Become an Xcoder

Xcode 开发员入门导引

作者/ Bert Altenburg, Alex Clarke, Philippe Mougin
翻译/ 刘珏

英文版版权声明

目录

前言

苹果电脑公司（Apple Computer, Inc.）为用户提供了全套免费的Cocoa程序开发工具，这套工具就是我们所说的Xcode。它随着Mac OS X一起发行，当然，你也可以在苹果公司的网站下载。

市面上已经有了许多关于为Mac微机编程的优秀读物，这些读物阅读的前提往往是要求你有一定的编程经验。但本书并不要求这个前提。因为本书以介绍Objective-C语言的基础知识为内容，即关于如何使用Xcode的基础知识。通过学习前五章，你将掌握如何在非图形界面下编写基本的代码。之后的一些章节将教给你如何在图形界面（GUI）下开发简单的程序。当你读过了整本书，你就已经可以再去阅读开头提到的那些相对高深的读物以提高自己。事实上，你非常有必要去读更多的书，因为编程有很多东西要学习。不过现在请不要紧张，本书的内容十分简单。

如何使用本书

如你所见，有些词组会被加上灰色底纹：

Some tidbits

我们建议读者每个章节至少阅读两次。第一次，跳过这些加上了灰色底纹的内容；第二次阅读的时候再来看这些部分。这样你将会很好的复习所学，同时第一次阅读时那些令你迷惑的部分会变成一些有意思的小技巧为你所学到。这样使用本书，能令你的学习少走弯路。

本书包含许多例子，这些例子由一行或多行语句行组成。为了确保说明和例子正确对应，每一个例子都用带有方括号的数字标注。对于多于两行语句行的例子，方括号中的第一个数字代表例子编号，第二个数字代表这个例子中语句行的行号。比如[4.3]表示例4中的第三行语句行。对于一个较长的程序片断，通常把这种编号置于一行行末，就像这样：

```
volume = baseArea * height; // [4.3]
```

编程不是一件简单的事。对你来说，需要坚持到底并亲自实践本书中提供的所有素材。学习驾驶汽车或者演奏钢琴靠纸上谈兵是不行的，学习编程也莫不如此。本书以电子版呈现，你就更没有理由不频繁的切换到Xcode中进行演练。因此，建议读者把前五章读上三遍。第二遍时要边读边把例子拿到电脑上实践，到了第三遍就要尝试对给定的例子进行些许修改，以进一步探求编程原理。

第 0 章 在开始之前

我们为读者写了这本免费的读物，作为回报，在进入正题前还要谈谈关于促进Mac微机发展的话题。每一位Mac微机的使用者都可以为促进自己钟爱的电脑平台的发展尽进绵薄。这里将告诉你如何去做。

1、Mac微机功能越强大，越容易令别人关注它。所以要及时浏览关于Mac微机的原创网站，阅读Mac微机的杂志。当然还要学好Objective-C和AppleScript。在工作上，AppleScript能为你节约大量的时间和金钱。到网上找找我的书《AppleScript初学者》（*AppleScript for Absolute Starters*），它同样是免费的，网址如下：

`http://www.macscripter.net/books`

2、通过视觉展示告诉世界并不是人人都用PC。在公共场合着一件以Mac微机为印花的T恤是个办法，但还有许多其他途径。如果运行“活动监视器（Activity Monitor）”（位于“应用程序”文件夹下的“实用程序”文件夹里面），你会注意到你的Mac微机只是偶尔才会满负荷运行。

科研人员正在推动几项“分散计算计划”（distributed computing projects，简称DC），比如Folding@home和SETI@home，就是利用Mac微机空闲的处理能力来为公众服务。你只需要下载一个被称做DC客户端（DC client）的免费小程序并开始处理工作。这些 DC客户端（DC client）占用很少的系统资源。如果你运行一个占用资源很大的程序， DC客户端（DC client）将自动中止，因此你大可不必在意它的运行。这项工作如何帮助Mac微机？通常这种DC项目的网站上回对各个团队的工作进度进行排名。如果你加入了一个Mac微机团队（你可以从他们的名字中区别来），你就可以帮助你的团队提升排名。其他平台的用户会看到Mac微机是如此之棒！DC项目的内容很丰富，有关于数学的，也有关于医疗的等等。你可以通过一下网址找到一个你感兴趣的DC项目：

`http://distributedcomputing.info/projects.html`

对于这个提议，唯一的问题是它可能会令你上瘾！

3、确保Mac微机拥有最好的软件。不仅仅是你自己写一些很酷的软件。而且要把向其他开发人员（礼貌的）反馈使用意见变成一种习惯。你对试用的某一款软件并不感冒，那么告诉它的开发者为什么你不喜欢这个软件。发现了bug也要及时报告，最好在报告中精确的描述一下你当时的操作过程。

4、为你使用的软件付费。只要Mac微机的软件市场能够生存下去，开发人员就会坚持不懈的提供优秀的软件。

5、请向至少3位对编写程序感兴趣的Mac微机用户推荐本书，并告诉他们哪里能够找到本书。或者建议他们履行以上4点建议。

好了，在后台下载DC客户端（DC client）的同时我们开始学习编程吧。

第 1 章 程序是一系列指令

当学习驾驶汽车的时候，你要学会同时处理不同的事情。你必须弄清楚离合器、油门和刹车。编程也需要你同一时间“一心多用”，否则，你就可能在编程的路上撞车。我们学习开车前往往已经了解了车子的内部结构，但在学习Xcode编程上你并不具备这个优势。为了不在一开始就令你觉得困窘，我们把编程的事情放到后面的章节，现在先来通过一些基本数学知识帮你熟悉Objective-C语言的代码。

小学的时候我们作过这样的填空题：

2＋6＝（ ）

（ ）＝3*4（星号“*”是计算机中乘号的标准写法）

到了中学，填空过时了，我们改用*x*、*y*这样的**变量**（**variables**）（我们称之为“代数”）来代替。回头看看，也许你很想知道为什么这样微小的记号的变化会令那么多人感到恐慌。

2＋6＝x

y=3*4

Objective-C语言同样使用变量。变量并不神秘，它是用来代替特定数据的名称，比如代替一个数。这里有一个Objective-C语言的**语句行**（**statement**）的例子，也就是一行代码，它的含义是给一个变量赋值。

[1]

```
x = 4;
```

上面例子中，变量*x*被赋了一个数值4。同时你也注意到了，这个语句的末尾带有一个分号。这是因为每个语句行都要以分号结尾。为什么要这样？尽管例[1]这个程序片断在你看来很小儿科，但电脑却完全不知道这段程序的含义。为此要借助一个被称为**编译器**（**compiler**）的特殊程序把你输入的文字转换成为Mac微机能够读懂的0和1。阅读并且理解人类键入的文本对编译器来说并不容易，所以要给出一些明显的标记来，比如标出语句行的结尾。这就是你要用上一个分号。

如果你忘了在代码中加上分号，哪怕只忘了一个分号，那么代码也不能被编译，也就是说不能生成被Mac微机执行的程序。但不要担心，编译器会提示错误。后面读到的章节会帮助你查找是哪里出了问题。

对于编译器来说变量本身并没有特别的意义，但是描述性变量可以令程序简单易读并易于理解。特别是你在代码中寻找错误的时候它就显得特别有用。

> 错误在程序中习惯被称为**臭虫(bugs)**。发现并修正这些错误被叫做**调试(debugging)**。

今后，在真正的代码中，我们应当避免像*x*这样的非描述性变量名。比如，表示图片宽度的变量可以被叫做*pictureWidth*，见例[2]。

```
[2]
pictureWidth = 8;
```

从编译器程序对语句行末尾分号的要求，你会发现编程是十分关注细节的。一个值得关注的细节就是代码是区分大小写的。也就是说问题的关键是你是否使用了大写字母。变量名*pictureWidth*与*pictureWIDTH*或者*PictureWidth*是不同。为了和大多数人的习惯一致，我使用变量名时大多把若干词组合在一起，第一个词的首字母不大，但其它所有词首字母大写，如同例[2]那样。

严格遵守这个约定，可以减少许多因为区分大小写带来的错误。

请注意变量名一般由单个单词构成（必要时也可以是一个字母）。

尽管你在使用变量名上有充分的自由，但依然要遵循一些规则。也许这些规则令人讨厌。最重要的一条规则是你不能使用 Objective-C 语言中的保留字（也就是在 Objective-C 语言中已经有了特殊含义的单词）。使用简明的单词组成变量名，比如 *pictureWidth*，通常是安全的。为了保证变量名的可读性，推荐在其中使用大写字母，这条规则可以让你程序减少错误。

下一条规则是一个变量名不能以数字开头，但数字可以出现在变量名中。另外，使用下划线“_”也是可以的。

下面举例说明以上规则。

合法的变量写法：*door8k*，*do8or*，*do_or*

不合法的变量写法：*door 8*（中间有空格），*8door*（以数字开头）

不推荐的变量写法：*Door8*（开头使用大写字母）

令人惊讶的是编译器并不挑剔空格（但变量名、关键字等除外），为了代码清晰易读，我们可以使用空格。

```
[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

现在请看看例[5]，特别注意前两行语句行。

```
[5]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

数字基本上分为两类：整数和分数。正如语句行[5.1]和[5.2]分别给出的。整数用来计数，比如给出重复某一特定指令的次数（见第七章）。分数或者称作浮点数用来计算例如棒球的击中率。

例[5]给出的代码还不能运行。因为编译器编译代码前需要你指出你会在程序中使用哪些变量名

称，还有这些变量指代什么类型的常量，也就是说指代整数还是浮点数。用术语讲，叫做“声明变量”。

[6]

```
int pictureWidth;
float pictureHeight, pictureSurfaceArea;
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

例[6.1]这一行**int**表示变量*pictureWidth*是一个**整型**（**int**）变量。在下一行我们一次声明了两个变量，中间只需要用逗号隔开。具体地说，语句行[6.2]把两个变量都定义为**单精度**（**float**）变量，也就是说数字部分是包含分数的。在这个例子中把*pictureWidth*与别的变量分别设定为不同类型本没有必要。但要注意，一个整数和一个分数相乘结果是分数，这就是在语句行[6.2]中必须把*pictureSurfaceArea*设定为单精度变量的原因。

为什么编译器程序要求声明变量类型？这是因为计算机程序需要占用部分内存。编译器程序要为每个变量预留出内存空间。不同的数据类型，也就是int型和float型，需要不同的存储空间和代码，编译器程序要预留出足够的空间并使用正确的代码。

如果我们用了一个很大的数或着是一个精度极高的十进制数会怎样？预留的内存空间会不够用么？答案是肯定的。对于这个问题，将有两种答案：一种是int型和float型变量有相对应的形式来存储很大的数或精度较高的数。很多系统采用常整型（**long long**）和双精度（**double**）。尽管第一种方法已经解决了问题，单还有另一个回答：作为一名程序员，关注问题是你的职责。在任何情况下，这一点都应该在一本手册的第一章指出。

整数和十进制数都可以是负数，比如你银行帐户里的数字。如果你知道有些变量的值不可以是负数，那么就可以更合理的安排内存的使用。

[7]

```
unsigned int chocolateBarsInStock;
```

负数对于计算巧克力的块数没有意义，所以无符号整型（unsigned int）变量可以被用在这里。unsigned int型变量的值都大于等于0。

我们可以在定义变量类型的时候同时为它赋值，见例[8]。

[8]

```
int x = 10;
float y= 3.5, z = 42;
```

这样着实减少少打许多字。

前面的例子中我们演示了乘法运算。下面这些符号是进行简单计算的合法运算符。

+ 加法运算

- 减法运算

/ 除法运算

* 乘法运算

使用上面的运算符，我们可以完成大部分计算。但如果你看到过专业Objective-C程序员编写的代码，你会看到许多富有特色的写法，也许因为程序员都是很懒的打字员。

程序员常常用例[9]和例[10]的写法代替$x = x + 1$。

```
[9]
x++;
[10]
++x;
```

这两个符号都表示：x累加1。但某些情况下，“++”符号出现在变量之前或之后是非常关键的信息。看看例[11]和例[12]。

```
[11]
x = 10;
y = 2 * (x++);
[12]
x = 10;
y = 2 * (++x);
```

在例[11]中，当所有过程执行完毕，*y*值等于20，*x*值等于11。（即先使用*x*值，再使$x = x + 1$。）但在语句行[12.2]中，x的值先加1，再与2相乘。(即先使$x = x + 1$，再使用*x*值参与其它运算。)所以，最后结果是*x*等于11，*y*等于22。例[12]的代码实际上等价于例[13]的形式。

```
[13]
x = 10;
x++;
y = 2 * x;
```

通常程序员把两行代码简化成一行。但我个人认为这使程序的可读性变差了。走捷径可以，可要知道这样做可能伴随着潜在的问题。

如果你顺利读完高中，这个问题对你来说易如反掌：括号可以用来规定运算顺序。通常*和 / 优先于＋和－。所以2 * 3 + 4等于10。而使用了括号，就可以使加法优先计算：2 * (3 + 4)等于14。

除法运算应该特别注意，整数和浮点数进行除法运算的结果是不同的。例[14]的结果等于2，而例[15]的结果是2.4。

[14]

```
int x = 5, y = 12, ratio;
ratio = y / x;
```

[15]

```
float x = 5, y = 12, ratio;
ratio = y / x;
```

你可能并不熟悉运算符“%”。运算符“%”用来求两个数的余数（除数不能为0）。

[16]

```
int x = 13, y = 5, remainder;
remainder = x % y;
```

可以算出例[16]的结果，即余数是3，这里x等于 $2 * y + 3$。

一些常见的余数运算结果：

21 % 7 = 0	30 % 2 = 0	50 % 9 = 5
22 % 7 = 1	31 % 2 = 1	60 % 29 = 2
23 % 7 = 2	32 % 2 = 0	
24 % 7 = 3	33 % 2 = 1	
27 % 7 = 6	34 % 2 = 0	

这些数字可以放在手边备用，但注意，这个运算符只对整数有效。

第 2 章 没有注释？那可不行！

使用带有明显含义的变量名可以增强代码的描述性，并使程序容易被理解，如例[1]。

[1]

```
float pictureWidth, pictureHeight, pictureSurfaceArea;
pictureWidth = 8.0;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

我们前面的例子都是一两行简短的代码，真正的代码要长的多。在编程时，首要关注的不仅仅是写出来的东西可以按照你的意图执行，而且要适当能够加入注释说明。以后，在没有看过完整程序又要对程序进行修改的时候，注释能告诉你某段代码是干什么的，为什么的放在这个位置。强烈建议在编程时加入注释，花费在这上面的时间在今后一定会得到回报。另外，如果你和别人分享程序代码，注释可以帮助别人迅速找到他需要的内容。养成写注释的习惯，记得以双斜杠开头。

```
// This is a comment
```

在Xcode中，注释以绿色显示。如果注释很长或者分行，还可以用这个符号把注释括起来：/* */。

```
/* This is a comment
extending over two lines */
```

下面简单谈一下调试程序，Xcode有很强大的功能用于程序调试。一种比较原始的方法就是“放入注释法”。把部分代码用/* */括起来，使其暂时失效（放入注释的程序部分是不被执行的），然后测试其余部分。这个办法可以帮助你发现错误。假如你放入注释的代码正好要给某个变量赋值，你可以临时加上一行代码，为变量临时赋一个合适值用来测试其余的部分。

注释的作用不应被夸大。通常一段关于语句行简明扼要的说明是非常有用的。一旦出了问题，你便可以一目了然而不用先去推测这段代码的作用是什么。你同样可以通过写注释来说明一些通过代码不好说明或不能说明的意思。例如你编写一个数学函数的程序，引用了某本著作中的内容，你就可以通过注释来标明出处。有的时候，在编程之前写注释也是很有用的，注释可以帮助你理顺思路，使编程工作变的简单。

