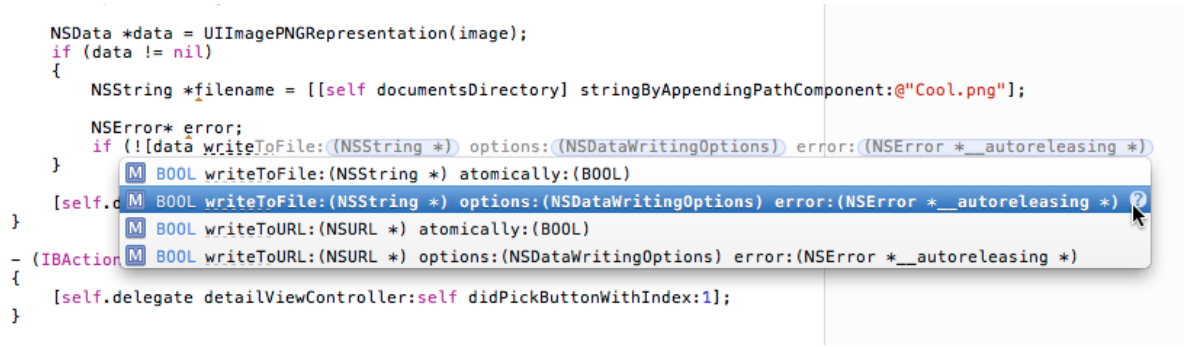
你应该很少需要使用 __autoreleasing，这个关键字主要用在输出参数（out-parameters）或按引用传递（pass-by-reference），典型的例子就是使用了 (NSError **) 参数的那些方法。

我们来看一个例子：

```
- (IBAction)coolAction
{
    NSData *data = UIImagePNGRepresentation(image);
    if (data != nil)
    {
        NSString *filename = [[self documentsDirectory] stringByAppendingPathComponent:@"Cool.png"];
        NSError *error;
        if (![data writeToFile:filename options:NSDataWritingAtomic error:&error])
        {
            NSLog(@"Error: %@", error);
        }
    }
}
```

NSData 的 writeToFile 方法需要一个 NSError **的参数，如果出现了某种错误，方法会创建一个新的 NSError 对象，并存储错误信息到这个对象中。这就是 out-parameter，通常用来函数返回更多的信息。

如果你在 Xcode 中输入上面代码，你可以看到自动完成提示，(NSError **) 参数实际上指定为：(NSError *__autoreleasing *)



这是 ARC 实现 out-parameters 的通用模式，它告诉编译器 writeToFile 方法返回的 NSError 对象是 autorelease 对象。通常你不需要担心和使用 __autoreleasing，只有你编写自己的 out-parameter 函数，或者遇到性能问题时，才需要考虑__autoreleasing。

在上面代码中，我们直接定义：

```
NSError *error;
```

这意味着 error 变量是 __strong 类型，然后将 error 传递给 writeToFile 方法时，后者接受的却是 __autoreleasing 变量。一个变量不可能同时既是 strong 又是 autoreleasing。因此编译器在这里实际上使用了一个临时变量，我们的代码经过编译器处理实际上如下：

```
NSError *error;
__autoreleasing NSError *temp = error;
BOOL result = ![data writeToFile:filename options:NSDataWritingAtomic
error:&temp];
error = temp;
if (!result)
{
    NSLog(@"Error: %@", error);
}
```

通常这个额外的临时变量没什么大不了，不过如果你确实希望避免编译器为你生成，可以如下编写代码：

```
__autoreleasing NSError *error;
if (![data writeToFile:filename options:NSDataWritingAtomic
error:&error])
{
    NSLog(@"Error: %@", error);
}
```

现在 error 变量和 writeToFile 的参数类型就一致了，因此编译器不会进行转换。当然，我个人不会这样使用__autoreleasing，多数情况下，上面代码都是不必要的优化。

如果你需要编写自己的 out-parameter 方法，可以采用下面代码模式：

```
- (NSString *)fetchKeyAndValue:(__autoreleasing NSNumber **)value
{
    NSString *theKey;
    NSNumber *theValue;

    // do whatever you need to do here

    *value = theValue;
    return theKey;
}
```

方法返回一个 NSString 对象，并且在 out-parameter 中返回一个 NSNumber 对象。使用示例如下：

```
NSNumber *value;
NSString *key = [self fetchKeyAndValue:&value];
```

