Programming withObjective-C

极客学院出版

前言

Objective-C 是你在为 OS X 和 IOS 系统编写应用程序时使用的主要编程语言。它是C语言的超集并具备面对对象的能力和动态运行的特性。 Objective-C 继承了 C 语言的语法，基本类型和控制流语句并且添加了定义类和方法的语法。并且 Objective-C 语言在提供了动态类型和延迟到运行时的绑定的同时，为对象图形管理和对象字面提供了语言层面上的支持。

本教程翻译自 Apple 官方发行的 Programming with Objective-C (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC)，为 iOS 开发者提供权威的编程指导。

	更新日期	更新内容
	2014-04-03	第一版发布

目录

1

关于 Objective-C

Objective-C 是你在为 OS X 和 IOS 系统编写应用程序时使用的主要编程语言。它是C语言的超集并具备面对对象的能力和动态运行的特性。 Objective-C 继承了 C 语言的语法，基本类型和控制流语句并且添加了定义类和方法的语法。并且 Objective-C 语言在提供了动态类型和延迟到运行时的绑定的同时，为对象图形管理和对象字面提供了语言层面上的支持。

概览

本文介绍了 Objective-C 语言并且就其使用提供了广泛的例子。你将会学习到如何创造你自己描述自定义对象的类别以及如何使用由 Cocoa 和 Cocoa Touch 所提供的框架类。尽管框架类是与语言本身所分开的，但是他们的使用与处理和 Objective-C 语言紧密相关，并且许多其他语言层面的特点依靠这些类别所提供的行为。

一个应用程序由一个网络对象创建

在为 OS X 或者 IOS 系统创建应用程序时，你会将大把时间花费在使用对象上。那些对象是 Objective-C 类的实体。其中的一些是由 Cocoa 或者 Cocoa Touch 提供的而其他的一些是由你自己编写的。

如果你在编写你自己的类，开始的时候先提供一个对于类的描述，要细化预期的公开接口到类的实体。这个公共接口的公共特性就是来封装相关数据以及一系列的方法。方法声明表明一个对象可以接受到的信息，也包括任何时候方法被调用时所需要的参数的信息。你也会提供一个类的实现，包括实现在接口声明的每一个方法的代码。

相关章节：类别定义 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/DefiningClasses/DefiningClasses.html#//apple_ref/doc/uid/TP40011210-CH3-SW1)，对象使用 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW1)，数据封存 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple_ref/doc/uid/TP40011210-CH5-SW1)

类的继承

去定义一个类在已有类上增加自定义行为，而不是创造一个全新的类来提供微小的附加能力。你可以用一个类为任何类添加方法，包括那些你没有初始实现源代码的类，比如框架类 `NSString` 。

如果你的确有初始源代码，你可以使用一个类的继承来增加新的特性，或者修饰已有特性的属性。类扩展被广泛用于隐藏私人行为在一个单个源代码文件中使用，或者在一个自定义框架的私有实现中使用。

相关章节： 现有类的定制 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html#//apple_ref/doc/uid/TP40011210-CH6-SW1)

协议定义消息规范

在 Objective-C 的应用程序中大部分的工作是由对象之间互相传递信息形成的。通常，这些信息是由在一个类接口中明确声明的方法来定义的。然而某些时候定义非直接相关一个特定类的一系列方法是有用处的。

Objective-C 使用协议来定义一组相关方法，例如一个对象请求代理的方法，是可选还是需要的。任何类可以表明它采用了协议，也就是说它必须提供所需要的所有方法的实现。

相关章节：使用协议 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html#//apple_ref/doc/uid/TP40011210-CH11-SW1)

值和集合通常作为 Objective-C 的对象

在 Objective-C 中使用 Cocoa 和 Cocoa Touch 类来声明值是很常见的。 `NSString` 类用于字符串的字符， `NSNumber` 类用于不同类型的数字，例如整数或浮点数，`NSValue` 类用于等其他值，例如 C 中的结构。你也可以使用任何由 C 语言定义的初始类型，例如 `int` ，`float` 或者 `char` 。

集合经常代表某一集合类的实体，例如 `NSArray` ，`NSSet` , 或者 `NSDictionary` ，这其中的每一个都被用于集合其他 Objective-C 对象。

相关章节：值和集合 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/FoundationTypesandCollections/FoundationTypesandCollections.html#//apple_ref/doc/uid/TP40011210-CH7-SW1)

Blocks 简化常规任务

Blocks 是引进到 C ，Objective-C 和 C++ 语言的一种语言功能，来代表工作的一个单元；它们把一块代码以捕获的状态封锁，这就使它们在其他程序语言中类似于关闭状态。Blocks常常被用于简化常规任务,例如集合枚举，分类和测试。同样它们使用多线程优化技术（ GCD ）也使得并发或异步执行规划任务更加简单。

相关章节：使用Blocks (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html#//apple_ref/doc/uid/TP40011210-CH8-SW1)

错误对象在运行时错误问题中的使用

尽管 Objective-C 包括异常处理的语法，仅在程序错误（如界外数组访问）时使用的 Cocoa 和 Cocoa Touch 出现异常，这些都需要在一个应用程序运行前确定。

尽管其他的错误—包括运行时错误,例如硬盘空间不足或者网络功能不可用，都由 `NSError` 类来体现。你的应用程序要为错误做好计划并且确定如何做到最佳处理，以达到在问题出现时提供尽可能最佳的用户体验。

相关章节：错误处理 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/ErrorHandling/ErrorHandling.html#//apple_ref/doc/uid/TP40011210-CH9-SW1)

Objective-C 代码遵守已有的规则

写 Objective-C 代码的时候，你应该记住大量已有的代码规则。例如，方法名称以小写字母开始，用大小写混合的方式来区分多个单词，就像 `doSomething` 或者 `doSomethingElse` 而且，重要的不仅仅是要关注大写，你还应该确定你的代码尽可能的能被读懂。也就是说，方法名称要能够传词达意，而不是特别冗长。 除此以外，如果你想利用语言或者结构特征，你还需要注意一些规则。例如，属性访问方法必须严格遵守命名规则，以便能够和 Key-Value Coding(KVC) 或者 Key-Value Observing (KVO)这些技术共同使用。

相关章节：规则 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html#//apple_ref/doc/uid/TP40011210-CH10-SW1)

先决条件

如果你对 OS X 或者 iOS 的发展还不熟悉，你可以在读这分文档前阅读 Start Developing Mac Apps Today 或者 Start Developing Mac Apps Today (https://developer.apple.com/library/mac/referencelibrary/GettingStarted/RoadMapOSX/index.html#//apple_ref/doc/uid/TP40012262)，以便对 iOS 和 OS X 的应用发展过程有一个大概的了解。除此以外，在继续这篇文档大部分章节后面的练习前，先熟悉 Xcode 。 Xcode 是一种用来给 iOS 和 OS X 建立应用程序的集成开发环境。你将用它编写代码，设计应用程序的用户界面，测试你的应用软件，以及调试任何问题。 虽然有一些与 C 语言或者基于 C 语言的一种语言会更好，比如：Java 或者 C#，这份文档包括基本 C 语言特征的内联例子，比如：流控制声明。如果你还掌握其他更高级的程序语言，比如 R

uby 或者 Python，你应该能够理解这些内容。 合理的通用性被用于一般面向对象编程原则，尤其是当它们用于 Objective-C 环境中。但是它假定你至少有与基本面向对象概念最小的相似点。如果你不熟悉这些概念，你应该读读 Concepts in Objective-C Programming (https://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010810) 中的相关章节。

相关文献

这份文档中的内容使用 Xcode 4.4 以及更高版本，并且假定你的目标是 OS X v10.7 及更高版本，或者 iOS 5 及更高版本。更多有关Xcode的信息，请关注 Xcode Overview (https://developer.apple.com/library/mac/documentation/ToolsLanguages/Conceptual/Xcode_Overview/index.html#//apple_ref/doc/uid/TP40010215) 。有关语言特征可利用性，请关注 Objective-C Feature Availability Index (https://developer.apple.com/library/mac/releasenotes/ObjectiveC/ObjCAvailabilityIndex/index.html#//apple_ref/doc/uid/TP40012243) 。

Objective-C 应用程序使用引用计数来决定对象的使用寿命。大多情况下，编译程序的自动引用计数特征会为你注意这个问题。如果你不能利用 ARC ,或者需要转换或保留手动管理对象记忆的遗留代码，你应该阅读 Advanced Memory Management Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i) 。

除了编译程序，Objective-C 语言使用一种 runtime system 来保证它的动态和面向对象特征。虽然你通常不需要担心 Objective-C 怎样“工作”，直接和这种 runtime system 互动是可以实现的。就像 Objective-C Runtime Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008048) 和 Objective-C Runtime Reference (https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/doc/uid/TP40001418) 里描述的一样。

2

定义类 – Defining Classes

当你为 OS X 或 iOS 系统编写软件时，你的大部分时间都花在了对象上。Objective-C 中的对象就像其他面向对象编程语言中的对象一样：它们把数据与相关的行为打包。

一个应用程序构成了一个巨大的联通对象的生态系统，他们之间相互通信以解决具体问题，如显示一个可视化界面、响应用户的输入、或存储信息。 OS X 或 iOS 开发中，您不需要从头开始创建对象来解决每个问题；Cocoa (for OS X) 和 Cocoa Touch (for iOS) 为你提供一个包含可供你使用的现有对象库。

其中一些对象是立即可用的，比如字符串和数字等基本数据类型，或像按钮和表视图这样的用户界面元素。一些是专为你以你需要的方式定制代码。软件开发过程涉及决定如何最好地定制和合并底层框架提供的对象和自己的对象，让你的应用具有其独特的特性和功能。

在面向对象的编程术语中，对象类的实例。本章示范了如何在 Objective - C 中通过声明一个用来描述你打算使用的类及其实例的接口来定义类。这个接口包含此类可以接收到的消息列表，所以你还需要提供类的实现，其中包含要执行的代码以响应每个消息。

类是对象的蓝图 – Classes Are Blueprints for Objects

类描述了特定类型的对象的常见行为和属性。字符串对象(在 Objective-C 中，这是类 NSString 的一个实例)，该类提供了各种方法来检查和转换内部的字符。同样，此类过去常用于描述一个数字对象(NSNumber)提供围绕内部的数值，如将该值转换为不同数字类型的功能。

同样的，多个由相同的蓝图构成的建筑在结构上是相同的，类的每个实例共享相同的属性和行为来作为该类的所有其他实例。每个 NSString 实例也一样，无论它的内部字符串如何。

任何特定的对象是用于以特定的方式而设计的。你可能知道一个字符串对象表示某些字符的字符串，但你不需要知道确切的内部机制，用来存储这些字符。你不懂的内部行为对象本身用于直接处理的特点，但你要知道如何你预期的交互作用与对象，也许是为了索取特定字符或请求一个新的对象，在其中所有原始字符转换为大写。

在 Objective-C 中，类接口指定一个给定的类型的对象是如何被其他对象使用的。换句话说，它定义了类的实例与外部世界之间的公共接口。

可变性决定了一个代表值能否被取代 – Mutability Determines Whether a Represented Value Can Be Changed

一些类定义对象是不可变的。这意味着当一个对象创建，并且随后不能由其他对象更改时，必须设置内部内容。在 Objective-C 中，所有基本的 NSString 和 NSNumber 对象都是不可变的。如果您需要代表一个不同的数字，则必须使用一个新的 NSNumber 实例。

不可变的类还提供了一个可变的版本。如果您一定需要更改在运行时的字符串内容，例如他们在收到通过网络连接时填加字符，您可以使用 NSMutableString 类的一个例子。这个例子像 NSString 对象一样，除此之外他们还提供更改此对象表示的字符。

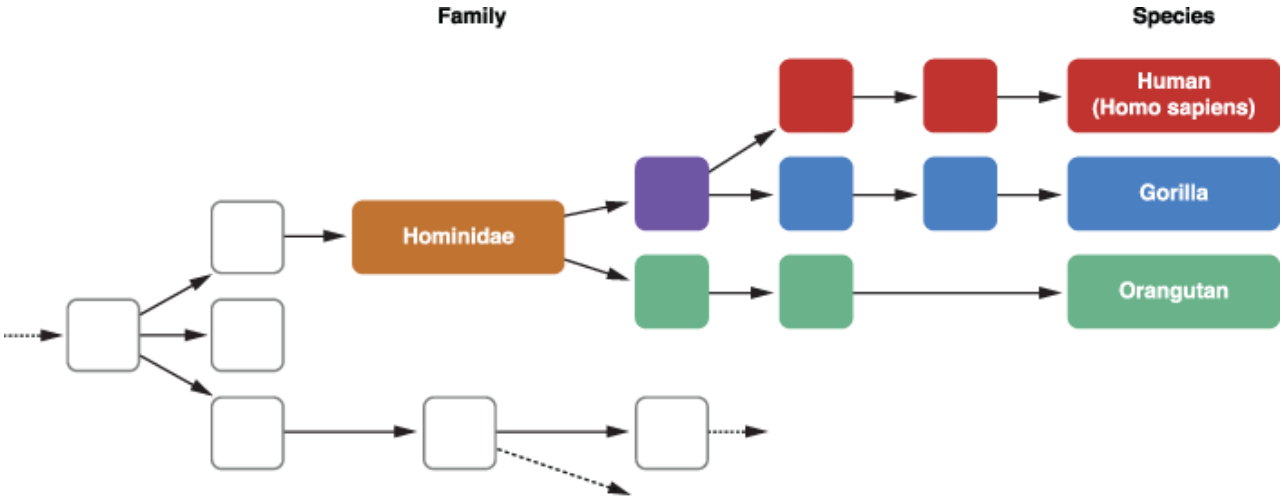
虽然 NSString 和 NSMutableString 是不同类，但他们有许多相似之处。你并不需要从零开始的编写两个独立类，只是有时候碰巧有一些类似的行为要写两个完全独立的类，它完全可以利用现有的东西完成。

类的继承 – Classes Inherit from Other Classes

在自然的世界里，将动物分为像种、 属、族这样的各个门类。这些群体是分层的，许多的物种可能都属于同一属，许多的属可能同是一个族。

举个例子，猩猩、狒狒和人类有明显的相似性。虽然他们都属于不同的物种，和甚至不同的属，部落和亚科，但是它们分类学相关，因为它们都属于同一个族（称为"人科"），如图 1–1 所示。

图 1–1 物种之间的分类关系



图片 2.1 物种之间的分类关系

在全世界的面向对象的编程中，对象则是也分成为各层次的组。对象被简单的分成许多类，并不会为了像 genus、species 这样不同的层次使用不一样的术语。同样的，人类作为人科家族成员继承着某些特征，那么就可以在从父母类功能性的传承中建立某一类。

当一个类从另一个类有所继承时，新的类就会继承旧类的全部行为和属性。它也有机会来定义它自己的其他的行为和属性，或也可以重写继承来的行为属性。

在 Objective–C 字符串类的情况下， NSMutableString 描述指定的类继承于 NSString，如图 1–2 所示。所有由 NSString 提供的功能都是可用于 NSMutableString 的，如查询特定的字符或要求更改新的大写字符串，但 NSMutableString 添加了可以让您附加、插入、替换或删除子字符串和单个字符的方式。

图 1-2 NSMutableString 的继承

图片 2.2 NSMutableString 的继承

The Root Class提供基本的功能 – The Root Class Provides Base Functionality

所有生物体都拥有着一些基本的 "life" 特点，有些功能对于 Objective-C 中的对象是很常见的。

当 Objective-C 对象需要使用另一个类的一个实例时，它需要其他类提供一定的基本特征和行为。为此，Objective-C 定义根类从中绝大多数的其他类继承，称为 NSObject 。当一个对象遇到另一个对象时，它将至少能够使用 NSObject 类描述所定义的基本行为进行交互。

当您定义您自己的类时，你应该至少从 NSObject 中继承。一般情况下，你应该找一个提供你所需最接近的功能的 Cocoa or Cocoa Touch 对象，然后从其中继承。

如果您想要在 iOS 应用程序中使用自定义的按钮，提供的 UIButton 类不能提供足够可自定义的属性以满足您的需要，从 NSObject 中创建一个新类比从 UIButton 中创建更有意义。如果你只是从 NSObject 简单的继承，你将需要由 UIButton 类定义的所有复杂的视觉交互和通信，这只是为了让你的行为方式按照用户的期望的按钮定义来实现。此外，通过从 UIButton 继承，你的子类会自动地获得任何未来的功能增强或可能应用于内部 UIButton 行为的 bug 修复。

UIButton 类本身定义继承 UIControl ，描述了在 iOS 上所有用户界面控件的常见基本行为。反过来， UIControl 类继承 UIView ，给在屏幕显示的对象提供常用功能。UIView 继承于 UIResponder，允许它响应用户输入，如水龙头、 手势。最后，也是最重要的，UIResponder 继承 NSObject，图 1-3 所示。

图 1-3 UIButton 类的继承

图片 2.3 UIButton 类的继承

这一连串的继承是指 UIButton 任何自定义子类将不仅是继承声明了 UIButton 本身的功能，也依次从每个超类继承功能。你会最终以一个像按钮的对象结束，它可以在屏幕上自我显示、 对用户输入作出响应以及与所有基本的 Cocoa Touch 对象进行通信。

对于你需要使用的类，要把它的继承链牢记于心，以便它可以把它所有能做的做好。类参考文档提供的 Cocoa and Cocoa Touch 允许从任何类可以轻松导航到每个超类。如果你在一个类接口或引用中找不到你需要的，它可能很好的在进一步的超类中定义了。

类接口定义预期交互 – The Interface for a Class Defines Expected Interactions

面向对象编程的诸多好处之一就是前面提到的想法 — 一要使用一个类，那么你所需要知道的就是如何与它的实例进行交互。更具体地说，应该设计一个让对象隐藏其内部实现的细节。

如果你在 iOS 应用程序中使用标准的 UIButton ，你不需要担心像素在屏幕上如何操纵按钮来显示。你需要知道的就是您可以更改某些属性，比如按钮的标题和颜色，并当你将它添加到您可视化界面时表示信任，它将会以你期望的方式正确显示。

当您定义您自己的类时，您需要先搞清楚这些类的公共属性和行为。你想访问哪些公开属性？你应该让这些属性改变吗？其他对象与您的类的实例如何沟通？

此信息进入您的类的接口 — — 它定义了你打算与你的类的实例中的其他对象进行交互的方式。公共界面由您的类的内部行为进行独立描述，它组成了类。在 Objective-C 中，接口和安装通常放置在独立的文件中，这样你只需要使接口开放。

基本语法 – Basic Syntax

在 Objective-C 中描述一个类接口的语法如下所示：

```
@interface SimpleClass : NSObject

@end
```

本例声明了一个从 NSObject 中 继承来的名为 SimpleClass 的类。

@interface 声明内部定义的公共属性和行为。在此示例中，没有指定属性或行为超出超类，所以唯一的可在 SimpleClass 上实现的实例预计功能是从 NSObject 继承来的。

属性控制对对象值的访问

对象通常具有属性供公众查阅。例如，如果您在一款记录保存软件中定义了一个类来表示人，那你可能需要一个决定字符串来表示一个人的姓和名的属性。

这些属性的声明应添加内部接口，像这样：

```
@interface Person : NSObject

@property NSString *firstName;
@property NSString *lastName;

@end
```

在此示例中，Person 类声明了两个公共属性，这两个属性是 NSString 类的实例。

这两个属性都是对 Objective-C 的对象而言的，所以他们使用星号以表明它们是 C 指针。他们也像在 C 语言中其他变量的声明语句一样，需要以分号结尾。

您可能会决定添加一个属性来表示一个人出生日期，以便可以按年龄给人分组，而不是只按名称进行排序。你可以对一个数字对象使用这样的属性：

```
@property NSNumber *yearOfBirth;
```

但这只是为了存储一个简单的数字值，可能算是矫枉过正。那么可以使用另一种由 C 语言提供替代方法，持一个整数标量值：

```
@property int yearOfBirth;
```

属性的特性表明数据可存取性和存储的注意事项 – Property Attributes Indicate Data Accessibility and Storage Considerations

这个示例展示了到目前为止所有为了完善公共访问所声明的属性。这意味着其他对象可以同时读取和更改属性的值。

在某些情况下，您可能希望声明一个属性不能被改变。在现实世界中，一个人必须填写大量的文书工作来更改其记录的第一个或最后一个名称。如果你正在写官方记录的应用程序，您可以选择指定一个人名字的公共属性为只读，任何更改都需要通过中介对象来负责验证请求并选择通过或拒绝它。

Objective-C 属性声明可以包含属性的特性，用于指示是否一个属性应设置为只读模式。在官方记录应用程序中，人名的类接口可能如下所示：

```
@interface Person : NSObject
@property (readonly) NSString *firstName;
@property (readonly) NSString *lastName;
@end
```

属性的特性在括号内的 @property 关键字后指定，完整地在声明的公共属性公开的数据中描述。

方法声明对象可接收消息 – Method Declarations Indicate the Messages an Object Can Receive

到目前为止的例子涉及到描述一个典型的模型对象或主要用于封装数据的对象的类。在 Person 类中，很可能有不需要能够访问这两个声明属性之外的任何功能。然而，大部分的类包括除了定义属性以外的行为。

既然 Objective-C 软件是由对象的大型网络所组成，标记那些能够通过发送信息相互作用的对象就非常重要了。在 Objective-C 术语中，一个对象发送消息到另一个对象是通过对该对象调用的方法实现的。

尽管语法有很大不同， Objective-C 在概念上还是类似于 C 和其他编程语言中的标准函数的。C 函数声明如下所示：

```
void SomeFunction();
```

与之相同的用 Objective-C 来定义如下所示：

```
- (void)someMethod;
```

在这种情况下，该方法不具有参数。Void 在声明之初写在小括号中来表示这个方式是不会再程序结束时返回任何值的。

在方法名称前面的减号 (-) 表示它是实例方法，可以在类的任何实例调用。这将它从类的方式中区别出来，可以调用类本身，Objective-C Classes Are also Objects (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/DefiningClasses/DefiningClasses.html#//apple_ref/doc/uid/TP40011210-CH3-SW18) . 中有所叙述。

正如 C 函数的原型，一个在 Objective-C 类的接口声明就像其他的 C 语句一样，需要分号终止。

Methods 可以使用参数 – Methods Can Take Parameters

如果您需要声明一个方法来带一个或多个参数，语法则是与一个典型的 C 函数非常不同的。 对于 C 函数的参数被指定在括号里，像这样：

```
void SomeFunction(SomeType value);
```

Objective-C 方式声明包括参数的名字，这里用到冒号，像这样：

```
- (void)someMethodWithValue:(SomeType)value;
```

正如返回类型，参数的类型被指定在括号内，就像标准的 C 类型转换一样。 如果您需要提供多个参数，语法又迥异于 C。将这些 C 函数参数依旧是指定在括号内，用逗号分开来彼此分开 ； 在 Objective-C 中，方式声明像这样采取了两种参数：

```
- (void)someMethodWithFirstValue:(SomeType)value1 secondValue:(AnotherType)value2;
```

在此示例中， value1 和 value2 是用在当方式被调用时实现访问值时的名字，它们就像变量。

一些编程语言允许函数用所谓的命名参数来定义； 要特别注意这不是 Objective-C。 方式中的变量被调用的顺序必须与方式的声明相匹配，事实上 secondValue： 方法声明的一部分是方法的名称：

```
someMethodWithFirstValue:secondValue:
```

这是一个有助于使 Objective-C 更加可读的语言，因为通过方法调用传递的值是指定的内联函数 ，旁边的方法名称的相关部分在You Can Pass Objects for Method Parameters (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW13) 中有所描述。

> 注： 上面使用的 value1 和 value2 值的名称不是严格意义的方法声明，这意味着它是不需要像你在执行中要使用完全相同值作为名称一部分。它唯一的要求是签名要相匹配，这意味着您必须保留该方法的参数名称，并保证返回类型完全相同。 下面的例子，这种方法具有相同的签名，就如上面所示：

```
- (void)someMethodWithFirstValue:(SomeType)info1 secondValue:(AnotherType)info2;
- (void)someMethodWithFirstValue:(SomeType)info1 anotherValue:(AnotherType)info2;
- (void)someMethodWithFirstValue:(SomeType)info1 secondValue:(YetAnotherType)info2;
```

类的名称一定要与众不同 – Class Names Must Be Unique

请特别注意在同一个应用程序中每个类的名称必须是唯一的，甚至跨越包括的库或框架也要这样。如果你试图用一个项目中的现有类相同的名称创建一个新类，您会收到一个编译器错误。

出于这个原因，也建议您使用三个或更多字母定义任何类的名称的前缀。这些字母可能涉及到您目前正在编写的应用程序，或者是可重复利用的代码框架名字。 在本文档的其余部分给出的所有例子都使用类名前缀，像这样：

```
@interface XYZPerson : NSObject
@property (readonly) NSString *firstName;
@property (readonly) NSString *lastName;
@end
```

