Objective-C 2.0 程序语言介绍

面向对象技术使程序开发和设计更接近自然的方式，也使程序开发更迅速，更容易修改，更容易读懂，大部分面向对象的开发环境都至少由下面三部分组成：

面向对象的程序语言，以及支持的库

一个基础对象框架库

一组开发工具

本文档就是关于第一部分--面向对象的程序语言Objective-C的介绍，介绍了程序语言本身以及它的运行环境，为进一步去学习第二部分-- Mac OS X Objective-C 应用开发框架库（Cocoa）打下基础。你能通过学习*Getting Started with Cocoa*.学到更多关于Cocoa 的开发技术。

开发Objective-C的两项主要工具是Xcode和Interface Builder，学习*Xcode User Guide*和*Interface Builder*可以熟悉这两项开发工具。

SHAPE * MERGEFORMAT

Objective-C语言是为写出优雅的面向对象程序而设计的简单的编程语言。Objective-C是在ANSI C语言的基础上进行小的但强大的扩展，它在C的基础上扩展了大量基于Smalltalk（第一种面向对象的编程语言之一）的面向对象的部分。Objective-C让C语言拥有全面的、并且通过简单容易的方式就可以实现的面向对象的能力。

对于没有面向对象编程经验的程序员，本文也能帮助你成为面向对象编程的能手，它能教会你如何进行面向对象程序的开发。

谁适合去阅读本文

本文档适合对下面几项内容有兴趣的读者：

学习面向对象的编程

学习基础的Cocoa应用开发框架

Objective-C语言编程

本文档既包括基于Objective-C的面向对象模型的介绍，也包括对Objective-C语言本身的介绍。集中介绍了Objective-C中扩展于C的部分，对C语言不做介绍。因为本文不是关于介绍C的文档，它假定读者已经熟悉C语言了，但不要求对C语言非常的精通，Objective-C在面向对象的程序设计上与ANSI C有很大的差别，即使您对C语言不精通，也不妨碍您使用Objective-C进行面向对象的编程。

文档结构

这个文档分为几个章节和两个附录。

下面几个章节描述的是Objective-C语言，它们涵盖了所有扩展于标准C和C++的特性，这些章节虽然只涉及到语言本身，但也必然接触到基础的运行时系统。

对象和类

定义一个类

属性

Protocols

Fast Enumeration

消息的工作方式

Enabling Static Behavior

异常处理

线程

Apple编译器是基于GNU Compiler Collection的编译器。Objective-C语法支持GNU C/C++语法，Objective-C编译器可以编译C、C++和Objective-C源代码。编译器通过源代码文件名的扩展名为.m识别源文件为Objective-C源文件，源代码文件名的扩展名为.c识别源文件为标准C源文件，源代码文件名的扩展名为.mm识别为在Objective-C中使用C++的源文件。详细的介绍参见章节“在Objective-C中使用C++”。

“运行时系统”关注类**NSObject**，主要介绍NSObject与运行时之间的关系，在特定条件下，它是如何管理对象的内存分配、在运行时如何动态的加载新类、以及对象间的消息传递。

附录包括一些对语言有助于理解的参考资料：

“语言概要”列出了所有Objective-C扩展于C语言的部分的概要。

“语法”列出了Objective-C扩展于C语言的部分的形式语法，它可以和C语言的参考手册一起来使用，C语言的参考手册参见由Brian W. Kernighan和Dennis M.Ritchie合著，Prentice Hall出版的C语言圣经*The C Programming Language*。

约定

文档中涉及到function、method以及其他程序元素时，使用斜体字。如下面语法：

```
@interface ClassName (CategoryName)
```

表示@interface有两个元素被要求，但也可以选择ClassName和CategoryName。

在一些代码中，省略号表示有大量部分被省略了，如下：

```
- (void)encodeWithCoder:(NSCoder *)coder
{
[super encodeWithCoder:coder];
...
}
```