顺便说下，out-parameter 的默认所有权是 __autoreleasing，因此你实际上可以简化方法定义为：

```
- (NSString *)fetchKeyAndValue:(NSNumber **)value
{
    . . .
}
```

反过来，如果你不想为 out-parameter 使用 __autoreleasing，不想把对象放到 autorelease pool，你可以使用 __strong 来声明 out-parameter：

```
- (NSString *)fetchKeyAndValue:(__strong NSNumber **)value
{
    . . .
}
```

对于 ARC 来说，它实际上并不关心 out-parameters 到底是 autorelease 还是 strong，编译器总是会做正确的事情。但是如果你想要在非 ARC 环境中使用这个方法，就需要你自己对返回的对象进行手动的 [release]。否则就会导致内存泄漏，因此 out-parameters 返回 retain 对象时必须给出明确的文档！

还有一件事情你需要知道，有些 API 方法会使用自己的 autorelease pool，例如 NSDictionary 的 enumerateKeysAndObjectsUsingBlock:方法首先设置一个 autorelease pool，然后再调用你提供的 block。这意味着你在 block 中创建的任何 autoreleased 对象都会被 NSDictionary 的 pool 释放，一般这也正是你想要的效果。但是下面情况例外：

```
- (void)loopThroughDictionary:(NSDictionary *)d error:(NSError **)error
{
    [d enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop)
    {
        // do stuff . . .

        if (there is some error && error != nil)
        {
            *error = [NSError errorWithDomain:@"MyError"
code:1 userInfo:nil];
        }
    }];
}
```

error 变量是一个 out-parameter，因此也是 autoreleased，由于 enumerateKeysAndObjectsUsingBlock:拥有自己的 autorelease pool，你创建的 error 对象会在方法返回之前被释放。要解决这个问题，你需要使用一个临时的 strong 变量来保存 NSError 对象：

```
- (void)loopThroughDictionary:(NSDictionary *)d error:(NSError **)error
{
    __block NSError *temp;
```

```
    [d enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop)
    {
        // do stuff . . .

        if (there is some error)
        {
            temp = [NSError errorWithDomain:@"MyError" code:1
userInfo:nil];
        }
    }];

    if (error != nil)
        *error = temp;
}
```

autorelease 对象还可能会比你想象中存活更长时间，在 iOS 中，每次 UI 事件（点击按钮等）都会清空一次 autorelease pool，但是如果你的事件处理器进行了大量操作，例如循环地创建许多对象，最好是使用你自己的 autorelease pool，避免应用面临内存不足：

```
for (int i = 0; i < 10000; i++)
{
    @autoreleasepool
    {
        NSString *s = [NSString stringWithFormat:. . .];
    }
}
```

在 ARC 之前的代码中，有时候你会见到循环和 autorelease pool 的一个特殊技巧，X 次循环迭代清空一次 autorelease pool。因为很多人认定 NSAutoreleasePool 很慢（实际上并不慢），觉得每次循环都清空 pool 效率低下。其实很没有必要！

现在@autoreleasepool 比 NSAutoreleasePool 要快 6 倍左右，因此直接在循环中使用@autoreleasepool 不会带来任何性能问题。

如果你创建了一个新线程，你需要使用 @autorelease 来封装代码到 autorelease pool 中，虽然语法不同，但原理是一样的。

ARC 对 autorelease 对象还有一些更进一步的优化，多数方法都会返回一个 autorelease 对象，但并不是所有这些对象都需要存放在 autorelease pool 中，假设你编写了下面代码：

```
NSString *str = [NSString stringWithFormat:. . .];
```

stringWithFormat 方法返回一个 autorelease 对象，但 str 是一个 strong 类型的本地变量。当 str 超出作用域时，string 对象会被释放。因此这里没有

必要再将 string 对象放进 autorelease pool。ARC 通过一些特定的技术手段，能够识别出这一类代码，不会将这些对象进行 autorelease。因此 @autoreleasepool 不仅更快，而且还能减少对象的存入。

最后注意 Core Foundation 对象不能 autorelease，autorelease 完全纯属于 Objective-C。有些人通过转换 Core Foundation 对象为 id，再调用 autorelease，然后再强制转换回 Core Foundation 对象：

```
return (CGImageRef)[(id)myImage autorelease];
```

在 ARC 中这明显不能工作，因为你不能调用 autorelease。如果你确实要试一试 autorelease 一个 Core Foundation 对象，可以编写自己的 CFAutorelease() 函数，并把函数放到一个设置了 -fno-objc-arc 的文件中进行编译。

