第1章 前 言

C程序设计语言是由AT&T贝尔实验室的Dennis Ritchie于20世纪70年代早期首创的。但是，直到70年代晚期，这种程序设计语言才获得了广泛的支持并流行开来。这是因为，在此之前，C编译器还不能用于贝尔实验室以外的商业用途。最初，UNIX操作系统同样的普及速度也在某种程度上促进了C语言的快速普及，UNIX操作系统几乎完全是由C语言编写的。

Objective-C语言是由Brad J.Cox于20世纪80年代早期设计的，它以一种叫做SmallTalk-80的语言为基础。Objective-C建立在C语言之上，意味着它是在C语言基础添加了扩展而创造出来的能够创建和操作对象的一门新的程序设计语言。

1988年，NeXT Software公司获得了Objective-C语言的授权，并开发出了Objective-C的语言库和一个名为NEXTSTEP的开发环境。1992年，自由软件基金会的GNU开发环境增加了对Objective-C的支持。所有Free Software Foundation（FSF）产品的版权归FSF所有。它根据GNU通用公共许可证发布产品。。

1994年，NeXT Computer公司和Sun公司联合发布了一个针对NEXTSTEP系统的标准规范，名为OPENSTEP。OPENSTEP在自由软件基金会的实现名称为GNUStep。有一个Linux版本，它包括Linux内核和GNUStep开发环境，这个Linux发行版被十分贴切地命名为LinuxSTEP。

1996年12月20日苹果公司宣布收购NeXT Software公司，NEXTSTEP/OPENSTEP环境将成为苹果操作系统下一个主要发行版本OS X的基础。这个开发环境的该版本被苹果公司称为Cocoa。它内置了对Objective-C语言的支持，并结合了Project Builder（或它的后继版本Xcode）和Interface Builder等开发工具，苹果公司为Mac OS X上的应用程序开发创建了一个功能强大的开发环境。

2007年，苹果公司发布了Objective-C语言的更新，并称之为Objective-C 2.0。本书即是基于该版本的。

iPhone于2007年发布时，开发人员们要求为这款革新性的设备开发应用程序。起初，苹果公司不欢迎第三方应用程序开发。苹果公司安抚那些超级崇拜iPhone的开发人员的办法是：允许他们开发基于Web的应用。这些基于Web的应用在iPhone内置的Safari Web浏览器下运行，但需要用户连接到托管该应用程序的网站。开发人员对基于Web的应用的很多固有限制非常不满，于是苹果公司不久之后就宣布，开发人员能够为iPhone开发所谓的本机应用。

本机应用是驻留在iPhone上并且在iPhone操作系统下运行的应用，其运行方式与该设备上运行的内置iPhone应用（如Contacts、iPod和Weather）相同。iPhone的操作系统实际上是某个Mac OS X版本，这意味着可以在MacBook Pro上开发并调试这些应用。实际上，苹果公司很快就提供了强大的软件开发套件（SDK），允许快速地开发iPhone应用并进行调试。iPhone模拟器使得开发人员直接在其开发系统上调试应用成为可能，无需在实际的iPhone或iPod Touch设备上下载并测试程序。

1.1 本书的内容

在计划编写这本有关Objective-C的教程时，我做出了一项重要决定。即在讲述Objective-C的内容之前，假定读者已经知道如何编写C语言程序。我将从使用丰富的例程库（例如Foundation框架和Application Kit框架）的角度讲解这门语言。我还将讲解如何使用一些开发工具（如Mac的Xcode和Interface Builder等）。

选用这种方法有以下几个问题：首先，学习Objective-C之前必须完整地学习C语言，这种说法是错误的。C语言是一门过程性的语言，有很多特性是在使用Objective-C进行程序设计时不必要的，特别是对于初学者。事实上，采用其中的某些特性违反了坚持良好的面向对象的程序设计方法的本质。同样，在学习面向对象编程语言之前，最好不要了解过程性语言的所有细节。这会导致程序员误入歧途，并在养成良好的面向对象的程序设计风格方面，造成错误的导向和思维定式。Objective-C仅仅只是C语言的扩展，并不意味着必须首先学习C语言！

因此，我决定既不首先讲述C，也不事先假定你具备了该语言的知识。相反，我决定采用一种非常规的方式，从面向对象编程的视角出发，将Objective-C和基础的C语言作为一门单独的集成语言来讲解。顾名思义，本书的目的是教你如何使用Objective-C 2.0进行程序设计。这并不表示我会详细介绍可用于开发和调试程序的开发工具，或者讲解如何使用Cocoa开发交互式图形应用。学会如何使用Objective-C编写程序后，所有这些资料都可在其他地方获得。事实上，在具备了如何使用Objective-C进行程序设计的坚实基础后，掌握这些知识是轻而易举的。本书并不假设需要编程经验，即使有，也不会很多。如果你是一名程序设计的初学者，应该可以将Objective-C作为第一门程序设计语言。

本书以示例的方式来讲述Objective-C语言。在介绍这门语言的每个新特性时，通常会提供一个完整的小例子来阐述这一特性。正如一图胜千言一样，一个经过严格筛选的例子也有如此功效。强烈建议运行每个程序（所有这些程序都可在线获得），并比较系统上获得的结果与本书中的结果。这么做，不仅可以学会Objective-C语言及其语法，而且还能熟悉编译和运行Objective-C程序的过程。

1.2 本书的组织方式

本书分为三个逻辑部分：第一部分是“Objective-C 2.0语言”，介绍了该语言的基础知识；第二部分是“Foundation框架”，讲述了如何使用构成Foundation框架的种类丰富的预定义类。第三部分是“Cocoa程序设计和iPhone SDK”，简要介绍了Cocoa的应用程序套件框架，然后逐步演示了如何使用UIKit框架开发简单的iPhone应用，以及如何使用Xcode和Interface Builder开发并调试代码。

框架就是一组从逻辑上组合在一起的类和例程，它们使开发程序更加容易。使用Objective-C进行程序设计时需要的许多能力都来源于大量可用的框架。

第2章首先讲述了如何使用Objective-C编写第一个程序。

因为本书并非主要讲解Cocoa程序设计的，所以在第三部分之前没有过多介绍图形用户界面，甚至很少提及它。这就需要使用一种方法实现程序输入并产生输出。本书中的大多数例子

都是从键盘获得输入并在一个窗口中产生输出：如果在命令行使用gcc，那么这个窗口是Terminal窗口；如果使用Xcode，那么这个窗口是控制台窗口。

第3章介绍了面向对象程序设计的基础。本章引入了一些术语，但数量保持了最少。本章还介绍了定义类的机制，以及向实例或对象发送消息的方式。教师或者有经验的Objective-C程序员将会注意到：本书使用静态类型声明对象。我认为这种方法是学生起步的最好方式，因为编译器能捕捉更多的错误，程序有更强的自文档化（self-documenting）功能，同时还能鼓励新程序员显式声明已知的数据类型。这样，id类型的概念及其强大功能直到第9章才会完全显现出来。

第4章描述了基本的Objective-C数据类型以及如何在程序中使用它们。

第5章介绍了可用在程序中的3种循环语句：for、while和do。

决定选择是任何计算机程序设计语言的基础。第6章详细讲述了Objective-C语言的if和switch语句。

第7章更深入地研究了类和对象的使用，详细讨论了方法、方法的多个参数以及本地变量的相关内容。

第8章介绍了继承的主要概念。这一特性使得程序的开发更容易，因为我们可以利用以前编写的代码。使用继承以及子类的概念可以方便地修改和扩展现有的类定义。

第9章讨论了Objective-C语言的3个重要特性。多态、动态类型以及动态绑定是本章的关键概念。

第10章至第13章对Objective-C深入讨论，既包含对象的初始化、协议、分类、预处理程序，还包括一些基本的C语言特性，如函数、数组、结构和指针。第一次开发面向对象的程序时，通常不必（最好避免）使用这些特性。建议你首次通读本书时略过第13章，只在需要了解这门语言的特殊特性的更多内容时再返回来学习它。

第二部分从第14章开始，这部分介绍了Foundation框架以及如何访问它的文档。

第15章至第19章讲解了Foundation框架的重要特性。这包括数字和字符串对象、集合、文件系统、内存管理以及对象的复制和归档。

学习完第二部分后，你将能够使用Foundation框架开发出相当复杂的Objective-C程序。

第三部分从第20章开始，本章简要介绍了应用程序套件，它提供了在Mac上开发复杂图形应用所需的各种类。

第21章介绍了iPhone SDK和Uikit框架。本章阐述了如何以迭代的方式编写简单的iPhone（或iTouch）应用，然后列举了一个计算器应用的示例，通过它可使用iPhone进行简单的分数算术运算。

因为面向对象的用语涉及大量术语，所以附录A提供了一些常用术语的定义。

附录B对Objective-C语言进行了总结，它用于快速参考。

附录C给出了本书第二部分开发并大量使用的两个类的源代码。这些类定义了地址卡和地址簿类。使用方法可以执行简单的操作，如在地址簿中添加和删除地址卡、查找某人、列出地址簿的内容等。

学会如何编写Objective-C程序后，可以继续向几个不同的方向发展。你可能想学习有关C

语言的更多内容，或开始编写在Mac OS X上运行的Cocoa程序，或者是开发更复杂的iPhone应用。不管是哪种情况，附录D都能引导你向正确的方向迈进。

1.3 致谢

我要感谢在本书第一版的准备阶段为我提供帮助的朋友们。首先，要感谢Tony Iannino和Steven Levy对原稿的审阅，并感谢Mike Gaines对本书的贡献。

其次，还要感谢本书的技术编辑Jack Purdum（第一版）和Mike Trent。我很幸运，因为Mike审阅了本书的两个版本。Mike对我所编写的每一本书都进行了最详细的审阅，他不仅指出了书中的不足之处，而且还十分慷慨地提出了建议。正是因为Mike对第一版提供的意见，我改变了讲解内存管理的方法，并尽力确保本书中的每个示例都是“无漏洞的”。Mike还为有关iPhone程序设计的章节提供了宝贵的贡献。

在第一版中，Catherine Babin提供了封面的图片，他还提供了许多有价值的图片供我选择。朋友为我制作的封面艺术效果，使得本书具有了更强的专业性。

我非常感谢Pearson的Mark Taber，他忍受了我的推迟交稿并且非常和蔼地让我按照自己的进度工作，并且容忍我在撰写第二版时又不断地将最终完成时间推后。对于Pearson的朋友，我还要感谢我的开发编辑Michael Thurston、文字编辑Krista Hansing以及我的项目编辑Mandie Frank，他出色地管理了很多乱糟糟的事情并最终漂亮地完成了任务。

和往常一样，当我整个夏天（这项工作一直到秋天才完成）都忙于此书时，我的孩子们表现出了令人难以置信的成熟和耐心！感谢Gregory、Linda和Julia，我爱你们！

Stephen G. Kochan

2008年10月

第一部分　Objective-C语言

第2章　Objective-C程序设计

本章我们将直接切入正题，并演示如何编写第一个Objective-C程序。你现在还不需要使用对象，对象是下一章讨论的主题。希望你能通过本章理解编写程序和编译并运行它的各个步骤。我们重点讲解了在Windows和Macintosh计算机上的这一过程。

首先，举一个十分简单的例子，在屏幕上显示短语“Programming is fun！”的程序。无需大费周折，代码清单2-1显示了用于完成此任务的Objective-C程序。

代码清单2-1

```
// First program example

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog (@"Programming is fun!");

   [pool drain];
```

```
    return 0;
}
```

2.1 编译并运行程序

在详细解释这个程序之前，先要学习编译和运行此程序的步骤。可以使用Xcode编译并运行程序，或者使用GNU Objective-C编译器在Terminal窗口中编译并运行程序。我们将用这两种方法分别实现这一系列步骤。然后，你可以确定如何处理本书其余部分的程序。

注意　这些工具应该预安装在所有使用OS X的Mac机上。如果单独安装OS X，则务必要安装Developer Tools。

2.1.1 使用Xcode

Xcode是一款功能齐全的应用程序，使用它可轻松输入、编译、调试和执行程序。如果想在Mac机上开发应用程序，那么学习使用这个强大的工具是值得的。这里只是介绍入门知识，后面会再次回到Xcode并逐步介绍如何使用它开发图形应用程序。

首先，Xcode位于位于Developer文件夹内一个名称为Applications的子文件夹中。图2-1显示了该工具的图标。

图2-1　Xcode图标

启动Xcode，在File菜单下，选择New Project（参见图2-2）。

此时出现一个窗口，如图2-3所示。

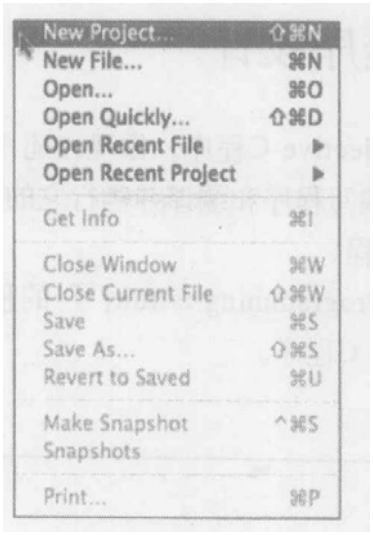

图2-2　启动一个新项目

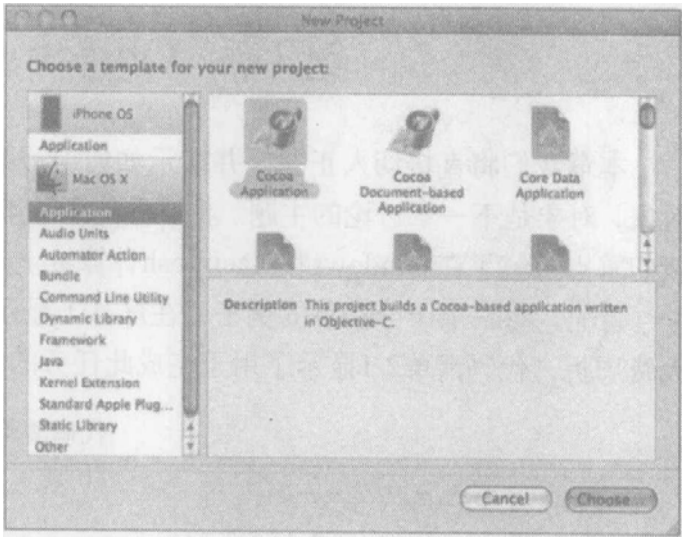

图2-3　启动一个新项目：选择应用程序类型

在左侧窗格中向下滚动，直至找到Command Line Utility。在右上窗格中，突出显示Foundation Tool，现在窗口应该如图2-4所示。

单击Choose，这会打开新窗口，如图2-5所示。

我们将第一个程序命名为prog1，所以在Save As字段中输入此名称。可以创建一个单独的文件夹来存储所有的项目。在我的系统上，我将本书中使用的项目保存在一个称为ObjC Progs

的文件夹中。

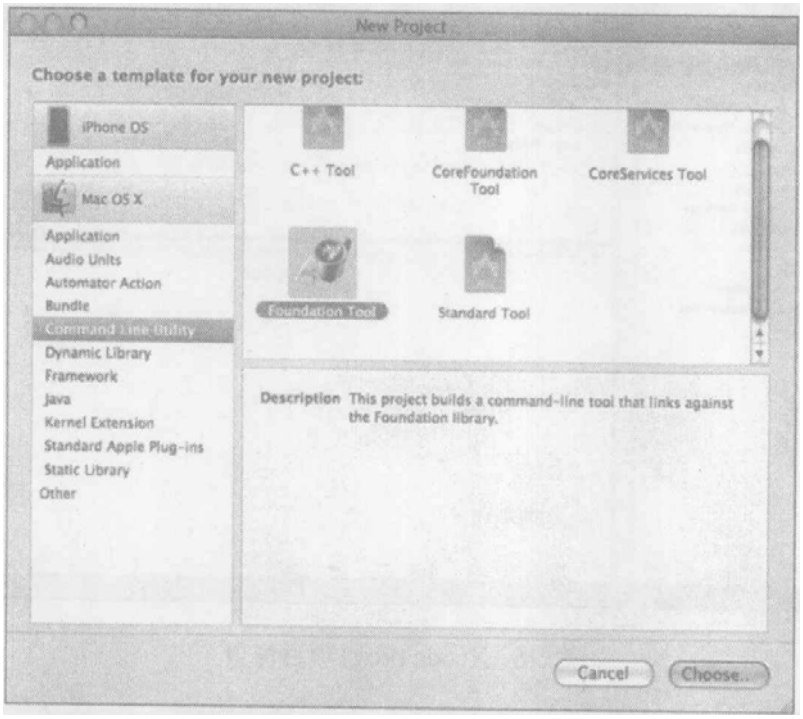

图2-4 启动一个新项目：创建Foundation Tool

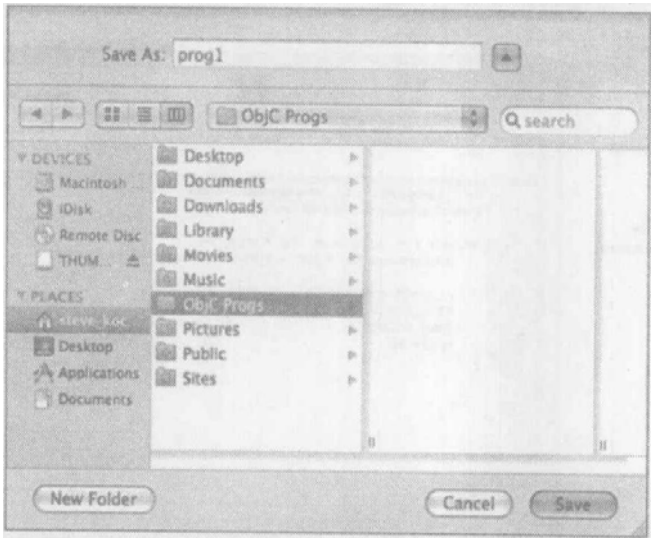

图2-5 Xcode文件列表窗口

单击Save按钮创建新项目，此时显示一个项目窗口，如图2-6所示。注意，如果你已经使用过Xcode或更改了其中的选项，那么显示的窗口可能与此处的稍有不同。

现在可以输入第一个程序了。在右上窗格中选择文件prog1.m，此时应该出现如图2-7所示的Xcode窗口。

Objective-C源文件使用.m作为文件名的最后两个字符（称为扩展名）。表2-1列出了其他常用的文件扩展名。

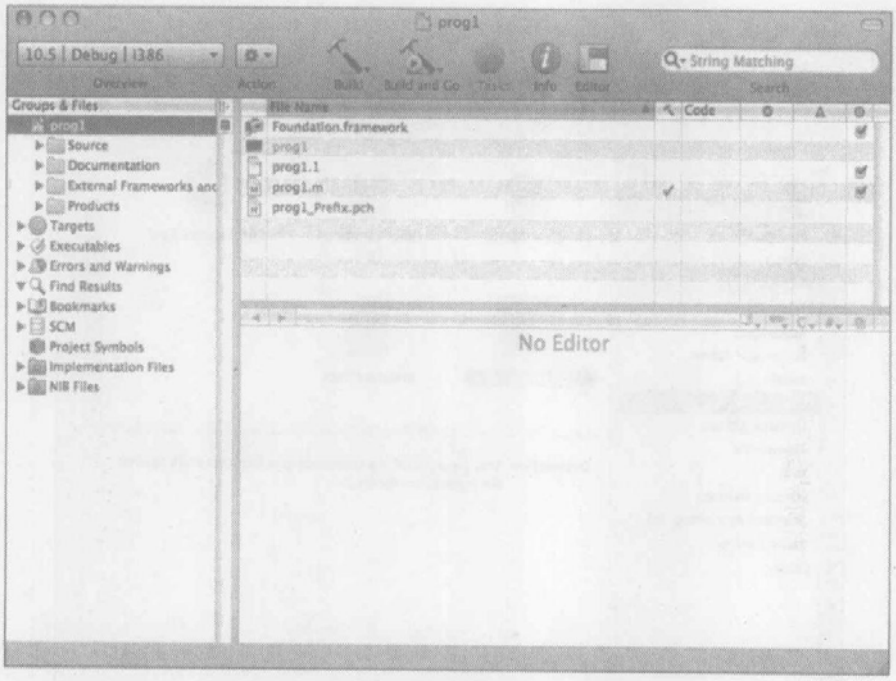

图2-6　Xcode prog1项目窗口

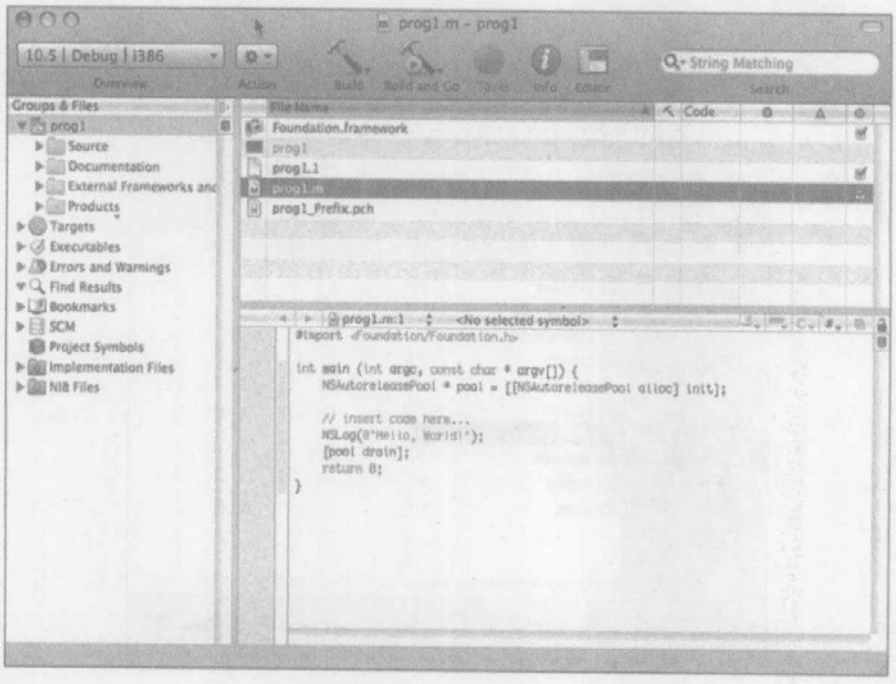

图2-7　文件prog1.m和编辑窗口

表2-1　常见的文件扩展名

扩展名	含　义	扩展名	含　义
.c	C语言源文件	.mm	Objective-C++源文件
.cc，.cpp	C++语言源文件	.pl	Perl源文件
.h	头文件	.o	Object（已编译的）文件
.m	Objective-C源文件		

返回Xcode项目窗口，窗口的右下侧显示了文件prog1.m并且包含以下代码：

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

int main (int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   // insert code here...
   NSLog (@"Hello World!");
   [pool drain];
   return 0;
}
```

注意 如果看到没有显示文件内容，则必须点击并向上拖动右下窗格，让编辑窗口显示出来。同样，如果以前使用过Xcode，可能会出现此情况。

可在该窗口内编辑文件。Xcode已创建了模板文件供你使用。

修改Edit窗口中显示的程序，使之与代码清单2-1相符。在prog1.m开始添加的行中，首先输入两个斜杠字符（//），这称为注释；稍后会更详细地介绍注释。

现在Edit窗口中的程序应该如下所示：

```
// First program example
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog (@"Programming is fun!");

   [pool drain];
   return 0;
}
```

不要担心屏幕上为文本显示的各种颜色。Xcode使用不同的颜色指示值、保留字等内容。

现在应该编译并运行第一个程序了，用Xcode的术语来说，称为构建并运行。但是，首先需要保存程序，方法是从File菜单中选择Save。如果在未保存文件的情况下尝试编译并运行程序，Xcode会询问你是否需要保存。

在Build菜单下，可以选择Build或Build and Run。我们选择后者，因为如果构建时不会出现任何错误，则会自动运行此程序。也可点击工具栏中出现的Build and Go图标。

注意 Build and Go意味着“构建，然后执行上次最后完成的操作”，这可能是Run、Debug、Run with Shark或Instruments等。首次为项目使用此图标时，Build and Go意味着构建并运行程序，所以此时使用这个操作没有问题。但是一定要知道“Build and Go”与“Build and Run”之间的区别。

如果程序中有错误，在此步骤期间会看到列出的出错消息。如果情况如此，可回到程序中解决错误问题，然后再次重复此过程。解决程序中的所有错误之后，会出现一个新窗口，其中

显示prog1-Debugger Console。这个窗口中包含了程序的输出，该内容应该与图2-8所示的类似。如果该窗口没有自动出现，可进入主菜单栏并从Run菜单中选择Console。稍后我们会讨论Console窗口的实际内容。

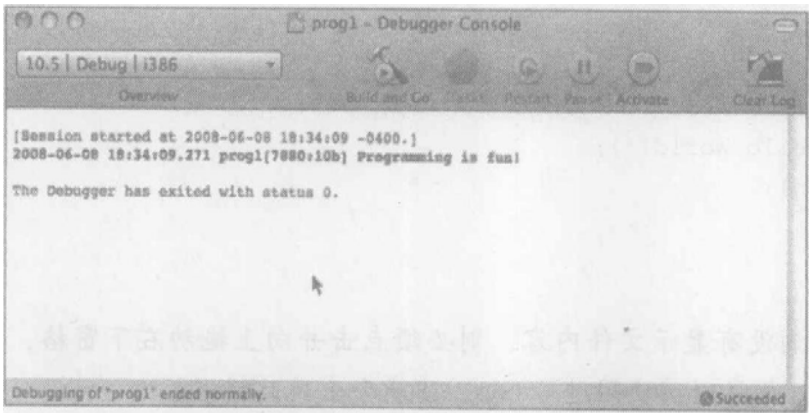

图2-8　Xcode Debugger Console窗口

使用Xcode编译并运行第一个程序的这部分过程已经完成了。下面总结一下使用Xcode创建新程序的步骤：

1. 启动Xcode应用程序。
2. 如果开发新项目，选择File、New Project。
3. 为应用程序类型选择Command Line Utility、Foundation Tool，然后点击Choose。
4. 选择项目名称，还可以选择在那个目录中存储项目文件，然后点击Save。
5. 在右上窗格中，会看到文件prog1.m（或者是你为项目起的其他名称，后面是.m）。突出显示该文件。在该窗口下面出现的编辑窗口中输入你的程序。
6. 选择File、Save，保存已完成的更改。
7. 选择Build、Build and Run，或者点击Build and Go按钮构建并运行程序。
8. 如果出现任何编译器错误或者输出内容不符合要求，对程序进行所需的更改并重复执行步骤6和7。

2.1.2　使用Terminal

有些人可能不想从学习使用Xcode开始Objective-C程序设计之旅。如果习惯于使用UNIX shell和命令行工具，可能会用Terminal应用程序编辑、编译和运行程序。下面说明如何使用Terminal执行上述操作。

图2-9　Terminal程序图标

第一步：启动Mac上的Terminal应用程序。Terminal应用程序位于Application文件夹下，存储在Utilities中。其图标如图2-9所示。

启动Terminal应用程序，将看到一个类似于图2-10的窗口。

命令要在每一行的$（或者%，这取决于Terminal应用程序的配置）之后键入。如果熟悉UNIX的使用，这是很容易理解的。

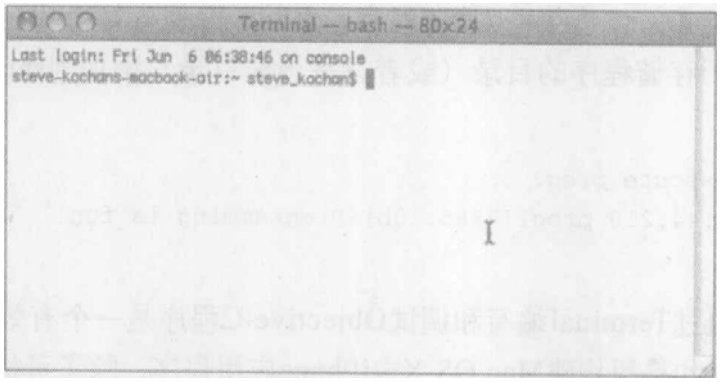

图2-10 Terminal窗口

首先，需要将代码清单2-1中的程序行输入到一个文件中。可以从创建一个用于存储这个程序例子的目录开始。然后，还必须运行一个文本编辑器（例如vi或者emacs）来输入这个程序：

```
sh-2.05a$ mkdir Progs       创建一个存储程序的目录
sh-2.05a$ cd Progs          转到新目录
sh-2.05a$ vi Prog1.m        启动文本编辑器来输入程序
..
```

注意 *在前一个例子以及本书的其余部分中，由你（即用户）输入的命令都用黑体表示。*

对于Objective-C文件，可以选择要使用的任何名；只要确保最后两个字符是.m即可。这样编译器就知道有一个Objective-C程序。

将程序输入到文件中之后，可以使用名为gcc的GNU Objective-C编译器来编译并链接这个程序。gcc命令的一般格式为：

```
gcc -framework Foundation files -o progname
```

该选项说明你要使用有关Foundation框架的信息：

```
-framework Foundation
```

要记住在命令行上使用该选项。files表示一列将要编译的文件。在我们的例子中，这样的文件只有一个，我们将其命名为prog1.m。如果编译时没有任何错误，那么包含这个可执行文件的文件名将是progname。

我们把这个程序名为prog1；则下面是用于编译第一个Objective-C程序的命令行：

```
$ gcc -framework Foundation prog1.m -o prog1 Compile prog1.m & call it prog1
$
```

命令提示符在返回时没有附带任何消息，这意味着编译器没有在程序中发现错误。现在，可以在命令提示符后键入名称prog1来继续执行这个程序：

```
$ prog1          Execute prog1
sh: prog1: command not found
$
```

除非事先已使用Terminal，否则，很可能得到上面的结果。导致这种情况的原因是：UNIX的shell（即运行该程序的应用程序）并不知道prog1位于何处（我们不在这里讲述所有细节），

所以此时有两种选择：其一，将字符./放在程序名之前，以便告知shell在当前目录查找要执行的程序。其二，将用于存储程序的目录（或者只是当前目录）添加到shell的PATH变量中。此处采用第一种方法：

```
$ ./prog1       Execute prog1
2008-06-08 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

应该注意的是，通过Terminal编写和调试Objective-C程序是一个有效的方法。但是，这并不是很好的长期策略。如果想构建Mac OS X或iPhone应用程序，除了可执行文件，还有很多文件都需要“打包”到应用程序软件包中。在Terminal中执行这些操作并不容易，而Xcode则对此很擅长。因此，我建议你首先学习如何使用Xcode开发程序。这样学起来有一定的难度，但所有努力最终会证明是值得的。

2.2 解释第一个程序

现在，你熟悉了编译并运行Objective-C程序的各个步骤，我们更详细地讲述这个程序。下面再次给出该程序。

```
// First program example
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSLog (@"Programming is fun!");
   [pool drain];
   return 0;
}
```

在Objective-C中，小写字母和大写字母是不同的。同样，Objective-C并不关心你在程序行的何处开始输入，程序行的任何位置都能输入语句。利用这个事实可以开发容易阅读的程序。

程序的第一行

```
// First program example
```

引入了注释的概念。

程序中使用的注释语句用于说明程序并增强程序的可读性。注释负责告诉该程序的读者，不管他是程序员还是其他负责维护该程序的人，这只是程序员在编写特定程序和特定语句序列时的想法。

向Objective-C程序中插入注释时，有两种方式。其一，使用两个连续的斜杠（//）。双斜杠后直到这行结尾的任何字符都将被编译器忽略。

注释还能以两个字符/和*开始。它们表示注释的开始，但必须终止这种注释。要终止注释，需要再次使用字符*和/，而且中间不能插入任何空格。开始/*和结束*/之间的所有字符都被看作注释语句的一部分，从而被Objective-C编译器忽略。注释跨越很多程序行时，通常使用这种

注释格式，例如以下所示：

```
/*
 This file implements a class called Fraction, which
 represents fractional numbers. Methods allow manipulation of
 fractions, such as addition, subtraction, etc.

 For more information, consult the document:
    /usr/docs/classes/fractions.pdf
*/
```

使用哪种风格的注释完全由你决定。只要注意：/*风格的注释不能嵌套使用。

在编写程序或者将其键入到计算机上时，应该养成在程序中插入注释的习惯。这种习惯有三方面的好处：首先，当特殊的程序逻辑在你的大脑中出现时就说明程序，要比程序完成后再回来重新思考这个逻辑简单得多。其次，通过在工作的早期阶段把注释插入程序中，可在调试阶段隔离和调试程序逻辑错误时受益匪浅。注释不仅可以帮助你（或者其他人）通读程序，而且还有助于指出逻辑错误的根源。最后，我甚至还发现有一个程序员实际上喜欢为程序添加注释。事实上，在调试完程序后，你很可能不喜欢再返回程序来插入注释。在开发程序过程中插入注释会使这项乏味的工作变得稍微容易。

代码清单2-1的下一个程序行

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

告诉编译器找到并处理名为Foundation.h的文件，这是一个系统文件，就是说，这个文件不是你创建的。#import表示将该文件的信息导入或包含到程序中，在这一点上，特别像把此文件的内容键入到程序中。要导入文件Foundation.h，因为它包含程序结尾处用到的其他类和函数的有关信息。

在代码清单2-1中，程序行

```
int main(int argc, const char *argv[])
```

指定程序的名称为main，它是一个特殊名称，用于准确地表示程序将在何处开始执行。main之前的保留字int指定main返回的值类型，该值为整型（很快将更加详细地讨论这个话题）。我们将暂时忽略出现在开始和结束圆括号之间的内容。它们与名为命令行参数的内容有关，我们将在第13章中再作讨论。

认识了针对系统的main,可以准确指定这个例程要执行哪些工作。把例程的所有程序语句放入到一对花括号中，可完成这项工作。在最简单的情况下，一条语句就是一个以分号结束的表达式。系统将把位于花括号中间的所有程序语句看作main例程的组成部分。代码清单2-1有4条语句。

代码清单2-1中的第一条语句是

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

这条语句为自动释放池在内存中保留了空间。我们会在第17章中全面讨论这个内容。作为模板的一部分，Xcode会将这行内容自动放入程序中，所以现在先不要管它。

下一条语句指定要调用名为NSLog的例程。传递或传送给NSLog例程的参数，或实参，是以下字符串：

```
@"programming is fun!"
```

此处@符号在位于一对双引号的字符串前面。这称为常量NSString对象。

注意　如果你有一定的C程序设计经验，可能会被前面的@字符所迷惑。如果没有前面的@字符，就是在编写常量C类型的字符串；有了这个符号，就是在编写NSString字符串对象。

NSLog例程是Objective-C库中的一个函数，它仅仅显示或记录其参数（或者参数列表，很快将会看到）。但是之前它会显示该例程的执行日期和时间、程序名，以及其他我们在此不会介绍的数值。在本书的后面部分中，我们不会列出NSLog在输出前面插入的这些文本。

Objective-C的所有程序语句必须使用分号（;）结束。这就是为什么分号会在NSLog调用的结束圆括号之后立即出现。

退出程序之前，应该使用类似于下面的语句释放已分配的内存池（以及与程序相关联的对象）：

```
[pool drain];
```

同样，Xcode会在程序中自动插入此行内容。我们在后面会详细介绍该语句的作用。

main中的最后一条程序语句是：

```
return 0;
```

它表示要终止main的执行并送回（或返回）一个状态值0。按照约定，0意味着程序正常结束。任何非零值通常表示出现了一些问题，例如，很可能无法找到程序所需的文件。

如果你正在使用Xcode，回顾Debug Console窗口（参见图2-8），会发现在NSLog输出的行之后将显示如下：

```
The Debugger has exited with status 0.
```

你应该理解这条消息在此处的含义。

现在，我们将结束第一个程序的讨论，对它进行修改以便同时显示短语“And programming in Objective-C is even more fun!”。这项工作可以通过简单地添加另一个对NSLog例程的调用来实现，如代码清单2-2所示。记住，每个Objective-C程序语句必须使用分号结束。

代码清单2-2

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

int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSLog (@"Programming is fun!");
    NSLog (@"Programming in Objective-C is even more fun!");
```

```
    [pool drain];
    return 0;
}
```

如果键入代码清单2-2，然后编译并运行它，你可能期望在终端上看到以下输出（同样，没有显示NSLog通常在输出前面加入的那些文本）。

代码清单2-2　输出

```
Programming is fun!
Programming in Objective-C is even more fun!
```

在下一个程序例子中将看到，无需为每行输出单独调用NSLog例程。

首先，让我们看看特殊的两字符序列。反斜杠（\）和字母n，合起来称为换行符。换行符通知系统要准确完成其名称所暗示的工作：转到一个新行。任何要在换行符之后输出的字符随后将出现在显示器的下一行。事实上，换行符非常类似于一台打字机上结束符（carriage return）的概念。

研究代码清单2-3中列出的程序，并在检验输出之前尝试预测结果（不要作弊，开始！）。

代码清单2-3

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSLog (@"Testing...\n..1\n...2\n....3");
   [pool drain];
   return 0;
}
```

代码清单2-3　输出

```
Testing...
..1
...2
....3
```

2.3　显示变量的值

用NSLog不仅能显示简单短语，还能显示变量的值以及计算结果。代码清单2-4使用NSLog例程显示两个数（即50和25）相加的结果。

代码清单2-4

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

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int sum;

    sum = 50 + 25;
    NSLog (@"The sum of 50 and 25 is %i", sum);
    [pool drain];

    return 0;
}
```

代码清单2-4　输出

```
The sum of 50 and 25 is 75
```

main中自动释放池后面的第一条程序语句将变量sum定义为整型。在应用程序中使用所有程序变量之前，都必须先定义它们。变量的定义向Objective-C编译器说明程序将如何使用这些变量。编译器需要这些信息来生成正确的指令，以便将值存储到变量中或者从变量中检索值。定义成int的变量只能用于存储整型值，就是说，没有小数位的值。整型值的例子有3、5、−20和0。带有小数位的数字，例如2.14、2.455和27.0，称为浮点数，它们是实数。

整型变量sum用于存储两个整数50和25相加的结果。我们故意在这个变量定义的下面留下一个空行，以便在视觉上区分例程的变量定义和程序语句；严格来讲，这是一个风格问题。有时候，在程序中添加单个空白行可使程序的可读性更强。

程序语句

```
sum = 50 + 25;
```

在作用上与其他大多数程序设计语言中一样：数字50和数字25相加（如加号所示），并把结果存储（如赋值运算符，或者等号所示）到变量sum中。

现在，代码清单2-4中的NSLog例程调用的圆括号中有两个参数。这些参数用逗号隔开。NSLog例程的第一个参数总是要显示的字符串。然而，在显示字符串的同时，通常还希望要显示某些程序变量的值。在这个例子中，你希望在显示字符之后，还要显示变量sum的值：

```
The sum of 50 and 25 is
```

第一个参数中的百分号是一个特殊字符，它可被NSLog函数识别。紧跟在百分号后的字符指定在这种情况下将要显示的值类型。在前一个程序中，字母i被NSLog例程识别，它表示将要显示的是一个整数。

只要NSLog例程在字符串中发现%i字符，它都将自动显示例程第二个参数的值。因为sum是NSLog的下一个参数，所以它的值将在显示字符The sum of 50 and 25 is之后自动显示。

现在，尝试预测代码清单2-5的输出。

代码清单2-5

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int value1, value2, sum;

   value1 = 50;
   value2 = 25;
   sum = value1 + value2;

   NSLog (@"The sum of %i and %i is %i", value1, value2, sum);

   [pool drain];
   return 0;
}
```

代码清单2-5 输出

```
The sum of 50 and 25 is 75
```

main中的第二条程序语句定义了3个int变量，分别名为value1、value2和sum。这条语句可以等价地表示成以下3条独立的语句：

```
int value1;
int value2;
int sum;
```

定义这3个变量后，程序将50赋值给变量value1，将25赋值给变量value2。然后计算两个变量的和并将结果赋值给变量sum。

现在，NSLog例程的调用包含4个参数。同样，第一个参数通常叫做格式字符串，它向系统描述其余参数的显示方式。value1的值将在The sum of之后立即显示。类似地，value2和sum两者的值将在适当的位置输出，该位置由格式字符串中后两个%i字符指定。

2.4 小结

本章是介绍性的一章，上述讨论包括了使用Objective-C开发程序的相关内容。至此，你应该对使用Objective-C进行程序设计有了很好的体验，而且应该能够独立地开发小程序。在下一章中，开始领略这个功能强大而且灵活的程序设计语言的一些更复杂的内容。但是首先，要尝试一下能否完成以下练习，确保你理解了本章解释的概念。

2.5 练习

1. 键入并运行本章出现的5个程序。将每个程序产生的输出与原文中每个程序后面列的输

出进行比较。

2. 编写一个可显示以下文本的程序：

```
In Objective-C, lowercase letters are significant.
main is where program execution begins.
Open and closed braces enclose program statements in a routine.
All program statements must be terminated by a semicolon
```

3. 以下程序输出什么内容？

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init;
   int i;

   i = 1;
   NSLog (@'Testing...');
   NSLog (@"....%i", i);
   NSLog (@"...%i",  i + 1);
   NSLog (@"..%i", i + 2);

   [pool drain];
   return 0;
}
```

4. 编写一个程序，执行87减15这个操作并显示其结果，同时还要显示一条适当的消息。

5. 找出以下程序中的语法错误。然后键入并运行改正之后的程序以确保找出了所有错误：

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

int main (int argc, const char *argv[]);
(
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   INT sum;
   /* COMPUTE RESULT //
   sum = 25 + 37 - 19
   / DISPLAY RESULTS /
   NSLog (@'The answer is %i' sum);

   [pool drain];
   return 0;
}
```

6. 以下程序输出什么结果？

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

```
int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int answer, result;

   answer = 100;
   result = answer - 10;

   NSLog (@"The result is %i\n", result + 5);

   [pool drain];
   return 0;
}
```

第 3 章　类、对象和方法

本章我们将学习面向对象程序设计的一些关键概念，并开始使用Objective-C中的类。你需要学习少量术语，我们将用非正式的形式向你介绍。本章只会讲解一些基本的术语，因为一下子讲太多你可能无法轻易接受。参见本书结尾的附录A，可以获得这些术语的更为精确定义。

3.1　到底什么是对象

对象就是一个事物。把面向对象的程序设计看成一个事物，而且想对这个事物做些工作。这与C语言不同，C语言通常称为过程性语言。在C语言中，通常是先考虑要做什么，然后才关注对象……这几乎总是与面向对象的思考过程相反。

我们举一个日常生活中的例子。假定你有一辆汽车，显然它是一个对象，而且是你拥有的一个对象。你并不是拥有任意一辆汽车，而是一辆特定的汽车，它由一家制造厂制造，可能在底特律，可能在日本，也可能在其他地方。你的汽车拥有一个车辆识别号码（vehicle identification number，VIN），它唯一标识你的汽车。

在面向对象的用语中，你的汽车是汽车的一个实例。如果继续使用术语，car就是类的名称，这个实例就是从该类创建的。因此，每制造一辆新汽车，就会创建汽车类的一个新实例，而且汽车的每个实例都称作一个对象。

你的汽车可能是银白色的，内部装饰为黑色，是辆敞篷车或者有金属顶盖，等等。此外，你还用这辆汽车执行特定的操作。例如，驾驶汽车、加油、（满怀希望地）洗车、接受维修，等等。这些情况在表3-1中做了描述。

表3-1所列的操作可以对你的汽车实现，也可以对其他汽车实现。例如，你姐姐会驾驶她的汽车、洗车、加油，等等。

表3-1　对象的操作

对象	使用对象执行的操作
你的汽车	驾驶
	加油
	洗车
	维修

3.2　实例和方法

类的独特存在就是一个实例，对实例执行的操作名为方法。在某些情况下，方法可以应用于类的实例或者类本身。例如，可将洗车作为一个实例（事实上，表3-1列出的所有方法都将作为实例方法）。如果把“找出一家制造厂制造了多少款汽车”应用于这个类，那么它将是一个类方法。

假设你有两辆使用装配线制作的汽车，它们看上去是一样的：都有相同的内部设计，相同的喷漆颜色，等等。它们可能同时启动，但是由于每部汽车是由它各自的主人驾驶的，这就使它们获得了自身独一无二的特性。例如，一辆汽车可能后来有了刮痕，而另一辆汽车可能行驶了更远的距离。每个实例或对象不仅包含从制造商那里获得的有关原始特性的信息，还包含它

的当前特性。这些特性可以动态改变。当驶汽车时，油箱的油渐渐耗尽，汽车越来越脏，轮胎也逐渐磨损。

对对象使用方法可以影响对象的状态。如果方法是“给汽车加油”，执行这个方法之后，汽车的油箱将会加满。这个方法影响了汽车油箱的状态。

这里的关键概念是：对象是类的独特表示，每个对象都包含一些通常对该对象来说是私有的信息（数据）。方法提供访问和改变这些数据的手段。

Objective-C采用特定的语法对类和实例应用方法：

```
[ ClassOrInstance method ];
```

在这条语句中，左方括号后要紧跟类的名称或者该类的实例的名称，它后面可以是一个或多个空格，空格后面是将要执行的方法。最后，使用右方括号和结束分号来终止。请求一个类或实例来执行某个操作时，就是在向它发送一条消息，消息的接收者称为接收者。因此，有另一种方式可以表示前面所描述的一般格式，如下所示：

```
[ receiver message ];
```

回顾上一个列表，使用这个新语法为它编写所有方法。在此之前，你需要获得一辆新车。去制造厂购买一辆，如下所示：

```
yourCar = [Car new];        得到一辆新车
```

向Car类（消息的接收者）发送一条消息请求它卖给你一辆新车。得到的对象（它代表你的独特汽车）将被存储到变量yourCar中。从现在开始，可用yourCar引用你的汽车实例，就是你从制造厂买的那辆汽车。

因为你到制造厂购买了一辆新汽车，所以这个新方法可叫做制造厂方法，或者类方法。对新车执行的其余操作都将是实例方法，因为它们应用于你的新车。下面是一些可能为这辆新车编写的示例消息表达式：

```
[yourCar prep];         准备好第一次使用
[yourCar drive];        驾驶汽车
[yourCar wash];         洗车
[yourCar getGas];       如果需要就给汽车加油
[yourCar service];      维修

[yourCar topDown];      是否为一辆敞篷车
[yourCar topUp];
currentMileage = [yourCar currentOdometer];
```

最后一个示例显示的实例方法可返回信息——即当前的行驶里程，这通过里程表（odometer）可看出来。我们将该信息存储在程序中的currentMileage变量内。

你姐姐Sue可以对她自己的汽车实例使用相同的方法：

```
[suesCar drive];
[suesCar wash];
[suesCar getGas];
```

将同一个方法应用于不同的对象是面向对象程序设计背后的主要概念之一，稍后将学到这

方面的更多内容。

你可能无需在程序中对汽车进行操作。你的对象很可能是面向计算机的对象，例如窗口、矩形、一段文本，或甚至是计算器或歌曲的播放列表。就像用于你的汽车的方法，这些方法可能看上去类似于：

```
[myWindow erase];                    清除窗口
[myRect getArea];                    计算矩形的面积
[UserText spellCheck];               对一些文本进行拼写检查
[deskCalculator clearEntry];         清除最后一次输入
[favoritePlaylist showSongs];        在播放列表中显示喜欢听的歌曲
[phoneNumber dial];                  拨号
```

3.3 用于处理分数的Objective-C类

现在我们将用Objective-C定义一个实际的类，并学习如何使用类的实例。

再次，我们将先学习过程。这样，实际的程序范例可能不是特别实用。那些更加实际的内容将在稍后讨论。

假设要编写一个用于处理分数的程序。可能需要处理加、减、乘、除等运算。如果不知道什么是类，那么可使用如下程序行来开始这个简单的程序：

代码清单3-1

```
// Simple program to work with fractions
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int  numerator = 1;
   int  denominator = 3;
   NSLog (@"The fraction is %i/%i", numerator, denominator);

   [pool drain];
   return 0;
}
```

代码清单3-1　输出

```
The fraction is 1/3
```

在代码清单3-1中，分数是以分子和分母的形式表示的。创建自动释放池之后，main中的前两行将变量numerator和denominator都声明为整型，并分别给它们赋予初值1和3。这两个程序行与下面的程序行等价：

```
int numerator, denominator;
```

```
numerator = 1;
denominator = 3;
```

通过将1存储到变量numerator中，将3存储到变量denominator中可表示分数1/3。如果需要在程序中存储多个分数，这种方法可能比较麻烦。每次要引用分数时，都必须引用相应的分子和分母。而且操作这些分数也相当困难。

如果能把一个分数定义成单个实体，用单个名称（例如myFraction）来共同引用它的分子和分母，那么会更好。这种方法可以使用Objective-C来实现，并从定义一个新类开始。

代码清单3-2使用一个名为Fraction的新类重写了代码清单3-1中的函数。下面给出这个程序，随后详细解释了它是如何工作的。

代码清单3-2

```
// Program to work with fractions - class version

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject
{
   int  numerator;
   int  denominator;
}

-(void)   print;
-(void)   setNumerator: (int) n;
-(void)   setDenominator: (int) d;

@end

//---- @implementation section ----


@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
   numerator = n;
}

-(void) setDenominator: (int) d
```

```
{
    denominator = d;
}

@end

//---- program section ----


int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction  *myFraction;

    // Create an instance of a Fraction

    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    // Set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Display the fraction using the print method

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
    return 0;
}
```

代码清单3-2 输出

```
The value of myFraction is:
1/3
```

从代码清单3-2的注释中可以看到，程序在逻辑上分为3个部分：

- @interface部分
- @implementation部分
- program部分

@interface部分用于描述类、类的数据成分以及类的方法；而@implementation部分包括实现这些方法的实际代码。最后，program部分包含实现程序预期目的的程序代码。

以上每个部分都是每个Objective-C程序的一部分，即使你可能不需要自己编写每一部分。

你将看到，每一部分通常放在它自己的文件中。然而，从现在起，我们将把它们放到一个单独的文件中。

3.4 @interface部分

定义新类时，必须做一些事情。首先，要通知Objective-C编译器这个类来自何处。就是说，必须命名它的父类。其次，必须确定这个类对象要存储的数据的类型。就是说，必须描述类成员将包含的数据。我们把这些成员叫做实例变量。最后，还必须定义在处理该类的对象时将要用到的各种操作或方法的类型。这些工作都在程序中名为@interface的特殊部分内完成。该部分的一般格式类似于：

```
@interface NewClassName: ParentClassName
{
    memberDeclarations;
}

methodDeclarations;
@end
```

按照约定，类名以大写字母开头，尽管没有要求这么做。但这种约定能使其他人在阅读你的程序时，仅仅通过观察名称的第一个字母就能把类名和其他变量类型区分开来。让我们暂时转换一下话题，先谈论一些在Objective-C中制定名称的有关规则。

3.4.1 选择名称

在第2章“用Objective-C进行程序设计”中，使用了几个变量来存储整型值。例如，在程序2.4中使用变量sum来存储两个数50和25相加的结果。

Objective-C语言还允许变量存储非整型的数据类型，只要在程序中使用变量之前，对它进行适当的声明即可。变量可以用于存储浮点数、字符，甚至是对象（或者更确切地说，是对对象的引用）。

制定名称的规则相当简单：名称必须以字母或下划线（_）开头，之后可以是任何（大写或小写）字母、下划线或者0到9之间的数字组合。下面是一列合法的名称：

- sum
- pieceFlag
- i
- myLocation
- numberOfMoves
- _sysFlag
- ChessBoard

另一方面，根据规定，以下名称是非法的：

- sum$value—$是一个非法字符。
- piece flag—名称中间不允许插入空格。

• 3Spencer—名称不能以数字开头。

• int—这是一个保留字。

int不能用作变量名，因为其用途对Objective-C编译器而言有特殊含义。这个使用称为保留名或保留字，一般来说，任何对Objective-C编译器有特殊意义的名称都不能作为变量名使用。附录B“Objective-C 2.0语言总结”，提供了这些保留名的完整列表。

应该始终记住，Objective-C中的大写字母和小写字母是有区别的。因此，变量名sum、Sum和SUM，都表示不同的变量。注意，在命名类时，类名要以大写字母开始。另一方面，实例变量、对象以及方法的名称，通常以小写字母开始。为使程序具有可读性，名称中要用大写字母来表示新单词的开始，如以下例子所示：

• AddressBook—可能是一个类名。

• currentEntry—可能是一个对象。

• current_entry——些程序员还使用下划线作为单词的分隔符。

• addNewEntry—可能是一个方法名。

确定名称时，要遵循同样的标准：千万不要偷懒。要找出能反应出变量或对象使用意图的名称。原因是明显的。就像使用注释语句一样，富有意义的名称可以显著增强程序的可读性，并可在调试和归档阶段受益匪浅。事实上，因为程序具有更强的自解释性（self-explanatory），所以归档的任务将很可能大大减少。

以下是代码清单3-2中的@interface部分：

```
//------- @interface section -------

@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

新类（NewClassName）的名称为Fraction，其父类为NSObject。（我们将在第8章“继承”中讲解有关父类的更详细内容。）NSObject类在文件NSObject.h中定义，导入Foundation.h文件时会在程序中自动包括这个类。

3.4.2 实例变量

memberDeclarations部分指定了哪种类型的数据将要存储到Fraction中，以及这些数据类型的名称。可以看到，这一部分放入自己的一组花括号内。对于Fraction类而言，声明

```
int numerator;
```

```
int denominator;
```

表示Fraction对象有两个名为numerator和denominator的整型成员。

在这一部分声明的成员称为实例变量。你将看到，每次创建新对象时，将同时创建一组新的实例变量，而且是唯一的一组。因此，如果拥有两个Fractions，一个名为fracA，另一个名为fracB，那么每一个都将有自己的一组实例变量。就是说，fracA和fracB各自将拥有独立的numerator和denominator。Objective-C系统将自动追踪这些实例变量，对使用对象而言，这是一件令人高兴的事。

3.4.3 类和实例方法

必须定义各种方法才能使用Fractions。需要能够将分数的值设为特定的值。因为你不能直接访问分数的内部表示（就是，直接访问它的实例变量），因此必须编写方法来设置分子和分母。还要编写一个名为print的方法来显示分数的值。下面是print方法的声明，应该位于接口文件中：

```
-(void) print;
```

开头的负号（-）通知Objective-C编译器，该方法是一个实例方法。其他唯一的选择是正号（+），它表示类方法。类方法是对类本身执行某些操作的方法，例如创建类的新实例。这类似于制造厂制造一辆新汽车，在这一点上，汽车就是一个类；而要制造新汽车便是类方法。

实例方法对类的特定实例执行一些操作，例如设置值、检索值和显示值等。在制造出一辆汽车后，引用这个汽车实例时，可能要执行给它加油的操作。这个加油操作是对特定的汽车执行的，因此它类似于实例方法。

返回值

声明新方法时，必须通知Objective-C编译器这个方法是否返回值，如果返回值，那么返回哪种类型的值。将返回类型放入开头的负号或者正号之后的圆括号中，可完成这项工作。因此，声明

```
-(int) retrieveNumerator;
```

指定名为retrieveNumerator的实例方法将返回一个整型值。类似地，程序行

```
-(double) retrieveDoubleValue;
```

声明了一个返回双精度值的方法（第4章“数据类型和表达式”会介绍有关这个数据类型的更多内容）。

使用Objective-C中的return语句可以从方法中返回一个值，这与前一个程序例子中从main内返回值的方式类似。

如果方法不返回值，可用void类型表明，如下所示：

```
-(void) print;
```

这条语句声明了一个名为print的方法，它不返回任何值。在这种情况下，无需在方法结尾执行一条return语句。相反，可以执行一条不带任何指定值的return语句，如下所示：

```
return;
```

无需为方法指定返回类型，尽管那样做是一个较好的编程习惯。如果没有指定任何类型，那么id将是默认类型。你将在第9章“多态、动态类型和动态绑定”中学到有关id数据类型的更多内容。基本上，id类型可用来引用任何类型的对象。

方法的参数

代码清单3-2的@interface部分声明了其他两个方法：

```
-(void) setNumerator:(int) n;
-(void) setDenominator:(int) d;
```

它们都是不返回值的实例方法。每个方法都有一个整型参数，这是通过参数名前面的（int）指明的。就setNumerator来说，其参数名是n。这个名称可以是任意的，它是用来引用参数的方法名称。因此，setNumerator的声明指定向该方法传递一个名为n的整型参数，而且该方法没有要返回的值。这类似于setDenominator的声明，不同之处是后者的参数名是d。

要注意声明这些方法的语法。每个方法名都以冒号结束，这个冒号通知Objective-C编译器该方法期望看到一个参数。然后，指定参数的类型，并将其放入一对圆括号中，这与为方法自身指定返回类型的方式十分相似。最后，使用象征性的名称来确定方法所指定的参数。整个声明以一个分号结束。这个语法在图3-1中做了描述。

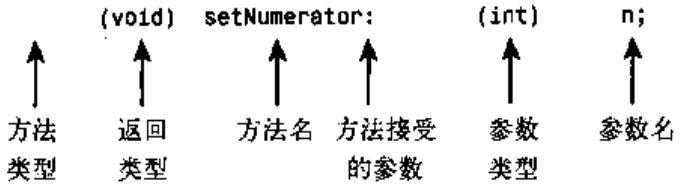

图3-1　方法的声明

如果方法接受一个参数，那么在引用该方法时在方法名之后附加一个冒号。因此，setNumerator:和setDenominator:是指定这两个方法的正确方式，每个方法都有一个参数。同样，在指定print方法时没有使用后缀的冒号，这表明此方法不带有任何参数。在第7章“类”中，将学习如何指定带有多个参数的方法。

3.5　@implementation部分

与注释的一样，@implementation部分包含声明在@interface部分的方法的实际代码。就像术语指出的那样，在@interface部分声明方法并在@implementation部分定义它们（就是说，给出实际的代码）。

@implementation部分的一般格式如下：

```
@implementation NewClassName
  methodDefinitions;
@end
```

NewClassName表示的名称与@interface部分的类名相同。可以在父类的名称之后使用冒号，如同在@interface部分使用的冒号一样：

```
@implementation Fraction: NSObject
```

然而，它是可选的而且通常并不这么做。

@implementation部分中的methodDefinitions部分包含在@interface部分指定的每个方法的代码。与@interface部分类似，每种方法的定义首先指定方法（类或者实例）的类型、返回类型、参数及其类型。然而，我们并没有使用分号来结束该行，而是将之后的方法代码放入一对花括号中。

以下是代码清单3-2的@implementation部分：

```
//---- @implementation section ----
@implementation Fraction
-(void) print
{
    NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end
```

print方法使用NSLog显示实例变量numerator和denominator的值。但是这个方法引用numerator还是denominator呢？它引用的实例变量包含在作为消息接收者的对象中。这是一个重要的概念，我们稍后再回来介绍。

setNumerator:方法具有一个名为n的整型参数，并简单地存储到实例变量numerator中。类似地，setdenominator:将其参数d的值存储到实例变量denominator中。

3.6 Program部分

program部分包含解决特定问题的代码，如果有必要，它可以跨越多个文件。前面提到，必须在其中一个地方有一个名为main的例程。通常情况下，这是程序开始执行的地方。以下是代码清单3-2的program部分：

```
//---- program section ----

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction  *myFraction;
```

```
    // Create an instance of a Fraction

    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    // Set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Display the fraction using the print method

    NSLog (@"The value of myFraction is:");
    [myFraction print];

    [myFraction release];
    [pool drain];

    return 0;
}
```

在main中，使用以下程序行定义了一个名为myFraction的变量：

```
Fraction *myFraction;
```

这个行表示myFraction是一个Fraction类型的对象；就是说，myFraction用于存储来自新的Fraction类变量。myFraction前面的星号（*）是必需的，但现在不用担心它的意义。在技术上，它实际表示myFraction是对Fraction的一个引用（或者指针）。

现在，你拥有一个用于存储Fraction的对象，需要创建一个分数，就像要求制造厂制造一辆汽车一样。可以用以下程序行实现：

```
myFraction = [Fraction alloc];
```

alloc是allocate的缩写。因为要为新分数分配内存存储空间。表达式

```
[Fraction alloc]
```

向新创建的Fraction类发送一条消息。你请求Fraction类使用alloc方法，但是你从未定义这个alloc方法，那么它来自何处呢？此方法继承自一个父类。第8章将会详细讨论这个主题。

将alloc消息发送给一个类时，便获得该类的新实例。在代码清单3-2中，返回值存储在变量myFraction中。alloc方法保证对象的所有实例变量都变成初始状态。然而，这并不意味着该对象进行了适当的初始化进而可以使用。在分配对象之后，还必须对它初始化。

这项工作可用代码清单3-2中的下一条语句来完成。如下：

```
myFraction = [myFraction init];
```

这里再次使用了一个并非自己编写的方法。init方法用于初始化类的实例变量。注意，你正将init消息发送给myFraction。就是说，要在这里初始化一个特殊的Fraction对象，因此它没

有发送给类，而是发送给了类的一个实例。继续下面内容之前，务必理解这一点。

init方法也可以返回一个值，即被初始化的对象。将返回值存储到Fraction的变量myFraction中。

下面两行代码分配了类的新实例并做了初始化，这两条消息在Objective-C中特别常见，通常组合到一起，如下所示：

```
myFraction = [[Fraction alloc] init];
```

内部消息表达式

```
[Fraction alloc]
```

将首先求值。可以看到，这条消息表达式的结果是已分配的实际Fraction。对它直接应用init方法，而不是像以前那样把分配结果存储到一个变量中。因此，再次先分配一个新的Fraction，然后对它初始化，最后把初始化的结果赋值给变量myFraction。

作为最终的简写形式，经常把分配和初始化直接合并到声明行，如下所示：

```
Fraction *myFraction = [[Fraction alloc] init];
```

这种编码的风格将在本书的其余部分广泛使用，因此理解这种风格非常重要。到目前为止，在每个程序中都看到了如何分配自动释放池：

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

此处将alloc消息发送给NSAutoreleasePool类，请求创建一个新实例。然后向新创建的对象发送init消息，以初始化该对象。

回到代码清单3-2，现在已经能够设置分数的值。程序行

```
//Set fraction to 1/3

[myFraction setNumerator:1];
[myFraction setDenominator:3];
```

用于完成这项工作。第一条消息语句把消息setNumerator:发送给myFraction。提供一个值为1的参数。然后将控制发送给为Fraction类定义的setNumerator:方法。因为Objective-C系统知道myFraction是Fraction类的对象，因此它知道setNumerator:是要从Fraction类使用的一个方法。

在setNumerator:方法中，将传递来的值1存储在变量n中。该方法中的唯一程序行获得该值并由实例变量numerator存储这个值。因此，myFraction的分子已经被有效地设置为1。

其后的消息用于调用myFraction的setDenominator:方法。在setDenominator:方法中，参数3被赋值给变量d。然后把这个值存储到实例变量denominator中，这样就将myFraction赋值为1/3。现在可以显示此分数的值，用代码清单3-2的如下代码行来实现：

```
//display the fraction using the print method

NSLog(@"The value of myFraction is:");
[myFraction print];
```

NSLog调用仅显示以下文本：

```
The value of myFraction is:
```

使用以下消息表达式调用print方法：

```
[myFraction print];
```

在print方法中，将显示实例变量numerator和denominator的值，并用斜杠字符（/）将其分隔。程序中的消息

```
[myFraction release];
```

用于释放Fraction对象使用的内存。执行这个操作是良好的程序设计风格的重要部分。只要创建一个新对象，都要请求为该对象分配内存。同样，在完成对该对象的操作时，必须释放它所使用的内存空间。尽管“程序以任何方式终止时，都将释放内存”是事实，但开始创建更复杂的应用程序之后，最终可以生成成百（或上千）的对象，它们占用大量内存。等到程序终止时才释放内存是对内存的浪费，这会减慢程序的执行速度，这种方式并非好的程序设计风格。因此，要从现在开始养成良好的习惯！

Apple的运行时系统提供了一个称为垃圾回收的机制，它可自动清理内存。但是，最好要学会如何自己管理内存的使用，而不是依赖于自动的机制。事实上，针对不支持垃圾回收的某些平台（如iPhone）进行程序设计时，就不能依赖于垃圾回收机制。所以，我们在本书的后面才会介绍这个机制。

似乎要在代码清单3-2中编写大量的代码才能完成代码清单3-1所实现的工作。对于此处这个简单的例子来说确实如此；然而，使用对象的最终目的是使程序易于编写、维护和扩展。以后你将认识到这一点。

本章最后一个例子展示如何在程序中使用多个分数。在代码清单3-3中，将一个分数设置为2/3，另一个设置为3/7，然后同时显示它们。

代码清单3-3

```
// Program to work with fractions - cont'd

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject
{
   int numerator;
   int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

```
//---- @implementation section ----

@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
   numerator = n;
}

-(void) setDenominator: (int) d
{
   denominator = d;
}

@end

//---- program section ----

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction  *frac1 = [[Fraction alloc] init];
   Fraction  *frac2 = [[Fraction alloc] init];

   // Set 1st fraction to 2/3

   [frac1 setNumerator: 2];
   [frac1 setDenominator: 3];

   // Set 2nd fraction to 3/7

   [frac2 setNumerator: 3];
   [frac2 setDenominator: 7];

   // Display the fractions

   NSLog (@"First fraction is:");
   [frac1 print];

   NSLog (@"Second fraction is:");
   [frac2 print];
```

```
    [frac1 release];
    [frac2 release];

    [pool drain];
    return 0;
}
```

代码清单3-3　输出

```
First fraction is:
2/3
Second fraction is:
3/7
```

@interface和@implementation部分和代码清单3-2一样。这个程序创建了两个名为frac1和frac2的Fraction对象，然后将它们分别赋值为2/3和3/7。注意下面这一点，当frac1使用setNumerator:方法将其分子设置为2时，实例变量frac1也将实例变量numerator设置为2。同样，frac2使用相同的方法将其分子设置为3时，它特有的实例变量numerator也被设置为3。每次创建新类时，该类就获得了自己特有的一组实例变量。图3-2描述了这个情况。

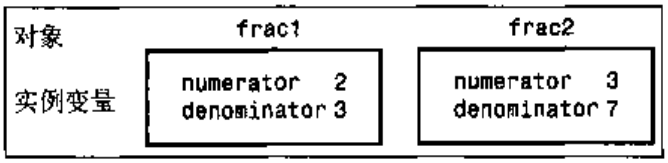

图3-2　特有的实例变量

根据要发送消息的对象，引用正确的实例变量。因此，在

```
[frac1 setNumerator: 2];
```

中，只要setNumerator:在方法中使用名称numerator，引用的都是frac1的numerator。这是因为frac1是此消息的接收者。

3.7　实例变量的访问以及数据封装

你已经看到处理分数的方法如何通过名称直接访问两个实例变量numerator和denominator的情况。事实上，实例方法总是可以直接访问它的实例变量。然而，类方法则不能，因为它只处理类本身，并不处理类的任何实例（仔细考虑一会儿）。但是，如果要从其他位置访问实例变量，例如，从main例程内部来访问，该如何实现？在这种情况下，不能直接访问这些实例变量，因为它们是隐藏的。将实例变量隐藏起来的这种做法实际上是称为数据封装的关键概念。它使得编写类定义的人在不必担心程序员（即类的使用者）是否破坏类的内部细节的情况下，扩展和修改他的类定义。数据封装提供了程序员和类的开发者之间的良好隔离层。

通过编写特殊方法来检索实例变量的值，可以用一种新的方式来访问它们。例如，创建两个非常恰当地命名为numerator和denominator的方法，它们用来访问作为消息接收者的Fraction的相应实例变量。结果将是相应整数值，你将返回这些值。以下是这两个新方法的声明：

```
-(int) numerator;
-(int) denominator;
```

下面是定义：

```
-(int) numerator
{
   return numerator;
}

-(int) denominator
{
   return denominator;
}
```

注意，它们访问的方法名和实例变量名是相同的。这样做不存在任何问题，事实上，这是很常见的情况。代码清单3-4用来测试这两个新方法。

代码清单3-4

```
// Program to access instance variables - cont'd

#import <Foundation/Foundation.h>
//---- @interface section ----

@interface Fraction: NSObject
{
   int  numerator;
   int  denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;

@end

//---- @implementation section ----

@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
```

```
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

@end


//---- program section ----

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction  *myFraction = [[Fraction alloc] init];

    // Set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Display the fraction using our two new methods

    NSLog (@"The value of myFraction is: %i/%i",
       [myFraction numerator], [myFraction denominator]);
    [myFraction release];
    [pool drain];

    return 0;
}
```

代码清单3-4 输出

```
The value of myFraction is 1/3
```

NSlog语句

```
NSLog (@"The value of myFraction is: %i/%i",
       [myFraction numerator], [myFraction denominator]);
```

显示发送给myFraction:的两条消息的结果，第一条消息检索numerator的值，而第二条则检索

denominator的值。

顺便说一下，设置实例变量值的方法通常总的称为设置函数（setter），而用于检索实例变量值的方法叫做获取函数（getter）。对于Fraction类而言，setNumerator:和setdenominator:是设置函数，numerator和denominator是获取函数。

注意 不久就会看到Objective-C 2.0一个很方便的特性，即允许自动创建设置函数方法和获取函数方法。

这里还应指出，还有一个名为new的方法可以将alloc和init的操作结合起来。因此，程序行

```
Fraction *myFraction = [Fraction new];
```

可用于分配和初始化新的Fraction。

用两步来实现分配和初始化的方式通常更好，这样可以在概念上理解正在发生两个不同的事件：首先创建一个对象，然后对它初始化。

3.8 小结

现在，你知道如何定义自己的类，如何创建该类的对象或实例，以及如何向这些对象发送消息。我们将在随后的章节中介绍Fraction类。你将了解到如何向某个方法传递多个参数，如何将类定义划分到不同的文件，同时还将了解到一些关键概念，如继承和动态绑定。然而，现在需要学习有关数据类型以及使用Objective-C编写表达式的更多内容。首先，尝试完成以下练习，测试是否已经理解本章所讲的重点。

3.9 练习

1. 下列名称中，哪些是不合法的？为什么？

Int	playNextSong	6_05
_calloc	Xx	alphaBetaRoutine
clearScreen	_1312	z
ReInitialize	_	AS

2. 根据本章中的汽车示例，想象一个每天都要使用的对象。为这个对象确定一个类，并编写5个用于处理该对象的操作。
3. 给定练习2中的程序清单，使用以下语法：

```
[instance method];
```

中的格式重写程序清单。

4. 设想你拥有一艘船、一辆摩托车和一辆汽车。列出对其中每个对象执行的操作。这些操作之间有重叠吗？
5. 根据问题4，设想有一个名为Vehicle的类和一个名为myVehicle的对象，这个对象可以是Car、Motorcycle或Boat中的任何一个。如果编写以下操作：

```
[myVehicle prep];
[myVehicle getGas];
```

```
[myVehicle service];
```

向来自这几个类中的某个对象应用操作，知道这样做的好处吗？

6. 在C这样的过程性语言中，思考涉及各种对象的操作，然后编写代码来执行这些操作。参见汽车例子，可用C编写洗交通工具的过程，然后在该过程中编写代码来处理清洗汽车、清洗游艇以及清洗摩托车等操作。如果采用这种方法，同时希望添加一种新的交通工具（参见以前的练习），那么能指出使用这种过程性的方法比使用面向对象的方法有什么好处和缺点吗？
7. 定义一个名为XYpoint的类，用来保存笛卡儿坐标（x, y），其中x和y均为整数。定义一些方法，分别用来设置点的坐标x和y，并检索它们的值。编写一个Objective-C程序，实现这个新类并测试它。

第 4 章　数据类型和表达式

本章中，我们将讲解Objective-C的基本数据类型，并描述一些用于构成算术表达式的基本规则。

4.1　数据类型和常量

你已经接触过Objective-C的基本数据类型int。回顾一下，声明为int类型的变量只能用于保存整型值，也就是没有小数位的值。

Objective-C还提供了另外3种基本数据类型：float、double和char。声明为float类型的变量可存储浮点数（即包含小数位的值）。double类型和float类型一样，只不过前者的精度约是后者精度的两倍而已。最后，char数据类型可存储单个字符，例如字母a、数字字符6或者一个分号（稍后将详细讨论有关内容）。

在Objective-C中，任何数字、单个字符或者字符串通常都称为常量。例如，数字58表示一个常量整数值，字符串@“Programming in Objective-C is fun.\n”表示一个常量字符串对象。完全由常量值组成的表达式叫做常量表达式。因此，表达式

```
128 + 7 - 17
```

是一个常量表达式，因为该表达式的每一项都是常量值。然而，如果将i声明为整型变量，那么表达式

```
128 + 7 - i
```

就不是一个常量表达式。

4.1.1　int类型

在Objective-C中，整数常量由一个或多个数字的序列组成。序列前的负号表示该值是一个负数。值158、−10和0都是合法的整数常量。数字中间不允许插入空格，大于999的值不能用逗号来表示（因此，值12 000是一个非法的整数常量，它必须写成12000）。

Objective-C中存在两种特殊的格式，它们用一种非十进的方式来表示整数常量。如果整型值的第一位是0，那么这个整数将用八进制计数法来表示，就是说用基数8来表示。在这种情况下，该值的其余位必须是合法的八进制数字，那必须是0到7之间的数字。因此，在Objective-C中以八进制表示的值50（等价于10进制的值40），表示方式为050。与此类似，八进制的常量0177表示十进制的值127（$1 \times 64 + 7 \times 8 + 7$）。通过在NSLog调用的格式字符串中使用格式符号%o，可在终端上用八进制显示整型值。在这种情况下，用八进制显示的值不带有前导0，而格式符号%#o将在八进制值的前面显示前导0。

如果整型常量以0和字母x（无论是小写字母还是大写字母）开头，那么这个值都将用十六

进制（以16为基数）计数法来表示。紧跟在字母x后的是十六进制值的数字，它可由0到9之间的数字和a到f（或A到F）之间的字母组成。字母表示的数字分别为10到15。因此，要给名为rgbColor的整型常量指派十六进制的值FFEF0D，可使用以下语句：

```
rgbColor = 0xFFEF0D;
```

格式符号%x将用十六进制格式显示一个值，该值不带前导的0x，并用a到f之间的小写字符表示十六进制数字。要使用前导0x显示该值，使用格式字符%#x，如下所示：

```
NSLog("Color is %#x\n", rgbColor);
```

%X或%#X中的大写字母X，可用于显示前导的x，而随后用大写字母表示的十六进制数字。

每个值，不管是字符、整数还是浮点数字，都有与其对应的值域。这个值域与存储特定类型的值而分配的内存量有关。一般来说，在语言中没有规定这个量，它通常依赖于所运行的计算机，因此叫做设备或机器相关量。例如，一个整数可在计算机上占用32位，或者可以使用64位存储。

永远不要编写假定数据类型大小的程序。然而，要保证为每种基本数据类型留出最小数量的内存。例如，要保证整型值存储在32位中。然而，这一点不能保证。参见附录B中的表B-3，了解有关数据类型大小的更多信息。

4.1.2 float类型

声明为float类型的变量可存储包含小数位的值。要区分浮点常量，可通过查看其是否包含小数点。可以省略小数点之前的数字，也可以省略之后的数字，然而，显然不能将它们全部省略。值3.、125.8及-.0001都是合法的浮点常量。要显示浮点值，可用NSLog转换字符%f。

浮点常量也能使用所谓的科学计数法来表示。值1.7e4就是使用这种计数法来表示的浮点值，它表示值1.7×10^4。位于字母e前的值称为尾数，而之后的值称为指数。指数前面可以放置正号或负号，指数表示将与尾数相乘的10的幂。因此，在常量2.25e-3中，2.25是尾数值，而-3是指数值。该常量表示值2.25×10^{-3}，或0.00225。顺便说一下，用于分隔尾数和指数的字母e，可用大写字母，也可用小写字母。

要用科学计数法显示值，应该在NSLog格式字符串中指定格式字符%e。使用NSLog格式字符串%g允许NSLog确定使用常用的浮点计数法还是使用科学计数法来显示浮点值。这一决定取决于指数的值：如果该值小于-4或大于5，采用%e（科学计数法）表示，否则采用%f（浮点计数法）。

十六进制的浮点常量包含前导的0x或0X，后面紧跟一个或多个十进制或十六进制数字，再后是p或P，最后是可以带符号的二进制指数。例如，0x0.3p10表示的值为$3/16 \times 2^{10} = 192$。

4.1.3 double类型

类型double与类型float非常相似，而在float变量所提供的值域不能满足要求时，就要使用double变量。声明为double类型的变量可存储的位数大概是float变量所存储的两倍多。大多数计算机使用64位来表示double值。

除非另有说明，否则，Objective-C编译器将所有的浮点常量均看作double值。要清楚地表示float常量，需要在数字的尾部添加一个f或F，例如：

```
12.5f
```

要显示double值，可用格式符号%f、%e或%g，它们与显示float值所用的格式符号是相同的。

4.1.4 char类型

char变量可存储单个字符。将字符放入一对单引号中就能得到字符常量。因此，'a'，';'和'0'都是合法的字符常量。第一个常量表示字母a，第二个表示分号，第三个表示字符0，它并不等同于数字0。不要把字符常量和C风格的字符串混为一谈，字符常量是放在单引号中的单个字符，而字符串则是放在双引号中的任意个数的字符。正如在上一章提及的，前面有@字符并且放在双引号中的字符串是NSString字符串对象。

注意 附录B讨论了存储字符的方法，这些字符来自扩展的字符集，包括特殊的转义序列、通用字符以及宽位字符。

字符常量'\n'（即换行符）是一个合法的字符常量，尽管它似乎与前面提到的规则矛盾。出现这种情况的原因是反斜杠符号是Objective-C系统中的特殊符号，实际上并不把它看成一个字符。换句话说，Objective-C编译器将字符'\n'看作单个字符，尽管它实际上由两个字符组成。其他的特殊字符由反斜杠字符开头。要了解全部特殊字符，可参见附录B。在NSLog调用中可以使用格式字符%c，以便显示char变量的值。

在代码清单4-1中，使用了基本的Objective-C数据类型。

代码清单4-1

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int    integerVar = 100;
    float  floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char   charVar = 'W';

    NSLog (@"integerVar = %i", integerVar);
    NSLog (@"floatingVar = %f", floatingVar);
    NSLog (@"doubleVar = %e", doubleVar);
    NSLog (@"doubleVar = %g", doubleVar);
    NSLog (@"charVar = %c", charVar);

    [pool drain];
    return 0;
}
```

代码清单4-1　输出

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = 'W'
```

在程序输出的第二行，你会注意到指派给floatingVar的值331.79，实际显示成了331.790009。事实上，实际显示的值是由使用的特定计算机系统决定的。出现这种不准确值的原因是：计算机内部使用特殊的方式表示数字。使用计算器处理数字时，很可能遇到相同的不准确性。如果用计算器计算1除以3，将得到结果.33333333，很可能结尾带有一些附加的3。这串3是计算器计算1/3的近似值。理论上，应该存在无限个3。然而该计算器只能保存这些位的数字，这就是计算机的不确定性。此处应用了相同类型的不确定性：在计算机内存中不能精确地表示一些浮点值。

4.1.5　限定词：long、long long、short、unsigned及signed

如果直接把限定词long放在int声明之前，那么所声明的整型变量在某些计算机上具有扩展的值域。一个long int声明的例子为：

```
long int factorial;
```

这条语句将变量fractorial声明为long的整型变量。就像float和double变量一样，long变量的具体精度也是由具体的计算机系统决定。在许多系统上，int与long int具有相同的值域，而且任何一个都能存储32位宽（$2^{31}-1$，或2,147,483,647）的整型值。

long int类型的常量值可通过在整型常量末尾添加字母L（大小写均可）来形成。但数字和L之间不允许有空格。因此，声明

```
long int numberOfPoints = 131071100L;
```

将变量numberOfPoints声明为long int类型，而且初值为131,071,100。

要用NSLog显示long int的值，使用字母l作为修饰符并放在整型格式符号i、o和x之前。这意味着格式符号%li用十进制格式显示long int的值，符号%lo用八进制格式显示值，而符号%lx则用十六进制格式显示值。

可以如下形式使用long long的整型数据类型：

```
long long int maxAllowedStorage;
```

这条语句把将指定的变量声明为具有特定扩展精度的变量，该扩展精度保证变量至少具有64位的宽度。NSLog字符串不使用单个字母l，而使用两个l来显示long long的整数，例如“%lli”。

同样可将long标识符放在double声明之前，如下：

```
long double US_deficit_2004;
```

long double常量可写成其尾部带有字母l或L的浮点常量，如下：

```
1.234e+7L
```

要显示long double的值，需要使用修饰符L。因此，%Lf用浮点计数法显示long double的值，%Le用科学计数法显示同样的值，而%Lg将告诉NSLog在%Lf和%Le之间任选一个使用。

把限定词short放在int声明之前时，它告诉Objective-C编译器要声明的特定变量用来存储相当小的整数。之所以使用short变量，主要原因是对节约内存空间的考虑，当程序员需要大量内存而可用的内存量又十分有限时，就可用short变量来解决这个问题。

在某些计算机上，short int占用的内存空间是常规int变量所占空间的一半。在任何情况下，确保分配给short int的空间数量不少于16位。

在Objective-C中，没有其他方法可显式地编写short int型常量。要显示short int变量，可将字母h放在任何普通的整型转换符号之前，如：%hi、%ho或%hx。换句话说，可用任何整型转换符号来显示short int，因为当它作为参数传递给NSLog例程时，可转换成整数。

最终限定词可放在int变量之前，当整数变量只用来存储正数的情况下使用最终限定符。以下语句

```
unsigned int counter;
```

向编译器声明：变量counter只用于保存正值。通过限制整型变量的使用，使它专门存储正整数，可以扩展整型变量的精度。

通过将字母u（或U）放在常量之后，可产生unsigned int常量，如下：

```
0x00ffU
```

编写整型常量时，可将字母u（或U）和l（或L）组合起来使用，因此

```
20000UL
```

告诉编译器将常量20000看作unsigned long。

如果整型常量之后不带有字母u、U、l或L中的任何一个，而且它太大以至于不适合用普通大小的int表示，那么编译器将把它看作unsigned int值。如果它太小以至于不适合用unsigned int表示，那么编译器将把它看作long int。如果仍然不适合用long int表示，编译器就会把它视为unsigned long int。

将变量声明为long int、short int或unsigned int类型时，关键字int可以省略。因此，unsigned变量counter可等价地声明为以下形式：

```
unsigned counter;
```

同样可将char变量声明为unsigned。

signed限定词可明确地告诉编译器特定变量是有符号的。它主要用在char声明前面，有关它的深层讨论已超出了本书的范围。

4.1.6 id类型

id数据类型可存储任何类型的对象。从某种意义说，它是一般对象类型。例如，程序行

```
id number;
```

将number声明为id类型的变量。可声明方法使其具有id类型的返回值，如下：

```
-(id) newObject: (int) type;
```

这个程序行声明了一个名为newObject的实例方法，它具有名为type的单个整型参数并有id类型的返回值。应该注意，对返回值和参数类型声明来说，id是默认的类型。因此，以下程序行

```
+allocInit;
```

声明了一个返回id类型值的类方法。

id数据类型是本书经常使用的一种重要数据类型。这里介绍该类型的目的是为了保持本书的完整性。id类型是Objective-C中十分重要的特性，它是多态和动态绑定的基础，这两个特性将在第9章“多态、动态类型和动态绑定”中详细讨论。

表4-1总结了基本数据类型和限定词。

表4-1　基本数据类型

类　型	常量实例	NSlog字符
char	'a', '\n'	%c
short int	—	%hi、%hx、%ho
unsigned short int	—	%hu、%hx、%ho
int	12、-97、0xFFE0、0177	%i、%x、%o
unsigned int	12u、100U、0XFFu	%u、%x、%o
long int	12L、-2001、0xffffL	%li、%lx、%lo
unsigned long int	12UL、100ul、0xffeeUL	%lu、%lx、%lo
long long int	0xe5e5c5e5LL、500ll	%lli、%llx、&llo
unsigned long long int	12ull、0xffeeULL	%llu、%llx、%llo
float	12.34f、3.1e-5f、0x1.5p10、0x1p-1	%f、%e、%g、%a
double	12.34、3.1e-5、0x.1p3	%f，%e，%g，%a
long double	12.341、3.1e-51	%Lf，$Le，%Lg
id	nil	%p

4.2　算术表达式

在Objective-C中，事实上与所有程序设计语言一样，在两个数相加时使用加号（+），在两个数相减时使用减号（-），在两个数相乘时使用乘号（*），在两个数相除时使用除号（/）。这些运算符称为二元算术运算符，因为它们运算两个值或项。

4.2.1　运算符的优先级

你已经看到如何在Objective-C中执行简单运算，例如加法。下面的程序进一步说明了减法、乘法和除法运算。在程序中执行的最后两个运算引入了一个运算符比另一个运算符有更高优先级或优先级的概念。事实上，Objective-C中的每一个运算符都有与之相关的优先级。

该优先级用于确定拥有多个运算符的表达式如何求值：优先级较高的运算符首先求值。如果表达式包含优先级相同的运算符，可按照从左到右或从右到左的方向来求值，具体按哪个方向求值取决于运算符。这就是通常所说的运算符结合性。附录B提供了有关运算符优先级及其结合规则的完整列表。

代码清单4-2

```
// Illustrate the use of various arithmetic operators

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int   a = 100;
   int   b = 2;
   int   c = 25;
   int   d = 4;
   int   result;

   result = a - b;   //subtraction
   NSLog (@"a - b = %i", result);

   result = b * c;   //multiplication
   NSLog (@"b * c = %i", result);

   result = a / c;   //division
   NSLog (@"a / c = %i", result);

   result = a + b * c;   //precedence
   NSLog (@"a + b * c = %i", result);

   NSLog (@"a * b + c * d = %i", a * b + c * d);

   [pool drain];
   return 0;
}
```

代码清单4-2 输出

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

在声明整型变量a、b、c、d及result之后，程序将a减b的结果指派给result，然后用恰当的NSLog调用来显示它的值。

下一条语句

```
result = b * c;
```

将b的值和c的值相乘并将其结果存储到result中。然后用NSLog调用来显示这个乘法的结果，到

目前为止，你应该很熟悉该过程了。

之后的程序语句引入了除法运算符——斜杠。执行100除以25得到结果4，可用NSLog语句在a除以c之后立即显示。

在某些计算机系统上，尝试将一个数除以0将导致程序异常终止或出现异常。即使程序没有异常终止，执行这样的除法所得的结果也毫无意义。在第6章“选择结构”中，将看到如何在执行除法运算之前检验除数是否为0。如果除数为0，可采用适当的操作来避免除法运算。

表达式

```
a + b * c
```

不会产生结果2550（102×25）；相反，相应的NSLog语句显示的结果为150。这是因为Objective-C与其他大多数程序设计语言一样，对于表达式中多重运算或项的顺序有自己规则。通常情况下，表达式的计算按从左到右的顺序执行。然而，为乘法和除法运算指定的优先级比加法和加法的优先级要高。因此，Objective-C认为表达式

```
a + b * c
```

等价于

```
a + (b * c)
```

（如果采用基本的代数规则，那么该表达式的计算方式是相同的。）

如果要改变表达式中项的计算顺序，可使用圆括号。事实上，前面列出的表达式是相当合法的Objective-C表达式。这样，可用表达式

```
result = a + (b * c);
```

替换代码清单4-2中的表达式，也可以获得同样的结果。

然而，如果用表达式

```
result = (a + b) * c;
```

来替换，则指派给result的值将是2550，因为要首先将a的值（100）和b的值（2）相加，然后再将结果与c的值（25）相乘。圆括号也可以嵌套，在这种情况下，表达式的计算要从最里面的一对圆括号依次向外进行。只要确保结束圆括号和开始圆括号数目相等即可。

从代码清单4-2中的最后一条语句可发现，对NSLog指定表达式作为参数时，无需将该表达式的结果先指派给一个变量，这种做法是完全合法的。表达式

```
a * b + c * d
```

可根据以上述规则使用

```
(a * b) + (c * d)
```

即

```
(100 * 2) + (25 * 4)
```

来计算。

求出的结果300将传递给NSLog例程。

4.2.2 整数运算和一元负号运算符

代码清单4-3巩固了前面讨论的内容，并引入了整数运算的概念。

代码清单4-3

```
// More arithmetic expressions

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int   a = 25;
   int   b = 2;
   int   result;
   float c = 25.0;
   float d = 2.0;

   NSLog (@"6 + a / 5 * b = %i", 6 + a / 5 * b);
   NSLog (@"a / b * b = %i", a / b * b);
   NSLog (@"c / d * d = %f", c / d * d);
   NSLog (@"-a = %i", -a);

   [pool drain];
   return 0;
}
```

代码清单4-3 输出

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

前三条语句中，在int和a、b及result的声明之间插入了额外的空格，以便对齐每个变量的声明。使用这种方法书写语句可使程序更容易阅读。还可以注意到，在迄今出现的每个程序中，每个运算符前后都有空格。这种做法同样不是必需的，仅仅是出于美观上的考虑。一般来说，在允许单个空格的任何位置都可以插入额外的空格。如果能使程序更容易阅读，敲击空格键的操作还是值得做的。

在代码清单4-3中，第一个NSLog调用中的表达式巩固了运算符优先级的概念。该表达式的计算按以下顺序执行：

1. 因为除法的优先级比加法高，所以先将a的值（25）除以5。该运算将给出中间结果4。

2. 因为乘法的优先级也大于加法，所以随后中间结果（5）将乘以2（即b的值），并获得新的中间结果（10）。

3. 最后，计算6加10，并得出最终结果（16）。

第二条NSLog语句引入了一个新误区。你希望a除以b再乘以b的操作返回a（已经设置为25）。但此操作并不会产生这一结果，在输出显示器上显示的是24。难道计算机在某个地方迷失了方向？如果这样就太不幸了。其实该问题的实际情况是：这个表达式是采用整数运算来求值的。

如果回头看一下变量a和b的声明，你会想起它们都是用int类型声明的。当包含两个整数的表达式求值时，Objective-C系统都将使用整数运算来执行这个操作。在这种情况下，数字的所有小数部分将丢失。因此，计算a除以b，即25除以2时，得到的中间结果是12而不是期望的12.5。这个中间结果乘以2就得到最终结果24，这样，就解释了出现“丢失”数字的情况。

在代码清单4-3的倒数第二个NSLog语句中看到，如果用浮点值代替整数来执行同样的运算，就会获得期望的结果。

决定使用float变量还是int变量应该基于变量的使用目的。如果无需使用任何小数位，可使用整型变量。这将使程序更加高效，换言之，它可以在大多数计算机上更加快速地执行。另一方面，如果需要精确到小数位，很清楚应该选择什么。此时，唯一必须回答的问题是使用float还是double。对此问题的回答取决于使用数据所需的精度以及它们的量级。

在最后一条NSLog语句中，使用了一元负号运算符对变量a的值求反。这个一元运算符是用于单个值的运算符，而二元运算符作用于两个值。负号实际上扮演了一个双重角色：作为二元运算符，它执行两个数相减的操作；作为一元运算符，它对一个值求反。

与其他算术运算符相比，一元负号运算符具有更高的优先级，但一元正号运算符（+）除外，它和算术运算符的优先级相同。因此，表达式

```
c = -a * b;
```

将执行-a乘以b。再次参见附录B，其中有一个包括各种运算符及其优先级的总结表。

4.2.3 模运算符

本章介绍的最后一个运算符是模运算符，它由百分号（%）表示。通过分析代码清单4-4的输出，尝试确定这种运算符的工作方式。

代码清单4-4

```
// The modulus operator

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int a = 25, b = 5, c = 10, d = 7;

   NSLog (@"a %% b = %i", a % b);
   NSLog (@"a %% c = %i", a % c);
   NSLog (@"a %% d = %i", a % d);
   NSLog (@"a / d * d + a %% d = %i", a / d * d + a % d);
```

```
    [pool drain];
    return 0;
}
```

代码清单4-4 输出

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

注意，main中的语句定义并初始化了变量a、b、c和d，这些工作均在一条语句内完成。

你已经知道，NSLog使用百分号之后的字符来确定如何输出下一个参数。然而，如果它后面紧跟另一个百分号，那么NSLog例程认为你其实想显示百分号，并在程序输出的适当位置插入一个百分号。

如果总结出用模运算符%的功能是计算第一个值除以第二个值所得的余数，就是正确的。在第一个例子中，25除以5所得的余数，显示为0。如果用25除以10，会得到余数5，输出中的第二行可以证实。执行25乘以7将得到余数4，它显示在输出的第三行。

现在，我们把注意力转移到最后一条语句求值的表达式上。你将回忆起Objective-C使用整数运算来执行两个整数间的任何运算。因此，两个整数相除所产生的任何余数将被完全丢弃。如果使用表达式a/b表示25除以7，将会得到中间结果3。将这个结果乘以d的值（即7），将会产生中间结果21。最后，加上a除以b的余数，该余数由表达式a % d来表示，会产生最终结果25。这个值与变量a的值相同并非巧合。一般来说，表达式

```
a / b * b + a % b
```

的值将始终与a的值相等，当然，这是在假定a和b都是整型值的条件下做出的。事实上，定义的模运算符%只用于处理整数。

就优先级而言，模运算符的优先级与乘法和除法的优先级相等。毫无疑问，这意味着表达式

```
table + value % TABLE_SIZE
```

等价于

```
table + (value % TABLE_SIZE)
```

4.2.4 整型值和浮点值的相互转换

要更有效地开发Objective-C程序，必须理解Objective-C中浮点值和整型值之间进行隐式转换的规则。代码清单4-5表明数值数据类型间的一些简单转换。

代码清单4-5

```
// Basic conversions in Objective-C

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

```
int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   float  f1 = 123.125, f2;
   int    i1, i2 = -150;

   i1 = f1;   // floating to integer conversion
   NSLog (@"%f assigned to an int produces %i", f1, i1);

   f1 = i2;   // integer to floating conversion
   NSLog (@"%i assigned to a float produces %f", i2, f1);

   f1 = i2 / 100;   // integer divided by integer
   NSLog (@"%i divided by 100 produces %f", i2, f1);

   f2 = i2 / 100.0;   // integer divided by a float
   NSLog (@"%i divided by 100.0 produces %f", i2, f2);

   f2 = (float) i2 / 100;   // type cast operator
   NSLog (@"(float) %i divided by 100 produces %f", i2, f2);

   [pool drain];
   return 0;
}
```

代码清单4-5　输出

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

在Objective-C中，只要将浮点值赋值给整型变量，数字的小数部分都会被删节。因此，在前一个程序中，把f1的值指派给i1时，数字123.125将被删节，这意味着只有整数部分（即123）存储到了i1中。程序输出的第一行验证了上述程序就是此种情况。

把整型变量指派给浮点变量的操作不会引起数字值的任何改变；该值仅由系统转换并存储到浮点变量中。程序输出的第二行验证了这一情况：i2的值（-150）进行了正确转换并存储到float变量f1中。

程序输出的后两行说明了在编写算术表达式时要记住的两点。第一点和整数运算有关，在前一章已经讨论了这一点。只要表达式中的两个运算数是整型（而且这一情况还适用于short、unsigned和long所修饰的整型），该运算就将在整数运算的规则下进行。因此，由乘法运算产生的任何小数部分都将删除，即使该结果指派给一个浮点变量（如同我们在程序中所做的那样）也是如此。当整型变量i2除以整数常量100时，系统将该除法作为整数除法来执行。因此，-150除以100的结果，即-1，将存储到float变量f1中。

在前一个程序中，后一个除法涉及一个整数变量和一个浮点常量。在Objective-C中，任何处理两个值的运算如果其中一个值是浮点变量或常量，那么这一运算将作为浮点运算来处理。因此，当i2的值除以100.0时，系统将除法作为浮点除法来计算，并产生结果1.5，该结果将指派给float变量f1。

4.2.5 类型转换运算符

你已经看到如何在声明和定义方法时将类型放入圆括号中来声明返回值和参数的类型。在表达式中使用类型时，它表示一个特殊的用途。

代码清单4-5中的最后一个除法运算

```
f2 = (float) i2 / 100;      // type cast operator
```

引入了类型转换运算符。

为了求表达式值，类型转换运算符将变量i2的值转换成float类型。该运算符永远不会影响变量i2的值；它是一元运算符，行为和其他一元运算符一样。因为表达式-a永远不会影响a的值，因此表达式（float）a也不会影响a的值。

类型转换运算符比所有算术运算符的优先级都高，但一元减号和一元加号运算符除外。当然，如果需要，可经常使用圆括号进行限制，以任何想要的顺序来执行一些项。

下面是使用类型转换运算符的另一个例子，表达式

```
(int) 29.55 + (int) 21.99
```

在Objective-C中等价于

```
29 + 21
```

因为将浮点值转换成整数的后果就是舍弃其中的浮点值。表达式

```
(float) 6 / (float) 4
```

得到的结果为1.5，与下列表达式的执行效果相同：

```
(float)6 / 4
```

类型转换运算符通常用于将一般id类型的对象转换成特定类的对象。例如，

```
id   myNumber;
Fraction *myFraction;
  ...
myFraction = (Fraction *) myNumber;
```

将id变量myNumber的值转换成一个Fraction对象。转换结果将指派给Fraction变量myFraction。

4.3 赋值运算符

Objective-C语言允许使用以下一般格式将算术运算符和赋值运算符合并到一起：

```
op=
```

在这个格式中，op是任何算术运算符，包括+、-、*、/和%。此外，op还可以是任何用于

移位和屏蔽操作的位运算符，这些内容将在以后讨论。

考虑下面这条语句：

```
count += 10;
```

通常所说的“加号等号”运算符（+=），将运算符右侧的表达式和左侧的表达式相加，再将结果保存到运算符左边的变量中。因此，上面的语句和以下语句等价：

```
count = count + 10;
```

表达式

```
counter -= 5
```

使用“减号等号”赋值运算符将counter的值减5，它和下面这个语句等价：

```
counter = counter - 5
```

下面是一个稍微复杂一些的表达式

```
a /= b + c
```

无论等号右侧出现何值（或者b加c的和），都将用它除以a，再把结果存储到a中。因为加法运算符比赋值运算符的优先级高，所以表达式会首先执行加法。事实上，除逗号运算符外的所有运算符都比赋值运算符的优先级高，而所有赋值运算符的优先级相同。

在这个例子中，该表达式的作用和下列表达式相同：

```
a = a / (b + c)
```

使用赋值运算符的动机有3个：首先，程序语句更容易书写，因为运算符左侧的部分没有必要在右侧重写。其次，结果表达式通常容易阅读。再次，这些运算符的使用可使程序的运行速度更快，因为编译器有时在计算表达式时能够产生更少的代码。

4.4 计算器类

现在定义一个新类。我们将创建一个Calculator类，它是一个简单的四则计算器，可用来执行加、减、乘和除运算。类似于常见的计算器，这种计算器必须能够记录累加结果，或者通常所说的累加器。因此，方法必须能够执行以下操作：将累加器设置为特定值、将其清空（或设置为0），以及在完成时检索它的值。代码清单4-6包括这个新类的定义和一个用于试验该计算器的测试程序。

代码清单4-6

```
// Implement a Calculator class

#import <Foundation/Foundation.h>

@interface Calculator: NSObject
{
    double accumulator;
}
```

```
// accumulator methods
-(void)   setAccumulator: (double) value;
-(void)   clear;
-(double) accumulator;

// arithmetic methods
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
-(void) setAccumulator: (double) value
{
    accumulator = value;
}

-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
```

```
@end

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Calculator *deskCalc;

    deskCalc = [[Calculator alloc] init];

    [deskCalc clear];
    [deskCalc setAccumulator: 100.0];
    [deskCalc add: 200.];
    [deskCalc divide: 15.0];
    [deskCalc subtract: 10.0];
    [deskCalc multiply: 5];
    NSLog (@"The result is %g", [deskCalc accumulator]);
    [deskCalc release];

    [pool drain];
    return 0;
}
```

代码清单4-6　输出

```
The result is 50
```

Calcauator类只有一个实例变量，以及一个用于保存累加器值的double变量。方法定义的本身非常直观。

要注意调用multiply方法的消息：

```
[deskCalc multiply: 5];
```

该方法的参数是一个整数，而它期望的参数类型却是double。因为方法的数值参数会自动转换以匹配期望的类型，所以此处不会出现任何问题。multiply:期望使用double值，因此调用该函数时，整数5将自动转换成双精度浮点值。即使自动转换过程会自己进行，然而在调用方法时提供正确的参数类型仍是一个较好的程序设计习惯。

要认识到与Fraction类不同，Fraction类可能使用多个不同的分数，在这个程序中可能希望只处理单个Calculator对象。然而，定义一个新类以便更容易地处理这个对象仍是有意义的。从某些方面讲，可能要为计算器添加一个图形前端，以便用户能够在屏幕上实际点击按钮，与系统或电话中已安装的计算器应用程序一样。

在以后的若干练习中，可以看到定义Calculator类的另一个好处和便于扩展有关。

4.5　位运算符

Objective-C语言中有各种各样的运算符可处理数字中的特定位。表4-2列出这些运算符。

表4-2 位运算符

符 号	运 算	符 号	运 算
&	按位与	~	一次求反
|	按位或	<<	向左移位
^	按位异或	>>	向右移位

表4-2列出的所有运算符，除一次求反运算符（~）外，都是二元运算符，因此需要两个运算数。位运算符可处理任何类型的整型值，但不能处理浮点值。

4.5.1 按位与运算符

对两个值执行与运算时，会逐位比较两个值的二进制表示。第一个值与第二个值对应位都为1时，在结果的对应位上就会得到1；其他的组合在结果中都得到0。如果b1和b2表示两个运算数的对应位，那么下表（称为真值表）就显示了在b1和b2所有可能值下对b1和b2执行与操作的结果。

```
b1      b2      b1 & b2
----------------------
0       0       0
0       1       0
1       0       0
1       1       1
```

例如，如果w1和w2都定义为short int，w1等于十六进制的15，w2等于十六进制的0c，那么以下C语句会将值0x04指派给w3：

```
w3 = w1 & w2;
```

将w1、w2和w3都表示为二进制后可更清楚地看到此过程。假设所处理的short int大小为16位：

```
w1    0000 0000 0001 0101     0x15
w2    0000 0000 0000 1100   & 0x0c
---------------------------------
w3    0000 0000 0000 0100     0x04
```

按位与运算经常用于屏蔽运算。就是说，这个运算符可轻易地将数据项的特定位设置为0。例如，语句

```
w3 = w1 & 3;
```

将w1与常量3按位与所得的值指派给w3。它的作用是将w3中的全部位（而非最右边的两位）设置为0，并保留w1中最左边的两位。

与Objective-C中使用的所有二元运算符相同，通过添加等号，二元位运算符可同样用作赋值运算符。因此语句