本书给出的例子并没有附加很多注释，因为它们已经被上下文的说明内容包围了。

第 3 章 函数

迄今为止我们见到的最长的一段代码也只包含5行语句，编写带有上千行代码的程序还是后话，但考虑到Objective-C语言的特性，我们要先讨论一下程序的组织方法。

如果程序中是很长的语句行，你将很难发现和修改其中的错误。此外，一段特定的代码可能会在整个程序的多个位置反复出现。如果这样一段代码出了问题，你将不得不一一修改，没准你就会落下一两处——这简直是噩梦！所以人们想出了一些方法来组织代码，有了错误可以方便的修改。

组织代码的具体方法是依照其在程序中的作用进行模块化处理。比如，你写了一组用来计算圆形面积的代码并想看看它运行是否正确，你不必重新阅读所有的语句行来试图发现错误。这段代码就是一个**函数**（**function**），函数有自己的名称，你可以通过函数名找到并运行这个函数。函数是一个十分重要的概念，一段程序中至少包含一个被称为**主函数**（**main( ) function**）的函数。当开始运行程序时，主函数的任务是告诉编译器从哪里开始执行程序。让我们通过例[1]来了解主函数。

```
[1]
main( )
{
// Body of the main( ) function. Put your code here.
}
```

语句行[1.1]给出了函数的名称，也就是“main”，下面跟着一组大括号“{ }”。“main”是一个保留字，而且main( )函数又是必须出现的，所以你在定义你自己的函数名称时应当选用其它的名字。在程序中使用大括号是有理由的，我们会在本章后面的部分讨论。两半大括号中间的部分被称为函数体，实际是我们编写的代码。把第一章中的一些代码放到函数中就成了例[2]。

```
[2]
main( )
    {
       // Variables are declared below
       float pictureWidth, pictureHeight, pictureSurfaceArea;
       // We initialize the variables (we give the variables a value)
       pictureWidth = 8.0;
       pictureHeight = 4.5;
       // Here the actual calculation is performed
       pictureSurfaceArea = pictureWidth * pictureHeight;
     }
```

如果持续不断的增加main( )函数的函数体，那结果依然是难于调试。我们说过要避免杂乱无章的程序组织方式，那么就下面的例[3]就以有序的方式来编写程序，除了必要的main( )函数外，我们还要创建一个circleArea( )函数。

[3]

```
main( )
{
   float pictureWidth, pictureHeight, pictureSurfaceArea;
        pictureWidth = 8.0;
        pictureHeight = 4.5;
        pictureSurfaceArea = pictureWidth * pictureHeight;
     }
     circleArea( ) // [3.9]
     {
 }
```

从语句行[3.9]开始的自定义函数并没有发挥作用。因为这个circleArea( )函数在main( )函数的函数体之外，或者说它没有被嵌套。

circleArea( )函数只用被main( )函数调用才能发挥作用，我们通过例[4]看看是怎么一回事儿。

[4]　　**注：程序其余部分没有写出，请参见例[3]。**

```
main( )
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
  circleRadius, circleSurfaceArea; // [4.4]
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  circleRadius = 5.0; // [4.7]
  pictureSurfaceArea = pictureWidth * pictureHeight;
  // Here we call our function!
  circleSurfaceArea = circleArea(circleRadius); // [4.11]
}
```

我们语句行[4.4]中新声明了一些变量，并且在语句行[4.7]给变量*circleRadius*赋值。语句行[4.11]最重要，它的作用是调用circleArea( )函数。如你说见，变量*circleRadius*放在小括号里面，被作为circleArea( )函数的一个参数。变量*circleRadius*的值被传送到circleArea( )函数。当circleArea( )函数被执行完，它还会返回一个结果。例[5]是我们补全了例[3]中的circleArea( )函数。

[5] **注：仅给出circleArea( )函数的内容。**

```
circleArea(float theRadius) // [5.1]
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius; // pi times r square [5.4]
  return theArea;
}
```

在语句行[5.1]我们定义了circleArea( )函数输入数据的类型是单精度，当接收到数就会被赋给变量*theRadius*。我们在语句行[5.4]使用了另外一个变量*theArea*用来存放计算结果，之前的语句行[5.3]要先它的声明变量的类型，就像例[4]中那样。你已经注意到了，在语句行[5.1]的小括号里就已经声明了变量*theRadius*的类型。语句行[5.5]的作用是把结果返回去，在语句行[4.11]中，变量*circleSurfaceArea*就是用来接收这个返回的数值的。

例[5]说完了，但还差一点儿。我们没有声明返回的数值类型。编译器要求我们必须进行声明，因此比无选择，具体办法见例[6]。

```
[6]
float circleArea(float theRadius)
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}
```

从语句行[6.1]第一个词就可以看出，这个函数返回的数值（也就是变量*theArea*的值）是一个单精度数。作为一名程序员，你还要做的就是确保main( )函数中第[4.8]行变量*circleSurfaceArea*的类型也是单精度，这样编译器对我们的程序就无所挑剔了。

并不是所有的函数都要求参数。即使没有参数也要保留一个空的小括号。

```
[7]
int throwDice()
{
  int noOfEyes;
  // Code to generate a random value from 1 to 6
  return noOfEyes;
}
```

并不是所有的函数都会返回一个数值。如果一个函数不返回结果，那么它应该被定义为“void”。“return”语句就成了可选的。如果你要写这句，那么只需要写出语句定义符，后面不要带任何数值或者变量名。

```
[8]
void beepXTimes(int x);
{
// Code to beep x times.
return;
}
```

如果一个函数有多个参数，那么要用逗号把每个参数分隔开。

```
[9]
float pictureSurfaceArea(float theWidth, float theHeight)
{
// Code here
}
```

习惯上，main( )函数也要返回一个整数值，所以它也需要“return”语句。它以返回0（见语句行[10.9]）表示行数运行正常。为了使main( )函数返回一个整数值，我们需要在它前面加上一个“int”声明（见语句行[10.1]）。下面给出的是一段完整的程序。

```
[10]
int main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
  circleRadius, circleSurfaceArea;
```

```
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
   circleSurfaceArea = circleArea(circleRadius); // [10.9]
    return 0;
    }
  float circleArea(float theRadius) // [10.13]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
     return theArea;
}
```

例[10]中，我们已经写好了一个main( )函数（见[10.1]）和另外一个我们自定义的函数（见[10.13]），但是当我们编译这段程序的时候，编译器还是会遇到困难。在执行语句行[10.9]的时候会报告无法识别名为circleArea( )的函数。显然这是因为编译器在执行main( )函数时突然遇到了一个不为它所指知的东西，但它不会继续向下看就开始报错。为此，我们需要在main( )函数之前再加上一句**函数声明**（**function declaration**）语句,，也就是语句行[11.1]。这行内容和语句行[10.13]完全相同，只是结尾要记得加上分号“；”。

[11] **注：只给出了部分程序，其他未给出部分与例[10]相同。**

```
float circleArea(float theRadius); // function declaration
int main()
{
```

现在可以把代码加以编译并运行了。

当你编程时，建议你要考虑让代码能被重复利用。前面的程序还可以加上一个在例[12]中给出的rectangleArea( )函数，这个函数也将能够被main( )函数调用。有时就算代码只在程序中被使用一次，这个思想也是十分有用的。它能使main( )函数变得简单易读；还能在你调试程序时，使其中的错误更容易被发现。你会发现你不用浏览整篇程序而只要检查大括号内的函数体。

[12]

```
float rectangleArea(float length, float width)
{
  return (length * width);
}
```

在比较简单的情况下中，一个语句行就可以同时完成计算和返回值两个步骤，如[12.3]所示。在前面语句行[10.5]中设置了一个多余的变量*theArea*是为了说明如何在函数中声明变量。

尽管这一章让我们发现创建函数是一件琐碎的事情，但实际上你在修改函数的时候不需要改动函数调用语句，也不用改动函数声明语句（程序最开头的语句行）。例如你可以修改函数体中的变量名，函数依然正常运行，同时也不会影响到程序的其他部分。别人写好的某个函数你可以直接移植到你的程序中使用而不必考虑这个函数的内容如何。你需要知道的不过是：

- 函数名称；
- 函数参数的个数，顺序和数值类型；
- 函数将返回什么值（比如这个值代表是矩形面积）以及值得类型。

如果你要使用例[12]给出的函数，你要知道的内容是：

- 函数名称：rectangleArea；
- 函数包含2个参数，都是单精度数，第一个参数代表长度，第二个代表宽度；
- 函数返回值是举行面积，值的类型依然是单精度（从语句行[12.14]可以看出）

函数中的代码只在函数中有效。这是Objective-C的一个重要特点。在第五章中，我们会再次讨论这个特点。但是现在，让我们启动Xcode运行以下例[11]写好的程序吧。

第 4 章 在屏幕上输出

写好了程序，下面要讨论的是如何显示程序运行的结果。在屏幕上输出运行结果有许多方式，本书介绍的是Cocoa提供的NSLog( )函数。在屏幕上显示出结果是很容易的，你不用担心。

NSLog( )函数函数最初被用来显示错误信息，而不是显示运行结果。因为它易学易用，所以本书把它拿来用作显示结果。今后精通了Cocoa，你还会接触到更多更复杂的技术。

下面就来看看如何使用NSLog( )函数吧。

[1]

```
int main()
{
   NSLog(@"Julia is a pretty actress.");
  return 0;
}
```

运行例[1]结果会显示：“Julia is a pretty actress.”在符号@"和"之间的部分叫做字符串。

除了字符串本身，NSLog( )函数还会附加显示其它信息，比如当时的日期、时间和应用程序名称等。在我的计算机上运行例[1]就会显示以下信息：

```
2005-12-22 17:39:23.084 test[399] Julia is a pretty actress.
```

字符串可以是空的，亦可以包含若干多字符。

[2]　　**注：下面仅节选了main( )函数的一部分。**

```
NSLog(@"");
NSLog(@" ");
```

语句行[2.1]没有包含字符，叫做空字符串，也就是说字符串长度为零。语句行[2.2]不是空字符串，尽管看起来它里面没有内容，实际上它里面包含有一个空格，所以这个字符串的长度为1。

一些特殊的字符序列放在字符串中有特殊意义。比如，要让字符串中最后一个单词另起一行，就可以通过加入“\n”来实现（见语句行[3.1]）。符号“\n”是“另起一行”的简写。

[3]

```
NSLog(@"Julia is a pretty \nactress.");
```

执行例[3]后屏幕应当显示：

```
Julia is a pretty
actress.
```

在语句行[3.1]中出现的反斜杠“\”叫做**转意字符**（**escape character**），它的作用是告诉NSLog( )函数反斜杠后面的字符不是显示在屏幕上的，而是转成另外的含义：这个例子中，字母“n”的意

义转变为“另起一行”。

有些时候你需要在屏幕上输出一个反斜杠“\”，这可能给你出了个难题，因为反斜杠后面的字符都会转变的带有特殊含义。那么怎样输出一个“\”？其实你只需要在前面再加上一个“\”。负负为正，这就告诉NSLog( )函数第二个“\”是被用来显示的，转义的功能将被忽略。

[4]

```
NSLog(@"Julia is a pretty actress.\\n");
```

执行例[4]后会显示：

```
Julia is a pretty actress.\n
```

到目前为止，我们能在屏幕上显示的只是静态的字符串，下面学习的是如何将从计算过程中得到的结果在屏幕上输出。

[5]

```
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@"The value of the integer is %d.", integerToDisplay);
```

请注意，在小括号里有一个字符串和一个变量名，中间用逗号“，”则隔开。字符串中包含一些有趣的信息：“%d”。类似反斜杠，百分号“%”也有特殊含义。如果后面跟一个“d”（十进制数的简写），执行程序以后在“%d”的位置会插入逗号后面的数值，在这个例子中就会插入变量*integerToDisplay*的值。运行例[5]的结果如下：

```
The value of the integer is 6.
```

要显示一个单精度数则用“%f”代替 “%d”。

[6]

```
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %f.", floatToDisplay);
```

输出单精度数时保留小数的位数取决于你的设定。如果只要求保留两位小数那么只需要在百分号“%”和“f”中间加上一个“.2”。

[7]

```
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %.2f.", floatToDisplay);
```

后面我们会介绍如何重复计算，你也许希望借此创建一张数值表。比如一张华氏和摄氏的换算表，如何让表格看起来更漂亮，就需要你控制输出的两栏数据的宽度。你可以通过在“%”和“f”（或者“%”和“d”）中间插入一个整数值来控制输出字符串的宽度。如果给定的宽度小于字符串的本身长度，则按照实际长度显示。

```
[8]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

例[8]运行结果如下：

```
123456
123456
123456
  123456
```

在前两行[8.1，8.2]中，我们给定的输出宽度远远小于实际需要，所以字符串是按照本身长度输出的。在第[8.4]行中宽度值大于实际值，我们看到字符串前面用空格补齐，以占满所有给定的宽度。

控制字符串输出的宽度也可以和保留小数的位数混合在一起定义。

```
[9]
float x=1234.5678
NSLog(@"Reserve a space of 10, and show 2 significant digits.";
NSLog(@"%10.2d", x);
```

当然，你可以同时输出多个数值（见[10.3]）。但你必须确定好输出数值的类型，对应int型或float型使用“%d”或“%f”。

```
[10]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %d, whereas the float value is %f.", x, pi);
```

使用正确的符号对应正确的变量类型十分重要。如果你把第一个弄错了，那么第二个值也不会正确显示！

```
[10b]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %f, whereas the float value is %f.", x, pi);
```

运行结果是：

```
The integer value is 0.000000, whereas the float value is 0.000000.
```

运行我们的第一个程序，还面临一个要解决的问题。如何让程序知道NSLog( )函数？答案是我们告诉它。为此我们编写的程序必须告诉编译器去引入一个包含NSLog( )函数功能的底层库文件。其形式如下：

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

这个语句行必须是程序的首行。

我们现在把本章学习的代码写到一起，同时这也引出了下面一个章节的内容。

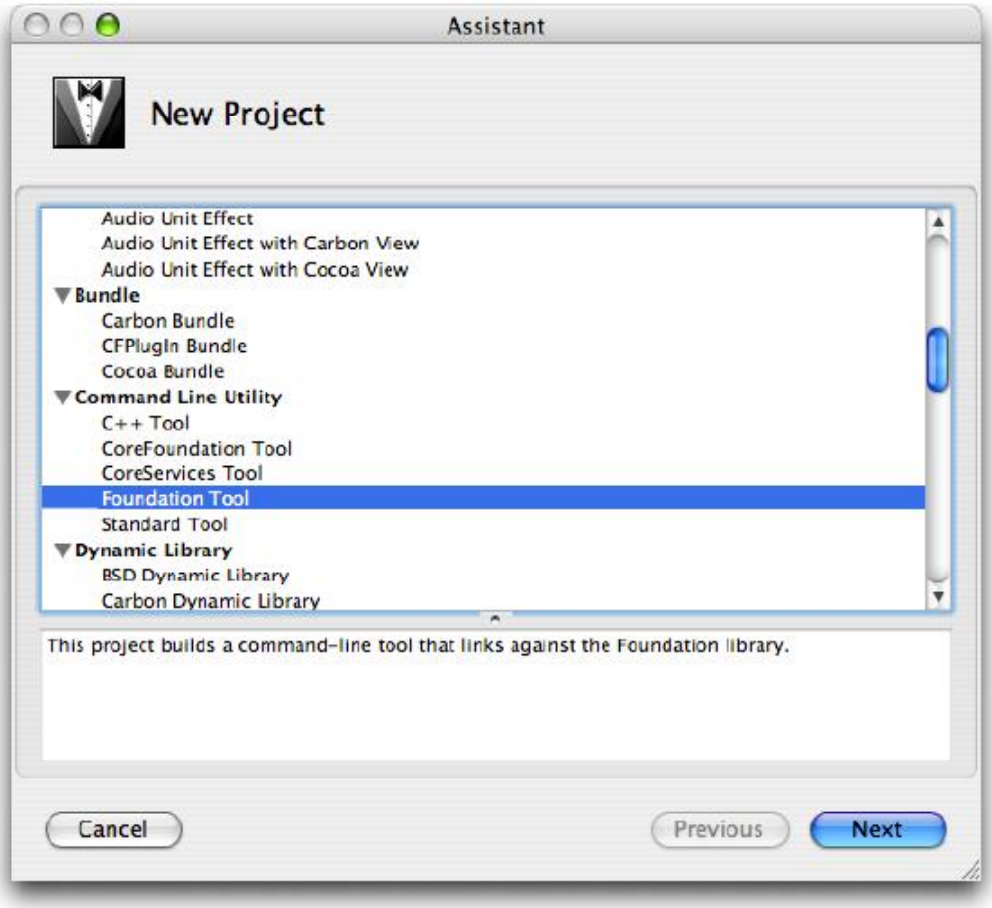
```
[11]
#import <Foundation/Foundation.h>

float circleArea(float theRadius)

float rectangleArea(float width, float height);

int main()
{
  float pictureWidth, pictureHeight, pictureSurfaceArea,
  circleRadius, circleSurfaceArea;
  pictureWidth = 8.0;
  pictureHeight = 4.5;
  circleRadius = 5.0;
  pictureSurfaceArea = rectangleArea(pictureWidth, pictureHeight);
  circleSurfaceArea = circleArea(circleRadius);
  NSLog(@"Area of circle: %10.2f.", circleSurfaceArea);
  NSLog(@"Area of picture: %f. ", pictureSurfaceArea);
```