历史注释： 如果你想知道为什么所以你遇到的许多类有一个 NS 前缀，其实因为它是过去的 Cocoa and Cocoa Touch 历史。Cocoa 开始生命的收集的框架，用于构建 NeXTStep 操作系统的应用程序。当苹果公司在 1996 年买下一回时下， 一步步骤大量被纳入OS X 上，包括现有的类名。介绍了可可触摸 iOS 等同于可可豆； 一些类，可在可可粉和可可触摸，虽然也有大量的类独有的每个平台。

两个字母前缀像 NS 和 UI（ 对于在 iOS 用户的界面元素 ）被苹果公司所使用。 与此相反的是，方法和属性的名称，只需在定义它们的类内唯一。虽然每个应用程序中的 C 函数必须具有唯一的名称，但在许多Object-C 中用

相同名字定义类是完全可以接受的（甚至是这样更好）。然而，你不能在相同的类声明中不止一次的定义同一个方法，虽然你想要重写从父类继承的方法，但您必须使用原始声明中使用的确切名称。

作为与方法，对象的属性和实例变量（多数属性受配于实例变量 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple_ref/doc/uid/TP40011210-CH5-SW6) 有所描述）需要在他们定义的类中是唯一的。但是，如果要使用的全局变量，这些必须在应用程序或项目内唯一名称。 更多的命名规定和建议详见 Conventions (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html#//apple_ref/doc/uid/TP40011210-CH10-SW1) 。

类的实现规定其内部的行为 – The Implementation of a Class Provides Its Internal Behavior

一旦你定义了一个类，其中包括的属性和方法都是供公众查阅的，您需要编写代码来实现的类行为。

如前所述，该接口的类通常放置在一个专用的文件中，通常被称为一个头文件，它一般都有文件扩展名 .h。写 Objective-C 类里面一个源代码文件以扩展名执行 . m。

每当在头文件中定义的接口时，你需要告诉编译器在试图编译的源代码文件中执行前请先阅读。为此，Objective-C 提供预处理器指令，#import。它类似于 C #include 指令，但可以确保文件是只包括在编译过程中一次。请注意预处理器指令有别于传统的 C 语句，不使用分号终止。

基本语法 – Basic Syntax

为一个类提供实现的基本语法如下所示：

```
#import "XYZPerson.h"

@implementation XYZPerson

@end
```

如果你声明一个类接口中的方式，您需要在这个文件内执行他们。

执行方法 – Implementing Methods

用这种方法，像这样的简单的类接口如下：

```
@interface XYZPerson : NSObject
- (void)sayHello;
@end
```

实现过程就像这样：

```
#import "XYZPerson.h"

@implementation XYZPerson
- (void)sayHello {
  NSLog(@"Hello， World!");
}
@end
```

本示例使用 NSLog() 函数将消息记录到控制台。它类似于标准的 C 库中的 printf () 函数，并采用可变数目的参数，其中第一个必须是 Objective-C 字符串。 方法实现类似于 C 函数的定义，相关的代码必须被大括号所包在内。此外，方法的名称必须与它的原型和参数和返回类型完全匹配。

Objective-C 从C语言中继承属性区分大小写 ，所以此方法：

```
- (void)sayhello {
}
```

会被视为由编译器作为完全不同于前面所示的 sayHello 方法。 一般情况下，方法名称应以小写字母开头。Objective-C 要求使用比你可能看到用于C中的函数更具描述性的名称。如果方法名称涉及到多个单词，请使用棕色大小写（每个新单词的首字母大写），使它们易于阅读。 请注意在 Objective-C 中那空白也是灵活的。习惯在代码中使用制表符或空格，或者各个块内每一行的缩进，你会经常看到左大括号位于单独的一行，像这样：

```
- (void)sayHello
{
  NSLog(@"Hello， World!");
}
```

Xcode，苹果公司的集成开发环境 (IDE) 用于创建 OS X 和 iOS 软件，它将基于一组自定义的用户首选项自动缩进代码。请参见更改缩进和制表符宽度Xcode Workspace Guide。 在下一章中你会看到更多例子的方法实现，Working with Objects (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW1) 。

Objective-C类也是对象 – Objective-C Classes Are also Objects

在Objective-C 中，类本身不透明的类型被称为类的对象。类不能具有属性以使用上文所示的实例，但他们的声明语法定义可以接收消息。 类方法的典型用途是作为工厂模式，它替代对象分配和初始化步骤在对象创建中有所描述。例如，NSString 类，有各种各样的工厂方法可用于创建一个空的字符串对象或使用特定的字符，包括初始化的字符串对象：

```
+ (id)string;
+ (id)stringWithString:(NSString *)aString;
+ (id)stringWithFormat:(NSString *)format， …;
+ (id)stringWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:(NSError **)error;
+ (id)stringWithCString:(const char *)cString encoding:(NSStringEncoding)enc;
```

如这些示例中所示，类方法通过使用 + 表示，这让他们从使用实例方法的标志中区别开来。 类方法原型可能包含在类接口中，就像实例方法原型。类方法实现与 @implementation 块的类的实例方法方式相同。

练习 – Exercises

> 注： 为了跟着每章结尾给出的练习，您可能也希望创建 Xcode 项目。这将让您确保您的代码编译没有错误。

使用 Xcode 的新项目模板窗口从可用的 OS X 应用程序项目模板创建一个命令行工具。出现提示时，指定为基础的项目类型。

1. 使用 Xcode 的新文件模板窗口创建名为的接口和实现文件。其中 XYZPerson 继承于 NSObject。
2. 向 XYZPerson 类接口添加属性为一个人的名字、 姓氏和 出生日期的类。（ 日期由 NSDate 类描述 ）的出生日期。
3. 声明 sayHello 的方法，并像前文所述那样实现它。
4. 添加一个声明为类称为"person"的工厂模式。不会不要紧，读读下一章!

> 注意：
> 如果您要编译代码，你可能会由于此缺少的实现而出现 "Incomplete implementation" 警告。

3

使用对象 – Working with Objects

在一个 Objective-C 应用中大部分的工作是由于消息跨越对象的生态系统被送回和送达而产生的。这些对象中的一部分是由 Cocoa 或者 Cocoa Touch 所提供的类的实例，一部分是你自己的类的实例。

前一章描述了来为一个类定义接口和实现的语法，其中也包括了实现包含为响应消息而需要被执行的代码的方法的语法。这一章解释了如何给一个对象发送这样一条消息，并涵盖了 Objective-C 的一些动态特征，包括动态类型和决定哪个方法应当在运行时被调用的能力。

在一个对象能够被使用前，它必须结合为它的属性所做的内存分配和内部值的初始化以被正确地创建。这一章描述了如何嵌套方法调用来分配和初始化一个对象来保证它是被正确配置的。

对象发送并且接收消息

尽管在 Objective-C 的对象间中有多种不同的方法来发送消息，到目前为止最普遍的方法是使用方括号的基础语法，像这样：

```
[someObject doSomething];
```

左边的引用，在这个案例中的 `someObject`，是消息的接收者。右边的消息，`doSomething`，是访问接收者的方法的名称。换句话说，当上面这行代码被执行的时候，`someObject` 将被发送消息 `doSomething`。

前一章描述了如何为一个类创建一个借口，像这样：

```
@implementation XYZPerson : NSObject
- (void)sayHello;
@end
```

和怎样为创建那个类的实现，像这样：

```
@implementation XYZPerson
- (void)sayHello {
    NSLog(@"Hello, world!");
}
@end
```

注解：这个例子使用了一个 Objective-C 的字符串字面量，`@“Hello, world!”`。字符串是在 Objective-C 中几个允许为它们的创建而使用的速记文字语法的其中之一。规定 `@“Hello, world!”` 概念上等同于“An Objective-C string object that represents the string Hello, world!.”

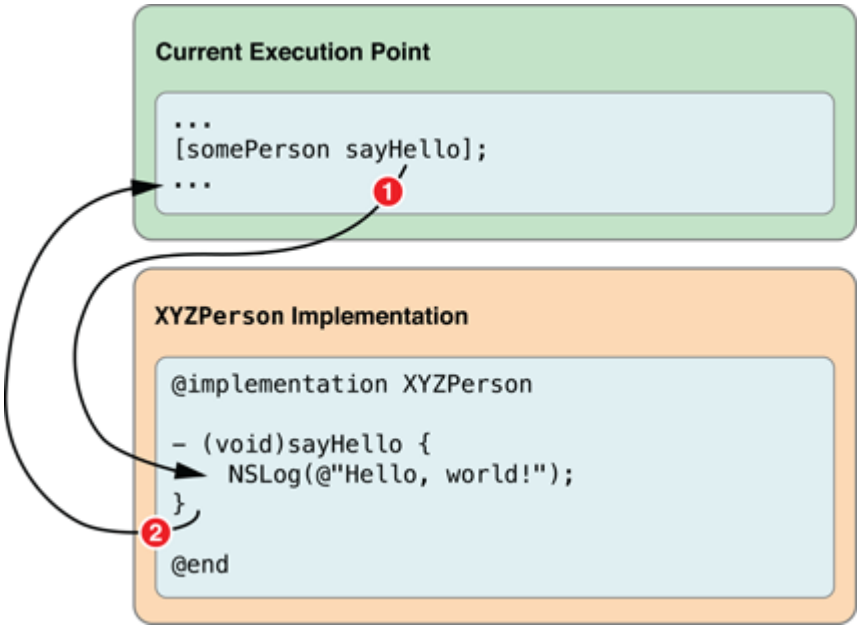
字面量和对象创建将在这一章之后的对象是被动态创建的 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW7) 中被进一步地解释。

假设你已经得到了一个 `XYZPerson` 的对象，你可以像这样给它发送 `sayHello` 的消息：

```
[somePerson sayHello];
```

发送一条 Objective-C 的消息概念上非常像调用一个函数。图2-1 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW4) 为消息 “sayHello” 展示了实际的程序流程。

图 2-1 基本的消息程序流程



图片 3.1 2-1

为了指定一条消息的接收者，理解指针在 Objective-C 中怎样被用来指向对象是非常重要的。

使用指针来跟踪对象

C 和 Objective-C 使用变量来跟踪数值，就像其他编程语言一样。

在标准C语言中有许多基本的标量变量被定义，包括整数型，浮点数型和字符型，它们像这样被声明和赋值：

```
int someInteger = 42;
float someFloatingPointNumber = 3.14f;
```

局部变量，是在一个方法或函数中被声明的变量，像这样：

```
- (void)myMethod {
    int someInteger = 42;
}
```

被限定在方法中被定义的作用域中。

在这个例子中，`someInteger` 在 `myMethod` 中被声明为一个局部变量。一旦执行到方法的闭合括号，`someInteger` 将不再可存取。当一个局部的标量变量（像一个 int 或者一个 float）消失，数值也将消失。

Objective-C对象，相比之下，则分配得稍许不同。对象通常比方法调用的简单作用域生命更长。特别的，一个对象经常需要比那些被创建来跟踪它的原始变量要存活更久的时间。因此，一个对象的存储空间是动态地被分配和解除分配的。

注解：如果你习惯于使用栈和堆，则当对象在堆上被分配时，一个局部变量是在栈上被分配的。

这需要你使用 C语言中的指针（指向存储空间的地址）来追踪它们在存储空间中的位置，像这样：

```
- (void)myMethod {
    NSString *myString = // get a string from somewhere...
    [...]
}
```

尽管指针变量 `myString`（星号指示它的指针）的作用域受 `myMethod` 作用域的限制，它在存储空间中所指的实际的字符串对象在作用域外可能会有更长的生存时间。举例来说，它可能已经存在，或者你可能需要在其他的方法调用中传递对象。

你可以给方法参数传递对象

如果你在发送一条消息时需要传递一个对象，你为方法参数中的一个提供一个对象指针。前一章描述了声明只有一个参数的方法的语法：

```
- (void)someMethodWithValue:(SomeType)value;
```

因此声明一个带有字符串对象的方法的语法，看起来是像这样的:

```
- (void)saySomething:(NSString *)greeting;
```

你可以像这样实现 `saySomething:` 的方法：

```
- (void)saySomething:(NSString *)greeting {
    NSLog(@"%@", greeting);
}
```

指针 `greeting` 表现得像一个局部变量并且被限制在 `saySomething:` 的作用域中。尽管它所指的真实的字符串对象先于方法调用存在，并且将在方法完成后继续存在。

注解： NSLog() 使用说明符来指示替代单元。就像C标准函数库中的 printf() 函数。记录到控制台的字符串是通过插入提供的值（剩余的参数）来修改格式字符串（第一个参数）的结果。

在 Objective-C 中有一个额外的可替代的单元，%@，用来指示一个对象。在运行时，这个说明符将随着调用 descriptionWithLocale: 方法（如果它存在）或者提供的对象上的 description 方法中的一个而被替换。description 方法由 NSObject 实现用来传回类和对象的存储空间，但是许多 Cocoa 和 Cocoa Touch 的类将它重写来提供更多有用的消息。在 NSString 的例子中，description 方法简单地返回了它所代表的字符串字符。

想要获取更多有关 NSLog() 和 NSString 类的说明符使用的消息，可参见 String Format Specifiers (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html#//apple_ref/doc/uid/TP40004265) 。

方法可以返回值

就像通过方法参数传递值，方法直接返回值是有可能的。在这章中到现在每一个展示出的方法都有一个返回值类型 void 。这个C语言的关键词 void 意味着一个方法不返回任何东西。

规定返回值类型为 int 意味着方法返回一个标量整形数值：

```
- (int)magicNumber;
```

方法的实现使用C语言中的 return 声明来指示在方法在结束执行之后被传回的值，像这样：

```
- (int)magicNumber {
    return 42;
}
```

忽略一个方法会返回值的事实是完全可以接受的。在这个例子中 magicNumber 方法除了返回一个值不做其他任何有用的事，但是像这样调用方法没有任何错误：

```
    int interestingNumber = [someObject magicNumber];
```

你可以用相同的方法从方法中返回对象。举个例子，NSString 类，提供了一个 uppercaseString 方法：

```
- (NSString *)uppercaseString;
```

当一个方法返回一个纯数值时做法相同，尽管你需要使用一个指针来追踪结果：

```
    NSString *testString = @"Hello, world!";
    NSString *revisedString = [testString uppercaseString];
```

当这个方法调用返回时，revisedString 将会指向一个代表 Hello,world! 的 NSString 对象。

记住当实现一个方法来返回一个对象时，像这样：

```
- (NSString *)magicString {
NSString *stringToReturn = // create an interesting string...
   return stringToReturn;
}
```

尽管指针 `stringToReturn` 出了作用域，字符串对象继续存在当它被作为一个返回值被传递时。

在这种状况下有很多存储空间管理的考虑：一个返回了的对象（在堆中被创建）需要存在足够长的时间使它被方法的原有调用者使用，但不是永久存在，这是因为那样会导致内存泄露。在很大程度上，具有自动引用计数（ARC）特征的 Objectiive-C 编译程序会为你照顾到这些需要考虑的地方的。

对象能给他们自己发送消息

无论何时你正在写一个方法的实现，你可以获得一个重要的隐藏值，`self`。概念上，`self` 是一个指向"已经接收到这个消息的对象"的方法。它是一个指针，就像上文的值 `greeting`，可以被用来在现在接收对象上调用方法。

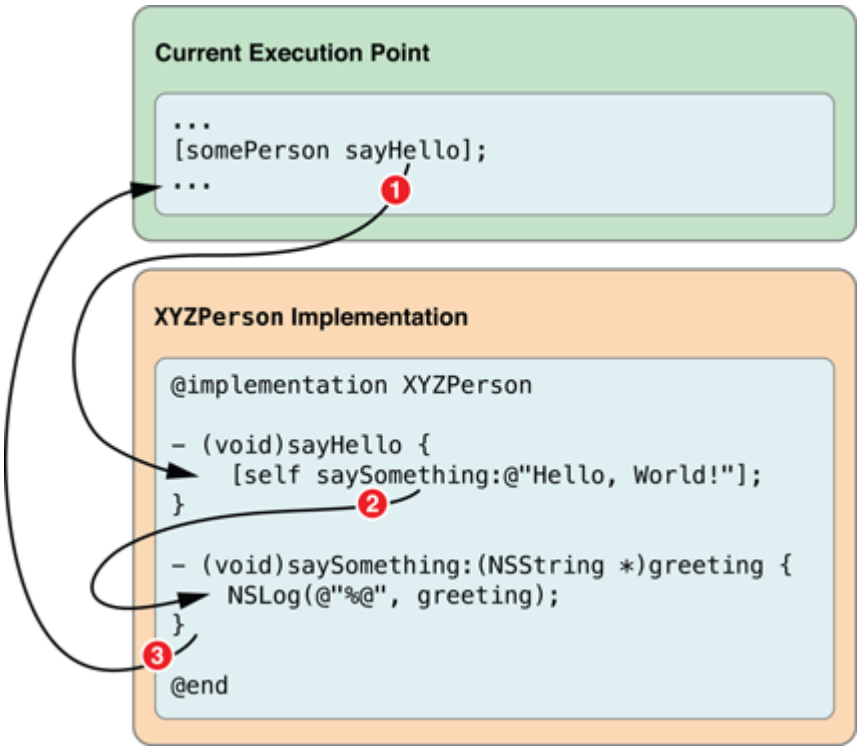
你也许决定通过修改 `sayHello` 方法来使用上文的 `saySomething:` 方法来重构 `XYZPerson` 的实现，因此将 `NSLog()` 的调用移动到不同的方法中。这将意味着你能进一步增加方法，像 `sayGoodbye`，那将每次联系 `saySomething:` 方法来解决真实的问候过程。如果你稍后想将每一个问候在用户接口中的每个文本框中展示出来，你只需要修改 `saySomething:` 方法而不是仔细检查并单独地调整每一个问候方法。

新的在当前对象上使用 `self` 来调用消息的实现将会是这样的：

```
@implementation XYZPerson
- (void)sayHello {
    [self saySomething:@"Hello, world!"];
}
- (void)saySomething:(NSString *)greeting {
   NSLog(@"%@", greeting);
}
@end
```

如果你用这种更新过的实现把消息 `sayHello` 发送给对象 `XYZPerson`，实际的程序流程将会在图2-2 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW8) 中显示。

图 2-2 和自己通信时的程序流程



图片 3.2 2-2

对象可以调用被它们的超类实现的方法

在 Objective-C 中对你来说有另外一个重要的关键词，叫作 `super`。发送一条消息给 `super` 是调用一个被进一步完善继承链的超类所定义的方法的途径。`super` 最普遍的用法是重写一个方法的时候。

比方说你想要创建一个新类型的 person 类，“shouting person”类，它每一句问候都用大写字母展示出来。你可以复制整个 `XYZPerson` 类并修改每个方法中的每个字符串使它们是大写的。但是最简单的方法是创建一个新的继承自 `XYZPerson` 的类，只要重写 `saySomething:` 方法这样它就会以大写的形式展现出来，像这样：

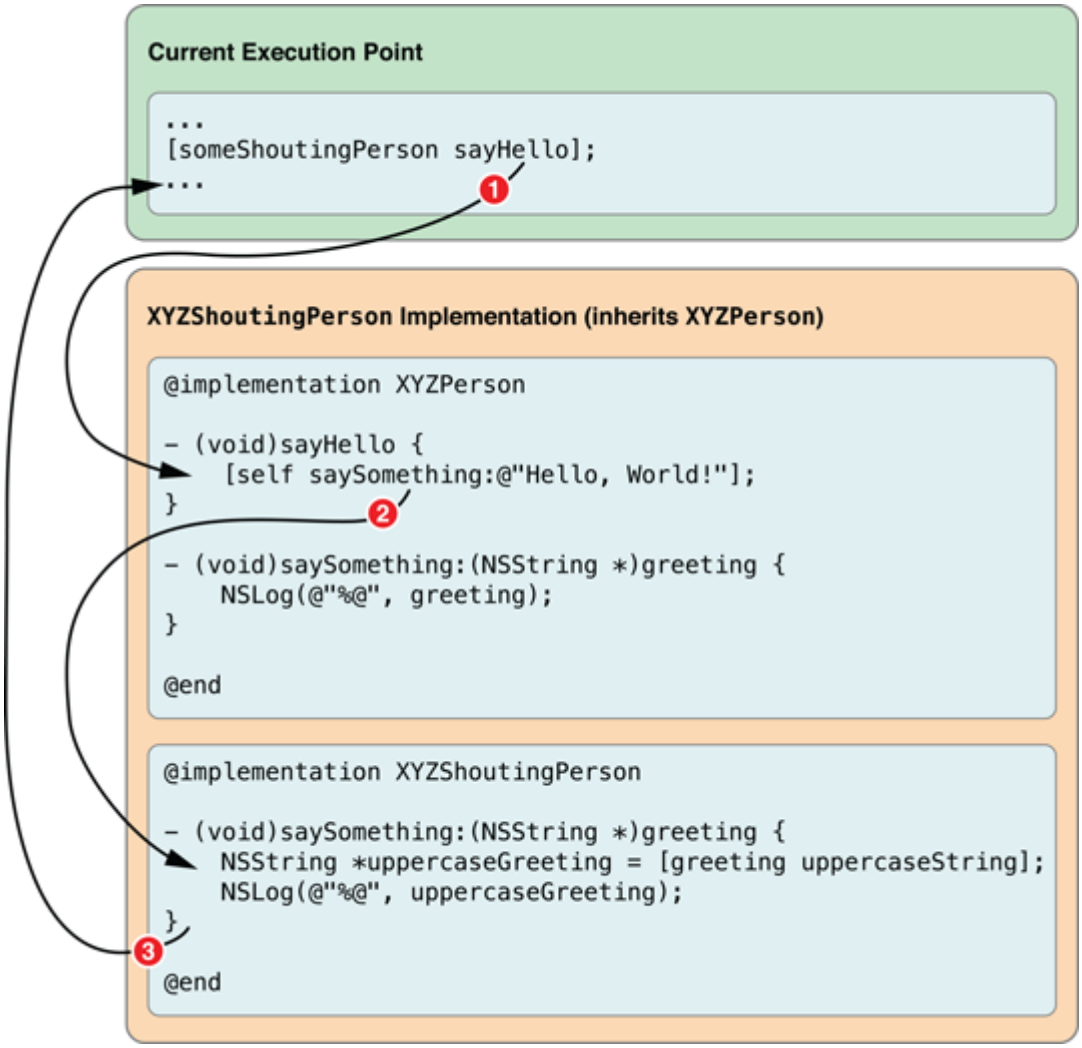
```
@interface XYZShoutingPerson : XYZPerson
@end
```

```
@implementation XYZShoutingPerson
- (void)saySomething:(NSString *)greeting {
    NSString *uppercaseGreeting = [greeting uppercaseString];
    NSLog(@"%@", uppercaseGreeting);
}
@end
```

这个例子声明了一个额外的字符串指针， uppercaseGreeting ，并且将发给初始对象 greeting 的消息 upperc aseString message 返回的值赋给它。正如你早一些所见到的，这将成为一个将原始字符串中的每个字符转换为大写新的字符串对象。

因为 sayHello 由 XYZPerson 实现，而 XYZShoutingPerson 是用来继承 XYZPerson 的，你也可以在 XYZShoutingPerson 对象上调用 sayHello 对象。当你在 XYZShoutingPerson 对象上调用 sayHello 对象时， [self saySomething:...] 的调用将使用重写过的实现并且将问候显示为大写，实际的程序流程图在图2-3 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW9) 中显示。