```
word &= 15;
```

与下列语句

```
word = word & 15;
```

执行相同的功能。

此外，它还能将word的全部位设置为0，但最右边的四位除外。

4.5.2 按位或运算符

在Objective-C中对两个值执行按位或运算时，会逐位比较两个值的二进制表示。此时，只要第一个值或者第二个值的相应位是1，那么结果的对应位就是1。按位或操作符的真值表如下所示。

```
b1      b2     b1 | b2
-----------------------
0       0        0
0       1        1
1       0        1
1       1        1
```

所以，如果w1是short int，等于十六进制的19，w2也是short int，等于十六进制的6a，那么对w1和w2执行按位或会得到十六进制的7b，如下所示：

```
w1    0000 0000 0001  1001        0x19
w2    0000 0000 0110  1010    |   0x6a
-------------------------------------
      0000 0000 0111  1011        0x7b
```

按位或操作通常就称为按位OR，用于将某个词的特定位设为1。例如，以下语句将w1最右边的三位设为1，而不管这些位操作前的状态是什么都是如此。

```
w1 = w1 | 07;
```

当然，可以在语句中使用特殊的赋值运算符，如下面的语句所示：

```
w1 |= 07;
```

我们在后面会提供一个程序例子，演示如何使用按位或运算符。

4.5.3 按位异或运算符

按位异或运算符，通常称为XOR运算符，遵守以下规则：对于两个运算数的相应位，如果任何一个位是1，但不是两者全为1，那么结果的对应位将是1，否则是0。该运算符的真值表如下所示。

```
b1      b2     b1 ^ b2
-----------------------
0       0        0
0       1        1
1       0        1
1       1        0
```

如果w1和w2分别等于十六进制的5e和d6，那么w1与w2执行异或运算后的结果是十六进制

值e8，如下所示：

```
w1      0000 0000 0101  1110      0x5e
w2      0000 0000 1011  0110    ^ 0xd6
-----------------------------------
        0000 0000 1110  1000      0xe8
```

4.5.4 一次求反运算符

一次求反运算符是一元运算符，它的作用仅是对运算数的位“翻转”。将运算数的每个是1的位翻转为0，而将每个是0的位翻转为1。此处提供真值表只是为了保持内容的完整性。

```
b1    ~b1
--------
0     1
1     0
```

如果w1是short int，16位长，等于十六进制值a52f，那么对该值执行一次求反运算会得到十六进制值5ab0：

```
 w1    1010  0101  0010  1111      0xa52f
~w1    0101  1010  1101  0000      0x5ad0
```

如果不知道运算中数值的准确位大小，那么一次求反运算符非常有用，使用它可让程序不会依赖于整数数据类型的特定大小。例如，要将类型为int的w1的最低位设为0，可将一个所有位都是1、但最右边的位是0的int值与w1进行与运算。所以像下面这样的C语句在用32位表示整数的机器上可正常工作。

```
w1 &= 0xFFFFFFFE;
```

如果用

```
w1 &= ~1;
```

替换上面的语句，那么在任何机器上w1都会同正确的值进行与运算。

这是因为这条语句会对1求反，然后在左侧会加入足够的1，以满足int的大小要求（在32位机器上，会在左侧的31个位上加入1）。

现在，显示一个实际的程序例子，说明各种位运算符的用途（参见代码清单4-7）。

代码清单4-7

```
// Bitwise operators illustrated

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    unsigned int w1 = 0xA0A0A0A0, w2 = 0xFFFF0000,
```

```
        w3 = 0x00007777;

    NSLog (@"%x %x %x", w1 & w2, w1 | w2, w1 ^ w2);
    NSLog (@"%x %x %x", ~w1, ~w2, ~w3);
    NSLog (@"%x %x %x", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    NSLog (@"%x %x", w1 | w2 & w3, w1 | w2 & ~w3);
    NSLog (@"%x %x", ~(~w1 & ~w2), ~(~w1 | ~w2));

    [pool drain];
    return 0;
}
```

代码清单4-7　输出

```
a0a00000 ffffa0a0 5f5fa0a0
5f5f5f5f ffff ffff8888
0 a0a0 fffff7f7
a0a0a0a0 ffffa0a0
ffffa0a0 a0a00000
```

对代码清单4-7中的每个运算都演算一遍，确定你理解了这些结果是如何得到的。

在第四个NSLog调用中，需要注意重要的一点，即按位与运算符的优先级要高于按位或运算符，因为这会实际影响表达式的最终结果值。有关运算符优先级的内容，请参考附录B。

第五个NSLog调用展示了DeMorgan的规则：~(~a & ~b)等于a | b，~(~a | ~b)等于a & b。

4.5.5　向左移位运算符

对值执行向左移位运算时，按照字面的意思，值中包含的位将向左移动。与该操作关联的是该值要移动的位置（或位）数目。超出数据项的高位的位将丢失，而从低位移入的值总为0。因此，如果w1等于3，那么表达式

```
w1 = w1 << 1;
```

可同样表示成

```
w1 <<= 1;
```

结果就是3向左移一位，这样产生的6将赋值给w1。

```
w1        ... 0000 0011    0x03
w1 << 1   ... 0000 0110    0x06
```

<< 运算符左侧的运算数表示将要移动的值，而右侧的运算数表示该值所需移动的位数。如果将w1再向左移动一次，那么会得到十六进制值0c：

```
w1        ... 0000 0110     0x06
w1 << 1   ... 0000 1100    0x0c
```

4.5.6 向右移位运算符

顾名思义，向右移位运算符（>>）把值的位向右移动。从值的低位移出的位将丢失。把无符号的值向右移位总是左侧（就是高位）移入0。对于有符号值而言，左侧移入1还是0依赖于被移动数字的符号，还取决于该操作在计算机上的实现方式。如果符号位是0（表示该值是正的），不管哪种机器都将移入0。然而，如果符号位是1，那么在一些计算机上将移入1，而其他计算机上则移入0。前一类型的运算符通常称为算术右移，而后者通常称为逻辑右移。

警告 对于系统使用算术右移还是逻辑右移，千万不要进行猜测。如果进行此类的假设，那么在一个系统上可正确进行有符号右移运算的程序，有可能在其他系统上运行失败。

如果w1是unsigned int，用32位表示它并且它等于十六进制的F777EE22，那么使用语句

```
w1 >>= 1;
```

将w1右移一位后，w1等于十六进制的7BBBF711，如下所示：

```
w1        1111 0111 0111 0111 1110 1110 0010 0010    0xF777EE22
w1 >> 1 0111 1011 1011 1011 1111 0111 0001 0001    0x7BBBF711
```

如果将w1声明为（有符号）的short int，在某些计算机上会得到相同的结果；而在其他计算机上，如果将该运算作为算术右移来执行，结果将会是FBBBF711。

应该注意到，如果试图用大于或等于该数据项的位数将值向左或向右移位，那么该Objective-C语言并不会产生规定的结果。因此，例如计算机用32位表示整数，那么把一个整数向左或向右移动32位或更多位时，并不会在计算机上产生规定的结果。还注意到，如果使用负数对值移位时，结果将同样是未定义的。

4.6 类型：_Bool、_Complex和_Imaginary

在本章结束之前，还将探讨这门语言中的其他3种类型：用于处理Boolean（即，0或1）值的_Bool；以及分别用于处理复数和抽象数字的_Complex和_Imaginary。

Objective-C程序员倾向于在程序中使用BOOL数据类型替代_Bool来处理Boolean值。这种“数据类型”本身实际上并不是真正的数据类型，它事实上只是char数据类型的别名。这是通过使用该语言的特殊关键字typedef实现的，而typedef将在第10章“变量和数据类型”中讲解。

4.7 练习

1. 下列常量中，哪些是非法的？为什么？

```
123.456     0x10.5     0X0G1
0001        0xFFFF     123L
0Xab05      0L         -597.25
123.5e2     .0001      +12
98.6F       98.7U      17777s
0996        -12E-12    07777
1234uL      1.2Fe-7    15,000
```

```
1.234L      197u        100U
0XABCDEFL   0xabcu      +123
```

2. 编写一个程序，使用以下公式将华氏温度（F）27o转换成摄氏温度（C）：

```
C = (F - 32) / 1.8
```

3. 以下程序将输出什么结果?

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

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   char c, d;

   c = 'd';
   d = c;
   NSLog (@"d = %c", d);

   [pool drain];
   return 0;
}
```

4. 编写一个程序，求以下多项式的值：

$3x^3 - 5x^2 + 6$

已知x = 2.55

5. 编写一个程序，求下列表达式的值并显示其结果（记住要使用指数格式显示结果）：

$$(3.31\times10^{-6}x+2.01\times10^{-7}) \;/\; (7.16\times10^{-6}+2.01\times10^{-8})$$

6. 复数包含两个部分：实部和虚部。如果a是实部，b是虚部，那么符号

```
a + b i
```

可用来表示复数。

编写一个Objective-C程序，定义一个名为Complex的新类。依照为Fraction类创建的范例，为该类定义以下方法：

```
-(void) setReal: (double) a;
-(void) setImaginary: (double) b;
-(void) print;  // display as a - bi
-(double) real;
-(double) imaginary;
```

编写一个测试程序测试这个新类和各个方法。

7. 假设你正开发操作图形对象的例程库。从定义名为Rectangle的新类开始。目前，仅记录矩形的宽和高即可。开发一些方法用于设置矩形的宽和高、检索这些值以及计算矩形的面积和周长。假定这些矩形对象使用整数坐标格栅来描述矩形，例如一台计算机屏幕。在这种情况下，假定矩形的宽和高都是整数值。

以下是Rectangle类的@interface部分：

```
@interface Rectangle: NSObject
{
    int  width;
    int  height;
}
-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(int) width;
-(int) height;
-(int) area;
-(int) perimeter;
@end
```

编写implementation部分，并编写一个测试程序来测试新类的方法。

8. 修改代码清单4-6中的add:、subtract:、muntiply:和divide:方法，使其返回累加器的结果值。测试这些新方法。
9. 完成练习8后，把以下方法添加到Calculator类中并测试它们：

```
-(double) changeSign; // change sign of accumulator
-(double) reciprocal; // 1/accumulator
-(double) xSquared;   // accumulator squared
```

10. 为代码清单4-6中的Calculator添加一项存储功能。实现以下方法声明并测试它们：

```
-(double) memoryClear;    // clear memory
-(double) memoryStore;    // set memory to accumulator
-(double) memoryRecall;   // set accumulator to memory
-(double) memoryAdd;      // add accumulator to memory
-(double) memorySubtract; // subtract accumulator from memory
```

第5章 循环结构

在Objective-C中，有若干方法可以用于重复执行一系列代码。本章的主题是这些循环功能，它们由以下几部分组成：

- for语句
- while语句
- do语句

从一个简单的例子——计数开始讨论。

如果要把15个大理石弹子排列成一个三角形，排列后的弹子可能如图5-1所示：

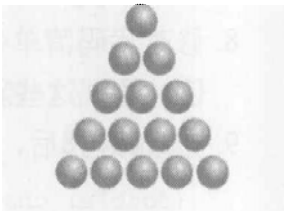

图5-1 三角形排列示例

三角形的第一行包含一个弹子，第二行包含两个弹子，依此类推。一般来说，包含n行的三角形可容纳的弹子总数等于1到n之间所有整数之和。这个和称为三角数（triangular number）。

如果从1开始，第4位三角数将等于1到4之间连续整数的和（1 + 2 + 3 + 4），即10。

假设要编写一个程序来计算第8位三角数的值，并在终端上显示。显然可以在头脑中计算这个值，但是为了学习参数，假设你用Objective-C编写一个带有参数的程序来执行这个任务。代码清单5-1显示了这个程序。

代码清单5-1

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

// Program to calculate the eighth triangular number

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int triangularNumber;

   triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

   NSLog (@"The eighth triangular number is %i", triangularNumber);

   [pool drain];
   return 0;
}
```

代码清单5-1 输出

```
The eighth triangular number is 36
```

如果计算相对较小的三角数，代码清单5-1中的方法工作良好，但是要找出第200位三角数的值，该程序将会如何处理呢？必须修改代码清单5-1，以便显式地将1到200之间的所有整数相加，这项工作肯定是冗长乏味的。值得庆幸的是，有一个比较简单的方法。

计算机的基本属性之一就是它能够重复执行一组语句。这种循环能力可使程序员开发出包含重复过程的简洁程序，这些过程能够以不同的方式执行成百上千的程序语句。Objective-C包含3种用于编写结构循环的程序语句。

5.1 for语句

看看使用for语句的程序。代码清单5-2的目的是计算第200位三角数。看看你是否可以找出for语句的工作方式。

代码清单5-2

```
// Program to calculate the 200th triangular number
// Introduction of the for statement

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int n, triangularNumber;

   triangularNumber = 0;

   for ( n = 1; n <= 200; n = n + 1 )
   triangularNumber += n;

   NSLog (@"The 200th triangular number is %i", triangularNumber);

   [pool drain];
   return 0;
}
```

代码清单5-2 输出

```
The 200th triangular number is 20100
```

需要对代码清单5-3进行一些解释。用于计算第200位三角数的方法其实与上一个程序中用于计算第8位三角数的方法是相同的，就是求1到200之间的整数之和。

在执行for语句之前，变量triangularNumber被设置为0。一般来说，在程序使用变量之前，需要将所有的变量初始化为某个值（和处理对象一样）。后面将会学到，某些类型的变量将给定默认的初始值，但是无论如何都应该为变量设置初始值。

for语句提供的机制可使你避免显式地写出1到200之间的每个整数。从某种意义上讲，这条

语句将为你生成这些数字。

for语句的一般格式如下：

```
for ( init_expression; loop_condition; loop_expression )
    program statement
```

圆括号中的3个表达式init_expression、loop_condition和loop_expression建立了程序循环的“环境”。其后的program statement（当然是以一个分号结束）可以是任何合法的Objective-C程序语句，它们组成循环体。这条语句执行的次数由for语句中设置的参数决定。

for语句的第一部分标着init_expression，它用于在循环开始之前设置初始值。在代码清单5-2中，for语句这个部分将n的初始值设置为1。可以看到，赋值是一种合法的表达式形式。

for语句的第二部分用于指定继续执行循环所需的条件。换言之，只要满足这个条件，循环就将继续执行。再次参见代码清单5-2，for中的loop_condition是由以下关系表达式指定的：

```
n <= 200
```

这个表达式可读做“n小于或等于200”。“小于或等于”运算符（由等号[=]和紧跟在其后的小于号[<]组成）仅是Objective-C程序设计语言提供的若干关系运算符中的一个。这些关系运算符用于测试特定的条件。如果满足条件，测试结果为真（或TRUE）；如果不满足条件，测试结果为假（或FALSE）。

表5-1列出了Objective-C中可用的所有关系运算符。

表5-1　关系运算符

运算符	含　义	例　子
==	等于	count == 10
!=	不等于	flag != DONE
<	小于	a < b
<=	小于或等于	low <= high
>	大于	points > POINT_MAX
>=	大于或等于	j >= 0

关系运算符的优先级比所有的算术运算符的优先级都低。这意味着，例如以下表达式

```
a < b + c
```

将按

```
a < ( b + c )
```

来求值。

这与你期望的相同。如果a的值小于b + c的值，表达式将为TRUE，否则表达式为FALSE。

要特别注意等于运算符（==），不要将其与赋值运算符（=）混淆。表达式

```
a == 2
```

用于测试a的值是否等于2，而表达式

```
a = 2
```

用于将值2赋值给变量a。

选择要使用哪个关系运算符由所做的具体测试决定，有的情况下由你的具体偏好决定。例如，关系表达式

```
n <= 200
```

可以等价地表示为

```
n < 201
```

回到前一个例子中，只要关系测试结果为TRUE，在这个例子中n的值小于或等于200时，形成for循环体的程序语句——triangularNumber +=;将被重复执行。这条语句的作用是将n的值和triangularNumber的值加到一起。

不再满足loop_condition时，程序在for循环之后的程序语句继续执行。在该程序中，该循环终止之后将继续执行NSLog语句。

for语句的最后一部分包含一个表达式，它在每次执行循环体之后求值。在代码清单5-2中，loop_expression的作用是将n的值加1。因此，每次把n的值加到triangularNumber之后，它的值都要加1，而且该值将从1一直增加到201。

值得注意的是，n的最终值（即201），将不会加到triangularNumber的值上，因为只要不再满足循环条件，或只要n等于201，循环就会终止。

总之，for语句将按以下步骤执行：

1. 先求初始表达式的值。这个表达式通常设置一个将在循环中使用的变量，对于某些初始值（例如0或1）来说，通常称作索引变量。

2. 求循环条件的值。如果条件不满足（即表达式为FALSE），循环就立即终止。然后执行在循环之后的程序语句。

3. 执行组成循环体的程序语句。

4. 求循环表达式的值。这个表达式通常用于改变索引变量的值，最常见的情况是，将索引变量的值加1或减1。

5. 返回到步骤2。

记住，循环条件要在进入循环时在第一次执行循环体之前立即求值。还要记住，不要在循环末尾处的结束圆括号后面放置分号，这将导致循环立即终止。

代码清单5-2在计算最终结果的过程中，实际生成了所有前200个三角数，所以，生成这些数字的一个表格是不错的主意。然而，为了节省空间，假定你只想打印一张包含前10个三角数的表。代码清单5-3执行这个任务。

代码清单5-3

```
// Program to generate a table of triangular numbers

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
```

```
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int n, triangularNumber;

    NSLog (@"TABLE OF TRIANGULAR NUMBERS");
    NSLog (@" n  Sum from 1 to n");
    NSLog (@"--  ---------");

    triangularNumber = 0;

    for ( n = 1; n <= 10; ++n ) {
        triangularNumber += n;
        NSLog (@" %i          %i", n, triangularNumber);
    }

    [pool drain];
    return 0;
}
```

代码清单5-3　输出

```
TABLE OF TRIANGULAR NUMBERS

n Sum from 1 to n
--- ---------------
1       1
2       3
3       6
4       10
5       15
6       21
7       28
8       36
9       45
10      55
```

在代码清单5-3中，前3个NSLog语句的目的仅仅是提供一个普通标题并标记输出列。

在显示适当的标题后，程序将计算前10个三角数。变量n用于记录当前的数字，你正在计算1到n的和，而变量triangularNumber用于存储第n个三角数的值。

for语句的执行首先是将变量n的值设置为1。前面提到过，在for语句之后的程序语句构成了程序循环的主体。但是如果不只想执行单个程序语句，而且想执行一组语句该怎么办呢？通过把这样的程序语句放入一对花括号中可达到这个目的。系统会把这组（或块）语句看作单个实体。一般来说，在Objective-C程序中能使用单个语句的任何位置均能使用语句块，不过要记住，语句块必须放在一对花括号中才能使用。

因此在代码清单5-3中，要把n加到triangularNumber值上的表达式和在程序循环构成体之后的NSLog语句放到一对花括号中。特别注意程序语句的缩进方式。快速扫视一下，可以轻易地

确定哪些语句构成了for循环。还应该注意程序员采用不同的编码风格：一些人更喜欢用以下方式输入循环：

```
for ( n = 1; n <= 10; ++n )
{
  triangularNumber += n;
  NSLog (@" %i %i", n, triangularNumber);
}
```

其中开始的花括号位于for的下一行。严格来说，这只是一个爱好问题，并不会影响程序。

通过简单地将n的值加到前一个三角数，可计算出下一个三角数。第一次遍历for循环时，上一个三角数为0，因此n等于1时triangularNumber的新值就是n的值，即1。然后显示n的值和triangularNumber，并带有适当数目的空格，这些空格将插入到格式字符串中，以确保这两个变量的值可以排列到相应的列标题之下。

因为现在执行的是循环体，所以随后将求循环表达式的值。然而，这条for语句中的表达式看上去有些奇怪。当然，这肯定是一个印刷错误，它意味着插入n = n + 1来替代这个看上去相当奇怪的表达式：

```
++n
```

事实是：++n其实是相当合法的Objective-C表达式。它引入了Objective-C程序设计语言中的一个新（而且相当独特）运算符——自增运算符。双加号（或叫做自增运算符）的作用是将其运算数加1。加1运算在程序设计中很常见，以至于创造了一个特殊符号专门完成这项任务。因此，表达式++n等价于表达式n = n + 1。乍一看时，可能觉得n = n + 1更易阅读，但是很快你就会习惯这种运算符，甚至更喜欢它的简洁性。

当然，没有哪种语言只提供自增运算符执行加1的操作，而不提供相应的运算符执行减1操作。正如你所猜测的，这种运算符叫做自减运算符，它由双减号来表示。因此，用Objective-C书写的表达式

```
bean_counter = bean_counter - 1
```

可用自减运算符等价地表示成以下形式：

```
--bean_counter
```

一些程序员喜欢将++或--放到变量名后面，如n++或bean_counter--。这种情况是可以接受的，只不过是个人喜好问题。

你可能已经注意到，代码清单5-3输出的最后一行没有对齐。使用以下NSLog语句来替代代码清单5-3中对应的语句，可改正这个小毛病。

```
NSLog ("%2i %i", n, triangularNumber);
```

要验证这个改变解决了上述问题，下面给出修改后的程序输出（名为代码清单5-3A）。

代码清单5-3A 输出

```
TABLE OF TRIANGULAR NUMBERS
```

```
 n  Sum from 1 to n
--- ---------------
 1          1
 2          3
 3          6
 4          10
 5          15
 6          21
 7          28
 8          36
 9          45
10          55
```

NSLog语句所做的主要改动是它包含了字段宽度说明。字符%2i告知NSLog例程：不仅在特定点显示整数值，而且要展示的整数应该占用显示器的两列。通常占用空间少于两列的任何整数（即，0到9之间的整数）在显示时都带有一个前导空格。这种情况称为向右对齐。

因而，通过使用字符宽度说明%2i，可以确保至少有两列将用于显示n的值，还能保证对齐triangularNumber的值。

5.1.1 键盘输入

代码清单5-2可计算出第200个三角数，但不能计算更多的数。如果要计算第50个或第100个三角数，该怎么办呢？好吧，如果是这种情况，就不得不更改程序，以便for循环可以执行合适的次数。还必须更改NSLog语句来显示正确的消息。

更加简单的解决方案可能是，在一定程度上允许程序向你询问要计算哪个三角数。然后，给出回答后，程序就可以计算出期望的三角数。使用一个名为scanf的例程，可以实现这样的解决方案。在概念上，scanf例程与NSLog例程类似。但是NSLog例程用于显示值，而scanf例程的用途是程序员可以把值输入到程序中。当然，如果使用图形用户界面（UI）编写Objective-C程序，如Cocoa或iPhone应用程序，那么在程序中可能根本不用NSLog或scanf。

代码清单5-4先询问用户要计算哪个三角数，然后计算该数并显示结果。

代码清单5-4

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int n, number, triangularNumber;

    NSLog (@"What triangular number do you want?");
    scanf ("%i", &number);

    triangularNumber = 0;
```

```
    for ( n = 1; n <= number; ++n )
        triangularNumber += n;

    NSLog (@"Triangular number %i is %i\n", number, triangularNumber);

    [pool drain];
    return 0;
}
```

在后面的程序输出中，用户键入的数字（100）用黑体显示，以便和由程序显示的输出相区别。

代码清单5-4 输出

```
What triangular number do you want?
100
Triangular number 100 is 5050
```

依照输出，可以看出数字100是由用户键入的。然后该程序计算第100个三角数并将结果5050显示在终端上。如果用户想要计算一个特定的三角数，可以键入10或者30这些数字。

在代码清单5-4中，第一个NSLog语句用于提示用户键入数字。当然，向用户提示输入内容总是好的。输出消息后，调用scanf例程。scanf的第一个参数是格式字符串，它不以@字符开头。NSLog的第一个参数始终是NSString，而scanf的第一个参数是C风格的字符串。在前面已经提及过，C风格的字符串前面不用加字符@。

格式字符串告知scanf要从控制台（或者是Terminal窗口，如果正在使用Terminal应用程序编译程序的话）读入的值类型。和NSLog一样，%i字符用于指定整型值。

scanf例程的第二个参数用于指定将用户键入的值存储在哪里。在这种情况下，变量number之前的&字符是必需的。但是不要担心它在此处的功能。在第13章“基本的C语言特性”中谈论指针时，我们将详细讨论这个字符，它实际上是一个运算符。

根据前面的讨论，可以看到代码清单5-4中的scanf调用指定要输入整型值并将其存储到变量number中。这个值代表用户希望计算哪个三角数。

键入这个数字之后（按下键盘上的Enter键，表示该数字的键入工作已完成），程序便计算指定的三角数。实现方式和代码清单5-2中的一样，唯一的不同是，此处没有用200作为界限，而是用number作为界限。

计算出期望的三角数之后，显示结果，然后程序的执行结束。

5.1.2 嵌套的for循环

代码清单5-4向用户提供了以下灵活性：使程序计算出任何想要的三角数。但是假设用户要计算5个三角数的列表，该怎么办呢？这种情况下，用户可简单地将程序执行5次，每次键入要计算的列表中下一个三角数即可。

实现相同目的有另一种方式，并且就学习Objective-C而言，它更为有趣，就是让程序处理

这种情况。通过向程序插入循环，将整个计算过程重复执行5次，可最好地完成这一任务。可以使用for语句来建立这样的循环。代码清单5-5及其输出说明了这一技术。

代码清单5-5

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

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int n, number, triangularNumber, counter;

   for ( counter = 1; counter <= 5; ++counter ) {
        NSLog (@"What triangular number do you want?");
        scanf ("%i", &number);

        triangularNumber = 0;

        for ( n = 1; n <= number; ++n )
             triangularNumber += n;

        NSLog (@"Triangular number %i is %i", number, triangularNumber);

    }

   [pool drain];
   return 0;
}
```

代码清单5-5 输出

```
What triangular number do you want?
12
Triangular number 12 is 78

What triangular number do you want?
25
Triangular number 25 is 325

What triangular number do you want?
50
Triangular number 50 is 1275

What triangular number do you want?
75
Triangular number 75 is 2850

What triangular number do you want?
```

```
83
Triangular number 83 is 3486
```

该程序包含两层for循环语句。最外层的for循环语句如下：

```
for ( counter = 1; counter <= 5; ++counter )
```

这条语句指定该程序循环正好执行5次。因为counter的值最初设为1，并且依次加1，直到它的值不再小于或等于5（换句话说，直到它到达6）为止，所以可以看出执行5次。

与上一个程序例子不同，程序的其他位置都没有使用变量counter。它的作用仅仅是充当for语句中的循环计数器。然而，因为它是一个变量，所以必须在程序中声明。

该程序循环实际上由其余所有的程序语句组成，如花括号所示。如果有以下概念，也许能更容易地理解该程序的工作方式：

```
For 5 times
{
   Get the number from the user.
   Calculate the requested triangular number.
   Display the results.
}
```

前面的循环部分指的是Calculate the requested triangular number，它实际上包括：将变量triangularNumber的值设为0，以及计算三角数的for循环。这样，这个for语句实际上包含另一个for语句。这在Objective-C中是相当合法的，而且可以继续嵌套，甚至可嵌套任何想要的层。

处理比较复杂的程序结构（如嵌套的for语句）时，适当地使用缩进变得尤为重要。扫视一下，就能轻易地确定每个for语句中包含哪些语句。

5.1.3 for循环的变形

在离开for循环的讨论之前，先探讨一下生成这种循环时允许的一些语法变化。编写for循环时，你可能发现在开始循环之前需要初始化多个变量，或者可能每次循环都要计算多个表达式。for循环的任何位置都可包含多个表达式，只要使用逗号分隔这些表达式。例如，使用以下形式开始的for循环

```
for ( i = 0, j = 0; i < 10; ++i )
   ...
```

在循环开始前，将i的值设为0，将j的值设为0。两个表达式i = 0和 j = 0通过逗号隔开，而且两者都是循环的init_expression部分。另一个例子，开始于

```
for ( i = 0, j = 100; i< 10; ++i, j -=10 )
   ...
```

的for循环设置了两个索引变量：i和j，在循环开始之前，它们分别被初始化为0和100。每次执行完循环体之后，i的值加1，j的值减10。

就像可能希望for循环的特定字段包含多个表达式一样，可能还需要省略语句中的一个或多个字段。通过省略指定的字段并使用分号标记其位置，可简单地实现这一点。省略for语句中某

个字段的最常见情形发生在无需计算初始表达式的值时。在这种情况下，init_expression字段可简单地保留空白，只要仍然包括分号即可，如下所示：

```
for ( ; j != 100; ++j )
   ...
```

如果在进入循环之前已经将j设置了初始值，则可采用这条语句。

省略looping_condition字段的for循环可有效地设置无限循环，就是理论上执行无限次的循环。只要有其他方式退出循环（例如，执行renturn、break或goto语句，将在本书后面讨论），就可以使用这一循环。

在for循环中，还可定义一个变量作为初始表达式的一部分。使用以前定义变量的传统方式可实现。例如，下面的语句可用于设置for循环，它定义了整形变量counter并将其初始化为1，如下所示：

```
for ( int counter = 1; counter <= 5; ++counter )
```

变量counter只在for循环的整个执行过程中是已知的（它名为局部变量），并且不能在循环外部访问。作为另一个例子，for循环

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )
   triangularNumber +=n;
```

定义了两个整型变量，并相应地设置了它们的值。

最后一种for循环变化可在对象集合上执行所谓的快速枚举，第15章“数字、字符串和集合”会详细介绍此内容。

5.2 while语句

while语句进一步扩展了Objective-C语言中的循环功能指令系统。这个经常使用的结构的语法如下：

```
while ( expression )
   program statement
```

圆括号中指定的expression将被求值。如果expression求值的结果为TRUE，则执行随后的program statement。执行完这条语句（或位于花括号中的语句组）后，将再次对expression求值。如果求值的结果为TRUE，将再次执行program statement。继续这个过程直到expression的最终求值结果是FALSE为止，此时循环将终止。然后，程序在program statement之后的语句继续执行。

作为它的用途示例，以下程序设置了一个while循环，它仅仅从1计数到5。

代码清单5-6

```
// This program introduces the while statement

#import <Foundation/Foundation.h>

#import <stdio.h>
```

```
int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int count = 1;

   while ( count <= 5 ) {
       NSLog (@"%i", count);
       ++count;
   }

   [pool drain];

   return 0;
}
```

代码清单5-6 输出

```
1
2
3
4
5
```

程序最初将count的值设为1，然后开始执行while循环。因为count的值小于或等于5，所以将执行它后面的语句。花括号将NSLog语句和对count执行加1操作的语句定义为while循环。从程序的输出可以看出：这个程序执行了5次，直到count的值是5为止。

从以上程序你可能认识到，使用for语句同样可以方便地完成该任务。事实上，for语句都可转换成等价的while语句，反之亦然。例如，下面这个普通的for语句

```
for (init_expression; loop_conditon; loop_expression )
     program statement
```

可用while语句的形式等价地表示，如以下所示：

```
init_expression;
while ( loop_condition )
{
     program statement
     loop_expression;
}
```

熟悉使用while语句之后，将对何时用while语句更合理以及何时用for语句有更好的认识。一般来说，最先选用for语句来实现执行预定次数的循环。同样，如果初始表达式、循环表达式和循环条件都涉及同一变量，那么for语句很可能是合适的选择。

下一个程序提供了使用while语句的另一个例子。这个程序计算两个整数值的最大公因子(greatest common divisor)。两个整数的最大公因子（此后将其缩写为gcd）是可整除这两个整数的最大整数值。例如，10和15的gcd是5，因为5是可整除10和15的最大整数。

可使用一个过程（或算法）来获得任意两个整数的gcd，它以Euclid于公元前300年左右首次研究出的一个方法为基础。其说明如下：

问题：找出两个非负整数u和v的最大公因子。

步骤1：若v等于0，则结束，gcd等于u。

步骤2：计算temp = u%v，u = v，v = temp并回到步骤1。

不要过分关注上述算法的运行细节，简单地相信它。此处我们更关注开发一个程序来找出最大公因子，而不是分析这一算法的实现方式。

使用算法描述找出最大公因子这一问题的解决方案之后，开发一个计算机程序成了一项十分简单的工作。算法步骤的分析揭示：只要v的值不等于0，就会重复执行步骤2。这一发现导致了该算法在Objective-C中使用while语句的自然实现。

代码清单5-7找出用户键入的两个整数的gcd。

代码清单5-7

```
// This program finds the greatest common divisor
// of two nonnegative integer values

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   unsigned int u, v, temp;

   NSLog (@'Please type in two nonnegative integers.');
   scanf ("%u%u", &u, &v);

   while ( v != 0 ) {
         temp = u % v;
         u = v;
         v = temp;
   }

   NSLog (@"Their greatest common divisor is %u", u);

   [pool drain];

   return 0;
}
```

代码清单5-7　输出

```
Please type in two nonnegative integers.
150 35
Their greatest common divisor is 5
```

代码清单5-7 输出（重新运行）

```
Please type in two nonnegative integers.
1026 540
Their greatest common divisor is 54
```

输入两个整型值并分别存储到变量u和v（使用%u格式字符读入一个无符号的整型值）之后，程序进入一个while循环来计算它们的最大公因子。退出while循环之后，u的值会显示出来，即代表v和u的原始值的gcd，并且显示一条适当的消息。

第7章“类”中，返回处理分数时，将再次使用这个算法来找出最大公因子。

对于下一个说明while语句的程序，设想的任务是翻转从终端输入的整数位。例如，如果用户键入数字1234，该程序将把这个数字的位颠倒过来，并显示结果4321。

注意 使用NSLog调用会导致每个数字出现在输出的单独行上。熟悉printf函数的C程序员可以使用该例程，而不是让数字连续地显示。

要编写这样的程序，首先必须提出一个算法来实现所陈述的工作。最常见的情况是，分析自己解决问题的方法可以产生一个算法。对于颠倒数字的位这项工作，解决方案可简单地陈述为“从右到左依次读取数字的位”。通过开发一个过程从数字最右边的位开始依次分离或取出该数字的每个位，计算机程序就可以依次读取数字的各个位。提取的位随后可以作为已颠倒数字的下一位显示在终端上。

通过将整数除以10之后取其余数，可提取整数最右边的数字。例如，1234 % 10会得出值4，就是1234最右边的数字。也是第一个要颠倒的数字（记住，可以使用模运算符得到一个整数除以另一个整数所得的余数）。通过将数字除以10这个过程，可以获得下一个数字。因此，1234 % 10的结果为123，而123 % 10的结果为3，它是颠倒数字的下一个数。

这个过程可一直继续执行，直到计算出最后一个数字为止。一般情况下，如果最后一个整数除以10的结果为0，那么这个数字就是最后一个要提取的数字。

代码清单5-8提示用户输入一个数值，然后从右向左依次显示该数值各个位的数字。

代码清单5-8

```
// Program to reverse the digits of a number

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int number, right_digit;

    NSLog (@"Enter your number.");
    scanf ("%i", &number);

    while ( number != 0 )  {
        right_digit = number % 10;
```

```
        NSLog (@"%i", right_digit);
        number /= 10;
    }

    [pool drain];
    return 0;
}
```

代码清单5-8　输出

```
Enter your number.
13579
9
7
5
3
1
```

5.3　do语句

迄今为止，本章讨论的这两个循环结构都要在循环开始前测试一组条件。因此，如果条件不满足，那么可能永远不会执行循环体。开发程序时，有时需要在循环结尾（而不是在开始）处执行测试。很自然，Objective-C语言也提供了专门的语言结构用于处理这种情况，即do语句。该语句的语法如下：

```
do
  program statement
while ( expression );
```

do语句按以下过程执行：首先，执行program statement。其次，求圆括号中expression的值。如果expression的求值结果为TRUE，循环将继续，并再次执行program statement。只要expression的计算结果始终为TRUE，就将重复执行program statement。当表达式求出的值为FALSE时，循环将终止并以正常顺序执行程序中的下一条语句。

do语句只是while语句的简单转置，它把循环条件放在循环的结尾而不是开头。

代码清单5-8使用while语句来翻转数字中的各个位。回到这个程序，并确定如果用户键入0而不是13579将会发生什么。在这种情况下，while循环中的语句将永远不会执行，输出中什么也不会显示。如果用do语句代替while语句，可确保程序循环要至少执行一次，从而保证在所有情况下都至少显示一个数字。代码清单5-9展示了如何使用do语句。

代码清单5-9

```
// Program to reverse the digits of a number

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
```

```
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int number, right_digit;

   NSLog (@"Enter your number.");
   scanf ("%i", &number);

   do {
        right_digit = number % 10;
        NSLog (@"%i", right_digit);
        number /= 10;
   }
   while ( number != 0 );

   [pool drain];
   return 0;
}
```

代码清单5-9 输出

```
Enter your number.
135
5
3
1
```

代码清单5-9 输出（重新运行）

```
Enter your number.
0
0
```

从该程序的输出可以看到，向程序键入0时，程序就会正确地显示数字0。

5.4 break语句

在执行循环的过程中，你希望只要发生特定的条件（例如，检测到错误条件或过早地到达数据末尾时），就立即退出循环。break语句可以用于实现这个目的。只要执行break语句，程序将立即退出正在执行的循环，而无论此循环是for、while还是do。循环中break之后的语句将被跳过，并且循环的执行也将终止，而转去执行循环之后的其他语句。

如果在一组嵌套循环中执行break语句，仅会退出执行break语句的最内层循环。

break语句的格式仅是在关键字break之后添加一个分号，如下所示：

```
break;
```

5.5 continue语句

continue语句和break语句类似，但它并不会使循环结束。执行continue语句时，循环会跳过该语句之后直到循环结尾处之间的所有语句。否则，循环将和平常一样执行。

continue通常用来根据某个条件绕过循环中的一组语句，否则，循环会继续执行。continue语句的格式如下：

```
continue;
```

只有熟悉了编写程序循环和如何从中正确退出之后，才使用break和continue语句。这些语句很容易滥用，并导致程序很难理解。

5.6 小结

既然，你已经熟悉了Objective-C语言提供的所有基本循环结构，可以继续学习这门语言中的其他类，使用它们能在程序运行过程中做出选择。在下一章中将详细介绍决定选择。

5.7 练习

1. 编写一个程序，为所有从1到100之间的整数n生成并显示n和n^2的表。确保能打印正确的列标题。
2. 使用以下公式，同样能为任何整数n生成三角数：

   ```
   triangularNumber = n (n + 1) / 2
   ```

   例如，第10三角数，也就是55，通过把上述公式中的n用10来代替，可以生成。编写一个程序，使用上述公式生成三角数表。用该程序在5到50之间每隔5个数生成一个三角数（就是说，生成第5、10、15，...，50个三角数）。
3. 整数n的阶乘可写成*n!*，它表示1到n之间所有连续整数的乘积。例如，5的阶乘可用以下方法计算：

   ```
   5! = 5 × 4 × 3 × 2 × 1 = 120
   ```

   编写一个程序，生成并打印前10个整数的阶乘表。
4. 字段宽度说明前面的负号能使字段按左对齐方式显示。用以下NSLog语句代替代码清单5-3中对应的语句，运行程序并比较这两种情况产生的结果：

   ```
   NSLog (@"%-2i %i", n, triangularNumber);
   ```

5. 代码清单5-5只允许用户键入5个不同的数字。修改这个程序，使用户能够键入要计算三角数的数字。
6. 重写代码清单5-2～5-5，用等价的while语句代替所有用到的for语句。运行每个程序，验证这两种方案是恒等的。
7. 如果在代码清单5-8中键入负数，会发生什么情况？试试看。
8. 编写一个程序，计算整数各个位上数字的和。例如，整数2155各个位上的数字和为2 + 1 + 5 +5，即13。该程序可接受用户键入的任意整数。

第 6 章 选 择 结 构

对于任何程序设计语言来说，其构造选择的能力都是一项基本特性。选择在执行循环语句时做出，以便确定何时终止循环。Objective-C程序设计语言也提供了其他几种用于构造选择的结构，本章将介绍它们：

- if语句
- switch语句
- conditional运算符

6.1 if语句

Objective-C程序设计语言提供了通用的构造选择能力，其语言结构就是所谓的if语句。这种语句的一般格式为

```
if ( expression )
    program statement
```

假设，要将“如果不下雨，我就去游泳”这样的句子转换成Objective-C语言。使用上述的if语句格式，可用Objective-C将这个句子“编写”成以下形式：

```
if ( 不下雨 )
    我就去游泳
```

if语句根据指定的条件规定程序语句（或括在花括号中的多条语句）的执行。如果不下雨，我就去游泳。类似地，在程序语句

```
if ( count > MAXIMUM_SONGS )
     [playlist maxExceeded];
```

中，只要count的值大于MAXIMUM_SONGS的值，就会将消息maxExceeded发送给playlist；否则，这条消息将被忽略。

一个实际的程序例子将帮助你理解这一点。假设想要编写一个程序，用于接受从键盘键入的整数，并随后显示这个整数的绝对值。计算整数绝对值的简明方法就是简单地在这个整数小于0时对它求反。在上述语句中使用的短语“如果它小于0”，表示程序必须做出选择。if语句可以影响这个选择，如以下程序所示。

代码清单6-1

```
// Calculate the absolute value of an integer

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
```

```
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int number;

    NSLog (@"Type in your number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        number = -number;

    NSLog (@"The absolute value is %i", number);

    [pool drain];
    return 0;
}
```

代码清单6-1　输出

```
Type in your number:
-100
The absolute value is 100
```

代码清单6-1　输出（重新运行）

```
Type in your number:
2000
The absolute value is 2000
```

程序执行了两次，以验证它能正确地运行。当然，你可能希望该程序执行更多次，以获得更高水准的可信度，这样就知道该程序确实能正确地工作，但是你至少应该知道，已经检查了该程序所做选择的两种可能结果。

向用户显示一条消息之后，用户将输入整数值，程序将该值存储到number中，然后程序测试number的值以确定该值是否小于0。如果这个值小于0，将执行以下程序语句，对number的值求反。如果number的值不小于0，将自动略过这条程序语句（如果这个值已经是正的，则无需对它求反）。随后，程序将显示number的绝对值，并终止运行。

看一看另一个使用if语句的程序。再向Fraction类添加一个名为convertToNum的方法。这个方法将提供一个用实数表示的分数值。换言之，该方法将用分子除以分母并用双精度的值返回结果。因此，对于分数1/2，该方法将返回值0.5。

这样的方法声明可能如下所示：

```
-( double ) convertToNum;
```

而且，以下就是它的定义：

```
-( double ) convertToNum
{
```

```
    return numerator / denominator;
}
```

当然，并非完全如此。实际上，定义这个方法时有两个重要问题。能发现它们吗？第一个和算术转换有关。你会想起，numerator和denominator都是整型的实例变量。因此，对这两个整数执行除法时，将出现什么情况？正确的情况，它们会作为整数除法来完成！因此，要将分数1/2转换成实数，上述代码将得出结果0！通过在执行除法之前，使用类型强制运算符将一个或两个运算数转换成浮点值，可轻松地解决这个错误。

```
(double) numerator / denominator
```

回忆一下，这种运算符的优先级相对较高，因此，在执行除法之前先将numerator转换成double类型。此外，无需转换denominator，因为算术运算的规则将替你完成这项工作。

使用这种方法要注意的第二个问题是应该检查被除数是否为0（总应该检查这种情况）。这个方法的调用者可能由于疏忽而忘记了设置此分数的分母或将分母设置为0，而且你并不希望程序异常终止。

修改后的convertToNum方法显示如下：

```
-(double) convertToNum
{
  if (denominator != 0)
     return (double) numerator / denominator;
  else
     return 0.0;
}
```

如果分数的denominator为0，此处规定它返回0.0。另外，还可以使用其他选择（如，打印一条错误消息、抛出一个异常，等等），但不在这里做进一步讨论了。

使用这个新方法。下面是用于测试它的测试程序。

代码清单6-2

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

@interface Fraction: NSObject
{
  int   numerator;
  int   denominator;
}

-(void)   print;
-(void)   setNumerator: (int) n;
-(void)   setDenominator: (int) d;
-(int)    numerator;
-(int)    denominator;
-(double) convertToNum;
@end
```

```
@implementation Fraction
-(void) print
{
     NSLog (@" %i/%i ', numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 0.0;
}
@end

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

    [aFraction setNumerator: 1];  // 1st fraction is 1/4
    [aFraction setDenominator: 4];

    [aFraction print];
    NSLog (@" =");
```

```
        NSLog (@"%g", [aFraction convertToNum]);

        [bFraction print];      // never assigned a value
        NSLog (@" =");
        NSLog (@"%g", [bFraction convertToNum]);
        [aFraction release];
        [bFraction release];

        [pool drain];
        return 0;
}
```

代码清单6-2 输出

```
1/4
 =
0.25
0/0
 =
0
```

将aFraction设置成1/4后，程序会使用convertToNum方法将此分数转换成一个十进制的值。这个值随后就会显示出来，即0.25。

在第二种情况下，没有明确地设置bFraction的值，因此，这个分数的分子和分母会初始化为0，这是实例变量默认的初始值。这就解释了print方法显示的结果。同时，它还导致convertToNum方法中的if语句返回值0，从输出可以验证。

6.1.1 if-else结构

如果有人问你特定的数是偶数还是奇数，最有可能的情况是，通过检查这个数的最后一位数字做出决定。如果最后一位数字是0、2、4、6或8中的任何一个，就能说这个数是偶数。否则，它是奇数。

确定特定的数是偶数还是奇数时，计算机使用一种更方便的方法，它并不检查这个数的最后一位数字来观察此数字是否是0、2、4、6或8，而是简单地通过检验这个数能否整除2来确定。如果能整除2，这个数是偶数；否则，就是奇数。

你已经知道，如何使用模运算符%来计算两个整数相除所得的余数。这种能力使它成为在确定整数能否整除2时使用的理想运算符。如果某个数除以2所得的余数为0，它就是偶数；否则，就是奇数。

现在，编写一个程序用于确定用户键入的整型值是偶数还是奇数，并随后在终端显示一条正确的消息，参见代码清单6-3。

代码清单6-3

```
// Program to determine if a number is even or odd
```

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

int main (int argc, char *argv[])

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int number_to_test, remainder;

    NSLog (@'Enter your number to be tested: ");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        NSLog (@"The number is even.");

    if ( remainder != 0 )
        NSLog (@"The number is odd.");

    [pool drain];
    return 0;
}
```

代码清单6-3　输出

```
Enter your number to be tested:
2455
The number is odd.
```

代码清单6-3　输出（重新运行）

```
Enter your number to be tested:
1210
The number is even.
```

键入一个数后，计算此数除以2所得的余数。第一条if语句测试了这个余数的值，检验它是否等于0。如果等于0，将显示消息“The number is even”。

第二条if语句测试余数，检验它是否不等于0，如果不等于0，就会显示一条消息声明这个数是奇数。

实际的情况是：只要第一条if语句成功，第二条if语句肯定失败；反之亦然。如果回顾本节开始处有关偶数/奇数的讨论，就会知道：如果一个数能被2整除，它就是偶数；否则就是奇数。

编写程序时，这个“否则”的概念使用得相当频繁，以至几乎所有的现代程序设计语言都提供了一个特殊结构来处理这一情况。在Objective-C中，它通常称为if-else结构，一般格式如下：

```
if ( expression )
    program statement 1
else
    program statement 2
```

实际上，if-else仅仅是if语句一般格式的一种扩展形式。如果表达式的计算结果是TRUE，将执行之后的program statement 1；否则，将执行program statement 2。在任何情况下，都会执行program statement 1或program statement 2两者中的一个，而不是两个都执行。

可将if-else语句结合到前面的程序中，用单个if-else语句替换程序中的两个if语句。你将看到，使用这种新的程序语句如何实际地帮助程序在一定程度上减少复杂性，同时又提高可读性。

代码清单6-4

```
// Determine if a number is even or odd (Ver. 2)

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int number_to_test, remainder;

    NSLog (@"Enter your number to be tested:');
    scanf ("%i', &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
       NSLog (@"The number is even.");
    else
       NSLog (@'The number is odd.");

    [pool drain];
    return 0;
}
```

代码清单6-4 输出

```
Enter your number to be tested:
1234
The number is even.
```

代码清单6-4 输出（重新运行）

```
Enter your number to be tested:
6551
The number is odd.
```

不要忘记，双等号（==）用于相等测试，而单个等号是赋值运算符。如果忘记了这一点，并因疏忽而在if语句中使用了赋值运算符，将会导致许多令人头痛的事。

6.1.2 复合条件测试

到目前为止，本章使用过的if语句都只是两个数之间的简单条件测试。代码清单6-1将number的值与0做了比较，而代码清单6-2则将分数的分母与0做了比较。有时候即使不是必需的，也需要设置多个复杂测试。例如，假设要计算一次考试中分数介于70到79之间的分数个数，包括70和79在内。在这种情况下，不是根据一个界限比较考试分数的值，而是根据两个界限70和79进行比较，以确保一个考试分数位于指定的范围之内。

Objective-C语言提供了用于执行这种复合条件测试所必需的机制。复合条件测试仅仅是用逻辑与或者逻辑或运算符中的一个连接起来的一个或多个简单条件测试。这两个运算符分别使用字符对&&和||（两个竖线字符）来表示。举一个例子，Objective-C语句

```
if ( grade >= 70 && grade <= 79 )
   ++grades_70_to_79;
```

只在grade的值大于等于70或小于等于79时才对grade_70_to_79的值加1。使用类似的方式，语句

```
if ( index < 0 || index > 99 )
     NSLog (@"Error - index out of range");
```

在index小于0或大于99时会执行NSLog语句。

在Objective-C中，复合运算符可用于形成极其复杂的表达式。Objective-C向程序员提供了在构成表达式时极大的灵活性，这种灵活性经常被滥用。比较简单的表达式往往更容易阅读和调试。

形成复合条件表达式时，可大量使用圆括号来加强表达式的可读性，并避免由于错误假设表达式中的运算符优先级而陷入麻烦之中（与任何算术运算符或关系运算符相比，&&运算符有更低的优先级，但比||运算符的优先级要高）。在表达式中还应该使用空格以加强表达式的可读性。&&和||运算符两旁的空格将在视觉上把用这些运算符连接起来的表达式与运算符分隔开。

要在实际的程序例子中说明复合条件测试的用途，编写一个程序来测试某个年份是不是闰年。我们知道，如果某个年份能被4整除，它就是闰年。然而，你可能没有认识到，能被100整除的年份并不是闰年，除非它能同时被400整除。

试着考虑，如何为这样的条件设置测试。首先，计算某个年份除以4、100和400所得的余数，并将这些值分别指派给名称合适的变量，如rem_4、rem_100和rem_400。然后，可以继续测试这些余数，以确定是否满足闰年的标准。

如果重新表达上述的闰年定义，可将它表示为：如果某个年份能被4整除但不能被100整除、或某个年份能被400整除，这一年就是闰年。暂停一会儿思考最后一个句子，并自己验证这个句子等价于前面陈述的定义。现在，已经用这些条件重新阐述了闰年的定义，将这些句子转换成以下程序语句的过程变得相对简单和直观，如下所示：

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
    NSLog (@"It's a leap year.");
```

子表达式

```
rem_4 == 0 && rem_100 != 0
```

两边的圆括号不是必需的，因为表达式就是以这种方式求值的，记住，&&比||有更高的优先级。

事实上，在这个特定的例子中，测试

```
if ( rem_4 == 0 && ( rem_100 != 0 || rem_400 == 0 ) )
```

将运作良好。

如果在测试之前添加几条语句，声明变量并允许用户从终端键入年份，最终就会生成一个用来确定某个年份是不是闰年的程序，如代码清单6-5所示。

代码清单6-5

```
// This program determines if a year is a leap year

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int year, rem_4, rem_100, rem_400;

    NSLog (@"Enter the year to be tested: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;

    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        NSLog (@"It's a leap year.");
    else
        NSLog (@"Nope, it's not a leap year.");

   [pool drain];
   return 0;
}
```

代码清单6-5 输出

```
Enter the year to be tested:
1955
Nope, it's not a leap year.
```

代码清单6-5　输出（重新运行）

```
Enter the year to be tested:
2000
It's a leap year.
```

代码清单6-5　输出（重新运行）

```
Enter the year to be tested:
1800
Nope, it's not a leap year.
```

上述例子使用了3个年份：一个并非闰年（1955），因为它不能被4整除；一个闰年（2000），因为它能被400整除；一个并非闰年（1800），因为它能被100整除但不能被400整除。要使测试用例完整，还应改检验能被4整除但不能被100整除的年份是否是闰年。这将作为练习由你来完成。

我们提到Objective-C在创建表达式方面为程序员提供了极大的灵活性。例如，在前一个程序中，不必计算中间结果rem_4、rem_100和rem_400，可能直接在if语句中进行了计算，如下所示：

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
```

用于分隔各种运算符而使用的空格还加强了上述表达式的可读性。如果决定不添加空格并删除非必需的圆括号，将会获得以下表达式：

```
if(year%4==0&&year%100!=0||year%400==0)
```

这个表达式是完全合法的，运行结果将等价于（不管你是不是相信）前面刚刚显示的表达式。显然，这些额外的空格在帮助理解复杂表达式方面起到很大作用。

6.1.3　嵌套的if语句

讨论if语句的一般格式时，我们指出：如果圆括号中表达式的求值结果是TRUE，将执行之后的语句。如果这条程序语句是另外一条if语句，也是完全合法的，如以下语句：

```
if ( [chessGame isOver] == NO )
     if ( [chessGame whoseTurn] == YOU )
          [chessGame yourMove];
```

如果向chessGame发送的isOver消息所返回的值为NO，将执行随后的语句，它是另一条if语句。这条if语句比较从whoseTrun方法返回的值与YOU。如果这两个值相等，将向chessGame对象发送yourMove消息。因此，只有在两边的对弈都未结束且轮到你移动棋子时，才发送yourMove消息。事实上，使用复合关系可将这条语句等价地表示成以下形式：

```
if ( [chessGame isOver] == NO && [chessGame whoseTurn] == YOU )
     [chessGame yourMove];
```

对于嵌套的if语句，更实际的例子可能是对上述例子添加了else子句之后的形式，如下所示：

```
if ( [chessGame isOver] == NO )
     if ( [chessGame whoseTurn] == YOU )
```

```
            [chessGame yourMove];
      else
            [chessGame myMove];
```

这条语句的执行方式和前面描述过的相同。然而，如果对弈没有结束并且不该你移动棋子，将执行else子句。这将向chessGame发送消息myMove。如果对弈结束，就会跳过后面整个if语句，包括对应的else子句。

要注意else子句如何与if语句对应，即用于测试从whoseTurn方法返回的值的if语句，而非用于测试对弈是否结束的if语句。一般规则是：else子句通常与最近的不包含else子句的if语句对应。

在上述例子中，可进一步为最外层的if语句添加else子句。这条else子句将在对弈结束时执行：

```
if ( [chessGame isOver] == NO )
      if ( [chessGame whoseTurn] == YOU )
            [chessGame yourMove];
      else
            [chessGame myMove];
else
      [chessGame finish];
```

当然，即使使用缩进来表明你认为Objective-C语言解释语句的方式，但它并不总是与系统实际解释语句的方式一致。例如，从上述例子中删除第一个else子句

```
if ( [chessGame isOver] == NO )
      if ( [chessGame whoseTurn] == YOU )
            [chessGame yourMove];
else
      [chessGame finish];
```

系统并不会以这种格式解释这条语句。而是使用以下形式解释这条语句：

```
if ( [chessGame isOver] == NO )
      if ( [chessGame whoseTurn] == YOU )
            [chessGame yourMove];
      else
            [chessGame finish];
```

这是因为else子句与最近的无else子句的if语句对应。在最内部的if语句不包含else子句，但外部的if语句却包含else子句的情况下，可用花括号强制表示不同的关联。花括号具有隔离if语句的作用。因此，

```
if ( [chessGame isOver] == NO ) {
      if ( [chessGame whoseTurn] == YOU )
            [chessGame yourMove];
}
else
      [chessGame finish];
```

可实现所希望的结果。

6.1.4 else if结构

你已经看到，测试两个可能的条件（任何数不是偶数，就是奇数；任何年份或者是闰年，或者不是）时，else语句是如何工作的。然而，必须制定的程序设计选择并不总是这么明确。考虑以下任务：编写一个程序，在用户键入的数小于0时显示-1；在此数等于0时显示0；在大于0时显示1（这实际上是所谓的sign函数的一种实现）。显然，在这种情况下，必须做3次测试以确定键入的数是否是负数、0或正数。此时，简单的if-else结构就不再适用。当然，在这种情况下，总是可以使用3条独立的if语句，但这种解决方案并不总是可行的，特别是在做出测试并非相互排斥的时候更是如此。

通过向else子句添加一条if语句，就能处理这种情况。我们提到过，在else子句之后的语句可以是任何合法的Objective-C程序语句，为什么不能是另一条if语句呢？因此，在一般情况下，可编写以下形式：

```
if ( expression 1 )
    program statement 1
else
    if ( expression 2 )
        program statement 2
    else
        program statement 3
```

这种形式有效地扩展了if语句，使其从双值逻辑判定变成了3个值的逻辑判定。可以用刚才显示的方式继续向else子句添加if语句，以便有效地将这种选择扩展成n个值的逻辑判定。

上述结构的使用相当频繁，它通常称作else if结构，其格式经常与上面显示的不同，例如：

```
if ( expression 1 )
    program statement 1
else if ( expression 2 )
    program statement 2
else
    program statement 3
```

后面这种格式提高了语句的可读性，并使做出的具有3个路线的选择更加清晰。

之后的程序通过实现前面讨论的sign函数说明了else if结构的使用。

代码清单6-6

```
// Program to implement the sign function

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int number, sign;
```

```
    NSLog (@'Please type in a number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else        // Must be positive
        sign = 1;

    NSLog (@"Sign = %i", sign);
    [pool drain];
    return 0;
}
```

代码清单6-6 输出

```
Please type in a number:
1121
Sign = 1
```

代码清单6-6 输出（重新运行）

```
Please type in a number:
-158
Sign = -1
```

代码清单6-6 输出（重新运行）

```
Please type in a number:
0
Sign = 0
```

如果输入的数小于0，将给sign指派值−1；如果此数等于0，将给sign指派0；否则，这个数一定大于0，因此给它指派值1。

下面的程序分析从终端键入的字符，并根据字母符号（a～z或A～Z）、数字（0～9）或特殊字符（其他任何字符）将其分类。要从终端读取单个字符，需要在scanf调用中使用格式字符%c。

代码清单6-7

```
// This program categorizes a single character
//       that is entered from the keyboard

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
```

```
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    char c;

    NSLog (@"Enter a single character:");
    scanf ("%c", &c);

    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        NSLog (@"It's an alphabetic character.");
    else if ( c >= '0' && c <=  9' )
       NSLog (@"It's a digit.");
    else
       NSLog (@"It's a special character.");

   [pool drain];
   return 0;
}
```

代码清单6-7　输出

```
Enter a single character:
&
It s a special character.
```

代码清单6-7　输出（重新运行）

```
Enter a single character:
8
It's a digit.
```

代码清单6-7　输出（重新运行）

```
Enter a single character:
B
It's an alphabetic character.
```

读入字符之后构建的第一个测试确定了char变量c是否是字母符号。这项工作通过测试该字符是否是小写字母或大写字母来完成。测试的前半部分由以下表达式组成：

```
( c >= 'a' && c <= 'z' )
```

如果c位于字符'a'和'z'之间，表达式为TRUE；就是说，如果c是小写字母，表达式为TRUE。测试的后半部分由下面这个表达式组成：

```
( c >= 'A' && c <= 'Z' )
```

如果c位于字符‘A’和‘Z’之间，表达式为TRUE；就是说，如果c是大写字母，表达式为TRUE。这些测试适用于以ASCII的格式存储字符的计算机系统上。

如果变量c是字母符号，且第一个if测试获得成功，将显示消息“It's an alphabetic character.”。如果测试失败，将执行else if子句。这个子句确定此字符是否为数字。注意，这个

测试将字符c和字符‘0’和‘9’进行比较，而不是整数0和9。这是因为字符是从终端读入的，并且字符‘0’到‘9’不同于数字0到9。事实上，在ASCII中，字符‘0’在内部实际表示为数字48，字符‘1’表示为49，依此类推。

如果c是数字字符，将显示短语“It's a digit.”。否则，如果c不是字母符号并且不是数字字符，将执行最后的else子句，并在终端显示短语“It's a special character.”。然后，程序就会结束。

应该注意到，尽管此处使用scanf读取单个字符，但键入字符之后仍需按下Enter键以便向程序发送输入。一般来说，无论何时从终端读入数据，在按下Enter键之前，程序都不会看到在数据行键入的任何数据。

假设在下一个例子中希望编写程序，允许用户使用以下形式键入简单的表达式：

```
number operator number
```

程序将计算表达式并在终端显示结果。可以识别的运算符是普通的加法、减法、乘法和除法运算符。在这里使用第4章“数据类型和表达式”的代码清单4-6中的Calculator类。因此，每个表达式都向计算器进行计算。

以下程序使用具有多个else if子句的大型if语句来确定要执行的运算。

注意 最好使用标准库中的例程islower和isupper，并彻底避免内部表示问题。为此，需要在程序中包含程序行 #import<ctype.h>。然而，放到这里只是为了说明的目的。

代码清单6-8

```
// Program to evaluate simple expressions of the form
//        number operator number

// Implement a Calculator class

#import <Foundation/Foundation.h>

@interface Calculator: NSObject
{
    double accumulator;
}

// accumulator methods
-(void)  setAccumulator: (double) value;
-(void)  clear;
-(double) accumulator;

// arithmetic methods
-(void)  add: (double) value;
-(void)  subtract: (double) value;
-(void)  multiply: (double) value;
-(void)  divide: (double) value;
@end
```

```
@implementation Calculator
-(void) setAccumulator: (double) value
{
    accumulator = value;
}

-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
@end

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    double      value1, value2;
    char        operator;
    Calculator  *deskCalc = [[Calculator alloc] init];

    NSLog (@"Type in your expression.");
    scanf ("%lf %c %lf", &value1, &operator, &value2);
```

```
    [deskCalc setAccumulator: value1];
    if ( operator == '+' )
        [deskCalc add: value2];
    else if ( operator == '-' )
        [deskCalc subtract: value2];
    else if ( operator == '*' )
       [deskCalc multiply: value2];
    else if ( operator == '/' )
       [deskCalc divide: value2];

    NSLog (@"%.2f", [deskCalc accumulator]);
    [deskCalc release];

    [pool drain];
    return 0;
}
```

代码清单6-8 输出

```
Type in your expression.
123.5 + 59.3
182.80
```

代码清单6-8 输出（重新运行）

```
Type in your expression.
198.7 / 26
7.64
```

代码清单6-8 输出（重新运行）

```
Type in your expression.
89.3 * 2.5
223.25
```

scanf调用指定了要读入到变量value1、operator和value2中的3个值。double值可用%lf格式字符读入。这就是用于读入变量value1值的格式，而变量value1是表达式的第一个运算数。

接下来，读入运算符。因为运算符是字符（+、−、*或/）而非数字，因此需要将它读入到字符变量运算符中。%c格式字符可告诉系统从终端读入下一个字符。格式字符串中的空格表示输入中允许存在任意个数的空格。这样在键入这些值时可以使用空格将运算数和运算符分隔开。

读入两个值和运算符之后，程序将第一个值存储到计算器的累加器中。然后，将operator的值和4个允许的运算数做比较。如果存在正确的匹配，相应的消息就会发送给计算器来执行运算。在最后一个NSlog中，检索累加器的值用于显示。然后程序就会结束。

在这里，我们将简单地讨论一下有关程序的彻底性（thoroughness）的话题。尽管以上程序确实可以完成设置的任务，但是因为它并没有考虑用户所犯的错误，所以实际并不完善。例

如，如果用户为运算符错误地键入了一个“？”，将发生什么情况？执行if语句时，程序只会失败，而且终端上不会显示消息警告用户他错误地键入了表达式。

另一种被忽略的情况是，用户键入一个使用0作为除数的除法运算。至此，你知道在Objective-C中永远不要试图用某个数去除以0。该程序应该能检查这种情况。

尽量预测程序可能失败或产生非预期结果的情形，然后采取预防性措施应付这些情况，是产生优秀而可靠的程序的必要部分。对程序运行大量的测试用例，通常可以找出没有考虑的特定情形。但它不仅仅如此。编写程序时，总是询问“如果……将发生什么情况？”并插入必要的程序语句来适当地处理该情况时，这就成为自律的问题。

代码清单6-8A，即修改之后的代码清单6-8，考虑了除以0和键入未知运算符的情形。

代码清单6-8A

```
// Program to evaluate simple expressions of the form
//    value  operator  value

#import <Foundation/Foundation.h>

// Insert interface and implementation sections for
// Calculator class here

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    double     value1, value2;
    char       operator;
    Calculator *deskCalc = [[Calculator alloc] init];

    NSLog (@"Type in your expression.");
    scanf ("%lf %c %lf", &value1, &operator, &value2);

    [deskCalc setAccumulator: value1];

    if ( operator == '+' )
       [deskCalc add: value2];
    else if ( operator == '-' )
       [deskCalc subtract: value2];
    else if ( operator == '*' )
       [deskCalc multiply: value2];
    else if ( operator == '/' )
       if ( value2 == 0 )
            NSLog (@"Division by zero.");
       else
            [deskCalc divide: value2];
    else
        NSLog (@"Unknown operator.");
```

```
    NSLog (@"%.2f", [deskCalc accumulator]);
    [deskCalc release];

    [pool drain];
    return 0;
}
```

代码清单6-8A 输出

```
Type in your expression.
123.5 + 59.3
182.80
```

代码清单6-8A 输出（重新运行）

```
Type in your expression.
198.7 / 0
Division by zero.
198.7
```

代码清单6-8A 输出（重新运行）

```
Type in your expression.
125 $ 28
Unknown operator.
125
```

键入的运算符是斜杠时，对除法而言，要执行另一个测试以确定value2是否为0。如果是0，将在终端显示一条适当的消息；否则，将执行除法运算并显示结果。在这种情况下，要特别注意嵌套的if语句和对应的else子句。

程序结尾的else子句会捕获所有失败。因此，任何与测试的4个字符不匹配的operator值都会导致执行else子句，并导致在终端上显示”Unknown operator.”。

处理除0这个问题的较好方法是，在计算除法的方法内部执行测试。因此，可将divide：方法修改成以下形式：

```
-(void) divide: (double) value
{
   if (value != 0.0)
      accumulator /= value;
   else {
      NSLog (@"Division by zero.");
      accumulator = 99999999.;
   }
}
```

如果value非0，将执行除法；否则，将显示消息并将累加器设置为99999999。这是随意的，可以将它设置成0，或可能设置一个特殊值来表示出现一个错误条件。一般来说，最好让方法

处理特殊的情况，而不是依赖程序员使用方法。

6.2 switch语句

这种类型的if-else语句和上一个程序例子有关（其中一个变量的值与不同的值连续进行比较），开发包含Objective-C语言中特有程序语句的程序，以便精确地执行这种功能时，经常用到这种功能。该语句的名称为switch语句，它的一般格式如下：

```
switch ( expression )
{
    case value1:
        program statement
        program statement
          ...
        break;
    case value2:
        program statement
        program statement
          ...
        break;
     ...
    case valuen:
        program statement
        program statement
          ...
        break;
    default:
        program statement
        program statement
          ...
        break;
}
```

括在圆括号中的expression与value1、value2，……，valuen连续进行比较，后者必须是单个常量或常量表达式。如果发现某种情况下某个Value的值与expression的值相等，就执行该情况之后的程序语句。注意，包含多条这样的程序语句时，它们不必括在圆括号中。

break语句表示一种特定情况的结束，并导致switch语句的终止。记住，每种情况的结尾都要包含break语句。如果忘记为特定的情况执行这项操作，只要执行这种情况，程序就会继续执行下一种情况。有时需要有意这么做，如果选择这样的方式，请务必插入注释以便将你的目的告知他人。

如果expression的值不与任何情况的值匹配，将执行一种特殊的、名为default的可选情况。这在概念上等价于前一个例子中使用的else。事实上，switch语句的一般格式可以等价地表示成以下if语句：

```
if ( expression == value1 )
{
```

```
        program statement
        program statement
            ...
    }
    else if ( expression == value2 )
    {
        program statement
        program statement
            ...
    }
       ...
    else if ( expression == valuen )
    {
        program statement
        program statement
            ...
    }
    else
    {
        program statement
        program statement
            ...
    }
```

记住前面的代码，可以将代码清单6-8A中的大型if语句转换成等价的switch语句。我们把这个新程序叫做代码清单6-9。

代码清单6-9

```
// Program to evaluate simple expressions of the form
//       value operator  value

#import <Foundation/Foundation.h>

// Insert interface and implementation sections for
// Calculator class here

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    double  value1, value2;
    char    operator;
    Calculator *deskCalc = [[Calculator alloc] init];

     NSLog (@"Type in your expression.");
    scanf ("%lf %c %lf", &value1, &operator, &value2);
```

```
    [deskCalc setAccumulator: value1];

    switch ( operator ) {
       case '+':
          [deskCalc add: value2];
          break;
       case '-':
          [deskCalc subtract: value2];
          break;
       case '*':
          [deskCalc multiply: value2];
          break;
       case '/':
          [deskCalc divide: value2];
          break;
       default:
          NSLog (@"Unknown operator.");
          break;
    }

    NSLog (@"%.2f", [deskCalc accumulator]);
    [deskCalc release];

    [pool drain];
    return 0;
}
```

代码清单6-9　输出

```
Type in your expression.
178.99 - 326.8
-147.81
```

在读入表达式之后，operator的值会连续与每种情况指定的值进行比较。发现一个匹配的值时，将执行包含在这种情况中的语句。然后，break语句将终止switch语句的执行，程序也会在此处结束。如果不存在匹配operator值的情况，将执行可显示“Unkown operator.”的default语句。

以上程序中，default情况中的break语句实际上不是必需的，因为这种情况之后的switch中不存在任何语句。然而，记住在每种情况的结尾都包含break语句是一种良好的程序设计习惯。

编写switch语句时，应该记住任何两种情况的值都不能相同。然而，可以将多个情况的值与一组程序语句关联起来。简单地在要执行的普通语句之前列出多个情况的值（每种情况中值的前面都使用关键字case，而且后面要有一个冒号）就能实现该任务。作为一个例子，在以下switch语句中，

```
switch ( operator )
{
```

```
    ...
  case '*':
  case 'x':
      [deskCalc multiply: value2];
      break;
    ...
}
```

如果operator等于星号或是小写字母x，将执行multiply:方法。

6.3 Boolean变量

几乎所有学习程序设计的人很快就会发现自己要解决这样一个问题，即编写一个程序生成素数表。为了提醒你，回顾一下：如果一个正整数p不能被1和它本身之外的其他任何整数整除，就是一个素数。第一个素数规定为2。下一个素数是3，因为它不能被1和3之外的任何整数整除；而4不是素数，因为它能被2整除。

可以采用几种方法来生成素数表。例如，如果你的任务是生成50以内的所有素数，那么最直接的（也是最简单的）算法是生成这样一个表：它仅仅对每个整数p测试是否能被2到p-1间的所有整数整除。如果这样的任何整数能整除p，那么p就不是素数；否则，p就是素数。

代码清单6-10生成了2到50之间的素数列表。

代码清单6-10

```
// Program to generate a table of prime numbers

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int   p, d, isPrime;

   for ( p = 2; p <= 50; ++p ) {
        isPrime = 1;

        for ( d = 2; d < p; ++d )
              if ( p % d == 0 )
                    isPrime = 0;

        if ( isPrime != 0 )
              NSLog (@"%i ", p);
   }

   [pool drain];
   return 0;
}
```

代码清单6-10　输出

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

关于代码清单6-10，有几个要点值得注意。最外层的for语句建立了一个循环，周期性地遍历2到50间的整数。循环变量p表示当前正在测试以检查是否是素数的值。循环中的第一条语句将值1指派给变量isPrime。你很快就会明白这个变量的用途。

建立的第二个循环用于将p除以从2到p-1间的所有整数。在该循环中，要执行一个测试以检查p除以d的余数是否为0。如果为0，就知道p不可能是素数，因为它能被不同于1和它本身的整数整除。为了表示p不再可能是素数，可将变量isPrime的值设置为0。

执行完最内层的循环时，测试isPrime的值。如果它的值不等于0，表示没有发现能整除p的整数，否则，p肯定是素数，并显示它的值。

你可能已经注意到，变量isPrime只接受值0或值1，而不是其他值。只要p还有资格成为素数，它的值就是1。但是一旦发现它有整数约数时，它的值将设为0，以表示p不再满足成为素数的条件。以这种方式使用的变量一般称作Boolean变量。通常，一个标记只接受两个不同值中的一个。此外，标记的值通常要在程序中至少测试一次，以检查它是on（TRUE或YES）还是off（YES或TRUE），而且根据测试结果采取的特定操作。

在Objective-C中，标记为TRUE或FALSE的概念大部分被自然地转换成值1或0。因此在代码清单6-10中，在循环中将isPrime的值设置为1时，实际上将把它设置成TRUE表示p“是素数”。如果在内层for循环的执行过程中发现了一个偶数约数，isPrime的值将设置为FALSE以表示p不再“是素数”了。

以下情况并不一致：值1通常用于表示TRUE或on状态，而0用于表示FALSE或off状态。这种表示与计算机内部单个比特的概念对应。当比特位于开状态时，它的值是1；当位于关状态时，值是0。但在Objective-C中，有一种甚至更令人信服的理由来支持这些逻辑值。这与Objective-C语言处理TRUE和FALSE概念的方式有关。

在开始本章的讨论时我们注意到，如果满足if语句内部指定的条件，将执行其后的程序语句。但是满足的确切含义是什么？在Objective-C中，满足意味着非零，而不是其他的值。因此，

语句

```
if ( 100 )
    NSLog (@"This will always be printed.");
```

将导致执行NSLog语句，因为if语句中的条件（本例中仅指值100）非零，因此它是满足的。

本章的每一个程序中，都会用到这个概念，即“非零意味着满足”和“零意味着不满足”。这是因为，只要对Objective-C中的关系表达式进行求值，如果满足表达式时结果将为1，不满足时将为0。因此，语句

```
if ( number < 0 )
   number = -number;
```

的求值实际上按以下步骤进行：求关系表达式number<0的值。如果条件满足，就是，如果number小于0，表达式的值将是1；否则，它的值是0。

if语句测试了表达式求值的结果。如果结果非零，将执行其后的语句；否则，将略过这条语句。

前面的讨论还用到for、while和do语句中的条件求值。如以下语句中的复合关系表达式的求值

```
while ( char != 'e' && count != 80 )
```

也会收到前面指出的结果。如果指定的两个条件都满足，结果是1；但如果有任何一个条件不满足，求值的结果就是0。然后，检查求值的结果。如果结果为0，while循环就会终止；否则，它会继续执行。

回到代码清单6-10和标记的概念上，使用表达式测试标记的值是否为TRUE，这在Objective-C中是相当合法的，例如

```
if ( isPrime )
```

等价于

```
if ( isPrime != 0 )
```

要方便地测试标记的值是否为FALSE，需要使用逻辑非运算符。在以下表达式中，

```
if ( ! isPrime )
```

使用逻辑非运算符来测试isPrime的值是否为FALSE（这条语句可读做“如果非isPrime”）。一般来说，如下表达式

```
! expression
```

用于对expression的逻辑值求反。因此，如果expression为0，逻辑非运算符将产生1。并且如果expression的求值结果非零，逻辑非运算符就会产生0。

可以使用逻辑非运算符轻易地跳过标记的值，如在以下表达式中：

```
my_move = ! my_move;
```

正如你期望的，这种运算符的优先级和一元运算符相同。这意味着与所有二元算术运算符和关系运算符相比，它的优先级较高。因此，要测试变量x的值是否不小于变量y的值，如在

```
! ( x < y )
```

中，圆括号是必需的，它用于确保表达式的正确求值。当然，也可将上述语句等价地表示为

```
x >= y
```

Objective-C中有一对内置的特性可以使Boolean变量的使用更容易一些。一种是特殊类型BOOL，它可以用于声明值非真即假的变量。另一种是内置值YES或NO。在程序中使用这些预定义的值可使它们更易于编写和读取。下面是使用这些特性重写的代码清单6-10。

注意　类型BOOL其实是由一种称为预处理程序的机制添加的。

代码清单6-10A

```
// Program to generate a table of prime numbers
// second version using BOOL type and predefined values

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int    p, d;
   BOOL  isPrime;

   for ( p = 2; p <= 50; ++p ) {
      isPrime = YES;

      for ( d = 2; d < p; ++d )
         if ( p % d == 0 )
            isPrime = NO;

      if ( isPrime == YES )
          NSLog (@"%i ", p);
    }

   [pool drain];
   return 0;
}
```

代码清单6-10A　输出

```
2
3
5
7
11
13
17
```

```
19
23
29
31
37
41
43
47
```

6.4 条件运算符

Objective-C语言中最普通的运算符很可能是一种名为关系运算符的运算符。与Objective-C中的其他运算符不同（它们不是一元运算符就是二元运算符），关系运算符是三元运算符；就是说，它要接受3个运算数。用于表示这种运算符的两个符号是问号（？）和冒号（:）。第一个运算数放在？之前，第二个运算数放在？与：之间，而第三个运算符放在：之后。

条件运算符的一般格式为：

```
condition ? expression1 : expression2
```

在这个语法中，condition表示一个表达式，通常是关系表达式，只要遇到关系运算符，Objective-C系统就会先对它求值。如果condition求值的结果是TRUE（就是说，它是非零的），将执行expression1而且其结果将成为该运算的结果。如果condition的求值结果为FALSE（就是说，它是零），将执行expression2，而且它的结果会成为该运算的结果。

条件表达式最常用在根据条件将两个值中的一个指派给变量时。例如，假设有一个整型变量x和另一个整型变量s。如果希望在x小于0时将-1指派给s，否则将x2的值指派给s，可以写成以下语句：

```
s = ( x < 0 ) ? -1 : x * x;
```

执行上述语句时，首先要测试条件x<0。通常在条件表达式两边放置圆括号来增强语句的可读性。虽然，这一般不是必需的，因为条件运算符的优先级非常低，实际上，它低于其他所有运算符的优先级，但赋值运算符和逗号运算符除外。

如果x的值小于0，就会求？之后表达式的值。这个表达式仅仅是常量整数值-1，它将在x小于0时指派给变量s。

如果x的值不小于0，将求：之后的表达式的值，并将该值赋值给s。因此，如果x大于或等于0，将把x*x（或x2）的值指派给s。

作为使用条件运算符的另一个例子，使用以下语句将a和b的最大值指派给变量max_value：

```
max_value = ( a > b ) ? a : b;
```

如果在：（即“else”部分）之后使用表达式包含另一个条件运算符，就可以达到else if子句的效果。例如，代码清单6-6中实现的sign函数可以使用两个条件运算符写到一个程序行上，如下所示：

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

如果number小于0，将给sign指派值−1；否则，如果number等于0，将给sign指派值0；否则，将给它指派值1。上面的表达式中，“else”部分两旁的圆括号实际上是不必要的。这是因为条件运算符是从右到左结合的，这意味着在单个表达式中可以使用多个运算符，如在

```
e1 ? e2 : e3 ? e4 : e5
```

中，条件运算符从右到左结合，因此用以下形式求值：

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

条件表达式并非必须用在赋值运算符的右边，它们可以用在可以使用的表达式的任何位置。这意味着可以在没有将变量number的符号指派给变量的情况下使用NSLog语句来显示它，如下所示：

```
NSLog (@"Sign = %i", ( number < 0 ) ? -1
                                    : ( number == 0 ) ? 0 : 1);
```

使用Objective-C编写预处理程序宏指令时，条件预算符非常便利。可以在第12章“预处理程序”看到它们的详细内容。

6.5 练习

1. 编写一个程序，请求用户在终端键入两个整数。测试这两个值以确定第一个数是否可以被第二个数整除，然后在终端显示一条适当的消息。
2. 即使输入非法运算符或试图除以0时，代码清单6-8A也会显示累加器中的值。修改该程序。
3. 修改Fraction类的print方法，使其同样可以显示整数（因此，分数5/1可以只显示成5）。再修改这个方法使其能将分子是0的分数显示成0。
4. 编写一个程序，使其作为简单的打印计算器。该程序应该允许用户键入以下形式的表达式：

   ```
   number operator
   ```

   程序还应该可以识别以下运算符：

   ```
   +   -   *   /   S   E
   ```

   S运算符通知程序将累加器设置成键入数字模式，而E运算符通知程序终止运行。使用键入的数字作为第二个运算数对累加器的内容执行算术运算。以下是显示程序应该如何运算的示例操作：

   ```
   Begin Calculations
   10 S                Set Accumulator to 10
   = 10.000000         Contents of Accumulator
   2 /                 Divide by 2
   = 5.000000          Contents of Accumulator
   55 -                Subtract 55
   = -50.000000
   100.25 S            Set Accumulator to 100.25
   ```

```
= 100.250000
4 *                 Multiply by 4
= 401.000000
0 E                 End of program
= 401.000000
End of Calculations.
```

确信该程序可以检测除数为0的情况，还能对未知的运算符进行检查。使用代码清单6-8中开发的Calculator类来执行运算。

5. 开发代码清单5-9用于翻转从终端键入数的各个位。然而，如果键入负数，这个程序就不能很好地运行。找出这种情况下发生了什么事情，然后修改这个程序以便正确地处理负数。正确地处理，指的是如果（例如）键入数-8645，程序的输出将是5468-。
6. 编写一个程序，用于接受从终端键入的整数，提取并用英语显示这个数的每一个数字。因此，如果用户键入932，程序就会显示以下内容：

```
nine
three
two
```

（记住，用户只键入一个0时将显示zero）。注意：这个练习很难！

7. 代码清单6-10存在几个低效的方面。其中一个低效的方面是由检查偶数引起的。因为任何大于2的偶数显然不能是素数，因此可以简单地略过偶数作为可能的素数和可能的除数。内层的for循环也是低效的，因为始终使用2到p-1之间的所有d值除以p。如果在for循环的条件中添加用于判断isPrime值的测试，可以避免这种低效性。使用这种方式，只要没有发现除数而且d的值小于p，for循环就将继续执行。修改代码清单6-10，将这两种修改合并到一起，然后运行程序以验证它的运算。

第7章 类

在本章中，你将继续学习如何使用类以及如何编写方法。还将用到以前章节中所学的一些概念，例如程序循环、构造选择和使用表达式。首先，讨论一下将程序分成多个文件，使较大的程序更容易处理。

7.1 分离接口和实现文件

现在，是时候将类的声明和定义放在单独的文件中。

如果正在使用Xcode，可新建一个称为FractionTest的项目。在文件FractionTest.m中键入以下程序。

代码清单7-1 主测试程序：FractionTest.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction  *myFraction = [[Fraction alloc] init];

    // set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // display the fraction

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
    return 0;
}
```

注意，该文件未包括Fraction类的定义。然而，它确实导入了一个称为Fraction.h的文件。

通常，类的声明（即，@interface部分）要放在它自己的名为class.h的文件中。而类的定义（即，@implementation部分）通常放在相同名称的文件中，但扩展名要使用.m。所以，把Fraction类的声明放到Fraction.h文件中，把Fraction类的定义放到Fraction.m文件中。

要在Xcode中完成该工作，在File菜单中选择New File。在左侧窗格中，选择Cocoa。在右

上窗格中，选择Objective-C类。现在窗口应该如图7-1所示。

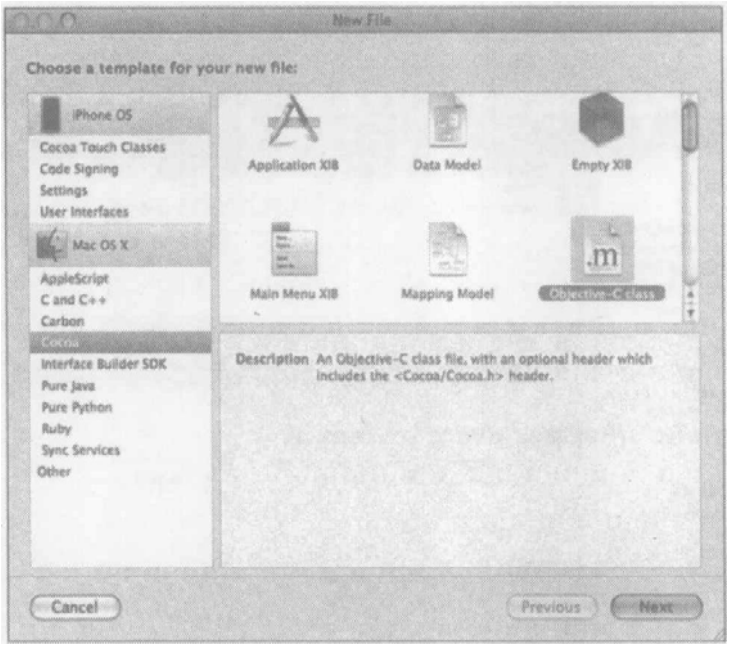

图7-1 Xcode New File菜单

点击Next，键入Fraction.m作为文件名。选中Also Create Fraction.h框。该文件所在的位置应该与FractionTest.m文件的文件夹相同。现在窗口应该如图7-2所示。

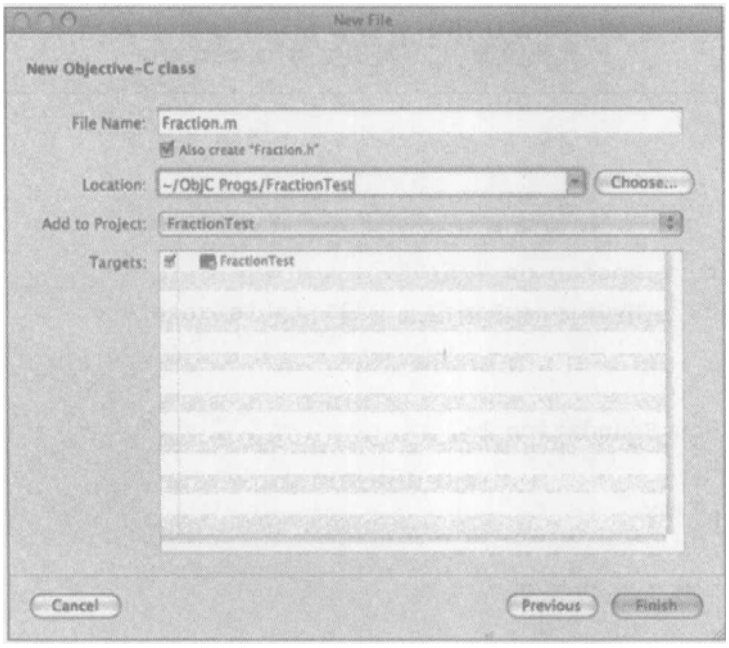

图7-2 在项目中添加新类

现在点击Finish。Xcode在项目中添加了两个文件：Fraction.h和Fraction.m。图7-3显示了这两个文件。

在此我们不使用Cocoa，所以将Fraction.h中的

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

一行内容改为

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

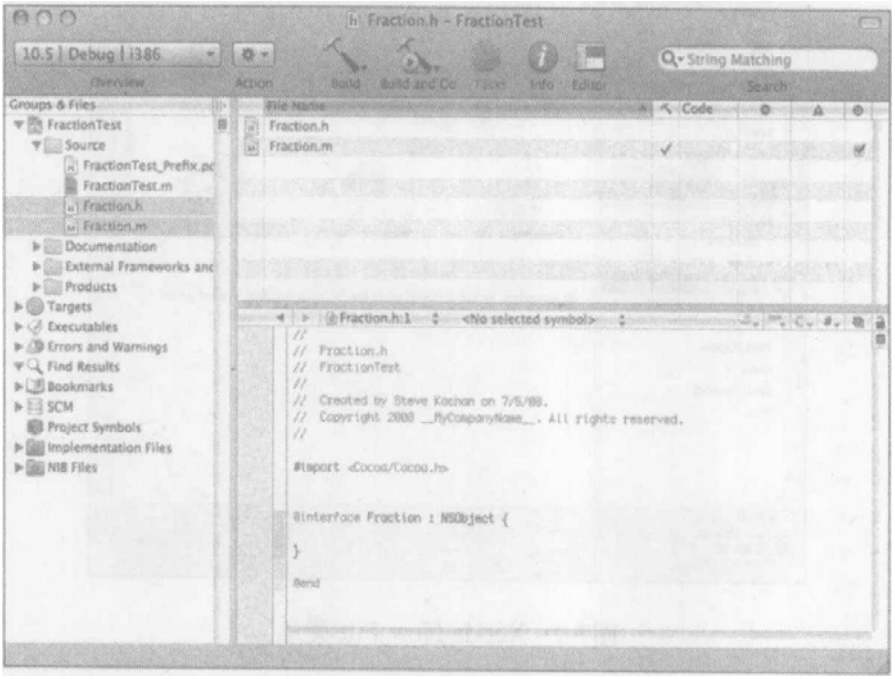

图7-3 Xcode为新类创建了文件

在同一文件中（Fraction.h），现在可输入类的接口部分，如代码清单7-1所示：

代码清单7-1　接口文件Fraction.h

```
//
//  Fraction.h
//  FractionTest
//
//  Created by Steve Kochan on 7/5/08.
//  Copyright 2008 __MyCompanyName__. All rights reserved.
//

#import <Foundation/Foundation.h>

// The Fraction class

@interface Fraction : NSObject
{
  int  numerator;
  int   denominator;
}
-(void)    print;
-(void)    setNumerator: (int) n;
-(void)    setDenominator: (int) d;
-(int)     numerator;
```

```
-(int)      denominator;
-(double)   convertToNum;

@end
```

接口文件告诉编译器（并告知其他程序员，以后你将学到这些内容）Fraction类的外观特征，它包括两个名为numerator和denominator的实例变量，而且它们都是整数。它还包含6个实例方法：print、setNumerator:、setDenominator:、numerator、denominator和convertToNum。前3个方法不返回值；之后两个方法返回int值；最后一个方法返回double值。setNumerator:和setDenominator:方法各接收一个整型参数。

Fraction类的实现细节位于文件Fraction.m中。

代码清单7-1 实现文件：Fraction.m

```
//
//  Fraction.m
//  FractionTest
//
//  Created by Steve Kochan on 7/5/08.
//  Copyright 2008 __MyCompanyName__. All rights reserved.
//

#import "Fraction.h"

@implementation Fraction
-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
   numerator = n;
}

-(void) setDenominator: (int) d
{
   denominator = d;
}

-(int) numerator
{
   return numerator;
}

-(int) denominator
{
```

```
    return denominator;
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 1.0;
}
@end
```

注意，使用以下语句将接口文件导入到实现文件中：

```
#import "Fraction.h"
```

这样做的目的是，使编译器知道为Fraction类声明的类和方法，同时还能确保这两个文件的一致性。还要提醒一下，一般不能（虽然可以这么做）在实现部分重复声明类的实例变量，所以编译器需要从Fraction.h中包含的接口部分获得信息。

需要注意的另一件事是：导入的文件要用一对引号括起来，而不是<Foundation/Foundation.h>中的<和>字符。双引号用于本地文件（你自己创建的文件），本地文件与系统文件相对，并且它们告诉编译器在哪里找到指定的文件。使用双引号，编译器一般会首先在当前目录寻找指定文件，然后再转到其他位置查找。如果有必要，可以指定编译器要查找的实际位置。

下面是这个例子的测试程序，我们已经将它键入文件FractionTest.m中。

代码清单7-1　main测试程序：FractionTest.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
     Fraction  *myFraction = [[Fraction alloc] init];

    // set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // display the fraction

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
    return 0;
}
```

还要注意，测试程序FractionTest.m包括接口文件Fraction.h，而不包括实现文件Fraction.m。现在，该程序分成了三个独立的文件。对于小程序例子而言，这似乎要花费很多工作，但是，着手处理较大的程序并和其他程序员共享类声明时，这种实用性就会变得十分明显。

现在可以编译并运行程序，方法与以前使用的一样：选择Build菜单中的Build and Go，或者点击主Xcode窗口中的Build and Go图标。

如果使用命令行编译程序，要为Objective-C编辑器提供这两个".m"文件名。如果使用gcc，则命令行可能如下：

```
gcc -framework Foundation Fraction.m FractionTest.m -o FractionTest
```

这将构建一个名为FractionTest的可执行文件。下面就是运行该程序的输出：

代码清单7-1 输出

```
The value of myFraction is:
1/3
```

7.2 合成存取器方法

从Objective-C 2.0开始，可自动生成设置函数方法和获取函数方法（统称为存取器方法）。我们到目前为止都没有介绍如何实现，原因是知道如何自己编写这些方法非常重要。然而，该语言提供了一个很方便的功能，所以现在应该学习如何充分利用这个功能了。

第一步是在接口部分中使用@property指令标识属性。这些属性通常是实例变量。在Fraction类中，两个实例变量numerator和denominator都属于此类属性。下面是新的接口部分，其中添加了新的@property指令。

```
@interface Fraction : NSObject
{
    int numerator;
    int denominator;
}

@property int numerator, denominator;

-(void)    print;
-(double)  convertToNum;
@end
```

注意，我们没有包括下列设置函数方法和获取函数方法的定义：numerator、denominator、setNumerator：和setDenominator：。我们要让Objective-C 2.0编译器为我们自动生成或合成这些方法。如何完成呢？只需在实现部分中使用@synthesize指令即可，如下所示。

```
#import "Fraction.h"

@implementation Fraction
```

```
@synthesize numerator, denominator;

-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(double) convertToNum
{
   if (denominator != 0)
      return (double) numerator / denominator;
   else
      return 1.0;
}
@end
```

下面这行内容告诉Objective-C编译器，为两个实例变量（numerator和denominator）的每一个生成一对设置函数方法和获取函数方法：

```
@synthesize numerator, denominator;
```

通常，如果有称为X的实例变量，那么在实现部分包括以下行会导致编译器自动实现一个获取函数方法X和一个设置函数方法setX：

```
@synthesize x;
```

即使此处看起来并非什么大事，但是让编译器完成这项工作是值得的，因为生成的存取器方法是高效的，并且在使用多个核心的多台机器上，使用多线程时也可正常运行。

现在回到代码清单7-1并按照上述内容对接口和实现部分进行更改，为你合成存取器方法。确定最终的输出与没有任何变化的FractionTest.m是相同的。

7.3　使用点运算符访问属性

Objective-C语言允许你使用非常简便的语法访问属性。要获得myFraction中存储的numerator的值，可使用以下语句：

```
[myFraction numerator]
```

这会向myFraction对象发送numerator消息，从而返回所需的值。从Objective-C 2.0开始，现在可以使用点运算符编写以下等价的表达式：

```
myFraction.numerator
```

一般格式为：

```
instance.property
```

还可使用类似的语法进行赋值：

```
instance.property = value
```

这等价于编写以下表达式：[instance setProperty: value]

在代码清单7-1中，使用以下两行代码将分数的numerator和denominator设置为1/3：

```
[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

下面是两行等价的代码：

```
myFraction.numerator = 1;
myFraction.denominator = 3;
```

我们为合成方法使用这些新功能，并在本书的后面部分都使用这种方法访问属性。

7.4 具有多个参数的方法

继续使用Fraction类并对它做一些补充。你已经定义了6个方法。如果有一个方法能够只用一条消息既可设置numerator同时又可设置denominator，就太好了。通过列出每个连续的参数并用冒号将其连起来，就可以定义一个接收多个参数的方法。用冒号连接的参数将成为这个方法名的一部分。例如，方法名addEntryWithName:andEmail:表示接收两个参数的方法，这两个参数可能是姓名和电子邮件地址。方法addEntryWithName:andEmail:andPhone:是接收以下3个参数的方法，一个姓名、一个电子邮件地址和一个电话号码。

同时设置numerator和denominator的方法可以命名为setNumberator:andDenominator:，你可能采用以下形式：

```
[myFraction setNumerator: 1 andDenominator: 3];
```

这种方法还不错。实际上，这是命名方法的首选方式。但是必须为方法指定更易阅读的名称。例如，觉得setTo:over:如何？这个名称乍一看并没有什么吸引力，通过将myTraction设置为1/3，比较这个名称和前面的名称：

```
[myFraction setTo: 1 over: 3];
```

我认为这个名称的可读性更强，但是，究竟选择哪种命名方式由你决定（实际上，有些人更喜欢第一种命名方式，因为它明确地引用了类中的实例变量名称）。再者，选择好的方法名对整个程序的可读性是很重要的。写出实际的消息表达式可帮助你找出一个最好的。

让我们应用这种新方法。首先，在接口文件中添加setTo:over:声明。如代码清单7-2所示。

代码清单7-2 接口文件:Fraction.h

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

// Define the Fraction class

@interface Fraction : NSObject
{
    int numerator;
    int denominator;
}

@property int numerator, denominator;
```

```
-(void)    print;
-(void)    setTo: (int) n over: (int) d;
-(double)  convertToNum;
@end
```

然后，在实现文件中添加新方法定义。

代码清单7-2　实现文件:Fraction.m

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
   NSLog (@"%i/%i", numerator, denominator);
}

-(double) convertToNum
{
   if (denominator != 0)
     return (double) numerator / denominator;
   else
     return 1.0;
}

-(void) setTo: (int) n over: (int) d
{
   numerator = n;
   denominator = d;
}
@end
```

新的setTo:over:方法仅接收两个整型参数，n和d，并把它们赋值给该分数对应的域numerator和denominator。

下面是新方法的测试程序。

代码清单7-2　测试文件：main.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   Fraction *aFraction = [[Fraction alloc] init];
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

```
    [aFraction setTo: 100 over: 200];
    [aFraction print];

    [aFraction setTo: 1 over: 3];
    [aFraction print];
    [aFraction release];

    [pool drain];
    return 0;
}
```

代码清单7-2 输出

```
100/200
1/3
```

7.4.1 不带参数名的方法

创建方法名时，参数名实际上是可选的。例如，可以如下声明一个方法：

```
-(int) set: (int) n: (int) d;
```

注意，和先前的例子不同，这个方法的第二个参数没有名字。这个方法名为set::，两个冒号表示这个方法有两个参数，虽然没有全部命名。

要调用set::方法，可以使用冒号作为参数分隔符，如下所示：

```
[aFraction set: 1 : 3];
```

在编写新方法时，省略参数名不是一种好的编程风格，因为它使程序很难读懂并且很不直观，特别是当使用的方法参数特别重要时更是如此。

7.4.2 关于分数的操作

继续探讨上面提到的Fraction类。首先，编写一个方法，将一个分数与另一个分数相加，将一个方法命名为add:，并且把一个分数作为参数。这个新方法的声明如下所示：

```
-(void) add: (Fraction *) f;
```

注意参数f的声明：

```
(Fraction *) f
```

这条语句说明add:方法的参数类型是Fraction类。星号是必需的，所以声明

```
(Fraction) f
```

是不正确的。你将把一个分数作为参数传递给add:方法，而这一方法将其添加到消息的接收器中。所以消息表达式

```
[aFraction add: bFraction];
```

将Fraction bFraction和Fraction aFraction相加。就像一个快速数学复习程序，它用于将分数a/b和c/d相加，如下执行计算：

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd}$$

下面是@implementation部分的新方法代码：

```
// add a Fraction to the receiver

- (void) add: (Fraction *) f
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   numerator = numerator * f.denominator
               + denominator * f.numerator;
   denominator = denominator * f.denominator;
}
```

不要忘了通过域numerator和donominator，可以指定Fraction作为消息的接收器。而另一方面，不能用这种方法直接指定参数f的实例变量。相反，必须通过对f应用点运算符来获得实例变量（或者通过向f发送相应的消息）。

假设将新add:方法的声明和定义添加到接口和实现文件中。代码清单7-3是示例测试和输出。

代码清单7-3　测试文件:FractionTest.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction *aFraction = [[Fraction alloc] init];
   Fraction *bFraction = [[Fraction alloc] init];

   // Set two fractions to 1/4 and 1/2 and add them together

   [aFraction setTo: 1 over: 4];
   [bFraction setTo: 1 over: 2];

   // Print the results

   [aFraction print];
   NSLog (@"+");
   [bFraction print];
   NSLog (@"=");

   [aFraction add: bFraction];
```

```
    [aFraction print];
    [aFraction release];
    [bFraction release];

    [pool drain];
    return 0;
}
```

代码清单7-3 输出

```
1/4
+
1/2
=
6/8
```

这个测试程序非常易懂。两个分数，名为aFraction和bFraction的两个Fraction被分配空间和初始化。然后分别被赋值为1/4和1/2。然后将Fraction bFraction与Fraction aFraction相加，之后显示相加的结果。注意add:方法将消息对象的参数相加，所以对象被修改了。在main程序尾打印aFraction的值时将证明该点。应该注意，要在调用add:方法之前显示aFraction的值，这样可以在add:方法改变aFraction的值之前显示其原值。本章的结尾，你将重新定义add:方法，以便add:方法不影响其接收器的值。

7.5 局部变量

你也许注意到1/4和1/2相加的结果显示为6/8，而不是期望的3/4，这是因为加法例程只执行算术不做其他处理，它不会将结果相约。所以继续练习如何编写有关分数运算的新方法。编写一个新的reduce方法，将分数相约到它的最简形式。

回顾一下中学学过的数学课程，通过找到能同时整除分子和分母的最大数来约简分数，然后使用这个数除分子和分母。技术上，要找出分子和分母的最大公约数（gcd）。代码清单5-7已告知如何得到它。稍一回忆也许就会想起这个程序。

根据这些算法，可以编写出名为reduce的新方法：

```
- (void) reduce
{
    int  u = numerator;
    int  v = denominator;
    int  temp;

    while (v != 0) {
        temp = u % v;
        u = v;
        v = temp;
    }
```

```
    numerator /= u;
    denominator /= u;
}
```

注意这个reduce方法中有一些新东西：它声明了三个整型变量u、v和temp。这些变量是局部变量，意思是它们的值只在reduce方法运行时才存在并且只能在定义它们的方法中访问。从这个意义上来说，它们类似于在main例程中定义的变量；这些变量对main来说是局部变量，可以在main例程中直接进行访问。你定义的其他方法都不能访问main中的局部变量。

局部变量没有默认的初始值，所以在使用前要先赋值。reduce程序中有三个局部变量在使用之前被赋值，所以没有问题。和实例变量不同（它们通过方法调用赋值），这些局部变量不存在于内存中，也就是，当方法返回时，这些变量的值都消失了。每次调用方法，该方法中的局部变量（如果有的话）都使用变量声明初始化一次。

7.5.1 方法的参数

方法的参数名也是局部变量。执行方法时，通过方法传递的任何参数都被复制到局部变量中。因为方法使用参数的副本，所以不能改变通过方法传递的原值。这是一个重要概念。假设有个名为calculate:的方法，其定义如下：

```
-(void) calculate: (double) x
{
    x *= 2;
    ...
}
```

并假设使用以下消息表达式来调用它：

```
[myData calculate: ptVal];
```

执行calculate方法时，ptVal变量所含的任何值都被复制到局部变量x。所以改变calculate中x的值，对ptVal的值没有影响，只影响x变量的数据副本。

顺便提及，如果参数是对象，可以更改其中的实例变量值，我们在下一章中将了解更多内容。

7.5.2 static关键字

在变量声明前加上关键字static，可以使局部变量保留多次调用一个方法所得的值。例如

```
static int hitCount = 0;
```

声明整数hitCount是一个静态变量。和其他常见局部变量不同，静态变量的初始值为0，所以前面显示的初始化是多余的。此外，它们只在程序开始执行时初始化一次，并且在多次调用方法时保存这些数值。

所以编码序列

```
-(void) showPage
{
    static int pageCount = 0;
    ...
```

```
    ++pageCount;
      ...
}
```

可能出现在一个showPage方法中，它用于记录该方法的调用次数（在这种情况下，还可能是要打印的页数）。只在程序开始时局部静态变量设置为0，并且在连续调用showPage方法时获得新值。

注意，将pageCount设置为局部静态变量和实例变量之间的区别。对于前一种情况，pageCount能记录调用showPage方法的所有对象打印页面的数目。对后一种情况，pageCount变量可以计算每个对象打印的页面数目，因为每个对象都有自己的pageCount副本。

记住只能在定义静态和局部变量的方法中访问这些变量。所以，即使静态变量pageCount，也只能在showPage函数中访问。可以将变量的声明移到所有方法声明的外部（通常放在implementation文件的开始处），这样所有方法都可以访问它们，如：

```
#import "Printer.h"
static int pageCount;

@implementation Printer
  ...
@end
```

现在，该文件中包含的所有实例或者类方法都可以访问变量pageCount。第10章“变量和数据类型”详细地讲述了变量作用域的主题。

返回关于分数的讨论，将reduce方法的代码结合到实现文件Fraction.m中。不要忘了在接口文件Fraction.h中声明reduce方法，之后，就可以在代码清单7-4中测试这个新方法。

代码清单7-4 测试文件FractionTest.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction *aFraction = [[Fraction alloc] init];
   Fraction *bFraction = [[Fraction alloc] init];

   [aFraction setTo: 1 over: 4];   // set 1st fraction to 1/4
   [bFraction setTo: 1 over: 2];   // set 2nd fraction to 1/2

   [aFraction print];
   NSLog (@"+");
   [bFraction print];
   NSLog (@"=");

   [aFraction add: bFraction];

   // reduce the result of the addition and print the result
```

```
    [aFraction reduce];
    [aFraction print];

    [aFraction release];
    [bFraction release];

    [pool drain];
    return 0;
}
```

代码清单7-4　输出

```
1/4
+
1/2
=
3/4
```

这次更好了！

7.6　self关键字

在代码清单7-4中，我们决定在add:方法之外约简分数，还可以在add:中进行约简。怎么做完全是随意的。然而，如何确定使用reduce方法约简的分数？我们知道如何通过变量名直接确定方法中的实例变量，但是不知道如何直接确定消息的接收者。

关键字self用来指明对象是当前方法的接收者。如果在add:方法中编写

```
[self reduce];
```

就可以对Fraction应用reduce方法，它正是你希望的add:方法的接收者。在学习本书过程中，可以看到关键字self多么有用。现在，在add:方法中使用它。下面是修改后的结果：

```
- (void) add: (Fraction *) f
{
    // To add two fractions:
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    numerator = (numerator * [f denominator]) +
        (denominator * [f numerator]);
    denominator = denominator * [f denominator];

    [self reduce];
}
```

这样，执行加法之后，分数被约简了。

7.7　在方法中分配和返回对象

我们注意到add:方法改变了接收该消息的对象值。创建新版本的add:函数，这个方法创建

了一个新的分数来存储相加的结果。在这种情况下，需要向消息的发送者返回新的Fraction。下面是新方法add:的定义：

```
-(Fraction *) add: (Fraction *) f
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   // result will store the result of the addition
   Fraction   *result = [[Fraction alloc] init];
   int        resultNum, resultDenom;

   resultNum = numerator * f.denominator +
       denominator * f.numerator;
   resultDenom = denominator * f.denominator;

   [result setTo: resultNum over: resultDenom];
   [result reduce];

   return result;
}
```

方法定义的第一行是：

```
-(Fraction *) add: (Fraction *) f;
```

这一行说明add:方法将返回一个名为Fraction的对象，并且说明它还使用一个Fraction作为参数。这个参数将与消息的接收者相加。这个接收者同样是Fraction。

这一方法分配并初始化了一个新的Fraction对象，名为result，然后定义两个名为resultNum和resultDenom的局部变量。它们将用于存储程序相加运算产生的分子分母。

和以前一样执行完加法运算之后，将产生的分子和分母复制给局部变量，然后使用以下消息表达式设置result的值：

```
[result setTo: resultNum over: resultDenom];
```

约简结果之后，将结果返回给消息的发送者。

注意在add:方法中分配给Fraction result的空间被返回，并且没有被系统释放。不能从add:方法中释放，因为add:方法的调用者需要它。因此这个方法的使用者知道返回的对象是一个新变量，并将被随后释放。用户通过可用的合适文档，可以了解这些信息。

代码清单7-5测试了新方法add:。

代码清单7-5 测试文件main.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

```
    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

    Fraction *resultFraction;

    [aFraction setTo: 1 over: 4];    // set 1st fraction to 1/4
    [bFraction setTo: 1 over: 2];    // set 2nd fraction to 1/2

    [aFraction print];
    NSLog (@"+");
    [bFraction print];
    NSLog (@"=");

     resultFaction = [aFraction add: bFraction];
    [resultFraction print];

    // This time give the result directly to print
    // memory leakage here!