```
 return 0;
}

float circleArea(float theRadius) // first custom function
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height) // second custom function
{
  return width*height;
}
```

第 5 章 编译和运行一个程序

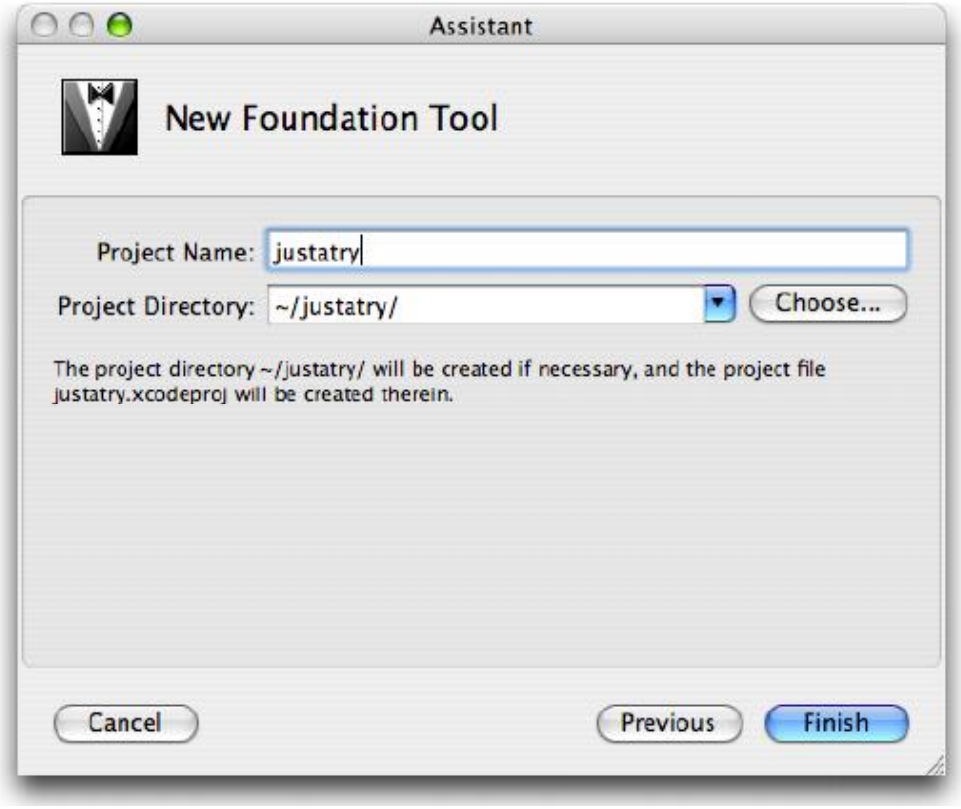
我们写好的代码目前为止还只是人类可以读懂的文字。对我们来说，这些代码比不上散文诗，但对Mac微机来说更糟糕——压根儿就看不懂。一个被称为编译器的特殊程序专门用来把你编写的代码转换成Mac微机可以执行的形式。它是苹果免费开发环境Xcode的一部分。你可以从Mac OS X系统盘中安装，也可以从从苹果网站的开发者专区http://developer.apple.com下载到最新版，从网站下载时要求预先注册一个免费的Apple ID。

现在打开“Developer”文件夹，启动Xcode。当第一次启动Xcode编程环境，会要求你完成一些设定。建议使用默认设置即可，今后有需要时还可以在“预置”面板中进行调整。Xcode启动后，选择“File”菜单中“New Project”命令，会弹出“Assitant”对话框，里面包含了各种可以创建的工程类型。



Xcode的“Assitant”对话框帮你创建一个工程

我们先从最简单的非图形界面的Objective-C程序开始，向下移动滚动条找到并选择“Foundation Tool”标题下的“Command Line Utility”项。



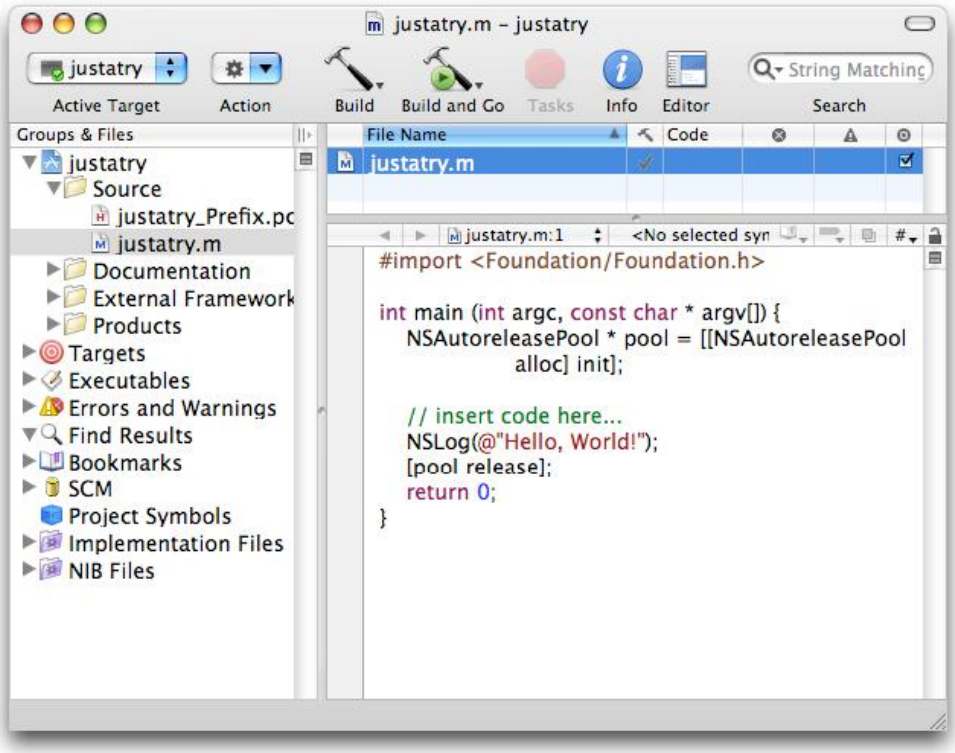
设定新建工程的名称和路径

为你的程序起一个名字，比如叫“justatry”，在选择一个保存路径，之后单击“Finish”。

如果希望编写的程序也能在Terminal上运行，需要你确保工程的名称只有一个单词组成，习惯上还不可以使用大写字母。而为图形界面编写的程序的名字一般以大写字母开头。

现在会有一个窗口呈现在屏幕上，这个窗口将是一名程序员最常用到的。窗口分为两个部分。左边一栏叫做“Groups & Files”，用来处理组成你整个程序的各个素材文件。当前这里面没有太多内容，等后面你开始写多语言图形程序时，将在这一栏里找到图形界面程序和多语言程序需要的素材文件。如果你要在硬盘上一一去查找分散的素材文件可能会比较麻烦，Xcode中提供了虚拟文件夹（被称为“组”（Group））概念，素材文件被分类存放在这些Groups中，可以帮助你方便的组织这些素材。

从左侧“Groups & Files”栏中打开名为“justatry”的组，再打开其下名为“Source”的组，找到其中名为“justatr. m”的文件（见例[1]）。记得么，每一个程序必须包含main( )函数。这个文件的内容就是main( )函数。后面我们还会学习到如何修改这个文件以把我们编写的代码包含进去。如果你是通过双击“justatry. m”的图标打开它的，那么你会惊讶的发现main( )函数的基本内容已经创建好了。



显示在Xcode编成窗口的main( )函数

[1]

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

int main (int argc, const char * argv[]) // [1.3]
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init]; // [1.5]
  // insert code here...
   NSLog(@"Hello, World!");

  [pool release]; //[1.9]
   return 0;
}
```

在上面这段程序中你看到了也注意到了以下内容：

- 被比如NSLog( )函数等函数需要的一个import声明语句，它是以井号“#”开头的一行语句。
- main( )函数
- 括住函数体的大括号

- 一行注释行，告诉我们将函数体放在哪里

- 一个将把一串字符串在屏幕上输出的NSLog( )函数

- 一句内容是“return 0”语句

但是你可能忽略了一些东西：

- 在main( )函数的括号里有两个看起来很有意思的变量声明（见[1.3]）

- 有一句以“NSAutoreleasePool”开头的语句行（见[1.5]）

- 另外一个语句行，其中包含“pool”和“release”字样（见[1.9]）

就我个人而言，如果我是读者，我可不喜欢作者在书上写满了我不熟悉的语句行还信誓旦旦的说后面我就会明白了。所以请你相信，这就是为什么我特意让你先熟悉了函数的概念，这样你就不必和太多的概念撞车了。你现在已经知道了函数是一种组织程序的方式，而且知道每一个程序都包含有main( )函数和其它类似的函数。尽管如此，我还是要承认我现在无法解释清楚你在例[1]中看到的每一项内容。很抱歉，现在只能要求你暂时忽略[1.3、1.5和1.9]这些语句行。因为这里还有些别的关于Objective-C语言的内容你要先熟悉一下，以便着手编写简单的程序。来说一条令人高兴的消息，你已经攻克了难度很大的两章内容，在开始学习更为复杂的内容前，我们马上要学到的三章内容都十分的简单。

如果你好求甚解，可以参考一下这里的简单解释。main( )函数中出现的从前没有见过的内容与程序在Terminal中运行有关。程序运行会占用内存，运行完毕又会把内存交给其它需要的程序。程序员一项重要的工作是要为程序运行预留出内存空间，同等重要的是在程序运行后把内存释放出来。这就是那些包含“pool”字样的语句行的作用。

现在我们就运行一下例[1]。单击标有“Build and Go”标签的锤子图标来（创建）编译并运行程序。

“Build and Go”按钮

程序运行后在“Run Log”窗口显示结果，同时显示的还有一些附加信息。最后一句的含义是程序返回了一个0并退出（终止）。在这里看到的0值是main( )函数返回的，这在第三章例[7.9]中讨论过。我们的程序把它放到了最后一行而没有提前，真是好极了！

现在回过头来再看例[1]，看看假如程序中有错误是什么样的情况。比如我重写NSLog( )函数，并人为的“忘记”在语句行结尾加上分号。

[2]

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

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

{
```

```
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // insert code here...
    NSLog(@"Julia is a pretty actress") //Whoops, forgot the semicolon!
    [pool release]; //[2.9]
    return 0;
}
```

单击工具栏上的“Build and Go”按钮来运行程序，一个红色的圆点出现在了语句行[2.9]前面。

Xcode提示有一处编译错误

如果你单击这个红点，工具栏下方会给出一个简短的错误提示：

```
error: parse error before "release".
```

分析是编译器要做的第一件事：浏览所有的代码并检查是否每一行都可以被理解。帮助编译器理解每个部分的含义，需要你提供线索。import语句要以井号“#”开头（见[2.1]）。为了明确第[2.8]行的结尾，你要加上一个分号。现在编译器停在了第[2.9]行，说明有错误出现。尽管这一行没有错误，但是前面一行少了一个分号。这告诉我们很重要的一点，编译器试图给出可知的错误反馈，但是这个反馈不是对问题的精确描述也不是错误所在的精确位置（尽管会给出相近的位置）。把少了的分号加上再运行一下程序，看看是不是一切正常了。

现在我们把上一章的代码拿来放到Xcode提供的程序（即例[1]）中，这就是下面的例[3]。

```
[3]
#import <Foundation/Foundation.h>

float circleArea(float theRadius); // [3.3]

int main (int argc, const char * argv[]) // [3.5]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  int pictureWidth;
  float pictureHeight, pictureSurfaceArea,
  circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;
```

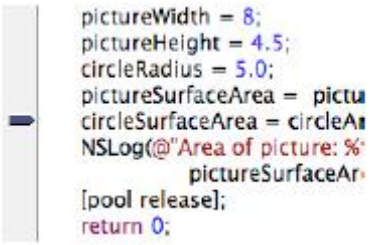
```
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);
    NSLog(@"Area of picture: %f. Area of circle: %10.2f.",
    pictureSurfaceArea, circleSurfaceArea);
    [pool release];
    return 0;
}
float circleArea(float theRadius) // [3.22]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
float rectangleArea(float width, float height) // [3.29]
{
    return width*height;
}
```

花点时间看看自己是不是已经明白了这段程序的结构。我们在开头第[3.3，3.4]行，main( )函数之前，给出了对自定义的circleArea( )函数和rectangleArea( )的函数声明。我们自定已的这两个函数是在main( )函数之外的，而main( )函数的函数体则直接放入到Xcode指定的位置。

运行程序的结果如下：