这些约定也适用于附录中。

参见

*Object-Oriented Programming with Objective-C*从一个Objective-C程序员视角讲述了面向对象的程序开发。

*Objective-C 2.0 Runtime Reference*描述了Objective-C运行时库的数据结构和函数，你的程序可以通过这些接口与Objective-C运行时库进行交互。例如：你能增加类或方法，或是获取所有加载类的类定义列表。

*Garbage Collection Programming Guide*介绍了用Objective-C实现的垃圾收集系统。

*Objective-C Release Notes*描述了在最近的Mac OS X系统中Objective-C运行时的改变。

对象和类

对象

从字面上理解，面向对象编程是围绕对象构建的。一个对象包含了数据和一组操作，这组操作可以使用或是修改对象的数据。在Objective-C中，这组操作被称为对象的方法（method），它们能影响的数据是它们的实例变量（instance variables），本质上，一个对象就是一个数据结构（instance variables）和一组程序（method）打包而成的一个自包含程序单元。

比如你要写一个绘图程序，这个程序允许用户通过直线、圆、矩形、文本、位图等组件进行绘图，你需要创建许多用户能操作的基本形状类。比方说，矩形对象，有许多的实例变量：标识矩形位置、矩形长和宽，矩形颜色、是否填充，以及用于绘制矩形的线型样式。矩形类也需要许多方法：设置位置、尺寸、颜色、是否填充、线型样式，以及绘制自己。

在Objective-C中，对象的实例变量定义在对象的内部，通常，只能通过对象的方法才能访问这些变量（你也可以让子类和其他对象直接访问对象的实力变量，这需要使用范围定义指令，详细请参见“实例变量范围”）。因此其他对象要获得对象的信息，这个对象必须提供了一个这样的方法。例如：一个矩形应该有方法可以获取到它的尺寸和位置。

此外，仅有针对对象设计的方法才能被看见，不能错误的将其他类型的方法使用在这个这个对象上，Objective-C中的实例变量就像C语言中的局部变量一样，被隐藏起来不为其他程序可见。对象隐藏了它的实例变量和它的方法的实现。

id

在Objective-C中，对象标识被作为一个特殊的数据类型：id。这个数据类型定义为引用对象的指针。实际上，是指向对象实例变量的指针。像一个C函数和数组一样，任何对象都有一个它自己的地址标识。所有的对象，不管它的实例变量和方法，都统一使用id作为它的标识。

对于基于面向对象的Objective-C，像方法返回值类型，id取代int作为缺省数据类型（对于纯的C语言部分，像函数返回值类型，缺省仍然为int）。

关键字nil被定义作为空对象，表示id的值为0.id，nil以及其他Objective-C的基本类型被定义在头文件objc/objc.h中。

动态类型

id类型是完全没有限制的，就自身而言，它不提供关于这个对象的任何信息，除非，它自己是一个对象。

但是对象并不都是一样的，一个矩形不可能和一个位图对象有同样的方法和实例变量。在一些情况下，程序需要找出关于这个对象所拥有的更多的信息，如这个对象的实例变量有哪些，它提供了哪些方法等等，即使id类型不能提供这些信息给编译器，每个对象自己也是可以提供这些信息给运行时的。

这是可能的，因为每个对象都通过携带一个isa的实例变量标识出对象的类（class）--它描述了对象的类型。每个矩形对象都能够告诉运行时系统它是一个矩形，每个圆都能告诉运行时系统它是一个圆。同类型的对象都拥有相同的行为（method）和相同类型的数据（instance variable），同类型的对象都是这个类型中的一员（member）。

Isa指针也能为对象提供反射功能—找出他们自己（或其他对象）的信息，编译器为运行时系统记录关于类定义的数据结构，运行时系统函数使用isa，去找出在运行时这些对象的信息。例如，使用运行时系统你能找出这个对象是否有制定的方法，或是找出它的超类的名字。

对象的类的更多的细节的讨论请参看章节“类”。

通过静态类型给编译器关于一个对象在源代码中使用的类名也是可行的，类被看成一种特殊的对象，类名能被提供通过类型名来提供，具体参见“类类型”和“Enabling Static Behavior”。