    [[aFraction add: bFraction] print];
    [aFraction release];
    [bFraction release];
    [resultFraction release];

    [pool drain];
    return 0;
}
```

代码清单7-5　输出

```
1/4
+
1/2
=
3/4
3/4
```

这里要做一点补充，首先，你定义了两个Fraction类，aFraction和bFraction，并将它们的值分别设为1/4和1/2，然后又定义了一个名为resultFraction的Fraction类(为什么不必为它分配空间并初始化呢)。这个变量将用来存储相加操作的结果。

以下代码

```
resultFraction = [aFraction add: bFraction];
[resultFraction print];
```

首先发送add:消息到aFraction类，同时将类Fraction bFraction作为它的参数。该方法返回的结果Fraction存储到resultFraction中，然后，通过向它发送一条print消息，显示这个结果。注意虽

然你没有在main例程中为它分配空间，但是在程序的末尾一定要将resultFraction释放。它是由add:方法分配空间的，但是由你负责清除它。消息表达式

```
[[aFraction add: bFraction] print];
```

看起来很不错，但是它实际上存在一个问题。因为你使用add:方法返回的Fraction类，并向其发送一条要print的消息，你没有办法释放add:方法创建的Fraction类。这是一个内存泄漏的例子。如果程序中有很多这样的嵌套消息，最终这些分数的存储空间就会累加，这些分数的内存没有被释放。每次，你将添加或者泄漏少许内存（它们不能直接恢复）。

问题的一个解决方案是将print方法返回它的接收者，然后可以将它释放。但这有点兜圈子。一个更好的解决方案是将嵌套的消息分成两个单独的消息，和在程序中以前做的一样。

随便说一句，可以在add:方法中避免使用临时变量resultNum和resultDenom。相反，单条消息调用

```
[result setTo: numerator * f.denominator + denominator * f.numerator
         over: denominator * f.denominator];
```

将实现这个目的！并不推荐编写如此简明的代码。但是，如果查看其他程序员的编码，可能会发现这种代码，所以了解如何阅读和理解这些强大的表达式很有用。

我们介绍本章最后一个分数。例如，考虑计算以下内容：

$$\sum_{i=1}^{n} 1/2^i$$

Σ符号是总和的缩写。用在这里是表示将$1/2^i$加到一起，i从1到n。即1/2+1/4+1/8……。如果n足够大，这个序列的总和应该接近于1。试验不同的n值，来看看有多接近1。

代码清单7-6提示你提供不同的n值，并执行相应的计算。

代码清单7-6 FractionTest.m

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction *aFraction = [[Fraction alloc] init];
   Fraction *sum = [[Fraction alloc] init], *sum2;
   int i, n, pow2;

   [sum setTo: 0 over: 1]; // set 1st fraction to 0

   NSLog (@"Enter your value for n:");
   scanf ("%i", &n);

   pow2 = 2;
```

```
    for (i = 1; i <= n; ++i) {
        [aFraction setTo: 1 over: pow2];
        sum2 = [sum add: aFraction];
        [sum release];  // release previous sum
        sum = sum2;
        pow2 *= 2;
    }

    NSLog (@"After %i iterations, the sum is %g", n, [sum convertToNum]);
    [aFraction release];
    [sum release];

    [pool drain];
    return 0;
}
```

代码清单7-6　输出

```
Enter your value for n:
5
After 5 iterations, the sum is 0.96875
```

代码清单7-6　输出（重新运行）

```
Enter your value for n:
10
After 10 iterations, the sum is 0.999023
```

代码清单7-6　输出（重新运行）

```
Enter your value for n:
15
After 15 iterations, the sum is 0.999969
```

当分子为0分母为1时Fraction Sum被设为0（如果你将分子分母都设为0会怎样？）。然后程序提示用户输入n值，并使用scanf读取它。然后进入for循环来计算这个序列的总和。首先，将变量pow2初始化为2。这个变量用来储存2^i的值。所以，每循环一次它的值被乘以2。

for循环从1开始到n结束，每循环一次，将aFraction设为1/pow2或$1/2^i$。然后这个值通过前面定义的add:方法与累积的sum变量相加。Add:的结果被赋值给sum2，而不是sum以免内存漏洞问题（如果你直接赋值给sum会怎样）。然后释放旧的sum，sum的新值sum2将赋值给sum，以便进行下一轮迭代。学习在代码中释放分数的方式，就可以熟悉用来避免内存泄漏的策略。如果这是一个执行成百上千次的for循环，而你不懂得如何释放分数，就会迅速积累很多浪费的内存空间。

for循环结束时，使用convertToNum方法将结果以十进制值的方式显示。那时只有两个对

象要释放，aFraction和最后存储在sum中的Fraction对象。然后程序执行结束。

输出展示了在MacBook Air中独立运行该程序三次后所发生的情况。第一次，计算出这个序列的和，显示结果值0.96875。第三次使用n值为15运行该程序，计算结果非常接近1。

扩展类的定义和接口文件

现在，你已经开发了处理分数的一个小方法库。下面是接口文件，它是完整地列出的，所以可以看到该类实现的所有方法：

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

// Define the Fraction class

@interface Fraction : NSObject
{
  int  numerator;
  int  denominator;
}

@property int numerator, denominator;

-(void)       print;
-(void)       setTo: (int) n over: (int) d;
-(double)     convertToNum;
-(Fraction *) add: (Fraction *) f;
-(void)       reduce;
@end
```

你可能不需要处理分数，但是这些例子告诉你如何通过加入新方法来定义和扩展一个类。其他处理分数的人将使用这个接口文件，并且使用这个文件足够编写他自己的处理分数的程序。如果他需要添加新方法，通过直接扩展类定义或者定义自己的子类并添加自己的新方法直接地扩展该类，可以实现该目的。下一章将学习如何实现。

7.8 练习

1. 将下列方法加到Fraction类，以扩展关于分数的算术运算。在每个例子中都约简结果。

```
// Subtract argument from receiver
-(Fraction *) subtract: (Fraction *) f;
// Multiply receiver by argument
-(Fraction *) multiply: (Fraction *) f;
// Divide receiver by argument
-(Fraction *) divide: (Fraction *) f;
```

2. 从Fraction类中修改print方法，使之能够接受一个可选的BOOL参数，它表明是否应该约简该分数显示它。如果要约简它，一定不要对它进行永久更改。

3. 修改代码清单7-8，将结果sum显示为分数而不是实数。
4. Fraction类对负分数适用吗？例如−1/4和−1/2能得出正确结果吗？如果想出答案，编写测试程序进行尝试。
5. 修改Fraction类的print方法，以便显示比1大的分数，即5/3能显示为1 2/3。
6. 第4章的练习6定义了一个名为Complex的新类，它处理带虚数的复数。添加一个名为add:的方法，它可以用来使两个复数相加。要使两个复数相加，只需将它们的实部和虚部分别相加，如下所示：

```
(5.3 + 7i) + (2.7 + 4i) = 8 + 11i
```

根据以下方法声明，使add:方法存储并返回一个新的Complex值。

```
-(Complex *) add: (Complex *) complexNum;
```

一定要解决测试程序中任何可能的内存泄漏问题。

7. 给定第4章练习6的Complex类和该章在练习6中所作的扩展，创建Complex.h和Complex.m接口文件和实现文件。创建单独的测试程序文件来验证。

第8章 继 承

在本章将学到使面向对象编程如此强大的一个重要原理。通过继承这个概念，将学会如何使用现有的类定义并使其适于自己的应用程序。

8.1 一切从根类开始

在第3章“类、对象和方法”中学过父类的概念。父类自身也可以有父类。没有父类的类位于类层次结构的最顶层，称为根（root）类。在Objective-C中，允许定义自己的根类，但通常你不想这么做，而是想要利用现有的类。至此我们所定义的类都属于名为NSObject的根类的派生类，这个根类通常如下在接口文件中指定：

```
@interface Fraction: NSObject
...
@end
```

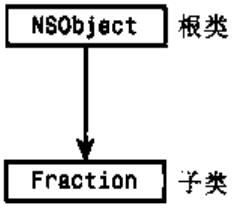

图8-1 根类和子类

类Fraction就是从类NSObject派生来的。因为NSObject是层次结构的最顶端（也就是，它上面没有任何类），因此称它为根类，如图8-1所示。类Fraction称为子或者子类（subclass）。

从术语的角度而言，可以将一个类称做子类和父类。相似地，还可以将类称作子类和超类。你应该熟悉这两种术语。

只要定义一个新类（不是一个新的根类），该类都会继承一些属性。例如，很明显父类的所有实例变量和方法都成为新类定义的一部分。这意味着子类可以直接访问这些方法和实例变量，就像直接在类定义中定义了这些子类一样。

举一个简单的例子，虽然不太自然，但有助于解释继承这个关键概念。

下面是一个名为ClassA的对象的声明，它有一个方法initVar：

```
@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
@end
```

initVar方法简单地把100赋值给ClassA的实例变量：

```
@implementation ClassA
-(void) initVar
{
    x = 100;
}
```

```
@end
```

现在，再定义一个名为ClassB的类：

```
@interface ClassB: ClassA
-(void) printVar;
@end
```

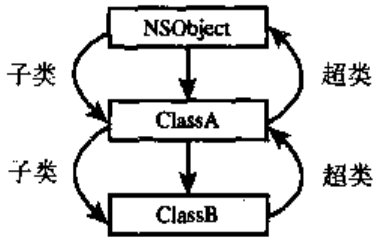

图8-2　子类及超类

声明的第一行

```
@interface ClassB: ClassA
```

说明ClassB并非NSObject的子类，而是ClassA的子类。所以，尽管ClassA的父类（或超类）是NSObject，但是ClassB的父类却是ClassA，如图8-2所示。

如图8-2所示，根类没有超类，而ClasssB，它位于继承的最底部，没有子类。因此，ClassA是NSObject的子类，而ClassB是ClassA的子类，也是NSObject的子类（从学术上讲，它是子类的子类或者孙类）。同样，NSObject是ClassA的超类，也是ClassB的超类。因为ClassB位于NSObject层次结构的下方，所以NSObject也是ClassB的超类。

下面是ClassB的完整声明，ClassB定义了一个名为printVar的方法：

```
@interface ClassB: ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
    NSLog (@"x = %i", x);
}
@end
```

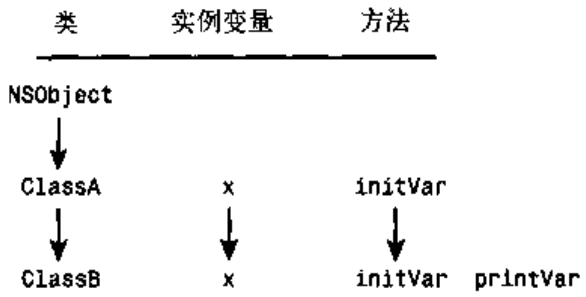

图8-3　继承实例变量和方法

虽然在ClassB中没有定义任何实例变量，但可以通过printVar方法输出实例变量x的值。这是由于ClassB是ClassA的子类，因此它继承ClassA的所有实例变量（在这个例子中，实例变量只有一个）。如图8-3中所示。

（当然，图8-3并没有显示从NSObject类所继承的任何方法或实例变量，虽然NSObject有几个方法及实例变量。）

通过把它集中在一个完整的程序例子中，看看它的工作方式。为了简洁起见，将所有类声名和定义放在单个文件中（参见代码清单8-1）。

代码清单8-1

```
// Simple example to illustrate inheritance

#import <Foundation/Foundation.h>

// ClassA declaration and definition
```

```
@interface ClassA: NSObject
{
   int  x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
  x = 100;
}
@end

// Class B declaration and definition

@interface ClassB : ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
  NSLog (@"x = %i", x);
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   ClassB  *b = [[ClassB alloc] init];

   [b initVar];      // will use inherited method
   [b printVar];     // reveal value of x;

   [b release];

   [pool drain];
   return 0;
}
```

代码清单8-1 输出结果

```
x = 100
```

首先，定义b为一个ClassB对象。在分配空间并初始化b后，发送一条消息，向它应用initvar方法。但是回过头来看看ClassB的定义，就会发现从未定义过这样的方法。事实上，initVar定义在ClassA中，由于ClassA是ClassB的父类，所以ClassB可以使用ClassA的所有方法。对于ClassB而言，initVar是继承来的方法。

注意　这里只简短地提到，但是alloc和init是你用过的从未在类中定义过的方法。这是因为你利用了它们都是继承方法的事实。

向b发送initVar消息之后，调用printVar方法显示实例变量x的值。输出结果x = 100证实printVar能够访问这个实例变量。这是因为和initVar方法一样，这个变量也是继承的。

记住，继承的概念作用于整个继承链。因此，如果如下所示定义一个父类是ClassB的新类ClassC：

```
@interface ClassC: ClassB;
   ...
@end
```

那么ClassC将继承ClassB的所有方法和实例变量，同时也依次继承ClassA的所有方法和实例变量，它依次集成NS Object的所有方法和实例变量。

一定要理解以下的事实：类的每个实例都拥有自己的实例变量，即使这些实例变量是继承来的。因此，对象ClassC与对象ClassB具有完全不同的实例变量。

找出正确的方法

向对象发送消息时，你可能想知道如何选择正确的方法来应用到该对象。规则其实很简单。首先，检查该对象所属的类，以查看在该类中是否明确定义了一个具有指定名称的方法。如果有，就使用这个方法。如果这里没有定义，就检查它的父类。如果父类中有定义，就用这个方法。否则，将继续寻找。

直到发现下面两种情况中的一种，才会检查父类：发现包含指定的方法的类，或者一直搜索到根类也没有发现任何方法。如果是第一种情况，就会停止查找；如果是第二种情况，说明存在问题，就会生成类似下面的警告消息：

```
warning: 'ClassB' may not respond to '-inity'
```

在这个例子中，你无意中向类ClassB传递了一条名为inity的消息。编译器告知你：该类型的类不能响应这种方法。同样，这是在检查ClassB的方法及直到根类（在此例中，根类为NSObject）的父类的方法之后确定的。

在某些情况下，如果没有发现该方法不会生成消息。这涉及所谓的传递的概念，这将在第9章“多态、动态类型和动态绑定”中简要地讨论。

8.2　通过继承扩展——添加新方法

继承通常用于扩展一个类。举个例子，假设你刚接受一项任务：开发一些处理2D图形对象（如：矩形、圆形和三角形）的类。现在，只考虑矩形。实际上，回顾第4章“数据类型和表达

式”的练习7，从以下例子开始@interface部分：

```
@interface Rectangle: NSObject
{
    int width;
    int height;
}

@property int width, height;
-(int) area;
-(int) perimeter;
@end
```

当前的方法可以设置矩形的宽与高，返回它们的值并计算其面积与周长。再添加一个方法，它允许你使用相同的消息调用来设置矩形的宽度与长度值，如下：

```
-(void) setWidth: (int) w andHeight: (int) h;
```

假设将新类声明键入名为Rectangle.h的文件。实现文件Rectangle.m将如下面所示：

```
#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h
{
    width = w;
    heighe = h;
}

-(int) area
{
    return width * height;
}

-(int) perimeter
{
    return (width + height) * 2;
}
@end
```

每个方法定义都很直观。代码清单8-2显示了一个测试它的main例程。

代码清单8-2

```
#import "Rectangle.h"

int main (int argc, char *argv[])
{
```

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

    Rectangle *myRect = [[Rectangle alloc] init];

    [myRect setWidth: 5 andHeight: 8];

    NSLog (@ "Rectangle:w = %i, h = %i",
            myRect.width, myRect.height);
    NSLog (@"Area = %i, Perimeter = %i",
              [myRect area], [myRect perimeter]);
    [myRect release];

    [pool drain];
    return 0;
}
```

代码清单8-2　输出结果

```
Rectangle: w = 5, h = 8
Area = 40, Perimeter = 26
```

给myRect分配内存并将其初始化，然后将其宽设为5、高设为8。输出的第一行验证了这一点。然后，使用合适的消息调用，计算矩形的面积和周长，返回值由NSLog进行显示。

处理矩形之后，假设现在需要处理正方形。可以定义一个名为Square的新类，并在其中定义同Rectangle类相似的方法。或者，认识到正方形只是长方形的特例，长和宽恰好相同的长方形。

因此，简单的处理方法就是定义一个名为Square的新类，并使它成为Rectangle的子类。这样除了定义自己的方法和变量之外，可以使用Rectangle类中的所有方法和变量。现在，可能要添加的唯一方法可能是将正方形的边设置特定的值，并检索该值。代码清单8-3显示了Square类的接口文件和实现文件。

代码清单8-3　Square.h接口文件

```
#import "Rectangle.h"

@interface Square: Rectangle

-(void) setSide: (int) s;
-(int) side;
@end
```

代码清单8-3　Square.m实现文件

```
#import "Square.h"

@implementation Square: Rectangle;
```

```
-(void) setSide: (int) s
{
    [self setWidth: s andHeight: s];
}

-(int) side
{
    return width;
}
@end
```

注意此处所做的工作。你将Square定义为Rectangle的子类，这是在头文件Rectangle.h中声明的。这里不必添加任何实例变量，但是添加了两个名为setSide和side的新方法。

虽然正方形只有一个边，但在内部也可用两个数表示，这样也可以。这对于Square类的用户是隐藏的。如果需要，以后随时可以重新定义Square类。根据前面介绍的封装概念，任何用户都无须担心内部的细节问题。

setSide:方法利用从Rectangle类继承的方法设定矩形宽度与长度的值。所以setSide:调用Rectangle类的setWidth:andHeight:方法，同时传递参数作为宽度与长度值。实际上不必再进行其他任何操作。处理Square对象的人通过利用setSide:方法，可以设置正方形的大小，并利用Rectangle类的方法计算正方形的面积、周长，等等。代码清单8-3展示了新的Square类的测试程序和输出。

代码清单8-3 测试程序

```
#import "Square.h"
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Square *mySquare = [[Square alloc] init];

   [mySquare setSide: 5];

   NSLog (@"Square s = %i", [mySquare side]);
   NSLog (@"Area = %i, Perimeter = %i",
       [mySquare area], [mySquare perimeter]);
   [mySquare release];

   [pool drain];
   return 0;
}
```

代码清单8-3　输出结果

```
Square s = 5
Area = 25, Perimeter = 20
```

定义Square类的方式是在Objective-C中使用类的基本技巧：扩展自己或其他人以前实现的类，使其适合自己的需要。另外，所谓的分类（category）机制允许你以模块方式向现有类定义添加新方法，也就是，不必经常给同一接口和实现文件增加新定义。当想要对你没有源代码访问权限的类添加新定义时，这特别方便。在第11章“分类和协议”中将学到分类。

8.2.1　Point类和内存分配

Rectangle类只存储矩形大小。在实际的图形应用中，可能需要保存各种附加消息，如：矩形的填充色、线条颜色、窗口中的位置（原点），等等。可以方便地扩展这些类来处理这些情况。现在，处理矩形原点的概念。假设“原点”是指笛卡尔坐标系（x, y）中矩形左下角的位置。如果正在编写绘图应用程序，这一点可能代表矩形在窗口中的位置。如图8-4所示。

在图8-4中，矩形的原点是(x_1, y_1)。

可以扩展Rectangle类，将矩形原点（x, y）保存为两个不同的值。或许你已经认识到在开发图形应用程序的过程中，要处理很多坐标，因此要定义一个名为XYPoint的类（在第3章的练习7中出现过这个问题）：

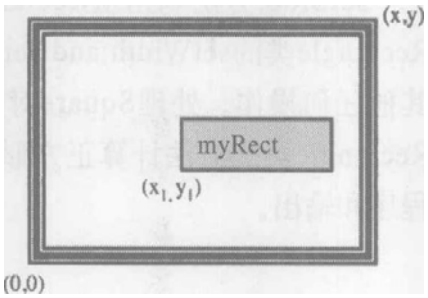

图8-4　窗口中绘制的矩形

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

@interface XYPoint: NSObject
{
   int x;
   int y;
}
@property int x, y;

-(void) setX: (int) xVal andY: (int) yVal;
@end
```

现在，回到Rectangle类。希望能够存储矩形的原点，所以，必须向Rectangle类的定义添加另一实例变量origin：

```
@interface Rectangle: NSObject
{
   int width;
   int height;
   XYPoint *origin;
}
      ...
```

应该再添加一个方法，设置矩形的原点并对其进行检索。为突出重点，我们这里不综合原

点的存取器方法，我们将自行编写。

8.2.2 @class指令

现在可以设置矩形（或正方形）的宽、高及原点。首先，完整地看一下接口文件Rectangle.h：

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

@class XYPoint;
@interface Rectangle: NSObject
{
   int width;
   int height;
   XYPoint *origin;
}

@property int width, height;

-(XYPoint *) origin;
-(void) setOrigin: (XYPoint *) pt;
-(void) setWidth: (int) w andHeight: (int) h
-(int)  area;
-(int)  perimeter;
@end
```

在Rectangle.h头文件中使用一个新的指令：

```
@class XYPoint;
```

这是因为编译器在遇到为Rectangle定义的实例变量XYPoint时，必须了解XYPoint是什么。类名还分别用在setOrigin:和origin方法的参数及返回类型声明中。你还有另一个选择。可以如下所示导入头文件来代替它：

```
#import "XYPoint.h"
```

使用@class指令提高了效率，因为编译器不需要处理整个XYPoint.h文件（虽然它很小），而只需知道XYPoint是一个类名字。如果需要引用XYPoint类中方法，@class指令是不够的，因为编译器需要更多消息。它需要知道该方法中有多少参数、它们是什么类型、方法的返回类型是什么。

下面填充新的XYPoint类和Rectangle方法的空白，这样就可以在一个程序中测试所有内容。首先，代码清单8-4显示了XYPoint类的实现文件。

代码清单8-4显示了Rectangle类的新方法。

代码清单8-4 Rectangle.m添加的方法

```
#import "XYPoint.h"

-(void) setOrigin: (XYPoint *) pt
```

```
{
   origin = pt;
}

-(XYPoint *) origin
{
    return origin;
}
@end
```

以下是完整的XYPoint和Rectangle类定义，接着是测试程序。

代码清单8-4　XYPoint.h接口文件

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

@interface XYPoint: NSObject
{
   int x;
   int y;
}
@property int x, y;

-(void) setX: (int) xVal andY: (int) yVal;
@end
```

代码清单8-4　XYPoint.m实现文件

```
#import "XYPoint.h"

@implementation XYPoint

@synthesize x, y;
-(void) setX: (int) xVal andY: (int) yVal
{
   x = xVal;
   y = yVal;
}
@end
```

代码清单8-4　Rectangle.h接口文件

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

@class XYPoint;
@interface Rectangle: NSObject
{
   int width;
```

```
    int height;
    XYPoint *origin;
}

@property int width, height;

-(XYPoint *) origin;
-(void) setOrigin: (XYPoint *) pt;
-(void) setWidth: (int) w andHeight: (int) h;
-(int)  area;
-(int)  perimeter;
@end
```

代码清单8-4 Rectangle.m实现文件

```
#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h
{
    width = w;
    height = h;
}

-(void) setOrigin: (XYPoint *) pt
{
    origin = pt;
}

-(int) area
{
    return width * height;
}

-(int) perimeter
{
    return (width + height) * 2;
}

-(XYPoint *) origin
{
    return origin;
}
@end
```

代码清单8-4 测试程序

```
#import "Rectangle.h"
#import "XYPoint.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Rectangle *myRect = [[Rectangle alloc] init];
   XYPoint *myPoint = [[XYPoint alloc] init];

   [myPoint setX: 100 andY: 200];

   [myRect setWidth: 5 andHeight: 8];
   myRect.origin = myPoint;

   NSLog (@"Rectangle w = %i, h = %i",
      myRect.width, myRect.height);

   NSLog (@"Origin at (%i, %i)",
      myRect.origin.x, myRect.origin.y);

   NSLog (@"Area = %i, Perimeter = %i",
      [myRect area], [myRect perimeter]);
      [myRect release];
   [myPoint release];

   [pool drain];
   return 0;
}
```

代码清单8-4 输出结果

```
Rectangle w = 5, h = 8
Origin at (100, 200)
Area = 40, Perimeter = 26
```

在main例程中，为名为myRect的矩形和名为myPoint的点分配了空间并进行初始化。使用方法setX:和Y:，将myPoint值设置为(100, 200)。在分别将矩形的宽和长设置为5和8之后，调用setOrigin方法，将矩形原点设置为myPoint中指定的点。通过三个NSlog方法调用检索并输出值。表达式

```
myRect.origin.x
```

接收存取器方法origin方法返回的XYPoint对象，并对其应用方法x来取得矩形原点的x坐标。以同样的方式，表达式

```
myRect.origin.y
```

检索矩形原点的y坐标。

8.2.3 具有对象的类

能够解释代码清单8-5的输出结果吗？

代码清单8-5

```
#import "Rectangle.h"
#import "XYPoint.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Rectangle *myRect = [[Rectangle alloc] init];
   XYPoint *myPoint = [[XYPoint alloc] init];
   [myPoint setX: 100 andY: 200];

   [myRect setWidth: 5 andHeight: 8];
   myRect.origin = myPoint;

   NSLog (@"Origin at (%i, %i)",
      myRect.origin.x, myRect.origin.y);
   [myPoint setX: 50 andY: 50];
   NSLog (@"Origin at (%i, %i)",
      myRect.origin.x, myRect.origin.y);
   [myRect release];
   [myPoint release];

   [pool drain];
   return 0;
}
```

代码清单8-5 输出结果

```
Origin at (100, 200)
Origin at (50, 50)
```

将XYPoint myPoint从原程序中的(100, 200)改为(50, 50)，显然也改变了矩形的原点！但是为什么出现这种情况？你并没有显式地使用另一个setOrigin:方法调用重新设置矩形的原点，所以，为什么矩形的原点会改变？如果回顾setOrigin:方法的定义，也许就会明白：

```
-(void) setOrigin: (XYPoint *) pt
{
   origin = pt;
}
```

当使用以下表达式调用setOrigin:方法时，

```
myRect.Origin = myPoint;
```

myPoint的值做为参数传递给该方法。这个值指向存储XYPoint对象的内存，如图8-5所示。

存储在myPoint（它是一个指向内存的指针）中的值被复制到在该方法中定义的本地变量pt中。现在pt与myPoint都引用内存中相同的数据。如图8-6所示。

在方法中将origin变量设置为pt时，pt中存储的指针被复制到实例变量origin，如图8-7所示。

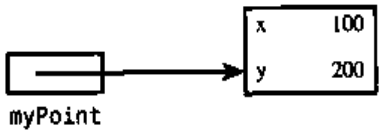

图8-5　内存中的XYPoint myPoint

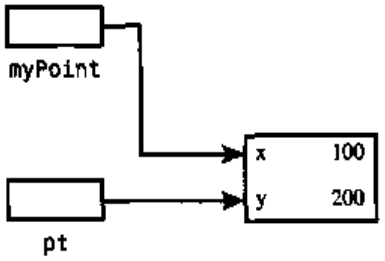

图8-6　将矩形原点传递给该方法

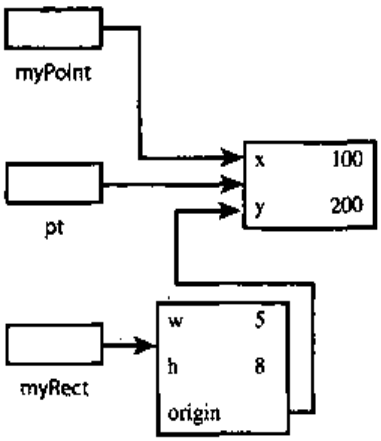

图8-7　设置矩形原点

因为myPoint和存储在myRect中的origin变量引用内存中的同一区域（与局部变量pt相同），所以随后将myPoint的值改为(50, 50)时，矩形的原点也被更改。

避免这一问题的办法就是修改setOrigin:方法，以便分配自己的点并将原点设置为该点，如下所示：

```
-(void) setOrigin: (XYPoint *) pt
{
   origin = [[XYPoint alloc] init];

   [origin setX: pt.x andY: pt.y];
}
```

这个方法首次分配并初始化一个新XYPoint。消息表达式

```
[origin setX: pt x andY: pt.y];
```

将新分配的XYPoint设置为该方法参数的x、y坐标。研究该消息表达式，直到完全了解它的工作原理。

setOrigin:方法的改变意味着每个Rectangle实例现在都有它的origin XYPoint实例。既然它负责为XYPoint分配内存，所以还应该负责释放该内存。通常，当一个类包含其他对象时，有时你希望拥有部分或全部对象。以矩形为例，它有自己的原点很合理，因为这是矩形的基本属性。

但是，如何释放origin占用的内存呢？释放矩形内存，并不同时释放为原点分配的内存。释放内存的一种方式是在main中插入以下一行：

```
[[myRect origin] release];
```

这样可以释放由原点方法返回的XYPoint对象。但必须在Rectangle对象释放自己的内存之前释放XYPoint对象，因为在对象释放内存后它包含的所有实例都是无效的。因此，正确的代码顺序应该如下所示：

```
[[myRect origin] release]; // Release the origin's memory
[myRect release];          // Release the rectangle's memory
```

要记住释放原点的内存是个负担。毕竟，不是你为原点分配的内存，而是由Rectangle类分配的。在下一节“重载方法”中，将学习如何释放Rectangle的内存。

使用修改后的方法，重新编辑和运行代码清单8-5，生成以下出错消息，如图8-8所示。

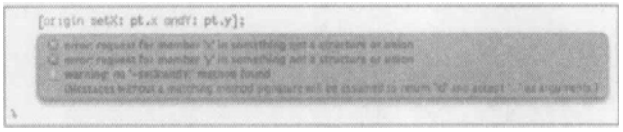

图8-8 编译器出错消息

糟了，这么多问题！这里的问题是由于在修改后的方法中使用了XYPoint类的一些方法，所以，现在编译器需要的信息多于@class指令提供的。在这个例子中，返回并用import代替这个指令，如：

```
#import "XYPoint.h"
```

代码清单8-5 输出结果

```
Origin at (100, 200)
Origin at (100, 200)
```

这样就好多了！这次在main例程中将myPoint的值更改为(50, 50)对矩形原点没有影响，因为在矩形的setOrigin:方法中创建了该点的副本。

顺便说一下，这里我们没有合成origin方法，因为合成的setter setOrigin:方法的行为将与你之前编写的那个方法一样。也就是说，在默认情况下，合成setter只是简单地复制对象指针而不是对象本身。

你可以合成另一种setter方法，而不是制作对象的副本。但是，如果那样做，就需要学习如何编写特殊的复制方法。第17章“内存管理”将再次讨论这一主题。

8.3 重载方法

前面一节提到过，不能通过继承删除或减少方法。但可以利用重载来更改继承方法的定义。

回头看这两个类：ClassA与ClassB。假定要为ClassB编写自己的initVar方法。你已经知道ClassB将继承定义在ClassA中的initVar方法，但是否可以新建一个同名的方法来替代继承的方法呢？答案是可以，只要定义一个同名的新方法即可。使用和父类相同的名称定义的方法代替或重载了继承的定义。新方法必须具有相同的返回类型，并且参数的数目与重载的方法相同。

代码清单8-6展示了简单的例子来说明这个概念。

代码清单8-6

```
// Overriding Methods

#import <Foundation/Foundation.h>

// ClassA declaration and definition

@interface ClassA: NSObject
{
   int x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
    x = 100;
}
@end

// ClassB declaration and definition

@interface ClassB: ClassA
-(void) initVar;
-(void) printVar;
@end

@implementation ClassB
-(void) initVar    // added method
{
   x = 200;
}

-(void) printVar
{
   NSLog (@"x = %i", x);
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc]] int];

   ClassB *b = [[ClassB alloc] init];
```

```
    [b initVar]; // uses overriding method in B

    [b printVar]; // reveal value of x;
    [b release];

    [pool drain];
    return 0;
}
```

代码清单8-6 输出

```
x = 200
```

显然消息

```
[b initVar];
```

导致使用定义在ClassB中的initVar方法，而不是使用ClassA中所定义的方法，前一示例也是如此。如图8-9所示。

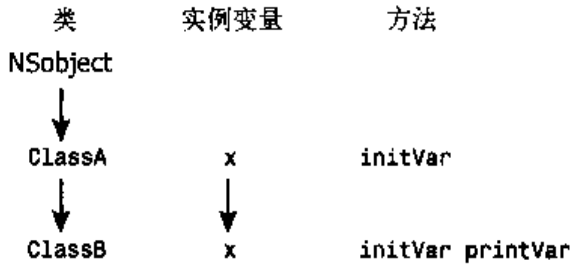

图8-9 重载initVar方法

8.3.1 择哪个方法

前面曾讲到系统如何上溯类层次查找应用于对象的方法。如果在不同的类中有名称相同的方法，则根据作为消息的接收者的类选择正确的方法。代码清单8-7使用与前面的ClassA和ClassB相同的类定义。

代码清单8-7

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

// insert definitions for ClassA and ClassB here

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    ClassA *a = [[ClassA alloc] init];
    ClassB *b = [[ClassB alloc] init];
```

```
    [a initVar]; // uses ClassA method
    [a printVar]; // reveal value of x;

    [b initVar]; // use overriding ClassB method
    [b printVar]; // reveal value of x;
    [a release];
    [b release];

    [pool drain];
    return 0;
}
```

编译该程序时会得到以下警告消息：

```
warning: 'ClassA' may not respond to '-printVar'
```

这里有什么问题？前一节讨论过此问题。观察ClassA的声明：

```
// ClassA declaration and definition

@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
@end
```

注意没有声明printVar方法。该方法声明并定义在ClassB中。因此，尽管ClassB对象及其派生类可以通过继承使用此方法，但是ClassA对象却不能使用此方法，这是由于此方法是沿着类层次定义的。

注意 你可以通过某种方式强制使用此方法，但是我们这里不对此进行讨论，此外，这也不是一个好的编程实践。

回到例子，为ClassA添加一个printVar方法，以便显示实例变量的值：

```
// ClassA declaration and definition

@interface ClassA: NSObject
{
    int x;
}

-(void) initVar;
-(void) printVar;
@end

@implementation ClassA
-(void) initVar
```

```
{
   x = 100;
}

-(void) printVar
{
   NSLog (@"x = %i", x);
}


@end
```

ClassB的声明与定义保持不变。现在，再次尝试编译并运行该程序。

代码清单8-7 输出结果

```
x = 100
x = 200
```

现在可以讨论这个实际例子。首先，分别将a和b定义为ClassA及ClassB对象。经过内存分配与初始化后，向a发送一条消息，让它应用initVar方法。此方法在ClassA中定义，所以，它是所选的方法。此方法只是将实例变量值设为100然后返回。然后调用刚刚添加到ClassA中的printVar方法来显示的x值。

ClassB 对象b类似，经过内存分配及初始化后，将实例变量x设为200，最后显示其值。

一定理解如何按照a与b所属的类选择相应的方法。这是Objective-C中面向对象编程的基础概念。

作为练习，考虑从ClassB中删除printVar方法。可行吗？为什么可行，或为什么不可行？

8.3.2 重载dealloc方法和关键字super

既然知道如何重载方法后，那么返回代码清单8-5学习释放origin所占内存的更好办法。setOrigin:方法现在为自己的XYPoint origin对象分配内存，并且你负责释放它的内存。代码清单8-6中使用的方法就是使用以下语句让main释放该内存：

```
[[myRect origin] realease];
```

所以不必担心释放所有单独的类成员，可以重载继承的dealloc方法（它是从NSObject继承的）并在其中释放origin的内存。

注意 你不重载release方法，而是重载dealloc方法。在后续章节中你将了解，release有时释放对象使用的内存，有时却不。只有在其他人引用某个对象时，release才释放该对象所占用的内存。这通过调用该对象的dealloc方法来完成，实际上是由dealloc来释放内存。

如果决定重载dealloc方法，必须确保不仅要释放自己的实例变量所占用的内存，而且释放继承的变量所占的内存。

为此，需要利用关键字super，它引用消息接收者的父类。可以向super传递消息来执行重

载的方法。这就是此关键字最常见的用途。所以，在方法内部使用消息表达式

```
[super release];
```

时，调用定义在父类中的（或是父类继承的）release方法。此方法是对消息的接收者调用的，换言之，是对self调用。

因此，为Rectangle类重载dealloc方法的策略是，首先要释放origin所占的内存，然后调用父类的dealloc方法完成这项任务。这就释放了Rectangle对象自身所占的内存。下面是这个新方法：

```
-(void) dealloc
{
   if (origin)
      [origin release];
   [super dealloc];
}
```

定义dealloc方法没有返回值。通过查看头文件<NSObject.h>可以了解这点。在dealloc方法中，进行一项测试来查看释放origin之前它是否非零。很可能从未设置矩形的原点，这种情况下，它拥有默认值零。然后调用父类的dealloc方法，如果没有重载，则Rectangle类将继承此方法。

应该指出，还可以如下更简单地编写这个dealloc方法：

```
(void) dealloc
{
   [origin release];
   [super dealloc];
}
```

因为可以向nil对象发送消息。此外，要注意此时是release原点，而不是dealloc它。如果没有其他人使用原点，release就会在原点调用dealloc方法来释放它的空间。

有了这个新方法，现在只需释放分配了内存的矩形，而无须担心其中包含的XYPoint对象。代码清单8-5所示的两条release消息：

```
[myRect release];
[myPoint release];
```

足以释放在程序中分配内存的所有对象，包括setOrigin所创建的XYPoint对象。

还有一个问题：如果在程序的执行期间将单个Rectangle对象的原点设置为不同的值，那么在分配和指定新的原点之前，必须释放旧原点所占的内存。例如，在下面的代码序列中：

```
myRect.Origin= startPoint;
   ...
myRect.Origin= endPoint;
   ...
[startPoint release];
[endPoint release];
[myRect release];
```

XYPoint startPoint的副本保存在myRect的成员origin中，它从未被释放，这是因为它被存储在

其中的第二个原点（endPoint）所重写了。根据新的release方法，只当释放矩形自身时才能释放原点。

需要确保在设置矩形的新原点之前释放旧的原点。这可以在setOrigin:方法中如下进行处理：

```
-(void) setOrigin: (XYPoint *) pt
{
   if (origin)
     [origin release];

   origin = [[XYPoint alloc] init];

   [origin setX: pt.x andY: pt.y];
}
```

幸运的是，在合成存取器方法时，还可以让编译器自动处理这类问题。

8.4 通过继承扩展：添加新的实例变量

你不仅可以添加新方法来有效地扩展类定义，还可以添加新的实例变量。以上两种情况的影响是累加的。不能通过继承减少方法或实例变量，只能添加，对于方法来说，可以添加或者重载。

回到简单的ClassA和ClassB类并做些修改。可以如下向ClassB添加一个新实例变量：

```
@interface ClassB: ClassA
{
   int y;
}

-(void) printVar;
@end
```

尽管根据前面的声明，ClassB看起来可能只有一个实例变量y，它实际上可能有2个：从ClassA继承的变量x加上自己的实例变量y。

注意 当然它还包含从NSObject类继承的实例变量，但暂时不管这些细节。

把它们放到一个简单的例子中来说明这个概念（参见代码清单8-8）。

代码清单8-8

```
// Extension of instance variables

#import <Foundation/Foundation.h>

// Class A declaration and definition

@interface ClassA: NSObject
{
  int  x;
```

```
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
  x = 100;
}
@end

// ClassB declaration and definition

@interface ClassB: ClassA
{
  int  y;
}
-(void)  initVar;
-(void)  printVar;
@end

@implementation ClassB
-(void) initVar
{
   x = 200;
   y = 300;
}

-(void) printVar
{
   NSLog (@"x = %i", x);
   NSLog (@"y = %i", y);
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    ClassB *b = [[ClassB alloc] init];

    [b initVar];  // uses overriding method in ClassB
    [b printVar];  // reveal values of x and y;

    [b release];
    [pool drain];
```

```
    return 0;
}
```

代码清单8-8 输出结果

```
x = 200
y = 300
```

ClassB对象b通过调用定义在ClassB中的initVar方法进行初始化。回忆这个方法重载了ClassA的initVar方法。该方法还将x（它是从ClassA继承来的）设为200，将y（在ClassB中定义）设为300。然后，使用printVar方法显示这两个实例变量的值。

选择正确的方法以响应一条消息涉及许多微妙之处，特别是接收者可以是几个类之一。这是个很强大的概念，名为动态绑定。下一章将介绍这个主题。

8.5 抽象类

除了术语外还有什么更好的办法总结本章内容呢？这里介绍这个概念，是因为它与继承的概念直接相关。

有时，创建类只是使创建子类更容易。因此，这些类名为抽象（abstract）类，或等价地称作抽象超类（abstract superclasses）。方法和实例变量定义在该类中，但不期望任何人从该类创建实例。例如，考虑根对象Object。能想出从该类定义对象的任何用途吗？

Foundation框架将在第二部分讲述，它包含几个所谓的抽象类。举一个例子，Foundation的NSNumber类是为了将数字作为对象处理而创建的抽象类。整数与浮点数字通常有不同的内存需求。每种数字类型都有单独的NSNumber子类。因为这些子类与它们的抽象超类不同，这些子类是具体存在的，它们名为具体子类。每个具体子类属于NSNumber类，总起来名为簇（cluster）。向NSNumber类发送消息来创建新的整数对象时，使用合适的子类为整数对象分配必需的空间，并正确地设定其值。这些子类实际上是私有的。你自己无法直接访问这些子类，只能通过抽象的超类间接访问。抽象超类提供了处理所有数字对象类型的公共接口，你就无须了解存储在数字对象中的数字类型及如何设置与检索该值。

的确，这个讨论看起来可能有些“抽象”（报歉！），不用担心，这里只需掌握基础的概念就可以了。

8.6 练习

1. 向代码清单8-1添加一个名为ClassC的新类，它是ClassB的子类。创建一个initVar方法，它将实例变量x的值设置为300。编写一个测试例程，它声明对象ClassA、ClassB及ClassC，并且调用相应initVar方法。
2. 使用高分辨率的设备时，可能需要使用允许将点指定为浮点值（而不是简单的整数）的坐标系。修改本章的XYPoint与Rectangle类，以处理浮点数字。矩形的宽度、高度、面积与周长也都使用浮点数字进行处理。

3. 修改代码清单8-1，向其添一个名为ClassB2的新类，ClassB2和ClassB一样，都是ClassA的子类。

   ClassB与ClassB2之间有什么关系？

   指出NSObject类、ClassA、ClassB及ClassB2之间的层次关系。

   ClassB的超类是什么？

   ClassB2的超类是什么？

   一个类可以有多少个子类、多少个超类？

4. 编写一个名为translate的Rectangle方法。它使用向量XYPoint (xv,yv)为参数。通过指定的向量平移矩形的原点。

5. 定义一个名为GraphicObject的新类，使其成为NSObject的子类。在新类中定义如下一些实例变量：

```
int  fillColor;   // 32-bit color
BOOL filled;      // Is the object filled?
int  lineColor;   // 32-bit line color
```

   编写一个方法，设定并检索前面定义的变量。

   使Rectangle类成为GraphicObject的子类。

   定义两个新类Circle和Triangle，它们都是GraphicObject的子类。编写一些方法来设定及检索这些对象的各种参数，并计算圆的圆周、面积及三角形的周长、面积。

6. 编写一个名为intersect:的Rectangle方法。它使用一个矩形作为参数，并返回代表两个矩形的重叠区域。例如，给定图8-10所展示的两个矩形，这个方法应该返回原点位于(400,420)的矩形，其宽度为50、高度为60。如果矩形没有相交，返回宽度与高度均为零的矩形，其原点在(0,0)。

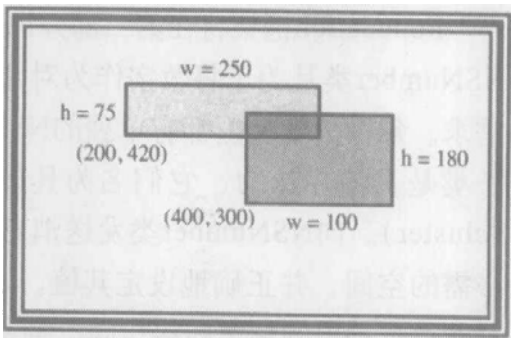

图8-10　相交矩形

7. 为Rectangle类编写一个名为draw的方法，此方法使用虚线与垂直的条形字符绘制矩形。以下代码序列

```
Rectangle *myRect = [[Rectangle alloc] init];
[myRect setWidth: 10 andHeight: 3];
[myRect draw];
[myRect realease];
```

   将产生如下的输出结果：

```
----------
|        |
|        |
|        |
----------
```

注意：应该使用printf绘制字符，因为每次调用NSlog时它都会显示一个新行。

第 9 章　多态、动态类型和动态绑定

本章你将了解到Objective-C语言的一些特性。这些特性使它成为一门功能强大的编程语言，并且使其有别于其他面向对象的程序设计语言，如C++。本章中讲述了三个关键概念：多态、动态类型和动态绑定。多态使得能够开发以下程序：来自不同类的对象可以定义共享相同名称的方法。动态类型能使程序直到执行时才确定对象所属的类；动态绑定则能使程序直到执行时才确定要对对象调用的实际方法。

9.1　多态：相同的名称，不同的类

代码清单9-1显示了名为Complex的类的接口文件，它用于表示程序中的复数。

代码清单9-1　接口文件Complex.h

```
// Interface file for Complex class

#import <Foundation/Foundation.h>

@interface Complex: NSObject
{
   double real;
   double imaginary;
}

@property double real, imaginary;
-(void)   print;
-(void)   setReal: (double) a andImaginary: (double) b;
-(Complex *) add: (Complex *) f;
@end
```

你应该已经完成了第7章的练习6和练习7。我们在那个练习中加入一个补充的setReal:andImaginary:方法，使用它就可以用一条消息和合成存取器方法设置数字的实数和虚数部分，如下所示。

代码清单9-1　实现文件Complex.m

```
// Implementation file for Complex class

#import "Complex.h"

@implementation Complex

@synthesize real, imaginary;
```

```
-(void) print
{
   NSLog (@"%g + %gi ", real, imaginary);
}

-(void) setReal: (double) a andImaginary: (double) b
{
   real = a;
   imaginary = b;
}

-(Complex *) add: (Complex *) f
{
   Complex *result = [[Complex alloc] init];

   [result setReal: real + [f real]
          andImaginary: imaginary + [f imaginary]];

   return result;
}
@end
```

代码清单9-1 测试程序main.m

```
// Shared Method Names: Polymorphism

#import "Fraction.h"
#import "Complex.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction *f1 = [[Fraction alloc] init];
   Fraction *f2 = [[Fraction alloc] init];
   Fraction *fracResult;
   Complex *c1 = [[Complex alloc] init];
   Complex *c2 = [[Complex alloc] init];
   Complex *compResult;

   [f1 setTo: 1 over: 10];
   [f2 setTo: 2 over: 15];

   [c1 setReal: 18.0 andImaginary: 2.5];
   [c2 setReal: -5.0 andImaginary: 3.2];

   // add and print 2 complex numbers
```

```
    [c1 print]; NSLog (@"+"); [c2 print];
    NSLog (@"---------");
    compResult = [c1 add: c2];
    [compResult print];
    NSLog (@"\n");

    [c1 release];
    [c2 release];
    [compResult release];

    // add and print 2 fractions
    [f1 print]; NSLog (@"+"); [f2 print];
    NSLog (@"----");
    fracResult = [f1 add: f2];
    [fracResult print];

    [f1 release];
    [f2 release];
    [fracResult release];

    [pool drain];
    return 0;
}
```

代码清单9-1 输出

```
18 + 2.5i
       +
-5 + 3.2i
---------
13 + 5.7i

1/10
   +
2/15
----
7/30
```

注意，Fraction和Complex类都包含add:和print方法。所以，当执行以下消息表达式

```
compResult = [c1 add: c2];
[compResult print];
```

时，系统如何知道执行哪个方法呢？其实很简单：Objective-C运行时知道第一条消息的接收者c1是一个Complex对象。因此，选择定义在Complex类中的add:方法。

Objective-C的运行时系统还确定compResult是一个Complex的对象。因而，它选择定义在complex类中的print方法来显示加法的结果。同样的论述也适用于以下消息表达式：

```
fracResult = [f1 add: f2];
[fracResult print];
```

注意 第13章将更加详尽描述，系统总是携带有关“一个对象属于哪个类”这样的信息。这信息能使系统在运行时而不是在编译时做出这些关键性的决定。

选择Fraction类中相应的方法来计算基于f1和fracResult类的消息表达式。

前面提到过，使不同的类共享相同方法名称的能力称为多态。它能使你开发一组类，这组类中的每一个类都能响应相同的方法名。每个类的定义都封装了响应特定方法所需的代码，这就使得它独立于其他的类定义。多态还允许你后来添加新类，这些新类能响应相同的方法名。

注意 在结束本节之前，注意Fraction类和Complex类应该负责释放它们add:方法而不是测试程序生成的结果。事实上，这些对象应该被自动释放。我们将在第17章对此进行更加详细的讨论。

9.2 动态绑定和id类型

第4章简略谈到了id数据类型，并指出这是一种通用的对象类型。也就是，它可以用来存储属于任何类的对象。当以这种方式在一个变量中存储不同类型的对象时，在程序的执行期间，这种数据类型的真正优势就出现了。研究代码清单9-2和它对应的输出。

代码清单9-2

```
// Illustrate Dynamic Typing and Binding

#import "Fraction.h"
#import "Complex.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   id dataValue;
   Fraction *f1 = [[Fraction alloc] init];
   Complex  *c1 = [[Complex alloc] init];

   [f1 setTo: 2 over: 5];
   [c1 setReal: 10.0 andImaginary: 2.5];

   // first dataValue gets a fraction

   dataValue = f1;
   [dataValue print];

   // now dataValue gets a complex number
```

```
    dataValue = c1;
    [dataValue print];

    [c1 release];
    [f1 release];

    [pool drain];
    return 0;
}
```

代码清单9-2 输出

```
2/5
10 + 2.5i
```

变量dataValue被声明为id对象类型。因此，dataValue可用来保存程序中任何类型的对象。务必注意，声明中并没有使用星号：

```
id dataValue;
```

Fraction f1被设为2/5，并且Complex数C1被设为(10 + 2.5i)。赋值语句

```
dataValue = f1;
```

将Fraction f1存储到dataValue中。现在，使用datavalue可以做什么呢？其实，可以对dataValue调用可以用于Fraction对象的任何方法，即使dataValue是一个id类型，不是Fraction。然而，如果dataValue可以存储任何类型的对象，那么系统如何知道该调用哪一个方法呢？就是说，当系统遇到消息表达式

```
[dataValue print];
```

时，如何知道该调用哪个print方法呢？你知道，Fraction和Complex类中都定义有print方法。

前面提到，答案就在于以下事实：Objective-C系统总是跟踪对象所属的类。答案同样存在于动态类型和动态绑定的概念之中，就是说，先判定对象所属的类，并因此确定运行时而不是编译时需要动态调用的方法。

因此，在程序执行期间，当系统准备将print消息发送给dataValue时，它首先检查dataValue中存储的对象所属的类。在代码清单9-2的第一种情况中，这个变量保存一个Fraction，因此使用在Fraction类中定义的print方法。这一点使用程序的输出即可验证。

在第二种情况下，发生了同样的事情。首先，Complex数c1被指派给dataValue。接下来，执行以下消息表达式

```
[dataValue print];
```

这次，因为dataValue包含属于complex的对象，所以选择该类相应的print方法来执行。

这是一个简单的例子，但是我认为你可以将这个概念推广到更复杂的应用。结合多态、动态绑定和动态类型时，你就能轻易地编写出可向来自不同类的对象发送相同消息的代码。

例如，考虑可用来在屏幕上绘制图形对象的draw方法。你可能为每个图形对象（比如文本、

圆、矩形、窗口，等等）定义了不同的draw方法。如果要绘制的特定对象存储在一个名为currentObject的id变量中，仅仅通过发送draw消息就可以在屏幕上进行绘制，如下所示：

```
[currentObject draw];
```

甚至可以首先测试它，确保存储在currentObject中的对象的确响应draw方法。在本章后面将学到如何这样做。

9.3 编译时和运行时检查

因为存储在id变量中的对象类型在编译时无法确定，所以一些测试推迟到运行时进行，就是说，推迟到程序执行时。

考虑下列代码序列：

```
Fraction *f1 = [[Fraction alloc] init];
[f1 setReal: 10.0 andImaginary: 2.5];
```

回顾一下，setReal:andImaginary: 方法应用于复数而不是分数，当编译包含这些语句的程序时，会显示以下的消息

```
prog3.m: In function 'main':
prog3.m:13: warning: 'Fraction' does not respond to 'setReal:andImaginary:'
```

Objective-C编译器知道f1是Fraction的对象，因为它就是这样声明的。编译器同样知道，当遇到消息表达式

```
[f1 setReal: 10.0 andImaginary: 2.5];
```

时，Fraction类并不包含setReal:andImaginary:方法（并且也没有继承该方法），所以生成前面所示的警告消息。

现在考虑下面的代码序列：

```
id dataValue = [[Fraction alloc] init];
 ...
[dataValue setReal: 10.0 andImaginary: 2.5];
```

这些代码行在编译时，编译器不会产生警告消息。这是因为编译器在处理源文件时并不知道储存在dataValue中的对象的类型。

直到运行包含有这些代码的程序时，才会出现如下的出错消息：

```
objc: Fraction: does not recognize selector -setReal:andImaginary:
dynamic3: received signal: Abort trap
When attempting to execute the expression
[dataValue setReal: 10.0 andImaginary: 2.5];
```

运行时系统首先检查存储在dataValue中的对象类型。因为在dataValue中存储有Fraction，所以运行时系统检查并确定setReal:andImaginary:方法是定义在Fraction类中的一个方法。因为事实并非如此，所以显示前面的出错消息并且终止程序的运行。

9.4 id数据类型与静态类型

如果id数据类型可以用来存储任何类型的对象，为什么不把所有的对象都声明为id类型呢？为什么不要养成滥用这种通用数据类型的习惯，有以下的几个原因。

首先，将一个变量定义为特定类的对象时，使用的是静态形态。“静态”这个词指的是这个变量总是用于存储特定类的对象。这样，存储在这种形态中的对象的类是预定的，也就是静态的。使用静态类型时，编译器尽可能确保变量的用法在程序中始终保持一致。编译器能够通过检查来确定应用于对象的方法是由该类定义的或者由该类继承，否则它将显示警告消息。这样，在程序中声明名为myRect的Rectangle变量时，编译器就会检查对myRect调用的每个方法是定义在Rectangle类中或者是从父类继承的。

注意 有一些方法可以调用变量指定的方法，在这种的情况下编译器不会进行检查。

但是，如果检查是在运行时执行的，为什么关心静态类型呢？关心静态形态是因为它能更好地在程序编译阶段而不是在运行时指出错误。如果把它留到运行时，你可能在运行程序时没有发现错误。假如程序能够投入应用，也许就会有一些可怜的信任你的使用者发现在他运行程序时特定的对象不能够识别的方法。

使用静态类型的另一个原因是它能够提高程序的可读性。考虑以下声明：

```
id f1;
```

和

```
Fraction *f1;
```

你认为哪个声明更容易理解，也就是说，对于读者来讲哪个更能清楚地说明使用f1变量的目的？结合使用静态类型和有意义的变量名（在以前的例子中，我们并没有刻意选择合适的变量名）将会对程序的自说明性产生深远的影响。

动态类型的参数和返回类型

如果使用动态类型来调用一个方法，需要注意以下的规则：如果在多个类中实现名称相同的方法，那么每个方法都必须符合各个参数的类型和返回值类型。这样编译器才能为消息表达式生成正确的代码。

编译器会对它所遇到的每个类声明执行一致性检查。如果有一个或多个方法在参数或者返回类型上存在冲突，编译器就会显示警告消息。例如：Fraction和Complex类中都包含add：方法。然而，Fraction类的参数和返回类型都是Fraction对象，而Complex类的参数和返回类型是Complex对象。如果frac1和myFracrt都是Fraction的对象，而comp1和myComplex都是Complex的对象，那么以下声明

```
result = [myFract add: frac1];
```

和

```
result = [myComplex add: comp1];
```

不会导致编译器显示任何警告消息，这是因为，在这两种情况下消息的接收者都是静态类型，

并且编译器可以检查这些方法的使用是否和在接收者类中定义的一致。

如果dataValue1和dataValue2是id变量，那么语句

```
result = [dataValue1 add: dataValue2];
```

会导致编译器生成代码来将参数传递给add:方法并通过假设处理其返回值。

在运行时，Objective-C运行时系统仍然检查存储在dataValue1中对象所属的确切类，并选择相应的方法来执行。然而，编译器可能生成不正确的代码来向方法传递参数或处理返回值。当一个方法选取对象作为它的参数，而另一个方法选取浮点数作为参数时，很有可能发生这种情况。如果这两个方法之间的不一致性仅在于对象类型的不同（例如，Fraction的add:方法使用Fraction对象作为其参数和返回值，而Complex的add：方法使用Complex对象作为参数），编译器仍然能够生成正确的代码，因为传递给对象的引用是内存地址（即指针）。

9.5 有关类的问题

开始使用可以包含来自不同类的对象的变量时，可能会遇到以下问题：

- 这个对象是矩形吗?
- 这个对象支持print方法吗?
- 这个对象是Graphics类或是其子类的成员吗?

这些问题的答案可以用来执行不同的代码序列，避免错误或在运行程序时检查程序的完整性。

表9-1总结了NSObject类所支持的一些基本方法，它们用来提出这类问题。在这个表中，class-object是一个类对象（通常是由class方法产生的），selector是一个SEL类型的值（通常是由@selector指令产生的）。

表9-1　处理动态类型的方法

方法	问题或行为
-(BOOL) isKindOf: class-object	对象是不是class-object或其子类的成员?
-(BOOL) isMemberOfClass: class-object	对象是不是class-object的成员?
-(BOOL) respondsToSeletor: selector	对象是否能够响应selector所指定的方法?
+(BOOL) instancesRespondToSelector:selector	指定的类实例是否能响应selector?
+(BOOL) isSubclassOfClass:class-object	对象是指定类的子类吗?
-(id) performSelector: selector	应用selector指定的方法
-(id) performSelector: selector withObject: object	应用selector指定的方法，传递参数object
-(id) performSelector: selector withObject: object1 withObject: object2	应用selector指定的方法，传递参数object1和object2

这里没有描述其他可用的方法，这些方法允许你提出关于是否符合某项协议的问题（协议将在第11章中讲述），还有一些方法允许你提出有关动态解析方法（本文中不讨论）的问题。

要根据类名或另一个对象生成一个类对象，可以向它发送class消息。所以，要从名为Square的类中获得类对象，可以编写如下代码：

```
[Square class]
```

如果mySquare是Square对象的实例，可以通过如下代码知道它所属的类：

```
[mySquare class]
```

要查看存储在变量obj1和obj2中的对象是不是相同的类实例，可以编写如下的代码：

```
if ([obj1 class] == [obj2 class])
...
```

要查看变量myFract是不是Fraction类的实例，可如下测试表达式的结果：

```
[myFract isMemberOf: [Fraction class]]
```

要生成表9-1中列出的所谓的selector，可以对一个方法名应用@selector指令。例如，

```
@selector (alloc)
```

为名为alloc的方法生成一个SEL类型的值，你知道这个方法是从NSObject类继承的。表达式

```
@selector (setTo:over:)
```

为setTo:over:方法生成一个selector，这个setTo:over:方法是在Fraction类中实现的（切记方法名称中的冒号字符）。

要查看Fraction类的实例是否响应setTo:over方法，可以如下测试表达式的返回值：

```
[Fraction instancesRespondToSelector: @selector (setTo:over:)]
```

记住，测试包括继承的方法，并不是只测试直接定义在类中的方法。

performSelector:方法和它的变形版本（在表9-1中没有显示）允许你向对象发送消息，这条消息可以是存储在变量中的selector。例如，考虑以下代码序列：

```
SEL      action;
id       graphicObject;
 ...
action = @selector (draw);
 ...
[graphicObject performSelector: action];
```

在这个例子中，SEL变量action所指定的方法被发送到存储在grphicObject中的任何图形对象。根据推测，即使已经把这个行为指定为draw，在程序执行时，行为也可能发生变化，可能依赖于用户的输入。要先确定对象是否可以响应这个动作，可能使用以下方式：

```
if ([graphicObject respondsToSelector: action] == YES)
    [graphicObject perform: action]
else
    // error handling code here
```

注意 还可能由于重载doesNotRecognize:方法而捕获错误。只要向一个类发送一条未识别的消息并传递未识别的selector作为其参数，就会调用这个方法。

还可以采用其他策略：可以使用forward:方法将消息转发给其他人去处理，或者试着发送方法并捕获发生的异常。稍后将简要介绍这个技术。

代码清单9-3对在第8章“继承”中定义的Square和 Rectangle类提出了一些问题。在查看实

际输出之前，尝试预测程序的结果（现在可不能偷看！）。

代码清单9-3

```
#import "Square.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Square *mySquare = [[Square alloc] init];

   // isMemberOf:

   if ( [mySquare isMemberOfClass: [Square class]] == YES )
      NSLog (@"mySquare is a member of Square class");

   if ( [mySquare isMemberOfClass: [Rectangle class]] == YES )
      NSLog (@"mySquare is a member of Rectangle class");

   if ( [mySquare isMemberOfClass: [NSObject class]] == YES )
      NSLog (@"mySquare is a member of NSObject class");

   // isKindOf:

   if ( [mySquare isKindOfClass: [Square class]] == YES )
      NSLog (@"mySquare is a kind of Square");

   if ( [mySquare isKindOfClass: [Rectangle class]] == YES )
      NSLog (@"mySquare is a kind of Rectangle");

   if ( [mySquare isKindOfClass: [NSObject class]] == YES )
      NSLog (@"mySquare is a kind of NSObject");

   // respondsTo:

   if ( [mySquare respondsToSelector: @selector (setSide:)] == YES )
      NSLog (@"mySquare responds to setSide: method");

   if ( [mySquare respondsToSelector: @selector (setWidth:andHeight:)] == YES )
      NSLog (@"mySquare responds to setWidth:andHeight: method");

   if ( [Square respondsToSelector: @selector (alloc)] == YES )
      NSLog (@"Square class responds to alloc method");

   // instancesRespondTo:
```

```
    if ([Rectangle instancesRespondToSelector: @selector (setSide:)] == YES)
       NSLog (@"Instances of Rectangle respond to setSide: method");

    if ([Square instancesRespondToSelector: @selector (setSide:)] == YES)
       NSLog (@"Instances of Square respond to setSide: method");

    if ([Square isSubclassOfClass: [Rectangle class]] == YES)
       NSLog (@"Square is a subclass of a rectangle");

    [mySquare release];

    [pool drain];
    return 0;
}
```

一定要使用Square、Rectangle和XYPoint类（它们都出现在第8章“继承”中）的实现文件编译这个程序。

代码清单9-3 输出

```
mySquare is a member of Square class
mySquare is a kind of Square
mySquare is a kind of Rectangle
mySquare is a kind of NSObject
mySquare responds to setSide: method
mySquare responds to setWidth:andHeight: method
Square class responds to alloc method
Instances of Square respond to setSide: method
Square is a subclass of a rectangle
```

代码清单9-3的输出应该很清晰。记住isMemberOf:测试类中的直接成员关系，而isKindOf:检测继承层次中的关系。所以mySquare是Square类的成员；但是它同样是“某种” Square、Rectangle和NSObject，因为它存在于这些类的层次结构中（很明显，所有对象都应该在有关NSObject类的isKindOf:的测试时返回YES，除非定义了新的根对象）。

测试语句

```
if ( [Square respondsTo: @selector (alloc)] == YES )
```

检测Square类是否响应alloc类方法，确实响应，因为Square类是从根对象NSObject继承来的。现在你认识到总能在消息表达式中直接使用类名作为接收者，并且在以前的表达式中不必编写

```
[Square class]
```

不过如果你想编写，也可以编写。这是唯一可以这样做的地方。在其他地方就需要应用class的方法来获取类对象。

9.6 使用@try处理异常

好的编程实践是指能预测程序中可能出现的问题。为此，你可以测试使程序异常终止的条件并处理这些情况，可能要记录一条消息并完全终止程序，或者采取其他正确措施。例如，本章前面讲过如何测试来查看一个对象是否响应特定的消息。以避免错误为例，在程序运行时执行测试可以避免向对象发送未识别的消息。当试图发送这类未识别消息时，程序通常会立即终止并抛出一个异常。

看一下代码清单9-4。Fraction类中未定义任何名为noSuchMethod的方法。当你编译程序时，就会得到相关警告消息。

代码清单9-4

```
#import "Fraction.h"

int main (int argc, char *argv [])

{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Fraction *f = [[Fraction alloc] init];
   [f noSuchMethod];
   NSLog (@"Execution continues!");
   [f release];
   [pool drain];
   return 0;
}
```

你可以不管警告消息继续运行程序。如果这样做，程序可能会异常终止，并出现类似如下的错误：

代码清单9-4　输出

```
-[Fraction noSuchMethod]: unrecognized selector sent to instance 0x103280
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
     reason: '*** -[Fraction noSuchMethod]: unrecognized selector sent
             to instance 0x103280'
Stack: (
   2482717003,
   2498756859,
   2482746186,
   2482739532,
   2482739730
)
Trace/BPT trap
```

为了避免在这类情况下程序异常终止，可以在一个特殊的语句块中加入一条或多条语句，格式如下：

```
@try {
    statement
    statement
    ...
}
@catch (NSException *exception) {
    statement
    statement
    ...
}
```

在@try块中加入这些statement后，程序正常执行。但是，如果块中的某一条语句抛出异常，执行不会终止，而是立即跳到@catch块，在那里继续执行。在@catch块内可以处理异常。这里可行的执行顺序是记录出错消息、清除和终止执行。

代码清单9-5演示了异常处理。紧跟着的是程序的输出。

代码清单9-5 异常处理

```
#import "Fraction.h"

int main (int argc, char *argv [])

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction *f = [[Fraction alloc] init];

    @try {
        [f noSuchMethod];
    }
    @catch (NSException *exception) {
        NSLog(@"Caught %@%@", [exception name], [exception reason]);
    }
    NSLog (@"Execution continues!");
    [f release];
    [pool drain];
    return 0;
}
```

代码清单9-5 输出

```
*** -[Fraction noSuchMethod]: unrecognized selector sent to instance 0x103280
Caught NSInvalidArgumentException: *** -[Fraction noSuchMethod]:
```

```
unrecognized selector sent to instance 0x103280
Execution continues!
```

出现异常时，@catch块被执行。包含异常信息的NSException对象作为参数传递给这个块。如你所见，name方法检索异常的名称，reason方法给出原因（运行时系统还会将原因自动输出）。

这是一个非常简单的例子，演示了如何在程序中捕获异常。可以使用@finally块包含是否执行抛出异常的@try块中的语句代码。

@throw指令允许你抛出自己的异常。可以使用该指令抛出特定的异常，或者在@catch块内抛出带你进入类似如下代码块的异常：

```
@throw;
```

自行处理异常后（例如，可能是在执行清除工作后）可能需要这么做。然后便可以让系统处理其余的工作。最后，可以使用多个@catch块按顺序捕获并处理各种异常。

9.7 练习

1. 如果在代码清单9-1中执行加法之后（但是在释放compResult的内存之前）插入以下消息表达式，将发生什么情况？

   ```
   [compResult reduce];
   ```

   试一试并查看结果。

2. 可以将代码清单9-2中定义的id变量dataValue分配给在第8章中定义的Rectangle对象吗？就是说，表达式

   ```
   dataValue = [ [Rectangle alloc] init];
   ```

   是否合法？为什么？

3. 给第8章中定义的XYPoint类中添加一个print方法。让它以格式（x，y）显示一个点。然后修改代码清单9-2来结合一个XYPoint对象。使修改后的程序创建一个XYPoint对象，设置其值，把这个值分配给id变量dataValue，最后显示它的值。

4. 基于本章关于参数和返回类型的讨论，修改Fraction和Complex类的add:方法来选取并返回id对象。然后编写一个程序并添加以下代码序列：

   ```
   result = [dataValue1 add: dataValue2];
   [result print];
   ```

   其中，result、dataValue1和dataValue2都是id对象。确保在程序中适当地设置dataValue1和dataValue2，并且在程序结束之前释放所有的对象。

   注意：必须将方法名改为非add:的其他名称。这是因为系统NSObjectController类也有一个add:方法。如9.4小节所述，如果在不同的类中有多个同名的方法，并且在编译时不知道接收器的类型，那么编译器就会执行一致性检查，确保参数和返回类型在名称类似的方法之间保持一致。

5. 根据本章中使用的Fraction和Complex类定义以及如下定义

```
Fraction *fraction = [[Fraction alloc] init];
Complex *complex = [[Complex alloc] init];
id number = [[Complex alloc] init];
```

确定以下消息表达式的返回值。然后将它们键入一个程序，验证结果。

```
[fraction isMemberOfClass: [Complex class]];
[complex isMemberOfClass: [NSObject class]];
[complex isKindOfClass: [NSObject class]];
[fraction isKindOfClass: [Fraction class]];
[fraction respondsToSelector: @selector (print)];
[complex respondsToSelector: @selector (print)];
[Fraction instancesRespondToSelector: @selector (print)];
[number respondsToSelector: @selector (print)];
[number isKindOfClass: [Complex class]];
[number respondsToSelector: @selector (release)];
[[number class] respondsToSelector: @selector (alloc)];
```

第10章　变量和数据类型

本章中，我们将讨论有关变量的作用域、对象的初始化方法以及数据类型的详细内容。

对象的初始化是本章中需要特别关注的内容。在第7章“类”中，已简要地介绍了实例变量、静态变量以及局部变量的作用域。这里将更深入地讲述静态变量，并且将引入全局变量和外部变量的概念。同时，还将给出一些指令，以便Objective-C编译器能够更准确地控制实例变量的作用域。本章还将讲述这些指令。

使用枚举数据类型，可以定义只存储一系列特定值的数据类型名称。Objective-C语言的typedef语句允许你对内置或派生的数据类型指派自己的名称。最后，在本章，我们将更详细地描述在表达式的求值过程中转换数据时，Objective-C编译器所遵循的确切步骤。

10.1　类的初始化

前面已经出现过这种模式：使用以下常见序列，分配对象的新实例，然后对它初始化：

```
Fraction *myFract = [[Fraction alloc] init];
```

调用这两个方法之后，通常向这个新对象指派一些值，如下所示：

```
[myFract setTo: 1 over: 3];
```

初始化对象之后为其设置初值的过程通常可合并到一个方法中。例如，你可以定义一个initWith::方法，它初始化一个分数，并将其分子和分母设置为两个给定的参数（没有给出名称）。

包含很多方法和实例变量的类通常还有几个初始化方法。例如，Foundation框架中的NSArray类包含了以下6个初始化方法：

```
initWithArray:
initWithArray:copyItems:
initWithContentsOfFile:
initWithContentsOfURL:
initWithObjects:
initWithObjects:count:
```

很可能会用下面的语句序列完成数组的空间分配和初始化工作：

```
myArray = [[NSArray alloc] initWithArray: myOtherArray];
```

常见的编程习惯是类中所有初始化方法都以init...开头。可以看到，NSArray的初始化方法遵循这个惯例。在编写初始化方法时，应该遵循以下两个策略。

如果你的类包含多个初始化方法，其中一个就应该是指定的（designated）初始化方法，并且其他所有初始化方法都应该使用这个方法。通常，它是最复杂的初始化方法（一般是参数最多的初始化方法）。通过创建指定的初始化方法，可以把大部分初始化代码集中到单个方法

中。然后，任何人要想从该类派生子类，就可以重载这个指定的初始化方法，以便保证正确地初始化新的实例。

一定要恰当地初始化任何继承来的实例变量。最简单的方式就是首先调用父类指定的初始化方法，大多数情况下是init方法，然后，可以初始化自己的实例变量。

基于这个讨论，Fraction类的初始化方法initWith::可能如下所示：

```
-(Fraction *) initWith: (int) n: (int) d
{
   self = [super init];

   if (self)
      [self setTo: n over: d];

   return self;
}
```

这个方法首先调用父类的初始化方法，也就是NSObject的init方法（回忆一下，它是Fraction的父类）。初始化的结果需要指派回self，因为初始化方法有权更改或移动内存中的对象。

完成Super的初始化（返回的非零值表示初始化成功）之后，使用setTo:over:方法设置Fraction的分子和分母。和其他初始化方法一样，希望由你返回初始化的对象，在这里你就是这样做的。

代码清单10-1测试了新的初始化方法initWith::。

代码清单10-1

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [NSAutoreleasePool alloc] init];

   Fraction *a, *b;

   a = [[Fraction alloc] initWith: 1: 3];
   b = [[Fraction alloc] initWith: 3: 7];

   [a print];
   [b print];

   [a release];
   [b release];

   [pool drain];
   return 0;
}
```

代码清单10-1　输出

```
1/3
3/7
```

开始执行程序时，它向所有类发送initialize调用方法。如果存在一个类及相关的子类，则父类首先得到这条消息。该消息只向每个类发送一次，并且向该类发送其他任何消息之前，保证向其发送初始化消息。这样做的目的是在程序开始时执行所有类的初始化工作。例如，你可能想在开始时初始化与类有关的静态变量。

10.2　作用域回顾

可以使用几种方式影响程序中变量的作用域。可以改变实例变量以及定义在函数外部或内部的普通变量的作用域。在下面的讨论中，我们使用术语模块（module）来引用包含在一个源文件中任何数目的方法或者函数定义。

10.2.1　控制实例变量作用域的指令

目前，你知道实例变量的作用域只限于为该类定义的实例方法。因此，任何实例方法都能直接通过变量名来访问该类的实例变量，而无需特别的操作。

你还知道，实例变量可通过子类进行继承。继承来的实例变量同样可以通过变量名在该子类定义的方法中直接访问。同样，这也无需执行其他特别的操作。

在接口部分声明实例变量时，可以把以下三个指令放在实例变量之前，以便更精确地控制其作用域，它们是：

- @protected——这个指令后面的实例变量可被该类及任何子类中定义的方法直接访问。这是默认的情况。
- @private——这个指令后面的实例变量可被定义在该类的方法直接访问，但是不能被子类中定义的方法直接访问。
- @public——这个指令后面的实例变量可被该类中定义的方法直接访问，也可被其他类或模块中定义的方法直接访问。
- @package——对于64位图像，可以在实现该类的图像的任何地方访问这个实例变量。

如果要定义一个名为Printer的类，它包含两个私有实例变量pageCount和tonerLevel，并且只有Printer类中的方法才能直接访问它们，那么可以如下使用接口部分：

```
@interface Printer: NSObject
{
@private
   int pageCount;
   int tonerLevel;
@protected
   // other instance variables
}
 ...
@end
```

由于这两个实例变量均为私有的，所以任何从Printer派生子类的人都无法访问它们。

这些特殊的指令和“开关”一样，所有出现在这些指令之后的变量（直到标志着变量的声明结束的右花括号为止）都有指定作用域，除非使用另一个指令。在前面的例子中，@protected指令确保它后面和)符号之前的实例变量可以被Printer的类方法访问，也可以被子类访问。

@public指令使得其他方法或函数可以通过使用指针运算符（->）访问实例变量，这些内容将在第13章“基本的C语言特性”中详细讲述。将实例变量声明为公共的并不是良好的编程习惯，因为这违背了数据封装的思想(即，类隐藏它的实例变量)。

10.2.2 外部变量

如果在程序的开始处（所有方法、类定义和函数定义之外）编写以下语句：

```
int gMoveNumber = 0;
```

那么这个模块中的任何位置都可以引用这个变量的值。在这种情况下，我们说gMoveNumber被定义为全局变量。为了向阅读程序的人说明变量的作用域，按照惯例，用小写的g作为全局变量的首字母。

实际上，这样的定义使得其他文件也可以访问变量gMoveNumber的值。确切地说，前面语句不仅将gMoveNumber定义为全局变量，而且将其定义为外部全局变量。

外部变量是可被其他任何方法或函数访问和更改其值的变量。在需要访问外部变量的模块中，变量声明和普通方式一样，只是需要在声明前加上关键字extern。这就告知系统，要访问其他文件中定义的全局变量。下面这个例子说明如何将gMoveNumber声明为外部变量：

```
extern int gMoveNumber;
```

现在，包含前面这个声明的模块就可以访问和改变gMoveNumber的值。同样，通过在文件中使用类似的extern声明，其他模块也可以访问gMoveNumber的值。

使用外部变量时，必须遵循下面这条重要原则：变量必须定义在源文件中的某个位置。这是通过在所有方法和函数外部声明变量，并且前面不加关键字extern，如下所示：

```
int gMoveNumber;
```

这里，前面显示过，可以为这个变量指派初始值。

定义外部变量的第二种方式是在所有函数之外声明变量，在声明前面加上关键字extern，同时显式地为变量指派初始值，如下所示：

```
extern int gMoveNumber = 0;
```

然而，这并不是首选的方法。编译器将给出警告消息：提示你已将变量声明为extern的，并同时为变量赋值。这是因为使用关键字extern的表明这条语句是变量的声明而不是定义。记住，声明不会引起分配变量的存储空间，而定义会引起变量存储空间的分配。因此，前面的例子强行将声明当作定义处理（通过指派初始值），所以违背了这个规则。

处理外部变量时，变量可以在许多地方声明为extern，但是只能定义一次。

通过观察一个小程序例子来说明外部变量的用法。假设我们定义了一个名为Foo的类，并

将以下代码键入一个名为main.m的文件中：

```
#import "Foo.h"

int gGlobalVar = 5;

int main (int argc, char *argc[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Foo *myFoo = [[Foo alloc] init];
    NSLog (@"%i ", gGlobalVar);

    [myFoo setgGlobalVar: 100]
    NSLog (@'%i', gGlobalVar);
    [myFoo release];
    [pool drain];
    return 0;
}
```

在前面的程序中，gGlobalVar定义为全局变量，因此任何方法（或函数）只要正确地使用extern声明，都可以访问它。假设Foo方法setgGlobalVar:如下所示：

```
-(void) setgGlobalVar: (int) val
{
   extern int gGlobalVar;
   gGlobalVar = val;
}
```

该程序将在终端生成以下结果：

```
5
100
```

这就证明了方法setgGlobalVar:可以访问和改变外部变量gGlobalVar的值。

如果有很多方法需要访问gGlobalVar的值，只在文件的开始进行一次extern声明将比较简便。但是如果只有一个或少数几个方法要访问这个变量，就应该在其中每个方法中单独进行extern声明。这样将使程序的组织结构更清晰，并且使实际使用变量的不同函数单独使用这个变量。注意，如果变量定义在包含访问该变量的文件中，那么不需要单独的extern声明。

10.2.3 静态变量

前面所示的例子与数据封装原则以及良好的面向对象编程技术相违背。然而，可能需要下面这种变量：它们的值在经过不同的方法调用时是共享的。虽然在Foo类中将gGlobalVar定义为实例变量似乎也不太合理，但是，更好的方法可能是通过将访问限制在类中定义的setter和getter方法中，将实例变量“隐藏”在Foo类中。

现在，你知道在方法之外定义的变量不仅是全局变量而且是外部变量。然而，在很多情况下你想要将变量定义为全局变量但不是外部变量。换句话说，希望定义的全局变量只在特定模

块（文件）中是全局的。这种变量在下面的情况中很有意义：除了特定类中的方法，再没有其他方法需要访问这个特定变量。要做到这一点，可以在包含这个特定类实现的文件中将该变量定义为static。

如果语句

```
static int gGlobalVar = 0;
```

声明在任何方法（或函数）之外，那么在该文件中，所有位于这条语句之后的方法或函数都可以访问gGlobalVar的值，而其他文件中的方法和函数则不行。

你会想起类方法不能访问实例变量（你可能考虑为什么又是这样）。然而，可能希望类的方法可以设定和访问一些变量。简单的例子是类的分配器方法，它要记录类已经分配空间的对象数目。实现这项任务的方式是在类的实现代码文件中设定静态变量。由于这个变量不是实例变量，所以分配器方法可以直接访问它。类的用户不用知道这个变量。因为它是定义在实现文件中的静态变量，作用域只是文件内部。因此，用户不能直接访问该变量，也就没有违背数据封装的概念。如果需要从类之外访问该变量，则可以编写一个方法来获取该变量的值。

代码清单10-2对Fraction的类定义进行了扩充，增加了两个新方法。allocF类方法分配一个新的Fraction对象，同时记录分配了多少Fraction，count方法则返回这个数的值。注意，后者也是类方法。也可以作为实例方法实现，然而，与向类的特定实例发送消息相比，询问类已经分配了多少实例更有意义。

下面是在头文件Fraction.h中添加的这两个新的类方法声明：

```
+(Fraction *) allocF;
+(int) count;
```

你可能注意到，继承来的alloc方法并没有被重载；相反，你定义了自己的分配器方法。这个方法就可以利用继承来的alloc方法。下面是需要在实现文件Fraction.m中加入的代码：

```
static int gCounter;

@implementation Fraction;

+(Fraction *) allocF
{
   extern int gCounter;
   ++gCounter;

   return [Fraction alloc];
}

+(int) count
{
   extern int gCounter;

   return gCounter;
}
```

```
// other methods from Fraction class go here
  ...
@end
```

注意　重载alloc并不是好的编程实践，因为这个方法处理内存的物理分配。你不应该涉足那个领域。

count声明为静态使得定义在执行文件中的方法可以访问它，但是该文件之外都不可以访问。allocF方法仅仅递增gCounter变量，然后使用alloc方法创建一个新的Fraction，并返回结果；gCounter方法只是返回计数器的值，这样就隔离了来自用户的直接访问。

回忆一下，因为gCounter变量定义在该文件中，因此不需要在这两个方法中使用extern声明。声明只是为了让阅读该方法的人明白：他访问的变量是定义在该方法之外。变量名加g前缀也是出于同样的目的；因此，大多数程序员一般不使用extern声明。

代码清单10-2测试这个新方法。

代码清单10-2

```
#import "Fraction.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Fraction *a, *b, *c;

   NSLog (@"Fractions allocated: %i", [Fraction count]);

   a = [[Fraction allocF] init];
   b = [[Fraction allocF] init];
   c = [[Fraction allocF] init];

   NSLog (@"Fractions allocated: %i", [Fraction count]);
   [a release];
   [b release];
   [c release];

   [pool drain];
   return 0;
}
```

代码清单10-2　输出

```
Fractions allocated: 0
Fractions allocated: 3
```

程序开始运行时，gCounter的值会自动置为0（你应该还记得，如果要将类作为整体进行任何特殊初始化，例如，将其他静态变量的值设置为一些非零值，可以重载继承类的initialize方

法)。使用allocF方法分配三个Fraction实例之后，count方法检索counter变量的值，它被正确地设置为3。如果要重置计数器或将其设为特定的值，可以在类中添加一个setter方法。然而，在这个程序中并不需要这么做。

10.3 存储类说明符

你已经遇到过一些可以放在变量前面的存储类说明符，如extern和static。下面将讨论更多的说明符，它们给出有关在程序中使用变量的编译器信息。

10.3.1 auto

这个关键字用来声明一个自动局部变量，与static相反。这是函数或方法内部变量的默认声明方式，并且没有人使用它。下面是一个例子：

```
auto int index;
```

该语句声明index为一个自动局部变量，意味着在进入该块（即一段位于花括号之内的语句、方法或函数序列）时，自动为它分配存储空间，并在退出该块时自动解除分配。因为在块中是默认的，因此语句

```
int index;
```

和语句

```
auto int index;
```

是等效的。

静态变量有默认的初始值0，而自动变量没有默认的初始值。除非显式地给自动变量赋值，否则它们的值是不确定的。

10.3.2 const

编译器允许你给对程序中值不变的变量设置const特性。这样，就告诉编译器，指定的变量在程序运行期间都有恒定的值。在初始化变量之后，如果尝试给const变量指派一个值，或试图将其增1或减1，编译器就会给出警告消息。举一个const特性的例子，代码

```
const double pi = 3.141592654;
```

声明了一个const变量pi。这就告诉编译器，程序不会修改该变量。当然，因为随后不能更改const变量的值，因此必须在定义变量时进行初始化。

将变量定义为const变量在自文档编制过程（self-documentation process）中很有帮助。这让读程序的人知道该变量的值不会被程序改变。

10.3.3 volatile

这个类型和const正好相反。它明确地告诉编译器，指定类型变量的值会改变。在语言中加入这个关键字是为了防止编译器优化掉看似多余的变量赋值，同时避免重复地检查值没有变化的变量。I/O端口就是一个很好的例子，这涉及到对指针的理解（参见第13章）。

假设在程序中，将输出端口的地址存储在一个名为outPort的变量中。如果要向这个端口写两个字符（一个O后面接一个N），可能会写出下面的代码：

```
*outPort = 'O';
*outPort = 'N';
```

第一行表示在outPort指定的内存地址存储字符O。第二行则表示在同一位置存储字符N。一个智能的编译器可能发现对同一地址进行了两次连续的赋值。因为outPort在这两者之间并没有被修改，所以编译器将第一个赋值语句从程序中删除。要防止这种情况发生，应该将outPort声明为一个volatile变量，如下所示：

```
volatile char *outPort;
```

10.4 枚举数据类型

Objective-C语言使得你可以将一系列值指派给一个变量。枚举数据类型的定义以关键字enum开头，之后是枚举数据类型的名称，然后是标识符序列（包含在一对花括号内），它们定义了可以给该类型指派的所有允许值。例如，语句

```
enum flag { false, true };
```

定义了一个数据类型flag。从理论上说，在程序中这个数据类型只能指派true和false两种值，不能指派其他值。遗憾的是，即使违背了这个规则，Objective-C编译器也不会发出警告消息。

要声明一个enum flag类型的变量，仍需要用到关键字enum，之后是枚举类型名称，最后是变量序列。所以语句

```
enum flag endOfData, matchFound;
```

定义了两个flag类型的变量endOfData和matchFound。能指派给这两个变量的值只有（理论上是这样）true和false。因此，

```
endOfData = true;
```

和

```
if ( matchFound == false )
   ...
```

之类的语句都是合法的。

如果希望一个枚举标识符对应一个特定的整数值，那么可以在定义数据类型时给该标识符指定整数值。在列表中依次出现的枚举标识符被指派了以特定整数值开始的序列数。

在定义

```
enum direction { up, down, left = 10, right };
```

中，定义了一个包含值up、down、left和right的枚举数据类型direction。因为up在序列中位于首位，所以编译器给它赋值为0；down接着up，因此赋给它的值为1；由于left明确指定了一个整数，赋给它值就是10；right的值由列表中前一个enum的值递增得到，因此给它赋的值为11。

枚举标识符可以共享相同的值。例如，

```
enum boolean { no = 0, false = 0, yes = 1, true = 1 };
```

给enum boolean变量指派no和false时，就是向其赋值0，指派yes和true时赋值1。

再举另一个枚举数据类型定义的例子。下面定义了类型enum month。对于这种类型的变量，可以指派的值为一年中十二个月的名字：

```
enum month { january = 1, february, march, april, may, june, july,
        august, september, october, november, december };
```

Objective-C编译器实际上将枚举标识符作为整型常量来处理。如果你的程序包含以下两行

```
enum month thisMonth;
   ...
thisMonth = february;
```

那么给thisMonth赋的值是整数2（而不是february这个名字）。

代码清单10-3展示了使用枚举数据类型的简单程序。该程序首先读取一个月份数，然后进入switch语句来判断要进入哪个月份。回忆一下，编译器把枚举值当作整型常量处理，因此它们都是有效的case值。将变量days赋值为该月的天数，在switch退出后显示days的值。程序中包含特定的测试，用来查看该月是否为二月。

代码清单10-3

```
#import <Foundation/Foundation.h>
// print the number of days in a month
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    enum month { january = 1, february, march, april, may, june,
                 july, august, september, october, november,
                 december };
    enum month amonth;
    int     days;

    NSLog (@"Enter month number: ");
    scanf ("%i", &amonth);
    switch (amonth) {
       case january:
       case march:
       case may:
       case july:
       case august:
       case october:
       case december:
                days = 31;
                break;
       case april:
       case june:
```

```
        case september:
        case november:
                days = 30;
                break;
        case february:
                days = 28;
                break;
        default:
                NSLog (@"bad month number");
                days = 0;
                break;
    }

    if ( days != 0 )
       NSLog (@"Number of days is %i", days);

    if ( amonth == february )
       NSLog (@"...or 29 if it's a leap year");

    [pool drain];
    return 0;
}
```

代码清单10-3　输出

```
Enter month number:
5
Number of days is 31
```

代码清单10-3　输出（再次运行）

```
Enter month number:
2
Number of days is 28
...or 29 if it's a leap year
```

可以明确地给枚举类型的变量指派一个整数值，这应该使用类型转换运算符。因此，如果monthValue是值为6的整型变量，那么表达式

```
lastMonth = (enum month) (monthValue - 1);
```

是允许的。如果不使用类型转换运算符，编译器也不会有异议（很遗憾）。

使用包含枚举数据类型的程序时，尽量不要依赖枚举值被当作整数这个实事。相反，尽量把它们当作独立的数据类型。枚举类型提供了一种方法，使你能把整数值和有象征意义的名称对应起来。如果以后需要更改这个整数的值，只能在定义枚举的地方更改。如果根据枚举数据类型的实际值进行假设，就丧失了使用枚举带来的好处。

在定义枚举数据类型时，也允许有所变化：可以省略数据类型的名称，定义该类型时，可

以将变量声明为特定枚举数据类型中的一个。举出一个同时展示这两种选择的例子，语句

```
enum { east, west, south, north } direction;
```

定义了一个（未命名的）枚举数据类型，它包含的值为east、west、south和north，同时还声明了该类型的变量direction。

在代码块中定义的枚举数据类型的作用域限于块的内部。另一方面，在程序的开始及所有块之外定义的枚举数据类型对于该文件是全局的。

定义枚举数据类型时，必须确保枚举标识符与定义在相同作用域之内的变量名和其他标识符不同。

10.5 typedef语句

Objective-C提供了一种能力：编程者可以为数据类型另外指派一个名称。这是通过所谓的typedef语句实现的。语句

```
typedef int Counter;
```

定义名称Counter等价于Objective-C数据类型int。随后的变量就可以声明为Counter类型，如在以下语句中：

```
Counter j, n;
```

Objective-C编译器实际上是将变量j和n的声明当作前面显示的普通整型变量。在这种情况下使用typedef语句的主要好处是增加了变量定义的可读性。从j 和n的定义中就可以清晰地看出这些变量在程序中的使用目的。用传统方式将变量定义为int类型不能很清晰地表示出它们的用途。

下面的typedef将一个名为NumberObject的类型定义为Number对象：

```
typedef Number *NumberObject;
```

随后将一些变量声明为NumberObject类型，如在语句

```
NumberObject myValue1, myValue2, myResult;
```

中，它们的使用方式和以常规方式在程序中声明一样，如下所示：

```
Number *myValue1, *myValue2, *myResult;
```

使用typedef定义一个新类型名，按照下面的步骤：

1. 像声明所需类型的变量那样编写一条语句。
2. 在通常应该出现声明的变量名的地方，将其替换为新的类型名。
3. 在语句的前面加上关键字typedef。

作为这个过程的例子，定义一个名为Direction的枚举数据类型，它包含4个方向：东、南、西和北。写出枚举类型的声明，在通常出现变量名称的地方使用名称Direction替代。在开始其他工作之前，在语句前加上关键字typedef：

```
typedef enum { east, west, south, north } Direction;
```

将typedef放在合适的位置之后，就可以声明Direction类型的变量了，如以下语句所示：

```
Direction step1, step2;
```

Foundation框架在它的一个头文件中使用typedef对NSComparisonResult进行了如下定义：

```
enum _NSComparisonResult {
   NSOrderedAscending = -1, NSOrderedSame, NSOrderedDescending
};

typedef NSInteger NSComparisonResult;
```

Foundation框架中一些用于比较的方法会返回一个该类型的值。例如，Foundation的字符串比较方法名为compare:，它在完成两个NSString对象字符串的比较之后返回一个NSComparisonResult类型的值。该方法的声明如下所示：

```
-(NSComparisonResult) compare: (NSString *) string;
```

要测试两个名为userName 和savedName的NSString对象是否相等，可以在程序里编写以下代码行：

```
If ([userName compare: savedName] == NSOrderedSame){
    // The names match
    ...
}
```

这实际上测试compare:方法的返回值是否为0。

10.6 数据类型转换

第4章“数据类型和表达式”曾简要地说明，在表达式求值的过程中，系统有时会进行隐式的数据类型转换。你验证的例子是float和int数据类型。你看到了涉及一个float和一个int的运算如何作为浮点运算进行求值，整型数被自动转换成为浮点型。

你还看到了如何使用类型转换运算符显式地控制转换。因此，假设total和n都是整型变量，

```
average = (float) total / n;
```

在运算之前，变量total的值被转换为float型，这就保证了将除法运算作为浮点运算求值。

10.6.1 转换规则

对含有不同类型数据的表达式求值时，Objective-C编译器会遵循一些非常严格的规则。

下面总结了表达式求值过程中，不同类型操作数发生转换的先后顺序：

1. 如果其中一个操作数是long double型，则另一操作数被转换为long double型，计算结果也是这种类型。

2. 如果其中一个操作数是double型，则另一操作数转换为double型，计算结果也是这种类型。

3. 如果其中一个操作数是float型，则另一操作数转换为float型，计算结果也是这种类型。

4. 如果其中一个操作数是_Bool、char、short int、bit field[⊖]或枚举数据类型，则全部转换为int型。

5. 如果其中一个操作数是long long int型，则另一操作数转换为long long int型，计算结果

⊖ 第13章节简要地讨论了bit field。

也是这种类型。

6. 如果其中一个操作数是long int型，则另一操作数转换为long int型，计算结果也是这种类型。

7. 如果到达这一步，则可知两个操作数均为int型，计算结果也是这种类型。

以上列出的实际上只是表达式中操作数类型转换过程涉及步骤的简化版本。涉及unsigned操作数时，规则还会更复杂。在附录B“Objective-C语言总结”中，可以看到完整的转换规则。

从这一系列步骤中可以认识到，只要到达“计算结果也是这种类型”，转换过程就结束了。

举一个例子说明如何遵循这些步骤。观察以下表达式的求值方式。其中f定义为float变量，i为int变量，l为long int变量，s为short int变量：

```
f * i + l / s
```

首先考虑f和i的相乘运算，这是float和int相乘。由步骤3可知，由于f是float型，另外一个操作数（i）会被转换为float类型，相乘的结果也是float。

下一步，考虑l和s的除法运算，这是long int除以short int。由步骤4可知short int会被转换为int，然后，步骤6表明，由于其中一个操作数（l）是long int类型，所以另一个操作数将被转换为long int，计算结果也是这个类型。因此，这个除法的结果是long int类型，实际上删节了除法运算结果的小数部分。

最后，按照步骤3，如果表达式中一个操作数为float类型（变量f和i相乘的结果），则另一个操作数将被转换为float类型，运算结果也是float类型。因此，在l和s的除法运算完成之后，其结果将被转换为float类型，然后和f和i的乘积相加。整个表达式的最终计算结果是float类型的值。

记住，总能使用类型转换运算符显式地强制类型转换，从而控制特定表达式的求值方式。

因此，在前面的表达式求值过程中，如果不希望l和s相除得到截取的结果，可以将其中一个变量转换为float类型，这样就能强制表达式作为浮点除法求值，如下所示：

```
f * i + (float) l / s
```

在这个表达式中，由于类型转换运算符比除法运算符的优先级高，所以l首先转换为float，然后进行除法运算。因为这个除法运算中一个操作数变成了float类型，所以，其他一个或多个操作数也会自动转成float类型，计算结果也是这个类型。

10.6.2 符号扩展

只要将有符号的int或short int转换成更多字节的整型，在执行转换的过程中符号位就会扩展至左侧。这就保证值为−5的short int转换成long int之后，它的值仍保持为−5。可以想到，将unsigned的整数转换为更多字节的整数类型时，不会发生符号扩展。

在一些计算机上（如在当前Mac系列使用的英特尔处理器上，以及在iPhone和iTouch当前使用的ARM处理器上），字符是作为有符号的量处理的。这就意味着将字符转换为整数时，会发生符号扩展。只要字符都按照标准ASCII字符集定义，就不会产生问题。但是，如果使用的字符值不是标准字符集的一部分，将其转换为整型时，它的符号就可能会扩展。例如，在Mac

机上，字符常量‘\377’会被转换为值−1，这是因为作为有符号的8-bit数时，它的值是负的。

回忆一下，Objective-C语言中允许将字符变量声明为unsigned，这就避免了这个潜在的问题。也就是，将unsigned char变量转换为整型时，永远不进行符号扩展，因为它在数值上总是大于或等于0。对于典型的8-bit字符，有符号字符变量的取值范围是−128至+127，包括这两个数。无符号字符变量的取值范围是0至255，包括这两个数。

如果要强制对字符变量进行符号扩展，则可将这些变量声明为signed char型。这保证将字符变量转换为整型时发生符号扩展，即使在默认情况下不这样做的计算机上也适用。

在第15章“数字、字符串和集合”中，你将学习处理多字节Unicode字符。这是处理一些字符串的首选方式，这些字符串中的字符可以来自包含数百万字符的字符集。

10.7 练习

1. 使用第8章“继承”中的Rectangle类，根据下面的声明增加一个初始化方法：

```
-(Rectangle *) initWithWidth: (int) w andHeight: (int) h;
```

2. 假设将练习1中的初始化方法标记为Rectangle类的指定初始化方法，根据第8章定义的Square和Rectangle类，结合下面的声明，为Square类增加一个初始化方法：

```
-(Square *) initWithSide: (int) side;
```

