

1, 前言

相信 iPhone 不久就要在国内发布了,和我们在国内可以通过正规渠道买得到的 iPod Touch 一样, iPhone 也是一个激动人心的产品。iPhone 发布的同时,基于 iPhone 的程序也像雨后春笋一样在 iTunes 里面冒出来。

你将来也许会考虑买一个 iPhone,体验一下苹果的富有创意的种种应用;你也许会考虑向 iTunes 的社区的全世界的人们展示一下你非凡的创意,当然也可以通过你的创意得到一些意想不到的收益。

OK,你也许迫不及待的准备开发了。但是先等一下,让我们回忆一下最初的电影是怎么拍摄的。这个很重要,因为和 iPhone 的开发比较类似。

在最初因为器材比较原始,所以拍摄电影需要很高的技术,那个时候的电影的导演基本上是可以熟练操作摄影器材的人。随着器材的完善,使用也简单起来。于是器材的使用不是决定一个电影的质量的唯一的因素,取而代之的是故事或者说电影的创意。

iPhone 的开发也是这样。当然从入门到掌握的过程来说任何事情都是开始比较难,随着掌握程度的加深,你将会觉得开发 iPhone 应用程序是一件简单而且轻松的事情,到了那个时候,你的主要的制胜武器就不是开发技术,而是你的创意了。对于你来说,我在这里写的东西都是有关“摄影器材”也就是介绍如何使用 iPhone 的平台来开发应用程序。

iPhone 的开发语言是 Objective-C。Objective-C 是进行 iPhone 开发的主要语言,掌握了 Objective-C 的基本语法以及数据结构之后,你需要熟悉一下 iPhone 的 SDK。笔者很难做到在一篇文章里面把所有的东西都介绍清楚,所以笔者打算分成两个主题,一个是 Objective-C,一个是 iPhone 开发。

本系列将侧重于 Objective-C。当然,任何一种开发语言都无法脱离于运行环境,Objective-C 也不例外。所以在本系列当中也会穿插的介绍一些 SDK 里面的一些特性,主要是数据结构方面,比如说 NSString, NSArray 等等。看到 NSString, NSArray 这些名词,你也许会感到有些茫然,不过没有关系,随着本系列的深入介绍,你会发现你非常喜欢这些东西。

1.1,谁会考虑阅读本系列

如果你对 iPhone 感兴趣，如果你考虑向全世界的人们展示你的创意，如果你有一颗好奇心，如果你打算通过开发 iPhone 程序谋生，如果你觉得苹果比 Windows 酷，如果你认为不懂苹果的话那么就有些不时尚的话，那么可以考虑阅读本系列。

老手也可以考虑花一点时间阅读一下，可以发帖子和笔者交流切磋。笔者发布的文章属于公益写作，旨在为大家介绍 iPhone 开发的一些基础知识，如果可以提供宝贵意见，笔者将不胜感激。

1.2,需要准备的东西

工欲善其事，必先利其器。 《论语·魏灵公》

第一，你需要一台苹果电脑。当然这个不是必需的条件，如果你可以在你的 Intel PC 上成功安装 MAC OS 的话，那么请忽略这一条。

第二，你需要去苹果网站上下载开发工具 XCODE。注意，XCODE 是完全免费的，但是需要你去注册一个账号才可以下载。由于 XCODE 不时的在更新，所以如果你的 MAC OS 不支持你下载的 XCODE 的话，那么你也也许需要考虑买一个最新的 MAC OS。

第三，你需要至少有 C,C++,或者 JAVA 的背景知识。不过如果你没有，那么也不用担心，相信阅读了笔者的文章之后应该也可以掌握。

最后需要的东西就不是必须的了，当然有的话会更好一些。这些东西是，开发者账户（需要付费），iPhone 手机（在部分国家可以免费获得，但是中国会怎么样，笔者不是很清楚），iPod Touch（需要购买）。

1.3 ,关于笔者的写作

笔者利用业余时间进行写作，所以无法对文章发布的时间表做出任何保证，还请各位读者谅解。但是笔者会尽最大努力在短时间之内完成写作。

由于笔者经验才识所限，在本教程当中难免会遇到遗漏，错误甚至荒谬的地方，所以还请同学们批评指正。

对于已经完成的章节，基于一些条件的改变或者勘误，或者大家提出的意见，笔者也会考虑做出适当的修改。

在每一个章节都会有代码的范例，笔者注重阐述基本概念所以代码难免会有不完整或者错误的地方，同学们可以任意的在自己的代码中使用笔者所写的代码，但是笔者不承担由于代码错误给同学们带来的损失。同学们在阅读本教程的时候，可以直接下载范例代码运行，但是为了熟悉编码的环境以及代码的规范，笔者强烈建议同学们按照教程自己亲自输入代码。

Objective-C 的概念比较多，而且很多概念都相互交叉。比如说讲解概念 A 的时候，需要概念 B 的知识，讲解概念 B 的时候需要概念 C 的知识，讲解概念 C 的时候需要概念 A。这样就给本教程的写作带来了一定的麻烦，很明显笔者无法在某一个章节里面把所有的概念都讲述清楚，所以每一章都有侧重点，大家在阅读的时候需要抓住每一章的侧重点，忽略一些和本章内容无关的新的概念和知识。

1.4,本系列的结构

第 1 章，也就是本章

第 2 章，从 [Hello,World!](#) 开始

第 3 章，类的声明和定义

第 4 章，继承

第 5 章，Class 类型，选择器 [Selector](#) 以及函数指针

第 6 章，[NSObject](#) 的奥秘

第 7 章，对象的初始化以及实例变量的作用域

第 8 章，类方法以及私有方法

第 9 章，内存管理

第 10 章，到目前为止出现的内存泄漏事件

第 11 章，字符串，数组以及字典

第 12 章，属性

第 13 章，类目(Categories)

第 14 章，协议(Protocols)

第 15 章，Delegate

第 16 章，线程

第 17 章，文件系统

第 18 章，数据序列化以及保存用户数据

第 19 章，网络编程

第 20 章，XML 解析

2,从 Hello,World! 开始

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，[请点击这里](#)。

现在笔者假设大家已经有了开发的环境。好了，我们开始构筑我们的第一个程序。

在开始第一个程序之前，笔者需要提醒大家一下，如果手里面有开发环境的话并且是第一次亲密接触 Xcode 的话，为了可以熟悉开发环境，强烈建议按照笔者的步骤一步一步的操作下去。尽管如此，笔者还是为大家准备了已经做好的代码，[点击这里](#)下载。

2.1,构筑 Hello, World

第一步，启动 Xcode。初次启动的时候，也许会弹出一个“Welcome to Xcode”的一个对话框。这个对话框和我们的主题没有关系，我们可以把它关掉。

第二步，选择屏幕上部菜单的“File->New Project”，出现了一个让你选择项目种类的对话框。你需要在对话框的左边选择“Command Line Utility”，然后在右边选择“Foundation Tool”，然后选择“Choose...”按钮。如图 2.1 所示。

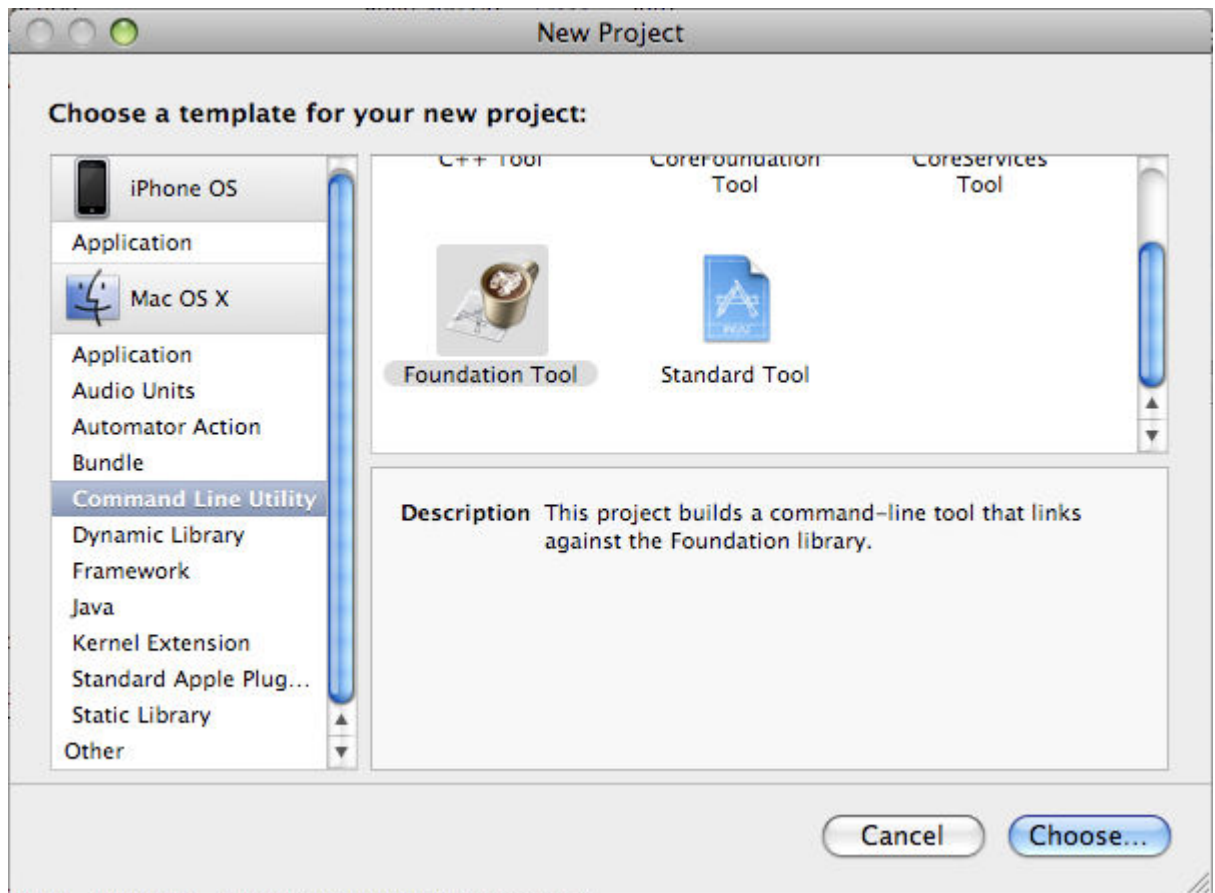


图 2-1，新建项目

注意也许有人会问，你不是要讲解 iPhone 的开发，那么为什么不选择“iPhone OS”下面的“Application”呢？

是这样的，在这个系列当中，笔者主要侧重于 Objective-C 的语法的讲解，为了使得讲解简单易懂，清除掉所有和要讲解的内容无关的东西，所以笔者在这里只是使用最简单的命令行。

第三步，Xcode 会提问你项目的名字，在“Save As”里面输入“02-Hello World”，然后选择“Save”。如图 2-2 所示

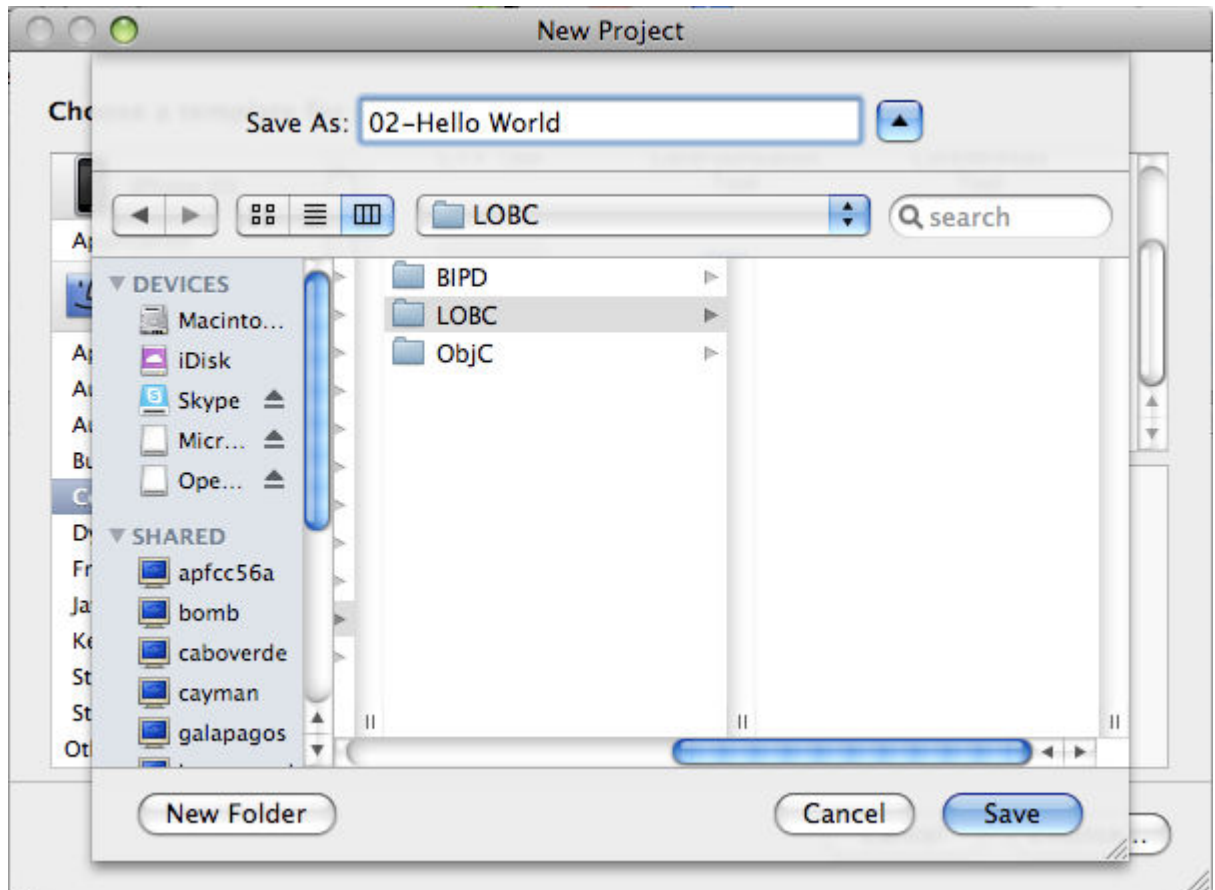


图 2-2，输入项目的名字

第四步，得到一个如图 2-3 所示的一个画面。尝试一下用鼠标分别点击左侧窗口栏里面的“02-Hello World”，“Source”，“Documentation”，“External Frameworks and Libraries”，“Products”，然后观察一下右边的窗口都出现了什么东西。一般来说，“02-Hello World”就是项目的名字下面是项目所有的文件的列表。项目下面的子目录分别是和这个项目相关的一些虚拟或者实际上的目录。为什么我说是虚拟的呢？大家可以通过 Finder 打开你的工程文件的目录，你会发现你的所有文件居然都在根目录下，根本就不存在什么“Source”之类的目录。

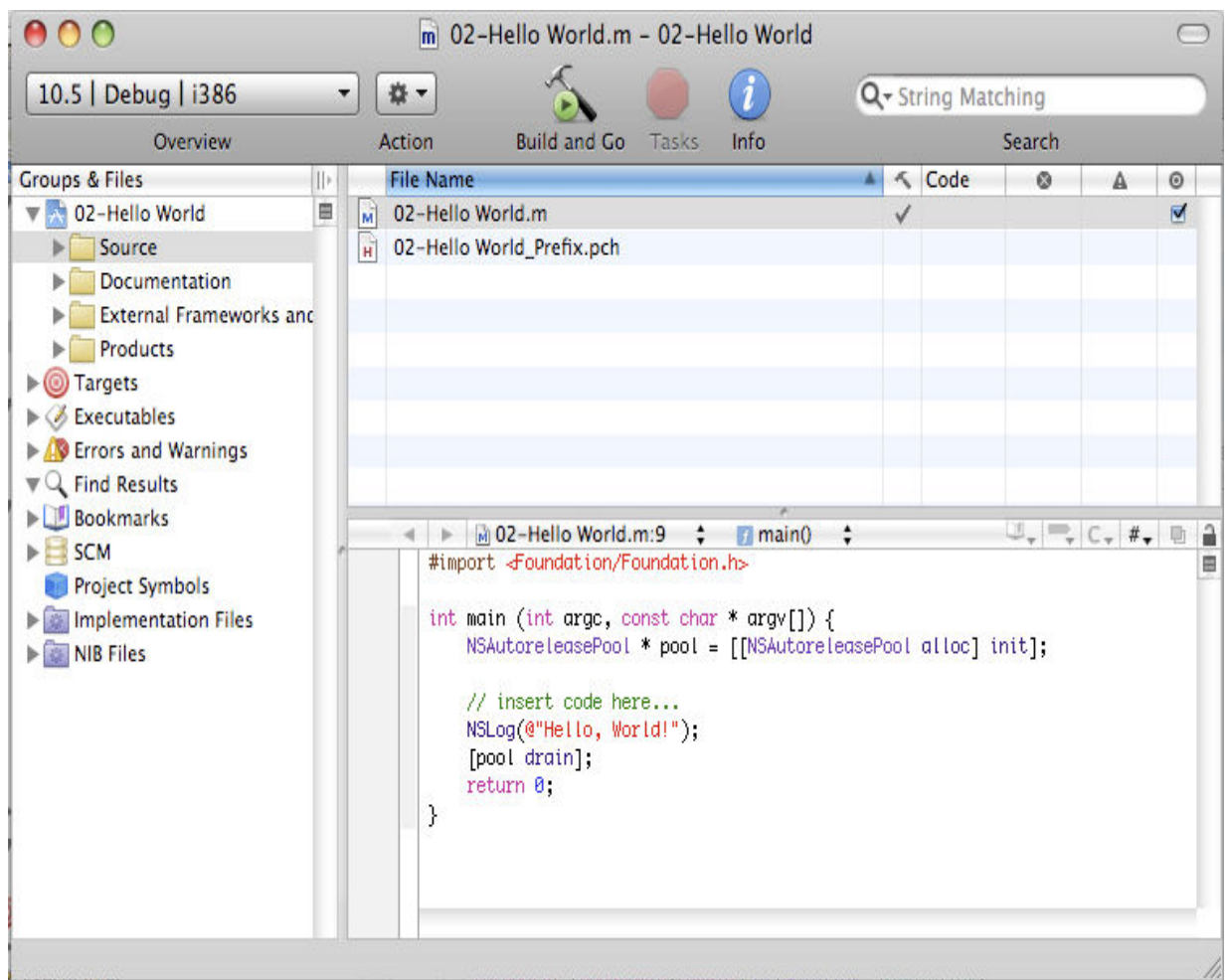


图 2-3，项目浏览窗口

第五步，选择屏幕上方菜单的“Run”然后选择“Console”，出现了如图 2-4 所示的画面，用鼠标点击窗口中间的“Build and Go”按钮。

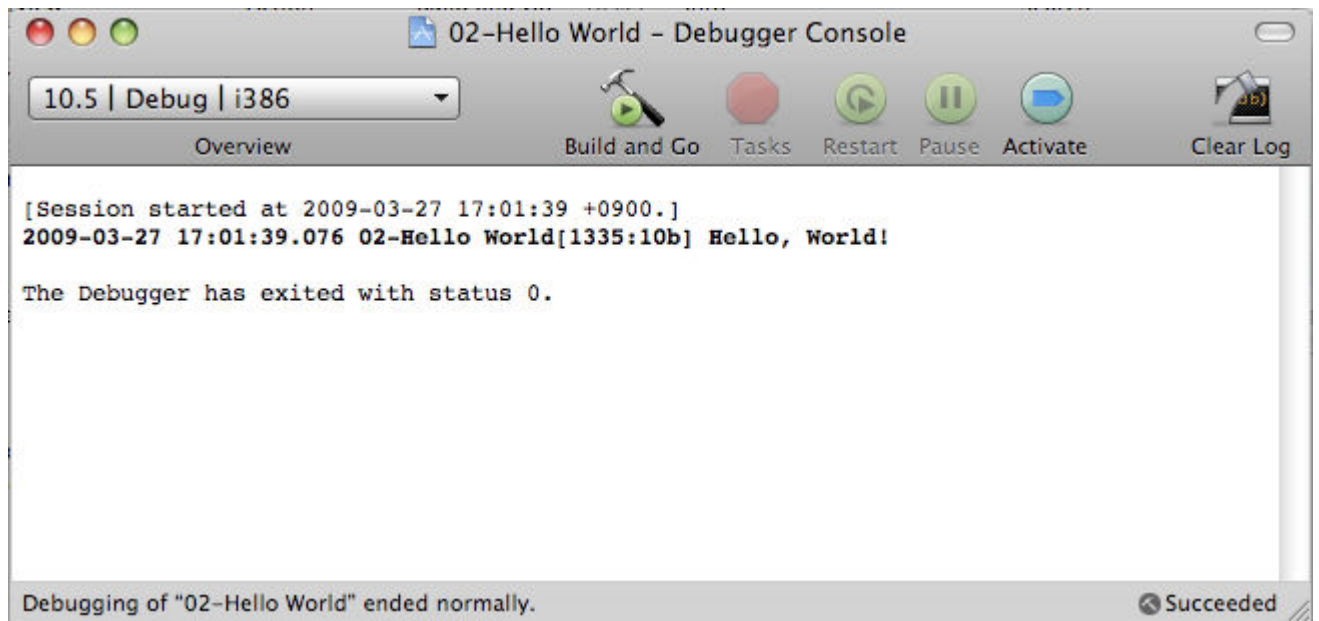


图 2-4，运行结果画面

如果不出什么意外的话，大家应该看到我们熟悉得不能再熟悉的“Hello Wolrd!”。由于我们没有写任何的代码，所以从理论上来说，这部分代码不应该出现编译错误。好的，从下面开始，笔者要开始对这个 Hello World 里面的一些新鲜的东西进行讲解。

2.2，头文件导入

在 Java 或者 C/C++ 里面，当我们的程序需要引用外部的类或者方法的时候，需要在程序源文件中包含外部的类以及方法的包（java 里面的 jar package）或者头文件（C/C++ 的 .h），在 Objective-C 里面也有相类似的机制。笔者在这一节里面将要向大家介绍在 Objective-C 里面，头文件是怎样被包含进来的。

请同学们到 Xcode 开发环境的左侧窗口里面，点击 Source 文件夹，然后就在右侧部分看到了代码源文件的列表，找到 02-Hello World.m 之后单击会在 Xcode 的窗口里面出现，双击鼠标代码会在一个新窗口出现，请同学们按照这种方法打开“02-Hello World.m”。

对于 Java 程序来说，源程序的后缀为 .java，对于 C/C++ 代码来说，后缀为 c/cpp，现在我们遇到了 .m。当 Xcode 看到了 .m 文件之后，就会把这个文件当作 Objective-C 文件来编译。同学们也许会猜到，当 Xcode 遇到 c/cpp，或者 java 的时候也会对应到相应的语言的。

好的，我们顺便提了一下 Xcode 对 .m 文件的约定，现在我们开始从第一行代码讲起，请参看下列代码：


```

1 #import <Foundation/Foundation.h>
2
3 int main (int argc, const char * argv[]) {
4     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
5
6     // insert code here ...
7
8     NSLog(@"Hello, World!");
9
10    [pool drain];
11    return 0;
12 }

```



有过 C/C++ 经验的同学看到第一行，也许会觉得有些亲切；有过 Java 经验的同学看到第一行也许也会有一种似曾相识的感觉。同学们也许猜到了这是干什么用的，没错，这个正是头文件。不过，在 C/C++ 里面是 `#include`，在 java 里面是 `import`，这里是 `#import`。

在 C/C++ 里面会有 `#include` 互相包含的问题，这个时候需要 `#ifdef` 来进行编译的导向，在 Xcode 里面，同学们可以“放心的”包含各种东西，这个没有关系，因为我们的编译器有足够的“聪明”，因为同一个头文件只是被导入一次。除了 `#import` 变得聪明了一点之外，和 `#include` 的功能是完全一样的。

我们再来看看我们的另外一个新的朋友---`Foundation.h`。这个是系统框架 **Foundation framework** 的头文件，有了它你可以免费的获取系统或者说苹果公司为你精心准备的一系列方便你使用的系统功能，比如说字符串操作等等。`Foundation` 框架从属于 **Cocoa** 框架集，**Cocoa** 的另外一个框架为 **Application Kit**，或者是 **UIKit**，其中前者的应用对象为 **MAC OS**，后者的应用对象为 **iPhone OS**。本系列入门指南将只是使用 **Foundation**，因为笔者需要向同学们介绍 **Objective-C** 的基本使用方法，为了避免过多的新鲜东西给同学们造成阅读上的困难，所以命令行就已经足够了。

说到这里，笔者需要澄清一点，其实 **MAC OS** 的 **Cocoa** 和 **iPhone** 的 **Cocoa** 是不一样的，可以说，其中 **iPhone** 是 **MAC OS** 的一个子集。

2.3, main 函数

有过 C/C++ 或者 java 经验的同学们对第 3 行代码应该很熟悉了，是的大家都一样主程序的入口都是 main。这个 main 和 C/C++ 语言里面的 main 是完全一样的，和 java 语言在本质上也是完全一样的。因为 Objective-C 完全的继承了 C 语言的特性。确切的说，不是说 Objective-C 和 C 语言很相似，而是 Objective-C 和 C 语言是完全兼容的。

关于 main 函数是干什么用的，笔者就不在这里罗嗦了，有兴趣的同学可以找一本 C 语言的书看看。

2.4，关于 NSAutoreleasePool

自己动手，丰衣足食---

在第 4 行，我们遇到了另外一个新鲜的东西，这就是 NSAutoreleasePool。

让我把这个单词分为三部分，NS，Autorelease 和 Pool。

当我们看到 NS 的时候，也许不知道是代表着什么东西。NS 其实只是一个前缀，为了避免命名上的冲突。NS 来自于 NeXTStep 的一个软件，NeXT Software 的缩写，NeXT Software 是 Cocoa 的前身，一开始使用的是 NS，为了保持兼容性所以 NS 一直得以保留。在多人开发的时候，为了避免命名上的冲突，开发组的成员最好事先定义好各自的前缀。但是，最好不要有同学使用 NS 前缀，这样会让其他人产生误解。