```
Area of picture: 36.000000. Area of circle: 78.54.
justatry has exited with status 0.
```

程序越是复杂就越难以调试。有时你希望知道程序运行时内部的情况，Xcode让你很容易就能实现这个想法。在想知道运行过程中某个步骤变量的值，只需要在句行前面的灰色条上单击鼠标，语句行前面会出现的蓝色的箭头，标明Xcode在这里插入了一个中断点（breakpoint）。



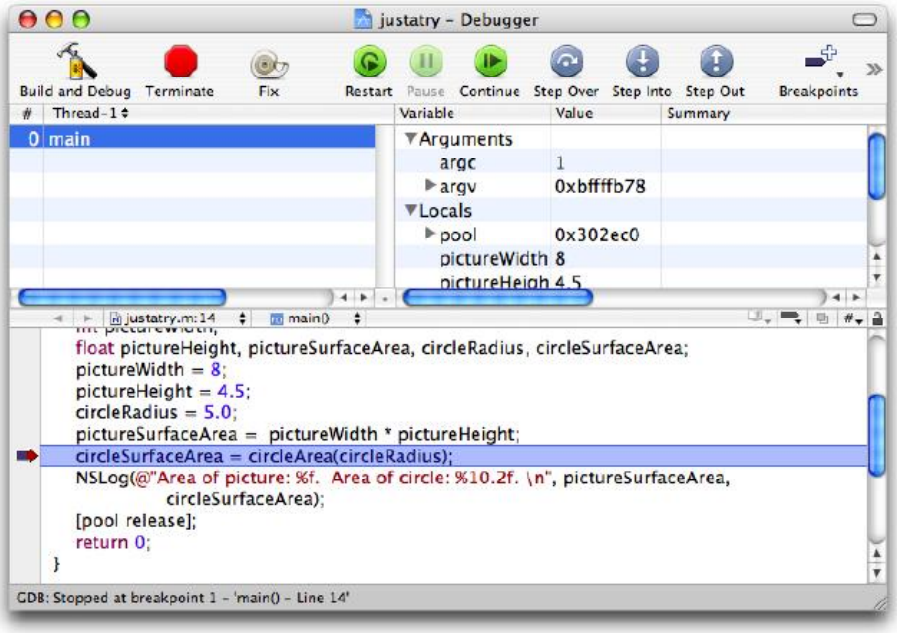
在代码中设置中断点

请注意，你看到的变量值是中断点所在语句行被执行之前的，所以你需要把中断点设定在靠后的一行才可以看到你感兴趣的结果。

现在，用鼠标按住“Build and Go”按钮，一个下拉菜单会弹开。

“Build and Go”按钮列出的下拉菜单

选择“Build and Debug”一项，你会看到下面这个窗口。



Xcode调试器（Debugger）允许你单步运行程序，并随时看到每步结果

程序会在遇到第一个中断点时停下来。你可以通过顶部右侧的窗格擦看变量值。数值和变量的值都是以红色显示的中断点之前设定或计算得出的。单击“Continue”按钮可以向下继续执行。调试器（Debugger）是一个很有用的工具，多试用几次，熟悉它的功能。

我们已经讲完了关于运行和调试简单的Mac OS X程序的所有内容。如果你不想写图形界面程序，那么后面你的工作就是继续积累Objective-C语言的知识，使你能够编写复杂的非图形界面程序。后面的章节我们将涉及这方面内容。再往后将讲述以图形界面为基础的编程。

第 6 章 条件语句

有些时候，我们希望程序在特定的条件下可以完成特定的工作。第[1.2]行中提供的语句就是这个作用。

```
[1]
int age = 42;
if (age > 30) // The > symbol means "greater than"
{
   NSLog(@"age is older than thirty."); //[1.4]
}
NSLog(@"Finished."); //[1.6]
```

在[1.2]行出现的“if”语句是一个条件语句。后面的小括号中列出你给定的条件，整个小括号的逻辑值是真。在例[1]中如果条件“age > 30”满足，则第[1.4]行中的字符串将被输出。但不论条件是否满足，第[1.6]行的字符串都会被输出，因为它没有被包含在if语句的大括号中。

条件满足时执行某个程序，条件不满足时执行另外一段程序，要让“if”语句能够“二选一”就需要使用例[2]中的“if…else”语句。

```
[2]
int age = 42;
if (age > 30)
{
  NSLog(@"age is older than thirty."); //[2.4]
}
Else
{
  NSLog(@"age is not older thirty."); //[2.7]
}
NSLog(@"Finished."); //[1.6]
```

第[2.7]行的字符串只有在条件未满足时才会被输出，实际上在例[2]中没有实际意义。

除了第[2.2]行中的大于号，下面这些比较符号都可以随意使用。

==	等于	>=	大于等于
>	大于	<=	小于等于
<	小于	!=	不等于

特别注意一下“等于”用两个等号“==”表示，这一点很容易被人忘记而只写一个等号。不巧的是“=”是赋值符号，它会把一个特定值赋给变量。对初学者来说，这点儿区别常常被混淆并且造成错误。现在和我一起大声地说：我不会忘了用两个等号表示等于！

比较运算在重复使用某些语句的时候是十分有用的。这是下面一章的话题。现在先来说说“if”语句其他方面的内容，这些东西早晚用得上的。

我们深入的看一下比较运算，一个比较运算无非有两个结果：真或者假。

在Objective-C语言中，真和假分别以1和0代替。它们属于一个特殊的数值类型——布尔型数值（BOOL）。“真”可以用1或者YES代替；“假”可以用0或者“NO”代替。

```
[3]
int x = 3;
BOOL y;
y = (x == 4); // y will be 0.
```

我们可以一次比较多个条件。如果要同时满足一个以上条件，使用逻辑与运算（AND），AND也可以用符号“&&”代替；如果至少满足一个条件则使用逻辑或运算（OR），OR可以使用双竖线“||”代替。

```
[4]
if ( (age >= 18) && (age < 65) )
{
  NSLog(@"Probably has to work for a living.");
}
```

我们也可以嵌套条件语句。下面是把一个条件嵌套进另外一个条件之中的一个简单例子。先判断外层的条件，如果满足，再判断里面一层，以此类推。

```
[5]
if (age >= 18)
{
   if (age < 65)
   {
    NSLog(@"Probably has to work for a living.");
   }
}
```

第 7 章 循环

到目前为止，我们所有的代码都是“一次性”的。我们可以通过重复写函数中的代码来实现重复调用的目的。

```
[1]
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
```

但是即使这样，你还要重复使用调用。有时候你需要执行性一个或几个语句行若干次，如同其它编程语言，Objective-C语言为这种需求提供了解决方案。

如果你知道要重复使用某个或某些语句行的次数，你可以使用例[2]中的“for”语句加上循环的次数。需要注意的是循环次数必须是整数，因为你没有办法循环2.7次。

```
[2]
int x;
for (x = 1; x <= 10; x++)
{
  NSLog(@"Julia is a pretty actress.");
}
NSLog(@"The value of x is %d", x);
```

在例[2]中，第[2.4]行的字符串会被输出10次。首先，*x*被赋值为1。电脑依据后面的公式（*x*<=10）进行判断。此时条件满足（因为这时*x*的值是1），在大括号中的语句行（循环体）被执行。之后*x*的值增加，根据*x*++可以知道是每次加1。随后，*x*的值已经是2了，再与10相比，依然小于10，循环体再次被执行。就这样直到*x*的值累加到11，不再满足*x*<=10的条件。第[2.6]行就是为了证明一下在这个循环结束时*x*的值是11而不是10。

有些时候，我们需要比*x*++更复杂的步骤。你要做的就是用你需要的表达式来代替。例[3]给出的是摄氏度和华氏度的换算程序。

```
[3]
float celsius, tempInFahrenheit;
for (tempInFahrenheit = 0; tempInFahrenheit <= 200; tempInFahrenheit
=tempInFahrenheit + 20)
{
  celsius = (tempInFahrenheit - 32.0) * 5.0 / 9.0;
```

```
    NSLog(@"%10.2f -> %10.2f", tempInFahrenheit, celsius);
}
```

例[2]输出的结果如下：

```
    0.00 ->     -17.78
   20.00 ->      -6.67
   40.00 ->       4.44
   60.00 ->      15.56
   80.00 ->      26.67
  100.00 ->      37.78
  120.00 ->      48.89
  140.00 ->      60.00
  160.00 ->      71.11
  180.00 ->      82.22
  200.00 ->      93.33
```

Objective-C语言还提供了另外两种循环语句：`while () { }`和`do {} while ()`。

前者与我们刚刚讨论的“for”循环语句相当。开始时先判断条件，条件不满足就不会执行循环体（大括号中内容）。

[4]

```
int counter = 1;
while (counter <= 10)
{
  NSLog(@"Julia is a pretty actress.\n");
  counter = counter + 1;
}
NSLog(@"The value of counter is %d", counter);
```

在这个例子中，最后counter的值是11。

而`do {} while ()`循环中，循环体最少被执行1次。

[5]

```
int counter = 1;
do
{
  NSLog(@"Julia is a pretty actress.\n");
  counter = counter + 1;
}
while (counter <= 10);
NSLog(@"The value of counter is %d", counter);
```

这里counter最后还是等于11。

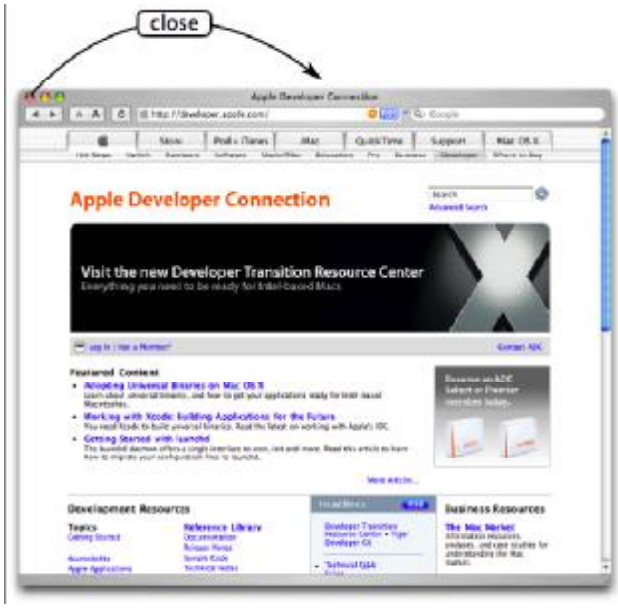
你又学到了一些编程技巧，所以现在要去处理一些更复杂的问题了。下面一章就来写我们的一个图形界面程序。

第 8 章 带有图形界面的程序

随着知识的积累，我们下面将开始讨论如何编写带有图形界面的程序。我要承认，Objective-C语言是C语言的变种。之前我们讨论的很多东西都是纯粹的C语言的内容。那么Objective-C语言和纯粹的C语言有什么不同？区别就在“Objective”上。Objective-C语言把抽象的概念当作对象。

直到现在，我们主要在处理数字。如你所知，Objective-C语言自然支持数字的概念。也就是它允许你在内存中创建数，并通过数学公式和数学函数操纵这些数。如果你的程序用作处理数字，比如计算器程序，那这就很很有用了。但是如果你的程序是一个音乐播放器，用来处理歌曲、播放列表、艺术家名单等等，那该怎么办？亦或者你的程序是一套航空管制系统，要与飞机、航班、机场打交道，又该怎么办？是不是在Objective-C中编写这样的程序如同编写处理数字的程序一样容易呢？上面这些元素（歌曲、艺术家、飞机、机场等等）都是所谓的“对象”。通过Objective-C语言，你可以定义各种你需要处理的对象并为它们编写程序。

通过一个例子看看Objective-C语言编写的程序是如何处理窗口的，比如Safari浏览器的窗口。Mac微机上一个打开的Safari窗口左上角都有三个按钮。红色的是关闭按钮。如果点击这个按钮来关闭一个窗口会发生什么？一条信息会被发到窗口，窗口接着运行一段使自己关闭的代码来回应这段信息。



一条“关闭窗口”的信息被发送到窗口

窗口是一个对象，你可以随意的拖放它。三个按钮也是对象，你可以点击它们。这些对象都有一个在屏幕上可以看见的代表物，但并不是所有的对象都有可以被看见的代表物。例如存在于Safari浏览器和Web服务器之间的连接，就是一个不可见的对象。

一个对象（比如窗口）可以包含其它的对象（比如按钮）

你可以随你的意愿打开任意多的Safari窗口。那么你觉得苹果的开发人员是a.为每一个窗口编写代码，绞尽脑汁预测你最多打开多少个窗口；还是b.创建一个模板使Safari可以随时创建窗口对象？

很显然答案是b。他们编写一些代码，被称作“类”（class），这些代码定义一个窗口的外观和行为。当你新建一个窗口，实际上是这个“类”为你创建了一个窗口。这个“类”就代表了窗口这个概念，或者说每一个窗口都是这个“类”的一个实例。（就像说76是数这个概念的一个实例。）

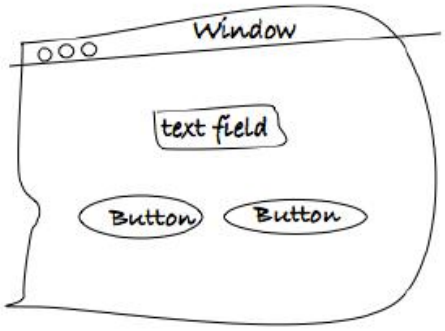
你创建的窗口都在屏幕上有自己特定的位置。如果你把窗口最小化到Dock上又把它还原，它会回到它在屏幕上原来的位置。这是如何实现的？类定义了专门的变量来记录窗口在屏幕上的位置。类的实例，也就是对象，包含有这些变量的值。每个窗口有自己的值，不同的窗口也就有不同的值赋给变量。

类不仅仅创建窗口对象，而且还给它提供一系列动作。其中一个动作就是关闭。当你点击窗口上“close”按钮，按钮就向窗口发送一条“关闭”信息。能够被对象执行的动作叫做**方法**（**methods**）。它们和函数很相似，所以如果你一直跟着我们学到现在，学习它就不会有太多障碍。

当你类创建一个窗口，它同时在内存（RAM）中留出空间来存放窗口的位置信息和其他信息。但是并不包含关闭窗口这个程序代码。那样会浪费计算机的内存空间，因为所有的窗口都是用同样的程序代码执行关闭命令。用来关闭程序的代码只被需要一次，但是每个窗口对象都有通向类中相关代码的途径。

像前面一样，本章下面要看到的例子依然包含着一些涉及内存预留和释放的语句。但是我们仍然要留到后面才讨论。抱歉！

下面要写的这段程序有两个按钮和一个文本区域。当按下第一个按钮，一个值会显示在文本区域；按下第二个按钮另外一个值被显示在文本区域。你可以把它想像成只有两个按钮的计算器，当然它并不能计算。后面你学的多了就可以编写一个真正的计算器了，我只是举一个简单的例子。



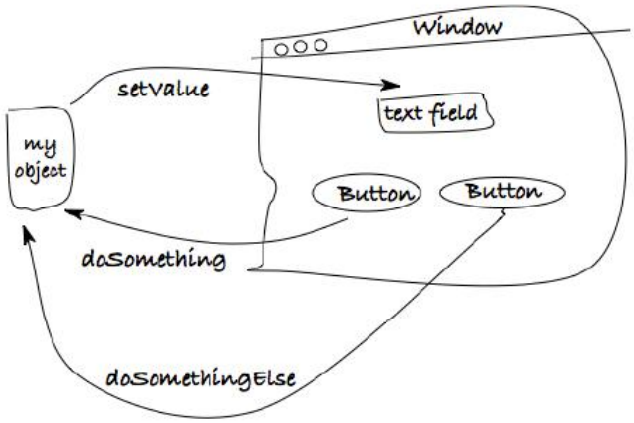
我们即将编写的程序的外观草图

程序上的任何一个按钮被按下，都会传送一条信息。这条信息包含要被执行的方法的名称。那

么这样一条信息将被发送到什么地方？在关闭窗口的例子中，关闭窗口这条信息被发送给了作为窗口类的实例的窗口对象。这里我们同样需要一个对象来接收来自两个按钮的信息，并且还能告诉文本区域显示哪个值。

所以，我们要先创建一个我们自己定义的类，并且创建一个作为它实例的对象。这个对象会作为来自按钮的信息的接收方。（参考下面的草图。）像窗口对象一样，我们的类的实例也是一个对象，但相比之下我们的类的实例在程序运行时并不能像窗口对象那样在屏幕上显示出来。它只是Mac微机内存中的东西。

当我们的类的实例收到来自按钮的信息，相对应的方法就会被执行。方法的代码是包含在类中的而不是包含在类的实例中。运行时，这个方法发送一条信息给文本区域对象。除了文本区域对象的名称，信息还包含文本区域类的一个方法的名称。接收到这条消息，文本区域对象就会执行方法。我们需要文本区域对象在我们单击按钮时现实一个值。所以这条消息除了包含对象名称、方法名称，还会包含一个方法会使用到的参数（值）。



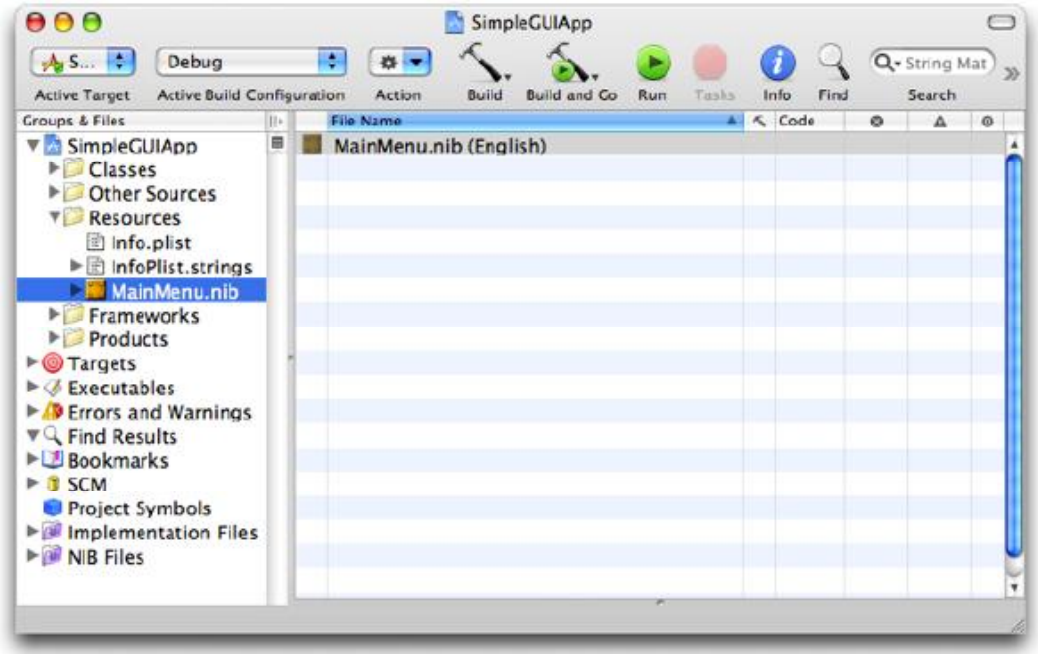
对象间信息传递示意图

下面给出的是Objective-C语言信息传递的一般格式，[1.1]不带参数，[1.2]是带参数的形式。