3. 为Fraction类的add:方法增加一个计数器来计算它的调用次数。如何获取这个变量的值？
4. 使用typedef和枚举数据类型定义一个名为Day的类型，可能的值为Sunday、Monday、Tuesday、Wednesday、Thursday、Friday和Saturday。
5. 使用typedef和枚举数据类型定义名为FractionObj的类型，该类型允许你编写如下语句：

```
FractionObj f1 = [[Fraction alloc] init],
            f2 = [[Fraction alloc] init];
```

6. 根据下面的定义

```
float     f = 1.00;
short int i = 100;
long int  l = 500L;
double    d = 15.00;
```

和本章讲解表达式中操作数类型转换时列举的7个步骤，确定以下表达式的类型和值。

```
f + i
l / d
i / l + f
l * i
f / 2
i / (d + f)
l / (i * 2.0)
l + i / (double) l
```

7. 编写一个程序，确定在计算机上有符号的char变量是否进行符号扩展。

第11章　分类和协议

在本章中，你将学习如何通过使用分类（category）以模块的方式向类添加方法，以及如何创建标准化的方法列表供其他人实现。

11.1　分类

有时候在处理类定义时，可能想要为其添加一些新方法。例如，对于Fraction类，除了将两个分数相加的add:方法之外，还想要拥有将两个分数相减、相乘、相除的方法。

再举一个例子，假如你参与一个大型程序设计项目，并且作为该项目的一部分，正在定义一个新类，它包含许多方法。你的任务就是为该类编写处理文件系统的方法。其他项目成员的任务负责以下方法：创建和初始化该类实例、对该类中的对象执行操作以及在屏幕上绘制该类对象的表示。

最后一个例子，假如你已经知道如何使用库中的类（例如，Foundation的数组类，名为NSArray），并且认识到你希望该类实现了一个或多个方法。当然，可以编写NSArray类的新子类并实现新方法，但是可能存在更简单的方式。

以上所有情况的实用解决方案是一个：分类。分类提供了一种简单的方式，用它可以将类的定义模块化到相关方法的组或分类中。它还提供了扩展现有类定义的简便方式，并且不必访问类的源代码，也无需创建子类。分类是一个功能强大且简单的概念。

回到第一个例子，展示如何为Fraction类添加新分类，以处理基本的四则数学运算。首先，展示原始的Fraction接口部分：

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

// Define the Fraction class

@interface Fraction : NSObject
{
   int numerator;
   int denominator;
}
@property int numerator, denominator;
-(void) setTo: (int) n over: (int) d;
-(Fraction *) add: (Fraction *) f;
-(void)   reduce;
-(double) convertToNum;
-(void)   print;
@end
```

然后，从接口部分删除add:方法，并将其添加到新分类，同时添加其他三种要实现的数学

运算。新MathOps分类的接口部分应该如下所示：

```
#import "Fraction.h"
@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end
```

注意，这既是接口部分的定义，也是现有接口部分的扩展。因此，必须包括原始接口部分，这样编译器就知道Fraction类（除非直接将新分类结合到原始Fraction.h头文件，这是一种选择）。

在#import之后，有下面这一行：

```
@interface Fraction (MathOps)
```

这告诉编译器你正在为Fraction类定义新的分类，而且它的名称为MathOps。这个名称括在类名称之后的一对圆括号中。注意此处没有列出Fraction的父类，因为编译器已从Fraction.h中知道此内容。而且，你没有向编译器告知实例变量，因为在以前定义的接口部分中已经这样做了。实际上，如果尝试列出父类或实例变量，将收到编译器发出的语法错误。

这个接口部分告知编译器，你正在名为MathOps的分类下为名为Fraction的类添加扩展。MathOps分类包括4个实例方法：add:、mul:、sub:和div:。每种方法均使用一个分数作为参数并返回一个分数。

可以将所有方法的定义放在一个实现部分。也就是，可以在一个实现文件中定义Fraction.h接口部分中的所有方法，以及MathOps分类中的所有方法。或者，在单独的实现部分定义分类的方法。在这种情况下，这些方法的实现部分还必须找出方法所属的分类。与接口部分一样，通过将分类名称括在类名称之后的圆括号中来确定方法所属的分类，如下所示：

```
@implementation Fraction (MathOps)
   // code for category methods
   ...
@end
```

在代码清单11-1中，新的MathOps分类的接口和实现部分组合在一起，连同测试例程都放在一个文件中。

代码清单11-1　MathOps分类和测试程序

```
#import "Fraction.h"

@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end
```

```
@implementation Fraction (MathOps)
-(Fraction *) add: (Fraction *) f
{
   // To add two fractions:
   // a/b + c/d = ((a*d) + (b*c)) / (b * d)

   Fraction *result = [[Fraction alloc] init];
   int     resultNum, resultDenom;

   resultNum = (numerator * f.denominator) +
      (denominator * f.numerator);
   resultDenom = denominator * f.denominator;

   [result setTo: resultNum over: resultDenom];
   [result reduce];

   return result;
}

-(Fraction *) sub: (Fraction *) f
{
   // To sub two fractions:
   // a/b - c/d = ((a*d) - (b*c)) / (b * d)

   Fraction *result = [[Fraction alloc] init];
   int     resultNum, resultDenom;

   resultNum = (numerator * f.denominator) -
      (denominator * f.numerator);
   resultDenom = denominator * f.denominator;

   [result setTo: resultNum over: resultDenom];
   [result reduce];

   return result;
}

-(Fraction *) mul: (Fraction *) f
{
   Fraction *result = [[Fraction alloc] init];

   [result setTo: numerator * f.numerator
            over: denominator * f.denominator];
   [result reduce];

   return result;
}
```

```
-(Fraction *) div: (Fraction *) f
{
   Fraction *result = [[Fraction alloc] init];

   [result setTo: numerator * f.denominator
            over: denominator * f.numerator];
   [result reduce];

   return result;
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   Fraction *a = [[Fraction alloc] init];
   Fraction *b = [[Fraction alloc] init];
   Fraction *result;

   [a setTo: 1 over: 3];
   [b setTo: 2 over: 5];

   [a print]; NSLog (@ "+"); [b print]; NSLog (@ "-----");
   result = [a add: b];
   [result print];
   NSLog (@ "\n ");
   [result release];

   [a print]; NSLog (@ "-"); [b print]; NSLog (@ "-----");
   result = [a sub: b];
   [result print];
   NSLog (@ "\n ");
   [result release];

   [a print]; NSLog (@ "*"); [b print]; NSLog (@ "-----");
   result = [a mul: b];
   [result print];
   NSLog (@ "\n ");
   [result release];

   [a print]; NSLog (@ "/"); [b print]; NSLog (@ "-----");
   result = [a div: b];
   [result print];
   NSLog (@ "\n ");
   [result release];
   [a release];
   [b release];
```

```
    [pool drain];
    return 0;
}
```

代码清单11-1 输出

```
 1/3
   +
 2/5
-----
11/15

 1/3
   -
 2/5
-----
-1/15

1/3
  *
2/5
-----
2/15

1/3
  /
2/5
-----
5/6
```

再次注意，以下语句在Objective-C语言中是合法的：

```
[[a div: b] print];
```

这行代码将直接打印分数a除以b的结果，因此避免了对变量result的中间赋值，在代码清单11-1中有这项操作，但需要执行这个中间赋值，这样可以获得结果Fraction，并随后释放它的内存。否则，每次对分数执行数学运算，程序都会泄漏一些内存。

代码清单11-1把新分类的接口和实现部分与测试程序放在一个文件中。前面提到过，这个分类的接口部分可以放在原始的Fraction.h头文件中（这样，所有方法都在一个位置声明），也可以放在自己的头文件中。

如果将分类放到一个主类定义文件中，那么这个类的所有用户都将访问这个分类中的方法。如果不能直接修改原始的头文件（考虑从一个库向现有类添加一个分类，如第二部分“Foundation框架”中所示），除了单独保存它之外，别无选择。

关于分类的一些注意事项

关于分类有几点值得注意。首先，尽管分类可以访问原始类的实例变量，但是它不能添加

自身的任何变量。如果需要添加变量，可以考虑创建子类。

另外，分类可以重载该类中的另一个方法，但是通常认为这种做法是拙劣的设计习惯。其一，重载了一个方法之后，再也不能访问原来的方法。因此，必须小心地将被重载方法中的所有功能复制到替换方法中。如果确实需要重载方法，正确的选择可能是创建子类。如果在子类中重载方法，仍然可以通过向super发送消息来引用父类的方法。因此，不必了解要重载方法的复杂内容，就能够调用父类的方法，并向子类的方法添加自己的功能。

如果喜欢，可以拥有许多分类，只要遵守此处指出的规则。如果一个方法定义在多个分类中，该语句不会指定使用哪个分类。

和一般接口部分不同的是，不必实现分类中的所有方法。这对于程序扩展很有用，因为可以在该分类中声明所有方法，然后在一段时间之后才实现它。

记住，通过使用分类添加新方法来扩展类不仅会影响这个类，同时也会影响它的所有子类。例如，如果为根对象NSObject添加新方法，就存在潜在的危险性，因为每个人都将继承这些新方法，无论你是否愿意。

通过分类为现有类添加新方法可能对你有用，但它们可能和该类的原始设计或意图不一致。例如，通过添加一个新分类和一些方法修改Square类的定义，将Square变成Circle（确实有些夸张），并非好的编程习惯。

同样，对象/分类命名对必须是唯一的。但是，在给定的Objective-C名称空间中，只能存在一个NSString（私有的）分类。这样做可能比较复杂，因为，Objective-C名称空间是程序代码和所有库框架和插件共享的。对于编写屏幕保护首选窗格和其他插件的Objective-C程序员，这尤为重要，因为这些代码将插入到他们无法控制的应用程序或框架代码中。

11.2 协议

协议是多个类共享的一个方法列表。协议中列出的方法没有相应的实现；由其他人来实现（比如你！）。协议提供一种方式来使用指定的名称定义一组多少有点相关的方法。这些方法通常有文档说明，所以你知道它们将如何执行，因此，如果需要，可在自己的类定义中实现它们。

如果决定实现特定协议的所有方法，也就意味着要遵守（confirm to）或者采用（adopt）这项协议。

定义一个协议很简单：只要使用@protocol指令，之后是你给出的协议名称。然后，和处理接口部分一样，声明一些方法。@end指令之前的所有方法声明都是协议的一部分。

如果选择使用Foundation框架，你将发现一些已定义的协议。其中一个名为NSCopying，而且它声明了一个方法，如果你的类要支持使用copy（或者copyWithZone:）方法来复制对象，则必须实现这个方法。我们将在第18章“复制对象”中讲述复制对象的主题。

下面是在标准的Foundation头文件NSObject.h中定义NSCopying协议的方式：

```
@protocol NSCopying
- (id)copyWithZone: (NSZone *)zone;
@end
```

如果你的类采用NSCopying协议，则必须实现名为copyWithZone:的方法。通过在

@interface行的一对尖括号(<...>)内列出协议名称，可以告知编译器你正在采用一个协议。这项协议的名称放在类名和它的父类名称之后，如下所示：

```
@interface AddressBook: NSObject <NSCopying>
```

这说明，AddressBook是父类为NSObject的对象，并且它遵守NSCopying协议。因为系统已经知道以前为这个协议定义的方法（在这个例子中，它是从头文件NSObject.h中得知的），所以不必在接口部分声明这些方法。但是，要在实现部分定义它们。

因此，在这个例子中，在AddressBook的实现部分，编译器期望找到定义的copyWithZone:方法。

如果你的类采用多项协议，只需把它们都列在尖括号中，用逗号分开，如下所示：

```
@interface AddressBook: NSObject <NSCopying, NSCoding>
```

以上代码告知编译器AddressBook类采用NSCopying和NSCoding协议。这次，编译器将期望在AddressBook的实现部分看到为这些协议列出的所有方法的实现。

如果定义了自己的协议，则不必由自己实际实现它。但是，这就告诉其他程序员：如果要采用这项协议，则必须实现这些方法。这些方法可以从超类继承。这样，如果一个类遵守NSCopying协议，则它的子类也遵守NSCopying协议（不过这并不意味着对于该子类，这些方法得到了正确的实现）。

如果希望继承你的类的用户实现一些方法，则可以使用协议定义这些方法。可以为你的GraphicObject类定义一个Drawing协议，并且可以在其中定义paint、erase和outline方法，如下代码所示：

```
@protocol Drawing
-(void) paint;
-(void) erase;
@optional
-(void) outline;
@end
```

作为GraphicObject类的创建者，你不必实现这些绘制方法。但是，需要指定一些方法，要求从GraphicObject创建子类的人实现这些方法，以便符合他要创建的绘图对象的标准。

注意 注意这里使用了@protocol指令。下面列出的所有方法的指令都是可选的。也就是说，采用Drawing方法不一定要实现outline方法来遵守该协议（之后可以通过在协议定义内使用@required指令来列出需要的方法）。

因此，如果创建名为Rectangle 的GraphicObject的子类，并宣传说（也就是，文档说明）这个Rectangle类遵守Drawing协议，那么这个类的用户就知道他们可以向这个类的实例发送paint、erase和outline（可能有）消息。

注意 无论如何，这是理论。编译器允许你声明遵守一项协议，并且只有当你没有实现这些方法时，才发出警告消息。

注意协议不引用任何类，它是无类的（classless）。任何类都可以遵守Drawing协议，不仅

仅是GraphicObject的子类。

可以使用conformsToProtocol:方法检查一个对象是否遵循某项协议。例如，如果有一个名为currentObject的对象，并且想要查看它是否遵循Drawing协议，可以向它发送绘图消息，可以如下编写：

```
id currentObject;
 ...
if ([currentObject conformsToProtocol: @protocol (Drawing)] == YES)
{
  // Send currentObject paint, erase and/or outline msgs
  ...
}
```

这里使用的专用@protocol指令用于获取一个协议名称，并产生一个Protocol对象，conformsToProtocol:方法期望这个对象作为它的参数。

通过在类型名称之后的尖括号中添加协议名称，可以借助编译器的帮助来检查变量的一致性，如下所示：

```
id <Drawing> currentObject;
```

这告知编译器currentObject将包含遵守Drawing协议的对象。如果向currentObject指派静态类型的对象，这个对象不遵守Drawing协议（假定有一个Square对象，它不遵守Drawing协议），编译器将发出一条警告消息，如下所示：

```
warning: class 'Square' does not implement the 'Drawing' protocol
```

这里存在一项编译器校验，所以向currentObject指派一个id变量不会产生这条信息，因为编译器无法知道存储在id变量中的对象是否遵守Drawing协议。

如果这个变量保存的对象遵守多项协议，则可以列出多项协议，如以下代码所示：

```
id <NSCopying, NSCoding> myDocument;
```

定义一项协议时，可以扩展现有协议的定义。所以，以下协议定义

```
@protocol Drawing3D <Drawing>
```

说明Drawing3D协议也采用了Drawing协议。因此，任何采用Drawing3D协议的类都必须实现此协议列出的方法，以及Drawing协议的方法。

最后，分类也可以采用一项协议，如下所示：

```
@interface Fraction (Stuff) <NSCopying, NSCoding>
```

此处，Fraction拥有一个分类Stuff（当然，并非最佳的名称），这个分类采用了NSCopying和NSCoding协议。

和类名一样，协议名必须是唯一的。

非正式协议

你可能在读物中遇到过非正式（informal）协议的概念。它实际上是一个分类，列出了一组方法但并没有实现它们。每个人（或者几乎每个人）都继承相同的根对象，因此非正式分类

通常是为根类定义的。有时，非正式协议也称作抽象（abstract）协议。

如果查看头文件<NSScriptWhoseTests.h>，可能会发现如下所示的一些方法声明：

```
@interface NSObject (NSComparisonMethods)
- (BOOL)isEqualTo:(id)object;
- (BOOL)isLessThanOrEqualTo:(id)object;
- (BOOL)isLessThan:(id)object;
- (BOOL)isGreaterThanOrEqualTo:(id)object;
- (BOOL)isGreaterThan:(id)object;
- (BOOL)isNotEqualTo:(id)object;
- (BOOL)doesContain:(id)object;
- (BOOL)isLike:(NSString *)object;
- (BOOL)isCaseInsensitiveLike:(NSString *)object;
@end
```

这些代码为NSObject类定义了一个名为NSComparisonMethods的分类。这项非正式协议列出了一组方法（这里列出了9个），可以将它们实现为协议的一部分。非正式协议实际上仅仅是一个名称之下的一组方法。这在文档说明和模块化方法时，可能有所帮助。

声明非正式协议的类自己并不实现这些方法，并且选择实现这些方法的子类需要在它的接口部分重新声明这些方法，同时还要实现这些方法中的一个或多个。和正式协议不同，编译器不提供有关非正式协议的帮助；这里没有遵守协议或者由编译器测试这样的概念。

如果一个对象采用正式协议，则它必须遵守协议中的所有信息。这可以在运行时以及编译时强制执行。如果一个对象采用非正式协议，则它可能不需要采用此协议的所有方法，具体取决于这项协议。可以在运行时强制要求遵守一项非正式协议（借助respondsToSelector:），但是在编译时不可以。

注意 前面描述的@optional指令添加到了Objective-C 2.0语言中，用于取代非正式协议的使用。你可以看到几个UIKit类（UIKit是Cocoa Touch框架的一部分）使用了这一点。

11.3 合成对象

你已经学习了通过派生子类和分类等技术扩展类定义的几种方式。有另一项涉及定义一个包含其他类的一个或多个对象的技术。这个新类的对象就是所谓的合成（composite）对象，因为它是由其他对象组成的。

比如，考虑第8章中定义的Square类。将这个类定义为Rectangle的子类，因为你知道正方形就是等边的矩形。定义子类时，它继承了父类的所有实例变量和方法。在一些情况下，这种做法不合适。例如，在父类中定义的一些方法可能不适合子类使用。Square类继承了Rectangle的setWidth:andHeight:方法，但并不适用（即使它能够正常工作）。此外，创建子类时，必须确保所有被继承的方法能够正常工作，因为该类的用户可能会访问它们。

作为创建子类的替代方式，可以定义一个新类，它包含要扩展类的实例变量。然后，只需在新类中定义适合该类的方法。返回Square例子，下面是定义Square的另一种方式：

```
@interface Square: NSObject
```

```
{
   Rectangle *rect;
}
-(int) setSide: (int) s;
-(int) side;
-(int) area;
-(int) perimeter;
@end
```

此处定义的Square类有4个方法。和子类版本不同，子类版本允许你直接访问Rectangle的方法（setWidth:、setHeight:、setWidth:andHeight:、width和height），但是这个Square的定义中不包括这些方法。这样做很有意义，因为处理square时这些方法确实不适合。

如果以这种方式定义Square，就需要为它包含的矩形分配存储空间。例如，如果不重载方法，以下语句

```
Square *mySquare = [[Square alloc] init];
```

分配了一个新的Square对象，但是没有为存储在其实例变量中的Rectangle对象rect分配存储空间。

一个解决方案就是重载init或添加initWithSide:之类的新方法来分配空间。这个方法可以为Rectangle rect分配存储空间，并相应地设置它的边。还需要重载dealloc方法（第8章中学习了如何对Rectangle类进行处理），以便在释放Square时，释放Rectangle rect占用的存储空间。

在Square类中定义方法时，可以使用Rectangle的方法。例如，下面说明了如何实现area方法：

```
-(int) area
{
   return [rect area];
}
```

其他方法的实现作为练习留给你（参见随后的练习5）。

11.4 练习

1. 扩展代码清单11-1中的MathOps分类，使之包含一个invert方法，这个方法返回一个Fraction，它是接收者的倒置。
2. 向类Fraction添加一个名为Comparison的分类。根据以下声明，在这个分类中添加两个方法：

   ```
   -(BOOL) isEqualTo: (Fraction *) f;
   -(int) compare: (Fraction *) f;
   ```

   如果两个分数相同，第一个方法应该返回YES，否则，返回NO。注意分数的比较方式（如，比较3/4和6/8应当返回YES）。

   如果接收者小于参数传递来的分数，则第二个方法应当返回-1，如果二者相等，应返回0，如果接收者大于参数，则应当返回1。
3. 通过添加遵守非正式协议NSComparisonMethods（本章前面所列出的）的方法来扩展Fraction类。根据该协议实现前6个方法（isEqualTo:、isLessThanOrEqualTo:、

isLessThan:、isGreaterThanOrEqualTo:、isGreaterThan:、isNotEqualTo:）并测试它们。

4. 函数sin()、cos()和tan()是Standard Library的一部分（与scanf()一样）。这些函数在头文件<math.h>中声明，应该使用以下语句把这个头文件导入程序：

```
#import <math.h>
```

这些函数分别可以用来计算用弧度表示的double参数的sine、cosine或者tangent值。返回的结果也是一个双精度的浮点值。所以，

```
result = sin (d);
```

可用于计算d的sine值，角度d的值用弧度表示。为第6章“选择结构”中的Calculator类添加一个名为Trig的分类。根据以下声明，为这个分类添加一些方法来计算sine、cosine和tangent的值：

```
-(double) sin;
-(double) cos;
-(double) tan;
```

5. 根据本章对合成对象的讨论以及以下接口部分：

```
@interface Square: NSObject
{
    Rectangle *rect;
}
-(Square*) initWithSide: (int) s;
-(int) setSide: (int) s;
-(int) side;
-(int) area;
-(int) perimeter;
-(void)  dealloc; // Override to release the Rectangle object's memory
@end
```

编写Square的实现部分，以及用来检验其方法的测试程序。

第12章　预处理程序

预处理程序提供了一些工具。使用这些工具能够开发那些更易于开发、阅读、修改以及移植到不同系统的程序。还能使用预处理程序从文字上定制Objective-C语言，以适合特定的编程应用或适应你自己的编程风格。

预处理程序是Objective-C编译过程的一部分，它可以识别散布在程序中的特定语句。顾名思义，预处理程序实际上是在分析Objective-C程序之前处理这些语句。预处理程序语句是以井号(#)标记的，这个符号必须是该行的第一个非空格字符。你将看到，预处理程序语句的语法稍微不同于Objective-C语句。我们将从研究#define语句开始。

12.1　#define语句

#define语句的基本用途之一就是给符号名称指派程序常量。预处理程序语句

```
#define TRUE 1
```

定义了名称TRUE，并使它等于值1。随后，名称TRUE能够用于程序中任何需要常量1的地方。只要出现这个名称，预处理程序自动在该程序中将这个名称替换为预定义的值1。例如，可能遇到下面这条Objective-C语句，该语句使用了预定义的名称TRUE：

```
gameOver = TRUE;
```

这条语句向gameOver指派了值TRUE。你自己无需关注为TRUE定义的确切值。但是因为已经知道它定义为1，所以前面语句的作用就是将1赋给gameOver。预处理程序语句

```
#define FALSE 0
```

定义了名称FALSE，随后在程序中它就等价于0。因此，语句

```
gameOver = FALSE;
```

将FALSE的值赋给gameOver，并且语句

```
if (gameOver == FALSE)
...
```

比较gameOver的值和FALSE的预定义值。

预定义名称不是变量。因此，不能为它赋值，除非替换指定值的结果实际上是一个变量。只要在程序中使用预定义名称，在#define语句中预定义名称右边的所有字符都会被预处理程序自动替换到程序中。这类似于在文本编辑器中进行搜索和替换，在这种情况下，预处理程序将出现的所有预定义名称替换为相应的文本。

你将发现#define语句的语法很特别：将TRUE赋值为1没有用到等号。而且，在语句末尾也没有出现分号。不过你马上就会明白这种特别语法存在的原因。

#define语句经常放在程序的开始，#import或#include语句之后。这并不是必需的，它们可

以出现在程序的任何地方。但是，在程序引用这个名称之前，必须先定义它们。预定义的名称和变量的行为方式不同：没有局部定义之类的说法。在定义一个名称之后，随后就可以在程序的任何地方使用它。大多数程序员把定义放在头文件中，以便在多个源文件中使用它们。

举另一个使用预定义名称的例子。假设想要编写两个方法来计算Circle对象的面积和周长。这两个方法都需要使用常量π，但是它不容易记住，因此，合理的情况是，在程序的开始部分定义该常量的值，然后根据需要在每个方法中使用该值。

所以，可以在程序中包含以下代码：

```
#define PI    3.141592654
```

然后就可以如下所示在两个Circle方法中使用它了（下面假设Circle类中有一个名为radius的实例变量）：

```
-(double) area
{
    return PI * radius * radius;
}

-(double) circumference
{
    return 2.0 * PI * radius;
}
```

给符号名称指派一个常量，每次想在程序中使用它们时，就不必记住这个特定常量的值。此外，如果需要更改常量的值（例如，你可能发现使用了错误的值），则只需要在程序的一个地方更改这个值：那就是在#define语句中。如果没有这种方式，将不得不从头到尾搜索程序，并在使用该值的地方显式地修改这个常量的值。

你可能已经注意到了，目前为止显示的所有定义（TRUE、FALSE和PI）都是大写字母组合。这是为了从视觉上区分预定义的值和变量。一些程序员有这样的习惯：所有预定义名称都用大写，这样就容易区分一个名称是变量名、对象名、类名，还是预定义名称。另一种常见的惯例是在定义之前加字母k。这种情况下，之后的字符并不用全部大写。kMaximumValues和kSignificantDigits是符合这种惯例的两个预定义名称例子。

对常量值使用预定义名称有助于加强程序的可扩展性。例如。学习如何使用数组时，可以不通过硬编码分配数组大小，而是如下定义：

```
#define MAXIMUM_DATA_VALUES 1000
```

这样所有引用都可以以这个数组的大小为基础（如在内存中分配该数组的内存），并且根据这个预定义的值确定数组的有效下标。

并且，假设程序在任何用到数组大小的地方都使用MAXIMUM_DATA_VALUES，如果后来需要改变数组的大小，程序中唯一必须改动的语句就是前面的定义。

12.1.1 更高级的定义类型

名称的定义不仅能够包括简单的常量值。你很快就会看到，它可以包括表达式和其他任何

东西。

下面的语句将名称TWO_PI定义为2.0与3.141592654的积：

```
#define TWO_PI 2.0 * 3.141592654
```

随后就可以在程序中任何表达式2.0*3.141592654有效的地方，使用这个预定义名称。因此，可以使用以下语句替换前面例子中circumference方法的return语句：

```
return TWO_PI * radius ;
```

在Objective-C程序中遇到预定义的名称时，使用#define语句中预定义名称右边的所有字符字面替换程序中该点的名称。因此，当预处理程序遇到前面所示的return语句中的名称TWO_PI时，它使用#define语句中相应于该名称的全部字符替换这个名称。因此，只要程序中出现TWO_PI，预处理程序就将其字面替换为2.0*3.141592654。

预定义名称一出现，预处理程序就执行文本替换。这可以解释为什么通常不能使用分号结束#define语句。如果使用了分号，只要出现预定义名称，分号也将替换到程序中。如果如下定义PI：

```
#define PI    3.141592654;
```

然后这样编写代码：

```
return 2.0 * PI * r;
```

那么预处理程序将使用3.151592654;替换预定义名称PI。因此在预处理程序完成替换之后，编译器将把这条语句看作：

```
return 2.0 * 3.141592654; * r;
```

这会导致语法错误。记住，除非十分确定需要分号，否则不要在定义语句的末尾添加分号。

预处理程序定义的右面不必是合法的Objective-C表达式，只要使用它的时候，结果表达式正确就可以了。例如，可以如下设置定义：

```
#define AND    &&
#define OR     ||
```

然后可以如下编写表达式：

```
if ( x > 0 AND x < 10 )
  ...
```

和

```
if ( y == 0 OR y == value )
  ...
```

甚至可以包含如下定义，用于测试相等性：

```
#define EQUALS ==
```

然后，可以编写如下表达式：

```
if (y EQUALS 0 OR y EQUALS value)
  ...
```

这样就消除使用单个等号“=”错误进行等价判断的可能性。

虽然这些例子显示了#define的强大功能，但是应该注意以这种方式重新定义底层语言语法的行为通常是不好的编程习惯。而且会使其他人难以理解你的代码。

要想更有趣，预定义的值本身可以引用另一个预定义的值。所以，以下两个定义：

```
#define PI     3.141592654
#define TWO_PI  2.0 * PI
```

是完全合法的。名称TWO_PI是按照前面的预定义名称PI定义的，这样就不必重复拼写值3.141592654。

如果把这两个定义的顺序颠倒一下，如下所示：

```
#define TWO_PI  2.0 * PI
#define PI     3.141592654
```

也是合法的。规则就是：只要在程序中使用预定义名称时所有符号都是定义过的，那么就可以在定义中引用其他预定义的值。

合理地使用定义通常可以减少程序中对注释的需要。考虑如下语句：

```
if ( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
...
```

这个表达式检测变量year是不是闰年。现在，考虑以下定义以及后续的if语句：

```
#define IS_LEAP_YEAR  year % 4 == 0 && year % 100 != 0 \
                    || year % 400 == 0
  ...
if ( IS_LEAP_YEAR )
  ...
```

通常，预处理程序假设定义包含在程序的一行中。如果需要第二行，那么上一行的最后一个字符必须是反斜线符号。这个字符告诉预处理程序这里存在一个后续，否则将被忽略。对于多个后续行，也是如此；每个要继续的行都必须以反斜线结尾。

这条if语句远比前面的if语句更容易理解。因为该语句很清楚，所以无需注释。当然，这个定义只能限于测试变量year来判断该年是不是闰年。最好能够编写一个定义，它能够判定任何一年是否是闰年，而不只是变量year。实际上，可以编写带有一个或多个自变量的定义。这就引出下一个讨论要点。

可以将IS_LEAP_YEAR定义为带有一个名为y的参数：

```
#define IS_LEAP_YEAR(y)  y % 4 == 0 && y % 100 != 0 \
                       || y % 400 == 0
```

和方法定义不同，这里没有定义参数y的类型，因为此时仅执行字面文本替换，并没有调用函数。注意，在定义带有参数的名称时，预定义名称和参数列表的左半括号之间不允许空格。

依照前面的定义，可以编写如下语句：

```
if ( IS_LEAP_YEAR (year) )
  ...
```

该语句判断year的值是不是闰年。或者，可以如下编写来测试nextYear的值是不是闰年：

```
if ( IS_LEAP_YEAR (nextYear) )
  ...
```

在前面的语句中，IS_LEAP_YEAR的定义直接替换到if语句中，同时只要定义中出现y，就使用参数nextYear替换它。这样，编译器实际上将这个if语句看作：

```
if ( nextYear % 4 == 0 && nextYear % 100 != 0 || nextYear % 400 == 0 )
  ...
```

预定义（definition）通常称作“宏”。这个术语经常用于带有一个或多个参数的定义。

这个宏名为SQUARE，它简单地将参数乘方：

```
#define SQUARE(x) x * x
```

虽然SQUARE的宏定义简单明了，但是在定义宏时有一个有趣的陷阱，必须小心地避开。我们描述过，语句

```
y = SQUARE (v);
```

把V^2的值赋给y。你认为对于以下语句会发生什么情况？

```
y = SQUARE (v + 1);
```

这个语句并不像你期望的那样，把$(V + 1)^2$的值赋给y。因为预处理程序对宏定义的参数实行文本替换，前面的表达式实际上是如下求值的：

```
y = v + 1 * v + 1;
```

这显然不能得到期望的结果。要正确解决这个问题，需要在SQUARE宏的定义中加入括号：

```
#define SQUARE(x) ((x) * (x))
```

虽然这个定义看起来可能有点奇怪，但要记住定义中任何出现x的任何地方都要使用整个表达式进行字面替换，和SQUARE宏定义的一样。使用新的SQUARE宏定义，语句

```
y = SQUARE(v+1) ;
```

将正确地作为以下表达式进行求值：

```
y = ((v+1)*(v+1));
```

以下的宏允许你方便地根据Fraction类动态地创建新分数：

```
#define MakeFract(x,y) ([[Fraction alloc] initWith: x over: y])
```

然后，可以如下编写表达式：

```
myFract = MakeFract (1, 3); // Make the fraction 1/3
```

或者甚至使用

```
sum = [MakeFract (n1, d1) add: MakeFract (n2, d2)];
```

把分数n1/d1和n2/d2相加。

定义宏时，使用条件表达式的运算符可以非常方便。以下语句定义了一个名为MAX的宏，

它给出两个值的最大值：

```
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
```

这个宏允许随后写出这些语句：

```
limit = MAX(x+y, minValue);
```

这个式子把x+y和minValue的最大值赋给limit。用括号把整个MAX定义括起来是为了确保正确地计算如下表达式：

```
MAX(x,y)*100
```

每个自变量都用括号括起来是为了确保正确地计算如下表达式：

```
MAX(x&y,z)
```

&运算符是按位AND运算符，它的优先级低于宏中使用的>运算符。如果宏定义中没有括号，>运算符将在按位AND之前求值，从而导致错误的结果。

以下宏测试字符是不是小写字母：

```
#define IS_LOWER_CASE(x) ( ((x) >= 'a') && ((x) <= 'z') )
```

因此，允许编写以下表达

```
if ( IS_LOWER_CASE (c) )
   ...
```

甚至可以在另一个宏定义中使用这个宏把字符从小写转换为大写，同时不改变非小写字符：

```
#define TO_UPPER(x) ( IS_LOWER_CASE (x) ? (x) - 'a' + 'A' : (x) )
```

这里再次用到标准ASCII字符集。在第二部分学习Foundation字符串对象时，将看到如何对国际（Unicode）字符集执行大小写转换。

12.1.2 #运算符

如果在宏定义中参数之前放置一个#，那么在调用该宏时，预处理程序将根据宏参数创建C风格的常量字符串。例如，定义：

```
#define str(x) # x
```

使得预处理程序将随后的调用

```
str (testing)
```

扩展为

```
"testing"
```

因此printf调用

```
printf (str (Programming in Objective-C is fun.\n));
```

等价于

```
printf ("Programming in Objective-C is fun.\n");
```

预处理程序在实际的宏参数两侧插入双引号。参数中的任何双引号或反斜线符号都是预处

理程序的保留字符。所以

```
str ("hello")
```

将产生

```
"\"hello\""
```

下面这个宏定义是使用#运算符的更实用例子：

```
#define printint(var) printf (# var " = %i\n", var)
```

该宏用于显示整型变量的值。如果count是值为100的整型变量，那么语句

```
printint (count);
```

将扩展为

```
printf ("count" " = %i\n", count);
```

编译器把两个相邻的字符串连接到一起，形成单个字符串。因此，对两个相邻的字符串执行连接之后，语句将变成下面的样子：

```
printf ("count = %i\n", count);
```

12.1.3 ##运算符

在宏定义中，##这个运算符用于把两个标记（token）连在一起。它的前面（或后面）是宏的参数名称。预处理程序使用调用该宏时提供的实际参数，并且根据该参数和##之后（或之前）的标记创建单个标记。

例如，假设有一个从x1到x100的变量列表，可以编写一个名为printx的宏，它简单地使用一个1～100的整数值作为它的参数，并如下显示对应的x变量：

```
#define printx(n) printf("%i\n", x ## n)
```

该定义的以下部分

```
x ## n
```

表示使用##之前和之后的标记（分别是字母x和参数n），并依据它们构造一个标记。因此调用

```
printx(20);
```

将被扩展成以下表达式：

```
printf("%i\n", x20);
```

printx宏甚至可以使用前面定义的printint宏来获得变量名及其显示的值：

```
#define printx(n) printint(x ## n)
```

调用

```
printx(10);
```

首先扩展为

```
printint(x10);
```

然后扩展为

```
printf ("x10" " = %i\n", x10);
```

最终变成

```
printf ("x10 = %i\n", x10);
```

12.2 #import语句

经过一段时间的Objective-C编程后，你将发现自己开发了一组宏，你想在每一个程序中使用它们。但是不必在编写的每个新程序中键入这些宏，相反，预处理程序允许你将所有定义收集到一个单独文件中，然后使用#import语句把它们包含在程序中。这些文件类似于前面遇到的、但是不必是你自己编写的程序，它们通常以.h结尾，人们将其称为头文件（header file）或包含文件（include file）。

假设你正在编写一系列程序，用于执行各种量度转换。你可能想为执行转换所需要的各种常量设置一些#define语句：

```
#define INCHES_PER_CENTIMETER  0.394
#define CENTIMETERS_PER_INCH  (1 / INCHES_PER_CENTIMETER)

#define QUARTS_PER_LITER       1.057
#define LITERS_PER_QUART      (1 / QUARTS_PER_LITER)

#define OUNCES_PER_GRAM        0.035
#define GRAMS_PER_OUNCE       (1 / OUNCES_PER_GRAM)
  ...
```

假设将前面的定义输入到系统中一个名为metric.h的独立文件中。随后任何需要使用包含在metric.h中定义的程序都只需简单地使用以下预处理程序指令：

```
#import "metric.h"
```

在引用metric.h中的定义之前，必须出现这条语句，并且通常放在源文件的开始处。预处理程序在系统中寻找指定的文件，并且有效地把该文件的内容拷贝到程序中出现#import语句的确切位置。这样，该文件中的所有语句似乎都是直接在程序中该位置键入的。

头文件名两侧的双引号指示预处理程序在一个或者多个文件目录（通常首先在包含源文件的目录中查找，但是通过修改适当的“项目设置”，可以用Xcode指定预处理程序搜索的确切位置）中寻找指定的文件。

把文件名放在<和>字符之间，例如

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

将导致预处理程序只在特殊的“system”头文件目录中寻找包含文件，当前目录不会被搜索。同样，使用Xcode可以通过从菜单中选择“项目”、“编辑项目设置”来更改这些目录。

注意 编译本部分的程序时，我会从系统目录/Developers/SDKs/MacOSX10.5.sdk/System/Library/Frameworks/Foundation.framework/Versions/C/Headers中导入Foundation.h文件。

要查看实际程序例子中如何使用包含文件，将前面给出的6条#define语句键入名为metric.h的文件中，然后以常规方式键入并运行代码清单12-1。

代码清单12-1

```
/* Illustrate the use of the #import statement
  Note: This program assumes that definitions are
  set up in a file called metric.h        */

#import <Foundation/Foundation.h>
#import "metric.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   float liters, gallons;

   NSLog (@ "*** Liters to Gallons *** ");
   NSLog (@ "Enter the number of liters:");
   scanf ("%f", &liters);

   gallons = liters * QUARTS_PER_LITER / 4.0;
   NSLog (@ "%g liters = %g gallons", liters, gallons);

   [pool drain];
   return 0;
}
```

代码清单12-1　输出

```
*** Liters to Gallons ***
Enter the number of liters:
55.75
55.75 liters = 14.7319 gallons.
```

代码清单12-1相当简单，因为它只显示一个预定义值（QUARTS_PER_LITER），该值引自包含文件metric.h。不管怎么样，可以得出这样一点：将定义输入metric.h文件之后，就能够在任何使用合适的#import语句的程序中使用它们。

导入文件最出色的能力之一是它使你能够把定义集中起来，从而确保所有程序引用相同的值。此外，包含文件中发现的任何值错误只需在该位置修改，从而不必更正每个使用该值的程序。任何引用这个错误值的程序只需简单地重新编译一下，而不必重新编辑。

其他系统包含文件包含存储在底层C系统库中的各种函数的声明。例如，limits.h包含一些系统相关值，它们指定各种字符和整型数据类型的大小。例如，在这个文件中，名称INT_MAX定义了int型的最大大小，ULONG_MAX定义了unsigned long int型的最大大小，等等。

头文件float.h给出关于浮点数据类型的信息。例如，FLT_MAX指定了最大的浮点数，

FLT_DIG指定了float类型小数的十进制位的精度数。

文件string.h包含执行字符串操作（如复制、比较和连接）的库例程的声明。如果专门使用Foundation字符串类（将在第15章“数字、字符串和集合”中讨论），那么程序中可能不需要使用这些例程。

12.3 条件编译

Objective-C预处理程序提供了一项名为条件编译（conditional compilation）的功能。

条件编译通常用于创建可以在不同计算机系统上编译运行的程序。它还经常用来开关程序中的各种语句，例如用来输出变量值或跟踪程序执行流程的调试语句。

12.3.1 #ifdef、#endif、#else和#ifndef语句

遗憾的是，程序有时必须依靠系统相关的参数，这些参数需要在不同的处理器（例如，Power PC与英特尔）或特定版本的操作系统（例如，Tiger与Leopard）上分别指定。

如果有大型程序，它对计算机系统的特定硬件和／或软件有很多这样的依赖（应尽可能减少这种依赖性），最终你可能会有许多这样的定义，在程序移植到其他计算机系统时，它们的值不得不更改。

通过利用预处理程序的条件编译能力，能够减少在程序移植过程中不得不更改这些定义的问题，并且能够把每种机器关于这些定义的值结合到程序中。举个简单的例子，如果前面已经定义了符号MAC_OS_X，下面的语句

```
#ifdef MAC_OS_X
#  define DATADIR  "/uxn1/data"
#else
#  define DATADIR  "\usr\data"
#endif
```

就把DATADIR定义为“/uxn1/data”，否则就定义为“\usr\data”。

这里可以看到，允许在标志预处理语句开始的#符号之后放置一个或多个空格。

#ifdef、#else和#endlif语句的行为和你期望的一样。如果程序编译时，#ifdef行中所指定的符号已经通过#define语句或命令行定义了，那么编译器将处理从此处开始到#else、#elif或#endif的程序段，否则就忽略这个程序段。

要为预处理程序定义符号POWER_PC，使用如下语句：

```
#define POWER_PC 1
```

或者仅仅

```
#define POWER_PC
```

就足够了。正如你所见，定义的名称之后没有必要出现文本来满足#ifdef检测。编译器还允许使用编译器命令的特殊选项在程序编译时为预编译器定义名称。命令行：

```
gcc -framework Foundation -D POWER_PC program.m -
```

为预处理程序定义了名称POWER_PC，它使program.m中的所有#ifdef POWER_PC语句都判断

为TRUE（注意，在命令行中，-D POWER_PC必须在程序名称之前键入）。该技术使得不必编辑源程序就可以定义名称。

使用Xcode，通过选择“项目设置”下的“添加用户定义设置”，就可以添加新的预定义名称并指定它们的值。

#ifndef语句与#ifdef在一条线上。这个语句的使用方式类似，只是如果指定的符号没有定义，它就导致程序处理后续行。

前面提到过，在调试程序时，条件编译很有用。你可能在程序中嵌入了很多NSLog调用，用于显示中间结果并跟踪执行流程。如果定义了一个特定名称（如DEBUG），通过将这些语句条件编译到程序中，就可以打开它们。例如，只要程序在编译时定义了名称DEBUG，就可以使用如下一系列语句来显示一些变量的值：

```
#ifdef DEBUG
  NSLog(@"User name = %s, id = %i\n", userName, userId);
#endif
```

程序中可能有很多这样的调试语句。无论何时调试这个程序，都能够通过DEBUG使得所有调试语句都编译。当程序已经能正确工作时，就可以不加DEBUG重新进行编译。这样做还可以削减程序大小，因为没有编译所有调试语句。

12.3.2 #if和#elif预处理程序语句

#if预处理程序语句提供控制条件编译的更通用方法。if语句可以用来检测常量表达式是不是非零。如果表达式的结果非零，就会处理到#else、#elif或#endif为止的所有后续行；否则将跳过它们。

举一个如何使用它的例子，Foundation头文件NSString.h中有以下行：

```
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_5
#define NSMaximumStringLength   (INT_MAX-1)
#endif
```

这将预定义变量MAC_OS_X_VERSION_MIN_REQUIRED的值与预定义变量MAC_OS_X_VERSION_10_5的值进行比较。如果前者小于后者，就处理随后的#define，否则就跳过它。如果程序在MAC OS X 10.5或更高版本上编译，这大概将一个字符串的最大长度设置为整型的最大大小减1。

特殊运算符

```
defined (name)
```

也能够用在#if语句中。预处理程序语句集

```
#if defined (DEBUG)
  ...
#endif
```

和

```
#ifdef DEBUG
```

```
...
#endif
```

的作用相同。

以下语句出现在NSObjcRuntime.h头文件中，用于根据使用的特定编译器定义NS_INLINE（如果之前未定义）：

```
#if !defined(NS_INLINE)
    #if defined(__GNUC__)
        #define NS_INLINE static __inline__attribute__((always_inline))
    #elif defined(__MWERKS__) || defined(__cplusplus)
        #define NS_INLINE static inline
    #elif defined(_MSC_VER)
        #define NS_INLINE static __inline
    #elif defined(__WIN32__)
        #define NS_INLINE static __inline__
    #endif
#endif
```

#if另一种常见用法是在如下代码序列中：

```
#if defined (DEBUG) && DEBUG
...
#endif
```

这使得#if到#endif之间的语句只有在定义了DEBUG而且具有非零值时才被处理。

12.3.3 #undef语句

在一些情况下，可能需要使一些已经定义的名称成为未定义的，通过使用#undef语句就可以这么做。要消除特定名称的定义，编写如下语句：

```
#undef  name
```

这样，语句

```
#undef  POWER_PC
```

将消除POWER_PC的定义。之后的#ifdef POWER_PC或#if defined(POWER_PC)语句都将判断为假。

这里总结了关于预处理程序的讨论。这里没有描述的一些其他预处理程序语句将在附录B“Objective-C 语言总结”中进行描述。

12.4 练习

1. 在机器上找到系统头文件limits.h、和float.h。检查这些文件，看看其内容。如果这些文件包含其他头文件，确保跟踪它们并查看它们的内容。
2. 定义一个名为MIN的宏，它给出两个值的最小值。然后编写一个程序来测试这个宏定义。

3. 定义一个名为MAX3的宏，它给出三个值的最大值。然后编写一个程序来测试这个宏定义。
4. 编写一个名为IS_UPPER_CASE的宏，作用是如果字符是大写字母就给出非零值。
5. 编写一个名为IS_ALPHABETIC的宏，其作用是如果一个字符是字母就给出非零值。使该宏使用在本章定义的IS_LOWER_CASE宏和练习4中定义的IS_UPPER_CASE宏。
6. 编写一个名为IS_DIGIT的宏，其作用是如果字符是0～9的数字就给出非零值。在另一个名为IS_SPECIAL的宏定义中使用它。如果字符是一个特殊字符（也就是说，它既不是字母也不是数字），IS_SPECIAL将给出非零结果。一定要使用练习5中定义的IS_ALPHABETIC宏。
7. 编写一个名为ABSOLUTE_VALUE的宏，其作用是计算参数的绝对值。确保能够正确计算ABSOLUTE_VALUE(x + delta)之类的表达式的值。
8. 考虑本章关于printint宏的定义：

```
#define printx(n) printf("%i\n", x##n)
```

下面的程序能够用来显示x1～x100这100个变量的值吗？为什么？

```
for(i=i;i<=100;++i)
  printx(i);
```

第13章　基本的C语言特性

本章将介绍在编写Objective-C程序时并非必须知道的特性。事实上，这些特性大部分来源于基本的C语言。函数、结构、指针、联合和数组之类的特性最好在必须知道时才学习。因为C语言是一门过程式语言，所以有些特性与面向对象编程的思想是相对立的。这些特性也会妨碍Foundation框架实现的策略，比如内存分配方式或处理包含多字节字符的字符串。

注意　在C级别有多种使用多字节字符的方式，但是Foundation通过其NSString类提供了比较好的解决方案。

另一方面，也许一些应用程序为了优化要求使用底层方法。比如，如果使用大型的数据数组，可能想要使用C语言的内置数组结构，而不是Foundation的数组对象（详见第15章）。如果使用恰当，函数也可以方便地组合重复使用的操作并将程序模块化。

建议你先浏览本章的内容，然后在读完第二部分之后再学习本章。或者是完全忽略本章。并开始学习第二部分。如果后来需要支持其他人编写的代码，或开始研究Foundation框架的头文件，将接触到本章所讲的一些结构。很多Foundation数据类型，如NSRange、NSPoint和NSRect，都要求对本章所讲的结构有一些基本了解。在这些情况下，可以回到本章，并阅读相关部分以了解所需的概念。

13.1　数组

Objective-C提供了一项功能，它允许用户定义一组有序的数据项，即数组。本节将讲述如何定义和操纵数组。在后面几节中，我们会进一步讨论数组，以阐述它们如何与函数、结构、字符串及指针一起使用。

假设你有一组成绩要录入计算机，并要对这些成绩执行一些操作，比如按升序排列、计算平均值或查找中值。在排序过程中，除非录入所有成绩，否则排序将无法进行。

在Objective-C中，可以定义一个名为grades的变量，它代表的不是一个成绩值，而是一组成绩值。然后通过名为索引或下标的数字来引用其中各个元素。在数学上，有下标的变量x_i，指的是x集合中的第i个元素；而在Objective-C语言中，类似的符号为：

```
x[i]
```

所以表达式

```
grades[5]
```

（读做“grades sub 5”）指的是名为grades的数组中索引为5的元素。在Objective-C语言中，数组元素以0索引开头，所以

```
grades[0]
```

实际上指的是该数组的第一个元素。

任何一个数组元素都可以用在可用常规变量的地方。比如，可以使用以下语句将数组值赋给另一个变量：

```
g = grades[50];
```

此表达式使用grades[50]的值，并将它赋给变量g。更广泛地说，如果将i声明为整型变量，那么语句

```
g = grades[i];
```

使用索引为i的grades数组元素的值，并将其指定给g。

通过在等号的左侧指定数组元素，就可以将数值存储到数组元素中。下面的语句

```
grades[100] = 95;
```

将数值95存储到grades数组中索引为100的元素中。

通过改变作为数组下标的变量值，可以轻松地浏览数组中的元素。因此，如下for循环

```
for ( i = 0; i < 100; ++i )
  sum += grades[i];
```

依次浏览数组grades的前100个元素（元素0～99），并且将每个成绩相加之和赋给变量sum。当for循环结束时，变量sum包含数组grades的前100个元素值之和（假设在循环开始之前sum值设为0）。

和其他变量类型一样，必须在使用之前先声明数组。数组的声明涉及声明数组所包含元素的数值类型，如int、float或者对象，以及将存储在数组中的最大元素数目。

定义

```
Fraction *fracts[100];
```

规定fracts为包含100个分数的数组。通过使用下标0～99，可以有效地引用该数组的元素。

表达式

```
fracts[2] = [fracts[0] add: fracts[1]];
```

调用Fraction类的add:方法：将数组fracts的前两个分数相加，并将结果存储到数组第三个位置。

代码清单13-1产生斐波纳契数的前15个值列表，尝试预测输出结果，表中每个数值之间有什么关系？

代码清单13-1

```
// Program to generate the first 15 Fibonacci numbers
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int Fibonacci[15], i;

   Fibonacci[0] = 0; /* by definition */
   Fibonacci[1] = 1; /* ditto */
```

```
    for ( i = 2; i < 15; ++i )
        Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

    for ( i = 0; i < 15; ++i )
        NSLog (@ "%i", Fibonacci[i]);

    [pool drain];
    return 0;
}
```

代码清单13-1 输出

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

前两个斐波纳契数，我们称之为F_0和F_1，分别定义为0和1。此后的每个斐波纳契数F_i都定义为前两个斐波纳契数F_{i-2}和F_{i-1}之和。所以，F_0和F_1数值之和是F_2的值。对于前面的程序，通过计算Fibonacci[0]和Fibonacci[1]之和，就可以直接计算出Fibonacci[2]。这个计算公式是在for循环中执行的，它计算出F_2到F_{14}的值（或者，相当于Fibonacci[2]到Fibonacci[14]的值）。

13.1.1 数组元素的初始化

在声明变量时可以给它赋初值，也可以给数组元素赋初值。通过简单地列出数组元素的初值，并以第一个元素开始，就可以实现。列表中的值由逗号隔开，并且整个列表放于一对大括号之内。

语句

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

将0赋给integers[0]，1赋给integers[1]，2赋给integers[2]，依此类推。

字符数组以同样的方式初始化，所以表达式

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

定义了字符数组letters，并将5个元素分别初始化为字符‘a’、‘b’、‘c’、‘d’和‘e’。

不必完全初始化整个数组。如果指定了较少的初始化值，那么只初始化等量的元素，并且数组中其余元素被设为0。因此，声明

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

将数组sample_data的前3个元素初始化为100.0、300.0和500.5，其余497个元素被设为0。

通过将元素编号放在一对大括号中可以以任何顺序初始化指定的数组元素。比如

```
int x = 1233;
int a[] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

定义了一个名为a的10个元素数组（根据数组中最大索引得出的），并将数组的最后一个元素初始化为x+1（1234）。此外，它的前3个元素分别被初始化为1、2和3。

13.1.2 字符数组

代码清单13-2的目的是阐明如何使用字符数组。然而，下面有一个值得讨论的问题，你能发现它吗？

代码清单13-2

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

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   char word[] = { 'H', 'e', 'l ,  l', 'o', '! };
   int i;

   for ( i = 0; i < 6; ++i )
      NSLog (@ "%c", word[i]);

   [pool drain];
   return 0;
}
```

代码清单13-2　输出

```
H
e
l
l
o
!
```

上面的程序最值得注意的一点是字符数组word的声明。它没有指出数组中的元素个数。Objective-C语言允许定义没有指明元素个数的数组。如果是这样定义的，则自动根据初始化元素的数目确定该数组大小。因为代码清单13-2为数组word列出了6个初始值，所以Objective-C

语言隐式地将该数组定义为6元素。

只要在定义数组时初始化了数组中的每个元素，这种方式就没有问题。但如果不是这种情况，就必须显式地给出数组的大小。

如果在字符数组结尾添加一个终止空字符（'\0'），就产生了一个通常称为字符串的变量。如果将代码清单13-2中数组word的初始化语句替换为

```
char word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

随后，就可以如下使用printf语句显示这个字符串：

```
NSLog(@"%s", word);
```

因为%s格式字符告诉printf持续显示字符直到到达终止空字符，也就是在word数组最后添加的字符，所以该语句没有问题。

13.1.3 多维数组

到目前为止看到的数组都是线性数组，也就是，它们都是一维的。C语言允许定义任意维的数组。本节将讲述二维数组。

二维数组最自然的应用之一是矩阵。考虑下面的4×5矩阵：

10	5	−3	17	82
9	0	0	8	−7
32	20	1	0	14
0	0	8	7	6

在数学中，通常通过双下标来引用矩阵中的元素。如果将上面的矩阵命名为M，符号$M_{i,j}$则代表在第i行第j列的元素，其中i的范围从1到4，j的范围从1到5。符号$M_{3,2}$指的是值20，它位于矩阵第3行第2列。类似地，$M_{4,5}$指的是第4行第5列的元素（值为6）。

在Objective-C语言中，在引用二维数组的元素时也使用类似的概念。然而，因为Objective-C语言都是从0开始计数的，所以矩阵的第一行实际上是0行，而第一列是0列。那么上面矩阵的行列标识如下图表所示：

行（i） 列（j）

	0	**1**	**2**	**3**	**4**
0	10	5	−3	17	82
1	9	0	0	8	−7
2	32	20	1	0	14
3	0	0	8	7	6

在数学上使用符号$M_{i,j}$，而Ojective-C语言中使用等价的符号：

```
M[i][j]
```

记住，第一个索引指的是行数，而第二个索引指的是列数。因此，语句

```
sum = M[0][2] + M[2][4];
```

将0行2列的数值（-3）和2行4列的数值（14）相加，并将结果11赋给变量sum。

定义二维数组的方式和一维数组的相同，于是

```
int M[4][5];
```

声明M为4行5列的二维数组，总共包含20个元素。数组中的每个位置都包含整型值。

初始化二维数组的方法类似于一维等价数组的初始化方式。列出初始化元素时，它们的值是根据行排列的。大括号对用来分隔初始化的各行。因此，要将数组M定义和初始化为前面表中列出的元素，可以使用下面的表达式：

```
int M[4][5] = {
                { 10, 5, -3, 17, 82 },
                { 9, 0, 0, 8, -7 },
                { 32, 20, 1, 0, 14 },
                { 0, 0, 8, 7, 6 }
    };
```

特别注意上一个语句的语法。除了最后一行，每行结束的大括号之后必须加上逗号。实际上，内层括号可以省略。如果没有内层括号，将按行进行初始化。所以上一个语句可以如下编写：

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
                20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

和一维数组一样，也不必初始化整个二维数组。如下语句

```
int M[4][5] = {
        { 10, 5, -3 },
        { 9, 0, 0 },
        { 32, 20, 1 },
        { 0, 0, 8 }
    };
```

只初始化了矩阵每行的前三个元素。其余的值都设为0。注意，在这种情况下，需要内层花括号对，以强制正确地初始化。如果没有这些括号，将初始化前两行和第三行的前两个元素（自己验证这种说法。）

13.2 函数

到目前为止，每个程序中接触到的NSLog例程就是一个函数例子。事实上，每个程序还用到一个名为main的函数。回顾你编写的第一个程序（代码清单2-1），它在终端显示“programming is fun.”这个短语：

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSLog (@ "Programming is fun.");
    [pool drain];
```

```
    return 0;
}
```

下面是名为printMessage的函数，它产生同样的结果：

```
void printMessage (void)
{
    NSLog (@ "Programming is fun.");
}
```

函数printMessage和代码清单2-1 main函数的唯一区别在第一行。第一行函数定义告诉编译器关于该函数的4件事情：

- 谁可以调用这个函数；
- 函数的返回值类型；
- 函数的名称；
- 函数使用的参数数目和类型。

函数printMessage的第一行告诉编译器printMessage是这个函数的名称，并且该函数不返回任何值（第一个void关键字的用途）。与方法不同，不必将函数的返回值类型放在一对圆括号中。事实上，如果这么做，将得到一条编译错误消息！

告知编译器printMessage不返回任何值之后，第二个void关键字的用途说明该函数没有任何参数。

回忆一下，在Objective-C系统中，main是一个特别的名称，它总是指示程序开始运行的位置。每个程序都必须有一个main函数。所以，可以向前面的代码添加一个main函数，以完成这个程序，如代码清单13-3所示。

代码清单13-3

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

void printMessage (void)
{
   NSLog (@ "Programming is fun.");
}

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   printMessage ();
   [pool drain];
   return 0;
}
```

代码清单13-3 输出

```
Programming is fun.
```

代码清单13-3包含两个函数：printMessage和main。前面提到过，调用函数并不是新的概念。因为printMessage不带有参数，所以可以简单地在名称后面添加一对左右圆括号来调用它。

13.2.1 参数和局部变量

在第5章“循环结构”中，开发了一个用于计算三角数的程序。这里你将定义一个产生三角数的函数，并且恰当地将其命名为calculateTriangularNumber。通过该函数的一个参数，指定要计算哪个三角数。然后这个函数计算出所求的数值并显示结果。代码清单13-4显示了完成这项任务的函数和测试它的main例程。

代码清单13-4

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

// Function to calculate the nth triangular number

void calculateTriangularNumber (int n)
{
   int i, triangularNumber = 0;

    for ( i = 1; i <= n; ++i )
        triangularNumber += i;

    NSLog (@ "Triangular number %i is %i", n, triangularNumber);
}

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);

    [pool drain];
    return 0;
}
```

代码清单13-4　输出

```
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

函数calculateTriangularNumber的第一行是

```
void calculateTriangularNumber (int n)
```

它告知编译器calculateTriangularNumber是一个函数，它不返回任何值（关键字void）并且带有一个名为n的int参数。再次注意，不能像编写方法那样，将参数类型放在圆括号中。

左花括号表示函数定义的开始。因为你想要计算第n个三角数，所以必须设置一个变量，以便在计算过程中储存三角数的值。还需要一个变量作为循环的索引。为此，定义了变量TriangularNumber和i，并将其声明为整型变量。这些变量的定义及初始化方式和在前面程序的main函数中定义和初始化变量的方式相同。

函数中局部变量的行为同方法中的一样：如果在函数内给变量赋予初始值，那么每次调用该函数时，都会指定相同的初始值。

在函数中（和在方法中一样）定义的变量称为自动局部变量，因为每次调用该函数时，它们都自动“创建”，并且它们的值对于函数来说是局部的。

静态局部变量用关键字static声明，它们的值在函数调用的过程中保留下来，并且初始值默认为0。

局部变量的值只能在定义该变量的函数中访问。它的值不能从函数之外访问。

回到我们的程序示例，定义局部变量之后，该函数计算三角数并且在终端显示结果。右花括号表示函数结束。

在main函数中，数值10是在第一次调用函数calculateTriangularNumber时作为参数传递的。然后，执行直接转换到该函数，数值10成为函数中形参n的值。然后，该函数计算出第10个三角数的值并显示结果。

再次调用函数calculateTriangularNumber时，传递参数20。经过与前面描述相似的过程，该数值成为函数中n值。然后，该函数计算出第20个三角数的值并显示答案。

13.2.2 函数的返回结果

和方法一样，函数也可以返回值。Return语句返回的值类型必须和函数声明的返回类型一致。如下函数

```
float kmh_to_mph (float km_speed)
```

定义了函数kmh_to_mph，它使用一个名为km_speed的float参数，并返回浮点型小数。类似地，

```
int gcd (int u, int v)
```

定义了一个名为gcd的函数，它传递整型参数u和v并返回一个整型值。

使用函数重新编写代码清单5-7中求最大公约数的算法。该函数的两个参数是想要计算最大公约数（gcd）的两个数（参见代码清单13-5）。

代码清单13-5

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

// This function finds the greatest common divisor of two
//  nonnegative integer values and returns the result

int gcd (int u, int v)
```

```
{
    int temp;

    while ( v != 0 )
    {
       temp = u % v;
       u = v;
       v = temp;
    }

   return u;
}

main ()
{
     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
     int result;

     result = gcd (150, 35);
     NSLog (@"The gcd of 150 and 35 is %i", result);

     result = gcd (1026, 405);
     NSLog (@"The gcd of 1026 and 405 is %i", result);

     NSLog (@"The gcd of 83 and 240 is %i", gcd (83, 240));
     [pool drain];
     return 0;
}
```

代码清单13-5 输出

```
The gcd of 150 and 35 is 5
The gcd of 1026 and 405 is 27
The gcd of 83 and 240 is 1
```

函数gcd规定带有两个整型参数。该函数通过形参名称u和v来指明这些参数。将变量temp声明为整型之后，该程序将在终端显示参数u、v的值和相关消息。然后，这个函数计算并返回这两个整数的最大公约数。

表达式

```
result = gcd (150, 35);
```

使用参数150和35来调用函数gcd，并且将返回值存储到变量result中。

倘若省略函数的返回类型声明，如果该函数确实返回任何值，编译器就会假设该值为整数。许多程序员利用这个事实，省略返回整数的函数返回类型声明。但是，这是不好的编程习惯，应该避免。

函数的默认返回类型与方法的默认返回类型不同。回想一下，如果没有为方法指定返回类

型，编译器就假设它返回id类型的值。同样，应该总是声明方法的返回类型，而不是依赖于这个事实。

声明返回类型和参数类型

前面提到过Ojective-C语言编译器假设函数的默认返回值是整型值。更确切地说，无论什么时候调用一个函数，编译器都会假设这个函数的返回类型为int，除非发生以下两种情况之一：

- 在遇到函数调用之前，已经在程序中定义了该函数。
- 在遇到函数调用之前，已经声明了该函数的返回值类型。声明函数的返回值类型和参数类型称为原型（prototype）声明。

函数声明不仅用于声明函数的返回类型，而且用于告知编译器，该函数带有多少参数及其类型。这类似于定义新类时在@interface部分中声明方法。

要将absoluteValue定义为一个返回float型值并带有一个float类型参数的函数，可以使用以下原型声明：

```
float absoluteValue (float);
```

可以看到，只需要在圆括号中指定参数类型，而不是参数名称。如果愿意，可以选择在类型之后指定“伪”名称：

```
float absoluteValue (float x);
```

这个名称和函数定义所用到的不必相同，反正编译器会忽略它。

编写原型声明最简单的方法是简单地复制函数实际定义的第一行代码。记住在结尾处添加分号。

如果函数的参数数目不定（比如NSLog和scanf），必须告知编译器。如下声明

```
void NSLog (NSString *format, ...);
```

告知编译器：NSLog将使用一个NSString对象作为它的第一个参数，之后是任意数目的附加参数（...的用途）。NSLog和scanf在一个特殊文件Foundation/Foundation.h[⊖]中声明，这就是为什么要在每个程序的开头添加下面一行语句。

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

如果没有这一行语句，编译器可以假设NSLog使用固定数目的参数，这可以导致产生不正确的代码。

只有已经在调用函数之前添加了函数的定义或声明了该函数及其参数类型时，编译器才会在调用该函数时自动将参数转换成相应的类型。

下面是关于函数的一些注意事项和建议：

- 默认情况下，编译器假设函数返回int。
- 定义返回值为int的函数时，直接将它定义为int。
- 当定义没有返回值的函数时，将它定义为void。
- 只有当前面已经定义或声明了这个函数，编译器才会将参数转换成函数认可的类型。

⊖ 从技术上讲，是在文件NSObjCRuntime.h中定义，该文件是从文件Foundation.h内部导入的。

为了安全起见，在程序中声明所有函数，即使它们在被调用之前已经定义了（将来你可能决定将这些函数移到文件的其他位置或者移到另一个文件中）。好的策略是将函数声明放到一个头文件中，然后只将这个头文件导入（import）你的模块即可。

默认情况下，函数是外部的。即，函数的默认作用域是任何与该函数链接在一起的文件中的任何函数或方法都可以调用它。通过将其定义为static（静态）可以限制函数的作用域。将关键字static放在函数声明前即可，如下所示：

```
static int gcd (int u, int v)
{
   ...
}
```

静态函数只可以由和该函数定义位于同一文件的其他函数或者方法调用。

13.2.3 函数、方法和数组

要向函数或方法传递单个数组元素，使用常规方式将数组元素指定为参数。因此，如果有一个用来计算平方根的函数squareRoot，并且想要计算averages[i]的平方根并将结果赋给名为sq_root_result的变量，使用如下表达式即可：

```
sq_root_result = squareRoot (averages[i]);
```

向函数或方法传递整个数组是完全不同的情况。要传递数组，只需在函数调用或者方法调用中列出数组名称，并且不需要任何下标。举个例子，如果假设前面将grade_scores定义为包含100个元素的数组，那么下面的表达式：

```
minimum (grade_scores)
```

实际上将数组grade_scores中的100元素都传递给名为minimum的函数。很自然，从另一方面讲，函数minimum必须使用整个数组作为参数，也必须有适当的形参声明。

下面有一个函数，它寻找包含指定元素个数的数组中的最小整数值：

```
// Function to find the minimum in an array

int minimum (int values[], int numElements)
{
    int minValue, i;

    minValue = values[0];

    for ( i = 1; i < numElements; ++i )
        if ( values[i] < minValue )
           minValue = values[i];

    return (minValue);
}
```

函数minimum定义为带有两个参数：第一个是要查找最小数的数组，第二个是数组中的元

素个数。在函数头中，values之后的一对方括号用来告知Objective-C编译器：values是整型数组。编译器并不关心这个数组有多大。

形参numElements用做for语句的上限。这样，for语句依次查找values[1]到数组最后一个元素，即values[numElements-1]。

如果函数或者方法更改了数组元素的值，那么这个变化将影响到传递到该函数或方法的原始数组，而且这个变化在函数或方法执行完之后依然有效。

值得讨论一下数组的行为和单个变量或数组元素（函数或方法不能更改它们的值）不同的原因。我们说过，调用函数或方法时，作为参数传递的值将被复制到相应的形参中。这个论述依然是有效的。但是，使用数组时，并非将整个数组的内容复制到形参数组中。而是传递一个指针，它表示数组所在的计算机内存地址。所以，对形参数组所作的所有更改实际上都是对原始数组而不是数组的副本执行的。因此，函数或方法返回时，这些变化仍然有效。

多维数组

多维数组元素可以像任何普通变量或一维数组元素那样传递给函数或方法。语句

```
result = squareRoot (matrix[i][j]);
```

调用squareRoot函数，同时传递matrix[i][j]中包含的值作为参数。

整个多维数组可以像一维数组那样当作参数传递：只要列出数组名称即可。比如，如果将矩阵measuredValues声明为整型二维数组，那么Objective-C语句

```
scalarMultiply (measuredValues, constant);
```

可以用来调用一个函数，它通过值constant求出该矩阵每个元素的乘积。当然这暗示该函数本身可以更改数组measuredValues中的值。关于一维数组的讨论在这里也适用：在函数中任何对形参数组元素的赋值操作都会永久地更改向该函数传递的数组。

将一个单维数组声明为形参时，规定不需指定数组的实际大小。简单地使用一对空方括号来告知编译器这个参数实际上是个数组就足够了。这并不完全适用于多维数组的情况。对于二维数组，数组的行数可以省略，但是声明必须包括数组的列数。如下声明

```
int arrayValues[100][50]
```

和

```
int arrayValues[][50]
```

对于100行50列的形参数组arrayValues都是合法声明；但是下面的声明

```
int arrayValues[100][]
```

和

```
int arrayValues[][]
```

就不是合法声明，因为必须指明数组的列数。

13.3 结构

除了数组之外，Objective-C语言还提供了另一种组合元素的工具。结构就是这种工具，它构成了本节讨论的基础。

假设要在程序中存储日期（比如说7/18/03），它也许用于程序输出的开头或是出于计算需要。存储日期的自然方法就是：将一个月份赋给名为month的整型变量，日期赋给整型变量day，年份赋给整型变量year。那么语句

```
int month = 7, day = 18, year = 2009;
```

可以实现该任务，这是完全可接受的方式。但是，如果你的程序还需要存储很多日期，该怎么办？如果可以使用某种方式将这三个变量组合起来就更好了。

在Objective-C语言中，可以定义一个名为date的结构，它包含三个分别代表年、月、日的成分。这种定义的语法相当直观，如下所示：

```
struct date
{
   int month;
   int day;
   int year;
};
```

date结构恰好定义了三个整型元素，分别名为month、day和year。在某种意义上说，date的定义指出了Objective-C语言中的新类型，这是由于，随后就可以如下将变量声明为struct date类型了：

```
struct date today;
```

还可以如下另外定义一个名为purchaseDate的同类型变量：

```
struct date purchaseDate;
```

或者，可以直接在同一行中包含两个定义，如下面一行所示：

```
struct date today, purchaseDate;
```

不同于int、float或char型变量，处理结构变量时需要特殊语法。通过指明变量名称，在之后加上句点（称为点运算符）来访问结构成员。举个例子，要将变量today的day值设置为21，可以编写如下语句：

```
today.day = 21;
```

注意变量名、句点和成员名称之间不允许出现空格。

看一下，我们使用同样的运算符来调用对象的属性。回忆一下，我们可以编写语句

```
myRect.width = 12;
```

来调用Rectangle对象的setter方法（称为setWidth），给它传递参数值12。这里的意思很明了：编译器确定点运算符左边的是一个结构还是一个对象，然后进行相应的处理。

返回到struct date示例，要将today中的year赋值为2010，可以使用如下表达式

```
today.year = 2010;
```

最后，要检测month的值是不是等于12，可以使用如下表达式：

```
if ( today.month == 12 )
next_month = 1;
```

代码清单13-6将前面的讨论合成一个实际的程序。

代码清单13-6

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    struct date
    {
        int month;
        int day;
        int year;
    };

    struct date today;

    today.month = 9;
    today.day = 25;
    today.year = 2009;

    NSLog (@ "Today's date is %i/%i/%.2i.", today.month,
              today.day, today.year % 100);

    [pool drain];
    return 0;
}
```

代码清单13-6 输出

```
Today's date is 9/25/09.
```

main函数的第一条语句定义了名为date的结构，它包含三个整型成员，分别是month、day和year。在第二条语句中，变量today声明为struct date类型。所以，第一条语句只是简单地向Objective-C编译器说明了date结构的外观，并没有在计算机中分配存储空间。第二条语句定义了一个struct date类型的变量，所以导致内存中分配了空间，以便存储结构变量today的三个整型成员。

在赋值完成后，通过调用适当的NSLog语句来显示包含在结构中的值。Today.year除以100的余数是在传递给NSLog函数之前计算的，这样可以使年份只显示09。NSLog中的%.2i格式符号指明了最少显示两位字符，从而强制显示年份开头的0。

谈到表达式的求值时，结构成员遵循的法则和Objective-C语言中普通变量一样。这样，将整型的结构成员除以整数的除法和整数除法一样，如下所示：

```
century = today.year / 100 + 1;
```

假设要编写一个简单的程序，它接收今天的日期作为输入数据，并向用户显示明天的日期。第一眼看上去，这似乎是一项非常简单的任务。可以让用户输入今天的日期，然后通过一系列语句计算出明天的日期，如下：

```
tomorrow.month = today.month;
tomorrow.day   = today.day + 1;
tomorrow.year  = today.year;
```

当然，对于大多数日期来讲，上面的语句都可以得出正确结果，但是不能正确处理以下两种情况：

- 如果今天的日期是一个月的最后一天；
- 如果今天的日期是一年的最后一天（即，今天的日期是12月31日）。

确定今天的日期是不是一个月最后一天的一种简便方法是设置对应于每月天数的整型数组。在数组中对查找特定月份就可以得到当月的天数（参见代码清单13-7）。

代码清单13-7

```
// Program to determine tomorrow's date

#import <Foundation/Foundation.h>

struct date
{
    int month;
    int day;
    int year;
};

// Function to calculate tomorrow's date

struct date dateUpdate (struct date today)
{
    struct date tomorrow;
    int numberOfDays (struct date d);

    if ( today.day != numberOfDays (today) )
    {
       tomorrow.day = today.day + 1;
       tomorrow.month = today.month;
       tomorrow.year = today.year;
    }
    else if ( today.month == 12 ) // end of year
    {
       tomorrow.day = 1;
       tomorrow.month = 1;
       tomorrow.year = today.year + 1;
    }
```

```
    else
    {                   // end of month
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    return (tomorrow);
}

// Function to find the number of days in a month

int numberOfDays (struct date d)
{
    int answer;
    BOOL isLeapYear (struct date d);
    int daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == YES && d.month == 2 )
        answer = 29;
    else
        answer = daysPerMonth[d.month - 1];
    return (answer);
}

// Function to determine if it's a leap year

BOOL isLeapYear (struct date d)
{
if ( (d.year % 4 == 0 && d.year % 100 != 0) ||
        d.year % 400 == 0 )
    return YES;
 else
    return NO;
}

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    struct date dateUpdate (struct date today);
    struct date thisDay, nextDay;

    NSLog (@ "Enter today's date (mm dd yyyy):");
    scanf ( "%i%i%i ", &thisDay.month, &thisDay.day,
                &thisDay.year);
```

```
    nextDay = dateUpdate (thisDay);

    NSLog (@ "Tomorrow's date is %i/%i/%.2i.",nextDay.month,
                nextDay.day, nextDay.year % 100);

    [pool drain];
    return 0;
}
```

代码清单13-7　输出

```
Enter today's date (mm dd yyyy):
2 28 2012
Tomorrow's date is 2/29/12.
```

代码清单13-7　输出（重新运行）

```
Enter today's date (mm dd yyyy):
10 2 2009
Tomorrow's date is 10/3/09.
```

代码清单13-7　输出（重新运行）

```
Enter today's date (mm dd yyyy):
12 31 2010
Tomorrow's date is 1/1/10.
```

即使没有用到该程序中的任何类，仍然导入了Foundation.h这个文件，因为你想要使用BOOL类型，并定义YES和NO。它们定义在Foundation.h文件中。

注意date结构的定义最先出现并且在所有函数之外。这是因为结构定义的行为与变量定义非常类似：如果在特定函数中定义结构，那么只有这个函数知道它的存在。这是一个局部结构定义。如果将结构定义在函数之外，那么该定义是全局的。使用全局结构定义，该程序中随后定义的任何变量（不论是在函数之内或之外）都可以声明为这种结构类型。多个文件共用的结构定义都集中放在一个头文件中，然后想要使用这些结构的文件导入这个头文件。

在main例程中，如下声明

```
struct date dateUpdate (struct date today);
```

告知编译器函数dateUpdate使用date结构作为它的参数，并且返回date结构的值。这里并不需要这个声明，因为编译器已经在文件中知道了实际的函数定义。然而，这仍然是好的编程习惯。比如，如果将函数定义和main函数分别放在不同的源文件中，那么在这种情况下，声明将是必要的。

和普通的变量一样（与数组不同），函数对于结构参数的任何更改都不会影响原结构。这些变化只影响到调用该函数时所产生的结构副本。

将日期输入并存储在date结构变量thisDay中之后，如下调用函数dateUpdate：

```
nextDay = dateUpdate (thisDay);
```

这个语句调用函数dateUpdate，同时传递date结构变量thisDay的值。

在dateUpdate function中，原型声明

```
int numberOfDays (struct date d);
```

告诉Objective-C编译器函数numberOfDays返回一个整型值，并且带有一个date结构类型的参数。

语句

```
if ( today.day != numberOfDays (today) )
```

指明today结构将作为参数传递给函数numberOfDays。在这个函数中，必须进行适当声明来告知系统参数的类型是一个结构，如下所示：

```
int numberOfDays (struct date d)
```

函数numberOfDays首先确定该日期是否为闰年并且是否为二月。前一判断是通过调用另一个名为isLeapYear的函数实现的。

函数isLeapYear非常直观。它简单地测试作为参数传递来的date结构中所包含的年份信息，如果是闰年则返回YES，反之返回NO。

一定要理解代码清单13-7中函数调用的层次结构：main函数调用dateUpdate函数，dateUpdate又调用numberOfDays函数，numberOfDays调用函数isLeapYear。

13.3.1 结构的初始化

初始化结构与初始化数组类似，将元素列在一对花括号之中，元素之间以逗号相隔。

要将date结构的变量初始化为2011年7月2日，可以使用下面的语句：

```
struct date today = { 7, 2, 2011 };
```

和数组的初始化一样，列出的值可以少于结构中包含的元素个数。所以，语句

```
struct date today = { 7 };
```

将today.month初始化为7，但是没有给today.day或者today.year赋初值。在这种情况下，它们的默认初始值是未定义的（undefined）。

在初始化列表中，用下面的表示方式

```
.member = value
```

可以以任意顺序初始化结构中指定的成员，如下所示：

```
struct date today = { .month = 7, .day = 2, .year = 2011 };
```

和

```
struct date today = { .year = 2011 };
```

最后一个语句只将该结构中的year元素设置为2011。你知道，其余两个元素是未定义的。

13.3.2 结构数组

结构数组的使用方法非常直观。定义

```
struct date birthdays[15];
```

说明数组birthdays中包含15个date结构类型的元素。引用数组特定结构元素的方法非常简单。要将数组birthdays中的第二个生日赋值为1996年2月22日，使用以下语句序列即可：

```
birthdays[1].month = 2;
birthdays[1].day = 22;
birthdays[1].year = 1996;
```

语句

```
n = numberOfDays (birthdays[0]);
```

将数组中第一个日期发送给函数numberOfDays，以找出该日期指定的月份有多少天。

13.3.3 结构中的结构

Objective-C语言在定义结构方面提供了极大的灵活性。比如，可以定义一个结构，它本身包含其他结构作为自己的一个或多个成员，或者可以定义包含数组的结构。

你已经知道如何将月、日和年逻辑地组合在名为date的结构中。假设有一个名为time的类似结构，用来组合小时、分钟和秒以表示时间。在一些应用程序中，也许需要在逻辑上将日期和时间组合在一起。比如，可能要设置一组在特定日期和时间发生的事件列表。

前面的讨论暗示你想要找到一种简便的方式将日期和时间结合在一起。在Objective-C语言中，通过定义新结构（可能命名为date_and_time）可以实现该任务。这个结构包含两个元素：日期和时间。下面给出定义：

```
struct date_and_time
{
    struct date sdate;
    struct time stime;
};
```

该结构的第一个成员是struct date类型的，名为sdate，第二个成员是struct time类型的，名为stime。date_and_time结构的定义要求先向编译器定义date结构和time结构。

现在，可以将变量定义为date_and_time结构了，如下所示：

```
struct date_and_time event;
```

要引用变量event中的date结构，语法是一样的：

```
event.sdate
```

这样，就可以使用一条语句，将该日期作为参数调用函数dateUpdate，然后将结果赋值。如下所示：

```
event.sdate = dateUpdate (event.sdate);
```

可以对date_and_time结构中的time结构进行同样的操作：

```
event.stime = timeupdate (event.stime);
```

要引用这些结构中的一个成员，需要在成员名称之后添加句点：

```
event.sdate.month = 10;
```

这个表达式将event包含的date结构中的month成员设置为10月，下面的语句

```
++event.stime.seconds;
```

将time结构中的seconds成员加1。

变量event可以用如下方式初始化：

```
struct date_and_time event =
     { { 12, 17, 1989 }, { 3, 30, 0 } };
```

该语句将变量event中的date成员设置为1989年12月17日，并且将time成员设置为3点30分0秒。

很自然，可以设置一个date_and_time结构数组，如下声明所示：

```
struct date_and_time events[100];
```

数组events包含有100个struct date_and_time类型的元素。数组中第4个date_and_time元素可以用通常的方式引用为events[3]，数组中第25个元素可以使用如下语句发送给函数dateUpdate：

```
events[24].sdate = dateUpdate (events[24].sdate);
```

要将该数组中的第一个时间设置为中午，可以使用以下的语句序列：

```
events[0].stime.hour = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

13.3.4 关于结构的补充细节

这里应该提到定义结构时有一些灵活性。首先，将变量定义为特定结构类型的同时，声明这个结构是合法的。只需要将变量名称放在结构定义的终止分号之前即可。比如，表达式

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate, purchaseDate;
```

定义了date结构，同时也声明了变量todaysDate和purchaseDate为这个类型。还可以按常规方式来给变量赋初值。因此

```
struct date
{
    int month;
    int day;
    int year;
} todaysDate = { 9, 25, 2010 };
```

定义了date结构，同时也将变量todaysDate赋予如上初值。

如果定义结构时，也定义了该结构类型的所有变量，那么可以省略结构名称。所以表达式

```
struct
{
    int month;
    int day;
    int year;
} dates[100];
```

定义了包含100个元素的数组dates。每个元素都是包含月、日和年三个整型成员的结构。因为没有为这个结构提供名称，所以定义同类型变量的唯一方式就是再次显式地定义这个结构。

位字段

在Objective-C语言中有两种包装信息的方式。一种方式是在整数内表示这些数据，然后使用第4章“数据类型和表达式”中提到的位运算符来访问所需的位。

另一种方式是用Objective-C语言所谓的位字段（bit field）来定义包装信息的结构。这种方法在定义结构时用到了特殊语法，它允许定义一个位字段并给该字段指定名称。

要定义位字段，可以定义一个名为packedStruct的结构，比如：

```
struct packedStruct
{
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int type:4;
   unsigned int index:9;
};
```

结构packedStruct定义为包含5个成员。第一个成员名为f1，是unsigned int。成员名称之后的：1表示这个成员将占用1位。类似地，标志f2和f3定义为1位。成员type被定义占用4位，而成员index定义为9位。

编译器自动地将前面的位字段定义包装在一起。这种方式好的方面是定义为packedStruct类型的字段变量可以和一般结构成员那样进行访问。所以，如果如下声明一个名为packedData的变量：

```
struct packedStruct packedData;
```

那么，可以使用这个简单的表达式方便地将packedData中的type字段设置为7：

```
packedData.type = 7;
```

还可以使用类似的表达式将该字段赋值为n：

```
packedData.type = n;
```

后面这种情况，不必考虑n的值对于type字段是否过大，只有n的低4位赋值给packedData.type。

从位字段中提取数值同样也是自动执行的，所以语句

```
n = packedData.type;
```

提取packedData的type字段（自动根据需要将它移到低位）并将之赋给n。

位字段可以用在正则表达式中，并且自动转换成整型数据。因此语句

```
i = packedData.index / 5 + 1;
```

是完全合法的，下面的表达式也一样：

```
if ( packedData.f2 )
...
```

这个语句测试标志f2是开还是关。关于位字段值得注意的一个事项是：不能保证字段在内部赋值时是从左到右还是从右到左。因此，如果位字段是从右向左赋值，那么f1将位于最低位，f2位于f1左边的位置，依此类推。除非处理由不同程序或其他机器产生的数据，否则这并不会产生问题。

还可以在包含位字段的结构中包含正常的数据类型。因此，如果想要定义包含一个int、一个char和两个1位标志的结构，下面的定义是合法的：

```
struct table_entry
{
   int      count;
   char     c;
   unsigned int f1:1;
   unsigned int f2:1;
};
```

位字段在结构定义中被包装成单位（units），单位的大小由实现来定义，并且最可能是一个字大小。Objective-C编译器并不会因为尝试优化存储空间，而重新组织位字段定义。

可以指定没有名称的位字段来跳过字中的某些位，比如下面的语句

```
struct x_entry
{
   unsigned int type:4;
   unsigned int :3;
   unsigned int count:9;
};
```

定义了一个x_entry结构，它包含一个名为type的4位字段和一个名为count的9位字段。未命名的字段表示分开type和count字段的3个位。

字段规范的最后一点涉及到长度为0的未命名字段这一特殊情况。它可以用来强制调整结构中的下一个字段作为单位边界的开始点。

13.3.5 不要忘记面向对象编程思想

现在你知道如何定义结构来存储日期，并且编写了各种例程来操纵这些date结构。但是，面向对象编程又体现在哪里？难道不应该建立一个名为Date的类，然后构造方法来使用Date对象？这难道不是一种更好的方法？当然，答案是肯定的。本节讲述在程序中存储日期的讨论时，这是希望你能够思考的问题。

当然，如果必须在程序中处理大量日期，那么定义一个类和方法是更好的途径。事实上，

Foundation框架有两个类NSDate和NSCalendarDate用于这个目的。设计一个Date类来以对象方式而不是结构方式来处理日期留作练习。

13.4 指针

指针允许你高效地表示复杂的数据结构，更改作为参数传递给函数和方法的值，并且更准确而高效地处理数组。在本章的结尾处，我们还会提示在Objective-C语言中，指针对于对象实现的重要性。

第8章“继承”中讲述了Point和Rectangle类，以及如何对同一对象实现多个指引时，介绍了指针的概念。

要了解指针操作方式，首先必须明白间接寻址（indirection）的概念。在日常生活中我们已经习惯了这种说法。比如，假设需要为我的打印机买一个彩色墨盒。在我工作的公司，所有的采购活动都由采购部门负责的。所以，可以打电话给负责采购的Jim，让他为我订购一个新墨盒。Jim将打电话给本地供应商店来订购该墨盒。这样，事实上我获得新墨盒的方式就是间接的，因为我并没有直接从供应商店处订购墨盒。

这种间接方式同样是Objective-C中指针的工作方式。指针提供了间接访问特定数据项值的途径。有理由认为通过采购部门订购墨盒很有道理（比如，不必了解从哪家供应商店订购墨盒），所以也有很好的理由认为有时在Objective-C中使用指针很有道理。

说得足够多了，应该看看指针的工作方式了。假设已经定义了一个名为count的变量，如下所示：

```
int count = 10;
```

还可以通过以下声明定义一个名为intPtr的变量，它将允许间接访问count的值：

```
int *intPtr;
```

星号向Objective-C系统定义变量intPtr是int的指针类型。这表示intPtr在这个程序中用于间接访问一个或多个整型变量的值。

在前面的程序中，学习了如何在scanf调用中使用&运算符。在Objective-C语言中，它是一元运算符，又称为地址运算符，用来得出变量的指针。所以，如果x是特定类型的变量，那么符号&x就是该变量的指针。如果需要，符号&x可以赋值给任何已经声明为与x同类的指针变量。

所以，给出count和intPtr的定义，可以编写如下表达式

```
intPtr = &count;
```

设置intPtr和count之间的间接引用。地址运算符的作用是将指向变量count的指针而不是count的值赋给变量intPtr。图13-1描述了intPtr和count之间的关系。方向线说明intPtr并不直接包含count的值，而是包含变量count的指针这一概念。

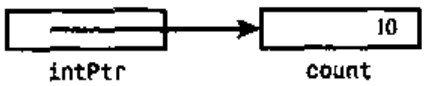

图13-1　指向整型数据的指针

要通过指针变量intPtr引用count的内容，可以使用间接寻址运算符，即星号（*）。如果x是int类型，那么语句

```
x = *intPtr;
```

会将intPtr间接指向的值赋给变量x。因为之前将intPtr设置为指向count，所以这个语句的作用就是将变量count的数值10赋给变量x。

代码清单13-8结合了之前的语句，演示两个基本的指针运算符：地址运算符（&）和间接寻址运算符（*）。

代码清单13-8

```
// Program to illustrate pointers

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int count = 10, x;
   int *intPtr;

   intPtr = &count;
   x = *intPtr;

   NSLog (@ "count = %i, x = %i", count, x);

   [pool drain];
   return 0;
}
```

代码清单13-8 输出

```
count = 10, x = 10
```

变量count和x以常规方式声明为整型变量。在下一行，变量intPtr声明为“int指针”类型。注意这两个声明行可以合并为一行，如下所示：

```
int count = 10, x, *intPtr;
```

然后，对变量count应用地址运算符，它的作用就是创建该变量的指针。然后程序将该指针赋给变量intPtr。

程序中下一条语句

```
x = *intPtr;
```

的执行过程如下：间接运算符告知Objective-C系统创建变量intPtr，它包含指向另一个数据项的指针。然后这个指针用来访问所需的数据项，该数据项的类型是由指针变量的声明指定的。因为在声明该变量时，已告知编译器intPtr指向整数，所以编译器知道表达式*intPtr指向的是整型数据值。而且，因为在前面的程序语句中，已经将intPtr设为指向整型变量count的指针，所以count的值可以使用表达式*intPtr间接访问。

代码清单13-9演示了指针变量一些有趣的属性。这里用到了指向字符的指针。

代码清单13-9

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

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    char c = 'Q';
    char *charPtr = &c;

    NSLog (@ "%c %c", c, *charPtr);

    c = '/';
    NSLog (@ "%c %c", c, *charPtr);

    *charPtr = '(';
    NSLog (@ "%c %c", c, *charPtr);

    [pool drain];
    return 0;
}
```

代码清单13-9　输出

```
Q Q
/ /
( (
```

定义了字符变量c并且将其初始化为字符‘Q’。在程序的下一行代码中，变量charPtr定义为“char指针”类型，意思是无论存储在该变量中的是什么值，都应该看作字符的间接引用(即指针)。你将注意到可以使用常规方式给这个变量赋初值。在程序中赋给charPtr的值是指向变量c的指针，它是通过在变量c前面加上地址运算符得到的（注意如果在该语句之后定义c，那么这个初始化语句将产生编译错误，因为必须在表达式中引用变量的值之前声明这个变量)。

变量charPtr的声明和初始值的分配都可以等效地使用如下两个语句表示：

```
char *charPtr;
charPtr = &c;
```

(而不是像前一行声明暗示的那样，通过语句

```
char *charPtr;
*charPtr = &c;
```

表示)。

始终记住在Objective-C语言中，将指针设置指向一些值之前，指针的值是没有意义的。

第一个NSLog调用仅仅显示变量c的内容以及charPtr所引用的变量内容。因为你将charPtr设为指向变量c，所以显示的就是c的内容，在程序输出的第一行就能得到验证。

该程序的下一行，将字符‘/’赋给字符变量c。因为charPtr仍然指向变量c，所以最后的NSLog语句中显示*charPtr的值就正确地显示为c的新值。这是非常重要的概念。除非更改charPtr的值，否则表达式*charPtr总是访问c的值。这样，当c的值发生变化时，*charPtr的值也相应地改变。

前面的讨论有助于理解程序中的下一条语句的工作方式。我们提到过，除非更改charPtr，否则表达式*charPtr将总是引用c的值。所以，在以下表达式中

```
*charPtr = '(';
```

将左括号赋给c。从形式上说是将字符‘(’赋给charPtr指向的变量。你知道这个变量是c，因为在程序的开始将c的指针存入了charPtr。

上述概念是理解指针操作的关键。如果还不是很清楚，再回顾一下这些知识。

13.4.1 指针和结构

你已经看到如何将指针定义为指向基本数据类型（如int和char）。但是指针还可以指向结构。在本章的前面，定义了如下date结构：

```
struct date
{
   int month;
   int day;
   int year;
};
```

与使用下面的语句定义struct date类型的变量一样

```
struct date todaysDate;
```

也可以定义指向struct date变量的指针变量：

```
struct date *datePtr;
```

然后就可以用期望的方式使用刚才定义的变量datePtr。比如，可以使用下面的赋值语句将其设为指向变量todaysDate的指针：

```
datePtr = &todaysDate;
```

如上赋值之后，就可以用如下方式通过datePtr间接访问date结构的任何成员：

```
(*datePtr).day = 21;
```

这个语句的作用是将datePtr指向的date结构中的day成员设置为21。括号是必需的，因为结构成员运算符比间接寻址运算符*的优先级别高。

要测试存储在datePtr指向的date结构中month成员的值，可以使用如下语句：

```
if ( (*datePtr).month == 12 )
     ...
```

因为经常用到结构指针，所以该语言中存在着一个特殊的运算符。结构指针运算符->，即短划线和紧跟其后的大于号，它允许将表达式

```
(*x).y
```

更清楚地表示为

```
x->y
```

所以，前面的if语句可以方便地写为：

```
if ( datePtr->month == 12 )
   ...
```

第一个说明结构的代码清单13-6通过结构指针的概念被重新改写。这个程序被改写成这里的代码清单13-10。

代码清单13-10

```
// Program to illustrate structure pointers
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   struct date
   {
      int month;
      int day;
      int year;
   };

   struct date today, *datePtr;

   datePtr = &today;
   datePtr->month = 9;
   datePtr->day = 25;
   datePtr->year = 2009;

   NSLog (@ "Today's date is %i/%i/%.2i.",
         datePtr->month, datePtr->day, datePtr->year % 100);
   [pool drain];
   return 0;
}
```

代码清单13-10　输出

```
Today's date is 9/25/09.
```

13.4.2　指针、方法和函数

可以按一般方式将指针作为参数传递给方法或函数，并且可以让函数或者方法返回指针。

alloc和init方法一直都在这么做，也就是返回指针。本章结尾对此有更详细的介绍。

现在考虑代码清单13-11。

代码清单13-11

```
// Pointers as arguments to functions
#import <Foundation/Foundation.h>

void exchange (int *pint1, int *pint2)
{
   int temp;

   temp = *pint1;
   *pint1 = *pint2;
   *pint2 = temp;
}

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   void exchange (int *pint1, int *pint2);
   int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

   NSLog (@ "i1 = %i, i2 = %i", i1, i2);

   exchange (p1, p2);
   NSLog (@ "i1 = %i, i2 = %i", i1, i2);

   exchange (&i1, &i2);
   NSLog (@ "i1 = %i, i2 = %i", i1, i2);

   [pool drain];
   return 0;
}
```

代码清单13-11 输出

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

函数exchange的目的是交换由两个参数指向的两个整型值。局部整型变量temp用于在交换时存放其中一个整数的值。它的值设为等于pint1指向的整型值。然后，pint2指向的整数被复制到pint1指向的整数中，最后temp的数值被复制到pint2指向的整数中，这样就完成了交换。

main例程定义了整数i1和i2，并给它们分别赋值-5和66。然后，定义了两个整型指针p1和p2，并分别将其设为指向i1和i2。然后这个程序显示i1和i2的值，并且传递两个指针（p1和p2）作为参数来调用函数exchange。函数exchange交换p1指向的整数值和p2指向的整数值。因为p1

指向i1，p2指向i2，所以函数交换的是i1和i2的值。第二个NSLog调用的输出验证交换函数运行良好。

第二次调用exchange函数比较有意思。这次，传递给该函数的参数是通过对i1和i2应用地址运算符手动创建的指针。因为表达式&i1产生了指向整型变量i1的指针，所以它符合函数所期望的第一个参数类型（整型指针）。这同样适用于第二个参数。从程序的输出结果中可以看出，函数exchange完成了这项任务，将i1和i2的数值交换回它们原本的值。

详细研究代码清单13-11。它使用一个小例子阐明了Objective-C语言中处理指针所需要了解的主要概念。

13.4.3 指针和数组

如果有一个包含100个整数的数组values，那么可以定义一个名为valuesPtr的指针，它可以通过下面的表达式访问数组中的整数：

```
int *valuesPtr;
```

定义用来指向数组元素的指针时，不能将指针定义为“数组指针”，而是要将指针定义为数组中所包含的元素类型。

如果你有一个Fraction对象数组fracts。同样，可以通过下面的语句定义一个指针，用于指向fracts中的元素：

```
Fraction* *fractsPtr;
```

注意这是也用来定义Fraction对象的声明。

要将valuesPtr设为指向数组values的第一个元素的指针，可以简单地编写为：

```
valuesPtr = values;
```

在这个例子中并没有用到地址运算符，因为Objective-C编译器将没有下标的数组名称看作是指向数组第一个元素的指针。所以，仅仅指明values而不带下标的作用就是产生一个指向values第一个元素的指针。

产生指向values首元素指针的另一个等效方式是对数组第一个元素应用地址运算符。因此，语句

```
valuesPtr = &values[0];
```

用来将数组values第一个元素的指针存放到指针变量valuesPtr中。

要显示数组fracts中fractsPtr指向的Fraction对象，可以编写下面的表达式：

```
[*fractsPtr print];
```

对数组使用指针的真正好处体现在按顺序访问数组元素时。如果按照前面提到的方法定义valuesPtr，并且将其设置为指向values的第一个元素，那么表达式

```
*valuesPtr
```

访问数组values的第一个整数，即values[0]。要通过变量valuesPtr引用values[3]，可以将valuesPtr加3，然后应用间接寻址运算符：

```
*(valuesPtr + 3)
```

更广泛的情况是，可以使用表达式

```
*(valuesPtr + i)
```

来访问values[i]的数值。

所以，要将values[10]设置为27，显然可以如下编写表达式：

```
values[10] = 27;
```

或者使用valuesPtr，写为：

```
*(valuesPtr + 10) = 27;
```

要使valuesPtr指向数组values中的第二个元素，可以在values[1]之前加上地址运算符并将结果赋给valuesPtr：

```
valuesPtr = &values[1];
```

如果valuesPtr指向values[0]，通过将valuesPtr的数值加1，可以让它指向values[1]：

```
valuesPtr += 1;
```

在Objective-C语言中，这是完全合法的表达式，并且可以用于指向任何数据类型的指针。

所以，一般来说，如果a是元素类型为x的数组，px是“x指针”类型，并且i和n都是变量的整数常量，则表达式：

```
px = a;
```

将px设置为指向a的第一个元素，而后，表达式

```
*(px + i)
```

指向a[i]中包含的值。此外，表达式

```
px += n;
```

将px设为指向数组中比原来指向的元素多了n个元素的指针，无论数组中包含的元素是什么类型。

假设fractsPtr指向存储在分数数组中的分数。另外，假设想要将它与数组的下一个元素包含的分数相加，并将结果赋给Fraction对象result。通过编写下面的表达式即可实现：

```
result = [*fractsPtr add: *fractsPtr + 1];
```

处理指针时，运用自增和自减运算符（++和--）非常方便。对指针应用自增运算符相当于将指针加1，而对指针应用自减运算符则相当于将指针减1。所以，如果将textPtr定义为char指针，并将其设置为指向chars数组text的第一个字符，则表达式

```
++textPtr;
```

会使textPtr指向数组text的下一个字符，即text[1]。类似地，表达式

```
--textPtr;
```

会将textPtr指向数组text的上一个字符，当然要假设在这条语句执行之前，指针textPtr并没有指向数组text的首字符。

在Objective-C语言中，比较两个指针变量的做法是完全合法的。这在比较指向同一数组的

两个指针时是非常有用的。比如，你可以测试指针valuesPtr，看它的指向是否超出了包含有100个元素的数组的范围，方法是将它与指向数组最后一个元素的指针相比较。所以如果valuesPtr超出了数组values的最后元素，表达式

```
valuesPtr > &values[99]
```

的结果将为TRUE（非零），反之表达式的值为FALSE（零）。根据前面的讨论，可以将上面的表达式相应地改写为：

```
valuesPtr > values + 99
```

这是可能的，因为values不带有下标，它是指向数组values首字符的指针（记住这和写成&values[0]是一样的）。

代码清单13-12举例说明了指向数组的指针。函数arraySum计算整型数组所包含的元素之和。

代码清单13-12

```
// Function to sum the elements of an integer array

#import <Foundation/Foundation.h>

int arraySum (int array[], int n)
{
   int sum = 0, *ptr;
   int *arrayEnd = array + n;

   for ( ptr = array; ptr < arrayEnd; ++ptr )
       sum += *ptr;

   return (sum);
}

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int arraySum (int array[], int n);
   int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

   NSLog (@"The sum is %i", arraySum (values, 10));
   [pool drain];
   return 0;
}
```

代码清单13-12　输出

```
The sum is 21
```

在函数arraySum中，定义了整型指针arrayEnd，并使其指向数组最后一个元素之后的指针。然后，设置for循环来顺序浏览array的元素，当循环开始时，ptr的值被设置为array的首字符。

每循环一次，ptr所指向的array元素的值都被加到sum中。然后for循环自动递增ptr的值，将它设置为指向array的下一个元素。当ptr超出了数组范围时，就退出for循环，并将sum的值返回给调用者。

是数组还是指针

要将数组传递给函数，只要和前面调用函数arraySum一样，简单地指定数组名称即可。但是在本节中还提到过，要产生指向数组的指针，只需指定数组名称即可。这暗示着在调用函数arraySum时，传递给函数的实际上是数组values的指针。这确切地解释了为什么能够在函数中更改数组元素的值。

但是，如果事实情况是将数组的指针传递给函数，那么为什么函数中的形参不声明为指针呢？换句话说，函数arraySum的array声明中，为什么没有用到以下声明：

```
int *array;
```

在函数中，是不是所有对数组的引用都是通过指针变量实现的？

要回答这些问题，首先必须重申前面提到的关于指针和数组的话题。我们提到过，如果valuesPtr指向的数据类型和values数组中包含的元素的类型相同，并且假设将valuesPtr设为指向values的首字符，那么表达式*(valuesPtr + i)与符号values[i]完全相同。然后，还可以用表达式*(values + i)来引用数组values的第i个元素，并且，一般说来，如果x是任意类型的数组，则在Objective-C语言中，可以将表达式x[i]等价地表示为*(x+i)。

可以看到，在Objective-C语言中指针和数组是密切相关的，这就是为什么可以在函数arraySum中将array声明为“int数组”类型，或者是“int指针”类型。在前面的例子中，以上每种声明都可以正常工作——尝试并查看结果。

如果要使用索引数来引用数组元素，那么要将对应的形参声明为数组。这能更准确地反应该函数对数组的使用情况。类似地，如果要将参数作为指向数组的指针，则要将其声明为指针类型。

字符串指针

指向数组的指针最广泛的应用之一就是作为字符串指针。原因是符号表达的便利和效率。要说明字符串指针使用起来多方便，编写一个名为copyString的函数，它将一个字符串复制到另一个字符串之中。如果使用常规数组索引方式编写这个函数，则可以如下编码：

