

DEVDIV

iOS 5 Programming Cookbook

第一章 基础入门

版本 1.0

翻译时间：2012-05-02

DevDiv 翻译：

kyelup cloudhsu 耐心摩卡 wangli2003j3 xiebaochun

DevDiv 校对：laigb kyelup

写在前面

目前，移动开发被广大的开发者们看好，并大量的加入移动领域的开发。

鉴于以下原因：

- 国内的相关中文资料缺乏
- 许多开发者对 E 文很是感冒
- 电子版的文档利于技术传播和交流

DevDiv.com 移动开发论坛特此成立了翻译组，翻译组成员具有丰富的移动开发经验和英语翻译水平。组员们利用业余时间，把一些好的相关英文资料翻译成中文，为广大移动开发者尽一点绵薄之力，希望能对读者有些许作用，在此也感谢组员们的辛勤付出。

关于 DevDiv

DevDiv 已成长为国内最具人气的综合性移动开发社区

更多相关信息请访问 [DevDiv 移动开发论坛](http://DevDiv.com)。

技术支持

首先 DevDiv 翻译组对您能够阅读本文以及关注 DevDiv 表示由衷的感谢。

在您学习和开发过程中，或多或少会遇到一些问题。DevDiv 论坛集结了一流的移动专家，我们很乐意与您一起探讨移动开发。如果您有什么问题和需要技术支持的话，请访问网站 www.devdiv.com 或者发送邮件到 BeyondVincent@DevDiv.com，我们将尽力所能及的帮助您。

关于本文的翻译

感谢 kyelup、cloudhsu、耐心摩卡、wangli2003j3 和 xiebaochun 对本文的翻译，同时非常感谢 laigb 和 kyelup 在百忙中抽出时间对翻译初稿的认真校验，指出了文章中的错误。才使本文与读者尽快见面。由于书稿内容多，我们的知识有限，尽管我们进行了细心的检查，但是还是会存在错误，这里恳请广大读者批评指正，并发送邮件至 BeyondVincent@devdiv.com，在此我们表示衷心的感谢。

目录

| | | |
|-----------|----------------------------------|----|
| 写在前面 | 2 | |
| 关于 DevDiv | 2 | |
| 技术支持 | 2 | |
| 关于本文的翻译 | 2 | |
| 目录 | 3 | |
| 第 1 章 | 基础入门 | 7 |
| 1.0. | 章节介绍 | 7 |
| 1.1. | 利用 Xcode 创建一个简单的 iOS 应用程序 | 7 |
| 1.1.1. | 问题 | 7 |
| 1.1.2. | 方案 | 8 |
| 1.1.3. | 讨论 | 8 |
| 1.1.4. | 参考 | 10 |
| 1.2. | 熟悉使用 Interface Builder(图形界面开发工具) | 11 |
| 1.2.1. | 问题 | 11 |
| 1.2.2. | 方案 | 11 |
| 1.2.3. | 讨论 | 11 |
| 1.2.4. | 参考 | 15 |
| 1.3. | 编译 iOS 项目 | 15 |
| 1.3.1. | 问题 | 15 |
| 1.3.2. | 方案 | 15 |
| 1.3.3. | 讨论 | 15 |
| 1.3.4. | 参考 | 16 |
| 1.4. | 在模拟器中运行你的应用程序 | 16 |
| 1.4.1. | 问题 | 16 |
| 1.4.2. | 方案 | 16 |
| 1.4.3. | 讨论 | 16 |
| 1.4.4. | 参考 | 17 |
| 1.5. | 在 iOS 设备中运行你的应用程序(真机调试) | 18 |
| 1.5.1. | 问题 | 18 |
| 1.5.2. | 方案 | 18 |
| 1.5.3. | 讨论 | 18 |
| 1.5.4. | 参考 | 20 |
| 1.6. | 打包并发布你的 app 为发布到 appstore 做准备. | 21 |
| 1.6.1. | 问题 | 21 |
| 1.6.2. | 方案 | 21 |
| 1.6.3. | 讨论 | 21 |
| 1.6.4. | 参考 | 24 |
| 1.7. | Objective-C 语言中如何声明变量 | 24 |
| 1.7.1. | 问题 | 24 |
| 1.7.2. | 方案 | 24 |
| 1.7.3. | 讨论 | 25 |
| 1.7.4. | 参考 | 25 |
| 1.8. | 利用 objective-c 中的 IF 逻辑判断分支结构 | 25 |
| 1.8.1. | 问题 | 25 |
| 1.8.2. | 方案 | 26 |
| 1.8.3. | 讨论 | 26 |
| 1.8.4. | 参考 | 28 |
| 1.9. | For 循环的结构 | 28 |
| 1.9.1. | 问题 | 28 |
| 1.9.2. | 方案 | 28 |

| | | |
|---------|----------------------|----|
| 1.9.3. | 讨论 | 28 |
| 1.9.4. | 参考 | 29 |
| 1.10. | 采用 While Loop 的循环结构. | 29 |
| 1.10.1. | 问题 | 30 |
| 1.10.2. | 方案 | 30 |
| 1.10.3. | 讨论 | 30 |
| 1.10.4. | 参考 | 31 |
| 1.11. | 创建自定义类 | 32 |
| 1.11.1. | 问题 | 32 |
| 1.11.2. | 方案 | 32 |
| 1.11.3. | 讨论 | 32 |
| 1.11.4. | 参考 | 35 |
| 1.12. | 定义类的功能 | 35 |
| 1.12.1. | 问题 | 35 |
| 1.12.2. | 方案 | 35 |
| 1.12.3. | 讨论 | 35 |
| 1.12.4. | 参考 | 37 |
| 1.13. | 定义具有相同名称的两个或多个方法 | 37 |
| 1.13.1. | 问题 | 37 |
| 1.13.2. | 方案 | 38 |
| 1.13.3. | 讨论 | 38 |
| 1.13.4. | 参考 | 39 |
| 1.14. | 分配和初始化对象 | 40 |
| 1.14.1. | 问题 | 40 |
| 1.14.2. | 方案 | 40 |
| 1.14.3. | 讨论 | 40 |
| 1.14.4. | 参考 | 41 |
| 1.15. | 添加类的属性 | 41 |
| 1.15.1. | 问题 | 41 |
| 1.15.2. | 方案 | 41 |
| 1.15.3. | 讨论 | 41 |
| 1.15.4. | 参考 | 43 |
| 1.16. | 将手动引用计数修改为自动引用计数 | 43 |
| 1.16.1. | 问题 | 43 |
| 1.16.2. | 方案 | 44 |
| 1.16.3. | 讨论 | 44 |
| 1.16.4. | 参考 | 46 |
| 1.17. | 自动引用计数的类型转换 | 46 |
| 1.17.1. | 问题 | 46 |
| 1.17.2. | 方案 | 46 |
| 1.17.3. | 讨论 | 47 |
| 1.17.4. | 参考 | 48 |
| 1.18. | 使用协议委托任务 | 48 |
| 1.18.1. | 问题 | 48 |
| 1.18.2. | 方案 | 48 |
| 1.18.3. | 讨论 | 48 |
| 1.18.4. | 参考 | 53 |
| 1.19. | 检测实例或类方法是否有效 | 53 |
| 1.19.1. | 问题 | 53 |

| | | |
|---------|------------------------------|----|
| 1.19.2. | 方案 | 53 |
| 1.19.3. | 讨论 | 53 |
| 1.19.4. | 参考 | 54 |
| 1.20. | 确认类是否可在运行期使用 | 54 |
| 1.20.1. | 问题 | 54 |
| 1.20.2. | 方案 | 55 |
| 1.20.3. | 讨论 | 55 |
| 1.20.4. | 参考 | 55 |
| 1.21. | 申请、使用字符串 | 55 |
| 1.21.1. | 问题 | 55 |
| 1.21.2. | 方案 | 55 |
| 1.21.3. | 讨论 | 55 |
| 1.21.4. | 参考 | 58 |
| 1.22. | 申请、使用数字 | 58 |
| 1.22.1. | 问题 | 58 |
| 1.22.2. | 方案 | 58 |
| 1.22.3. | 讨论 | 58 |
| 1.22.4. | 参考 | 59 |
| 1.23. | 分配、使用数组 | 60 |
| 1.23.1. | 问题 | 60 |
| 1.23.2. | 方案 | 60 |
| 1.23.3. | 讨论 | 60 |
| 1.23.4. | 参考 | 63 |
| 1.24. | 分配和使用 Dictionaries | 63 |
| 1.24.1. | 问题 | 63 |
| 1.24.2. | 方案 | 63 |
| 1.24.3. | 讨论 | 63 |
| 1.24.4. | 参考 | 65 |
| 1.25. | 分配和使用 Sets | 65 |
| 1.25.1. | 问题 | 65 |
| 1.25.2. | 方案 | 65 |
| 1.25.3. | 讨论 | 65 |
| 1.25.4. | 参考 | 67 |
| 1.26. | 创建程序包 | 67 |
| 1.26.1. | 问题 | 67 |
| 1.26.2. | 方法 | 67 |
| 1.26.3. | 讨论 | 67 |
| 1.26.4. | 参考 | 68 |
| 1.27. | 从主文件包加载数据 | 68 |
| 1.27.1. | 问题 | 68 |
| 1.27.2. | 方法 | 68 |
| 1.27.3. | 讨论 | 68 |
| 1.27.4. | 参考 | 70 |
| 1.28. | 从其他文件包下载数据 | 70 |
| 1.28.1. | 问题 | 70 |
| 1.28.2. | 方法 | 70 |
| 1.28.3. | 讨论 | 70 |
| 1.28.4. | 参考 | 72 |
| 1.29. | 通过 NSNotificationCenter 发送通知 | 72 |
| 1.29.1. | 问题 | 72 |
| 1.29.2. | 方法 | 72 |
| 1.29.3. | 讨论 | 72 |
| 1.29.4. | 参考 | 74 |

| | | |
|---------|-------------------------------|----|
| 1.30. | 收听 NSNotificationCenter 发出的通知 | 74 |
| 1.30.1. | 问题 | 74 |
| 1.30.2. | 方法 | 74 |
| 1.30.3. | 讨论 | 74 |
| 1.30.4. | 参考 | 76 |

DevDiv 翻译

第 1 章 基础入门

1.0. 章节介绍

目前由于 iOS5 新添加的一些特性,iPhone,iPad 和 iPodTouch 程序开发已经出现了很多新的变化。程序运行过程的一些变化也戏剧性的导致了我们在利用 Object-C 编写程序时做必要的调整。ARC(Automatic Reference Counting)自动内存回收机制现如今已经当做一种行业机制添加到 LLVM 编译器中来。从而在某些地方给我们带来了很大的灵活性来编程,但是在某些方面却又带来了一些比较棘手的问题。在本章节中,我们将从简单和基础的讲一下如何在 ARC 的机制下编写我们相关的应用代码。

万事万物,一切皆为对象。因此,对象和实物我们通常也可以交换着来讲。但是,从根本上讲,一个对象其实是一个类的一个抽象化概念,也就是通常所说的实体。从而我们从抽象的概念上给实体采集出了许多属性啊,方法等等。在面向对象的编程语言当中,类通常是抽象出来的,就好比说一个人的一些行为和特征是从他的父母那里遗传的一样。



Object-C 语言不允许多继承,也就是同时继承多个父类。因此,每一个类只有一个最直接的抽象父类。

所有 Objective-C 的超父类就是 NSObject 类,这个类运行在 iOS 所提供的一个环境当中。作为这个原因,所有继承或间接继承与 NSObject 都应该有这个特征。这就是我们在后面会提到,所有继承与超基类 NSObject 的类都能充分的利用 Objective-C 的多样化的内存环境。

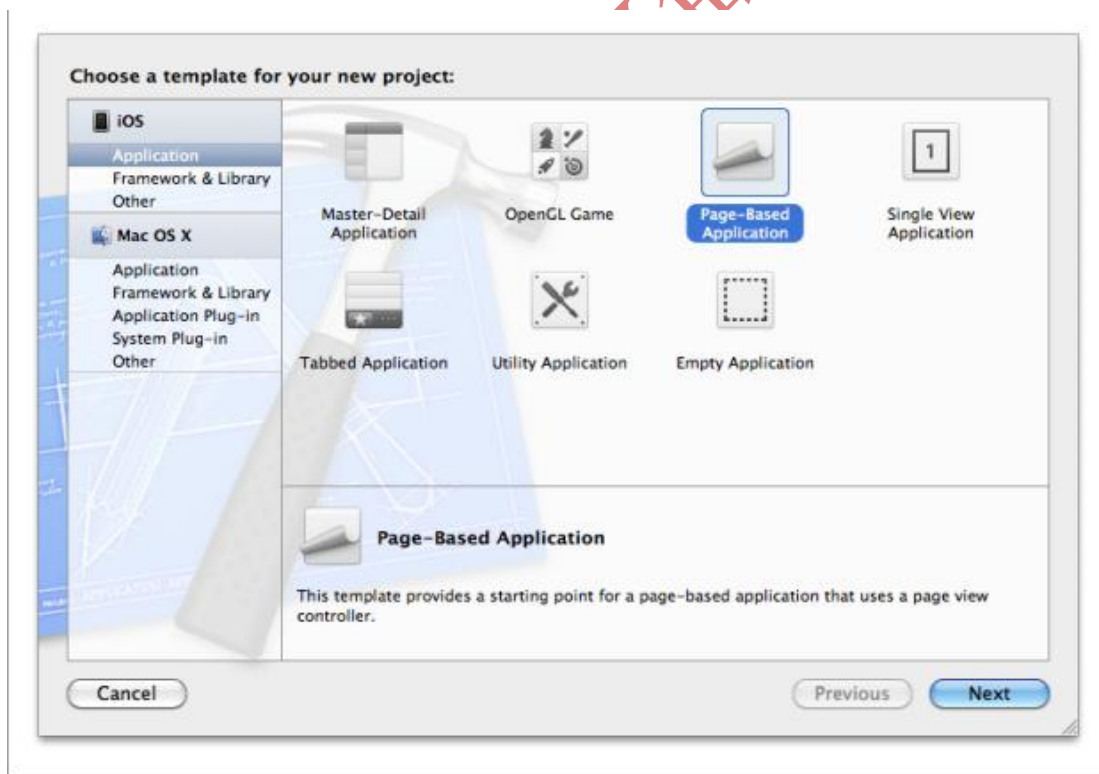


图 1-1 在 Xcode 中创建一个新的工程

1.1. 利用 Xcode 创建一个简单的 iOS 应用程序

1.1.1. 问题

你已开始学习 iOS 程序开发,并且准备利用 Xcode 创建一个简单的 iOS 应用程序。


1.1.2. 方案


利用 Xcode 创建一个简单的 iOS 工程然后利用快捷键 `Command+Shift+R` 来打开模拟器。

1.1.3. 讨论

我假设你目前已经有了台苹果电脑，并且已经安装了相应的 Xcode 开发工具。现在你需要创建一个 iOS 工程并且把你的应用程序发布到模拟器中运行并查看。这些需要如下步骤。

 如果你没有打开你的 Xcode 那么请你打开你的开发工具。

 选取菜单栏中的 File 选项，然后选择 New，然后选择创建一个新的应用程序。你将会发现一个类似于图 1-1 的界面。

 类似于图 1-1，中，选择一个 iOS,Application,然后在右边选择一个 Page-Based Application.然后选择 Next 选项。

4. 然后如图 1-2 所示，你需要给你的应用程序起一个名字，然后起一个类似你们公司名字的标示符。这个独特的标示符用来代表你们公司的一些产品应用。然后其他的选项按照示例图中的选项来操作。其中 Device Family 用来表示你的应用使用与那些设备，比如说 ipad,iphone,或者适用于所有的设备。

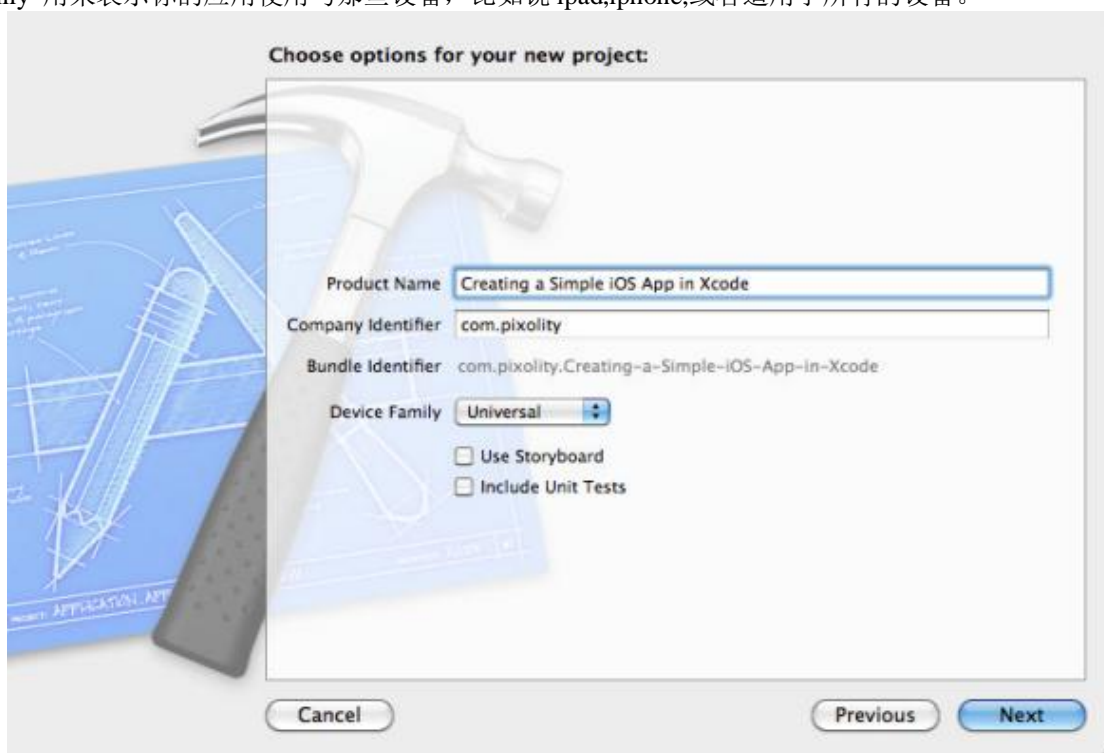


图 1-2 创建一个新工程的一些相关设置

5. 然后你需要选在一个磁盘位置来保存你所创建的项目,如示例图 3 一样,选择你一个比较理想的位置，然后 Xcode 将会帮你创建一些工程文件一些相关的工程目录结构。

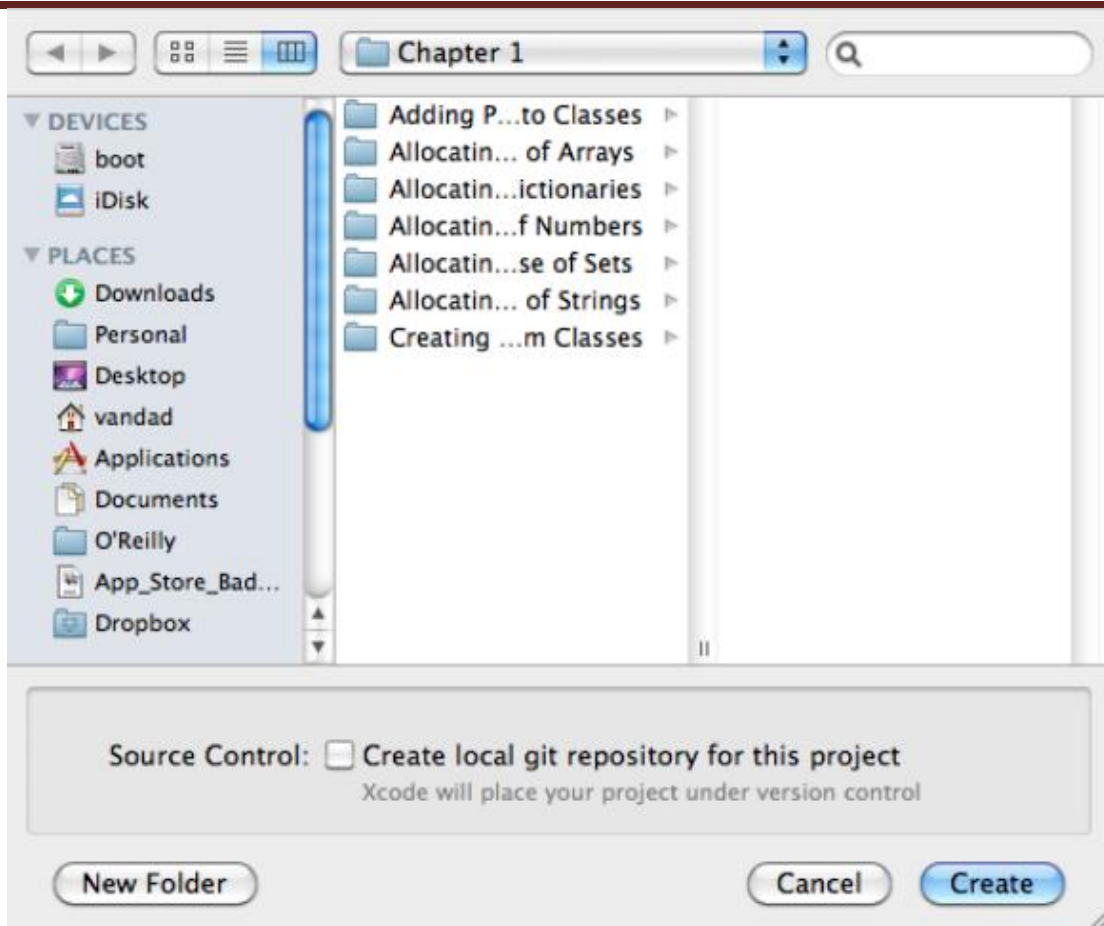


图 1-3 选择一个磁盘位置保存你的项目

6. 现在在你准备运行你的应用程序的时候，你需要确保你已经把一些外接设备，比如说 `ipad`, `iphone` 已经卸载了。因为，如果你已经连接了这些设备，那么 Xcode 将会直接在你的外接设备中来运行你的程序而不是在你的模拟器中运行。如果你并没有为你的发布配置一些运行设备，你将会没有办法来运行你的程序，一般默认是配置的有的，例如 `iphone`, `ipad`。
7. 如图 1-4 所示，通过 Xcode 左上角的那个下拉列表，选择一个 iPad 模拟器来运行你的应用程序。

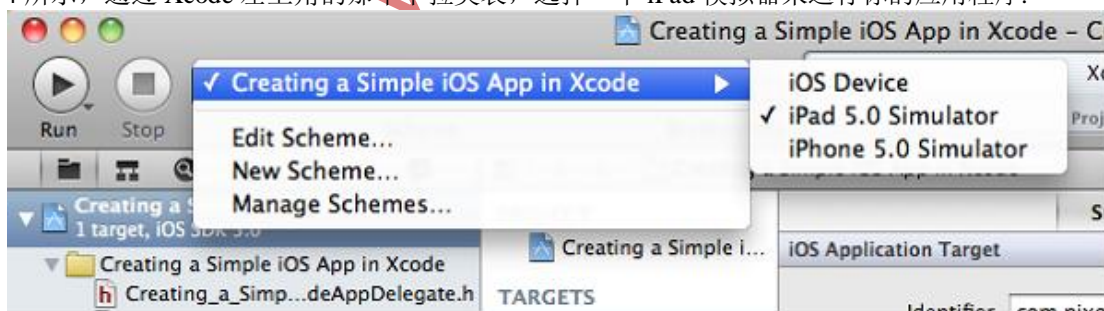


图 1-4 选择一个磁盘位置保存你的项目

8. 现在所有的事情都准备好了，然后通过快捷键 `Command+Shift+R` (如图 1-5 所示) 来进入到程序运行菜单，然后点击 `Run` 按钮来运行程序。

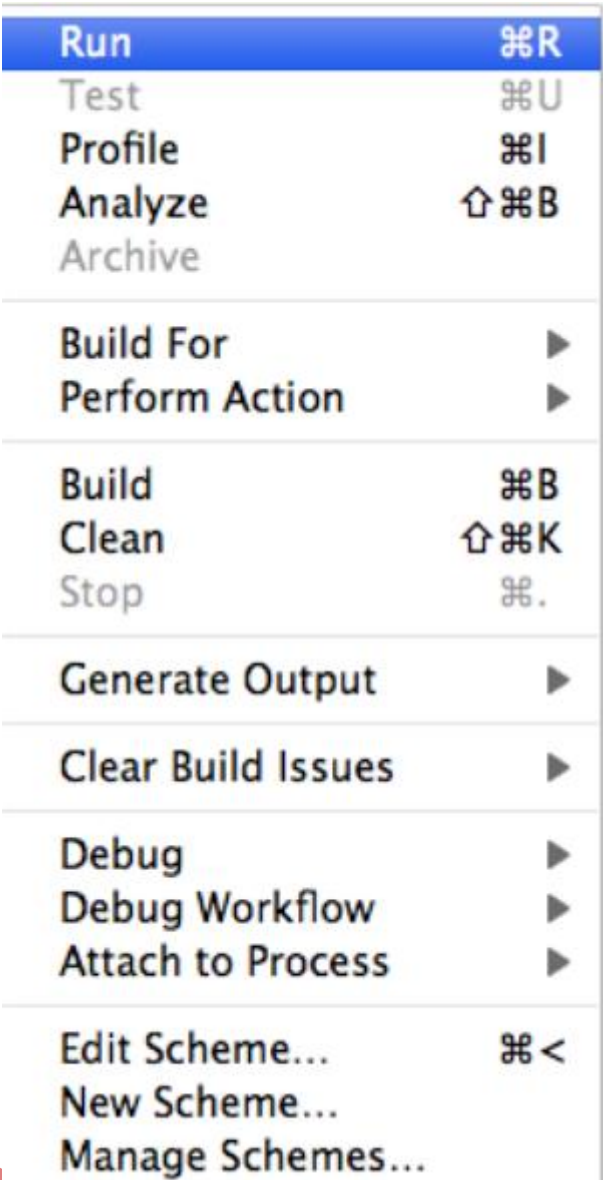


图 1-5 运行程序的菜单选项.