Cocos2D 和 Box2D

前面讲的是 ARC 与基于 UIKit 的 App，对于 Cocos2D 游戏，ARC 的规则是一样的，但由于当前版本的 Cocos2D 并没有 100%与 ARC 兼容，因此将 Cocos2D 库整合到你的 ARC 应用时，仍然需要一点点额外工作。

注意：随着时间和 Cocos2D 项目的进展，与 ARC 的兼容肯定会越来越好，因此某些代码可能已经根据 ARC 进行了变化。另外 Cocos2D 2.0beta 和 1.1beta 版本都已经完全兼容 ARC，不过暂时都还不是稳定版。

无论如何，除非你完全重写 Cocos2D，使其全面支持 ARC（这明显是不现实的）。否则你的项目使用 Cocos2D 时，都需要对 Cocos2D 源代码禁止 ARC，只有你自己的代码才能享受 ARC 带来的好处。

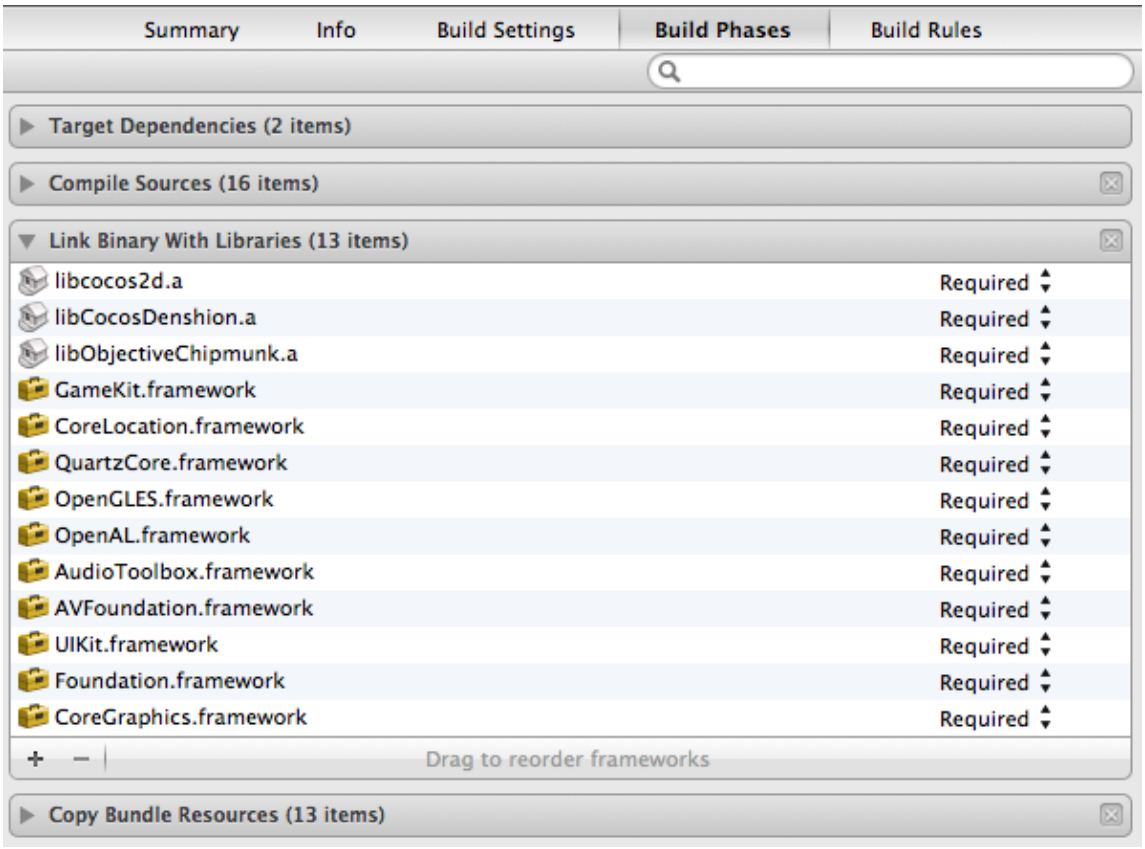
Cocos2D 下载及安装请参考：
http://www.cocos2d-iphone.org/wiki/doku.php/prog_guide:lesson_1._install_test