```
void copyString (char to[], char from[])
{
   int i;

   for ( i = 0; from[i] != '\0'; ++i )
       to[i] = from[i];

   to[i] = '\0';
}
```

for循环在将空字符复制到数组to之前退出，这就解释了该函数中最后一行代码存在的必要性。

如果使用指针编写copyString，那么不需要使用索引变量i。代码清单13-13展示了该程序的

指针版本。

代码清单13-13

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

void copyString (char *to, char *from)
{
    for ( ; *from != '\0'; ++from, ++to )
        *to = *from;

    *to = '\0';
}

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   void copyString (char *to, char *from);
   char string1[] = "A string to be copied.";
   char string2[50];

   copyString (string2, string1);
   NSLog (@"%s", string2);

   copyString (string2, "So is this.");
   NSLog (@"%s", string2);

   [pool drain];
   return 0;
}
```

代码清单13-13　输出

```
A string to be copied.
So is this.
```

函数copyString将两个形参to和from定义为字符指针，而不是像前一个copyString那样定义为字符数组。这反映出该函数将如何使用这两个变量。

然后进入for循环（没有初始条件），它将from指向的字符串复制到to指向的字符串中。每循环一次，指针from和to都会分别自增1。这样from指针就指向源字符串中下一个要复制的字符，而指针to则指向目标字符串中下一个要存储的字符。

当指针from指向空字符时，for循环将退出。然后，该函数在目标字符串的结尾处放置一个

⊖ 注意程序中字符串“A string to be copied”和“So is this”的使用。它们不是字符串对象，而是C样式的字符串，区分方式是，字符串对象前面没有@。这两种类型是不能互换的。如果函数期望用字符数组作为参数，就应该给函数传递一个char类型的数组，或者一个C样式的字符串，而不是一个字符串对象。

空字符。

在main例程中，函数copyString被调用了两次，第一次用来将string1的内容复制到string2中，而第二次用来将字符串常量“So is this.”复制到string2中。

字符串常量和指针

前面的程序中，调用函数

```
copyString (string2, 'So is this.');
```

可以正常工作的事实意味着向函数传递字符串作为参数时，事实上传递的是指向该字符串的指针。这种情况不仅正确，而且还可以推广到只要在Objective-C语言中用到字符串，就会产生指向该字符串的指针。

以下观点可能有些让人混乱，但是正如第4章提到的：这里我们提到的字符串常量称为C样式字符串。它们不是对象。你知道，通过在字符串前面添加标志@（如@“This is okey.”），就可以创建一个常量字符串对象。不能使用一个替代另一个。

所以，如果将textPtr声明为字符指针，如下所示：

```
char *textPtr;
```

那么表达式

```
textPtr = 'A character string.";
```

则将textPtr设为指向字符串常量“A character string.”的指针。需要注意字符指针和字符数组之间的区别，因为前面显示的赋值类型对于字符数组并不合法。例如，如果将text定义为chars数组，语句如下：

```
char text[80];
```

那么就不能编写如下表达式：

```
text = 'This is not valid.";
```

只有在初始化字符数组时，Objective-C语言才允许对字符数组使用这种赋值方式，如下所示：

```
char text[80] = "This is okay.';
```

以这种方式初始化text并没有在text中存储指向字符串“This is okay.”的指针，而是在相应的text数组元素中存储实际的字符本身及最后的终止空字符。

如果text是一个字符指针，则使用以下表达式初始化text：

```
char *text = 'This is okay.';
```

将赋给它一个指向字符串“This is okay.”的指针。

回顾自增和自减运算符

到目前为止，在使用自增或自减运算符时，它们都是表达式中唯一出现的运算符。编写表达式++x时，你知道这将使变量x的值加1。你刚刚学过，如果x是指向数组的指针，这将使x指向数组的下一个元素。

自增和自减运算符也可以用于有其他运算符出现的表达式中。在这种情况下，更准确地了解这两个运算符的作用方式是非常重要的。

使用自增和自减运算符时，总是将它们放在自增或自减变量之前。所以，要使变量i自增，只需要写：

```
++i;
```

事实上，将自增运算符放在变量的后面同样合法，比如：

```
i++;
```

两种表达式都是合法的，而且都有着相同的结果，也就是，将i的数值加1。第一种情况，即++放在操作数前，可以更准确地将自增运算定义为前缀自增。第二种情况，即++放在操作数之后，可以将自减运算定义为后缀自增。

自减运算符也一样。所以，语句

```
--i;
```

实现了i的前缀自减操作，而语句

```
i--;
```

则实现了i的后缀自减操作。它们的最终结果相同，即i的数值减1。

在更复杂的表达式中使用自增和自减运算符时，运算符前缀和后缀的不同之处就体现出来了。

假设有两个整数i和j。如果将i的值设为0，然后编写语句

```
j = ++i;
```

那么赋给j的值是1，不是你可能认为的0。使用前缀自增运算符时，变量的值在被用到之前先自增。所以，在前面的语句中，i的值先从0加到1，然后再将它的值赋给j，结果与下面两个表达式一样：

```
++i;
j = i;
```

如果在语句中使用后缀自增运算符

```
j = i++;
```

那么i在它的数值被赋给j之后才自增。所以，如果执行前面的语句之前，i的值为0，那么j将被赋值为0，然后i的值再加1，就像使用下面两个表达式

```
j = i;
++i;
```

另举个例子，如果i等于1，那么语句

```
x = a[--i];
```

的结果是把a[0]的值赋给x，因为变量i的数值在用作a的索引之前，值已经减1。语句

```
x = a[i--];
```

会将a[1]的值赋给x，因为i是在用作a的索引之后自减1的。

作为描述前缀和后缀自增自减运算符区别的第三个例子，函数调用语句

```
NSLog (@ "%i", ++i);
```

将i的值自增1，然后将该值发送给NSLog函数。而调用

```
NSLog (@ "%i", i++);
```

则在将i的值发送给函数之后才自增1。所以，如果i等于100，那么第一个NSLog语句将显示101，而第二个NSLog将显示100。在以上任何情况下，执行该语句之后，i的数值都会等于101。

在介绍程序之前再举一个关于此话题的例子，如果textPtr是一个字符指针，那么表达式

```
*(++textPtr)
```

首先将textPtr的值加1，然后获取它所指向的字符，而表达式

```
*(textPtr++)
```

在textPtr自增之前，先获取它指向的字符。这两种情况都不要求必须存在括号，因为*和++运算符的优先级相同，按照从左向右的顺序进行运算。

回顾代码清单13-13的copyString函数，重写这个函数，使它将自增运算直接合并到赋值语句中。

因为每次执行for循环中的赋值语句之后，指针to和from都会自增1，所以它们应该以后缀自增形式合并到赋值语句中。将代码清单13-13 的for循环重写为：

```
for ( ; *from != '\0'; )
    *to++ = *from++;
```

循环中的赋值语句将按以下方式执行：先检索from指向的字符，然后from将自加1，从而指向源字符串的下一个字符。引用的字符将被存储到to指向的位置，然后to自加1，从而指向目标字符串的下一个地址。

前面的for语句看上去并不实用，因为它没有初始表达式，也没有循环表达式。事实上，使用while循环的形式来表示，逻辑性可能会更明显。代码清单13-14就是这样实现的，该程序给了函数copyString的新版本。while循环用到了空字符等于数值0这一事实，熟练的Objective-C编程人员经常这样使用。

代码清单13-14

```
// Function to copy one string to another
//          pointer version 2

#import <Foundation/Foundation.h>
void copyString (char *to, char *from)
{
     while ( *from )
          *to++ = *from++;
     *to = '\0';
}

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   void copyString (char *to, char *from);
```

```
    char string1[] = "A string to be copied.";
    char string2[50];

    copyString (string2, string1);
    NSLog (@ "%s", string2);

    copyString (string2, "So is this.");
    NSLog (@  %s", string2);
    [pool drain];
    return 0;
}
```

代码清单13-14　输出

```
A string to be copied.
So is this.
```

13.4.4　指针运算

在本章学过，可以给指针加减整型数值。此外，可以比较两个指针来查看它们是否相等或者检测一个指针是大于还是小于另一个指针。指针所允许的其他唯一操作就是相同类型的两个指针的减法。在Objective-C语言中，两指针相减的结果是它们之间所包含的元素个数。这样，如果a是指向任意类型的元素数组，而b指向同一数组中索引值更大的其他元素，那么表达式b-a代表的就是这两个指针之间的元素个数。比如，如果p指向数组x中的某个元素，那么语句

```
n = p - x;
```

赋给变量n（假设为整型变量）的是数组x中p指向元素的索引数。所以，如果通过以下语句将p设置为指向x中的第100个元素：

```
p = &x[99];
```

那么经过前面的减法运算，n的值应该为99。

函数指针

函数指针的概念有些高级，但出于完整性的考虑，我们在这里介绍它。处理函数指针时，Objective-C编译器不但需要知道指向函数的指针变量，而且要知道函数返回值的类型和参数的数目及类型。要声明变量fnPtr为“指向返回int并且不带参数的函数的指针”，可以编写下面的声明来实现：

```
int (*fnPtr) (void);
```

*fnPtr两侧的括号是必需的，否则，Objective-C编译器就会认为，在上述语句中，名为fnPtr的函数声明返回一个int指针（因为函数调用运算符()比指针间接寻址运算符*的优先级高）。

要使函数指针指向特定函数，可以简单地将函数的名称赋给该指针。因此，如果lookup是返回int并且不带参数的函数，则语句

```
fnPtr = lookup;
```

将指向lookup函数的指针存入函数指针变量fnPtr。编写一个函数名称，并且不带随后的一对括号，类似于编写没有下标的数组名称那样。Objective-C编译器将自动产生指向特定函数的指针。允许在函数名称之前添加&标志，但这不是必需的。

如果之前在程序中没有定义函数lookup，则必须在实现前面的赋值运算之前声明该函数。如下语句

```
int lookup (void);
```

在将函数指针赋给变量fnPtr之前是必需的。

通过对指针变量应用函数调用运算符，同时在括号内列出该函数的所有参数，可以间接引用指针变量引用的函数。比如

```
entry = fnPtr ();
```

调用fnPtr所指向的函数，并将返回值存储在变量entry中。

函数指针的一个常见应用是将其作为参数传递给其他函数。Standard Library在函数qsort中用到此应用，该函数实现数组元素的快速排序。它将一个函数指针作为一个参数，只要qsort需要比较待排序的数组中两个元素时，它就会调用这个函数。使用这种方式，qsort可以用来对任何类型的数组进行排序，因为数组中任意两个元素的比较是由用户提供的函数实现的，而不是函数qsort本身。

在Foundation框架中，一些方法使用函数指针作为参数。比如，方法sortUsingFunction:context:定义在类NSMutableArray中，并且无论何时待排序的数组中两个元素需要比较时，就调用指定的函数。

函数指针的另一个常见应用是建立分派表。不能将函数本身保存在数组元素中。然而，可以在数组中存储函数指针。这样，就可以创建包含要调用的函数指针的表。比如，可能创建一个表，用于处理用户输入的不同命令。该表中每项都可以包含命令名称和指向处理这项特定命令要调用的函数的指针。现在，只要用户输入一条命令，就可以查询该表中的命令，并调用相应的函数去处理它。

13.4.5 指针和内存地址

在结束Objective-C中的指针讨论之前，应该指出实际如何实现指针的细节。计算机的内存可以理解为存储单元的顺序集合。计算机内存的每个单元都有一个相关的编号，称为地址。通常，计算机内存的首地址为0。在大多数计算机系统中，一个单元就是一字节。

计算机使用内存来存储计算机程序的指令和程序相关变量的值。所以，如果声明变量count为int数据，那么在程序执行时，系统会分配内存地址来存储count的值。比如，这个位置也许是内存地址$1000FF_{16}$。

幸运的是，你自己不需要考虑指定给变量的特定内存地址，它们是系统自动处理的。然而，掌握每个变量都有唯一内存地址这个知识，对于理解指针的工作方式会有所帮助。

在Objective-C语言中，对变量应用地址运算符，产生的值是变量在计算机内存中的实际存储地址（显然，这就是地址运算符名称的由来）。所以，语句

```
intPtr = &count;
```

向intPtr分配指定给变量count的计算机内存地址。这样，如果count位于地址$1000FF_{16}$，那么这个语句将数值0x1000FF赋给intPtr。

对指针变量应用间接寻址运算符，表达式

```
*intPtr
```

的作用就是将包含在指针变量中的数值当作内存地址。然后获取该内存地址中存储的值并按照指针变量声明的类型进行解释。所以，如果intPtr是int指针，那么系统将存储在内存地址中由*intPtr给出的值解释为整型数据。

13.5 联合

在Objective-C编程语言中，最不寻常的构造之一就是联合（union）。这种结构主要用在需要在相同存储区域存放不同类型数据的更高级编程应用中。比如，如果要定义一个名为x的变量，它可以用来存储单个字符、一个浮点数或是一个整数，那么首先要定义一个联合，可能名为mixed，如下所示：

```
union mixed
{
   char c;
   float f;
   int i;
};
```

联合的声明和结构的声明一样，除了关键字用的是union而不是结构声明中所用的struct。联合和结构的真正区别在于，联合必须涉及到内存的分配方式。如下语句

```
union mixed x;
```

声明一个union mixed类型的变量，它并没有定义x包含三个不同成员c、f和i；而是定义x为包含有成员c、f、i其中之一。这样，变量x可以用来存储char型、float型或int型的数据，但不是同时存储这三个数据（也不是存储其中两个数据）。可以使用如下表达式将一个字符存储到变量x中：

```
x.c = 'K';
```

要将一个浮点数存储到x中，可以使用符号x.f：

```
x.f = 786.3869;
```

最后，要将整数count除以2的结果存储到x中，使用如下语句：

```
x.i = count / 2;
```

因为x的float、char和int成员共存于内存中的同一位置，所以每次只能有一个数值存入x。此外，必须确保从联合获得的值与联合中最后存入的类型一致。

定义联合时，并不要求有联合名称，而且在声明联合的同时可以定义联合变量。也可以声明联合的指针，它们执行运算的的语法和规则与结构的相同。最后，可以如下初始化联合变量：

```
union mixed x = { '#' };
```

该语句将x的第一个成员（即c）设置为字符#。还可以通过名称初始化特定成员，如下所示：

```
union mixed x = {.f=123.4;};
```

还可以将自动联合变量初始化为同类型的另一个联合变量。

使用联合允许你定义一个存储不同数据类型元素的数组。比如，语句

```
struct
{
    char *name;
    int type;
    union
    {
        int i;
        float f;
        char c;
    } data;
} table [kTableEntries];
```

设置了一个包含kTableEntries元素的数组table。数组中的每个元素都包含一个结构，它包含一个名为name的字符指针、一个名为type的整型数和一个名为data的联合成员。数组中的每个data成员可以存储一个int型、float型或char型数据。整数成员type也许用来记录存储在成员data中的数据类型。例如：如果data中存储的是int型数据，可以将它赋值为INTEGER（假设已经适当地定义了）；如果存储的是float型数据，可以赋值为FLOATING；如果存储的是char型数据，则赋值为CHARACTER。通过这个信息可以知道如何应用特定数组元素中的特定data成员。

要将字符‘#’存入table[5]，并且随后设置type字段，以说明该位置存储的是字符，可以使用下面两条语句：

```
table[5].data.c = '#';
table[5].type = CHARACTER;
```

依次访问table中的元素时，通过编写合适的测试语句序列，可以确定每个元素中存储的data值类型。比如，下面的循环将在终端显示每个名称及table中相关的值：

```
enum symbolType { INTEGER, FLOATING, CHARACTER };
  ...

for ( j = 0; j < kTableEntries; ++j )
{
   NSLog (@ "%s", table[j].name);

   switch ( table[j].type )
   {
      case INTEGER:
           NSLog (@ "%i", table[j].data.i);
           break;
      case FLOATING:
           NSLog (@ "%g", table[j].data.f);
```

```
            break;
        case CHARACTER:
            NSLog (@ "%c", table[j].data.c);
            break;
        default:
            NSLog (@ 'Unknown type (%i), element %i',
                    table[j].type, j );
            break;
    }
}
```

前面说明的应用类型对于存储符号表可能很实用，它可能包含每个符号的名称、类型和值（也许还有该符号的其他信息）。

13.6 它们不是对象

你现在已经知道了如何定义数组、结构、字符串和联合，以及如何在程序中操纵它们。只要记住一个基本原则：它们不是对象。这意味着不能给它们传递消息，也不能利用它们获得Foundation框架提供的内存分配策略之类的最大优势。这是我鼓励你跳过本章，以后再学习的原因之一。概括地说，我更希望你学习如何使用Foundation中将数组和字符串定义为对象的类，而不是使用该语言中内置的类。只应该在真正需要时才使用本章所定义的类型。并且希望你不要遇到这种情况！

13.7 其他语言特性

一些语言特性不能归入其他章节，所以就在这里介绍它们。

13.7.1 Compound Literal

Compound Literals是包含在括号之内的类型名称，之后是一个初始化列表。它创建特定类型的未命名值，它的作用域限于创建它的块；如果它是在所有程序块之外定义的，则是全局作用域。在后一种情况下，初始化表达式必须都是常量表达式。

下面是一个例子：

```
(struct date) {.month = 7, .day = 2, .year = 2004}
```

这个表达式产生struct date类型的结构，并且有一些初始值。可以将它赋值给另一个struct date，如下所示：

```
theDate = (struct date) {.month = 7, .day = 2, .year = 2004};
```

或者，它可以传递给带有struct date参数的函数或方法，如下所示：

```
setStartDate ((struct date) {.month = 7, .day = 2, .year = 2004});
```

还可以定义结构之外的其他类型，例如，如果intPtr为int *类型，那么语句

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

（它可以出现在程序中的任何位置）将intPtr设置为指向包含100个整数的数组，数组的前三

个元素初始化为特定的数值。

如果数组的大小没有说明，则由初始列表来确定。

13.7.2 goto语句

goto语句的执行导致在程序中产生一个到达特定点的直接分支。要确定程序中分支所在的位置，需要一个标签。标签是根据生成变量名称相同的规则生成的名称，它之后必须紧跟一个冒号。标签直接放在分支语句之前，而且必须在goto之类的函数或方法之前。

比如，语句

```
goto out_of_data;
```

使程序立即跳到标签out_of_data之后的语句分支；这个标签可以放在函数或方法中的任何地方，无论是在goto语句之前或之后，并且可以如下使用该语句：

```
out_of_data: NSLog (@ "Unexpected end of data.");
  ...
```

懒惰的程序员经常滥用goto语句来转移到代码的其他部分。goto语句打断了程序的常规顺序流程，结果就会造成程序难以读懂。在程序中使用很多goto语句会使程序变得难于解释。由于这个原因，所以使用goto语句并不认为是好的编程方式。

13.7.3 空语句

Objective-C语言允许将孤立的分号放在可以出现常规语句的地方。这种称为空语句的语句不做任何操作。这看上去没有什么用，但是程序员经常将它用在while、for和do语句。比如，下面语句的作用是将所有从标准输入（默认为终端）读入的字符存储到指针text指向的字符数组，直到出现换行字符为止。它使用库例程getchar每次从标准输入读入并返回单个字符：

```
while ( (*text++ = getchar ()) != '' )
  ;
```

所有操作都是在while语句的循环条件部分中实现的，需要有空语句是因为编译器认为循环语句后的下一条语句是循环体。如果没有空语句，无论下一条语句是什么，都会被编译器认为是循环体。

13.7.4 逗号运算符

优先级列表的最底层是逗号运算符。在第5章“循环结构”中，我们提到过在for语句中通过使用逗号将每条语句分开，可以在任一部分包含多条表达式。比如，开始于

```
for ( i = 0, j = 100; i != 10; ++i, j -= 10 )
  ...
```

的for语句在循环开始前将i初始化为0，j初始化为100，然后，每次执行循环体之后，i的值自减1，j的值自减10。

因为在Objective-C语言中，所有运算符都产生一个值，所以逗号运算符的值是最右边的表

达式值。

13.7.5 sizeof运算符

尽管不应该假设程序中数据类型的大小，但是有的时候需要知道这些信息。使用库例程（如malloc）来实现动态内存分配，或对文件读出或写入数据时，可能需要这些信息。Objective-C语言提供了sizeof运算符，它可以用来确定数据类型或对象的大小。Sizeof运算符返回的是指定项的字节大小。sizeof运算符的参数可以是变量、数组名称、基本数据类型名称、对象、派生数据类型名称或表达式。比如，

```
sizeof (int)
```

给出了存储整型数据所需的字节数。在我的MacBook Air计算机上，上述符号会产生结果4（或32位）。如果x声明为包含100个int数据的数组，则表达式

```
sizeof (x)
```

将给出存储x中的100个整数所需要的存储空间。

假设myFract是一个Fraction对象，它包含两个int实例变量（分子和分母），那么表达式

```
sizeof (myFract)
```

在任何使用4字节表示指针的系统中都会产生值4。事实上，这是sizeof对任何对象产生的值，因为这里询问的是指向对象数据的指针大小。要获得实际存储Fraction对象实例的数据结构大小，可以编写以下语句：

```
sizeof (*myFract)
```

在我的MacBook Air计算机上，上述表达式给出了值12。即分子和分母分别用4个字节，加上另外的4个字节存储继承来的isa成员，本章最后的“工作原理”一节中将提到这个成员。

表达式

```
sizeof (struct data_entry)
```

的值将是存储data_entry结构所需的空间总数。如果将data定义为包含struct data_entry元素的数组，则表达式

```
sizeof (data) / sizeof (struct data_entry)
```

将给出包含在data（data必须是前面定义的，并且不是形参也不是外部引用的数组）中的元素个数。表达式

```
sizeof (data) / sizeof (data[0])
```

也产生同样的结果。

尽可能地使用sizeof运算符来避免必须在程序中计算和硬编码数据大小。

13.7.6 命令行参数

很多时候，程序都要求用户在终端输入少量信息。这些信息可能包含一些数字，它们表示你想要计算的三角数或者是想要在字典中查询的单词。

除了让程序请求用户输入相关信息，还可以在程序执行时给该程序提供信息。这种能力是由命令行参数提供的。

我们提到过main函数唯一的与众不同之处在于它的名称很特别：它指明了程序开始执行的位置。事实上，main函数是在程序开始执行时，由运行时系统调用，就像在自己的程序中调用函数一样。当main执行完毕时，控制权返回给运行的系统，这样系统便知道程序已经执行完毕了。

当运行时系统调用main函数时，系统向该函数传递两个参数。第一个参数，按照规定称为argc（argument count的简写），是一个整型值，它指明了从命令行键入的参数个数。第二个传递给main的参数是一个字符指针数组，按照规定称为argv（argument vector的简写）。另外，这个数组中包含argc+1个字符指针。该数组的第一个元素是执行程序的名称指针，如果你的系统中没有程序名称，则是空串。数组的其他项指向由启动程序执行的命令行所指定的值。数组argv中的最后一个指针argv[argc]规定为空。

要访问命令行参数，必须将main函数适当地声明为带有两个参数。使用本书所有程序中用到的惯例声明就可以了，如下所示：

```
int main (int argc, char *argv[])
{
  ...
}
```

记住，argv的声明定义了一个包含“char指针”类型元素的数组。命令行参数的一个实际用途是，假设你开发了一个程序，它在字典中查询某个单词并显示单词的含义，那么可以使用命令行参数，这样在执行程序同时就可以指定想要查找含义的单词，如以下命令所示：

```
lookup aerie
```

这就不必提示用户键入一个单词，因为单词已经从命令行键入了。

如果执行上面的命令，那么系统自动向main函数传递argv[1]中字符串“aerie”的指针。你可能会想起，argv[0]包含的是指向程序名称的指针，在这个例子中是“lookup”。

main函数可能如下所示：

```
#include <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
   struct entry dictionary[100] =
     { { "aardvark", "a burrowing African mammal"},
       {"abyss", "a bottomless pit"},
       {"acumen", "mentally sharp; keen"},
       {"addle", "to become confused"},
       {'aerie", "a high nest"},
       {'affix", "to append; attach"},
       {"agar", "a jelly made from seaweed'},
       {"ahoy", "a nautical call of greeting"},
```

```
        {"aigrette", "an ornamental cluster of feathers"},
        {"ajar", "partially opened"} };

    int entries = 10;
    int entryNumber;
    int lookup (struct entry dictionary [], char search[],
                int entries);

    if ( argc != 2 )
    {
         NSLog (@ "No word typed on the command line.");
         return (1);
    }

    entryNumber = lookup (dictionary, argv[1], entries);

    if ( entryNumber != -1 )
       NSLog (@ "%s", dictionary[entryNumber].definition);
    else
       NSLog (@ "Sorry, %s is not in my dictionary.", argv[1]);
  return (0);
}
```

执行程序时，main例程测试以确保在程序名称之后键入一个单词。如果没有，或键入了多个单词，那么argc的数值将不等于2。在这种情况下，程序将写明出错消息并终止运行，返回终止状态1。

如果argc等于2，将调用函数lookup，在字典中寻找argv[1]指向的单词。如果找到了该单词，则显示它的定义。

需要记住命令行参数总是存储为字符串。所以，带有命令行参数2和16的power程序：

```
power 2 16
```

的执行结果是将字符串“2”的指针保存在argv[1]中，将字符串“16”的指针保存在argv[2]。如果程序将参数解释为数字（就像我们猜想程序power一样），程序本身必须转换它。程序库中有一些例程可以用来实现这种转换，如sscanf、atof、atoi、strtod和strtol。在第二部分中，将学习如何使用名为NSProcessInfo的类来以字符串对象而不是C字符串的形式访问命令行参数。

13.8 工作原理

如果没有先尝试将一些事物联系在一起就结束本章，那么我们就没有尽到责任。因为Objective-C语言以C语言为基础，所以值得讨论前者和后者的关系。下面是一些可以忽略的实现细节，或可以用来更好地理解系统工作方式的细节，就像学习指针实际上是内存地址可以帮助你更好地理解指针一样。我们没有涉及到太多详细内容，只是阐明关于Objective-C语言和C语言联系的4个事实。

事实#1：实例变量存储在结构中

定义一个新类和它的实例变量时，这些实例变量实际上存放在一个结构中。这说明了可以如何处理对象；对象实际上是结构，结构中的成员是实例变量。所以继承的实例变量加上你在类中添加的变量就组成了一个结构。使用alloc分配新对象时，系统预留了足够的空间来存储这些结构。

结构中继承的成员（从根对象中获得的）之一是名为isa的保护成员，它确定对象所属的类。因为它是结构的一部分（因此也是对象的一部分），所以由对象携带。这样，运行时系统只需通过查看isa成员，就可以确定对象的类（即使将其赋给通用的id对象变量）。

通过将成员定义为@public，可以获得对象结构成员的直接存储权限（详见第10章“变量和数据类型”）。举个例子，如果对Fraction类中的numerator和denominator成员进行了这项操作，那么可以在程序中编写如下表达式

```
myFract->numerator
```

来直接访问Fraction对象myFract的numerator成员。但是，我们强烈建议不要这么做！在第10章中提到过，它违反了数据封装的特性。

事实#2：对象变量实际上是指针

定义Fraction之类的对象变量时，如

```
Fraction *myFract;
```

事实上是定义了一个名为myFract的指针变量。这个变量定义为指向Fraction类型的数据，即你的类名称。使用

```
myFract = [Fraction alloc];
```

来创建Fraction的新实例时，是在为Fraction对象的新实例分配存储内存（即，存放结构的空间），然后使用结构的指针，并将指针变量myFract存储在其中。

将对象变量赋给另一个对象变量时，如下所示

```
myFract2 = myFract1;
```

只是简单地复制了指针。这两个变量最后都指向存储在内存中的同一结构。因此，改变myFract2引用的（即指向的）一个成员，将更改myFract1引用的同一个实例变量（即结构成员）。

事实#3：方法是函数，而消息表达式是函数调用

方法实际上是函数。调用方法时，是在调用与接收者类相关的函数。传递给函数的参数是接收者（self）和方法的参数。所以，无论是函数还是方法，关于传递参数给函数、返回值以及自动和静态变量的规则都是一样的。Objective-C编译器通过类名称和方法名称的组合为每个函数产生一个唯一的名称。

事实#4：id类型是通用指针类型

因为通过指针，也就是内存地址来引用对象，所以可以自由地将它们在id变量之间来回赋值。因此返回id类型值的方法只是返回指向内存中某对象的指针。然后可以将该值赋给任何对象变量。因为无论在哪里，对象总是携带它的isa成员，所以即使将它存储在id类型的通用对象变量中，也总是可以确定它的类。

13.9 练习

1. 编写一个函数，计算包含10个浮点数的数组的平均值并返回结果。
2. Fraction类中的方法reduce用来找出分子和分母的最大公约数来简约分数。修改这个方法，使其使用代码清单13-5中的函数gcd。你认为应该在什么地方定义该函数呢？将函数定为static有什么好处？你认为哪种方式更好，使用函数gcd还是像以前一样将代码直接合并到方法中呢？为什么？
3. 可以使用一种名为Sieve of Erastosthenes的算法产生素数。这个过程算法如下所示。编写一个程序来实现这个算法。假设程序找出150之前的所有质数，比较本文中其他计算质数的算法，如何评价这个算法？

   步骤1：定义整型数组P。将所有元素Pi置为0、2 <= i <= n。

   步骤2：将i设为2。

   步骤3：如果$i > n$，算法终止。

   步骤4：如果P_i等于0，则i是质数。

   步骤5：对于所有满足$i \cdot j < n$的正整数j，将P_{ixj}设为1。

   步骤6：i加1并且回到步骤3。
4. 编写一个函数，将所有传递给它的数组中的Fractions相加，并以Fraction类型返回结果。
5. 为名为Date的struct date编写定义typedef，它允许在程序中进行如下声明

```
Date todaysDate;
```

6. 在文中提到过，定义Date类而不是date结构更符合面向对象的编程思想。定义这样的类，并定义适当的setter和getter方法。同时，添加一个名为dateUpdate的方法来返回参数之后的日期。

   能看出将Date定义为类而不是结构的好处吗？

   能看出这样做的缺点吗？
7. 给出下列定义：

```
char *message = "Programming in Objective-C is fun";
char message2[] = "You said it";

int  x = 100;
```

确定下面语句集中的NSLog语句是否都合法，产生的输出是否和其他语句一样。

```
/*** set 1 ***/
```

```
NSLog (@"Programming in Objective-C is fun");
NSLog (@"%s","Programming in Objective C is fun");
NSLog (@"%s", message);
/*** set 2 ***/

NSLog (@"You said it");
NSLog (@"%s", message2);
NSLog (@"%s", &message2[0] ;

/*** set 3 ***/
NSLog (@"said it");
NSLog (@"%s", message2 + 4);
NSLog (@"%s", &message2[4]);
```

8. 编写程序，它在终端输出所有命令行参数，每行一个。注意引号中包含空格字符的封装参数有什么作用。
9. 下面哪组语句会生成输出This is a test?并说明原因。

```
NSLog (@"This is a test");
NSLog ("This is a test");

NSLog (@"%s", "This is a test");
NSLog (@"%s", @"This is a test");

NSLog ("%s", "This is a test");
NSLog ("%s", @"This is a test");

NSLog (@"%@", @"This is a test");
NSLog (@"%@", "This is a test");
```

第二部分　Foundation框架

第14章　Foundation框架简介

框架是由许多类、方法、函数、文档按照一定的逻辑组织起来的集合，以便使研发程序变得更容易。在OS X下的Mac操作系统中，大约存在80个框架，这些框架可以用来开发应用程序，处理Mac的Address Book结构、刻制CD、播放DVD、使用QuickTime播放电影、播放歌曲，等等。

为所有程序开发奠定基础的框架称为Foundation框架。该框架是本书第二部分的主题。它允许使用一些基本对象，如数字和字符串，以及一些对象集合，如数组、字典和集合。其他功能包括处理日期和时间、自动化的内存管理、处理基础文件系统、存储(或归档) 对象、处理几何数据结构（如点和长方形）。

Application Kit框架包含广泛的类和方法，它们用来开发交互式图形应用程序，使得开发文本、菜单、工具栏、表、文档、剪贴板和窗口之类的过程变得十分简便。在Mac OS X操作系统中，术语Cocoa总的来说指的是Foundation框架和Application kit框架。术语Cocoa Touch是指Foundation框架和UIKit框架。本书的第三部分会介绍有关这一主题的详细信息。附录D“资源”中也列出了关于这个主题的许多资源。

Foundation文档

为了参考，应该知道Foundation头文件存储在目录`/System/Library/Frameworks/Foundation.framework/Headers`中。

注意　头文件实际上与其存储位置的其他目录相链接，但是这对你的操作没有任何影响。

查看这个目录，熟悉它的内容，应该利用存储在系统上的Foundation框架文档。该文档存储在你的系统中（位于/Developer/Documentation目录中），Apple的网站上也提供了。大多数文档为HTML格式的文件，可以通过浏览器查看，同时也提供了Acrobat pdf文件。这个文档中包

含Foundation的所有类及其实现的所有方法和函数的描述。

如果正在使用Xcode开发程序，可以通过Xcode的Help菜单中的Documentation窗口轻松访问文档。通过这个窗口，可以轻松搜索和访问存储在计算机本机中或者在线的文档。图14-1显示了在Xcode文档窗口中搜索字符串“foundation framework”的结果。从这个显示头文件Foundation Framework Reference的面板中，可以轻松访问所有Foundation类的文档。

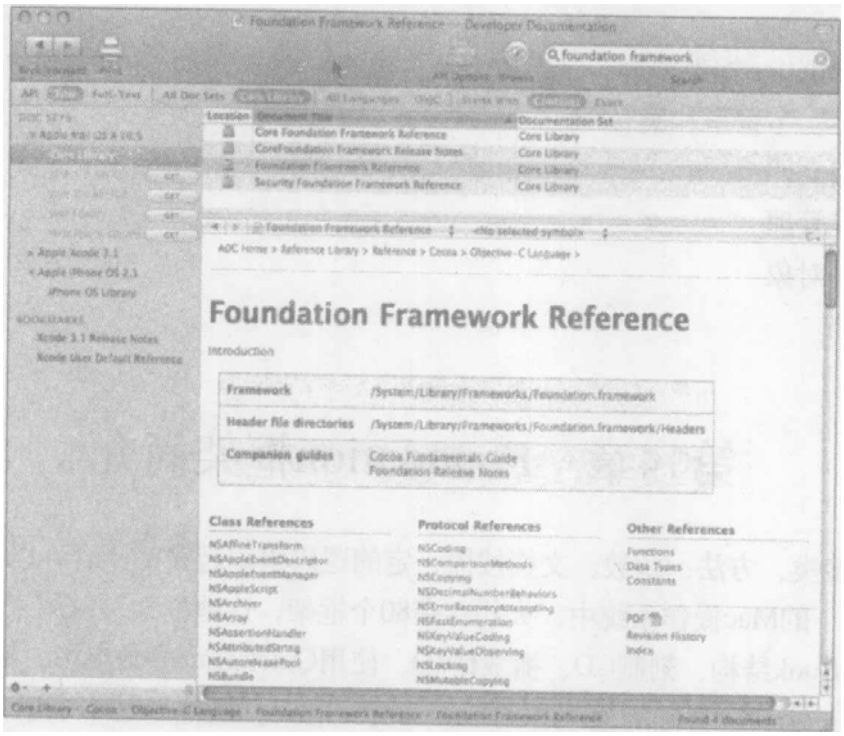

图14-1　使用Xcode访问Foundation参考文档

如果正在Xcode中编辑文件并且想要快速访问某个特定头文件、方法或类的文档，可以通过高亮显示编辑器窗口中的文本并右键单击的方法来实现。在出现的菜单中，可以适当选择Find Selected Text in Documentation或者Find Selected Text in API Reference。Xcode将搜索文档库并显示与查询相匹配的结果。

看一看它是如何工作的。NSString类是一个Foundation类，可以使用它来处理字符串（下一章将详细介绍这部分内容）。假设正在编辑某个使用该类的程序，并且想要获得更多关于这个类及其方法的信息，无论何时，当单词NSString出现在编辑窗口时，都可以将其高亮显示并右键单击。如果从出现的菜单中选择Find Selected Text in API Reference，会得到一个外观与图14-2类似的文档窗口。

如果向下滚动标有NSString Class Reference的面板，将发现（在其他内容中间）一个该类所支持的所有方法的列表。这是一个能够获得有关类实现哪些方法等信息的便捷途径，包括它们如何工作以及它们的预期参数。

还可以在线访问文档。地址是developer.apple.com/referencelibrary，打开Foundation参考文档（通过Cocoa、Frameworks、Foundation Framework Reference链接）。在这个站点中，还能够发现一大类介绍某些特定编程问题的文档，比如内存管理、字符串和文件管理。

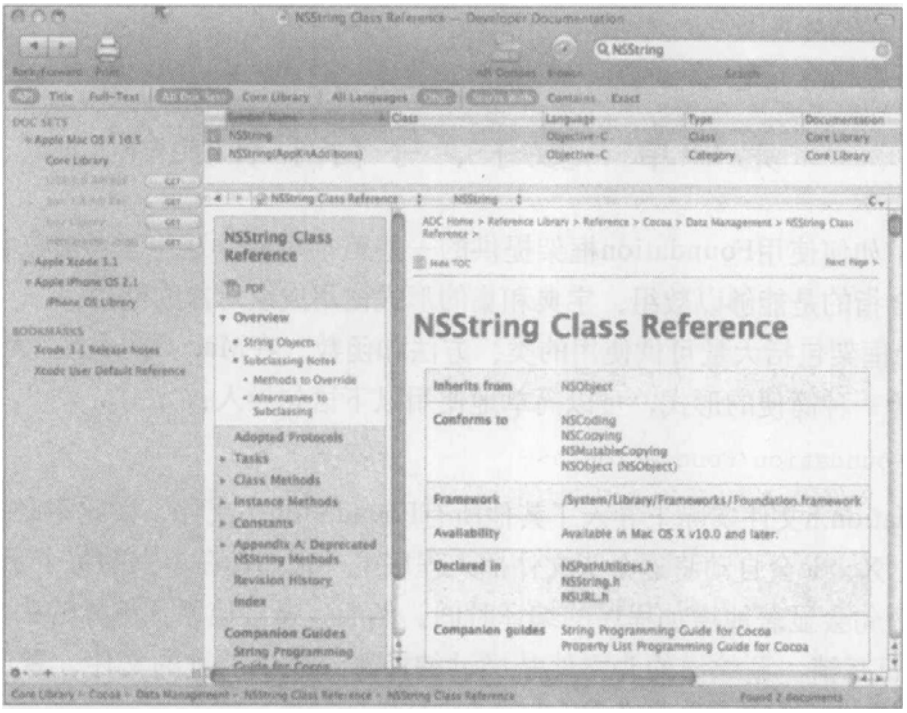

图14-2 NSString类的文档

除非订阅的是某个特定文档集，否则从时间上讲在线文档要比存储在计算机硬盘中的文档更新。

以上对Foundation框架进行了简短介绍。现在，可以了解有关它的类的一些信息以及如何使它们在应用程序中工作。

第15章　数字、字符串和集合

本章讲解了如何使用Foundation框架提供的一些基本对象。这些基本对象包括数字、字符串和集合，集合指的是能够以数组、字典和集的形式使用成组对象的能力。

Foundation框架包括大量可供使用的类、方法和函数。在Mac OS X上，大约有125个可用的头文件。作为一种简便的形式，可以简单地使用以下语句导入：

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

因为Foundation.h文件实际上导入了其他所有Foundation头文件，因此不必担心是否导入了正确的头文件。Xcode会自动将这个头文件插入程序中，正如你在本书的每个示例中所见到的。

使用这条语句会显著地增加程序的编译时间。然而，通过使用预编译的头文件，可以避免这些额外的时间开销。预编译的头文件是经过编译器预先处理过的文件。默认情况下，所有Xcode项目都会受益于预编译的头文件。

本章使用的每个对象都将用到这些特定的头文件。这有助于你熟悉每个头文件所包含的内容。

注意　如果喜欢，可以继续导入语句Foundation.h，如果确实导入了每个例子中显示的单个文件，还应该删除Project_name_prefix.pch文件，该文件是在创建新的Foundation Tool项目时Xcode自动包含进去的。在从项目中删除文件时，需要确保在Xcode提示时选择Delete Reference。

15.1　数字对象

到目前为止，我们所讨论过的所有数字数据类型，如int型、float型和long型都是Objective-C语言中的基本数据类型，也就是说，它们都不是对象。例如，不能向它们发送消息。然而，有时需要作为对象使用这些值。例如，使用Foundation的对象NSArray，可以设置一个用于存储值的数组。这些值必须是对象，因此不能将任何基本数据类型直接存储到这些数组中。要存储任何基本数据类型（包括char数据类型），可以使用NSNumber类根据这些数据类型来创建对象（参见代码清单15-1）。

代码清单15-1

```
// Working with Numbers

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSValue.h>

#import <Foundation/NSString.h>
```

```
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSNumber          *myNumber, *floatNumber, *intNumber;
    NSInteger          myInt;

    // integer value

    intNumber = [NSNumber numberWithInteger: 100];
    myInt = [intNumber integerValue];
    NSLog (@ "%li ", (long) myInt);

    // long value

    myNumber = [NSNumber numberWithLong: 0xabcdef];
    NSLog (@ "%l x ", [myNumber longValue]);

    // char value

    myNumber = [NSNumber numberWithChar: 'X'];
    NSLog (@ "%c ", [myNumber charValue]);

    // float value

    floatNumber = [NSNumber numberWithFloat: 100.00];
    NSLog (@ "%g ", [floatNumber floatValue]);

    // double

    myNumber = [NSNumber numberWithDouble: 12345e+15];
    NSLog (@"%lg", [myNumber doubleValue]);

    // Wrong access here

    NSLog (@"%i", [myNumber integerValue]);

    // Test two Numbers for equality

    if ([intNumber isEqualToNumber: floatNumber] == YES)
        NSLog (@"Numbers are equal");
    else
        NSLog (@"Numbers are not equal");

    // Test if one Number is <, ==, or > second Number

    if ([intNumber compare: myNumber] == NSOrderedAscending)
        NSLog (@"First number s less than second");
```

```
    [pool drain];
    return 0;
}
```

代码清单15-1　输出

```
100
abcdef
X
100
1.2345e+19
0
Numbers are equal
First number is less than second
```

使用NSNumber类中的对象时，接口文件<Foundation/NSValue.h>是必需的。

自动释放池一览

代码清单15-1中的第一个程序行在本书中的任何一个程序中都会出现。下面这行代码为你分配给pool的自动释放池（autorelease pool）预留了内存空间：

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

自动释放池可以自动释放添加到该池中的对象所使用的内存。向对象发送一条autorelease消息时，就将该对象放到这个池中。释放这个池时，添加到该池的所有对象也会一起释放。因此，所有这样的对象都会被销毁，除非已指明这些对象所在的作用域超出自动释放池（如，使用引用计数指明）。

一般来说，无须担心需要释放Foundation方法返回的对象。有时候，对象由返回它的方法所拥有。其他情况下，对象是由方法新创建的并被添加到自动释放池中。第一部分中讲过，使用完用alloc方法显式地创建的任何对象（包括 Foundation对象）之后，仍然需要释放它们。

注意　此外，还需要释放由复制行创建的对象，这部分内容将在第17章介绍。

第17章将十分详细地讲述引用计数和自动释放池。

回到代码清单15-1。NSNumber类包含多个方法，它们允许使用初始值创建NSNumber对象。例如，程序行：

```
intNumber = [NSNumber numberWithInteger: 100];
```

创建了一个值为100的整数对象。

从NSNumber对象获得的值必须和存储在其中的值类型一致。因此，在程序中该语句之后的printf语句中，消息表达式

```
[intNumber integerValue]
```

检索存储在intNumber中的整型值，并将其存储在NSInteger变量myInt中。注意，NSInteger不是一个对象，而是基本数据类型的typedef。它被typedef成64位的long或者32位的int。存在一个

类似的NSInteger typedef用于处理程序中那些未签名的整数。

在NSLog调用中，将NSInteger转换为long并使用格式字符%li，以确保值可以传递并正确显示，即使程序编译后是32位架构的。

对于每个基本值，类方法都为它分配了一个NSNumber对象，并将其设置为指定的值。这些方法以numberWith开始，之后是该方法的类型，如numberWithLong:、numberWithFloat:等。此外，可以使用实例方法为以前分配的NSNumber对象设置指定的值。这些都是以initWith开头的，如initWithLong:和initWithFloat:。

表15-1列出了为NSNumber对象设置值的类和实例方法，以及检索这些值的相应实例方法。

表15-1 NSNumber的创建方法和检索方法

创建和初始化类方法	初始化实例方法	检索实例方法
numberWithChar:	initWithChar:	charValue
numberWithUnsignedChar:	initWithUnsignedChar:	unsignedCharValue
numberWithShort:	initWithShort:	shortValue
numberWithUnsignedShort:	initWithUnsignedShort:	unsignedShortValue
numberWithInteger:	initWithInteger:	integerValue
numberWithUnsignedInteger:	initWithUnsignedInteger:	unsignedIntegerValue
numberWithInt:	initWithInt:	intValue
numberWithUnsignedInt:	initWithUnsignedInt:	unsignedIntValue
numberWithLong:	initWithLong:	longValue
numberWithUnsignedLong:	initWithUnsignedLong:	unsignedLongValue
numberWithLongLong:	initWithLongLong:	longlongValue
numberWithUnsignedLongLong:	initWithUnsignedLongLong:	unsignedLongLongValue
numberWithFloat:	initWithFloat:	floatValue
numberWithDouble:	initWithDouble:	doubleValue
numberWithBool:	initWithBool:	boolValue

回到代码清单15-1，该程序接下来使用类方法创建了long、char、float和double NSNumber对象。注意，使用程序行

```
myNumber = [NSNumber numberWithDouble: 12345e+15];
```

创建double对象后将出现什么情况？然后尝试（不正确地）使用如下程序行来检索并显示它的值：

```
NSLog (@"%i", [myNumber integerValue]);
```

将得到以下输出：

```
0
```

并且，系统也没有给出出错消息。一般来说，你负责确保正确地进行检索，如果在NSNumber对象中存储了一个值，那么也要用一致的方式进行检索。

在if语句中，消息表达式

```
[intNumber isEqualToNumber: floatNumber]
```

使用isEqualToNumber:方法根据数值比较两个NSNumber对象。该程序测试返回的Boolean值，

以查看这两个值是否相等。

可用compare:方法来测试一个数值型的值是否在数值上小于、等于或大于另一个值。消息表达式

```
[intNumber compare: myNumber]
```

在intNumber中的值小于myNumber中的值时，返回值NSOrderedAscending；如果这两个数相等，则返回值NSOrderedSame；如果第一个值大于第二个值，则返回值NSOrderedDescending。在头文件NSobject.h中已经定义了这些返回值。

应该注意不能重新初始化前面创建的NSNumber对象的值。例如，不能使用下面的语句设置存储在NSNumber对象myNumber中的整数：

```
[myNumber initWithInt: 1000];
```

当程序执行时，这条语句将产生一条错误。所有数字对象都必须是新创建的，这意味着必须对NSNumber类调用表15-1第一列列出一个方法，或者对alloc方法的结果调用第二列列出的方法中，如下所示：

```
myNumber = [[NSNumber alloc] initWithInt: 1000];
```

当然，基于前面的讨论，如果使用这种方式创建myNumber，则在使用完之后，你需要使用以下语句来释放它：

```
[myNumber release];
```

本章其余部分将再次在程序中遇到NSNumber对象。

15.2　字符串对象

前面在程序中已经遇到过字符串，只要使用一对双引号括住一组字符串时，如下所示

```
@"Programming is fun"
```

就是使用Objective-C语言创建了一个字符串。Foundation框架支持一个名为NSString的类，它用于处理字符串对象。然而C-string由char字符组成，NSString对象是由unichar字符组成。Unichar字符是符合Unicode标准的多字节字符。这样，就可以处理包含数百万字符的字符集。幸运的是，不必担心字符串的内部表示，因为NSString类[⊖]已经自动地做了这些工作。使用该类的方法，可以更容易地开发能够本地化的应用程序，即，使其能够在全世界不同语言环境下使用。

要使用Objective-C语言创建一个常字符串对象，需要在字符串开头放置一个@字符，于是，表达式

```
@"Programming is fun"
```

创建了一个常量字符串对象。在特殊情况下，它是属于NSConstantString类的常量字符串对象。NSConstantString类是字符串对象NSString类的子类。在你的程序中使用字符串对象，要包括

⊖ 目前，unichar字符占用16位，而Unicode标准提供的大小大于16位。因此，将来，unichar字符可能大于16位，底线是绝对不要对Unicode字符的大小作任何假定。

下面的一行：

```
#import <Foundation/NSString.h>
```

15.2.1 NSLog函数

随后的代码清单15-2，展示了如何定义NSString对象，并向其指派一个初始化值。还展示了如何使用格式字符%@来显示NSString对象。

代码清单15-2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *str = @ "Programming is fun ";

   NSLog (@ "%@ ", str);

   [pool drain];
   return 0;
}
```

代码清单15-2 输出

```
Programming is fun
在以下代码行中
  NSString *str = @'Programming is fun";
```

常量字符串对象Programming is fun被赋值给NSString变量str。然后使用NSLog来显示它的值。

NSLog格式字符%@不仅能显示NSString对象，而且可以显示其他对象。例如，给定以下语句：

```
NSNumber *intNumber = [NSNumber numberWithInteger: 100];
```

NSLog调用

```
NSLog (@"%@", intNumber);
```

将产生如下输出：

```
100
```

格式字符%@甚至能够显示数组、字典和集合的全部内容。事实上，通过重载你的类所继承的方法，还可使用这些格式字符显示你自己的类对象。如果不重载方法，NSLog仅仅显示类名和该对象在内存中的地址（那是从NSObject类继承的description方法的默认实现）。

15.2.2 可变对象与不可变对象

通过编写如下语句：

```
@"Programming is fun"
```

创建字符串对象时，就创建了一个内容不可更改的对象。这称作不可变对象。NSString类处理不可变字符串。你经常需要处理字符串并更改字符串中的字符。例如，可能想从字符串中删除一些字符或对字符串执行搜索替换操作。这些类型的字符串是使用NSMutableString类处理的。

代码清单15-3显示在程序中处理不可变字符串的基本方式。

代码清单15-3

```
// Basic String Operations

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   NSString *str1 = @ "This is string A ";
   NSString *str2 = @ "This is string B ";
   NSString *res;
   NSComparisonResult compareResult;

   // Count the number of characters

   NSLog (@ "Length of str1: %lu ", [str1 length]);

   // Copy one string to another

   res = [NSString stringWithString: str1];
   NSLog (@ "copy: %@ ", res);

   // Copy one string to the end of another

   str2 = [str1 stringByAppendingString: str2];
   NSLog (@ "Concatentation: %@ ", str2);

   // Test if 2 strings are equal

   if ([str1 isEqualToString: res] == YES)
      NSLog (@ "str1 == res ");
   else
      NSLog (@ "str1 != res ");
```

```
    // Test if one string is <, == or > than another

    compareResult = [str1 compare: str2];

    if (compareResult == NSOrderedAscending)
       NSLog (@ "str1 < str2 ");
    else if (compareResult == NSOrderedSame)
       NSLog (@ "str1 == str2 ");
    else // NSOrderedDescending
       NSLog (@ "str1 > str2 ");
       .
    // Convert a string to uppercase

    res = [str1 uppercaseString];
    NSLog (@ "Uppercase conversion: %s ", [res UTF8String]);

    // Convert a string to lowercase

    res = [str1 lowercaseString];
    NSLog (@ "Lowercase conversion: %@ ", res);

    NSLog (@ "Original string: %@ ", str1);

    [pool drain];
    return 0;
}
```

代码清单15-3 输出

```
Length of str1: 16
Copy: This is string A
Concatentation: This is string AThis is string B
str1 == res
str1 < str2
Uppercase conversion: THIS IS STRING A
Lowercase conversion: this is string a
Original string: This is string A
```

代码清单15-3首先定义了3个不可变的NSString对象：str1、str2和res。前两个初始化为常字符串对象。声明

```
NSComparisonResult compareResult;
```

声明了compareResult保存该程序后面将要执行的字符串比较操作的结果。

Length方法可以用来对字符串中的字符进行计数。输出验证了字符串

```
@"This is string A"
```

包含16个字符。

语句

```
res = [NSString stringWithString: str1];
```

展示了如何使用另一个字符串的内容来生成一个新字符串。结果NSString对象被赋值给res，然后显示以验证结果。实际的字符串内容复制是在这里进行的，而不是对内存中的同一字符串的引用。即str1和res指向两个不同的字符串对象，这与简单地执行如下赋值操作是不同的：

```
res = str1;
```

这仅仅创建了内存中同一对象的另一个引用。

stringByAppendingString:方法可以用来连接两个字符串。所以，表达式

```
[str1 stringByAppendingString: str2]
```

创建了一个新对象，这个对象由str1之后是str2的字符组成，返回结果。这项操作没有改变原字符串对象str1和str2（它们不能更改，这是因为它们都是不可变字符串对象）。

然后使用isEqualToString:方法来检测两个字符串是否相等，即，是否包含相同的字符。如果需要确定两个字符串的顺序，例如，如果要对字符串数组进行排序。可以使用compare:方法来代替。与前面用来比较两个NSNumber对象的compare:方法相似：如果从词汇上说第一个字符串小于第二个字符串，则结果是NSOrderedAscending；如果两个相等，则结果是NSOrderedSame；如果第一个字符串大于第二个，则结果是NSOrderedDescending。如果不想执行大小写敏感的比较，则使用caseInsensitiveCompare:方法而不是compare:方法。在这个例子中，使用caseInsensitiveCompare:比较两个字符串对象@“Gregory”和@“gregory”，结果是相等。

uppercaseString和lowercaseString方法是代码清单15-3中使用的最后两个NSString方法，它们分别将字符串转换成大写字符和小写字符。再次提醒一下，该操作没有改变原字符串，最后一行输出可以证明。

代码清单15-4举例说明额外的字符串处理方法。这些方法允许你提取字符串中的子字符串，并且能够在一个字符串中搜索另一个字符串。

一些方法需要指定范围来确定子字符串。范围包括开始索引数和字符计数，索引数以0开始，因此使用数字对{0, 3}指定字符串中前3个字符。NSString类（和其他的Foundation类）中的一些方法使用了特殊的数据类型NSRange来创建范围指定。它定义在<Foundation/NSRange.h>（<Foundation/NSString.h>中已经包括这个头文件）中，实际上，它是结构的typedef定义，该结构包含location和length两个成员。代码清单15-4中使用了这个数据类型。

注意 第13章讲述了结构。然而，你能够从本章之后的讨论中学到足够信息来处理它们。

代码清单15-4

```
// Basic String Operations - Continued

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
```

```
int main (int argc, char *argv[])
{
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   NSString *str1 = @ "This is string A ";
   NSString *str2 = @ "This is string B ";
   NSString *res;
   NSRange subRange;

   // Extract first 3 chars from string

   res = [str1 substringToIndex: 3];
   NSLog (@ "First 3 chars of str1: %@ ", res);

   // Extract chars to end of string starting at index 5

   res = [str1 substringFromIndex: 5];
   NSLog (@ "Chars from index 5 of str1: %@ ", res);

   // Extract chars from index 8 through 13 (6 chars)

   res = [[str1 substringFromIndex: 8] substringToIndex: 6];
   NSLog (@ "Chars from index 8 through 13: %@ ", res);

   // An easier way to do the same thing

   res = [str1 substringWithRange: NSMakeRange (8, 6)];
   NSLog (@ "Chars from index 8 through 13: %@ ", res);

   // Locate one string inside another

   subRange = [str1 rangeOfString: @ "string A "];
   NSLog (@ "String is at index %lu, length is %lu ",
      subRange.location, subRange.length);


   subRange = [str1 rangeOfString: @ "string B "];

   if (subRange.location == NSNotFound)
      NSLog (@ "String not found ");
   else
      NSLog (@ "String is at index %lu, length is %lu ",
         subRange.location, subRange.length);

   [pool drain];
   return 0;
}
```

代码清单15-4 输出

```
First 3 chars of str1: Thi
Chars from index 5 of str1: is string A
Chars from index 8 through 13: string
Chars from index 8 through 13: string
String is at index 8, length is 8
String not found
```

substringToIndex:方法创建了一个子字符串，它包括首字符直到执行的索引数，但是不包括这个字符。因为索引数是从0开始的，所以参数3表示从字符串中提取字符0、1和2，并返回结果字符串对象。对于所有采用索引数作为参数的字符串方法，如果提供的索引数对该字符串无效，就会获得Range or index out of bounds的出错消息。

substringFromIndex:方法返回了一个子字符串，它从接收者的指定索引的字符开始，直到字符串的结尾。

表达式

```
res = [[str1 substringFromIndex: 8] substringToIndex: 6];
```

显示了如何结合这两个方法，提取字符串内部的子字符串。首先使用substringFromIndex:方法从索引数8开始直到字符串结尾的字符，然后对结果应用substringToIndex:方法，以获得前6个字符。最终结果是一个子字符串，它是原字符串中{8, 6}范围的字符。

substringWithRange:方法用一步完成我们刚刚用两步所做的工作：它接受了一个范围，并返回指定范围的字符。特殊函数

```
NSMakeRange (8,6)
```

根据其参数创建了一个范围，并返回该结果。这个结果可以作为substringWithRange:方法的参数。

要在另一个字符串中查找一个字符串，可以使用rangeOfString:方法。如果在接收者中找到了指定的字符串，则返回的范围精确地指定找到它的位置。然而，如果没有找到这个字符串，则返回范围的location成员被设为NSNotFound。

所以，语句

```
subRange = [str1 rangeOfString: @"string A"];
```

把该方法返回的NSRange结构的赋值给NSRange变量subRange。一定要注意subRange不是对象变量，而是一个结构变量（并且程序中的subRange声明不包括星号）。通过使用结构成员操作符（.），可以检索其成员。所以，表达式subRange.location给出了该结构中成员location的值，subRange.length给出该结构中成员length的值。这些值被传递给NSLog函数以显示。

15.2.3 可变字符串

NSMutableString类可以用来创建可以更改字符的字符串对象。因为该类是NSString类的子类，所以可以使用NSString类的所有方法。

讲述可变与不可变字符串对象时，我们讲到了更改字符串中的实际字符。任一可变或不可变字符串对象在程序执行期间，总是可以被设为完全不同的字符串对象。例如，考虑以下代码：

```
str1 = @"This is a string";
       ...
str1 = [str1 substringFromIndex: 5];
```

在这个例子中，首先将str1设置为一个常量字符串对象。后来在程序中将其设为一个子字符串。在这个例子中，str1可以声明为可变的字符串对象，也可以声明为不可变的字符串对象。一定要理解这一点。

代码清单15-5展示了处理程序中可变字符串的几种方式。

代码清单15-5

```
// Basic String Operations - Mutable Strings

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *str1 = @ "This is string A ";
   NSString *search, *replace;
   NSMutableString *mstr;
   NSRange substr;

   // Create mutable string from nonmutable

   mstr = [NSMutableString stringWithString: str1];
   NSLog (@ "%@ ", mstr);

   // Insert characters

   [mstr insertString: @ "mutable "atIndex: 7];
   NSLog (@ "%@ ", mstr);

   // Effective concatentation if insert at end

   [mstr insertString: @ "and string B "atIndex: [mstr length]];
   NSLog (@ "%@ ", mstr);

   // Or can use appendString directly

   [mstr appendString: @ "and string C "];
   NSLog (@ "%@ ", mstr);

   // Delete substring based on range

   [mstr deleteCharactersInRange: NSMakeRange (16, 13)];
```

```
NSLog (@ "%@ ", mstr);

// Find range first and then use it for deletion

substr = [mstr rangeOfString: @ "string B and"];

if (substr.location != NSNotFound) {
   [mstr deleteCharactersInRange: substr];
   NSLog (@ "%@ ", mstr);
}

// Set the mutable string directly

[mstr setString: @ "This is string A "];
NSLog (@ "%@ ", mstr);

// Now let's replace a range of chars with another

[mstr replaceCharactersInRange: NSMakeRange(8, 8)
                    withString: @ "a mutable string "];
NSLog (@ "%@ ", mstr);

// Search and replace

search = @ "This is ";
replace = @ "An example of ";

substr = [mstr rangeOfString: search];

if (substr.location != NSNotFound) {
   [mstr replaceCharactersInRange: substr
                       withString: replace];
   NSLog (@ "%@ ", mstr);
}

// Search and replace all occurrences

search = @ "a";
replace = @ "X";

substr = [mstr rangeOfString: search];

while (substr.location != NSNotFound) {
   [mstr replaceCharactersInRange: substr
                       withString: replace];
   substr = [mstr rangeOfString: search];
}
```

```
    NSLog (@ "%@ ", mstr);

    [pool drain];
    return 0;
}
```

代码清单15-5 输出

```
This is string A
This is mutable string A
This is mutable string A and string B
This is mutable string A and string B and string C
This is mutable string B and string C
This is mutable string C
This is string A
This is a mutable string
An example of a mutable string
An exXmple of X mutXble string
```

声明

```
NSMutableString *mstr;
```

将mstr变量声明为一个变量，用来存储在程序执行过程中值可能更改的字符串对象。语句行

```
mstr = [NSMutableString stringWithString: str1];
```

将mstr设置为字符串对象，其内容是str1中的字符的副本，或”This is string A”。将stringWithString:方法发送给NSMutableString类时，返回了一个可变的字符串对象。而将stringWithString:方法发送给NSString类时，如代码清单15-5所示，则返回一个不可变的字符串对象。

insertString:atIndex:方法将指定的字符串插入接收者，插入点从指定的索引值开始。在这个例子中，在字符串的索引数7或者第8字符处插入字符串@“mutable”。与不可变字符串对象不同，这里没有返回值，因为被修改的是接收者，因为它是可变的字符串对象，所以可以这么做。

第二个insertString:atIndex:调用使用length方法将一个字符串插入另一个字符串结尾。appendString:使得这个任务变得简单一些。

通过使用deleteCharactersInRange:方法，可以删除字符串中指定数目的字符。对以下字符串

```
This is mutable string A and string B and string C
```

应用范围{16, 13}时，从索引数16（或者字符串中的第17个字符）开始删除13个字符“string A and”。如图15-1所示。

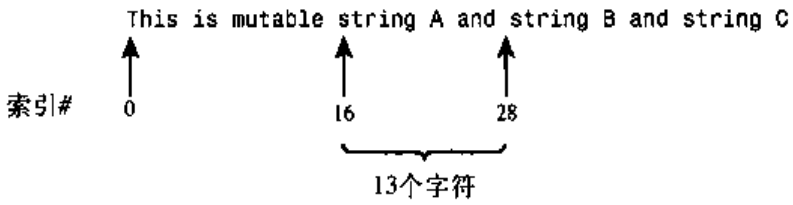

图15-1 字符串中的索引

代码清单15-5之后的rangeOfString:方法用来展示如何首先找到然后删除字符串。首先验证mstr中的确存在字符串@“string B and”，然后使用rangeOfString:方法返回的范围作为deleteCharactersInRange:方法的参数来删除这个字符串。

setString:方法可以用来直接设置可变字符串对象的内容。使用该方法将mstr设置为字符串@“This is string A”之后，replaceCharactersInRange:方法用另一个字符串来替换这个字符串中的部分字符。字符串的大小不必相同，可以使用大小相等或不等的字符串来替换另一个字符串。因此，语句

```
[mstr replaceCharactersInRange: NSMakeRange(8, 8)
                    withString: @"a mutable string"];
```

的结果是8个字符“string A”被替换成16个字符“a mutable string”。

这个程序例子中的其余几行展示了如何执行搜索和替换操作。首先，在字符串mstr中（它包含@“This is a mutable string”）查找字符串@“This is”。如果在搜索字符串中找到该内容，就用字符串@“An example of”替换匹配的字符串。最终mstr中包含的字符串变为@“An example of a mutable string”。

然后，程序设置了一个循环来显示如何实现替换并全部替换操作。搜索字符串被设置为@“a”，替换字符串被设置为@“x”。

如果替换字符串还包括搜索字符串（例如，考虑使用字符串“aX”替换字符串“a”），那么将会陷入无限循环。

其次，如果替换字符串为空（也就是，不包含字符），那么将有效地删除所有搜索字符串。通过没有空格隔开的相邻引号可以指定空的常量字符串，如下所示：

```
replace = @"";
```

当然，如果只想删除字符串，则可以使用deleteCharactersInRange:方法，前面我们已经学过。

最后，NSMutablestring类还包含一个名为replaceOccurrencesOfString:withString:options:range:的方法，它可以用来执行搜索并全部替换。实际上，代码清单15-5中的while循环可以替换为如下一行代码：

```
[mstr replaceOccurrencesOfString: search
                      withString: replace
                         options: nil
                           range: NSMakeRange (0, [mstr length])];
```

这将获得相同的结果，并且避免了潜在的无限循环，因为该方法阻止这样的事情发生。

15.2.4 所有对象到哪里去了

代码清单15-4和15-5处理各种NSString和NSMutableString方法生成并返回的字符串对象。本章开始讨论过，你不负责释放这些对象使用的内存，该对象的创建者负责释放。推测起来，所有被创建者添加到自动释放池的对象，将在系统释放池时全部释放。然而应该意识到，如果开发的程序中创建了许多临时的对象，那么这些对象使用的内存就会累积起来。在这种情形下，可能需要采取不同的策略，允许在程序执行过程中释放内存，而不只是在程序结尾释放。这个

概念将在第17章中描述。现在，只需认识到在执行这个程序时，这些对象占用的内存会扩张。

NSString类包含100多个方法，它可以用来处理不可变的字符串对象。表15-2总结了一些更常用的方法，表15-3列出了NSMutableString类提供的一些附加方法。其他一些NSString方法（例如处理路径名和将文件的内容读入一个字符串）将在本书剩余部分进行介绍。

在表15-2和15.3中，url是一个NSURL对象，path是指明文件路径的NSString对象，nsstring是一个NSString对象，i是表示字符串中有效字符数的NSUInteger值，enc是指明字符编码的NSStringEncoding对象，err是描述所发生错误的NSError对象，size和opts是NSUInteger，range是指示字符串中有效范围的NSRange对象。

表15-2 常见的NSString方法

方 法	描 述
+(id) stringWithContentsOfFile: path encoding: enc error: err	创建一个新字符串并将其设置为path指定的文件的内容，使用字符编码enc，如果非零，则返回err中的错误
+(id) stringWithContentsOfURL: url encoding: enc error: err	创建一个新字符串，并将其设置为url的内容，使用字符编码enc，如果非零，则返回err中的错误
+(id) string	创建一个新的空字符串
+(id) stringWithString: nsstring	创建一个新字符串，并将其设置为nsstring
-(id) initWithString: nsstring	将新分配的字符串设置为nsstring
-(id) initWithContentsOfFile: path encoding: enc error: err	将字符串设置为path指定的文件的内容
-(id) initWithContentsOfURL: url encoding: enc error: err	将字符串设置为url (NSURL *) url的内容，使用字符编码 enc，如果非零，则返回err中的错误
-(UNSIgned int) length	返回字符串中的字符数目
-(unichar) characterAtIndex: i	返回索引i的Unicode字符
- (NSString *)substringFromIndex: i	返回从i开始直到结尾的子字符串
- (NSString *)substringWithRange: range	根据指定范围返回子字符串
-(NSString *) substringToIndex: i	返回从该字符串开始位置到索引i的子字符串
-(NSComparator *)caseInsensitiveCompare: nsstring	比较两个字符串，忽略大小写
-(NSComparator *) compare: nsstring	比较两个字符串
-(BOOL) hasPrefix: nsstring	测试字符串是否以nsstring开始
-(BOOL) hasSuffix: nsstring	测试字符串是否以nsstring结尾
-(BOOL) isEqualToString: nsstring	测试两个字符串是否相等
- (NSString *) capitalizedString	返回每个单词首字母大写的字符串（每个单词的其余字母转换为小写）
-(NSString *) lowercaseString	返回转换为小写的字符串
-(NSString *) uppercaseString	返回转换为大写的字符串
-(const char *) UTF8String	返回转换为UTF-8字符串的字符串
-(double) doubleValue	返回转换为double的字符串
-(float) floatValue	返回转换为浮点值的字符串
-(NSInteger) integerValue	返回转换为NSInteger整数的字符串
-(int) intValue	返回转换为整数的字符串

表15-3中的方法创建或修改NSMutableString对象。

表15-3 常见的NSMutableString方法

方 法	描 述
+(id) stringWithCapacity: size	创建一个字符串，初始包含size的字符
-(id) initWithCapacity: size	使用初始容量为size的字符来初始化字符串
-(void) setString: nsstring	将字符串设置为nsstring
-(void) appendString: nsstring	在接收者的末尾附加nsstring
-(void) deleteCharactersInRange: range	删除指定range中的字符
-(void) insertString: nstring atIndex: i	以索引i为起始位置插入nsstring
-(void) replaceCharactersInRange: range withString: nsstring	使用nsstring替换range指定的字符
-(void) replaceOccurrencesOf String: nsstring withString: nsstring2 options: opts range: range	根据选项opts。使用指定range中的nssring2替换所有nsstring。选项可以包括NSBackwardsSearch（从范围的结尾开始搜索）、NSAnchoredSearch（nsstring必须匹配范围的开始）、NSLiteralSearch（执行逐字节比较以及NSCaseInsensitiveSearch的按位或组合）

NSString对象广泛地用于本文的其余部分。如果需要把字符串分解为语言符号，可以查看Foundation的NSScanner类。

15.3 数组对象

Foundation数组是有序的对象集合。最常见的情况是，一个数组中的元素都是一个特定类型，但这不是必需的。就像存在可变字符串和不可变字符串，同样也存在可变数组和不可变数组。不可变数组是由NSArray类处理的，而可变数组是由NSMutableArray处理的。后者是前者的子类，就是说后者继承前者的方法。要在你的程序中使用数组对象，包括下面这行代码：

```
#import <Foundation/NSArray.h>
```

代码清单15-6设置一个数组，用来存储一年中的月份名字，然后显示这些月份。

代码清单15-6

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
    int i;
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // Create an array to contain the month names

    NSArray *monthNames = [NSArray arrayWithObjects:
        @"January ", @ "February ", @ "March ", @ "April ",
```

```
        @"May", @ "June", @ "July", @ "August", @ "September",
        @"October", @ "November", @ 'December', nil ];

    // Now list all the elements in the array

    NSLog (@ 'Month Name ");
    NSLog (@ "===== ==== ");

    for (i = 0; i < 12; ++i)
        NSLog (@ "%2i %@ ", i + 1, [monthNames objectAtIndex: i]);

    [pool drain];
    return 0;
}
```

代码清单15-6 输出

```
Month   Name
=====   ====
 1      January
 2      February
 3      March
 4      April
 5      May
 6      June
 7      July
 8      August
 9      September
10      October
11      November
12      December
```

类方法arrayWithObjects:可以用来创建使用一列对象作为元素的数组。在这种情况下，按顺序列出对象并使用逗号隔开。这是方法使用的特殊语法，这个方法可以接受可变数目的参数。要标记参数列表的结束，必须把该列表的最后一个值指定为nil，它实际上并不存储在数组中。

在代码清单15-7中，monthNames被设置为arrayWithObjects:的参数所指定的12个字符串。

数组中的元素是由它们的索引数确定的。与NSString对象类似，索引从0开始。所以，包含12个元素的数组的有效索引数是0-11。要使用数组索引数来检索其中的元素，用objectAtIndex:方法。

程序仅仅使用objectAtIndex:方法简单地执行了一个for循环来从数组中提取每个元素。每个检索到的元素都使用NSLog进行显示。

代码清单15-7生成了一个素数表。因为要在生成素数时，把它们添加到数组中，所以需要一个可变数组。使用arrayWithCapacity:方法分配NSMutableArray的素数。你给出的参数20指定了数组的初始化大小；在程序运行时，可变数组的容量会根据需要自动增长。

即使素数是整数，也不能直接在数组中存储int值。你的数组只能容纳对象。因此，需要在primes数组中存储NSNumber整数对象。

代码清单15-7

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSValue.h>

#define MAXPRIME 50
int main (int argc, char *argv[])
{
   int    i, p, prevPrime;
   BOOL    isPrime;
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   // Create an array to store the prime numbers

   NSMutableArray *primes =
      [NSMutableArray arrayWithCapacity: 20];

   // Store the first two primes (2 and 3) into the array

   [primes addObject: [NSNumber numberWithInteger: 2]]
   [primes addObject: [NSNumber numberWithInteger: 3]];

   // Calculate the remaining primes

   for (p = 5; p <= MAXPRIME; p+= 2) {
      // we're testing to see if p is prime

      isPrime = YES;

      i = 1;

      do {
         prevPrime = [[primes objectAtIndex: i] integerValue];

         if (p % prevPrime == 0)
            isPrime = NO;

         ++i;
      } while ( isPrime == YES && p / prevPrime >= prevPrime);

      if (isPrime)
         [primes addObject: [NSNumber numberWithInteger: p]];
```

```
    }

    // Display the results

    for (i = 0; i < [primes count]; ++i)
        NSLog (@"%li", (long) [[primes objectAtIndex: i] integerValue]);

    [pool drain];
    return 0;
}
```

代码清单15-7 输出

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

将MAXPRIME定义为希望程序计算的最大素数，在这个例子中是50。

在分配primes数组之后，使用如下语句设置数组开始的两个元素：

```
[primes addObject: [NSNumber numberWithInteger: 2]];
[primes addObject: [NSNumber numberWithInteger: 3]];
```

addObject:方法向数组的末尾添加了一个对象。下面分别添加由整数2和3所创建的NSNumber对象。

然后，该程序进入一个for循环，来查找以5开始，直到MAXPRIME为止的素数，并且跳过(p += 2)之间的的偶数。

对每个可能的素数p，你想要了解它能否被前面找到的素数整除。如果能整除，则它不是素数。作为额外的优化，仅使用前面的素数，直到该数的平方根来测试这个数。这是因为，如果一个数不是素数，则它一定能够被小于或等于其平方根（哈，又回到了高中数学）的素数整除。所以，只要prevPrime小于或等于p的平方根，表达式

```
p / prevPrime >= prevPrime
```

总是为真。

如果do-while循环退出，而标志isPrime仍然等于YES时，你就发现了另一个素数。在这种情况下，将p加到primes数组，并且继续执行程序。

这里正好有一个关于程序效率的评论。Foundation类为使用数组提供了许多便利。然而，当使用复杂的运算法则操纵大型数字数组时，学习如何用该语言提供的低级数组构造来执行这种任务可能更加有效，对于内存使用和执行速度来说，都是如此。参考第13章中标题为“数组”的小节，以获取更多信息。

制作地址簿

看一个例子，它结合目前为止学到的知识，生成地址簿。你的地址簿将包含地址卡。为简单起见，地址卡中仅包含某人的姓名和email地址[⊖]。将这个概念扩展到其他信息，如地址和电话号码很简单，但是这作为本章末尾的练习题留给你来完成。

生成一个地址卡片

你打算从定义一个名为AddressCard的新类来开始。你需要以下的功能：创建一个新的地址卡片、设置卡片的姓名字段和email字段、检索这些字段的内容，并打印卡片。在图形化的环境下，可以使用一些友好的例程（比如Application Kit框架所提供的例程），在屏幕上绘制卡片。但在这里，继续使用简单的终端界面来显示地址卡片。

代码清单15-8显示了新的AddressCard类的接口文件。这里不会介绍访问器方法，你可以自己编写并从中学习更多的知识。

代码清单15-8　接口文件AddressCard.h

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>

@interface AddressCard: NSObject
{
  NSString *name;
  NSString *email;
}

-(void) setName: (NSString *) theName;
-(void) setEmail: (NSString *) theEmail;

-(NSString *) name;
-(NSString *) email;

-(void) print;

@end
```

这与代码清单15-8中的实现文件一样简单直观。

⊖ Mac OSX提供了完整的地址簿框架，该框架提供了极强的处理地址簿的功能。

代码清单15-8 实现文件AddressCard.m

```
#import"AddressCard.h "

@implementation AddressCard

-(void) setName: (NSString *) theName
{
   name = [[NSString alloc] initWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
   email = [[NSString alloc] initWithString: theEmail];
}

-(NSString *) name
{
   return name;
}

-(NSString *) email
{
   return email;
}

-(void) print
{
    NSLog (@ "========================================== ");
    NSLog (@ "|                                        | ");
    NSLog (@ "| %-31s | ", [name UTF8String]);
    NSLog (@ "| %-31s | ", [email UTF8String]);
    NSLog (@ "|                                        | ");
    NSLog (@ "|                                        | ");
    NSLog (@ "|                                        | ");
    NSLog (@ "|       O                    O           | ");
    NSLog (@ "========================================== ");
}
@end
```

可以使用如下方法定义，使setName和setEmail方法直接将这些对象存储在各自的实例变量中：

```
-(void) setName: (NSString *) theName
{
   name = theName;
}
```

```
-(void) setEmail: (NSString *) theEmail
{
   email = theEmail;
}
```

但是，AddressCard对象并不拥有它自己的成员对象。在第8章“继承”中，通过拥有自己的origin对象的Rectangle类，讨论了对象要获得所有权的动机。

用以下方式定义的两个方法：

```
-(void) setName: (NSString *) theName
{
   name = [NSString stringWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
   email = [NSString stringWithString: theEmail];
}
```

仍然是不正确的途径，因为AddressCard方法仍然没有获得它们的姓名和email对象——而NSString将拥有这些对象。

回到代码清单15-8，print方法尝试用一种类似于Rolodex卡片（还记得它们吗？）的格式向用户显示出具有良好展示效果的地址卡片。NSLog中的“%-31s”字符表明要用31个字符的字段宽度且左对齐的方式打印UTF8 C-字符串，这确保输出时地址卡片的右边缘是整齐的。

使用AddressCard类，可以编写一个测试程序来创建一个地址卡片、设置卡片的值以及显示卡片（参见代码清单15-8）。

代码清单15-8　测试程序

```
#import"AddressCard.h "
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString   *aName = @ "Julia Kochan ';
   NSString   *aEmail = @ "jewls337@axlc.com ';
   AddressCard   *card1 = [[AddressCard alloc] init];

   [card1 setName: aName];
   [card1 setEmail: aEmail];

   [card1 print];

   [card1 release];
   [pool drain];
   return 0;
}
```

代码清单15-8 输出

```
====================================
|                                  |
| Julia Kochan                     |
| jewls337@axlc.com                |
|                                  |
|                                  |
|                                  |
|      O              O            |
====================================
```

代码清单15-8中使用的程序行

```
[card1 release];
```

用于释放地址卡片占用的内存。从先前的讨论中，你应该认识到用这种方法释放AddressCard对象的同时并没有释放分配给它的name和email成员的内存。为了使AddressCard无漏洞，需要重载名为dealloc的方法，这样，无论何时释放AddressCard对象的内存，其成员的内存都会一并释放。

下面是为AddressCard类的dealloc方法：

```
-(void) dealloc
{
   [name release];
   [email release];
   [super dealloc];
}
```

在用super销毁对象本身之前，dealloc方法必须先释放自己的实例变量。这是因为释放对象之后，它就不再有效了。

要使AddressCard无漏洞，还必须修改setName:和setEmail:方法以释放存储在相应实例变量中的对象所占用的内存。如果某人更改了卡片上的姓名，就需要在使用新姓名代替旧姓名之前，释放旧姓名所占用的内存。email地址与此类似，在使用新地址代替旧地址之前，也必须释放旧email地址占用的内存。

下面是新的setName:、setEmail:方法，它们将确保拥有可以正确地进行内存管理的类：

```
-(void) setName: (NSString *) theName
{
   [name release];
   name = [[NSString alloc] initWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
   [email release];
   email = [[NSString alloc] initWithString: theEmail];
}
```

你可以给空对象发送消息，因此，即使没有预先设置name和email，消息表达式

```
[name release];
```

和

```
[email release];
```

都是正确的。

15.4 同步AddressCard方法

我们已经讨论了编写访问器方法setName:和setEmail:，你也已经理解了那些重要原理，那么可以返回操作并使系统生成访问器方法。考虑AddressCard界面文件的第二个版本：

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>

@interface AddressCard: NSObject
{
   NSString    *name;
   NSString    *email;
}

@property (copy, nonatomic) NSString *name, *email;
-(void) print;
@end
```

程序行

```
@property (copy, nonatomic) NSString *name, *email;
```

列出属性的copy属性和nonatomic属性。如你所编写的版本一样，copy属性将在setter方法内生成实例变量的副本。默认行为不会生成副本，而是仅仅执行分配（为默认assign属性），这是我们当前讨论的一个不正确方法。

nonatomic属性指明在返回值之前，getter方法不会保留或自动释放实例变量。本书第18章将详细介绍这个议题。

代码清单15-9是新的AddressCard实现文件，该文件指明访问器方法将被同步。

代码清单15-9　带同步方法的实现文件AddressCard.m

```
#import "AddressCard.h"

@implementation AddressCard

@synthesize name, email;
-(void) print
{
    NSLog (@"+-------------------------------------+");
    NSLog (@"|                                     |");
    NSLog (@"|  %-31s |", [name UTF8String]);
```

```
    NSLog (@"|  %-31s |", [email UTF8String]);
    NSLog (@"|                                    |");
    NSLog (@"|                                    |");
    NSLog (@"|                                    |");
    NSLog (@"|       O                 O          |");
    NSLog (@"======================================");

}
@end
```

这里为你准备了一个练习，即验证新的AddressCard定义与其同步访问器方法处理代码清单15-9中测试程序的情况。

现在，为AddressCard类添加另一个方法。你可能想使用一个调用同时设置卡片的姓名和email这两个字段，为此，可添加一个新方法：setName:andEmail:。[⊖]下面就是这个新地方法：

```
-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
   self.name = theName;
   self.email = theEmail;
}
```

通过使用同步setter方法设置适当的实例变量（而不是直接在方法中设置它们），会增加一定的抽象性，因此使程序更进一步地独立于它的内部数据结构。你也可以利用同步方法的属性，在本例中，是为实例变量复制而不是分配值。

代码清单15-9测试了新方法。

代码清单15-9 测试程序

```
#import <Foundation/Foundation.h>
#import"AddressCard.h "

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString  *aName = @ "Julia Kochan ";
   NSString  *aEmail = @ 'jewls337@axlc.com ';
   NSString  *bName = @ "Tony Iannino ';
   NSString  *bEmail = @ "tony.iannino@techfitness.com ";

   AddressCard  *card1 = [[AddressCard alloc] init];
   AddressCard  *card2 = [[AddressCard alloc] init];

   [card1 setName: aName andEmail: aEmail];
   [card2 setName: bName andEmail: bEmail];
```

⊖ 你可能还想有一个名为initWithName:andEmail:的初始化方法，但在这里不显示这个方法。

```
    [card1 print];
    [card2 print];
    [card1 release];
    [card2 release];
    [pool drain];
    return 0;
}
```

代码清单15-9 输出

```
=====================================
|                                   |
| Julia Kochan                      |
| jewls337@axlc.com                 |
|                                   |
|                                   |
|                                   |
|       O           O               |
=====================================

=====================================
|                                   |
| Tony Iannino                      |
| tony.iannino@techfitness.com      |
|                                   |
|                                   |
|                                   |
|       O           O               |
=====================================
```

你的AddressCard类看起来工作良好，但如果要使用很多AddressCard，该怎么办呢？把它们集中到一起很合理，通过定义一个名为AddressBook的新类就可以实现这项任务。AddressBook类存储地址簿的名字和一个AddressCard集合，你将这个集合存储在一个数组对象中。首先，需要以下功能：创建新的地址簿，向其添加地址卡片，计算地址簿的记录数，列出地址簿的内容；随后，可能需要更多的功能，如：搜索地址簿，删除记录，可能编辑现有记录，将记录排序，甚至拷贝记录内容。

首先看一个简单的interface文件（参见代码清单15-10）。

代码清单15-10 Addressbook.h 接口文件

```
#import <Foundation/NSArray.h>
#import"AddressCard.h "

@interface AddressBook: NSObject
{
    NSString *bookName;
    NSMutableArray *book;
```

```
}

-(id) initWithName: (NSString *) name;
-(void) addCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;
-(void) dealloc;

@end
```

initWithName:方法设置了初始数组来存放地址卡片和簿的名称，而addCard:方法向簿添加AddressCard。entries方法报告簿中地址卡片的数量，而list方法给出簿中全部内容的简明列表。AddressBook类的实现文件参见代码清单15-10。

代码清单15-10 Addressbook.m 实现文件

```
#import"AddressBook.h"

@implementation AddressBook;

// set up the AddressBook's name and an empty book

-(id) initWithName: (NSString *) name
{
   self = [super init];

   if (self) {
      bookName = [[NSString alloc] initWithString: name];
      book = [[NSMutableArray alloc] init];
   }

   return self;
}

-(void) addCard: (AddressCard *) theCard
{
   [book addObject: theCard];
}

-(int) entries
{
    return [book count];
}

-(void) list
{
   NSLog (@"======== Contents of: %@ =========", bookName);
```

```
    for ( AddressCard *theCard in book )
        NSLog (@"%-20s %-32s", [theCard.name UTF8String],
                    [theCard.email UTF8String]);
    NSLog (@"==========================================");
}

-(void) dealloc
{
    [bookName release];
    [book release];
    [super dealloc];
}
@end
```

initWithName:方法首先调用超类的init执行初始化；然后创建一个字符串对象（使用alloc函数，这样它就拥有这个对象），并通过作为name传递，将地址簿名称设置为这个字符串；随后分配并初始化一个可变的空数组，用以存储实例变量book。

定义的initWithName:方法返回一个id对象而不是AddressBook对象。如果创建AddressBook的子类，那么initWithName:的参数不是AddressBook对象，它的类型是子类对象。因此，需将返回类型定义为一般对象类型。

还要注意在initWithName:方法中，通过使用alloc函数，可获得bookName和book实例变量的所有权。比如，使用NSMutableArray的array方法为book创建数组，见

```
book = [NSMutableArray array];
```

但你还不是book数组的拥有者，NSMutableArray拥有它，因此释放AddressBook对象的内存时，你无权释放它的内存。

addCard:方法获取作为参数提供给它的AddressCard对象，并将其添加到地址簿中。

count方法返回数组中的元素个数。entries方法使用这个方法返回存储在地址簿中的地址卡片数目。

15.4.1 快速枚举

list方法的for循环展示一个你以前从来没有见过的结构：

```
for ( AddressCard *theCard in book )
      NSLog (@"%-20s    %-32s", [theCard.name UTF8String],
                  [theCard.email UTF8String]);
```

这里对book数组中的每个元素序列使用了一个被称为快速枚举的技术。它的语法非常简单：先定义一个能够依次保留数组中每个元素的变量（AddressCard *theCard）。使用关键字in跟随，然后列出数组的名称。当for循环执行时，它会为指定的变量分配数组的第一个元素并执行循环体。然后它会再为变量分配数组的第二个元素并执行循环体。这会一直持续，直到数组的所有元素都被分配给变量并且每一次都执行了循环体。

注意，如果theCard之前已经被定义为AddressCard对象，那么for循环将变得更加简单，如下：

```
for ( theCard in book )
    ...
```

代码清单15-10是新AddressBook类的测试程序。

代码清单15-10 测试程序

```
#import "AddressBook.h "
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString  *aName = @ "Julia Kochan";
   NSString  *aEmail = @ "jewls337@axlc.com";
   NSString  *bName = @ "Tony Iannino";
   NSString  *bEmail = @ "tony.iannino@techfitness.com';
   NSString  *cName = @ "Stephen Kochan";
   NSString  *cEmail = @ "steve@kochan-wood.com";
   NSString  *dName = @ "Jamie Baker";
   NSString  *dEmail = @ "jbaker@kochan-wood.com";

   AddressCard *card1 = [[AddressCard alloc] init];
   AddressCard *card2 = [[AddressCard alloc] init];
   AddressCard *card3 = [[AddressCard alloc] init];
   AddressCard *card4 = [[AddressCard alloc] init];

   AddressBook *myBook = [AddressBook alloc];

   // First set up four address cards

   [card1 setName: aName andEmail: aEmail];
   [card2 setName: bName andEmail: bEmail];
   [card3 setName: cName andEmail: cEmail];
   [card4 setName: dName andEmail: dEmail];

   // Now initialize the address book

   myBook = [myBook initWithName: @"Linda's Address Book"];

   NSLog (@ "Entries in address book after creation: %i",
              [myBook entries]);

   // Add some cards to the address book

   [myBook addCard: card1];
   [myBook addCard: card2];
```

```
    [myBook addCard: card3];
    [myBook addCard: card4];

    NSLog (@ "Entries in address book after adding cards: %i ",
              [myBook entries]);

    // List all the entries in the book now

    [myBook list];

    [card1 release];
    [card2 release];
    [card3 release];
    [card4 release];
    [myBook release];
    [pool drain];
    return 0;
}
```

代码清单15-10 输出

```
Entries in address book after creation: 0
Entries in address book after adding cards: 4

======== Contents of: Linda's Address Book =========
Julia Kochan          jewls337@axlc.com
Tony Iannino          tony.iannino@techfitness.com
Stephen Kochan        steve@kochan-wood.com
Jamie Baker           jbaker@kochan-wood.com
=====================================================
```

这个程序设置了4个地址卡片，然后创建一个名为Linda's Address Book的新地址簿。随后使用addCard:方法在地址簿中添加4张卡片，list方法用于列出地址簿的内容并校验其内容。

在地址簿中查询某人

你有一本容量较大的地址簿，每次查询某人时不想列出簿中的所有内容，因此增加一个方法很有意义。我们将这种方法叫做lookup:，并将要查找的姓名作为参数。这个方法搜索整个地址簿来寻找匹配（忽略大小写），如果匹配成功，则返回该记录。如果电话簿中不存在所给的姓名，使其返回nil。

下面是新的lookup:方法。

```
// lookup address card by name - assumes an exact match

-(AddressCard *) lookup: (NSString *) theName
{
    for ( AddressCard *nextCard in book )
       if ( [[nextCard name] caseInsensitiveCompare: theName] == NSOrderedSame )
```

```
            return nextCard;

    return nil;
}
```

如果将该方法声明放在接口文件中，而把定义放在实现文件中，则可以编写一个测试程序来试验这个新方法。代码清单15-11显示了这个程序，输出结果紧随其后。

代码清单15-11 测试程序

```
#import"AddressBook.h "
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{

   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString  *aName = @ "Julia Kochan";
   NSString  *aEmail = @ "jewls337@axlc.com";
   NSString  *bName = @ "Tony Iannino";
   NSString  *bEmail = @ "tony.iannino@techfitness.com";
   NSString  *cName = @ "Stephen Kochan";
   NSString  *cEmail = @ "steve@kochan-wood.com";
   NSString  *dName = @ "Jamie Baker";
   NSString  *dEmail = @ "jbaker@kochan-wood.com";
   AddressCard  *card1 = [[AddressCard alloc] init];
   AddressCard  *card2 = [[AddressCard alloc] init];
   AddressCard  *card3 = [[AddressCard alloc] init];
   AddressCard  *card4 = [[AddressCard alloc] init];

   AddressBook  *myBook = [AddressBook alloc];
   AddressCard  *myCard;

   // First set up four address cards

   [card1 setName: aName andEmail: aEmail];
   [card2 setName: bName andEmail: bEmail];
   [card3 setName: cName andEmail: cEmail];
   [card4 setName: dName andEmail: dEmail];

   myBook = [myBook initWithName: @ "Linda's Address Book"];

   // Add some cards to the address book

   [myBook addCard: card1];
   [myBook addCard: card2];
```

```
    [myBook addCard: card3];
    [myBook addCard: card4];

    // Look up a person by name

    NSLog (@ "Look up: Stephen Kochan ");
    myCard = [myBook lookup: @ "stephen kochan "];

    if (myCard != nil)
       [myCard print];
    else
       NSLog (@ "Not found! ");

    // Try another lookup

    NSLog (@ "Lookup:Haibo Zhang ");
    myCard = [myBook lookup: @ "Haibo Zhang "];

    if (myCard != nil)
       [myCard print];
    else
       NSLog (@ 'Not found! ");

    [card1 release];
    [card2 release];
    [card3 release];
    [card4 release];
    [myBook release];

    [pool drain];
    return 0;
}
```

代码清单15-11 输出

```
Lookup: Stephen Kochan
=====================================
|                                   |
| Stephen Kochan                    |
| steve@kochan-wood.com             |
|                                   |
|                                   |
|                                   |
|       O           O               |
=====================================

Lookup: Haibo Zhang
Not found!
```

通过lookup:方法在地址簿中找到Stephen Kochan（注意我们利用匹配时不区分大小写的事实），使用AddressCard的print方法显示所得的结果。第二次查询时，没有找到姓名Haibo Zhang，因此返回了上面的结果信息。

这个lookup消息非常简单，因为它必须找到整个name的精确匹配。更好的方法可以完成部分匹配，也可以处理多重匹配。比如记录Steve Kochan、Fred Stevens和steven levy都可以满足消息表达式

```
[myBook lookup: @"steve"]
```

的匹配条件。因为可能存在多重匹配，所以有效的方法可能是创建一个包含所有匹配的数组，并将该数组返回给方法的调用者（参见本章最后的练习2），如下：

```
matches = [myBook lookup: @"steve"];
```

从地址簿中删除某人

具有添加记录功能的地址簿管理程序如果没有删除记录的功能，那么它是不完整的。可以构造一个removeCard:方法从地址簿中删除特定的AddressCard。或创建一个remove:方法，它根据名字删除记录（见本章最后的练习6）。

因为已经对接口文件中做了几次改动，所以代码清单15-12再次显示包含新的removeCard:方法的接口文件。之后是新的removeCard:方法。

代码清单15-12 Addressbook.h接口文件

```
#import <Foundation/NSArray.h>
#import"AddressCard.h "

@interface AddressBook: NSObject
{
   NSString       *bookName;
   NSMutableArray *book;
}

-(id) initWithName: (NSString *) name;

-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;

-(AddressCard *) lookup: (NSString *) theName;
-(int) entries;
-(void) list;

@end
```

下面是新的removeCard:方法：

```
-(void) removeCard: (AddressCard *) theCard
{
  [book removeObjectIdenticalTo: theCard];
}
```

关于什么是同一对象，我们的观点是占用同样的内存位置。所以，两个包含相同信息的地址卡片对象处于不同的内存单元时（比如，复制地址卡片对象时，就会出现这种情况），removeObjectIdenticalTo:方法并不把它们视为同一对象。

顺便提及，removeObjectIdenticalTo:方法用于删除和其参数相同的所有对象。但只有对象数组中多次出现一个对象时才会用到这个方法。

通过使用removeObject:方法，然后编写isEqual:方法用于测试两个对象的相等性可以使方法更完善。如果使用removeObject:方法，则系统会自动为数组中的每个元素调用isEqual:方法，同时提供要比较的两个元素。在这个例子中，因为你的地址簿包含AddressCard对象作为其成员，所以必须将isEqual:方法添加到该类中（应该重载从NSObject继承的方法）。该方法可以自己决定如何确定等同性。将相应的name字段和email字段比较，如果都相等，则可以从方法返回YES，否则，返回NO。你的方法可以这样写：

```
(BOOL) isEqual (AddressCard *) theCard
{
   if ([name isEqualToString: theCard.name] == YES &&
         [email isEqualToString: theCard.email] == YES)
      return YES;
   else
      return NO;
}
```

下面应该注意NSArray方法，如：containsObject:方法和indexOfObject:方法，都依赖isEqual:策略来决定两个对象是否相等。

代码清单15-12测试了新的removeCard:方法。

代码清单15-12　测试程序

```
#import"AddressBook.h "
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString  *aName = @ "Julia Kochan ";
   NSString  *aEmail = @ "jewls337@axlc.com ";
   NSString  *bName = @ "Tony Iannino ";
   NSString  *bEmail = @ "tony.iannino@techfitness.com ";
   NSString  *cName = @ "Stephen Kochan ";
   NSString  *cEmail = @ "steve@kochan-wood.com ";
   NSString  *dName = @ "Jamie Baker ";
   NSString  *dEmail = @ "jbaker@kochan-wood.com ";

   AddressCard *card1 = [[AddressCard alloc] init];
   AddressCard *card2 = [[AddressCard alloc] init];
   AddressCard *card3 = [[AddressCard alloc] init];
```

```
    AddressCard *card4 = [[AddressCard alloc] init];

    AddressBook *myBook = [AddressBook alloc];
    AddressCard *myCard;

    // First set up four address cards

    [card1 setName: aName andEmail: aEmail];
    [card2 setName: bName andEmail: bEmail];
    [card3 setName: cName andEmail: cEmail];
    [card4 setName: dName andEmail: dEmail];

    myBook = [myBook initWithName: @ 'Linda's Address Book "];

    // Add some cards to the address book

    [myBook addCard: card1];
    [myBook addCard: card2];
    [myBook addCard: card3];
    [myBook addCard: card4];

    // Look up a person by name

    NSLog (@ 'Lookup: Stephen Kochan ");
    myCard = [myBook lookup: @ "Stephen Kochan '];

    if (myCard != nil)
       [myCard print];
    else
       NSLog (@ "Not found! ");

    // Now remove the entry from the phone book

    [myBook removeCard: myCard];
    [myBook list]; // verify it's gone

    [card1 release];
    [card2 release];
    [card3 release];
    [card4 release];
    [myBook release];
    [pool drain];

return 0;
}
```

代码清单15-12　输出

```
Lookup: Stephen Kochan
=================================
:                                 |
| Stephen Kochan
| steve@kochan-wood.com           |
|                                 |
|                                 |
|                                 |
|       O           O             |
=================================

======== Contents of: Linda's Address Book =========
Julia Kochan        jewls337@axlc.com
Tony Iannino        tony.iannino@techfitness.com
Jamie Baker         jbaker@kochan-wood.com
================================================
```

在地址簿中查询Stephen Kochan并检测出它在簿中后，将所得结果AddressCard传给新removeCard:方法来删除它。结果地址簿列表验证了删除成功。

15.4.2　数组排序

如果地址簿包含大量的记录，那么按字母将其排序可能很方便。通过在AddressBook类中增加sort方法并利用NSMutableArray类中名为 sortUsingSelector：的方法，可以很容易地实现这项功能。sort方法使用一个selector作为其参数，sortUsingSelector：方法使用这个selector来比较两个元素。数组可包含任何类型的对象，所以实现一般排序方法的唯一途径就是先判定数组中的元素是否有序。为此，必须添加一个方法用于比较数组中的两个元素[⊖]。这个方法返回的结果是NSComparisonResult类型的值。如果希望排序方法使排序后原数组中的第一条记录位于第二条之前，那么，方法的返回值应是NSOrderedAscending；如果这两条记录相等，那么返回NSOrderedSame；如果排序后的原数组中的第一条记录放在第二条之后，那么返回值NSOrderedDescending。

首先，下面是AddressBook类中的新sort方法：

```
-(void) sort
{
   [book sortUsingSelector: @selector(compareNames:)];
}
```

和在第9章“多态动态类型和动态绑定”中学到的一样，表达式

```
@selector(compareNames:)
```

创建一个SEL类型的selector，它来自一个指定的方法名，sortUsingSelector:使用该方法比较数

⊖ 还有一个名为sortUsingFunction:context:的方法，它允许你使用函数代替方法来实现比较功能。

组中的两个元素。如果sortUsingSelector:方法需要完成这样的比较，它先调用这个指定的selector方法，然后向数组（接收者）的第一条记录发送消息，比较其参数和此记录。前面描述过，返回值应该为NSComparisonResult类型。

因为地址簿的元素是AddressCard对象，所以还必须向AddressCard类添加Comparision方法。因此，应该回到AddressCard类，并为其添加compareNames:方法。下面给出具体实现：

```
// Compare the two names from the specified address cards
-(NSComparisonResult) compareNames: (id) element
{
    return [name compare: [element name]];
}
```

因为执行的是地址簿中两个名字字符串的比较，所以可以用NSString类的compare:方法来实现此功能。

如果向AddressBook类添加了sort方法，并且向AddressCard类添加了compareNames:方法，那么可以编写一个程序来测试一下（参见代码清单15-13）。

代码清单15-13 测试程序

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{

   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString  *aName = @ "Julia Kochan";
   NSString  *aEmail = @ "jewls337@axlc.com";
   NSString  *bName = @ "Tony Iannino";
   NSString  *bEmail = @ "tony.iannino@techfitness.com";
   NSString  *cName = @ "Stephen Kochan ";
   NSString  *cEmail = @ "steve@kochan-wood.com";
   NSString  *dName = @ "Jamie Baker";
   NSString  *dEmail = @ "jbaker@kochan-wood.com";

   AddressCard *card1 = [[AddressCard alloc] init];
   AddressCard *card2 = [[AddressCard alloc] init];
   AddressCard *card3 = [[AddressCard alloc] init];
   AddressCard *card4 = [[AddressCard alloc] init];

   AddressBook *myBook = [AddressBook alloc];

   // First set up four address cards

   [card1 setName: aName andEmail: aEmail];
   [card2 setName: bName andEmail: bEmail];
```

```
    [card3 setName: cName andEmail: cEmail];
    [card4 setName: dName andEmail: dEmail];

    myBook = [myBook initWithName: @ "Linda's Address Book "];

    // Add some cards to the address book

    [myBook addCard: card1];
    [myBook addCard: card2];
    [myBook addCard: card3];
    [myBook addCard: card4];

    // List the unsorted book

    [myBook list];

    // Sort it and list it again
    [myBook sort];
    [myBook list];

    [card1 release];
    [card2 release];
    [card3 release];
    [card4 release];
    [myBook release];
    [pool drain];
    return 0;
}
```

代码清单15-13 输出

```
======== Contents of: Linda's Address Book =========
Julia Kochan        jewls337@axlc.com
Tony Iannino        tony.iannino@techfitness.com
Stephen Kochan      steve@kochan-wood.com
Jamie Baker         jbaker@kochan-wood.com
=====================================================

======== Contents of: Linda's Address Book =========
Jamie Baker         jbaker@kochan-wood.com
Julia Kochan        jewls337@axlc.com
Stephen Kochan      steve@kochan-wood.com
Tony Iannino        tony.iannino@techfitness.com
=====================================================
```

应能注意到排列是升序的，但是，可以很容易地修改AddressCard类中的compareNames:方法，使返回的值呈降序排列。

现在，处理数组对象有50多个方法，表15-4和表15-5分别列出了不变数组和可变数组的常用方法。因为NSMutableArray类是NSArray类的子类，所以前者继承了后者的方法。

表15-4和表15-5中的obj、obj1和obj2是任意对象，i是呈现数组中有效索引号的NSUInteger整数，selector是SEL类型的selector对象，size是一个NSUInteger整数。