图 2-3 对于一个覆写方法的程序流程



图片 3.3 2-3

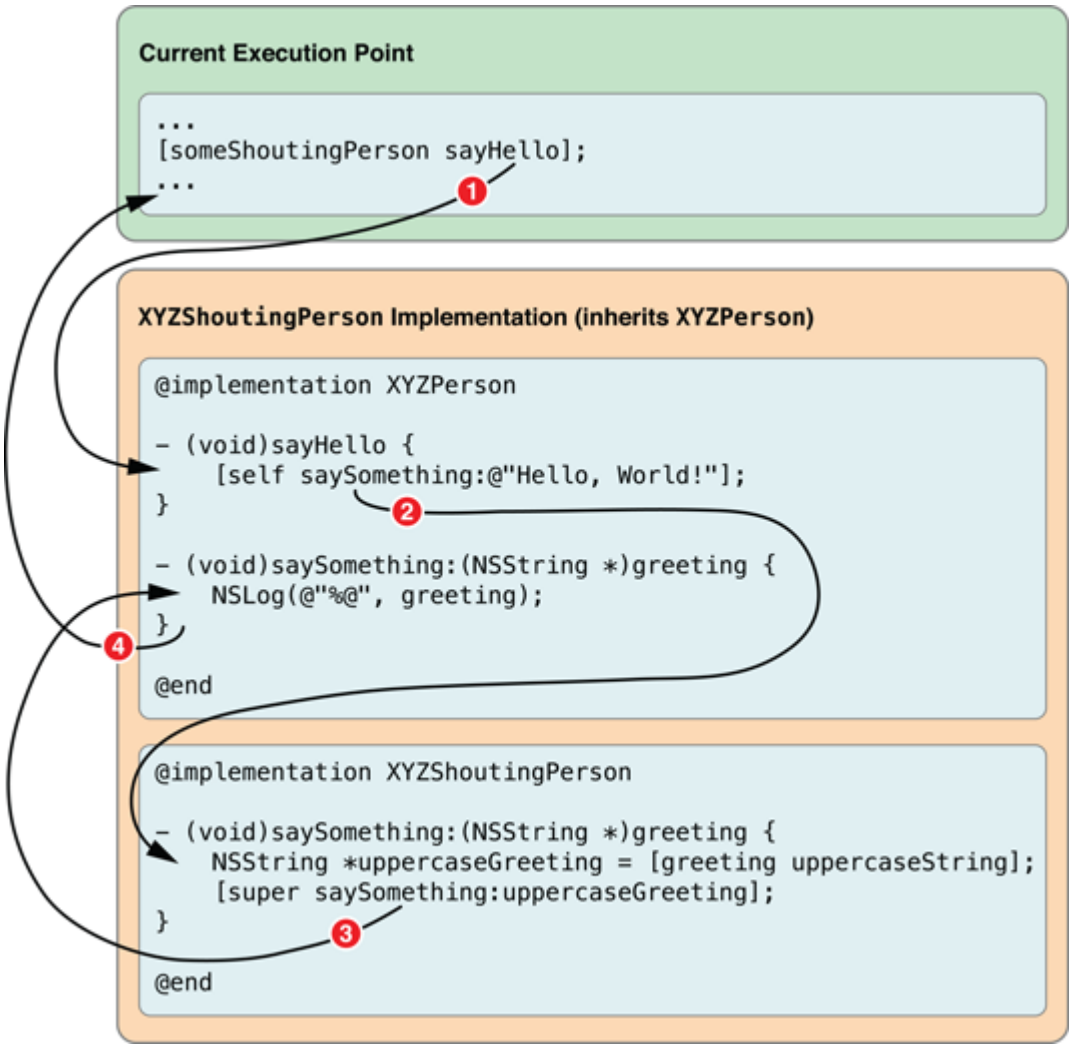
新的实现并不是理想的，是因为如果你确实稍后决定修改 saySomething 的 XYZPerson 实现，用用户接口元素来展示问候而不是通过 NSLog() ，你也将需要修改 XYZShoutingPerson 的实现。

一个更好的想法将会是改变 saySomething 的 XYZShoutingPerson 版本来调用超类(XYZPerson)实现来处理实际的问候：

```
@implementation XYZShoutingPerson
- (void)saySomething:(NSString *)greeting {
   NSString *uppercaseGreeting = [greeting uppercaseString];
   [super saySomething:uppercaseGreeting];
}
@end
```

由于给对象 XYZShoutingPerson 发送消息 sayHello 而来的实际的程序流程如图2-4 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW12) 所示。

图 2-4 和超类通信时的程序流程



图片 3.4 2-4

对象是被动态创建的

正如在这章早些时候描述的，给 Objective-C 对象的存储空间的分配是动态的。创建一个对象的第一步是确认有足够的存储空间，不仅是对被一个对象的类所定义的属性来说，也要满足在它的继承链中在每一个超类上所定义的属性。

根类 `NSObject` 提供了一个类方法， `alloc` ，为你处理这一过程：

```
+ (id)alloc;
```

注意到这个方法的返回值类型为 `id` 。这在 Objective-C 中是一个特殊的关键词，表示“一些类型的对象”。它是一个对象的指针，就像(NSObject *),但是又是特别的因为它不使用星号。它在这章的稍后，Objective-C 是一种动态语言 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW18) 中会被更仔细地描述。

`alloc` 方法有另外一个重要的任务，就是通过将存储空间设为零来清空为对象的属性而分配的存储空间。这避免了一个寻常的问题，即存储空间含有之前曾存储的垃圾。但是这不足以完全初始化一个对象。

你需要将对 `alloc` 的调用和对另一个 `NSObject` 的方法 `init` 的调用结合起来：

```
- (id)init;
```

`init` 方法被类使用以来确认它的属性在创建时有合适的初始值，它将在下一章被详细介绍。

注意到 `init` 也返回 `id` 。

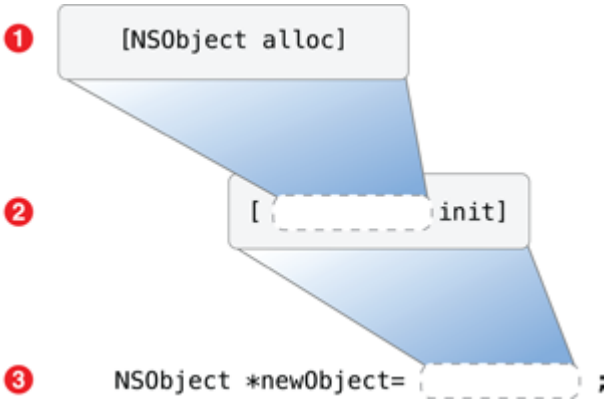
如果一个方法返回一个对象指针，将那个方法的调用作为接收者嵌套进另一个方法的调用时有可能的，由此在一个声明中结合了多个消息的调用。正确分配和初始化一个对象的方法是在对 `init` 的调用中嵌套对 `init` 的调用，像这样：

```
    NSObject *newObject = [[NSObject alloc] init];
```

这个例子设置了 `newObject` 对象来指向一个新被创建的 `NSObject` 实例。

最内部的调用第一个被实现，所以 `NSObject` 类被送到返回一个新被分配的 `NSObject` 实例的 `alloc` 方法。这个返回的对象之后被作为 `init` 消息的接收者被使用，它自己返回对象并赋给 `newObject` 指针，正如在图2-5 (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html#//apple_ref/doc/uid/TP40011210-CH4-SW14) 中显示的那样。

图 2-5 嵌套 alloc 和 init 消息



图片 3.5 2-5

注解： init 返回一个由 alloc 创建的不同的对象是有可能的，所以正如展示的那样，嵌套调用是最好的尝试。

永远不要在没有将任何指针赋给对象的情况下初始化一个对象。作为一个例子，不要这样做：

```
NSObject *someObject = [NSObject alloc];
[someObject init];
```

如果对 init 的调用返回了一些其他的对象，你将留下一个初始被分配过但从没有初始化的对象的指针。

初始化方法可以携带参数

一些对象需要用需要的值来初始化。一个 NSNumber 对象，举例来说，必须用它需要代表的数值来创建。

NSNumber 类定义了几种初始化方法，包括：

```
- (id)initWithBool:(BOOL)value;
- (id)initWithFloat:(float)value;
- (id)initWithInt:(int)value;
- (id)initWithLong:(long)value;
```

带有参数的初始化方法的调用和普通的 init 方法是一样的———一个 NSNumber 对象像这样被分配和初始化：

```
NSNumber *magicNumber = [[NSNumber alloc] initWithInt:42];
```

类工厂方法是分配和初始化的一个选择

正如在前边的章节中所提到的，一个类也可以定义工厂方法。工厂方法提供了传统的 alloc]， init] 过程的不许嵌套两个方法的选择。

`NSNumber` 类定义了几个类工厂方法来匹配它的初始化方法，包括：

```
+ (NSNumber *)numberWithBool:(BOOL)value;
+ (NSNumber *)numberWithFloat:(float)value;
+ (NSNumber *)numberWithInt:(int)value;
+ (NSNumber *)numberWithLong:(long)value;
```

一个工厂方法像这样被使用：

```
NSNumber *magicNumber = [NSNumber numberWithInt:42];
```

这实际上和之前使用 `alloc]` `initWithInt:]` 的例子相同。类工厂方法通常指直接调用 `alloc` 和相关的 `init` 方法，它为方便使用而被提供。

如果初始化不需要参数那么使用 new 来创建一个对象

创建一个类的实例时使用类方法 `new` 是有可能的。这个方法由 `NSObject` 提供并且在你自己的超类中不需要被覆写。

这实际上和调用没有参数的 `alloc` 和 `init` 是一样的：

```
XYZObject *object = [XYZObject new];
// is effectively the same as:
XYZObject *object = [[XYZObject alloc] init];
```

文字提供了一个简洁的对象——创建语法

一些类允许你使用简洁的，文字的语法来创建实例。

你可以创建一个 `NSString` 实例，举例来说，使用一个特殊的文字记号，像这样：

```
NSString *someString = @"Hello, World!";
```

这实际上和分配，初始化一个 `NSString` 或者使用它的类工厂方法中的一个相同：

```
NSString *someString = [NSString stringWithCString:"Hello, World!"
                                           encoding:NSUTF8StringEncoding];
```

`NSNumber` 类也允许各种各样的文字：

```
NSNumber *myBOOL = @YES;
NSNumber *myFloat = @3.14f;
```

```
NSNumber *myInt = @42;
NSNumber *myLong = @42L;
```

此外，这些例子中的每一个实际上和使用相关的初始化方法或者一个类工厂方法相同。

你也可以使用一个框表达式来创建一个 NSNumber ，像这样：

```
NSNumber *myInt = @(84 / 2);
```

在这种情况中，表达式被用数值表示，而且一个 NSNumber 实例伴随着结果被创建。

Objective-C 也支持文字来创建不可变的 NSArray 和 NSDictionary 对象；这些将在 Values and Collections 中进一步讨论。

Objective-C 是一种动态语言

正如之前所提到的，你需要使用一个指针来追踪存储空间中的一个对象。因为 Objective-C 的动态特征，你为那个指针使用什么特定的类型都没有关系——当你给它发送消息时，正确的方法将总是在相关的对象上被调用。

id 型定义了一个通用的对象指针。当声明一个变量时使用 id 是有可能的，但你会失去关于对象编译时的信息。

考虑以下的代码：

```
id someObject = @"Hello, World!";
[someObject removeAllObjects];
```

在这种情况下， someObject 将指向一个 NSString 实例，但是编译器不知道任何事，而事实上它是对象的某一类型。消息 removeAllObjects 由一些 Cocoa 或者 Cocoa Touch 对象（例如 NsMutableArray ）定义所以编译器不会抱怨，尽管这个代码因为一个 NSString 对象不能响应 removeAllObjects 而会在运行时会生成一个异常。

使用一个静态类型来重写编写代码：

```
NSString *someObject = @"Hello, World!";
[someObject removeAllObjects];
```

意味着编译器将会由于 removeAllObjects 没有在任何它所知道的公共的 NSString 接口中被声明而生成一个错误。

因为对象的类是在运行时被决定的，当创建或者使用一个实例时你无论给变量指定什么类型都没有差别。要使用这章早些时候描述的 XYZPerson 和 XYZShoutingPerson 类，你可能要使用下面的代码：

```
XYZPerson *firstPerson = [[XYZPerson alloc] init];
XYZPerson *secondPerson = [[XYZShoutingPerson alloc] init];
[firstPerson sayHello];
[secondPerson sayHello];
```

尽管 firstPerson 和 secondPerson 作为 XYZPerson 的对象都是静态类型的， secondPerson 在运行时将会指向一个 XYZShoutingPerson 对象。当 sayHello 方法在每一个对象上被调用时，正确的实现将会被使用；对于 secondPerson ，这意味着 XYZShoutingPerson 的版本。

确定对象相等

如果你需要确定一个对象是否和另一个对象相同，记住你在使用指针是重要的。

标准的C语言等号运算符 == 被用来检测两个变量的值是否相同，像这样：

```
if (someInteger == 42) {
    // someInteger has the value 42
}
```

当处理对象的时候，运算符 == 被用来检测两个单独地指针是否指向一个相同的对象：

```
if (firstPerson == secondPerson) {
    // firstPerson is the same object as secondPerson
}
```

如果你需要检测两个对象是否代表相同的数据，你需要调用一个像 isEqual: 的方法，从 NSObject 可以获得：

```
if ([firstPerson isEqual:secondPerson]) {
    // firstPerson is identical to secondPerson
}
```

如果你需要比较一个对象是否比另一个对象代表了更大或更小的值，你不能使用标准C语言的比较运算符 > 和 <。相反，像 NSNumber , NSString 和 NSDate 这样的基本类型，提供了一个方法 compare: ：

```
if ([someDate compare:anotherDate] == NSOrderedAscending) {
    // someDate is earlier than anotherDate
}
```

使用 nil

在你声明它们的时候初始化纯量变量总是一个好主意，否则它们的初始值将包含前一个堆栈的垃圾内容：

```
BOOL success = NO;
int magicNumber = 42;
```

这对对象指针来说不是必要的，因为编译器将自动把变量设为 `nil` 如果你不规定任何其他的初始值：

```
XYZPerson *somePerson;
// somePerson is automatically set to nil
```

`nil` 的值对初始化一个对象指针来说是最安全的，如果你没有另一个值来使用的话。因为在 Objective-C 中发送消息给 `nil` 是完全可以接受的。如果你确实给 `nil` 发送了消息，显然什么都不会发生。

注解：如果你期待从发送给 `nil` 的消息中获得一个返回值，返回值将会是对象返回类型的 `nil` 型，数字类型的 0, `BOOL` 类型的 `NO` 。返回的结构拥有所有初始化为 0 的成员。

如果你需要检查确认一个对象不是 `nil` （一个变量指向存储空间中的一个对象），你可以使用标准C语言中的不等运算符：

```
if (somePerson != nil) {
    // somePerson points to an object
}
```

或者简单地提供变量：

```
if (somePerson) {
    // somePerson points to an object
}
```

如果 `somePerson` 变量是 `nil` ，它的逻辑值是 `0` （false）。如果它有地址，它就不是零，所以被认为是 true。

相同的，如果你需要检查一个 `nil` 变量，你可以使用相等运算符：

```
if (somePerson == nil) {
    // somePerson does not point to an object
}
```

或只是使用 C 语言逻辑非操作符：

```
if (!somePerson) {
    // somePerson does not point to an object
}
```

练习

1.在你的项目里从上一章最后的练习中打开 `main.m` 文件并且找到 `main()` 函数。作为对于任何用 C 语言写的可执行的程序，这个函数代表了你应用的起点。

使用 `alloc` 和 `init` 创建一个新的 `XYZPerson` 实例，然后调用 `sayHello` 方法。

注解：如果编译器没有自动提示你，你将需要在 `main.m` 的顶部导入头文件(包含 `XYZPerson` 接口)

2.实现在这一章早些时候展示的方法 `saySomething:` ,并且重新编写方法 `sayHello` 来使用它。添加各种其他的问候并且在你之前创建的每一个实例上调用它们。

3.为 `XYZShoutingPerson` 类创建新的类文件，设置成继承自 `XYZPerson` 。

覆写 `saySomething:` 方法来展示大写的问候，并且测试在 `XYZShoutingPerson` 实例上的行为。

4.实现在前一章你声明的 `XYZPerson` 类 `person` 工厂方法，来返回一个被正确分配和初始化的 `XYZPerson` 类的实例，然后在 `main()` 中使用方法来代替嵌套的 `alloc` 和 `init` 。

提示：尝试使用 [[self alloc] init] 而不是使用 [[XYZPerson alloc] init]。

在一个类工厂方法中使用 `self` 意味着你在指向类本身。

这意味着你不必在 `XYZShoutingPerson` 实现中覆写 `person` 方法来创建正确的实例。通过检查以下代码来测试这个：

```
XYZShoutingPerson *shoutingPerson = [XYZShoutingPerson person];
```

创建正确类型的对象。

5.创建一个新的局部的 `XYZPerson` 指针，但是不包括任何其他赋值。 使用一个转移指令（ `if` 声明）来检查变量是否自动被赋为 `nil` 。

4

数据封装 – Encapsulating Data

除了前面一章提到的消息行为（ messaging behavior ）外，对象还可以通过属性封装数据。

这一章节描述了 ObjC 中用于声明对象属性的语法，阐明了属性如何通过默认的访问方法和实例变量的生成实现。如果实例变量支持该属性，那么该变量必须在所有初始化方法中正确的设定。

如果一个对象需要通过属性包含一个连向其他对象的链接，那么考虑这两个对象之间的关系性质将会变得很重要。尽管ObjC的内存管理将会通过自动引用计数（ Automatic Reference Counting (ARC) ）为你处理大部分类似情况，但你仍需要了解这些，以避免类似强引用环（ strong reference cycle ）导致内存溢出等问题的出现。这一章还介绍了对象的生命周期，以及如何从利用关系来管理对象图的角度思考。

用属性封装对象的值

大部分对象都会通过跟踪信息来执行任务。一些对象被设计为一个以上具体值的模型，如Cocoa 中的用来保存数值的NSNumber类 或用来代表一个有姓、名的人的自定义类 XYZperson 。还有些对象作用域更为广泛，甚至可能用来处理用户界面和其显示信息之间的交互，但即使这样的对象，仍需要跟踪用户界面元素或相关模式对象的信息。

为外部数据声明公共属性

ObjC的属性提供了一种定义信息的方式，而这些信息正是类中将要封装的。正如你在 Properties Control Access to an Object’s Values (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/DefiningClasses/DefiningClasses.html#//apple_ref/doc/uid/TP40011210-CH3-SW7) 一节中看到的，类的接口（ interface ）包含了属性声明，例如：

```
@interface XYZPerson : NSObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

在这个例子中XYZPerson类声明了string 属性来保存一个人的名和姓。 这是面向对象编程中一个非常基本的原则——将对象的内部运作隐藏在它的公共接口之后，所以我们总是使用一个对象的外部特性来访问它的属性，而不是直接尝试去访问它内部的值。

通过访问器（ accessor ）方法来获得或设置属性的值

你通过访问器（ accessor ）方法来访问或设定对象的属性：

```
NSString *firstName = [somePerson firstName];
[somePerson setFirstName:@"Johnny"];
```

默认地编译器自动为你生成访问器方法，除了在类接口中用 @propertys 声明属性外你不需要做任何事情。 生成中遵循的特定命名惯例：

- 用于访问值的方法（ getter 方法 ）拥有与属性一样的名字 对于上例中，名为firstName的属性它的 gette r 方法名也为firstName 。
- 用于设置值的方法（ setter 方法 ）用“ set ”开头的单词命名，并且其中的属性名首字母要大写。

对于上例中，名为 firstName 的属性它的setter方法可以取名 setFirstName 。 如果你不想属性通过setter方法改变值，可以在属性声明中增加一个特征（ attribute ） readonly 表明其不能修改只能读取：

```
@property (readonly) NSString *fullName;
```

除了告诉其他的对象他们应该怎样和一个属性交互以外，特征还告诉了编译器如何合成相应的访问方法。在这种情况下编译器将会合成一个名为fullName getter方法，而不是一个setFullName方法。

注： 与 readonly 相对的是 readwrite ，因为 readwrite 特征是默认设置的，所以没有必要再去指明它。 如果你想给访问器方法另外命名，通过给相应属性添加特征，来确定自定义名称是可行的。对于布尔属性（ 属性值只有 YES 或 NO ），它的 getter 方法按照惯例应该以“ is ”开头，例如，对于一个名为 finished 属性，它的gette r方法，应该命名为“ isFinished ”。 同样的，给属性添加一个特征是也是可行的：

```
@property (getter=isFinished) BOOL finished;
```

如果你需要指明多个特征，只需把他们排成一排并用逗号分隔开，像下面这样：

```
@property (readonly, getter=isFinished) BOOL finished;
```

在这种情况下，编译器仅会合成一个 isFinished 方法，而不是 setFinished 方法。

注： 一般来说属性访问方法遵循键/值编码，也就是说它的命名是遵循一定的命名惯例的。参看Key-Value Coding Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html#//apple_ref/doc/uid/10000107i) 获得更多信息

点语法对于访问器方法调用是简洁的选择

除了明确的访问器（ accessor ）方法调用，ObjC 还提供了另外一种选择来访问对象属性——点语法 你可以这样使用点语法访问属性：

```
NSString *firstName = somePerson.firstName;
somePerson.firstName = @"Johnny";
```

点语法仅是对访问器方法做了一个单纯的包装，当你使用它的时候，你仍是在使用前面提到的 getter 和 setter 方法，来对属性进行访问或变更。

- 通过somePerson.firstName 获得一个值与使用 [somePerson firstName]是相同的
- 通过 somePerson.firstName = @"Johnny" 赋值与使用 [somePerson setFirstName:@"Johnny"] 是相同的 这意味着通过点语法访问属性也是由属性特征控制的。比如一个属性标记为readonly ，那么当你试图通过点语法对该属性进行设置时，将会出现编译错误。

实例变量支持大多数的属性

默认情况下，一个可读写属性会获得实例变量的支持，这些会由编译器自动生成。 实例变量是一种在对象的生命周期中一直存在并保存着值的变量。实例变量的存储空间是在初次创建时被分配（通过 alloc ），在 dealloc 时被释放。 你可以另外指定其他的名称，或者实例变量将会生成与属性一样的名字，但实例变量的名字之前还会有一个下划线前缀。例如，实例变量的一个可能的名字 _firstName 尽管通过访问器方法或点语法来访问对象自身的属性是最好的实践，但直接通过类实现中的任意实例方法来访问实例变量也是可行的。下划线前缀表明了你访问的是一个实例变量而不是其他的，诸如局部变量之类的变量

```
- (void)someMethod {
    NSString *myString = @"An interesting string";

    _someString = myString;
}
```

在这个例子中很明显 myString 是一个局部变量 _someString 是一个实例变量。一般来说，你会使用点语法或访问方法进行属性访问，即使是在对象自身的实现里进行也是这样，这时将会用到 self：

```
- (void)someMethod {
NSString *myString = @"An interesting string";

self.someString = myString;
// or
[self setSomeString:myString];
}
```

对于这条规则使用的特例是，初始化、释放空间或自定义访问器（accessor）方法，这些情况将会在后面的部分讲到。

你可以自定义生成的实例变量的名字

正如前面提到的，一个可写属性的默认行为是使用一个名为 _propertyname 的实例变量。 如果你想为实例变量取另外的名字，你需要指导编译器在你的实现中生成一个使用下列语法的变量：

```
@implementation YourClass
@synthesize propertyName = instanceVariableName;
...
@end
```

例如

```
@synthesize firstName = ivar_firstName;
```

在这种情况下，这个属性仍然要被命名为 firstName ，并仍可以通过访问器（accessor）方法—— firstName 和 setFirstName 或点语法访问。但这种情况下，属性是通过一个名为 ivar_firstName 的实例变量获得支持的。 **重要：** 如果你使用 @synthesize 关键字时，不去指明实例变量的名字的话，像下面这样：

```
@synthesize firstName;
```

那么实例变量将会被命成与属性一样的名字。 在这个例子中，实例变量将会被命名为 firstName ,并且不会加上下划线前缀。

你可以不通过属性来定义实例变量

当你需要跟踪一个值或者对象时，最好的实现是通过使用另一个对象的属性。 如果你确实需要定义一个你自己单独的实例变量，即不通过声明属性获得，你可以将他们声明在，类接口或实现的那个大括号的最开始，例如：

```
@interface SomeClass : NSObject {
NSString *_myNonPropertyInstanceVariable;
}
...
@end

@implementation SomeClass {
NSString *_anotherCustomInstanceVariable;
}
...
@end
```

注意： 你还需要将这样的实例变量声明在拓展类的最开始，参见 Class Extensions Extend the Internal Implementation (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html#//apple_ref/doc/uid/TP40011210-CH6-SW3) 。

可以直接通过初始化方法来访问实例变量

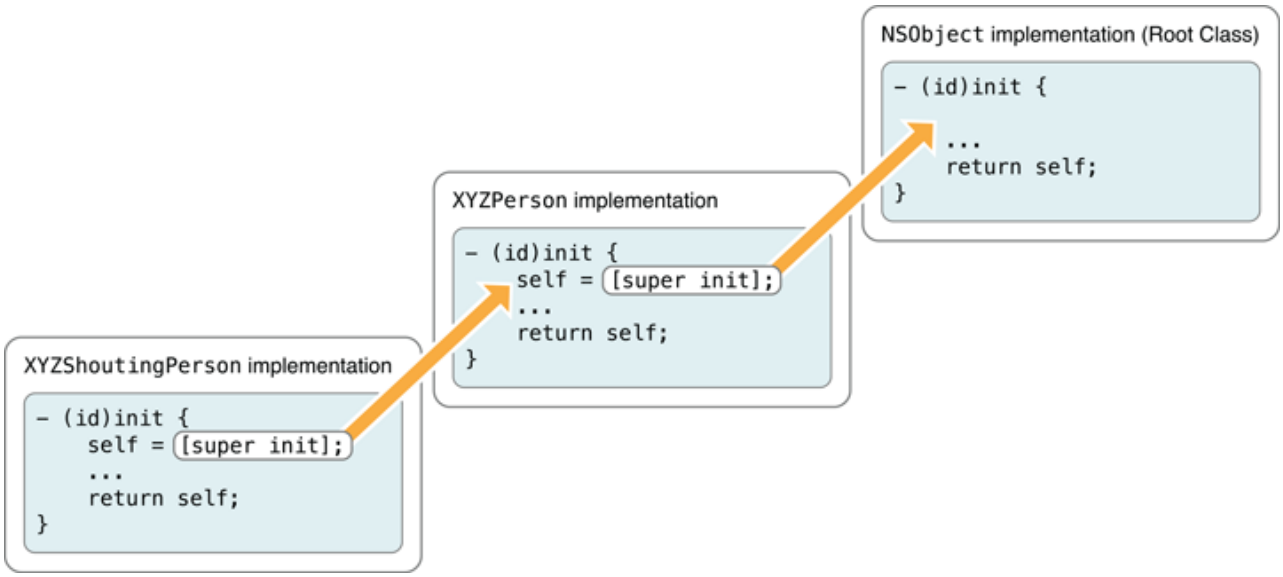
Setter 方法可能会产生额外的副作用，它可能会触发 KVC 通知，或在你编写了自定义方法时执行进一步的任务。 你应该总是从一个初始化方法中直接访问实例变量，因为当属性被设置好时，一个对象的其余部分可能还未

初始化完全。即使你不提供自定义访问器（ accesso ）方法,或对自定义类中可能产生的副作用有一定了解，之后产生的子类还是很可能会覆写行为（ behavior ）。 一个典型的init方法会像下面这样：