恭喜，现在如果一切正常的情况下，你就能很正常的看见你的应用程序在 iOS 模拟器中的安装，运行的过程。正如图 1-1 所示，你可以选择创建许多个模板类型的工程文件。

如下是一些你可以直接使用的一些工程模板。

Master-Detail Application

这个工程模板将会给你创建一个 split View Controller 左右切换分割的应用程序，split View Controller 将会在我们第二个章节的内容中提到。

Page-Based Application

这个模板将会帮你创建一个类似翻页的基础应用程序，你将会通过我们的第二章节学到更多关于这个模板的一些知识。

Empty Application

这个模板是一个空的工程程序，只是创建了一些简单的工程文件。没有一些基础的设置，一些基础的设置你都可以按照你的要求去添加。

1. 1. 4. 参考

暂无

1.2. 熟悉使用 Interface Builder(图形界面开发工具)

1.2.1. 问题

你想开始来为你的应用程序设计一些交互界面，但是又不想花费太多的时间来进行硬编码。

1.2.2. 方案

使用界面编辑工具(Interface Builder)

1.2.3. 讨论

Interface Builder 或者是 IB 都是 Xcode 所集成的一些工具或者文件用来进行用户的交互界面开发。这些工具目前只有 Mac 版本的，并且主要是用来针对 iOS app 的开发所设计的。

IB 工具能很好的操控.xib 文件。这些文件在以前的版本中一般称作 nib 文件。其实一个 nib 文件是由 XML 配置文件组成的，不过一般是由 IB 工具来进行管理的。

让我开始来使用 IB 工具吧，在想尝试这个之前，你必须利用 Single View Application 模板创建一个简单的 iOS 工具。如果不会创建，那么轻参照我们的 1.1 小节内容，创建一个工程并且选择一个磁盘位置来保存你的项目。



确保你的工程是一个 Universal 的应用，也就是针对所有的设备，类似图 1-2 所示。

当你的工程创建好了呢，那么首要的一件事情就是确保这个工程能够在 iPhone 模拟器中运行，并且做了正确的设置。如图 1-6 所示。

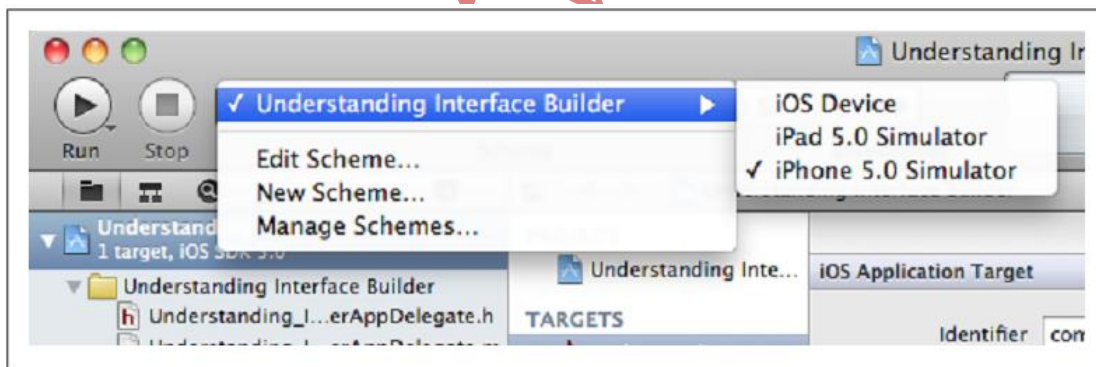


图 1-6 配置相应的运行模拟器

当配置好了之后呢，请使用快捷键 **Command + Shift + R** 来运行你的程序，你将会看到如图 1-7 中所示，在模拟器中能看到你的这一个空模板的程序。



图 1-7 正在模拟器中运行的 Single View 模板的应用程序

现在找到工程文件中的 `Understanding_Interface_BuilderViewController_iPhone.xib` 文件，并且双击打开它。Interface Builder 工具将会在你的 Xcode 工具中打开这个文件，并且展示一个用户的窗体界面给你。现在 IB 工具其实就已经打开了。然后选在 Xcode 右上脚的 View 菜单选项，然后看到一个新的 Menu 窗体打开，然后选择 Object Library，如图 1-8 所示

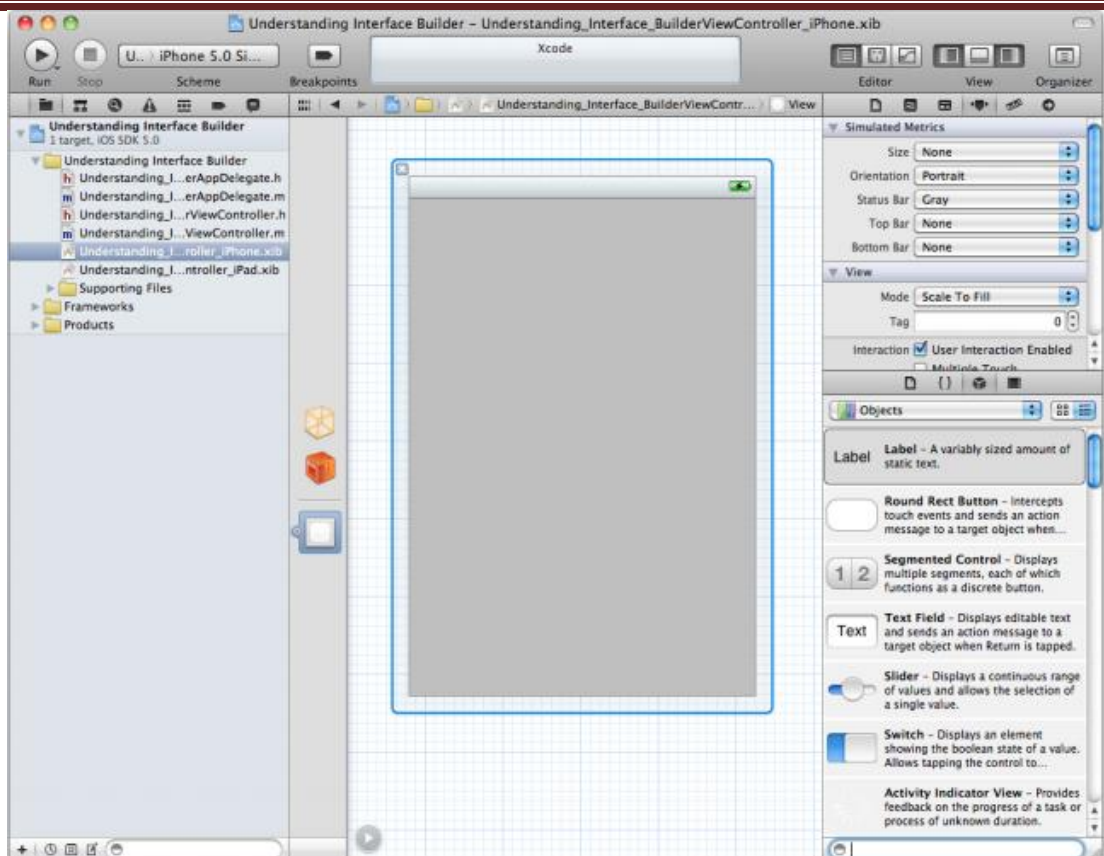


图 1-8 IB 工具以及所展示的用户界面及一些常用的控件

现在如果你已经看到了 Object Library,你已经有了很多控件的可选项来丰富你的用户交互窗体。你可以发现其中有许多控件,例如,按钮, on/off 的交换器,进度条,表格,等等。然后你可以用鼠标选择一个控件,然后拖动到你的用户窗体上面。就好比图 1-9 所示的那样

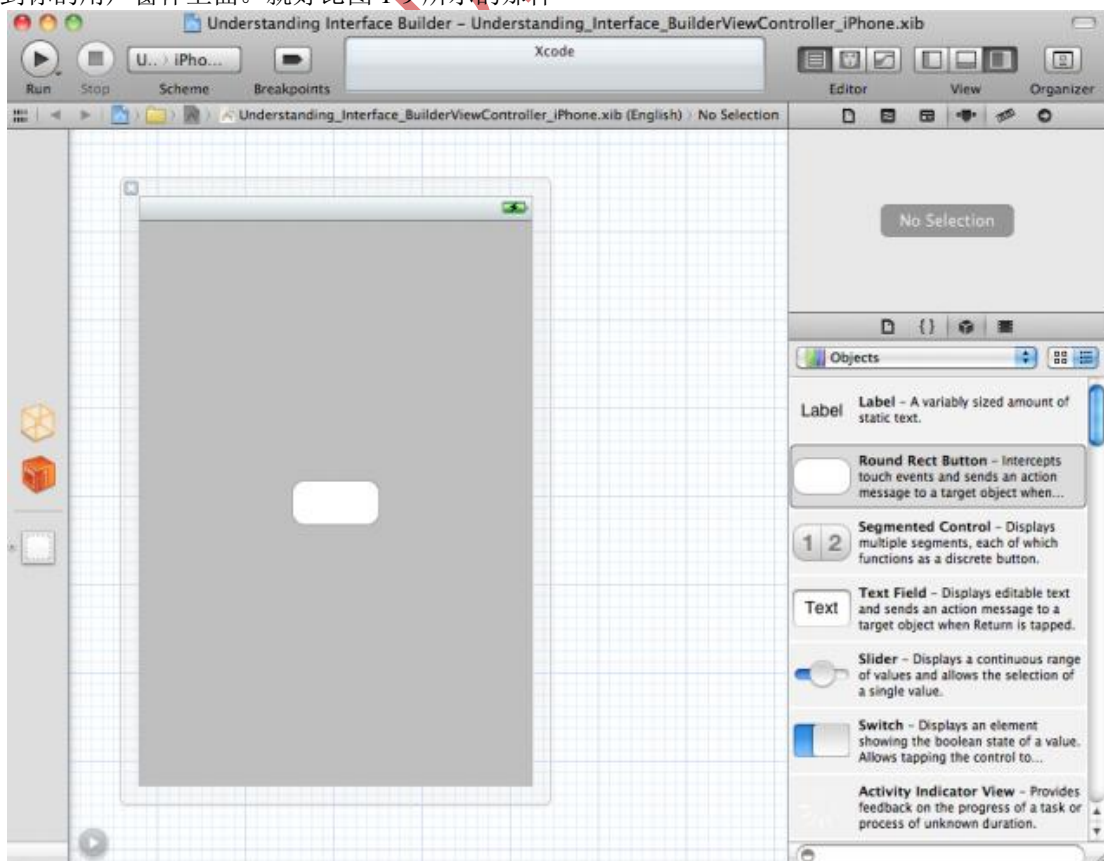


图 1-9 给 xib(用户交互界面)添加一个按钮

操作完这一步之后然后通过 File-->Save,然后来保存你刚刚修改过的文件。然后按照上面介绍的步骤运行你的程序。效果如图 1-10 所示。



图 1-10 模拟器中运行的添加一个 button 之后的效果.

目前为止，你肯能很困惑，但是别着急啊，更多的关于 Interface Builder 的介绍随后就会来。

1.2.4. 参考

暂无

1.3. 编译 iOS 项目

1.3.1. 问题

目前你可能已经知道如何创建一个工程，如何利用模拟器运行你的程序，但是你比较奇怪这个是如何进行的呢。

1.3.2. 方案

利用 Apple 的最新的编译器来编译和运行你的 iOS 应用程序，然后在模拟器中测试你的程序，使它能够更好的在你的设备中展现。

1.3.3. 讨论

创建一个 iOS 程序一般分为如下几个步骤.

1. 初步的计划
2. 原型的设计
3. 详细的设计
4. 功能的实现以及单元测试
5. 发布

在编码实现你程序功能的极端，你一般需要在你的模拟器中运行或者在许多个终端设备上运行从而能够保证你的程序能支持多种设备。从而根据你或者你们整个项目组中的设计方案来确保你们的应用程序更加的牢固和符合苹果 appstore 的规范。



程序突然终止或者崩溃将会是苹果 appstore 拒绝你发布应用的最主要的原因。通常 appstore 会主要拒绝那些经常崩溃的应用。因此为了避免这些问题，你就需要更多的利用模拟器和真机进行你程序的测试。

我们在编写代码的同时呢，我们也是一个学习的过程，women 需要确保我们所编写的是正确的。Xcode 把我们的代码编成一个可执行的代码的过程叫做编译。编译器做这个编译的工作。在 Xcode 中我们通常用许多不同的命令来编译我们的应用程序。

编译运行

采用这个编译器，你将能够断点测试你的应用程序在模拟器设备中。采用断点调试，你能够很好的来发现你程序中的一些问题。

编译测试

采用这个编译设置来运行你为程序所编写的单元测试用例，Xcode 将会在你发布应用程序之前运行这些单元测试用例。

编译配置

如果你想测试你应用程序的性能，需要利用这个来进行一些配置，然后你可以根据这个编译结果能够找出相关的程序瓶颈，内存消耗情况，和一些其他单元测试所不能发现的一些问题。

编译发布

当你确定你的应用程序已经没有什么问题之后，并且准备把他传递给测试人员测试的时候需要用到这个配置。

为了编译你的应用程序，在 Xcode 工具当中，最简单的方式就是选择 **Produce** 菜单类表，然后选择 **Build For** 然后选择一些编译配置来完成相关的步骤。

当你的代码中有一些错误的时候你该怎么做呢？在本章的第一小节中，我们创建了一个简单的 **Page-Based** 模板程序，因此，我们可以再次利用这些代码，现在我们打开 **RootViewController.m** 文件，可以看到如下的代码。

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}
```

然后把代码更改成下面这个样子。

```
- (void)viewWillAppear:(BOOL)animated
{
    [super nonExistantMethod];
    [super viewWillAppear:animated];
}
```

然后在重新在模拟器中运行这个程序的时候，你就会发现如下的代码错误。