```
[1]
[receiver message];
[receiver message:argument];
```

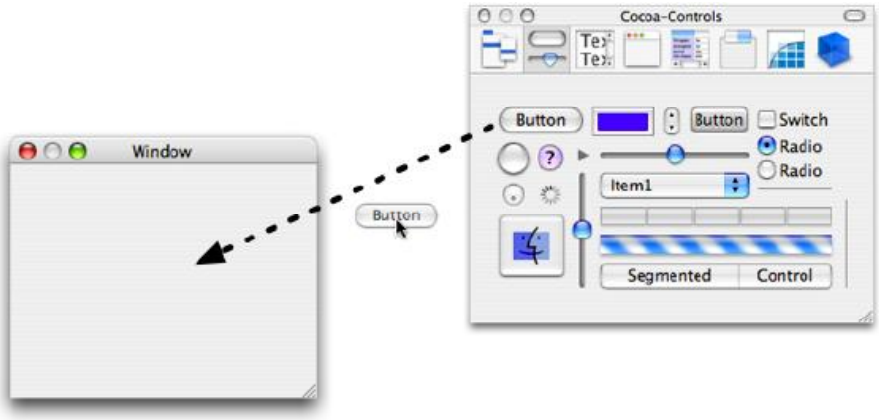
如你所见，所有的工作部分都放在中括号里面，而且要在结尾加上那个永恒不变的分号。在括号里，接收对象的名称放在首位，紧随其后的是方法。如果方法要求跟有参数，那么要使用[1.2]那样的形式。

具体看看上面讲的东西是如何实现的。启动Xcode并创建一个新的工程。在“Application”标题下选择“Cocoa Application”。给你的工程取个名字，按照惯例，图形界面程序的名字一般以大写字母开头。在“Groups & Files”一栏打开“Resources”文件夹。双击“MainMenu.nib”一项。



在Xcode中双击MainMenu.nib文件

另一个名为界面创建器（Interface Builder）的程序就会启动。好多窗口同时出现，你可能都想从“文件”菜单选择“隐藏其它窗口”了。屏幕上应当显示出三个窗口。一个叫做“Window”,你的程序的使用者会看到这个窗口。它有点儿大，你可以调整它。“Window”窗口的右边是一个以“Cocoa-”字样开头的窗口。 它被称作“调板”，里面存放着可以用于你的图像界面的各种对象。单击调板工具栏上第二个图标，并且拖拽两个按钮到你的图形界面窗口“Window”上。单击调板工具栏上第三个图标，把标有“系统字体文本（System Font Text）”字样的文本区域拖拽到图形界面窗口。



从调板拖拽图形界面对象到你自己的应用程序窗口

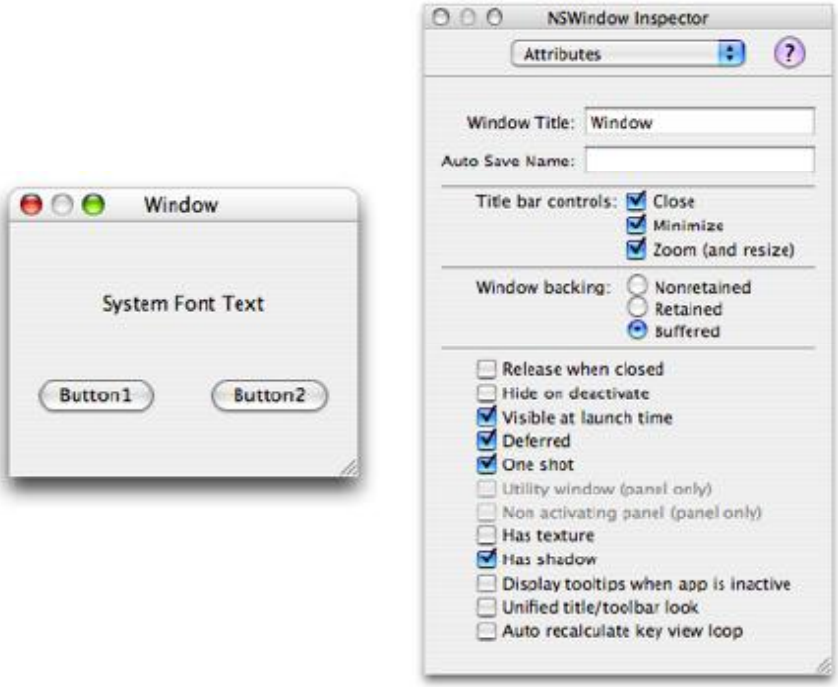
在后台，从调板上拖拽一个按钮实际上是创建了一个新的按钮对象并把这个对象放到了你的程序窗口中。从调板拖拽文本区域或者其它对象情况也是一样的。

请注意，当你把光标停留在调板中的图标上，图标的名字就会被显示出来，比如“NSButton”

或者“NSTextView”等等。这都是苹果提供的类的名称。本章的后面我们会学习在我们自己编写的程序需要执行某一动作时如何从这些类里找到相关的方法。

把“Window”窗口中你刚刚拖拽过来的对象排列美观，并把它们的尺寸调整适当。双击按钮对象可以更改它们的名字，但是要一次一个。我建议你在我们写完这个程序之后继续探索一下调板，掌握其它对象的添加方法。

选择一个对象，按组合键command-shift-i就可以修改它的属性。这个也要好好探究一下。比如选择了“Window”窗口（你可以在左下角的窗口的Instances标签页里看到它已经被选中），按组合键command-shift-i。如果窗口顶部的弹出式菜单上是“Attributes”，你就可以选择勾选“Textured Window”一项，这个选项使窗口呈现金属外观。也就是说，不用写一行代码就可以在很多方面定义你的应用程序的外观。



界面编辑器中的检视窗口和我们编写的“Window”窗口

下面要编写一个类，为此我们还要深入的研究一下类的工作原理。

为了节省编程耗费的精力，最好就是能够在已有的东西上创建你的新东西，而不是从草稿开始着手。举个例子，你要写一个具有特殊属性（功能）的窗口，只要添加那些定义特殊属性的代码就足够了。而不用再去写那些定义其它内容的代码，比如最小化或者关闭窗口。在其他程序员的基础上工作，你可免费继承他们已经写好的东西。这也是Objective-C语言与C语言的不同。

具体的说，这里有一个窗口类（NSWindow），你就能写一个继承它功能的类。假如你添加了某些行为到你自己写的窗口类中。当你自己编写的窗口接收到一条关闭信息时会发生什么？你没有写也没有拷贝任何代码到你的类中。很简单，你编写的窗口的类如果没有包含常规行为的代码时，信息会自动地上溯到它所继承功能的原始的类那里（称作“父类superclass”）。如果有必要会一直

追溯到方法被找到（或者追溯到最为原始的那个类为止）。

如果方法没有被找到，那就意味着你发送了一条无法被处理的信息。这好比你到汽车修理厂修理雪橇，他们却爱莫能助。这个时候，Objective-C语言会报错。

如何去执行你自己定义的行为而不是从父类那里继承来的方法？很容易你就可以跳过个别方法。例如你可以写一段这样的代码：点击关闭按钮只是把窗口隐藏而不真正关闭它。你自己编写的方法要使用和苹果定义的关闭窗口方法一样的方法名称。这样你自己写的窗口收到关闭信息时就会执行你定义的方法，而不是苹果定义的那个。所以这个窗口只会移出你的视线而不是真正关闭。

嘿！真正关闭的方法苹果已经写好了。除了我们写的自己的关闭方法，我们还可以援引父类中提供的关闭方法，这就需要另一个不同的调用语句来确保我们写的关闭方法没有被先使用。

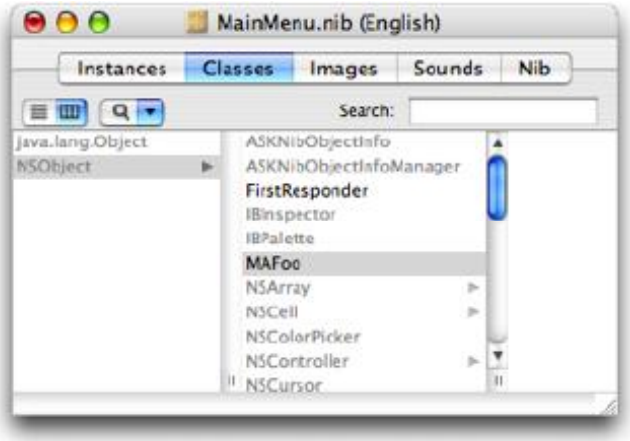
```
[2]
// Code to move the window out of sight here.
[super close]; // Use the close method of the superclass.
```

这些内容对于这本入门级小册子来讲都是些进阶内容，我们可不指望你凭此寥寥数语就学会。

所有类里面，顶级的是被称作“对象类”（NSObject）的类。几乎所有的你创建或使用的类都直接或间接的是对象类的“子类”（subclass）。比如类NSWindow是类NSResponder的子类，类NSResponder又是类NSObject的子类。类NSObject定义了适用所有对象的一般方法。（比如产生一段对对象的文本描述，询问对象是否能理解所给信息等等。）

在这些长篇大论的理论是你厌烦之前，看看如何创建一个类。

回到“MainMeni.nib”窗口，选择“Classes”标签页。在第一栏中你会看到“NSObject”。选择它，并在“Classes”菜单中选择“Subclass NSObject”。回到“MainMeni.nib”窗口，给你创建的类取个名字。我叫它“MAFoo”。



创建名叫MAFoo的类

“MAFoo”这个名字中前面两个大写字母是My Application的缩写。你可以随意给你的类起名字。一旦你开始写程序了，我们建议你使用相似的方法取名（也就是说选择两到三个大写字母作为所有你编写的类的名称的固定的前缀，以免与现有的类发生冲突）。但是不要使用“NS”做前缀，以免引起混淆。NS是苹果自有的类专用的，它代表NeXTStep。Mac OS X就是在NeXTStep操作系统的基础上发展起来的，苹果公司收购了NeXT公司，此举使乔布斯（Steve Jobs）重返苹果并重坐第一把交椅。

在CocoaDev wiki网站上提供了哪些应当避免作为前缀出现的词汇列表。你可以在编程前访问`http://www.cocoadev.com/index.pl?ChooseYourPrefix`进行查询。

当你创建一个类的时候，你应当给它取一个能够传达有效信息的名称。例如在Cocoa中，描绘窗口的类叫做NSWindow。另一个例子是描绘颜色的类叫做NSColor。在我们的例子中，类MAFoo是用来说明对象在应用程序中是怎样互相通讯的。所以我们给他取了一个没什么特殊含义的名字。

回到界面编辑器中在Classes菜单选择“Instantiate MAFoo”一项。就像你现在在“Instances”标签页中看到的一样，你有了一个新的叫做MAFoo的图标。这个图标代表了我们刚刚创建的类的实例。

创建了一个叫做MAFoo的类

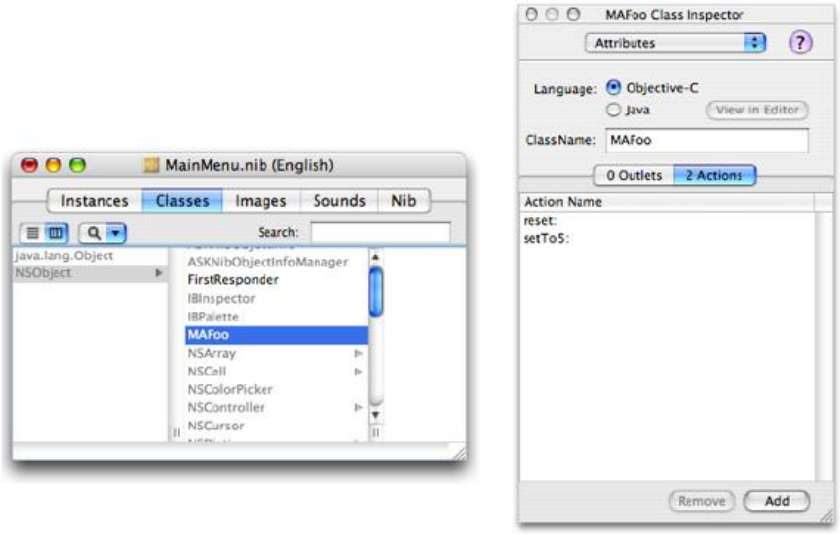
我们下面要作的就是把发送信息的的按钮和我们创建的对象MAFoo关联起来。此外，我们还要创建一个对象MAFoo和文本区域之间的连接，因为还有一条信息回被传送给文本区域对象。如果没有索引，一个对象无法发送信息给其它对象。在按钮和对象MAFoo之间创建关联就是为按钮提供一个通向对象MAFoo的索引。使用这个索引按钮就可以向对象MAFoo发送信息。同样的，建立对象MAFoo与文本区域的关联可以让前者给后者发送信息。

再从新走一遍程序的工作流程。按钮被单击时发送一条定义具体动作的信息。这个信息包含将被执行的类MAFoo中的方法的名称。信息被送到我们创建的类MAFoo的实例——对象MAFoo那里。对象MAFoo本身并不包含执行动作的代码，但类MAFoo中有。所以发送到对象MAFoo的信息转向请求类MAFoo中的方法去做事情：在这个例子中是向文本区域发送内容。像所有的信息一样，这条

信息包含着方法的名称。本例中文本区域对象的方法的任务是显示一个值。这个值被作为信息内容的一部分和方法的名称一起被文本区域援引。

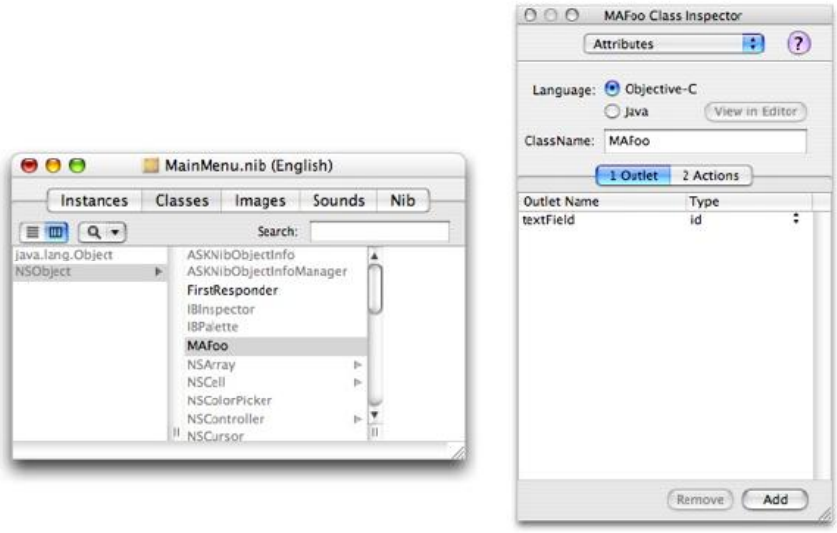
我们的类需要两个动作（方法），这两个方法会被两个按钮分别调用。类还需要一个出口，通过一个变量来记录信息被发向哪个对象。

在“Mainfile.nib”窗口的“Classes”标签页中选中MAFoo。在键盘上按下组合键command-shift-i来调出检视窗口。在检视窗口中选择“Action”标签页并点击“Add”按钮来给类MAFoo添加一个动作（也就是一个方法）。重新命名由界面编辑器默认的名称使名字富有含义。（比如可以叫做“setTo5:”，因为我们会通过这个方法在文本区域显示数字5。）添加其它方法并命名。（比如叫“reset:”，我们会用这个方法在文本区域中显示数字0。）注意方法的名称后面以冒号“：”结尾，后面我们再详细讨论。



在类MAFoo中添加方法

现在在检视窗口中选择“Outlet”标签页，添加出口并命名（比如叫“textField”）。



在类MAFoo中添加出口

在对象之间建立起关联之前，我们还要给两个按钮取名。第一个按钮要在文本区域显示数字5，所以叫“Set to 5”（方法是双击并按钮并键入新名字）。同样，第二个按钮改叫“Reset”。注意，上面给按钮改名这些步骤并不是程序正确运行的要求，而只是想让我们的用户界面看起来尽可能像是面对最终用户的。

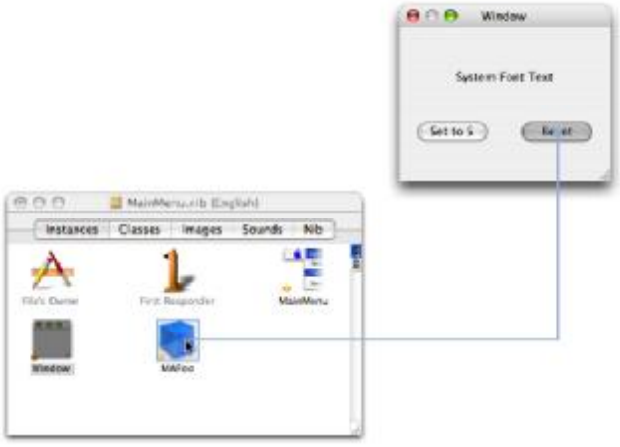
现在我们已经可以创建下面这些关联了：

a）按钮“Reset”和MAFoo的实例之间的关联；

b）按钮“Set to 5”和MAFoo的实例之间的关联；

c）MAFoo的实例和文本区域之间的关联。

在“MainFile.nib”窗口中单击“Instances”标签页。按住键盘上的Ctrl键并用鼠标将“Reset”按钮拖拽到MAFoo的实例上。（千万不要使用别的方法创建关联！）一条代表关联关系的线会显示在屏幕上，确认这条线是从按钮连到了MAFoo的实例上就可以松开鼠标了。



创建关联

当松开鼠标，检视窗口会显示关联调板，调板中列出了对象MAFoo中可用的方法。选择正确的方法并单击“Connect”按钮来完成关联过程。

在检视窗口中完成创建关联

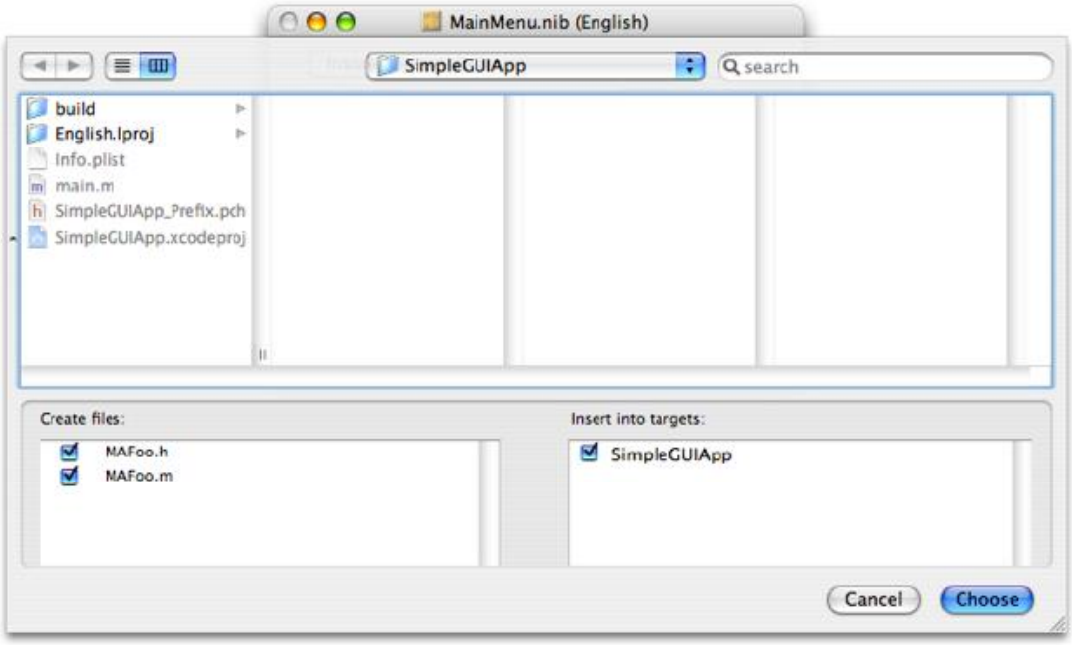
现在按钮已经有了通到对象MAFoo的索引，任何时候被单击都会发送包含“reset:”这一方法名称的信息。

用同样的方法创建“Set to 5”按钮和对象MAFoo的关联。

创建对象MAFoo和文本区域之间的关联时要按住Ctrl键并将对象MAFoo拖拽到文本区域上，再单击“Connect”按钮。

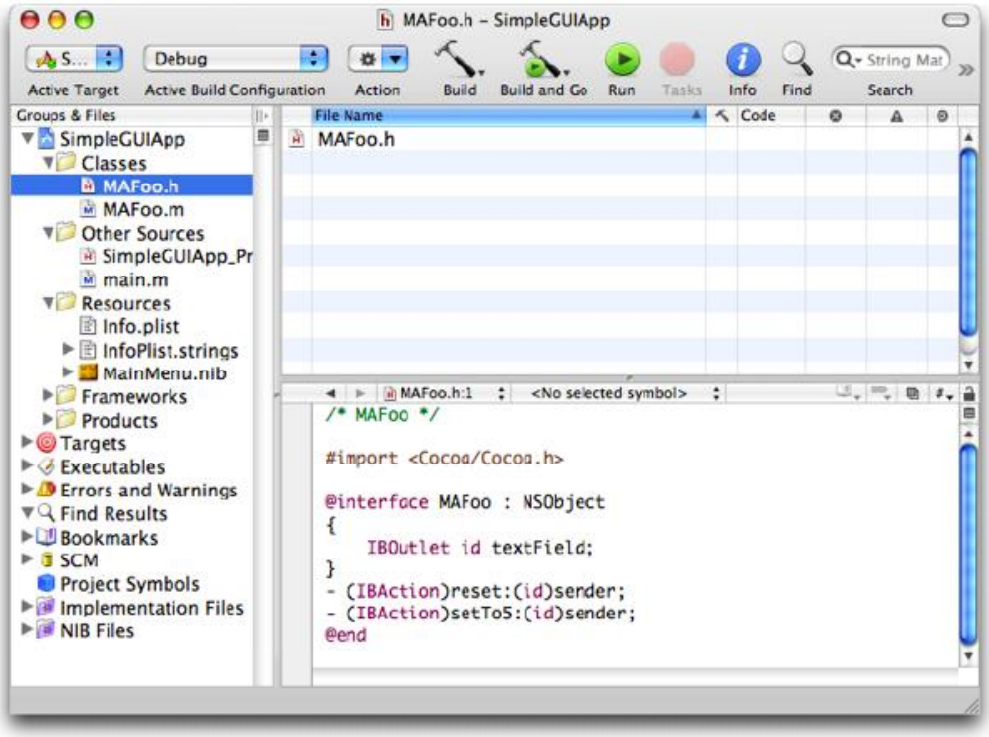
上面你做的就是没有写一行代码就创建了关联。

在“MainMenu.nib”窗口选中MAFoo的实例，切换到“Classes”标签页，你会看到在列表中类MAFoo被选中。在“Classes”菜单中选择“Create Files for MAFoo”。界面编辑器会接着询问生成的文件的保存路径。默认情况下，这些文件会保存到我们保存程序工程的文件夹中，这也正是我们需要的。



为类MAFoo生成代码

现在返回到Xcode中，你可以在工程窗口中看到你创建的文件，这些文件放在“Other Sources”组中。如果需要，你可以把他们拖拽到“Classes”组中，因为你创建的文件本身就代表类。



生成的文件出现在Xcode窗口

现在我们要回顾一下第四章的内容，我们在那一章讨论的是函数。你还记得例[11.1]讲到的函数使用时，你必须在使用函数前告诉编译器引入一些东西。在我们刚刚生成的两个文件中有一个叫

“MAFoo.h”,它是一个头文件：包含关于我们的类的信息。比如，下面第[3.5]行有NSObject字样，表明我们的的类继承自类NSObject。