对象消息

这个章节主要介绍发送消息的语法，包括你如何去构造消息表达式。也讨论对象实例变量的“能见度”问题、多态和动态绑定的概念。

消息语法

让一个对象去执行一些动作，你需要通过它提供的方法向它发送一条消息，在Objective-C中，消息表达式需要使用中括号括起来：

```
[receiver message]
```

receiver是一个对象，message告诉它要去做什么。在源代码中，消息是方法名和需要传递给它的参数。当一条消息被发送时，运行时系统从接收者的方法列表中选取合适的方法引用它来处理消息。

例如下面的方法就是告诉myRect对象去执行display方法，让myRect将自己绘制出来。

```
[myRect  display];
```

方法也能携带参数。下面的例子是发消息设置myRect的位置在窗口上的坐标为(30.0, 50.0)：

```
[myRect  setOrigin:30.0  :50.0];
```

在上面的例子中，setOrigin::是方法名，每个参数前面都有一个冒号，参数跟在冒号后面。这个方法是用了非标签参数，非标签参数将难以描述参数的类型和目的。更一般的是使用带标签的参数，参数标签在冒号之前。如setWidth:height:很清楚的描述了两个参数的意图：

```
[myRect  setWidth:10.0  height:15.0];
```

方法也可以带可变个数的参数，但是它们比较稀有。可变的参数通过逗号区隔放在放在方法名的后面（与冒号不同，逗号不是方法名的一部分）。在下面的例子中，假定makeGroup有一个固定参数（group）和三个可选参数。

```
[receiver  makeGroup:group,  memberOne,  memberTwo,  memberThree];
```

跟标准C函数一样，方法也可以返回值。下面的例子假定myRect画一个实心举行返回YES，否则返回NO。

```
BOOL  isFilled;
isFilled  =  [myRect  isFilled];
```

注意：方法和变量可以有相同的名字。

消息表达式也支持嵌套。下面的例子是设置一个矩形的颜色为另一个矩形的颜色：

```
[myRect  setPrimaryColor:[otherRect  primaryColor]];
```

Objective-C 2.0也支持点操作符，点操作符为引用对象提供紧密（compact）、简单的存取方法，典型的应用是在声明的属性特征中在联合中使用。详情参见“属性”和“圆点语法”。

发送消息到**nil**

在Objective-C中，发送一条消息到nil（空值）也是有效的的，但是在运行时不会有任何的反应（也不会有异常）。在实际的应用中，这会给Cocoa带来一些好处。发送给nil的消息，其返回值也可能是有效的：

如果方法返回值是：对象、任何指针类型、任何存储小于等于sizeof(void*)的整数标量、

float、double、long double、long long,发送给nil的消息，都将返回0.

如果方法返回值是在*Mac OS X ABI Function Call Guide* 中注册定义的struct，发送给nil的消息，返回值为0， 0将填充返回的struct中的每一个字段.返回其他struct数据类型的，将不被0填充。

除了上面说明的外，其他发送给nil的消息，返回值都是未定义的（undefined）.

下面的代码片段示意发送消息给nil也是允许的。

```
id anObjectMaybeNil = nil;
// this is valid
if ([anObjectMaybeNil methodThatReturnsADouble] == 0.0)
{
    // implementation continues...
}
```

接收者实例变量

方法将自动分配接收对象实例变量，不需要声明它们作为方法的参数。举个例子，primaryColor方法没有任何参数，你获取otherRect的填充颜色（primary color）并将它返回，每个方法都假定接收者和它的实例变量已经分配好了，不需要将它们作为参数进行声明。

```
[myRect setPrimaryColor:[otherRect primaryColor]];
```

这项约定简化了Objective-C源代码，它也被支持在面向对象的程序设计中的对象和消息。消息被发送给接收者，就像一封信被邮递到你的家，消息参数带给接收者之外的信息，并不需要将接收者自己带给它自己。

方法仅自动分配接收者实例变量，如果它需要的信息存储在另一个对象中，它必须发送消息给这个对象让它暴露变量的内容。primaryColor和isFilled显示了刚才这个意图的用法。