整合 Cocos2D 到你的项目有以下几种方式：

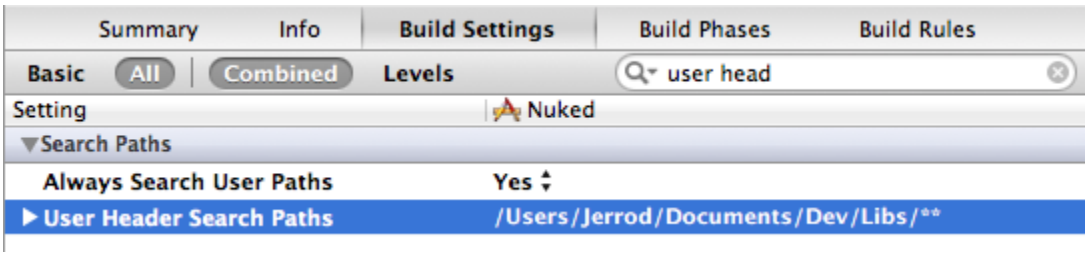
1. 使用 Cocos2D 提供的 Xcode project template 来创建你的项目，这是官方推荐使用的方法。不过这样做可能需要你在项目配置中对每个 Cocos2D 源文件设置 -fno-objc-arc 编译标志，以禁止 Cocos2D 使用 ARC。
2. 直接把 Cocos2D 安装目录下的 cocos2d-ios.xcodeproj 文件拖拉到你的 Xcode 工程中，这样会把整个 Cocos2D 工程显示你的工程之下。这样做的好处是你可以直接禁止整个 Cocos2D 项目使用 ARC，而不用对每个源文件进行设置。
3. 直接把 Cocos2D 的源代码拖拉到你的 Xcode 工程中，这是最不推荐的方法。

我们使用方法 2，详细步骤如下：

- 拖拉 cocos2d-ios.xcodeproj 工程文件到你的 Xcode 工程中。
- 在 Xcode 工程的 Build Phases 设置中添加 Cocos2D 库文件到 Link Binary With Libraries 中。



- 在 Xcode 工程的 Build Settings 中，设置 Always Search User Paths 选项为 YES，然后添加 Cocos2D 源目录到 User Header Search Paths 中，并且设置为 Recursive：



现在你的项目就已经配置好了，Cocos2D 不使用 ARC，你的项目则使用 ARC。但是如果你进行编译的话，会出现 ARC 相关的编译错误。因为你的源文件需要包含 cocos2d.h （cocos2d.h 则包含了所有你需要的头文件）。

Cocos2D 并没有完全兼容 ARC，因此只有禁用 ARC，才能够成功编译。但你的源文件包含了 cocos2d.h，在 ARC 环境中编译就自然会出现 ARC 不兼容的错误。下面我们需要修改一些 Cocos2D 的头文件和源文件，使我们的项目能够成功在 ARC 中编译。

注意这里的代码修改仅供参考，需要根据你使用的 Cocos2D 版本进行调整。

CCDirectorIOS.h：

```
@interface CCDirectorFast : CCDirectorIOS
{
    BOOL isRunning;
    NSAutoreleasePool *autoreleasePool;
}
```

这里明显不能使用 NSAutoreleasePool，CCDirectorFast.m 源文件仍然需要使用，所以需要将它移动到 CCDirectorFast.m 中去：

CCDirectorFast.m：

```
@implementation CCDirectorFast
{
    NSAutoreleasePool *autoreleasePool;
}
```

CCActionManager.h

```
typedef struct _hashElement
{
    struct ccArray *actions;
    __unsafe_unretained id target;
    NSUInteger actionIndex;
    __unsafe_unretained CCAction *currentAction;
    BOOL currentActionSalvaged;
    BOOL paused;
    UT_hash_handle hh;
} tHashElement;
```

因为结构体在 ARC 中不能再包含 Objective-C 对象，我们需要添加 __unsafe_unretained 修饰符，告诉编译器我们知道自己在做什么，不要报错！

ccCArray.h

```
typedef struct ccArray {
    NSUInteger num, max;
    __unsafe_unretained id *arr;
} ccArray;
```

做完这些，你应该就可以成功编译你的 Cocos2D 项目了。不过假如你还使用了其它第三方库，可能还需要进行其它 ARC 相关的调整。自己去研究下吧。

静态库 static library

目前并不是所有第三方库都带有 .xcodeproj 文件，能够用来构建一个静态库。你可以把第三方库的源文件添加到你的工程，然后手动设置这些源文件的 ARC 编译选项 -fno-objc-arc，能用但有点麻烦！