```
[3]
/* MAFoo */

#import <Cocoa/Cocoa.h> // [3.3]

@interface MAFoo : NSObject // [3.5]
{
    IBOutlet id textField; // [3.7]
}
- (IBAction)reset:(id)sender; // [3.9]
- (IBAction)setTo5:(id)sender; // [3.10]
@end
```

在第[3.7]行，这里有一个去往文本区域的出口。“id”指示对象。“IB”代表界你用来创建代码的面编辑器。

第[3.9,3.10]行中“IBAction”在这里是无效的。没有信息会返回给作为信息发送方的对象：我们程序中的这些按钮不会从对象MAFoo那里得到回应信息。

你还会看到这里有两个界面编辑器动作（Interface Builder Actions）。

前面我们见过和第[3.3]行相似的内容：`#import <Foundation/Foundation.h>`。尖括号中的内容前者是用于非图形界面程序的，后者用于图形界面程序。

让我们看看第二个文件：MAFoo.m。又是一些免费代码。

```
[4]
#import "MAFoo.h"

@implementation MAFoo

-  (IBAction)reset:(id)sender // [4.5]
{
}
-  (IBAction)setTo5:(id)sender
```

```
{
}
@end
```

首先，头文件MAFoo.h被输入，所以编译器知道要做什么。两个方法“reset:”和“setTo5:”将可以被识别。它们是我们的类的方法。它们和函数类似，所以你需要把代码写在大括号中。在我们的程序中，当你按下一个按钮，按钮会发送一条信息到对象MAFoo，要求执行一个方法。但我们并没有写相关的代码，我们做的仅仅是在界面编辑器中把按钮和对象MAFoo连接起来了。尽管如此，我们必须写出从对象MAFoo发送到文本区域对象的信息的代码。我们在第[5.7,5.12]行给出了这些代码。

```
[5]
#import "MAFoo.h"