表15-4 常用的NSArray方法

方 法	说 明
+(id) arrayWithObjects: obj1, obj2, ... nil	创建一个新数组，obj1、obj2，…是其对象
-(BOOL) containsObject: obj	确定数组中是否包含对象obj（使用isEqual方法）
-(NSUInteger) count	数组中元素的个数
-(NSUInteger) indexOfObject: obj	第一个包含对象obj的元素索引号（使用isEqual方法）
-(id) objectAtIndex: i	存储在元素i的对象
-(void)makeObjectsPerformSelector: (SEL) selector	将selector指示的消息发送给数组中的每个元素
-(NSArray*) sortedArrayUsingSelector: (SEL) selector	根据selector指定的比较方法对数组进行排序
-(BOOL) writeToFile: path atomically: (BOOL) flag	将数组写入指定的文件中，如果flag为YES，那么先创建一个临时文件

表15-5 常用的NSMutableArray方法

方 法	说 明
+(id) array	创建一个空数组
+(id) arrayWithCapacity: size	使用指定的初始size创建一个数组
-(id) initWithCapacity: size	使用指定的初始size初始化新分配的数组
-(void) addObject: obj	将对象obj添加到数组的末尾
-(void) insertObject: obj atIndex: i	将对象obj插入数组的i元素
-(void) replaceObjectAtIndex: i withObject: obj	将数组中序号为i的对象用对象obj替换
-(void) removeObject: obj	从数组中删除所有obj
-(void) removeObjectAtIndex: i	从数据中删除元素i，将序号为i+1的对象移至数组的结尾
-(void) sortUsingSelector: (SEL) selector	用selector指示的比较方法将组排序

15.5 词典对象

dictionary是由键-对象对组成的数据集合。正如在词典中查找单词的定义一样，可通过对象的键从Objective-C词典中获取所需的值（即那个对象）。词典中的键必须是单值的，尽管它们通常是字符串，但还可以是任何对象类型。和键关联的值可以是任何对象类型，但它们不能为nil。

词典可以是固定的，也可以是可变的。可变词典中记录可动态的添加和删除。可基于特定的键对词典进行搜索，也可以枚举它们的内容。代码清单15-14创建了一个可变词典，它用作Objective-C的术语表，前三条记录已添加到词典中。

要在程序中使用词典类，需将下面这行代码添加到程序中：

```
#import <Foundation/NSDictionary.h>
```

代码清单15-14

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSMutableDictionary *glossary = [NSMutableDictionary dictionary];

   // Store three entries in the glossary

   [glossary setObject: @ "A class defined so other classes can inherit from it"
               forKey: @ "abstract class"];
   [glossary setObject: @ "To implement all the methods defined in a protocol"
               forKey: @ "adopt"];
   [glossary setObject: @ "Storing an object for later use "
               forKey: @ "archiving"];

   // Retrieve and display them

   NSLog (@ "abstract class: %@", [glossary objectForKey: @ "abstract class"]);
   NSLog (@"adopt: %@", [glossary objectForKey: @"adopt"]);
   NSLog (@"archiving: %@", [glossary objectForKey: @"archiving"]);

   [pool drain];
   return 0;
}
```

代码清单15-14 输出

```
abstract class: A class defined so other classes can inherit from it
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use
```

表达式

```
[NSMutableDictionary dictionary]
```

创建了一个空的可变词典。随后使用setObject:forKey:方法将键-值对添加到词典中。生成词典之后，可以使用objectForKey:方法检索给定键的值。代码清单15-14显示了如何检索和显示Objective-C术语表中的三条记录。在更为实际的应用程序中，用户键入他要定义的单词，程序会搜索该术语表以寻找其定义。

枚举词典

代码清单15-15阐述了如何使用dictionaryWithObjectsAndKeys:方法和带有初始键-值对的词典，这样就创建了一个不可变词典，该程序还显示如何使用快速枚举循环一次一个键地从词典检索各个元素。不像数组对象，词典对象是无序的。所以，枚举词典时，放到词典中的第一个键-对象对并不一定是第一个提取的。

代码清单15-15

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSDictionary *glossary =
    [NSDictionary dictionaryWithObjectsAndKeys:
     @"A class defined so other classes can inherit from it",
     @"abstract class",
     @"To implement all the methods defined in a protocol",
     @"adopt",
     @"Storing an object for later use",
     @"archiving",
     nil
   ];

   // Print all key-value pairs from the dictionary

      for ( NSString *key in glossary )
         NSLog ( @"%@%@ ", key, [glossary objectForKey: key]);

   [pool drain];
   return 0;
}
```

代码清单15-15 输出

```
abstract class: A class defined so other classes can inherit from it
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use
```

dictionaryWithObjectsAndKeys:的参数是object-key对的列表（是的，就是这种顺序），每个object-key用逗号隔开。列表必须以特定的nil对象结束。

程序创建词典后，它设置一组循环语句用于检索词典的内容。已经说过，键是从词典中依

次被检索的，而没有特定顺序。如果要以字母顺序显示词典中的内容，可以先检索词典中的所有键，将其排序，然后为所有已排序键检索对应的值。keysSortedByValueUsingSelector:方法可以帮你完成一半工作，它可以依据你的排序标准返回已排序的键。

我们已经显示了词典的一些基本操作。表15-6和表15-7分别总结出使用不变和可变词典的一些常用方法。因为NSMutableDictionary类是NSDictionary类的子类，所以它继承了NSDictionary类的方法。

表15-6和表15-7中的key、key1、key2、obj、obj1和obj2是任意对象，size是一个NSUInteger整数。

表15-6　常用的NSDictionary方法

方　法	说　明
+(id) dictionaryWithObjectsAndKeys: obj1, key1, obj2, key2, ..., nil	使用键-对象对{key1,obj1}、{key2,obj2}...创建词典
-(id) initWithObjectsAndKeys: obj1, key1, obj2, key2,..., nil	将新分配的词典初始化为键-对象对{key1,obj1}{key2,obj2}...
-(unsigned int) count	返回词典中的记录数
-(NSEnumerator *) keyEnumerator	为词典中所有键返回一个NSEnumerator对象
-(NSArray *) keysSortedByValueUsingSelector: (SEL) selector	返回词典中的键数组，它根据selector指定的比较方法进行了排序
-(NSEnuerator *) objectEnumerator	为词典中的所有值返回一个NSEnumerator对象
-(id) objectForKey: key	返回指定key的对象

表15-7　常用的NSMutableDictionary方法

方　法	说　明
+(id) dictionaryWithCapacity: size	使用一个初始指定的size创建可变词典
-(id) initWithCapacity: size	将新分配的词典初始化为指定的size
-(void) removeAllObjects	删除词典中所有的记录
-(void) removeObjectForKey: key	删除词典中指定key对应的记录
-(void) setObject: obj forKey: key	向词典为key键添加obj，如果key已存在，则替换该值

15.6　集合对象

set是一组单值对象的集合，并且它可以是可变的，也可以是不变的。操作包括：搜索、添加、删除集合中的成员（仅用于可变集合），比较两个集合，计算两个集合的交集和并集等。

要在程序中使用set类，需在程序中加入这一行：

```
#import <Foundation/NSSet.h>
```

代码清单15-16显示了集合的一些基本操作。假定你想在程序执行过程中几次显示集合的内容，因此决定创建一个名为print的新方法。通过创建一个名为Printing的新category，可以将print方法加入NSSet类中。NSMutableSet类是NSSet类的子类，所以可变集合也可以使用这个新print方法。

代码清单15-16

```
#import <Foundation/NSObject.h>
#import <Foundation/NSSet.h>
#import <Foundation/NSValue.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>

// Create an integer object
#define INTOBJ(v) [NSNumber numberWithInteger: v]

// Add a print method to NSSet with the Printing category

@interface NSSet (Printing)
-(void) print;
@end

@implementation NSSet (Printing)
-(void) print {
    printf ("{");

    for (NSNumber *element in self)
        printf ("%li", (long) [element integerValue]);

    printf ("}\n ");
}
@end

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSMutableSet *set1 = [NSMutableSet setWithObjects:
          INTOBJ(1), INTOBJ(3), INTOBJ(5), INTOBJ(10), nil];
    NSSet *set2 = [NSSet setWithObjects:
          INTOBJ(-5), INTOBJ(100), INTOBJ(3), INTOBJ(5), nil];
    NSSet *set3 = [NSSet setWithObjects:
          INTOBJ(12), INTOBJ(200), INTOBJ(3), nil];
    NSLog (@ "set1:");
     [set1 print];
    NSLog (@ "set2:");
     [set2 print];

    // Equality test
    if ([set1 isEqualToSet: set2] == YES)
       NSLog (@ "set1 equals set2 ");
    else
       NSLog (@ "set1 is not equal to set2 ");
```

```
    // Membership test

    if ([set1 containsObject: INTOBJ(10)] == YES)
       NSLog (@ "set1 contains 10 ");
    else
       NSLog (@ "set1 does not contain 10 ");

    if ([set2 containsObject: INTOBJ(10)] == YES)
       NSLog (@ "set2 contains 10 ");
    else
       NSLog (@ "set2 does not contain 10 ");

    // add and remove objects from mutable set set1

    [set1 addObject: INTOBJ(4)];
    [set1 removeObject: INTOBJ(10)];
    NSLog (@ "set1 after adding 4 and removing 10:");
    [set1 print];

    // get intersection of two sets

    [set1 intersectSet: set2];
    NSLog (@ "set1 intersect set2:");
    [set1 print];

    // union of two sets

    [set1 unionSet:set3];
    NSLog (@ "set1 union set3:");
    [set1 print];

    [pool drain];
    return 0;
}
```

代码清单15-16 输出

```
set1:
{ 3 10 1 5 }
set2:
{ 100 3 -5 5 }
set1 is not equal to set2
set1 contains 10
set2 does not contain 10
set1 after adding 4 and removing 10:
{ 3 1 5 4 }
set1 intersect set2:
{ 3 5 }
```

```
set1 union set3:
( 12 3 5 200 )
```

print方法使用了前面描述的快速枚举技术检索集合中的每个元素。还可以定义一个名为INTOBJ的宏，它根据整数值创建整型的对象，这会使得你的程序更加简洁，并且节省一些不必要的打字工作。当然，因为你的print方法只用于元素类型为整型的集合，所以它不太通用。但在这里它是一个很好的暗示，提醒我们如何通过创建category将方法添加到类中（注意：在print方法中，使用C库中的printf例程显示每个单线集合中的元素）。[⊖]

setWithObjects:根据一个以nil结尾的对象列表创建新集合。创建三个集合后，该程序使用新的print方法显示前两个集合的内容。然后用isEqualToSet:方法检测集合set1是否和set2相等，结果它们不等。

首先使用containsObject:方法查询整数10是否在集合set1中，然后查询是否在集合set2中。该方法返回的Boolean值证实了整数10在集合set1中而不是set2中。

然后，该程序使用addObject:方法和removeObject:方法分别从集合set1添加4和删除10，显示的集合内容证实了操作的成功。

intersect:和union:方法用于计算两个集合的交集和并集。在这两种情况下，运算的结果取代了消息的接收者。

Foundation框架同样提供了一个名为NSCountedSet的类。这些集合中同一对象可出现多次，然而，并非在集合中多次存放这个对象，而是维护一个次数计数。所以，第一次将对象添加到集合中时，它的count值被置为1，随后，每次将这个对象添加到集合中，count的值就会增1，而每次从集合中删除对象时，count就对应减1，如果count值为零时，实际的对象本身就被删除了。countForObject:方法用于在集合中检索某个对象的count值。

计数集合的一个应用程序是单词计数器。每次在文本中发现一个单词，就将它添加到计数集合中，扫描完文本之后，就可以从集合中检索每个单词及其计数，计数表明该词在文本中出现的次数。

这里只是显示了集合的一些基本操作。表15-8和表15-9分别总结了用于不变和可变集合的一些常用方法。因为NSMutableSet类是NSSet类的子类，所以它继承了NSSet类的方法。

在表15-8和表15-9中，obj、obj1和obj2是任意对象，nsset是NSMutableSet类或NSSet类的对象，size是一个NSUInteger整数。

表15-8 常用的NSSet方法

方 法	说 明
+(id) setWithObjects: obj1, obj2, ..., nil	使用一列对象创建新集合
-(id) initWithObjects: obj1, obj2, ..., nil	使用一列对象初始化新分配的集合
-(NSUInteger) count	返回集合的成员个数
-(BOOL) containsObject: obj	确定集合是否包含obj

⊖ 更通用的方法是调用每个对象的description方法来显示集合中的每个元素。这允许用清楚的格式显示其中包含任意类型对象的集合，还要注意，只需调用NSLog一次也可显示任意集合的内容，方法是用“打印对象”格式符“%@”。

（续）

方 法	说 明
-(BOOL) member: obj	使用isEqual:方法确定集合是否包含obj
-(NSEnumerator *) objectEnumerator	为集合中的所有对象返回一个NSEnumerator对象
-(BOOL) isSubsetOfSet: nsset	确定receiver的每个成员是否都出现在nsset中
-(BOOL) intersectsSet: nsset	确定是否receiver中至少一个成员出现在对象nsset中
-(BOOL) isEqualToSet: nsset	确定两个集合是否相等

表15-9 常用的NSMutableSet方法

方 法	说 明
-(id) setWithCapacity: size	创建新集合，使其具有存储size个成员的初始空间
-(id) initWithCapacity: size	将新分配的集合设置为size个成员的存储空间
-(void) addObject: obj	将对象obj添加到集合中
-(void) removeObject: obj	从集合中删除对象obj
-(void) removeAllObjects	删除接收者的所有成员
-(void) unionSet: nsset	将对象nsset的所有成员添加到接收者
-(void) minusSet: nsset	从接收者中删除nsset的所有成员
-(void) intersectSet: nsset	将接收者中所有不属于nsset的元素删除

15.7 练习

1. 在文档中查找NSCalendarDate类，然后向类添加一个名为ElapsedDays的新category。在这个新category中，根据以下方法声明添加一个方法：

```
-(unsigned long) numberOfElapsedDays: (NSCalendarDate *) theDate;
```

让新方法应返回接收者到该方法的参数之间经过的天数。并编写一个测试程序来测试新方法。（提示：可参见years:months:days:hours:minutes:seconds:sinceDate:方法。）

2. 修改本章为类AddressBook开发的lookup:方法，使之能够找出name的部分匹配。消息表达式

```
[myBook lookup: @"steve"]
```

应该匹配名称中任何位置包含字符串“steve”的记录。

3. 修改本章为类AddressBook开发的lookup:方法，使之能够搜索地址簿以找到所有匹配，返回值为包含所有匹配的地址卡片的数组，或者若匹配不成功时，则返回nil。
4. 在AddressCard类添加你选择的新字段，一些建议是将name字段分隔成姓氏和名字字段，并添加地址（可能包含单独的州、城市、邮编、国家字段）和电话号码字段。编写合适的setter和getter方法，并确保print方法和list方法能恰当地显示这些字段。
5. 完成练习3之后，修改练习2的lookup：方法，使之能够对地址卡片中的所有字段进行搜索。能否想出一种方式设计AddressCard类和AddressBook类，使得后者不必了解存储在前者中的所有字段。
6. 给定以下声明：

```
-(BOOL) removeName: (NSString *) theName;
```

在AddressBook类中添加RemoveName：方法，以便从地址簿中删除某条记录。

7. 使用在第一部分中定义的Fraction类，根据任意一些值创建一个分数数组，编写代码计算数组中所有分数的和。
8. 使用第一部分中定义的Fraction类，根据任意一些值创建一个可变的分数数组。使用类NSMutableArray的sortUsingSelector:方法给数组排序，向Fraction类中添加一个Comparison类型，并实现comparison方法。
9. 定义三个新类，分别名为Song、PlayList和MusicCollection。Song对象包含着关于特定歌曲的信息，比如歌曲名、艺术家、专辑、歌曲长度等；PlayList对象包含播放列表名称和一个歌曲的集合；MusicCollection对象包含播放列表集合，它包括一个名为Library的主播放列表，这个列表包含该集合中的所有歌曲。定义上述的三个类，并编写方法实现下列任务：
   - 创建一个Song对象，并设置其信息。
   - 创建一个PlayList对象，并对播放列表添加和删除歌曲。如果一首新歌不在主列表中，那么将其添加进去。确保从主播放列表中删除一首歌时，也要从音乐集合中的其他播放列表中删除此歌曲。
   - 创建一个MusicCollection对象，并对该集合添加和删除播放列表。
   - 搜索并显示关于所有歌曲、播放列表或整个音乐集合的信息。
   - 确保所有你定义的类都不产生内存漏洞。
10. 编写一个程序，它使用NSNumber对象的NSArray（其中每个NSNumber代表一个整数）并生成一个频率图表，列出每个整数和它在数组中的出现次数。

第16章 使 用 文 件

Foundation框架允许你利用文件系统对文件或目录执行基本操作。这些基本操作是由NSFileManager类提供的，这个类的方法具有如下功能：

- 创建一个新文件
- 从现有文件中读取数据
- 将数据写入文件中
- 重新命名文件
- 删除文件
- 测试文件是否存在
- 确定文件的大小及其他属性
- 复制文件
- 测试两个文件的内容是否相同

上述多数操作也可以直接对目录进行。例如，可以创建目录，读取其中的内容，或者删除目录。另一个重要特性是链接文件的能力，也就是，同一个文件可以以不同的名字存在，这两个文件甚至可以位于不同的目录中。

用NSFileHandle类提供的方法，可以打开文件并对其执行多次读写操作。NSFileHandle类的方法可以实现如下功能：

- 打开一个文件，执行读、写或更新（读写）操作
- 在文件中查找指定位置
- 从文件中读取特定数目的字节，或将特定数目的字节写入文件中

NSFileHandle类提供的方法也可以用于各种设备或套接字。然而，本章我们只讨论普通文件的处理。

16.1 管理文件和目录：NSFileManager

对于NSFileManager，文件或目录是使用文件的路径名唯一地标识的。每个路径名都是一个NSString对象，它既可以是相对路径名，也可以是完整路径名。相对路径名是相对于当前目录的路径名。所以，文件名copy1.m意味着当前目录中的文件copy1.m。斜线字符用于隔开路径中的目录列表。文件名ch16/copy1.m也是相对路径，它标识存储在目录ch16中的文件copy1.m，而Ch16包含在当前目录中。

完整路径名，也称为绝对路径名，以斜线/开始。斜线实际上就是一个目录，称为根目录。在我的Mac上，主目录的完整路径名为/Users/stevekochan。这个路径名指定了三个目录：/（根目录）、Users和stevekochan。

这个特殊的代字符（~）用作用户主目录的缩写。因此，~linda表示用户linda的主目录的

缩写，这个目录的路径可能是/Users/linda。单个代字符表示当前用户的主目录，这意味着路径名~/copy1.m将引用存储在当前用户主目录中的文件copy1.m。其他特殊的UNIX风格的路径名字符，如表示当前目录的“.”和表示父目录的“..”，应该在Foundation中文件处理方法使用路径之前，从路径名中删除。还可以使用一些路径实用工具，它们将在本章后面讨论。

在程序中，应该尽量避免使用硬编码的路径名。本章前面讲到过，可使用方法和函数来获取当前目录的路径名、用户的主目录以及可以用来创建临时文件的目录。应该尽可能地利用这些函数和方法。在本章后面会看到，Foundation有一个函数，它用于获取一列特殊的目录，如用户的Documents目录。

表16-1总结了一些基本的NSFileManager方法，这些方法用于处理文件。在表16-1中，path、path1、path2、from和to都是NSString对象；attr是一个NSDictionary对象；handler是一个回调处理程序，它允许你使用自己的方式来处理错误。如果对handler指定nil，就会采取默认的行为，即如果该操作成功，那么返回BOOL的方法就会返回YES，失败，就会返回NO。本章并不涉及编写自己的处理程序。

表16-1 常见的NSFileManager文件方法

方 法	描 述
-(BOOL) contentsAtPath: path	从一个文件中读取数据
-(BOOL) createFileAtPath: path contents: (BOOL) data attributes: attr	向一个文件写入数据
-(BOOL) removeFileAtPath: path handler: handler	删除一个文件
-(BOOL) movePath: from toPath: to handler: handler	重命名或者移动一个文件（to可能是已存在的）
-(BOOL) copyPath: from toPath: to handler: handler	复制文件（to不能是已存在的）
-(BOOL) contentsEqualAtPath: path1 andPath: path2	比较这两个文件的内容
-(BOOL) fileExistsAtPath: path	测试文件是否存在
-(BOOL) isReadableFileAtPath: path	测试文件是否存在，并且是否能执行读操作
-(BOOL) isWritableFileAtPath: path	测试文件是否存在，并且是否能执行写操作
-(NSDictionary *) fileAttributesAtPath: path traverseLink: (BOOL) flag	获取文件的属性
-(BOOL) changeFileAttributes: attr atPath: path	更改文件的属性

每个文件方法都是对NSFileManager对象的调用，而NSFileManager对象是通过向类发送一条defaultManager消息创建的，如下所示：

```
NSFileManager  *fm;
   ...
 fm = [NSFileManager defaultManager];
```

例如，要从当前目录删除名为todolist的文件，首先要创建一个NSFileManager对象（如前面所示），然后调用removeFileAtPath方法，代码如下：

```
[fm removeFileAtPath: @"todolist" handler: nil];
```

可以测试返回结果，以确保成功地删除该文件。

```
if ([fm removeFileAtPath: @"todolist" handler: nil] == NO) {
   NSLog (@"Couldn't remove file todolist");
   return 1;
}
```

除了其他事情之外，属性字典还允许你指定要创建的文件的权限，以便获取或者更改现有文件的信息。对于文件创建，如果将该参数指定为nil，将会为该文件设置默认权限。fileAttributesAtPath:traverseLink:方法返回一个包含指定文件属性的字典。对于符号链接(symbolic link)，traverseLink:参数的值为yes或no。如果该文件是一个符号链接，则指定yes，并且返回链接到的文件属性。如果指定no，则返回链接本身的属性。

对于现有的文件，属性字典包括各种信息，如文件的所有者、文件大小、文件的创建日期，等等。字典的每个属性都可以根据其键来提取，而所有键都定义在头文件<Foundation/NSFileManager.h>中。例如，表示文件大小的键为NSFileSize。

代码清单16-1展示了一些基本的文件操作。这个例子假设当前目录中存在一个名为testfile的文件，文件的内容如下：

```
This is a test file with some data in it.
Here's another line of data.
And a third.
```

代码清单16-1

```
// Basic file operations
// Assumes the existence of a file called "testfile"
// in the current working directory

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSDictionary.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString          *fName = @"testfile";
   NSFileManager     *fm;
   NSDictionary      *attr;

   // Need to create an instance of the file manager

   fm = [NSFileManager defaultManager];

   // Let's make sure our test file exists first

   if ([fm fileExistsAtPath: fName] == NO) {
```

```
        NSLog (@"File doesn't exist!");
        return 1;
    }

    // Now let's make a copy

    if ([fm copyPath: fName toPath: @"newfile"handler: nil] == NO) {
        NSLog (@"File copy failed!");
        return 2;
    }

    // Let's test to see if the two files are identical

    if ([fm contentsEqualAtPath: fName andPath: @ "newfile"] == NO) {
        NSLog (@'Files are not equal! ");
        return 3;
    }

    // Now let's rename the copy

    if ([fm movePath: @"newfile'toPath: @"newfile2 "
            handler: nil] == NO) {
        NSLog (@'File rename failed!");
        return 4;
    }

    // Get the size of newfile2

    if ((attr = [fm fileAttributesAtPath: @"newfile2'
                traverseLink: NO]) == nil) {
        NSLog (@"Couldn't get file attributes!");
        return 5;
    }

    NSLog (@"File size is %i bytes",
              [[attr objectForKey: NSFileSize] intValue]);

    // And finally, let's delete the original file

    if ([fm removeFileAtPath: fName handler: nil] == NO) {
        NSLog (@"File removal failed!");
        return 6;
    }

    NSLog (@"All operations were successful!");

    // Display the contents of the newly-created file
```

```
    NSLog (@"%@", [NSString stringWithContentsOfFile: @"newfile2" encoding:
            NSUTF8StringEncoding error: nil]);

    [pool drain];
    return 0;
}
```

代码清单16-1　输出

```
File size is 84 bytes
All operations were successful!

This is a test file with some data in it.
Here's another line of data.
And a third.
```

这个程序首先测试testfile文件是否存在。如果存在，则复制testfile文件，然后比较原文件和复制文件是否相等。经验丰富的UNIX用户都知道，不能只通过为方法copyPath:toPath:和movePath:toPath:指定目标目录，就将文件移动或复制到这个目录中；必须明确地指定目标目录中的文件名。

注意　可以使用Xcode创建testfile，方法是选择File菜单中的New File...。在出现的左窗格中，突出显示Other，然后选择右侧窗格中的Empty File。键入testfile作为文件名，并且创建时该文件所在的目录一定要与可执行文件所处的目录相同。这将是项目的Build/Debug文件夹。

movePath:toPath:方法可以用来将文件从一个目录移到另一个目录中（也可以用来移动整个目录）。如果两个路径引用同一目录中的文件（如本例所示），其结果仅仅是重新命名这个文件。因此，在代码清单16-1中，使用这个方法将文件newfile重新命名为newfile2。

如表16-1所示，在执行复制、重命名或移动操作时，目标文件不能是已存在的；否则，操作将失败。

newfile2的大小是通过使用fileAttributesAtPath:traverseLink:方法确定的。测试并确保返回了一个非nil目录，然后使用NSDictionary类中的方法objectForKey:，并用键NSFileSize从字典中获得文件的大小。最后，显示字典中表示文件大小的整数值。

程序使用removeFileAtPath:handler:方法来删除原始文件testfile。

最后，使用NSString的stringWithContentsOfFile:方法将文件newfile2的内容读入一个字符串对象，然后这个对象作为参数传递给要显示的NSLog。

在代码清单16-1中，测试每个文件操作以检查它是否成功。如果任何一个操作失败，就会使用NSLog来记录错误，并且程序将通过返回一个非零的退出状态而退出。根据约定，每个非零值都表示一次程序失败，并且根据错误类型，这个值都是唯一的。如果正在编写命令行工具，这将是一项有用的技术，因为可以由另一个程序来测试返回值，比如从一个shell脚本中测试。

16.1.1 使用NSData类

使用文件时，需要频繁地将数据读入一个临时存储区，它通常称为缓冲区。当收集数据，以便随后将这些数据输出到文件中时，通常也使用存储区。Foundation的NSData类提供了一种简单的方式，它用来设置缓冲区、将文件的内容读入缓冲区，或将缓冲区的内容写到一个文件。有一点不要奇怪，对于32位应用程序，NSDATA缓冲区最多可存储2GB的数据。对于64位应用程序，最多可存储8EG（注意是EB），即80亿GB的数据。

正如你所期望的，我们既可以定义不可变缓冲区（使用NSData类），也可以定义可变的缓冲区（使用NSMutableData类）。在本章和后续几章将介绍这个类的方法。

代码清单16-2展示了如何方便地将文件的内容读入内存缓冲区。

这个程序读取文件newfile2的内容，并将其写入一个名为newfile3的新文件中。从某种意义来说，它实现了文件的复制操作，尽管它采取的方式并不像方法copyPath:toPath:handler:那样直接。

代码清单16-2

```
// Make a copy of a file

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSFileManager     *fm;
   NSData            *fileData;

   fm = [NSFileManager defaultManager];

   // Read the file newfile2

   fileData = [fm contentsAtPath: @"newfile2"];

   if (fileData == nil) {
         NSLog (@"File read failed!");
         return 1;
   }

   // Write the data to newfile3

   if ([fm createFileAtPath: @"newfile3"contents: fileData
                  attributes: nil] == NO) {
```

```
        NSLog (@"Couldn't create the copy!");
        return 2;
    }

    NSLog (@"File copy was successful!");

    [pool drain];
    return 0;
}
```

代码清单16-2　输出

```
File copy was successful!
```

NSData的contentsAtPath:方法仅仅接受一个路径名，并将指定文件的内容读入该方法创建的存储区；如果读取成功，这个方法将返回存储区对象作为结果，否则（例如，该文件不存在或者你不能读取），将返回nil。

方法createFileAtPath:contents:attributes:创建了一个具有特定属性（或者如果attributes参数提供为nil，则采用默认的属性值）的文件。然后，将指定的NSData对象内容写入这个文件中。在本例中，数据区包含前面读取的文件内容。

16.1.2　使用目录

表16-2总结了NSFileManager提供的用于处理目录的一些方法。其中大多数方法和用于普通文件的方法相同，如表16-1所示。

表16-2　常见的NSFileManager目录方法

方　法	描　述
-(NSString *) currentDirectoryPath	获取当前目录
-(BOOL) changeCurrentDirectoryPath: path	更改当前目录
-(BOOL) copyPath: from toPath: to handler: handler	复制目录结构，to不能是已存在的
-(BOOL) createDirectoryAtPath: path attributes: attr	创建一个新目录
-(BOOL) fileExistsAtPath: path isDirectory: (BOOL *) flag	测试文件是不是目录（flag中存储结果YES/NO）
-(NSArray *)directoryContentsAtPath: path	列出目录内容
-(NSDirectoryEnumerator *) enumeratorAtPath: path	枚举目录的内容
-(BOOL) removeFileAtPath: path handler: handler	删除空目录
-(BOOL) movePath: from toPath: to handler: handler	重命名或移动一个目录，to不能是已存在的

代码清单16-3展示了一些使用目录的基本操作。

代码清单16-3

```
// Some basic directory operations

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
```

```
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString          *dirName = @"testdir";
   NSString          *path;
   NSFileManager     *fm;

   // Need to create an instance of the file manager

   fm = [NSFileManager defaultManager];

   // Get current directory

   path = [fm currentDirectoryPath];
   NSLog (@"Current directory path is %@ ", path);

   // Create a new directory

   if ([fm createDirectoryAtPath: dirName attributes: nil] == NO) {
         NSLog (@"Couldn't create directory!");
         return 1;
   }

   // Rename the new directory

   if ([fm movePath: dirName toPath: @"newdir"handler: nil] == NO) {
         NSLog (@"Directory rename failed!");
         return 2;
   }

   // Change directory into the new directory

   if ([fm changeCurrentDirectoryPath: @ "newdir"] == NO) {
         NSLog (@"Change directory failed! ");
         return 3;
   }

   // Now get and display current working directory

   path = [fm currentDirectoryPath];
   NSLog (@"Current directory path is %@ ", path);

   NSLog (@"All operations were successful!");
```

```
    [pool drain];
    return 0;
}
```

代码清单16-3 输出

```
Current directory path is /Users/stevekochan/progs/ch16
Current directory path is /Users/stevekochan/progs/ch16/newdir
All operations were successful!
```

代码清单16-3很容易理解。出于获得信息的目的，首先获取当前的目录路径，然后，在当前目录中创建一个名为testdir的新目录。然后使用movePath:toPath:handler:方法将新目录testdir重命名为newdir。记住，这个方法还可以用来将整个目录结构（这意味着包括目录的内容）从文件系统的一个位置移动到另一个位置。

重命名新目录之后，程序使用changeCurrentDirectoryPath:方法将这个新目录设置为当前目录。然后，显示当前目录路径，以验证修改是否成功。

16.1.3 枚举目录中的内容

有时需要获得目录的内容列表。使用enumeratorAtPath:方法或者directoryContentsAtPath:方法，都可以完成枚举过程。如果使用第一种方法，一次可以枚举指定目录中的每个文件，默认情况下，如果其中一个文件为目录，那么也会递归枚举它的内容。在这个过程中，通过向枚举对象发送一条skipDescendants消息，可以动态地阻止递归过程，从而不再枚举目录中的内容。

对于directoryContentsAtPath:方法，使用这个方法，可以枚举指定目录的内容，并在一个数组中返回文件列表。如果这个目录中的任何文件本身是个目录，这个方法并不递归枚举它的内容。

代码清单16-4演示了如何在程序中使用这两个方法。

代码清单16-4

```
// Enumerate the contents of a directory

#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSArray.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool     * pool = [[NSAutoreleasePool alloc] init];
    NSString              *path;
    NSFileManager         *fm;
    NSDirectoryEnumerator *dirEnum;
    NSArray               *dirArray;
```

```
    // Need to create an instance of the file manager

    fm = [NSFileManager defaultManager];

    // Get current working directory path

    path = [fm currentDirectoryPath];

    // Enumerate the directory

    dirEnum = [fm enumeratorAtPath: path];

    NSLog (@"Contents of %@: ", path);
     while ((path = [dirEnum nextObject]) != nil)
          NSLog (@"%@ ", path);

     // Another way to enumerate a directory
     dirArray = [fm directoryContentsAtPath:
                                  [fm currentDirectoryPath]];
     NSLog (@'Contents using directoryContentsAtPath: ");

   for ( path in dirArray )
          NSLog (@'%@ ", path);

    [pool drain];
    return 0;
}
```

代码清单16-4 输出

```
Contents of /Users/stevekochan/mysrc/ch16:
a.out
dir1.m
dir2.m
file1.m
newdir
newdir/file1.m
newdir/output
path1.m
testfile

Contents using directoryContentsAtPath:
a.out
dir1.m
dir2.m
file1.m
newdir
path1.m
testfile
```

让我们仔细看看以下代码内容：

```
dirEnum = [fm enumeratorAtPath: path];

NSLog (@"Contents of %@:", path);

while ((path = [dirEnum nextObject]) != nil)
    NSLog (@"%@", path);
```

通过向文件管理器对象（此处是fm）发送enumeratorAtPath:消息来开始目录的枚举过程。enumeratorAtPath:方法返回了一个NSDirectortyEnumerator对象，这个对象存储在dirEnum中。现在，每次向该对象发送nextObject消息时，都会返回所枚举的目录中下一个文件的路径。没有其他文件可供枚举过程使用时，会返回nil。

从代码清单16-4的输出中，可以看到这两种枚举技术的不同之处。enumeratorAtPath:方法列出了newdir目录中的内容，而方法directoryContentsAtPath:没有。如果newdir包含子目录，那么方法enumeratorAtPath:也会枚举其中的内容。

前面提到过，在代码清单16-4中while循环的执行过程中，通过对代码做如下更改，可以阻止任何子目录中的枚举。

```
while ((path = [dirEnum nextObject]) != nil) {
    NSLog (@"%@ ", path);

     [fm fileExistsAtPath: path isDirectory: &flag];

    if (flag == YES)
        [dirEnum skipDescendents];
}
```

这里，flag是一个BOOL变量。如果指定的路径是目录，则fileExistsAtPath:在flag中存储yes，否则存储NO。

另外提醒一下，无需像在这个程序中那样进行快速枚举，使用以下NSLog调用可显示整个dirArray的内容：

```
NSLog (@"%@", dirArray);
```

16.2 使用路径：NSPathUtilities.h

NSPathUtilities.h包含了NSString的函数和分类扩展，它允许你操作路径名。应该尽可能地使用这些函数，以便使程序更独立于文件系统结构以及特定文件和目录的位置。代码清单16-5展示了如何使用NSPathUtilities.h提供的几种函数和方法。

代码清单16-5

```
// Some basic path operations

#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
```

```
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSPathUtilities.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString          *fName = @"path.m";
   NSFileManager     *fm;
   NSString          *path, *tempdir, *extension, *homedir, *fullpath;
   NSString          *upath = @"~stevekochan/progs/../ch16/./path.m";

   NSArray           *components;

   fm = [NSFileManager defaultManager];

   // Get the temporary working directory

   tempdir = NSTemporaryDirectory ();

   NSLog (@"Temporary Directory is %@", tempdir);

   // Extract the base directory from current directory

   path = [fm currentDirectoryPath];
   NSLog (@"Base dir is %@", [path lastPathComponent]);

   // Create a full path to the file fName in current directory

   fullpath = [path stringByAppendingPathComponent: fName];
   NSLog (@"fullpath to %@ is %@", fName, fullpath);

   // Get the file name extension

   extension = [fullpath pathExtension];
   NSLog (@"extension for %@ is %@", fullpath, extension);

   // Get user's home directory

   homedir = NSHomeDirectory ();
   NSLog (@"Your home directory is %@", homedir);

   // Divide a path into its components

   components = [homedir pathComponents];

   for ( path in components)
```

```
        NSLog (@"%@", path);

    // "Standardize" a path

    NSLog (@"%@ => %@", upath ,
           [upath stringByStandardizingPath] );

    [pool drain];
    return 0;
}
```

代码清单16-5 输出

```
Temporary Directory is /var/folders/HT/HTyGLvSNHTuNb6NrMuo7QE+++TI/-Tmp-/
Base dir is examples
fullpath to path.m is /Users/stevekochan/progs/examples/path.m
extension for /Users/stevekochan/progs/examples/path.m is m
Your home directory is /Users/stevekochan
/
Users
stevekochan
~stevekochan/progs/../ch16/./path.m => /Users/stevekochan/ch16/path.m
```

函数NSTemporaryDirectory返回系统中可以用来创建临时文件的目录路径名。如果在这个目录中创建临时文件，一定要在完成任务之后将它们删除。另外，还要确保文件名是唯一的，特别是在应用程序的多个实例同时运行时（参见本章末尾的练习5）更应如此。如果多个用户登录到系统，并且运行同一个应用程序，这种情况就很容易发生。

方法lastPathComponent用来从路径中提取最后一个文件名。当你有一个绝对路径名，并且只想从中获取基本文件名时，这个函数很有用。

stringByAppendingPathComponent:方法用于将文件名附加到路径的末尾。如果指定为接收者的路径名不是以斜线结束，那么该方法将在路径名中插入一个斜线，将路径名和附加的文件名分开。结合使用CurrentDirectory方法和stringByAppendingPathComponent:方法，可以在当前目录中创建文件的完整路径名。这种技术展示在代码清单16-5中。

PathExtension方法给出了指定路径名的文件扩展名。在这个例子中，文件path.m的扩展名为m，该方法返回这个扩展名。如果所给的文件没有扩展名，那么方法仅仅返回一个空字符串。

NSHomeDirectory函数返回当前用户的主目录。使用NSHomeDirectoryForUser函数，同时提供用户名作为函数的参数，可以获得任何特定用户的主目录。

PathComponents方法返回一个数组，这个数组包含指定路径的每个组成部分。代码清单16-5按顺序显示了返回数组的每一元素，并且在单独的输出行上显示每个路径组成部分。

最后，和前面讨论的一样，有时路径名包含代字符（~）。FileManager方法将~用作用户主目录的缩写，或将~user用作指定用户的主目录。如果路径名包含代字符，那么使用stringByStandardizingPath方法可以解析它们。这个方法返回一个路径，同时删除这些特殊字符，

即将其标准化。如果路径名中出现代字符，则可以使用方法stringByExpandingTildeInPath扩展代字符。

16.2.1 常用的路径处理方法

表16-3总结了许多常用的使用路径方法。其中，components是一个NSArray对象，它包含路径中每一部分的字符串对象；path是一个字符串对象，它指定文件的路径；ext是表示路径扩展名的字符串对象（如，@"mp4"）。

表16-3 常用的路径工具方法

方 法	描 述
+(NSString *) pathWithComponents: components	根据components中的元素构造有效路径
-(NSArray *) pathComponents	析构路径，获得组成此路径的各个部分
-(NSString *) lastPathComponent	提取路径的最后一个组成部分
-(NSString *) pathExtension	从路径的最后一个组成部分中提取其扩展名
-(NSString *) stringByAppendingPathComponent: path	将path添加到现有路径的末尾
-(NSString *) stringByAppendingPathExtension: ext	将指定的扩展名添加到路径的最后一个组成部分
-(NSString *) stringByDeletingLastPathComponent	删除路径的最后一个组成部分
-(NSString *) stringByDeletingPathExtension	从文件的最后一部分删除扩展名
-(NSString *) stringByExpandingTildeInPath	将路径中的代字符扩展成用户主目录（~）或指定用户的主目录（~user）
-(NSString *) stringByResolvingSymlinksInPath	尝试解析路径中的符号链接
-(NSString *) stringByStandardizingPath	通过尝试解析~、..（父目录符号）、.（当前目录符号）和符号链接来标准化路径

表16-4展示了一些函数，它可用于获取用户、用户的主目录和存储临时文件的目录信息。

表16-4 常用的路径工具函数

函 数	描 述
NSString *NSUserName (void)	返回当前用户的登录名
NSString *NSFullUserName (void)	返回当前用户的完整用户名
NSString *NSHomeDirectory (void)	返回当前用户主目录的路径
NSString *NSHomeDirectoryForUser (NSString *user)	返回用户user的主目录
NSString *NSTemporaryDirectory (void)	返回可用于创建临时文件的路径目录

你可能还想查看Foundation函数NSSearchPathForDirectoriesInDomains，它可用于查找系统的特殊目录，如Application目录。

16.2.2 复制文件和使用NSProcessInfo类

代码清单16-6说明了如何使用命令行工具来实现简单的文件复制操作。这个命令的用法如下：

```
copy from-file to-file
```

与NSFileManager的copyPath:toPath:handler:方法不同，命令行工具允许to-file是目录名。

在这个例子中，文件以名称from-file被复制到to-file目录中；与copyPath:toPath:handler:方法不同的还有，如果to-file目录已存在，允许重写其内容。这更像标准UNIX的复制命令cp。

通过在main函数中使用argv和argc参数，可以从命令行中获得文件名。这两个参数分别包括了命令行上键入的参数个数（包括命令名），以及指向C风格字符串数组的指针。

并非必须处理C字符串（使用argv参数时，必须处理c字符串），而是利用Foundation中的类NSProcessInfo。NSProcessInfo类中包含一些方法，它们允许你设置或检索正在运行的应用程序（即进程）的各种类型的信息。表16-5总结了这些方法。

表16-5 NSProcessInfo类方法

方 法	说 明
+(NSProcessInfo *) processInfo	返回当前进程的信息
-(NSArray *) arguments	以NSString对象数字的形式返回当前进程的参数
-(NSDictionary *) environment	返回变量/值对词典，以描述当前的环境变量（比如PATH和HOME）及其值
-(int) processIdentifier	返回进程标识符，它是操作系统赋予进程的唯一数字，用于识别每个正在运行的进程
-(NSString *) processName	返回当前正在执行的进程名称
-(NSString *)globallyUniqueString	每次调用这个方法时，都返回不同的单值字符串。可以用这个字符串生成单值临时文件名（参见练习5）
-(NSString *) hostname	返回主机系统的名称（在我的Mac OS X系统中，返回的是Steve-Kochans-Computer.local）
-(unsigned int) operatingSystem	返回表示操作系统的数字（在我的Mac上，返回的值是5）
-(NSString *) operatingSystemName	返回操作系统的名称（在我的Mac机子上，返回常量NSMACH-OperatingSystem，其中可能的返回值定义在NSProcessInfo.h中）
-(NSString *) operatingSystemVersionString	返回操作系统的当前版本（在我的Mac OS X系统中，返回Version 10.5.4(Build 9E17)）
-(void) setProcessName:(NSString *) name	将当前进程名称设置为name。应该谨慎地使用这个方法，因为关于进程名称存在一些假设（比如用户默认的设置常常假设进程名称）

代码清单16-6

```
// Implement a basic copy utility

#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSPathUtilities.h>
#import <Foundation/NSProcessInfo.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSFileManager     *fm;
   NSString          *source, *dest;
```

```
    BOOL                isDir;
    NSProcessInfo       *proc = [NSProcessInfo processInfo];
    NSArray             *args = [proc arguments];

    fm = [NSFileManager defaultManager];

    // Check for two arguments on the command line

    if ([args count] != 3) {
       NSLog (@"Usage: %@ src dest ", [proc processName]);
       return 1;
    }

    source = [args objectAtIndex: 1];
    dest = [args objectAtIndex: 2];

    // Make sure the source file can be read

    if ([fm isReadableFileAtPath: source] == NO) {
       NSLog (@"Can't read %@ ", source);
       return 2;
    }

    // See if the destination file is a directory
    // if it is, add the source to the end of the destination

    [fm fileExistsAtPath: dest isDirectory: &isDir];

    if (isDir == YES)
       dest = [dest stringByAppendingPathComponent:
             [source lastPathComponent]];

    // Remove the destination file if it already exists

    [fm removeFileAtPath: dest handler: nil];

    // Okay, time to perform the copy

    if ([fm copyPath: source toPath: dest handler: nil] == NO) {
       NSLog (@"Copy failed! ");
       return 3;
    }

    NSLog (@"Copy of %@ to %@ succeeded! ", source, dest);

    [pool drain];
    return 0;
}
```

代码清单16-6　输出

```
$ ls -l          see what files we have
total 96
-rwxr-xr-x 1 stevekoc staff 19956 Jul 24 14:33 copy
-rw-r--r-- 1 stevekoc staff 1484 Jul 24 14:32 copy.m
-rw-r--r-- 1 stevekoc staff 1403 Jul 24 13:00 file1.m
drwxr-xr-x 2 stevekoc staff 68 Jul 24 14:40 newdir
-rw-r--r-- 1 stevekoc staff 1567 Jul 24 14:12 path1.m
-rw-r--r-- 1 stevekoc staff 84 Jul 24 13:22 testfile
$ copy        try with no args
Usage: copy src dest
$ copy foo copy2
Can't read foo
$ copy copy.m backup.m
Copy of copy.m to backup.m succeeded!
$ diff copy.m backup.m    compare the files
$ copy copy.m newdir      try copy into directory
Copy of copy.m to newdir/copy.m succeeeded!
$ ls -l newdir
total 8
-rw-r--r-- 1 stevekoc staff 1484 Jul 24 14:44 copy.m
$
```

NSProcessInfo类中的argments方法返回一个字符串对象数组。数组的第一个元素是进程名称；其余的元素是在命令行键入的参数。

首先检查，以确保在命令行键入这两个参数。这是通过测试数组args的大小实现的，这个数组是从方法arguments返回的。如果测试成功，那么程序将从数组args中提取源文件名和目标文件名，并将它们的值分别赋给source和test。

然后程序测试源文件是否能够读取。如果不能，则生成一条出错消息，并退出程序。

语句

```
[fm fileExistsAtPath: dest isDirectory: &isDir];
```

检查dest指定的文件是否是目录。前面讲过，答案YES或NO将存储到变量isDir中。

如果dest是目录，则要将源文件名的最后一部分附加到dest目录名的末尾，使用路径工具方法stringByAppendingPathComponent:来实现这个功能。因此，如果Source的值是ch16/copy1.m，dest的值是/Users/stevekochan/progs，并且后者是个目录，那么dest的值将更改为/Users/stevekochan/progs/copy1.m。

方法copyPath:ToPath:handler:不容许重写文件。因此，要避免错误，程序尝试用方法removeFileAtPath:handler:来删除目标文件。没有必要担心这个方法能否成功；因为如果目标文件不存在，它将失败。

到达程序末尾时，可以假设程序的所有部分都运行良好，并为此生成一条消息。

16.3 基本的文件操作：NSFileHandle

利用NSFileHandle类提供的方法，允许更有效地使用文件。在本章的开始，我们列举了利用这些方法可以完成的一些操作。

一般而言，我们处理文件时都要经历以下三个步骤：

1. 打开文件，并获取一个NSFileHandle对象，以便在后面的I/O操作中引用该文件。
2. 对打开的文件执行I/O操作。
3. 关闭文件。

表16-6总结了一些常用的NSFileHandle方法。在这个表中，fh是一个NSFileHandle对象，data是一个NSData对象，path是一个NSString对象，offset是一个unsigned long long。

表16-6 常用的NSFileHandle方法

方 法	描 述
+(NSFileHandle *) fileHandleForReadingAtPath: path	打开一个文件准备读取
+(NSFileHandle *) fileHandleForWritingAtPath: path	打开一个文件准备写入
+(NSFileHandle *)fileHandleForUpdatingAtPath: path	打开一个文件准备更新（读取和写入）
-(NSData *) availableData	从设备或者通道返回可用的数据
-(NSData *) readDataToEndOfFile	读取其余的数据直到文件的末尾（最多UINT_MAX字节）
-(NSData *) readDataOfLength: (unsigned int) bytes	从文件中读取指定数目bytes的内容
-(void) writeData: data	将data写入文件
-(unsigned long long) offsetInFile	获取当前文件的偏移量
-(void) seekToFileOffset: offset	设置当前文件的偏移量
-(unsigned long long) seekToEndOfFile	将当前文件的偏移量定位到文件的末尾
-(void) truncateFileAtOffset: offset	将文件的长度设置为offset字节（如果需要，可以填充内容）
-(void) closeFile	关闭文件

上表中并未列出获取NSFileHandle以用于标准输入、标准输出、标准错误和空设备的方法。它们的格式为fileHandleWithDevice，其中Device可以是StandardInput、StandardOutput、StandardError或NullDevice。

这里没有列出用于后台（也就是，异步）读取和写入的方法。

应该注意到FileHandle类并没有提供创建文件的功能。前面描述过，必须使用FileManager方法来创建文件。因此，方法fileHandleForWritingAtPath：和fileHandleForUpdatingAtPath：都假定文件已存在，否则返回nil。对于这两个方法，文件的偏移量都设为文件的开始，所以都是在文件的开始位置开始写入（或更新模式的读取）。另外，如果在UNIX系统下编程应该注意，打开用于读取的文件，不要截断文件；如果想要这么做，不得不自己完成这项操作。

代码清单16-7打开本章开始创建的原始testfile文件，读取它的内容，并将其复制到名为testout的文件中。

代码清单16-7

```
// Some basic file handle operations
// Assumes the existence of a file called "testfile"
// in the current working directory

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileHandle.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSFileHandle      *inFile, *outFile;
   NSData            *buffer;

   // Open the file testfile for reading

   inFile = [NSFileHandle fileHandleForReadingAtPath: @ "testfile"];

   if (inFile == nil) {
        NSLog (@"Open of testfile for reading failed");
        return 1;
   }

   // Create the output file first if necessary

   [[NSFileManager defaultManager] createFileAtPath: @"testout"
        contents: nil attributes: nil];

   // Now open outfile for writing

   outFile = [NSFileHandle fileHandleForWritingAtPath: @ "testout"];

   if (outFile == nil) {
        NSLog (@"Open of testout for writing failed");
        return 2;
   }

   // Truncate the output file since it may contain data

   [outFile truncateFileAtOffset: 0];

   // Read the data from inFile and write it to outFile
```

```
    buffer = [inFile readDataToEndOfFile];

    [outFile writeData: buffer];

    // Close the two files

    [inFile closeFile];
    [outFile closeFile];

    // Verify the file's contents
    NSLog (@"%@", [NSString stringWithContentsOfFile: @"testout" encoding:
            NSUTF8StringEncoding error: nil]);

    [pool drain];
    return 0;
}
```

代码清单16-7 输出

```
This is a test file with some data in it.
Here's another line of data.
And a third..
```

方法readDataToEndOfFile:每次从文件中读取最多UINT_MAX个字节的数据，这个量定义在头文件<limits.h>中，并且在许多系统中值等于$FFFFFFFF_{16}$。这个值对于你编写的任何应用程序而言，已经足够大了。还可以中断这项操作，以执行少量读取和写入。利用方法readDataOfLength:，甚至可以设置循环，一次在文件之间传输一缓冲区的字节。缓冲区的大小可能是8192（8Kb）字节，也可以是131072（128Kb）字节。经常使用的是2的乘方，这是因为底层的操作系统通常以块为单位执行I/O操作的，而块的大小一般为2的乘方个字节。可能要在系统上试用不同的值，以查看哪个值最适合。

如果读取方法到达文件的末尾，并且没有读到任何数据，那么这个方法将返回一个空的NSData对象（也就是，缓冲区中没有字节）。可以对这个缓冲区应用length方法，并测试长度是否等于零，以查看该文件中是否还剩有数据可以读取。

如果打开一个要更新的文件，则文件的偏移量要被设为文件的开始。通过在文件中定位(seeking)，可以更改偏移量，然后执行该文件的读写操作。因此，要定位到文件（文件的句柄为databaseHandle）的第10个字节，可以编写如下消息表达式：

```
[databaseHandle seekToFileOffset: 10];
```

通过获得当前的文件偏移量，然后加上或者减去这个值，就得到相对文件位置。因此，要跳过文件中当前位置之后的128个字节，编写如下代码：

```
[databaseHandle seekToFileOffet:
            [databaseHandle offsetInFile] + 128];
```

要在文件中向回移动5个整数所占的字节数，编写如下代码：

```
[databaseHandle seekToFileOffet:
            [databaseHandle offsetInFile] - 5 * sizeof (int)];
```

代码清单16-8将一个文件的内容附加到另一个文件中。通过打开另一个用于写入的文件，然后定位到该文件的结尾，最后将第一个文件中的内容写入第二个文件中来实现。

代码清单16-8

```
// Append the file "fileA"to the end of "fileB"

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileHandle.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool     * pool = [[NSAutoreleasePool alloc] init];
   NSFileHandle          *inFile, *outFile;
   NSData                *buffer;

   // Open the file fileA for reading

   inFile = [NSFileHandle fileHandleForReadingAtPath: @ "fileA"];

   if (inFile == nil) {
      NSLog (@"Open of fileA for reading failed");
      return 1;
   }

   // Open the file fileB for updating

   outFile = [NSFileHandle fileHandleForWritingAtPath: @ "fileB"];

   if (outFile == nil) {
         NSLog (@'Open of fileB for writing failed");
         return 2;
   }

   // Seek to the end of outFile

   [outFile seekToEndOfFile];

   // Read inFile and write its contents to outFile

   buffer = [inFile readDataToEndOfFile];
```

```
        [outFile writeData: buffer];

        // Close the two files

        [inFile closeFile];
        [outFile closeFile];

        [pool drain];
        return 0;
}

Contents of FileA before running Program 16.8

This is line 1 in the first file.
This is line 2 in the first file.

Contents of FileB before running Program 16.8

This is line 1 in the second file.
This is line 2 in the second file.
```

代码清单16-8 输出

```
Contents of fileB
This is line 1 in the second file.
This is line 2 in the second file.
This is line 1 in the first file.
This is line 2 in the first file.
```

从输出可知，第一个文件的内容成功地附加到第二个文件的末尾。顺便说明一下，在搜索操作执行完毕之后，seekToEndOfFile返回当前文件的偏移量。选择忽略这个值；如果需要，可以使用这个信息来获得程序中文件的大小。

16.4 练习

1. 修改代码清单16-6中开发的复制程序，以便它像标准的UNIX命令一样，可以接收多个要复制到该目录的源文件。如下命令

```
$ copy copy1.m file1.m file2.m progs
```

应该将三个文件copy1.m、file1.m和fiel2.m复制到目录progs中。如果指定了多个源文件，那么最后一个参数实际上是现有的目录。

2. 编写一个名为myfind命令行工具，它带有两个参数。第一个是开始搜索的初始目录，第二个参数是需要定位的文件名。命令行

```
$ myfind /Users proposal.doc
/Users/stevekochan/MyDocuments/proposals/proposal.doc
$
```

首先搜索/users的文件系统以查找文件proposal.doc。如果找到该文件，则输出该文件的完整路径名；如果没有找到，则输出一条合适的消息。

3. 编写自己的标准UNIX工具，basename和dirname。
4. 使用NSProcessInfo编写一个程序，用于显示每个取值函数方法所返回的所有信息。
5. 给定本章中介绍过的NSPathUtilities.h函数NSTemporaryDirectory和NSProcessInfo方法globallyUniqueString，将名为TempFiles的分类添加到NSString中，并定义一个名为temporaryFileName的方法，每次调用这个方法都返回单值的文件名。
6. 修改代码清单16-7，以便该文件一次读取和写入kBufSize个字节，其中kBufSize定义在程序的开始部分。一定要对大型文件测试这个程序（也就是大于kBufSize字节的文件）。
7. 打开一个文件，一次从中读取128个字节，并将其写到终端。利用FileHandle的方法fileHandleWithStandardOutput来获得终端输出的句柄。

第17章 内存管理

对于内存管理这个主题的关注贯穿全书。现在，应该已经明白什么时候由你负责释放对象，以及什么时候不由你释放。虽然本书中所有的例子都很小，但我们仍然强调内存管理的重要性，以便讲述好的程序设计习惯，并开发无内存泄漏的程序。

根据编写的应用程序类型，明智的内存使用可能很重要。例如，假如编写一个交互式的制图应用程序，这个程序在执行过程中创建很多对象，如果不够小心，那么你的程序在运行过程中可能会不断地消耗越来越多的内存资源。在这种情况下，明智地管理这些资源并在不再需要它们时释放就成为你的责任。这就意味着，在程序执行的过程中而不是等到程序结束时释放资源。

本章将更详细地讲解Foundation的内存分配策略。这涉及到对自动释放池的详细讨论和保持对象（retaining object）的概念。还将学习对象的引用计数。最后，我们会介绍称为垃圾回收的机制，这可缓解因保留对象并在使用完后释放这些对象的压力。然而你会看到，不能为iPhone应用程序使用垃圾回收，所以你仍需要理解本书中介绍的内存管理技术（本章会详细介绍这些内容）。

17.1 自动释放池

通过本书第二部分的程序例子，你已经熟悉了自动释放池。在处理Foundation程序时，为了使用Foundation对象，必须设置自己的池。系统使用这个池来跟踪对象，以便以后释放。在应用程序中，可以通过调用来建立这个池，如下所示：

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

建立了自动释放池之后，Foundation将自动为这个池添加特定的数组、字符串、字典以及其他对象。使用完该池时，可以发送drain消息来释放它使用的内存：

```
[pool drain];
```

任何标记为自动释放并因此添加到池中的对象，将在池本身被释放的同时也自动释放，自动释放池的名称就是由此而来的。事实上，程序中可以有多个自动释放池，并且它们还可以是嵌套的。

如果你的程序产生了大量的临时对象（这种情况在执行循环中的代码时经常发生），可能需要在程序中创建多重自动释放池。例如，代码片段

```
NSAutoreleasePool  *tempPool;
  ...
for (i = 0; i < n; ++i) {
  tempPool = [[NSAutoReleasePool alloc] init];
     ...  // lots of work with temporary objects here
  [tempPool drain];
}
```

说明了如何创建几个自动释放池来释放由for循环的每一次迭代而产生的临时对象。

应该注意到自动释放池并不包含实际的对象本身，仅仅是对释放池时要释放的对象的引用。

通过向当前的自动释放池发送一条autorelease消息，可以将一个对象添加到其中，以便以后释放：

```
[myFraction autorelease];
```

然后，系统将myFraction添加到自动释放池中以便将来自动释放。你将看到，autorelease方法对于在某个方法内标记对象以便以后进行处理是非常有用的。

17.2 引用计数

谈到基本的Objective-C对象类NSObject时，注意到可以使用alloc方法进行内存分配，并且随后可以使用release消息来释放分配的内存。遗憾的是，事情并不总是这么简单。例如，正在运行的应用程序可以在多个位置引用你创建的对象；该对象可能存储在一个数组中，也可能被其他位置的实例变量引用。只有在确定使用该对象的每个人都使用完之后，才能释放它占用的内存。

幸运的是，Foundation框架提供了一个巧妙的解决方案，用于跟踪对象的引用次数。它涉及一个相当简单直观的技术，称为引用计数。其概念如下：创建对象时，将它的引用次数设置为1。每一次必须保持该对象时，就发送一条retain消息，使其引用次数加1，如下所示：

```
[myFraction retain];
```

Foundation框架提供的其他一些方法也可以增加对象的引用次数，例如，把对象添加到数组中时。

不再需要对象时，可以通过发送release消息，使对象的引用次数减1，如下所示：

```
[myFraction release];
```

当对象的引用次数达到0时，系统就知道不再需要这个对象（因为在理论上它不再被引用），因此系统就会释放（deallocates）它的内存。这是通过向对象发送一条dealloc消息而实现的。

对于程序员而言，要成功地运用这个策略需要足够的勤勉，以保证在程序运行期间正确地增减对象的引用次数。你将看到，系统会自动处理一些对象的引用次数，但并不是对所有对象都是如此。

了解一下引用计数的更多细节。通过向对象发送retainCount消息，可以获得这个对象的引用（或者保持）计数。通常情况下，永远也不会需要使用这个方法，但在这里，它对于说明问题非常有用（参见代码清单17-1）。注意，它返回一个NSUInteger类型的无符号整数。

代码清单17-1

```
// Introduction to reference counting

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
```

```
#import <Foundation/NSValue.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSNumber          *myInt = [NSNumber numberWithInteger: 100];
    NSNumber          *myInt2;
    NSMutableArray    *myArr = [NSMutableArray array];

    NSLog (@ "myInt retain count = %lx ",
        (unsigned long) [myInt retainCount]);

    [myArr addObject: myInt];
    NSLog (@ "after adding to array = %lx ",
        (unsigned long) [myInt retainCount]);

    myInt2 = myInt;
    NSLog (@ "after asssignment to myInt2 = %lx ",
        (unsigned long) [myInt retainCount]);

    [myInt retain];
    NSLog (@ "myInt after retain = %lx ",
        (unsigned long) [myInt retainCount]);

    NSLog (@ "myInt2 after retain = %lx ",
        (unsigned long) [myInt2 retainCount]);

    [myInt release];
    NSLog (@ 'after release = %lx ',
        (unsigned long) [myInt retainCount]);

    [myArr removeObjectAtIndex: 0];
    NSLog (@ 'after removal from array = %lx ",
        (unsigned long) [myInt retainCount]);

    [pool drain];
    return 0;
}
```

代码清单17-1 输出

```
myInt retain count = 1
after adding to array = 2
after assignment to myInt2 = 2
myInt after retain = 3
myInt2 after retain = 3
after release = 2
after removal from array = 1
```

NSNumber对象myInt被设置为整数100，并且程序输出显示它的初始保持计数为1。然后，使用addObject:方法将该对象加到数组myArr中。注意引用次数随之变为2。这是通过addObject:方法自动完成的。如果检查addObject:方法的文档，将发现这里说明的事实。将对象添加到任何类型的集合都会使该对象的引用次数增加。这意味着，如果随后释放了添加的对象，那么数组中仍然存在着该对象的有效引用，并且对象不会被释放。

接下来，将myInt赋值给变量myInt2。要注意这个操作并没有使myInt对象的引用次数增加，这可能会在以后造成潜在的麻烦。例如，如果对myInt的引用次数减少到0，并且它占用的空间被释放，那么myInt2将拥有无效的对象引用（请记住，将myInt赋值给myInt2的操作并没有复制实际的对象，只是指向myInt在内存中位置的指针）。

现在，因为myInt对象有另一个引用（通过myInt2），所以可以通过发送retain消息来增加它的引用次数。在代码清单17-1中使用过这个方法。可以看到，发送retain消息之后，它的引用数变为3。第一个引用是实际的对象本身，第二个是数组中的引用，第三个来自赋值引用。虽然将元素存储到数组会使对象的引用次数自动增加，但把它赋值给另一个变量时，引用次数不会自动增加，因此必须自己增加对象的引用次数。注意程序输出中myInt和myInt2的引用次数都是3，这是因为它们引用的是内存中的同一个对象。

假设在程序中已经使用完myInt对象。通过向其发送一条release消息，可以告知系统。可以看到，对象的引用次数从3降为2。因为引用次数并不为0，所以对象的其他引用（从数组和通过myInt2）仍然有效。只要对象具有非零的引用次数，系统就不会释放对象使用的内存。

如果使用removeObjectAtIndex:方法来删除数组myArr中的第一个元素，那么你将注意到对象myInt的引用值自动减为1。一般而言，从任何集合中删除对象都能够使其引用计数减小。这意味着下面的代码序列

```
myInt = [myArr ObjectAtIndex: 0];
 ...
[myArr removeObjectAtIndex: 0]
  ...
```

可能会产生麻烦。这是因为，在这种情况下，如果在调用removeObjectAtIndex:方法之后，对象的引用值减为0，那么myInt引用的对象将成为无效的。当然，此处的解决方案是从数组检索该值后，保持myInt，这样不管其他位置这个引用发生了什么事情，都不会产生任何影响。

17.2.1 引用计数和字符串

代码清单17-2演示了如何对string对象应用引用计数。

代码清单17-2

```
// Reference counting with string objects

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
```

```
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSString           *myStr1 = @ "Constant string ";
    NSString           *myStr2 = [NSString stringWithString: @ "string 2 "];
    NSMutableString    *myStr3 = [NSMutableString stringWithString: @ "string 3 "];
    NSMutableArray     *myArr = [NSMutableArray array];

    NSLog (@ "Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx ",
             (unsigned long) [myStr1 retainCount],
             (unsigned long) [myStr2 retainCount],
             (unsigned long) [myStr3 retainCount]);

    [myArr addObject: myStr1];
    [myArr addObject: myStr2];
    [myArr addObject: myStr3];

    NSLog (@ "Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx ",
           (unsigned long) [myStr1 retainCount],
           (unsigned long) [myStr2retainCount],
             (unsigned long) [myStr3 retainCount]);

    [myArr addObject: myStr1];
    [myArr addObject: myStr2];
    [myArr addObject: myStr3];

    NSLog (@ "Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx ",
           (unsigned long) [myStr1 retainCount],
           (unsigned long) [myStr2retainCount],
             (unsigned long) [myStr3 retainCount]);

    [myStr1 retain];
    [myStr2 retain];
    [myStr3 retain];

    NSLog (@ "Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx ",
           (unsigned long) [myStr1 retainCount],
           (unsigned long) [myStr2 retainCount],
             (unsigned long) [myStr3 retainCount]);

    // Bring the reference count of myStr3 back down to 2
    [myStr3 release];

    [pool drain];
    return 0;
}
```

代码清单17-2　输出

```
Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 1
Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 2
Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 3
```

将NSString对象myStr1赋值为NSConstantString @"Constant string"。内存中常量字符串的空间分配与其他对象不同。它们没有引用计数机制，因为永远不能释放这些对象，这就是为什么向myStr1发送消息retainCount，而它返回0xffffffff的原因（实际上，在标准头文件<limits.h>中，值0xffffffff被定义为最大的无符号整数，或者UINT_MAX）。

注意　显然，在某些系统上，为代码清单17-2中的常量字符串返回的保持值是0x7fffffff（不是0xffffffff），这是最大的无符号整数值，或INT_MAX。

注意，这也同样适用于使用常量字符串初始化的不可变字符串对象；这种对象也没有保持值，为myStr2显示的保持值可以证明这一点。

注意　此处系统比较聪明，确定了不可变字符串对象是由常量字符串对象初始化的。Leopard发布之前，这种优化是没有的，mystr2会有一个保持值。

在语句

```
NSMutableString  *myStr3 = [NSMutableString stringWithString: @"string 3"];
```

中，变量myStr3被设置为常量字符串@"string 3"的副本。制作字符串副本的原因是向NSMutableString类发送了stringWithString:消息，表明该字符串的内容可能在程序执行期间发生变化。由于常量字符串的内容是不能改变的，所以系统不能将变量myStr3设为指向常量字符串@"string 3"，而myStr2是可以的。

所以字符串对象myStr3没有引用计数，从程序输出可以证实。通过将这些字符串添加到数组中，或者向它们发送retain消息，可以更改这些引用计数，通过最后两行NSLog调用可以得到证实。在创建这两个对象时，通过Foundation的stringWithString:方法将这两个对象添加到自动释放池。通过Foundation的array方法将数组myArr添加到池中。

在释放自动释放池之前，myStr3被释放一次。这导致它的引用计数减为2。随后，自动释放池的释放使得这些对象的引用计数减为0，这导致它被释放。这是如何发生的呢？释放自动释放池时，每次向它发送autorelease消息时，池中的每个对象都会收到一条release消息。由于在stringWithString:方法创建字符串对象myStr3时，将它们添加到自动释放池，因此它们都会收到一条release消息。这导致它们的引用计数减为1。当自动释放池中的数组被释放时，数组中的每个元素也被释放。因此，从池中释放myArr时，该数字的每个元素（包括myStr3）都将收到一条release消息。这使其引用计数减为0，随之这个对象将被释放。

必须小心地不要过度释放对象。在代码清单17-2中，如果在释放该池之前，myStr3的引用计数少于2，那么该池将包含一个无效对象的引用。然后，释放该池时，无效对象的引用非常可能导致程序以一条分段识别错误而异常终止。

17.2.2 引用计数与实例变量

处理实例变量时，必须注意它们的引用计数。例如，回想一下类AddressCard中的setName:方法：

```
-(void) setName: (NSString *) theName
{
    [name release];
    name = [[NSString alloc] initWithString: theName];
}
```

假设使用下面这种方式定义setName:方法，并且它没有name对象的所有权。

```
-(void) setName: (NSString *) theName
{
   name = theName;
}
```

这个版本的setName:方法使用代表人名的字符串，并将它存储到name实例变量中。这看起来简单明了，然而考虑下面的方法调用：

```
NSString *newName;
   ...
[myCard setName: newName];
```

假设newName是一个临时存储空间，它存储要添加到地址卡上的人名，并且后来你要释放它。你认为myCard中的实例变量name将发生什么呢？因为它引用了已经销毁的对象，所以它的name域将不再有效。这就是你的类必须拥有自己的成员对象的原因：这样就不用担心无意中修改或释放这些对象。

下面几个例子更详细地说明了该点。首先定义一个新类ClassA，这个类有一个实例变量：字符串对象str。必须为这个变量编写取值方法和赋值方法。我们在此没有合并这些方法，然而我们自己写出了这些方法，这样可清楚地知道发生了什么。

代码清单17-3

```
// Introduction to reference counting

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>

@interface ClassA: NSObject
{
   NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
@end
```

```
@implementation ClassA
-(void) setStr: (NSString *) s
{
   str = s;
}

-(NSString *) str
{
   return str;
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleas ePool * pool = [[NSAutoreleasePool alloc] init];
   NSMutableString *myStr = [NSMutableStr ing stringWithString: @ "A string"];
   ClassA   *myA = [[ClassA alloc] init];

   NSLog (@ "myStr retain count: %x', [myStr retainCount]);

   [myA setStr: myStr];
   NSLog (@ "myStr retain count: %x', [myStr retainCount] ;

   [myA release];
   [pool drain];
   return 0;
}
```

代码清单17-3　输出

```
myStr retain count: 1
myStr retain count: 1
```

这个程序仅仅分配了一个ClassA对象myA，然后调用它的赋值方法，将其设置为myStr指定的NSString对象。和你期望的一样，在调用setStr方法前后，myStr的引用计数都为1，这是因为该方法只是将它的参数值存储到实例变量str中。然而，再次声明，如果程序在调用setStr方法之后释放myStr，那么存储在实例变量str中的值将变成无效的，这是因为它的引用计数将减为0，并且将释放它引用的对象所占用的内存空间。

在代码清单17-3中，当自动释放池被释放时，情况的确如此。即使我们没有显式地将对象添加到自动释放池，使用stringWithString:方法创建字符串对象myStr时，该对象也被添加到自动释放池中。所以，该池被释放时，myStr也被释放。因此，在池被释放后对它的任何访问都将无效。

代码清单17-4更改了setStr:方法，以保持str的值。这样，以后其他人释放str引用的对象时，你不会受到影响。

代码清单17-4

```
// Retaining objects

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>

@interface ClassA: NSObject
{
   NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
@end

@implementation ClassA
-(void) setStr: (NSString *) s
{
   str = s;
  [str retain];
}

-(NSString *) str
{
   return str;
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *myStr = [NSMutableString stringWithString: @ 'A string'];
   ClassA *myA = [[ClassA alloc] init];

   NSLog (@ "myStr retain count: %x", [myStr retainCount]);

   [myA setStr: myStr];
   NSLog (@ "myStr retain count: %x", [myStr retainCount]);

   [myStr release];
   NSLog (@ "myStr retain count: %x", [myStr retainCount]);

   [myA release];
   [pool drain];
   return 0;
}
```

代码清单17-4 输出

```
myStr retain count: 1
myStr retain count: 2
myStr retain count: 1
```

可以看到调用setStr:方法之后，myStr的引用值升为2。因此，这个问题就得到了解决。随后，释放程序中的myStr时，因为它的引用计数为1，所以通过这个实例变量的引用仍然有效。

因为在程序中使用alloc为myA分配了资源，所以你仍要负责释放它。要想不必自己释放它，可以通过向它发送一条autorelease消息，将它添加到自动释放池：

```
[myA autorelease];
```

如果想要，可以在给对象分配资源后立即将它添加到自动释放池中。记住，将对象添加到自动释放池中并没有释放它或使它无效，这仅仅是为以后释放它做了标记。当池被释放时，如果对象的引用值恰好变为0，则释放对象的资源。在此之前，可以继续使用这个对象。

你可能看出，代码清单17-4中还有一些潜在的问题。方法setStr:的作用是保持了作为参数传入的字符串对象，然而什么时候释放这个对象呢？还有，重写实例变量str时，原来的值如何？不应该释放它的值以收回它占用的内存吗？代码清单17-5提供了这些问题的解决方法。

代码清单17-5

```
// Introduction to reference counting

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>

@interface ClassA: NSObject
{
   NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
-(void) dealloc;
@end

@implementation ClassA
-(void) setStr: (NSString *) s
{
   // free up old object since we're done with it
   [str autorelease];

   // retain argument in case someone else releases it
   str = [s retain];
```

```
}

-(NSString *) str
{
   return str;
}

-(void) dealloc {
   NSLog (@ 'ClassA dealloc ");
   [str release];
   [super dealloc];
}
@end

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *myStr = [NSMutableString stringWithString: @ "A string '];
   ClassA *myA = [[ClassA alloc] init];

   NSLog (@ "myStr retain count: %x ", [myStr retainCount]);
   [myA autorelease];

   [myA setStr: myStr];
   NSLog (@ "myStr retain count: %x ", [myStr retainCount]);

   [pool drain];
   return 0;
}
```

代码清单17-5 输出

```
myStr retain count: 1
myStr retain count: 2
ClassA dealloc
```

无论实例变量Str中当前存储的是什么，setStr:方法首先接受这个对象，并将它添加到自动释放池中。也就是，这使得以后可以自动释放str变量。如果在整个程序执行过程中多次调用setStr:方法，以便同一个域设置不同值，那么这项操作尤为重要。每次存储新值时，变量的旧值应该被标记为自动释放。释放旧的值之后，将保持新值并将其存储到str域中。消息表达式

```
str = [s retain];
```

利用了retain方法返回它的接收者这个事实。

注意 如果变量str为nil，这没有问题。Objective-C运行时将所有变量都初始化为nil，向nil发送消息是没有问题的。

Dealloc方法并不是新出现的，在第15章“数字、字符串和集合”中使用类AddressBook和AddressCard时，曾经遇到过它。重载的dealloc方法为你提供了一种简洁的方式，用于在释放str实例变量时（也就是，当它的引用计数变为0时），处理该对象引用的最后一个对象。在这种情况下，系统调用dealloc方法，这个方法继承自NSObject，通常情况下，不需要重载这个方法。对于保持的对象，使用alloc方法分配的对象，或者在方法中复制（使用下一章将讨论的一个复制方法）的对象，可能需要重载dealloc方法，以便有机会释放它们。语句：

```
[str release];
[super dealloc];
```

首先释放str实例变量，然后调用父类的dealloc方法来完成这项工作。

调用dealloc方法时，该方法中的NSLog语句用来输出一条消息。这样做的目的仅仅是为了验证释放自动释放池时正确地释放了ClassA对象。

对于赋值方法setStr，你可能已经发现它最后的一个缺陷。再看一下代码清单17-5。假设myStr是可变字符串而不是不变字符串，进一步假设，在调用setStr方法之后，字符串myStr中的一个或多个字符更改了。更改myStr所引用的字符串，也会影响到其他引用此字符串的实例变量，因为它们实际上引用的是同一对象。重读最后一句，以确保已经理解了这点。并且，认识将myStr设置为全新的字符串对象，不会产生这个问题。仅当通过某种方式修改字符串中的一个或多个字符，才会出现这个问题。

如果想要保护字符串，并且使它完全独立于设置方法的参数，那么这个问题的解决方案就是在设置方法中生成字符串的全新副本。这就是在第15章中，为什么我们选择在类AddressCard的方法setName:和setEmail:方法中创建name和email成员的全新副本。

17.3 自动释放池示例

看一下本章最后一个执行例子，以保证真正地理解对象的引用计数、保持和释放/自动释放的工作方式。检查代码清单17-6，该程序定义了一个名为Foo的虚类，它有一个实例变量以及一个继承来的方法。

代码清单17-6

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>

@interface Foo: NSObject
{
   int x;
}
@end

@implementation Foo
@end

int main (int argc, char *argv[])
```

```
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Foo     *myFoo = [[Foo alloc] init];

    NSLog (@ "myFoo retain count = %x ", [myFoo retainCount]);

    [pool drain];
    NSLog (@ "after pool drain = %x ", [myFoo retainCount]);

    pool = [[NSAutoreleasePool alloc] init];
    [myFoo autorelease];
    NSLog (@ "after autorelease = %x ", [myFoo retainCount]);

    [myFoo retain];
    NSLog (@ "after retain = %x ", [myFoo retainCount]);

    [pool drain];
    NSLog (@ "after second pool drain = %x ", [myFoo retainCount]);

    [myFoo release];
    return 0;
}
```

代码清单17-6 输出

```
myFoo retain count = 1
after pool drain = 1
after autorelease = 1
after retain = 2
after second pool drain = 1
```

该程序分配了一个新的Foo对象，并且将它赋值给变量myFoo。前面已经看到过，它的初始保持计数为1。然而，这个对象并不是自动释放池的一部分，因此，释放池时，这个对象并没有失效。随后，定义了一个新池，并且向myFoo发送一条autorelease消息将其添加到池中。请注意，它的引用计数并没有改变，这是因为将对象添加到自动释放池时，并不影响它的引用数，仅仅是为了以后释放而进行了标记。

然后，向myFoo发送retain消息。这使得它的引用数变为2。随后再次释放池时，myFoo的引用数减为1，这是因为在前面向它发送了autorelease消息，所以，当池被释放时，会向myFoo发送release消息。

因为myFoo在释放池之前是保持的，所以它的引用数在减1后仍大于0。因此，myFoo对象在池释放后仍然有效。当然，现在你必须亲自释放它，在代码清单17-6中，就是这样来释放对象并避免内存泄漏的。

如果对此仍有些疑惑，请重新阅读关于自动释放池的解释。理解代码清单17-6之后，就彻底理解了自动释放池及其工作机理。

17.4 内存管理规则摘要

总结一下，在本章中已学过的内存管理知识：

- 释放对象，可以释放它所占用的内存，如果你的程序在运行期间创建了很多对象，应该关注对象的释放。良好的规则就是，不再使用创建或保持的对象时，就释放它们。
- 发送一条release消息不一定销毁对象，当一个对象的引用计数变为0时，才销毁这个对象。系统通过向该对象送一条dealloc消息来释放它所占用的内存。
- 自动释放池用于在释放池本身时自动释放池中的对象。每次释放池时，系统通过向池中的每一个对象发送一条release消息来实现这个功能。同时，系统会向池中每个引用计数为0的对象发送一条dealloc消息来销毁这个对象。
- 如果你的方法中不再需要一个对象，但需要返回它，那么向其发送一条autorelease消息，将它标记为以后释放。消息autorelease并不影响对象的引用计数。因此，它允许消息的发送者使用这个对象，然而仍然在以后当自动释放池时，释放这些对象。
- 无论对象是否添加到自动释放池，应用程序终止时，都会释放程序中对象占用的所有内存。
- 开发更复杂的应用程序时（例如Cocoa应用程序），可以在程序的运行期间创建和销毁自动释放池（对Cocoa应用程序，每次发生一个事件，就会产生这种情况）。在这样的情况下，如果想要保证在自动释放池被释放时对象仍然存在，则应该显式地保持它。如果对象的引用计数大于收到的autorelease消息数目，则在池被释放时会保留下来。
- 如果使用alloc或copy方法（或使用allocWithZone:、copyWithZone:或mutableCopy方法）直接创建对象，则由你负责释放它。每次retain对象时，应该release或者autorelease它。
- 除上一个规则中提到的方法之外，不必费心地释放其他方法返回的对象。这不是你的责任；这些对象应当被它们的方法自动释放。这就是为什么首先需要在程序中创建自动释放池的原因。stringWithString:之类的方法自动向新创建的字符串对象发送一条autorelease消息，把它们添加到自动释放池。如果事先没有创建池，就会收到一条出错消息，这个消息通知你在没有池的情况下尝试自动释放对象。

17.5 垃圾回收

本书到目前为止，你已经创建了在可管理的内存运行时环境中运行的程序。上一节中总结的内存管理规则适用于这种环境，在这种环境中需要处理自动释放池，与保持和释放对象的相关问题，以及对象所有权。

从Objective C 2.0开始，提供了一种替代的内存管理形式，称为垃圾回收。有了垃圾回收，就不必考虑有关保持和释放对象、自动释放池或保持计数的问题了。系统可自动跟踪拥有其他哪些对象的那些对象，并且在程序执行期间需要内存空间时，自动释放（即垃圾回收）不再被引用的对象。

如果事情如此简单，为什么我们不在本书的例子中都使用垃圾回收，去掉所有关于内存管理的内容呢？其中有3个原因：首先，即使环境支持垃圾回收，最好也要知道是谁拥有对象，并跟踪你何时不再需要它们了。这样在编写代码时就会更加小心，因为你了解程序中对象之间

的关系以及它们的生命周期。

第二个原因前面说过，就是iPhone运行时环境不支持垃圾回收，所以为该平台开发程序时不能使用这个功能。

第三个原因适用于要编写库例程、插件或共享代码的人。因为这些代码可能会被加载到已经垃圾回收或未被垃圾回收的进程中，所以必须要编写在这两种环境中使用的代码。这就意味着，你需要使用本书中介绍的内存管理技巧编写代码，还意味着必须在禁用和启用垃圾回收的情况下测试代码。

如果决定要使用垃圾回收，那么使用Xcode生成程序时必须开启该功能。可通过Project，Edit Project Settings菜单完成该任务。在“GCC 4.0—Code Generation”设置下，会看到一个称为Objective-C Garbage Collection的设置。将该设置从默认值Unsupported改为Required，指定在程序生成时启用自动垃圾回收功能（参见图17-1）。

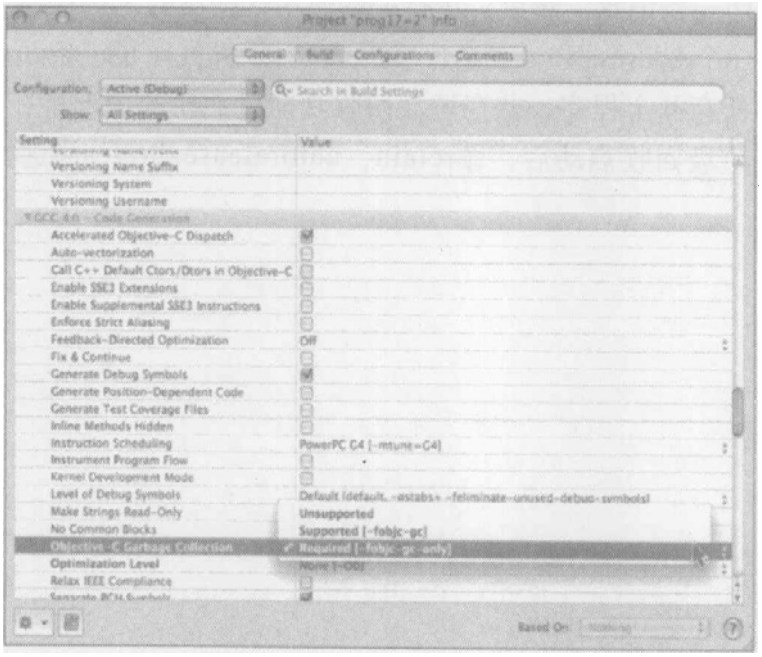

图17-1 启用垃圾回收功能

启用垃圾回收功能后，程序仍可进行retain、autorelease、release和dealloc方法调用。然而这些调用会被忽略。这样，可以开发同时能在内存管理和垃圾回收环境中运行的程序。然而这也意味着，如果需要代码能在这两种环境中运行，那么就不能完成dealloc方法中实现的所有功能。原因前面已经说过，启用垃圾回收后会忽略dealloc调用。

注意 本章介绍的内存管理技术对于大多数应用程序足够了。然而在更为复杂的情况中，如编写多线程应用程序时，可能还需要更多的技术。要想了解有关这些问题以及与垃圾回收有关的其他信息，请参见附录D“资源”。

17.6 练习

1. 编写一个程序，测试添加和移除字典中的条目时，对添加和移除对象的引用计数有什么

影响。

2. 你认为NSArray的replaceObjectAtIndex:withObject:方法对数组中被替换的对象的引用计数有什么影响？对放置到数组中的对象有什么影响呢？编写一个程序以进行测试，然后参考这个方法的文档来检验你的结果。
3. 返回第一部分“Objective-C 2.0语言”中一直使用的Fraction类，为了方便起见，这个类在附录D“资源”中列出，修改这个类，使它在Foundation框架下工作。然后，给各种MathOps分类的方法添加合适的消息，将每步操作产生的小数添加到自动释放池中。完成这些任务后，可以编写如下语句：

```
[[fractionA add: fractionB] print];
```

不会导致内存泄漏吗？解释你的答案。

4. 返回到第15章中的例子AddressBook和AddressCard。修改每个dealloc方法，使得当调用这个方法时打印一条消息。然后，运行一些使用这些类的示例程序来确保在主程序main结束之前给每一个AddressBook和 AddressCard对象都发送了一条dealloc消息。
5. 选择本书中的任意两个程序，并在垃圾回收功能已启用的情况下在Xcode中生成并运行它们。确定在垃圾回收启动后，像retain、autorelease和release这样的方法调用都被忽略了。

第18章 复制对象

本章将讨论复制对象相关的一些细节。我们将介绍浅复制和深复制的概念，并讨论如何在Foundation框架下实现对象复制。

在第8章“继承”中，我们讨论了使用简单的赋值语句将对象赋值给另一个对象时发生的情况，比如：

```
origin = pt;
```

在这个例子中，origin和pt都是XYPoint对象，其定义如下：

```
@interface XYPoint: NSObject
{
    int x;
    int y;
};
  ...
@end
```

还记得，这样赋值的结果仅仅是将对象pt的地址复制到origin中。在赋值操作结束时，两个变量都指向内存中的同一个地址。使用一条消息对实例变量进行修改，如：

```
[origin setX: 100 andY: 200];
```

改变了origin和pt变量都引用的XYPoint对象的x、y坐标，因为它们都引用内存中的同一个对象。

这同样适用于Foundation对象：将一个变量赋值给另一个对象仅仅创建另一个对这个对象的引用（但这并不像第17章“内存管理”中所讨论的，将增加引用计数）。所以，如果dataArray和dataArray2都是NSMutableArray对象，那么语句

```
dataArray2 = dataArray;
[dataArray2 removeObjectAtIndex: 0];
```

将从这两个变量引用的同一个数组中删除第一个元素。

18.1 copy和mutableCopy方法

Foundation类实现了名为copy和mutableCopy的方法，可以使用这些方法创建对象的副本。通过实现一个符合<NSCopying>协议（用于制作副本）的方法来完成此任务。如果类必须区分要产生对象的是可变副本还是不可变副本，还需要根据<NSMutableCopying>协议实现一个方法。本章后面将学习如何实现上述方法。

回顾Foundation类的copy方法，给定前面描述的两个NSMutableArray对象dataArray2和dataArray，语句

```
dataArray2 = [dataArray mutableCopy];
```

在内存中创建了一个新的dataArray副本，并复制了它的所有元素。随后，执行语句

```
[dataArray2 removeObjectAtIndex: 0];
```

删除了dataArray2中的第一个元素，但却不删除dataArray中的。代码清单18-1说明了这种情况。

代码清单18-1

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
      @"one", @ "two", @ "three", @ "four", nil];
   NSMutableArray *dataArray2;

   // simple assignment

   dataArray2 = dataArray;
   [dataArray2 removeObjectAtIndex: 0];

   NSLog (@"dataArray: ");
   for ( NSString *elem in dataArray )
      NSLog (@"%@", elem);

   NSLog (@"dataArray2: ");

   for ( NSString *elem in dataArray2 )
      NSLog (@"%@", elem);

   // try a Copy, then remove the first element from the copy

   dataArray2 = [dataArray mutableCopy];
   [dataArray2 removeObjectAtIndex: 0];

   NSLog (@"dataArray: ");

   for ( NSString *elem in dataArray )
      NSLog (@"%@", elem);

   NSLog (@"dataArray2: ");

   for ( NSString *elem in dataArray2 )
      NSLog (@"%@", elem);
```

```
    [dataArray2 release];
    [pool drain];
    return 0;
}
```

代码清单18-1 输出

```
dataArray:
    two
    three
    four
dataArray2:
    two
    three
    four
dataArray:
   two
   three
   four
dataArray2:
    three
    four
```

这个程序定义了可变数组对象dataArray，并分别将其元素设置为字符串对象@"one"、@"two"、@"three"和@"four"。

前面讨论过，赋值语句

```
dataArray2 = dataArray;
```

仅仅创建了对内存中同一数组对象的另一个引用。从dataArray2中删除第一个对象并随后输出两个数组对象中的元素，不出意料，这两个引用中的第一个元素（字符串@"one"）都会消失。

接下来，创建一个dataArray的可变副本并将它赋值给dataArray2的最终副本。这就在内存中创建了两个截然不同的可变数组，两者都包含三个元素。现在，删除dataArray2中的第一个元素时，不会对dataArray的内容有任何影响，正如程序输出的最后两行验证的一样。

注意，产生一个对象的可变副本并不要求被复制的对象本身是可变的。这种情况同样适用于不可变副本：可以创建可变对象的不可变副本。

还要注意在产生数组的副本时，数组中每个元素的保持计数将通过复制操作自动增1。因此，如果产生数组的副本并随即释放原始数组，那么副本仍然包含有效的元素。

因为dataArray的副本是在程序中使用mutableCopy方法产生的，所以你要负责释放它的内存。上一章讲述了你负责释放自己使用其中一个copy方法创建的对象这条规则。这解释了下面这个程序行

```
[dataArray2 release];
```

的实质，它位于代码清单18-1的结尾。

18.2 浅复制与深复制

代码清单18-1使用不可变字符串来填充dataArray的元素（回忆一下，常量字符串对象是不可变的）。在代码清单18-2中，将使用可变字符串代替它来填充数组，这样就可以改变数组中的一个字符串。观察代码清单18-2，看自己能否理解程序的输出结果。

代码清单18-2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSMutableArray    *dataArray = [NSMutableArray arrayWithObjects:
      [NSMutableString stringWithString: @ "one"],
      [NSMutableString stringWithString: @ "two"],
      [NSMutableString stringWithString: @ "three'],
      nil
   ];
   NSMutableArray    *dataArray2;
   NSMutableString   *mStr;

   NSLog (@"dataArray:    ");
   for ( NSString *elem in dataArray )
      NSLog (@"    %@", elem);

   // make a copy, then change one of the strings

   dataArray2 = [dataArray mutableCopy];


   mStr = [dataArray objectAtIndex: 0];
   [mStr appendString: @ "ONE"];

   NSLog (@"dataArray:    ");
   for ( NSString *elem in dataArray )
      NSLog (@"    %@", elem);

   NSLog (@"dataArray2:    ");
   for ( NSString *elem in dataArray2 )
      NSLog (@"    %@", elem);

   [dataArray2 release];
   [pool drain];
```

```
    return 0;
}
```

代码清单18-2 输出

```
dataArray:
    one
    two
    three
dataArray:
    oneONE
    two
    three
dataArray:
    oneONE
    two
    three
```

使用下面的语句检索dataArray的第一个元素：

```
mStr = [dataArray objectAtIndex: 0];
```

然后，使用下面的语句将字符串@“ONE”附加到给这个元素：

```
[mStr appendString: @'ONE"];
```

注意，原始数组及其副本中第一个元素的值：它们都被修改了。或许你能理解为什么dataArray的第一个元素发生改变，但不明白为什么它的副本也会改变。从集合中获取元素时，就得到了这个元素的一个新引用，但并不是一个新副本。所以，对dataArray调用objectAtIndex:方法时，返回的对象与dataArray中的第一个元素都指向内存中的同一个对象。随后，修改string对象mStr的副作用就是同时改变了dataArray的第一个元素，从输出结果中可以看到。

但你制作的副本怎么样了？为什么它的第一个元素也会改变？这与默认的浅复制方式有关。它意味着使用mutableCopy方法复制数组时，在内存中为新的数组对象分配了空间，并且将单个元素复制到新数组中。然而将原始数组中每个元素复制到新位置意味着：仅将引用从一个数组元素复制到另一个数组元素。这样做的最终结果，就是这两个数组中的元素都指向内存中的同一个字符串。这与将一个对象赋值给另一个对象没什么不同，就像我们本章开始提到的一样。

要为数组中每个元素创建完全不同的副本，需要执行所谓的深复制。这就意味着要创建数组中的每个对象内容的副本，而不仅是这些对象的引用的副本（并且考虑一下，如果一个数组中的元素本身是数组对象时，深复制意味着该如何处理）。然而使用Foundation类的copy或mutableCopy方法时，深复制并不是默认执行的。在第19章“归档”中，我们将向你展示如何使用Foundation的归档功能来创建对象的深复制。

例如，复制一个数组、字典或集时，会获得这些集合的新副本。然而，如果想要更改其中一个集合而不是它的副本，那么可能需要为单个元素创建自己的副本。例如，假设想要更改代码清单18-2中dataArray2的第一个元素，但不更改dataArray的第一个元素，可以创建一个新字符串（使用stringWithString:之类的方法）并将它存储到dataArray2的第一个位置，如下所示：

```
mStr = [NSMutableString stringWithString: [dataArray2 objectAtIndex: 0]];
```

然后，可以更改mStr，并使用replaceObject:atIndex:withObject:方法将它添加到数组中，如下所示：

```
[mStr appendString @"ONE"];
[dataArray2 replaceObjectAtIndex: 0 withObject: mStr];
```

如果顺利的话，你会发现即使替换了数组中的对象之后，mStr和dataArray2的第一个元素仍指向内存中的同一个对象。这意味着随后在程序中对mStr做的修改也将更改数组的第一个元素。如果这不是你想要的，则可以总是释放mStr，并分配新实例，因为对象会被replaceObject:atIndex:withObject:方法自动保持。

18.3 实现<NSCopying>协议

如果尝试使用自己类（例如，地址簿）中的copy方法，如下所示：

```
NewBook = [myBook mutableCopy];
```

将会收到一条出错消息，它可能如下所示：

```
*** -[AddressBook copyWithZone:]:  selector not recognized
*** Uncaught exception:
*** -[AddressBook copyWithZone:]: selector not recognized
```

正如注释的那样，要实现使用自己的类进行复制，必须根据<NSCoping>协议实现其中一两个方法。

我们将展示如何为Fraction类添加copy方法，这个类在第一部分“Objective-C 2.0语言”中广泛地使用。注意，这里描述的复制策略的技巧非常适用于你自己的类。如果这些类是任何Foundation类的子类，那么可能需要实现较为复杂的复制策略。必须考虑这样一个事实：超类可能已经实现了它自己的复制策略。

还记得Fraction类包含两个整型实例变量，分别名为numberator和denominator。要产生其中一个对象的副本，需要分配一个新分数的空间并简单地将这两个整数的值复制到新分数中。

实现<NSCopying>协议时，类必须实现copyWithZone:方法来响应copy消息。（这条copy消息仅将一条带有nil参数的copyWithZone:消息发送给你的类）。注意，如果想要区分可变副本和不可变副本，还需要根据<NSMutableCoping>协议实现mutableCopyWithZone:方法。如果两个方法都实现，那么copyWithZone:应该返回不可变副本，而mutableCopytWithZone:将返回可变副本。产生对象的可变副本并不要求被复制的对象本身也是可变的（反之亦然），想要产生不可变对象的可变副本是很合理的（例如，考虑字符串对象）。

@interface指令应该如下所示：

```
@interface Fraction: NSObject <NSCopying>
```

Fraction是NSObject的子类，并且符合NSCopying协议。

在实现文件Fracion.m中，为新方法添加下列定义：

```
-(id) copyWithZone: (NSZone *) zone
```

```
{
    Fraction *newFract = [[Fraction allocWithZone: zone] init];

    [newFract setTo: numerator over: denominator];
    return newFract;
}
```

参数zone与不同的存储区有关，你可以在程序中分配并使用这些存储区。只有在编写要分配大量内存的应用程序并且想要通过将空间分配分组到这些存储区中来优化内存分配时，才需要处理这些zone。可以使用传递给copyWithZone:的值，并将它传给名为allocWithZone:的内存分配方法。这个方法在指定存储区中分配内存。

分配新的Fraction对象之后，将接收者的numerator和denominator变量复制到其中。copyWithZone:方法应该返回对象的新副本，这个对象就是在你的方法中实现的。

代码清单18-3测试了你的新方法。

代码清单18-3

```
// Copying fractions

#import"Fraction.h"
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction *f1 = [[Fraction alloc] init];
    Fraction *f2;

    [f1 setTo: 2 over: 5];
    f2 = [f1 copy];

    [f2 setTo: 1 over: 3];

    [f1 print];
    [f2 print];

    [f1 release];
    [f2 release];
    [pool drain];
    return 0;
}
```

代码清单18-3 输出

```
2/5
1/3
```

该程序创建了一个名为f1的Fraction对象并将其设置为2/5。然后，它调用copy方法来产生副本，copy方法向你的对象发送copyWithZone:消息。这个方法产生了一个新Fraction，将f1的值复制到其中，并返回结果。回到main函数中，你将这个结果赋给f2。随即，将f2中的值设置为分数1/3，这样就验证了这些操作对原始分数f1没有影响。将程序中的这一行

```
f2 = [f1 copy];
```

简单地改成

```
f2 = f1;
```

并在程序结尾删除f2的release语句，观察获得的不同结果。

如果你的类可以产生子类，那么copyWithZone:方法将被继承。在这种情况下，该方法中的程序行

```
Fraction *newFract = [[Fraction allocWithZone: zone] init];
```

应该改为

```
Fraction *newFract = [[[self class] allocWithZone: zone] init];
```

这样，可以从该类分配一个新对象，而这个类是copy的接收者（例如，如果它产生了一个名为NewFraction的子类，那么应该确保在继承的方法中分配了新的NewFraction对象，而不是Fraction对象）。

如果编写一个类的copyWithZone:方法，而该类的超类也实现了<NSCopying>协议，那么应该先调用超类的copy方法以复制继承来的实例变量，然后加入自己的代码以复制想要添加到该类中的任何附加的实例变量（如果有的话）。

你必须确定是否在类中实现浅复制或深复制，并为其编写文档，以告知类的其他使用者。

18.4　用赋值方法和取值方法复制对象

只要实现赋值方法或取值方法，都应该考虑实例变量中存储的内容、要检索的内容以及是否需要保护这些值。例如，考虑使用setName:方法来设置AddressCard对象的名称时：

```
[newCard setName: newName];
```

假设newName是一个字符串对象，它包含新地址卡片的名称。假定在赋值方法例程内，你只是简单地将参数赋值给相应的实例变量：

```
-(void) setName: (NSString *) theName
{
    name = theName;
}
```

现在，如果随后更改了程序的newName中包含的一些字符，那么你认为会发生什么？对，这样也会无意间改变地址卡片中对应的域，因为它们都引用相同的string对象。

前面已经学过，为了避免无意间对程序产生影响，比较安全的途径是在赋值方法例程中产生对象的副本。通过使用alloc方法来创建新的string对象，然后使用initWithString:函数将该方法提供的参数值赋给它，可以实现在赋值对象中产生副本。

还可以编写如下的setName:方法来使用copy：

```
-(void) setName: (NSString *) theName
{
    name = [theName copy];
}
```

当然，要使赋值方法例程的内存管理友好一些，首先应该自动释放旧的值，如下所示：

```
-(void) setName: (NSString *) theName
{
    [name autorelease];
    name = [theName copy];
}
```

如果在属性声明中为实例变量指定了copy属性，那么合并后的方法会使用类的copy方法（你编写的copy方法或继承的copy方法）。所以下列property声明

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

会生成一个合并的方法，其行为类似于

```
-(void) setName: (NSString *) theName
{
    if (theName != name) {
        [name release]
        name = [theName copy];
    }
}
```

此处使用nonatomic是为了告诉系统不要使用mutex（互斥）锁定保护属性存取器方法。编写线程安全代码的人会使用mutex锁定来防止同一代码中的两个线程同时执行，如果同时执行了，会导致可怕的问题。然而这种锁定也会让程序变慢，如果知道这个代码只会在单线程中运行，就可避免使用这种锁定方法。

如果未指定nonatomic或者指定了atomic（这是默认值），那么会使用mutex锁定保护实例变量。另外，合并的取值方法将被保持，并在实例变量的值返回前自动释放该实例变量。在没有使用垃圾回收的环境中，这可保护实例变量不会被赋值方法调用所重写，这种赋值方法在设置新值前会释放实例变量的旧值。在取值方法中的这种保持确保了旧的值不会被销毁。

注意 虽然保持/自动释放问题与垃圾回收环境无关（还记得这些方法调用都会被忽略），然而Mutex锁定问题不是这样。因此，如果代码在多线程环境中运行时，要考虑使用atomic存取器方法。

关于保护实例变量值的讨论同样适用于取值例程。如果返回对象，那么必须确保对返回值的更改不影响你的实例变量的值。在这样的情况下，可以生成实例变量的副本，并将它用作返回值。

回到copy方法的实现，如果正在复制的实例变量包含不可变的对象（如，不可变的string对象），那么可能不需要生成这个对象内容的新副本。仅通保持该对象来生成它的新引用，可能

就足够了。例如，为AddressCard类实现copy方法时，这个类包含name和email成员，下面实现的copyWithZone:方法就足够了：