参见“定义类”了解更多关于实例变量的信息。

多态性

基于上面的例子的描述，在Objective-C中消息的看上去和标准C中的函数调用相似，但因为方法属于一个对象，消息的行为与函数调用还是不同。

在一些情况下，一些对象仅能对针对它设计的方法起作用，它不能被其他类的方法使用，即使这些类有同样名称的方法。这意味着两个对象对同样的消息有不同的响应。例如，每个类的display消息都通过自己的方式“显示”它自己，圆和矩形对同一个消息有不同的画线轨迹。

这个特性，被称为多态性，是面向对象程序设计中的一项重要的特性。和动态绑定配合使用，可以使你写的代码提供给任意数量不同类型的对象使用，这些对象不需要在你写代码时就已定义好，它们可以是你以后再开发的对象，也可以是其他项目的其他对象。例如你可以写代码给id变量发送display消息，任何有display方法的对象都是一个潜在的接收者。

动态绑定

函数和消息的最根本不同是：函数和它的参数是在编译时就已经确定下来了，但是消息和接收对象直到程序已经在运行、消息在传递时才能确定下来。因此，消息的方法绑定只有在运行时才能确定，并不是在编译时。

确切的消息方法实现依赖消息的接收者，不同的接收者可能有有相同的方法名不同的方法实现（多态性）。对于编译器而言，要找到消息对应的确切的方法实现，它就必须知道接收对象的类型—它所属的类，但是，接收者的信息（包括它的类型）只有在消息接收时才能确定（动态类型），通过源代码的类型定义并不能完全确认接收者的类型。

方法的实现的选择发生在运行时，当消息发送时，运行时消息例程查找消息接收者和消息方法，找到消息方法实现的机器码，呼叫这个方法，传递给它消息实例变量指针。（更多详细的了解，请参见“消息如何工作”）。

方法名被用于作为方法实现的选择，因为这个原因，消息的方法名经常被提及为“selectors”。

动态绑定和多态性相辅相成，为面向对象程序设计提供了许多灵活性和强大的能力。即使每个对象都有它自己版本的方法，也可以通过一个程序获取多样的结果，这样做并不需要多个程序，只需要多样性的接收者对象即可。在程序运行时才确定接收者，能够根据用户的不同行为了调整在执行的方法实现。

当执行基于应用套件的代码时，例如，用户点击菜单上的命令“剪切、拷贝和粘贴”产生的消息，无论当前选择的什么对象接收到消息，都可以根据自己的方式进行响应，显示文字的对象和显示扫描图像的对象对“拷贝”消息的响应是不同的，描绘一组形状的对象和描绘一个矩形的对象响应也是不同的。在运行前，消息都没选择方法（方法未绑定到消息），这些差别是因为这些方法都是独立的响应这些消息的。发送消息的代码不需要关注它们，也不需要对各种可能进行列举，每个应用都可以创造自己的对象，通过它们自己的方式响应copy消息。

Objective-C的动态绑定更进一步，它甚至允许发送的消息是可变的，消息在运行时才决定（方法选择器），关于这些更多的讨论参见“消息如何工作”。

类

面向对象程序构建在多种对象之上的，基于Cocoa框架的程序可能使用NSMatrix对象、NSWindow对象、NSDictionary对象、NSFont对象、NSText对象、以及许多其他的对象。程序之中经常用到同一个类型或类的超过一个的对象 — 比方说几个NSArray对象，几个NSWindow对象。

在Objective-C中，通过定义类来定义对象，类定义是一组对象的蓝本，它声明了类的每一个成员为实例变量，定义了这个类所有对象能使用的方法。

编译器为每个类创建一个“类对象（class object）”，类对象知道如何去创建一个新的属于这个类的对象（基于这个原因，传统上称类对象为对象工厂（factory object））。类对象是类的编译版本，它构建的对象是类的实例，在你的程序中执行主要工作的是这些由类对象在运行时创建的对象实例。