```
error: Automatic Reference Counting Issue: Receiver type 'UIViewController'
for instance message does not declare a method
with selector 'nonExistantMethod'
```

这个问题就是编译器检查出来的错误：你所编写的代码并不能正常的编译，并且不能够正常的给 CPU 发送一些正常的指令。在这个正常的案例中，是由于编译器并不能检索到什么是 **nonExistantMethod**。这个足矣说明编译器能够查找那些能使你的应用程序出错的信息。

1.3.4. 参考

暂无。

1.4. 在模拟器中运行你的应用程序

1.4.1. 问题

你已经创建好了一个应用程序，并且很期待能让他在模拟器中运行，确保运行结果正常。

1.4.2. 方案

你需要使用 Xcode 左上角的 **Schema** 导航按钮来运行你的程序，然后选择任意一种模拟器来运行你的程序 (iPhone 或者 iPad)

1.4.3. 讨论

参考如下的步骤

1. 选择 Xcode 左上角的 **Schema** 导航栏选项，如图 1-11 所示。

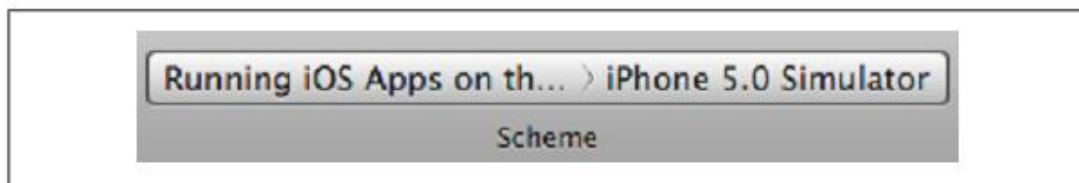


图 1-11 Xcode 工具中的 Schema 导航栏按钮。

2. 在 Xcode 开发工具中，你可以在一个工作空间中同时拥有多个项目，例如，在编写这个书的时候呢，

我是针对每个知识节点创建一个小的工程，然后在把他们添加到一个大的工程。左上角的这个 Schema 按钮所显示的内容就是你当前所选择的工程。因此，如果你点击了这个按钮，那么你将会看到如图 1-12 所实例的一样。所以，根据 Schema 按钮点击后的内容，然后选择你想运行在模拟器中的工程。

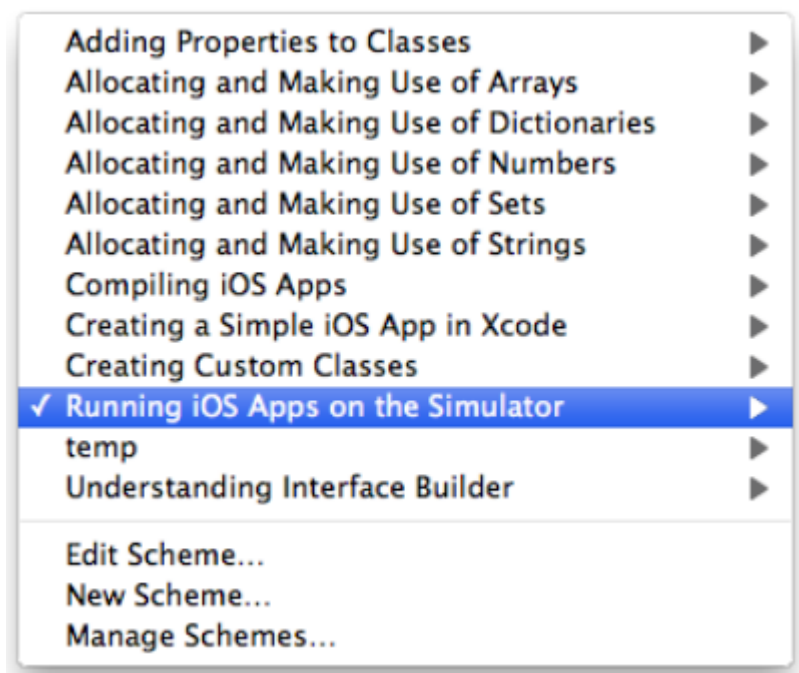


图 1-12 选择你期望运行的工程.

■ Schema 右边的按钮的是模拟器选择按钮，你可以选择 ipad 或者 iphone 模拟器来运行你的程序，我们在这本书中一般都是选择 iPhone 模拟器。所以尽量的多去点点，然后看看在不同模拟器中运行的效果。



注意这个选项，这个选项的依据就是你当初创建工程的时候所选择的 Device Family 的选项。如果你选择的 Universal 那么就表明你的程序适用于所有设备 iPad 或者 iPhone. 如果选择其一，那么就只能在你所选择的那个设备上运行测试。

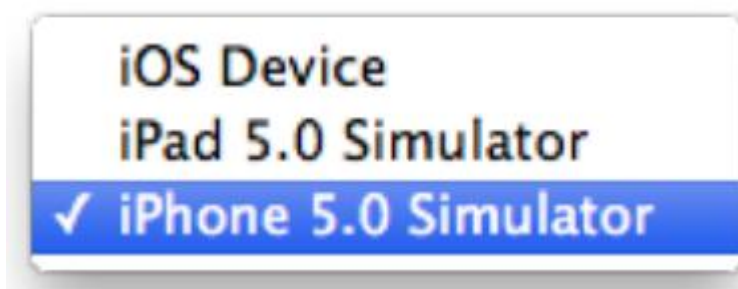


图 1-13 模拟器的选择

到目前为止你已经知道根据你的项目来选择不同的模拟器设备。其实很简单，选择你要运行的项目，然后选择一个模拟器，最后点击运行按钮，那么不出意外的话，你的模拟器就能正常的运行你的程序了。

1.4.4. 参考

暂无

1.5. 在 iOS 设备中运行你的应用程序(真机调试)

1.5.1. 问题

你已经创建好了你的 iOS 应用程序，但是你不知道怎么在你的真机中运行并测试

1.5.2. 方案

用 USB 把你的电脑和你的设备连接起来，请参照本章的第四小节确保你已经选择好了你需要测试运行的工程。这次不需要选择你的模拟器，而是选择 Device 然后进入到 Produce, 点击运行。

1.5.3. 讨论

每一个 Xcode 工具都会提供许多个 iOS SDK 的版本。由于我们一般都是使用最新的版本，而实际上，在 Xcode 上又不能把所有的版本都安装上去，也就是说类似 SDK3, SDK4 等等，Xcode 对于这些版本有一个数量上的限制。所以我们都是根据目前的情况来进行调整。

一个最简单的方法来检测你的 Xcode 是否已经连接了你的设备，就是当你用 USB 连接好了你的设备后，去看看，你的设备名字是否已经显示到 Scheme 的按钮上面。如果你连接好了你的设备，等同步完成后，过了一会发现已经有了你设备的名字，那么为了确保你的设备能够用来测试你的应用程序，你还需要按照下面几个步骤在做一些额外的设置。

1. 选择 Window 菜单选项\
2. 在 Window 菜单中选择 Organizer 选项。
3. 在新打开的 Organize 窗口中，在选择 Device 项，如图 1-14 所示

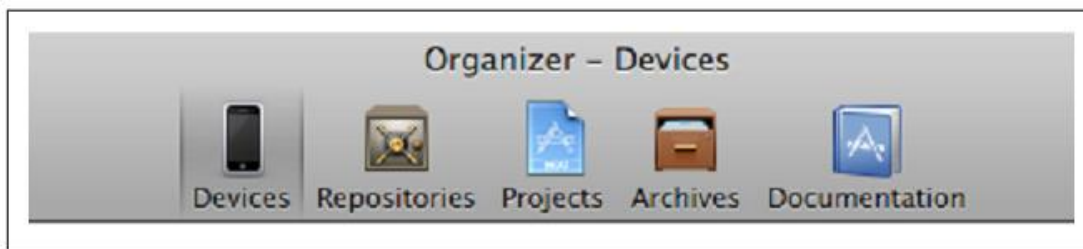


图 1-14

在 Organizer 窗体中选择 Devices 选项

4. 在左边的列表中，确保你已经点击选择了你的设备。如图 1-15 所示



图 1-15 通过点击选择你的设备\

5. 确保你能看到你的设备名称，并且能够看到右边打指示按钮是绿色的。如果是灰色的，表示目前这个设备并没有配置好。通过点击这个设备名称，你可以看到一个按钮上面表示将利用这个设备来发布程序，然后点击它。然后你会发现一个进度条展示出来，然后 Xcode 开始来检测这个设备。

6. 在进度条进行的时候，会有一个登录的提示框，需要让你录入你的开发者账号。这也就是说，Xcode 将会检测你的设备是否已经添加到你的开发者账号中，如果没有添加，那么你需要登录你的开发者账号，然后把你的设备添加进去。如图 1-16 所示，你需要录入你的开发者账号，然后点击确定进行登录。

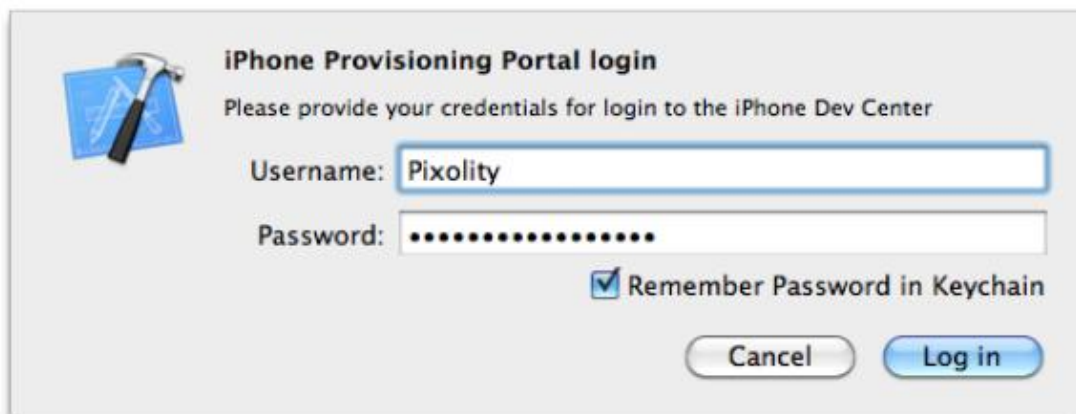


图 1-16 录入你的开发者账号

- 如果所有的配置都做了正常的设置，Xcode 将会检测你设备的 iOS 版本，看看是否支持运行。如果所有的配置都符合规范，那么左边的按钮将会变成绿色。如图 1-17 所示



图 1-17 设置好的设备

8. 现在请关闭你的 Organizer 并且返回到你的 Xcode 中，如果你点击 Scheme 上的导航按钮，那么你将看到你的设备名称也在里面。如图 1-18 所示。

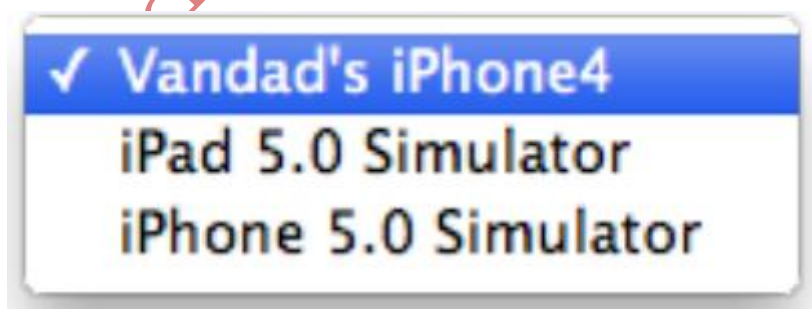


图 1-18 等待运行的设备

如果 Xcode 检测不到你的 iOS 设备的版本，那么它将会指示一个黄颜色的灯。在这种情形下，你需要自己确保让你的 Xcode 程序的版本和你的 iOS 设备的版本相匹配，要么，你需要更改一下你的设备的版本来匹配 Xcode 编译的版本。一般你能够直接看到 Xcode 所支持的一些版本。事实上当你操作的时候，如果版本不匹配，那么 Xcode 也会弹出提示框来说明不匹配的原因。如果这些原因是你的设备造成的，那么提示框也会说你需要什么版本的环境才能支持你所需要编译运行的程序

1. 5. 4. 参考

暂无

1.6. 打包并发布你的 app 为发布到 appstore 做准备.

1.6.1. 问题

如果你打算传递你的 app 给其他的人, 让他们帮你来测试, 或者体验一下它, 所以你需要做一下签名认证, 然后才能发布到 appstore 里面。

1.6.2. 方案

你需要给你的程序打包, 签名, 认证。

1.6.3. 讨论

为了能够打包, 签名你的程序, 你需要按照如下步骤来操作。

1. 首先要确保你的程序已经完全的在模拟器中正常的运行, 并且比较稳定, 不会出现类似内存泄露等明显的问题。

2. 然后你需要一个 iOS 设备的 UDID(产品唯一编码)以便让你的应用在那个设备上测试。如果你没有你也可以问你的朋友或者同学借用一下。

3. 在你 iOS 的设置界面, 添加这个 UDIDs.

4. 创建一个分发加密文件, 这个文件是一个二进制的 XML 文件, 有了这个文件才能让你的应用在 iOS 认证过后的设备上运行。

5. 当你已经创建好了这个以.mobileprovision 结尾的分发加密文件之后, 把这个文件添加到 xcode 中, 从而让 xcode 能够做正常的发布使用。

6. 在 xcode 中, 选择 Produce 菜单, 然后选择 Archive. Xcode 将会发布你的应用程序, 当发布完成的时候, Organizer 窗口将会展示出来。然后你可以把你的应用以.ipa 结尾的文件导出来, 有了这个文件你可以通过 iTunes 把应用安装到 iPhone 或者 iPad 等设备中。

为了让你的应用能够分发到你的测试人员或者你的朋友, 你需要创建这个认证加密文件, 如果你需要创建这个文件你需要参考如下步骤。

1. 通过浏览器登录苹果开发者中心。

2. 选择 iOS Provision Portal 通过右边的菜单列表。

3. 如果你并没有创建一个分发权限认证, 那么你需要按照如下步骤创建。

A. 通过左边的 iOS 产品配置菜单, 选择 Certificates.

B. 在右边展示的窗口中, 选择发布子选项。

C. 按照操作步骤, 录入相关数据, 然后需要你创建一个 Keychain 操作的权限文件, 你需要上传你电脑上的这个文件到表单中。然后操作完成之后你就创建了的分发证明。

D. 然后下载这个文件, 并保存在你的电脑中, 然后双击安装。

4. 然后你需要通过左边的菜单选项, 点击 Devices 节点。

5. 然后点击添加一个设备。

6. 你需要录入一个设备名字和这个设备的 UDID 号, 如果你需要同时添加多个设备, 那么只需要点击+号来再添加一组数据。但是苹果终端已经做了一个限制, 你最多只能添加 100 个不同的设备, 如果你需要添加超过 100 个的话, 那么你需要一个企业开发级的账号, 需要 299 美元。



当你的设备已经跟你的这个开发者账号关联起来了, 你不能把他删除掉, 除非过了一年。当过了一年之后或者你重新续费, 这个时候你可以删除它们来添加新的。这样可以确保你不是没有事在那添加着玩的。

5. 当你把这些事情都操作完成之后, 点击提交按钮。

6. 然后点击左边菜单栏的 Provisioning

7. 然后点击 Distribution 标签。

8. 然后点击 New Profile 按钮。

9.在创建 iOS 发布配置界面的时候, 确保发布的方法是 Ad Hoc

10.然后为你的分发加密文件添加一个描述.

11.在 App ID 下拉列表中, 选择 Xcode Wildcard AppID.这个的作用是为了让这个文件能够适用于你所有的 iOS 应用.

12.在 Devices 选项中, 你需要选择那些设备可以适用于这个加密权限文件, 从而让你的设备能够运行你的这个 app.

13.在你操作所有如上步骤之后, 你需要点击确认按钮发布。

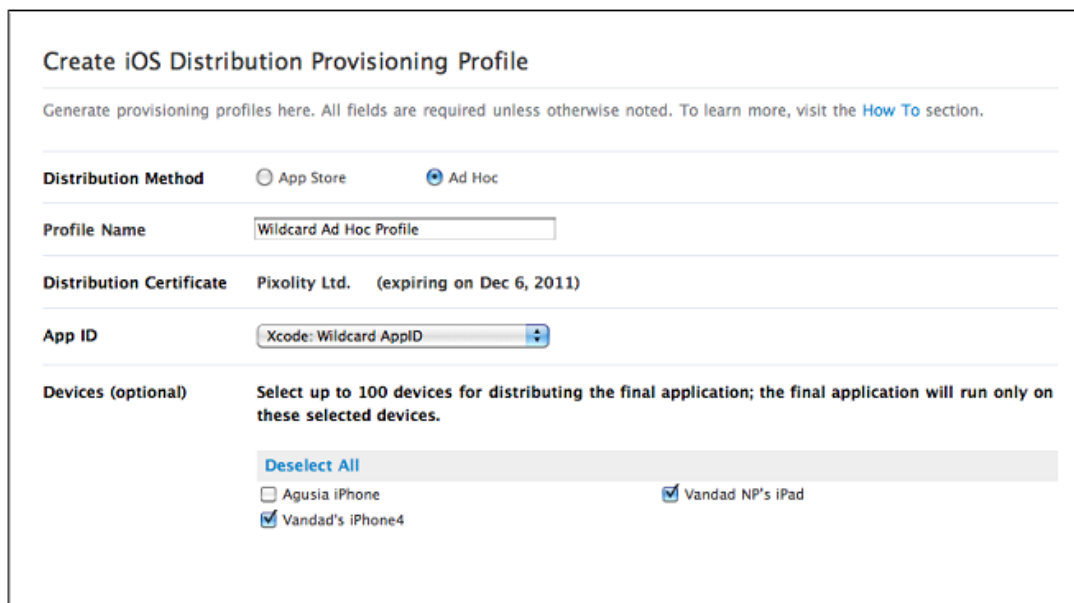


图 1-19 创建一个新的 Ad Hoc Provision 文件

14.然后返回到 Distribution 标签, 然后下载你刚才创建的加密权限文件。如果没有那个文件就刷新一下页面。

15.然后你把这个文件拖到 iTunes,iTunes 将会安装这个文件。

所有操作完成之后, 你可以创建一个发布文件。参照如下步骤.

第 1 章 选择你 Xcode 中的你的工程项目主文件.

第 2 章 现在你将会看到 targets 列表项,

第 3 章 选择 Build Settings 如图 1-20 所示.

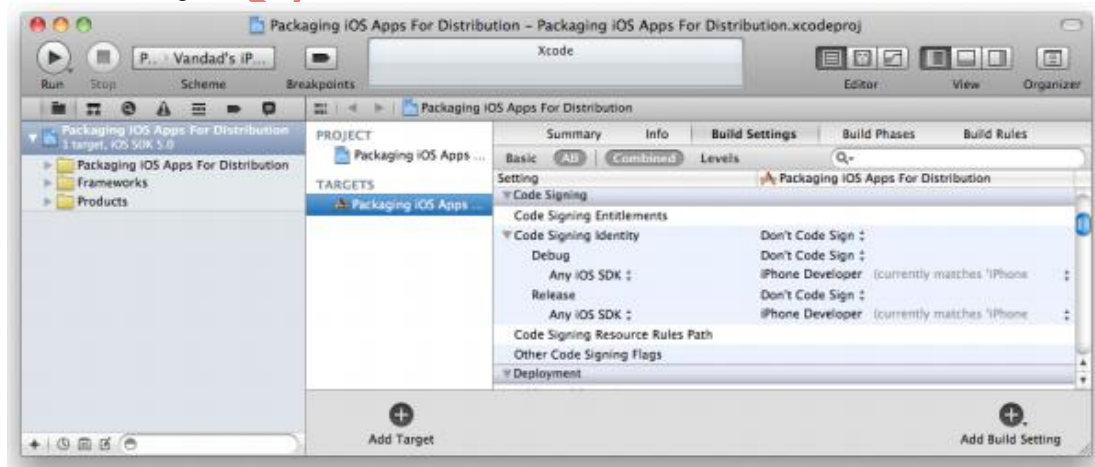


图 1-20 Build settings view

4.如 1-20 所示, 然后选择 build Setting ,然后下拉选择 Code Signing category 目录
5.然后依次 Code Signing Identity→Release and Code Signing Identity→Release→Any
iOS SDK 如图 1-20 所示.

■ 如图 1-11 所示.,然后确保在 Scheme 导航栏上显示的是你的 iOS 设备的名字。

■ 然后选择 Produce 菜单栏, 选择 Archive

在操作完这些之后, 你将会通过 Xcode 的 Organizer 查看你的配置情况, 如图 1-21 所示

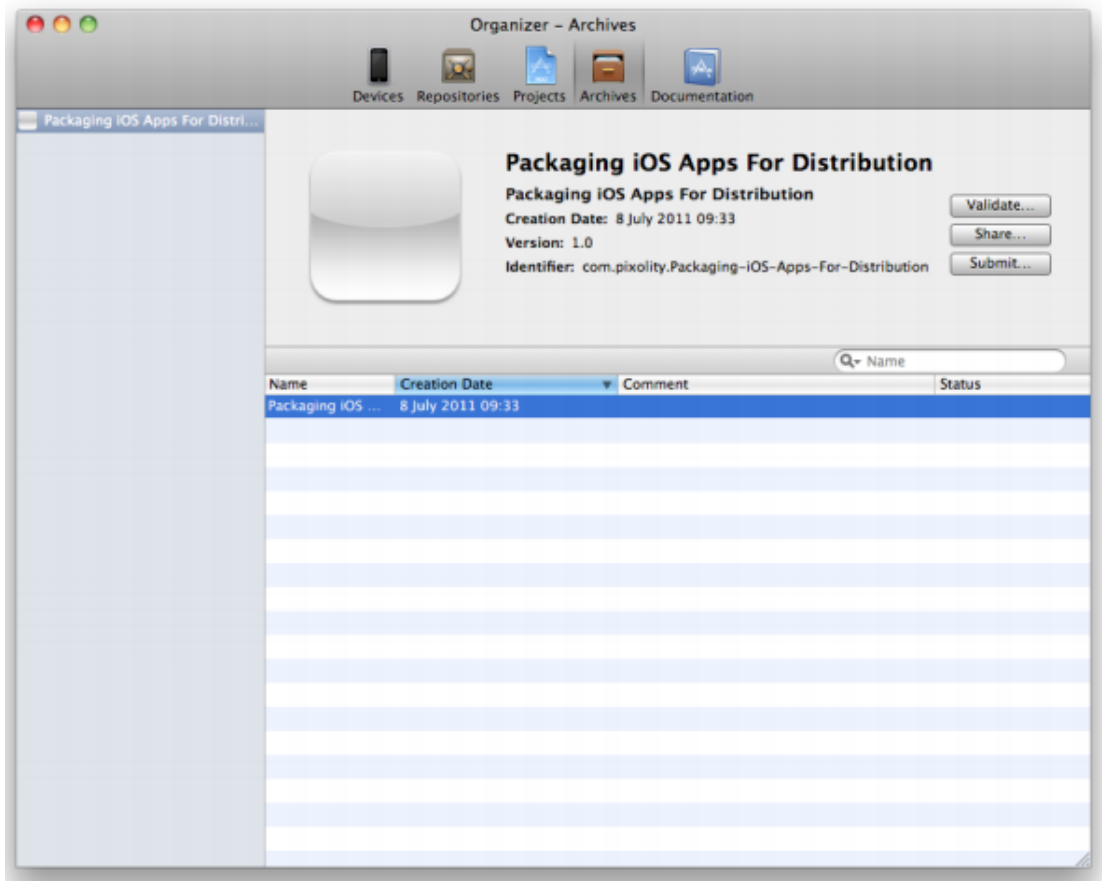


图 1-21 Organizer 配置成功界面.

8.然后选择 Share 按钮, 接着按照图 1-22 所示操作.

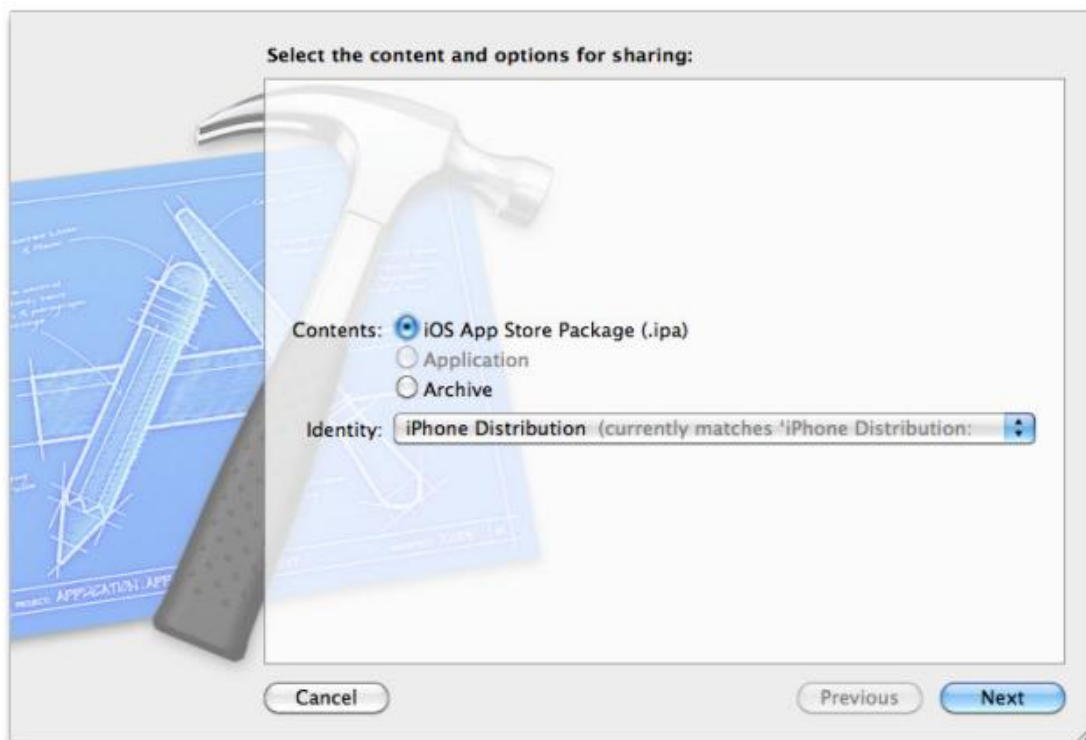


图 1-22 选择相应的发布终端界面.

9.按照如上图的选项进行相关操作.

所有的操作都完成之后你将会拥有一个以.ipa 结尾的文件, 然后你可以把这个文件共享给你的朋友们。确保你已经在你的开发者账号中关联了他们的设备序列号..他们将会需要你的这个.ipa 文件以及.mobileprovision 文件才能安装和运行你的程序.

1.6.4. 参考

无.

1.7. Objective-C 语言中如何声明变量

1.7.1. 问题

你想在你的程序中声明一些变量, 但不知道如何操作。

1.7.2. 方案

在苹果的开发框架中, 有自己一套的命名规则和不同的数据类型定义, 比如, integer, array, string 等等。在 xcode 程序中, 你需要首先给一个变量定义一个数据类型, 然后在给这个变量起一个名字。变量名字需要遵守以下规则。

- 遵守如下规则, 如果这个变量名字是一个单词, 那么所有的单词都必须小写, 如果这个变量名称是有多个单词组成, 那么第一个单词必须全部小写, 后面单词的每一个开头字母需要大写, 其他的小写。比如"first counter"可以声明它为"firstCounter"
- 变量名一般最好不要有下划线, 例如 my_variable_name 应该修改成 myVariableName.
- 变量名称最好只包含英文字母和数字, 不要什么特殊符号。

让我们一起来看几组实例, 有几组私有的数据类型, 其实是 objective-c 延伸过来的, 比如说 NSInteger 表示 integer., 精度大一点的可以使用 NSUInteger.



带符号的整数可以表示复数，但是无符号的就不行。

```
NSInteger signedInteger = -123; //可以表示负数
```

```
NSUInteger unsignedInteger = 123; //不可以表示负数。
```

还有许多其他的规则，就像前文提到的一样，一般命名都是采用驼峰规则，但是你也可以发挥自己的想法，在符合规范的前提下。最后，我建议无论你目前是有你自己的公司或者你在别人的公司工作，都最好能按照如下规范来。

1. 给你的变量一个很明确的描述，一看就知道这个变量是用来干什么的，避免使用一些简单的类似'i'或者'x'，这样让人看到很不能理解的变量名称。因为这些变量只有你自己能看明白，而别人并不一定能看明白。编译器一般都是能支持变量名称在小于 50 个字符的名称编译，所以，不需要考虑长度问题，尽管去做得明了些吧！

2. 避免创建一些模糊的变量名称，例如，如果你想给一个人的名字起一个变量名称，你不能起成这样“theString”或者“theGuy”，因为这个一点意义都没有，你可以起成“firstAndLastName”等等，因为这样别人一看都能明白是什么意思。

3. 避免你的命名会导致一些错误，比如“fullName”就比“_____full_____name”好，一般要避免使用下划线在多个单词之间。

1.7.3. 讨论

变量其实就是你文件的一个标示符，是需要消耗一定内存的，比如说，如果你想删除 100 个文件，而这 100 个文件又是以 1.png, 2.png, 3.png 等等这样的形式来命名字的，难道你说你想去起一百个变量。并不是你想要的，你仅仅需要做的就是起一个变量，然后循环迭代来删除就可以啦。程序员一般使用一些变量来进行数学计算，或者一些简单的字符串的拼接，就好比把 firstName 和 lastName 组合起来创建一个完整的名字。

每一个变量其实都应该有一个数据类型，这个数据类型可以告诉编译器如何来解析运行这个程序。你的变量类型是由你的实际数据结构来选择的。例如，如果你想用 integer 来保存 123 就是可以的，但是保存 123.456 就是不行的，因为，数据类型的精度没有达到。如果我们创建一个浮点型的数据就能够来保存这个数据了。

Integer 和 floating 我们在 objective-c 中一般用 NSInteger 和 float 表示。如下就是一些数据类型的案例。

NSInteger 能用来保存有符号的（正或者负）的整形变量。

NSUInteger 用来表示无符号的，要么是正，要么是负的整形变量。

Float 浮点型的数据，例如 1.23

NSString string 类型的数据，比如“Mrs Thomson”

NSArray 其实也就是一个数组，比如说你有十个文件对象，那么你可以把他们保存在这个里面。

NSSet 你可以保存唯一的，没有重复的集合对象。

1.7.4. 参考

无

1.8. 利用 objective-c 中的 IF 逻辑判断分支结构

1.8.1. 问题

你需要比较两个变量在 objective-c 的程序中，并进入到不同的分支结构中

1.8.2. 方案

使用 if 条件判断。具体使用方法，请参考如下小节中的内容。多种分支结构，根据判断条件来确定分支的进入情况。

1.8.3. 讨论

其实，我们日常的生活中很多地方都使用到了 if 这样的逻辑分支判断。例如，有些时候你可能会这么说“如果我见到他，我肯定会告诉他.....”或者“如果我很久都不回来的话，我会把电脑睡眠了”所有的这些声明都是有一个条件的，这个就如同在 Objective-C 中的一个条件判断一样，你可以利用这点让程序更复杂化，然后确保你的程序能按照相应的逻辑条件执行相应的分支结构代码。如下是一个常用的结构。

```
if (<replaceable>condition</replaceable>){  
    /* Your code to get executed if the <replaceable>condition</replaceable> is met */  
}
```



只要不是 zero/nil/NULL,而是一些其他的参数值，这个分支结构中的代码都会运行的。

如果一个 if 结构有一个其他的分支结构，那么一般是这样写的

```
if (<replaceable>condition</replaceable>){  
    /* Your code to get executed if the <replaceable>condition</replaceable> is met */  
} else {  
    /* Code to get executed if <replaceable>condition</replaceable> is not met */  
}
```

在 else 的分支中还可以继续分出很多个分支结构出来，这将会是一件很有趣的事情，这点你可以用生活中的例子来想象,例如“如果 coffee 店是开着的话，我将会去喝咖啡，如果那一家经常去的没开门的话，我就换一家，如果都没开门的话，那么我就只能回家了，然后自己泡点茶喝。”这其实就是一个分支结构的描述。如下 objective-c 代码可以来描述这些场景。

```
if (Coffee place A is open){  
    Get a Tall Latte from coffee place A  
} else if (Coffee place B is open){  
    Get a Cappuccino from coffee place B  
} else {  
    Come back home and make tea  
}
```

无论判断条件中的结构是什么，那么都需要是一个确切的 Boolean 类型的值，要么是 YES，要么是 NO，类似于如下结构。

```
if (YES){  
    /* This code will get executed whenever the app gets to it */  
} else {  
    /* The app will NEVER get here */  
}
```

为了做一个更明确的实例，你可以参考如下的代码

```
NSInteger integer1 = 123;  
NSInteger integer2 = 456;
```

```
if (integer1 == integer2){
    NSLog(@"Integers are equal.");
} else {
    NSLog(@"Integers are not equal.");
}
```



我们使用双等于号，其实就是表示一个判断的过程，判断两个值是否相等。这样编译器在编译之后能得到一个 YES 或者 NO 的判断结果值。

如果你比较的是一些对象，那么你最好使用 isEqual: 方法来进行判断。实例代码如下：

```
NSObject *object1 = [[NSObject alloc] init];
NSObject *object2 = object1;
if ([object1 isEqual:object2]){
    NSLog(@"Both objects are equal.");
} else {
    NSLog(@"Objects are not equal.");
}
```



目前你不需要疑惑这两个类到底是个什么东西，我们将会在后面的一个小节中介绍。

一些其他的类，例如 String，因为他们已经有自己的比较方法，我们可能不适合这种方法来比较，如果你有两个 string 类型的变量，但是变量的内容是一样的，当你使用 isEqual: 的时候，实际返回的结果却是 NO，因为他们是不同的类，即使他们有同样的类容，正因为这个原因，objective-c 的操作中给这个类又定义了一个比较的方法。如果你想了解更多关于类的相关知识，请参照本章的 1.11 小节。

分支结构也是可以写成没有中括号的 {} 形式的，他们的区别就是，如果你有了 {} 号来包围你的分支，你可以在里面写多个操作，如果没有那么只能写一行。

如下例子展示了没有 {} 的标准写法。

```
NSString *shortString = @"Hello!";
if ([shortString length] == 0)
    NSLog(@"This is an empty string");
else
    NSLog(@"This is not an empty string.")
```



了，如下

```
NSString *shortString = @"Hello!";
if ([shortString length] == 0)
    //NSLog(@"This is an empty string");
else
    //NSLog(@"This is not an empty string.");
```

那么你的这段代码将会出现问题, 这样会带来一些不必要的麻烦, 所以我们建议一般还是要使用 {} 来控制你的分支作用域.

1.8.4. 参考

无

1.9. For 循环的结构

1.9.1. 问题

你需要用代码来实现如果从一个集合或者其他的数据类型中迭代出其中的每一个子元素

1.9.2. 方案

使用如下的循环结构方式, 我们称之为 For 循环

```
for (<replaceable>code to execute before loop</replaceable>;  
    <replaceable>condition to be met for the loop to terminate</replaceable>;  
    <replaceable>code to execute in every iteration of the loop</replaceable>){  
}
```



for 循环中的三个参数是可选的换句话说就是你可以一个参数都不录入, 如下
`for (;;) { YOUR CODE HERE }`

如上的这个循环就是一个通常所说的“死循环”, 因为没有有一个终止的条件, 所以这种类似的代码结构最好不要出现我们的程序当中, 也尽量避免这种操作.

1.9.3. 讨论

循环在我们的程序中是非常有用的, 我们可以从一个变量节点到另外一个节点。或者从一个开始点到结束点。例如你可以循环的查找一个变量中到底有多少个 'A' 这样的字符。或者你可以来循环的输出在一个目录下面有多少个文件。利用循环我们能找到有多少个文件, 然后可以循环迭代从第一个文件到最后一个文件.

通常, 我们都需要给循环添加一个计数器, 例如, 如果你想分别读出字符串 "C-String" 中的每一个字符的话, 你需要创建一个计数器, 也就是一个临时变量. 如果你的字符串有 10 个字符, 那么你需要创建一个索引的计数变量从 0 到 9. 由于你的字符串的长度可能会发生变化, 你可以把它放在一个方法中来判断长度, 如下所示:

```
char *myString = "This is my string";  
NSUInteger counter = 0;  
for (counter = 0; /* Start from index 0 */  
     counter < strlen(myString); /* Exit loop when we reach last character */  
     counter++){ /* Increment the index in every iteration */  
    char character = myString[counter];  
    NSLog(@"%c", character);  
}
```

其中计算长度的方法 `strlen(myString)` 会在循环开始之前就计算好。事实上, `for` 循环中的三个参数都是可选的, 但是我们建议你根据你的设计, 然后合理的利用这三个参数。

下面让我们来研究一下, 当我们把第一个参数置空将会出现什么情况。如同上段代码一样, 我们设置了一

个变量从 0 开始，然后开始我们的循环，如下代码，我们仍然这么操作，如果按照如下代码，我们也并不会出现什么问题，这也就表明，如果我们没有给这个参数设置 0，如果不影响操作的话，我们也可以不用设置这个参数。

```
char *myString = "This is my string";
NSUInteger counter = 0;
for (; /* empty section */ 这个地方为空了.
    counter < strlen(myString); /* Exit loop when we reach last character */
    counter++){ /* Increment the index in every iteration */
    char character = myString[counter];
    NSLog(@"%c", character);
```

For 循环中的第二个参数就是比较重要的，因为这个参数关系到这个循环什么时候停止。如果没有第二个参数，那么可能就表示一个“死循环”，将导致这个循环一直进行下去。因此我们应该根据实际情况来设置条件，并要考虑一些异常的情况发生。

For 循环的第一个参数无论怎么开始的最终都应该包含在第二个参数的范围之内，如下：

```
for (NSUInteger counter = 0;
    counter < 10;
    counter++){
    NSLog(@"%lu", (unsigned long)counter);
}
/* "counter" is NOT accessible here. This line will throw compile time error */
NSLog(@"%lu", (unsigned long)counter);
```

For 循环的第三个参数也是比较重要的，因为这个关系到每次循环之后要做的一些操作。这个标示这计数器参数每次要做什么样的变化，如下：

```
NSUInteger counter = 0;
for (counter = 0;
    counter < 4;
    counter++){
    NSLog(@"%lu", (unsigned long)counter);
}
NSLog(@"%lu", (unsigned long)counter);
上段代码将会打出如下信息：
```

```
0
1
2
3
4
```

所以我们的计数器将会达到 4 这个数，但是我们的循环应该比 4 小啊。因为我们是从 0 开始的。这个就表明，每次当循环完成之后，第三个参数都会让计数器参数做一个逻辑规则运算，然后看看是否符合第二个参数的要求，是否在第二个参数的范围之内，如果不在那么循环将会停止，如果还在的话，循环将会继续进行。

1.9.4. 参考

无。

1.10. 采用 While Loop 的循环结构。

1. 10. 1. 问题

你将想让你的程序根据一个变量的值的判断来进行一直的循环。

1. 10. 2. 方案

你需要采用 `while` loop 的结构来实现你的方案，结构如下。