```
-(AddresssCard *) copyWithZone: (NSZone *) zone
{
    AddressCard *newCard = [[AddressCard allocWithZone: zone] init];
    [newCard retainName: name andEmail: email];
    return newCard;
}
-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail
{
    name = [theName retain];
    email = [theEmail retain];
}
```

这里没有使用setName:andEmail:方法来复制实例变量，因为这个方法生成了它的参数的新副本，这将违反整个练习的目的。可替换为，只需使用一个称为retainName:andEmail：的新方法保持两个变量（可以直接在copyWithZone:方法中设置newCard中的两个实例变量，然而这涉及指针操作，而目前我们要避免使用指针。当然指针操作可能效率更高，并且不会将方法[retainName:andEmail:]暴露给类的用户，这个[retainName:andEmail:]方法原义并不是public使用的，因此有些时候，可能需要学习如何这样使用它——但不是现在！）

认识到这里可以侥幸成功地保持实例变量（而不是产生它们的完整副本），因为被复制的地址卡的所有者不能影响原始对象的name和email成员。可能想要仔细地想一下，以验证这个例子（提示：这与赋值方法有关）。

18.5 练习

1. 根据NSCopying协议为AddressBook类实现一个copy方法。也实现mutableCopy方法行不行得通？为什么行或者为什么不行？
2. 修改第8章中定义Rectangle类和XYPoint类，使其符合<NSCopying>协议的要求。然后为每个类添加copyWithZone:方法。确保Rectangle使用XYPoint的copy方法复制它的XYPoint成员origin。为这些类实现可变和不可变副本是否行得通？解释原因。
3. 创建一个NSDictionary字典对象，并使用键/对象对来填充它，然后产生可变和不可变副本。这些复制是深复制还是浅复制？验证你的答案。
4. 谁负责释放本章在copyWithZone：方法中为新AddressCard对象分配的内存？为什么？

第19章 归 档

在Objective-C术语中，归档是指用某种格式来保存一个或多个对象，以便以后还原这些对象的过程。通常，这个过程包括将（多个）对象写入文件中，以便以后读回该对象。我们将在本章讨论两种归档数据的方法：属性列表和带键值的编码。

19.1 使用XML属性列表进行归档

Mac OS X上的应用程序使用XML属性列表（或plists）来存储诸如默认参数选择、应用程序设置和配置信息这样的数据，因此，了解如何创建和读回这些数据很有用。然而，这些列表的归档用途是有限的，因为当为某个数据结构创建属性列表时，没有保存特定的对象类，没有存储对同一对象的多个引用，也没有保持对象的可变性。

注意 所谓的“老式”属性列表存储数据的格式与XML属性列表不同。如果可能，尽量在程序中使用XML属性列表。

如果你的对象是NSString、NSDictionary、NSArray、NSData或NSNumber对象，你可以使用在这些类中实现的writeToFile:atomically：方法将数据写到文件中。在写出某个字典或数组的情况下，该方法可以使用XML属性列表的格式写出数据。代码清单19-1显示了如何将在第15章“数字、字符串和集合”中作为简易术语表而创建的字典作为属性列表写入文件中。

代码清单19-1

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>
int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSDictionary *glossary =
   [NSDictionary dictionaryWithObjectsAndKeys:
      @"A class defined so other classes can inherit from it.", @ "abstract class",
      @"To implement all the methods defined in a protocol", @ "adopt",
      @'Storing an object for later use.", @ "archiving",
      nil
   ];
   if ([glossary writeToFile: @ "glossary "atomically: YES] == NO)
      NSLog (@ "Save to file failed! ");
   [pool drain];
   return 0;
}
```

writeToFile:atomically:encoding:error:消息被发送给字典对象glossary，使字典以属性列表的形式写到文件glossary中。atomically参数被设为YES，表示希望首先将字典写入临时备份文件中，并且一旦成功，将把最终数据转移到名为glossary的指定文件中。这是一种安全措施，它保护文件在一些情况下（如系统在执行操作的过程中崩溃时）免受破坏。在这种情况下，原始的glossary文件（如果该文件已存在）不会受到损害。

如果查看代码清单19-1中创建的glossary文件，它的内容可能如下：

```
<?xml version= "1.0"encoding= 'UTF-8'?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN "
              "http://www.apple.com/DTDs/PropertyList-1.0.dtd ">
<plist version= "1.0">
<dict>
    <key>abstract class</key>
    <string>A class defined so other classes can inherit from it.</string>
    <key>adopt</key>
    <string>To implement all the methods defined in a protocol</string>
    <key>archiving</key>
    <string>Storing an object for later use. </string>
</dict>
</plist>
```

从所创建的XML文件中可以看到，是以一种键（<key>...</key>）值（<string>...</string >）对的形式将字典写入文件的。

根据字典创建属性列表时，字典中的键必须全都是NSString对象。数组中的元素或字典中的值可以是NSString、NSArray、NSDictionary、NSData、NSDate或NSNumber对象。

要将文件中的XML属性列表读入你的程序，使用dictionaryWithContentsOfFile:或arrayWithContentsOfFile:方法。要读回数据，使用dataWithContentsOfFile:方法，要读回字符串对象，使用stringWithContentsOfFile:方法。代码清单19-2读回了代码清单19-1中编写的术语表，然后输出其内容。

代码清单19-2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSEnumerator.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSDictionary *glossary;

   glossary = [NSDictionary dictionaryWithContentsOfFile: @ "glossary"];

   for ( NSString *key in glossary )
```

```
    NSLog (@ "%@: %@ ", key, [glossary objectForKey: key]);

    [pool drain];
    return 0;
}
```

代码清单19-2 输出

```
archiving: Storing an object for later use.
abstract class: A class defined so other classes can inherit from it.
adopt: To implement all the methods defined in a protocol
```

你的属性列表不必从Objective-C程序中创建，属性列表可以来自任何的源。可以使用简单的文本编辑器，或使用Mac OS X系统中位于/Developer/Applications/Utilities目录下的Property List Editor程序来创建属性列表。

19.2 使用NSKeyedArchiver归档

将各种类型的对象存储到文件中，而且不仅仅是字符串、数组和字典类型，有一种更灵活的方法。就是利用NSKeyedArchiver类创建带键（keyed）的档案来完成。

Mac OX X从版本10.2开始支持带键的档案。在此之前，要使用NSArchiver类创建连续的（sequential）归档。连续的归档需要完全按照写入时的顺序读取归档中的数据。

在带键的档案中，每个归档字段都有一个名称。归档某个对象时，会为它提供一个名称，即键。从归档中检索该对象时，是根据这个键来检索它的。这样，可以按照任意的顺序将对象写入归档并进行检索。另外，如果向类添加了新的实例变量或删除了实例变量，程序也可以进行处理。

注意，iPhone SDK中没有提供NSArchiver。如果想在iPhone上使用归档功能，则必须使用NSKeyedArchiver。

要想使用带键的档案，需要导入

```
<Foundation/NSKeyedArchiver.h>.
```

代码清单19-3展示了如何使用NSKeyedArchiver类中的archiveRootObject:toFile:方法将glossary存储到磁盘上。要使用该类，在你的程序中包含以下文件

```
#import <Foundation/NSKeyedArchiver.h>
```

代码清单19-3

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
```

```
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSDictionary *glossary =
      [NSDictionary dictionaryWithObjectsAndKeys:
        @"A class defined so other classes can inherit from it",
        @"abstract class",
        @"To implement all the methods defined in a protocol',
        @"adopt',
        @'Storing an object for later use',
        @"archiving",
        nil
      ];

    [NSKeyedArchiver archiveRootObject: glossary toFile: @ "glossary.archive"];

    [pool release];
    return 0;
}
```

代码清单19-3并不在终端产生任何输出。但是，语句

```
[NSKeyedArchiver archiveRootObject: glossary toFile: @"glossary.archive"];
```

将字典glossary写入文件glossary.archive中。可以为该文件指定任何路径名。在本例中，文件被写入当前目录下。

以后通过NSKeyedUnarchiver的unArchiveObjectWithFile:方法将创建的归档文件读入执行程序中，如代码清单19-4所示。

代码清单19-4

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSEnumerator.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSDictionary *glossary;

    glossary = [NSKeyedUnarchiver unarchiveObjectWithFile:
                    @"glossary.archive"];

    for ( NSString *key in glossary )
            NSLog (@ "%@: %@", key, [glossary objectForKey: key]);
```

```
    [pool drain];
    return 0;
}
```

代码清单19-4 输出

```
abstract class: A class defined so other classes can inherit from it.
adopt: To implement all the methods defined in a protocol
archiving: Storing an object for later use.
```

语句

```
glossary = [NSKeyedUnarchiver unarchiveObjectWithFile: @"glossary.archive"];
```

将指定的文件打开并读取文件的内容。该文件必须是前面归档操作的结果。可以为文件指定完整路径名或相对路径名，如本例所示。

在恢复glossary之后，程序可以简单地通过枚举其内容来验证恢复是否成功。

19.3 编码方法和解码方法

可以使用刚刚描述的方式归档和恢复NSString、NSArray、NSDictionary、NSSet、NSDate、NSNumber和NSData之类的基本Objective-C类对象。这还包括嵌套的对象，如包含字符串，甚至其他数组对象的数组。

这意味着不能直接使用这种方法归档AddressBook，因为Objective-C系统不知道如何归档AddressBook对象。如果在程序中插入如下一行来尝试归档它：

```
[NSKeyedArchiver archiveRootObject: myAddressBook toFile: @"addrbook.arch"];
```

运行该程序时将会得到以下所示的消息：

```
*** -[AddressBook encodeWithCoder:]: selector not recognized
*** Uncaught exception: <NSInvalidArgumentException>
*** -[AddressBook encodeWithCoder:]: selector not recognized
archiveTest: received signal: Trace/BPT trap
```

从这些出错消息中，可以看到系统正在AddressBook类中查找一个名为encodeWithCoder:的方法，但你从未定义过这样的方法。

要归档前面没有列出的对象，必须告知系统如何归档（或编码）你的对象，以及如何解归档（或解码）它们。这是按照<NSCoding>协议，在类定义中添加encodeWithCoder:方法和initWithCoder:方法实现的。对于我们地址簿的例子，必须向AddressBook类和AddressCard类添加这些方法。

每次归档程序想要根据指定类编码对象时，都将调用encodeWithCoder:方法，该方法告知归档程序如何进行归档。类似地，每次从指定的类解码对象时，就会调用initWithCoder:方法。

一般而言，编码方法应该指定如何归档想要保存的对象中的每个实例变量。幸运的是，这些都有帮助可查。对于前面描述的基本Objective-C类，可以使用encodeObject:forKey:方法。相反，对于基本的C数据类型（如整型和浮点型），可以使用表19-1中列出的某种方法。解码方法

initWithCoder:的工作方式正好相反：它使用decodeObject:forKey:来解码基本的Objective-C类，使用19-1中列出的相应解码方法来解码基本的数据类型。

表19-1　在带键的档案中编码和解码基本数据类型

编码方法	解码方法
encodeBool:forKey:	decodeBool:forKey:
encodeInt:forKey:	decodeInt:forKey:
encodeInt32:forKey:	decodeInt32:forKey:
encodeInt64: forKey:	decodeInt64:forKey:
encodeFloat:forKey:	decodeFloat:forKey:
encodeDouble:forKey:	decodeDouble:forKey:

代码清单19-5为AddressCard类和AddressBook类都添加了两个编码和解码方法。

代码清单19-5　Addresscard.h接口文件

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>

@interface AddressCard: NSObject <NSCoding, NSCopying>
{
   NSString *name;
   NSString *email;
}

@property (copy, nonatomic) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;
-(void) print;

// Additional methods for NSCopying protocol
-(AddressCard *) copyWithZone: (NSZone *) zone;
-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail;

@end
```

下面是要添加到AddressCard类实现文件的两个新方法：

```
-(void) encodeWithCoder: (NSCoder *) encoder
{
   [encoder encodeObject: name forKey: @ "AddressCardName"];
   [encoder encodeObject: email forKey: @ "AddressCardEmail"];
}

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

```
    name = [[decoder decodeObjectforKey: @ "AddressCardName"] retain];
    email = [[decoder decodeObjectforKey: @ "AddressCardEmail"] retain];

    return self;
}
```

该程序向编码方法encodeWithCoder:传入一个NSCoder对象作为参数。由于AddressCard类直接继承自Nsobject，所以无需担心编码继承的实例变量。如果的确担心，并且知道类的子类符合NSCoding协议的要求，那么应该用下面的语句开始编码方法，确保继承的实例变量也被编码：

```
[super encodeWithCoder: encoder];
```

对于地址簿来说，有两个名为name和email的实例变量。因为它们都是NSString对象，所以使用encodeObject:forKey:方法依次对它们进行编码，然后将这两个实例变量添加到归档文件中。

encodeObject:forKey:方法编码对象并将其存储在指定的键下，以后可使用该键检索对象。键名是任意的，所以只要在检索（编码）数据时使用的名称与归档（编码）时使用的名称相同，就可以指定任何键名。唯一可能出现冲突的情况是，为正在编码的对象子类使用了相同的键。为了防止这种情况出现，制订归档的键时，可将类名放在实例变量名的前面，代码清单19-5就是这样做的。

注意，encodeObject:ForKey:方法可以用于任何在其类中实现对应encodeWithCoder:方法的对象。

解码的过程刚好相反。传递给initWithCoder:的参数也是NSCoder对象。不必担心这个参数，只要记住它是获得该消息（对于每个想要从归档文件中提取的对象）的对象。

同样，由于AddressCard类直接继承自NSObject，所以不必担心解码继承的实例变量。如果的确担心，那么应在解码方法的开始插入下列行（假设类的超类符合NSCoding协议的要求）：

```
self = [super initWithCoder: decoder];
```

通过调用decodeObject:ForKey:方法并传递在编码变量时使用的相同键，就可解码每个实例变量。

类似于AddressCard类，你为AddressBook类添加了两个编码和解码方法。在接口文件中只需更改@interface指令，以声明现在AddressBook类已经符合NSCoding协议。更改如下所示：

```
@interface AddressBook: NSObject <NSCoding, NSCopying>
```

下面是实现文件中所含的方法定义：

```
-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject: bookName forKey: @ "AddressBookBookName"];
    [encoder encodeObject: book forKey: @ "AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    bookName = [[decoder decodeObjectForKey: @ "AddressBookBookName"] retain];
```

```
    book = [[decoder decodeObjectForKey: @ "AddressBookBook"] retain];

    return self;
}
```

以下代码清单19-6是它的测试程序。

代码清单19-6　测试程序

```
#import "AddressBook.h"
#import <Foundation/NSAuoreleaePool.h>

int main (int argc, char *argv[])
{

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSString *aName = @ "Julia Kochan";
    NSString *aEmail = @ "jewls337@axlc.com";
    NSString *bName = @ "Tony Iannino";
    NSString *bEmail = @ "tony.iannino@techfitness.com";
    NSString *cName = @ 'Stephen Kochan";
    NSString *cEmail = @ "steve@steve_kochan.com";
    NSString *dName = @ "Jamie Baker";
    NSString *dEmail = @ "jbaker@hitmail.com';

    AddressCard *card1 = [[AddressCard alloc] init];
    AddressCard *card2 = [[AddressCard alloc] init];
    AddressCard *card3 = [[AddressCard alloc] init];
    AddressCard *card4 = [[AddressCard alloc] init];

    AddressBook *myBook = [AddressBook alloc];

    // First set up four address cards

    [card1 setName: aName andEmail: aEmail];
    [card2 setName: bName andEmail: bEmail];
    [card3 setName: cName andEmail: cEmail];
    [card4 setName: dName andEmail: dEmail];

    myBook = [myBook initWithName: @ "Steve's Address Book"];
    // Add some cards to the address book

    [myBook addCard: card1];
    [myBook addCard: card2];
    [myBook addCard: card3];
    [myBook addCard: card4];

    [myBook sort];
```

```
    if ([NSKeyedArchiver archiveRootObject: myBook toFile: @ "addrbook.arch"] == NO)
        NSLog (@ "archiving failed");

    [card1 release];
    [card2 release];
    [card3 release];
    [card4 release];
    [myBook release];

    [pool drain];
    return 0;
}
```

这个程序创建了一个地址簿，然后将它归档到文件addrbook.arch中。在创建归档文件的过程中，注意AddressBook类和AddressCard类中的编码方法都被调用了。如果想要验证，可以向这些方法添加一些NSLog调用。

代码清单19-7展示了如何将归档读入内存以根据文件创建地址簿。

代码清单19-7

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
    AddressBook         *myBook;
    NSAutoreleasePool   * pool = [[NSAutoreleasePool alloc] init];

    myBook = [NSKeyedUnarchiver unarchiveObjectWithFile: @ "addrbook.arch"];

    [myBook list];

    [pool drain];
    return 0;
}
```

代码清单19-7 输出

```
======== Contents of: Steve's Address Book =========
Jamie Baker         jbaker@hitmail.com
Julia Kochan        jewls337@axlc.com
Stephen Kochan      steve@steve_kochan.com
Tony Iannino        tony.iannino@techfitness.com
================================================
```

在解码地址簿的过程中，自动调用向两个类添加的解码方法。注意将地址薄读回程序是多么容易。

前面说过，encodeObject:forKey:方法作用于内置类以及根据NSCoding协议为其编写编码和解码方法的类。如果你的实例包含基本数据类型，如整型或浮点型，那么需要知道如何对它们进行编码和解码（参见表19-1）。

下面是一个类的简单定义，这个类名为Foo，它包含三个实例变量：一个是NSString类型，一个int型，一个float型。这个类包含一个赋值方法、三个取值方法以及两个用于归档的编码/解码方法：

```
@interface Foo: NSObject <NSCoding>
{
   NSString *strVal;
   int      intVal;
   float    floatVal;
}

@property (copy, nonatomic) NSString *strVal;
@property int intVal;
@property float floatVal;
@end
```

实现文件如下：

```
@implementation Foo

@synthesize strVal, intVal, floatVal;

-(void) encodeWithCoder: (NSCoder *) encoder
{
   [encoder encodeObject: strVal forKey: @ "FoostrVal"];
   [encoder encodeInt: intVal forKey: @ "FoointVal"];
   [encoder encodeFloat: floatVal forKey: @ "FoofloatVal"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
   strVal = [[decoder decodeObjectForKey: @ "FoostrVal"] retain];
   intVal = [decoder decodeIntForKey: @ "FoointVal"];
   floatVal = [decoder decodeFloatForKey: @ "FoofloatVal"];

   return self;
}
@end
```

编码例程首先使用前面用过的encodeObject:forKey:对象来编码字符串值strVal，如上面内容所示。

在代码清单19-8中，我们创建了一个Foo对象，把它归档到一个文件，解归档，然后显示。

代码清单19-8 测试程序

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>
#import "Foo.h "// Definition for our Foo class

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Foo *myFoo1 = [[Foo alloc] init];
   Foo *myFoo2;

   [myFoo1 setStrVal: @ "This is the string"];
   [myFoo1 setIntVal: 12345];
   [myFoo1 setFloatVal: 98.6];

   [NSKeyedArchiver archiveRootObject: myFoo1 toFile: @ "foo.arch"];

   myFoo2 = [NSKeyedUnarchiver unarchiveObjectWithFile: @ "foo.arch"];
   NSLog (@ "%@\n%i\n%g", [myFoo2 strVal], [myFoo2 intVal], [myFoo2 floatVal]);
   [myFoo1 release];
   [pool drain];
   return 0;
}
```

代码清单19-8 输出

```
This is the string
12345
98.6
```

以下消息归档了对象的3个实例变量：

```
[encoder encodeObject: strVal forKey: @ "FoostrVal"];
[encoder encodeInt: intVal forKey: @ "FoointVal"];
[encoder encodeFloat: floatVal forKey: @ "FoofloatVal"];
```

一些基本数据类型，如char、short、long和long long在表19-1中没有列出。你必须确定数据对象的大小并使用相应的例程。例如，short int通常是16位的，而int和long可以是32位或64位，long long是64位的（可以使用第13章介绍的sizeof运算符确定任何数据类型的大小）。所以要归档short int的数据，首先将其存储在int中，然后使用encodeInt:forKey:归档它。反向执行该过程可恢复它：使用decodeIntForKey:，然后将其赋值给short int变量。

19.4 使用NSData创建自定义档案

有时可能不想和前面示例程序一样，使用archiveRootObject:ToFile:方法将对象直接写入文件。

比如，可能想收集一些或全部对象，并将其存储到单个档案文件中。在Objective-C中，通过使用名为NSData的通用数据流对象类，可以实现上述功能，在第16章，我们简单地提到过这个类。

正如第16章所提到的，NSData对象可以用来保留一块内存空间以备后来存储数据。这些数据空间的典型应用是作为一些数据的临时存储空间，如随后将被写入文件，或可能用于容纳从磁盘读取的文件内容。创建可变数据空间的最简单方式是使用data方法：

```
dataArea = [NSMutableData data];
```

该语句创建一个空缓冲区，其大小随程序执行需要而扩展。

作为一个简单的例子，假设你想将地址簿和一个Foo对象归档到同一个文件。假设对于这个例子，你已经向AddressBook和AddressCard类添加了一个带键的归档方法（参见代码清单19-9）。

代码清单19-9

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSCoder.h>
#import <Foundation/NSData.h>
#import "AddressBook.h"
#import "Foo.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Foo               *myFoo1 = [[Foo alloc] init];
   Foo               *myFoo2;
   NSMutableData     *dataArea;
   NSKeyedArchiver   *archiver;
   AddressBook       *myBook;

   // Insert code from Program 19.7 to create an Address Book
   // in myBook containing four address cards

   [myFoo1 setStrVal: @ "This is the string"];
   [myFoo1 setIntVal: 12345];
   [myFoo1 setFloatVal: 98.6];

   // Set up a data area and connect it to an NSKeyedArchiver object
   dataArea = [NSMutableData data];

   archiver = [[NSKeyedArchiver alloc]
             initForWritingWithMutableData: dataArea];
   // Now we can begin to archive objects
   [archiver encodeObject: myBook forKey: @ "myaddrbook"];
   [archiver encodeObject: myFoo1 forKey: @ "myfoo1"];
```

```
    [archiver finishEncoding];

    // Write the archived data area to a file
    if ( [dataArea writeToFile: @ "myArchive" atomically: YES] == NO
         NSLog (@ "Archiving failed!  );

    [archiver release];

    [myPool release];
    [pool drain];
    return 0;
}
```

分配一个NSKeyedArchiver对象之后，发送initForWritingWithMutableData:消息，以指定要写入归档数据的存储空间。这就是前面创建的NSMutabledata空间dataArea。此时，就可以向存储在archiver中的NSKeyedArchiver对象发送编码消息，以归档该程序中的对象。实际上，所有编码消息在收到finishEncoding消息之前都被归档并存储在指定的数据空间内。

这里，有两个对象需要编码——一个是地址簿，另一个是Foo对象。对于这些对象可以使用encodeObject:forKey:，因为在前面你已经为AddressBook、AddressCard和Foo类实现了编码方法和解码方法（理解这个概念很重要）。

在归档这两个对象时，向archiver对象发送一条finishEncoding消息。之后，就不能编码其他对象，你需要发送此消息以完成归档过程。

此时，你预留的那块名为dataArea的空间包含归档对象，这些对象可以一种可写入文件的格式存在。消息表达式

```
[dataArea writeToFile: @"myArchive" atomically: YES]
```

向你的数据流发送writeToFile:atomically:encoding:error:消息，请求它把它的数据写入指定文件，这个文件名为myArchive。

从if语句可以看到，writeToFile:atomically:encoding:error:方法返回一个BOOL值：如果写操作成功就返回YES，如果失败（可能是指定了无效的路径名或文件系统已满）就返回NO。

从档案文件中恢复数据很简单——所做的工作只需和归档文件相反。首先，需要像以前那样分配一个数据空间。然后，把档案文件中的数据读入该数据空间，然后，需要创建一个NSKeyedUnarchiver对象，并告知它从指定空间解码数据。必须调用解码方法来提取和解码归档的对象，做完之后，向NSKeyedUnarchiver对象发送一条finishDecoding消息。

代码清单19-10实现了所有任务。

代码清单19-10

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSCoder.h>
```

```
#import <Foundation/NSData.h>
#import "AddressBook.h"
#import "Foo.h"

int main (int argc, char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSData            *dataArea;
   NSKeyedUnarchiver *unarchiver;
   Foo               *myFoo1;
   AddressBook       *myBook;
   // Read in the archive and connect an
   // NSKeyedUnarchiver object to it

   dataArea = [NSData dataWithContentsOfFile: @ "myArchive"];
   if (! dataArea) {
      NSLog (@ "Can't read back archive file!");
      return (1);
   }

   unarchiver = [[NSKeyedUnarchiver alloc]
                        initForReadingWithData: dataArea];

   // Decode the objects we previously stored in the archive
   myBook = [unarchiver decodeObjectForKey: @ 'myaddrbook"];
   myFoo1 = [unarchiver decodeObjectForKey: @ "myfoo1"];

   [unarchiver finishDecoding];

   [unarchiver release];

   // Verify that the restore was successful
   [myBook list];
   NSLog ( "%@\n%i\n%g", [myFoo1 strVal],
           [myFoo1 intVal], [myFoo1 floatVal]);

   [pool release];
   return 0;
}
```

代码清单19-10 输出

```
======== Contents of: Steve's Address Book =========
Jamie Baker       jbaker@hitmail.com
Julia Kochan      jewls337@axlc.com
Stephen Kochan    steve@steve_kochan.com
Tony Iannino      tony.iannino@techfitness.com
============================================================
```

```
This is the string
12345
98.6
```

输出结果验证了你的地址簿和Foo对象已成功地从档案文件中恢复了。

19.5 使用归档程序复制对象

在代码清单18-2中，尝试创建包含可变字符串元素的数组副本，并且了解了如何进行浅复制。也就是，没有复制实际的字符串本身，只是复制对它们的引用。

可以使用Foundation的归档能力来创建对象的深复制。例如，代码清单19-11通过dataArray归档到一个缓冲区，然后把它解归档，将结果指派给dataArray2，从而将dataArray复制给dataArray2。对于这个过程，不需要使用文件。归档和解归档过程都可以在内存中发生。

代码清单19-11

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSArray.h>

int main (int argc, char *argv[])
{
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   NSData       *data;
   NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
        [NSMutableString stringWithString: @ "one"],
        [NSMutableString stringWithString: @ "two"],
        [NSMutableString stringWithString: @ "three"],
        nil
   ];

   NSMutableArray *dataArray2;
   NSMutableString *mStr;

   // Make a deep copy using the archiver

   data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
   dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];

   mStr = [dataArray2 objectAtIndex: 0];
   [mStr appendString: @ "ONE"];

   NSLog (@ "dataArray: ");
   for ( NSString *elem in dataArray )
      NSLog (@ "%@", elem);
```

```
    NSLog (@ "\ndataArray2: ");
    for ( NSString *elem in dataArray2 )
        NSLog (@ "%@", elem);

    [pool drain];
    return 0;
}
```

代码清单19-11　输出

```
dataArray:
one
two
three

dataArray2:
oneONE
two
three
```

这个输出结果验证了更改dataArray2的第一个元素对dataArray的第一个元素并没有影响，这是因为在归档和解归档过程中产生的是字符串的新副本。

代码清单19-11中的复制操作是通过以下两行实现的：

```
data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];
```

甚至可以避免中间赋值，只用一条语句来执行复制，如下所示：

```
dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData:
                  [NSKeyedArchiver archivedDataWithRootObject: dataArray]];
```

下次需要生成一个对象（或不支持NSCopying协议的对象）的深复制时，应该记住这项技术。

19.6　练习

1. 在第15章，代码清单15-7生成了一个素数表。修改此程序，将结果数组作为XML属性列表写入文件primes.pl中。然后，检查该文件的内容。
2. 编写一个程序，该程序从练习1中创建的XML属性列表中读出数据，并将它们的值存储到一个数组对象。打印这个数组的所有元素，以验证存储操作成功。
3. 修改代码清单19-2，显示/Library/Preferences文件夹中存储的某个XML属性列表（.plist文件）的内容。
4. 写一个程序，使它读取已归档的AddressBook，并根据在命令行提供的名称进行查找，如：

```
$ lookup gregory
```

第三部分　Cocoa和iPhone SDK

第20章　Cocoa简介

本书中开发的所有程序都具有简单的用户界面，并依靠NSLog例程在控制台上显示消息。这个例程虽然有用，但其功能十分有限。毫无疑问，Mac上使用的其他程序都是用户友好的。事实上，Mac的良好声誉正是建立在其用户友好的对话框和易用性上。幸运的是，XCode和Interface Builder应用程序的组合可以满足这个要求。这种组合不仅提供了一个包含编辑和调试工具的强大程序开发环境，可以方便地访问在线文档，而且还提供了一个可轻松开发复杂图形用户界面（GUI）的环境。

Cocoa是一种支持应用程序提供丰富用户体验的框架，它实际上由两个框架组成：你已经熟知的Foundation框架，以及Application Kit（或AppKit）框架。后者提供与窗口、按钮、列表等相关的类。

20.1　框架层

一般使用示意图来说明最顶层应用程序与底层硬件之间的各个层次，如图20-1所示。

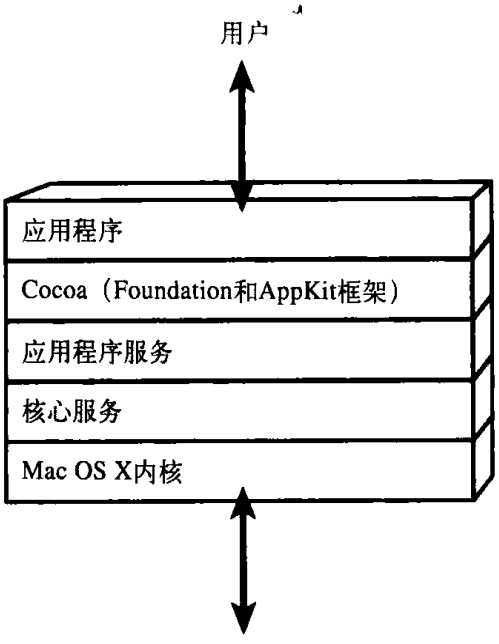

图20-1　应用程序层次结构

内核以设备驱动程序的形式提供与硬件的底层通信。它负责管理系统资源，包括调度要执行的程序、管理内存和电源，以及执行基本的I/O操作。

顾名思义，核心服务提供的支持比它上面层次更加底层或更加“核心”。例如，这里提供对集合、网络、调试、文件管理、文件夹、内存管理、线程、时间和电源的管理。

应用程序服务层包含对打印和图形呈现的支持，包括Quartz、OpenGL和Quicktime。

Cocoa层直接位于应用程序层之下。正如图20-1所示，Cocoa包括Foundation和AppKit框架。Foundation框架提供的类用于处理集合、字符串、内存管理、文件系

统、存档等。AppKit框架提供的类用于管理视图、窗口、文档和让Mac OS X闻名于世的多信息用户界面。

根据上面的描述，有些层之间的功能似乎有重复。Cocoa层和核心服务层中都存在集合。然而，后者是前者的基础。此外，在某些情况下也可以绕过某一层。例如，有些Foundation类，比如处理文件系统的那些类，直接依赖于核心服务层中的功能，实际上绕过了应用程序服务层。很多情况下，Foundation框架为底层核心服务层（主要用过程化的C语言编写）中定义的数据结构定义了一种面向对象的映射。

20.2 接触Cocoa

iPhone包含一台运行Mac OS X缩小版本的计算机。iPhone硬件中的有些功能，比如它的加速器，是电话中独一无二的，而且在其他Mac OS X计算机（如MacBook或iMac）中也找不到。

注意　Mac笔记本电脑实际上包含一个加速器，用于当计算机跌落到地上时对硬盘进行保护，但不能直接从程序中访问这个加速器。

Cocoa框架用于Mac OS X桌面与笔记本电脑的应用程序开发，而Cocoa Touch框架用于iPhone与ipod Touch的应用程序开发。

Cocoa和Cocoa Touch都有Foundation框架。然而在Cocoa Touch下，UIKit代替了AppKit框架，以便为很多相同类型的对象提供支持，比如窗口、视图、按钮、文本域等。另外，Cocoa Touch还提供使用加速器（它与GPS和WiFi信号一样都能跟踪你的位置）的类和触摸式界面，并且去掉了不需要的类，比如支持打印的类。

对于Cocoa概况的简介到此结束。在下一章中，你将了解如何使用iPhone SDK包含的模拟器为iPhone编写应用程序。

第21章　编写iPhone应用程序

本章将会开发两个简单的iPhone应用程序。第一个应用程序说明了一些基础性概念，目的是让你熟悉如何使用Interface Builder、建立连接，并理解委托、出口和操作。第二个应用程序是一个分数计算器，它很好地结合了你在开发第一个应用程序的过程中所学的知识和本书余下部分中学到的知识。

21.1　iPhone SDK

要编写iPhone应用程序，必须先安装Xcode和iPhone SDK。这个SDK可以从Apple的Web站点上免费下载。下载之前，你需要首先注册成为Apple开发人员，这个过程同样也是免费的。为了找到正确的链接，你可以从developer.apple.com开始访问，然后导航到正确的地点。熟悉该站点是一个明智的选择。附录D中列出了一些链接，它们直接指向该站点上你可能感兴趣的特定位置。

本章中的讨论基于Xcode 3.1.1和iPhone SDK for iPhone OS 2.1。它们的新版本与我们描述的内容也是兼容的。

21.2　第一个iPhone应用程序

第一个应用程序说明了如何在iPhone的屏幕上放置一个黑色窗口，只要用户按下一个按钮，该窗口中就会显示一些文本。

> **注意**　第二个应用程序更有趣！你需要使用从第一个应用程序中学到的知识构建一个进行分数运算的简单计算器。你可以使用本书中早先用过的Fraction类，以及一个经过修改的Calculator类。这次，你的计算器要知道如何处理分数。

让我们开始探讨第一个程序。本章不会面面俱到，因为篇幅不允许这么做。相反，我们会依次讲解每个步骤，从而让你拥有必要的基础知识，能够通过一些单独的Cocoa或iPhone程序设计介绍来自主探讨和学习更多的概念。

图21-1显示了要开发的第一个iPhone应用程序，它运行在iPhone模拟器（很快就会详细介绍）上。

这个应用程序的设计目的是：按下标为“1”的按钮时，显示屏上就会出现相应的数字（参见图21-2）。这就是它的全部功能！这个简单的应用程序为第二个分数计算器应用程序打下了基础。

你将使用Xcode创建这个应用程序，并使用Interface Builder创建用户界面。本书写到这里，如果一直使用Xcode输入和测试程序，你应该对它已经相当熟悉了。如前所述，Interface Builder这个工具通过将各种UI元素（如表、标签和按钮）放入组成iPhone屏幕的窗口中，从而设计出用户界面。与任何其他强大的开发工具一样，使用Interface Builder时也有一个习惯的过程。

Apple在iPhone SDK中发布了一个iPhone模拟器。该模拟器复制了iPhone环境的大部分，包括它的主屏幕、Safari Web浏览器、Contacts应用程序等。该模拟器使调试应用程序变得更加轻松，因为进行调试时，不必将应用程序的每次使用都下载到真正的iPhone设备上。这可以省下大量的时间和工作量。

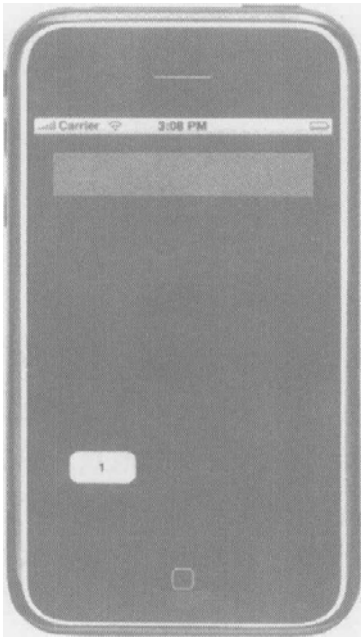

图21-1　第一个iPhone应用程序

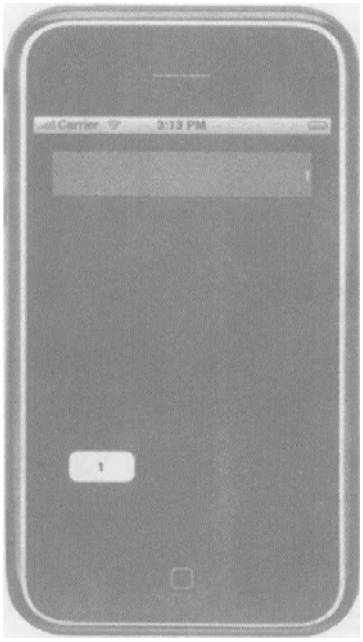

图21-2　iPhone应用程序的运行结果

为了在iPhone设备上运行应用程序，需要注册iPhone开发人员计划，并付给Apple 99美元的费用（此数字截止到本书撰写之际）。接下来，你将会收到一个激活码，它让你能够获得一个iPhone开发证书，从而可以在iPhone上测试并安装应用程序。不幸的是，如果没有完成这个过程，即使是在你自己的iPhone上也无法开发应用程序。注意，本章中我们开发的应用程序将在iPhone模拟器上进行加载和测试，而不是在iPhone设备上。

21.2.1　创建新的iPhone应用程序项目

让我们回到第一个应用程序的开发上。安装iPhone SDK后，启动Xcode应用程序。从File菜单中选择New Project。在iPhone OS下（如果在左窗格中看不到这个区域，证明你还没有安装iPhone SDK），单击Application，此时应该看到一个如图21-3所示的窗口。

这个窗口中显示了为不同类型应用程序提供起点的各种模板，表21-1对此进行了总结。

表21-1　iPhone应用程序模板

应用程序类型	说　明
Navigation-Based	针对使用导航控制器的应用程序。Contacts应用程序就是这类模板的一个例子
OpenGL ES	针对基于OpenGL图形的应用程序，比如游戏
Tab Bar	针对使用选项卡栏的应用程序。iPod应用程序就是一个例子
Utility	针对拥有反面视图的应用程序。Stock Quote应用程序就是这类模板的一个例子
View-Based	针对拥有单一视图的应用程序。绘制视图后便将它显示在窗口中
Window-Based	针对只从iPhone主窗口开始的应用程序。可以使用此模板作为任意应用程序的起点

回到New Project窗口上，选择位于最右上角的Window-based Application，然后单击Choose按钮。接着在Save As框中提示输入项目名称时，输入文本iPhone_1，然后单击Save按钮。默认情况下这也会成为应用程序的名称。从使用Xcode创建的前一个项目得知，现在创建的新项目中包含你要使用的文件模板，如图21-4所示。

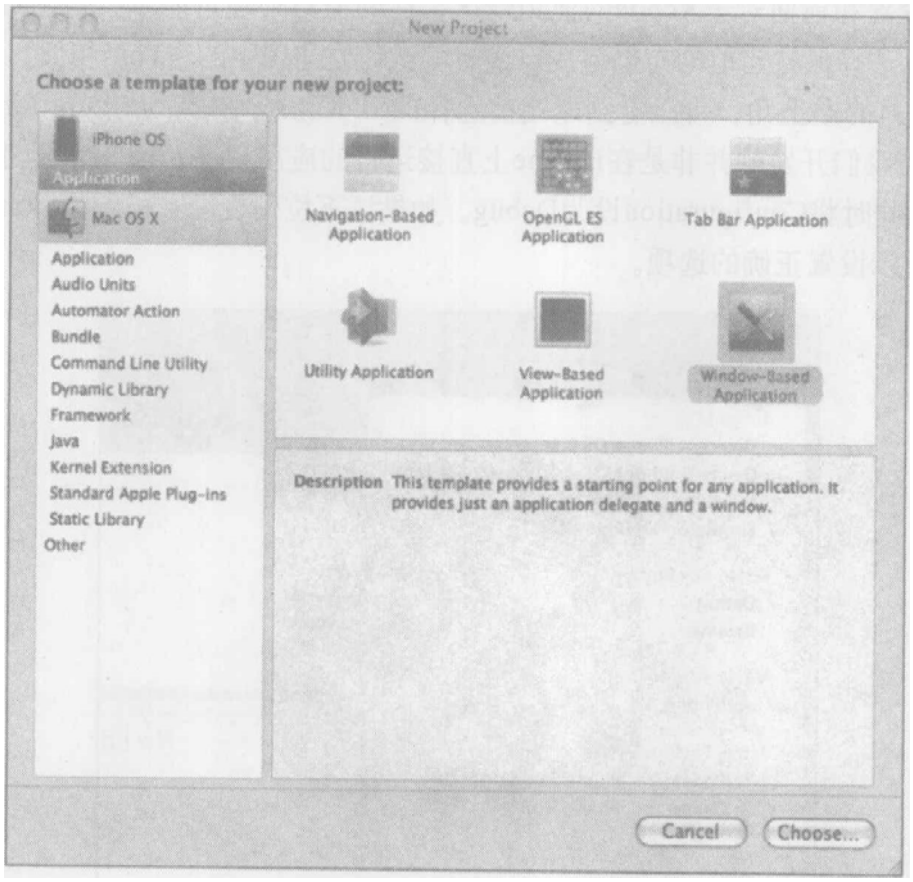

图21-3 开始一个新的iPhone项目

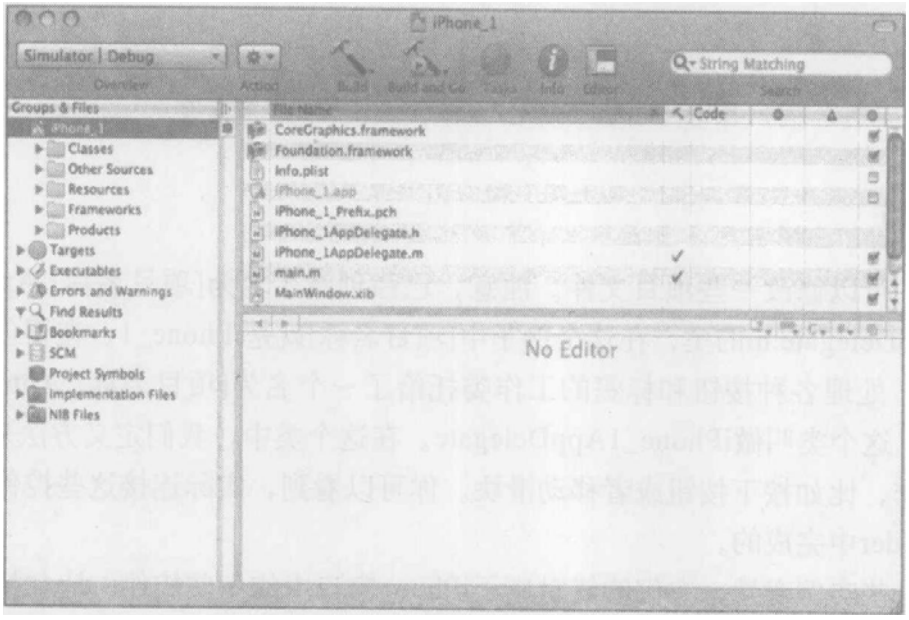

图21-4 创建新的iPhone项目iPhone_1

回到New Project窗口上，选择位于最右上角的Window-based Application，然后单击Choose按钮。接着在Save As框中提示输入项目名称时，输入文本iPhone_1，然后单击Save按钮。默认情况下这也会成为应用程序的名称。从使用Xcode创建的前一个项目得知，现在创建的新项目中包含你要使用的文件模板，如图21-4所示。

根据你的设置和前面对于Xcode的使用情况，你的窗口外观可能不会与图21-4中显示的完全一样。你可以选择保持当前的布局组成，或者尝试让它与图中的窗口更相似。

在Xcode窗口的左上角，可以看到以当前选择的SDK和Active Configuration为标记的一个下拉列表。因为我们开发的并非是在iPhone上直接运行的应用程序，因此需要将SDK设为运行iPhone模拟器，同时将Configuration设为Debug。如果该下拉列表没有标示为Simulator | Debug，请按照图21-5所示设置正确的选项。

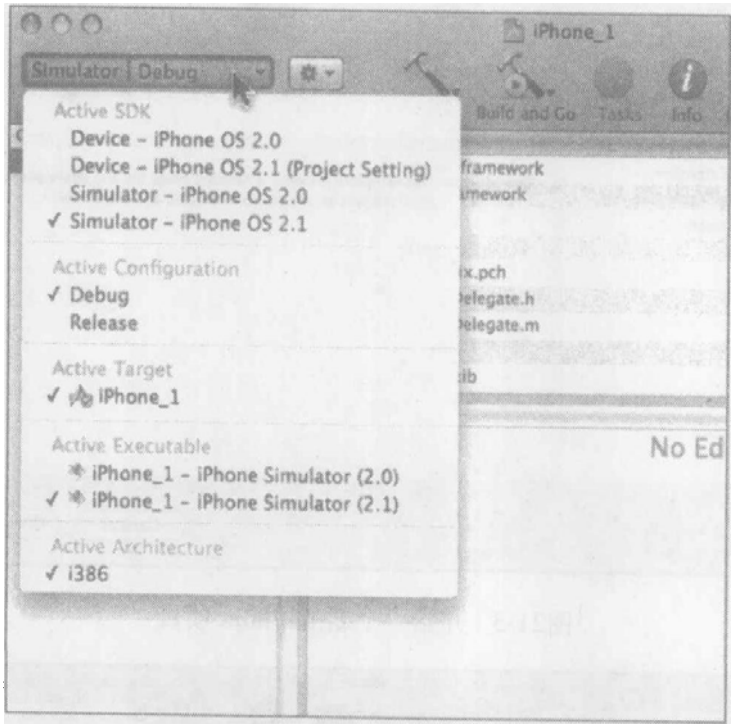

图21-5　为iPhone_1项目设置SDK和配置选项

21.2.2　输入代码

现在，我们可以修改一些项目文件。注意，已经创建了名为[项目名称] AppDelegate.h和[项目名称] AppDelegate.m的类，在这个例子中[项目名称]就是iPhone_1。在要创建的基于窗口的应用程序中，处理各种按钮和标签的工作委托给了一个名为[项目名称] AppDelegate的类，在这个例子中，这个类叫做iPhone_1AppDelegate。在这个类中，我们定义方法来响应iPhone窗口中出现的操作，比如按下按钮或者移动滑块。你可以看到，实际连接这些控件和响应方法是在Interface Builder中完成的。

该类还有一些实例变量，它们的值对应于iPhone窗口中的一些控件，比如标签名称或可编辑文本框内显示的文本。这些变量称为出口，而且与操作例程一样，实例变量与iPhone窗口中

实际控件的连接是在Interface Builder中完成的。

对于第一个应用程序，我们需要一个方法来响应按下标示为1的按钮的操作。我们还需要一个出口变量，用于包含我们在iPhone窗口顶部创建的标签中显示的文本和其他信息。

编辑文件iPhone_1AppDelegate.h，添加一个名为display的新UILabel变量，并声明一个名为click1:的操作方法，以便响应按下按钮的操作。接口文件的内容应该如代码清单21-1所示（在文件头部自动插入的注释行在此不显示。）

代码清单21-1 iPhone_1AppDelegate.h

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

@interface iPhone_1AppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    UILabel *display;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UILabel *display;

- (IBAction) click1: (id) sender;

@end
```

注意，iPhone应用程序导入了头文件<UIKit/UIKit.h>。这个头文件又导入了其他UIKit头文件，导入方式与Foundation.h头文件导入其他所需头文件（如NSString.h和NSObject.h）时类似。如果想要查看这个文件的内容，必须费一番功夫。下面是撰写本书时，它在我的系统上的安装位置：

```
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulat
or2.1.sdk/System/Library/Frameworks/UIKit.framework/Headers/UIKit.h
```

iPhone_1AppDelegate类现在有两个实例变量。第一个变量是一个名为window的UIWindow对象。该实例变量是在创建项目时自动创建的，它引用的是iPhone的主窗口。你添加的另一个名为display的实例变量属于UILabel类。这是一个出口变量，将连接到一个标签。设置这个变量的文本字段时，它会更新窗口中标签的对应文本。为UILabel类定义的其他方法允许设置和获取标签的其他属性，比如它的颜色、行数和字体大小。

当你学习了我们在此没有介绍的其他iPhone编程内容时，可能会在界面中使用其他的类。这些类中有一些的名称就表明了它的用途，比如UITextField、UIFont、UIView、UITableView、UIImageView、UIImage和UIButton。

window和display实例变量都是出口，而且在这两个变量的属性声明中，要注意IBOutlet标识符的使用。实际上在UIKit头文件UINibDeclarations.h中，IBOutlet的定义没有任何意义（也就是说从字面上，预处理器可以在源文件中将它替换成空白。）然而，Interface Builder在读取头文件时，只有定义为IBOutlet的变量才可以用作出口。

在接口文件中，注意我们声明了一个名为click1:的方法，它带有一个参数sender。调用click1:方法时，与事件相关的信息就是通过这个参数传递给该方法的。例如，对于用于响应按下不同按钮操作的例程，可以查询此参数来确定按下的是哪个按钮。

click1:方法的返回值类型定义为IBAction。(UINibDeclarations.h头文件中定义为void。)和IBOutlet一样，读取头文件时，Interface Builder使用这个标识符来确定可以用作操作的方法。

现在，是时候为你的类修改相应的iPhone_1AppDelegate.m实现了。在这里，你可以为display变量编写存取器方法（window访问方法已经合成了），并添加click1:方法的定义。

按照代码清单21-1中的内容编辑你的实现文件。

代码清单21-1 iPhone_1AppDelegate.m

```
#import *"iPhone_1AppDelegate.h ""
@implementation iPhone_1AppDelegate

@synthesize window, display;

- (void) applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after application launch
    [window makeKeyAndVisible];
}

-(IBAction) click1: (id) sender
{
    [display setText: @ "1"];
}

- (void) dealloc {
    [window release];
    [super dealloc];
}

@end
```

iPhone运行时，系统将自动调用一次applicationDidFinishLaunching:方法。正如其名所示，应用程序已经结束了启动。在这里，你可以初始化实例变量，在屏幕上绘制内容，并让窗口可见以显示其内容。最后一个操作是通过在方法结尾把makeKeyAndVisible消息发送给窗口来完成的。

click1:方法使用UILabel的setText:方法将出口变量display设为字符串1。在将按钮的按下动作连接到这个方法的调用上后，它能够执行预定的操作，将一个1放入iPhone窗口的显示中。为了进行连接，你现在必须学习如何使用Interface Builder。在这样做之前，编译程序，以便排除所有的编译器警告或出错消息。

21.2.3 设计界面

在图21-4和Xcode主窗口中，注意一个名为MainWindow.xib的文件。xib文件包含了与程序

的用户界面有关的所有信息，包括关于它的窗口、按钮、标签、选项卡栏、文本字段等信息。当然，你现在还没有用户界面！这是接下来要完成的工作。

双击MainWindow.xib文件，这将启动另一个名为Interface Builder的应用程序。还可以从项目的Resources文件夹访问XIB文件。

Interface Builder启动时，屏幕上会出现一系列窗口，如图21-6、图21-7和图21-8所示。实际打开的窗口可能与这些图略有出入。

Library窗口提供了一组可用于界面的控件。图21-6显示了这个窗口的一种显示格局。

MainWindow.xib窗口（图21-7）是在应用程序代码与界面之间建立连接的控制窗口，如下所示。

只是标示为Window的窗口显示了iPhone主窗口的布局。因为你尚未给iPhone窗口设计任何内容，该窗口开始为空，如图21-8所示。

我们要做的第一件事情是将iPhone窗口设置为黑色。为此，首先在名为Window的窗口内单击。现在，从Tools菜单中选择Inspector。这应该会打开如图21-9所示的Inspector窗口。

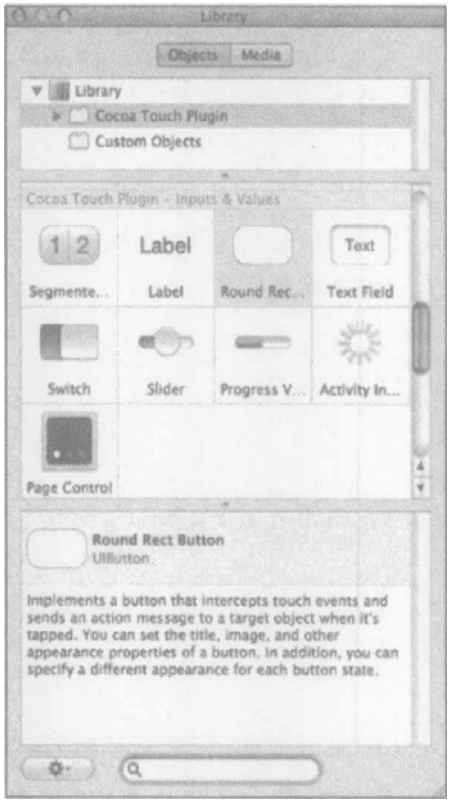

图21-6 Interface Builder Library窗口

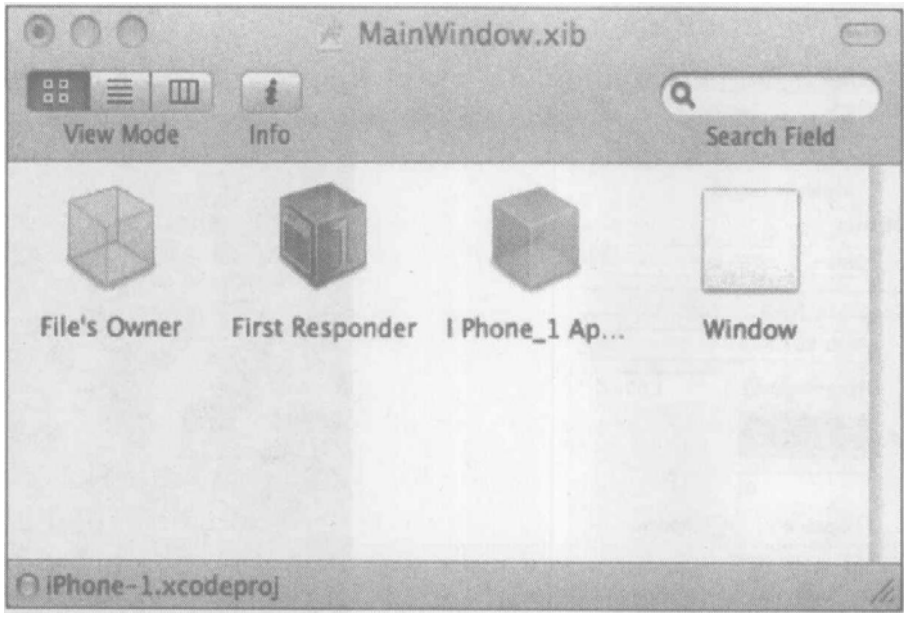

图21-7 Interface Builder MainWindow.xib窗口

确保Inspector窗口的名称显示为Windows Attributes，如图21-9所示。如果不是，单击顶部选项卡栏中最左侧的选项卡，以便显示正确的窗口。

如果向下看到窗口的View部分，可以看到一个标示为Background的属性。如果双击Background旁边的白色实心矩形，可以打开颜色拾取器。从拾取器中选择黑色，这将把

Inspector窗口中Backgournd属性旁边的矩形从白色变为黑色（参见图21-10）。

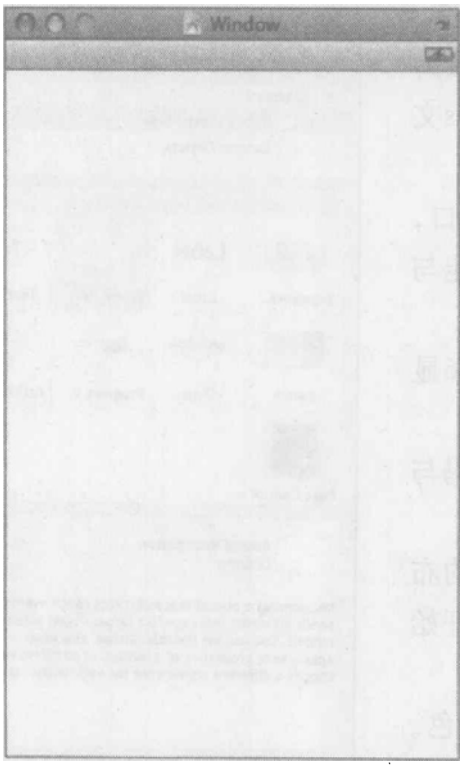

图21-8　Interface Builder iPhone窗口

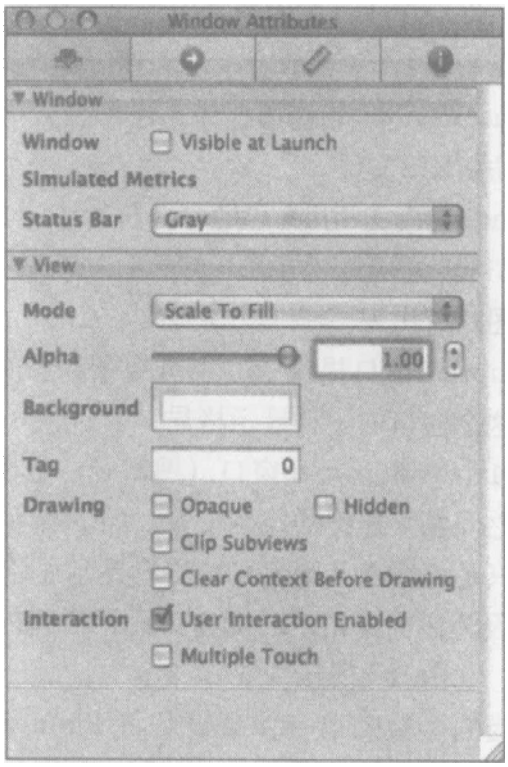

图21-9　Interface Builder Inspector窗口

如果看一看代表iPhone显示窗口的名为window的窗口，可以看到它已经变为黑色，如图21-11所示。

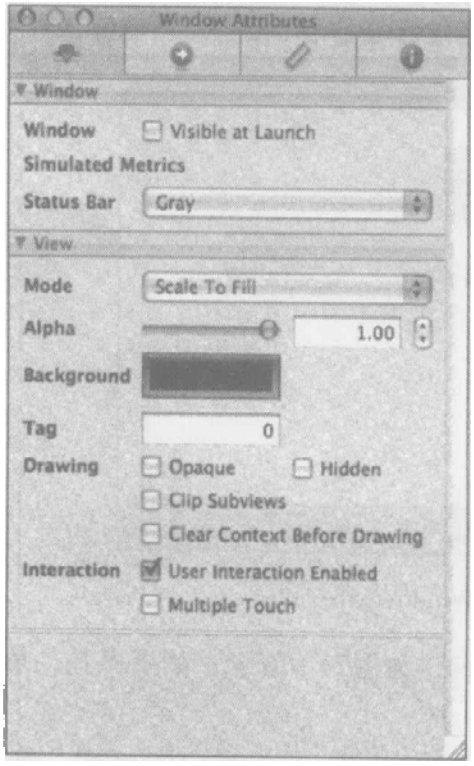

图21-10　修改窗口的背景颜色

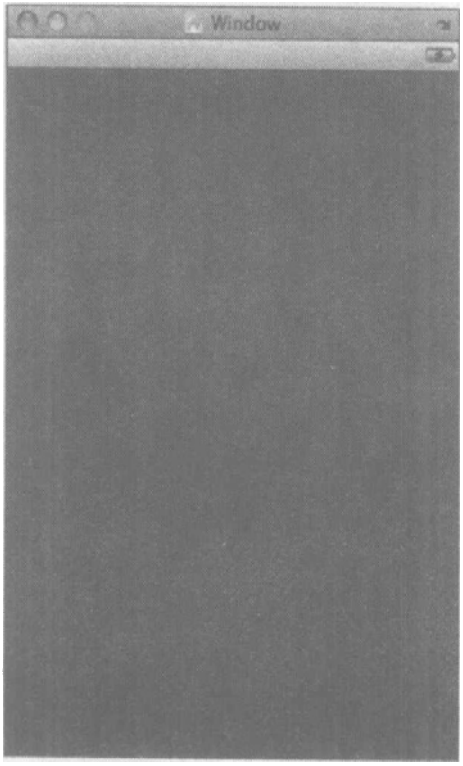

图21-11　界面窗口变为黑色

现在可以关闭Colors窗口。

在Library窗口中单击一个对象并将其拖到iPhone窗口中，可以在iPhone界面中创建新的对象。现在单击并拖动一个标签。当标签接近窗口中心顶部的位置时释放鼠标，如图21-12所示。

在窗口内四处移动标签时，会出现蓝色的导线。有时候，显示导线是为了将对象与前面在窗口中放置的其他对象对齐，而在其他时候是为了确保对象与其他对象和窗口边缘之间具有足够的间隔，从而与Apple的界面导线保持一致。

以后随时可以改变标签在窗口中的位置，方法是单击将其拖动到窗口内的其他位置。

现在让我们设置这个标签的一些属性。如果窗口中当前没有选中标签，单击你刚刚创建的标签以选中它。注意，Inspector窗口的内容会自动改变，从而提供关于窗口中当前选中对象的信息。我们不想让这个标签默认显示任何文本，因此将Text值修改为一个空字符串（也就是说，从Inspector窗口中显示的文本字段中删除字符串Label）。

对于Layout属性，选择Right-justified（右对齐）作为对齐方式。最后，将标签的背景颜色改为蓝色（或者你选择的其他颜色），就像把窗口的背景颜色改为黑色一样。Inspector窗口的外观应该如图21-13所示。

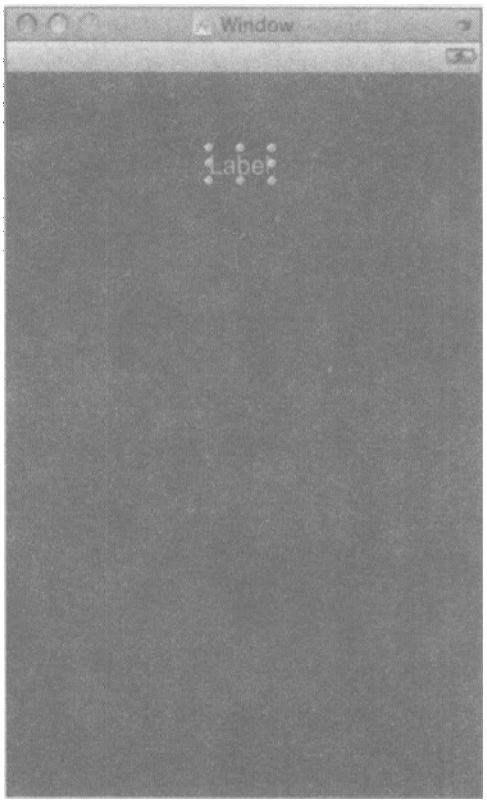

图21-12 添加一个标签

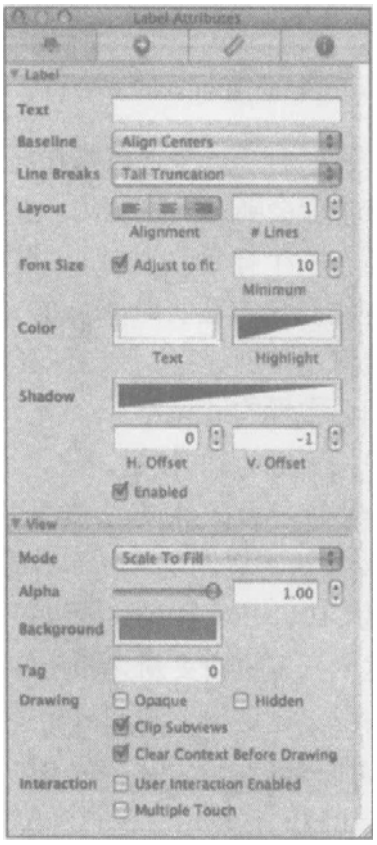

图21-13 修改标签的属性

现在让我们修改标签的大小。返回Window窗口，只要向外拉伸它的边角即可改变标签大小。改变标签的大小和位置，直到如图21-14所示为止。

现在我们给界面添加一个按钮。从Library窗口中单击一个Round Rect Button对象并将其拖到界面窗口中，把它放置在窗口的左下角，如图21-15所示。修改按钮上的标签有两种途径：

双击按钮然后输入文本，或者在Inspector窗口中设置Title字段。任意选择一种途径，让窗口变得如图21-15所示。

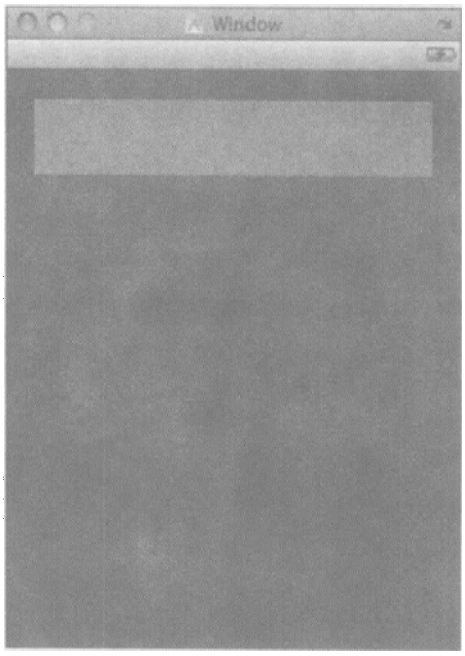

图21-14　改变标签的大小和位置

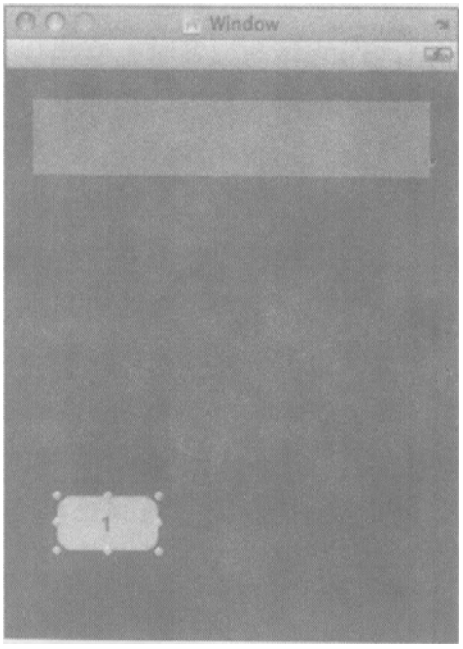

图21-15　给界面添加一个按钮

现在，我们有了一个连接到程序中display实例变量的标签，这样只要在程序中设置变量的值，标签文本就会随之发生变化。

我们还有一个标示1的按钮，只要按下它就会调用我们的click1:方法。该方法将display的文本字段设置为1。而且因为该变量将连接到标签，所以标签文本也会更新。概言之，下面是我们要实现的过程：

1. 用户按下标为1的按钮。
2. 这个事件导致click:方法被调用。
3. click:方法将实例变量display的文本修改为字符串1。
4. 因为UILabel对象display连接到了iPhone窗口中的标签，所以这个标签也会更新为相应的文本值，也就是值1。

为了实现上述过程，我们只需要建立两个连接。让我们讨论一下如何去做。

首先，让我们将按钮连接到IBAction方法click1:，方法是按下Ctrl键的同时单击按钮，然后将出现在屏幕上的蓝线拖动到MainWindow.xib窗口中的应用程序委托，如图21-16所示。

在Delegate立方体上释放鼠标后，会出现一个下拉列表，允许你选择一个IBAction方法连接到这个按钮。在我们的例子中，我们只有一个这样的方法叫做click1:，因此下拉列表中只会出现它。选择该方法以建立连接，如图21-17所示。

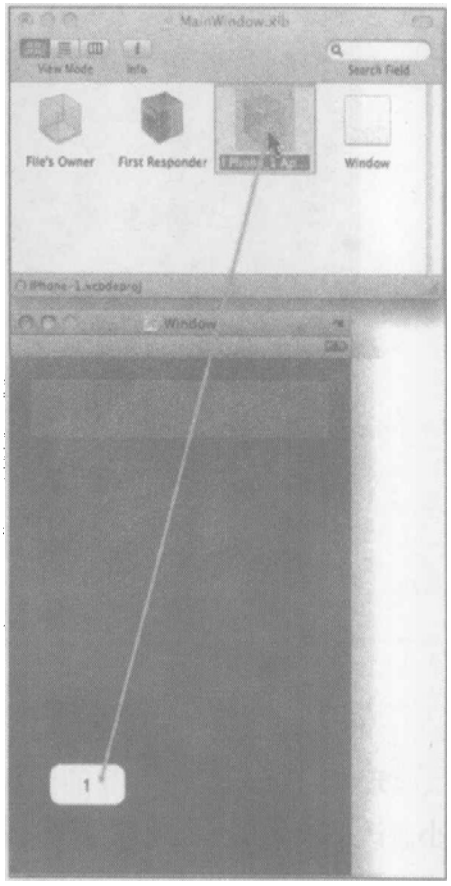

图21-16　为按钮添加一个操作

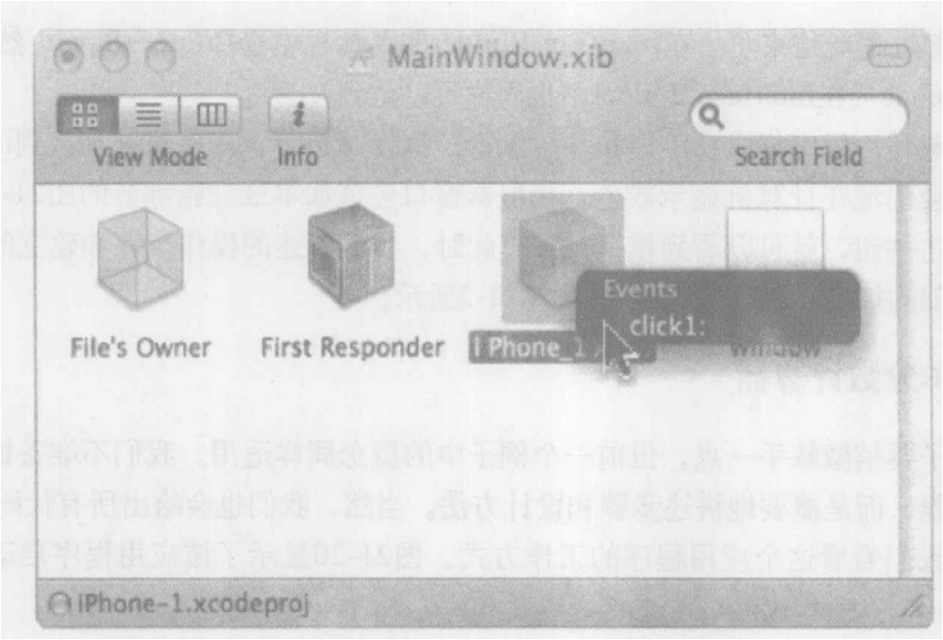

图21-17 将事件连接到方法

现在，让我们将display变量连接到标签。只要按下按钮就会执行应用程序中的一个方法（也就是说，操作的流程是从界面到应用程序委托），而设定应用程序中实例变量的值会更新iPhone窗口中的标签（在这里，操作流程是从应用程序委托到界面）。因此，一开始要在按下Ctrl键的同时单击应用程序委托图标，并将出现的蓝线拖动到Window中的标签，如图21-18所示。

释放鼠标时，你将看到对应类的IBOutlet变量的一个列表，作为可供选择的控件（UILabel）。我们的程序中有一个这样的变量，叫做display。选择此变量（如图21-19所示）并建立连接。

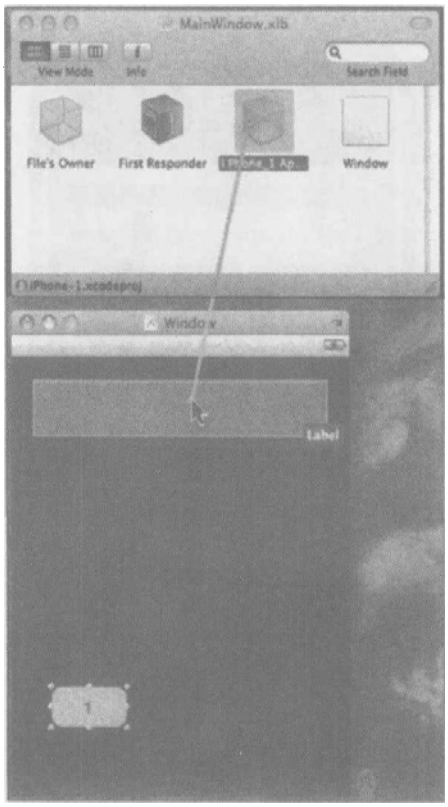

图21-18 连接一个出口变量

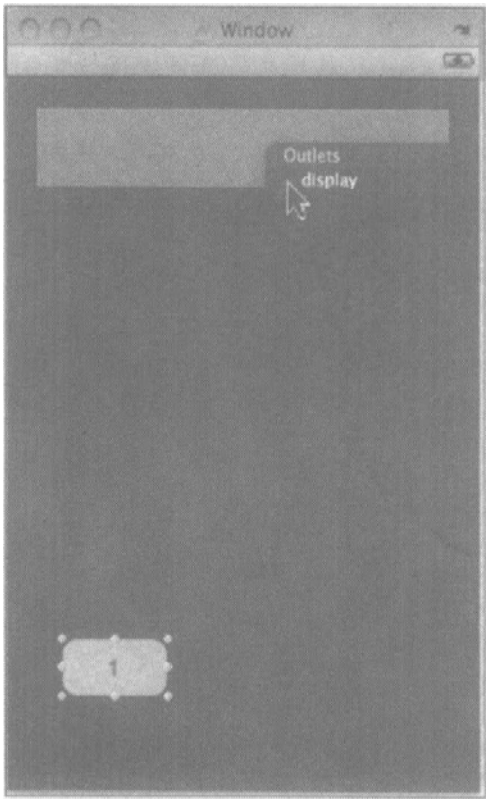

图21-19 完成连接

到这里全部过程就结束了！在Interface Builder的菜单栏中选择File→Save，然后选择Build and Go from Xcode（在Interface Builder中也同样可以做到）。

如果一切顺利，程序将成功编译和开始执行。执行开始时，程序将被加载到iPhone模拟器中，该模拟器会出现在计算机显示器上。模拟器窗口应该如本章开始部分的图21-1所示。只要在模拟器中单击按钮，就可以看到模拟效果。此时，我们概述的操作步骤和建立的连接，会导致在显示器顶部的标签中显示字符串1，如图21-2所示。

21.3 iPhone分数计算器

下一个例子要稍微棘手一点，但前一个例子中的概念同样适用。我们不准备说明创建这个例子的所有步骤，而是概要地讲述步骤和设计方法。当然，我们也会给出所有代码。

首先，让我们看看这个应用程序的工作方式。图21-20显示了该应用程序启动后在模拟器中的样子。

计算器应用程序的使用方法是首先输入分子，按下标有Over的键，然后输入分母。例如输入分数2/5的步骤是，按下2数字键，接着按下Over键，然后按下5数字键。可以看到，和其他计算器不同，这个计算器会在显示器上原样显示分数，因此2/5就显示为2/5。

输入一个分数后，接下来可以选择运算——加法、减法、乘法或除法，只要分别按下标有+、−、×或÷的键即可。

输入第二个分数后，按下=键即可完成运算，这一点与标准计算器相同。

注意　这个计算器的设计方式是在两个分数之间进行一次运算。本章末尾有一个练习，是让你去掉这种限制。

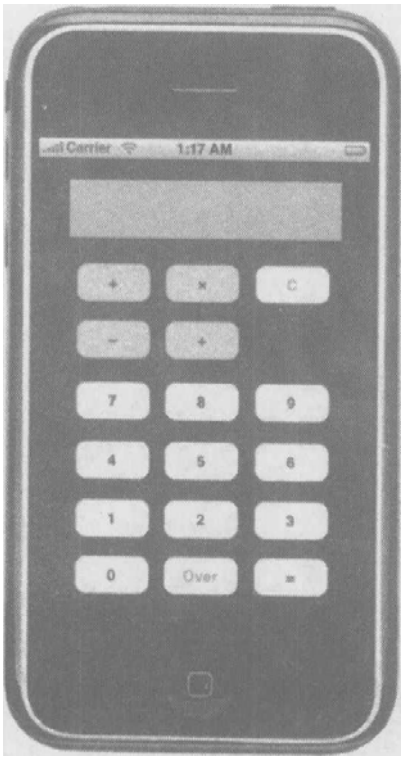

图21-20　完成连接

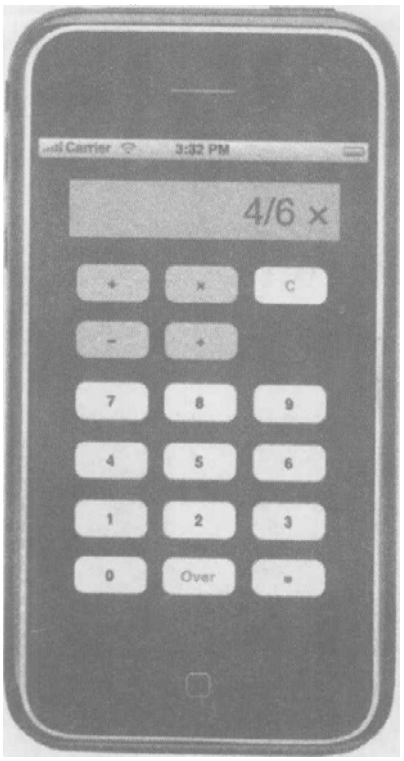

图21-21　一次运算中的输入

按下键时显示会不断更新。图21-21显示了输入分数4/6并按下乘法键后的显示。

图21-22显示了分数4/6和2/8相乘后的结果。你会注意到，结果为1/6表明结果被简化了。

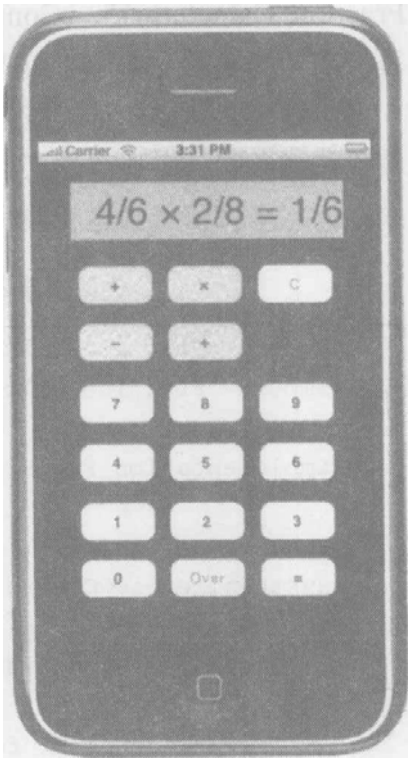

图21-22 两个分数相乘的结果

21.3.1 启动新的Fraction_Calculator项目

第一个iPhone程序是从基于Windows的项目模板开始的。在这个模板中，你在应用程序控制器（AppDelegate类）中直接进行（最少的）UI工作。这不是开发多信息UI应用程序的首选方法。AppDelegate类通常只处理与应用程序本身相关的改动，比如应用程序结束启动或者它将变为不活动。

应该在视图控制器（使用UIViewController类实现）中执行与UI相关的操作。这可能是显示文本，重新响应按钮的按下，或者在iPhone屏幕上绘制一个全新的视图。

这第二个程序示例将从创建一个新项目开始。这一次，从New Project窗口中选择View-Base Application。给新项目起名为Fraction_Calculator。

创建项目时，这次可以注意到已经定义好了两个类模板：Fraction_CalculatorAppDelegate.h和Fraction_ Calculator-AppDelegate.m定义了项目的应用程序控制器类，而Fraction_CalculatorViewController.h和Fraction_CalculatorViewController.m定义了项目的视图控制器类。如前所述，所有工作是在后一个类中进行的。

我们将首先从应用程序控制器类开始。它包含两个实例变量：一个用于引用iPhone窗口，另一个用于引用视图控制器。这两个变量都是由Xcode建立的。事实上，不需要对应用程序控制器的.h或.m文件做任何改动。

Fraction_CalculatorAppDelegate接口文件如代码清单21-2所示。

代码清单21-2 Fraction_CalculatorAppDelegate.h接口文件

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

@class Fraction_CalculatorViewController;

@interface Fraction_CalculatorAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet Fraction_CalculatorViewController *viewController;
}

@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) Fraction_CalculatorViewController *viewController;

@end
```

UIWindow实例变量window的用途与第一个程序示例相同，它代表着iPhone窗口。Fraction_CalculatorViewController实例变量代表将管理与用户的所有交互以及显示的视图控制器。在类的实现文件中将放入与这些任务相关的所有工作。

代码清单21-2显示了应用程序控制器类的实现文件。如前所述，我们在这个文件中不会做像代码清单21-1中那样的工作，因为这些工作均已经委托给了视图控制器。因此这个文件原封未动，完全保留着创建新项目时Xcode生成它时的原样。

代码清单21-2 Fraction_CalculatorAppDelegate.h实现文件

```
#import "Fraction_CalculatorAppDelegate.h"
#import "Fraction_CalculatorViewController.h"

@implementation Fraction_CalculatorAppDelegate

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [viewController release];
    [super dealloc];
}
@end
```

21.3.2 定义视图控制器

现在让我们编写视图控制器Fraction_CalculatorViewController的代码。我们将从接口文件开始，如代码清单21-2所示。

代码清单21-2 Fraction_CalculatorViewController.h接口文件

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

@interface Fraction_CalculatorViewController : UIViewController {
    UILabel         *display;
    char            op;
    int             currentNumber;
    NSMutableString *displayString;
    BOOL            firstOperand, isNumerator;
```

```
    Calculator      *myCalculator;
}

@property (nonatomic, retain) IBOutlet UILabel *display;
@property (nonatomic, retain) NSMutableString *displayString;

-(void) processDigit: (int) digit;
-(void) processOp: (char) theop;
-(void) storeFracPart;

// Numeric keys

-(IBAction) clickDigit: (id) sender;

// Arithmetic Operation keys

-(IBAction) clickPlus: (id) sender;
-(IBAction) clickMinus: (id) sender;
-(IBAction) clickMultiply: (id) sender;
-(IBAction) clickDivide: (id) sender;

// Misc. Keys

-(IBAction) clickOver: (id) sender;
-(IBAction) clickEquals: (id) sender;
-(IBAction) clickClear: (id) sender;

@end
```

有一些内务管理变量分别用于构造分数（currentNumber、firstOperand和isNumerator）和构造显示字符串（displayString）。还有一个Calculator对象（myCalculator）用于执行两个分数间的实际计算。我们将把一个名为clickDigit:的方法关联到处理按下数字键0～9的操作上。最后，我们定义了一些方法，用于保存要执行的运算（clickPlus:、clickMinus、clickMultiply:、clickDivide:）；当按下=键时，执行实际计算（clickEquals:），清除当前运算（clickClear:）；当按下Over键时，分离分子与分母（clickOver:）。还有一些方法（processDigit:、processOp:和storeFracPart）用于帮助处理上面提到过的一些杂事。

代码清单21-2显示了这个控制器类的实现文件。

代码清单21-2 Fraction_CalculatorViewController.h实现文件

```
#import "Fraction_CalculatorViewController.h"
@implementation Fraction_CalculatorViewController
@synthesize display, displayString;
-(void) viewDidLoad {
   // Override point for customization after application launch
```

```
    firstOperand = YES;
    isNumerator = YES;
    self.displayString = [NSMutableString stringWithCapacity: 40];
    myCalculator = [[Calculator alloc] init];

}

-(void) processDigit: (int) digit
{
    currentNumber = currentNumber * 10 + digit;

    [displayString appendString: [NSString stringWithFormat: @ "%i", digit]];
    [display setText: displayString];
}

- (IBAction) clickDigit:(id)sender
{
    int digit = [sender tag];

    [self processDigit:digit];
}

-(void) processOp: (char) theOp
{
    NSString *opStr;

    op = theOp;

    switch (theOp) {
        case '+':
            opStr = @ "+";
            break;
        case '-':
            opStr = @ "-";
            break;
        case '*':
            opStr = @ "×";
            break;
        case '/':
            opStr = @ "÷";
            break;
    }

    [self storeFracPart];
    firstOperand = NO;
    isNumerator = YES;
```

```
    [displayString appendString: opStr];
    [display setText: displayString];
}

-(void) storeFracPart
{
    if (firstOperand) {
        if (isNumerator) {
            myCalculator.operand1.numerator = currentNumber;
            myCalculator.operand1.denominator = 1; // e.g. 3 * 4/5 =
        }
        else
           myCalculator.operand1.denominator = currentNumber;
    }
    else if (isNumerator) {
        myCalculator.operand2.numerator = currentNumber;
        myCalculator.operand2.denominator = 1; // e.g. 3/2 * 4 =
    }
    else {
        myCalculator.operand2.denominator = currentNumber;
        firstOperand = YES;
    }

    currentNumber = 0;
}

-(IBAction) clickOver: (id) sender
{
    [self storeFracPart];
    isNumerator = NO;
    [displayString appendString: @ "/"];
    [display setText: displayString];
}

// Arithmetic Operation keys

-(IBAction) clickPlus: (id) sender
{
    [self processOp: '+'];
}

-(IBAction) clickMinus: (id) sender
{
    [self processOp: '-'];
}

-(IBAction) clickMultiply: (id) sender
```

```
{
    [self processOp: '*'];
}

-(IBAction) clickDivide: (id) sender
{
    [self processOp: '/'];
}

// Misc. Keys

-(IBAction) clickEquals: (id) sender
{
    [self storeFracPart];
    [myCalculator performOperation: op];
    [displayString appendString: @ " = "];
    [displayString appendString: [myCalculator.accumulator convertToString]];
    [display setText: displayString];

    currentNumber = 0;
    isNumerator = YES;
    firstOperand = YES;
    [displayString setString: @ ""];
}

-(IBAction) clickClear: (id) sender
{
    isNumerator = YES;
    firstOperand = YES;
    currentNumber = 0;
    [myCalculator clear];

    [displayString setString: @""];
    [display setText: displayString];
}

- (void)dealloc {
    [myCalculator release];
    [super dealloc];
}

@end
```

和前一个应用程序一样，计算器窗口仍然包含一个标签，我们仍然称之为display。当用户逐位地输入一个数字时，我们需要同步构造这个数字。变量current_Number保存了这个输入中的数字，而BOOL变量firstOperand和isNumerator分别用于跟踪输入的是第一个还是第二个操作数，以及用户当前输入的是该操作数的分子还是分母。

在计算器上按下一个数字按钮时，我们会将它组装起来，这样可以将一些识别信息传递给clickDigit:方法，从而识别出按下的是哪个数字按钮。这是通过将每个数字按钮的tag属性（使用Interface Builder的Inspector）设为一个唯一值来实现的。在这个例子中，我们把tag设为了对应的数字。因此标为0的按钮的tag将被设为0，而标为1的按钮的tag将被设为1，依此类推。接下来，把tag消息发送给clickDigti:方法的sender参数，就可以获得按钮tag的值。这是在clickDigit:方法中完成的，如下所示：

```
- (IBAction) clickDigit:(id)sender
{
    int digit = [sender tag];

    [self processDigit:digit];
}
```

代码清单21-2中的按钮比第一个应用程序中要多得多。视图控制器实现文件中的复杂性大部分来自于构造和显示分数。如前所述，按下数字按钮0～9时，就会执行操作方法clickDigit:。该方法会调用processDigit:方法并将相应的数字附加到变量currentNumber中正在构造的数字末尾。该方法还会把数字添加给变量displayString中保存的当前显示字符串，并更新显示内容：

```
-(void) processDigit: (int) digit
{
    currentNumber = currentNumber * 10 + digit;

    [displayString appendString: [NSString stringWithFormat: @ "%i", digit]];
    [display setText: displayString];
}
```

按下=键时，就会调用clickEquals:方法执行运算。计算器将执行两个分数之间的运算，并将结果保存在其累加器中。这个累加器的值是在clickEquals:方法中获取的，然后把结果添加到显示中。

21.3.3 Fraction类

Fraction类在以前的例子和这个例子中几乎没有什么变化，只是增加了一个新的convertToString方法，用于将分数转换为相应的字符串表示。代码清单21-2显示了Fraction接口文件和对应的实现文件。

代码清单21-2 Fraction.h接口文件

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

@interface Fraction : NSObject {
    int numerator;
    int denominator;
}

@property int numerator, denominator;
```

```
-(void)     print;
-(void)     setTo: (int) n over: (int) d;
-(Fraction *)     add: (Fraction *) f;
-(Fraction *)     subtract: (Fraction *) f;
-(Fraction *)     multiply: (Fraction *) f;
-(Fraction *)     divide: (Fraction *) f;
-(void)        reduce;
-(double)      convertToNum;
-(NSString *) convertToString;

@end
```

代码清单21-2 Fraction.m实现文件

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
    denominator = d;
}

-(void) print
{
    NSLog (@ "%i/%i", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 1.0;
}

-(NSString *) convertToString;
{
    if (numerator == denominator)
        if (numerator == 0)
            return @ "0";
        else
            return @ "1";
```

```
    else if (denominator == 1)
        return [NSString stringWithFormat: @ "%i", numerator];
    else
        return [NSString stringWithFormat: @ "%i/%i",
                numerator, denominator];
}

// add a Fraction to the receiver

-(Fraction *) add: (Fraction *) f
{
    // To add two fractions:
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    // result will store the result of the addition
    Fraction   *result = [[Fraction alloc] init];
    int        resultNum, resultDenom;
    resultNum = numerator * f.denominator + denominator * f.numerator;
    resultDenom = denominator * f.denominator;

    [result setTo: resultNum over: resultDenom];
    [result reduce];

    return [result autorelease];
}

-(Fraction *) subtract: (Fraction *) f
{
    // To sub two fractions:
    // a/b - c/d = ((a*d) - (b*c)) / (b * d)

    Fraction *result = [[Fraction alloc] init];
    int   resultNum, resultDenom;

    resultNum = numerator * f.denominator - denominator * f.numerator;
    resultDenom = denominator * f.denominator;

    [result setTo: resultNum over: resultDenom];
    [result reduce];
    return [result autorelease];
}

-(Fraction *) multiply: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    [result setTo: numerator * f.numerator over: denominator
```

```
                * f.denominator];
    [result reduce];

    return [result autorelease];
}

-(Fraction *) divide: (Fraction *) f
{
    Fraction *result = [[Fraction alloc] init];

    [result setTo: numerator * f.denominator over: denominator * f.numerator];
    [result reduce];

    return [result autorelease];
}

- (void) reduce
{
    int u = numerator;
    int v = denominator;
    int temp;

    if (u == 0)
        return;
    else if (u <0)
     u = -u;
    while (v != 0) {
       temp = u % v;
        u = v;
        v = temp;
    }

    numerator /= u;
    denominator /= u;
}

@end
```

convertToString:方法用于检查分数的分子和分母，以产生更适合查看的结果。如果分子和分母相等（但不是0），我们将返回@“1”。如果分子为0，则返回字符串@“0”。如果分母为1，则分数是一个整数，因此不需要显示分母。

convertToString:方法内部使用的stringWithFormat:方法返回一个指定格式的字符串（类似于NSLog）和一个用逗号隔开的参数列表。将参数传递给一个参数数量不定的方法时，可以使用逗号将多个参数隔开，这与将参数传递给NSLog函数时的做法是一样的。

21.3.4 处理分数的Calculator类

接下来，到了了解Calculator类的时候了。这个类的概念类似于本书中我们以前开发的相同名称的类。然而在这个例子中，我们的计算器必须知道如何处理分数。下面给出了我们新Calculator类的接口与实现文件。

代码清单21-2 Calculator.h接口文件

```
#import <UIKit/UIKit.h>
#import 'Fraction.h"

@interface Calculator : NSObject {
    Fraction *operand1;
    Fraction *operand2;
    Fraction *accumulator;
}

@property (retain, nonatomic) Fraction *operand1, *operand2, *accumulator;

-(Fraction *) performOperation: (char) op;
-(void) clear;

@end
```

代码清单21-2 Calculator.m实现文件

```
#import "Calculator.h"

@implementation Calculator
@synthesize operand1, operand2, accumulator;

-(id) init
{
    self = [super init];

    operand1 = [[Fraction alloc] init];
    operand2 = [[Fraction alloc] init];
    accumulator = [[Fraction alloc] init];

    return self;
}

-(void) clear
{
    if (accumulator) {
        accumulator.numerator = 0;
        accumulator.denominator = 0;
```

```
    }
}

-(Fraction *) performOperation: (char) op
{
    Fraction *result;

    switch (op) {
        case '+':
            result = [operand1 add: operand2];
            break;
        case '-':
            result = [operand1 subtract: operand2];
            break;
        case '*':
            result = [operand1 multiply: operand2];
            break;
        case '/':
            result= [operand1 divide: operand2];
            break;
    }

    accumulator.numerator = result.numerator;
    accumulator.denominator = result.denominator;

    return accumulator;
}

-(void) dealloc
{
    [operand1 release];
    [operand2 release];
    [accumulator release];
    [super dealloc];
}