@implementation MAFoo

-  (IBAction)reset:(id)sender
{
   [textField setIntValue:0]; // [5.7]
}

-  (IBAction)setTo5:(id)sender
{
   [textField setIntValue:5]; // [5.12]
}
@end
```

如你所见，我们发送了一条消息以出口名称“textField”为参数的信息。因为我们已经在界面编辑器中关联了这个出口到真正的文本区域，所以信息会被发送到正确的对象。这条信息包含方法的名字“setIntValue:”，同时还有一个整数值作为参数。方法setIntValue:能在文本区域对象上显示一个整数值。下面一章我们会讨论如何创建这样的方法。

你现在已经可以编译程序并启动它了。按下Xcode工具条上的“Build and Go”按钮。Xcode可能需要几秒中时间来创建并且运行你编写的程序。最后，程序会出现在屏幕上，你可以试试了。

我们的程序在运行

简单的说，我们创建了一个（很基本的）程序，你仅仅为它写了两行代码。

第 9 章 寻找方法

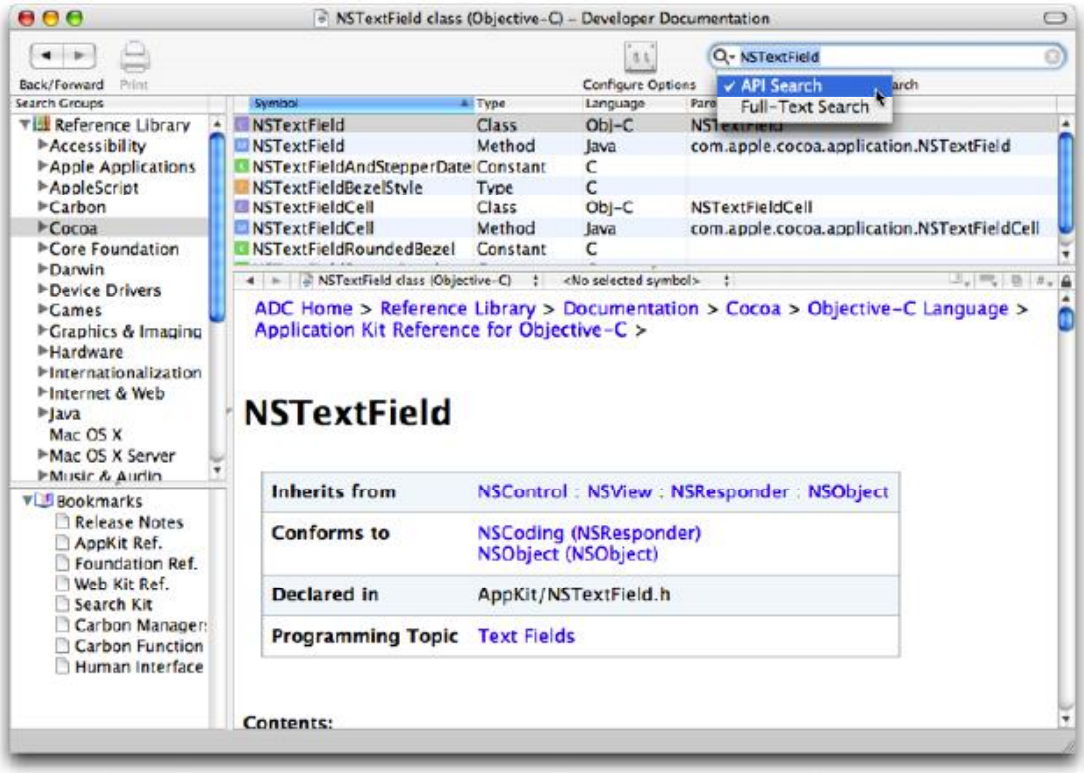
前面的章节我们已经学到了一些关于方法的知识。我们自己写了两段方法体，但是我们只用了一个苹果提供的方法。setIntValue:是一个用来在文本区域对象上显示数值的方法。那么如何找到合适的方法？

> 记住，我们用到的所有方法都是苹果创建的，你不需要写一句代码。而且，是基本上不会有错误在里面的。因此，选用正确的方法比自己编程更加有效。

在界面编辑器中，如果你把光标停留在调板中的某个对象上，一个小标签就会弹出。比如光标停在按钮上，弹出标签上就会显示“NSButton”；如果光标放在名为“System Font Text”的文本区域上，你就会看到弹出标签上显示“NSTextField”。这些都是类的名字。我们一起看看有哪些方法适合类NSTextField。

到Xcode中，选择“Help”菜单下面的“Documentation”。在左侧的栏中选择“Cocoa”，在右上角的搜索栏输入NSTexField（要确保使用API-Search模式）。输入的过程中，列表不断根据关键词缩小范围，最终NSTextField显示在了首行。

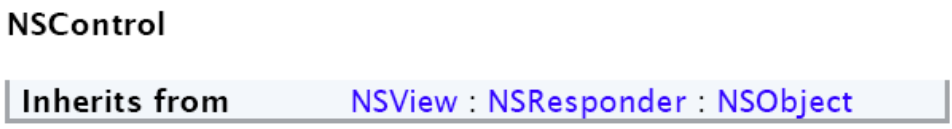
单击“NSTexField”（类型是Classes）的这一行，就可以获得更多信息。



在Cocoa说明文件中浏览

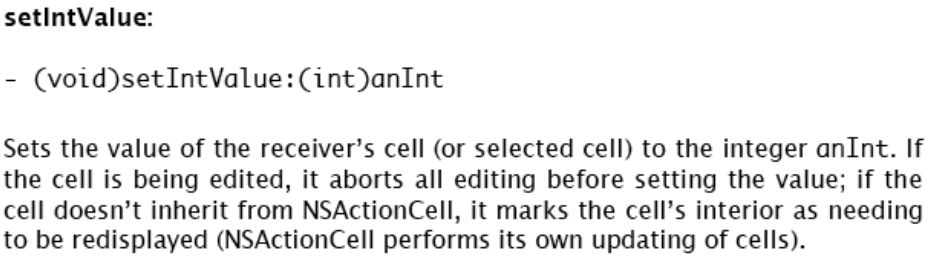
首先你注意到了这个类继承自一系列的父类。最后一个，处于支配地位的是类NSObject。向下滚动屏幕你会看到这个标题：Method Types。

我们就从这里开始我们的学习。快速浏览一下子标题你会发现这里没有在文本区域显示数值的方法。根据继承原则，我们需要访问父类NSControl（如果还没有，那就继续上溯到类NSControl的父类NSView查找）。因为所有的文件都是超文本文件，我们只需要点击带下划线的单词NSControl（它被显示在“Inherits from”列表中）就会被带领到类NSControl的说明信息中。



滚动屏幕，你会注意到方法列表中有这样一行子标题：Setting the control’s value。

这就是我们要的，我们想做的就是设定数值。在这个子标题下我们发现“`- setIntValue:`”看起来差不多。单击“setIntValue:”链接进一步查看说明。



我们的程序中，对象NSTextField是接收方，我们需要发送给它一个整数值。我们也可以从这个方法的语法（signature）上看出来：“`- (void)setIntValue:(int)anInt`”。

在Objective-C语言中负号“－”写在**实例方法**（**instance method**）的前头（与**类方法**（**class method**）相反，我们在后面讨论这个问题）。“void”表示没有信息从这个对象的援引者那里返回。就是说，当我们发送了“setIntValue:”信息到“textField”，我们的对象MAFoo不会收到一条从文本区域对象返回的信息。好的。在冒号后面，“（int）”表示变量***anInt***必须是一个整型的。在我们的例子中，我们使用的0或5都是整数，所以没有问题。

有时找到合适的方法可能有些难度。当你对说明文件（documentation）的内容比较熟悉以后就会变得容易些，所以要加强练习。

如果要读取文本区域对象“textField”中的值该怎么办？还记得么，函数中的变量只能用在这个函数中。这个特点也适用于方法。通常对象中会有一对相关的方法，它们互称做是对方的“Accessors”，比如一个用来读取值，另一个用来设定值。我们已经知道了后者，就是“setIntValue:”方法，前者则是这样的形式：

```
[1]
- (int) intValue
```

这个方法返回一个整数值。如果你想看到与“textfield”对象相关的整数值，可以发送这样的信息：

```
[2]
resultReceived = [textField intValue];
```

重申一下，在函数和方法中的所有变量只在这个函数和方法中有效。这一点很有用，因为你不必担心程序中某一部分使用过的变量会干扰其他函数中同名的变量。尽管如此，函数的名称必须唯一。Objective-C不允许同一个类包含重名的方法，但是不同的类中可以使用同名的方法。这一点对大型程序十分有用，你不必为每一个类单独写一段代码，因为不必担心方法的名字会冲突。

在不同的类中不同的方法可以使用一样的名字被称为“同质异像”（polymorphism），它是使面向对象编程与众不同的特征之一。它允许你编写代码模块而不必考虑将来控制哪些对象、哪些类。所有要做的就是运行时让对象读懂你发给它们的消息。为了应用这个优势特性，你的程序需要写的灵活并且有可扩展性。比如我们创建的图形界面程序，如果用别的对象置换文本区域，但这个新对象可以可以应用“setIntValue:”信息，那么我们的程序不用修改代码，甚至不用重新编译就可以正常运行。我们可以在程序运行过程中不间断程序而替换对象。这就是面向对象编程的魅力所在。

第 10 章 awakeFromNib 方法

苹果已经做好了很多东西使编程更简单。在你编写程序时，不必为在屏幕上画窗口和按钮或者其它的东西而担心。这样的大部分工作都已经由两个架构（framework）完成了。在第4章例[12]中使用的“基本工具”（Foundation Kit）提供了大部分与非图形界面编程相关的服务。另外一个架构被称做“应用程序工具”（Application Kit），用于处理图形界面编程的各种对象。两种架构都带有详细的说明。

让我们回到我们的图形界面程序。假设我们的程序在运行后窗口刚刚显示在屏幕上时就立刻在文本区域显示一个值。

关于窗口的所有信息都储存在nib文件中（“nib”是NeXT界面编辑器的缩写）。这也就是说我们需要的方法可能就包含在“应用程序工具”（Application Kit）中。让我们看看如何获取关于这个架构的信息。在Xcode中选择“Help”菜单下“Documentation”选项。

在“Documentation”窗口，启动“Full-Text Search”模式（单击窗口右上角的查找框的下面的，在弹出菜单上选择），在查找框中输入“Application Kit”并回车。Xcode会给出你相关的结果。其中有一个名为“Application Kit Reference for Objective-C”文档，在这个文档中你会看到这个架构所提供的项目列表。在标题“Protocols”之下是一个名为“NSNibAwakening”的链接。

NSNibAwaking

Declared in	AppKit/NSNibLoading.h
Programming Topic	Loading Resources

Contents:

Protocol Description
Method Types
Instance Methods

Protocol Description

This informal protocol consists of a single method, awakeFromNib. Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

如果执行这个方法，它会在对象被装入时从它的nib文件中被调用。因此我们可以利用它来达成我们的目标：在程序启动时在文本区域中显示一个特定值。

我不建议总是零碎的查找正确的方法。通常我们需要适当的浏览或者创造性的使用关键字查找，以此确定需要的方法。因此，熟悉关于这两个结构的说明文件十分重要，这样你就可以知道哪些类或方法是适用的。也许现在还用不到，但今会在编程时给你很大帮助。

好了，现在我们开始找我们的的方法，我们要做的是把方法添加到可执行文件“MAFoo.m”，见例[1.15]。

```
[1]
#import "MAFoo.h"

@implementation MAFoo

- (IBAction)reset:(id)sender
{
   [textField setIntValue:0];
}

- (IBAction)setTo5:(id)sender
{
   [textField setIntValue:5];
}

- (void)awakeFromNib // [1.15]
{
  [textField setIntValue:0];
}
@end
```

当窗口打开，方法awakeFromNib就会自动被调用。结果是你看到文本区域里显示了一个零“0”。

第 11 章 指针

注意！本章内容包含更深层次的概念并将处理一些C语言底层的概念，这些东西往往让初学者头痛。如果你现在还看不懂也请先不要着急。尽管从大体上说理解指针的工作原理很有用，但对Objective-C语言初级编程并不是那么重要。

当你定义一个变量，你的Mac微机就会将这个变量和一个内存空间关联起来以便储存这个变量的值。

看看下面的例子：

[1]

```
int x = 4;
```

为了执行这个语句，你的Mac微机在内存中找到没有被占用的空间并在那里储存变量*x*的值（当然我们可以使用其它的变量名）。再看例[1]，指明变量的类型（这里定义的是整型）就是让电脑知道要留出多少空间来储存变量*x*的值。如果数值被定义为长整型或者双精度，那么则需要更多的内存空间。

语句行“x = 4”的任务是将数值4存入预留的空间。当然，你的电脑知道在内存的哪个空间储存着变量*x*的值，换句话说，它知道*x*的**地址**（**address**）。这样，每次在程序中使用x，你的电脑就能找到正确的位置（在正确的地址）并找到*x*的确切值。

一个**指针变量**就是一个包含其它的变量地址的变量。

只要存在一个变量，你能够通过在它前面写上符号“&”来得到它的地址。比如要得到*x*的地址则写成“*&x*”。

电脑为表达式*x*赋值，变量*x*就会返回一个数值（在我们的例子中返回的数值是4）。相比之下，当电脑为表达式*&x*赋值，将返回变量*x*的在内存中的地址而不是存储在其中的数值。地址是表示电脑的内存一个特定位置的数值（如同房间号表示旅馆的某一个特定房间）。

这样来定义一个指针变量：

[2]

```
int *y;
```

上面的语句行定义了一个名为*y*的变量，它将代表另一个整型变量的地址。将变量*x*的地址存储到指针变量*y*中（术语叫做将*x*的地址指派给y）你需要这样做：

[3]

```
y = &x;
```

在指针变量前面加一个星号“*”得到的是指针所指向的内存存储空间内的数值。表达式“*y”的值是4，它等同于表达式“x”。同理，如果执行了语句“*y = 5”则等同于“x = 5”。

指针之所以有用是因为有时候你不需要变量的数值，但需要用到变量的地址。比如编写一个函数，内容是1加上一个变量，你就会用到变量地址。这是因为你会更改变量的值，而不总是使用当前值。因此，我们这样使用指针：

[4]

```
void increment(int *y)
{
  *y = *y + 1;
}
```

之后，你可以调用这个函数：

[5]

```
int x = 4;
increment(&x);
// now x is equal to 5
```

第 12 章 字符串

至此，我们已经见到过很多数据类型了：整型，长整型，单精度，双精度，布尔型；以及上一章我们介绍的指针。这里我们还要接触另外一种类型：字符串。本章只讨论与NSLog ( )函数有关的字符串操作。这个函数允许我们将字符串输出到屏幕上，格式控制字符串使用一个带百分号“%”的符号开头，比如%d，后面加上一个值。

```
[1]
float piValue = 3.1416;
NSLog(@"Here are three examples of strings printed to the screen.\n");
NSLog(@"Pi approximates %10.4f.\n", piValue);
NSLog(@"The number of eyes of a dice is %d.\n", 6);
```

我们有充分的理由作为一种数据类型来讨论。它们也可以被看作是由类NSString或者类NSMutableString创建的对象。我们下面就研究一下这两个类，先从NSString开始。

```
[2]
NSString *favoriteComputer;
favoriteComputer = @"Mac!";
NSLog(favoriteComputer);
```

上面的程序第二行很好理解，但是第一行（语句行[2.1]）需要一些解释。记得么，我们定义一个指针变量的时候，需要声明指针指向的数据是何种类型。这是第11章例[2]的语句行：

```
[3]
int *y;
```

这个语句告诉编译器指针变量*y*包含的内存地址内存储的是一个整型数值。通过语句行[2.1]我们同样是告诉编译器指针变量*favoriteComputer*包含的内存地址存储的是一个NSString类型的对象。我们使用指针来储存字符串还是因为Objective-C语言并不直接操作对象，而总是借助指针。

好了，为什么“@”这个有趣的符号总是出现？在Objective-C语言是C语言的一个变种，它自有一套处理字符串的方法。为了区分这种新型的字符串，Objective-C语言使用了“@”符号。

那么Objective-C语言的字符串相比C语言的字符串有哪些改进？Objective-C语言的字符串使用Unicode编码来代替ASCII编码。Unicode字符串几乎不受语言限制，甚至可以是中文或者是罗马字母。

当然，可以同时声明并初始化一个用于字符串操作的指针变量，参见例[4]。

[4]

```
NSString *favoriteActress = @"Julia";
```

指针变量*favoriteActress*指向内存中的一个位置，这个位置存储着字符串“Julia”。

当变量，也就是*favoriteComputer*，被初始化以后，你也许要赋给它另一个字符串，但是你并不能改变原有的字符串本身，因为它是类NSString的一个实例。来看看例[5]。

[5]

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  NSString *x;
  x = @"iBook"; // [5.7]
  x = @"MacBook Pro Intel"; // Hey, I'm just trying to make
                            // this book look up to date!
  NSLog(x);
  [pool release];
  return 0;
}
```

运行结果如下：

```
MacBook Pro Intel
```

由类NSString创建的字符串被叫做固定字符串，因为它不可修改。

不可以修改的字符串有什么优点？应当说这样的字符串更容易为操作系统处理，所以你的程序也可以更快的运行。事实上，当你使用Objective-C编程，你会发现大多数时候你并不需要修改字符串。

当然，有时你确实要修改它们。你可以使用另外一个类——NSMutableString，这个类创建的字符串是可以被修改的。我们在本章后面再讨论这个类的使用。现在先来让你搞明白为什么字符串是一种对象。作为对象，我们可以向他们发送信息。例[6]中我们就向字符串对象发送一条length信息。

[6]

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

```
    NSString * foo;
    foo = @"Julia!";
    theLength = [foo length]; // [6.10]
    NSLog(@"The length is %d.", theLength);
    [pool release];
    return 0;
}
```

运行结果是

```
The length is 6.
```

人们在编程时常常用*foo*和*bar*作为变量名称。事实上，它们是很糟糕的变量名，因为它们的描述性很差，就如同*x*。我们在这里使用它们，就是希望你今后在因特网上看到关于它们的讨论时不要感到迷惑。

在第[10.6]行，我们向对象foo发送了一条信息“length”。方法length定义在类NSString中：

```
- (unsigned int)length

Returns the number of Unicode characters in the receiver.
```

你也可以将字符串中的字母全部转成大写（参见例[7]）。向字符串对象发送合适的方法，也就是uppercaseString，你自己可以在说明文件中找到它（查找可以用于类NSString的方法）。接收到包含这个方法名的信息，字符串会声称一个新的串，内容与原字符串相同，但是字母全部变成大写。

```
[7]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *foo, *bar;
    foo = @"Julia!";
    bar = [foo uppercaseString];
    NSLog(@"%@ is converted into %@.", foo, bar);
    [pool release];
    return 0;
}
```

运行结果是

```
Julia! is converted into JULIA!
```

有时候，你可能希望修改一个已经存在的字符串，而不是在创建一个。这时就要用到类NSMutableString的对象来代表你的字符串。类NSMutableString提供了若干修改字符串内容的方法。比如，方法appendString：可以在作为方法信息接收方的字符串末尾加上新的内容。