略微有些遗憾的是，Objective-C 不支持 namespace 关键字，不知道后续的版本是否会支持。

下面我们讨论一下 Autorelease 和 Pool。

程序在执行的时候，需要向系统申请内存空间的，当内存空间不再被使用的时候，毫无疑问内存需要被释放，否则有限的内存空间会很快被占用光光，后面的程序将无法得到执行的有效内存空间。从计算机技术诞生以来，无数的程序员，我们的无数先辈都在为管理内存进行努力的工作，发展到现在，管理内存的工作已经得到了非常大的完善。

在 Objective-C 或者说 Cocoa 里面，有三种内存的管理方式。

第一种，叫做“Garbage Collection”。这种方式 and java 类似，在你的程序的执行过程中，始终有一个高人在背后准确地帮你收拾垃圾，你不用考虑它什么时候开始工作，怎样工作。你只需要明白，我申请了一段内存空间，当我不再使用从而这段内存成为垃圾的时候，我就彻底的把它忘记掉，反正那个高人会帮我收拾垃圾。遗憾的是，那个高人需要消耗一定的资源，在携带设备里面，资源是紧俏商品所以 iPhone 不支持这个功能。所以“Garbage Collection”不是本入门指南的范围，对“Garbage Collection”内部机制感兴趣的同学可以参考一些其他的资料，不过说老实话“Garbage Collection”不大合适初学者研究。

第二种，叫做“**Reference Counted**”。就是说，从一段内存被申请之后，就存在一个变量用于保存这段内存被使用的次数，我们暂时把它称为计数器，当计数器变为 0 的时候，那么就是释放这段内存的时候。比如说，当在程序 A 里面一段内存被成功申请完成之后，那么这个计数器就从 0 变成 1（我们把这个过程叫做 **alloc**），然后程序 B 也需要使用这个内存，那么计数器就从 1 变成了 2（我们把这个过程叫做 **retain**）。紧接着程序 A 不再需要这段内存了，那么程序 A 就把这个计数器减 1（我们把这个过程叫做 **release**）；程序 B 也不再需要这段内存的时候，那么也把计数器减 1（这个过程还是 **release**）。当系统(也就是 **Foundation**)发现这个计数器变成了 0，那么就会调用内存回收程序把这段内存回收（我们把这个过程叫做 **dealloc**）。顺便提一句，如果没有 **Foundation**，那么维护计数器，释放内存等工作需要你手工来完成。

这样做，有一个明显的好处就是，当我们不知道是 A 先不使用这段内存，还是 B 先不使用这段内存的时候，我们也可以非常简单的控制内存。否则，当我们在程序 A 里面释放内存的时候，还需要看看程序 B 是否还在使用这段内存，否则我们在程序 A 里面释放了内存之后，可怜的程序 B 将无法使用这段内存了。这种方式，尤其是在多线程的程序里面很重要，如果多个线程同时使用某一段内存的时候，安全的控制这些内存成为很多天才的程序员的梦魇。

如果有同学搞过 COM 的话，那么应该对 **Release/AddRef** 很熟悉了，其实 **Objective-C** 和他们的机制是一样的。

接下来，我需要解释一下 **Autorelease** 方式。上述的 **alloc->retain->release->dealloc** 过程看起来比较令人满意，但是有的时候不是很方便，我们代码看起来会比较罗嗦，这个时候就需要 **Autorelease**。**Autorelease** 的意思是，不是立即把计数器减 1 而是把这个过程放在线程里面加以维护。当线程开始的时候，需要通知线程（**NSAutoreleasePool**），线程结束之后，才把这段内存释放（**drain**）。Cocoa 把这个维护所有申请的内存的计数器的集合叫做 **pool**，当不再需要 **pool**(水池)的时候就要 **drain**（放水）。

笔者想要说的是，虽然 iPhone 支持 **Autorelease** 但是我们最好不要使用。因为 **Autorelease** 方式从本质上来说是一种延迟释放内存的机制，手机的空间容量有限，我们必须节约内存，确定不需要的内存应该赶快释放掉，否则当你的程序使用很多内存的情况下也许会发生溢出。这一个习惯最好从刚刚开始学习使用 **Objective-C** 的时候就养成，否则长时间使用 **Autorelease** 会让你变得“懒散”，万一遇到问题的时候，解决起来会非常耗费时间的。所以，还是关于内存管理，我们还是自己动手，丰衣足食。当然笔者不是说绝对不可以使用，而是当使用 **Autorelease** 可以明显减低程序复杂度和易读性的时候，还是考虑使用一下换一下口味。

说到这里，可能有的同学已经开始发晕了，认为这个东西比较难以理解。是的，笔者在这里只是介绍一个大概的东西，在这里只要了解计数器的概念就可以了，笔者将在随后的章节里面对这个功能加以详细论述，请同学们放心，这个东西和 **Hello World** 一样简单。

关于 Pool

在使用 **Pool** 的时候，也要记住系统给你的 **Pool** 的容量不是无限大的，从这一点来说和在现实世界的信用卡比较相似。

你可以在一定程度透支，但是如果“忘记掉”信用卡的额度的话，会造成很大的系统风险。

第三种，就是传统而又原始的 C 语言的方式，笔者就不在这里叙述了。除非你在 **Objective-C** 里面使用 C 代码，否则不要使用 C 的方式来申请和释放内存，这样会增加程序的复杂度。

线程是什么东西？线程指的是进程中一个单一顺序的控制流。它是系统独立调度和分派的基本单位。同一进程中的多个线程将共享该进程中的全部系统资源，比如文件描述符和信号处理等等。一个进程可以有很多线程，每个线程并行执行不同的任务。

2.5，关于[[NSAutoreleasePool alloc] init];

关于程序第 4 行等号右边出现的括弧以及括弧里面的内容，笔者将在后续的章节里面介绍。在这里，同学们可以理解为，通过告诉 **Objective-C** 编译器[[NSAutoreleasePool alloc] init]，编译器就会成功的编译生成 **NSAutoreleasePool** 对象的代码就可以了。

2.6，Objective-C 里面的注释

同学们在第 6 行看到了//的注释，这个和 C++ 以及 Java 是一样的，表示这一行的内容是注释，编译器将会忽略这一行的内容。笔者在上面说过 **Objective-C** 完全兼容 C 语言，所以 C 语言里面传统的 **/**/** 在 **Objective-C** 里面也是有效的。

2.7，命令行输出

第 7 行，我们看到了 **NSLog** 这个函数。**NS** 上面已经讲过了，我们都知道 **Log** 是什么意思，那么这段代码的意思就是输出一个字符串，Xcode 的代码生成器自己把字符串定义为“**Hello, World!**”。**NSLog** 相当于 C 语言里面的 **printf**，由于我们是在使用 **Objective-C** 所以笔者将会和同学们一起，在这里暂时忘记掉我们过去曾经熟悉的 **printf**。

有眼光锐利的同学会发现在字符串的前面多了一个@符号，这是一个什么东西呢？

如前所述，Objective-C 和 C 是完全兼容的，但是 NSLog 这个函数需要的参数是 NSString，这样就产生了一个问题，如果使用 C 的字符串方式的话，为了保持和 C 的兼容性编译器将会把字符串理解为 C 的字符串。为了和 C 的字符串划清界限，在 C 的字符串前面加上 @ 符号，Objective-C 的编译器会认为这是一个 NSString，是一个 NSLog 喜欢的参数。

为什么 NSLog 或者 Cocoa 喜欢使用 NSString？因为 NSString 封装了一系列的字符串的方法比如字符串比较，字符串和数字相互转换等等的方法，使用起来要比 C 的字符串方便的多。

2.8, 本章总结

非常感谢同学们耐心的看到这里！

通过理解本章的内容，同学们应该可以使用 Xcode 创建一个命令行的工程，理解 .m 文件的基本要素，理解内存的管理方法的思路，还有 Objective-C 的注释的写法，以及命令行的输出方法。

是不是很简单又很有趣呢？笔者将会尽最大努力把看起来复杂的东西讲解的简单一些，并且真心的希望大家可以从中找到乐趣。

Objective-C 2.0 with Cocoa Foundation --- 3,类的声明和定义

3,类的声明和定义

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，[请点击这里](#)。

上一章我们写了一个非常简单的 Objective-C 下面的 Hello, World! 的小程序，并且对里面出现的一些新的概念进行了解释。这一章，我们将要深入到 Objective-C 的一个基本的要素，也就是类的声明和定义。通过本章的学习，同学们应该可以定义类，给类加上变量，还有通过方法访问类的变量。不过准确的说，变量和方法的名词在 Objective-C 里面并不是最准确的称呼，我们暂时引用 Java 的定义，稍后我们将统一我们的用语定义。

3.1,本章的程序执行结果。

我们将构筑一个类，类的名字叫做 Cattle,也就是牛的意思，今年是牛年而且我还想给在股市奋战的同学们一个好的名字，所以我们暂时把这个类叫做牛类。

我们在 `main` 里面初始化这个牛类，然后调用这个类的方法设定类的变量，最后调用这个类的一个方法，在屏幕上输出，最终输出的结果如下图 3-1 所示

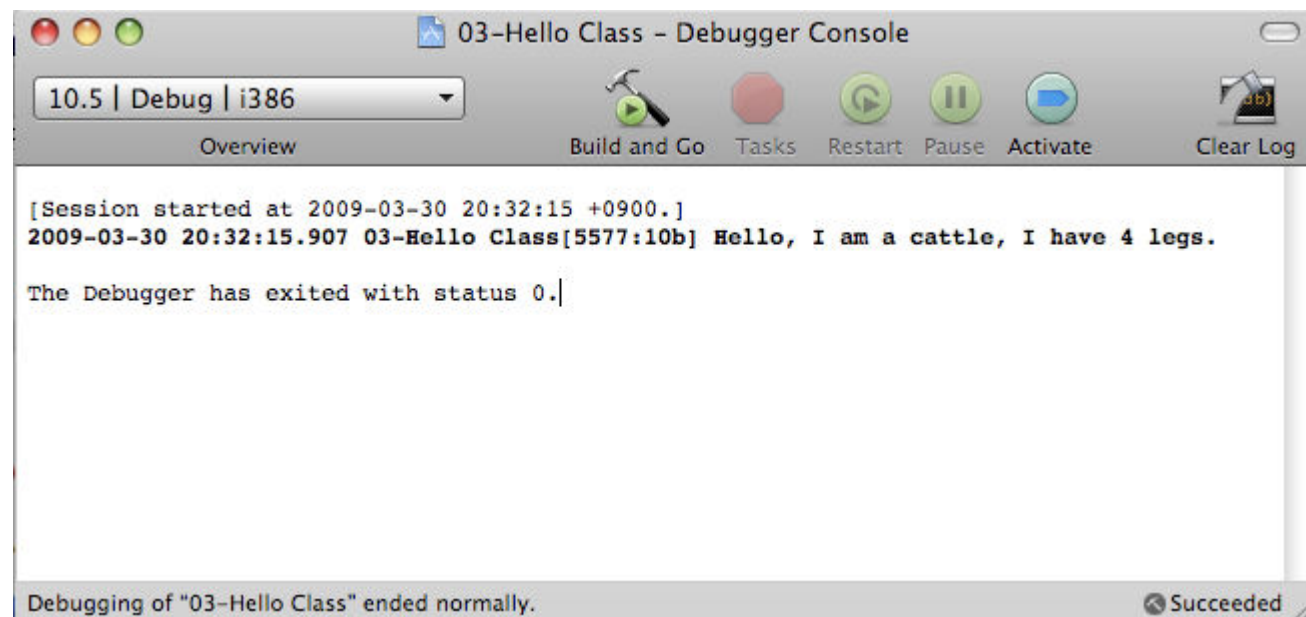


图 3-1，牛类的输出结果

完整的代码在这里。不过为了熟悉编辑环境以及代码，笔者强烈建议同学们按照下面的步骤自己输入。

3.2，实现步骤

第一步，按照我们在第二章所述的方法，新建一个项目，项目的名字叫做 `03-Hello Class`。当然，你也可以起一个别的更好听的名字，比如说 `Hello Cattle` 等等，这个并不妨碍我们的讲解。如果你是第一次看本系列文章，请到这里参看第二章的内容。

第二步，把鼠标移动到左侧的窗口的“`Source`”目录，然后单击鼠标右键，选择“`Add`”，然后界面上会出来一个子菜单，在子菜单里面选择“`New File...`”。如图 3-2 所示：

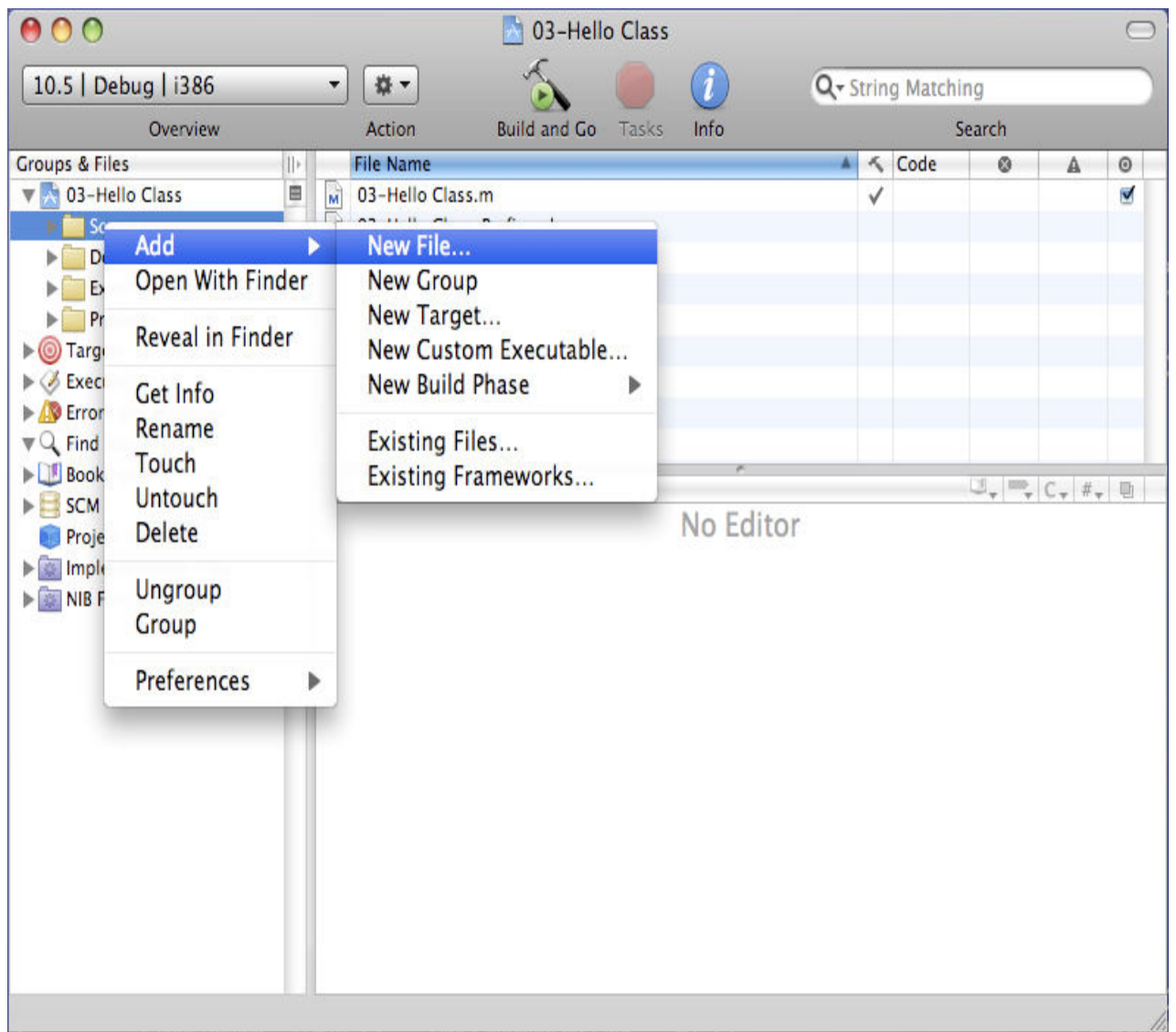
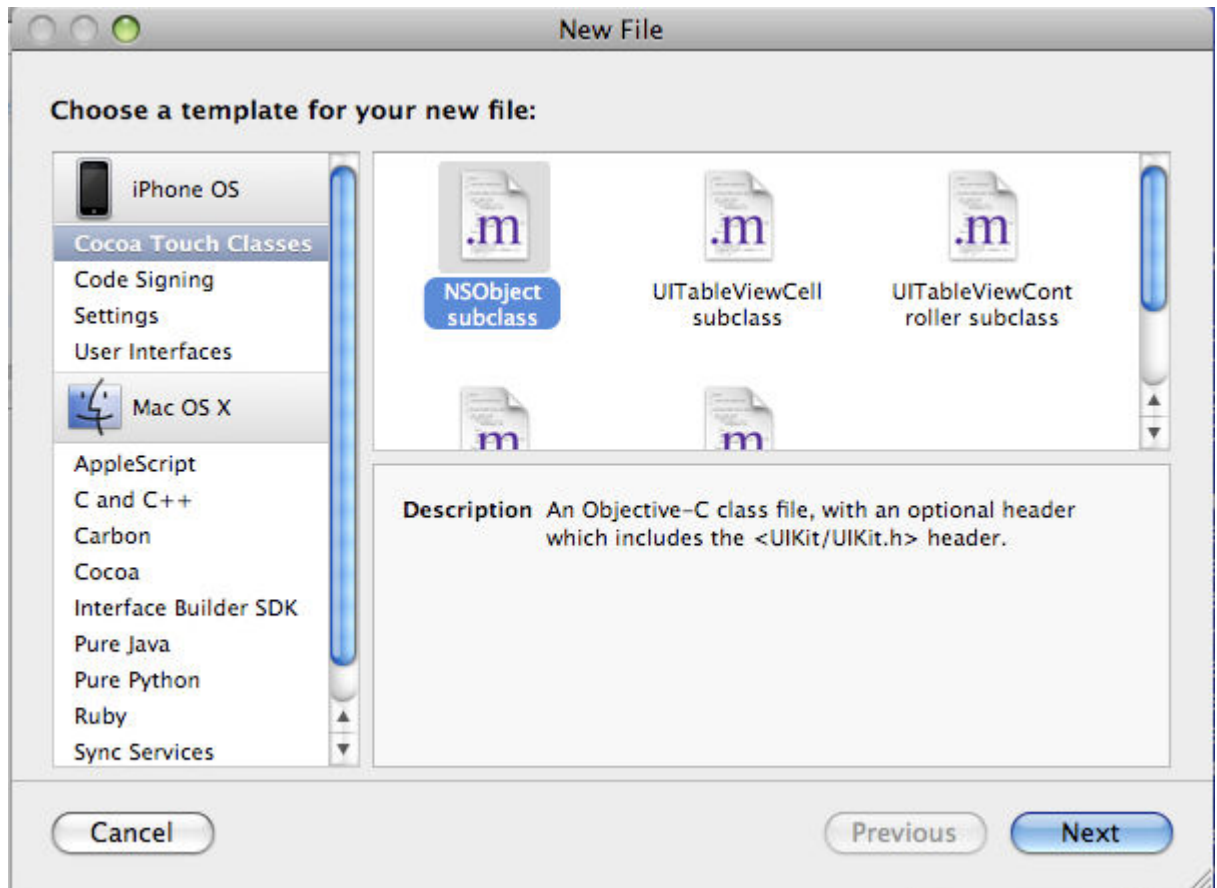
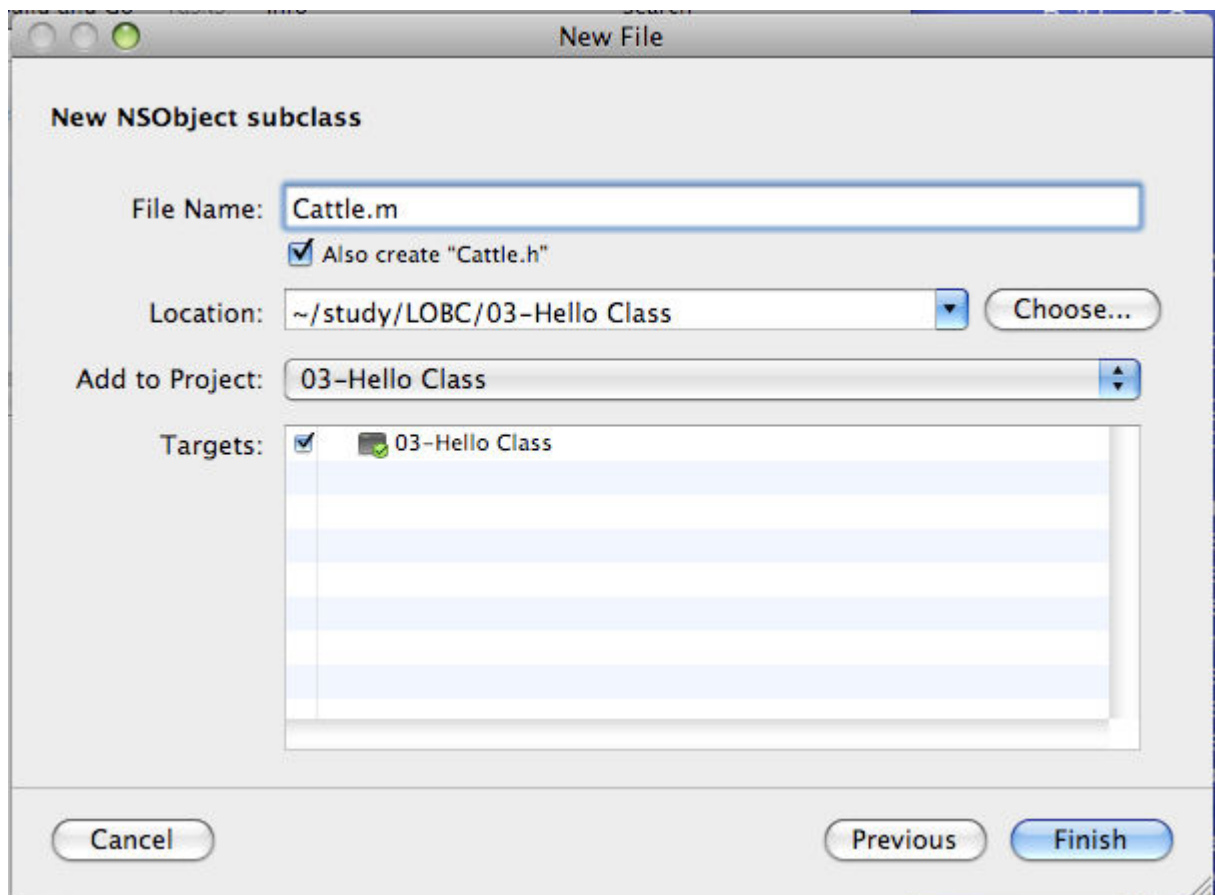


图 3-2，新建文件

第三步，在新建文件对话框的左侧选择“Cocoa Touch Classes”，然后在右侧窗口选择“NSObject subclass”，然后单击“Next”。如图 3-3 所示：



第四步，在“New File”对话框里面的“File Name”栏内输入“Cattle.m”。注意，在确省状态下，Xcode 为你加上了“.m”的后缀，这个也是编译器识别 Objective-C 源文件的方法，没有特殊理由请不要修改这个后缀，否则会让编译器感到不舒服。另外请确认文件名字输入栏的下方有一个“Also create “Cattel.h””选择框，请保持这个选择框为选择的状态。如图 3-4 所示。



第 5 步，在项目浏览器里面选择“Cattle.h”文件，把文件改为如下代码并且保存(Command 键+S)：

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

@interface Cattle : NSObject {
    int legsCount;
}

- (void) saySomething;
- (void) setLegsCount:(int) count;
@end
```

为什么 `legsCattle` 者，牛也；`legs` 者，股也。不过牛股里面的牛正确的英文说法应该是 **Bull**，请大家不要着急，我们会在类的继承里面命名一个 **Bull** 类的。

第六步，在项目浏览器里面选择“Cattle.m”文件，把文件改为如下代码并且保存(Command 键+S):

```
#import "Cattle.h"

@implementation Cattle

-(void) saySomething
{
    NSLog(@"Hello, I am a cattle, I have %d legs.", legsCount);
}

-(void) setLegsCount:(int) count
{
    legsCount = count;
}

@end
```

第七步，在项目浏览器里面选择“03-Hello Class.m” 文件，把文件改为如下代码并且保存 (Command 键+S):

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

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

    id cattle = [Cattle new];
    [cattle setLegsCount:4];
    [cattle saySomething];

    [pool drain];

    return 0;
}
```

第八步，选择屏幕上方菜单里面的“Run”，然后选择“Console”，打开了 Console 对话框之后，选择对话框上部中央的“Build and Go”，如果不出什么意外的话，那么应该出现入图 3-1 所示的结果。如果出现了什么意外导致错误的话，那么请仔细检查一下你的代码。如果经过仔细检查发现还是不能执行的话，可以到这里下载笔者为同学们准备的代码。如果笔者的代码还是不能执行的话，请告知笔者。

3.3，类的声明

从 Objective-C 名字我们就可以得知，这是一个面向对象的语言。面向对象的一个最基础的要素就是类的概念，Objective-C 也不例外。所谓的类的概念，其实是从 C 语言的结构体发展而来的。我们知道，C 语言里面的结构体仅仅有数据的概念，面向对象的语言不仅仅支持数据，还可以在结构体里面封装用于存取结构体数据的方法。结构体的数据和方法结合，我们把整个结构体称为类(Class)。仅仅有了类，是不能执行任何操作的，我们必须把类进行实体化，实体化后的类我们称之为对象(Object)。从这个角度上来说，我们可以认为类是对象的模版。

如果要使用类，那么和构造体相类似，我们必须声明这个类。

请参照“Cattle.h”文件：

```
1 #import <Foundation/Foundation.h>
2
3
4 @interface Cattle : NSObject {
5     int legsCount;
6 }
7 - (void) saySomething;
8 - (void) setLegsCount:(int) count;
9 @end
```