```
– (id)init {
self = [super init];

if (self) {
    // initialize instance variables here
}

return self;
}
```

一个 init 方法应该在开始它自己的初始化之前，首先用 self 响应父类的初始化方法请求 。一个父类在初始化对象失败之后会返回一个 nil ，所以应当总是在执行你自己类的初始化之前，检查 self 的返回值。

图 3–1 初始化过程



图片 4.1 初始化过程

正如你在前面的章节所看到的，对象的初始化有两种方式，通过调用 init ,或调用为对象初始化具体值的方法。 在XYNPerson例子中，提供一个可以设置初始名和姓的初始化方法是行的通的。

```
– (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName;
```

你可以这样实现这个方法：

```
– (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName {
self = [super init];

if (self) {
```

```
    _firstName = aFirstName;
    _lastName = aLastName;
}

return self;
}
```

指定初始器是最主要的初始化方法

如果一个对象声明了一个以上的初始化方法，你应该确定一个方法作为指定初始器方法，它通常是为初始化提供最多选择的方法（像是拥有最多参数的方法），并会被其他的便利方法调用。你通常还需要覆写 init 方法来通过合适的默认值调用指定初始器。

如果一个XYZPerson 类还包含一个用来表示生日的属性，那么它的指定初始器方法应该是这样：

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName
                                    dateOfBirth:(NSDate *)aDOB;
```

这个方法还会设置相应的实例变量值，像上面展示的一样。如果仍希望提供一个只包含姓、名的指定初始器，你需要像下面这样调用指定初始器，来实现这个方法。

```
- (id)initWithFirstName:(NSString *)aFirstName lastName:(NSString *)aLastName {
return [self initWithFirstName:aFirstName lastName:aLastName dateOfBirth:nil];
}
```

你或许还需要实现一个标准的 init 方法来提供合适的默认值，例如：

```
- (id)init {
return [self initWithFirstName:@"John" lastName:@"Doe" dateOfBirth:nil];
}
```

如果你需要在一个使用多个 init 方法的子类创建时，编写初始化方法，那么，你可以覆写父类的指定初始器来实现你自己的初始化，或者选择再另外添加一个你自己的初始器。无论哪种方法，你都需要在开始你自己的初始化之前，调用父类的指定初始器（在此处 [super init] 调用）。

你可以实现你自定义的访问器方法

属性也不总是被他们自己的实例变量支持的。举一个例子，XYZPerson 类可以为一个人的全名定义一个只读属性：

```
@property (readonly) NSString *fullName;
```

比起每次姓或名一变更就不得不更新 fullName 属性，只使用自定义访问器（accessor）方法来建立一个要求的全名字符串就显得容易许多。

```
- (NSString *)fullName {
return [NSString stringWithFormat:@"%@ %@", self.firstName, self.lastName];
}
```

这个简单的例子使用格式字符串和标识符，建立了一个包含了以空格隔开的姓名的字符串。

注意： 尽管这是一个很实用的例子，但认识到它的语境特殊性是很重要的，并且它只适用于那些将名放在姓之前的国家。 如果你为一个确实需要实例变量的属性自定义了一个访问器方法，你必须直接从这个方法的内部来访问这个属性的实例变量。 例如，将属性的初始化推延到它第一次被调用时是普遍的做法，这种做法叫做“ lazy访问器”：

```
- (XYZObject *)someImportantObject {
if (!_someImportantObject) {
   _someImportantObject = [[XYZObject alloc] init];
}

return _someImportantObject;
}
```

在返回值之前这个方法会首先检查 _someImportantObject 实例变量的值是不是 nil ，如果是，它将会分配一个对象。

注意： 当编译器生成了至少一个访问器（ accessor ）方法时，它都会再自动生成一个实例变量。但当你为一个读写属性实现了 getter , setter 方法，或为只读属性实现了 getter 方法，编译器都会假设你想要控制属性的实现，并不再会自动生成一个实例变量。此时，如果你仍需要一个实例变量，你需要手动请求生成：

```
@synthesize property = _property;
```

属性是默认的原子单元

属性是 ObjC 中默认的原子单元。

```
@interface XYZObject : NSObject
@property NSObject *implicitAtomicObject;          // atomic by default
@property (atomic) NSObject *explicitAtomicObject; // explicitly marked atomic
@end
```

这意味着生成的访问器（ accessor ）会确保值总是被 getter 方法取出，或被 setter 方法设置，即使它此时正在被不同的线程调用。由于原子访问器的内部实现和同步是私有的，所以将自己实现的访问器方法，同编译器生成的结合到一起是不太可能的。如果你这样尝试了，你将会得到编译器警告，例如这种情况：为一个原子读写属性提供一个自定义的 setter 方法，却将 getter 方法留给编译器生成。

你可以使用非原子（ nonatomic ）属性特征来为生成的访问器方法指明，它是要直接设置一个值还是返回一个值，但当同样的一个值被不同的线程访问时，不保证会发生什么情况。正是由于这个原因，访问一个非原子属性比访问原子属性快，并且将生成的 setter 方法与其他方法结合是可行的，例如，与你自己实现的 getter 方法。

```
@interface XYZObject : NSObject
@property (nonatomic) NSObject *nonatomicObject;
@end

@implementation XYZObject
- (NSObject *)nonatomicObject {
return _nonatomicObject;
}
// setter will be synthesized automatically
@end
```

注意： 属性原子性与对象线程安全性（ thread safety ）并不是同义词。 考虑 XYZPerson 对象的例子，如果出现一个人的名和姓被一个线程的原子访问器方法修改的情况。此时，另一个线程也在访问这两个字符串，原子的 getter 方法会返回完整的字符串，但并不保证这时的这两个值彼此之间是正确对应的。比如名是在另一个线程访问前变更的，而姓则是在之后，那么你最终得到的将会是一对前后不一，错误匹配的姓和名。

这个例子很简单，但如果考虑了关联对象的关系网后，线程安全性问题将会变得更加复杂。更多细节参看Concurrency Programming Guide (https://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008091) 。

通过所有权和责任来管理对象图

正如你所看到的， ObjC 为对象划分的内存是动态分配的（通过堆），这意味着你需要通过指针来跟踪对象的地址。与标量值不同，通过指针变量作用域来确定对象生命周期不一定可行。相反，一个对象只要是被其他对象调用了，就要始终在内存中保持活跃。

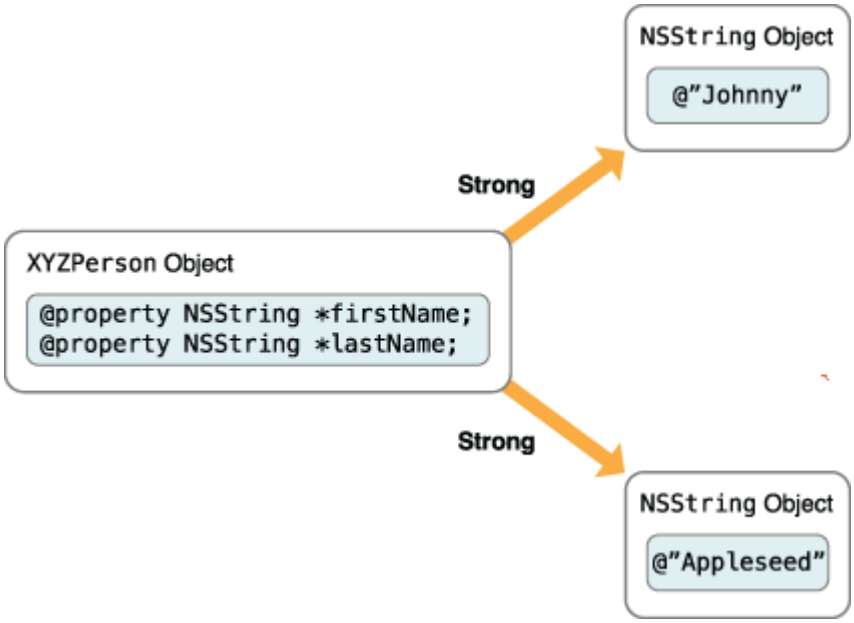
相比起去关心如何手动管理每个对象的生命周期，你更应该去认真考虑对象之间的关系。

就拿 XYZPerson 的例子来说， firstName 和 lastName 这两个字符串属性可以说是有效的被 XYZPerson 实例所“拥有”，这意味着只要 XYZPerson 对象存在在内存中， 这两个属性就同样存在。

当一个对象，通过有效地掌握其他对象的所有权的方式，依赖于其他对象时，这个对象就被称为强引用（ strong reference ）其他对象。

在 ObjC 中，只要有至少一个对象强引用了另一个对象，那么后面这个对象都会一直保持活跃。 XYZPerson 实例与两个 NSString 对象的关系图见图 3–2：

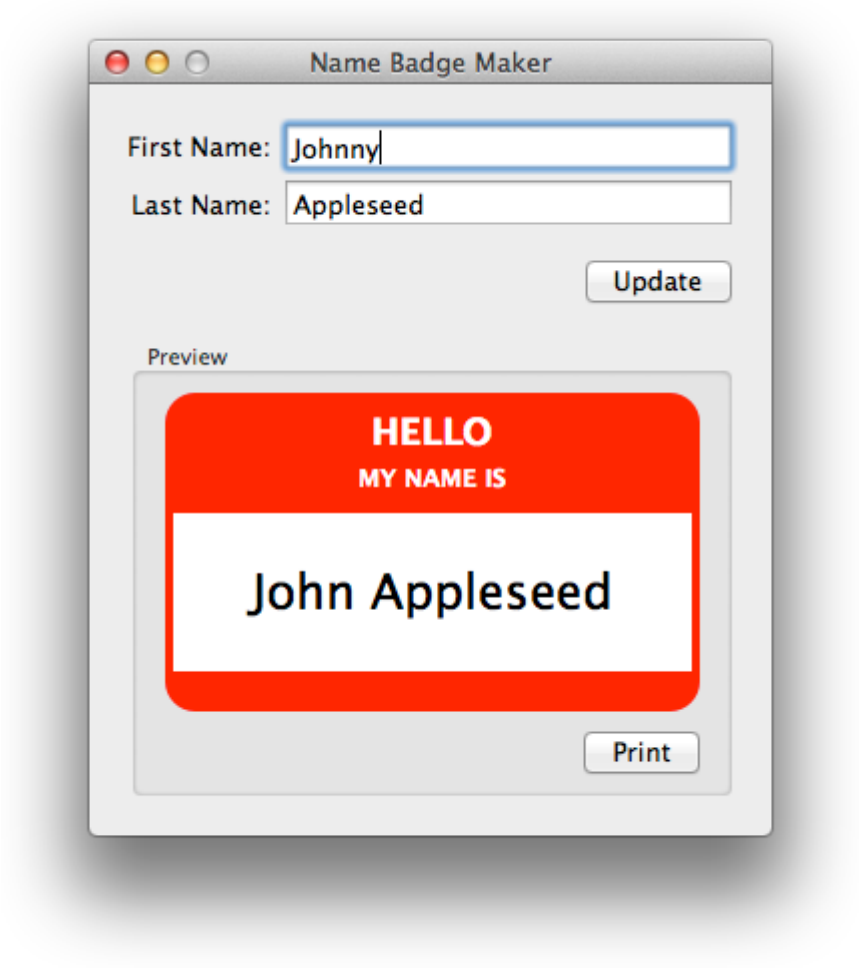
图 3–2 强参考



图片 4.2 strongpersonproperties

当一个 XYZPerson 对象从内存中被释放时，假设这时不再有其他任何对象强引用他们，那么这两个字符串对象也会同样被释放。再为这个例子增加一点复杂性，考虑一下下面展示的这个应用的对象图：

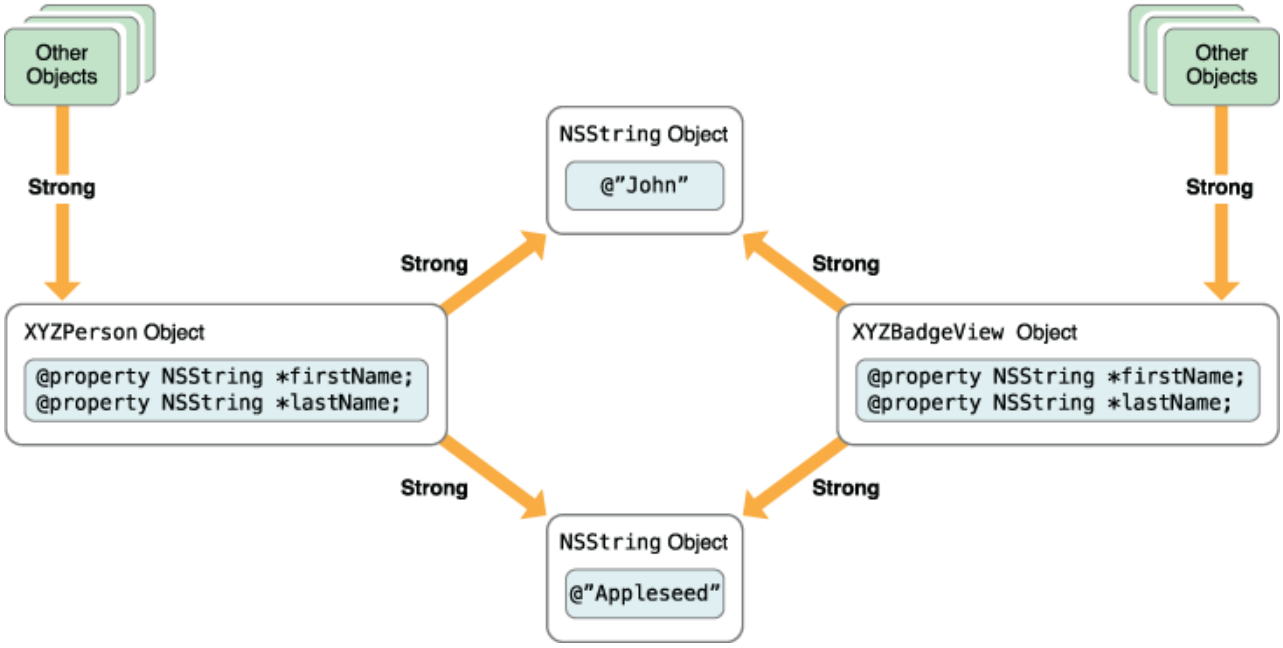
图 3–3 姓名标志制作 应用



图片 4.3 namebadgemaker

当用户点击了更新按钮时，这个标志的预览图就会根据相应的姓名信息更新。当一个 person 对象的信息第一次输入并点击更新按钮时，简化的的对象图可能会是图3-4中的样子，

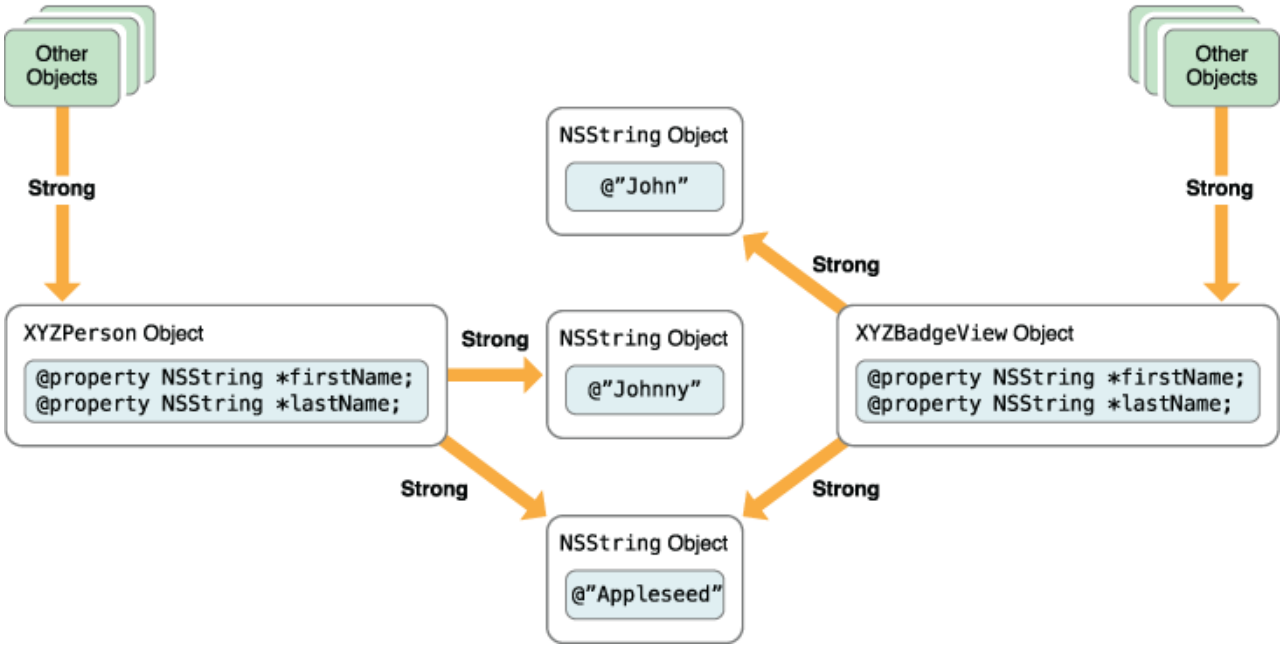
图 3-4 初始 XYZPerson 创建时的关系图



图片 4.4 simplifiedobjectgraph1

当用户调整了输入的名时，关系图会变成图3–5 中的样子

图 3–5 当名改变时的简化关系图

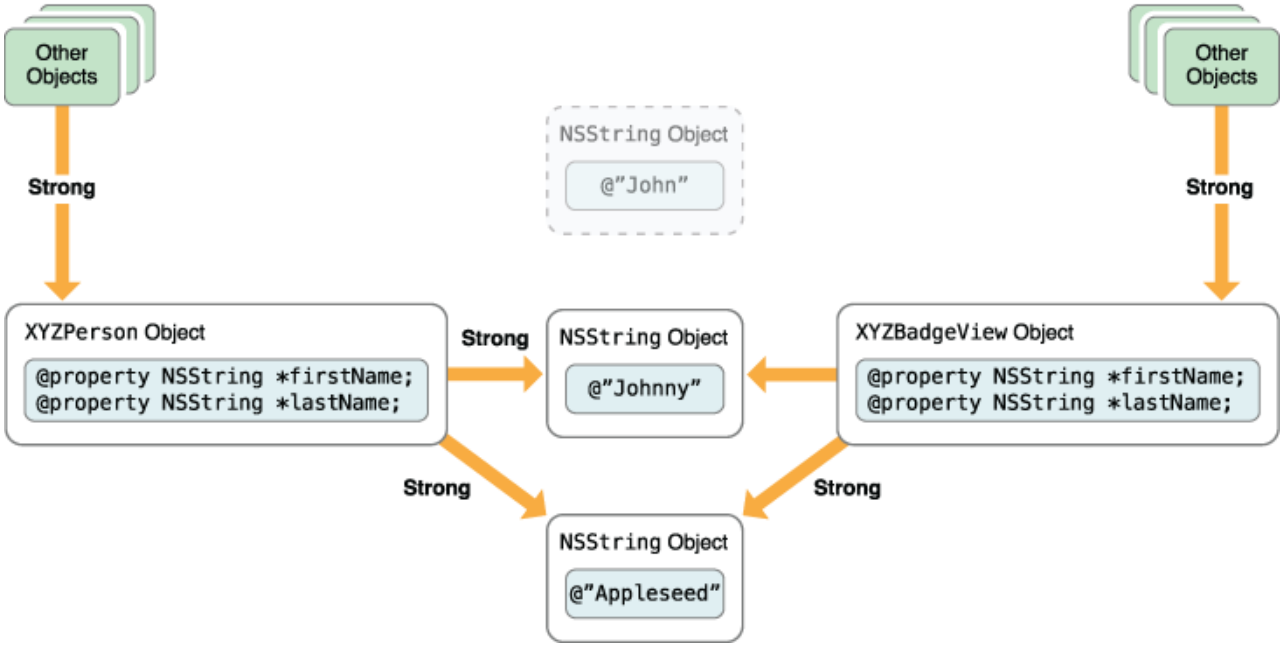


图片 4.5 simplifiedobjectgraph2

尽管此时 XYZPerson 对象已经有了一个不同的 firstName ， 标志视图仍然还包含着一个跟最开始的 @”Joh n” 字符对象的强引用。这意味着 @”John” 对象仍在内存中，并且还被标志视图用来输出名字。

一旦用户第二次点击了更新按钮，标志视图界面将会被告知更新它的内部属性以达到与 person 对象匹配。这时关系图会是图3-6中的样子：

图 3-6 更新了标志视图后的简化对象图



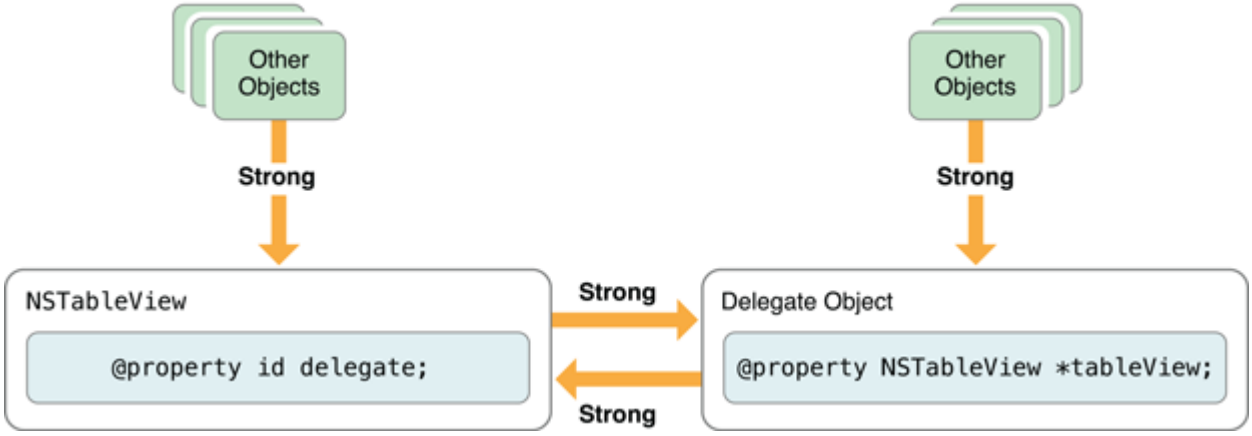
图片 4.6 simplifiedobjectgraph3

这时不再会有任何强引用与最开始的 @”John” 对象关联，它将会从内存中移出。

避免强引用环(strong reference cycles)

尽管对于对象之间的单向关系，强引用表现的很好，但当你在处理一组互相联系的对象时你就需要小心了。当一组对象是通过强引用环联系时，那么即使外接已经不存在任何跟他们的强引用，他们还是因为对彼此的强引用而保持活跃。 一个明显的例子就是，视图表对象与其授权对象（ delegate ）之间可能隐含的引用环（ iOS 中的UI TableView 和 OS X 中的 NSTableView ）。为了使通用视图表类在大多数情况下都可用，它会将一些决定授权给其他外部对象处理。这意味着视图表对象依赖于其他对象来决定显示什么样的内容，或者当用户与视图表中的特定项发生交互时应该做什么。 一个常见的情况是视图表引用了他的授权对象，相应的授权对象也会发出一个关联视图表的引用。