把第三方库构建成独立的静态库，并禁止其使用 ARC，是更加方便合理的选择。

下面我们仍然以 Cocos2D 为例，但是我们从头开始创建一个自己的静态库工程：

Xcode 中选择 New Project，工程类型使用 iOS\Framework & Library\Cocoa Touch Static Library

Product Name 可以直接使用"cocos2d"，禁止 Automatic Reference Counting

Xcode 为我们自动生成了一些默认文件，我们首先在 Xcode 中删除 cocos2d 源文件目录

然后选择 Xcode 工程中的 Add Files，浏览到你下载解压出来的 Cocos2D 安装目录，并选择添加 "cocos2d" 目录，注意确保选中了 Copy items into destination group's folder，同时 Add to targets 选中"cocos2d"

重复这个操作，把 CocosDenshion 和 external/FontLabel 目录也添加进来，另外再添加 external/Box2d/Box2D 目录，注意这里是两个 Box2D，其它的 Testbed 文件我们并不打算添加进来。

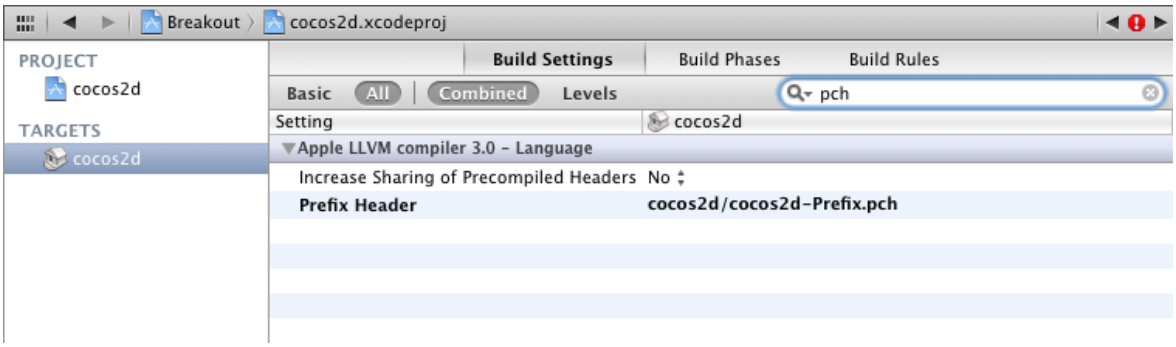
下面需要修改一些编译设置，选中 cocos2d 工程，进入 Build Settings

- Search Paths\Always Search User Paths 设置为 YES
- Search Paths\User Header Search Paths 设置为 ./**

这样设置后，Box2D 才能找到自己需要的头文件

同时请确保 "Skip Install" 设置为 YES，否则你将不能为 App Store 打包发布文件，因为静态库不应该被安装到 package 中。

最后，默认的 Xcode 静态库工程配置了一个预编译头文件（Prefix file），但 Cocos2D 并没有使用，之前我们已经把这个预编译头文件删除了，因此需要修改编译选项，告诉编译器不要使用预编译头文件，在 Build Settings 中查找 "pch"：



直接删掉 Prefix Header 选项就行了。

好了，Cmd+B 应该可以编译出自己的 Cocos2D 静态库了！

最后？

ARC 是对 Objective-C 语言的一个重要改进，虽然 iOS 5 中的其它新东西也很 cool，但 ARC 几乎完全改进了我们编写 App 的方式。虽然现在 ARC 仍然处于过渡期，使用的过程中会遇到一些兼容性问题。但根据 Apple 的态度，以及 iOS 平台的发展，ARC 在不久的将来肯定会成为主流！

最后再教你一招！如果你维护一个可重用的库，并且不想切换到 ARC（还不想吗？），你可以使用预处理指令在必要时保持与 ARC 兼容：

```
#if __has_feature(objc_arc)
// do your ARC thing here
#endif
```

或者假如你还想支持老的 GCC compiler：

```
#if defined(__has_feature) && __has_feature(objc_arc)
// do your ARC thing here
#endif
```

参考资料

- Transitioning to ARC Release Notes: https://developer.apple.com/library/mac/#releasenotes/ObjectiveC/RN-TransitioningToARC/_index.html
- Cocos2D and ARC: http://www.tinytimgames.com/2011/07/22/cocos2d-and-arc/
- 《iOS 5 By Tutorials》：http://www.raywenderlich.com/store/ios-5-by-tutorials。本文实际上是《iOS 5 By Tutorials》一书中 ARC 两章的笔记，在此感谢作者！