@end
```

21.3.5 设计UI

你可能已经注意到了，项目的Resources文件夹中有两个xib文件：一个叫做MainWindow.xib，另一个叫做Fraction_CalculatorViewController.xib。前一个文件你根本不必理会，只要双击后者的文件名打开这个文件即可。Interface Builder启动时，可以看到标示为Fraction_CalculatorViewController.xib的窗口中显示了一个标有View的标签，如图21-23所示。

如果View窗口没有打开，双击图标打开它。在这个View窗口内部，可以设计计算器的UI。

在每个数字按钮与clickDigit:方法之间建立连接，方法是在按下Ctrl键的同时，依次单击每个按钮并将其拖到Fraction_CalculatorViewController.xib窗口中的File's Owner图标，然后从Events下拉列表中选择clickDigit:。此外，还要在Inspector窗口中将每个数字按钮的Tag值设定为对应于按钮标示的数字。因此，标示为0的数字按钮的Tag值将被设为0，而标示为1的数字按钮的Tag值被设为1，依此类推。

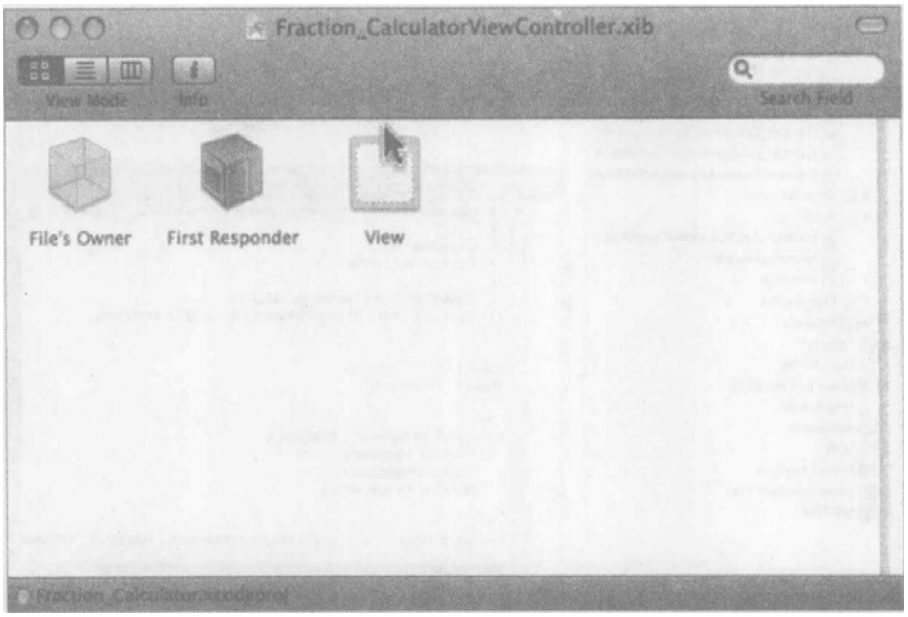

图21-23 Fraction_CalculatorViewController.xib窗口

在View窗口中绘制其他的按钮并建立相应的连接。不要忘记插入一个标签，以供计算器显示之用，并在按下Ctrl键的同时，单击File's Owner图标并将其拖动到标签上。从随后出现的Outlets下拉列表中选择display。

以上就是全部过程！你的界面设计已经完成，现在可以使用这个分数计算器应用程序了。

21.4 小结

图21-24显示了Xcode项目窗口，在其中可以看到与Fraction计算器项目相关的所有文件。

下面总结了创建iPhone分数计算器应用程序的步骤：

1. 创建一个新的基于视图的应用程序。
2. 在Fraction_CalculatorViewController.h和.m文件中输入UI代码。
3. 给项目添加Fraction和Calculator类。
4. 在Interface Builder中打开Fraction_CalculatorViewController.xib，以便创建UI。
5. 将View窗口的背景改为黑色。
6. 创建一个标签和多个按钮，并在View窗口中排好它们的位置。
7. 在按下Ctrl键的同时，单击File's Owner图标并拖到在View窗口中创建的标签上，然后将标签设置为display。
8. 在View窗口中，在按下Ctrl键的同时，依次单击每个按钮并将其拖到File's Owner图标中，并建立到正确操作方法的连接。为每个数字按钮选择clickDigit:方法。此外，将每

个数字按钮的tag属性设置为对应的数字0～9，这样clickDigit:方法就能识别按下的是哪个按钮。

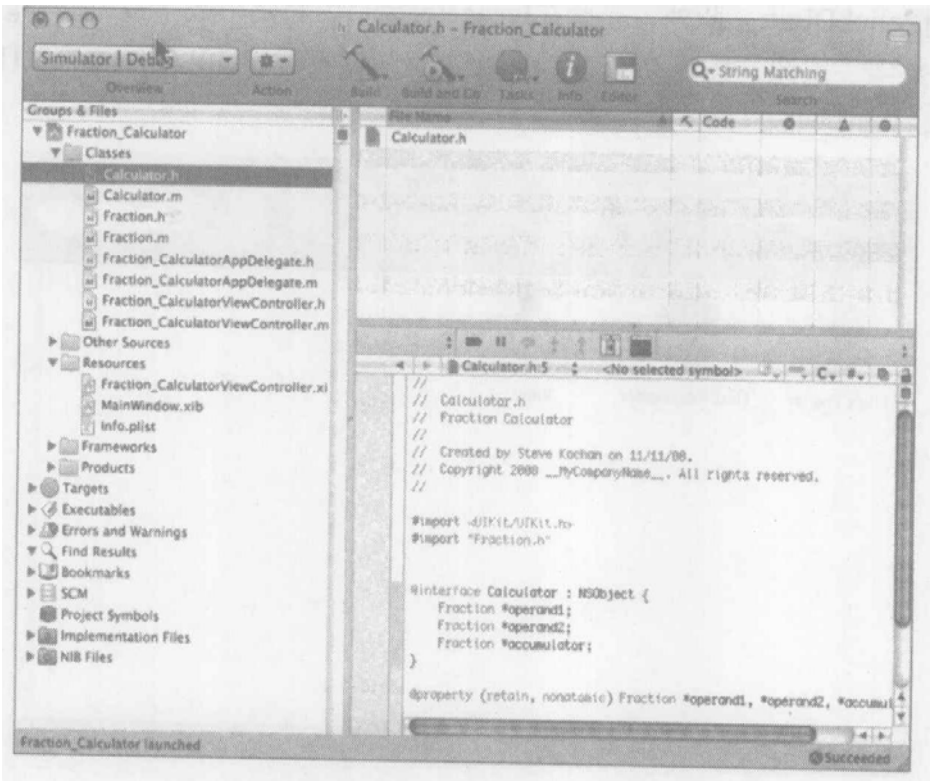

图21-24　Fraction计算器项目文件

学习如何使用视图控制器是值得的，即使这样做的工作量比仅仅在应用程序控制器中实现这一切的要大也是如此。然而，如果要在应用程序中实现某些复杂的任务，比如动画、响应屏幕旋转、使用导航控制器或者构建选项卡式的界面，就需要用到视图控制器。

希望这些对于开发iPhone应用程序的简要介绍，能够为你编写自己的iPhone应用程序提供一个良好的起点。如前所述，UIKit中提供的功能非常多，等待着你去探讨！

我们的分数计算器具有几个限制，在下面的练习中需要解决其中的一些限制。

21.5　练习

1. 给分数计算器应用程序添加一个Convert按钮。按下此按钮时，使用Fraction类的convertToNum方法生成分数形式结果的数字表示形式。将得到的结果转化为一个字符串并显示在计算器的显示屏中。
2. 按照下面的要求修改分数计算器应用程序：如果在输入分子之前按下-键，可以输入一个负的分数。
3. 如果输入0值作为第一或第二个操作数的分母，在分数计算器的显示屏中显示字符串Error。
4. 修改分数计算器应用程序，让它可以进行连续计算。例如，允许输入以下运算：
   1/5 + 2/7 − 3/8 =
5. 给应用程序添加一个图标，并让它出现在iPhone的主屏幕上。方法是在应用程序的Resources文件夹中添加一幅图片以用作图标（.png文件），然后把信息属性列表

（Resources文件夹中的Info.plist文件）中“Icon file”键的值设为这个图像文件，如图21-25所示。

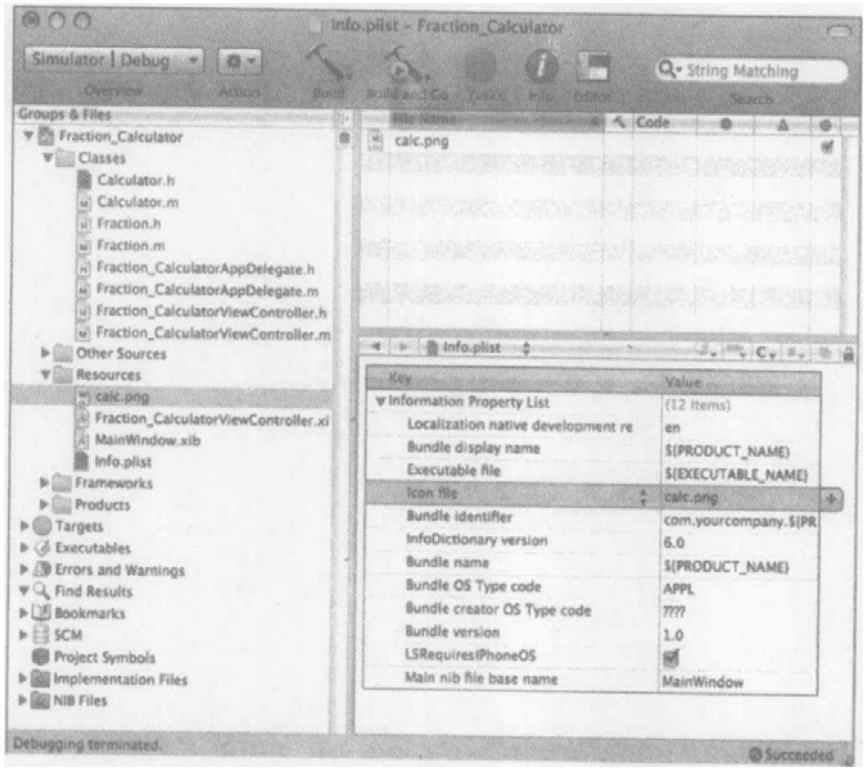

图21-25 添加一个应用程序图标

在Internet上找到一幅合适的计算器图像，将其用作你的应用程序图标。

第四部分 附 录

附录A 术 语 表

本附录包含了很多将会用到的术语的非正式定义。有些术语直接与Objective-C语言本身有关，而其他术语则有自己的语源，这些语源来自于面向对象程序设计的规范。在后一种情况中，术语的含义只在术语明确地应用于Objective-C时，才提供定义。

抽象类 为了更方便地创建子类，而定义的类。实例是从子类创建的，而不是从抽象类。同时参见具体子类。

存取器方法 可获得或设置实例变量值的方法。使用存取器方法可设置和检索实例变量的值，这与数据封装的方法论是相一致的。

Application Kit 用于开发应用程序用户界面的框架，用户界面包含各种对象，如菜单、工具栏和窗口。该框架是Cocoa的组成部分，通常称为AppKit。

归档 将对象数据的表示转换成一种以后可恢复（解归档）的格式。

数组 一个有序的值集合。数组可定义为Objective-C中的基本类型，并且通过NSArray和NSMutableArray类实现为Foundation下的对象。

自动变量 进入一个语句块时自动分配空间、退出语句块时自动释放的变量。自动变量的作用域限于定义它的程序块之内，并且这些变量没有默认的初始值。它们的前面可以可选地放置关键字auto。

自动释放池 定义在Foundation框架中的对象，它跟踪当池本身被释放时将要释放的对象。通过向对象发送autorelease消息可将对象添加到池中。

位域 包含一个或多个具有指定位宽度的整数域的结构。可以访问和操作位域，其方式与其他结构成员所用的方式相同。

分类 组合到特定名称下的一组方法。分类可以模块化方法的定义，并可用于向现有类添加新方法。

字符串 一种以null结尾的字符序列。

类 一组实例变量和可访问这些变量的方法。定义类之后，即可创建类的实例（即对象）。

类方法 对类对象调用的方法（定义有一个前导的+号）。同时参见实例方法。

类对象 标识特定类的对象。可以将类名用作消息的接收者来调用类方法。在其他地方，可以对该类调用class方法来创建类对象。

群集 组合了一组私有具体子类的抽象类，它通过抽象类向用户提供了一个简单的接口。

Cocoa 一种开发环境，它由Foundation和

Application Kit框架组成。

Cocoa Touch　一种开发环境，它由Foundation和UIKit框架组成。

集合　一种Foundation框架对象，可以是数组、字典或用于分组和操作相关对象的集。

编译时　分析源代码并将其转换成所谓目标码的底层格式的时期。

合成类　由来自其他类的对象组成的类。通常，它用作子类的替代物。

具体子类　抽象类的子类。可从具体子类创建实例。

符合　如果类通过直接实现或间接通过继承，采用了协议中的所有方法，则称该类符合这项协议。

常量字符串　括在一对双引号中的字符序列。如果以@character开头，通常定义NSConstantString类型的常量字符串对象。

数据封装　将对象的数据存储在对象的实例变量中，并只能通过对象的方法进行访问的概念。这样可维护数据的完整性。

委托　指导另一个对象实现某项行为的对象。

指定的初始化函数　将调用类或子类（通过向super发送消息）中的其他所有初始化方法的方法。

字典　在Foundation下，利用NSDictionary和NSMutableDictionary实现的键/值对集合。

指令　Objective-C中的一种特殊结构，它以at符号（@）开始。@interface、@implementation、@end和@class都是指令的例子。

分布式对象（Distributed Object）　一个应用程序中的Foundation对象与另一个（很可能是运行在另一台计算机上）应用程序的Foundation对象进行通信的能力。

动态绑定　在运行时而不是编译时确定对象将要调用的方法。

动态类型　在运行时而不是编译时确定对象所属的类。同时参见静态类型。

封装　同时参见数据封装。

extern变量　同时参见全局变量。

工厂方法　同时参见类方法。

工厂对象　同时参见类对象。

正式协议　使用@protocol指令定义在一个名称下的相关方法集。不同的类（不必是相关的）可以采用一个正式协议，只要实现（或继承）这个正式协议的所有方法即可。同时参见非正式协议。

转发　向另一个方法发送一条消息及相关（多个）参数，以便执行的过程。

Foundation框架　类、函数和协议的集合，这些类、函数和协议形成了应用程序开发的基础，同时提供了各种基本工具性程序，如内存管理、文件和URL访问、归档，以及集合、字符串、数字和日期对象的使用。

框架　类、函数、协议、文档、头文件和其他所有相关资源的集合。如，Cocoa框架是用于在Mac OS X下开发交互式图形应用程序的框架。

函数　利用一个名称标识的语句块，它可以接受通过值传递的一个或多个参数，并且可以选择返回一个值。对于定义函数的文件而言，函数可以是局部的（静态的）也可以是全局的，在后一种情况，可以从定义在其他文件中的函数或方法调用这些函数。

垃圾回收　一种内存管理系统，可自动释放未被引用的对象所使用的内存。iPhone运行时环境中不支持垃圾回收。

gcc　它是Free Software Foundation（FSF）开发的一种编译器的名称。gcc支持很多程序设计语言，包括C、Objective-C和C++。gcc是在Mac OS X上编译Objective-C程序所用的标准编译器。

gdb　由gcc编译的程序的标准调试工具。

取值方法　一种存取器方法，可检索实例变量的值。同时参见赋值方法。

全局变量　在所有方法或函数外部定义的变量，同一个源文件中或将该变量定义为extern的其他源文件中的任何方法或函数都可以访问

该变量。

头文件 包含常见定义、宏和变量声明的文件，可使用#import或#include语句将其包含到程序中。

id 通用数据类型，可容纳指向任何类型对象的指针。

不可变对象 不能修改其值的对象。如，Foundation框架中包含的NSString、NSDictionary和NSArray对象。同时参见可变对象。

实现部分 用于定义类的部分，它包含声明在相应接口部分（或者由协议定义所指定的）的方法的实际代码（即实现）。

非正式协议 作为一个分类（通常作为根类的分类）声明的逻辑上相关的方法集。与正式协议不同，非正式协议中的方法不必全部实现。同时参见正式协议。

继承 将一个类的方法和实例变量从根对象开始向下传递到子类的过程。

实例 类的具体表示。实例通常通过向类对象发送一条alloc或new消息来创建的对象。

实例方法 可被类实例调用的方法。同时参见类方法。

实例变量 在接口部分（它包含该对象的每个实例）声明的（或从父类继承来的）变量。实例方法可直接访问它们的实例变量。

Interface Builder Mac OS X下为应用程序构建图形用户界面的工具。

接口部分 用于声明类、类的超类、实例变量和方法的部分。对每个方法而言，还声明参数类型和返回类型。同时参见实现部分。

国际化 参见本地化。

isa 在根对象中定义并且所有对象都要继承的一个特殊的实例变量。isa变量用于在运行时识别对象所属的类。

链接 利用一个或多个对象文件并将它们转换成可执行程序的过程。

局部变量 作用域限于定义它的程序块之内的变量。对于方法、函数或语句块局部，变量可以是局部的。

本地化 使程序适合在特定的地理区域内执行的过程，通常是通过将消息转换成本地语言，并处理各种情况（如当地时区、货币符号、日期格式等）实现的。有时本地化只是指语言翻译过程，而术语国际化则表示这一过程的其余方面。

消息 发送给对象（接收者）的方法及相应的参数。

消息表达式 括在一对方括号中的表达式，它指定对象（接收者）和发送给该对象的消息。

方法 属于某个类的过程，通过向该类的对象或实例发送消息，可以执行方法。同时参见类方法和实例方法。

可变对象 值可更改的对象。Foundation框架支持可变和不可变数组、集、字符串和字典。同时参见不可变对象。

nil 一个id类型的对象，用来表示无效对象。它的值定义为0。可向nil发送消息。

通知 当发生特殊事件时，向已注册的可收到警告（通知）的对象发送消息的过程。

NSObject Foundation框架下的根对象。

空字符 值为0的字符。空字符常量用‘\0’表示。

空指针 无效的指针值。通常定义为0。

对象 一组变量和相应的方法。可以向对象发送消息来执行它的方法。

面向对象的程序设计 一种基于类、对象并对对象执行操作的程序设计方法。

父类 被其他类继承的类。也可称作超类。

指针 用于引用另一个对象或数据类型的值。指针在内存中作为特定对象或值的地址来实现。类的实例是一个指针，它指向内存中保存对象数据的位置。

多态 来自不同类的对象可接受同一消息的能力。

预处理程序 首次执行源代码处理行的程序，它以一个#开始，还可能包含特殊的预处理程序语句。常见的用途是使用#define来定义宏指令，包括用#import和#include导入其他源文件，以及用#if、#ifdef和#ifndef有条件地包含源程

序行。

过程式程序设计语言 使用过程和函数定义程序的语言，过程和函数可操作一组数据。

属性声明 这种方法可指定实例变量的属性，允许编译器为实例变量生成无内存泄漏并且线程安全的存取器方法。属性声明也可用于声明存取器方法的属性，这些方法在运行时动态加载。

属性列表 使用标准化的和可移植的格式的不同类型对象的表示。通常，属性列表以XML格式进行存储。

协议 为了符合协议或采用协议而必须实现的方法列表。协议提供了跨多个类标准化接口的方式。同时参见正式协议和非正式协议。

接收者 消息发送给的对象。可以从调用的方法内部作为self来引用接收者。

引用计数 参见保持计数。

保持计数 关于引用对象次数的计数。通过向该对象发送retain消息对其执行加1操作，发送release消息执行减1操作。

根对象 位于继承层次结构中最顶层且没有父类的对象。

运行时 程序正在执行的那段时间；也指负责执行程序指令的机制。

选择程序（selector） 用于选择对象要执行的方法名称。编译的选择程序是SEL类型的，并且可用@selector指令生成。

self 在方法内使用的变量，用于引用消息的接收者。

集 单值对象的无序集合，可使用NSSet、NSMutableSet和NSCountedSet类在Foundation下实现。

赋值方法 这种存取器方法可设置实例变量的值。同时参见取值方法。

语句 以分号结束的一个或多个表达式。

语句块 括在一对卷曲花括号内的一条或多条语句。局部变量可以在语句块内声明，而它们的作用域也会限制在该语句块中。

静态函数 使用关键字static声明的函数，只能由定义在同一源文件中的其他函数或方法调用它。

静态类型 在编译时显式地识别对象所属的类。同时参见动态类型。

静态变量 其作用域限制在定义它的块或模块内的变量。静态变量具有默认的初始值0，且在方法或函数的调用过程中会保持它们的值。

结构 一种集合数据类型，它包含不相同类型的成员。可将结构赋值给其他结构，作为参数传递给函数和方法，还可由函数和方法返回。

子类（subclass） 也称为“child class”，子类从它的父类或超类继承方法和实例变量。

super 方法中使用的关键字，用于引用接收者的父类。

超类 特定类的父类。同时参见super。

合并方法 编译器自动创建的一种赋值方法或取值方法。Objective 2.0语言中添加了这个方法。

UIKit 在iPhone和iTouch上开发应用程序的框架。除了提供可使用普通UI元素（如窗口、按钮和标签）的类以外，它还定义了处理设备特定功能的类型，如加速计和触摸界面。UiKit是Cocoa Touch的一部分。

Unicode字符 一种用于表示字符的标准，它来自包含数百万的字符的集。可使用NSString和NSMutableString类来处理包含在Unicode字符中的字符串。

联合 一种与结构类似的集合数据类型，它包含的成员共享一个存储区。任意时间只有一个成员可以占用此存储区。

Xcode 一种用于程序开发的编译和调试工具，Mac OS X附带提供。

XML 可扩展标记语言。它是Mac OS X上生成的属性类表的默认格式。

存储区（zone） 为分配数据和对象而指定的内存区域。一个程序可以使用多个存储区以便更有效地管理内存。

附录B　Objective-C 2.0语言概览

本附录以一种便于快速参考的方式总结了Objective-C语言。它不是这门语言的完整定义，而更像是该语言特性的非正式描述。读完本书正文之后，应该详细地阅读本附录中的资料。这样做不仅可巩固已学的内容，而且还可使你对Objective-C有一个更好的整体性理解。

这个概览基于ANSI C99（ISO/IEC 9899：1999）标准和Objective-C 2.0语言扩展。在编著本书时，在我的Mac OS X v10.5.5系统上，GUN gcc编译器的最新版本是4.0.1。

B.1　连字字符和标识符

B.1.1　连字字符

以下特殊的两字符序列（连字）等价于列出的单字符标点等价：

连字	含义	连字	含义
<:	[	%>	}
:>	]	%:	#
<%	{	%:%	##

B.1.2　标识符

Objective-C中的标识符包括：字母序列（大写或小写）、通用字符名称（1.2.1）、数字或下划线字符。标识符的第一个字符必须是字母、下划线或通用字符名称。在外部名称中，标识符的前31个字符很重要，而在内部标识符或宏名称中，前63个字符很重要。

1. 通用字符名称

通过位于4个十六进制数字之后的字符\u或8个十六进制数字之后的字符\U，可形成通用字符名称。如果标识符的第一个字符是利用通用字符指定的，它的值就不可能是数字字符。当通用字符在标识符名称中使用时，不能同时指定值小于$A0_{16}$（24_{16}、40_{16}或60_{16}除外）的字符或范围从$D800_{16}$一直到$DFFF_{16}$（包括$D800_{16}$和$DFFF_{16}$在内）的字符。

通用字符名称可用于标识符名称、字符常量和字符串。

2. 关键字

下面列出的标识符是关键字，它在Objective-C编译器中具有特殊的含义。

```
_Bool
_Complex
_Imaginary
auto
break
bycopy
```

```
byref
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
in
inline
inout
int
long
oneway
out
register
restrict
return
self
short
signed
sizeof
static
struct
super
switch
typedef
union
unsigned
void
volatile
while
```

3. 指令

编译器指令以@符号开始，特别用在使用类和对象时。表B-1对这些指令做了总结。

4. 预定义标识符

表B-2列出了Objective-C程序中具有特殊含义的标识符。

表B-1 编译器指令

指 令	含 义	例 子
@"chars"	实现常量NSSTRING字符串对象（相邻的字符串已连接）	NSString *url = @"http://www.kochan-wood.com";
@class c1, c2,...	将c1、c2……声明为类	@class Point, Rectangle;
@defs (class)	为class返回一个结构变量的列表	struct Fract { @defs(Fraction); } *fractPtr; fractPtr = (struct Fract *) [[Fraction alloc] init];
@dynamic names	用于names的存取器方法，可动态提供	@dynamic drawRect;
@encode (type)	将字符串编码为type类型	@encode (int *)
@end	结束接口部分、实现部分或协议部分	@end
@implementation	开始一个实现部分	@implementation Fraction;
@interface	开始一个接口部分	@interface Fraction:Object <Copying>
@private	定义一个或多个实例变量的作用域	参见“实例变量”
@protected	定义一个或多个实例变量的作用域	
@public	定义一个或多个实例变量的作用域	
@property (list) names	为names声明list中的属性	property (retain, nonatomic) NSSTRING *name;
@protocol	为指定的protocol创建一个Protocol对象	@protocol (Copying)]){...} if ([myObj conformsTo: (protocol)
@protocol name	开始name的协议定义	@protocol Copying
@selector (method)	指定method的SEL（选择）对象	if ([myObj respondsTo: @selector (allocF)]) {...}
@synchronized (object)	通过单线程开始一个块的执行。Object已知是一个互斥（mutex）的旗语	
@synthesize names	为names生成存取器方法，如果未提供的话	@synthesize name, email;参见“实例变量”
@try	开始执行一个块，以捕捉异常	参见“异常处理”
@catch (exception)	开始执行一个块，以处理exception	
@finally	开始执行一个块，不管上面的@try块是否抛出异常都会执行	
@throw	抛出一个异常	

表B-2 特殊的预定义标识符

标识符	含 义
_cmd	在方法内自动定义的本地变量，它包含该方法的选择程序
__func__	在函数内或包含函数名或方法名的方法内自动定义的本地字符串变量
BOOL	Boolean值，通常以YES和NO的方式使用
Class	类对象类型
id	通用对象类型
IMP	指向返回id类型值的方法的指针
nil	空对象
Nil	空类对象
NO	定义为(BOOL) 0

（续）

标识符	含 义
NSObject	定义在<Foundation/NSObject.h>中的根Foundation对象
Protocol	存储协议相关信息的类的名称
SEL	已编译的选择程序
self	在用于访问消息接收者的方法内自动定义的本地变量
super	消息接收者的父类
YES	定义为(BOOL) 1

B.2 注释

有两种方法可向程序中插入注释。注释可以以两个字符//开始，在这种情况下，编译器会忽略该行这两个字符之后的任何字符。

注释还可以以两个字符/*开始，并在遇到字符*/时结束。任何字符都可包含到这种注释内，它可扩展到多个程序行。注释可用在程序中允许使用空格的任何位置。然而，注释不能嵌套使用，这意味着不管你使用了多少/*字符，只要遇到第一个*/字符，注释就会结束。

B.3 常量

B.3.1 整型常量

整型常量是数字序列，前面可选地带有正号或负号。如果第一个数字为0，这个整数将被看作八进制常量，在这种情况中，它后面的所有数字必须在0-7之间。如果第一个数字是0且紧随其后的是字母x（或X），这个整数将作为十六进制常量，且后面的数字可以在范围0-9或a-f（或A-F）之间。

后缀字母l或L可以添加到十进制整型常量的末尾，使其成为long int常量。如果该值对于long int不适合，将作为long long int处理。如果将后缀字母l或L添加到八进制或十六进制常量的末尾，这些常量将在适合时作为long int，而在不适合时作为long long int。最后，如果它还不适合long long int，就会被看作unsigned long long int常量。

将后缀字母ll或LL添加到十进制整型常量的末尾可使其成为long long int。添加到八进制或十六进制常量的末尾时，这些常量将被首先看作long long int，如果此时不适合，就会被看作unsigned long long int常量。

后缀u或U可以添加到整型常量的末尾，使其成为unsigned。如果常量太大，不适合unsigned int，就会被看作unsigned long int。如果它大大超过了unsigned long int，它将被看作unsigned long long int。

unsigned和long后缀可同时添加到整型常量，以使其成为unsigned long int。如果常量太大不适合unsigned long int，它将作为unsigned long long int处理。

unsigned和long-long后缀可同时添加到整型常量，使其成为unsigned long long int。

如果无后缀的十进制整型常量太大，不适合有符号的int，它将被看作long int。如果它太大不适合long int，它将被看作long long int。

如果无后缀的八进制或十六进制整型常量太大不适合有符号的int，它将被看作unsigned int。如果它太大不适合unsigned int，它将被看作long int，如果它还是太大不适合long int，它将被看作unsigned long int。如果它太大不适合unsigned long int，它将被看作long long int。最后，如果它还太大不适合long long int，这个常量就会被作为unsigned long long int来处理。

B.3.2 浮点常量

浮点常量由十进制数字序列、小数点和另一个十进制数字序列组成。负号可以放在值的前面来表示负值。此外，既可以省略小数点前的数字序列，也可以省略小数点后的数字序列，但两者不能同时省略。

如果浮点常量后面紧跟字母e（或者E）和一个可选的符号整数，这个常量就是以科学计数法表示的。这个整数（指数）表示10的幂，它将与字母e（尾数）开始的值相乘（例如，1.5e-2表示$1.5 \cdot 10^{-2}$或者0.015）。

十六进制浮点常量依次由开始的0x或0X、一个或多个十进制或十六进制数字、p或者P和一个有符号的二进制指数组成。例如，0x3p10表示值$3 \cdot 2^{10}$。浮点常量将被编译器作为double精度的值来处理。可以为其添加后缀字母f或F，以指定float常量而非double常量，还可为其添加后缀l或L以指定long double常量。

B.3.3 字符常量

括在单引号内的字符是字符常量。如何处理单引号内包含多个字符的情况是实现定义的(implementation-defined)。通用字符可使用在字符常量中，以指定标准字符集中不包含的字符。

1. 转义序列

通过反斜杠字符可以识别或者引入特殊的转义字符。下面列出了这些转义字符：

字 符	含 义	字 符	含 义
\a	声音警报	\"	双引号
\b	退格	\'	单引号
\f	换页	\?	问号
\n	换行	\nnn	八进制字符值
\r	回车	\unnnn	通用字符名称
\t	水平制表符	\Unnnnnnnn	通用字符名称
\v	垂直制表符	\xnn	十六进制字符值
\\	反斜杠		

在八进制字符中，可以指定1个到3个八进制数字。在最后3种情况内，可以使用十六进制数字。

2. 宽字符常量

宽字符常量被写成L'x'。这种常量的类型是wchar_t，如在标准头文件<stddef.h>中定义的那样。宽字符常量提供了一种表示字符集中字符的方法，而这个字符集不能完全使用普通的char类型来表示。

B.3.4 字符串常量

括在双引号内的0个或多个字符序列表示字符串常量。字符串内可以包含任何合法的字符，包括前面列出的任何转义字符。编译器将在字符串末尾自动插入一个空字符（'\0'）。

通常，编译器会生成一个指向字符串中第一个字符的指针，且类型为“char指针”。然而，当字符串常量与sizeof运算符一起使用来初始化字符数组，或者与&运算符一起使用时，字符串常量的类型为“char数组”。

程序不能修改字符串常量。

1. 字符串连接

预处理程序会自动将相邻的字符串常量连接到一起。字符串可通过0或多个空格字符来分隔。因此，下面3个字符串

```
"a" " character "
"string"
```

在连接后将等价于单个字符串

```
"a character string"
```

2. 多字节字符

实现定义的字符序列可以在字符串内的不同状态之间进行转换，以便可以包含多字节字符。

3. 宽字符串常量

扩展字符集中的字符串常量可使用格式L“...”来表示。这种常量的类型是“wchar_t指针”，其中wchar_t定义在<stddef.h>中。

4. 常量字符串对象

常量字符串对象可通过在常量字符串前面放置一个@字符来创建。这种对象的类型是NSConstantString。可以将相邻的字符串常量连接到一起，所以下面3个字符串

```
@"a" @" character "
  @"string"
```

在连接后将等价于单个字符串

```
@"a character string"
```

B.3.5 枚举常量

定义为枚举类型值的标识符被看作这种特殊类型的常量；否则，编译器将把这个标识符作为类型int处理。

B.4 数据类型和声明

本节总结了基本数据类型、派生数据类型、枚举数据类型和typedef。本节中还总结了声明变量的格式。

B.4.1 声明

定义特定的结构、联合、枚举数据类型或typedef时，编译器不会自动保留任何内存。定义只通知编译器有关的特定数据类型及其相关的名称（可选地）。这种定义既可放在函数和方法的内部也可放在外部。在前一种情况中，只有函数或方法知道它的存在；后一种情况中，文件其余部分都知道它的存在。

在定义之后，可将变量声明为这个特殊的数据类型。变量可以被声明成任何一种数据类型，而且系统还将为其保留内存；除非变量是用extern声明的，在这种情况下，可能会为它分配内存，也可能不会（参见“存储类和作用域”一节）。

这门语言还能在定义的特定结构、联合或枚举数据类型分配内存。只要在定义的结束分号之前列出这些变量，即可实现这一点。

B.4.2 基本数据类型

表B-3总结了基本的Objective-C数据类型。使用以下格式，可将变量声明为特定的数据类型：

```
type name = initial_value;
```

表B-3 基本数据类型总结

类 型	含 义
int	整数值，也就是不包含小数点的值；保证包含至少32位的精度
short int	精度减少的整数值；在一些机器上占用的内存是int的一半；保证至少包含16位的精度
long int	精度扩展的整数值；保证包含至少32位的精度
long long int	精度扩展的整数值；保证包含至少64位的精度
unsigned int	正整数值，能存储的最大整数值是int两倍；保证包含至少32位的精度
float	浮点值；就是，可以包含小数位的值；保证包含至少6位数字的精度
double	精度扩展的浮点值；保证包含至少10位数字的精度
Long double	具有附加扩展精度的浮点值；保证包含至少10位数字的精度
char	单个字符值，在某些系统上，在表达式中使用它时可以发生符号扩展
unsigned char	除了它能确保作为整型提升的结果不会发生符号扩展之外，与char相同
signed char	除了它能确保作为整型提升的结果会发生符号扩展之外，与char相同
_Bool	Boolean类型；它足够存储值0和1
float _Complex	复数
double _Complex	具有扩展精度的复数
long double _Complex	具有附加扩展精度的复数
void	无类型，用于确保在需要返回值时不使用那些不返回值的函数或方法，或者显式地抛弃表达式的结果；还可用于一般指针类型（void *）

给变量指派初始值是可选的，而且它遵守“变量”一节中总结的规则。使用下面的一般格式，可同时声明多个变量：

```
type name = initial_value, name = initial_value, .. ;
```

在类型声明之前，还可以指定一个可选的存储类，如同“变量”一节中所总结的。如果指定了存储类而且变量的类型为int，那么可以忽略int。例如

```
static counter;
```

将counter声明为static int变量。

注意，修饰符signed还可以放在short int、int、long int和long long int类型的前面。因为这些类型在默认情况下是带符号的，这对它们没有影响。

_Complex和_Imaginary允许声明和操纵复数和虚数，它们具有该库中支持这些类型进行运算的函数。通常，应该将文件<complex.h>包含到程序中，该文件为使用复数和虚数定义了宏指令并声明了函数。例如，使用下面的语句，可将c1声明为double_Complex变量并初始化为值5 + 10.5i：

```
double _Complex c1 = 5 + 10.5 * I;
```

然后，可使用creal和cimag这样的库例程来分别提取c1的实部和虚部。

并不需要一个实现来支持类型_Complex和_Imaginary，并且它可以可选地支持其中的一个类型，但不是另一个。

B.4.3 派生数据类型

派生数据类型是根据一个或多个基本数据类型构建的数据类型。派生数据类型包括数组、结构、联合和指针（它包含对象）。还可将返回特定类型值的函数或方法看作派生的数据类型。除了函数和方法以外，其他所有派生数据类型都在下面的各段中进行了总结。在“函数”和“类”两节中会分别涉及到作为派生数据类型的函数和方法。

1. 数组

（1）一维数组

可以定义数组使其包含任何一种数据类型或者任何一种派生数据类型。包含函数的数组是不允许的（尽管允许包含函数指针的数组）。

数组声明的基本形式如下：

```
type name[n] = { initExpression, initExpression, .. };
```

数组名中的表达式*n*确定元素个数，通过提供一列指定的初始值可以省略表达式n。在这种情况下，数组的大小是由列出的初始值个数决定的，在使用指定的初始方法时，是基于被引用元素的最大索引来决定。

如果定义的是全局数组，那么初始值必须是常量表达式。当初始值的个数少于数组中元素的个数时，可将初始值列在初始化列表中，但如果情况相反，则不能采用这种方法。如果指定了较少的值，就会只初始化这么多数组元素——其余的元素将设置为0。

数组初始化的特殊情况出现在字符数组中，即字符数组可以使用常量字符串来初始化。例如

```
char today[] = "Monday";
```

将today声明为一个字符数组。这个数组分别用字符‘M’、‘O’、‘n’、‘d’、‘a’、‘y’和‘\0’来初始化。

如果你显式地指定字符数组的维数，并且没有为结尾的空字符保留空间，编译器就不会在数组末尾放置空字符：

```
char today[6] = "Monday";
```

这条语句将today声明为一个包含6个字符的数组，并将其元素分别设为字符‘M’、‘o’、\&‘n’、‘d’、‘a’和‘y’。

通过将数组成员括在一对方括号中，可以用任何顺序初始化特定的数组元素。例如

```
int x = 1233;
int a[] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

定义了一个包含10个元素的数组a（基于数组中的最大索引），并将最后一个元素初始化为值x + 1（1234），前3个元素分别初始化为1、2和3。

(2) 长度可变数组

在函数、方法或程序块内，可以使用包含变量的表达式来确定数组的维数。这种情况下，数组的大小将在运行时计算。例如，函数

```
int makeVals (int n)
{
   int valArray[n];
 ...
}
```

定义了一个名为valArray的自动数组，它包含n个元素，而n将在运行时计算，并且可以在函数调用中改变。长度可变的数组不能被初始化。

(3) 多维数组

声明多维数组的一般格式如下：

```
type name[d1][d2]...[dn] = initializationList;
```

数组名定义为包含特定类型的d1 x d2 x...x dn个元素。例如

```
int three_d [5][2][20];
```

定义了一个3维数组three_d，该数组包含200个整数。

通过将每一维的指定下标括在该维的方括号中，可以引用多维数组中的特定元素。例如，语句

```
three_d [4][0][15] = 100;
```

将100存储到数组three_d的指定元素。

多维数组初始化的方式可以和一维数组相同。可以使用嵌套的花括号来控制数组中元素的赋值。

下面将matrix声明为一个包含4行3列的二维数组：

```
int matrix[4][3] =
      { { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 } };
```

该声明将matrix的第1行元素分别设为1、2和3；第2行元素分别设为4、5和6；第3行分别设为7、8和9。第4行元素设为0，因为没有为那一行指定值。声明

```
int matrix[4][3] =
     { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

将matrix初始化为相同的值，因为多维数组的元素是按维的顺序来初始化的——即，从最左边的维到最右边的维。

声明

```
int matrix[4][3] =
      { { 1 },
        { 4 },
        { 7 } };
```

将第1行的第1个元素设为1，第2行的第1个元素设为4，第3行的第1个元素设为7。其余所有的元素按默认值设为0。

最后，声明

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

将指出的数组元素初始化为特定的值。

2. 结构

一般格式：

```
struct name
{
   memberDeclaration
   memberDeclaration
...
} variableList;
```

结构name被定义为包含由memberDeclaration指定的成员。每一个这样的声明都包括位于一个或多个成员名列表之后的类型说明。

定义结构的同时可以声明变量，通过在结束分号之前列出它们或者使用以下格式随后声明它们即可：

```
struct name variableList;
```

如果在定义结构时省略name，就不能使用这种格式。在这种情况下，该结构类型的所有变量必须使用定义来声明。

声明结构变量的格式与声明数组变量的格式类似。结构的成员可以通过将初始值列表括在一对花括号内的方式来声明。如果定义的是全局结构，那么列表中的值必须是常量表达式。

声明

```
struct point
{
   float x;
   float y;
} start = {100.0, 200.0};
```

定义了一个名为point的结构和一个名为start的struct point变量，该变量具有指定的初始值。

可以使用表示法

```
.member = value
```

在初始化列表中以任何顺序为初始化指定特定成员，如

```
struct point end = { .y = 500, .x = 200 };
```

中一样。

声明

```
struct entry
{
  char *word;
  char *def;
} dictionary[1000] = {
   { "a",     "first letter of the alphabet" },
   { "aardvark", "a burrowing African mammal" },
   { "aback",    "to startle"          }
};
```

声明了包含1000个entry结构的dictionary，且它的前3个元素被初始化为特定的字符串指针。使用指定的初始化方法，还可将上述定义写成以下形式：

```
struct entry
{
  char *word;
  char *def;
} dictionary[1000] = {
   [0].word = "a",     [0].def = "first letter of the alphabet",
   [1].word = "aardvark", [1].def = "a burrowing African mammal",
   [2].word = "aback",  [2].def = "to startle"
};
```

或等价地声明为：

```
struct entry
{
   char *word;
   char *def;
} dictionary[1000] = {
  {  {.word = "a",     .def = "first letter of the alphabet" },
     {.word = "aardvark", .def = "a burrowing African mammal"},
     {.word = "aback",  .def = "to startle"}
};
```

可将自动结构变量初始化为相同类型的另一个结构，如下：

```
struct date tomorrow = today;
```

这条语句声明了date结构变量tomorrow，并将（以前声明的）date结构变量today的内容指派给该变量。

具有格式

```
type fieldName : n
```

的memberDeclaration在结构内定义了n位宽的field，其中n是一个整数。在某些计算机上，字段是可以从左到右组装的，而在其他计算机上则是从左到右组装的。如果省略fieldName，位的特定成员将会保留下来，但不能引用它。如果省略fieldName并且n是0，那么随后的字段将会对与下一个存储unit的边界对齐，而unit是实现定义的。字段的类型可以是int、signed int或unsigned int。不管字段被看作带符号或无符号，它都是实现定义的。地址运算符（&）不能应用于字段，并且不能定义字段的数组。

3. 联合

一般格式：

```
union name
{
   memberDeclaration
   memberDeclaration
   ...
} variableList;
```

这定义了一个名为name的联合，它的成员通过每一个memberDeclaration来指定。联合的每一个成员共享重叠的存储空间，编译器确保了保留足够大的空间以包含联合中的最大成员。

变量可以在定义联合时声明，或者随后使用以下表示法来声明

```
union name variableList;
```

来声明，前提是在定义联合时提供了联合命名。

确保从联合内检索的值与最后一个存储在联合内的值一致，这个工作是由程序员负责完成的。联合的第一个成员可以通过将其初始值（在全局联合变量的情况中，该初始值必须是常量表达式）放在一对花圆括号内来初始化：

```
union shared
{
long long int l;
long int w[2];
} swap = { 0xffffffff };
```

用指定成员名的方式取而代之，可以初始化不同的变量，如同在

```
union shared swap2 = {.w[0] = 0x0, .w[1] = 0xffffffff};
```

中一样。

这条语句声明了联合变量swap并将它的第1个成员设为十六进制值ffffffff。

自动联合变量还可以初始化为相同类型的联合，如

```
union shared swap2 = swap;
```

中一样。

4. 指针

声明指针变量的基本格式如下：

```
type *name;
```

标识符name声明为“指向type的指针”类型，该类型可以是基本数据类型也可以是派生数据类型。例如

```
int *ip;
```

将ip声明为指向int的指针，而声明

```
struct entry *ep;
```

将ep声明为一个指向entry结构的指针。如果将Fraction定义为一个类，则声明

```
Fraction *myFract;
```

将myFract声明为Fraction类型的对象，或者更明确地讲，当创建指派给该变量的对象实例之后，myFract将用于保存指向对象数据结构的指针。

指向数组中元素的指针声明为指向数组中包含的元素类型。例如，上述ip的声明也可以用来声明指向整型数组的指针。

同样允许更高级的指针声明格式。例如，声明

```
char *tp[100];
```

将tp声明为一个包含100个字符指针的数组，而声明

```
struct entry (*fnPtr) (int);
```

将fnPtr声明为一个指向函数的指针，该函数返回一个entry结构并接受单个int参数。

通过将指针与一个值为0的常量表达式进行比较，可以测试它是否为空。实现可以选择使用非0值来内部表示一个空指针。然而，这样内部表示的空指针和值为0的常量之间的比较必须是相等的。

指针转换成整数和整数转换成指针所采用的方式依赖于计算机，容纳指针所需的整数大小也是一样。

类型“指向void的指针”是通用指针类型。这门语言保证任何类型的指针都能被赋值给一个void指针，并且可在不更改其值的情况下转换为原来类型。

类型id是通用对象指针。任何类的任何对象都可以指派为一个id变量，反之亦然。

与这两种特殊情况不同，不允许指定不同的指针类型，如果尝试这样做，编译器通常会产生一条警告消息。

B.4.4 枚举数据类型

一般格式：

```
enum name { enum_1, enum_2, .. } variableList;
```

利用枚举值enum_1、enum_2……来定义枚举类型name，每一个枚举值都是一个标识符或者后面带有等号或常量表达式的标识符。variableList是声明为类型enum name的变量的可选列

表（具有可选的初始值）。

编译器向枚举标识符指派以0开始的序数。如果标识符的后面是=和一个常量表达式，那么表达式的值就指派给该标识符。随后的标识符将从常量表达式加1开始赋值。编译器将把枚举标识符作为常量整数处理。

如果想要将变量声明为上述定义（并命名）的枚举类型，可以使用下面的构造：

```
enum name variableList;
```

声明为特定枚举类型的变量可以只为其指派一个相同数据类型的值，尽管编译器可能不会将这种情况标记为错误。

B.4.5 typedef

typedef语句用于为基本或派生数据类型指派新的名称。typedef没有定义新类型，而仅仅是为现有类型指派的新名称。因此，编译器对声明为这个新类型的变量的处理方式就像将其声明为和这个新名称关联的类型一样。

在形成typedef的定义时，就像进行了常规变量声明一样继续处理。然后，在通常出现变量名称的位置放置新类型名称。最后，在这些语句前面，放置关键字typedef。

作为例子，

```
typedef struct
{
   float x;
   float y;
} POINT;
```

将名称POINT与包含两个浮点型成员x和y的结构关联起来。随后可以将变量声明为POINT类型，例如：

```
POINT origin = { 0.0, 0.0 };
```

B.4.6 类型修饰符：const、volatile和restrict

关键字const可以放在类型声明之前，通知编译器这个值不能被修改。因此，声明

```
const int x5 = 100;
```

将x5声明为一个整数常量（也就是，在程序执行期间，不再给它指派其他任何值）。编译器无须标记更改const变量值的尝试。

修饰符volatile显式地告知编译器值的更改（通常是动态地）。在表达式中使用volatile变量时，它的值是从该变量出现的任何位置访问的。

要将port17声明为“指向char的volatile指针”类型，可以编写下面的程序行：

```
char *volatile port17;
```

关键字restrict可以和指针一起使用。它提示编译器进行优化（类似于变量的register关键字）。关键字restrict向编译器说明指针将是特定对象的惟一引用，也就是，该变量不再被同一作用域

内的其他任何变量引用。程序行

```
int * restrict intPtrA;
int * restrict intPtrB;
```

告知编译器，在定义intPtrA和intPtrB的作用域范围内，它们将永远不会访问同一个值。它们用于指向整数时（例如，在数组中）是相互排斥的。

B.5 表达式

变量名、函数名、消息表达式、数组名、常量、函数调用、数组引用以及结构和联合引用都可以看作表达式。将一元运算符（在合适的位置）应用到其中一个表达式中的其中，它仍是一个表达式，用二元或三元运算符将两个或多个这样的表达式组合到一起也是这样。最后，括在圆括号内的表达式仍然是表达式。

标识一个数据对象的任何非void类型的表达式名为左值（lvalue）。如果可以为它指派值，它是所谓的可修改左值。

可修改的左值表达式在特定的地方是必需的。位于赋值运算符左侧的表达式必须是一个可修改的左值。一元地址运算符只能应用于可修改的lvalue或函数名。最后，自加自减运算符只能应用于可修改的左值。

B.5.1 Objective-C运算符总结

表B-4总结了Objective-C语言中的各种运算符。这些运算符按其优先级降序列出，组合在一起的运算符具有相同的优先级。

表B-4 Objective-C运算符总结

运算符	描 述	结合性
()	函数调用	从左到右
[]	数组元素引用或者消息表达式	
->	指向结构成员引用的指针	
.	结构成员引用或方法调用	
-	一元负号	从右到左
+	一元正号	
++	加1	
--	减1	
!	逻辑非	
~	求反	
*	指针引用（间接）	
&	取地址	
sizeof	对象的大小	
(type)	类型强制转换（转换）	
*	乘	从左到右
/	除	
%	取模	

（续）

运算符	描　述	结合性
+	加	从左到右
−	减	
<<	左移	从左到右
>>	右移	
<	小于	
<=	小于等于	从左到右
>	大于	
>=	大于等于	
==	相等性	从左到右
!=	不等性	
&	按位AND	从左到右
^	按位XOR	从左到右
|	按位OR	从左到右
&&	逻辑 AND	从左到右
||	逻辑OR	从左到右
?:	条件	从左到右
= *= /= %= += −= &= ^= |= <<= >>=	赋值运算符	从右到左
,	逗号运算符	从右到左

作为演示如何使用表B-4的例子，考虑下面的表达式：

```
b | c & d * e
```

与按位OR和按位AND运算符相比，乘法运算符有更高的优先级，因为在表B-4中它出现在这两个位运算符之前。类似地，与按位OR运算符相比，按位AND运算符有更高的优先级，因为在表中前者出现在后者的前面。因此，这个表达式将以

```
b | ( c & ( d * e ) )
```

来求值。

现在，考虑下面的表达式：

```
b % c * d
```

因为在表B-4中取模和乘法运算符出现在同一个组，所以它们具有相同的优先级。这些运算符的结合性是从左到右，表示该表达式以如下方式求值：

```
( b % c ) * d
```

来求值。

作为另一个例子，表达式

```
++a->b
```

将以

```
++(a->b)
```

来求值，因为->运算符的优先级比++运算符的优先级高。

最后，因为赋值运算符从右到左结合，所以语句

```
a = b = 0;
```

将以

```
a = (b = 0);
```

来求值，这最终导致a和b的值被设为0。对于表达式

```
x[i] + ++i
```

中，它没有定义编译器将先求加号运算符的左侧的值还是右侧的值。此处，求值的方式会影响结果，因为i的值可能在求x[i]的值之前已经加1。

下面展示另一个在表达式中没有定义求值顺序的例子：

```
x[i] = ++i
```

在这种情况下，没有定义i的值是在x中的索引使用它的值之前还是之后加1。

函数和方法参数的求值顺序也是未定义的。因此，在函数调用

```
f (i, ++i);
```

或消息表达式

```
[myFract setTo: i over: ++i];
```

中，可能首先将i加1，因此导致将同一个值作为两个参数发送给函数或方法。

Objective-C语言确保&&和||运算符按照从左到右的顺序求值。此外，在使用&&时，保证如果第一个运算数为0，就不会求第二个运算数的值；在使用||时，保证如果第一个运算数非0，就不会求第二个运算数的值。在生成表达式时应该记住这个事实，例如

```
if ( dataFlag || [myData checkData] )
...
```

因为，在这种情况下，只有在dataFlag的值是0时才调用checkData。作为另一个例子，如果定义数组对象a包含n个元素，则以

```
if (index >= 0 && index < n && ([a objectAtIndex: index] == 0))
 ...
```

开始的语句只有在index是数组中的有效下标时才引用元素。

B.5.2 常量表达式

常量表达式是每一项都是常量值的表达式。在下列情况中，常量表达式是必需的：

(1) 作为switch语句中case之后的值

(2) 指定数组的大小

(3) 为枚举标识符指派值

(4) 在结构定义中，指定位域的大小

(5) 为外部或静态变量指派初始值

(6) 为全局变量指派初始值

(7) 在#if预处理程序语句中，作为#if之后的表达式

在前4种情况中，常量表达式必须由整数常量、字符常量、枚举常量和sizeof表达式组成。在此只能使用以下运算符：算术运算符、按位运算符、关系运算符、条件表达式运算符和类型强制转换运算符。

在第5和第6种情况中，除了上面提到的规则之外，还可以显式地或隐式地使用取地址运算符。然而，它只能应用于外部或静态变量或函数。因此，假设x是一个外部或静态变量，表达式

```
&x + 10
```

将是合法的常量表达式。此外，表达式

```
&a[10] - 5
```

在a是外部或静态数组时将是合法的常量表达式。最后，因为&a[0]等价于表达式a，所以

```
a + sizeof (char) * 100
```

也是一个合法的常量表达式。

对于最后一种需要常量表达式（在#if之后）的情况，除了不能使用sizeof运算符、枚举常量和类型强制转换运算符以外，其余规则与前4种情况的规则相同。然而，它允许使用特殊的defined运算符（参见“#if指令”一节）。

B.5.3 算术运算符

假设

a、b　　表示除void之外任何数据类型的表达式；

i、j　　表示任何整型数据类型的表达式；

则表达式

−a　　对a的值求反；

+a　　给出a的值；

a + b　　a加b；

a − b　　a减b；

a * b　　a乘以b；

a / b　　a除以b；

i % j　　给出i除以j的余数。

在每个表达式中，都会为运算数执行普通的算术转换（参见“基本数据类型转换”一节）。如果a是无符号的，那么通过首先将a提升为整型，然后从提升类型的最大值减去a，最后将结果加1来计算−a。

如果两个整数相除，其结果将会被截取。如果其中一个运算数为负，那么截取方向将是不确定的（也就是，−3/2将在某些计算机上产生−1，在另一些计算机上产生−2）；否则，截取

的结果将总是朝向0的（3/2总是产生1）。可参见“基本运算和指针”一节中对算术运算和指针的总结。

B.5.4 逻辑运算符

假设

a、b　　表示除void之外任何基本数据类型的表达式，或者表示两个指针；

则表达式

a && b　　在a和b都非0时，值为1；否则值为0（并且只在a非0时才对b求值）；

a || b　　在a或b其中之一非0时，值为1；否则值为0（并且只在a为0时才对b求值）；

! a　　在a为0时，值为1；否则值为0。

对a和b应用普通的算术转换（参见“基本数据类型的转换”一节）。其结果的类型在任何情况都是int。

B.5.5 关系运算符

假设

a、b　　是除void之外任何数据类型的表达式，或者是两个指针；

则表达式

a < b　　在a小于b时，值为1；否则值为0；

a <= b　　在a小于等于b时，值为1；否则值为0；

a > b　　在a大于b时，值为1；否则值为0；

a >= b　　在a大于等于b时，值为1；否则值为0；

a == b　　在a等于b时，值为1；否则值为0；

a != b　　在a不等于b时，值为1；否则值为0。

对a和b执行了普通的算术变换（参见“基本数据类型的转换”一节）。对指针而言，只有a和b都指向同一个数组或同一个结构或联合的成员时，前4个关系测试才是有意义的。其结果的类型在每种情况下都是int。

B.5.6 按位运算符

假设

i、j、n　　是任何整型数据类型的表达式；

则表达式

i & j　　执行i和j的按位AND操作；

i | j　　执行i和j的按位OR操作；

i ^ j　　执行i和j的按位XOR操作；

~I　　对i求反；

i << n　　将i左移n位；

i >> n　　将i右移n位；

上述操作对运算数执行常见的算术变换，但<<和>>除外，这两种情况只是对每一个运算数进行整数提升（参见“基本数据类型的转换”一节）。如果移位计数是负的或者大于等于被移动对象包含的位数，则移位结果是不确定的。在某些计算机上，右移是一种算术运算（填入正负号），而在其他计算机上则是一种逻辑运算（填入0）。移位操作结果的类型就是提升的左侧运算数的类型。

B.5.7 自加和自减运算符

假设

l　　是一个可修改的左值表达式，其类型没有使用const来限定；

则表达式

++l　　将l加1，然后将结果作为表达式的值；

l++　　用l作为表达式的值，然后将l减1；

--l　　将l减1，然后将结果作为表达式的值；

l--　　用l作为表达式的值，然后将l减1；

“基本运算和指针”一节描述了对指针执行这些运算的情况。

B.5.8 赋值运算符

假设

l　　是一个可修改的左值表达式，它的类型没有使用const来限定；

op　　是可以用作赋值运算符的任何运算符（参见表B-4）；

a　　是一个表达式；

则表达式

l = a　　将a的值存储到l中；

l op= a　　对l和a应用op，再将结果存储到l中。

在第一个表达式中，如果a是一个基本数据类型（除void之外），它将被转换以匹配l的类型。如果l是一个指针，则a必须是一个与l类型相同的指针、void指针或空指针。

如果l是void指针，则a可以是任何类型的指针。第二个表达式除了只对l求值一次之外，可写为l = l op (a)并按此情况处理（考虑x[i++] += 10）。

B.5.9 条件运算符

假设

a、b、c　　为表达式；

则表达式

a ? b : c　　在a非0时，值为b；否则为c。仅表达式b或c其中之一被求值。

表达式b和c必须具有相同的数据类型。如果它们的类型不同，但都是算术数据类型，就要对其执行常见的算术转换以使其类型相同。如果一个是指针，另一个为0，则后者将被看作是与前者具有相同类型的空指针。如果一个是指向void的指针，另一个是指向其他类型的指针，

则后者将被转换成指向void的指针并作为结果类型。

B.5.10 类型强制转换运算符

假设

type 表示一个基本数据类型、枚举数据类型（前面以关键字enum开始）、typedef定义的类型或者派生数据类型的名称；

a 是一个表达式；

则表达式

(type)a 将a转换成指定类型。

注意，用在方法声明或定义中的括在圆括号中的类型并不是类型强制转换运算符的应用例子。

B.5.11 sizeof运算符

假设

type 与上一节描述的一样；

a 是一个表达式；

则表达式

sizeof(type) 包含特定类型值所需的字节数；

sizeof a 保存a的求值结果所必需的字节数；

如果type为char，则结果将被定义为1。如果a是（显式地或者通过初始化隐式地）维数确定的数组名称，而不是形参或未确定维数的数组名称，那么sizeof a会给出将元素存储到a中必需的位数。

如果a是一个类名，则sizeof (a)会给出保存a的实例所必需的数据结构大小。

通过sizeof运算符产生的整数类型是size_t，它在标准头文件<stddef.h>中定义。

如果a是长度可变的数组，那么在运行时对表达式求值，否则在编译时求值，因此它可以用在常量表达式中（参见“常量表达式”一节）。

B.5.12 逗号运算符

假设

a、b 是表达式；

则表达式

a, b 导致对a求值，再对b求值。表达式的类型和值即为b的类型和值。

B.5.13 数组的基本运算

假设

a 被声明为一个包含n个元素的数组；

i 是任何整型数据类型的表达式；

v 是一个表达式；

则表达式

a[0]	引用a的第一个元素；
a[n－1]	引用a的最后一个元素；
a[i]	引用a的第i个元素；
a[i] = v	将v的值存储到a[i]中。

在每种情况下，结果的类型都是包含在a中元素的类型。参见B.5.15节，它总结了对指针和数组执行的运算。

B.5.14　基本运算和结构

注意　这些内容同样适用于联合。

假设

x	是struct s类型的可修改左值表达式；
y	是struct s类型的表达式；
m	是结构s中一个成员的名称；
obj	是任何一个对象；
M	是任何一个方法；
v	是一个表达式；

则表达式

x	引用整个结构并且它是struct s类型的；
y.m	引用结构y的成员m而且它的类型是为成员m声明的类型；
x.m = v	把v指派给x的成员m而且类型是为成员声明m的类型；
x = y	将y赋值给x，该表达式是struct s类型；
f (y)	调用函数f，作为参数传递结构y的内容（在f中，形参必须被声明为struct s类型）；
[obj M: y]	调用对象obj的方法M，同时作为参数传递结构y的内容（在方法内，必须将参数声明为struct s类型）；
return y;	返回结构y（为函数或方法声明的返回类型必须是struct s）；

B.5.15　基本运算和指针

假设

x	是t类型的左值表达式；
pt	是“指向t的指针”类型的可修改左值表达式；
v	是一个表达式；

则表达式

&x	产生一个指向x的指针，其类型为“指向t的指针”；
pt = &x	设置pt指向x，其类型为“指向t的指针”；
pt = 0	为pt赋值空指针；

pt == 0	测试pt是否为空；
*pt	引用pt指向的值，pt的类型为t；
*pt = v	将v的值存储到pt指向的位置中，pt的类型为t；

1. 指向数组的指针

假设

a	是一个元素类型为t的数组；
pa1	是一个“指向t的指针”类型的可修改左值表达式，它指向a中的一个元素；
pa2	是一个“指向t的指针”类型的左值表达式，它指向a中的一个元素或指向a中结束的最后一个元素；
v	是一个表达式；
n	是一个整型表达式；

则表达式

a、&a、&a[0]	每个都产生一个指向第一个元素的指针；
&a[n]	产生一个指向a中元素成员n的指针，其类型为“指向t的指针”；
*pa1	引用pa1指向的a中的元素，其类型为t；
*pa1 = v	将v的值存储到pa1指向的元素，其类型为t；
++pa1	不论a中包含的元素是哪种类型，都会设置pa1指向a的下一个元素，其类型为“指向t的指针”；
--pa1	不论a中包含的元素是哪种类型，都设置pa1指向a的前一个元素，其类型为“指向t的指针”；
*++pa1	将pa1加1，再引用pa1所指的a中的值，其类型为t；
*pa1++	在pa1加1之前引用pa1所指向的a的值，其类型为t；
pa1 + n	产生一个指针，它指向数组a中pa1所指元素之后的第n个元素，其类型为“指向t的指针”；
pa1 − n	产生一个指针，它指向数组中pa1所指元素之前的第n个元素，其类型为“指向t的元素”；
*(pa1 + n) = v	将v的值存储到pa1 + n所指的元素中，其类型为t；
pa1 < pa2	测试是否pa1所指元素位于pa2所指元素之前（任何关系运算符都可以用来比较两个指针）。
pa2 − pa1	产生包含在指针pa2和pa1之间的元素个数（假设pa2所指元素位于pa1所指元素之前），其类型为整型；
a + n	产生一个指向a的元素成员n的指针，其类型为“指向t的指针”，而且在各个方面都等价于&a[n]；
*(a + n)	引用a的元素成员n，其类型为t，而且在各方面都等价于a[n]。

两个整数相减所产生的实际整数类型由ptrdiff_t指定，它定义在标准头文件<stddef.h>中。

2. 结构指针

假设

x	是一个struct s类型的左值表达式；
ps	是一个“指向的指针”类型的可修改左值表达式；
m	是结构的一个成员的名称，且类型为t；
v	是一个表达式；
则表达式	
&x	产生一个指向x的指针，其类型为“指向struct s的指针”；
ps = &x	设置ps使其指向x，其类型为“指向struct s的指针”；
ps->m	引用ps所指的结构的成员m，其类型为t；
(*ps).m	同样引用这个成员，且各方面都等价于表达式ps->m；
ps->m = v	将v的值存储到ps所指的结构的成员m中，其类型为t；

B.5.16 复合常量

复合常量是括在圆括号中的类型名称，之后跟有初始化列表。它创建一个未命名的具有特定类型的值，该值的作用域限于定义它的程序块内，如果在任何程序块之外定义它，则具有全局作用域。在后一种情况下，复合常量的初始值必须全都是常量表达式。

作为一个例子，

```
(struct point) {.x = 0, .y = 0}
```

是一个用于产生结构的表达式，该结构的类型为struct point并且具有指定的初始值。这个表达式可以赋值给另一个struct point结构，如下：

```
origin = (struct point) {.x = 0, .y = 0};
```

此外，还可将它传递给期望接受struct point参数的函数或方法，如下：

```
moveToPoint ((struct point) {.x = 0, .y = 0});
```

还可定义非结构的其他类型，如，如果intPtr是int *类型的，则语句

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

（这条语句可以出现在程序的任何位置）设置intptr使其指向一个包含100个整数的数组，它的3个元素初始化为特定的值。

如果没有指定数组的大小，则是由初始化函数列表确定的。

B.5.17 基本数据类型的转换

Objective-C语言以一种预定义的顺序转换算术表达式中的运算数，这种转换称为普通算术转换：

（1）如果其中一个运算数的类型为long double，则其他运算数将会转换为long double并以此作为结果类型。

（2）如果其中一个运算数的类型为double，则其他运算符将会转换为double并以此作为结果类型。

（3）如果其中一个运算数的类型为float，则其他运算数将会转换为float并以此作为结果类型。

(4) 如果其中一个运算数为_Bool、char、short int、int位域或者枚举数据类型，则会在int可以表示其值域时将其转换为int；否则，将其转换为unsigned int。如果两个运算数具有相同的类型，这个类型就会作为结果的类型。

(5) 如果两个运算数都是有符号的或者都是无符号的，则较小的整数类型将转换成较大的整数类型，并以较大整数类型作为结果类型。

(6) 如果无符号的运算数在大小上等于或大于有符号的运算数，则有符号的运算数将转换成无符号运算数的类型，并以无符号运算数的类型作为结果类型。

(7) 如果有符号运算数可以表示无符号运算数中的所有值，则在前者可以完全表达后者的值域时将后者转换成前者的类型，并以前者的类型作为结果类型。

(8) 如果到达这一步，则将两个运算数转换成与有符号类型对应的无符号类型。

第4步较为正式的叫法是整型提升。

大多数情况下，运算数的转换都会获得很好的效果，但是要注意下列要点：

(1) 在某些计算机上，char到int的转换可能会涉及到符号扩展，除非将char声明为unsigned。

(2) 有符号整数到长整数的转换会导致符号向左侧扩展；无符号整数到长整数的转换会导致其左侧填充0。

(3) 任何值到_Bool的转换，都会在该值为0时产生0，否则产生1。

(4) 长整型到某个短整型的转换，会导致长整型在左侧进行截取。

(5) 浮点值到整型的转换，将导致浮点值在小数部分进行截取。如果整型的范围不够大，不能包含转换后的浮点值，则结果是不确定的，将负的浮点值转换成一个无符号整型产生的结果也是这样。

(6) 较长浮点值到较短浮点值的转换，可能会导致较长浮点值在发生截取前进行舍入，但也可能不会。

B.6 存储类和作用域

术语存储类指的是编译器为变量分配内存的一种方式，并且指的是特定函数或方法定义的作用域。存储类的类型为auto、static、extern和register。存储类可以在声明中省略，并指派默认的存储类。有关这方面的内容将在后面讨论。

术语作用域指的是程序中特定标识符的含义的范围。定义在所有函数、方法或语句块（此处称作BLOCK）之外的标识符，随后可以在文件中的任何位置引用。定义在BLOCK中的标识符，对于该BLOCK而言是本地的，并且可以在其中重新定义在其外部定义的标识符。标签的名称在整个BLOCK中都是已知的，形参名称也是一样。标签、实例变量、结构和结构成员名，联合和联合名称以及枚举类型名不必相互区分，也没有必要和变量、函数或方法名区分开。然而，枚举标识符却要与定义在同一作用域内的变量名和其他枚举标识符区分。具有全局作用域的类名也会因为其他的全局特性而必须与同一作用域内的其他变量和类型名区分。

B.6.1 函数

如果在定义函数时指定一个存储类，它必须是static或extern。声明为static的函数只能在包

含该函数的同一文件中引用。指定为extern的函数（或未指定类的函数）可以被其他文件中的函数或方法调用。

B.6.2 变量

表B-5总结了各种各样的存储类，它们可用于声明变量、变量的作用域和初始化方法。

表B-5　变量：存储类、作用域和初始化总结

存储类的类型	变量的声明位置	可以引用位置	初始化方式	注　释
static	任何BLOCK外部 BLOCK内部	文件中的任何位置 在BLOCK内	只能使用常量表达式	变量只能在程序执行开始时初始化一次，变量在整个BLOCK中保持值，其默认值为0
extern	任何BLOCK外部 BLOCK内部	文件中的任何位置 在BLOCK内	只能使用常量表达式	在不使用关键字extern的情况下，变量必须至少在一个位置声明，或者使用关键字extern在一个位置声明并指派初始值
auto	BLOCK内部	在BLOCK内	任何有效的表达式	变量要在每次进入BLOCK时进行初始化，无默认值
register	BLOCK内部	在BLOCK内	任何有效的表达式	不保证register的赋值情况，可以声明变量的类型使其具有变化的限制，不能获得register变量的地址，在每次进入BLOCK时进行初始化，无默认值
omitted	任何BLOCK外部	文件中的任何位置，或由包含适当声明的其他文件引用	只能使用常量表达式	这个声明可以只能出现在一个位置，变量在程序执行开始时进行初始化，其默认值为0，默认定义为auto
	BLOCK内部	（参见auto）	（参见auto）	

B.6.3 实例变量

实例变量可以由任何定义给类的实例方法在显式定义变量的interface部分或为类创建的分类中进行访问。继承来的实例变量也可以直接被访问，而无需任何特别声明。类方法则不能访问实例变量。

可以使用特殊指令@private、@protected和@public来控制实例变量的作用域。在这些指令出现之后，变量仍然有效，直到实例变量的声明遇到一个用于终止声明的右花括号或者直到使用上列3个指令中的另一个指令才终止。例如：

```
@interface Point: NSObject
{
@private
    int internalID;
@protected
    float x;
    float y;
```

```
@public
    BOOL valid;
}
```

开始了一个类的接口声明，该类名为Point并包含4个实例变量。变量internalID是private，变量x和y是protected（默认类型），而变量valid是public。

这些指令在表B-6中做了总结。

表B-6 实例变量的作用域

如果变量在以下指令之后声明	则变量可能的引用方式	注 释
@protected	类中的实例方法、子类中的实例方法和类的分类扩展中的实例方法	这是默认的
@private	类中的实例方法和类的分类扩展中的实例方法，但并不是所有子类中的实例方法	该指令限制对类本身的访问
@public	类中的实例方法、子类中的实例方法和类的分类扩展中的实例方法；还可以通过对类的实例应用结构指针间接运算符（->），之后是实例变量的名称（如myFract->numerator），可以从其他函数或方法进行访问	除非必须，否则不要使用这个指令，它会破坏数据封装的概念

B.7 函数

本节总结了函数的语法及运算。

B.7.1 函数定义

一般格式：

```
returnType name (type1 param1 ,type2 param2 , .. )
{
    variableDeclarations

    programStatement
    programStatement
    ...
    return expression ;
}
```

以上定义了一个名为name的函数，它返回一个returnType类型的值，并声明其形参param1、param2……，使param1为type1类型、param2为type2类型，等等。

通常，本地变量在函数的开始部分声明，但这并不是必需的。它们可以在函数的任何位置声明，在这种情况下，只有在声明之后出现的语句才能访问它们。

如果函数不返回值，则将returnType指定为void。

如果圆括号中只指定了void，则函数不会接受参数。如果将...作为参数列表的最后（或者惟一）一个参数，则函数将接受参数的变量成员，如下所示：

```
int printf (char *format, ...)
{
```

```
    ...
}
```

声明一维数组的参数时，无需在数组中指定元素的个数。而声明多维数组时，除第一维以外，必须指定每一维的大小。

有关return语句的讨论，请参见“return语句”一节。

对于函数定义，Objecticve-C语言还支持以前使用的方法。其一般格式为：

```
returnType name (param1, param2, .. )
param_declarations
{
    variableDeclarations
    programStatement
    programStatement
    ...
    return expression;
}
```

这里，圆括号内只列出各个参数名。如果不接收参数，左右圆括号之间就不要出现任何内容。每个参数的类型要在函数外部进行声明，而且要在函数定义的左花括号之前声明。例如：

```
unsigned int rotate (value, n)
unsigned int value;
int n;
{
    ...
}
```

定义了一个名为rotate的函数，这个函数接受名为value和n的两个参数。第一个参数的类型为unsigned int，第二个参数的类型为int。

关键字inline可以作为对编译器的提示而放在函数定义之前。某些编译器使用函数本身的实际代码来代替函数调用，并因此加快了执行速度。下面展示一个例子：

```
inline int min (int a, int b)
{
    return ( a < b ? a : b);
}
```

B.7.2 函数调用

一般格式：

```
name ( arg1, arg2, .. )
```

这个格式调用一个名为name的函数，并将值arg1、arg2……作为参数传递给函数。如果函数不接收参数，只要指定开始和结束圆括号即可（例如initialize ()）。

如果要调用一个函数，而该函数定义在这个调用之后或者定义在另一个文件中，则应该将函数的原型声明包括进来，这种声明的一般格式为：

```
returnType name (type1 param1, type2 param2, .. );
```

这告知编译器该函数的返回类型、所带的参数数目和每个参数的类型。作为一个例子，程序行

```
long double power (double x, int n);
```

将power声明为一个函数，该函数返回一个long double值并接受两个参数——第一个参数类型为double，第二个参数类型为int。圆括号内的参数名实际上是假名，如果需要的话可以将其省略，因此

```
long double power (double, int);
```

作用是完全一样的。

如果编译器预先遇到函数定义或者函数的原型声明，那么在调用函数时，每一个参数的类型将会自动转换以便与函数要求的类型匹配。

如果编译器既没遇到函数的定义也没遇到原型声明，它将假定函数返回一个int类型的值，并将所有float类型的参数转换为double类型，而且如“基本数据类型转换”一节中所述的那样对任何整型参数进行整型提升。其他的函数参数在传递时则无需进行转换。

接受不定数目参数的函数必须这样声明。否则，编译器将基于调用中实际使用的参数数目，任意假定函数接受某个固定的参数数目。

如果函数是使用旧的格式定义的（参见“函数定义”一节），函数的声明将会采用以下格式：

```
returnType name ();
```

如前段所描述，这种函数的参数将被转换。

返回类型声明为void的函数，会使编译器标记任何尝试利用返回值的函数调用。

函数的所有参数是通过值传递的，因此，它们的值不能被函数更改。然而，如果将一个指针传递给某个函数，这个函数就可以更改指针引用的值，但它仍然不能更改指针变量本身的值。

B.7.3 函数指针

其后没有一对圆括号的函数名将会产生指向该函数的指针。地址运算符也可以应用到函数名，以产生指向该函数的指针。

如果fp是指向某个函数的指针，那么既可以用

```
fp ()
```

也可以用

```
(*fp) ()
```

调用这个函数。

如果该函数接受参数，则可以在圆括号中列出这些参数。

B.8 类

本节总结了与类有关的语法和语义。

B.8.1 类定义

类定义包括：在接口部分声明实例变量和方法，在实现部分定义每个方法的代码。

1. 接口部分

一般格式：

```
@interface className : parentClass <protocol, ...>
{
    instanceVariableDeclarations
}
 methodDeclaration
 methodDeclaration
   ...
@end
```

类className使用其父类parentClass来声明。如果className还采用了一个或多个正式协议，在parentClass之后的一对方括号中将列出该协议名称。在这种情况下，对应的实现部分必须包含（多个）协议中列出的所有这类方法的定义。

如果省略冒号和parentClass，将声明新的根类。

（1）实例变量声明

可选的instanceVariableDeclarations部分列出了该类每个实例变量的类型和名称。每个className的实例会获得它自己的一组变量，以及任何从parentClass继承来的变量。通过其名称，或通过定义在className内的实例方法，或通过className的任何子类，可以直接访问所有这些变量。如果使用@private指令对访问进行限制，那么子类将不能访问这样定义的变量（参见“实例变量”一节）。

类方法不能访问实例变量。

（2）属性声明

一般格式：

```
@property (attributes) nameList;
```

这个格式声明了多个属性，每个属性间用逗号隔开。nameList是用逗号隔开的、所声明的类型的属性名列表：

```
(type) propertyName1, propertyName2, propertyName3, ...
```

@property指令可以出现在类、协议或类别的方法声明部分中的任意位置。

可以仅指定assign、copy或retain中的一个属性。如果没有使用垃圾回收，那么应该显式地使用这些属性中的一个；否则编译器会发出警告。如果使用垃圾回收并且没有指定这些属性中的一个，那么将使用默认属性assign。此时，如果类符合NSCopying协议（此时需要复制而不是赋值该属性），编译器只会发出一个警告。

如果使用copy属性，合成赋值方法会使用对象的copy方法。得到的是一个不可变的副本。如果需要可变的副本，必须提供自己的赋值方法。

表B-7 Property属性

属 性	含 义
assign	使用简单的赋值语句为赋值方法中的实例变量设置值
copy	使用copy方法设置实例变量的值
getter=name	为取值方法使用名称name，而不是propertyName，后者是合并取值方法的默认名称
nonatomic	可以直接返回合并取值方法的值。如果没有声明该属性，那么存取器方法是atomic的，意味着对实例变量的存取是互斥锁定的。这样通过确保取值或赋值操作在单个线程中运行，可在多线程环境中提供保护。另外，在默认情况下，在非垃圾回收环境中，合并取值方法在返回属性的值前会保持并自动释放属性
readonly	不能设置属性的值。编译器不会运行赋值方法，也没有合并方法（这是默认属性）
readwrite	可以检索并设置属性的值。编译器需要你提供取值方法和赋值方法，如果使用了@synthesize，还会合成两个方法
retain	属性在赋值时应该保持。只能为Objective-C类型指定该属性
setter=name	为赋值方法使用名称name，而不是setPropertyName，后者是合并存取器方法的默认名称

(3) 方法声明

一般格式：

```
mType (returnType) name1 : (type1) param1 name2 : (type2) param2, ...;
```

这个格式声明了name1:name2: ..方法，该方法返回一个returnType类型的值，并声明其形参param1、param2……，将param1声明为type1类型，param2声明为type2类型，等等。

任何位于name1之后的名称（意思指name2……）都可以省略，这种情况下，仍然要使用冒号作为占位符，而且它将成为方法名的一部分（参见下面的例子）。

如果mType是+，将声明一个类方法，但如果mType是-，则声明一个实例方法。

如果声明的方法是从父类继承的，则父类的定义将被新定义重载。在这种情况下，仍然可以通过向super发送一条消息来访问父类中的方法。

类方法将在相应消息发送给类对象时调用，而实例方法则在将相应消息发送给类实例时调用。类方法和实例方法可以拥有相同的名称。

相同的方法名还可以被不同的类使用。不同类的对象可以响应名称相同的方法的能力称为多态。

如果方法不返回值，returnType就是void。如果函数返回一个id值，则可省略returnType，尽管将id指定为返回类型是一种较好的程序设计实践。

如果将，…用作参数列表的最后（或者惟一）一个参数，则方法将接受数目不定的参数，比如

```
-(void) print: (NSSTRING *) format, ...
{
 ...
}
```

作为类声明的例子，接口定义部分

```
@interface Fraction: NSObject
{
```

```
  int numerator, denominator;
}
+(Fraction *) newFract;
-(void) setTo: (int) n : (int) d;
-(void) setNumerator: (int) n andDenominator: (int) d;
-(int)  numerator;
-(int)  denominator;
@end
```

声明了一个名为Fraction的类，其父类为NSObject。Fraction类有两个名为numerator和denominator的整型实例变量。它有一个名为newFract的类方法，该方法返回一个Fraction对象。它还有两个名为setTo::和setNumerator:andDenominator:的实例方法，每个实例方法都接受两个参数且不返回值。此外，它还有两个名为numerator和numerator的实例方法，这两个方法不接受参数但都返回一个int值。

2. 实现部分

一般格式：

```
@implementation className ;
  methodDefinition
  methodDefinition
   ...
@end
```

这个格式定义的类名为className。通常，实现部分不必重新声明父类和实例方法（尽管可以重新声明），因为已经在前面的接口部分声明过。

除非正在实现分类中的方法（参见B.8.2节），否则实现部分必须定义在接口部分声明的所有方法。如果接口部分列出了一个或多个协议，则必须定义该协议的所有方法——既可隐式地通过继承，也可显式地在实现部分定义。

每个methodDefinition都包含调用方法时将要执行的代码。

(1) 方法定义

一般格式：

```
mType (returnType ) name1 : ( type1 ) param1 : name2 (type2 ) param2, ...
{
    variableDeclarations

    programStatement
    programStatement
    ...
    return expression;
}
```

这个格式定义了$name_1$:$name_2$: ..方法，该方法返回一个returnType类型的值，并声明其形参param1、param2……，其中param1为type1类型，param2为type2类型，等等。如果mType是+，将定义一个类方法；如果mType是-，则定义一个实例方法。这个方法声明必须和接口部分或

前面定义的协议定义中对应的方法一致。

实例方法可以引用类的实例变量以及它直接通过名称继承来的任何变量。如果正在定义类方法，则它不能引用任何实例变量。

标识符self可以用在方法中来引用该方法调用的对象——就是说，消息的接收者。

标识符super也可以用在方法中来引用该方法要调用对象的父类。

如果returnType不是void，则方法的定义中必须出现一条或多条return语句以及returnType类型的表达式。如果returnType是void，则return语句的使用是可选的，如果使用了return语句，方法就不能包含返回值。

作为方法定义的例子，

```
-(void) setNumerator: (int) n andDenominator: (int) d
{
    numerator = n;
    denominator = d;
}
```

定义了一个与声明一致的setNumerator:andDenominator:方法（参见“方法声明”一节）。这个方法将它的两个实例变量设置为提供的参数，并且不执行返回语句（尽管可以执行），因我们已将此方法声明为没有返回值的方法。

声明一维数组的参数时，不必指定数组中元素的个数。但对于多维数组而言，除第一维以外，每一维的大小都要在声明时指定。

本地变量可以在方法内进行声明，而且通常声明在方法定义的开始部分。自动本地变量是在调用方法时分配的，并在退出方法时解除分配。

可参见“return语句”一节，了解有关return语句的说明。

(2) 合并存取器方法

一般格式：

```
@synthesize property_1, property_2, ...
```

这个格式指定应该为所列的属性property_1, property_2, ...合并各个方法。

可以在列表中使用表示法

```
property=instance_var
```

来指定property将与实例变量instance_var相关联。根据使用前面的@property指令为Property声明的属性，合并方法将会特征化。

B.8.2 分类定义

一般格式：

```
@interface className (categoryName) <protocol, ...>
 methodDeclaration
 methodDeclaration
   ...
@end
```

这个格式通过className和列出的相关方法为该类指定了一个分类categoryName。如果列出了一个或多个协议，则分类将采用这个（些）列出的协议。

编译器必须通过前面为该类声明的@interface部分中知道className。

只要你愿意，可以在任意多个的不同源文件中定义任意多个分类。列出的方法将成为类的一部分，并将被子类继承。

分类由className和categoryName这对名称惟一定义的。例如，在指定的程序中可能只有一个NSArray（Private）分类。然而，有个别分类的名称可以被重用。因此，指定的程序可以包含一个NSArray（Private）分类和一个NSString（Private）分类，而且这两个分类将明显地相互区分开。

对于定义在分类中的那些你并不想使用的方法，无需实现它们。

分类可以只展开一个有附加方法的类定义，或重载类中现有的方法。但它不能为该类定义任何实例变量。

如果一个或多个分类将同一个类中的某个方法声明成同一个名称，那么没有定义调用时将执行哪个方法。

作为一个例子，

```
#import 'Complex.h"
@interface Complex (ComplexOps)
-(Complex *) abs;
-(Complex *) exp;
-(Complex *) log;
-(Complex *) sqrt;
@end
```

定义了Complex类的一个名为ComplexOps的分类，它有4个实例方法。大概，程序的某处会出现一个对应的实现部分，以实现这些方法中的一个或多个。

```
#import "ComplexOps.h"
@implementation Complex (ComplexOps)
-(Complex *) abs
{
   ...
}
-(Complex *) exp
{
   ...
}
-(Complex *) log
{
   ...
}
-(Complex *) sqrt
{
   ...
```

```
}
@end
```

如果一个分类定义的方法打算由其他子类实现，那么这个分类称为非正式协议或抽象分类。与正式协议不同，编译器并不对非正式协议的一致性进行任何检查。在运行时，可能以一个专用方法为基础来测试对象与非正式协议的一致性，但也可能不测试。例如，一个方法可能在运行时被请求，而同一个协议中的另一个方法则可能不被请求。

B.8.3 协议定义

一般格式：

```
@protocol protocolName <protocol, ...>
    methodDeclarations
@optional
    methodDeclarations
@required
    methodDeclarations
...
@end
```

利用关联的方法定义了名为protocolName的协议。如果列出其他协议，protocolName还会采用列出的协议。

这个定义名为正式协议定义。

如果类定义或继承了protocolName协议中定义的所有方法，以及其他任何列出协议的所有方法，说明类符合protocolName协议。如果类不符合声明的正式协议，编译器会检查其一致性并生成警告。可能在运行时测试对象与正式协议的一致性，也可能不测试。

@optional指令可以放在方法列表的前面，这些方法的实现是可选的。然后，可以使用@required指令再取得必需方法的列表，为了符合协议，必须实现这些方法。

通常，协议不与任何特定的类关联，除非提供一种方法以定义在各个类中共享的公共接口。

特殊类型修饰符

在协议中，可以使用表B-8中列出的类型修饰符来声明方法参数和返回类型。这些修饰符可用于分布式的对象应用程序。

表B-8 特殊协议类型修饰符

限定词	含 义
in	这个参数引用一个对象，这个对象的值将被发送者更改并发送（也就是，复制）回接收者
out	这个参数引用一个对象，这个对象的值将被发送者更改并发送回发送者
inout	这个参数引用一个对象，该对象的值将被发送者和接收者设置，并将被来回发送，这是默认的
oneway	它用于返回类型声明；一般情况下（one way void）它用于指定这个方法的调用者不必等待返回值——换言之，此方法可以异步执行
bycopy	这个参数或返回值将被复制
byref	这个参数或返回值将通过引用传递而且不被复制

B.8.4 对象声明

一般格式：

```
className *var1, *var2, ...;
```

这个格式将var1、var2……定义为className类的对象。注意，该格式声明了一些指针变量，而且不会为每个指针对象中包含的实际数据保留存储空间。声明

```
Fraction *myFract;
```

将myFract定义为Fraction对象，或从技术上讲，是将myFract定义为指向Fraction对象的指针。要想给Fraction的数据结构分配实际的存储空间，通常会调用该类的alloc或者new方法，如下：

```
myFract = [Fraction alloc];
```

这条语句为Fraction对象和指向它的指针保留了足够的存储空间，这个指针将被返回并指派给myFract。变量myFract通常称作一个对象或Fraction类的实例。由于定义了位于根对象中的alloc方法，所以最新分配的对象的所有实例变量将被设为0。然而，但这并不意味着已经对该对象进行了正确的初始化，并且应该在使用这个对象之前调用它的初始化方法（例如init）。

因为变量myFract已经被显式地声明为Fraction类的一个对象，所以可以说该变量是静态类型的。通过参照关于方法的正确使用以及它们的参数和返回类型的类定义，编译器可以检查所使用的静态类型的变量的一致性。

id对象声明

一般声明：

```
id <protocol,...> var1, var2, ...;
```

这个格式将var1、var2……声明为一个不确定类的对象，这个类与尖括号中列出的（多个）协议是一致的。协议列表是可选的。

任何类对象都可以指派给id变量，反之亦然。如果列出一个或多个协议，编译器将使用一种一致的方式——即，与正式协议中声明的方法的相关参数和返回类型相一致，来检查那些列出的协议所使用的方法以及所使用的任何声明的变量。

例如，在语句

```
id <MathOps> number;
    ...
result = [number add: number2];
```

中，编译器将检查MathOps协议是否定义了一个add:方法。如果定义了这个方法，随后编译器就会检查这个方法的参数和返回类型的一致性。因此，如果此方法接受一个整型参数，而你正在给它传递一个上述的Fraction对象，则编译器将发出警告。

系统会跟踪每个对象所属的类；因此，系统可在运行时确定对象的类，并随后选择要调用的正确方法。这两个过程通常分别称为动态类型和动态绑定。

B.8.5 消息表达式

格式1：

```
[receiver  name1: arg1 name2: arg2, name3: arg3 .. ]
```

这个格式调用由receiver指定的类中的方法$name_1$:$name_2$:$name_3$……，并传递arg1、arg2……作为方法的参数。这种形式称为消息表达式。表达式的值是该方法返回的值，或如果方法是这样声明的，并且没有返回值，则表达式的值是void。表达式的类型就是被调用的方法所声明的类型。

格式2：

```
[receiver name];
```

如果方法不接收参数，那么这种格式用于从接收者指定的类调用方法名。

如果receiver是id类型，编译器将在声明的类中查找特定方法的定义或者继承来的定义。如果没有找到这样的定义，编译器就会发出一条警告，即接收者可能不会响应特定的消息。进一步假设：这个方法返回一个id类型的值，同时将所有浮点型参数转换成双精度类型并对所有整型参数进行整型提升，如同前面在“基本数据类型的转换”一节中所概述的那样。其他的方法参数在被传递时则不会进行类型转换。

如果receiver是一个class对象（可以仅仅通过指定类名来创建这个对象），则会调用特定的class方法。否则，receiver将是某个类的一个实例，并会调用对应的实例方法。

如果receiver是一个静态类型的变量或表达式，编译器就会在类定义中查找该方法（或者查找任何继承来的方法），并转换所有（可能位置处的）参数使之与方法期望的参数匹配。因此，给一个期望接受浮点值的方法传递整数会使这个整数在此方法被调用时自动转换为浮点值。

如果receiver是一个空对象指针（即nil），便可以向它发送消息。如果与该消息相关的方法返回一个对象，则消息表达式的值将为nil。如果方法不返回对象，表达式的值将是不确定的。

如果在多个类中定义同一个方法（既可显式地定义也可通过继承定义），编译器就会在各个类中检查参数和返回类型的一致性。

给方法传递的所有参数都是通过值传递的；因此，方法不能更改它们的值。如果给方法传递一个指针，方法就可以更改指针引用的值，但它仍然不能更改指针本身的值。

格式3：

```
receiver.property
```

这个格式会调用receiver的取值方法（默认为property），除非将该表达式用作左值（参见格式4）。可以使用@property指令更改取值方法名，此时将使用被调用的方法。

如果使用默认的取值方法名，那么前面的表达式等效于以下表达式：

```
[receiver  property]
```

格式4：

```
receiver.property = expression
```

这个格式会调用与属性property相关联的赋值方法，将expression的值作为参数传递给它。默认情况下，会调用赋值方法setProperty:，除非使用了以前的@property指令为属性赋予了其他的赋值方法名。

如果使用默认的赋值属性名，那么前面的表达式等效于以下表达式：

```
[receiver setProperty: expression]
```

B.9 语句

程序语句可以是任何后面跟有一个分号的有效表达式（通常是一条赋值语句或者一个函数调用），或是以下描述的特殊语句。可以将标签放在任何语句之前，标签包括后面紧跟一个分号标识符（参见goto语句）。

B.9.1 复合语句

包含在一对花括号内的程序语句统称为复合语句或语句块，它们可以出现在程序中允许单条语句出现的任何地方。程序块可以有自己的一组变量声明，它可以重载外部程序块中定义的任何类似命名的变量。这些本地变量的作用域即是定义它们的程序块。

B.9.2 break语句

一般格式：

```
break;
```

在for、while、do或switch语句中，break语句的执行导致这些语句立即终止。然后继续执行紧跟在循环或者switch之后的语句。

B.9.3 continue语句

一般格式：

```
continue;
```

在一个循环中执行continue语句会导致跳过continue语句之后的语句。否则，循环将以正常方式继续执行。

B.9.4 do语句

一般格式：

```
do
   programStatement
while ( expression );
```

只要expression的计算结果非零，就会执行programStatement。注意一点：因为expression每次都在执行programStatement之后求值，所以保证了programStatement将至少被执行一次。

B.9.5 for语句

格式1：

```
for ( expression_1; expression_2; expression_3 )
    programStatement
```

一旦开始执行循环，就会计算expression_1。然后，求expression_2的值。如果它的值非零，

将执行programStatement并对expression_3求职。只要expression_2的值非零，就继续执行programStatement并计算随后的expression_3。因为expression_2每次都在执行programStatement之前计算，所以，如果expression_2的值在首次进入循环时为0，那么可能永远不会执行programStatement。

for循环本地的变量可以在expression_1中声明。这些变量的作用域为for循环的作用域。例如

```
for ( int i = 0; i < 100; ++i)
    ...
```

在循环开始时声明整型变量i而且将其初始值设为0。这个变量可以被循环内的任何语句访问，但当循环终止之后，它将不能再被访问。

格式2：

```
for ( var in expression )
    programStatement
```

这种for循环的变体设置了快速枚举。Var是一个变量，可以声明这个变量的类型，让它的作用域与for循环相同。expression是一个表达式，其结果符合NSFastEnumeration协议。通常expression是一个集合，如数组或字典。

每执行一次for循环，根据expression初始求值生成的下一个对象都会被分配给var，并且循环体（由programStatement表示）都会被执行。Expression中的所有对象都枚举后，执行终止。

注意，for循环不能更改集合的内容。如果更改了，则会抛出异常。

数组中的每个元素都会按顺序进行枚举。枚举字典对象会导致每个键都会被枚举，没有特定的顺序。枚举一个组会导致该组的每个成员都会被枚举，没有特定的顺序。

B.9.6 goto语句

一般格式：

```
goto identifier;
```

goto语句的执行使控制权直接传递给标记有identifier的语句。被标记的语句必须与goto位于同一个函数或方法内。

B.9.7 if语句

格式1：

```
if ( expression )
    programStatement
```

如果expression的计算结果非零，将执行programStatement；否则，将略过这条语句。

格式2：

```
if ( expression )
    programStatement_1
else
    programStatement_2
```

如果expression的值非零，将执行programStatement_1；否则，将执行programStatement_2。如果programStatement_2是另一条if语句，它更像是一个if-else if链，如下：

```
if ( expression_1 )
    programStatement_1
else if ( expression_2 )
    programStatement_2
    ...
else
    programStatement_n
```

else子句总是与最后一个不包含else的if语句关联。如果有必要，可以使用花括号改变这种关联关系。

B.9.8 空语句

一般格式：

```
;
```

空语句的执行不会对程序产生任何影响，它主要满足for、do或者while循环中程序语句的需求。下面的语句将from指向的字符串复制到to指向的字符串：

```
while ( *to++ = *from++ )
   ;
```

这条语句中，空语句用于满足while的循环表达式之后要出现一条程序语句这一需求。

B.9.9 return语句

格式1：

```
return;
```

执行return语句会使程序的执行返回到调用的函数和方法。这种格式只能用于从不返回值的函数或方法中返回。

如果程序的运行继续到函数或方法的结尾，并且没有遇到return语句，那么这个函数或方法就像已经执行这种格式的return语句那样返回。因此，这种情况下，不返回任何值。

格式2：

```
return expression;
```

这个格式将expression的值返回给调用函数或方法。如果expression的类型与函数或者方法声明中声明的返回类型不一致，它的值就会在返回之前自动转换为声明的类型。

B.9.10 switch语句

一般格式：

```
switch ( expression )
{
```

```
    case constant_1:
       programStatement
       programStatement
        ...
       break;
    case constant_2:
       programStatement
       programStatement
        ...
      break;
    ...
    case constant_n:
       programStatement
       programStatement
       ...
       break;
    default:
       programStatement
       programStatement
        ...
       break;
}
```

这个格式将求expression的值，并与常量表达式constant_1、constant_2……constant_n的值进行比较。如果expression的值与这些case值中的一个匹配，将执行紧跟在其后的程序语句。如果没有case值与expression的值匹配，将执行default情况，如果有default，就会处理。如果不包含default情况，则不会执行switch中包含的任何语句。

expression的计算结果必须是整型，并且两种case的值不能相同。省略某个特定case中的break语句，将导致程序继续执行下一个case。

B.9.11 while语句

一般格式：

```
while ( expression )
    programStatement
```

只要expression的值非零，就会执行programStatement。因为expression每次都在执行programStatement之前计算，所以可能永远不会执行programStatement。

B.10 异常处理

通过在@try块中包括可能会生成异常的语句，就可在运行时处理异常，其一般格式为：

```
@try
   programStatement 1
@catch (exception)
   programStatement 2
```

```
@catch (exception)
  ...
@finally
   programStatement n
```

如果programStatement 1抛出异常，那么会按顺序测试后面的@catch块，查看相应的exception是否与抛出的异常相符。如果相符，则会执行相应的programStatement。无论是否抛出并捕获了异常，都将执行@finally块（如果提供了）。

B.11 预处理程序

预处理程序用于在编译器真正地看到代码之前分析源文件。以下就是预处理程序执行的各种功能：

(1) 用等价形式代替连字序列（参见“复合语句”一节）。

(2) 将所有以斜线字符（\）结尾的行连接到同一行。

(3) 将程序划分到某个标记流内。

(4) 删除注释，并将注释替换成单个空格。

(5) 处理预处理程序指令（参见“预处理程序指令”一节）并扩展宏指令。

B.11.1 三连字序列

要处理非ASCII字符集，需要识别以下三字符序列（称为三连字字符，即trigraph），并且当这些字符出现在程序（以及字符串）的任何位置时都能对它们进行特别处理：

连字字符	含义	连字字符	含义
??=	#	??/	\
??(	[	??'	^
??)	]	??!	|
??<	{	??-	~
??>	}		

B.11.2 预处理程序指令

所有预处理程序指令都以字符#开头，它必须是命令所在行的第一个非空白字符。字符#之后可以有选择地跟随一个或多个空格或者制表符。

1. #define指令

格式1：

```
#define name text
```

这个格式向预处理程序定义标识符名称，并将该名称与在name到其所在行结束间的第一个空格之后出现的任何text关联起来。随后，在程序中使用name时，将会直接代替程序相应位置的text。

格式2：

```
#define name(param_1, param_2, ..., param_n) text
```

这个格式定义一个宏指令name，它接受由param_1、param_2……param_n指定的参数，每个参数是一个标识符。随后，在程序中使用带有参数列表的name时，将直接替换程序相应位置的text，并用宏指令调用的参数替换text中出现的所有对应参数。

如果该宏接受可变数目的参数，可以在参数列表结尾使用3个圆点。列表中其余参数将通过特殊标识符__VA_ARGS__在宏指令定义中集体引用。作为一个例子，下列语句定义一个名为myPrintf的宏，它接受一个后面跟有可变数目参数的前导格式字符串：

```
#define myPrintf(...) printf ("DEBUG:   __VA_ARGS__);
```

合法的宏用途包括

```
myPrintf ("Hello world!\n");
```

以及

```
myPrintf ("i = %i, j = %i\n", i, j);
```

如果定义需要多行，必须以斜杠字符结束以延续每一行。定义一个名称之后，该名称就可以用于文件中的任何位置。

运算符#允许用在以下#define指令中，这些指令接受参数并且之后是该宏的参数名称。调用宏时，传递给宏指令的实际值时，预处理程序将实际值放在双引号中。换言之，预处理程序将把这个值转换为一个字符串。例如，使用调用

```
#define printint(x) printf (# x " = %d\n", x)
```

的定义

```
printint (count);
```

将被预处理程序扩展成

```
printf ("count" " = %i\n", count);
```

或者，等价于

```
printf ("count = %i\n", count);
```

执行字符串化操作时，预处理程序将在任何“或\字符之前放置\字符。因此，使用声明

```
#define str(x) # x
```

可以将调用

```
str (The string "\t" contains a tab)
```

扩展成以下形式：

```
"The string \"\\t\" contains a tab"
```

在接收参数的#define指令中还允许使用##运算符。它放在某个宏指令的参数名称之前（或者在它后面跟有一个宏指令的名称）。预处理程序使用在调用宏时所传递的值，并根据参数为宏指令创建一个标记，以及后面（或前面）的标记。例如，使用调用

```
printx (5)
```

的宏指令定义

```
#define printx(n) printf ("%i\n", x ## n );
```

会产生以下形式：

```
printf ("%i\n", x5);
```

使用调用

```
printx(10)
```

的宏指令定义

```
#define printx(n) printf ("x" # n " = %i\n', x ## n );
```

当替换并连接字符串之后，会产生

```
printf ("x10 = %i\n", x10);
```

#和##字符两侧不允许出现空格。

2. #error指令

一般格式：

```
#error text
  ...
```

预处理程序将特定的text编写成一条出错消息。

3. #if指令

格式1：

```
#if constant_expression
  ...
#endif
```

这个格式首先计算constant_expression的值。如果结果非零，将处理#endif指令之前的所有程序行；否则，它们将被自动略过并且不被预处理程序或者编译器处理。

格式2：

```
#if constant_expression_1
  ...
#elif constant_expression_2
  ...
#elif constant_expression_n
  ...
#else
  ...
#endif
```

如果constant_expression_1非零，将处理#elif之前的所有程序行，在其之后直到#endif间的其余程序行将被略过。否则，如果constant_expression_2非零，将处理下一个#elif之前的所有程序行，并略过它之后直到#endif之前的其余程序行。如果任何语句表达式的计算结果都非零，

则处理#else之后的程序行（如果包括的话）。

特殊运算符defined可以用作常量表达式的一部分，因此对于

```
#if defined (DEBUG)
  ...
#endif
```

如果前面定义过标识符DEBUG（同时参见下一节定义的#ifdef），则处理#if和#endif之间的代码。标识符两侧不必使用圆括号，因此

```
#if defined DEBUG
```

将运行良好。

4. #ifdef指令

一般格式：

```
#ifdef identifier
  ...
#endif
```

如果identifier的值在前面定义过（或者编译程序时通过#define定义或使用-D命令行选项定义的），将处理#endif之前的所有程序行；否则，这些程序行将被略过。就像使用#if指令一样，#elif和#else指令也可以和#ifdef指令一起使用。

5. #ifndef指令

一般格式：

```
#ifndef identifier
  ...
#endif
```

如果前面没有定义identifier的值，将处理#endif之前的所有程序行；否则，这些程序行将被略过。就像使用#if指令一样，#elif和#else指令也可以和#ifndef指令一起使用。

6. #import指令[4]

格式1：

```
#import "fileName"
```

如果fileName指定的文件已经在前面包含到程序中，则这条语句将被略过。否则，预处理程序将搜索实现定义的目录（或多个目录）以查找fileName。通常，首先检索包含源文件的同一目录，如果没有发现该文件，将检索一系列实现定义的标准位置。找到这个文件之后，将在#import指令出现的确切位置将该文件的内容包含到程序中。然后分析包括到程序的文件中所含的预处理指令；因此，包括到程序中的文件本身也可以包含其他的#import或者#include指令。

格式2：

```
#import <fileName>
```

如果前面没有包含这个文件，预处理程序只在标准位置检索指定的文件。特别地，搜索忽略了当前源文件。这一操作将在找到该文件之后执行，否则，与前面所描述的完全一样。

其他格式中，可以提供前面定义的名称并且还会出现扩展名。因此，下列序列也会起作用：

```
#define ROOTOBJECT  <NSObject.h>
  ...
#import ROOTOBJECT
```

7. #include指令

除了不对前面包含的指定头文件进行检查之外，这个指令与#import具有相同的功能。

8. #line指令

一般格式：

```
#line constant "fileName"
```

这个指令会使编译器处理程序中随后的程序行，就像源文件的名称为fileName且随后程序行的行编号以constant开始一样。如果没有指定fileName，文件名将由最后的#line指令指定，或使用源文件的文件名（如果前面没有指定文件名）。

#line指令主要用于在显示编译器发出的出错消息时控制文件名和行编号。

9. #pragma指令

一般格式：

```
#pragma text
```

这个指令会使预处理程序执行一些实现定义的操作。例如，以下附注

```
#pragma loop_opt(on)
```

会在特定的编译器上执行特殊的循环优化。如果编译器遇到这个附注，该编译器识别不了附注loop_opt，就会忽略它。

10. #undef指令

一般格式：

```
#undef identifier
```

对于预处理程序而言，指定的identifier将成为未定义的。随后的#ifdef或者#ifndef指令在使用中会认为这个标识符是从未定义过的。

11. #指令

这个一条空指令，将被预处理程序忽略。

B.11.3 预定义标识符

以下标识符是由预处理程序定义的：

标识符	含　义
__LINE__	编译当前的行编号
__FILE__	编译当前源文件的名称
__DATE__	编译文件的日期，以"Mmm dd yyyy"格式表示
__TIME__	编译文件的时间，以"hh:mm:ss"格式表示
__STDC__	如果编译器符合ANSI标准，则定义为1；否则定义为0
__STDC_HOSTED__	如果实现被托管，则定义为1；否则定义为0
__STDC_VERSION__	定义为199901L

附录C　地址簿源代码

为了便于参考，下面给出本书第二部分中一直使用的地址簿例子的完整接口和实现文件。该附录包含AddressCard和AddressBook类的定义。你应该在你的系统上实现这些文件，然后扩展这些类定义，以使它们更实用、功能更强。这是你学习这门语言并熟悉生成程序、使用类和对象以及使用Foundation框架的一种极好方式。

C.1　AddressCard接口文件

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

@interface AddressCard : NSObject <NSCopying, NSCoding> {
    NSString *name;
    NSString *email;
}

@property (nonatomic, copy) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;

-(void) print;

@end
```

C.2　AddressBook接口文件

```
#import <Foundation/Foundation.h>
#import "AddressCard.h"

@interface AddressBook: NSObject <NSCopying, NSCoding>
{
    NSString          *bookName;
    NSMutableArray    *book;
}

@property (nonatomic, copy) NSString *bookName;
@property (nonatomic, copy) NSMutableArray *book;

-(id) initWithName: (NSString *) name;
-(void) sort;
```

```
-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;
-(AddressCard *) lookup: (NSString *) theName;

-(void) dealloc;

@end
```

C.3 AddressCard实现文件

```
#import "AddressCard.h"

@implementation AddressCard

@synthesize name, email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
    [self setName: theName];
    [self setEmail: theEmail];
}


// Compare the two names from the specified address cards
-(NSComparisonResult) compareNames: (id) element
{
    return [name compare: [element name]];
}

-(void) print
    NSLog (@"=====================================");
    NSLog (@"|                                   |");
    NSLog (@"|  %-31s |", [name  UTF8String]);
    NSLog (@"|  %-31s |", [email UTF8String]);
    NSLog (@"|                                   |");
    NSLog (@"|                                   |");
    NSLog (@"|                                   |");
    NSLog (@"|        O               O          |");
    NSLog (@"=====================================");
}

-(AddressCard *) copyWithZone: (NSZone *) zone
{
    AddressCard *newCard = [[AddressCard allocWithZone: zone] init];
```

```
    [newCard retainName: name andEmail: email];
    return newCard;
}

-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail
{
    name = [theName retain];
    email = [theEmail retain];
}

-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject: name forKey: @"AddressCardName"];
    [encoder encodeObject: email forKey: @"AddressCardEmail"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    name = [[decoder decodeObjectForKey: @"AddressCardName"] retain];
    email = [[decoder decodeObjectForKey: @"AddressCardEmail"] retain];

    return self;
}

-(void) dealloc
{
    [name release];
    [email release];
    [super dealloc];
}
@end
```

C.4 AddressBook实现文件

```
#import "AddressBook.h"

@implementation AddressBook

@synthesize book, bookName;

// set up the AddressBook's name and an empty book

-(id) initWithName: (NSString *) name{
    self = [super init];

    if (self) {
        bookName = [[NSString alloc] initWithString: name];
```

```
        book = [[NSMutableArray alloc] init];
    }

    return self;
}

-(void) sort
{
    [book sortUsingSelector: @selector(compareNames: )];
}

-(void) addCard: (AddressCard *) theCard
{
    [book addObject: theCard];
}

-(void) removeCard: (AddressCard *) theCard
{
    [book removeObjectIdenticalTo: theCard];
}

-(int) entries
{
    return [book count];
}

-(void) list
{
    NSLog (@"======== Contents of: %@ =========", bookName);

    for ( AddressCard *theCard in book )
        NSLog (@"%-20s %-32s", [theCard.name UTF8String],
               [theCard.email UTF8String]);

    NSLog (@"=============================================");
}

// lookup address card by name - assumes an exact match

-(AddressCard *) lookup: (NSString *) theName
{
    for ( AddressCard *nextCard in book )
        if ( [[nextCard name] caseInsensitiveCompare: theName] == NSOrderedSame )
            return nextCard;

    return nil;
}
```

```
-(void) dealloc
{
    [bookName release];
    [book release];
    [super dealloc];
}

-(void) encodeWithCoder: (NSCoder *) encoder
{
    [encoder encodeObject:bookName forKey: @"AddressBookBookName"];
    [encoder encodeObject:book forKey: @"AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
    bookName = [[decoder decodeObjectForKey: @ "AddressBookBookName"] retain];
    book = [[decoder decodeObjectForKey: @ "AddressBookBook"] retain];

    return self;
}

// Method for NSCopying protocol

-(id) copyWithZone: (NSZone *) zone
{
    AddressBook *newBook = [[self class] allocWithZone: zone];

    [newBook initWithName: bookName];
    [newBook setBook: book];

    return newBook;
}
@end
```

附录D 资　　源

本附录包含一个资源的选择列表，你可以从中获得更多信息。一些信息可能位于你的系统上、在线Web站点上或从某本书中获得。我们编辑了C语言、Objective-C语言、Cocoa和iPhone/iTouch程序设计的资源。在这里，该列表将给你提供一个良好的开端，帮助你定位你要查找的所有内容。

D.1 练习答案、勘误表等

你可以访问Web站点www.informit.com/register来获得本书的练习答案和勘误表。

D.2 Objective-C语言

下面列出的资源可让你获得更多有关Objective-C语言的信息。

D.2.1 书籍

- 《Objective-C 2.0 Programming Language》，Apple Computer，2008——这本书是现有的、有关Objective-C语言的最佳参考书，它很适合在读完本书后进行阅读。通过Xcode的Help->Documentation窗口查看该书的内容，或者在Apple的Web站点上查看该内容。下面是该书pdf版本的在线链接：http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf。
- 《Object-Oriented Programming:An Evolutionary Approach, Second Edition》，Brad Cox和Andy Novobilski、Addison-Wesley，1991——这是最初关于Objective-C的书，该语言的设计者Brad Cox是本书的合著者。
- 《Objective-C Pocket Reference》，Andrew M. Duncan。O'Reilly Associates公司，2003——这是一本Objective-C语言的简明参考书。

D.2.2 Web站点

- http://developer.apple.com/documentation/Cocoa/ObjectiveCLanguage-date.html——Apple Web站点的Objective-C语言专区。除了其他内容，还包含：在线文档、示例代码和技术札记。

D.3 C程序设计语言

因为C是最基础的程序设计语言，所以你可能想要更深入地研究它。该语言已经存在了25年，因此关于该主题的信息绝对不会缺乏。

书籍

- 《Programming in C, Third Edition》，Stephen Kochan。Sams出版社，2004——这是我写的第一本书（这是很久以前的事了），此后修订了好几次。这是一本指南，它包括很多C语言特性的大量细节，这些细节在第13章“基本的C语言特性”中集中到了一起。
- 《The C Programming Language，Second Edition》，Brian W. Kernighan和Dennis M. Ritchie。Prentice Hall公司。1988——它一直是该语言的圣经和参考。它是为C语言编写的第一本参考书，并由创造了这门语言的Dennis Ritchie合著。
- 《C:A Reference Manual, Fifth Edition》，Samuel P. Harbison III和Guy L. Steele, Jr。Prentice Hall，2002——C程序员的另一本优秀参考书。

D.4 Cocoa

如果你想认真学习Mac OS X下的应用程序开发，将需要学习如何使用Cocoa进行编程。有很多图书和Cocoa有关，并且新的图书随时都在出版。你可以在amazon.com的搜索窗口键入“Cocoa”并查看弹出的内容。以下是一些可以找到的图书。

D.4.1 书籍

- Introduction to Cocoa Fundamentals Guide。Apple Computer公司，2007——这本出色的书籍介绍了如何使用Cocoa开发应用程序。可从Xcode的Documentation窗口访问它。也可在线访问并获得该书的pdf版本：http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf。
- Cocoa Programming for Mac OS X, Third Edition。Aaron Hillegass。Addison-Wesley，2008——这是一本关于Cocoa的极好简介，是用易于阅读的方式编写的。
- Cocoa in a Nutshell。Michael Beam和James Duncan Davidson。O'Reilly &Associates公司，2003——本书是关于属于Cocoa开发系统一部分的许多不同类和方法的良好参考资源。
- Learning Cocoa with Objective-C, Second Edition。James Duncan Davidson和Apple计算机公司。O'Reilly & Associates公司，2002——这是一本关于Cocoa程序设计的介绍性图书。

D.4.2 Web站点

- http://developer.apple.com/cocoa/——Apple针对Cocoa开发人员的主Web站点，包括文档、示例代码、技术注解和很多信息。
- http://www.cocoadevcentral.com/——这是用于帮助人们学习如何利用Cocoa和Objective-C编程的Web站点。
- http://www.cocoadev.com/——这是一个开放的Web站点，它可以被任何人编辑。在那里可以找到很多好的信息。

D.5 iPhone和iTouch应用程序开发

iPhone的流行肯定导致现在有大量的文章在介绍如何为该设备开发应用程序。下面是在本书出版或要印刷时出现的一些内容。

D.5.1 书籍

- iPhone OS Programming Guide。Apple Computer公司，2008——这本出色的书籍介绍了iPhone的应用程序开发。可从Xcode的Documentation窗口访问它。也可在线访问并获得该书的pdf版本：http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf。
- The iPhone Developer's Cookbook: Building Applications with the iPhone SDK，Erica Sadun，Addison-Wesley Professional公司，2008——介绍了如何编写不同类型的iPhone应用程序。
- Beginning iPhone Development: Exploring the iPhone SDK，Dave Mark，Apress公司，2008——简介了如何为iPhone和iTouch编写应用程序。
- iPhone Application Development: Building Applications for the AppStore，Jonathan Zdziarski，2009，O'Reilly Media，2002——本书印刷时尚未出版。

D.5.2 Web站点

- http://developer.apple.com/iphone/——Apple针对iPhone开发人员的主Web站点（也称为iPhone DevCenter）。从中可以找到各种文档、指南视频、示例代码、技术注解和很多信息，还可在此下载iPhone SDK。
- http://www.iphonedevcentral.org/——该网站提供了免费的指南和论坛，可供你交流想法并提问。