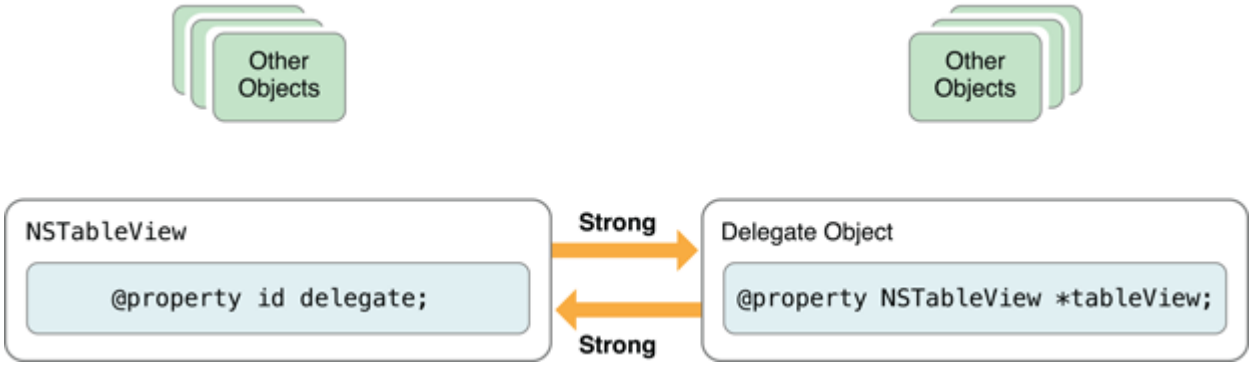
图 3-7 视图表与授权对象之间的强引用



图片 4.7 strongreferencecycle1

但是当其他对象撤销了他们与视图表与其授权对象之间的强引用时，问题就出现了

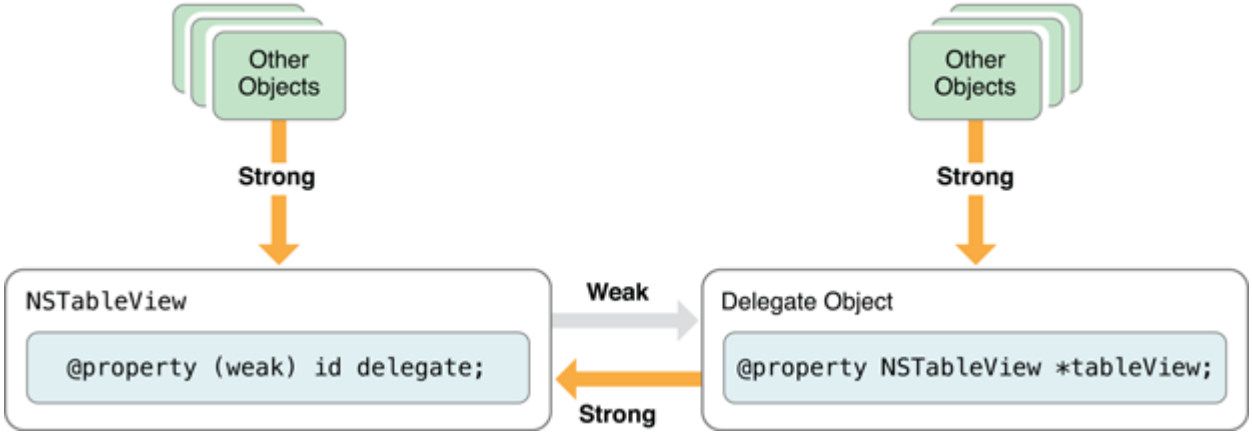
图 3-8 一个强引用环



图片 4.8 strongreferencecycle2

即使现在这两个对象已经没有任何在内存中存在的必要了——除了他俩之间还存在关联之外，已经没有任何对象与他们有强引用了，但他们却因为彼此之间存在的强引用而一直保持活跃。 当视图表将它与它授权对象之间的关联修改为弱关联（ weak relationship ）的时候（这也是 UITableView 和 NSTableView 如何解决这个问题的），最初的关系图将会变成图3-9。

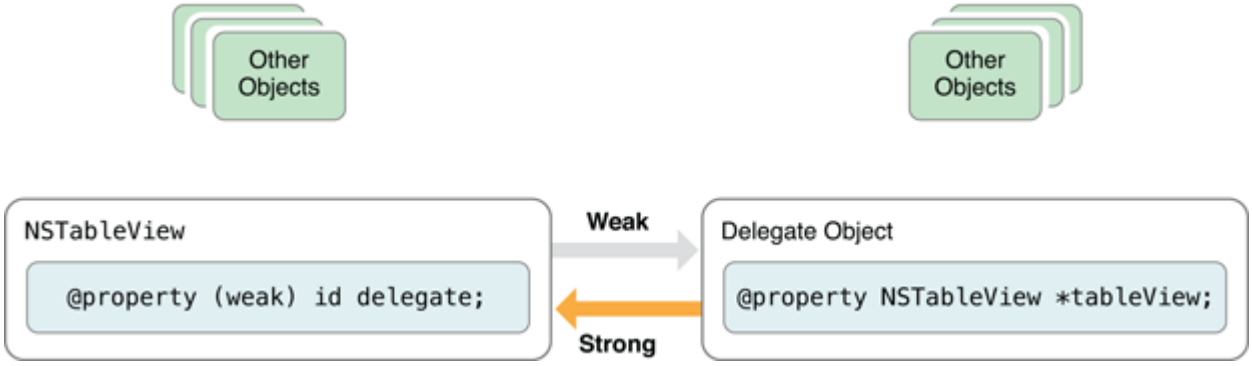
图 3-9 视图表与其授权对象之间的正确关系



图片 4.9 strongreferencecycle3

如果此时其他对象再撤销他们与视图表和其授权之间的强引用，视图表就不会再与它的授权有强引用了。如图 3–10

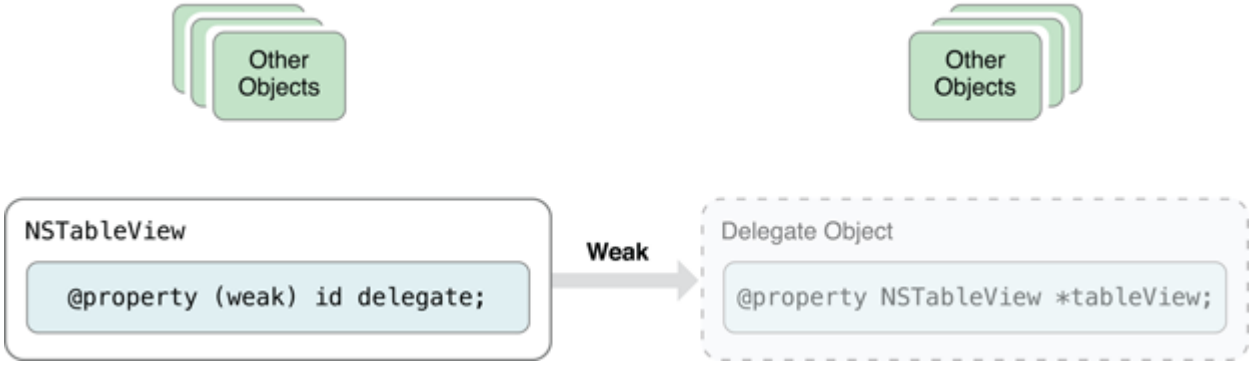
图 3–10 规避了强引用的环



图片 4.10 strongreferencecycle4

这意味着授权对象会被释放，因此它对视图表的强引用也会解除，如图3–11

图 3–11 释放授权对象



图片 4.11 strongreferencecycle5

一旦授权对象被释放，也就不再有关联于视图表的强引用了，因此它也会被释放。

通过强、弱声明（Strong and Weak Declaration）来管理所有权

对象属性的默认声明一般是下面这样：

```
@property id delegate;
```

为它生成的实例变量使用了强引用。你可以为属性添加一个特征，来声明弱关联，像这样：

```
@property (weak) id delegate;
```

注： 与弱（ weak ）相反的是强(strong)，由于强是默认的，所以不需要特别指明出强特征。 局部变量（还有非属性实例变量）同样也默认包含与对象的强引用，这意味着下面这段代码将会如你所期待的那样准确运行：

```
NSDate *originalDate = self.lastModificationDate;
self.lastModificationDate = [NSDate date];
NSLog(@"Last modification date changed from %@ to %@",
            originalDate, self.lastModificationDate);
```

在这个例子中，局部变量 originalDate 包含了一个与初始对象 lastModificationDate 的强引用。当 lastModificationDate 属性变更时，它不会再强引用原来的日期值，但这个日期仍然会被 originalDate 强变量保存着。

注： 只有当变量在作用域内、或还未分配给其他对象前、或值为 nil 的时候，它才会包含一个与对象的强引用。如果你不想一个对象包含强引用，你可以把它声明为 _weak ，像是这样：

```
NSObject * __weak weakVariable;
```

因为弱关联不会保证对象的活跃，所以有可能在关联还在使用的时候关联对象就被释放了。为避免出现悬空指针的情况，即指针指向的内存开始有对象后来却被释放了，当对象被释放时弱关联会自动被设置为 nil。

这意味着你在前面的例子中使用一个弱变量时，

```
NSDate * __weak originalDate = self.lastModificationDate;
self.lastModificationDate = [NSDate date];
```

originalDate 变量存在被设置成 nil 的可能性。当 self.lastModificationDate 被重新分配时，属性将不再会保有一个与原日期值相关的强引用。如果此时没有其他变量强引用该日期值，那么原日期将被释放， originalDate 也会被设置成 nil 。 弱变量可能会是混乱的来源，特别是在下面的编码中：

```
NSObject * __weak someObject = [[NSObject alloc] init];
```

在上面的例子中，新分配的对象没有任何与之相关的强引用，所以它会立即被释放， someObject 也会被置为nil。

注意： 与 _weak 相对的是 _strong ，再次重申，你不需要特地指明 _strong ，因为它是默认设置的。 同时去思考一个需要多次访问弱属性的方法含义也是很重要的，就像下面这样：

```
– (void)someMethod {
[self.weakProperty doSomething];
...
[self.weakProperty doSomethingElse];
}
```

在这种情况下，你可能需要将弱属性存放在一个强变量中，从而确保它在你需要使用的过程中一直保存在内存中。

```
– (void)someMethod {
NSObject *cachedObject = self.weakProperty;
[cachedObject doSomething];
...
[cachedObject doSomethingElse];
}
```

在上面的例子中， cachedObject 包含了一个与初始弱属性值关联的强引用，所以只要 cachedObject 变量还在它的作用域中（同时没有被重新赋予其他值），弱属性就不会被释放。 你需要特别记住的是，如果想确保弱属性在使用前它的值不是 nil ，去测试它还远远不够，像下面这样：

```
if (self.someWeakProperty) {
    [someObject doSomethingImportantWith:self.someWeakProperty];
}
```

因为在多线程应用中，属性可能会在测试与方法调用之间被释放掉，这样测试就无效了。因此你需要声明一个强局部变量来保存值，像是这样：

```
NSObject *cachedObject = self.someWeakProperty;          // 1
if (cachedObject) {                                      // 2
    [someObject doSomethingImportantWith:cachedObject];  // 3
}                                                        // 4
cachedObject = nil;                                      // 5
```

在这个例子中，强引用在第一行被创建，意味着将会在测试和方法调用的过程中，保证对象活跃。在第5行，cachedObject 被设置成 nil ，这样强引用就被解除了，如果此时初始对象并没有其他与之相关的强引用，它将会被释放 someWeakProperty 也会被设置成 nil 。

对一些类使用不安全、无保留的引用

在Cocoa 和 Coacoa Touch 中还存在一些类，至今还不支持弱关联，这意味着你不能通过声明弱属性或弱变量来跟踪他们。这些类包括 NSTextView，NSFont 和 NSColorSpace，想获得完整的相关类列表，参看Transitioning to ARC Release Notes. 如果你想对这些类使用弱关联，你必须使用不安全引用。对于一个属性，这意味着将要使用 unsafe_unretained 特征：

```
@property (unsafe_unretained) NSObject *unsafeProperty;
```

对于变量，你需要使用 __unsafe_unretained:

```
NSObject * __unsafe_unretained unsafeReference;
```

一个不安全引用与弱关联之间的相同点在于，它们都不会保证相应对象的活动。但当目标对象被释放时，不安全引用不会被设置成 nil 。这意味着将会留下一个悬空指针，指向内存中一块开始存有对象之后被释放的区域，这就是所谓的“不安全”。对一个悬空指针发送消息将会引起崩溃。

备份属性(Copy Properties)保有他们自己的备份

在一些情况下，一个对象可能会希望保存一份，为它的属性设置的其他所有对象的备份。 举个例子，早前在图3-4中提到的 XYZBadgeView 类的接口可能会是这样：

```
@interface XYZBadgeView : NSView
@property NSString *firstName;
@property NSString *lastName;
@end
```

上例声明了两个 NSString 属性，他们都包含了与自己对象的不明确强引用。 当有另一个对象也创建了一个字符串来设置标志视图中的一个属性，考虑会发生的情况，

```
NSMutableString *nameString = [NSMutableString stringWithString:@"John"];
self.badgeView.firstName = nameString;
```

这是完全有效的，因为 NSMutableString 是 NSString 的一个子类。尽管标志视图认为它正在处理的是 NSString 实例，但实际上它处理的是 NSMutableString 。 这意味着字符串可以通过这样的方式变更：

```
[nameString appendString:@"ny"];
```

在这个例子中，尽管最开始时为标志视图 firstName 属性设置的名字值是“ John ”，但因为可变字符串（ mutable string ）值被改变了，所以它现在变成了“Johnny ”。 你可能会选择为标志视图保存一份他自己的，包含

所有为它的 firstName 属性和 lastName 属性设置的字符串值的备份，这样就可以有效的捕捉属性被设置时字符串的值。通过为这两个属性声明一个 copy 特征可以达到这样的目的：

```
@interface XYZBadgeView : NSView
@property (copy) NSString *firstName;
@property (copy) NSString *lastName;
@end
```

现在标志视图包含了自己关于这两个字符串的备份了。即使可变字符串后来改变了，标志视图获取的都还是这两个字符串最初被设定的值。例如：

```
NSMutableString *nameString = [NSMutableString stringWithString:@"John"];
self.badgeView.firstName = nameString;
[nameString appendString:@"ny"];
```

这次，被标志视图保存的 firstName 值将会是来自于一份保有初始“ John ”字符串的抗影响备份。 Copy 特征表明，属性因为需要与新创建的对象保持一致，而使用了强引用。

注： 任何你希望设置备份属性的对象都需要支持 NSCopying ，这意味着它需要遵守 NSCopying协议。协议的描述在 Protocols Define Messaging Contracts (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html#//apple_ref/doc/uid/TP40011210-CH11-SW2) ，获取更多关于 NSCopying 的信息可以参看 NSCopying 或者 Advanced Memory Management Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i) 如果你需要直接设置一个备份属性的实例变量，例如在初始器方法中，不要忘记为原始的对象设置一个备份：

```
- (id)initWithSomeOriginalString:(NSString *)aString {
self = [super init];
if (self) {
    _instanceVariableForCopyProperty = [aString copy];
}
return self;
}
```

练习

1、为 XYZPerson 添加一个 sayHello 方法，使用人的名和姓来加载一句打招呼的话。 2、声明并实现一个新的指定初始器（ designated initializer ），用来创建一个XYZPerson类，该类使用指定姓、名、和生日日期， 同时还包含合适的类制造方法（ class factory method ） 3、测试当你设置可变字符串（ mutable string ）做为

人名，然后在调用你添加的 sayHello 方法之前，改变人名字符串的值会出现的情况。为 NSString 属性声明添加备份特征，再测试一次。 4、尝试使用main() 函数中各种强、弱变量来创建 XYZPerson 对象。验证强变量会按照你期望的那样，保持XYZPerson对象活动。 为了验证一个 XYZPerson 对象是在何时被释放的，你可能会想通过在 XYZPerson 实现中添加一个 dealloc 方法，来将它与对象的生命周期关联起来。当一个ObjC 对象从内存中释放时这个方法会被自动调用，同时该方法也可以用于释放你手动分配的内存，就像C中的 malloc() 函数一样，参看Advanced Memory Management Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i)

为了这种目的的练习，覆写 XYZPerson 中的 dealloc 方法来加载消息，像这样：

```
- (void)dealloc {
NSLog(@"XYZPerson is being deallocated");
}
```

尝试通过设置 XYZPerson 中的每一个指针变量为 nil ，来验证对象如你所期待的那样被 释放了。

注： 在 Xcode 工程中，为命令行提供的样板，在 main() 函数内使用 @autoreleasepool { } 块，以达到使用编译器的自动保留计数功能，来为你处理内存管理。你在main() 函数中写的任何代码都会进入自动释放池（ autoreleasepool ），这点是非常重要的。

自动释放池在本文档中没有涉及，更多细节参看Advanced Memory Management Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i)

当你正在编写 Cocoa 或 Cocoa Touch 应用而不是命令行时，你不需要过多的担心关于创建你自己的自动释放池的问题，因为那样你就在尝试进入对象的构架中，而这个构架却可以保证这个池的正确存在。 5、更改类的描述，使你可以跟踪配偶（ spouse ）或伙伴（ partner ）。你需要考虑怎样才能最好的模拟这种关系——你可以认真思考一下对象图管理。

5

自定义现有的类 – Customizing Existing Classes

每一个对象都应该有明确的任务，如为具体信息建模，显示可视化内容或者控制信息流。 另外正如你所知道的，一个类的接口定义了其利用一个对象来帮助它完成任务的连接方式。

有时你会发现你希望通过添加某些方法来扩展现有的类，但这只在某些情况下是有用的。 举个例子，你发现你的应用程序经常需要在一个可视化界面显示一些字符串信息。 那么除了在每次你需要显示字符串时创造一个字符串绘图对象来使用外，给 NSString 类自己本身赋予可以在屏幕上绘制字符的能力会更有意义。

在这种情况下，对原始的、主要的类的接口增加功能方法并不总是可行的。 因为在大多数应用字符串对象的程序中，绘图能力并不总是被要求的。 例如，在 NSString 类中，你不能修改原来的接口或是继承，因为它是一个框架类。

此外，上述方法对现有类的子类也是没有意义的，因为你可能希望你的绘图能力不仅对原始的 NSString 类有效，也有对该类子类有效，如 NSMutableString 类。 另外虽然 NSString 类在 OS X 和 iOS 两个操作系统内均可使用，但相关绘图能力的代码在每个操作系统内是不同的，所以你需要在每个操作系统内使用不同的子类。

然而，Objective-C 允许你通过 categories 和类扩展来对已有的类中添加你自定义的方法。

使用 Categories 对现有的类添加方法

如果你需要添加方法到现有类，是为了添加某些功能使你自己的应用程序在完成某些人任务时更容易，那么使用 category 是最方便的方法。

使用 @ interface 关键字来声明一个 category ，就像标准的 Objective-C 类的描述一样，但并不表示这个 category 从任何一个子类继承。 另外它指定 category 的名称在括号内，像这样：

```
@interface ClassName (CategoryName)

@end
```

一个 category 可以声明任何类，即使是在没有原始代码的类（如标准的 Cocoa 或 Cocoa Touch 的类）。 你在 category 中声明的任何方法都可以被原始类和任何原始类的子类所实例化。 同时在运行时，你在 category 里添加的方法和由原始类实现的方法之间是没有区别的。

请考虑在之前章节提过的 XYZPerson 类，它具有一个人的姓氏和名字的属性。 如果你在写一个记录的应用程序，你会发现你经常需要显示一个按姓氏排列的名单，像这样：

```
Appleseed, John
Doe, Jane
Smith, Bob
Warwick, Kate
```

如果你不想在每次显示这个列表的时候再编写代码来生成适当的 lastName ， firstName 字符串， 那么你可以向 XYZPerson 类如下所示添加一个 category：

```
#import "XYZPerson.h"

@interface XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString;
@end
```

在此示例中，名为 XYZPersonNameDisplayAdditions 的 category 声明了一个额外的方法以返回必需的字符串。

一个 category 通常是在单独的头文件中声明的，并在单独的源代码文件被实现。 例如在 XYZPerson 类中，你可能会声明一个 category 在 XYZPerson+XYZPersonNameDisplayAdditions.h 的头文件中。

虽然用 category 添加的任何方法都可用于此类及其子类的所有实例中，但你仍需要在任何要使用添加的方法的源代码文件中导入含有 category 的头文件，否则你可能会遇到编译器警告和错误。

Category 的实现如下所示：

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"

@implementation XYZPerson (XYZPersonNameDisplayAdditions)
- (NSString *)lastNameFirstNameString {
   return [NSString stringWithFormat:@"%@, %@", self.lastName, self.firstName];
}
@end
```

一旦你已经声明一个 category 并继承这些方法，你可以在此类的任何实例中使用这些方法，就好像他们是原始类接口的一部分一样：

```
#import "XYZPerson+XYZPersonNameDisplayAdditions.h"
@implementation SomeObject
- (void)someMethod {
   XYZPerson *person = [[XYZPerson alloc] initWithFirstName:@"John"
                                          lastName:@"Doe"];
   XYZShoutingPerson *shoutingPerson =
                  [[XYZShoutingPerson alloc] initWithFirstName:@"Monica"
                                             lastName:@"Robinson"];

   NSLog(@"The two people are %@ and %@",
      [person lastNameFirstNameString], [shoutingPerson lastNameFirstNameString]);
}
@end
```

除了可以向现有的类添加方法，你还可以使用 categories 把多功能的源代码文件中一个复杂的类拆分。 例如，你把一个自定义用户界面元素的绘图代码放在一个单独的 category 中，来执行一些其余的功能如几何计算、 颜色和渐变等，这就是一个特别复杂的类的例子。 另外你可以给类别的方法提供不同的实现，具体取决于你正在写一个 OS X 还是 iOS 的应用程序。

Categories 可用于声明类方法或成员方法，但并非通常适合声明附加属性。 在一个 category 的接口中包含属性声明时编译器不会报错，但是不能在一个 category 中声明一个附加的成员变量。 这意味着，编译器不会为该属性合成任何成员变量，也不合成任何属性访问方法。 在类的实现过程中，你可以编写你自己的访问方法，但是你不能来跟踪该属性的值，除非原始类中已有了该成员变量。

添加一个传统属性的唯一方式——也就是从现有类支持一个新的成员变量——是使用类扩展，如 Class Extensions Extend the Internal Implementation (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html#//apple_ref/doc/uid/TP40011210-CH6-SW3) 。

> 注： Cocoa 和 Cocoa Touch 包括了大量的主要框架类的 categories。
>
> 这一章的导言中提到的字符串绘图功能事实上已经由 OS X 中名为 NSStringDrawing 的 category 提供给 NSString 类了，其中包括 drawAtPoint:withAttributes: 和 drawInRect:withAttributes: 方法。 对于 iOS ，UIStringDrawing category 包括 drawAtPoint: withFont 方法和 drawInRect: withFont 方法。

避免 categories 方法名冲突

因为在一个 category 中声明的方法已经添加到现有的类中，所以你需要非常小心有关方法名的定义问题。

如果在一个 category 中声明的方法和在原始类中的方法或该类（甚至是在一个父类）的其他 category 中的方法名称相同，在运行时，编译哪种方法的指令将被认为是未定义的。 如果你正在使用你自己的类的 categories ，使用的 categories 会将方法添加到标准的 Cocoa 或 Cocoa Touch 类时导致问题。

例如当你的应用程序与远程 web 服务交互时，可能需要一种使用 Base64 编码技术来编码字符串的方法。 因此你可以通过在 NSString 类上定义一个 category ，添加一个称为 base64EncodedString 的实例方法以返回一个 Base64 编码的字符串。

但是如果你链接到另一个框架，恰巧也在 NSString 类的自定义 category 中包括了此方法 也称为 base64EncodedString 时 ，那么将会出现问题。 在运行时，只有一个方法会“赢”，并添加到 NSString 类中，另一个则成为未定义不起作用。

如果你添加方法到 Cocoa 或 Cocoa Touch 类和之后版本的原始类中，那么可能会出现另一个问题。

例如 NSSortDescriptor 类，它描述了一个对象的集合应该是如何排序的，包含有 aninitWithKey: accending 初始化方法。

但并没有在早期的 OS X 和 iOS 版本下提供相应的工厂类方法。

按照约定，工厂类方法应该叫做 sortDescriptorWithKey: accending ，所以为方便起见你要选择添加一个 category 到 NSSortDescriptor 类上来提供此方法。 这是在旧版本的 OS X 和 iOS 下操作的，但随着 Mac OS X 10.6 版本和 iOS 4.0 的发布，一个叫 sortDescriptorWithKey 的方法添加到原始的 NSSortDescriptor 类中，意味着在这些或更高版本操作系统上运行你的应用程序时，你不再会有命名冲突的问题。

为了避免未定义的行为，最佳的做法是给框架类 categories 中的方法名添加一个前缀，就像你向你自己的类的名称添加一个前缀一样。

你可以选择使用和你自己的类的前缀相同的三个字母，但要小写以遵循方法命名的规则，然后在方法名称的其余部分之间用一个下划线连接。

对于 NSSortDescriptor 的示例，你的 category 应该看起来像这样：

```
@interface NSSortDescriptor (XYZAdditions)
 (id)xyz_sortDescriptorWithKey:(NSString *)key ascending:(BOOL)ascending;
@end
```

这意味着你可以肯定你的方法在运行时可以使用。歧义将会被删除，你的代码现在看起来像这样：

```
  NSSortDescriptor *descriptor =
        [NSSortDescriptor xyz_sortDescriptorWithKey:@"name" ascending:YES];
```

用 extension 来实现类的扩展

类扩展与 category 有相似性，但在编译时它只能被添加到已有源代码的一类中（该类扩展和该类同时被编译）。

声明一个类扩展的方法在原始类 @ implementation 块中，所以你不能，举个例子，在框架类上声明一个类扩展，如 Cocoa 或 Cocoa Touch 的 NSString 类。

用于声明类扩展的语法类似于一个 category 声明的语法，看起来像这样：

```
@interface ClassName ()