类的所有对象都有相同的方法，它们也拥有从一套模子里拷贝出来的实例变量，每个对象存储它自己的实例变量，但是方法是共享的。

一般约定，类名都是首字母大写，实例名都是首字母小写。

继承

类的定义是累进的，任一个新类都是在另一个类的基础上定义的，新类继承了它的方法和实例变量。新类只是在继承类的基础上进行了增加或修改，它不需要去复制继承的代码。

继承使所有的类构成一颗树型结构，仅有一个类作为这棵树的根（root），基于Foundation框架的代码，根类指定是NSObject。每个类（除根类外）都有一个超类（superclass）更接近根类，任何类（包括根类）都可以作为其他类的超类更远离根类。下图（Figure 1-1）描述了绘图程序的几个类的层次关系：

这个图显示了Square类是Rectangle类的子类，Rectangle类是Shape类的子类，Shape是Graphic类的子类，Graphic类是NSObject的子类。继承是一个累进的过程。因此，Square对象有Rectangle、Shape、Graphic、和NSObject定义的方法和实例变量，这些方法和实例变量的定义也可以认为是针对Square的。也可以这么说，一个Square对象，不仅是一个Square，也是Rectangle、Shape、Graphic、NSObject。

每个类（除了NSObject）都是另外类的专有化和个性化改造，每一个相继的子类都累进了它继承的类更多的修改。Square类定义了由Rectangle转换为Square最小的要求。

当你定义类时，你就将它加入到它声明的超类的继承树上，你创建的每一个类都必须是另一个类的子类（除非你打算定义另一个根类）。提供大量丰富的类可以作为超类，Cocoa包括NSObject类和几个包含超过250个附加类的框架，一些类是“现货供应的”---直接应用到你的代码中，另一些需要根据你的需求定义新的子类。

一些框架类定义了绝大部分你需要的部分，但留了一部分优雅的实现需要在子类中实现，因此你能通过创造一些优雅的子类写少量的代码，重用框架程序。

NSObject类

NSObject是一个根类，并没有超类。它定义了基本的Objective-C框架对象和相关信息。它为从它继承的类和类的实例提供了具备对象行为和与运行时系统配合的能力。

即使不需要从其他类继承任何特别的行为的类也必须要是NSObject的子类。类的实例在运行时中至少要像Objective-C对象的行为，从NSObject类继承这种能力是最简单最可靠的方式，而不需要你重新去创造新的类定义。

继承实例变量（**Inheriting Instance Variables**）

当一个类对象创建一个新的实例时，新的对象不仅包括定义它的类的实例变量，还包括它的超类定义的实例变量，以及它超类的超类，这样一直追索到根类，因此isa实例变量是定义在NSObject中，成为每个对象的一部分，isa连接每个对象到它的类。

下图显示了Rectangle的一些实例变量的定义的特定实现，并且指出了它们是从哪里来。需要主要的是，这些变量哪些是Rectangle追加在Shape之上的，哪些是Shape追加在Graphic之上的，以此类推。

类也可以并不定义新的实例变量，它可以仅定义新的方法，使用它继承得到的实例变量，甚至它根本就不需要任何实例变量。例如，Square类就根本就不需要为自己声明任何实例变量。

继承方法

对象不仅可以使用定义它的类的方法，而且可以使用定义它的超类的方法，以及超类的超类的方法，以此类推，所有到根类的继承路径上的类的方法都可以使用。例如，Square对象能使用Rectangle、Shape、Graphic和NSObject类定义的方法，使用这些方法就跟类自己定义的一样。

在你的程序中你定义的任何新类都可以复用在继承树上的类写的这些代码。这种继承方式是面向对象的程序设计最主要的益处。当你使用Cocoa提供的面向对象的框架时，你的程序只需要在这些框架类的基础上加入少量的客户化代码就可以实现应用所需要的功