```
while (<replaceable>condition</replaceable>){  
    CODE  
}
```



这个 `while` 条件中的设置不应该是类似 `(zero/nil/NULL)` 这样的值。

1. 10. 3. 讨论

这个 `while` loop 的循环结构其实与上一个小节我们提到的 `for` 循环是一样的，因为 `while` loop 循环只是有一个判断条件来判断是否执行循环里面的过程，如果条件符合，那么将会执行循环里面的内容，如果不符合，那么循环将会终止。就好比 Mac OSX 操作系统里面的 Dock 快捷键标签盘样的，当你把鼠标指过去的时候，图标将会伸展出来，当你转移到别的图标上的时候这个图标将会收缩回去，而新的图标又会伸展出来一样。如果用户一直的把鼠标或者聚焦到某个快捷键图标上，那么这个图标将会一直伸展出来。

其实这个 `while` loop 的循环也并不是太好用，如果你有一个计数器的操作，那么还是建议使用 `For` 循环吧，如果你仍然坚持要使用 `while` loop 循环，那么你也可以创建一个计数器，然后在内部来管理它，代码如下

```
NSInteger counter = 0; //你自己创建的一个计数器  
while (counter < 10){  
    NSLog(@"Counter = %lu", (unsigned long)counter);  
    counter++; //需要你自己来进行维护管理。  
}
```

或者你可以为你的 loop 添加一个明确的标志位判断，代码如下：

```
BOOL shouldExitLoop = NO;  
NSInteger counter = 0;  
while (shouldExitLoop == NO){  
    counter++;  
    if (counter >= 10){  
        shouldExitLoop = YES;  
    }  
}  
NSLog(@"Counter = %lu", (unsigned long)counter);
```

上段代码将会打印出 `Counter = 10`

因此我们这段代码其实就是让循环一直操作，当我们的计数变量大于 10 的时候，那么这个循环将会终止。同时你也应该考虑到一个“死循环”的概念，防止循环不能终止。类似如下代码：

```
while (YES){  
    /* Infinite loop */  
}
```

在使用 `while` loop 循环的时候，你一定要确保你的条件有一个终止的值，因此你就必须要确保你所设置的条件能够当你在循环你的代码的过程中，经过一定的运算，最终会有一个终止的临界点，这样就能防止一些内

存的无限消耗。

如下是另外一个 while loop 的循环实例。

```
char *myString = "Some Random String";
NSUInteger counter = 0;
char character;
while ((character = myString[counter++]) != 'R' &&
      counter < strlen(myString)){
    /* Empty */
}
NSLog(@"Found the letter R at character #%lu", (unsigned long)counter+1);
```

如上代码，我们就是循环的输出一个字符串中的每一个字符，然后判断是否有一个字符是和 R 相等的，如果相等，那么这个循环将会退出并终止。因此我们碰到这种情况的时候为了防止无限循环，我们又添加了一个循环的长度的判断 strlen(myString) 来终止我们的循环，防止出现“死循环”的情况。

死循环实际上是程序中的一个 bug，假设我们的这个字符串中没有'R'这个字符，我们也并没有添加长度停止的判断，那么肯定就出现了“死循环”，因此，我们可以创建一个标志位的变量(YES/NO)来表示条件，如果符合，那么循环将会停止，如果不符合，那么将会循环到字符串的最后一个字符之后自动终止。这样就是一个比较好的规避无限循环的方法。

其实 while loop 这种循环对于遍历数组是比较有用的。例如变量“cString”是一组字节组成的，类似如下代码，如果你来查找这个变量中的每一个字节的话，你可以来利用 while loop 循环，然后将其中的每一个自己都迭代出来，然后进行比较。代码如下。

```
char *cString = "My String";
char *stringPointer = cString;
while (*stringPointer != 0x00){
    NSLog(@"%c", *stringPointer);
    stringPointer++;
}
```

这个代码将会把 cString 变量中的每一个字符都答应出来，然后打印完成之后停止。

在使用 while loop 的时候，你可以创建一些方法，类似于 strlen() 这个函数的功能一样，然后来判断一个字符串的长度，代码如下。

```
NSUInteger lengthOfCString(const char *paramString){
    NSUInteger result = 0;
    if (paramString == NULL){
        return 0;
    }
    char *stringPointer = (char *)paramString;
    while (*stringPointer != 0x00){
        result++;
        stringPointer++;
    }
    return result;
}
```

1.10.4. 参考

无

1. 11. 创建自定义类

1. 11. 1. 问题

为了方便现在或以后使用，需要打包一组具有相关功能的一个可重用实例。

1. 11. 2. 方案

创建自定义的类

1. 11. 3. 讨论

类是一个抽象实体。类包含方法和其他语言的具体结构。比如要编写一个计算器程序。在创建用户界面时，想让计算器上的每个按钮具有一个黑色的背景、白色的文本和凹凸效果的用户界面，就像一个真实的按钮。那是不是要将这些按钮之间的所有共同属性都设置到你的用户界面上呢？对！创建一个类来表示所有的按钮，编写一次代码并重复使用多次就可以了，这是一个很好的方法。

在 Objective-C 中，类通常由以下两部分组成：

头文件：这里是你定义的类，基本上是：接受用户输入，旋转形状，等等。但是头文件并不执行该功能。头文件是以 `.h` 为后缀的扩展名。

执行文件：在头文件中定义类的功能后，你在这里编写所有功能的实际代码。执行文件是以一个 `.m` 为后缀的扩展名。

现在继续按照以下步骤创建一个类来增加了解：

1. 打开 Xcode，在菜单栏的 File 里选择 New File。
2. 出现一个对话框，如图 1-23 所示。这里只需选择列表右边的 Objective-C 类。要确定在左侧的 iOS 是被选中的。然后按“下一步”按钮。

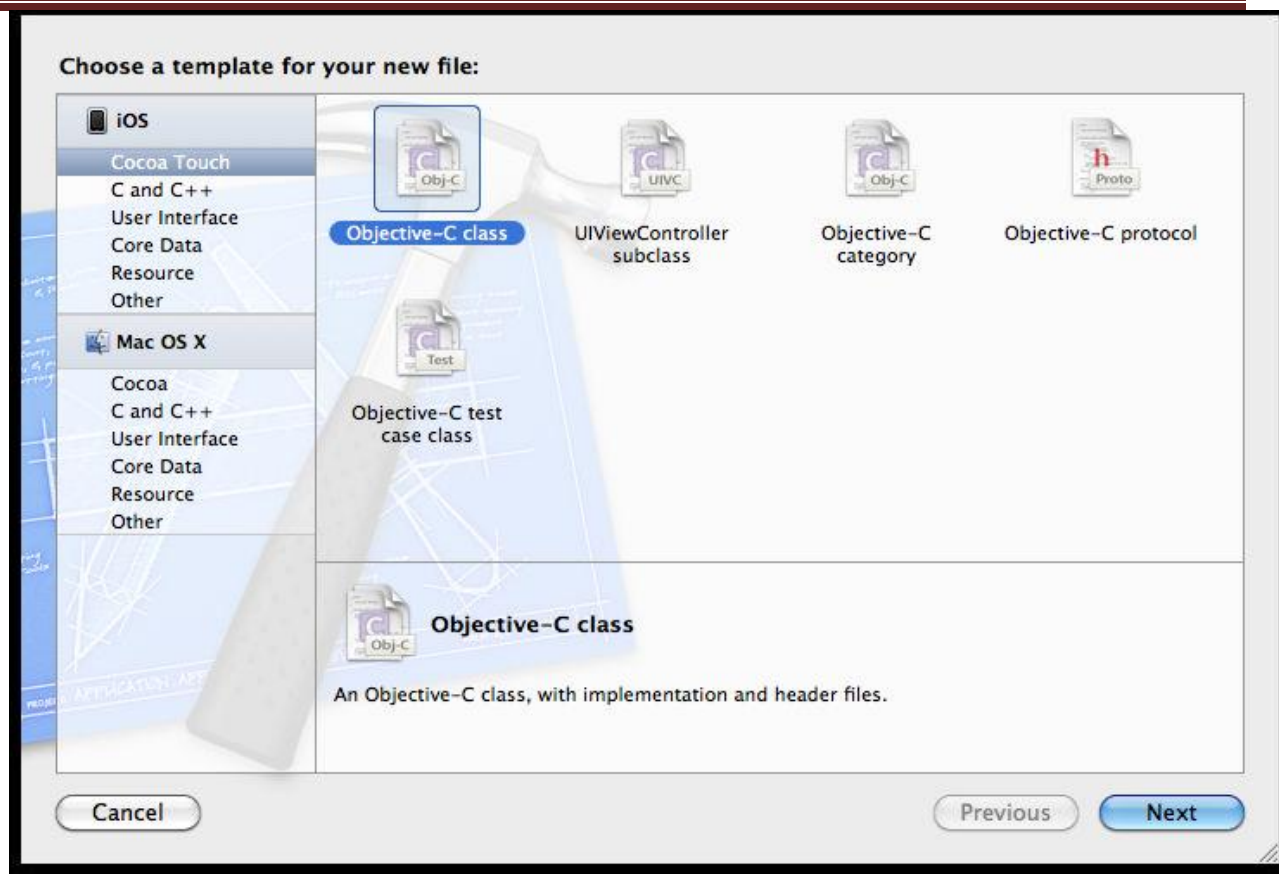


图 1-23. 在 Xcode 里的添加文件

3. 在接下来的对话框中，确定 Subclass of 后的文本框里显示的是 NSObject。然后按下 Next 按钮（如图 1-24）

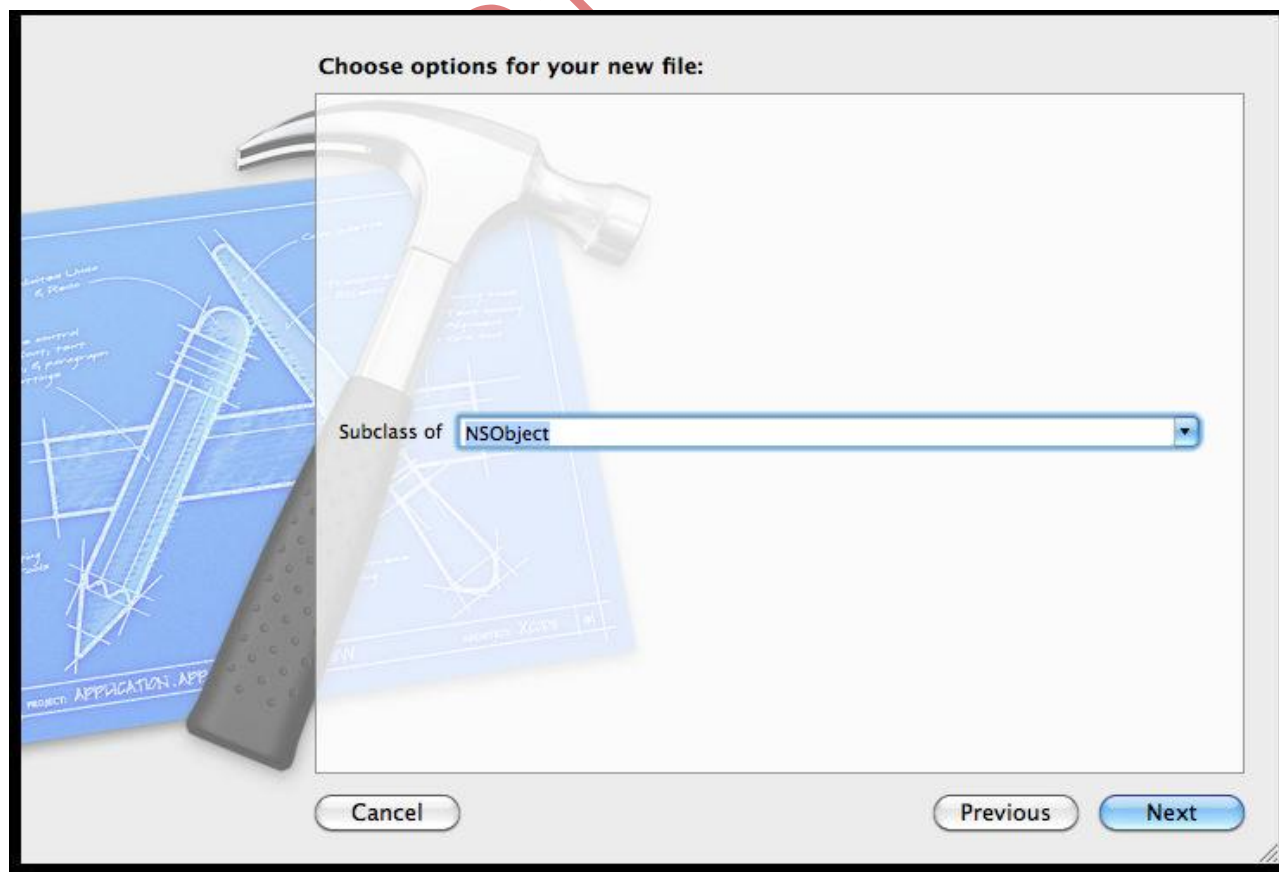


图 1-24. 选择新类的基类

4.在接下来的对话框中（如图 1-25），确定另存为文本框里显示的是 Person（这是类的名称）。在对话框的底部，确定类保存在正确的文件夹中。

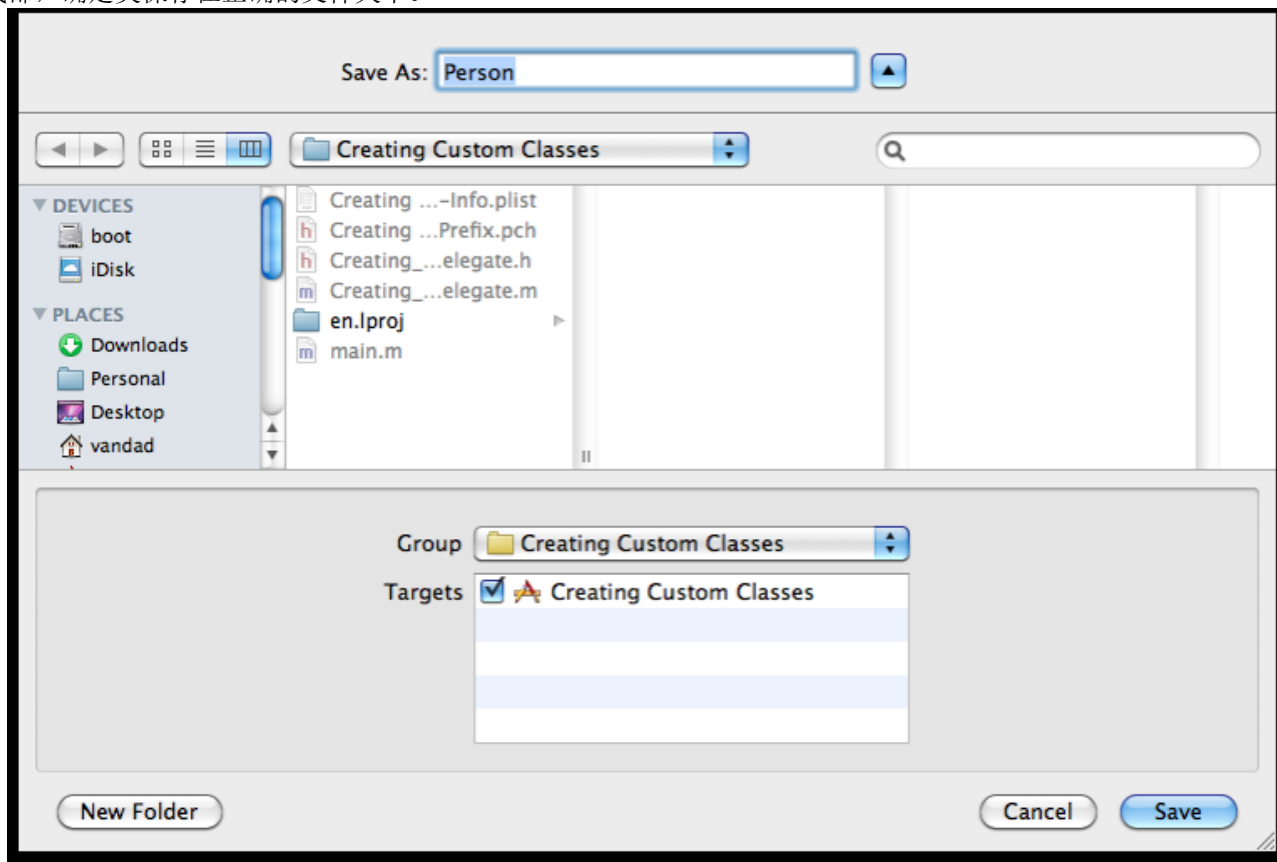


图 1-25。在 Xcode 里创建一个名为 Person 的类

现在这两个文件会自动添加到你的工程里。它们分别为 Person.h 和 Person.m 的文件。前者的是头文件，后者是执行文件。

以下是 Person.h 文件中的内容：

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
@end
```

下面是 Person.m 文件中的内容：

```
#import "Person.h"
@implementation Person
- (id)init
{
    self = [super init];
    if (self) {
        // Initialization code here.
    }
    return self;
}
@end
```

我们可以看到，Xcode 已经在这些文件预设一些内容。虽然我们还不了解这些内容，但这只是我们代码的开始部分。这里有一个名为 Person 的类。我们从哪里知道这个名字的呢？这并不是这个文件本身里面的，而是 Xcode 从图 1-25 中获取的文件名并用它表示类名。如果你再看看 Person.h 中的内容，你会发现有这一段代码：

```
@interface Person: NSObject
```

简而言之，在@interface 关键字后面就是类的名称。如果你不喜欢这个名称，只要右键单击它，然后选择“重构”，然后重命名即可。你可以通过向导来重命名你的类。

1.11.4. 参考

XXX

1. 12. 定义类的功能

1.12.1. 问题

为类定义一些功能并允许它们在以后可以被重用。

1.12.2. 方案

为了创建可重用的代码块，你可以创建类的实例或方法，或者干脆在你的程序中调用该方法。

1.12.3. 讨论

几乎每一个编程语言都会创建程序和函数去封装特定的功能，尤其是这些程序员经常重复使用到的功能。有些语言认为“程序”和“函数”是同一回事，但是有些语言会把它们区分开来。一个程序就是一个带有可选参数的代码块，它没有返回值。在 Objective-C 里，一个程序返回 void 来表示它不返回一个类似函数的值，但它实际上会返回值。这里是用 C 语言写的一个简单的程序（无主函数）：

```
Void sendEailTo(const char *paramTo,
                Const char *paramSubject,
                Const char *paramEmailMessage){
    /*send the email here...*/
}
```

这段程序是以 sendEmailTo 命名的一个函数并带有三个参数，参数分别为：ParamTo，paramSubJect 和 ParamEmailMessage。然后我们可以按下面方法来调用这段程序：

```
sendEmailTo("somebody@somewhere.com","My Subject","Please read my email");
```

把这段程序变成返回一个布尔值的函数，如下面一段代码：

```
Bool sendEmailTo(const char *parmTo,
Const char *paramSubject,
Const char *parmEmailMessage){
/* send the email here ... */
if (paramTo == nil ||
paramSubject == nil ||
paramEmailMessage == nil){
```

```
/* One or some of the parameters are nil */
NSLog(@"Nil parameter(s) is/are provided.");
return NO;
}
return YES;
}
```

调用这段函数和调用 `sendEmailTo` 这段函数相似（除了它含有一个函数），我们可以获取返回值，例如：

```
BOOL isSuccessful = sendEmailTo("somebody@somewhere.com",
"My Subject",
"Please read my email");
if (isSuccessful){
/* Successfully sent the email */
} else {
/* Failed to send the email. Perhaps we should display
an error message to the user */
}
```

在 Objective-C 中，任何方法都是为一个类而创建的。创建 Objective-C 的方法与其他编程语言（如 C 语言）编写程序和函数不一样。方法分为两类：实例或类。实例方法可以被称为一个类的实例（即，对每个基于类而创建的对象），而被称为类方法得到的类本身不需要程序员再为类而创建实例。要在 Objective-C 里创建一个方法，可以按以下几步在 .m 文件中创建目标类：

1. 类型（如果你想要一个实例方法或一个类的方法。）
2. 选择你的方法的返回类型，并括在括号内，例如：（void）没有返回值，（Bool）一个布尔值，（NSObject*）返回 NSObject 的一个实例，等等。
3. 选择一个方法的名称，以小写字母开头。在 Objective-C 中，方法名常以小写字母开头，例如：`sendEmailTo` 代替 `SendEmailTo`。
4. 如果你不想你的方法含有何参数，直接跳到步骤 9。
5. 给参数取两个名称。一个名称作为方法名称的一部分，将会被外部的方法所用（除了第一个参数外，其他所有参数都是可选的）。另一个名称将被作为方法内的一个参数名。但有一个例外，就是方法的第一个参数名是在第三步选择的方法的一部分。对于这第一个参数，你只能选择第二个名称作为方法内部的方法名。
6. 给参数取好名字后，给方法选择数据类型并括在括号内。
7. 在你的参数的第一个所选名称后加上冒号（如果有），并加上括号，括号里依次为方法的数据类型和参数的第二个名称。
8. 为其他参数重复第 5 至 7 步骤。
9. 在方法名和参数后插入一个开放的大括号（{）（假如有参数的话），并以一个右大括号（}）结束。

回到我们前面看到的 `sendEmailTo` 程序的例子，试着在 Objective-C 里创建一个相同的方法：

```
- (BOOL) sendEmailTo:(NSString *)paramTo
withSubject:(NSString *)paramSubject
andEmailMessage:(NSString *)paramEmailMessage{
/* Send the email and return an appropriate value */
if ([paramTo length] == 0 ||
[paramSubject length] == 0 ||
[paramEmailMessage length] == 0){
/* One or some of the parameters are empty */
NSLog(@"Empty parameter(s) is/are provided.");
return NO;
}
return YES;
}
```

这是一个返回一个布尔值（Bool）实例方法，这个方法名为 `sendEmailTo: WithSubject: andEmailMessage:`。并且它有三个参数。然后我们可以用下面的代码来调用这个方法：

```
[self sendEmailTo:@"someone@somewhere.com"  
withSubject:@"My Subject"  
andEmailMessage:@"Please read my email."];
```

如前所述，每一个参数的第一个名字（除了第一个）是可选的。换句话说，我们可以使用不同的名称来构造 `sendEmailTo:WithSubject:andEmailMessage:` 方法：

```
- (BOOL) sendEmailTo:(NSString *)paramTo  
:(NSString *)paramSubject  
:(NSString *)paramEmailMessage{  
/* Send the email and return an appropriate value */  
if (paramTo length] == 0 ||  
[paramSubject length] == 0 ||  
[paramEmailMessage length] == 0){  
NSLog(@"Empty parameter(s) is/are provided.");  
return NO;  
}  
return YES;  
}
```

强烈不建议写没有额外名称参数的方法。这是一个很糟糕的编程习惯，它会让你混淆，并且和你一起工作的团队会忽视你记录好的代码。我们可以这样调用这个方法：

```
[self sendEmailTo:@"someone@somewhere.com"  
:@"My Subject"  
:@"Please read my email."];
```

第一次执行很容易理解，因为你可以看到在调用里的每一个参数的名称。声明和实现一个类的方法类似声明和实现一个实例方法。当声明和实现一个类的方法时你需要注意以下几点：

- 1、一个类的方法类型标示符是+而不是实例方法类型标示符-。
- 2、可以自由访问类里的方法，但是类的方法自身只能被内部的类方法所执行。
- 3、为类提供新的方法实例时，类的方法是很有用的。例如，一个类名为 `allocAndInit` 方法既可以分配并初始化一个对象，也可以返回这个对象到其调用者。

1.12.4. 参考

XXX

1. 13. 定义具有相同名称的两个或多个方法

1. 13. 1. 问题

在一个对象里实现两个或两个以上具有相同名称的方法。在面向对象编程里，这被称为方法重载。然而，在 Objective-C 中，方法重载与其它编程语言（如 C++）中的方法重载不一样。

在此方案中讨论使用和解释到得技术只是创建有不同数量的参数或参数有不同的名字的方法，这只是给你一个怎样实现第一个名字段相同的方法的想法而已。

1. 13. 2. 方案

给方法使用相同的名称，但是要保证每个方法的参数的数量或名称一样，例如：

```
- (void) drawRectangle{
[self drawRectangleInRect:CGRectMake(0.0f, 0.0f, 4.0f, 4.0f)];
}
- (void) drawRectangleInRect:(CGRect)paramInRect{
[self drawRectangleInRect:paramInRect
withColor:[UIColor blueColor]];
}
- (void) drawRectangleInRect:(CGRect)paramInRect
withColor:(UIColor*)paramColor{
[self drawRectangleInRect:paramInRect
withColor:paramColor
andFilled:YES];
}
- (void) drawRectangleInRect:(CGRect)paramInRect
withColor:(UIColor*)paramColor
andFilled:(BOOL)paramFilled{
/* Draw the rectangle here */
}
```

这个例子是一个典型的重载模型。每个矩形都可以用填充（纯色）或空白（只显示它的边界）的方式来画出。

第一段程序是一个“易用程序”，它可以不用考虑如何去填充做这个矩形。在这段代码执行中，我们只调用第二个过程。

1. 13. 3. 讨论

方法重载是一种编程语言的功能，Objective-C、C++、Java 和其他一些语言都支持方法重载。使用此功能，程序员可以在同一个对象上创建具有相同名称的不同的方法。然而，在 Objective-C 中的方法重载与在 C++中方法重载有所区别。例如，在 C++中，重载一个方法，程序员需要保证同一种方法具有不同的参数数量或者至少有一个参数具有不一样的数据类型。

但是，在 Objective-C 中，你只需要改变至少有一个参数的名称。而更改参数的类型将无法工作，例如：

```
- (void) method1:(NSInteger)param1{
/* We have one parameter only */
}
- (void) method1:(NSString *)param1{
/* This will not compile as we already have a
method called [method1] with one parameter */
}
Changing the return value of these methods will not work either:
- (int) method1:(NSInteger)param1{
/* We have one parameter only */
return param1;
}
- (NSString *) method1:(NSString *)param1{
/* This will not compile as we already have a
method called [method1] with one parameter */
return param1;
}
```


结果表明，必须改变每个方法接受的参数的数量或至少一个参数的名称。这有一个已经改变过参数数量的范例：

```
- (NSInteger) method1:(NSInteger)param1 {
return param1;
}
- (NSString*) method1:(NSString *)param1
andParam2:(NSString *)param2{
NSString *result = param1;
if ([param1 length] > 0 &&
[param2 length] > 0){
result = [result stringByAppendingString:param2];
}
return result;
}
Here is an example of changing the name of a parameter:
- (void) drawCircleWithCenter:(CGPoint)paramCenter
radius:(CGFloat)paramRadius{
/* Draw the circle here */
}
- (void) drawCircleWithCenter:(CGPoint)paramCenter
Radius:(CGFloat)paramRadius{
/* Draw the circle here */
}
```