@end
```

因为没有在括号内给定名称，所以类扩展通常称为匿名类。

不像一般的 categories ，类扩展可以向类中添加其自己的属性和成员变量。如果你在类扩展中声明一个属性，要像这样：

```
@interface XYZPerson ()
@property NSObject *extraProperty;
@end
```

编译器会自动合成相关的访问方法，以及一个成员变量，继承到主要的类。

如果你在一个类扩展中添加任何方法，这些必须在主要类中继承。

也可以使用一个类扩展来添加自定义的成员变量。这些变量在类扩展接口中的大括号内声明：

```
@interface XYZPerson () {
    id _someCustomInstanceVariable;
}
...
@end
```

使用类扩展来隐藏私有信息

一个类的主要接口用于定义其他类将与之进行交互的方式。换句话说，它是类的公共部分。

类扩展通常用于扩展额外的私有方法或属性的公共接口以便在类本身的实现中使用。 例如，通常在界面中定义一个只读属性，但是为了在类的内部方法可以直接更改属性值，在继承上层的一个类扩展声明中定义该属性为读写属性。

举个例子，XYZPerson 类可以添加一个称为 uniqueIdentifier 的属性，用于跟踪信息，比如在美国的社会安全号码。

在现实世界中它通常需要大量的文书工作来给每一个人分配唯一的标识符，所以 XYZPerson 类接口可能会声明此属性为只读，并提供一些方法请求标识符分配，像这样：

```
@interface XYZPerson : NSObject
...
@property (readonly) NSString *uniqueIdentifier;
- (void)assignUniqueIdentifier;
@end
```

这意味着 uniqueIdentifier 不可能直接由另一个对象设置。 如果一个人还未有一个唯一的标识符，那么通过调用 assignUniqueIdentifier 方法将会作出分配一个标识符的请求。

为了 XYZPerson 类能够更改其内部的属性值，可以通过在类扩展中重新定义在顶层类继承的文件中被定义的属性值来实现：

```
@interface XYZPerson ()
@property (readwrite) NSString *uniqueIdentifier;
@end

@implementation XYZPerson
...
@end
```

> 注： 读写属性是可选的，因为它是默认值。为清楚起见你可以在想使用它时重新声明属性。

这意味着编译器现在将合成一个 setter 方法，所以在 XYZPerson 类执行内部的任何方法都能够直接使用 setter 方法或语法来设置该属性值。 通过为 XYZPerson 类继承的源代码文件声明类扩展， 使得 XYZPerson 类的信息是私有的。 如果另一种类型的对象试图设置该属性时，编译器将生成一个错误。

> 注： 如上所示通过添加类扩展，重新定义 uniqueIdentifier 属性为读写属性，一个名为 setUniqueIdentifier: 的方法将在运行时在每个 XYZPerson 对象上存在，无论其他源代码文件是否知道该类扩展的存在。

当其他源代码文件中的某段代码试图调用一个私有方法或设置一个只读属性的值时，编译器会报错，但利用动态运行功能使用其他方式调用这些方法是可以避免编译器错误的，例如通过使用由 NSObject 类提供的 performSelector 的方法。 你应该避免出现一个类的层次结构或者仅在必须的时候使用；相反主类接口应始终定义正确的"公共接口"。

> 如果你打算在选择其他类别时，“私有”方法或属性仍是可用的，例如在一个框架内的相关类中。 你可以在单独的头文件中声明一个类扩展，并在需要它的源文件中导入它。在一个类中有两个头文件并不罕见，例如， XYZPerson.h 和 XYZPersonPrivate.h 等。 当你释放框架时，你只需释放公共的 XYZPerson.h 头文件即可。

考虑其他办法来自定义类

Categories 和类扩展使得直接添加方法到一个现有的类变得很容易，但有时这并不是最好的选择。

面向对象编程的主要目标之一是编写可重用的代码，这意味着在各种情况下所有的类都尽可能地被重复使用。

如果你正在创建一个视图类来描述一个对象用于在屏幕上显示信息，那考虑一下这个类能在多种情况下可用是必要的。

除了将关于布局或内容的部分硬编码，一种可选择的方法是利用继承并将这些部分留在方法中，特别是子类重写的方法中。

虽然重用类并不会相对容易，因为每次你想要使用的那个原始的类时你仍然需要创建一个新的子类。

另一种选择是要对类使用一个 delegate 对象。

任何可能会限制可重用性的部分都可授权给另一个对象，也就是说可以在运行时编译这些部分。 一个常见的例子是标准表视图类 (OS X 的 NSTableView 和 iOS 的 UITableView)。 为了使一般表格视图 （ 使用一个或多个列和行显示信息的对象) 可用，它将内容部分留给另一个对象在运行时决定。 在下一章 Working with Protocols (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html#//apple_ref/doc/uid/TP40011210-CH11-SW1) 中会详细的介绍如何使用授权 。

直接与 Objective-C 运行库进行交互

Objective-C 通过其运行库系统提供动态功能。

许多决定并不在编译时作出，而在应用程序运行时决定，例如哪些方法调用时会发送消息的决定。 Objective- C 不仅仅是一种编译机器的语言代码，而且它还需要一个运行库系统来执行代码。

它是可以直接与运行库系统进行交互的，例如给对象添加关联引用。 不同于类扩展，关联引用不会影响原始类的声明和继承，这意味着你可以将它们用于你没有权限访问的原始源代码的框架类。

一个关联引用是用来链接两个对象的，类似于一个属性和成员变量。 获取更多的信息，请参阅关联引用部分。若要了解更多有关 Objective-C 的内容，请参阅 Objective-C Runtime Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008048) 。

练习

1. 添加一个 category 到 XYZPerson 类来声明和继承附加的功能，例如以不同的方式显示一个人的名字。
2. 向 NSString 类 添加一个 category，以添加一个方法来在给定位置绘制全部字母大写的字符串，通过调用到一个现有的 NSStringDrawing category 方法来执行实际的绘制。
   这些方法都记录在 iOS 的 NSString UIKit Additions Reference 中和 OSX 的 NSString Application Kit Additions Reference (https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ApplicationKit/Classes/NSString_AppKitAdditions/index.html#//apple_ref/doc/uid/TP40004121) 中。

3. 将两个只读属性添加到原始 XYZPerson 类的继承中，来代表一个人的身高和体重，分别是 measureWeight 和 measureHeight 方法。
   使用类扩展重新声明属性为读写属性，并继承以便将属性设置为适当的值。

6

使用协议 – Working with Protocols

在现实世界中，公务人员在处理某些情况时往往需要遵循严格的程序。例如，执法人员在进行询问或收集证据时要“遵循协议”。

在面向对象编程的世界，重要的是在一个给定的情况下能够定义一组对象的行为。举个例子，一个表视图为了查明它需要显示什么内容，希望能够与一个数据源对象通信。这意味着数据源必须响应表视图可能发送的一组特定的消息。

数据源可以是任何类的一个实例，比如视图控制器(子类 NSViewController OS X 或 UIViewController iOS)或者是一个刚从 NSObject 继承的专用的数据源类。为了使表视图知道一个对象是否为合适的数据源，重要的是能够声明对象实现的必要方法。

Objective-C 允许您定义协议，声明的方法将被用于一个特定的情况。本章介绍了定义一个正式协议的语法，并解释了如何标记一个类界面使其符合一个协议，这意味着该类必须执行要求的方法。

协议定义消息传递合同

一个类的接口声明这个类的方法和属性。相比之下，一个协议声明的方法和属性，独立于任何特定的类。

定义一个协议的基本语法如下:

```
@protocol ProtocolName
// list of methods and properties
@end
```

协议可以包括声明实例方法和类方法，以及属性。

作为一个例子，考虑一个定制的视图类，用于显示一个饼图，如图 5-1 所示。

图 5-1 饼图自定义视图

图片 6.1 image

为了使视图尽可能重用，所有的决策信息应该留给另一个作为数据源的对象。这意味着相同的视图类的多个实例可以显示不同的信息仅仅通过与不同数据源的交流。

饼图视图所需的最小信息包括分块的数量，每个分块的相对大小，每个分块的标题。饼图的数据源协议，因此，看起来像这样:

```
@protocol XYZPieChartViewDataSource
– (NSUInteger)numberOfSegments;
– (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
– (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

注:本协议使用无符号整数的 NSUInteger 值标量值。这种类型在下一章详细讨论。饼图视图类接口需要有一个记录数据源对象的属性。这个对象可以是任何类，所以基本属性类型为 id 。关于对象唯一知道的是其符合相关协议。

声明数据源属性视图的语法应该像这样:

```
@interface XYZPieChartView : UIView
@property (weak) id <XYZPieChartViewDataSource> dataSource;
...
@end
```

Objective-C 使用尖括号来表示与协议的一致性。本例为一个通用类对象指针声明一个弱属性符合 theXYZPieChartViewDataSource 协议。

> 注:接口和数据源属性通常标记为弱，根据前面提到的对象图管理原因，避免强引用周期。

指定与所需协议一致的属性，如果您试图将属性设置为一个对象， 这是不符合协议的，你会得到一个编译器警告，尽管基本属性类的类型是通用的。对象是否为 UIViewController orNSObject 的一个实例是无关紧要的。最重要的是，它符合协议，这意味着饼图视图知道它可以请求所需要的信息。

协议有可选择的方法

默认情况下，在一个协议中声明的所有方法都是需要的方法。这意味着符合协议的任何类必须实现这些方法。

在协议里指定可选方法也是可能的。这些方法是一个类只要需要就可执行的。

例如，您可能会决定，饼图的标题应该是可选的。如果数据源对象不实现 titleForSegmentAtIndex: ，则应该没有标题显示在视图中。

您可以使用 @optional 指令协议方法标记为可选，如下:

```
@protocol XYZPieChartViewDataSource
– (NSUInteger)numberOfSegments;
– (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
– (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

本例中，只有 titleForSegmentAtIndex: 方法标记为可选。前面的方法没有指令，所以被认为是必需的。

@optional 指令适用于遵循它的任何方法 ，直到协议定义的最后，或者遇到另一个指令之前，例如 @required。你可能会添加进一步的方法到协议中，如下:

```
@protocol XYZPieChartViewDataSource
– (NSUInteger)numberOfSegments;
– (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
@optional
– (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
– (BOOL)shouldExplodeSegmentAtIndex:(NSUInteger)segmentIndex;
@required
– (UIColor *)colorForSegmentAtIndex:(NSUInteger)segmentIndex;
@end
```

这个例子定义了一个有三种必需方法和两种可选方法的协议。

运行时检查可选方法的实现

如果一个方法在协议中被标记为可选的，您必须检查是否有对象在实现之前试图调用它。

例如，饼图视图测试分块标题的方法可能是这样的:

```
NSString *thisSegmentTitle;
    if ([self.dataSource respondsToSelector:@selector(titleForSegmentAtIndex:)]) {
        thisSegmentTitle = [self.dataSource titleForSegmentAtIndex:index];
    }
```

respondsToSelector:方法使用一个选择器，引用编译后的标识符的方法。您可以使用 @selector() 指令和指定方法的名称来提供正确的标识符。

如果在本例中数据源实现了这个方法，那么标题被使用;否则，标题仍然是零。

> 切记:本地对象变量自动初始化为零。

如果您试图调用有一个协议中 id 的 respondsToSelector: 方法，你会得到一个没有已知的实例方法的编译错误。一旦你有了一个协议中的 id ，所有静态类型检查复原，如果你试图调用任何未在指定协议中声明的方法，系统会报错。避免编译错误的一种方法是设置自定义协议采用 NSObject 协议。

协议继承其他协议

一个 Objective-C 类可以继承一个父类，以同样的方式，你还可以指定一个协议符合另一个协议。

作为一个例子，最佳的实践是定义您的协议符合 NSObject 协议(一些 NSObject 行为从它们的类接口划分到一个单独的协议; NSObject 类采用 NSObject 协议)。

通过表明自己的协议符合 NSObject 协议，这表明任何采用自定义协议的对象，还将提供每个 NSObject 协议方法的实现。因为你可能使用一些 NSObject 的子类，你不需要担心为这些 NSObject 提供自己的实现方法。不管怎样，对于前述的情况，采用的协议是有效的。

指定一个协议符合另一个协议，您需提供其他协议的名称在尖括号内，像这样:

```
@protocol MyProtocol <NSObject>
...
@end
```

在这个例子中，任何采用 MyProtocol 的对象，也有效地采用了 NSObject 协议中声明的所有方法。

符合协议

表示一个类采用了协议需再次使用尖括号括起来，像这样:

```
@interface MyClass : NSObject <MyProtocol>
...
@end
```

这意味着任何 MyClass 实例不仅响应在接口中特意声明的方法， MyClass 还在 MyProtocol 中提供了所需方法的实现。不需要在类的接口重新定义协议方法，采用协议就足够了。

> 注:编译器不会自动合成在采用的协议里声明的属性。

如果您需要一个类采用了多种协议，你可以用一个逗号分隔，像这样:

```
@interface MyClass : NSObject <MyProtocol， AnotherProtocol， YetAnotherProtocol>
...
@end
```

> 提示:如果您发现自己在一个类中采用大量的协议，它可能是一个信号，表明您需要重构那个过于复杂的类，可以通过必要的行为将其拆分到多个小类，每个小类都有明确的责任。

对新 OS X 和 iOS 开发者来说，一个相对常见的困难是使用一个应用程序委托类去包含一个应用程序的大部分功能(管理底层数据结构，提供数据到多个用户界面元素，以及响应手势和其他用户交互)。随着复杂性的增加，类变得更难以维护。

一旦您表明遵循某个协议，类必须至少为每个所需的协议方法提供实现方法，以及您选择的任何可选的方法。如果不能实现任何所需的方法，编译器会提醒您。

> 注:协议中的方法声明类似其他任何声明。协议中实现的方法名和参数类型必须与声明匹配。

Cocoa和Cocoa Touch定义大量的协议

Cocoa 和 Cocoa Touch 使用的协议针对各种不同情况的对象。例如，表视图类(NSTableView OS X and UITableView iOS)都使用一个数据源对象来为他们提供必要的信息。两者都定义自己的数据源协议，与上面的例子 XYZPieChartViewDataSource 协议使用差不多的方法。两者的表视图类还允许您设置一个委托对象，又必须符合相关 NSTableViewDelegate 或 UITableViewDelegate 协议。委托对象负责处理用户交互，或定制化显示某些条目。

一些协议是用于表示类之间没有相似之处。不是与特定类需求关联，一些协议与更普遍的 Cocoa 和 Cocoa Touch 通信机制关联，可能会采用多个不相关的类。

例如，许多框架模型对象(如集合类，如 NSArray 和 NSDictionary)支持 NSCoding 协议，这意味着它们可以编码和解码档案的属性或分布为原始数据。NSCoding 使它相对容易的将整个对象图写到磁盘中，提供每个对象采用协议的图表。

一些 Objective-C 语言级特性也依靠协议。为了使用快速枚举，例如，集合必须采用 NSFastEnumerationprotocol 协议，如 Fast Enumeration Makes It Easy to Enumerate a Collection 中所述。此外，一些对象可以被复制， 例如在使用一个属性和一个复制属性时，如 Copy Properties Maintain Their Own Copies 所述。你试图复制的任何对象必须采用 NSCopying 协议，否则你会得到一个运行异常。

协议用于匿名

对象的类不是已知的，或需要被隐藏的情况下协议也是很有用的。

比如，一个框架的开发人员可能会选择不发布框架内一个类的接口。因为类名称不清楚，框架的用户直接创建这个类的一个实例是不可能的。相反，框架内一些其他对象通常会指定返回一个现成的实例，像这样:

```
utility = [frameworkObject anonymousUtility];
```

为了让这个 anonymousUtility 对象是有用的，框架的开发人员就可以发布一个协议，显示它的一些方法。但不提供原始类接口，这意味着类保持匿名，对象仍是在有限的方式内被使用:

```
id <XYZFrameworkUtility> utility = [frameworkObject anonymousUtility];
```

如果您在写一个 iOS 应用程序，使用核心数据框架，例如，您可能会遇到 NSFetchedResultsController 类。这个类是为了帮助一个数据源对象提供存储数据到一个 iOS UITableView ，便于提供信息的行数。

如果您正在使用的表视图内容分为多个部分，您也可以访问一个获取结果控制器获取相关部分信息。而不是返回一个特定的类包含这部分信息， NSFetchedResultsController 类相反是返回一个匿名对象，它符合 NSFetchedResultsSectionInfo 协议。这意味着它仍然可以查询你需要的对象的信息，如一部分内的行数:

```
NSInteger sectionNumber = ...
    id <NSFetchedResultsSectionInfo> sectionInfo =
        [self.fetchedResultsController.sections objectAtIndex:sectionNumber];
    NSInteger numberOfRowsInSection = [sectionInfo numberOfObjects];
```

即使您不知道 sectionInfo 对象的类， NSFetchedResultsSectionInfo 协议规定，它可以应对 thenumberOfObjects 信息。

7

赋值与集合 – Values and Collections

尽管 Objective-C 是一种面向对象的编程语言，它也是C语言的加强版，这意味着你可以在 Objective-C 中使用标准 C 中任意的纯量类型（非对象的），例如 int，float，char。在 Cocoa 和 Cocoa Touch 应用中，你还可以使用一些额外的类型，例如 NSInteger，NSUInteger 和 CGFloat，在不同的系统结构中，他们的定义方式也不同。

纯量类型用于当你不需要用一个对象来表示值的时候。字符型经常作为 NSString 的实例而使用，数值被存储在纯量类型的局部变量或属性中。

你可以在 Objective-C 中定义与 C 语言类似的数组，但是在 Cocoa 和 Cocoa Touch 应用中，集合被用于实例化像 NSArray 或 NSDictionary 这样的类。这些类只能存放对象，这意味着在添加对象到集合之前，你就要用 NSValue, NSNumber 或 NSString 这样的类为对象赋好值。

在前面的章节中，我们多次使用了 NSString 类、它的初始化、方法函数库，@"string" 字符为创建 NSString 的实例提供了简单的语法。在本章中，我们会通过方法调用和赋值语句来示范如何使用 NSValue 和 NSNumber 类。

你可以使用C中的基本数据类型

在 Objective-C 中，C 的每一个 scalar 变量类型都是可以使用的：

```
int someInteger = 42;
float someFloatingPointNumber = 3.1415;
double someDoublePrecisionFloatingPointNumber = 6.02214199e23;
```

还有 C 的操作符也都可用：

```
int someInteger = 42;
someInteger++;          // someInteger == 43

int anotherInteger = 64;
anotherInteger--;       // anotherInteger == 63

anotherInteger *= 2;    // anotherInteger == 126
```

如果你要为 Objective-C 属性使用纯量类型 ，请这样做：

```
@interface XYZCalculator : NSObject
@property double currentValue;
@end
```

你也可以对属性使用 C 操作符，用点语法进行赋值操作，就像这样：

```
@implementation XYZCalculator
- (void)increment {
    self.currentValue++;
}
- (void)decrement {
self.currentValue--;
}
- (void)multiplyBy:(double)factor {
self.currentValue *= factor;
}
@end
```

点语法是一种纯粹用于存取器（accessor）调用方法的语法，所以这个例子中的每一条操作都是先使用 get accessor 方法获取实例变量值，运行程序后，再使用 set accessor 方法存储实例变量值作为结果。

Objective-C中其他的基本类型

BOOL 类型在 Objective-C 中用来表示布尔值：yes 和 no。正如你想的那样，yes 在逻辑上等于 true 和 1，no 等于 false 和 0；

在 Cocoa 和 Cocoa Touch 中，许多方法的参数也可以使用特殊的数据类型，例如 NSInteger 和 CGFloat。

例如， NSTableViewDataSource 和 UITableViewDataSource 协议（在之前章节提到的）都有令数据按行显示的方法：

```
@protocol NSTableViewDataSource <NSObject>
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView;
...
@end
```

像 NSInteger 和 NSUInteger 这样的类型，在不同的系统结构中有不同的定义方式。当为 32 位运行环境（例如 IOS）编程时，他们分别是 32 位的有符号整数和无符号整数；当为 64 位运行环境（例如 modern OS X runtime）编程时，他们分别是 64 位的有符号整数和无符号整数。

如果你希望在 API 边界间（包括内外部 API）传值，最好使用这种特定平台的类型，例如在你的应用代码和平台架构间，利用方法、函数调用传递参数或返回数值。

对于局部变量，例如循环中的计数值，如果你知道该值作用域的限制，使用 C 的基本类型定义它也是可以的。

C的结构可以存储基本数据

一些 Cocoa and Cocoa Touch API 使用 C 的数据结构存储数值。例如，一个字符串对象可以作为一个子字符串的取值域，就像这样：

```
NSString *mainString = @"This is a long string";

NSRange substringRange = [mainString rangeOfString:@"long"];
```

一个 NSRange 结构保存了一个地址和一个长度值。在这个例子中，substringRange 中会保存 {10,4} 这两个值：以 0 开始计算的位置，主字符串中的第 10 个字母为子字符串 @"long" 的第 1 个字母，@"long" 的长度为 4。

相似的是，如果你要编写自定义图形代码，你需要使用 Quartz，它要使用以 CGFloat 数据类型为基础的结构，就像 OS X 上的 NSPoint 和 NSSize ，IOS 上的 CGPoint 和 CGSize。CGFloat 也同样在不同的系统结构上有不同的定义方式。

想看更多关于 Quartz 2D 绘制机制？点击Quartz 2D Programming Guide (https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html#//apple_ref/doc/uid/TP30001066)

对象可以表示基本数值

如果你需要用 scalar 表示一个对象的值，你可以使用 Cocoa 和 Cocoa Touch 提供的基础赋值类，例如我们在下一个部分里面提到的集合类。

通过NSString类的实例表示Strings

就像之前章节里面提到的，NSString 是用来表示字符串的，例如 Hello World。创建 NSString 对象有很多种方式，包括分配空间并初始化，使用工厂方法或者纯语法：

```
NSString *firstString = [[NSString alloc] initWithCString:"Hello World!"

encoding:NSUTF8StringEncoding];

NSString *secondString = [NSString stringWithCString:"Hello World!"
```

```
encoding:NSUTF8StringEncoding];

NSString *thirdString = @"Hello World!";
```

每一个例子都高效的完成了同一件事：创建了一个字符串对象，其值为给定内容。

基本 NSString 类是不可变的，它的内容在创建时设好就不能再改变了。如果你想要一个不同的字符串，你必须创建一个新的对象，就像这样：

```
NSString *name = @"John";

name = [name stringByAppendingString:@"ny"]; // returns a new string object
```

NSMutableString 类是 NSString 的可变子类，允许在方法中修改，与 appendString 和 appendFormat 相似：

```
NSMutableString *name = [NSMutableString stringWithString:@"John"];

[name appendString:@"ny"];  // same object, but now represents "Johnny"
```

使用Format Strings创建来自其他对象和值的Strings

如果你需要创建一个带有变量的字符串，你要使用 format string。它允许你使用格式符来表示值是如何插入的：

```
int magicNumber = ...
NSString *magicString = [NSString stringWithFormat:@"The magic number is %i", magicNumber];
```

可用的格式符在这里String Format Specifiers (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html#//apple_ref/doc/uid/TP40004265) 。想了解更多关于strings，请点击String Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Strings/introStrings.html#//apple_ref/doc/uid/10000035i) 。

通过NSNumber类的实例表示数字

NSNumber 类用来表示所有 C 基础的 scalar 类型，包括 char，double，float，int，long，short 以及它们的无符号变量，还有 Objective-C 中的布尔类型，BOOL。

使用 NSString，你有很多种方式可以创建 NSNumber 实例，包括空间分配并初始化，以及使用工厂方法：

```
NSNumber *magicNumber = [[NSNumber alloc] initWithInt:42];
NSNumber *unsignedNumber = [[NSNumber alloc] initWithUnsignedInt:42u];
NSNumber *longNumber = [[NSNumber alloc] initWithLong:42l];
NSNumber *boolNumber = [[NSNumber alloc] initWithBOOL:YES];
NSNumber *simpleFloat = [NSNumber numberWithFloat:3.14f];
NSNumber *betterDouble = [NSNumber numberWithDouble:3.1415926535];
NSNumber *someChar = [NSNumber numberWithChar:'T'];
```

你也可以用 Objective-C 中的语法来创建 NSNumber 实例：

```
NSNumber *magicNumber = @42;
NSNumber *unsignedNumber = @42u;
NSNumber *longNumber = @42l;
NSNumber *boolNumber = @YES;
NSNumber *simpleFloat = @3.14f;
NSNumber *betterDouble = @3.1415926535;
NSNumber *someChar = @'T';
```

这些例子都等同于使用 NSNumber 类的工厂方法。

一旦你已经创建了一个 NSNumber 实例，你就可以使用 accessor 方法申请一个纯量值：

```
int scalarMagic = [magicNumber intValue];
unsigned int scalarUnsigned = [unsignedNumber unsignedIntValue];
long scalarLong = [longNumber longValue];
BOOL scalarBool = [boolNumber boolValue];
float scalarSimpleFloat = [simpleFloat floatValue];
```