能。

类对象也从他们的的类继承树上进行继承，但它们没有实例变量（只有实例动作），它们只继承方法。

方法重载

在继承中有一种特殊的情况：当你定义一个新类时，你定义了一个和父类（或祖先类）同名的方法，这个新的方法将覆盖父类（或祖先类）的方法，在新类中，新的方法将被实例化，新类的子类也继承的是这个新方法。这种现象被称为方法重载。

例如，Graphic类定义了display方法，Rectangle通过定义它自己版本的display方法覆盖了它。Graphic的方法对所有继承于它的对象都是有效的，但是Rectangle例外，Rectangle对象使用它们自己版本的display方法。

尽管重载阻断了原始版本方法的继承，但是在新类中的其他方法仍可以跳过重载的方法找到原始方法，详情请参见章节“消息发送给自己和超类”。

重载的方法也能完全正确的包含吸纳原始的方法，新方法只需要对原方法进行修正或修改，而不是完全取代它，继承树上的类定义了同样的方法，但每一个新版本都包含吸纳它覆盖的版本，方法的实现就能更有效的延伸到所有的类上。

尽管子类可以重载继承的方法，但是它不能重载继承的实例变量。即使对象会为每个它继承来的实例变量分配内存，你也不能声明一个新的同名的实例变量，如果你一定要这样做，编译器会报错误。

抽象类（**Abstract Classes**）

一些类被设计为仅或主要的目的是用于给其他类继承使用，这些类被称为抽象类。抽象类包含的方法和实例变量能为大量的不同的子类作为正常的定义来使用，抽象类通常是不完整的，但是它为子类提供了有用的代码来缩减子类的实现。

NSObject是最重要的抽象类例子，在程序中经常定义NSObject的子类，使用这些子类的实例，但是从不直接使用NSObject类的实例，NSObject实例并没有多大的价值，它是一个不能处理任何事情的普通对象。

NSView类是一个可以直接使用的抽象类的例子。

抽象类的代码经常帮助定义应用的结构，当你创建抽象类的子类，新类的实例将能很容易的融合到应用的结构中，自动的和其他类进行配合开始工作。

（因为抽象类经常需要有子类才能被使用，有时候也称它们为抽象超类（abstract superclasses））。

类类型

类定义是对一种对象的描述。事实上是类定义了一个数据类型，这个类型不仅基于这个类定义的数据结构（实例变量），还包括这个类定义的行为（方法）。

类名可以出现在C语言中任何类型可以出现的地方，例如：可以作为sizeof操作符的参数：

```
int  i  =  sizeof(Rectangle);
```

静态类型

你能使用类名代替id来标明对象的类型：

```
Rectangle  *myRect;
```

因为这种方式声明了对象的类型，告诉了编译器对象的类型信息，这种已经类型的变量类型称为静态类型（static typing），就像用id定义了一个对象的指针，而这个对象的类型是固定指向一个类的，对象总是被指针标识的（也只能被指针标识），静态类型使指针指向的对象的类明确化，id标识会隐藏指针指向的对象的类。

静态类型允许编译器对类型进行检查。例如，对象收到并不能给予响应的消息（类没这个方法），编译器会给予警告。如果用id了标识对象，编译器不会去做这些类型检查。另外，它也能让读你代码的其他人能更清楚的了解你的意图。当然，它也不能完全代替动态绑定和运行时的动态类型判定。

可以使用静态类型标识类及子孙类的对象。例如，Rectangle是Graphic的子孙类，Rectangle的静态类型可以是Graphic：

```
Graphic  *myRect;
```

Rectangle是Graphic是可能的，虽然它比Graphic多拥有Shape和Rectangle的实例变量和方法，但它仍然是Graphic。针对类型检查，编译器把myRect作为Graphic，但在运行时中，它被作为Rectangle来进行处理。

关于静态类型更多的信息以及它的好处，请参见“Enabling Static Behavior”。

类型反射（**Type Introspection**）

类型反射是指对象在运行时能获取它们自己的类型信息。NSObject类中定义的isMemberOfClass:方法可以检测消息接收者是否是指定类的实例。

```
if  (  [anObject  isMemberOfClass:someClass]  )
  ...
```