如果看过本系列第二章的同学们，第一行应该是一个老面孔了，我们知道我们需要这个东西免费获得苹果公司为我们精心准备的 Foundation Framework 里面的很多的功能。如果不使用这个东西的话，我们的工作将会很复杂。

同学们请看第 4 行和第 9 行的第一个字母，又出现了“@”符号。为什么说又呢，因为我们在第二章的字符串前面也看到过这个东西。字符串前面出现这个符号是因为我们需要和 C 语言的字符串定义区别开来，我们需要编译器导向。在这里，我要告诉同学们的是，这里的“@”符号的作

用还是同样是编译器导向。我们知道 Java 和 C++ 定义了一个关键字 `class` 用于声明一个类，在 Objective-C 里面，不存在这样的关键字。在 Objective-C 里面，类的定义从 `@interface` 开始到 `@end` 结束，也就是说，编译器看到了 `@interface` 就知道了这是类的定义的开始，看到了 `@end` 就知道，类的定义结束了。

我们这里类的名字是“Cattle”，我们使用了空格和 `@interface` 分开，通知编译器，我们要声明一个类，名字叫做 Cattle。在 Cattle 的后面，我们有“`: NSObject`”，这是在通知编译器我们的 Cattle 是从 NSObject 继承而来的，关于继承和 NSObject，我们将在后面的章节里面详细介绍，关于“`: NSObject`”我们现在可以理解为，通过这样写，我们免费获得了苹果公司为我们精心准备的一系列的类和对象的必备的方法。NSObject 被称为 `root class`，也就是根类。在 Java 或者 .NET 里面，根类是必备的，C++ 不需要。在 Objective-C 里面原则上，你可以不使用 NSObject，构筑一个你自己的根类，但是事实上这样做将会有很大工作量，而且这样做没有什么意义，因为苹果为你提供的 NSObject 经过了很长时间的检验。也许有好奇心的同学们想自己构筑根类，不过至少笔者不会有自己去构筑一个根类的欲望。

好的，大家现在来看第 5 行。我们以前把这个东西叫做变量，我们从现在开始，需要精确的使用 Objective-C 的用语了，这是实体变量（`instance variables`，在有的英文资料里面会简写为 `iVars`）。虽然作为一个 Cattle，它有不只一个实体变量，比如说体重等等，但是为了代码简洁，我们在这里声明一个就是牛腿也就是牛股的数目，这个实体变量是 `int` 型，表示一个整数，我们当然不希望有 4.5 个牛腿。

我们来看第 6 行，第 6 行的括弧和在第 4 行最后的括弧用来表示实体变量的定义区间，编译器认为在这两个括弧之间的定义是实体变量的定义。当然，如果你的类没有实体变量，那么这两个括弧之间允许什么都没有。和 Java 以及 C++ 不一样，Objective-C 要求在括弧里面不能有方法也就是函数的定义，那么 Objective-C 里面的方法的定义放在什么地方呢，请看第 7 行。

第 7 行的第一个字母是一个减号“-”。这个减号就是告诉编译器，减号后面的方法，是实体方法（`instance method`）。实体方法的意思就是说，这个方法在类没有被实体化之前，是不能运行的。我们在这里看到的是减号，在有减号的同时也有加号，我们把带加号的方法称为类方法（`class method`），和实体方法相对应，类方法可以脱离实体而运行。关于类方法，我们将在后面的章节里面讲解。大家也许可以想起来在 C++ 和 Java 里面同样也有类似的区分，不是么。

在 Objective-C 里面方法的返回类型需要用圆括号包住，当编译器看到减号或者加号后面的括号了之后，就会认为这是在声明方法的返回值。你也可以不声明返回值，Objective-C 的编译器会给没有写显式的返回值函数加上一个默认的回值，它的类型是 `id`，关于 `id` 类型我们将在后面讲解，不过笔者不推荐不写返回值的类型。

在第 7 行我们定义了这个方法的名字是 `saySomething`，当然 `Cattle` 说的话我们人类是听不懂的，笔者只是想让它在我们的控制台里面输出一些我们可以看得懂得字符串。方法的声明最后，需要分号来标识，这一点保持了和 `C` 没有任何区别。

我们再来看看第 8 行，第 8 行和第 7 行多了“`:(int) count`”。其中冒号放在方法的后面是用来表示后面是用来定义变量的，同样变量的类型使用括号给包住，如果不写变量的类型的化，编译器同样认为这是一个 `id` 类型的。最后的 `count`，就是变量的名字。如果有不只一个变量怎么办？答案就是在第一个变量后面加冒号，然后加圆括号包住变量的类型，接着是变量的名字。

好了，我们在这里总结一下，类的定义方法如下：

```
@interface 类的名字 : 父类的名字 {  
    实体变量类型 实体变量名字;  
    ...  
}  
- (返回值类型) 方法名字;  
+ (返回值类型) 方法名字;  
- (返回值类型) 方法名字: (变量类型) 变量名字 标签 1: (变量类型) 变量 1 名字;  
...  
@end
```

...的意思在本系列入门讲座里面，...表示省略了一些代码的意思。

3.4，类的定义

我们在前一节讲述了类的声明，我们下一步将要看一下类的定义。请同学们打开“`Cattle.m`”文件：

```
1 #import "Cattle.h"  
2  
3  
4 @implementation Cattle  
5 -(void) saySomething
```

```

6 {
7     NSLog(@"Hello, I am a cattle, I have %d legs.", legsCount);
8 }
9 -(void) setLegsCount:(int) count
10 {
11     legsCount = count;
12 }
13 @end
14

```

Cattle.m 文件的第一行就 **import** 了 **Cattle.h** 文件,这一点和 **C** 的机制是一样的,关于 **#import** 的说明请参照第二章。

我们来看第 4 行和第 13 行,和头文件里面的 **@** 一样,我们这里类的定义也是使用的编译导向。编译器会把从 **@implementation** 到 **@end** 之间的部分看作是类的定义。**@implementation** 的后面有一个空格,空格的后面是我们的类的名字 **Cattle**,这是在告诉编译器,我们要定义 **Cattle** 类了。第 4 行和第 13 行之间是我们在头文件里面定义的实体方法或者类方法的定义部分,当然我们的类如果没有任何的实体方法和类方法的话,我们也许要写上 **@implementation** 和 **@end**,把中间留为空就可以了。

第 5 行是我们定义的 **saySomething** 的实现,我们可以发现第 5 行的内容和头文件 **Cattle.h** 的第 7 行是一致的。笔者个人认为在编写实体方法和类方法的定义的时候,为了避免手工输入产生的误差,可以从头文件当中把声明的部分拷贝过来,然后删除掉分号,加上两个花括弧。我们知道地 6 行到第 8 行是方法的定义的部分,我们再来看看第 7 行。第 7 行和第二章的 **Hello, World** 输出有些相似,只不过多了一个 **%d**,还有实体变量 **legsCount**,这个写法和 **C** 语言里面的 **printf** 是类似的,输出的时候会使用 **legsCount** 来替代字符串里面的 **%d**。

第 9 行的内容和 **Cattle.h** 的第 8 行一致的,这个不需要再解释了。我们来看看第 11 行,第 11 行是在说,把参数 **count** 的数值赋值给实体变量 **legsCount**。我们可以通过使用 **setLegsCount** 方法来控制 **Cattle** 对象里面 **legsCount** 的数值。

这部分内容的关键点为 **@implementation** 和 **@end**,理解了这个东西,其余的就不难理解了。我们来总结一下,类的定义部分的语法:

```

@implementation 类的名字
- (方法返回值) 方法名字
{

```

```

        方法定义
        ...
    }
- (方法返回值) 方法名字:(变量类型) 变量名字
{
    方法定义
    ...
}
...
@end

```

3.5,类的实例化

我们在 3.3 和 3.4 节里面分别声明和定义了一个 **Cattle** 的类。虽然定义好的类，但是我们是不能直接使用这个类的。因为类的内容需要被调入到内存当中我们称之为内存分配(**Allocation**)，然后需要把实体变量进行初始化 (**Initialization**)，当这些步骤都结束了之后，我们的类就被实例化了，我们把实例化完成的类叫做对象(**Object**)。好的，我们知道了我们在类的实例化过程当中需要做哪些工作，我们接着来看看我们已经搞定的 **Cattle** 类的定义和声明是怎样被实例化的。

```

1  #import <Foundation/Foundation.h>
2  #import "Cattle.h"
3
4  int main (int argc, const char * argv[]) {
5      NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
6
7      id cattle = [Cattle new];
8      [cattle setLegsCount:4];
9      [cattle saySomething];
10
11     [pool drain];
12     return 0;
13 }

```

同学们请看第 7 行的第一个单词 `id`。`id` 是英文 `identifier` 的缩写，我们在很多地方都遇到过 `id`，比如说在博客园里面，我们都使用 `id` 来登陆系统的，我们的 `id` 就代表着系统的一个用户。由于 `id` 在一个系统当中是唯一的，所以系统获得我们的 `id` 之后就知道我们是谁了。`Objective-C` 也是一样的道理，使用 `id` 来代表一个对象，在 `Objective-C` 当中，所有的对象都可以使用 `id` 来进行区分。我们知道一个类仅仅是一些数据外加上操作这些数据的代码，所以 `id` 实际上是指向数据结构的一个指针而已，相当于 `void*`。

第 7 行的第二个单词是 `cattle`，就是我们给这个 `id` 起的一个名字。当然，你可以起系统保留的名字以外的任何名字，不过为了维持代码的可读性，我们需要一个有意义的名字，我们这里使用头文字为小写的 `cattle`。

第 7 行的[`Cattle new`]是创建对象，`new` 实际上是 `alloc` 和 `init` 的组合，在 `Objective-C` 里面创建对象是一个为对象分配内存和初始化的过程。`new`，`alloc` 还有 `init` 定义在 `Cattle` 的超类 `NSObject` 里面，笔者将要在第 7 章里面详细的解释一下如何创建对象。在第 7 章之前我们都是用 `new` 来创建对象。

`Objective-C` 里面的方法的使用和其他语言有些不同，`Objective-C` 使用消息（`Message`）来调用方法。所以笔者认为在讲解第 7 行等号右边的部分之前，需要首先向大家介绍一个我们的新朋友，消息（`Message`）。所谓的消息就是一个类或者对象可以执行的动作。消息的格式如下：

```
[对象或者类名字 方法名字:参数序列];
```

首先我们观察到有两个中括弧，最右边的括弧之后是一个分号，当编译器遇到了这个格式之后会把中间的部分当作一个消息来发送。在上文的表达式当中，包括中括弧的所有部分的内容被称作消息表达式（`Message expression`），“对象或者类名字”被称作接收器（`Receiver`），也就是消息的接受者，“方法名字:参数序列”被称为一个消息（`Message`），“方法名字”被称作选择器（`Selector`）或者关键字（`Keyword`）。`Objective-C` 和 C 语言是完全兼容的，C 语言里面的中括弧用于表示数组，但是数组的格式明显和消息的发送的格式是不一样的，所以我们可以放心，编译器不会把我们的消息发送当作一个数组。

我们来回忆一下 C 语言里面函数的调用过程，实际上编译器在编译的时候就已经把函数相对于整个执行包的入口地址给确定好了，函数的执行实际上就是直接从这个地址开始执行的。

`Objective-C` 使用的是一种间接的方式，`Objective-C` 向对象或者类（具体上是对象还是类的名字取决于方法是实体方法还是类方法）发送消息，消息的格式应该和方法相同。具体来说，第 7 行等号右边的部分[`Cattle new`]就是说，向 `Cattle` 类发送一个 `new` 的消息。这样当 `Cattle`

类接收到 `new` 的时候，就会查找它可以相应的消息的列表，找到了 `new` 之后就会调用 `new` 的这个类方法，分配内存和初始化完成之后返回一个 `id`，这样我们就得到一个对象。

Objective-C 在编译的过程当中，编译器是会去检查方法是否有效的，如果无效会给你一个警告。但是编译器并不会阻止你执行，因为只有在执行的时候才会触发消息，编译器是无法预测到执行的时候会发生什么奇妙的事情的。使用这样的机制给程序毫无疑问将给带来极大的灵活性，因为我们和任意的对对象或者类发送消息，只要我们可以保证执行的时候类可以准确地找到消息并且执行就可以了，当然如果找不到的话，运行会出错。

任何事物都是一分为二的 ---

任何事物都是一分为二的，在我们得到了灵活性的时候我们损失的是执行的时间。Objective-C 的这种方式要比直接从函数的入口地址执行的方式要消耗更多的执行时间，虽然编译器对寻找的过程作过一定的优化。

有的同学会觉得奇怪，我们在 `Cattle` 里面并没有定义 `new`，我们可以向 `Cattle` 发送这个类方法么？答案是可以，因为 `new` 在 `NSObject` 里面，实际上响应 `new` 消息的是 `NSObject`。实际上 `new` 类似于一个宏，并不是一个“原子”的不可再分的方法，关于详细的情况，我们将在后续的章节里面讲解。

有了第 7 行的讲解，那么第 8 行的内容就不难理解了，第 8 行实际上是想 `cattle` 对象发送一个 `setLegsCount` 的消息，参数是 4，参照 `Cattle.m`，我们可以发现这个时候我们希望实体变量 `legsCount` 是 4。第 8 行就更简单了，就是说向 `cattle` 对象发送一个 `saySomething` 的消息，从而实现了控制台的输出。

3.6，本章总结

通过本章的学习，同学们应该掌握如下概念

1. 如何声明一个类
2. 如何定义一个类
3. 实体变量的定义
4. 类方法和实体方法的定义
5. `id` 是什么
6. `NSObject` 的奇妙作用
7. 如何从类开始初始化对象
8. 消息的调用

感谢大家看到这里！我们下一章将要讲述继承的概念。

Objective-C 2.0 with Cocoa Foundation--- 4, 继承

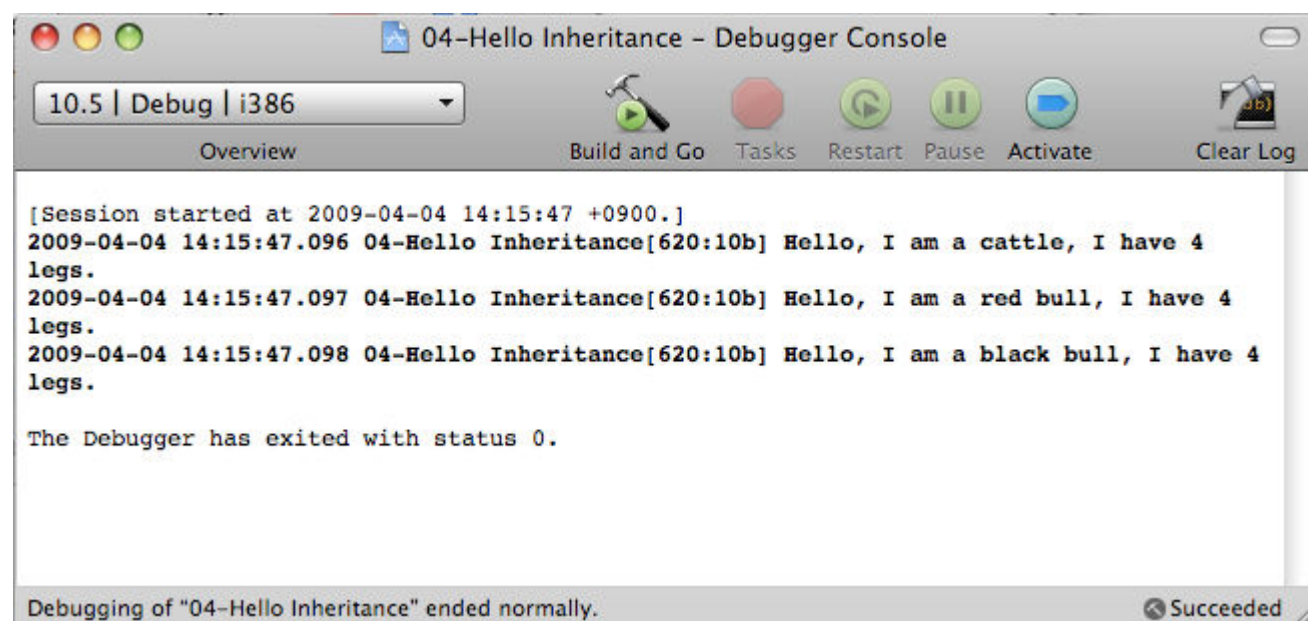
4, 继承

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，[请点击这里](#)。

上一章笔者介绍了一下在 Objective-C 里面的类的基本构造和定义以及声明的方法。我们知道在面向对象的程序里面，有一个很重要的需求就是代码的重复使用，代码的重复使用的重要方法之一就是继承。我们在这一章里面，将要仔细的分析一下继承的概念以及使用的方法。有过其他面向对象语言的同学，对这一章的内容应该不会感到陌生。

4.1,本章的程序的执行结果

在本章里面，我们将要重复使用第 3 章的部分代码。我们在第 3 章构筑了一个叫做 Cattle 的类，我们在这一章里面需要使用 Cattle 类，然后基于 Cattle 类，我们需要构筑一个子类，叫做 Bull 类。Bull 类里面，我们追加了一个实例变量，名字叫做 skinColor，我们也将要追加 2 个实例方法，分别 getSkinColor 还有 setSkinColor。我们然后需要更改一下我们的 main 函数，然后在 main 函数里面让我们的 Bull 做一下重要讲话。第 4 章程序的执行结果如图 4-1 所示：



```
[Session started at 2009-04-04 14:15:47 +0900.]
2009-04-04 14:15:47.096 04-Hello Inheritance[620:10b] Hello, I am a cattle, I have 4 legs.
2009-04-04 14:15:47.097 04-Hello Inheritance[620:10b] Hello, I am a red bull, I have 4 legs.
2009-04-04 14:15:47.098 04-Hello Inheritance[620:10b] Hello, I am a black bull, I have 4 legs.

The Debugger has exited with status 0.

Debugging of "04-Hello Inheritance" ended normally. Succeeded
```

图 4-1，本程序的执行结果

4.2，实现步骤

第一步，按照我们在第二章所述的方法，新建一个项目，项目的名字叫做 **04-Hello Inheritance**。

如果你是第一次看本篇文章，请到这里参看第二章的内容。

第二步，把鼠标移动到项目浏览器上面的“**Source**”上面，然后在弹出的菜单上面选择“**Add**”，然后在子菜单里面选择“**Existing Files**”，如图 4-2 所示

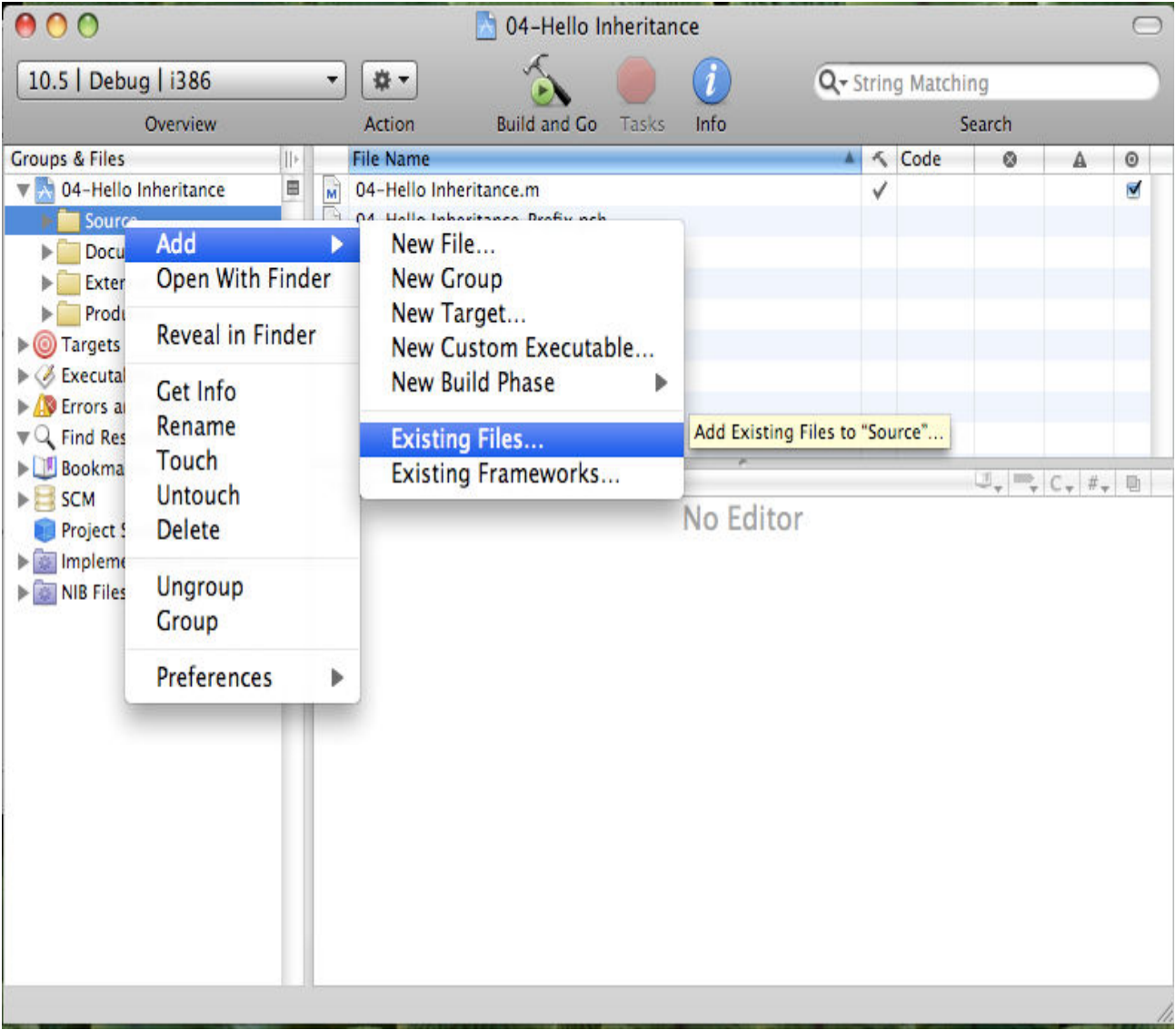


图 4-2，向项目追加文件

第三步，在文件选择菜单里面，选择第 3 章的项目文件夹“03-Hello Class”，打开这个文件夹之后，用鼠标和苹果电脑的 COMMAND 键，选泽文件“Cattle.h”和“Cattle.m”，然后按下“Add”按钮，如图 4-3 所示。如果你没有下载第 3 章的代码，请点击[这里](#)下载。

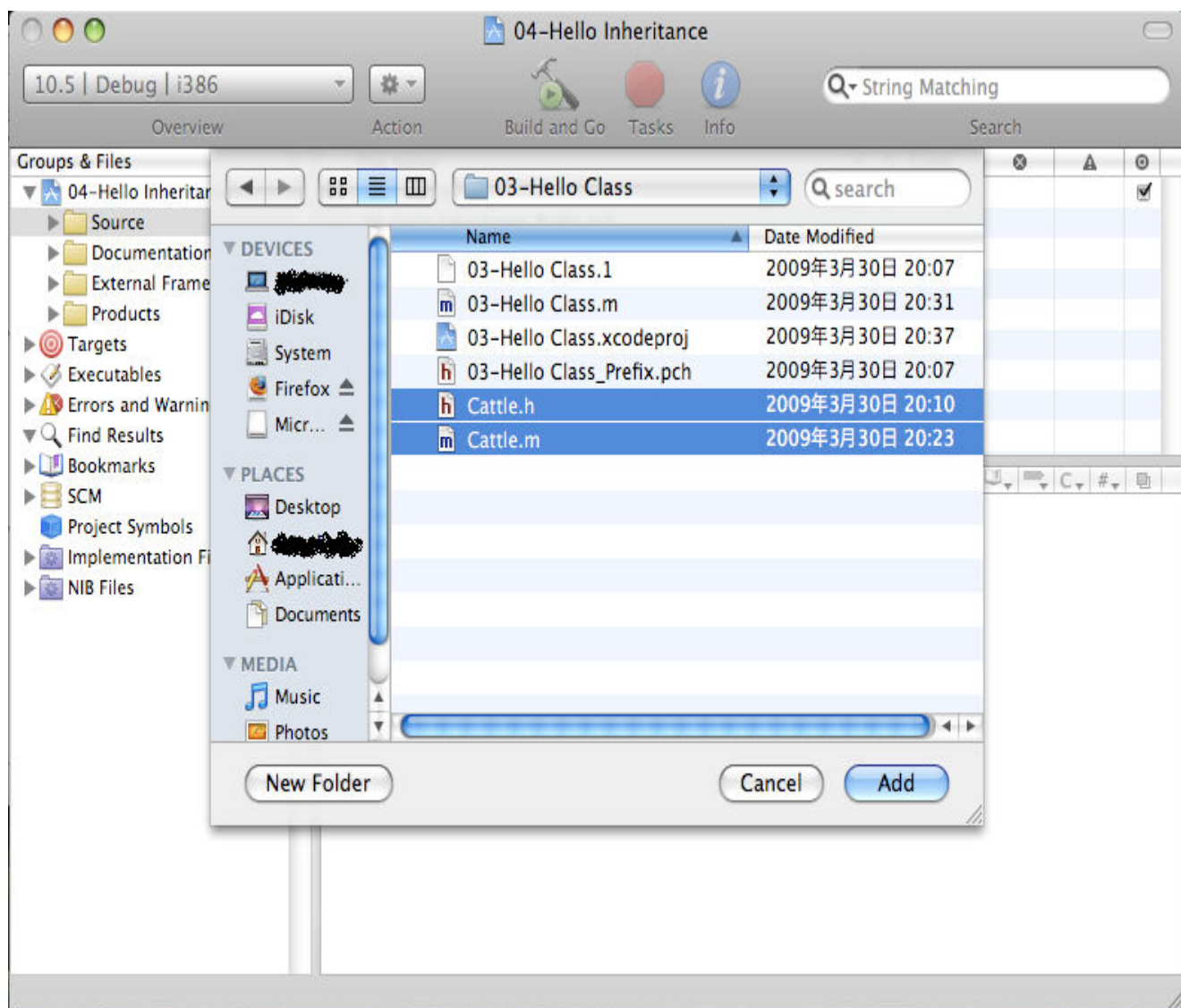


图 4-3，选择文件

第四步，在追加文件的选项对话框里面，让“Copy items into destination group's folder(if needed)”的单选框变为被选择的状态。这样就保证了我们在第三步里面选择的文件被拷贝到了本章的项目里面，可以避免我们不小心更改“Cattle.h”和“Cattle.m”对已经生效的第 3 章程序产生影响，虽然我们在本章里面不更改这 2 个代码。