```
double scalarBetterDouble = [betterDouble doubleValue];

char scalarChar = [someChar charValue];
```

NSNumber 类也为 Objective-C 额外的基本类型提供了方法。如果你想创建 NSInteger 和 NSUInteger 的对象，请使用如下方法：

```
NSInteger anInteger = 64;

NSUInteger anUnsignedInteger = 100;

NSNumber *firstInteger = [[NSNumber alloc] initWithInteger:anInteger];

NSNumber *secondInteger = [NSNumber numberWithUnsignedInteger:anUnsignedInteger];

NSInteger integerCheck = [firstInteger integerValue];

NSUInteger unsignedCheck = [secondInteger unsignedIntegerValue];
```

所有的 NSNumber 实例都是不可变的，并且它也没有可变的子类；如果你需要一个不同的数字，请使用另一个 NSNumber 实例。

> 提示
>
> NSNumber 实际上是一个类集。这意味着当你在运行时创建了一个实例时，你会得到一个合适的具体化的子类，值为给定值。只把创建的对象当做一个 NSNumber 的实例就可以了。

使用NSValue类的实例表示其他值

NSNumber 类是基础 NSValue 类的子类，它给单独的数据或数据项提供了包装对象。除了 C 的基本纯量类型，NSValue 也可以表示指针和结构体。

NSValue 类提供了很多的工厂方法来创建一个给定结构的值，这使得创建一个实例变得十分简单，例如之前例子中的 NSRange：

```
NSString *mainString = @"This is a long string";

NSRange substringRange = [mainString rangeOfString:@"long"];

NSValue *rangeValue = [NSValue valueWithRange:substringRange];
```

你也可以使用 NSValue 来创建自定义的结构体对象。如果你一定要用 C 中结构体存储信息，请这样做：

```
typedef struct {
    int i;
    float f;
} MyIntegerFloatStruct;
```

你可以通过指向结构体的指针或编好的 Objective-C 类型来创建一个 NSValue 实例。@encode() 这条编译器指令是用来创建正确的 Objective-C 类型的，如：

```
struct MyIntegerFloatStruct aStruct;

aStruct.i = 42;

aStruct.f = 3.14;

NSValue *structValue = [NSValue value:&aStruct
                         withObjCType:@encode(MyIntegerFloatStruct)];
```

标准 C 中的 & 符号在这里表示 aStruct 中 value 参数的地址。

大多数集合都是对象

尽管你可以使用 C 中的数组存放 scalar 数据的集合，甚至还可以存放对象指针，大多数 Objective-C 代码中的集合都是 Cocoa 和 Cocoa Touch 集合类中的一个，例如 NSArray, NSSet 和 NSDictionary。

这些类用来管理一组对象，这意味着你加入集合的任何一项都必须是 Objective-C 类的实例。如果你需要添加一个 scalar 值，你必须要先创建一个合适的 NSNumber 或 NSValue 实例。

集合类使用强大的引用持续追踪他们的内容，而不是不管怎样都为集合中的每一个对象做一份拷贝。这意味着你加入集合中的每一个对象的生命周期都将至少和集合的生命周期一样长， 正如这里描述的Manage the Object Graph through Ownership and Responsibility (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple_ref/doc/uid/TP40011210-CH5-SW3)

除了追踪他们的内容，Cocoa 和 Cocoa Touch 中的每一个集合类都可以简单地完成一些任务，例如列举，存取特定的项，查找某一个对象是否在这个集合中。

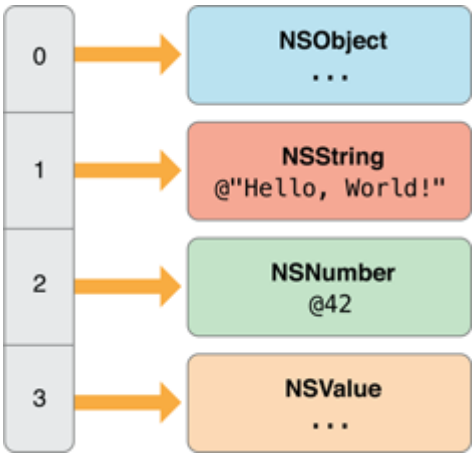
基本的 NSArray, NSSet 和 NSDictionary 类都是不可变的，它们的值在创建时就固定了。但它们都拥有可变的子类，你可以通过子类来添加和删除对象。

想查看更多 Cocoa 和 Cocoa Touch 中的集合类，请点击Collections Programming Topics (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Collections/Collections.html#//apple_ref/doc/uid/10000034i)

数组是有序的集合

NSArray 是用来表示对象的有序集合。对集合中内容唯一的要求是每一项都要是 Obejctive-C 的对象，但它们并不需要是同一个类的实例。

为了满足有序，每一个元素的存储都是从 0 开始记序，就像图 6-1 这样。



图片 7.1 图6-1 Objective-C 对象数组

创建数组

就像之前章节里写的赋值类那样，你可以通过分配空间并初始化，使用工厂类方法或纯语法方式来创建数组。

初始化方法和工厂方法有很多种，你可以根据对象的个数来选择不同的方法：

```
+ (id)arrayWithObject:(id)anObject;

+ (id)arrayWithObjects:(id)firstObject, ...;

- (id)initWithObjects:(id)firstObject, ...;
```

arrayWithObjects 和 initWithObjects: 这两种方法都采用零终止（nil-terminated），参数个数是可变的，这意味着你数组的最后一个值必须是 nil：

```
NSArray *someArray =
[NSArray arrayWithObjects:someObject, someString, someNumber, someValue, nil];
```

这个例子创建了一个类似图6-1中的数组。第一个对象：someObject，下标是 0；最后一个对象：someValue，下标是 3。

如果数组中有一个值为 nil，那么这个数组会在这里被截断：

```
id firstObject = @"someString";

id secondObject = nil;

id thirdObject = @"anotherString";

NSArray *someArray =
[NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];
```

在这里，someArray 数组只包含 firstObject，因为第二个元素为 nil，系统认为它是数组的结束。

语法

你可以用 Objective-C 的语法创建数组：

```
NSArray *someArray = @[firstObject,secondObject, thirdObject];
```

在这里你不能用 nil 来结束数组，事实上 nil 是非法的值。如果你运行下面的代码，系统会抛出异常：

```
id firstObject = @"someString";

id secondObject = nil;

NSArray *someArray = @[firstObject, secondObject];
// exception: "attempt to insert nil object"
```

如果你一定要使用 nil 值，你可以用 NSNull 类，详情请查看Represent nil with NSNull (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/FoundationTypesandCollections/FoundationTypesandCollections.html#//apple_ref/doc/uid/TP40011210-CH7-SW34)

查找数组对象

一旦你创建了数组，你可以查询有关的信息，例如对象个数或查找特定的元素：

```
NSUInteger numberOfItems = [someArray count];

if ([someArray containsObject:someString]) {
    ...
}
```

你也可以查询指定下标的元素。如果你查询的下标数是错误的，系统会抛出越界异常，所以你应该首先确认数组长度：

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

这个例子检查了数组长度是否大于 0。如果大于 0，它会记录下第一个元素，也就是下标为 0 的元素信息。

下标

还有另一种下标写法，使用 objectAtIndex：这与 C 中数组写法十分像。使用这种方法重写前面的例子：

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

给数组对象排序

NSArray 类提供了很多种排序的方法。因为 NSArray 是不可变的，所以所有的方法都会返回一个有序数组。

例如，你可以通过调用 compare 给字符串排序：

```
NSArray *unsortedStrings = @[@"gammaString", @"alphaString", @"betaString"];
NSArray *sortedStrings =
        [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

可变性

尽管 NSArray 类是不可变的，这并不会影响集合中的对象。如果你想添加一个可变的字符串，可以这样做：

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
NSArray *immutableArray = @[mutableString];
```

没有什么可以阻止你改变它：

```
if ([immutableArray count] > 0) {
```

```
    id string = immutableArray[0];
    if ([string isKindOfClass:[NSMutableString class]]) {
        [string appendString:@" World!"];
    }
}
```

使用 NSMutableArray 增删初始化后的数组对象，它提供了很多个方法添加、删除和替换一个或多个对象：

```
NSMutableArray *mutableArray = [NSMutableArray array];

[mutableArray addObject:@"gamma"];

[mutableArray addObject:@"alpha"];

[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

这个例子创建了以 @"epsilon", @"alpha",@"beta" 结尾的数组。

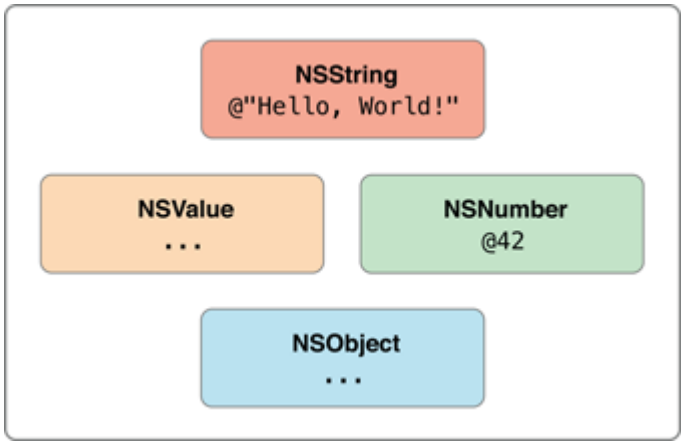
不用创建一个新的数组也可以给可变数组排序：

```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

这使得该数组以字母升序排序：@"alpha", @"beta", @"epsilon"。

无序集合：Sets

NSSet和数组很相似，但是它可以存储不同对象的无序组，如图6-2



图片 7.2 图6-2 一组对象

因为 sets 是无序的，所以当涉及到测试成员时，sets 比数组更有优势。

基础的 NSSet 类也是不可变的，所以它的值必须在创建时确定，使用空间分配并初始化或工厂方法：

```
NSSet *simpleSet =
  [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

和 NSArray, initWithObjects:,setWithObjects: 方法一样，NSSet 以 nil 作为结束，参数个数可变。可变的 NSSet 的子类是 NSMutableset。

Sets 中每一个对象只能存储一个引用,所以你不能多次添加同一个对象：

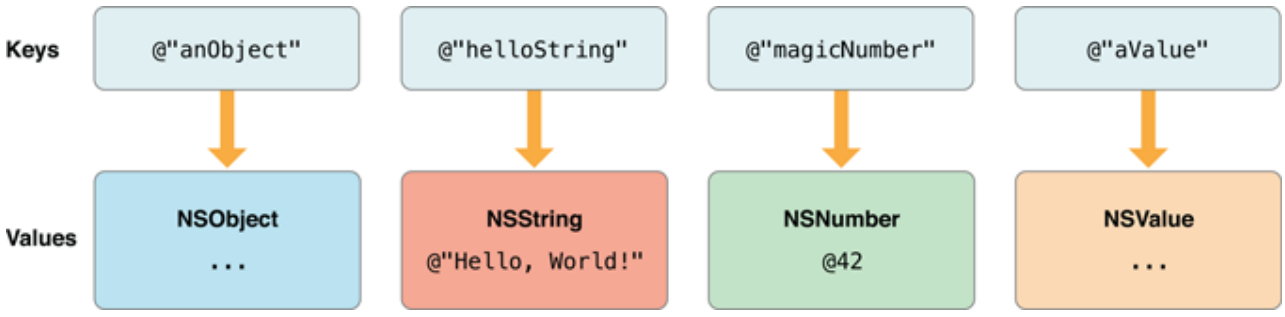
```
NSNumber *number = @42;

NSSet *numberSet =
  [NSSet setWithObjects:number, number, number, number, nil];
// numberSet only contains one object
```

查看更多关于sets： Sets: Unordered Collections of Objects (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Collections/Articles/Sets.html#//apple_ref/doc/uid/20000136)

字典是一种存储键值对的集合

NSDictionary 存储了对象和它们的关键字，而不是单单只保存了一个有序或无序的集合，这个功能可以用来恢复信息。

最好使用字符串作为字典的关键字，如图6-3：



图片 7.3 图6-3 对象词典

> 提示
>
> 使用其他对象作为关键字也可以，但值得注意的是关键字需要被字典复制，所以你的关键字一定要支持 NSCopying。

> 如果你的代码包含键值，你必须用字符串关键字作为字典对象，正如这里提到的： Key-Value Coding Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html#//apple_ref/doc/uid/10000107i)

字典的创建

你可以用空间分配并初始化或工厂方法创建字典：

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
            someObject, @"anObject",
        @"Hello, World!", @"helloString",
                @42, @"magicNumber",
             someValue, @"aValue",
                    nil];
```

注意使用 dictionaryWithObjectsAndKeys: 和 initWithObjectsAndKeys:方法时，每个对象在给定关键字前需要确定好，对象和关键字都必须以 nil 结束。

语法

Objective-C 也可以使用纯语法创建字典：

```
NSDictionary *dictionary = @{
            @"anObject" : someObject,
         @"helloString" : @"Hello, World!",
         @"magicNumber" : @42,
              @"aValue" : someValue
};
```

注意在字典语法中，关键字需要先确定，然后再确定对象，而且关键字不能以 nil 结束。

字典的查询

一旦你创建了字典，你可以从中给对象指定一个关键字：

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

如果没有找到该对象，objectForKey：方法会返回 nil。

objectForKey：方法还有另一种使用方法也可以完成这个功能：

```
NSNumber *storedNumber = dictionary[@"magicNumber"];
```

可变性

如果要在创建 dictionary 后增删对象，你需要使用 NSMutableDictionary 子类：

```
[dictionary setObject:@"another string" forKey:@"secondString"];

[dictionary removeObjectForKey:@"anObject"];
```

使用NSNull代替nil

在这一部分中，你不可以在集合类中添加 nil，因为在 Objectiv-C 中，nil 表示“没有对象”。如果你需要在集合中表示“没有对象”，你要使用 NSNull 类：

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

NSNull 是一个单例类，这意味着 null 方法总是返回同样的实例。这表示你可以用共享的 NSNull 实例来查找数组中的对象：

```
for (id object in array) {
    if (object == [NSNull null]) {
        NSLog(@"Found a null object");
    }
}
```

使用集合来保存对象的图形。

直接写 NSArray 和 NSDictionary 类是十分简单的：

```
NSURL *fileURL = ...

NSArray *array = @[@"first", @"second", @"third"];

BOOL success = [array writeToURL:fileURL atomically:YES];

if (!success)
{
    // an error occured...
}
```

如果集合中的某一个对象是 property list 类型（ NSArray，NSDictionary，NSString，NSData，NSDate 和 NSNumber ），你可以从磁盘中重新创建完整的层次：

```
NSURL *fileURL = ...
NSArray *array = [NSArra arrayWithContentsOfURL:fileURL];
if (!array) {
    // an error occurred...
}
```

查看更多关于property lists，请看Property List Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html#//apple_ref/doc/uid/10000048i)

如果你需要保存除了上面提到的标准property list类之外的对象类，你可以使用存档对象，例如 NSKeyedArchiver，来创建一个集合式的对象存档。

创建存档的唯一要求是每一个对象必须支持 NSCoding 协议。这意味着每一个对象必须知道如何在存档中编码（通过实现 encodeWithCoder: 方法），以及在读取存档时解码（通过 initWithCoder: 方法）。

NSArray，NSSet，NSDictionary 以及他们的可变子类，都支持 NSCoding，这意味着你可以通过存档来保存复杂的对象层次。如果你使用 Interface Builder 来布局窗口和视图，那么你的 nib 文件就是视觉化的层次对象存档。在程序执行时，对于使用有关类的对象层次，nib 文件是不能存档的。

查看更多关于 Archives，请看Archives and Serializations Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Archiving/Archiving.html#//apple_ref/doc/uid/10000047i)

使用最有效的集合列举技术

Objective-C、Cocoa、Cocoa Touch 提供了多种列举集合内容的方法。当然，使用 C 语言中传统的 for 循环来列举内容也是可以的：

```
int count = [array count];
for (int index = 0; index < count; index++) {
    id eachObject = [array objectAtIndex:index];
    ...
}
```

你最好可以练习使用这部分中描述的其他技术来实现这个功能。

快速列举使列举一个集合变得容易

许多集合类都符合 NSFastEnumeration 协议，包括 NSArray，NSSet 和 NSDictionary。这意味着你可以使用快速列举，一种 Objective-C 中特有的语言。

快速列举数组或集合中内容的语法：

```
for (<Type> <variable> in <collection>) {
    ...
}
```

使用快速列举描述每一个数组中对象：

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

其中的 eachObject 变量在每一层循环里自动地被设为当前对象，所以每一个对象都会列举出来。

如果你在字典中进行快速列举，你要像这样遍历字典的关键字：

```
for (NSString *eachKey in dictionary) {
    id object = dictionary[eachKey];
    NSLog(@"Object: %@ for key: %@", object, eachKey);
}
```

快速列举十分像 C 中的 for 循环，所以你可以用 break 来终止循环，用 continue 来进入下一层。

如果你要列举一个有序的集合，列举操作会按循序进行。对于 NSArray 类，列举操作的第一个对象是下标为 0 的那个元素，第二个对象是下标为 1 的那个，以此类推。如果你需要记录当前下标的值，只要简单的计算迭代次数：

```
int index = 0;
for (id eachObject in array) {
    NSLog(@"Object at index %i is: %@", index, eachObject);
    index++;
}
```

大多数集合也支持列举对象

你可以使用 NSEnumerator 对象来列举许多 Cocoa and Cocoa Touch 集合。

你也可以使用 NSArray 中的 objectEnumerator 或者 reverseObjectEnumerator 来实现快速列举：

```
for (id eachObject in [array reverseObjectEnumerator]) {
    ...
}
```

在这个例子中，循环会倒序输出集合中的对象，所以最后一个对象将会第一个被列举。

你也可以通过重复地调用 nextObject 方法来迭代内容：

```
id eachObject;
while ( (eachObject = [enumerator nextObject]) ) {
    NSLog(@"Current object is: %@", eachObject);
}
```

在这个例子中，while 用来设置下一个循环中对象的 eachObject 变量。当集合中没有剩下的对象时，nextObject 方法会返回 nil ，它被当成 false 所以循环终止。

> 提醒
>
> 由于在 C 程序中，等号（==）非常容易写错成赋值符号（=），当你在分支程序或循环中设定变量时，编译器会弹出警告：
> ```
> if (someVariable = YES)
> {
>  ...
> }
> ```
> ```
> 如果你真的想让逻辑值等于等号左边的变量，你可以将表达式用括号括起来：
> ```
> ```
> if ((someVariable = YES))
> {
>  ...
> }
> ```

```

当你使用快速列举时，你不可以改变集合。而且，使用快速列举法比人工列举对象要更快，因为你不用收集对象的名字。

**许多集合都支持基于块的列举**

你也可以使用块来列举 NSArray，NSSet 和 NSDictionary。关于块的详细信息会在下一章中介绍。
```

8

使用块 – Working with Blocks

一个 object-c 类定义了一个对象，这个对象整合了与行为相关的数据。有时，它仅仅代 表了一个简单的小任务或是一个行为单元而不是一个方法的集合。 区块是添加在 c，object-c 和 c++ 语言中的语言级别的形式，它允许你编写一个独特的代 码段，这个代码段能够在作为值方法和函数中传递。块是 object-c 的对象，这意味着他们能够被添加到像 `NASArray` 或是 `NSDictionary` 的集合中。他们也有从作用域中捕获值的 能力，使他们与其他编程语言关于 toclosures 或 lambda 表达式上的作用非常类似。 本章阐释了定义和引用块的语法，也展示了如何使用块简化一般的任务 ，比如集合枚举。欲了解更多的信息，请参阅Blocks Programming Topics。

块语法

块的定义语法是用脱字符（^）来定义的，如下：

```
^{
    NSLog(@"This is a block");
}
```

函数和方法定义时，大括号表明了块的起点和终点。在本例中，该块不返回任何值，也没有任何参数。 你也可以用同样的方式函数指针指向一个c函数，声明一个变量来跟踪块，如下：

```
void (^simpleBlock)(void);
```

如果你没有接触过c函数指针，那么这里的语法你也许会觉得有一些困扰。这个例子声明了一个变量 called `simpleBlock` 来调用一个没有参数和返回值的块，这意味着变量能够被像上面那样的块进行赋值，如下：

```
simpleBlock = ^{
    NSLog(@"This is a block");
};
```

这就像其他任何的变量赋值，所以语句必须被以右大括号后面的分号来进行结束。你也可以把变量的声明和赋值进行合并:

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is a block");
};
```

一旦你已经声明和赋值了一个块变量，你能够用它来调用这个块：

```
simpleBlock();
```

提示：如果你试图用没有被赋值的变量来调用一个块（零块变量），你的应用将会崩溃。

有参数和返回值的块

块也可以有参数和返回值就像其他的方法或是函数。 举一个例子，考虑一个变量来引用返回两个变量相乘结果的块：

```
double (^multiplyTwoValues)(double, double);
```

相应的块文字可能是这样的：

```
 ^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
 }
```

就像其他函数的定义那样,当块被调用时 `firstValue` 和 `secondValue` 是用来计算结果值的。在这个例子中，返回值类型是由块中返回语句来决定的。

根据你的个人所好，你可以通过在脱字符之后明确写出返回值类型：

```
 ^ double (double firstValue, double secondValue) {
    return firstValue * secondValue;
 }
```

一旦你已经声明或定义了一个块，你就能像一个函数一样来调用他：

```
double (^multiplyTwoValues)(double, double) =
                        ^(double firstValue, double secondValue) {
                            return firstValue * secondValue;
                        };

  double result = multiplyTwoValues(2,4);

  NSLog(@"The result is %f", result);
```

块能够从封闭区域中捕捉值

和包含可执行代码一样，块也有能够从封闭区域捕捉的能力。

比如，如果你在一个方法中声明了块，他可以捕捉到任何在方法域中可以访问到的值，如下：

```
\- (void)testMethod {
  int anInteger = 42;

  void (^testBlock)(void) = ^{
```

```
    NSLog(@"Integer is: %i", anInteger);
  };

  testBlock();
}
```

在这个例子中 anInteger 在块外进行定义，但是当块被定义时，该值也是被捕获了。

除非另行指定，值是一定会被捕获的。这意味着如果你在定义块的时间点和调用块的时间点之间改变外部变量的值，如下：

```
int anInteger = 42;

  void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
  };

  anInteger = 84;

  testBlock();
```

被块捕获的值是不受影响的。这意味着输出结果仍然会显示：

```
Integer is: 42
```

它同样意味着块不能够改变初始变量的值，或者是被捕获的值（被当做常量）。

使用__block变量共享存储

如果您需要能够从一个块中改变捕获变量的值，你可以使用 __block 存储类型修饰符对原变量声明。这意味着变量是存在于在作用域范围之内声明的块共享的存储空间。

举个例子，你可能会改写先前的例子，如下：

```
__block int anInteger = 42;

void (^testBlock)(void) = ^{
  NSLog(@"Integer is: %i", anInteger);
};

anInteger = 84;

testBlock();
```

由于 anInteger 声明为 __block 变量，其存储与块声明共享。这意味着，在日志的输出将现在显示：

```
Integer is：84
```

这也意味着，该块可以修改原始值，如下：

```
__block int anInteger = 42;

void (^testBlock)(void) = ^{
    NSLog(@"Integer is: %i", anInteger);
    anInteger = 100;
};

testBlock();
NSLog(@"Value of original variable is now: %i", anInteger);
```

这一次，输出将显示：

```
Integer is: 42
Value of original variable is now: 100
```

你可以把块作为方法或函数的参数

本章前面的例子，都是再定义一个块后，立即就调用它。实际中更常见的是在其他地方收到指令后在把块传递给方法或函数。比如，你可能用 GCD 技术来在后台调用一个块，或者定义一个块来表示一个任务被反复调用。比如，列举的集合时。并发性和枚举将本章后面介绍。

块也用于回调，定义了当一个任务结束时将被执行的代码。举一个例子，你的应用程序可能需要通过创建一个对象，执行一个复杂的任务，诸如从web服务请求信息，以响应用户操作。因为任务可能需要很长的时间，任务正在发生时，你应该显示某种进度指示器。一旦任务完成，就要来隐藏该指示器。

这将有可能使用授权做到这一点：你需要创建一个合适的委托协议，实施必要的方法，设置你的对象作为任务的代表，一旦任务完成了，那么就在你的对象中等待它调用委托方法。

使用块将使这方面内容非常容易，因为你可以在你初始化你的任务时，来定义回调行为，如下：

```
- (IBAction)fetchRemoteInformation:(id)sender {
    [self showProgressIndicator];

    XYZWebTask *task = ...

    [task beginTaskWithCallbackBlock:^{
        [self hideProgressIndicator];
    }];
}
```

这个例子调用一个方法来显示进度指示器，然后创建任务，并告诉它启动。毁掉快明确了在任务完成后将要执行的代码；在这种情况下，调用一个方法来隐藏进度指示器是十分简单的。请注意，此回调块捕获自己以便能够在被调用时能够调用 `hideProgressIndicator` 方法。捕获自己时应当非常小心，因为这非常容易形成一个强引用循环，这将在后面的 “在捕捉自时避免强引用循环” 中详细描述。