[8]

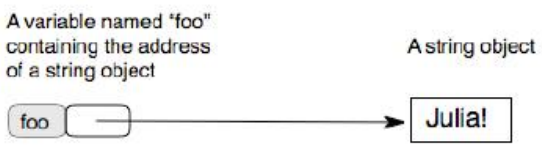
```
#import <Foundation/Foundation.h>

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

NSMutableString *foo; // [8.7]
  foo = [@"Julia!" mutableCopy]; // [8.8]
  [foo appendString:@" I am happy"];
  NSLog(@"Here is the result: %@.", foo);
  [pool release];
  return 0;
}
```

运行结果是

```
Here is the result: Julia! I am happy.
```

在第[8.8]行，方法mutableCopy（由类NSString提供）复制了一个新的可变字符串作为方法信息的接收方。执行了语句行[8.8]，变量*foo*就指向了一个可变字符串对象，这个字符串的内容是“Julia！”。

前面我们说了，Objective-C语言并不直接操作对象，而总是借助指针。这就是为什么，例如，我们在[8.7]行上使用指针标记。事实上，当我们提到Objective-C语言中的“对象”时，我们往往指的是“指向对象的指针”；但是由于我们总是通过指针来操作对象，所以就简称为“对象”了。至此，你还要理解一个重要的概念：若干变量可以同时代表同一个对象。比如执行了语句行[8.7]，变量*foo*就指向了代表字符串“Julia！”的对象，我们用下图说明：

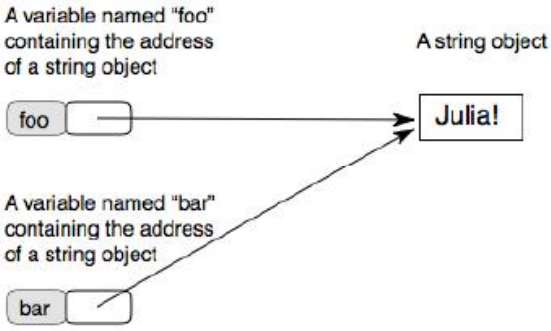


对象总是借助指针被操作

现在假设我们将*foo*的值赋给变量*bar*：

```
bar = foo;
```

结果是变量*foo*和*bar*都指向了同一个对象：



不同的变量可以指向同一个对象

在这种情况下，向对象发送信息，不论是一foo为接收方（比如 [foo dosomething]）还是一bar为接收方（比如 [bar dosomething]）效果完全相同，参见下面的例子：

[9]

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
   NSMutableString *foo = [@"Julia!" mutableCopy];
   NSMutableString *bar = foo;
   NSLog(@"foo points to the string: %@.", foo);
   NSLog(@"bar points to the string: %@.", bar);
   NSLog(@"-----------------------------");
   [foo appendString:@" I am happy"];
   NSLog(@"foo points to the string: %@.", foo);
   NSLog(@"bar points to the string: %@.", bar);
   [pool release];
   return 0;
}
```

运行结果是

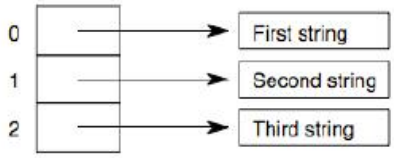
```
foo points to the string: Julia!
bar points to the string: Julia!
-----------------------------
foo points to the string: Julia! I am happy
bar points to the string: Julia! I am happy
```

从不同地点可以同时原因同一个对象是面向对象变成的语言的一个特性。实际上你在前面已经用到了这个特性。比如第8章中，我们让两个按钮关联到我们创建的MAFoo对象上。

第 13 章 数组

有些时候，你需要使用一系列数据。比如一堆字符串，将每一个串都赋一个变量将是一件很麻烦的事情。当然，这里有更好的解决方案：数组（array）。一个数组是一组有序的对象列表（更具体的说是一组对象指针的列表）。你可以向数组添加对象，移除对象，或者查看数组的给定位置上存放了哪个对象。你也可以查看数组里包含了几个元素。

我们数数习惯由1开始。在数组中，第一个元素的索引是0，第二个索引是1，以此类推。



举例：包含了三组字符串的一个数组

本章后面的部分还会有一些例子来说明从0开始计数的作用。

有两个类提供了数组：NSArray和NSMutableArray。带有字符串的数组分为固定数组和可变数组两种。在本章中，我们只体积可变数组。

通过执行下面这个步骤可以创建数组：

```
[NSMutableArray array]
```

这个语句会产生一个空数组。但是……请等等，这句代码有些古怪，不是么？实际上，这个例子中我们使用类NSMutalbeArray的名字来说明信息接收方。但是我们希望向类的一个实例发送信息，而不是类本身。

好了，我们又学到了一些新的东西：事实上，在Objective-C语言中，我们可以向类发送信息（原因是类本身也是对象，它是被元类（meta-class）的实例，但本书中不再就此问题做深入探讨）。在Cocoa的说明文件中，能够直接作用于类的方法（类方法）名称前用加号“＋”标示，与前面表示减号“－”的实例方法不同（见第8章例[4.5]）。 在说明文件中我们可以见到对方法array这样的描述：

array

```
+ (id)array
```

Creates and returns an empty array. This method is used by mutable subclasses of NSArray.

See Also: `+ arrayWithObject:`, `+ arrayWithObjects:`

再回到代码中，下面的程序用于创建一个空的数组，并将三个字符串储存在其中，之后显示数组元素个数。

```
[1]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"first string"];
  [myArray addObject:@"second string"];
  [myArray addObject:@"third string"];
  int count = [myArray count];
  NSLog(@"There are %d elements in my array", count);
  [pool release];
  return 0;
}
```

运行结果如下：

```
There are 3 elements in my array
```

接下来的程序和前面的大体一致，只是最后显示存储在索引0中的的字符串。为此，我们在语句行[2.13]中使用了方法“objectAtIndex:”。

```
[2]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"first string"];
  [myArray addObject:@"second string"];
  [myArray addObject:@"third string"];
  NSString *element = [myArray objectAtIndex:0]; // [2.13]
  NSLog(@"The element at index 0 in the array is: %@", element);
  [pool release];
  return 0;
}
```

这次的运行结果如下：

```
The element at index 0 in the array is: first string
```

你将会经常需要查看数组中的每一个元素以便进行其它操作。为此你可以参照下面例子，使用循环语句按照索引顺序显示数组中的每个元素：

[3]

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  NSMutableArray *myArray = [NSMutableArray array];
  [myArray addObject:@"first string"];
  [myArray addObject:@"second string"];
  [myArray addObject:@"third string"];
  int i;
  int count;
  for (i = 0, count = [myArray count]; i < count; i = i + 1)
  {
    NSString *element = [myArray objectAtIndex:i];
    NSLog(@"The element at index %d in the array is: %@", i, element);
  }
  [pool release];
  return 0;
}
```

运行结果如下：

```
The element at index 0 in the array is: first string
The element at index 1 in the array is: second string
The element at index 2 in the array is: third string
```

需要注意的是数组不仅仅可以用于字符串操作。它可以用于任何你希望用数组操作的对象。

类NSArray和类NSMutableArray提供了许多其它方法，希望你能够通过查看说明文件进一步学习与数组操作相关的知识。我们把置换数组内元素这个问题作为本章的结束。置换数组内元素要使用到方法“replaceObjectAtIndex:… withObject:…”。到目前为止，我们见到的方法最多只有一个参数，而这个方法不同，这就是为什么你在这里看到它，它带有两个参数。其实你可以看出来的，因为这个方法带有两个冒号。在Objective-C语言的方法可以使用任意多参数。下面是这个方法的语法：

```
[4]
[myArray replaceObjectAtIndex:1 withObject:@"Hello"];
```

运行这个方法后，索引1中的对象已经变成了字符串@”Hello”。当然，这个方法只能援引那些有效的索引。换句话说，原来索引中应当有对象，我们才可以用这个方法置换进新的对象。你可能发现了，Objective-C语言的方法的名字好似句子填空。当你要使用一个方法，只要填上你需要的值构成一个有意义的“句子”即可。这种给方法命名的方法来源于Smalltalk（一种由Xerox公司开发的面向对象的系统——译者按），也是Objective-C语言一大特色，它可以大大增强代码的描述性。当创建自己的方法时，你也应当尽量按照这种方法命名，不仅可以提高代码的可读性，而且使你的程序便于维护。

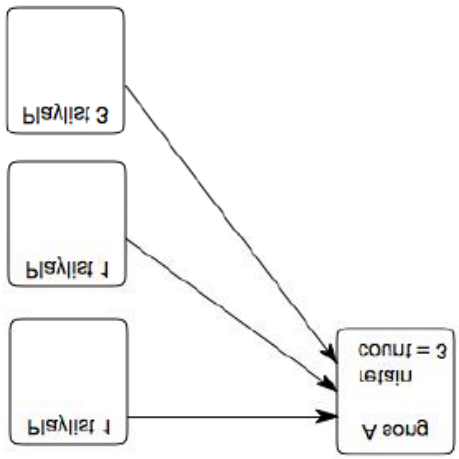
第 14 章 内存管理

前面很多章我都在向你致歉——因为例子中的一些语句行一直没有向你解释其用途。这些语句行就是用来处理内存的。你的程序并非你的Mac微机上唯一运行的程序，而且RAM又是珍贵的资源。当你的程序不再需要占用内存的时候，你需要把内存还给系统。当母亲教导你要对社区内的邻里有礼貌并和他们和谐相处的时候，她也在教给你如何编程。即使Mac微机上只运行了你的一个程序，拥挤的内存也会把你的程序挤到角落，Mac微机的响应速度也会变得很慢。

当你的程序创建出一个对象，对象会占用内存，你要在对象不被使用后释放出内存空间。也就是说当对象不再需要时你要毁掉它。尽管并不是那么好确定一个对象是否不再被使用了。比如在程序执行过程中你的对象会被多个其它对象援引，当被其它对象援引的可能性还存在时你就不能毁掉这个对象（如果你这样做可能导致程序崩溃或者出现不可遇见的结果）。

为了帮助你处理那些不在被使用的对象，Cocoa给每个对象关联了一个计数器，被称做“保留计数器”。编程时，给对象增加一条援引信息，就要让对象在它的计数器里加一；当减少一次援引，则减一。当保留计数器的计数为0的时候，对象就知道自己已经不再被援引了，可以被安全的毁掉了。这时候的对象会毁掉自己并释放出内存空间。

假设你的程序是一个音乐播放器，歌曲和播放列表代表对象。如果一首个被三个列表对象援引，你的歌曲对象的保留计数器的计数值是3。



有了保留计数器，对象知道自己将被援引多少词

要使对象的保留计数器增加数值，要做的就是发送一条“保留信息”（retain message）；减少计数器数值则要发送“释放信息”（release message）。Cocoa还提供了一个被称做“autorelease pool”的机制，可以发送延迟的释放信息——不是立即，而是迟一些。使用这个机制发送自动释放信息你要把对象注册到 autorelease pool之中。关于“autorelease pool”的语句行前面的例子中已经见过了，

那些语句是我们提供给系统以使系统建立起正确的“autorelease pool”机制。

本章介绍的Cocoa的内存管理技术就是通常所说的“援引计数”（reference counting）。你会在更高层次的资料中学习到全面的Cocoa的内存管理技术（参见第15章）。一些苹果工程师透露，苹果公司正在为Cocoa开发新的内存管理模式，被称做“无价值资料自动收集”（automatic garbage collection）。这种模式将会比现有技术更加有效，更易用并且更少出错。到本书编写时为止还苹果公司没有消息这种新技术何时可以面世。

第 15 章 信息资源

本书的目标是向读者介绍基于Objective-C语言的Xcode开发环境。如果你已经通读了两次以上本书，并且亲自尝试了书中的例子，那么你已经可以尝试创建自己的程序了。本书给了读者足够的知识去迅速的解决问题。本章内容将和你一起探寻更多的信息资源，下面的内容一定能引起你的兴趣。

在你着手编程前，给你一个建议：不要立即动手！先查检一下架构文件的内容，因为苹果可能已经帮你作好了一些事情，或者提供了现成的类来减轻你的负担；也许其他人已经写好了你要的东西，并把它变成了何以共享的源代码。所以，看说明文件和上网搜索可以节约你的宝贵时间。第一个应该访问的是苹果开发者主页：`http://www.apple.com/developer`。

我还强烈建议你在收藏夹中加入下列站点：

```
http://osx.hyperjeff.net/reference/CocoaArticles.php
http://www.cocoadev.com
http://www.cocoabuilder.com
http://www.stepwise.com
```

上述网站还提供大量的链接指向其他更多的资源。你还应该订阅“cocoa-dev”邮件列表（网址`http://lists.apple.com/mailman/listinfo/cocoa-dev`）。在这里你可以进行提问，热心的网友们总会尽己所能帮助你解决问题。当然，你要注意礼貌，而且应当首先自己试着解决（参考`http://www.cocoabuilder.com`）。`http://www.catb.org/~esr/faqs/smart-questions.html`页面上的“How To Ask Questions The Smart Way”则是告诉你如何恰当的提问。

还有一些优秀的关于Cocoa开发的书籍。Stephen Kochan编写的《Programming in Objective-C》是为初学者准备的。其它一些书则要求你具备本书所将到的基础知识。我们比较推崇Aaron Hillegass编写的《Cocoa Programming for Mac OS X》，他本人则是在Big Nerd Ranch教授这方面的课程。我们同样向您推荐James Duncan Davidson和苹果公司合作编写的，由O'Reilly出版的《Cocoa with Objective-C》一书。

最后还有一点。为Mac微机编程，不仅要保证代码中没有错误，还要使程序符合“苹果人机界面指南”（The Apple human interface guidelines）的要求。（可以在一下网页找到相关内容`http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html`）。写好的程序不要不好意思发表，别人的意见会帮助你进步。

我们衷心的希望你喜欢本书，并且可以热爱上Xcode编程。噢，要说的是别忘了第0章提及的内容。

作者：*Bert, Alex, Philippe*

译者后记

经过近二十个夜晚的“战斗”，基本完成了《Become an Xcoder》这本手册的中文翻译。说真的，当真正着手工作，我便后悔了，因为事情并不像想像中那么简单，虽然这本书仅有六十余页。对于我，一个学习文科的学生，难的不是整篇的英文单词，而是这些单词组成在一起形成的含义。

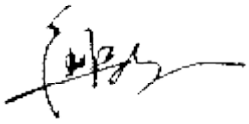
我自认为计算机水平不低，不论Windows还是Mac OS都玩得转，但编程毕竟比玩系统高深得多；我自以为有一定的编程基础，但从小到现在，从GW-Basic到Q-Basic再到VB，学了多年却连Basic族都弄不通；我自诩为英文高才生，但我的英国英语大脑面对美国口语表现好比配了68020芯片的古老Apple。——一切本以为的优势成了理所应当的劣势。

但是既然开始了，我想也只有硬着头皮上了，权当打发漫长无聊的假期吧。

在C编程方面，要感谢死党杜志佳积极而富有成效的指导，这位南京著名高校的高才生虽然至今没能教会我使用C语言，但是他在编程技术上对我的热心帮助却是本书得以早日翻译完成的保障。

还要感谢老弟乐天的支持，他本来答应帮我联系在洛阳攻读英语专业的老姐来做校对工作。因为种种原因，我不得不放弃这个本能令这部中文书稿蓬荜生辉的机会，实在遗憾，却依然要感谢。

最后，依然是那句话，但我说的十分真诚：鉴于水平之限，纰漏菲薄再所难免，还望不吝赐教，阔刀斧正。



二零零六年八月于河北唐山寓所