可以发现这两种方法的声明之间的区别吗？第一种方法的第二个参数被称为 radius（用小写 r 表示），而第二个方法的第二个参数被称为 Radius（用大写 R 表示）。这将会使这两个方法分开并允许工程得到编译。但是，苹果会引导构造方法该怎么做。更多信息，请参考苹果文档《Coding Guidelines for Cocoa》，有效连接为：<http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html>

下面是方法的构造和工作时要注意的一些简要事项：

- 使方法的名能清楚的描述出方法的作用，不要使用太多的术语和缩写。在代码向导里已经提供了一系列的可使用的缩写。
- 使每一个参数名能把它的参数和目的描述清楚，在一个有三个参数的方法里，如果这个方法被编写成执行两个独立的动作，那么可以使用最后一个参数开始的那个单词作为参数名。无论如何，都要避免使用开始的那个参数名。
- 方法名要以小写字母开始。
- 对于委托方法，方法名以这个委托方法所属的类为开始来命名。

1.13.4. 参考

章节 1.12

1. 14. 分配和初始化对象

1. 14. 1. 问题

为新的对象创建一个实例，而不清楚分配和初始化的区别，为什么要先分配和初始化一个对象才能使用它。

1. 14. 2. 方案

必须先分配和初始化一个对象才能使用它。可以使用 `alloc` 这个实例方法来分配一个对象。该类方法会为对象、对象的实例和方法分配内存空间。但是，被分配的内存并未定义。所以，每个对象必须初始化，就是给它赋上初值。

一个初始化方法必须是特定的初始化，通常是具有很多参数的初始化方法。例如：`initWithFrame`：这个方法是 `UIView` 类的对象的一个特定初始化方法。所以，在使用对象之前先分配和初始化它。

当执行一个新的对象时，不要重载 `alloc` 这个方法。这个方法是在声明在 `NSObject` 里的。相反的，而是为特定对象处理所需参数重载 `init` 方法和创建自定义方法。

1. 14. 3. 讨论

继承自 `NSObject` 的对象必须经过以下两步才能使用：

1. 分配
2. 初始化

这个实例是由于继承了 `NSObject` 类，然后实现了 `alloc` 这个父类方法然后才完成内存分配工作的。这个方法创建了这个新对象的内部构造器并使全部实例变量设为初值 0。完成这步后，`init` 方法负责给变量赋默认值并进行其他相关工作，比如实例化其他内部对象。

让我们看一个范例。创建一个 `MyObject` 类。这是一个 .h 文件：

```
#import <Foundation/Foundation.h>
@interface MyObject : NSObject
- (void) doSomething;
@end
```

下面是这个类的执行文件 (.m 文件)：

```
#import "MyObject.h"
@implementation MyObject
- (void) doSomething{
/* Perform a task here */
NSLog(@"%s", __FUNCTION__);
}
@end
```

`MyObject` 对象的 `doSomething` 实例方法会将当前函数名打印显示在控制窗口上。现在继续看下面的实例化 `MyObject` 类的一个对象的方法：

```
MyObject *someObject = [[MyObject alloc] init];
/* Do something with the object, call some methods, etc. */
[someObject doSomething];
```


这段代码是正确的。现在跳到初始化对象：

```
MyObject *someObject = [MyObject alloc];  
/* Do something with the object, call some methods, etc. */  
[someObject doSomething];
```

假如现在运行这段代码，会发现它也会正常运行。这是怎么回事呢？一般认为使用对象之前必须先初始化它。也许苹果的解释会更加清楚：

一个对象在没有初始化之前是不能被使用的。定义在类 `NSObject` 中的这个方法没有初始化；它仅仅返回它自身。

结论很简单，这说明 `init` 这个方法是一些在它们使用前需要执行的类的一个占位符，例如设定额外的数据结构或打开文件。`NSObject` 自带了许多要用到的类，所以不用特别地去初始化其他类。然而，如果其父类已经重载了一个自定义初始化的方法，在调用对象之后再运行 `init` 这个方法将是一个很好的编程习惯。请记住一个对象的初始化方法的返回值是 `id` 类型，设定初值的方法也许会返回一个与 `alloc` 方法返回不同的对象，这称为两阶段构造的技术，这是极其方便的。但是，讨论这个技术已经超出本书的内容。关于更多二阶段构造的技术请参考由 Erik M. Buck 和 Donald A. Yacktman (艾迪生 — 韦斯利专业) 编著的《Cocoa Design Patterns》。

1.14.4. 参考

XXX

1.15. 添加类的属性

1.15.1. 问题

需要利用 `XXX.AA` 的方式来直接访问类中的变量，而不是采用调用方法的方式来访问。

1.15.2. 方案

使用关键字 `@property` 给类定义属性。

1.15.3. 讨论

任何一个通过点表示的都是一个属性。属性是方法的快捷方式。什么意思呢？先看一个范例：

```
NSObject *myObject = [[NSObject alloc] init];  
myObject.accessibilityHint = @"Some string";
```

可以看到我们分配和初始化了一个 `NSObject` 类型的对象，并在这个对象使用点的方式去访问一个叫 `accessibilityHint` 的属性。然而 `accessibilityHint` 是从哪里来的呢？

这很简单。一个属性是用 `@property` 关键字定义的。实际上，假如你在 Xcode 里的键盘上按下命令键，并点击刚才看到的范例中的 `accessibilityHint` 属性，将会跳转到 `NSObject.h` 文件里的下面这段代码上：

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

到底属性是什么呢？当定义一个属性时，会告诉编译器我们将会给这个属性写上一个 setter 和 getter 方

法。如果在这个属性里设置一个值，运行时将执行setter方法。如果读取属性，则执行的 getter 方法。

为了加深理解。在1.11节，我们知道了如何创建类。我们创建了一个叫Person的类。在1.12节，我们学会了如何给类添加方法。可以结合这两个知识去学习更多的关于属性的内容。现在打开person.h文件并定义一个名为firstName的属性：

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
@property (nonatomic, strong) NSString *firstName;
@end
```

现在同时按下Shift+R键编译工程。注意，LLVM编译器会显示两个警告：

```
warning: property 'firstName' requires method 'firstName' to be defined - use @synthesize, @dynamic or provide warning: property 'firstName' requires method 'setFirstName:' to be defined - use @synthesize, @dynamic
```

注意：在1.16节你将会学到全新的关于自动引用数值的关键字，例如strong。

很明显，属性虽然创建了，但是当读取属性值或给属性设定值时，编译器并不知道怎么做。因此，必须写一个setter和一个getter方法。编译器会明确告诉我们setter方法该叫做setFirstName:，getter方法叫做firstName:。幸运的是，我们不必手动编写这两个方法的属性。我们可以在.m文件中使用@synthesize关键字让编译器自动为属性生成setter和getter方法：

```
#import "Person.h"
@implementation Person
@synthesize firstName;
- (id)init
{
    self = [super init];
    if (self) {
        // Initialization code here.
    }
    return self;
}
@end
```

接下来我们可以使用Person类了。这里有一个范例：

```
#import "SomeOtherClass.h"
#import "Person.h"
@implementation SomeOtherClass
- (void) makeNewPerson{
    Person *newPerson = [[Person alloc] init];
    newPerson.firstName = @"Andrew";
    NSLog(@"First name = %@", newPerson.firstName);
    NSLog(@"First name = %@", [newPerson firstName]);
}
@end
```

这段范例代码打印了两次newPerson的第一个名字，首先使用属性的第一个名字然后在对象上调用firstName的getter方法。两者都会指向在Person.m文件中使用@synthesize创建的方法。

注意：

在旧版本的Objective-C的运行库中，为了让@property正常工作，必须先定义一个实例变量。这个实例变量的内存管理是由程序员自己负责的。实例变量会对定义它们之外的类开放（即它们不会对任何仅仅是输入实

例变量的类公开)。实例变量一般被专业的Objective-C开发者称为ivars。Ivars的发音有点像I-VAR, VAE的发音有点像WAR, 后跟 a V音。

在新的运行库中, 我们不必再定义变量了。只要定义属性, LLVM会为我们定义变量。假如你使用的是GCC编译器, 那就完全不是一回事了, 你会发现LLVM处理变量的方式有很大的不同。例如, 在GCC4.2中, 变量对任何子类都无效, 而当你使用LLVM编译器, 子类将会使用超类的变量。所以你要确保你使用的是苹果的最新版本的LLVM编译器。

你可以根据需要自定义setter和getter方法。甚至你可以使用@synthesize让编译器自动生成setter 和getter属性方法, 并重载这个方法。例如, 改变Person1类setFirstName:方法的第一个属性名的setter方法, 例如:

```
#import "Person.h"
@implementation Person
@synthesize firstName;
- (void) setFirstName:(NSString *)paramFirstName{
    firstName = [paramFirstName stringByAppendingString:@" Jr"];
}
- (id)init
{
    self = [super init];
    if (self) {
        // Initialization code here.
    }
    return self;
}
@end
```

为给firstName 属性的字符串添加了一个“Jr”后缀重载了firstName 属性的setter方法。所以当setter和getter被调用时, 在这段代码之后:

```
Person *newPerson = [[Person alloc] init];
newPerson.firstName = @"Andrew";
NSLog(@"First name = %@", newPerson.firstName);
NSLog(@"First name = %@", [newPerson firstName]);
```

接下来控制窗口将会显示:

```
First name = Andrew Jr
First name = Andrew Jr
```

假如要定义一个只读属性, 你所需要做的就是用readonly关键字定义它, 例如:

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

注意:

对于一个只读属性, 改变它的值只有一个办法, 即在该类中定义属性成员的变量。

1.15.4. 参考

XXX

1.16. 将手动引用计数修改为自动引用计数

1.16.1. 问题

你希望学习苹果提供的新的编译器方案: 自动引用计数, 以解决另程序员头痛的 Objective-C 的对象与内存管理。

1.16.2. 方案

学习 LLVM 编译器提供的新存储属性:strong,weak 及 unsafe_unretained。

1.16.3. 讨论

在最新的 LLVM 编译器中使用自动引用计数(ARC), 我们必须处理存储属性是 strong,weak,unsafe 与 unretained。ARC 管理下的任何对象都会有一个存储属性。请参考下面的说明:

strong

声明为 Strong 类型的对象会在执行时期自动保留且在生命周期结束前都是有效的, 并在结束后会自动释放。这个关键字就像是 Objective-C 传统内存管理方法中的 retain。

weak

这是弱引用。若变量声明了这个关键字, 当对象的变量指针被释放时, 将会设为 nil。举例来说, 假如你有两个字符串属性, 分别设为 strong 引用和 weak 引用, 当 strong 属性被释放时,weak 属性将会被设为 nil。

unsafe_unretained

这属性表示只做简单的变量赋值。意思是在变量赋值时, 将不会做 retain 保留对象。

默认的情况下, 局部变量都是 strong 变量。相较之下, 属性必须明确指定存储属性, 因为编译器无法保证所有属性都默认为 strong 属性。看看下面关于 strong 存储属性的代码, 我们先声明两个属性叫 string1 与 string2:

```
#import <UIKit/UIKit.h>
@interface Moving_from_Manual_Reference_Counting_to_ARCAppDelegate
: UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) NSString *string1;
@property (nonatomic, strong) NSString *string2;
@end
```

现在我们初始化 string1 属性并赋值:String 1, 然后对 string2 属性赋值为 string1。我们可在此看到 strong 属性的功用, 即使 string1 属性被释放了, string2 的属性的值依然存在:

```
#import "Moving_from_Manual_Reference_Counting_to_ARCAppDelegate.h"
@implementation Moving_from_Manual_Reference_Counting_to_ARCAppDelegate
@synthesize window = _window;
@synthesize string1;
@synthesize string2;
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
self.string1 = @"String 1";
self.string2 = self.string1;
self.string1 = nil;
NSLog(@"String 2 = %@", self.string2);
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

这段程序的输出如下:

```
String 2 = String 1
```

声明属性时, 最常使用 strong,weak 与 unsafe_unretained。

你甚至可以利用这些存储属性来声明局部变量, 但必须改变指定的位。

strong 的内联相当于 __strong, weak 相当于 __weak, unsafe_unretained 相当于 __unsafe_unretained。

(在每个关键字前面加上两个底线字符)

参考下面范例:

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
```

/*

所有局部变量皆默认为 strong，在代码中指定__strong 只是为了更清楚表达，并没有任何影响。

```

*/
__strong NSString *yourString = @"Your String";
__weak NSString *myString = yourString;
yourString = nil;
__unsafe_unretained NSString *theirString = myString;
/* 此时，所有指针都会被设为 nil。 */
#endif;
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}

```

ARC 中的弱引用神奇的解决了很多程序员面临到的对象释放问题。

弱引用神奇的地方在于将弱引用的对象指针设为空(nil)时，对象将会被释放。被释放的对象会逐一将所有对象中使用弱引用的对象设为空(nil)。

unsafe_unretained 这个存储指定如其名，是真正不安全的。这是因为这种指针变量被释放后，并不会设为 nil，而是指向一块悬空的内存，因此当存取这指针可能会导致程序崩溃。一般而言，应该将变量指定为 weak 或__weak。

来看看下面的关于弱引用的范例，将 string2 属性的储存将 strong 改为 weak。

```

#import <UIKit/UIKit.h>
@interface Moving_from_Manual_Reference_Counting_to_ARCAppDelegate
: UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) NSString *string1;
@property (nonatomic, weak) NSString *string2;
@end

```

当我们的应用第一次执行时，将会初始化 string1 属性，并且赋值给 string2。我们将会对 string1 属性赋与空值(nil)，然后打印出 string2 的值，在这边 string2 也许会打印出不正确的结果，而不是空值。所以你必须确定在你应用执行期间，需去除所有无效对象。

为了实践这一目标，我们将会应用在后台执行时印出 string2 的值。(通常是用户将另一个应用弄到前台的时候。)一旦在后台执行，我们可以准确的知道内存中无效对象跟结果：

```

/* 3 */
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
self.string1 = [[NSString alloc] initWithUTF8String:"String 1"];
self.string2 = self.string1;
self.string1 = nil;
/* 此时，所有指针都为 nil */
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
}
- (void)applicationDidEnterBackground:(UIApplication *)application{
NSLog(@"String 2 = %@", self.string2);
}

```

现在执行应用，等一秒后，按下模拟器装置上的 Home 按键。

你将会在控制台视窗发现下面的结果：

```
String 2 = (null)
```

要证明 ARC 下的弱引用是否完美执行是很容易的。

现在要确认指定为 unsafe_unretained 会有多危险，继续修改 string2 属性的储存属性为

unsafe_unretained，并重复执行一次刚刚的步骤：

```

#import <UIKit/UIKit.h>
@interface Moving_from_Manual_Reference_Counting_to_ARCAppDelegate

```

```
: UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) NSString *string1;
@property (nonatomic, unsafe_unretained) NSString *string2;
@end
```

在前面的范例中，我们已经实践离开应用的委托(当应用到后台执行时打印出 `string2` 属性的值)，你可以重复同样的程序，开启你的应用并送到后台执行，这时程序将会崩溃！

这表示当程序到后台执行时，所打印的 `string` 属性是被指向一块无效的内存。

自从 `string2` 属性被设定为不安全与不保存(`unsafe_unretained`)，它就不知道它所指到的对象指针(`string1`)已经被释放，当 `string1` 被设为空值时。

除了上述三个存储规范之外，我们也可以使用 `__autoreleasing` 规范。

这种存储规范是最方便的，当我们想要传递一个引用对象给某个方法。

举例来说，假设你有个方法需要传递一个 `NSError` 型别，调用的方法将会接受一个未初始化且未配置的 `NSError` 实例。这表示呼叫者没有真正配置这错误，所以我们的方法应该负责配置。

要做到这点，你必须指定错误参数为可自动释放，当需要释放的时候：

```
- (void) generateErrorInVariable:(__autoreleasing NSError **)paramError{
    NSArray *objects = [[NSArray alloc] initWithObjects:@"A simple error", nil];
    NSArray *keys =
    [[NSArray alloc] initWithObjects:NSLocalizedStringKey, nil];
    NSDictionary *errorDictionary = [[NSDictionary alloc] initWithObjects:objects
    forKey:keys];
    *paramError = [[NSError alloc] initWithDomain:@"MyApp"
    code:1
    userInfo:errorDictionary];
}

- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSError *error = nil;
    [self generateErrorInVariable:&error];
    NSLog(@"Error = %@", error);
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

在这范例中，`application:didFinishLaunchingWithOptions:` 方法没有配置 `NSError` 的实例；由 `generateErrorInVariable` 方法处理。但编译器理解这个错误对象的范围，`generateErrorInVariable` 方法提醒编译器这个错误参数对象将会被自动释放，当它不再被需要时。

1.16.4. 参考

XXX

1.17. 自动引用计数的类型转换

1.17.1. 问题

你希望知道如何使用自动引用计数提供的新型别转换

1.17.2. 方案

使用 `__bridge`, `__bridge_transfer` 跟 `__bridge_retained` 转换

1.17.3. 讨论

类型转换是一个将类型 A 转换为类型 B 的过程。

举例来说，有一个 Code Foundation 字符串对象 `CFStringRef`，并且你想将它放入一个 Objective-C 的 `NSString` 字符串类型，这时你很容易犯错：

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    CFStringRef coreFoundationString =
    CFStringCreateWithCString(CFAllocatorGetDefault(),
    "C String",
    kCFStringEncodingUTF8);
    /* 编译期错误!!! */
    NSString *objCString = coreFoundationString;
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

在这边我们对类型 `NSString` 变量 `objCString` 赋与一个 Core Foundation 字符串变量，在这边编译器会觉得困惑，因为它不知道如果把内存分配给每一个对象。此外我们也会得到一些内存泄漏，因为 ARC 不支持 Code Foundation 对象。要记住 ARC 是无法支持 Core Foundation 对象的。如果想要这样做，就必须知道类型转换：

`__bridge`

简单的类型转换的等号右侧的对象赋予左值。这不会影响任何对象上的保留计数，不论是等号左边或右边的变量。

`__bridge_transfer`

这种类型转换会在赋值后将等号右边的变量释放。

所以假如跟前面说得一样有一个 Core Foundation 字符串被创建，并且要放入局部 `NSString` 变量(局部变量总是 strong 存储属性，参考章节 1.16)你应该用这个类型转换，后面将会举个例子。

`__bridge_retained`

这个转换方式类似 `__bridge_transfer`，唯一的差别就是会保留等号右边的变量。

接下来试着修改前面的代码。目标是将 Core Foundation 字符串变量放入 `NSString` 字符串变量。然后自动释放 Core Foundation 字符串变量。

为了达到这点，必须使用 `__bridge_transfer` 转换。

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    CFStringRef coreFoundationString =
    CFStringCreateWithCString(CFAllocatorGetDefault(),
    "C String",
    kCFStringEncodingUTF8);
    /* 编译期错误!!! */
    NSString *objCString = (__bridge_transfer NSString *)coreFoundationString;
    NSLog(@"String = %@", objCString);
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

当一个 CoreFoundation 类被创建时会发生什么？这时后保留值会是 1。

然后使用 `__bridge_transfer` 转换并赋值到局部 `NSString` 变量。

但这时候因为编译器发现了转换运算，因此它将先保留 Core Foundation 字符串并存入 `NSString` 变量，然后再赋值后释放 Core Foundation 字符串变量。

完美，这样就达成目标了！

现在来试试 `__bridge_retained`，这个转换方式会保留等号右边的变量。

参考下面范例：

```
- (BOOL) application:(UIApplication *)application
```

```
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    CFStringRef coreFoundationString =
    CFStringCreateWithCString(CFAllocatorGetDefault(),
    "C String",
    kCFStringEncodingUTF8);
    id unknownObjectType = (__bridge id)coreFoundationString;
    CFStringRef anotherString = (__bridge_retained CFStringRef)unknownObjectType;
    NSString *objCString = (__bridge_transfer NSString *)coreFoundationString;
    NSLog(@"String = %@", objCString);
    objCString = nil;
    CFRelease(anotherString);
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

这些代码会发生下列状况：

1. 配置一个 Core Foundation 字符串符，并存入 coreFoundationString 局部变量。

虽然一个 core Foundation 类，ARC 并不会为其加上存储属性，所以我们必须自行管理内存。

当变量被创建时，保留值会是 1。

2. 然后将 Core Foundation 字符串符转换为通用类型 id。注意，在这里只是做单纯的转换动作，这边并没有保存或是放这个对象，所以 unknownObjectType 跟 coreFoundationString 两个变量的保留值都是 1。3. 现在我们保留通用类型 id 的变凉，并且存入另一个 Core Foundation 字符串符。这时 coreFoundationString, unknownObjectType 与 anotherString 变量的保留值都是 2，而且存在同一块局部内存。

4. 当我们将 coreFoundationString 变量透过 __bridge_transfer 转换并存入一个 NSString 局部变量中会发生什么？可以确定的是在赋值后 coreFoundationString 会被释放(保留值会从 2 变 1)，并且再次被保留(因为局部变量存储属性为 strong，保留值会由 1 变 2)。现在 coreFoundationString, unknownObjectType, anotherString 与 objCString 变量的保留值都是 2。

5. 下一站，将 objCString 变量设为空值(nil)，这样做会释放这个变量并且保留值会回到 1。

所有其他局部变量依然是有效的，因为它们的保留值还是维持在 1。

6. 接下来明确指定释放 anotherString 变量。

这动作会将对象的保留值由 1 变 0，并且字符串符对象会被释放。

这时候就不应该再使用这里的任何变量，因为它们现在都指向一个被释放的对象，除了 objCString 之外，其他都应该手动设为空值(nil)。

1.17.4. 参考

XXX

1. 18. 使用协议委托任务

1.18.1. 问题

你希望确保类通过衍生其他类从而获得某些方法或属性。

1.18.2. 方案

使用协议。

1.18.3. 讨论

协议是一个声明某些方法及属性并储存在实体文档。(通常延伸档名是.h)

任何实践协议的对象，都必须实践协议提供的方法及属性(可在协议中指定是必须或可选)。

协议就像是一些规范，实践协议的类必须遵守这些规范。让我们看看这个简单的范例，接下来将继续声明一个协议叫做 `PersonProtocol`。

为此，你必须创建一个新的协议档，参照下列步骤：

1. 在 Xcode 中，打开你的工程，到 File 选单并选 New->New File
2. 现在新增文档对话框左边，选择 iOS 主分类然后选择 Cocoa Touch 子分类然后选择 Objective-C Protocol 选项，并按下一步(参考图 1-26)。

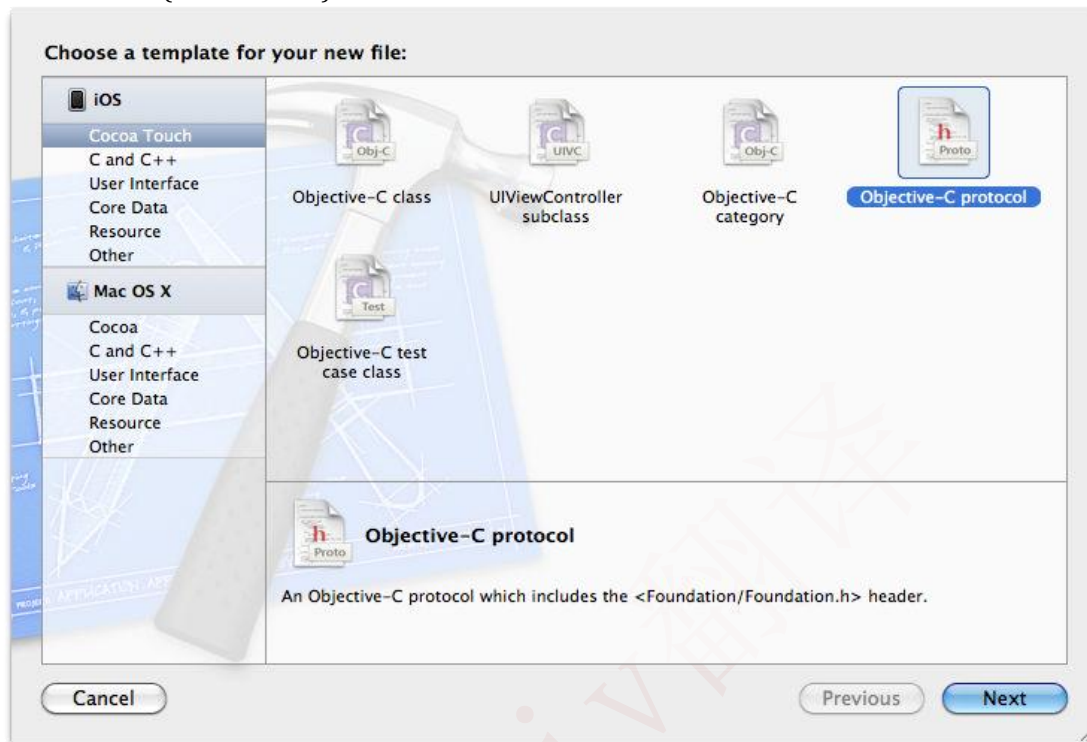


图 1-26. 创建一个新的协议

3. 现在你将被要求储存并指定一个文档名，命名为 `PersonProtocol` 并按下储存 (参考图 1-27)。

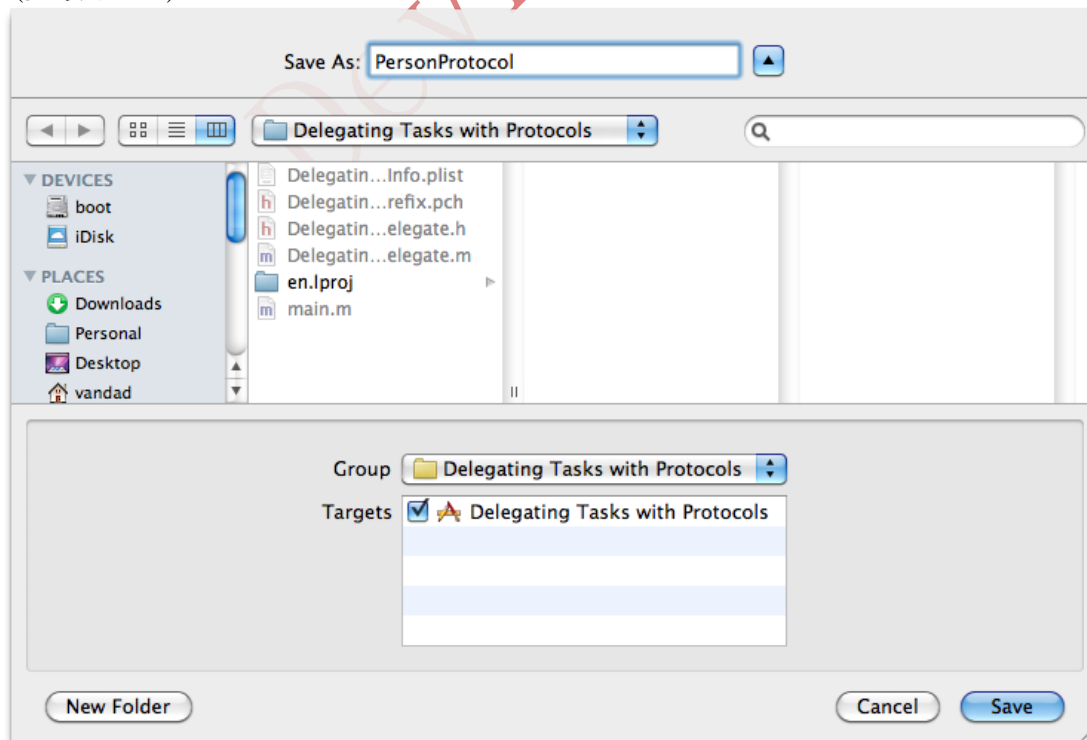


图 1-27. 储存新的协议

现在我们有了表头档。让我们实际声明协议吧。

我们希望实践这个 `PersonProtocol` 协议的任何类对象，是用来模拟人(Person)，换句话说就是个人的类。

例如，你可以把类命名为婴儿，母亲，父亲，儿子，女儿，陌生人等。同时可以让这些类实践 `PersonProtocol` 协议，让这些类实践协议的行为。比方说，每一个人至少要有一个名字，姓氏跟年龄：