在代码的可读性方面，该块可以在任务完成前和任务完成后很容易地看到在一个地方会发生什么。避免了通过跟踪和委托方式来查明将要放生什么的需要。

声明 `beginTaskWithCallbackBlock`：在这个例子中所示的方法是这样的：

```
- (void)beginTaskWithCallbackBlock:(void (^)
(void))callbackBlock;
```

在(void (^)(void)) 指定的参数是块不带任何参数或者返回任何值。该方法的实现可以用普通的方法来调用块：

```
- (void)beginTaskWithCallbackBlock:(void (^)(void))callbackBlock {
    ...
    callbackBlock();
}
```

用和块变量一样的方法来确定以一个或多个参数的块为参数的方法：

```
- (void)doSomethingWithBlock:(void (^)(double, double))block {
    ...
    block(21.0, 2.0);
}
```

块应该总是一个方法的最后一个参数

一个方法仅使用一个块参数为好。如果方法还需要其他的非块参数，块应该放在参数的最后一个。

```
- (void)beginTaskWithName:(NSString *)name completion:(void(^)(void))callback;
```

这使得指定块内嵌时的方法调用更容易阅读，如下：

```
[self beginTaskWithName:@"MyTask" completion:^{
        NSLog(@"The task is complete");
    }];
```

使用类型定义简化块语法

如果您需要定义具有相同签名多个块，你可能想给这个签名定义自己的类型。 举一个例子，你能够定义一个类型来创建一个没有参数和返回值的块，如下：

```
typedef void (^XYZSimpleBlock)(void);
```

然后，你可以用你自定义的类型作为方法参数，或作为创建的块变量，如下：

```
XYZSimpleBlock anotherBlock = ^{
    ...
};
```

```
- (void)beginFetchWithCallbackBlock:(XYZSimpleBlock)callbackBlock {

    ...
    callbackBlock();
}
```

当处理以块为参数或返回值类型的块是定义自定义类型尤其重要，考虑如下的例子：

```
void (^(^complexBlock)(void (^)(void)))(void) = ^ (void (^aBlock)(void)) {
    ...
    return ^{
        ...
    };
};
```

该 complexBlock 变量是指一个块需要另一块作为一个参数（ABLOCK）并返回另一个块。

重写代码使用类型定义使其可读性更高：

```
XYZSimpleBlock (^betterBlock)(XYZSimpleBlock) = ^ (XYZSimpleBlock aBlock) {
    ...
    return ^{
        ...
    };
};
```

对象用属性来跟踪块

定义一个属性来跟踪块的语法和定义块变量的语法是相似的：

```
@interface XYZObject : NSObject
@property (copy) void (^blockProperty)(void);
@end
```

注意：你应该指定副本的属性特征，因为一个块需要被复制来跟踪其在原始范围外抓获的状态。 使用自动引用计数的时候，你就不必这样做了，因为它会自动发生，但属性特征最好是来显示 其必然行为。想要得到更多的相关信息，请参考块编程条目。

一个块属性被初始化或调用的方法和其他块变量是相似的：

```
self.blockProperty = ^{
    ...
};
self.blockProperty();
```

它也可以使用块属性声明的类型定义，如下：

```
typedef void (^XYZSimpleBlock)(void);

@interface XYZObject : NSObject
@property (copy) XYZSimpleBlock blockProperty;
@end
```

在捕捉自时避免强引用循环

如果你需要在块中捕捉自，比如说定义一个回调块，考虑内存管理的影响十分重要。

块对任何捕获的对象都保持了强引用，包括自己，这意味着它很容易结束了一个很强的参考周期 举个例子，一个对象维护一个捕获自的块的副本属性：

```
@interface XYZBlockKeeper : NSObject
@property (copy) void (^block)(void);
@end
```

```
@implementation XYZBlockKeeper
- (void)configureBlock {
    self.block = ^{
        [self doSomething];    // capturing a strong reference to self
                               // creates a strong reference cycle
    };
}
...
@end
```

编译器会警告你的简单错误可能是这样的，但更复杂的例子可能涉及的对象创建周期之间的多个强引用，使之更难以诊断。为了避免这个问题，捕获一个弱参考是比较好的方法，如下：

```
- (void)configureBlock {
    XYZBlockKeeper * __weak weakSelf = self;
    self.block = ^{
        [weakSelf doSomething];   // capture the weak reference
                                  // to avoid the reference cycle
```

```
    }
}
```

通过捕获自我弱指针，块将无法保持牢固的关系去回到XYZBlockKeeper对象。如果该对象在 快被调用前释放，weakSelf指针会简单地设置为零。

块可以简化枚举

除了一般完成处理程序，许多 cocoa 和 cocoa Touch API 用块来简化一般的任务，如集合枚举 像是 `NSArray` 类，提供了三个基于块的方法，包括：

```
- (void)enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL *stop))block;
```

这个方法有一个块参数，对每个项目来说只掉用一次。

```
NSArray *array = ...
    [array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
        NSLog(@"Object at index %lu is %@", idx, obj);
    }];
```

块本身有三个参数，前两个参数表明了当前的对象和他在阵列中的索引。第三个参数为一个指向波尔变量的指针，

你可以用它来停止枚举：

```
[array enumerateObjectsUsingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
        if (...) {
            *stop = YES;
        }
    }];
```

它也可以通过使用 `enumerateObjectsWithOptions` 定制枚举。指定 `theNSEnumerationReverse` 选项。例如，以相反的顺序遍历集合。

如果枚举代码块是密集处理的，并且能安全并发执行，你可以使用 `NSEnumerationConcurrent` 选项：

```
    [array enumerateObjectsWithOptions:NSEnumerationConcurrent
                            usingBlock:^ (id obj, NSUInteger idx, BOOL *stop) {
        ...
    }];
```

这个标志指示枚举块调用可以分布在多个线程，如果代码块是密集处理型，那么该调用将提供一个潜在的性能提升。请注意，使用此选项时，枚举顺序是不确定的。

```
NSDictionary 类同样提供基于块的方法，包括：
NSDictionary *dictionary = ...
  [dictionary enumerateKeysAndObjectsUsingBlock:^ (id key, id obj, BOOL *stop) {
     NSLog(@"key: %@, value: %@", key, obj);
  }];
```

举个例子来说，相对于传统的循环，这样做能够更方便的枚举所有键值对。

模块可以简化并发任务

块代表了一个不同的工作单元，综合了可执行代码和周围范围可捕获的状态。这使得它非常适合使用OS X和iOS提供的并发选项中的异步调用。你可以使用块简单地定义你的任务，然后让系统执行这些任务的处理器资源可用，而不必弄清楚像线程低层次的机制怎样使用。

OS X和ios提供了多种并发技术，其中包括两个任务调度机制：操作队列和中央调度。这些机制是围绕着等待被调用的任务队列的一个想法。你按照调用顺序将你的块加入队列，然后在处理器资源可用时，你的系统将从队列中取出调用。

一个串行队列只允许同时执行一个任务--下一个任务将在上一个任务完成后才会在队列中取出调用。一个并发队列调用可以包括非常多的任务，并且不用等待前一个人物执行完毕。

使用块操作与运行队列

一个运行队列是 Cocoa 和 Cocoa Touch 提供的任务调度。要创建的 `NSOperation` 实例来封装工作单元以填充必要数据然后将该操作加入 `NSOperationQueen` 来执行。

尽管你可以创建有自己的自定义NSOperation子类来实现复杂任务，但也可以通过使用 `NSBlockOperation` 和块来创建操作，如下：

```
NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^{
    ...
}];
```

这虽然是可以手动执行的操作，但是操作也会被添加在正准备执行的已经存在的运行队列或是你自己创建的运行队列：

```
// schedule task on main queue:
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
[mainQueue addOperation:operation];

// schedule task on background queue:
```

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue addOperation:operation];
```

如果您使用的操作队列，可以配置操作的优先级或依赖性，比如限制一个操作在其他几项操作运行完成后才能执行。你同样可以通过键值观察来监视操作的状态改变，这样，当一个任务完成时，可以更加容易的更新进度指示器。 想要得到更多的关于操作运行队列的信息，请参考Operation Queues。

用GCD技术在调度队列调度块

如果你需要调用任意代码块来执行，你可以直接利用GCD技术控制的调度队列。对于请求者，调度队列可以很容易地同步或异步执行任务，并以先入先出的顺序执行他们的任务。

你可以创建你自己的调度队列，或是利用GCD提供的自动生成的队列。比如，如果你需要调度一个并发执行的任务，你能从已经存在一个队列中得到参考，通过使用 dispatch_get_global_queue()函数来调节队列优先级，如下：

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

你可以用 `dispatch_async()` 方法或是 `dispatch_sync()` 方法，调度块到队列中。 `dispatch_async()` 方法不用等到块被调用会直接立即返回。

```
dispatch_async(queue, ^{
    NSLog(@"Block for asynchronous execution");
});
```

`dispatch_sync()` 方法不会返回指导块完成执行；比如，在一个需要等待其他工作完成才能继续的并发块中你可以运用该方法。

想要得到更多关于 GCD 和调度队列的信息，请参考 `Dispatch Queues` 。

9

错误处理 – Dealing with Errors

几乎所有的 APP 都会出现错误。一些错误可能会在你的可控范围之外，例如硬盘空间耗尽或者网络连接中断。另一些错误却是可恢复的，例如无效用户输入。当开发者在不断追逐完美的过程中，也可能会伴随着偶尔的编程错误的出现。 如果你来自于其他的语言和开发平台，你也许会习惯于处理大多数错误处理中的异常。当你用 ObjC 编程的时候，异常仅会出现在编程错误中，就像数组越界访问或无效方法参数。这些就是在你的 APP 上线前的测试中，你需要排查并修复的问题。 NSError 类的实例代表了所有其他的错误。这一章我们将简单的介绍一下 NSError 对象的使用，包括怎样处理构造方法可能出现的失败和返回错误。更多信息参见 Error Handling Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ErrorHandlingCocoa/ErrorHandling/ErrorHandling.html#//apple_ref/doc/uid/TP40001806)

对大多数的错误使用 NSError

错误是任何APP 生命周期中不可避免的一部分，假设你需要向一个远程网络服务器请求数据，在这个过程中有多种潜在问题可能出现，包括：

- 无网络连接
- 远程网络服务器不可访问
- 远程网络服务器不能提供你请求的信息
- 得到的数据与你期望的不匹配
- 遗憾的是，建立所有可能问题的应急计划与方案是不现实的。相反你必须为可能出现的错误筹划并且知道如何解决，从而获得最好的用户体验。

一些授权方法（ delegate method ）向你提醒错误

如果你正实现一个授权对象，它是与一个执行某任务的构造类（ framework class ）一起使用的，比如这个对象需要从远程服务器上下载信息。通常，你会发现你需要至少实现一个错误关联方法（error-related method ）。例如 包括了一个 connection:didFailWithError: 方法的NSURLConnectionDelegate 协议：

```
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error;
```

当一个错误发生时，授权方法将会被调用，以向你提供一个 NSError 对象来描述这个错误。 一个NSError对象包含一个数字错误代码，域名和描述，以及封装在一个用户信息字典里的其他相关信息。 比起做出让所有可能错误都具有唯一的数字代码的要求，Cocoa 和 Cocoa Touch 的错误被划分成域。例如一个错误发生在 NSURLConnection 中，NSURLErrorDomain 域中的 connection:didFailWithError: 方法将会报一个错。错误对象还包含一个本地化的描述，例如“找不到指定主机名的服务器”。

一些方法可以通过引用传递错误

一些Cocoa 和 Cocoa Touch 的API 通过引用传回错误，举个例子，你可能决定通过 NSData 的 writeToURL:options:error: 方法，将你从网络服务器获得的信息通过存入硬盘的形式保存下来。这个方法的最后一个参数是一个指向 NSError 指针的引用。

```
- (BOOL)writeToURL:(NSURL *)aURL
options:(NSDataWritingOptions)mask
error:(NSError **)errorPtr;
```

在你调用这个方法之前，你需要创建一个合适的指针来传递地址：

```
NSError *anyError;
BOOL success = [receivedData writeToURL:someLocalFileURL
                        options:0
                        error:&anyError];
if (!success) {
NSLog(@"Write failed with error: %@", anyError);
// present error to user
}
```

当错误发生时，writeToURL:options:error: 方法会返回 NO ,并会更新你的 anyError指针，指向一个错误类来描述出现的问题。在处理应用传递的错误时，重要的是去检测方法的返回值以确定是否有错误发生。不要只是测试是否有设置指向错误的指针。

> 提示： 如果你并不关心出错的对象，则只需要将 NULL 传递给 error: 参数

如果可以，尽量将错误恢复或显示给用户

对你的 APP 来说最好的用户体验是隐蔽地从错误中恢复。例如，你正在向远程网络服务器发出请求，如果此时发生了错误，你可以试着再向另一个服务器发送请求，或者你可以向用户请求更多的信息，比如有效的用户名和密码，然后再次发出请求。 当从错误中恢复是不可能的时候，你才应该去警告用户。如果你正在使用 iOS的Cocoa Touch进行开发，你需要创建一个 UIAlertView 并调节它，从而向用户显示错误。如果你正在使用 OS X 的 Cocoa ，你可以在任何一个 NSResponder 对象（像是一个界面、窗口甚至是应用程序对象本身）上调用 presentError: ，然后这个错误便会被传输到响应链上，以进行进一步的调试或恢复。当它到达应用程序对象时，应用程序会通过一个警告板将错误呈现给用户。更多关于向用户呈现错误，参看 Displaying Information From Error Objects (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ErrorHandlingCocoa/CreateCustomizeNSError/CreateCustomizeNSError.html#//apple_ref/doc/uid/TP40001806-CH204-BAJCIGIA)

生成你自己的错误信息

为创建你自己的 NSError 类，你需要通过以下格式定义你自己的错误类：

```
com.companyName.appOrFrameworkName.ErrorDomain
```

你需要为每一个可能发生在你的错误域中的错误选择一个唯一的错误编码，同时还有合适的错误描述，关于错误的描述将会被存储在用户信息字典中。像下面这样：

```
NSString *domain = @"com.MyCompany.MyApplication.ErrorDomain";
NSString *desc = NSLocalizedString(@"Unable to…", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey : desc };

NSError *error = [NSError errorWithDomain:domain
                                     code:-101
                                 userInfo:userInfo];
```

这个例子使用了 NSLocalizedDescriptionKey 函数来查看来自 Localizable.strings 文件中，错误描述的本地化版本，即在本地化字符串资源中描述的那样。

如果你需要像早前描述的那样，通过引用将错误传回，你的方法标识中还需要包含一个为指针设置的参数，用来指向一个 NSError 对象。你同时还需要利用返回的值指明是成功还是失败，像是这样：

```
- (BOOL)doSomethingThatMayGenerateAnError:(NSError **)errorPtr;
```

当错误发生时，在返回表示失败的NO之前，你需要首先检查为错误参数提供的指针值是否为空（ NULL ），然后才能取消引用来设置错误信息：

```
- (BOOL)doSomethingThatMayGenerateAnError:(NSError **)errorPtr {
...
// error occurred
if (errorPtr) {
    *errorPtr = [NSError errorWithDomain:...
                                    code:...
                                userInfo:...];
  }
  return NO;
}
```

用于编程错误的异常

ObjC 用与其他编程语言类似的方式支持异常，并且与这些语言，如 C++ 、Java ，支持的语法也很相似。就像 NSError 一样，在 Cocoa 和 Cocoa Touch 里的异常，也是以 NSException 类实例为代表的对象。 如果你编写的代码可能导致异常抛出，你可以将这段代码封装在一个 try–catch 块内：

```
 @try {
    // do something that might throw an exception
}
@catch (NSException *exception) {
    // deal with the exception
}
@finally {
    // optional block of clean-up code
    // executed whether or not an exception occurred
}
```

如果异常抛出发生在 @try 块内，那么它将会被 @catch 块捕捉，以方便你对它的处理。比如你在用使用异常作为错误处理的低级别C++ 库进行开发，你可能会捕捉到异常，并生成一个合适的 NSError 对象以向用户显示异常。

如果一个异常被抛出但并未被捕获，那么默认的未捕捉异常处理器（ uncaught exception handler ）将会加载一个可以控制并终止应用的消息。

你不应该使用 try–block 块来代替 ObjC 的标准程序检测，以 NSArray 为例，你应该总是先使用数组的数目（ count ）来确定元素的数量，然后才通过给定索引访问一个对象。如果你发出了越界访问请求，那么 objectAtIndex: 方法将会抛出一个异常，这样你就可以在早期的开发过程中及时发现 bug ——你应该尽量避免在你已经上线的 APP 中出现异常抛出。 更多关于 ObjC 应用中的异常，参看 Exception Programming Topics (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Exceptions/Exceptions.html#//apple_ref/doc/uid/10000012i)

10

命名规则 – Conventions

使用框架类时，你会发现 Objective-C 代码非常易于阅读。它的类名和方法名比一般的 C 语言函数或 C 标准库要更加具有描述性。Camel Case 适用于带有多个单词的名称。在编写类的时候，你应当使用与 Cocoa 和 Cocoa Touch 相同的规则，这样可以使你的代码对于你?和其他需要与你协同工作的 Objective-C 开发者来说更加易读，并且可以保证代码库的一致性。

此外，请严格遵循命名规则，保证 Objective-C 框架功能的各种机制的正常工作。例如，访问方法的名称必须遵循命名规则来命名才能在 Key-Value Coding（KVC）或 Key-Value Observing（KVO）机制下正常工作。

本章介绍了一些广泛运用于 Cocoa 及 Cocoa Touch 编码中的命名规则，也列举了需要在整个应用程序项目（包括使用到的框架类）中唯一命名的情况。

一些名称必须在程序中唯一

创建一个新的类别、符号或是标识符时，你必须先考虑它的名称在所属区域中是否唯一。这个区域可能是整个应用程序（包括所涉及到的框架类）；也有可能仅仅是一个封装的类或是一个代码块。

类名应在程序中唯一

不论是在项目中还是在项目所包含的框架或 bundle 中，Objective-C 中的类名必须是唯一的。例如，避免使用如 ViewController 或 TextParser 作为类名，因为这些名称有可能是包含在 App 中的一个框架名。

在命名类时使用前缀命名可以保证类名唯一。你会注意到 Cocoa 及 Cocoa Touch 的类名通常都以 NS 或 UI开头。像这样两个字符组成前缀的命名方式在 Apple 中专门为框架类而保留。随着你学习更多关于 Cocoa 及 Cocoa Touch 的知识，你会遇到一些其他的特定框架的前缀，如下表所示。

Prefix	Framework
NS	Foundation (OS X and iOS) and Application Kit (OS X)
UI	UIKit (iOS)
AB	Address Book
CA	Core Animation
CI	Core Image

而你所创建的类的命名必须使用三个字符组成的前缀，可以是你的公司名和 App 名的结合，也可以是你的 App 中的某个组成部分。例如，如果你的公司叫做 Whispering Oak，你正在开发的一个游戏叫做 Zebra Surprise，那么你可以使用 WZS 或 WOZ 作为类名的前缀。

此外，你应该尽可能使用名词来命名你的类名，这样可以明确地显示这个类所代表的含义。例如下表中来自 Cocoa 及 Cocoa Touch中的例子。

NSWindow	CAAnimation:	NSWindowController	NSManagedObjectContext

如果你需要使用多个单词来命名你的类，那么每个单词的首字母必须是大写。

类中的方法名应具描述性且唯一

在保证了类名唯一后，同时这个类中所定义的方法名也必须是唯一的，但它可以与其他类中的某一个方法名相同。例如复写一个父类方法或是使用多态方法的情况。在不同类中具有相同功能的方法应当有相同的方法名、返回类型和参数类型。

方法名不需要包含前缀，且应当以小写字母开头，而对于多单词组成的名称，后续单词的首字母应该大写。下表为 NSString 类中的方法的命名。

length	characterAtIndex:	lengthOfBytesUsingEncoding:

如果一个方法中带有多个参数，那么方法名中必须提示每一个参数：

substringFromIndex:	writeToURL:atomically:encoding:error:	enumerateSubstringsInRange:options:usingBlock:

方法名的第一部分提示方法的主要作用或提示方法调用后的结果。例如，如果一个方法在调用后返回一个值，那么第一个单词应当表明该返回值的类型，如上文表格中的方法名 length，character...，substring...等等；如果你还需要在方法名中提示一些关于这个返回值的信息，那么应当使用多个单词来命名方法，如 NSString 类中的 mutableCopy 方法，capitalizedString 方法和 lastPathComponent 方法；如果一个方法用于执行某个操作，如写磁盘或列举目录，那么方法名的第一个单词必须提示这个操作，如 write... 方法或 enumerate... 方法。

如果方法中含有一个 error 指针参数来指示错误，那么应在其名称的最后加上 error 参数；如果方法中使用了 block，同样，为了使得能够应在方法名的最后加上 blcok 参数。

使用一个能够清楚表明方法的作用但又简洁明了的方法名非常重要，这不代表你需要取一个冗长的方法名，而同时方法名过于简洁明了也无法很好地表明一个方法的含义，因此最佳的方法是在这二者中找寻平衡点，如：

stringAfterFindingAndReplacingAllOccurrencesOfThisString:withThisString	Too verbose
strReplacingStr:str:	Too concise
stringByReplacingOccurrencesOfString:withString:	Just right

除非在不同语言和文化中某个单词的缩写是众所周知的，否则你应当避免在方法名中使用缩写。这里提供了一个常见缩写的列表：Acceptable Abbreviations and Acronyms (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/APIAbbreviations.html#//apple_ref/doc/uid/20001285) 。

在框架类的 Categories 中使用前缀命名方法

当使用 Category 来创建方法到一个已经存在的框架类时，你需要在方法名中使用前缀来避免冲突。

同一区域内局部变量名必须唯一

由于 Objective-C 是 C 语言扩展而来，因此 Objective-C 同样要遵守 C 语言中的变量命名规则。一个局部变量的变量名不允许与在相同范围内声明的其他变量名冲突。

```
(void)someMethod {
    int interestingNumber = 42;
    ...
    int interestingNumber = 44; // not allowed
}
```

尽管 C 语言允许一个局部变量的变量名与封装区域中的变量名相同。如下：

```
(void)someMethod {
    int interestingNumber = 42;
    ...
    for (NSNumber *eachNumber in array) {
        int interestingNumber = [eachNumber intValue]; // not advisable
        ...
    }
}
```

但这样会使得代码段变得混乱，不易于阅读，所以在编码中请尽量避免这种情况。

方法名

与类的命名相同，除了考虑唯一性之外，方法的命名还应该遵循严格的命名规则。除了 Cocoa 和 Cocoa Touch 的要求外，这些规则是为了满足一些 Objective-C 的基础机制，如编译和运行。

访问方法的名称

使用 @property 语法来声明对象的属性时（如 Encapsulating Data (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple_ref/doc/uid/TP40011210-CH5-SW1) ），编译器会自动合成 getter 和 setter 方法。如

果你需要使用自己提供的访问方法，那么你需要保证一个每一个属性都使用了正确的方法名，这样方法才可以被语法点调用。

除个别特殊情况之外，一个 getter 方法必须使用与属性相同的名称。例如，一个属性名为 firstname，那么它的访问方法名称也应为 firstname。但 Boolean 属性是一个特例，因为 Boolean 属性的 getter 方法名以 is 开头。例如，一个属性名为 paused，那么它的 getter 方法名为 isPaused。

一个属性的 setter 方法命名应使用形如 setPropertyName: 的形式。譬如，一个属性名为 firstname，那么它的 setter 方法应为 setFirstName:；同样对于一个 Boolean 型的属性 paused 而言，相应的 setter 方法应为 setPaused:。

尽管使用 @property 语法能够方便地设定不同的访问方法名称（你只需要特殊考虑 Boolean 型的情况），但你还是需要注意下述的一些规则，以免如 Key Value Coding（getter 名为 valueForKey:，setter名为 setValue:forKey:）这样的机制无法正常工作。了解更多有关 KVC 的信息，查阅 Key-Value Coding Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html#//apple_ref/doc/uid/10000107i) 。

创建对象的方法名

在之前的一些章节中介绍了许多不同创建一个类的实例的方法，你可以同时分配空间和初始化，例如：

```
NSMutableArray *array = [[NSMutableArray alloc] init];
```

或者你可以使用 new 方法来分配空间和初始化：

```
NSMutableArray *array = [NSMutableArray new];
```

一些类还提供了 class factory 方法：

```
NSMutableArray *array = [NSMutableArray array];
```

创建 class facotry 方法应以其所创建的类的名称开头（不使用前缀），以 NSArray 类为例，它的 factory 方法以 array 开头。而 NSMutableArray 类不定义任何它的类依赖 factory 方法，因此它创建一个可变 array 的 factory 方法仍然以 array 开头。

Objective-C 有许多不同的内存管理规则，这些规则让编译器能够保证对象有足够的的内存空间。编译器会根据创建方法的名称来判断遵循哪一条规则。因为 autorelease pool blocks 不同，由 factory 方法创建的对象与由传统的分配空间、初始化或 new 方法得来的对象在管理上会有细微的区别。学习更多有关 autorelease pool blocks 和一般的内存管理知识，查阅 Advanced Memory Management Programming Guide (https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i) 。