第五步，把鼠标移动到项目浏览器上面的“Source”上面，然后在弹出的菜单上面选择“Add”，然后在子菜单里面选择“New Files”，然后在新建文件对话框的左侧选择“Cocoa Touch Classes”，然后在右侧窗口选择“NSObject subclass”，选择“Next”，在“New File”对话框里

面的“File Name”栏内输入“Bull.m”。在这里笔者没有给出图例，在这里新建文件的步骤和第 3 章的第二步到第四步相同，只是文件名字不一样。第一次看到本篇文章的同学可以参照第 3 章。

第六步，打开 Bull.h 做出如下修改，并且保存。

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

#import "Cattle.h"

@interface Bull : Cattle {

    NSString *skinColor;

}

- (void) saySomething;

- (NSString*) getSkinColor;

- (void) setSkinColor:(NSString *) color;

@end
```

第七步，打开 Bull.m 做出如下修改，并且保存

```
#import "Bull.h"

@implementation Bull

- (void) saySomething

{

    NSLog(@"Hello, I am a %@ bull, I have %d legs.", [self getSkinColor], legsCount);

}

- (NSString*) getSkinColor

{

    return skinColor;

}

- (void) setSkinColor:(NSString *) color

{

    skinColor = color;

}
```

```
}  
@end
```

第八步，打开 **04-Hello Inheritance.m** 文件，做出如下修改，并且保存

```
#import <Foundation/Foundation.h>  
  
#import "Cattle.h"  
  
#import "Bull.h"  
  
int main (int argc, const char * argv[]) {  
  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
    id cattle = [Cattle new];  
    [cattle setLegsCount:4];  
    [cattle saySomething];  
  
    id redBull = [Bull new];  
    [redBull setLegsCount:4];  
    [redBull setSkinColor:@"red"];  
    [redBull saySomething];  
  
    Bull *blackBull = [Bull new];  
    [blackBull setLegsCount:4];  
    [blackBull setSkinColor:@"black"];  
    [blackBull saySomething];  
  
    [pool drain];  
    return 0;  
}
```

第九步，选择屏幕上方菜单里面的“Run”，然后选择“Console”，打开了 Console 对话框之后，选择对话框上部中央的“Build and Go”，如果不出什么意外的话，那么应该出现入图 4-1 所示

的结果。如果出现了什么意外导致错误的话，那么请仔细检查一下你的代码。如果经过仔细检查发现 还是不能执行的话，可以到这里[下载](#)笔者为同学们准备的代码。如果笔者的代码还是不能执行的话，请告知笔者。

4.3，子类 **Subclass** 和超类 **Superclass**

让我们首先回忆一下第 3 章的 **Cattle.h**，在 **Cattle.h** 里面我们有如下的代码片断：

```
@interface Cattle : NSObject {
```

这段代码是在告诉编译器，我们的 **Cattle** 是继承的 **NSObject**。在这段代码当中，**NSObject** 是超类，**Cattle** 是子类。通过这样写，我们曾经免费的得到了 **NSObject** 里面的一个方法叫做 **new**。

```
id cattle = [Cattle new];
```

在面向对象的程序设计当中，如果在子类当中继承了超类的话，那么超类当中已经生效的部分代码在子类当中仍然是有效的，这样就大大的提高了代码的效率。基于超类我们可以把我们需要追加的一些功能放到子类里面去，在本章里面，我们决定基于 **Cattle** 类，重新生成一个子类 **Bull**：

```
1 #import <Foundation/Foundation.h>
2 #import "Cattle.h"
3
4 @interface Bull : Cattle {
5     NSString *skinColor;
6 }
7 - (void) saySomething;
8 - (NSString*) getSkinColor;
9 - (void) setSkinColor:(NSString *) color;
10 @end
```

上段代码里面的第 2 行，是通知编译器，我们这个类的声明部分需要 **Cattle.h** 文件。这个文件我们已经很熟悉了，是我们在第 3 章曾经构筑过的，在本章里面，我们不会改变里面的任何内容。

第 4 行，就是在通知编译器，我们需要声明一个类名字叫做 **Bull**，从 **Cattle** 里面继承过来。

第 5 行，我们追加了一个实例变量 **skinColor**，用来保存 **Bull** 的颜色。

第 7 行，我们重载了在 `Cattle` 类里面已经有的`(void)saySomething` 实例方法。重载 `(void)saySomething` 方法的主要原因是，我们认为 `Bull` 说的话应该和 `Cattle` 有所区别。

第 8 行到第 9 行，我们为 `Bull` 类声明了两个新的方法`(NSString*) getSkinColor` 和 `(void) setSkinColor:(NSString *) color`，分别用来设定和读取我们的实例变量 `skinColor`。

好的，我们总结一下继承的时候的子类的格式。

```
@interface 类的名字 : 父类的名字 {  
    实体变量类型 实体变量名字;  
  
}  
- (返回值类型) 重载的方法名字;  
+ (返回值类型) 重载的方法名字;  
- (返回值类型) 其他的方法名字:(变量类型) 变量名字:(变量类型) 变量名字;  
  
@end
```

4.4, self 和 super

我们再来打开“`Bull.m`”，在 `saySomething` 的定义的部分，我们发现了如下的代码：

```
NSLog(@"Hello, I am a %@ bull, I have %d legs.", [self getSkinColor], legsCount);
```

我们在这句话当中，发现的第一个新朋友是`%@`，这是在告诉编译器，需要把`%@`用一个后面定义的字符串来替换，在这里我们给编译器提供的字符串是`[self getSkinColor]`。看到这里，同学们又会发现一个新的朋友 `self`。

在类的方法定义域之内，我们有的时候需要访问这个类自己的实例变量，或者是方法。在类被实例化之后，我们就可以使用一个指向这个类本身的一个指针，在 `Java` 或者 `C++` 里面的名字叫做 `this`，在 `Objective-C` 里面，这个名字是 `self`。`self` 本身是一个 `id` 类型的一个指针变量。我们在第 3 章里面讲解过，方法的调用格式如下：

```
[对象或者类名字 方法名字:参数序列];
```

在类的方法定义域里面，当我们需要调用类的其他方法的时候，我们需要指定对象或者类的名字，我们的方法是一个实例方法所以我们需要一个指向自己的对象，在这里我们需要使用 `self`。

我们假设,如果方法声明里面的参数序列里面有一个参数的名字和类的实例变量发生重复的情况下并且由于某种原因我们无法更改参数和实例变量的名字的话,我们应该如何应对呢? 答案是使用 **self**, 格式如下

```
self->变量名字
```

通过这样写,我们可以取得到类的变量的数值。当然如果没有名字冲突的话,我们完全可以省略 **self->**, **Xcode** 也足够的聪明能够识别我们的实例变量, 并且把我们代码里面的实例变量更改为相应的醒目的颜色。

如果我们在类的方法里面需要访问超类的方法或者变量(当然是访问对子类来说是可视的方法或者变量),我们需要怎样写呢? 答案是使用 **super**, **super** 在本质上也是 **id** 的指针, 所以, 使用 **super** 访问变量和方法的时候的书写格式, 和 **self** 是完全一样的。

“**Bull.m**”里面的其他的代码, 没有什么新鲜的东西, 所以笔者就不在这里赘述了。

4.5, 超类方法和子类方法的执行

我们来看一下 **04-Hello Inheritance.m** 的下面的代码片断

```
1      id redBull = [Bull new];
2      [redBull setLegsCount:4];
3      [redBull setSkinColor:@"red"];
4      [redBull saySomething];
5
6      Bull *blackBull = [Bull new];
7      [blackBull setLegsCount:4];
8      [blackBull setSkinColor:@"black"];
9      [blackBull saySomething];
```

第 1 行的代码在第 3 章里面讲解过, 我们来看看第 2 行的代码。

第 2 行的代码实际上是向 **redBull** 发送一个 **setLegsCount** 消息, 参数为 4。我们没有在 **Bull** 里面定义 **setLegsCount** 方法, 但是从控制台的输出上来看, **setLegsCount** 明显是得到了执行。在执行的时候, 我们给 **redBull** 发送 **setLegsCount** 消息的时候, **runtime** 会在 **Bull** 的映射表当中寻找 **setLegsCount**, 由于我们没有定义所以 **runtime** 找不到的。**runtime** 没有找到指定的方法的话, 会接着需要 **Bull** 的超类, 也就是 **Cattle**。值得庆幸的是, **runtime** 在 **Cattle**

里面找到了 `setLegsCount`，所以就被执行了。由于 `runtime` 已经寻找到了目标的方法并且已经执行了，所以它就停止了寻找。我们假设 `runtime` 在 `Cattle` 里面也没有找到，那么它会接着在 `Cattle` 的超类 `NSObject` 里面寻找，如果还是找不到的话，由于 `NSObject` 是根类，所以它会报错的。关于具体内部是一个怎样的机制，我们将在后面的章节里面讲解。

第 3 行的代码，是设定 `skinColor`。

第 4 行的代码是给 `redBull` 发送 `saySomething` 的消息。按照第 2 行的 `runtime` 的寻找逻辑，它首先会在 `Bull` 类里面寻找 `saySomething`，这一次 `runtime` 很幸运，它一次就找到了，所以就立即执行。同时 `runtime` 也停止了寻找的过程，所以，`Cattle` 的 `saySomething` 不会得到执行的。

在第 6 行里面，我们定义了一个 `blackBull`，但是这一次我们没有使用 `id` 作为 `blackBull` 的类型，我们使用了 `Bull *`。从本质上来说，使用 `id` 还是 `Bull *` 是没有任何区别的。但是，我们来想象，当我们的程序存在很多 `id` 类型的变量的话，我们也许就难以区分究竟是什么类型的变量了。所以，在没有特殊的理由的情况之下，我们最好还是显式的写清楚类的名字，这样可以方便其他人阅读。由于 `Bull` 从 `Cattle` 继承而来，我们也可以把地 6 行代码改为

```
Cattle *blackBull = [Bull new];
```

4.6，本章总结

感谢大家阅读到这里！我们本章学习了：

1. 超类，子类的概念以及如何定义和声明。
2. `self` 和 `super` 的使用方法以及使用的时机。
3. 超类和子类的方法的执行。

Objective-C 2.0 with Cocoa Foundation--- 5，Class 类型，选择器 Selector 以及函数指针

5，Class 类型，选择器 Selector 以及指针函数

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，[请点击这里](#)。

上一章笔者介绍了在 **Objective-C** 里面继承的概念。有了继承的知识我们可以重复的使用很多以前生效的代码，这样就大大的提高了代码开发的效率。在本章，笔者要向同学们介绍几个非常重要的概念，**Class** 类型， 选择器 **Selector** 以及指针函数。

我们在实际上的编程过程中，也许会遇到这样的场景，那就是我们在写程序的时候不能确切的知道我们需要使用什么类，使用这个类的什么方法。在这个时候，我们需要在我们的程序里面动态的根据用户的输入来创建我们在写程序不知道的类的对象，并且调用这个对象的实例方法。

Objective-C 为我们提供了 **Class** 类型， 选择器 **Selector** 以及指针函数来实现这样的需求，从而大大的提高了我们程序的动态性能。

在 **Objective-C** 里面，一个类被正确的编译过后，在这个编译成功的类里面，存在一个变量用于保存这个类的信息。我们可以通过一个普通的字符串取得这个 **Class**，也可以通过我们生成的对象取得这个 **Class**。**Class** 被成功取得之后，我们可以把这个 **Class** 当作一个已经定义好的类来使用它。

Selector 和 **Class** 比较类似，不同的地方是 **Selector** 用于表示方法。在 **Objective-C** 的程序进行编译的时候，会根据方法的名字（包括参数列表）确定一个唯一的身份证明（实际上就是一个整数），不用的类里面的相同名字相同声明的方法的身份证明是一样的。这样在程序执行的时候，**runtime** 就不用费力的进行方法的名字比较来确定是执行哪一个方法了，只是通过一个整数的寻找就可以马上定位到相应的方法，然后找到相应的方法的入口地址，这样方法就可以被执行行了。

笔者在前面的章节里面叙述过，在 **Objective-C** 里面消息也就是方法的执行比 **C** 语言的直接找到函数入口地址执行的方式，从效率上来讲是比较低下的。尽管 **Objective-C** 使用了 **Selector** 等招数来提高寻找效率，但是无论如何寻找的过程，都是要消耗一定的时间的。好在 **Objective-C** 是完全兼容 **C** 的，它也有指针函数的概念。当我们需要执行效率的时候，比如说在一个很大的循环当中需要执行某个功能的时候，我们可以放弃向对某一个对象发送消息的手段，用指针函数取而代之，这样就可以获得和 **C** 语言一样的执行效率了。

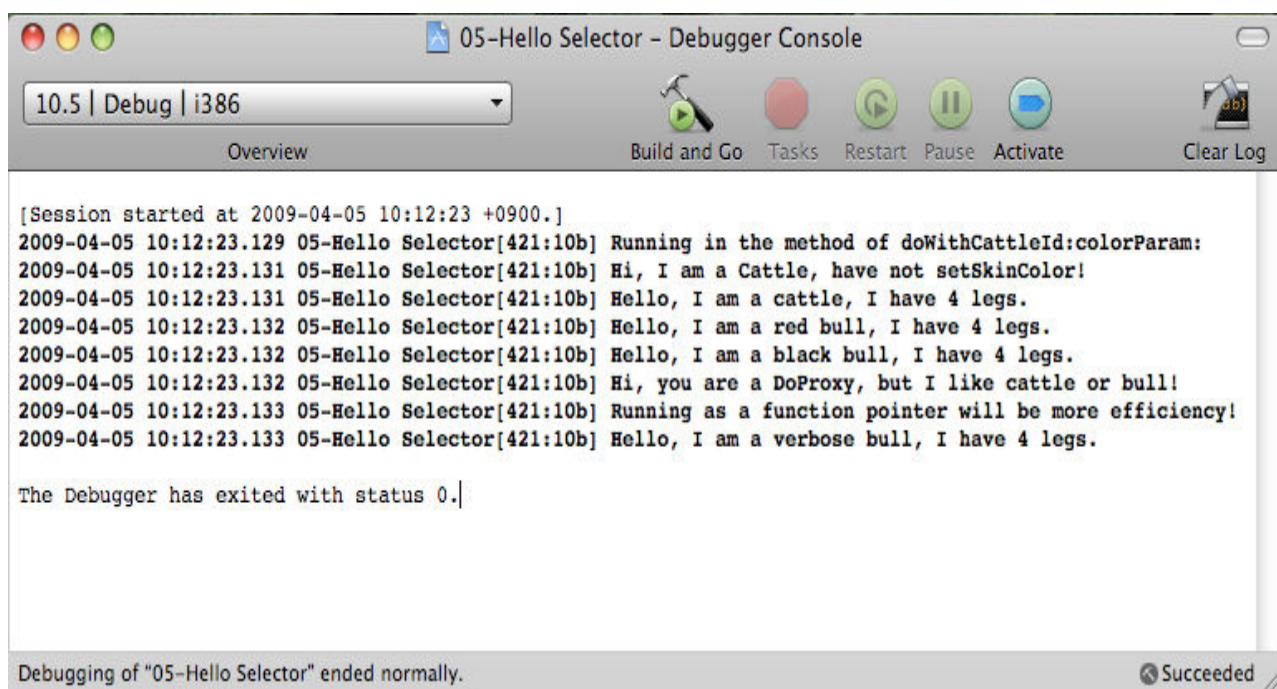
说到这里，可能有的同学已经有些茫然了。这些概念有些令人难以理解，但是它们确实是 **Objective-C** 的核心功能。掌握了这些核心的功能之后，同学们可以很轻松的看懂苹果的 **SDK** 里面的很多东西含义，甚至可以自己动手写一些苹果没有为我们提供的功能。所以建议大家仔细研读本章的内容，如果有什么问题，可以发个帖子大家可以共同探讨。

从笔者的观点上来看，对于有 **Java** 或者 **C++** 或者其他面向对象的语言的经验的同学们来说，前面的从第 1 到第 4 章的内容也许有些平淡无奇。从第 5 章开始，我们将要逐渐的深入到 **Objective-C** 的核心部分。笔者的最终目的，虽然是向大家介绍 **iPhone** 开发的入门，但是笔者

认为了解了 Objective-C 的基本概念以及使用方法之后，熟悉 iPhone 的应用程序的开发将是一件水到渠成的轻松的事情。否则如果你直接就深入到 iPhone 的开发的的话，在绝大多数时间你也许因为一个小小的问题就会困扰你几个小时甚至几天，解决这些问题的唯一方法就是熟悉 Objective-C 和 Cocoa Foundation 的特性。

好了，说了很多我们从下面就要开始，我们的手法和前面几章是一样的，我们首先要介绍一下本程序的执行结果。

5.1，本程序的执行结果



```
[Session started at 2009-04-05 10:12:23 +0900.]
2009-04-05 10:12:23.129 05-Hello Selector[421:10b] Running in the method of doWithCattleId:colorParam:
2009-04-05 10:12:23.131 05-Hello Selector[421:10b] Hi, I am a Cattle, have not setSkinColor!
2009-04-05 10:12:23.131 05-Hello Selector[421:10b] Hello, I am a cattle, I have 4 legs.
2009-04-05 10:12:23.132 05-Hello Selector[421:10b] Hello, I am a red bull, I have 4 legs.
2009-04-05 10:12:23.132 05-Hello Selector[421:10b] Hello, I am a black bull, I have 4 legs.
2009-04-05 10:12:23.132 05-Hello Selector[421:10b] Hi, you are a DoProxy, but I like cattle or bull!
2009-04-05 10:12:23.133 05-Hello Selector[421:10b] Running as a function pointer will be more efficiency!
2009-04-05 10:12:23.133 05-Hello Selector[421:10b] Hello, I am a verbose bull, I have 4 legs.

The Debugger has exited with status 0.
```

Debugging of "05-Hello Selector" ended normally. Succeeded

图 5-1，第 5 程序的执行结果

在本章里面，我们将要继续使用我们在前面几章已经构筑好的类 **Cattle** 和 **Bull**。为了灵活的使用 **Cattle** 和 **Bull**，我们将要构筑一个新的类，**DoProxy**。在 **DoProxy** 里面，我们将会引入几个我们的新朋友，他们分别是 **BOOL**，**SEL**，**IMP**，**CLASS**。通过这些新的朋友我们可以动态的通过设定文件取得 **Cattle** 和 **Bull** 的类，还有方法以及方法指针。下面将要介绍如何构筑本程序。同学们可以按照本章所述的步骤来构筑，也可以通过从这里下载。不过为了熟悉代码的写作，笔者强烈建议大家按照笔者所述的步骤来操作。

5.2，实现步骤

第一步，按照我们在第 2 章所述的方法，新建一个项目，项目的名字叫做 05-Hello Selector。如果你是第一次看本篇文章，请到这里参看第二章的内容。

第二步，按照我们在第 4 章的 4.2 节的第二，三，四步所述的方法，把在第 4 章已经使用过的“Cattle.h”，“Cattle.m”，“Bull.h”还有“Bull.m” 导入本章的项目里面。如果你没有第 4 章的代码，请到这里下载。如果你没有阅读第 4 章的内容，请参看这里。

第三步，把鼠标移动到项目浏览器上面的“Source”上面，然后在弹出的菜单上面选择“Add”，然后在子菜单里面选择“New Files”，然后在新建文件对话框的左侧选择“Cocoa Touch Classes”，然后在右侧窗口选择“NSObject subclass”，选择“Next”，在“New File”对话框里面的“File Name”栏内输入“DoProxy.m”。在这里笔者没有给出图例，在这里新建文件的步骤和第 3 章的第二步到第四步相同，只是文件名字不一样。第一次看到本篇 文章的同学可以参照第 3 章。

第四步，打开“DoProxy.h”做出如下修改并且保存

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

#define SET_SKIN_COLOR @"setSkinColor:"

#define BULL_CLASS @"Bull"

#define CATTLE_CLASS @"Cattle"

@interface DoProxy : NSObject {

    BOOL notFirstRun;

    id cattle[3];

    SEL say;

    SEL skin;

    void(*setSkinColor_Func) (id, SEL, NSString*);

    IMP say_Func;

    Class bullClass;

}

- (void) doWithCattleId:(id) aCattle colorParam:(NSString*) color;

- (void) setAllIvars;

- (void) SELFuncs;
```

```
- (void) functionPointers;

@end
```

第五步，打开“DoProxy.m”做出如下修改并且保存

```
#import "DoProxy.h"

#import "Cattle.h"

#import "Bull.h"

@implementation DoProxy

- (void) setAllIVars
{
    cattle[0] = [Cattle new];

    bullClass = NSClassFromString(BULL_CLASS);
    cattle[1] = [bullClass new];
    cattle[2] = [bullClass new];

    say = @selector(saySomething);
    skin = NSSelectorFromString(SET_SKIN_COLOR);
}

- (void) SELFuncs
{
    [self doWithCattleId:cattle[0] colorParam:@"brown"];
    [self doWithCattleId:cattle[1] colorParam:@"red"];
    [self doWithCattleId:cattle[2] colorParam:@"black"];
    [self doWithCattleId:self colorParam:@"haha"];
}

- (void) functionPointers
{
    setSkinColor_Func=(void (*)(id, SEL, NSString*)) [cattle[1] methodForSelector:skin];

    //IMP setSkinColor_Func = [cattle[1] methodForSelector:skin];
}
```

```

        say_Func = [cattle[1] methodForSelector:say];

        setSkinColor_Func(cattle[1], skin, @"verbose");

        NSLog(@"Running as a function pointer will be more efficiency!");

        say_Func(cattle[1], say);
    }

- (void) doWithCattleId:(id) aCattle colorParam:(NSString*) color
{
    if(notFirstRun == NO)
    {
        NSString *myName = NSStringFromSelector(_cmd);

        NSLog(@"Running in the method of %@", myName);

        notFirstRun = YES;
    }

    NSString *cattleParamClassName = [aCattle className];

    if([cattleParamClassName isEqualToString:BULL_CLASS] ||
        [cattleParamClassName isEqualToString:CATTLE_CLASS])
    {
        [aCattle setLegsCount:4];

        if([aCattle respondsToSelector:skin])
        {
            [aCattle performSelector:skin withObject:color];
        }
        else
        {
            NSLog(@"Hi, I am a %@, have not setSkinColor!", cattleParamClass
Name);
        }

        [aCattle performSelector:say];
    }
    else
    {

```

```

        NSString *yourClassName = [aCattle className];

        NSLog(@"Hi, you are a %@, but I like cattle or bull!", yourClassNam
e);
    }
}

@end

```

第六步，打开“05-Hello Selector.m” 作出如下修改并且保存

```

#import <Foundation/Foundation.h>

#import "DoProxy.h"

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

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

    DoProxy *doProxy = [DoProxy new];

    [doProxy setAllIVars];

    [doProxy SELFuncs];

    [doProxy functionPointers];

    [pool drain];

    return 0;
}

```

第七步，选择屏幕上方菜单里面的“Run”，然后选择“Console”，打开了 Console 对话框之后，选择对话框上部中央的“Build and Go”，如果不出什么意外的话，那么应该出现入图 5-1 所示的结果。如果出现了什么意外导致错误的话，那么请仔细检查一下你的代码。如果经过仔细检查发现 还是不能执行的话，可以到这里[下载](#)笔者为同学们准备的代码。如果笔者的代码还是不能执行的话，请告知笔者。

5.3, BOOL 类型

我们现在打开“DoProxy.h”文件。“DoProxy.h”文件的第 3 行到第 5 行是三个预定义的三个字符串的宏。我们将在程序当中使用这 3 个宏，为了实现代码的独立性，在实际的程序开发当中，我们也许考虑使用一个配置的文本文件或者一个 XML 来替代这些宏。但是现在由于笔者的主要

目的是讲解 Objective-C 的概念，为了避免较多的代码给大家带来理解主题的困难，所以笔者没有使用配置文件或者 XML 来表述这些可以设定的常量。

“DoProxy.h”的第 7 行对同学们来说也是老朋友了，是通知编译器，我们需要声明一个 DoProxy 类，从 NSObject 继承。

我们在第 8 行遇到了我们的一个新的朋友，BOOL：

```
BOOL notFirstRun;
```

我们定义了一个 notFirstRun 的实例变量，这个变量是布尔类型的。我们的实例方法 doWithCattleId 需要被执行多次，我们在第一次执行 doWithCattleId 的时候需要向控制台输出包含 doWithCattleId 的方法名字的字符串，关于这个字符串的内容，请参考图 5-1。

好的，我们现在需要看看在 Objective-C 里面 BOOL 是怎样定义的，我们把鼠标移动到 BOOL 上面，然后单击鼠标右键选择弹出菜单的“Jump to Definition”，然后 Xcode 会打开 objc.h 文件，我们看到下面的代码：

```
typedef signed char          BOOL;

// BOOL is explicitly signed so @encode(BOOL) == "c" rather than "C"
// even if -funsigned-char is used.

#define OBJC_BOOL_DEFINED

#define YES                  (BOOL) 1
#define NO                   (BOOL) 0
```

我们看到这段代码，我们可以这样理解，在 Objective-C 里面，BOOL 其实是 signed char，YES 是 1，NO 是 0。我们可以这样给 BOOL 赋值：

```
BOOL x = YES;

BOOL y = NO;
```

关于 BOOL，实际上就是一个开关的变量，但是我们需要注意下面 2 点：

第一点，从本质上来说 BOOL 是一个 8bit 的一个 char，所以我们在把其他比如说 short 或者 int 转换成为 BOOL 的时候一定要注意。如果 short 或者 int 的最低的 8 位 bit 都是 0 的话，尽管除了最低的 8 位以外都不是 0，那么经过转换之后，就变成了 0 也就是 NO。比如说我们有一个 int 的值是 0X1000，经过 BOOL 转换之后就变成了 NO。