```
#import <Foundation/Foundation.h>
@protocol PersonProtocol <NSObject>
@property (nonatomic, strong) NSString *firstName;
@property (nonatomic, strong) NSString *lastName;
@property (nonatomic, unsafe_unretained) NSUInteger age;
@end
```

现在建立一个类叫父亲(Father)，并确认这个类符合 `PersonProtocol` 协议。

参照下列步骤来建立一个类：

1. 在 Xcode 中，打开你的工程，到 File 选单并选 New->New File
2. 现在新增文档对话框左边，选择 iOS 主分类然后选择 Cocoa Touch 子分类然后选择 Objective-C class 选项，并按下一步(参考图 1-28)。

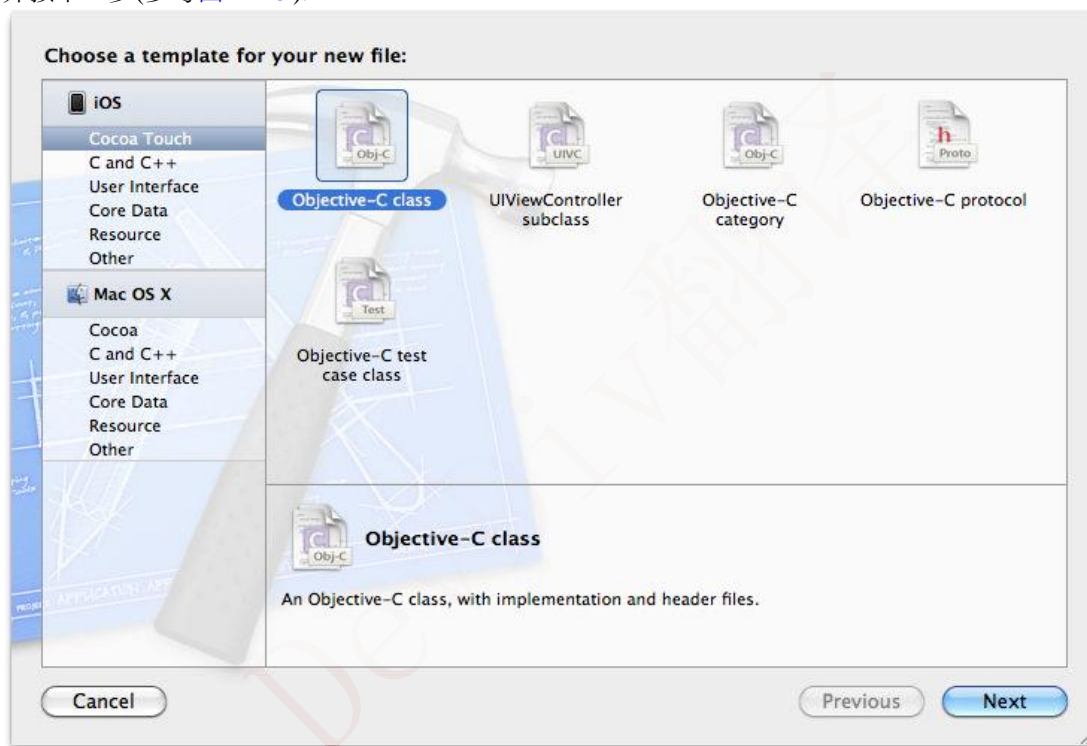


图 1-28. 创建父类

3. 在这个画面 (参考图 1-29)，确定我们创建的是 `NSObject` 的子类。一旦做到这点，按下下一步。

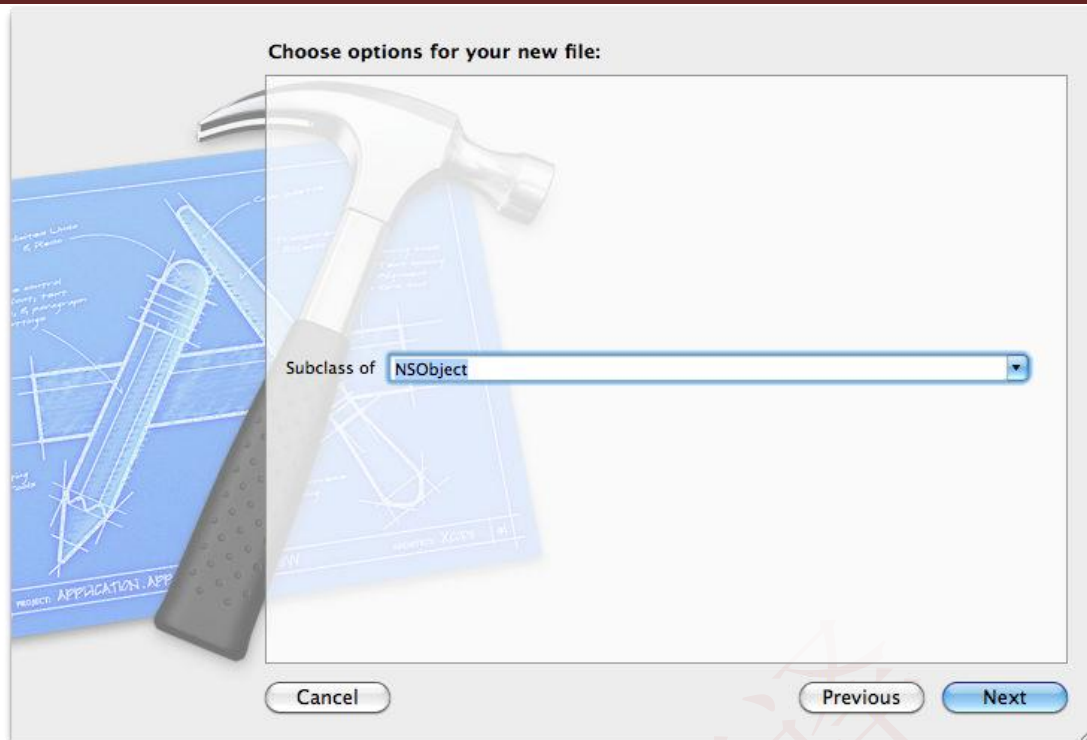


图 1-29. 继承 NSObject 来创建父类

4. 现在会要求你储存新的类。输入 **Father** 作为名称，并按下创建按钮(参考图 1-30)。

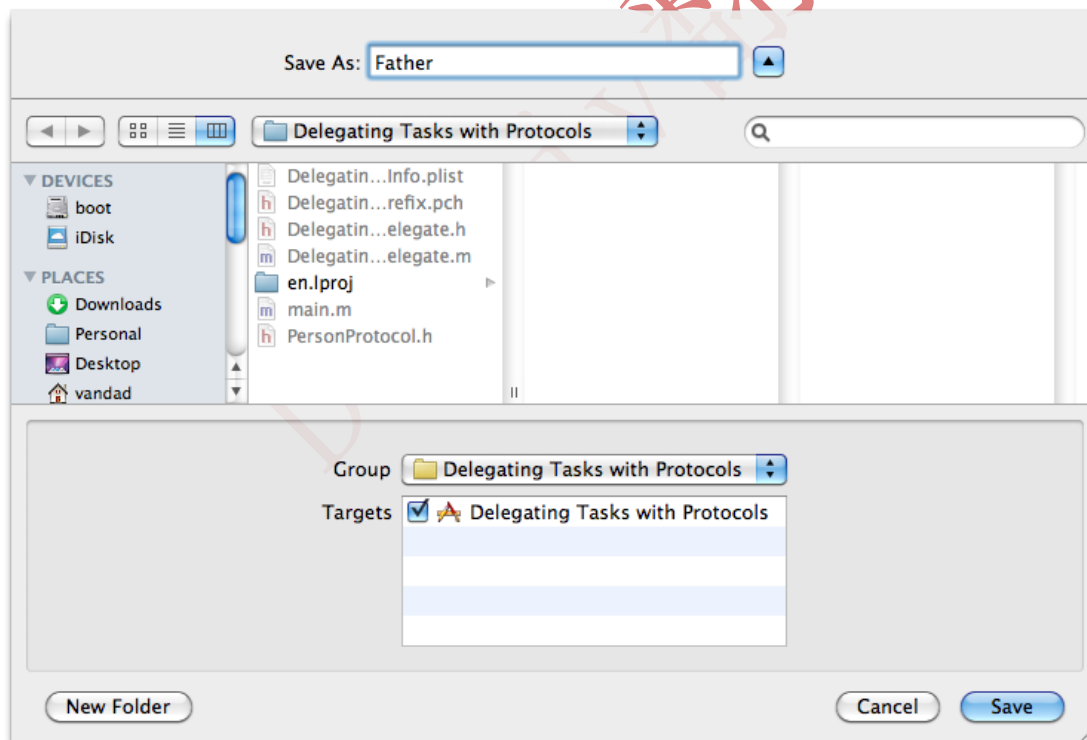


图 1-30. 储存父类

太棒了！现在我们有 **Father** 类跟 **PersonProtocol** 协议。开启 **Father** 类的表头档，并确认它符合 **PersonProtocol** 协议：

```
#import <Foundation/Foundation.h>
#import "PersonProtocol.h"
@interface Father : NSObject <PersonProtocol>
@end
```

假如现在你企图编译应用(按下快捷键 **Command+Shift+R**)，你将会看到如下图 1-31 显示的编译器错误。

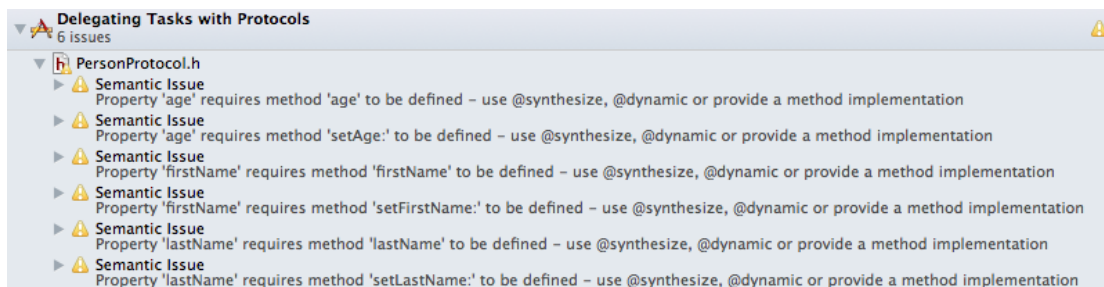


Figure 1-31. 跟协议有关的警告，应该会跟你的编译器讯息相同

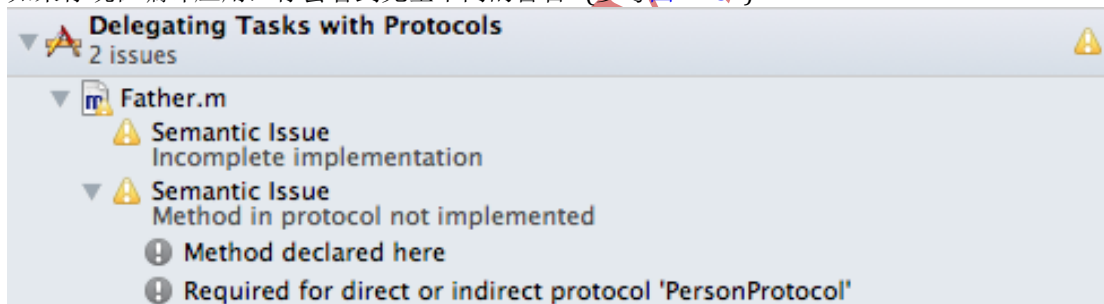
如你所见，编译器了解 `Father` 类想要符合 `PersonProtocol` 协议。然后 `Father` 类是不需要实践声明在 `PersonProtocol` 中属性的 getter 跟 setter 方法。

我们会看到这些警告是因为协议在默认的情况下，所有声明都是必须实现的。同时我们也可以用 `@required` 关键字明确指定必须的属性与方法。如果你希望协定的方法是可实践或不需实现的使用 `@optional` 关键字，就可以简单的设定。

让我们回到 `PersonProtocol.h` 并修改 `firstName`, `lastName` 与 `age` 属性为可选(`optional`)，也加入一个 `breathe`(呼吸)方法，设为必须的(`required`)，因为事实上，所有人都必须呼吸。

```
#import <Foundation/Foundation.h>
@protocol PersonProtocol <NSObject>
@optional
@property (nonatomic, strong) NSString *firstName;
@property (nonatomic, strong) NSString *lastName;
@property (nonatomic, unsafe_unretained) NSUInteger age;
@required
- (void) breathe;
@end
```

如果你现在编译应用，你会看到完全不同的警告 (参考图 1-32)。

图 1-32. `Father` 类没有实践声明在 `PersonProtocol` 中的 `breathe` 方法

假如你现在到 `Father` 中声明跟实践 `breathe` 方法，就算是空的方法，仍然可以编译通过。

记住，`Father` 类没有实践上面三个属性，因为它们是在 `PersonProtocol` 协议中是声明为可选的。下面是正确声明的 `Father` 类：

```
#import <Foundation/Foundation.h>
#import "PersonProtocol.h"
@interface Father : NSObject <PersonProtocol>
- (void) breathe;
@end
```

下面是正确实践的 `Father` 类：

```
#import "Father.h"
@implementation Father
- (void) breathe{
/* Implement this method here */
}
@end
```

试着去编译你的应用，你会发现已经可以顺利编译我们的代码了。

Cocoa Touch 给了 Objective-C 的协议一个很好的意义。在 Cocoa Touch 中协议是完美的声明委托对象的手段。委托对象是指一个对象，要使用或调用另一个对象发生的情况。举例来说，当你车坏了，你会委托修车厂

帮你修车。(虽然有些人喜欢自己修车,但一些较严重的还是需要委托修车厂)。

所以在 Cocoa Touch 中,当类期望委托方法是一样时,就会使用协议,以确保委托正确。

例如,在第 3 章中我们将看到,UITableView 类的定义和实现的属性是必须符合的 UITableViewDelegate 协议。这个协议定义了一些想要实践表视图的委托对象要遵守的法则。协议要求对象实践 certain 方法或在某些情况下,根据委托对象的需要指定实践一些可选择的方法或属性。

当用户选择了表中的一列的时候,UITableView 可以调用 didSelectRowAtIndexPath: 方法以及 UITableViewDelegate 提供的已实践方法。可能呼叫的方法是正确也可能是非正确的代码,但至少编译器不会报错。

1.18.4. 参考

XXX

1.19. 检测实例或类方法是否有效

1.19.1. 问题

你开发的 SDK 是最新版的,但你希望支持执行旧版 iOS SDK 的装置。

1.19.2. 方案

使用 NSObject 的 instancesRespondToSelector: 类方法检测指定的 selector 是否存在类实例中。要确认一个类是否响应本身的类方法,需使用 respondsToSelector: 类方法。你可以使用同样方式检测,在一个实例的实例方法,透过 instancesRespondToSelector: NSObject 的类方法

1.19.3. 讨论

这边有两个关于 iOS SDK 的重要概念要记住:

Base SDK(基底 SDK)

这个 SDK 是用来编译应用程序。可能是最新最大的 SDK,且能存取所有新的 API。

Deployment SDK/Target(部署 SDK)

这边的 SDK 使指定你希望编译后并执行的装置 SDK 版本。

因为就事实上,编译应用是基于 Base SDK 与 Deployment SDK 两个 SDK 编译。

这取决于你正在使用的设定(装置或模拟器)。程序必须不断检查实例或类方法是否存在,否则程序会是脆弱的,因为部署的装置的 SDK 版本,不一定是最新的,也许方法会不存在。

举个例子。NSArray 是 iOS SDK 中的一个类。正如你将看到的步骤 1.23,你可以简单的配置及初始化这个类型并且使用它的方法。NSMutableArray 类型是一个可变数组(建构后仍然可改变),并提供了排序机制,让你可以排序数组中的元素。它提供了一些排序方法,但有一些只适用于新版 SDK,因此你必须检测哪些方法是在运行环境中适用,并且可以利用他们来进行一些类似排序等操作:

```
NSMutableArray *array = [[NSMutableArray alloc] initWithObjects:
@"Item 1",
@"Item 4",
@"Item 2",
@"Item 5",
@"Item 3", nil];
NSLog(@"Array = %@", array);
if ([NSArray instancesRespondToSelector:@selector(sortUsingComparator:)]){
/* 使用 sortUsingComparator: 实例方法来排列数组*/
}
else if ([NSArray instancesRespondToSelector:
@selector(sortUsingFunction:context:)]){
/* 使用 sortUsingFunction:context: 实例方法做数组排序*/
}
else {
```

```
/* 处理其他状况 */  
}
```

因此，在这个例子中，我们正在检查存在的具体实例方法，使用 `instancesRespondToSelector:` `NSMutableArray` 类（这 `NSArray` 的子类）的类的方法。另外，我们可以使用数组实例方法 `respondToSelector:`。

```
NSMutableArray *array = [[NSMutableArray alloc] initWithObjects:  
    @"Item 1",  
    @"Item 4",  
    @"Item 2",  
    @"Item 5",  
    @"Item 3", nil];  
NSLog(@"Array = %@", array);  
if ([array respondsToSelector:@selector(sortUsingComparator:)]) {  
    /* 使用 sortUsingComparator: 实例方法来排列数组 */  
}  
else if ([array respondsToSelector:@selector(sortUsingFunction:context:)]) {  
    /* 使用 sortUsingFunction:context: 实例方法做数组排序 */  
}  
else {  
    /* 处理其他状况 */  
}
```

奇妙点。我们检查了存在的实例方法，那类方法呢？

`NSArray` 类有各种类方法，其中两个是 `arrayWithObjects:` 与 `arrayObjects:count:`。我们可以在运行时判断可用性以及初始化数组：

```
NSArray *array = nil;  
if ([NSArray respondsToSelector:@selector(arrayWithObjects:count:)]) {  
    NSString *strings[4];  
    strings[0] = @"String 1";  
    strings[1] = @"String 2";  
    strings[2] = @"String 3";  
    strings[3] = @"String 4";  
    array = [NSArray arrayWithObjects:strings  
        count:4];  
}  
else if ([NSArray respondsToSelector:@selector(arrayWithObjects:)]) {  
    array = [NSArray arrayWithObjects:  
        @"String 1",  
        @"String 2",  
        @"String 3",  
        @"String 4",  
        nil];  
}  
else {  
    /* 处理 else */  
}  
NSLog(@"Array = %@", array);
```

1.19.4. 参考 XXX

1. 20. 确认类是否可在运行期使用

1.20.1. 问题

你正在使用最新版的 SDK，但是你无法确定是否可以使用，因为无法肯定用户是否有安装最新版的

SDK。

1. 20. 2. 方案

使用 `NSStringFromClass` 函数。传入类的名称字符串。若是返回值为空(`nil`)，则表示这个类无法在这台装置上使用;反之，这个类则可照你希望的在这台装置上使用。参照下面范例代码：

```
if (NSStringFromClass(@"NSJSONSerialization") != nil){
/* 你可以使用这个类 */
[NSJSONSerialization JSONObjectWithData:... /* 在这传入数据 */
options:... /* 在这传入选项 */
error:...]; /* 这里处理错误 */
} else {
/* 该类不可用 */
}
```

1.20.3. 讨论

用户升级操作系统的速度缓慢，这已经不是什么秘密了。在不同公司工作过的我可以证实大约 30% 的 iOS 装置所安装的 iOS 版本比最新的慢上一年或一年半。

例如，若今天我们使用 iOS5 工作，仍然会有很多 iOS 装置是安装 iOS3。确保开发人员的收益是非常重要的，所以必须尽量也能支持旧版本的装置，好创造出更大的用户群，而不是在应用商店上卖只能执行 iOS5 的应用。

某些类只能在特定的 iOS 版本上执行。例如 `NSJSONSerialization` 这个类只能在安装 iOS5 SDK 的装置上执行。然而，若你打算支持 iOS4 与 5，就必须使用 `NSStringFromClass` 检测这个类是否可以使用。若返回值为空(`nil`)，表示不能在这台装置上使用该类。这种情况下，就必须使用替代方案，或自行实践该功能。

1. 20. 4. 参考

XXX

1. 21. 申请、使用字符串

1. 21. 1. 问题

你想在 Objective-C 中使用字符串

1. 21. 2. 方案

使用 `NSString` 和 `NSMutableString` 类

1. 21. 3. 讨论

`NSString` 和 `NSMutableString` 类允许你把一串字符存到内存中。`NSString` 类是不能更改的，`NSString` 类一旦被创建，内容就不能被修改了。可变字符串 `NSMutableString` 创建以后还可以被修改。我们接下来很快会看到两种类的例子。

Objective-C 字符串应该用双引号括起来。双引号的前面应该加上 `@` 符号。例如，`"Hello, world"` 这个字符串在 Objective-C 中应该表示为：

`@ "Hello, world"`

有很多种方法可以把一个字符串赋给 `NSString` 和 `NSMutableString` 类的实例。如下：

```
NSString *simpleString = @"This is a simple string";
```

```
NSString *anotherString =
[NSString stringWithString:@"This is another simple string"];
NSString *oneMoreString =
[[NSString alloc] initWithString:@"One more!"];
NSMutableString *mutableOne =
[NSMutableString stringWithString:@"Mutable String"];
NSMutableString *anotherMutableOne =
[[NSMutableString alloc] initWithString:@"A retained one"];
NSMutableString *thirdMutableOne =
[NSMutableString stringWithString:simpleString];
```

如果你正在使用字符串，运行时你可能不时的需要得到字符串的长度来做出某些操作。设想一下这个场景：你要求用户把他的名字输入到一个文本框里。当用户按下确认按钮时，你可能需要检查他是否真的输入了名字。通过调用 `NSString` 类或者其他子类例如 `NSMutableString` 的实例的 `length` 方法你可以实现这个功能。

如下：

```
NSString *userName = ...;
if ([userName length] == 0){
/* The user didn't enter her name */
} else {
/* The user did in fact enter her name */
}
```

另一个你可能想知道的关于字符串的功能是如何把一个字符串转换成整型。例如，字符串转 `int`、`float` 或 `double` 类型。你可以使用 `NSString`（或者它的子类）的 `integerValue`、`floatValue`，和 `doubleValue` 方法来获得 `int`、`float` 或 `double` 类型的值。

例如：

```
NSString *simpleString = @"123.456";
NSInteger integerOfString = [simpleString integerValue];
NSLog(@"integerOfString = %ld", (long)integerOfString);
CGFloat floatOfString = [simpleString floatValue];
NSLog(@"floatOfString = %f", floatOfString);
double doubleOfString = [simpleString doubleValue];
NSLog(@"doubleOfString = %f", doubleOfString);
```

上边这段代码的输出是：

```
integerOfString = 123
floatOfString = 123.456001
doubleOfString = 123.456000
```

如果你喜欢使用 C 语言库里的字符串，就不需要使用 @ 符号了。比如：

```
char *cString = "This is a C String";
```

如果你想把一个 `NSString` 转换成 C 库里的字符串，必须用 `NSString` 的 `UTF8String`。例如：

```
const char *cString = [@"Objective-C String" UTF8String];
NSLog(@"cString = %s", cString);
```

Tips:

你可以使用 `%S` 将一个 C 字符串格式化输出到控制台。相应的，使用 `%@` 来格式输出 `NSString` 对象。

要把一个 C 字符串转换成 `NSString`，必须使用 `NSString` 类的 `stringWithUTF8String:` 方法，如下：

```
NSString *objectString = [NSString stringWithUTF8String:"C String"];
NSLog(@"objectString = %@", objectString);
In order to find a string inside another string, you can use the rangeOfString: method
of NSString. The return value of this method is of type NSRange:
typedef struct _NSRange {
    NSUInteger location;
    NSUInteger length;
};
```

```
} NSRange;
```

如果你要查找的字符串（针）在一个目标字符串（草垛）中，NSRange 结构的 location 成员将被置为从零开始的索引用来表示第一个针在草垛中的位置。如果草垛中不包含针，location 值会给设为 NSNotFound。我们来看这个例子：

```
NSString *haystack = @"My Simple String";
NSString *needle = @"Simple";
NSRange range = [haystack rangeOfString:needle];
if (range.location == NSNotFound){
/* Could NOT find needle in haystack */
} else {
/* Found the needle in the haystack */
NSLog(@"Found %@ in %@ at location %lu",
needle,
haystack,
(unsigned long)range.location);
}
```

Tips: 用来查询的 NSString 类的 rangeOfString 方法是大小写敏感的。

如果你在查询字符串上拥有更多控制权的话，你可以使用 rangeOfString:options: 方法。其中 options 参数是 NSStringCompareOptions 类型的。

```
enum {
NSCaseInsensitiveSearch = 1,
NSLiteralSearch = 2,
NSBackwardsSearch = 4,
NSAnchoredSearch = 8,
NSNumericSearch = 64,
NSDiacriticInsensitiveSearch = 128,
NSWidthInsensitiveSearch = 256,
NSForcedOrderingSearch = 512,
NSRegularExpressionSearch = 1024
};
typedef NSUInteger NSStringCompareOptions;
```

就像你看到的，枚举中的这些值是可以一起使用的。这表示你可以用 OR 操作符（| 字符）连接它们。假设我们想再一个字符串中查找另外一个字符串，但是我们不关心大小写。我们想做的就是在一个字符串中查找另外一个，不管大小写是否一样。我们可以这么做：