定义在NSObject类中的isKindOfClass:方法检测消息接收者是否是指定类或子孙类的实例。

```
if  (  [anObject  isKindOfClass:someClass]  )
  ...
```

isKindOfClass:返回YES表示消息接收者是可以被这个类型静态标识。

反射并不仅提供类型信息，后面的章节将讨论类对象的方法，检查对象是否可以可以响应指定的消息，以及其他的信息。需要了解关于isKindOfClass:、isMemberOfClass:和相关

的其他方法，需要去查看Foundation框架手册中的NSObject类的说明。

类对象

类对象描述了类定义包含的各种信息，以及关于类的实例的信息：

类及它的超类的名字

一组实例变量样板的描述

方法名、返回值以及参数类型的描述

方法实现

这些信息由编译器编译并将它们记录到数据结构中供运行时系统使用。编译器创建了一个对象—类对象，来描绘类的这些信息，类对象包含了这个类的所有信息，包括这个类的实例像什么的主要信息，根据这些类定义信息可以依据要求生成新的实例。

虽然类对象保留了类实例的蓝本，但它自己并不是类的实例，它没有类的实例变量，也不提供类的实例所具有的方法。但是，在类的定义中已经包含了类对象方法的实现—类方法将被映射成实例方法，类对象从类及它的继承关系上继承了类方法，就像实例继承实例方法一样。

在源代码中，类对象通过类名来标识。在下面的例子中，Rectangle类通过从NSObject类继承来的方法返回类的版本号：

```
int versionNumber = [Rectangle version];
```

类对象通过类名来标识仅允许类对象作为消息表达式中的接收者。在别的地方，你需要根据实例或是类的方法返回类的id，下面两个例子都是返回class消息：

```
id aClass = [anObject class];
id rectClass = [Rectangle class];
```

上面的例子可以看出，类对象也能像其他所有的对象一样，通过id来标识。类对象也可以使用更确切的指定为Class数据类型：

```
Class aClass = [anObject class];
Class rectClass = [Rectangle class];
```

所有的类对象都是Class类型的，使用类的类型名等效于使用类名去标识实例的静态类型。

类对象是完完全全的对象，也支持动态绑定、消息接收、从其他类继承方法。它的特别之处在于它是由编译器创建的，除有从类定义中构建的外，缺乏它自己的数据结构（实例变量），它是在运行时产生实例的代理。

创建实例

类对象的最重要功能就是创建新的实例。下面的代码告诉Rectangle类去创建一个新的Rectangle实例，并把它赋值给myRect变量：

```
id    myRect;
myRect  =  [Rectangle  alloc];
```

alloc方法为新对象的实例变量动态的分配内存，并将这些实例变量全部初始化为0—除isa变量是连接到它的类的实例变量外的所有实例变量。如果这个对象要能被使用，通常还需要更完整的初始化，方法init就是这样的功能，典型的初始化过程是在内存分配后立即进行初始化：

```
myRect  =  [[Rectangle  alloc]  init];
```

上面的代码行在之前的例子中的myRect对象接收任何消息之前都是必须的。alloc方法返回一个新的实例，随后这个实例执行init方法设置它的初始状态。每个类对象至少有一个像alloc的方法能够去创建新的对象，每个实例至少有一个像init的方法能够去初始化实例让它可以被使用。初始化方法可以使用带标签的参数（如initWithPosition:size:，初始化一个新的Rectangle实例），但通常他们都是以“init”开始的。

定制类对象

将类使用对象来对待，并不是Objective-C语言的突发奇想，它是有意为之的一种选项，有时它有令人惊讶的好的益处。定制类对象也是可能的，在Application Kit中，NSMatrix对象就可以定制为含带NSCell对象。

NSMatrix对象可以创建一些特别的cells，无论是在matrix创建时还是创建后。画在屏幕上的matrix在运行时可以根据用户的行为拉伸和缩小，当在拉伸时，matrix需要创建一些新的对象去填充拉伸出来的空白区域。