第二点，Objective-C 里面的所有的逻辑判断例如 if 语句等等和 C 语言保持兼容，如果数值不是 0 判断为真，如果数值是 0 那么就判断为假，并不是说定义了 BOOL 值之后就变成了只有 1 或者 YES 为真。所以下面的代码的判断都为真：

```
if(0x1000)
if(2)
if(-1)
```

5.4，SEL 类型

让我们接着看“DoProxy.h”文件的下列代码：

```
1      id cattle[3];
2      SEL say;
3      SEL skin;
```

其中 id cattle[3]定义了一个数组用于存储 Cattle 或者 Bull 对象。这一行代码估计大家都很熟悉，笔者就不赘述了。像这样的传统的数组并不能完全满足我们的需求，当我们需要做诸如追加，删除等操作的时候，会很不方便。在随后的章节里面笔者将要向大家介绍传统数组的替代解决方案 NSArray。

上一段代码的第二行和第三行是本节所关注的，就是 SEL 类型。Objective-C 在编译的时候，会根据方法的名字（包括参数序列），生成一个用来区分这个方法的唯一的一个 ID，这个 ID 就是 SEL 类型的。我们需要注意的是，只要方法的名字（包括参数序列）相同，那么它们的 ID 都是相同的。就是说，不管是超类还是子类，不管是有没有超类和子类的关系，只要名字相同那么 ID 就是一样的。除了函数名字和 ID，编译器当然还要把方法编译成为机器可以执行的代码，这样，在一个编译好的类里面，就产生了如下图所示方法的表格示意图（本构造属于笔者推测，没有得到官方证实，所以图 5-2 为示意图仅供参考，我们可以暂时认为是这样的）。

Bull类		
方法名字	方法ID	地址
saySomething	1 001	0X2001
getSkinColor	1 002	0X2002
setSkinColor:	1 003	0X2003

图 5-2，方法的表格示意图

请注意 `setSkinColor` 后面有一个冒号，因为它是带参数的。由于存在这样的一个表格，所以在程序执行的时候，我们可以方便的通过方法的名字，获取到方法的 ID 也就是我们所说的 SEL，反之亦然。具体的使用方法如下：

```
1 SEL 变量名 = @selector(方法名字);
2 SEL 变量名 = NSSelectorFromString(方法名字的字符串);
3 NSString *变量名 = NSStringFromSelector(SEL 参数);
```

其中第 1 行是直接写在程序里面写上方法的名字，第 2 行是写上方法名字的字符串，第 3 行是通过 SEL 变量获得方法的名字。我们得到了 SEL 变量之后，可以通过下面的调用来给一个对象发送消息：

```
[对象 performSelector:SEL 变量 withObject:参数 1 withObject:参数 2];
```

这样的机制大大的增加了我们的程序的灵活性，我们可以通过给一个方法传递 SEL 参数，让这个方法动态的执行某一个方法；我们也可以通过配置文件指定需要执行的方法，程序读取配置文件之后把方法的字符串翻译成为 SEL 变量然后给相应的对象发送这个消息。

从效率的角度上来说，执行的时候不是通过方法名字而是方法 ID 也就是一个整数来查找方法，由于整数的查找和匹配比字符串要快得多，所以这样可以在某种程度上提高执行的效率。

5.5，函数指针

在讲解函数指针之前，我们先参看一下图 5-2，函数指针的数值实际上就是图 5-2 里面的地址，有人把这个地址成为函数的入口地址。在图 5-2 里面我们可以通过方法名字取得方法的 ID，同样我们也可以通过方法 ID 也就是 SEL 取得函数指针，从而在程序里面直接获得方法的执行地址。或者函数指针的方法有 2 种，第一种是传统的 C 语言方式，请参看“DoProxy.h”的下列代码片断：

```
1 void(*setSkinColor_Func) (id, SEL, NSString*);
2 IMP say_Func;
```

其中第 1 行我们定义了一个 C 语言里面的函数指针，关于 C 语言里面的函数指针的定义以及使用方法，请参考 C 语言的书籍和参考资料。在第一行当中，值得我们注意的是这个函数指针的参数序列：

第一个参数是 id 类型的，就是消息的接受对象，在执行的时候这个 id 实际上就是 self，因为我们将要向某个对象发送消息。

第二个参数是 **SEL**，也是方法的 **ID**。有的时候在消息发送的时候，我们需要使用 `_cmd` 来获取方法自己的 **SEL**，也就是说，方法的定义体里面，我们可以通过访问 `_cmd` 得到这个方法自己的 **SEL**。

第三个参数是 **NSString***类型的，我们用它来传递 **skin color**。在 **Objective-C** 的函数指针里面，只有第一个 **id** 和第二个 **SEL** 是必需的，后面的参数有还是没有，如果有那么有多少个要取决于方法的声明。

现在我们来介绍一下 **Objective-C** 里面取得函数指针的新的定义方法，**IMP**。

上面的代码的第一行比较复杂，令人难以理解，**Objective-C** 为我们定义了一个新的数据类型就是在上面第二行代码里面出现的 **IMP**。我们把鼠标移动到 **IMP** 上，单击右键之后就可以看到 **IMP** 的定义，**IMP** 的定义如下：

```
typedef id (*IMP)(id, SEL, ...);
```

这个格式正好和我们在第一行代码里面的函数指针的定义是一样的。

我们取得了函数指针之后，也就意味着我们取得了执行的时候的这段方法的代码的入口，这样我们就可以像普通的 **C** 语言函数调用一样使用这个函数指针。当然我们可以把函数指针作为参数传递到其他的方法，或者实例变量里面，从而获得极大的动态性。我们获得了动态性，但是付出的代价就是编译器不知道我们要执行哪一个方法所以在编译的时候不会替我们找出错误，我们只有执行的时候才知道，我们写的函数指针是否是正确的。所以，在使用函数指针的时候要非常准确地把握能够出现的所有可能，并且做出预防。尤其是当你在写一个供他人调用的接口 **API** 的时候，这一点非常重要。

5.6, Class 类型

到目前为止，我们已经知道了对应于方法的 **SEL** 数据类型，和 **SEL** 同样在 **Objective-C** 里面我们不仅仅可以使用对应于方法的 **SEL**，对于类在 **Objective-C** 也为我们准备了类似的机制，**Class** 类型。当一个类被正确的编译过后，在这个编译成功的类里面，存在一个变量用于保存这个类的信息。我们可以通过一个普通的字符串取得 这个 **Class**，也可以通过我们生成的对象取得这个 **Class**。**Class** 被成功取得之后，我们可以把这个 **Class** 当作一个已经定义好的类来使用它。这样的机制允许我们在程序执行的过程当中，可以 **Class** 来得到对象的类，也可以在程序执行的阶段动态的生成一个在编译阶段无法确定的一个对象。

因为 **Class** 里面保存了一个类的所有信息，当然，我们也可以取得一个类的超类。关于 **Class** 类型，具体的使用格式如下：

```
1    Class 变量名 = [类或者对象 class];
2    Class 变量名 = [类或者对象 superclass];
3    Class 变量名 = NSClassFromString(方法名字的字符串);
4    NSString *变量名 = NSStringFromClass(Class 参数);
```

第一行代码，是通过向一个类或者对象发送 **class** 消息来获得这个类或者对象的 **Class** 变量。

第二行代码，是通过向一个类或者对象发送 **superclass** 消息来获得这个类或者对象的超类的 **Class** 变量。

第三行代码，是通过调用 **NSClassFromString** 函数，并且把一个字符串作为参数来取得 **Class** 变量。这个在我们使用配置文件决定执行的时候的类的时候，**NSClassFromString** 给我们带来了极大的方便。

第四行代码，是 **NSClassFromString** 的反向函数 **NSStringFromClass**，通过一个 **Class** 类型作为变量取得一个类的名字。

当我们在程序里面通过使用上面的第一，二或者第三行代码成功的取得一个 **Class** 类型的变量，比如说我们把这个变量名字命名为 **myClass**，那么我们在以后的代码种可以把 **myClass** 当作一个我们已经定义好的类来使用，当然我们可以把这个变量作为参数传递到其他的方法当中让其他的方法动态的生成我们需要的对象。

5.7, DoProxy.h 里面的方法定义

DoProxy.h 里面还有一些实例方法，关于方法的定义的格式，同学们可以参照第三章。我们现在要对 **DoProxy.h** 里面定义的方法的做一下简要的说明。

```
1 - (void) doWithCattleId:(id) aCattle colorParam:(NSString*) color;
2 - (void) setAllIVars;
3 - (void) SELFuncs;
4 - (void) functionPointers;
```

第一行的方法，是设定 **aCattle**，也就是 **Cattle** 或者 **Bull** 对象的属性，然后调用 **saySomething** 方法，实现控制台的打印输出。

第二行的方法，是把我们定义的 **DoProxy** 类里面的一些变量进行赋值。

第三行的方法，是调用 **doWithCattleId** 方法。

第四行的方法，是调用了函数指针的方法。

好的，我们把 **DoProxy.h** 的内容介绍完了，让我们打开 **DoProxy.m**。

5.8, DoProxy.m 的代码说明

有了 DoProxy.h 的说明，同学们理解 DoProxy.m 将是一件非常轻松的事情，让我们坚持一下把这个轻松的事情搞定。由于篇幅所限，笔者在这里的讲解将会省略掉非本章的内容。

DoProxy.m 代码如下：

```
1 #import "DoProxy.h"
2 #import "Cattle.h"
3 #import "Bull.h"
4
5 @implementation DoProxy
6 - (void) setAllIVars
7 {
8     cattle[0] = [Cattle new];
9
10    bullClass = NSClassFromString(BULL_CLASS);
11    cattle[1] = [bullClass new];
12    cattle[2] = [bullClass new];
13
14    say = @selector(saySomething);
15    skin = NSSelectorFromString(SET_SKIN_COLOR);
16 }
17 - (void) SELFuncs
18 {
19     [self doWithCattleId:cattle[0] colorParam:@"brown"];
20     [self doWithCattleId:cattle[1] colorParam:@"red"];
21     [self doWithCattleId:cattle[2] colorParam:@"black"];
22     [self doWithCattleId:self colorParam:@"haha"];
23 }
24 - (void) functionPointers
25 {
26     setSkinColor_Func=(void (*)(id, SEL, NSString*)) [cattle[1] methodFor
Selector:skin];
```

```

27     //IMP setSkinColor_Func = [cattle[1] methodForSelector:skin];
28     say_Func = [cattle[1] methodForSelector:say];
29     setSkinColor_Func(cattle[1],skin,@"verbose");
30     NSLog(@"Running as a function pointer will be more efficiency!");
31     say_Func(cattle[1],say);
32 }

33 - (void) doWithCattleId:(id) aCattle colorParam:(NSString*) color
34 {
35     if(notFirstRun == NO)
36     {
37         NSString *myName = NSStringFromSelector(_cmd);
38         NSLog(@"Running in the method of %@", myName);
39         notFirstRun = YES;
40     }
41
42     NSString *cattleParamClassName = [aCattle className];
43     if([cattleParamClassName isEqualToString:BULL_CLASS] ||
44        [cattleParamClassName isEqualToString:CATTLE_CLASS])
45     {
46         [aCattle setLegsCount:4];
47         if([aCattle respondsToSelector:skin])
48         {
49             [aCattle performSelector:skin withObject:color];
50         }
51         else
52         {
53             NSLog(@"Hi, I am a %@, have not setSkinColor!", cattleParamCl
assName);
54         }
55         [aCattle performSelector:say];
56     }
57     else

```

```

58     {
59         NSString *yourClassName = [aCattle className];
60         NSLog(@"Hi, you are a %@, but I like cattle or bull!", yourClassName);
61     }
62 }
63 @end

```

第 10 行代码是通过一个预定义的宏 **BULL_CLASS** 取得 **Bull** 的 **Class** 变量。

第 11 和 12 行代码是使用 **bullClass** 来初始化我们的 **cattle** 实例变量数组的第 2 和第 3 个元素。

第 14 行是通过 **@selector** 函数来取得 **saySomething** 的 **SEL** 变量。

第 15 行是通过向 **NSSelectorFromString** 传递预定义的宏 **SET_SKIN_COLOR** 来取得 **setSkinColor** 的 **SEL** 变量。

第 22 行，笔者打算“戏弄”一下 **doWithCattleId**，向传递了不合适的参数。

第 26 行，笔者取得了传统的 C 语言的函数指针，也是使用了第 5.5 节所述的第一种取得的方法。

第 28 行，笔者通过 5.5 节所述的第二种取得的方法得到了函数指针 **say_Func**。

第 29 行和 31 行分别执行了分别第 26 行和 28 行取得的函数指针。

第 35 行是一个 **BOOL** 型的实例变量 **notFirstRun**。当对象被初始化之后，确省的值是 **NO**。第一次执行完毕之后，我们把这个变量设定成为 **YES**，这样就保证了花括号里面的代码只被执行一次。

第 37 行我们通过 **_cmd** 取得了 **doWithCattleId** 这个方法名字用于输出。当然同学们在设计方法的提供给别人使用的时候，为了防止使用方法的人把这个方法本身传递进来造成死循环，需要使用 **_cmd** 这个系统隐藏的变量判断一下。笔者在这里没有做出判断，这样写从理论上来说存在一定的风险。

第 42 行，我们通过向对象发送 **className** 消息来取得这个对象的类的名字。

第 43 行和第 44 行，我们通过 **NSString** 的方法 **isEqualToString** 来判断取得的类的名字是否在我们事先想象的范围之内，我们只希望接受 **Bull** 或者 **Cattle** 类的对象。

第 46 行，本来我们想通过 **SEL** 的方式来进行这个牛股的设定，但是由于它的参数不是从 **NSObject** 继承下来的，所以我们无法使用。我们会有办法解决这个问题的，我们将在后面的章节里面介绍解决这个问题的方法。

第 47 行的代码，有一个非常重要 `NSObject` 的方法 `respondToSelector`，通过向对象发送这个消息，加上一个 `SEL`，我们可以知道这个对象是否可以相应这个 `SEL` 消息。由于我们的 `Cattle` 无法相应 `setSkinColor` 消息，所以如果对象是 `Cattle` 类生成的话，`if` 语句就是 `NO` 所以花括号里面的内容不会得到执行。

第 59 行，我们通过类的名字发现了一个假冒的 `Cattle`，我们把这个假冒的家伙给揪出来，然后实现了屏幕打印。

5.9，本章总结

本章给同学们介绍了几个新的数据类型，以及使用方法，这些数据类型分别是 `BOOL`，`SEL`，`Class`，`IMP`。

本章的内容很重要，希望同学们花一点时间仔细的理解一下。应该说，本章的内容有些令人难以理解，或者说知道了 `SEL`，`Class`，`IMP` 之后也许不知道如何使用，遇到什么情况的时候需要使用。不过在学习 `Objective-C` 的初级阶段，不知道这些也没有关系，但是 `SEL`，`Class`，`IMP` 的概念需要掌握，否则当你遇到别人写的质量比较高的代码或者苹果官方的技术文档的时候，你会觉得理解起来比较吃力。

本章内容也是理解下一章内容的基础，下一章我们将要讲述 `NSObject` 的奥秘。

谢谢大家！

Objective-C 2.0 with Cocoa Foundation--- 6，NSObject 的奥秘

6，NSObject 的奥秘

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，[请点击这里](#)。

在上一章里面，笔者向大家介绍了在 `Objective-C` 里面的几个非常重要的概念，简单的说就是 `SEL`，`Class` 和 `IMP`。我们知道 `Objective-C` 是 `C` 语言的扩展，有了这 3 个概念还有我们以前讲过的继承和封装的概念，`Objective-C` 发生了翻天覆地的变化，既兼容 `C` 语言的高效特性又实现了面向对象的功能。

`Objective-C` 从本质上来说，还是 `C` 语言的。那么内部究竟是怎样实现 `SEL`，`Class` 和 `IMP`，还有封装和继承的？为了解答这个问题，笔者决定在本章向大家概要的介绍一下 `Objective-C` 的最主要的一个类，`NSObject`。

不过说实在话，如果同学们觉得本章的内容比较晦涩难懂的话，不阅读本章的内容丝毫不会对写程序产生任何不良的影响，但是如果掌握了本章的内容的话，对加深对 Objective-C 的理解，对于今后笔者将要讲述的内容而言，将会是一个极大的促进。

6.1，本程序的执行结果

在本章里面，我们将要继续使用我们在前面几章已经构筑好的类 **Cattle** 和 **Bull**。由于在现在的 Xcode 版本里面，把一些重要的东西比如说 **Class** 的原型定义都放到了 **LIB** 文件里面，所以这些东西的具体的定义，对于我们来说是不可见的。

我们首先把第 4 章的代码打开，然后打开“Cattle.h”文件，把鼠标移动到“NSObject”上面，单击鼠标右键，在弹出菜单里面选择“Jump to Definition”。然后会弹出一个小菜单，我们选择“interface NSObject”。我们可以看到如下代码

```
@interface NSObject <NSObject> {  
  
    Class      isa;  
  
    ...  
}
```

我们知道了，所谓的 **NSObject** 里面只有一个变量，就是 **Class** 类型的 **isa**。**isa** 的英文的意思就是 **is a pointer** 的意思。也就是说 **NSObject** 里面只有一个实例变量 **isa**。好的，我们需要知道 **Class** 是一个什么东西，我们把鼠标移动到“Class”上面，单击鼠标右键，在弹出菜单里面选择“Jump to Definition”，我们看到了如下的代码：

```
typedef struct objc_class *Class;  
  
typedef struct objc_object {  
  
    Class isa;  
  
} *id;  
  
...
```

我们在这里知道了，**Class** 实际上是一个 **objc_class** 的指针类型，我们把鼠标移动到“**objc_class**”上面，单击鼠标右键，在弹出菜单里面选择“Jump to Definition”，发现我们还是在在这个窗口里面，Xcode 并没有把我们带到 **objc_class** 的定义去，所以我们无从知道 **objc_class** 内部究竟是一个什么样的东西。

笔者顺便提一下，大家也许注意到了 **id** 的定义，**id** 实际上是 **objc_object** 结构的一个指针，里面只有一个元素那就是 **Class**。那么根据上面我们看到的，所谓的 **id** 就是 **objc_class** 的指针的指针。让我们回忆一下下面的代码：

```
id cattle = [Cattle new];
```

这句话是在初始化和实例化 **cattle** 对象，这个过程，实际上可以理解为，**runtime** 为我们初始化好了 **Class** 的指针，并且把这个指针返回给我们。我们初始化对象完成了之后，实际上我们得到的对象就是一个指向这个对象的 **Class** 指针。

让我们在回过头来说说这个神秘的 **Class**，我们无法在 **Xcode** 里面看到 **Class** 也就是 **objc_class** 的定义。庆幸的是这部分的定义是 **GCC** 代码，是开源的。笔者下载了开源的代码之后，把开源的代码作了一些小小的调整，然后把 **Class** 的定义等等放到了我们的工程文件里面去，通过类型转化之后，我们终于可以看到 **Class**，**SEL**，还有 **isa** 等等过去对我们来说比较“神秘”的东西的真正面目。

我们在前面几章里面在每一个章的第一节里面都要介绍一下本程序执行结果的屏幕拷贝，本章也是一样，但是本章的执行结果非常简单。因为对于本章而言重点应该是放在对 **NSObject** 机制的理解上。

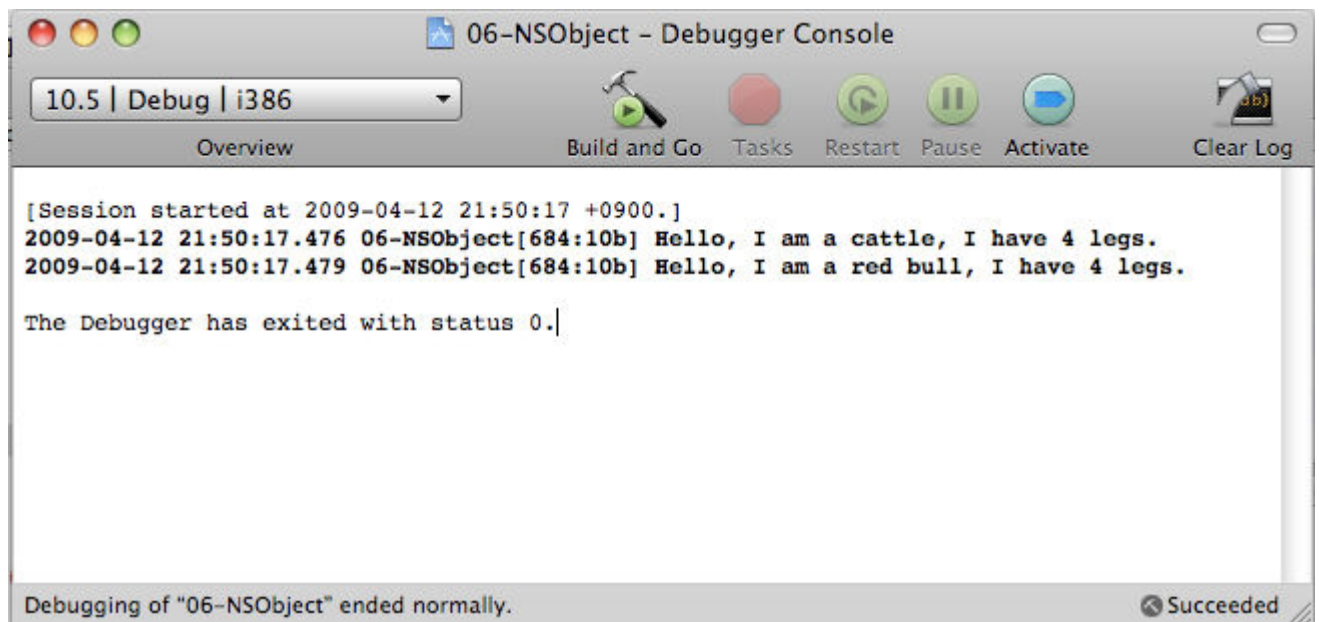


图 6-1，本程序运行结果

大家看到本程序的运行结果的屏幕拷贝的时候，也许会觉得很无趣，因为单单从结果画面，我们没有发现任何令人感到很有趣的东西，相反，都是同学们已经很熟悉的一些老面孔。但是本章所要讲述的东西也许是同学们在其他语言里面从来没有遇到过的东西，这些东西将会令人感到新鲜和激动。

6.2，实现步骤

第一步，按照我们在第 2 章所述的方法，新建一个项目，项目的名字叫做 06-NSObject。如果你是第一次看本篇文章，请到这里参看第二章的内容。

第二步，按照我们在第 4 章的 4.2 节的第二，三，四步所述的方法，把在第 4 章已经使用过的“Cattle.h”，“Cattle.m”，“Bull.h”还有“Bull.m” 导入本章的项目里面。如果你没有第 4 章的代码，请到这里下载。如果你没有阅读第 4 章的内容，请参看这里。

第三步，把鼠标移动到项目浏览器上面的“Source”上面，然后在弹出的菜单上面选择“Add”，然后在子菜单里面选择“New File”，然后在新建文件对话框的左侧最下面选择“Other”，然后在右侧窗口选择“Empty File”，选择“Next”，在“New File”对话框里面的“File Name”栏内输入“MyNSObject.h”。然后输入（或者是拷贝也可以，因为这是 C 的代码，如果你很熟悉 C 语言的话，可以拷贝一下节省时间）如下代码：

```
#include <stddef.h>

typedef const struct objc_selector

{

    void *sel_id;

    const char *sel_types;

} *MySEL;

typedef struct my_objc_object {

    struct my_objc_class* class_pointer;

} *myId;

typedef myId (*MyIMP)(myId, MySEL, ...);

typedef char *STR;                                /* String alias */

typedef struct my_objc_class *MetaClass;

typedef struct my_objc_class *MyClass;

struct my_objc_class {

    MetaClass class_pointer;

    struct my_objc_class* super_class;

    const char* name;

    long version;
```

```

    unsigned long      info;

    long               instance_size;

    struct objc_ivar_list* ivars;

    struct objc_method_list* methods;

    struct sarray *    dtable;

    struct my_objc_class* subclass_list;

    struct my_objc_class* sibling_class;

    struct objc_protocol_list *protocols;

    void* gc_object_type;
};

typedef struct objc_protocol {

    struct my_objc_class* class_pointer;

    char *protocol_name;

    struct objc_protocol_list *protocol_list;

    struct objc_method_description_list *instance_methods, *class_methods;

} Protocol;

typedef void* retval_t;

typedef void(*apply_t)(void);

typedef union arglist {

    char *arg_ptr;

    char arg_regs[sizeof (char*)];

} *arglist_t;

typedef struct objc_ivar* Ivar_t;

typedef struct objc_ivar_list {

    int    ivar_count;

    struct objc_ivar {

        const char* ivar_name;

        const char* ivar_type;

```

```

        int            ivar_offset;

    } ivar_list[1];
} IvarList, *IvarList_t;

typedef struct objc_method {
    MySEL            method_name;

    const char* method_types;

    MyIMP            method_imp;
} Method, *Method_t;

typedef struct objc_method_list {
    struct objc_method_list* method_next;

    int            method_count;

    Method method_list[1];
} MethodList, *MethodList_t;

struct objc_protocol_list {
    struct objc_protocol_list *next;

    size_t count;

    Protocol *list[1];
};

```

第四步，打开 **06-NSObject.m** 文件，输入如下代码并且保存

```

#import <Foundation/Foundation.h>

#import "Cattle.h"
#import "Bull.h"
#import "MyNSObject.h"

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

    id cattle = [Cattle new];