```
NSString *haystack = @"My Simple String";
NSString *needle = @"simple";
NSRange range = [haystack rangeOfString:needle
options:NSCaseInsensitiveSearch];
if (range.location == NSNotFound){
/* Could NOT find needle in haystack */
} else {
/* Found the needle in the haystack */
NSLog(@"Found %@ in %@ at location %lu",
needle,
haystack,
(unsigned long)range.location);
}
```

你可以看到我们用 NSCaseInsensitiveSearch 值使用了 NSString 类的 rangeOfString:options: 方法。这个方法告诉运行时我们在查找时不考虑大小写。

可变字符串和不可变字符串是相似的。不同的是可变字符串可以在运行时被修改。我们来看一个例子：

```
NSMutableString *mutableString =
[[NSMutableStringalloc] initWithString:@"My MacBook"];
```

```
/* Add string to the end of this string */
[mutableStringappendString:@" Pro"];
/* Remove the "My " string from the string */
[mutableString
replaceOccurrencesOfString:@"My "
withString:[NSString string] /* Empty string */
options:NSCaseInsensitiveSearch /* Case-insensitive */
range:NSMakeRange(0, [mutableString length])]; /* All to the end */
NSLog(@"mutableString = %@", mutableString);
```

当 mutableString 字符串打印到控制台后，我们能看到这个结果：

```
mutableString = MacBook Pro
```

你可以看到我们以“My MacBook”开始字符串，然后从原始字符串中删除了字符串“My”。所以现在只剩下“MacBook”。这之后我们再字符串的末尾加上“Pro”得到了最后结果“MacBook Pro”。

1. 21. 4. 参考 XXX

1. 22. 申请、使用数字

1. 22. 1. 问题

你需要在对象中使用整型或封装好的数字。

1. 22. 2. 方案

用 NSNumber 类来用面向对象的方法处理数字。如果你只需要简单的数字（而不是对象），用 NSInteger 类来操作有符号数（正数或者负数），用 NSUInteger 类来操作无符号数（正数或 0），用 CGFloat 类和 double 在操作浮点数。

1. 22. 3. 讨论

正如我们用 NSString 对象来存储字符串，我们可以用 NSNumber 对象来存储数字。你可能会问为什么？答案很简单：允许一个对象存储数字的值可以让我们方便的把数字存到磁盘，从磁盘取出。还能用一个简单的对象存储有符号、无符号以及浮点数，不要定义和转换各种不同类型的变量。变量的类型是无穷尽的。

让我们来看看 NSNumber 对象的构造函数：

```
NSNumber *signedNumber = [NSNumber numberWithInt:-123456];
NSNumber *unsignedNumber = [NSNumber numberWithInt:123456];
NSNumber *floatNumber = [NSNumber numberWithFloat:123456.123456f];
NSNumber *doubleNumber = [NSNumber numberWithDouble:123456.1234567890];
```

正如我们用 NSNumber 类存储有符号、无符号整型以及浮点数一样，我们可以通过调用 NSNumber 类的一些非常实用的方法来获取这些值。如下：

```
NSNumber *signedNumber = [NSNumber numberWithInt:-123456];
NSNumber *unsignedNumber = [NSNumber numberWithInt:123456];
NSNumber *floatNumber = [NSNumber numberWithFloat:123.123456f];
NSNumber *doubleNumber = [NSNumber numberWithDouble:123.1234567890];
NSInteger signedValue = [signedNumber integerValue];
```



```
NSNumber *unsignedNumber = [NSNumber numberWithInt:123456];
CGFloat floatValue = [floatNumber floatValue];
double doubleValue = [doubleValue doubleValue];
NSLog(@"signedValue = %ld, \n\"
      "unsignedValue = %lu \n\"
      "floatValue = %f \n\"
      "doubleValue = %f",
      (long)signedValue,
      (unsigned long)unsignedValue,
      floatValue,
      doubleValue);
```

在这段代码中用到的操作 `NSNumber` 实例的方法有以下几个：

`numberWithInteger:`

将一个整型值封装成一个 `NSNumber` 实例

`numberWithUnsignedInteger:`

将一个无符号整型值（正数或零）封装成一个 `NSNumber` 实例

`numberWithFloat:`

将一个浮点数封装成一个 `NSNumber` 实例

`numberWithDouble:`

将一个 `double` 类型的数封装成一个 `NSNumber` 实例

下面这些方法用来从一个 `NSNumber` 实例提取纯数字：

`integerValue`

从调用该函数的 `NSNumber` 实例中返回一个整型 `NSInteger` 类型值。

`unsignedIntegerValue`

从调用该函数的 `NSNumber` 实例中返回一个无符号整型 `NSUInteger` 类型值。

`floatValue`

从调用该函数的 `NSNumber` 实例中返回一个浮点数 `CGFloat` 类型值。

`doubleValue`

从调用该函数的 `NSNumber` 实例中返回一个双精度 `double` 类型值。

如果你想拿一个数字和一个字符串比较，可以把数字转换为你认为最近似于该数值的类型（整型或浮点型），然后将字符串格式化为合适的格式。例如，要将一个无符号整型转换为 `NSString`，你可以用 `%lu` 格式符，像这样：

```
NSNumber *unsignedNumber = [NSNumber numberWithInt:123456];
/* 将一个 NSNumber 对象中的无符号整型转换为 NSString */
NSString *numberInString =
[NSString stringWithFormat:@"%lu",
 (unsigned long)[unsignedNumber unsignedIntegerValue]];
NSLog(@"numberInString = %@", numberInString);
```

需要注意的是 `NSNumber` 类的任何一个以 `numberWith` 开头的方法都返回一个自动释放的 `NSNumber` 对象。要释放自动释放池中的内存，可以在分配空间后之后使用 `NSNumber` 的 `initWith...` 方法。像这样：

```
NSNumber *unsignedNumber =
[[NSNumber alloc] initWithUnsignedInteger:123456];
```

1. 22. 4. 参考

XXX

1. 23. 分配、使用数组

1. 23. 1. 问题

你想要把一系列对象存储到另外一个对象中以便使用。

1. 23. 2. 方案

用 `NSArray` 和 `NSMutableArray` 类把多个对象存储到操作相对更加方便的数组中。

1. 23. 3. 讨论

一个 `NSArray` 或者其子类的对象可以存储 `n` 个其他类型的对象。这些对象可以通过索引访问。例如，假设我们有 10 双袜子，我们刚所有的袜子从左到右放到一个平台上。我们一次叫它们：袜子 1，袜子 2，袜子 3 等等。所以最左边的袜子叫做袜子 1，挨着它的叫做袜子 2，以此类推。这样是不是比说“我红色袜子旁边的蓝色袜子”要简单的多？这就是数组做的事，它能使操作多个条目更加容易。

图图：你可以把任何 `NSObject` 类或者它子类的对象放到 `NSArray` 类型中。

`NSArray` 和 `NSMutableArray` 的主要区别是 `NSMutableArray` 可以在分配空间和初始化之后修改，然而 `NSArray` 不可以被修改。

让我没来看一个例子。让我们创建一个 `NSString` 实例，两个 `NSNumber` 实例并且把它们存到一个不可变数组中：

```
NSString *stringObject = @"My String";
NSNumber *signedNumber = [NSNumber numberWithInt:-123];
NSNumber *unsignedNumber = [NSNumber numberWithInt:123];
NSArray *array = [[NSArray alloc] initWithObjects:
stringObject,
signedNumber,
unsignedNumber, nil];
NSLog(@"array = %@", array);
```

当你运行这段代码的时候，能在控制台得到下面的输出：

```
array = (
    "My String",
    "-123",
    123
)
```

你可以看到，我们用 `initWithObjects:` 来初始化 `array`。当你用这个初始化函数时，把你要存储的对象一个一个传递进去。最后用一个 `nil` 符号结束这个列表以通知运行时列表什么时候结束。如果你不这么做，LLVM 编译器会抛出一个如下的异常：

```
warning: Semantic Issue: Missing sentinel in method dispatch
```

我们也可以使用 `NSArray` 类的 `arrayWithObjects:` 方法来创建一个可自动释放的数组。像这样：

```
NSArray *array = [NSArray arrayWithObjects:
stringObject,
signedNumber,
unsignedNumber, nil];
```

你可以调用数组的 `count` 方法类得到数组中的对象个数。

你也可以通过 `for` 循环或者枚举来遍历数组。让我们首先来看一下用 `for` 循环解决的第一种方案。

```
NSArray *array = [NSArray arrayWithObjects:
stringObject,
signedNumber,
unsignedNumber,nil];
NSUInteger counter = 0;
for (counter = 0;
counter < [array count];
counter++){
id object = [array objectAtIndex:counter];
NSLog(@"Object = %@", object);
}
```

这是输出:

```
Object = My String
Object = -123
Object = 123
```

就像你看到的, 我们用 `objectAtIndex:` 方法来获得一个指定位置的对象。

记住索引时从 0 开始的。换句话说, 当计数器到达-1 时循环必须停止因为在数组中没有负索引。

以前提到过, 你也可以利用快速枚举来遍历一个数组中的对象。快速枚举是 Objective-C 语言中的一个语言特性。它可以不用计数器或循环就可以实现枚举数组或字典 (或其他支持快速枚举的对象)。格式如下:

```
for (Type variableName in array/dictionary/etc){ ... }
```

帮助我们不用计数变量实现上面的功能。我们用快速枚举这么做:

```
for (id object in array){
NSLog(@"Object = %@", object);
}
```

这样做的得到的结果和我们用前面利用计数器的代码得到的结果完全一样。可变数组非常有趣, 就像你猜想的一样, 不可变字符串一旦被分配了内存且初始化之后就不能修改了。但是数组在分配空间初始化之后可以被修改。让我们来看个例子:

```
NSString *stringObject = @"My String";
NSNumber *signedNumber = [NSNumber numberWithInt:-123];
NSNumber *unsignedNumber = [NSNumber numberWithInt:123];
NSArray *anotherArray = [[NSArray alloc] initWithObjects:
@"String 1",
@"String 2",
@"String 3", nil];
NSMutableArray *array = [[NSMutableArray alloc] initWithObjects:
stringObject,
signedNumber, nil];
[array addObject:unsignedNumber];
[array removeObject:signedNumber];
[array addObjectsFromArray:anotherArray];
for (id object in array){
NSLog(@"Object = %@", object);
}
```

分析这段代码之前, 让我们看下输出:

```
Object = My String
Object = 123
Object = String 1
Object = String 2
Object = String 3
```

你可能会问发生了什么, 那么让我们来看看我们都用了 `NSMutableArray` 类的哪些方法:

addObject:

这个方法允许我们在个可变数组的末尾添加一个对象。

removeObject:

使用这个方法, 我们可以删除数组中的一个指定对象。记住删除对象时, 我们传递的参数是对象, 而不是

对象的索引。

想要通过索引来删除数组中的对象，必须使用 `removeObjectAtIndex` 方法。

`addObjectsFromArray`:

通过这个方法，我们可以将一个可变或不可变数组添加到一个可变数组中。

图图：请记住，在使用快速枚举或不可变数组的过程中，禁止向数组添加对象或许从数组删除对象，否则就会引起运行时错误。这是可变数组快速排序时的默认行为。系统会自动实现。

如果你对块对象感兴趣（我们在本书的后面会介绍它的好处），你也可以用 `enumerateObjectsUsingBlock` 方法来枚举数组中的对象。传递到该方法的块对象应该：

1. 无返回值

2. 有 3 个参数

a. 第一个参数是 `id` 类型，用来表示每次循环中的枚举的对象。

b. 第二个参数是 `NSUInteger` 类型，用来表示当前枚举对象的索引。

c. 最后一个（还可以有更多个）参数是 `*BOOL` 类型，用来停止枚举。这是一个 `BOOL` 指针，如果你需要继续枚举的话它的值必须为 `NO`。你可以随时将这个指针的值改为 `YES` 来停止枚举。可以用这个来在数组中检索一个对象。一旦找到对象马上停止检索因为如果你已经找到那个对象的话就没有指针来继续这个枚举了。

```
NSArray *myArray = [[NSArray alloc] initWithObjects:
```

```
@\"String 1\",
```

```
@\"String 2\",
```

```
@\"String 3\",
```

```
@\"String 4\", nil];
```

```
[myArray enumerateObjectsUsingBlock:
```

```
^(__strong id obj, NSUInteger idx, BOOL *stop) {
```

```
<!--
```

```
AO: *stop 不能在前面被设为 NO 吗？还是 *stop 默认值就是 NO
```

```
-->
```

```
NSLog(@"Object = %@", obj);
```

```
});
```

如果你需要为一个数组排序，简单的利用 `NSArray` 或 `NSMutableArray` 类的基于块对象的排序方法。需要注意的是 `NSArray` 类的排序方法返回一个新的额 `NSArray` 实例，原来的数组没有改变。这是因为 `NSArray` 一旦初始化之后就不能被改变，然而排序是要改变一个数组的。相对应的，`NSMutableArray` 类的排序方法返回的就是要排序的数组，而不会返回一个新的数组。我们来看一个不可变数组排序的例子：

```
NSMutableArray *myArray = [[NSMutableArray alloc] initWithObjects:
```

```
@\"String 2\",
```

```
@\"String 4\",
```

```
@\"String 1\",
```

```
@\"String 3\", nil];
```

```
[myArray sortUsingComparator:
```

```
^NSComparisonResult(__strong id obj1, __strong id obj2) {
```

```
NSString *string1 = (NSString *)obj1;
```

```
NSString *string2 = (NSString *)obj2;
```

```
return [string1 compare:string2];
```

```
});
```

```
NSLog(@"myArray = %@", myArray);
```

输出到控制台的结果如下：

```
myArray = (
```

```
\"String 1\",
```

```
\"String 2\",
```

```
\"String 3\",
```

```
\"String 4\"
```

```
)
```

那么发生了什么呢？我们只是简单的调用了数组的 `sortUsingComparator` 方法。

这个方法传进来一个块对象（用 `^` 字符表示），而且必须返回一个 `NSComparisonResult` 类型的值。

这个值可以是一下几种：

NSOrderedSame

参与比较的两个值相等。

NSOrderedAscending

左边的值比右边的小。这么来记：从左到右是升序的，意味着左边的值更小一些。

NSOrderedDescending

右边的值比左边的小。换句话说：从左到右是降序的，意味着左边的值更大一些。

如果我们把 String 3 当作 value 1，String1 当作 value2，结果是升序、降序还是相等呢？毫无疑问是降序因为 value 1（String 3）到 value 2（String 1）是递减的。

注意不管 compare:方法得到的那个字符串，调用的是左边的对象或者 value 1。传递到 sortUsingComparator:方法的块对象有两个参数：

First Object of type id

这是每次迭代中参与比较的第一个对象。

Second Object of type id

这是每次迭代中参与比较的第二个对象。

所以当为数组排序时，就用基于块的方法。这是 Apple 公司推动开发者与他们共同前进的方法。所以了解块儿对象非常有用。

1.23.4. 参考 XXX

1.24. 分配和使用 Dictionaries

1.24.1. 问题

你需把一组对象以 K-Value(键值对)的形式保存起来，数组没办法做到。

1.24.2. 方案

使用 NSDictionary 类以及可改变长度的 NSMutableDictionary 类。

1.24.3. 讨论

一个 dictionary 是一个特殊数组，其中的每个项都包含一个 key。这事 dictionary 和 array 唯一不同的地方。array 的每一项（对象）都有一个数字的索引，然而 dictionary 中的每一项（对象）都有一个 key。解释一下，假设我们要把一个人的姓、名字以及年龄存储到一个 array 和一个 dictionary 中。我们应该按以下方式存储：

```
NSArray *person = [[NSArray alloc] initWithObjects:
@"Anthony",
@"Robbins",
NSNumber numberWithInt:51, nil];
NSLog(@"First Name = %@", [person objectAtIndex:0]);
NSLog(@"Last Name = %@", [person objectAtIndex:1]);
NSLog(@"Age = %@", [person objectAtIndex:2]);
```

可以看到我们用索引来获取 array 中的对象。使用 dictionaries 时，每个对象都有一个 key 值。这个 key 值是个对象，我们可以用它来访问 dictionaries 中的值。

我们来看相同的例子用 dictionaries 怎么做。我们有一个 key"First Name"对应"Anthony"值，以此类推：

```
NSNumber *age = [NSNumber numberWithInt:51];
NSDictionary *person = [[NSDictionary alloc] initWithObjectsAndKeys:
@"Anthony", @"First Name",
@"Robbins", @"Last Name",
age, @"Age",
nil];
```

```
NSLog(@"First Name = %@", [person objectForKey:@"First Name"]);
NSLog(@"Last Name = %@", [person objectForKey:@"Last Name"]);
NSLog(@"Age = %@", [person objectForKey:@"Age"]);
The results will then be printed out as shown here:
First Name = Anthony
Last Name = Robbins
Age = 51
```

可以看到，我们用值和键的对儿初始化 `dictionary`。我们在先传递值，然后传值对应的 `key`。当我们用 `NSLog` 输出时，用 `key` 作参数调用 `objectForKey:` 方法来输出 `key` 对应的 `value` 值。`NSDictionary` 对应的可改变 `dictionary` 类 `NSMutableDictionary` 在分配内存和初始化之后可以被修改。例如，如果在 `dictionary` 初始化之后想从其中删除 `Age` 这个 `key` 对应的对象，我们应该这么做：

```
NSNumber *age = [NSNumber numberWithInt:51];
NSMutableDictionary *person = [[NSMutableDictionary alloc]
initWithObjectsAndKeys:
@"Anthony", @"First Name",
@"Robbins", @"Last Name",
age, @"Age",
nil];
[person removeObjectForKey:@"Age"];
NSLog(@"First Name = %@", [person objectForKey:@"First Name"]);
NSLog(@"Last Name = %@", [person objectForKey:@"Last Name"]);
NSLog(@"Age = %@", [person objectForKey:@"Age"]);
```

我们方便的删除了与 `Age` 这个 `key` 对应的对象。输出如下：

```
First Name = Anthony
Last Name = Robbins
Age = (null)
Note that "Age" is not just empty, but totally missing.
```

注意到 `"Age"` 不只为空了，而且以及不存在了。

如果想枚举 `dictionary` 中所有的 `key`，可以简单用 `enumerateKeysAndObjectsUsingBlock:` 方法。在前文体到的 `array` 中，`enumerateKeysAndObjectsUsingBlock` 方法会输出 `"First Name"` 和 `"Last Name"`，但没有 `"Age"`，因为已经被删除了。这个方法的参数是一个包含 3 个参数而且没有返回值的块对象。3 个参数如下：

Key

标识当前正在枚举的 `key` 的一个 `id`。

Object

也是一个 `id`，标识当前 `key` 对应的对象。

一个 `BOOL` 型指针

在枚举的过程中，你可以随时通过给这个指针传一个 `YES` 值来停止这个进程。如果想遍历 `dictionary` 中所有 `key` 就别动这个指针。

我们来看个例子：

```
NSNumber *age = [NSNumber numberWithInt:51];
NSDictionary *person = [[NSDictionary alloc] initWithObjectsAndKeys:
@"Anthony", @"First Name",
@"Robbins", @"Last Name",
age, @"Age",
nil];
[person enumerateKeysAndObjectsUsingBlock:
^(__strong id key, __strong id obj, BOOL *stop) {
NSLog(@"Key = %@", Object For Key = %@", key, obj);
}];
```

输出结果如下：

```
Key = Last Name, Object For Key = Robbins
```



```
Key = First Name, Object For Key = Anthony  
Key = Age, Object For Key = 51
```

如果你不想用块对象来快速枚举，可以采用 dictionary 的 allKeys 方法来做。一旦得到所有的 key 就可以通过 objectForKey

Key: 方法并用这些 key 来找到对应的对象。像这样：

```
for (id keyInDictionary in [person allKeys]){  
    idobjectForKey = [person objectForKey:keyInDictionary];  
    NSLog(@"Key = %@, Object For Key = %@", keyInDictionary, objectForKey);  
}
```

需要注意的是遍历 dictionary 中 key 的方法有很多种。前面已经介绍了 2 种。还有一种方法：调用 dictionary 的 keyEnumerator 方法来获得一个 NSEnumerator 对象。来看个例子：

```
NSEnumerator *keys = [person keyEnumerator];  
idkeyInDictionary = nil;  
while ((keyInDictionary = [keys nextObject]) != nil){  
    idobjectForKey = [person objectForKey:keyInDictionary];  
    NSLog(@"Key = %@, Object For Key = %@", keyInDictionary, objectForKey);  
}
```

使用一个可变 dictionary 的 keyEnumerator 方法时，不允许在遍历 key 的过程中改变 dictionary 中 key 对应的值。如果你记得的话，可变的 arrays 也有同样的规则。

1. 24. 4. 参考 XXX

1. 25. 分配和使用 Sets

1. 25. 1. 问题

你需要存储一组对象但是每个对象只存一次（剔除重复的数据）。

1. 25. 2. 方案

用集合（sets）来代替数组。

1. 25. 3. 讨论

Sets 和 array 非常相似。二者最大的区别就是 sets 中相同对象只能被添加一次。当你第二次添加同一个对象时，sets 会拒绝添加。

我们使用 NSSet 类表示不可改变的 sets，用 NSMutableSet 类表示可变 sets。我们来看一个不可变 sets 的例子：

```
NSString *hisName = @"Robert";  
NSString *hisLastName = @"Kiyosaki";  
NSString *herName = @"Kim";  
NSString *herLastName = @"Kiyosaki";  
NSSet *setOfNames = [[NSSet alloc] initWithObjects:  
    hisName,  
    hisLastName,  
    herName,  
    herLastName, nil];  
NSLog(@"Set = %@", setOfNames);
```

我们创建了一个不可改变的 set 并传了 4 个字符串对象来初始化它。我们来看看 NSLog 函数输出了什么：

```
Set = {(  
    Kim,  
    Robert,  
    Kiyosaki
```

```
}}
```

你可以看到名字 Kiyosaki 只添加了一次。Set 拒绝将同一个对象加入列表两次。一个 set 不仅仅比较对象在内存中的位置，而且比较这些对象的内容，了解这点非常重要。hisLastName 和 herLastName 是两个不同的变量，它们在内存中的地址不同。然而 set 知道我们传递了两个 NSString 的实例给它，而且可以比较这些字符串从而发现我们已经添加过了一个 Kiyosaki 名字。所以 set 中只留下了一个实例。

现在来看一下如何构造一个可变 sets:

```
NSMutableSet *setOfNames = [[NSMutableSet alloc] initWithObjects:
hisName,
hisLastName, nil];
[setOfNames addObject:herName];
[setOfNames addObject:herLastName];
```

我们可以方便的通过 NSMutableSet 类的 addObject: 方法类像 set 添加一个新的对象。你也可以用 removeObject 方法来删除对象。同样要记住对象的内容，不只是地址。所以如果想从一个 set 中删除一个字符串，可以将字符串传递给 removeObject: 方法，即使新字符串是内存中的不同变量。只要这个字符串/对象的内容相同你就可以得到想要的结果。如下:

```
NSMutableSet *setOfNames = [[NSMutableSet alloc] initWithObjects:
hisName,
hisLastName,
herName,
herLastName, nil];
[setOfNames removeObject:@"Kiyosaki"];
NSLog(@"Set = %@", setOfNames);
```

输出结果是:

```
Set = {
Kim,
Robert
}
```

如果你想快速遍历一个 set 中所有的对象，可以使用 enumerateObjectsUsingBlock: 方法。传递给这个方法的块对象没有返回值，有两个参数:

- 一个 id 类型的 key
- 包含 set 中当前枚举的对象。

一个 BOOL 类型的指针

你可以随时将一个 boolean 值 YES 传给这个指针来停止遍历。

我们来看个例子。假设我们想在一个 set 中查找字符串 Kiyosaki:

```
[setOfNames enumerateObjectsUsingBlock:^(__strong id obj, BOOL *stop) {
if ([obj isKindOfClass:[NSString class]]){
NSString *string = (NSString *)obj;
if ([string isEqualToString:@"Kiyosaki"]){
NSLog(@"Found %@ in the set", string);
*stop = YES;
}
}
}];
```

如果能在在 set 中找到一个 Kiyosaki 值，我们将字符串打印到控制台。然后通过将 YES 值传给遍历块对象的第二个参数来结束遍历。还有其他手动方法。用 count 方法来获得 set 中的对象个数。也可以用 allObjects 方法来获得 set 中包含所有对象的数组。

如果想在 set 中获取一个对象，调用 set 的 anyObject 方法。就像函数名代表的，这个函数返回 set 中一个随机的对象。如果 set 为空，则返回空。

注意不可变的空 set 绝对是没有用的。它是空的而且在它的生命周期中一直为空。

1. 25. 4. 参考 XXX

1. 26. 创建程序包

1. 26. 1. 问题

你想把资源按照层次结构分组以便在运行时能够随时调用。

1. 26. 2. 方法

按照下面的步骤成功创建一个程序包：

1. 在磁盘中创建给一个根文件夹，也是之后的程序包；例如，我们把这个文件夹命名为资源。
2. 在资源文件夹下创建 3 个命名为图片、视频和声音的子文件夹；
3. 在上面提到的 3 个子文件夹内分别放入相应的资源；例如在图片文件夹放入一张或者多张图片，在视频文件夹里放入一个或者多个视频文件等等。
4. 完成以上步骤之后，将资源文件夹重新命名为 `Resource.bundles`，一旦给文件命名时加上这个扩展名，OS X 会要求你确认并在显示器上弹出一个如图 1-33 所示的对话框，点击“增加”以便在文件夹中能够增加.bundles 扩展名。

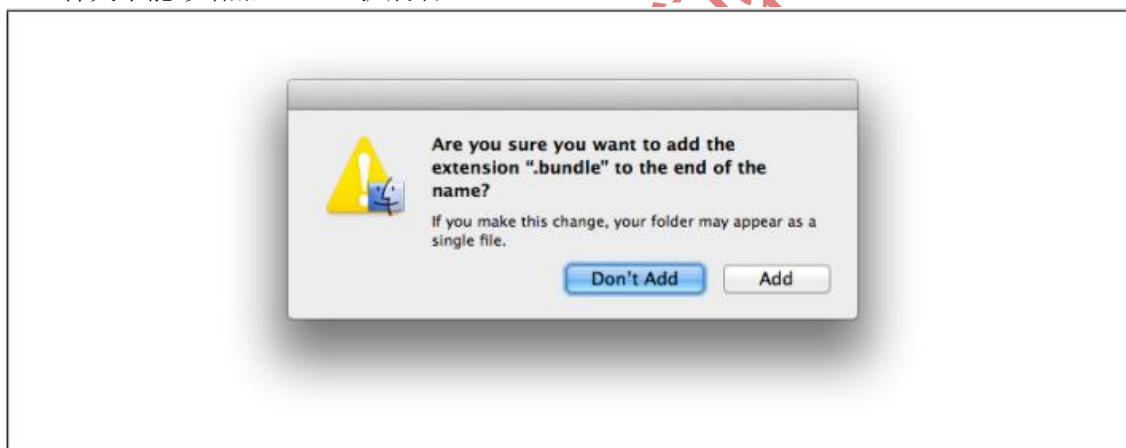


图 1-33 在文件名增加一个.bundle 扩展名

1. 26. 3. 讨论

文件包是以一个.bundle 为扩展名的文件包。和普通文件夹相比它们有 2 个主要特点：

10. Cocoa Touch 提供了一个界面，通过这个界面你可以进入相应的文件包简单获取里边已有资源。

11. 如果在 Xcode 左侧导航窗口增加一个文件包，任何文件增加或者移出文件包将分别立即出现或者消失在 Xcode 导航窗口。相反，如果你在 Xcode 导航增加了一个文件夹，然后再去删除磁盘中这个文件夹中的文件，在没有 Xcode 的帮助下，这个文件将会变成红色并且不能立即删除。文件包非常有用，特别是你想在文件夹中通过 Finder 而非 Xcode 手动增加文件。

每个 iOS 应用至少有一个文件包，叫做主文件包。主文件包包含你的应用软件中的二进制代码和其他在应用软件中使用的资源，比如影像、声音、HTML 文件和其他相关文件。换言之，主文件包包含了你提交给 AppStore 或者发布到自己组织内部的最终二进制文件中的资源。这些资源随后可以用 `NSBundle` 类的 `mainBundle` 类的方法动态加载。



虽然你能给一个 iOS 对象的 2 个或者更多的文件包增加一个相同的名字，但是最好不要这样做，因为会把事情变复杂，当我们开始从文件包加载资源时，首先需要找到对应的文件包；如果两个文件包共

用一个名字，查询起来就相当困难的，无法辨别出哪一个是哪一个是哪一个。所以一个好的解决办法就是在给 Xcode 对象命名时确保文件包使用不同的名称。

1.26.4. 参考

XXX

1.27. 从主文件包加载数据

1.27.1. 问题

已经在 Xcode 对象已经保存了资源（比如图片），现在，在运行时，就可以调用这些资源了。

1.27.2. 方法

使用 `NSBundle` 类中的 `MainBundle` 类实例方法调用主文件包。一旦这个步骤完成，使用 `pathForResource:ofType:` 实例方法从主文件包中获得具体资源的路径，路径明确之后，根据资源的类别可以获取具体文件比如 `UIImage` 或者 `NSData`，或者通过 `NSFileManager` 手动获取文件。



极力推荐对文件包中的每个资源都独立命名。例如，在主文件包一个命名为 `Default.png` 的文件在不只一个地方出现，这种做法不推荐。使用不同方法从一个文件包加载一种资源会导致不同结果。因此，要确保任何一个文件包中的文件能够区别命名，不论是在主文件包或自己创建的自定义文件包。（参考 1.26 中的方法）

1.27.3. 讨论

可以使用 `NSBundle` 类中的 `mainBundle` 类方法获取主文件包，可以从所有 `NSBundle` 类文件包和自己的文件包中加载资源。



每个 APP 在提交 AppStore 时它的主文件包在磁盘中有一个水平层次结构，这就意味着所有装在 App 文件包的文件放在主文件包的根文件夹内。也就是说，主文件包有一个唯一的文件夹—根文件夹，所有文件和资源都放在这个文件夹里。即使你在磁盘中有一个只有几张图片的文件夹，无论拖进或者拖出 Xcode，这个文件夹中的文件都将放在主文件夹里，而不是在这个文件夹自己内部。

比如在桌面上有一个叫做 `AlanSugar.png` 的图片，把它拖入或者拖出 Xcode，此时 Xcode 会弹出一个对话框，提问你这个文件夹会增加什么东西，如果需要是否需要用这个文件复制对象文件夹。对话框内容如图 1-34 所示。

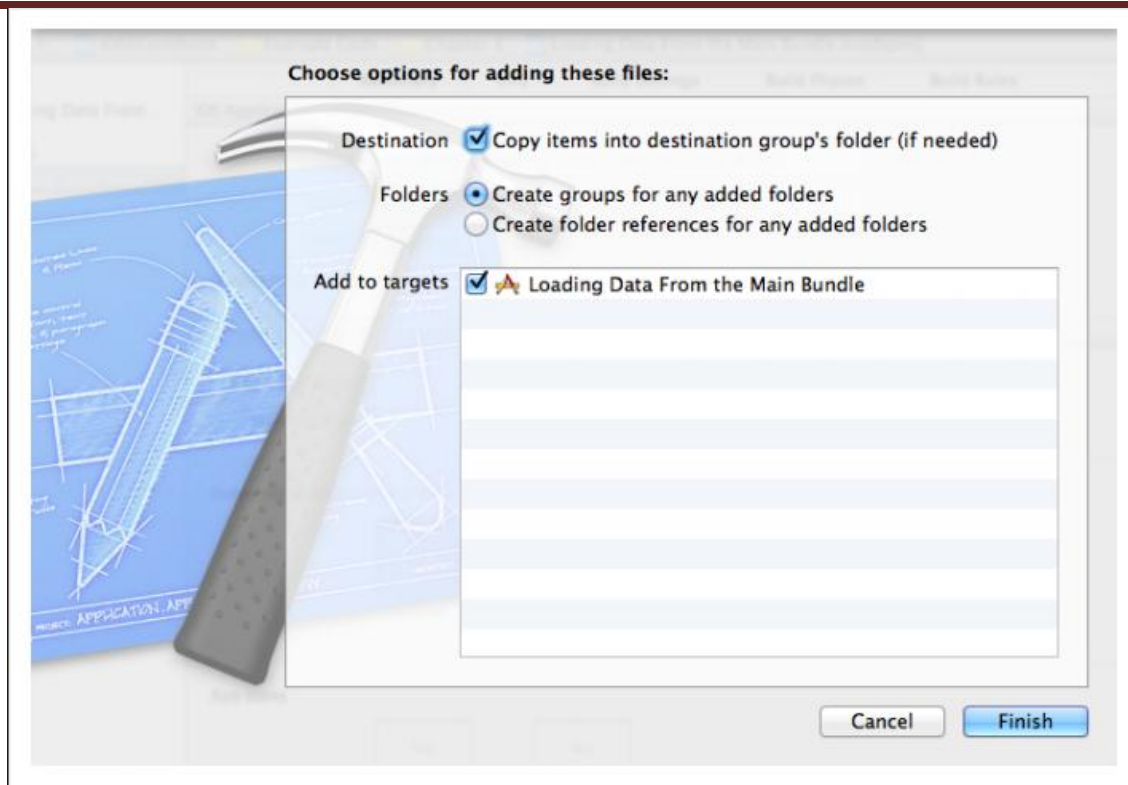


图 1-34 Xcode 提问文件夹必须增加什么内容

在这个对话框中，一定要选择“复制项目到目标群文件夹（如需要）”。这样会把之前放到 Xcode 的文件复制到目标 App 文件。现在，即使在桌面删除了这个文件，它并从对象中删除掉，因为对象已经有了复制了该文件。总体上来说这是个比较好的实例除非因为特别原因有了其他决定（我自己就经历过各种理由无法这样做）。在拖进或者拖出这个文件之后，AlanSugar.png 文件在对象主文件包，你可以通过下面的办法取得路径：

```
- (BOOL) application:(UIApplication*)
didFinishLaunchingWithOptions:(NSDictionary*) LaunchOptions {

    NSString *alanSugarFilePath =
        [[NSBundle mainBundle] pathForResource:@"AlanSugar"
        ofType:@"png"];
    if ([alanSugarFilePath length] > 0){
        UIImage *image = [UIImage imageWithContentsOfFile:alanSugarFilePath];
        if (image != nil){
            NSLog(@"Successfully loaded the file as an image.");
        } else {
            NSLog(@"Failed to load the file as an image.");
        }
    } else {
        NSLog(@"Could not find this file in the main bundle.");
    }
    1.27 Loading Data From the Main Bundle | 95
}

self.window = [[UIWindow alloc] initWithFrame:
                [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

如果具体资源在目标文件包中无法找到，输出 `pathForResource: ofType: NSBundle` 的方法会非常有效又可能没有任何意义。所以在使用这个方法后，最好能检查下是否真正获取路径。如果得到路径，为了能把 AlanSugar.png 以图片格式加载到记忆中，显现的代码会通过文件路径到达 UIImage 类。类似的，如果想把数据加载到内存中，可以改用 NSData 类代替获取图片对象：

```
- (BOOL) application:(UIApplication *)application
```

```
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
NSString *alanSugarFilePath =
    [[NSBundle mainBundle] pathForResource:@"AlanSugar"
ofType:@"png"];
if ([alanSugarFilePath length] > 0){
NSError *readError = nil;
NSData *dataForFile =
    [[NSData alloc] initWithContentsOfFile:alanSugarFilePath
options:NSMappedRead
error:&readError];
if (readError == nil &&
dataForFile != nil){
NSLog(@"Successfully loaded the data.");
    } else if (readError == nil &&
dataForFile == nil){
NSLog(@"No data could be loaded.");
    } else {
NSLog(@"An error occurred while loading data. Error = %@", readError);
    }
    } else {
NSLog(@"Could not find this file in the main bundle.");
    }
self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

1. 27. 4. 参考 XXX

1. 28. 从其他文件包下载数据

1. 28. 1. 问题

在运行时可能想去调用那些在主文件包中一个独立文件包内的已有的一些图片或者其他资源。

1. 28. 2. 方法

在运行时使用 `pathForResource ofType:` 的办法查找在主文件包到达目标文件包的路径，一旦找到路径，通过 `NSBundle` 中 `BundleWithPath:` 类的办法可以方便获取资源。



在继续时候这个方法之前，请根据 1.26 中的指导创建一个 `Resources.Bundle` 的文件包并把它放到主文件包下。

1. 28. 3. 讨论

按照 1.26 提供的方法你会有一个命名为 `Resources.Bundle` 的文件包，其中包含一个图片文件夹。现在在这个文件夹中放一张图片，在一张名为 `AlanSugar.png` 的图片放进文件包之后，图 1-35 会展现文件包中的内容。

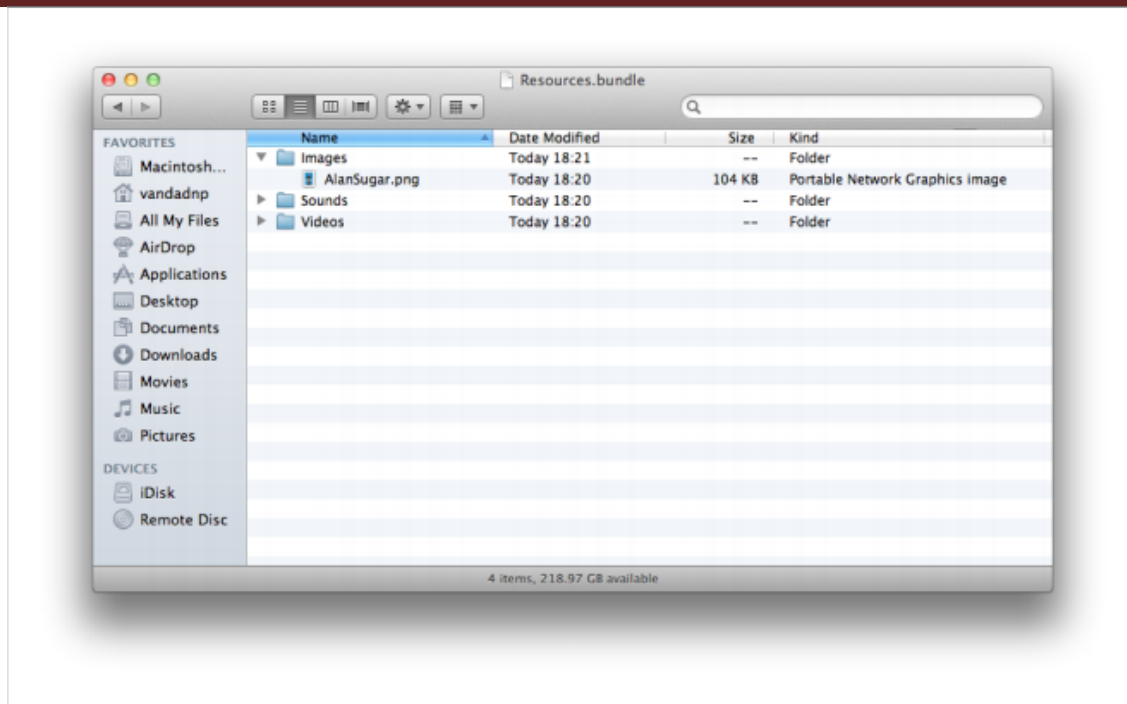


图 1-35. 在之前创建的文件包中放入一张图片

由于 Resources.bundle 已经增加到了应用的主文件包中，需要使用主文件包查找到达 Resources.bundle 的路径；一旦完成，可以直接获取这个文件包中的文件（现在只能获得 AlanSugar.png）。除了主文件包其他文件包可以本身有子文件包，因此获取除了主文件包之外其他文件包内的文件时最好使用 NSBundle 中 pathForResource ofType:inDirectory: 的实例方法，特别是要直接获取那些制定文件包已经存在的文件和资源。

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *resourcesBundlePath =
        [[NSBundle mainBundle] pathForResource:@"Resources"
        ofType:@"bundle"];
    if ([resourcesBundlePath length] > 0){
        NSBundle *resourcesBundle = [NSBundle bundleWithPath:resourcesBundlePath];
        if (resourcesBundle != nil){
            NSString *pathToAlanSugarImage =
                [resourcesBundle pathForResource:@"AlanSugar"
                ofType:@"png"
                inDirectory:@"Images"];
            if ([pathToAlanSugarImage length] > 0){
                UIImage *image = [UIImage imageWithContentsOfFile:pathToAlanSugarImage];
                if (image != nil){
                    NSLog(@"Successfully loaded the image from the bundle.");
                } else {
                    NSLog(@"Failed to load the image.");
                }
            } else {
                NSLog(@"Failed to find the file inside the bundle.");
            }
        } else {
            NSLog(@"Failed to load the bundle.");
        }
    } else {
        NSLog(@"Could not find the bundle.");
    }

    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
}
```

```
return YES;
}
```

如果你尝试查找文件包内特定文件夹里保存的所有资源，可以使用 `NSBundle` 类的实例方法 `pathsForResourceOfType:inDirectory:`

在这段代码中我们尝试查找资源文件包下图片文件夹里所有.png 文件的路径：

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    NSString *resourcesBundlePath =
        [[NSBundle mainBundle] pathForResource:@"Resources"
        ofType:@"bundle"];
    if ([resourcesBundlePath length] > 0){
        NSBundle *resourcesBundle = [NSBundle bundleWithPath:resourcesBundlePath];
        if (resourcesBundle != nil){
            NSArray *PNGPaths = [resourcesBundle pathsForResource:@"png"
            inDirectory:@"images"];
            [PNGPaths
            enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
                NSLog(@"Path %lu = %@", (unsigned long)idx+1, obj);
            }];
        } else {
1.28 Loading Data From Other Bundles | 99
            NSLog(@"Failed to load the bundle.");
        }
    } else {
        NSLog(@"Could not find the bundle.");
    }
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

`NSArray` 的 `enumerateObjectsUsingBlock:` 的实例方法接受一个锁定对象作为参数，更多关于 `enumerateObjectsUsingBlock:` 和它接受的锁定对象的信息请查看 1.25。

1.28.4. 参考 XXX

1.29. 通过 NSNotificationCenter 发送通知

1.29.1. 问题

通过 App 发布一条通知同时允许其他对象接收通知并采取行动，这取决于你发布的通知。

1.29.2. 方法

使用 `NSNotificationCenter` 中 `default notificationcenter` 的 `postNotificationName:object:userInfo:` 的实例方法发布一条通知，其中携带一个对象（通常此对象激活通知）和一条用户信息词典，词典中包含了关于词条通知和/或者激活通知的对象的额外信息。

1.29.3. 讨论

通知中心是通知对象的分散中心，例如，在用户使用 App 时如果键盘突然随处弹，iOS 会发送一条通知到你的应用，应用中的任何对象会根据词条通知自动跑到错误通知中心成为特殊通知的观察员。一旦对象生命终结它必须自动离开通知中心的发发表。这样一条通知就是成了通过通知中心发给对象的信息。通知中心是 `NSNotificationCenter` 类的一个例子。我们通过 `NSNotificationCenter` 的 `defaultCenter` 类的实例方法获取错误通知中心对象；通知本身就是 `NSNotification` 类的对象。一条通知对象有一个名称（标记为 `NSString`），它携带

2 条重要信息:

你可以不需要使用 API 就给自己的通知命名, 只要确保此条通知命名足够特别到不会与系统信息冲突。

发送对象

发送对象就是携带通知内容. 观察者可以通过 `NSNotification` 类的 `objectinstance` 方法获得对象。

用户信息词典

这个一个选择性词典, 发送对象可以创建并同时发送一条通知。这个词典通常包含关于此条通知的更多信息。例如, when a keyboard is about to get displayed in iOS for any component inside your app, iOS 发送 `UIKeyboardWillShowNotification` 的通知到错误通知中心。这条通知的用户信息词典中包含了一些数值比如在动画前后键盘和动画效果中键盘的数值。当键盘要在显示器上展现, 通过使用这些数据观察者可以决定对那些可能受到阻碍的 UI 组建做些什么事情。通知功能是一个执行去除代码的最好方法。我的意思是通过通知你可以处理完整的条件处理程序和委托。

然而, 关于通知有一个潜在问题: 不会即时发送出去。通知从通知中心发送出去, `NSNotificationCenter` 的执行被应用程序员隐藏, 发布可能会有时会延迟几毫秒或极端情况下会延迟几秒 (这种情况我从未碰到过)。结果就是, 在何处使用或者不使用通知的地点由你自己决定。

为了建立 `NSNotification` 类的通知, 要使用 `NSNotification` 的实例方法 `notificationWithName:object:userInfo:` class, 例如: 最好在通知名称前以 `Notification` 作为后缀。例如, 允许通知名称类似于 `ResultOfAppendingTwoStrings`。然而, 清楚的指出这个名称 `ResultOfAppendingTwoStringsNotification` 属于什么会更好。

我们看个例子, 简单用一个首名称和末名称, 两个追加在一起可以组成一个字符串 (首名称+末名称) 然后使用错误通知中心发布通知。当用户使用我们的 APP 时我们会执行应用程序委派。

```
#import "AppDelegate.h"
@implementation AppDelegate
@synthesize window = _window;
/* The notification name */
constNSString *ResultOfAppendingTwoStringsNotification =
    @"ResultOfAppendingTwoStringsNotification";
/* Keys inside the dictionary that our notification sends */
constNSString
    *ResultOfAppendingTwoStringsFirstStringInfoKey = @"firstString";
constNSString
    *ResultOfAppendingTwoStringsSecondStringInfoKey = @"secondString";
constNSString
    *ResultOfAppendingTwoStringsResultStringInfoKey = @"resultString";
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    NSString *firstName = @"Anthony";
    NSString *lastName = @"Robbins";
    NSString *fullName = [firstName stringByAppendingString:lastName];
    NSArray *objects = [[NSArray alloc] initWithObjects:
        firstName,
        lastName,
        fullName,
        nil];
    NSArray *keys = [[NSArray alloc] initWithObjects:
        ResultOfAppendingTwoStringsFirstStringInfoKey,
        ResultOfAppendingTwoStringsSecondStringInfoKey,
        ResultOfAppendingTwoStringsResultStringInfoKey,
        nil];
    NSDictionary *userInfo = [[NSDictionary alloc] initWithObjects:objects
        forKey:keys];
    NSNotification *notificationObject =
        [NSNotification
        notificationWithName:(NSString *)ResultOfAppendingTwoStringsNotification
        object:self
        userInfo:userInfo];
    [[NSNotificationCenter defaultCenter] postNotification:notificationObject];
}
102 | Chapter 1: The Basics
self.window = [[UIWindow alloc] initWithFrame:
```

```
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

当然，对每一条想要发布的通知没有必要明确对象或者用户信息词典。然而，如果你是和一个开发团队开发相同的应用软件或者在写一个固定程序库，建议你全面归档通知并且清楚说明这条通知是否携带对象和/或用户信息词典。如果是的话，必须说明每条通知携带的对象是什么，其中的用户信息词典中有那些关键信息和数值。

如果不打算发送对象和用户信息词典，则建议使用 `NSBundle` 中 `postNotificationName:object:` 的实例方法。明确指出代表通知名称的字符串作为第一个参数，通知对象应该携带的 `Nil` 为第二个参数

请看举例：

```
#import "AppDelegate.h"
@implementation AppDelegate
@synthesize window = _window;
/* The notification name */
const NSString *NetworkConnectivityWasFoundNotification =
    @"NetworkConnectivityWasFoundNotification";
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [[NSNotificationCenter defaultCenter]
postNotificationName:(NSString *)NetworkConnectivityWasFoundNotification
object:nil];
self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

1. 29. 4. 参考 XXX

1. 30. 收听 NSNotificationCenter 发出的通知

1. 30. 1. 问题

收听使用 `NSNotificationCenter` 不同系统和客户通知发出的通知。

1. 30. 2. 方法

在通知发布之前使用 `NSNotificationCenter` 的 `addObserver:selector:name:object:` 实例方法在通知中心增加观察对象。要停止观察通知，使用 `NSNotificationCenter` 的 `removeObserver:name:object:` 的实例方法，传入你的观察者对象，然后传入的想停止观察的通知名称和最初订阅的对象（这一点会在下面的讨论部分详细解释）。

1. 30. 3. 讨论

任何发布通知的对象和任何相同应用中的对象都能通过有确定名称的通知收听。有相同名称的两条通知可以发布，但是它们必须来自两个不同对象。例如，你有一个从两个类发出的名称为 `DOWNLOAD_COMPLETED` 的通知，一个是从网络下载图片的下载管理器，另一个从与 `iOS` 设备连接的附件下载数据的下载管理器。一个观察者可能只对从一个确定对象发出此名称的通知有兴趣，例如，只关心从附件下载数据的下载管理器。开始收听通知时，使用通知中心的 `addObserver:selector:name:object:` 实例方法中对参数指定源对象（广播）。

此处是关于 `addObserver:selector:name:object: accepts:` 每个参数的简单描述

`addObserver`

接收通知的对象（观察者）。

selector

当通知观察者发布或者接收通知时观察者会调用选择器（方法），这种 `NSNotification` 实例方法将单独讨论。

name

要发布的通知的名称

object

有选择的明确发布通知的源代码，如果参数为 `Nil`，在不考虑发布对象的情况下观察者将接收有确定名称的通知；如果已经设置了参数，那么只有那些有了由指定对象明确命名的通知被观察到。

在 1.29 我们学习了如何发布通知。现在试着观察信息我们已经学过的发布到此处的通知。

```
#import "AppDelegate.h"
@implementation AppDelegate
@synthesize window = _window;
/* The notification name */
constNSString *ResultOfAppendingTwoStringsNotification =
    @"ResultOfAppendingTwoStringsNotification";
/* Keys inside the dictionary that our notification sends */
constNSString
    *ResultOfAppendingTwoStringsFirstStringInfoKey = @"firstString";
constNSString
    *ResultOfAppendingTwoStringsSecondStringInfoKey = @"secondString";
constNSString
    *ResultOfAppendingTwoStringsResultStringInfoKey = @"resultString";
- (void) broadcastNotification{
    NSString *firstName = @"Anthony";
    NSString *lastName = @"Robbins";
    NSString *fullName = [firstName stringByAppendingString:lastName];
    NSArray *objects = [[NSArrayalloc] initWithObjects:
        firstName,
        lastName,
        fullName,
        nil];
    NSArray *keys = [[NSArrayalloc] initWithObjects:
        ResultOfAppendingTwoStringsFirstStringInfoKey,
        ResultOfAppendingTwoStringsSecondStringInfoKey,
        ResultOfAppendingTwoStringsResultStringInfoKey,
        nil];
    NSDictionary *userInfo = [[NSDictionaryalloc] initWithObjects:objects
        forKeys:keys];
    NSNotification *notificationObject =
        [NSNotification
        notificationWithName:(NSString *)ResultOfAppendingTwoStringsNotification
        object:self
        userInfo:userInfo];
    [[NSNotificationCenterdefaultCenter] postNotification:notificationObject];
}
- (void) appendingIsFinished:(NSNotification *)paramNotification{
    NSLog(@"Notification is received.");
    NSLog(@"Notification Object = %@", [paramNotification object]);
    NSLog(@"Notification User-Info Dict = %@", [paramNotificationuserInfo]);
}
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    /* Listen for the notification */
    [[NSNotificationCenterdefaultCenter]
    addObserver:self
    selector:@selector(appendingIsFinished:)
    name:(NSString *)ResultOfAppendingTwoStringsNotification
    object:self];
    [selfbroadcastNotification];
}
```

```
self.window = [[UIWindow alloc] initWithFrame:
                [[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
- (void)applicationWillTerminate:(UIApplication *)application{
    /* We no longer observe ANY notifications */
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

在运行这个应用的时候，会在控制窗口看到和下文类似的情况：

```
Notification is received.
Notification Object = <AppDelegate: 0x7408490>
Notification User-Info Dict = {
    firstString = Anthony;
    resultString = AnthonyRobbins;
    secondString = Robbins;
}
```

如果你想在观察过程中从正在删除的对象中指定通知，简单调用通知中心的 `removeObserver:name:object:` 实例方法，指定取消订阅的通知名称以及（可选择）发送通知的对象。

你已经看到了，我们使用了通知中心的 `removeObserver:` 实例方法把对象转换成所有通知的观察者。从观察者链上删除对象有不同方法；可以马上停止，和我们此处的做法一样——也就是观察通时完全转换对象，或者在 APP 的生命周期内在特定时间从观察确定通知中删除对象。

1. 30. 4. 参考

XXX



DEVDIV

点击这里访问：DevDiv.com 移动开发论坛