```

```

id redBull = [Bull new];

SEL setLegsCount_SEL = @selector(setLegsCount:);

IMP cattle_setLegsCount_IMP = [cattle methodForSelector:setLegsCount_SE
L];

IMP redBull_setLegsCount_IMP = [redBull methodForSelector:setLegsCount_S
EL];


[cattle setLegsCount:4];

[redBull setLegsCount:4];

[redBull setSkinColor:@"red"];


Class cattle_class = cattle->isa;
MyClass my_cattle_class = cattle->isa;

SEL say = @selector(saySomething);

IMP cattle_sayFunc = [cattle methodForSelector:say];

cattle_sayFunc(cattle, say);


Class redBull_class = redBull->isa;
MyClass my_redBull_class = redBull->isa;


IMP redBull_sayFunc = [redBull methodForSelector:say];

redBull_sayFunc(redBull, say);


[pool drain];

return 0;
}

```

第五步，在 06-NSObject.m 文件的窗口的“[pool drain];”代码的左侧单击一下窗口的边框，确认一下是否出现一个蓝色的小棒棒，如果有的话那么断点被选择好了。如图 6-2 所示

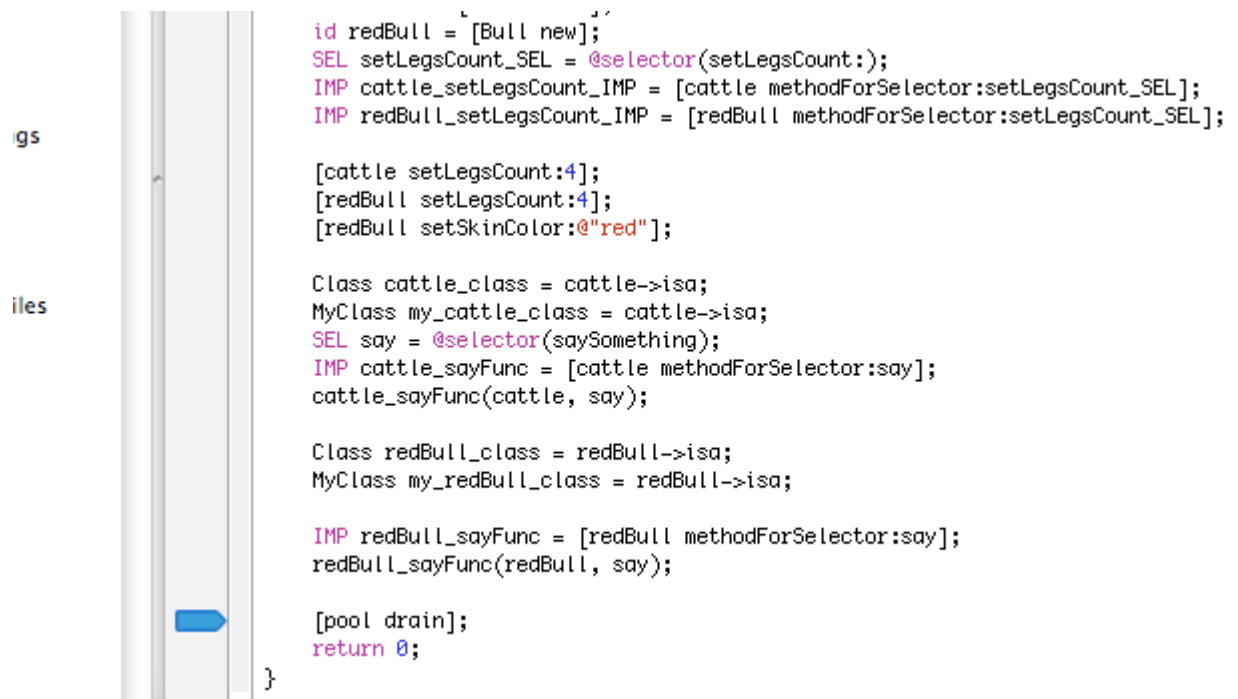


图 6-2，选择执行断点

第六步，选择 Xcode 上面的菜单的“Run”，然后选择“Debugger”，在 Debugger 窗口里面选择“Build and Go”。

好的，大家就停在这里，不要做其他的操作，我们把程序中断在程序几乎执行到最后的断点上，我们将要通过 Debugger 来看看 Objective-C 内部究竟发生了什么样的奇妙的魔法。

注意在从编译到执行的过程当中，会出现一些警告。由于本程序指示用来阐述一些 NSObject 内部的东西，所以请忽略掉这些警告。当然，我们在写自己的程序的时候，编译产生的警告一般是不能被忽略的。

6.3，超类方法的调用

我们现在打开“06-NSObject.m”文件，发现下面的代码：

```

SEL setLegsCount_SEL = @selector(setLegsCount:);

IMP cattle_setLegsCount_IMP = [cattle methodForSelector:setLegsCount_SEL];
IMP redBull_setLegsCount_IMP = [redBull methodForSelector:setLegsCount_SEL];

```

这一段代码，对同学们来说不是什么新鲜的内容了，我们在第 5 章里面已经讲过，这个是 SEL 和 IMP 的概念。我们在这里取得了 cattle 对象和 redBull 对象的 setLegsCount: 的函数指针。

如果大家现在已经不在 Debugger 里面的话，那么请选择 Xcode 菜单里面的，“Run”然后选择“Debugger”。

我们注意到在 Debugger 里面，cattle_setLegsCount_IMP 的地址和 redBull_setLegsCount_IMP 是完全一样的，如图 6-3 所示：

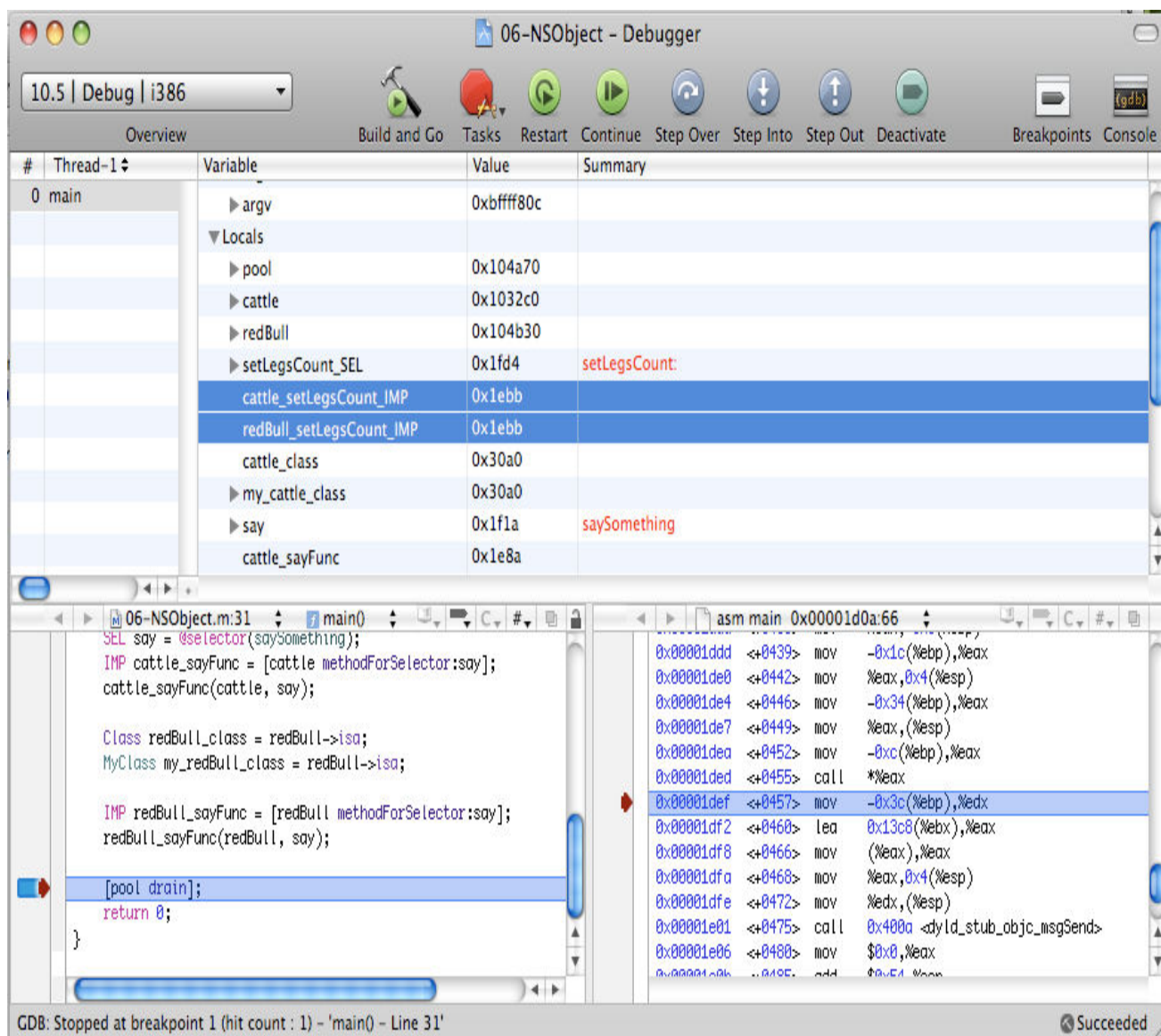


图 6-3，cattle_setLegsCount_IMP 和 redBull_setLegsCount_IMP 的地址。

注意由于环境和执行的时候的内存情况不同，所以同学们的电脑上显示的地址的数值可能和图 6-3 的数值不一样。

cattle_setLegsCount_IMP 和 redBull_setLegsCount_IMP 的地址完全一样，说明他们使用的是相同的代码段。这种结果是怎样产生的呢？大家请打开“MyNSObject.h”，参照下列代码：

```
struct my_objc_class {  
    MetaClass      class_pointer;  
};
```

```

struct my_objc_class*  super_class;

const char*           name;

long                  version;

unsigned long          info;

long                  instance_size;

struct objc_ivar_list* ivars;

struct objc_method_list* methods;

struct sarray *        dtable;

struct my_objc_class* subclass_list;

struct my_objc_class* sibling_class;

struct objc_protocol_list *protocols;

void* gc_object_type;

};

```

笔者在这里把开源代码的名字的定义加上了“my_”前缀，仅仅是为了区分一下。

“MyNSObject.h”里面的代码问题很多，笔者从来没有也不会在实际的代码里面使用这段代码，使用这些代码的主要目的是为了向大家讲解概念，请大家忽略掉代码里面的种种问题。

我们注意到这里的 **methods** 变量，里面包存的就是类的方法名字（SEL）定义，方法的指针地址(IMP)。当我们执行

```
IMP cattle_setLegsCount_IMP = [cattle methodForSelector:setLegsCount_SEL];
```

的时候，runtime 会通过 **dtable** 这个数组，快速的查找到我们需要的函数指针，查找函数的定义如下：

```

__inline__ IMP
objc_msg_lookup(id receiver, SEL op)
{
    if(receiver)

        return sarray_get(receiver->class_pointer->dtable, (idx)op);

    else

        return nil_method;

...

```

好的，现在我们的 **cattle_setLegsCount_IMP** 没有问题了，那么 **redBull_setLegsCount_IMP** 怎么办？在 **Bull** 类里面我们并没有定义实例方法 **setLegsCount:**，所以在 **Bull** 的 **Class** 里面，

runtime 难道找不到 `setLegsCount:` 么？答案是，是的 runtime 直接找不到，因为我们在 `Bull` 类里面根本就没有定义 `setLegsCount:`。

但是，从结果上来看很明显 runtime 聪明的找到了 `setLegsCount:` 的地址，runtime 是怎样找到的？答案就在：

```
struct my_objc_class*  super_class;
```

在自己的类里面没有找到的话，runtime 会去 `Bull` 类的超类 `cattle` 里面去寻找，庆幸的是它成功的在 `cattle` 类里面 runtime 找到了 `setLegsCount:` 的执行地址入口，所以我们得到了 `redBull_setLegsCount_IMP`。`redBull_setLegsCount_IMP` 和 `cattle_setLegsCount_IMP` 都是在 `Cattle` 类里面定义的，所以他们的代码的地址也是完全一样的。

我们现在假设，如果 runtime 在 `cattle` 里面也找不到 `setLegsCount:` 呢？没有关系，`cattle` 里面也有超类的，那就是 `NSObject`。所以 runtime 会去 `NSObject` 里面寻找。当然，`NSObject` 不会神奇到可以预测我们要定义 `setLegsCount:` 所以 runtime 是找不到的。

在这个时候，runtime 并没有放弃最后的努力，再没有找到对应的方法的时候，runtime 会向对象发送一个 `forwardInvocation:` 的消息，并且把原始的消息以及消息的参数打成一个 `NSInvocation` 的一个对象里面，作为 `forwardInvocation:` 的唯一的参数。

`forwardInvocation:` 本身是在 `NSObject` 里面定义的，如果你需要重载这个函数的话，那么任何试图向你的类发送一个没有定义的消息的话，你都可以在 `forwardInvocation:` 里面捕捉到，并且把消息送到某一个安全的地方，从而避免了系统报错。

笔者没有在本章代码中重写 `forwardInvocation:`，但是在重写 `forwardInvocation:` 的时候一定要注意避免消息的循环发送。比如说，同学们在 `A` 类对象的 `forwardInvocation` 里面，把 `A` 类不能响应的消息以及消息的参数发给 `B` 类的对象；同时在 `B` 类的 `forwardInvocation` 里面把 `B` 类不能响应的消息发给 `A` 类的时候，容易形成死循环。当然一个人写代码的时候不容易出现这个问题，当你在一个工作小组里面做的时候，如果你重写 `forwardInvocation:` 的时候，需要和小组的其他人达成共识，从而避免循环调用。

6.4，重载方法的调用

让我们继续关注“06-NSObject.m”文件，请大家参考一下下面的代码：

```
1 Class cattle_class = cattle->isa;
2 MyClass my_cattle_class = cattle->isa;
```

```

3 SEL say = @selector(saySomething);

4 IMP cattle_sayFunc = [cattle methodForSelector:say];

5 cattle_sayFunc(cattle, say);

6

7 Class redBull_class = redBull->isa;

8 MyClass my_redBull_class = redBull->isa;

9

10 IMP redBull_sayFunc = [redBull methodForSelector:say];

11 redBull_sayFunc(redBull, say);

```

本节的内容和 6.3 节的内容比较类似，关于代码部分笔者认为就不需要解释了，如果同学们有所不熟悉的话，可以参考一下第 5 章的内容。

在我们的 **Cattle** 类和 **Bull** 类里面，都有 **saySomething** 这个实例方法。我们知道只要方法的定义相同，那么它们的 **SEL** 是完全一样的。我们根据一个 **SEL say**，在 **cattle** 和 **redBull** 对象里面找到了他们的函数指针。根据 6.3 节的讲述，我们知道当 **runtime** 接收到寻找方法的时候，会首先在这个类里面寻找，寻找到了之后寻找的过程也就结束了，同时把这个方法的 **IMP** 返回给我们。所以，在上面的代码里面的 **cattle_sayFunc** 和 **redBull_sayFunc** 应该是不一样的，如图 6-4 所示：

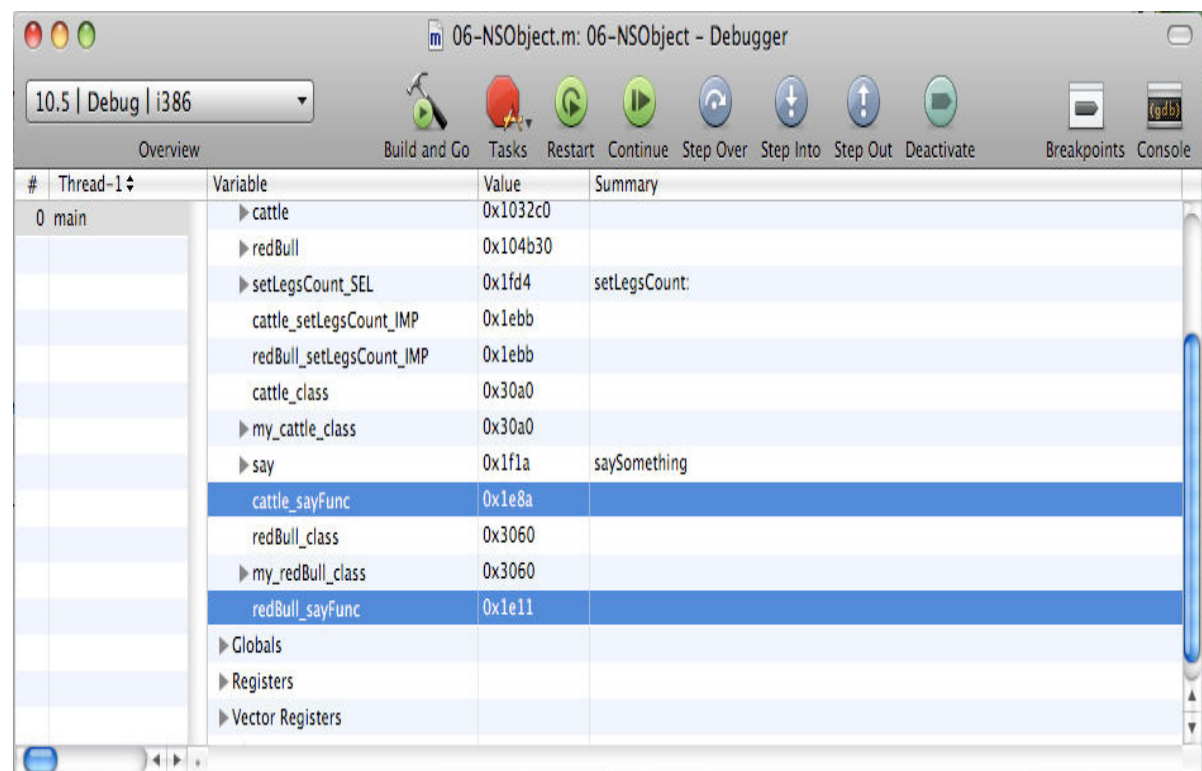


图 6-4， cattle_sayFunc 和 redBull_sayFunc 的地址

6.5，超类和子类中的 Class

在类进行内存分配的时候，对于一个类而言，runtime 需要找到这个类的超类，然后把超类的 Class 的指针的地址赋值给 isa 里面的 super_class。所以，我们的 cattle 里面的 Class 应该和 redBull 里面的 Class 里面的 super_class 应该是完全相同的，请参照图 6-5:

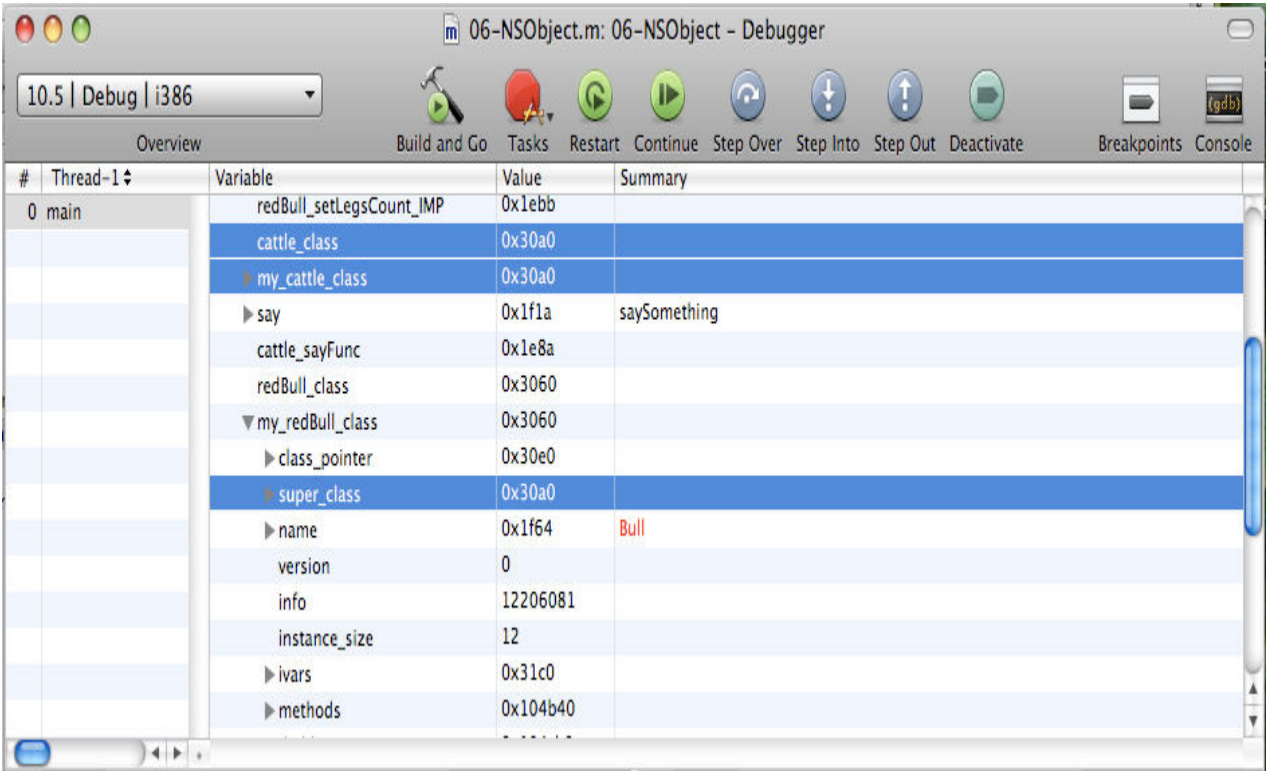


图 6-5， cattle 里面的 Class 和 redBull 里面的 Class 里面的 super_class

6.6，实例变量的内存分配的位置

我们先来回忆一下对象是怎样被创建的。创建对象的时候，类的内容需要被调入到内存当中我们称之为内存分配(Allocation)，然后需要把实体变量进行初始化(Initialization)，当这些步骤都结束了之后，我们的类就被实例化了，我们把实例化完成的类叫做对象(Object)。

对于内存分配的过程，runtime 需要知道分配多少内存还有各个实例变量的位置。我们回到“MyNSObject.h”，参照如下代码：

```
1 typedef struct objc_ivar* Ivar_t;  
2 typedef struct objc_ivar_list {
```

```

3     int    ivar_count;
4     struct objc_ivar {
5         const char* ivar_name;
6         const char* ivar_type;
7         int        ivar_offset;
8     } ivar_list[1];
9 } IvarList, *IvarList_t;

```

我们仔细看看第 5 行的 `ivar_name`，顾名思义这个是实例变量的名字，第 6 行的 `ivar_type` 是实例变量的类型，第 7 行的 `ivar_offset`，这个就是位置的定义。`runtime` 从类的 `isa` 里面取得了这些信息之后就知道了如何去分配内存。我们来看看图 6-6：

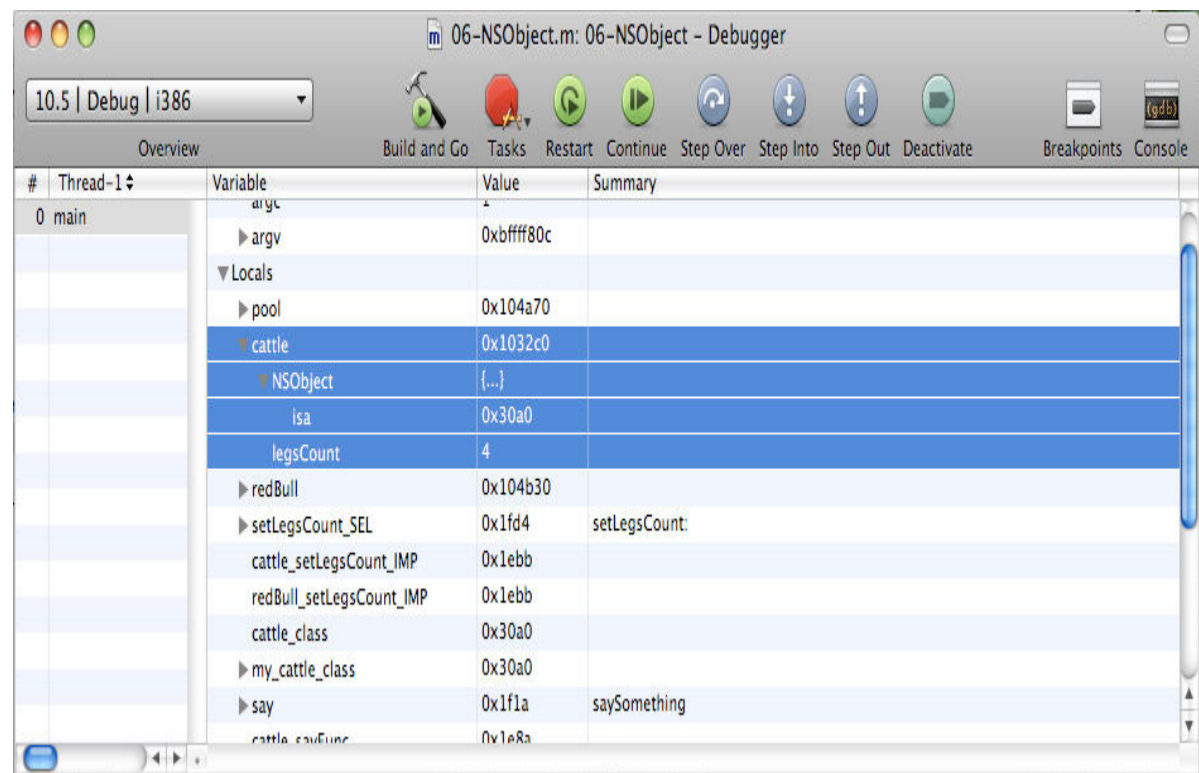


图 6-6，实例变量在内存中的位置

在 `cattle` 里面，我们看到了第一个实例变量是 `isa`，第二个就是我们定义的 `legsCount`。其中 `isa` 是超类的变量，`legsCount` 是 `Cattle` 类的变量。我们可以看出来，总是把超类的变量放在前头，然后是子类的变量。

那么对于 `redBull` 而言是什么样子呢？我们来看看图 6-7

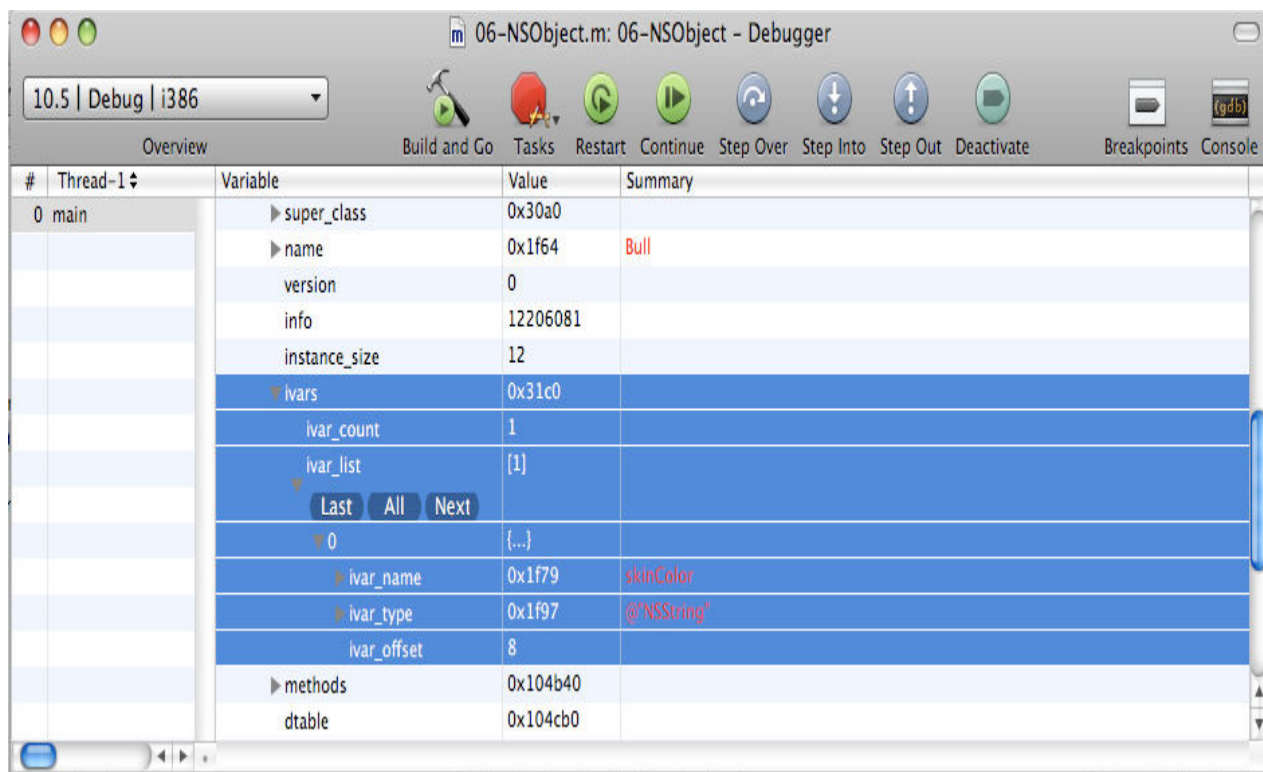


图 6-7，redBull 里面的实例变量的位置

我们通过图 6-7 可以发现 redBull 的 Class 里面的 skinColor 的位置偏移是 8，很明显，runtime 为 isa 和 legsCount 预留了 2 个位置。

6.7 本章总结

非常感谢大家！

在本章里面，笔者通过一个小小的“把戏”为同学们揭开了 NSObject 的神秘的面纱。本章的内容，虽然对理解 Objective-C 不是必需的，但是对以后的章节的内容的理解会有一个非常好的辅助作用，希望同学们花费一点点心思和时间阅读一下。

另外，笔者需要再次强调一下，由于笔者没有得到官方的正式的文档说明，所以不能保证本章的内容是完整而且准确的，本章内容仅供大家参考和娱乐，希望大家谅解。

Objective-C 2.0 with Cocoa Foundation--- 7，对象的初始化以及实例变量的作用域

7，对象的初始化以及实例变量的作用域

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，请点击[这里](#)。

到目前为止，我们都使用的是下列方式创建对象

```
[类名 new];
```

这种 **new** 的方式，实际上是一种简化的方式。笔者在这里总结一下前面几章里面曾经提到过关于创建对象的 2 个步骤：

第一步是为对象分配内存也就是我们所说的 **allocation**，**runtime** 会根据我们创建的类的信息来决定为对象分配多少内存。类的信息都保存在 **Class** 里面，**runtime** 读取 **Class** 的信息，知道了各个实例变量的类型，大小，以及他们的在内存里面的位置偏移，就会很容易的计算出需要的内存的大小。分配内存完成之后，实际上对象里面的 **isa** 也就被初始化了，**isa** 指向这个类的 **Class**。类里面的各个实例变量，包括他们的超类里面的实例变量的值都设定为零。

需要注意的是，分配内存的时候，不需要给方法分配内存的，在程序模块整体执行的时候方法部分就作为代码段的内容被放到了内存当中。对象的内容被放到了数据段当中，编译好的方法的汇编代码被放到了代码段当中。在 **Objective C** 里面，分配内存使用下列格式：

```
id 对象名=[类名 alloc];
```

NSObject 已经为我们提供了诸如计算内存空间大小以及初始化 **isa** 还有把各个实例变量清零，毫无疑问 **NSObject** 已经非常出色的完成了内存分配的工作，在一般情况下，我们不需要重写 **alloc** 方法。

第二步是要对内存进行初始化也就是我们所说的 **Initialization**。初始化指的是对实例变量的初始化。虽然在 **alloc** 方法里面已经把各个实例变量给清零了，但是在很多情况下，我们的实例变量不能是零（对于指针的实例变量而言，就是空指针）的，这样就需要我们对实例变量进行有意义的初始化。

按照 **Objective-C** 的约定，当初初始化的时候不需要参数的话，就直接使用 **init** 方法来初始化：

```
[对象名字 init];
```

init 是一个定义在 **NSObject** 里面的一个方法，**NSObject** 明显无法预测到派生类的实例变量是什么，所以同学们在自己的类里面需要重载一下 **init** 方法，在 **init** 方法里面把实例变量进行初始化。

但是，需要强调的是，由于某种原因我们的 **init** 也许失败了，比如说我们需要读取

CNBLOGS.COM 的某个 RSS，用这个 RSS 来初始化我们的对象，但是由于用户的网络连接失

败所以我们的 `init` 也许会失败，在手机应用当中的一些极端的情况下比如说有同学写一个读取网页内容的程序，在网页内容非常大的时候，那么 `alloc` 也有可能会失败，为了可以方便的捕获这些失败，所以我们在程序当中需要把上面的过程写在一起：

```
id 对象名 = [[类名 alloc] init];  
  
if (对象名)  
...  
  
else  
...
```

加上了上面的 `if` 语句我们的初始化过程就是完美的，当然我们有的时候不需要这个 `if` 语句。当我们的 `alloc` 和 `init` 永远不会失败的时候。关于初始化的时候的错误捕获，笔者将在后面的章节里面论述。

为了我们写程序方便和简洁，在创建一个从 `NSObject` 派生的类的对象的时候，苹果公司把 `alloc` 和 `init` 简化成为 `new`，我们在程序代码当中使用任何一种方式都是可以的，具体怎么写是同学们的喜好和自由。

到这里，有同学会问，如果我们的 `init` 需要参数怎么办？按照 `Objective-C` 的约定，我们需要使用 `initWith...`。也就是带参数的变量初始化，这个也是本章的主要内容。

本章在讲述 `initWith` 的同时，也将会顺便的给大家介绍一下实例变量的作用域。

7.1，本程序的执行结果

在本章里面，我们将要继续使用我们在第 4 章已经构筑好的类 `Cattle` 和 `Bull`。从一般的面向对象的角度上来说，是不鼓励我们改写已经生效的代码的。但是本章的目的是为了使同学们可以很好的理解主题，所以笔者在这里暂时违反一下规则改写了一下 `Cattle` 类，在里面追加了 `initWith` 方法，笔者也在 `Cattle` 类里面追加了一些实例变量为了阐述实例变量的作用域的问题。由于在 `Cattle` 类里面笔者追加了一些东西，所以在 `Bull` 类里面改写了 `saySomething` 这个函数，让我们的 `Bull` 可以说更多的内容。我们的 `redBull` 是这样说的：

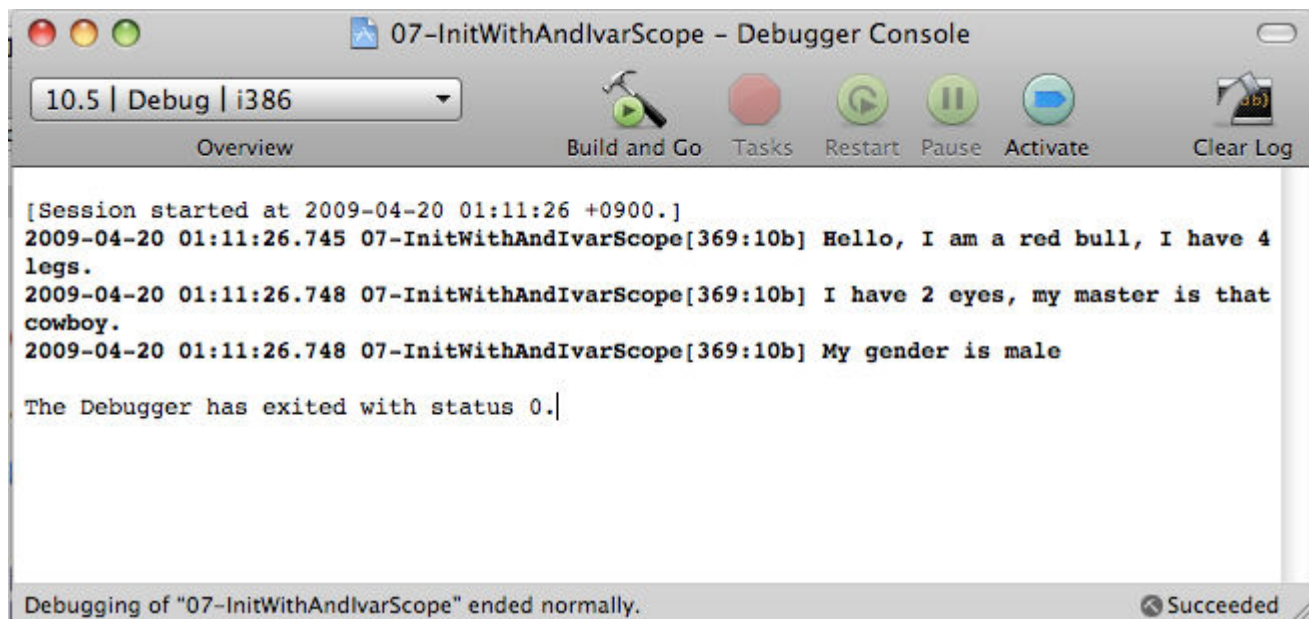


图 7-1，本程序的执行结果

本程序代码请点击[这里](#)下载。

再次强调在实际的编程过程中，尤其是写大型程序多人合作的时候，除非发现 BUG，否则不要改写已经生效的代码。这样会产生一些意想不到的结果，从而使其他的弟兄们或者姐妹们对你充满怨言。

7.2，实现步骤

第一步，按照我们在第 2 章所述的方法，新建一个项目，项目的名字叫做

07-InitWithAndIvarScope。如果你是第一次看本篇文章，请到这里参看第二章的内容。

第二步，按照我们在第 4 章的 4.2 节的第二，三，四步所述的方法，把在第 4 章已经使用过的“Cattle.h”，“Cattle.m”，“Bull.h”还有“Bull.m”，导入本章的项目里面。然后把第 6 章里面的“MyNSObject.h”也导入到项目当中。

第三步，打开“Cattle.h”，修改成为下面的代码并且保存：

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

@interface Cattle : NSObject {

    int legsCount;

private
```

```

    bool gender;    //male = YES female = NO

    @protected

    int eyesCount;

    @public

    NSString *masterName;
}

- (void) saySomething;

- (void) setLegsCount:(int) count;

- (id) initWithLegsCount:(int) theLegsCount

                gender:(bool) theGender

                eyesCount:(int) theEyesCount

                masterName:(NSString*) theMasterName;

@end

```

第 4 步，打开“Cattle.m”，修改成下面的代码并且保存：

```

#import "Cattle.h"

@implementation Cattle

- (void) saySomething
{
    NSLog(@"Hello, I am a cattle, I have %d legs.", legsCount);
}

- (void) setLegsCount:(int) count
{
    legsCount = count;
}

- (id) init
{
    [super init];

    return [self initWithLegsCount:4

                                gender:YES

                                eyesCount:2

```

```

        masterName:@"somebody"];
}

- (id)initWithLegsCount:(int) theLegsCount
        gender:(bool) theGender
        eyesCount:(int) theEyesCount
        masterName:(NSString*)theMasterName
{
    legsCount = theLegsCount;
    gender = theGender;
    eyesCount = theEyesCount;
    masterName = theMasterName;
    return self;
}

@end

```

第五步，打开“Bull.m”， ， 修改成下面的代码并且保存：

```

#import "Bull.h"

@implementation Bull

-(void) saySomething
{
    NSLog(@"Hello, I am a %@ bull, I have %d legs.", [self getSkinColor], legsCount);

    NSLog(@"I have %d eyes, my master is %@.", eyesCount, masterName);

    //List below is illegal

    //NSLog(@"My gender is %@", gender ? @"male" : @"female");
}

-(NSString*) getSkinColor
{
    return skinColor;
}

- (void) setSkinColor:(NSString *) color

```

```
{  
  
    skinColor = color;  
  
}  
  
@end
```

第六步，打开“07-InitWithAndIvarScope.m”，修改成下面的代码并且保存：

```
#import <Foundation/Foundation.h>  
  
#import "Bull.h"  
#import "Cattle.h"  
#import "MyNSObject.h"  
  
int main (int argc, const char * argv[]) {  
  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
    Bull *redBull = [[Bull alloc] initWithLegsCount:4  
  
                                gender:YES  
  
                                eyesCount:2  
  
                                masterName:@"that cowboy"];  
  
    [redBull setSkinColor:@"red"];  
    [redBull saySomething];  
  
    //legal, but not good  
    redBull->masterName = @"that cowgirl";  
    //legal, but bad  
    //redBull->eyesCount = 3;  
  
    //Trying to access a private ivar, VERY bad thing  
    //MyClass bullClass = redBull->isa;  
    bool *redBullGender = (bool *) (redBull) + 8;  
  
    NSLog(@"My gender is %@", *redBullGender ? @"male" : @"female");  
  
    [pool drain];  
}
```

```
    return 0;
}
```

第七步，选择屏幕上方菜单里面的“Run”，然后选择“Console”，打开了 Console 对话框之后，选择对话框上部中央的“Build and Go”，如果不出什么意外的话，那么应该出现入图 7-1 所示的结果。如果出现了什么意外导致错误的话，那么请仔细检查一下你的代码。如果经过仔细检查发现 还是不能执行的话，可以到这里下载笔者为同学们准备的代码。如果笔者的代码还是不能执行的话，请告知笔者。

7.3，实例变量的作用域(Scope)

对于 Objective-C 里面的类的实例变量而言，在编译器的范围里面，是有作用域的。和其他的语言一样，Objective-C 也支持 public, private 还有 protected 作用域限定。

如果一个实例变量没有任何的作用域限定的话，那么缺省就是 protected。

如果一个实例变量适用于 public 作用域限定，那么这个实例变量对于这个类的派生类，还有类外的访问都是允许的。

如果一个实例变量适用于 private 作用域限定，那么仅仅在这个类里面才可以访问这个变量。

如果一个实例变量适用于 protected 作用域限定，那么在这个类里面和这个类的派生类里面可以访问这个变量，在类外的访问是不推荐的。

我们来看看“Cattle.h”的代码片断：

```
1    int legsCount;
2    @private
3    bool gender;    //male = YES female = NO
4    @protected
5    int eyesCount;
6    @public
7    NSString *masterName;
```

第一行的 legsCount 的前面没有任何作用域限定，那么它就是 protected 的。

第二行是在说从第二行开始的实例变量的定义为 private 的，和其他的关键字一样，Objective-C 使用@来进行编译导向。

第三行的 gender 的作用域限定是 private 的，所以它适用于 private 作用域限定。

第四行是在说从第四行开始的实例变量的定义为 **protected** 的，同时第二行的 **private** 的声明作废。

第五行的 **eyesCount** 的作用域限定是 **protected** 的，所以它适用于 **protected** 作用域限定。

第六行是再说从第六行开始的实例变量的定义为 **public** 的，同时第四行的 **protected** 的声明作废。

第七行的 **masterName** 的作用域限定是 **public** 的，所以它适用于 **public** 作用域限定。

我们再来看看在派生类当中，**private**，**protected** 还有 **public** 的表现。**Bull** 类继承了 **Cattle** 类，笔者改写了一下“**Bull.m**”用来说明作用域的问题，请参看下面的代码：

```
1 -(void) saySomething
2 {
3     NSLog(@"Hello, I am a %@ bull, I have %d legs.", [self getSkinColor], legsCount);
4     NSLog(@"I have %d eyes, my master is %@.", eyesCount, masterName);
5     //List below is illegal
6     //NSLog(@"My gender is %@", gender ? @"male" : @"female");
7 }
```

我们来看看第 3 还有第 4 行代码，我们可以访问 **legsCount**，**eyesCount** 还有 **masterName**。

在第 6 行代码当中，我们试图访问 **gender** 这个 **Cattle** 的私有(**private**)属性，这行代码产生了编译错误，所以我们不得不注释掉第 6 行代码。

好的，我们再来看看类的外部 **private**，**protected** 还有 **public** 的表现。请同学们打开

“07-InitWithAndIvarScope.m”，参考一下下面的代码：

```
1     //legal, but not good
2     redBull->masterName = @"that cowgirl";
3     //legal, but bad
4     //redBull->eyesCount = 3;
5
6     //Trying to access a private ivar, VERY bad thing
7     //MyClass bullClass = redBull->isa;
8     bool *redBullGender = (bool *) (redBull) + 8;
9     NSLog(@"My gender is %@", *redBullGender ? @"male" : @"female");
```

在第二行里面，我们访问了 `masterName`，由于在 `Cattle` 里面 `masterName` 是 `public` 的，`Bull` 继承了 `Cattle`，所以我们可以直接访问 `masterName`。但是这不是一种好的习惯，因为这不面向对象的基本思想。实际上，如果没有特殊的理由，我们不需要使用 `public` 的。

第四行，我们试图在类的外边访问 `protected` 变量 `eyesCount`，在这里笔者的 `Xcode` 只是轻轻的给了一个警告，编译成功并且可以运行。同样，这种在类的外边访问类的 `protected` 变量是一个很糟糕的做法。

我们还记得在 `Bull` 的 `saySomething` 里面我们曾经试图访问过 `gender`，但是编译器无情的阻止了我们，因为 `gender` 是私有的。但是，这仅仅是编译器阻止了我们，当我们有足够的理由需要在类的外边访问 `private` 实例变量的时候，我们还是可以通过一些强硬的方法合法的访问私有变量的，我们的方法就是使用指针偏移。

我们首先回忆一下第 6 章 的 6.6 节的内容，`isa` 里面保存了对象里面的实例变量相对于对象首地址的偏移量，我们得到了这个偏移量之后就可以根据对象的地址来获得我们所需要的实例变量的地址。在正常情况下，我们需要通过访问类本身和它的超类的 `ivars` 来获得偏移量的，但是笔者在这里偷了一个懒，先使用第七行的代码 `MyClass bullClass = redBull->isa;` 通过 `Debugger` 获得 `gender` 的偏移量，数值为 8。然后在第 8 行里面，笔者通过使用指针偏移取得了 `gender` 的指针然后在第 9 行实现了输出。

由此可见，在 `Objective-C` 里面，所谓的 `private` 还有 `protected` 只是一个 `Objective-C` 强烈推荐的一个规则，我们需要按照这个规则来编写代码，但是如果我们违反了 this 规则，编译器没有任何方法阻止我们。

笔者认为在类的外部直接访问任何实例变量，不管这个实例变量是 `public`，`private` 还是 `protected` 都是一个糟糕的做法，这样会明显的破坏封装的效果，尽管这样对编译器来说是合法的。

7.4, initWith...

`NSObject` 为我们准备的不带任何参数的 `init`，我们的类里面没有实例变量，或者实例变量可以都是零的时候，我们可以使用 `NSObject` 为我们准备的缺省的 `init`。当我们的实例变量不能为零，并且这些实例变量的初始值可以在类的初始化的时候就可以确定的话，我们可以重写 `init`，并且在里面为实例变量初始化。

但是在很多时候，我们无法预测类的初始化的时候的实例变量的初始值，同时 `NSObject` 明显无法预测到我们需要什么样的初始值，所以我们需要自己初始化类的实例变量。

请同学们打开“Cattle.m”，我们参考一下下面的代码：

```
1 -(id)init
2 {
3     [super init];
4     return [self initWithLegsCount:4
5             gender:YES
6             eyesCount:2
7             masterName:@"somebody"];
8 }
9 - (id)initWithLegsCount:(int) theLegsCount
10    gender:(bool) theGender
11    eyesCount:(int) theEyesCount
12    masterName:(NSString*)theMasterName
13 {
14     legsCount = theLegsCount;
15     gender = theGender;
16     eyesCount = theEyesCount;
17     masterName = theMasterName;
18     return self;
19 }
```

从第 3 行到第 7 行，笔者重写了一下 `init`。在 `init` 里面，笔者给通过调用 `initWith`，给类的各个实例变量加上了初始值。这样写是很必要的，因为将来的某个时候，也许有人（或者是自己）很冒失的使用 `init` 来初始化对象，然后就尝试使用这个对象，如果我们没有重写 `init`，那么也许会出现一些意想不到的事情。

从第 9 行到第 19 行，是我们自己定义的 `initWith`，代码比较简单，笔者就不在这里赘述了。需要注意的一点是，笔者没有在这里调用 `[super init];`。原因是 `Cattle` 的超类就是 `NSObject`，初始化的过程就是初始化实例变量的过程，`runtime` 已经为我们初始化好了 `NSObject` 的唯一实例变量 `isa`，也就是 `Cattle` 的类的信息，所以我们不需要调用 `[super init];`。在某些时候，超类的变量需要初始化的时候，请同学们在子类的 `init` 或者 `initWith` 里面调用 `[super init];`。

请同学们再次打开“07-InitWithAndIvarScope.m”，参考下面的代码片断：

```
1    Bull *redBull = [[Bull alloc] initWithLegsCount:4
2
3                                gender:YES
4                                eyesCount:2
5                                masterName:@"that cowboy"];
6    [redBull setSkinColor:@"red"];
7    [redBull saySomething];
```

从第 1 行到第 4 行就是调用的 `initWith` 来初始化我们的 `redBull`。

7.5, 本章总结

非常感谢大家对笔者的支持！

我们在本章里面介绍了 2 个比较轻松的话题，一个是实例变量的作用域，这个概念笔者个人认为对有一点面向对象编程经验的人来说，不是什么新鲜的概念了。但是需要注意的是，**Objective-C** 并没有强制我们遵守它的规则，他仍旧为我们提供了违反规则的机会，这一点上根 **C++** 比较类似。只要支持指针，就无法避免使用者违反规则。事务都是一分为二的，当我们得到了访问任何变量的自由之后，我们必须为访问这些变量承担后果。

第二个话题就是 `initWith`。和其他的面向对象的语言不同，**Objective-C** 没有构造函数，它通过 `init` 还有 `initWith` 来初始化变量，我们应该根据具体情况进行具体的分析，从而编写我们的 `init` 还有 `initWith` 方法。

Objective-C 2.0 with Cocoa Foundation--- 8, 类方法以及私有方法

8, 类方法以及私有方法

本系列讲座有着很强的前后相关性，如果你是第一次阅读本篇文章，为了更好的理解本章内容，笔者建议你最好从本系列讲座的第 1 章开始阅读，请点击[这里](#)。

Objective-C 里面区别于实例方法，和 **Java** 或者 **C++** 一样，也支持类方法。类方法(Class Method) 有时被称为工厂方法(Factory Method)或者方便方法(Convenience method)。工厂方法的称谓明显和一般意义上的工厂方法不同，从本质上来说，类方法可以独立于对象而执行，所以在其他的语言里面类方法有的时候被称为静态方法。就像 `@interface` 曾经给我们带来的混乱一样，现在我们就不要去追究和争论工厂方法的问题了，我们看到 **Objective-C** 的文章说工厂方法，就把它当作类方法好了。

在 **Objective-C** 里面，最受大家欢迎的类方法应该是 **alloc**，我们需要使用 **alloc** 来为我们的对象分配内存。可以想象，如果没有 **alloc**，我们将要如何来为我们的类分配内存！

和其他的语言类似，下面是类方法的一些规则，请大家务必记住。

- 1，类方法可以调用类方法。
- 2，类方法不可以调用实例方法，但是类方法可以通过创建对象来访问实例方法。
- 3，类方法不可以使用实例变量。类方法可以使用 **self**，因为 **self** 不是实例变量。
- 4，类方法作为消息，可以被发送到类或者对象里面去（实际上，就是可以通过类或者对象调用类方法的意思）。

如果大家观察一下 **Cocoa** 的类库，会发现类方法被大量的应用于方便的对象创建和操作对象的，考虑到类方法的上述的特性，同学们在设计自己的类的时候，为了谋求这种方便，可以考虑使用类方法来创建或者操作对象。笔者认为，这个就是类方法的潜规则，在本章的范例程序里面，笔者将要遵守这个潜规则。

在上一章我们讲了一下实例变量的作用域，实例变量的作用域的方式和其他面向对象的语言没有什么不同。对于方法，非常遗憾的是，**Objective-C** 并没有为我们提供诸如 **public**，**private** 和 **protected** 这样的限定，这就意味着在 **Objective-C** 里面，从理论上来说所有的方法都是公有的。但是，我们可以利用 **Objective-C** 的语言的特性，我们自己来实现方法的私有化。当然我们自己的私有化手段没有得到任何的编译器的支持，只是告诉使用者：“这是一个私有的方法，请不要使用这个方法”。所以，无论作为类的设计者和使用者都应该清楚在 **Objective-C** 里面的方法私有化的所有手段，这样就在类的设计者和使用者之间达成了一种默契，这种方式明显不是 **Objective-C** 语法所硬性规定的，所以也可以把这种手法成为一种潜规则。

关于潜规则经常看英文文档的同学，应该可以遇到这样一个词，**de facto standard**，也就是笔者所说的潜规则。

本章所述的方法的私有化是一种有缺陷的手段，有一定的风险而且也没有完全实现私有化，在后面的章节里面笔者会陆续的给出其他的实现方法私有化的方法。

另外，**Objective-C** 里面有一个其他不支持指针的语言没有的一个动态特性，那就是程序在运行的时候，可以动态的替换类的手段。动态的方法替换有很多种应用，本章实现了一个类似 **java** 里面的 **final** 函数。和 **final** 函数不同的是，如果子类重写了这个方法，编译器不会报错，但是执行的时候总是执行的你的超类的方法。

类方法，方法私有化和动态方法替换将是本章的主题。

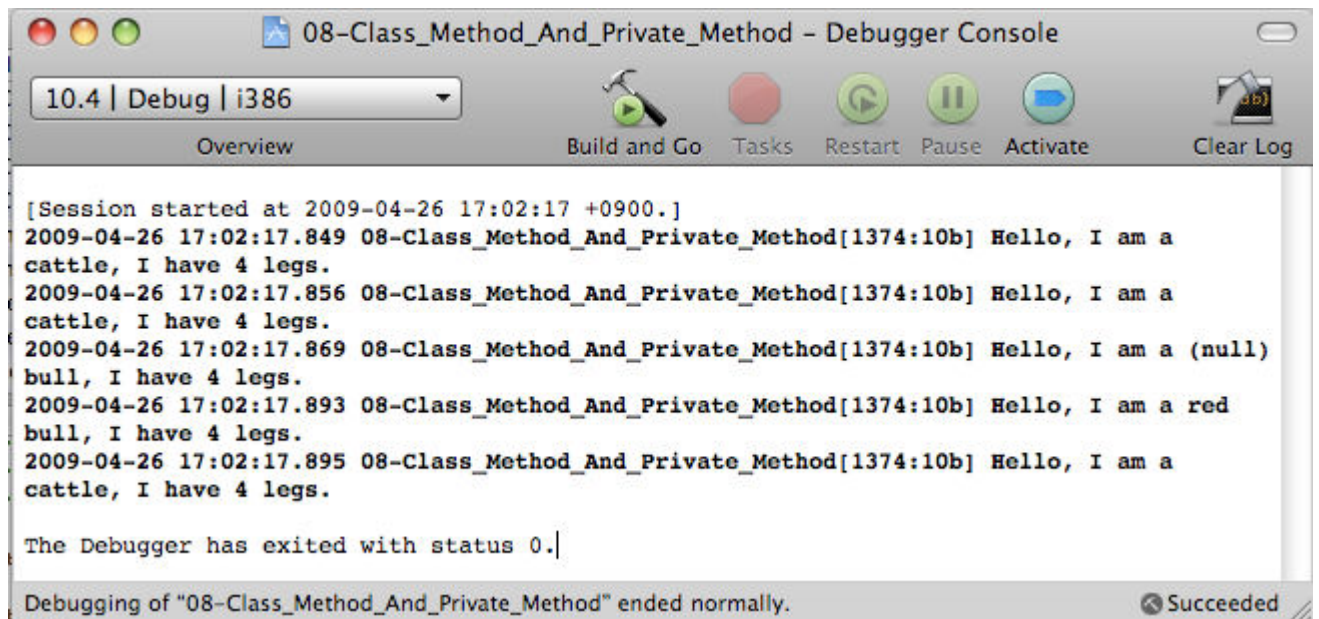
8.1，本程序的执行结果

在本章里面，我们将要继续使用我们在第 4 章已经构筑好的类 `Cattle` 和 `Bull`。

笔者在这里暂时违反一下不修改已经生效的代码规则改写了一下 `Cattle` 和 `Bull` 类，在里面追加了一些类方法，用于创建 `Cattle` 系列的对象。

笔者也改写了 `Cattle` 的头文件用来实现方法的私有化。

面向对象的程序有一个很大的特色就是动态性，但是由于某种原因我们在设计超类的时候，也许会考虑把某个方法设定成为静态的，这样就有了诸如 `final` 的概念。在本章我们将要使用动态的方法替换来实现这个功能。我们将要构筑一个新类，名字叫做 `UnknownBull`，我们使用动态方法替换导致即使 `UnknownBull` 重载了 `Cattle` 类的 `saySomething`，但是向 `UnknownBull` 发送 `saySomething` 的时候，仍然执行的是 `Cattle` 的 `saySomething`。本程序的执行结果请参照下图：



```
[Session started at 2009-04-26 17:02:17 +0900.]
2009-04-26 17:02:17.849 08-Class_Method_And_Private_Method[1374:10b] Hello, I am a
cattle, I have 4 legs.
2009-04-26 17:02:17.856 08-Class_Method_And_Private_Method[1374:10b] Hello, I am a
cattle, I have 4 legs.
2009-04-26 17:02:17.869 08-Class_Method_And_Private_Method[1374:10b] Hello, I am a (null)
bull, I have 4 legs.
2009-04-26 17:02:17.893 08-Class_Method_And_Private_Method[1374:10b] Hello, I am a red
bull, I have 4 legs.
2009-04-26 17:02:17.895 08-Class_Method_And_Private_Method[1374:10b] Hello, I am a
cattle, I have 4 legs.

The Debugger has exited with status 0.

Debugging of "08-Class_Method_And_Private_Method" ended normally. Succeeded
```

图 8-1，本程序的执行结果。

本程序可以点击[这里](#)下载。

8.2，实现步骤

第一步，按照我们在第 2 章所述的方法，新建一个项目，项目的名字叫做

`07-InitWithAndIvarScope`。如果你是第一次看本篇文章，请到这里参看第二章的内容。

第二步，按照我们在第 4 章的 4.2 节的第二，三，四步所述的方法，把在第 4 章已经使用过的“Cattle.h”，“Cattle.m”，“Bull.h”还有“Bull.m”， 导入本章的项目里面。

第三步，打开“Cattle.h”和“Cattle.m”，分别修改成为下面的代码并且保存：

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

@interface Cattle : NSObject {

    int legsCount;

}

- (void) saySomething;

+ (id) cattleWithLegsCountVersionA:(int) count;
+ (id) cattleWithLegsCountVersionB:(int) count;
+ (id) cattleWithLegsCountVersionC:(int) count;
+ (id) cattleWithLegsCountVersionD:(int) count;

@end

#import "Cattle.h"

#import <objc/objc-class.h>

@implementation Cattle

- (void) saySomething

{

    NSLog(@"Hello, I am a cattle, I have %d legs.", legsCount);

}

- (void) setLegsCount:(int) count

{

    legsCount = count;

}

+ (id) cattleWithLegsCountVersionA:(int) count

{

    id ret = [[Cattle alloc] init];

    //NEVER DO LIKE BELOW

    //legsCount = count;
```

```

        [ret setLegsCount:count];

        return [ret autorelease];
    }

+ (id) cattleWithLegsCountVersionB:(int) count
{
    id ret = [[[Cattle alloc] init] autorelease];

    [ret setLegsCount:count];

    return ret;
}

+ (id) cattleWithLegsCountVersionC:(int) count
{
    id ret = [[self alloc] init];

    [ret setLegsCount:count];

    return [ret autorelease];
}

+ (id) cattleWithLegsCountVersionD:(int) count
{
    id ret = [[self alloc] init];

    [ret setLegsCount:count];

    if([self class] == [Cattle class])
        return [ret autorelease];

    SEL sayName = @selector(saySomething);

    Method unknownSubClassSaySomething = class_getInstanceMethod([self class], sayName);

    //Change the subclass method is RUDE!

    Method cattleSaySomething = class_getInstanceMethod([Cattle class], sayName);

    //method_imp is deprecated since 10.5

    unknownSubClassSaySomething->method_imp = cattleSaySomething->method_imp;
}

```

```

        return [ret autorelease];
    }
@end

```

第四步，打开“Bull.h”和“Bull.m”，分别修改成为下面的代码并且保存：

```

#import <Foundation/Foundation.h>

#import "Cattle.h"

@interface Bull : Cattle {
    NSString *skinColor;
}

- (void) saySomething;
- (NSString*) getSkinColor;
- (void) setSkinColor:(NSString *) color;
+ (id) bullWithLegsCount:(int) count bullSkinColor:(NSString*) theColor;
@end

```

```

#import "Bull.h"

@implementation Bull

- (void) saySomething
{
    NSLog(@"Hello, I am a %@ bull, I have %d legs.", [self getSkinColor], legsCount);
}

- (NSString*) getSkinColor
{
    return skinColor;
}

- (void) setSkinColor:(NSString *) color
{
    skinColor = color;
}

```

```

}

+ (id) bullWithLegsCount:(int) count bullSkinColor:(NSString*) theColor
{
    id ret = [self cattleWithLegsCountVersionC:count];
    [ret setSkinColor:theColor];
    //DO NOT USE autorelease here!
    return ret;
}

@end

```

第五步，创建一个新类，名字叫做“UnknownBull”，然后分别打开“UnknownBull.h”和“UnknownBull.m”，分别修改成为下面的代码并且保存：

```

#import <Foundation/Foundation.h>

#import "Bull.h"

@interface UnknownBull : Bull {

}

- (void) saySomething;

@end

```

```

#import "UnknownBull.h"

@implementation UnknownBull

- (void) saySomething
{
    NSLog(@"Hello, I am an unknown bull.");
}

@end

```

第六步，打开“08-Class_Method_And_Private_Method.m”，修改成为下面的样子并且保存


```

#import <Foundation/Foundation.h>

#import "Cattle.h"

#import "Bull.h"

#import "UnknownBull.h"

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

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

    id cattle[5];

    cattle[0] = [Cattle cattleWithLegsCountVersionA:4];
    cattle[1] = [Bull cattleWithLegsCountVersionB:4];
    cattle[2] = [Bull cattleWithLegsCountVersionC:4];
    cattle[3] = [Bull bullWithLegsCount:4 bullSkinColor:@"red"];
    cattle[4] = [UnknownBull cattleWithLegsCountVersionD:4];

    for(int i = 0 ; i < 5 ; i++)
    {
        [cattle[i] saySomething];
    }

    [pool drain];

    return 0;
}

```

第七步，选择屏幕上方菜单里面的“Run”，然后选择“Console”，打开了 Console 对话框之后，选择对话框上部中央的“Build and Go”，如果不出什么意外的话，那么应该出现入图 8-1 所示的结果。如果出现了什么意外导致错误的话，那么请仔细检查一下你的代码。如果经过仔细检查发现 还是不能执行的话，可以到这里下载笔者为同学们准备的代码。如果笔者的代码还是不能执行的话，请告知笔者。

8.2，方法的私有化

在讲述方法私有化之前，我们首先要提到一个 Objective-C 里面的一个概念，动态类型和静态类型。

所谓的动态类型，就是使用 id 来定义一个对象，比如说

```
id cattle = [[Cattle alloc] init];
```

所谓的静态类型，就是使用已知变量的的类型来定义对象，比如说

```
Cattle cattle = [[Cattle alloc] init];
```

动态类型和静态类型各有好处，动态类型实现了多态性，使用静态类型的时候编译器会为你检查一下也许会出现危险的地方，比如说向一个静态类型的对象发送一个它没有定义的消息等等。

好的，我们现在打开“cattle.h”，大家可以发现，和以前的版本相比，我们的“cattle.h”少了一个方法的定义，那就是`-(void) setLegsCount:(int) count;`。笔者在本章的范例程序里面实现私有方法的手段比较简单，直接把`-(void) setLegsCount:(int) count`从“cattle.h”给删除掉了。

大家打开“cattle.m”，可以看到里面`-(void) setLegsCount:(int) count`是有实现部分的。实现部分和过去的版本没有任何区别的。

我们本章里面讲述的实现方法私有化的手段，就是从头文件当中不写方法的声明。这样做会导致如下几个现象

- 1，在类的实现文件.m 里面，你可以向平常一样使用`[self setLegsCount:4]` 来发送消息，但是确省设定的编译器会很不礼貌的给你一个警告。
- 2，你可以向 `Cattle` 以及从 `Cattle` 继承的类的静态对象发送 `setLegsCount:4` 的消息，但是同样，确省设定的编译器会很不礼貌的给你一个警告。
- 3，你可以向 `Cattle` 以及从 `Cattle` 继承的类的动态对象发送 `setLegsCount:4` 的消息，编译器不会向你发送任何警告的。

说到这里，同学们也许会觉得这一节的方法私有化有一点奇怪，因为在上面的第二条里面，不能阻止对对象的私有方法进行调用。令我们更为恼火的是，居然在我们自己的类的实现文件里面需要调用的时候产生诸如第一条的警告！

让我们冷静一下。

我们说，在面向对象的程序里面，一般而言类的使用者只关心接口，不关心实现的。当我们类的实现部分的某个方法，在头文件里面没有定义的话，那么由于我们的类的使用者只是看头文件，所以他不应该用我们定义的所谓的私有方法的。这一点，对于其他的语言来说也是一样的，其他的语言的私有方法和变量，如果我们把它们改为 `public`，或者我们不修改头文件，使用指针也可以强行的访问到私有的变量和方法的，从这个角度上来说，私有化的方法和变量也只不过是一个摆设而已，没有人可以阻止我们去访问他们，探求埋藏在里面的奥秘。所谓的私有化只不过是一个潜规则而已，在正常的时候，我们大家都会遵守这个潜规则的。但是被逼无奈走投无路的

时候我们也许会除了访问私有的东西无可选择。但是也不能过分，我们显然不可以把访问私有变量和函数当作一种乐趣。

说到这里，我想大家应该可以理解这种私有化方法的定义了。它只不过是一种信号，告诉类的使用者，“这是一个私有的函数，请不要使用它，否则后果自负”。我们在看到别人的代码的时候看到了这种写法的时候，或者别人看到我们的代码的时候，大家都需要做到相互理解对方的隐藏私有部分的意图。还是还是这句话，在大多数时候，请不要破坏潜规则。

8.3， 类方法

我们现在转到本章最重要的主题，类方法。我们将要首先关注一下类方法的声明，现在请同学们打开"Cattle.h"文件，可以发现下面的代码：

```
1 + (id) cattleWithLegsCountVersionA:(int) count;
2 + (id) cattleWithLegsCountVersionB:(int) count;
3 + (id) cattleWithLegsCountVersionC:(int) count;
4 + (id) cattleWithLegsCountVersionD:(int) count;
```

类方法和实例方法在声明上的唯一的区别就是，以加号+为开始，其余的部分是完全一致的。笔者在这里定义了 4 个不同版本的类方法，从功能上来说都是用来返回 Cattle 类或者其子类的对象的，其中 cattleWithLegsCountVersionA 到 C 是我们这一节讲解的重点。

让我们首先打开"Cattle.m"，关注一下下面的代码：

```
1 + (id) cattleWithLegsCountVersionA:(int) count
2 {
3     id ret = [[Cattle alloc] init];
4     //NEVER DO LIKE BELOW
5     //legsCount = count;
6     [ret setLegsCount:count];
7     return [ret autorelease];
8 }
9 + (id) cattleWithLegsCountVersionB:(int) count
10 {
11     id ret = [[[Cattle alloc] init] autorelease];
12     [ret setLegsCount:count];
```

```
13     return ret;
14 }
```

我们需要使用类方法创建对象，所以在第 3 行，我们使用了我们比较熟悉的对象的创建的方法创建了一个对象。大家注意一下第 5 行，由于类方法是和对象是脱离的所以我们是无法在类方法里面使用实例变量的。第 6 行，由于我们创建了对象 **ret**，所以我们可以向 **ret** 发送 **setLegsCount**:这个消息，我们通过这个消息，设定了 **Cattle** 的 **legsCount** 实例变量。在第 7 行，我们遇到了一个新的朋友，**autorelease**。我们在类方法里面创建了一个对象，当我们返回了这个对象之后，类方法也随之结束，类方法结束就意味着在我们写的类方法里面，我们失去了对这个对象的参照，也就永远无法在类方法里面控制这个对象了。在 **Objective-C** 里面有一个规则，就是谁创建的对象，那么谁就有负责管理这个对象的责任，类方法结束之后，除非和类的使用者商量好了让类的使用者释放内存，否则我们无法直接的控制这个过程。

记忆力好的同学应该可以回忆起来，笔者曾经在第二章提到过一种延迟释放内存的技术，这个就是 **autorelease**。关于 **autorelease** 以及其他的内存管理方法，我们将在下一章放到一起讲解。到这里大家记住，使用类方法的潜规则是你要使用类方法操作对象，当你需要使用类方法创建一个对象的时候，那么请在类方法里面加上 **autorelease**。

我们来看看 **cattleWithLegsCountVersionB** 的实现部分的代码，和 **cattleWithLegsCountVersionA** 唯一区别就是我们在创建的时候就直接的加上了 **autorelease**。这样符合创建对象的时候“一口气”的把所有需要的方法都写到一起的习惯，采取什么方式取决于个人喜好。

我们再打开“08-Class_Method_And_Private_Method.m”，参看下面的代码

```
1     cattle[0] = [Cattle cattleWithLegsCountVersionA:4];
2     cattle[1] = [Bull cattleWithLegsCountVersionB:4];
```

我们在回头看看本程序的执行结果，心细的同学也许发现了一个很严重的问题，我们在第 2 行代码里面想要返回一个 **Bull** 的对象，但是输出的时候却变成了 **Cattle**，原因就是我们在 **cattleWithLegsCountVersionB** 里面创建对象的时候，使用了 **id ret = [[[Cattle alloc] init] autorelease]**。由于 **Bull** 里面没有重写 **cattleWithLegsCountVersionB**，所以除非我们重写 **cattleWithLegsCountVersionB** 否则我们向 **Bull** 发送 **cattleWithLegsCountVersionB** 这个类方法的时候，只能得到一个 **Cattle** 的对象。我们可以要求我们的子类的设计者在他们的子类当中重写 **cattleWithLegsCountVersionB**，但是这样明显非常笨拙，失去了动态的特性。我们当然有办法解决这个问题，现在请大家回到“**Cattle.m**”，参照下列代码：

```

1 + (id) cattleWithLegsCountVersionC:(int) count
2 {
3     id ret = [[self alloc] init];
4     [ret setLegsCount:count];
5     return [ret autorelease];
6 }

```

我们的解决方案就在第 3 行，我们不是用静态的 **Cattle**，而是使用 **self**。说到这里也许大家有些糊涂了，在其他的语言当中和 **self** 比较类似的是 **this** 指针，但是在 **Objective-C** 里面 **self** 和 **this** 有些不大一样，在类函数里面的 **self** 实际上就是这个类本身。大家可以打开 **debugger** 观察一下，**self** 的地址就是 **Bull** 的 **Class** 的地址。所以程序执行到上面的代码的第 3 行的时候，实际上就等同于 `id ret = [[[Bull class] alloc] init];`

我们可以在类方法里面使用 **self**，我们可否通过使用 **self->legsCount** 来访问实例变量呢？答案是不可以，因为在这个时候对象没有被创建也就是说，没有为 **legsCount** 分配内存，所以无法访问 **legsCount**。

由于 **Bull** 类在程序被调入内存的时候就已经初始化好了，**Bull** 类里面的实例函数应该被放到了代码段，所以从理论上来说，我们可以通过使用 `[self setLegsCount:count]` 来调用实例方法的，但是不幸的是 **Objective-C** 没有允许我们这样做，我们在类方法中使用 **self** 来作为消息的接收者的时候，消息总是被翻译成为类方法，如果发送实例方法的消息的话，会在执行的时候找不到从而产生异常。这样做是有一定的道理的，因为一般而言，实例方法里面难免要使用实例变量，在类方法当中允许使用实例方法，实际上也就允许使用实例变量。

关于 **self** 大家需要记住下面的规则：

- 1，实例方法里面的 **self**，是对象的首地址。
- 2，类方法里面的 **self**，是 **Class**。

尽管在同一个类里面的使用 **self**，但是 **self** 却有着不同的解读。在类方法里面的 **self**，可以翻译成 **class self**；在实例方法里面的 **self**，应该被翻译成为 **object self**。在类方法里面的 **self** 和实例方法里面的 **self** 有着本质上的不同，尽管他们的名字都叫 **self**。

请同学们再次回到图 8-1，可以发现通过使用神奇的 **self**，我们动态的创建了 **Bull** 类的对象。但是等一下，我们的程序并不完美，因为 **Bull** 类的 **skinColor** 并没有得到初始化，所以导致了 **null** 的出现。我们在设计 **Cattle** 类也就是 **Bull** 的超类的时候，明显我们无法预测到 **Bull** 类的特征。消除这种问题，我们可以在得到了 **Bull** 对象之后使用 **setSkinColor:** 来设定颜色，当然我们也可以直接写一个 **Bull** 类的方法，来封装这个操作，请同学们打开“**Bull.h**”：

```
+ (id) bullWithLegsCount:(int) count bullSkinColor:(NSString*) theColor;
```

我们追加了一个类方法， `bullWithLegsCount:bullSkinColor:`用于创建 `Bull` 对象，请同学们打开“`Bull.m`”：

```
1 + (id) bullWithLegsCount:(int) count bullSkinColor:(NSString*) theColor
2 {
3     id ret = [self cattleWithLegsCountVersionC:count];
4     [ret setSkinColor:theColor];
5     //DO NOT USE autorelease here!
6     return ret;
7 }
```

上面这一段代码相信大家都可以看明白，笔者就不在这里赘述了。但是笔者需要强调一点，在这里我们不需要调用 `autorelease` 的，因为我们没有在这里创建任何对象。

经过了这个改造，通过在“`08-Class_Method_And_Private_Method.m`”里面我们使用

```
cattle[3] = [Bull bullWithLegsCount:4 bullSkinColor:@"red"];
```

使得我们的代码终于正常了，请参照图 8-1 的第 4 行输出。

8.4，使用动态方法替换实现 `final` 功能

首先请同学们打开“`Cattle.m`”，参照下面的代码片断：

```
+ (id) cattleWithLegsCountVersionD:(int) count
{
    id ret = [[self alloc] init];
    [ret setLegsCount:count];

    if([self class] == [Cattle class])
        return [ret autorelease];

    SEL sayName = @selector(saySomething);
    Method unknownSubClassSaySomething = class_getInstanceMethod([self class], sayName);
    //Change the subclass method is RUDE!
```

```

    Method cattleSaySomething = class_getInstanceMethod([Cattle class], sayName);

    //method_imp is deprecated since 10.5

    unknownSubClassSaySomething->method_imp = cattleSaySomething->method_imp;

    return [ret autorelease];
}
@end

```

在 `cattleWithLegsCountVersionD` 里面，我们将要通过使用动态的方法替换技术来实现 `final` 方法。

第 3,4 行代码，是用于创建 `Cattle` 或者从 `Cattle` 类继承的对象，并且设定实例变量 `legsCount`。

第 6,7 行代码，是用来判断调用这个类方法的 `self` 是不是 `cattle`，如果是 `cattle` 的话，那么就直接返回，因为我们要在这个方法里面把子类的 `saySomething` 替换成为 `Cattle` 的 `saySomething`，如果类是 `Cattle` 的话，那么很明显，我们不需要做什么事情的。

第 9 行代码是老朋友了，我们需要得到方法的 `SEL`。

第 10 行和第 12 行，我们需要通过 Objective-C 的一个底层函数 `class_getInstanceMethod` 来取得方法的数据结构 `Method`。让我们把鼠标移动到 `Method` 关键字上面，点击鼠标右键，选择“Jump to definition”，我们可以看到在文件“`objc-class.h`”里面的 `Method` 的定义。

`Method` 实际上是类方法在 `Class` 里面的数据结构，系统会使用 `Method` 的信息来构筑 `Class` 的信息。在 `Method` 类型的声明里面，我们看到了下面的代码

```

typedef struct objc_method *Method;

struct objc_method {
    SEL method_name;
    char *method_types;
    IMP method_imp;
};

```

其中 `SEL` 和 `IMP` 我们已经很熟悉了，`method_types` 是方法的类型信息，Objective-C 使用一些预定义的宏来表示方法的类型，然后把这些信息放到 `method_types` 里面。

需要强调的是，苹果在 10.5 之后就降级了很多 Objective-C 底层的函数，并且在 64 位的应用当中使得这些函数失效，笔者对剥夺了众多程序员的自由而感到遗憾。

第 14 行的代码，我们把子类的函数指针的地址替换成为 Cattle 类的 saySomething，这样无论子类是否重写 saySomething， 执行的时候由于 runtime 需要找到方法的入口地址，但是这个地址总是被我们替换为 Cattle 的 saySomething，所以子类通过

cattleWithLegsCountVersionD 取得对象之后，总是调用的 Cattle 的 saySomething，也就实现了 final。当然，这种方法有些粗鲁，我们强行的不顾后果的替换了子类的重写。

重要本节提到的 final 的实现方法，没有任何苹果官方的文档建议这样做，纯属笔者自创仅供大家参考，如果使用风险自担。

替换的结果，就是虽然我们在“08-Class_Method_And_Private_Method.m”里面的 cattle[4]l 里面使用 UnknownBull 是图返回 UnknownBull 对象，我们也确实得到了 UnknownBull 对象，但是不同的是，我们在 cattleWithLegsCountVersionD 里面狸猫换太子，把 UnknownBull 的 saySomething 变成了 Cattle 的 saySomething。

让我们回到图 8-1，我们发现最后一行的输出为 Cattle 的 saySomething。

关于 final 的实现方式，我们当然可以使用一个文明的方法来告知子类的使用者，我们不想让某个方法被重写。我们只需要定义一个宏

```
#define FINAL
```

类的使用者看到这个 FINAL 之后，笔者相信在绝大多数时候，他会很配合你不会重写带 FINAL 定义的方法的。

8.5，本章总结

我们在本章里面讲述了方法私有化，类方法的定义和使用，动态方法替换等技术手段，也给大家强调和澄清了 self 的概念。

更重要的是，笔者向大家介绍了一些潜规则，希望大家可以遵守。

非常感谢大家这些天对我的鼓励以及支持！