但是使用什么对象去填充呢？每个matrix都只显示一种NSCell，但是，NSCell有很多的种类（子类），所有的这些子类都可以被matrix用来填充：

当matrix在创建NSCell对象时，它可能是显示按钮的NSButtonCell对象，可能是显示允许用户输入和编辑的编辑框NSTextFieldCell对象，也可能是其他的NSCell对象，NSMatrix对象必须允许任何的cell对象，甚至是还未声明和开发的类型的对象。

这个问题的一个解决方案是，将NSMatrix设计为所有cell对象的抽象类，要求每个子类都实现了产生新的cell的方法。它们只有都实现了这些方法，才能保证创造正确类型的对象。

但是要求其他的类去完成NSMatrix类应该去完成的工作，可能引起子类的爆炸性增长，应用可能需要多个NSMatrix类，每一个都有不同的NSCell类，它可能造成大量杂乱的NSMatrix子类，每次你创造一个新的NSCell类，你可能都需要定义一个新的NSMatrix类，

另一方面，在不同的程序中，你需要做大量重复的工作，所有的这些仅是为了弥补NSMatrix类的不足。

一个更好的解决方案是，允许NSMatrix实例和一个种类的NSCell一起初始化，它定义了一个setCellClass:去传递NSCell对象类型给NSMatrix去填充拉伸造成的空白：

```
[myMatrix  setCellClass:[NSButtonCell  class]];
```

在初始化或其他任何改变尺寸时需要新的cell对象时，NSMatrix使用类对象去创建新的cell。如果类不是对象，不能分配变量，不能作为消息进行传递，处理这种情况将是非常困难的。

变量和类对象

当你定义一个新的类时，你可以指定实例变量，每个类的实例都能维护它定义的变量的拷贝—每个对象都能控制它自己的数据。例外的是类变量，类变量与实例变量不同，它仅是对象内部的数据结构，只能由类的定义对它进行初始化，也只能由类对它进行访问。另一方面，类对象不能访问任何实例的实例变量，不能初始化、读、修改实例变量。

要为所有的类去定义共享的数据，你必须定义一个外部变量，最简单的方式是在类实现文件中像下面的例子一样声明一个变量：

```
int  MCLSGlobalVariable;
@implementation  MyClass
//  implementation  continues
```

更优雅的实现方法是，可以声明一个变量是static，再提供类方法去管理它。声明一个变量是static的将限制它的范围只能是这个类—仅类所在的实现文件的那部分，因此静态变量不像实例变量那样能继承，直接访问和创建子类，这种模式通常用来定义一个类的共享变量，如单例模式：

```
static  MyClass  *MCLSSharedInstance;
@implementation  MyClass
+  (MyClass  *)sharedInstance
{
    //  check  for  existence  of  shared  instance
    //  create  if  necessary
    return  MCLSSharedInstance;
}
//  implementation  continues
```

静态变量帮助类对象实现更多的功能，包括工厂模式，它能通过它自己的权限去完整的创建多个对象。类对象能够去协调它创建的这些类的关系，管理应用进程关于这个类的信息。如果你仅需要类的一个对象，你可以放所有的对象状态为静态变量，而只是用它的方法，这样可以不用去分配和初始化实例。

初始化类对象

如果类对象使用在分配实例之外被使用，需要像一个普通实例一样被初始化。尽管程序没有分配类变量，但是Objective-C为程序提供了一种方式去初始化他们。

如果类作为静态变量或是全局变量，初始化方法的地方就是设置他们的初始值的好地方。比如，一个类要维护一组实例，初始化方法能创立好数组，甚至分配一两个缺省实例，让它们准备好。

在类收到任何消息之前，运行时系统都在它的超类收到initialize消息之后给类对象发送一个initialize消息，这是在类被用之前给的一个设置运行时环境的机会。如果没有初始化的需求，你没有必要去写一个initialize方法去响应这个消息。

因为继承的关系，initialize消息发送给没有实现initialize方法的类时，会将消息转发给它的超类，即使超类已经收到过initialize消息了。举个例子：假定类A实现了initialize方法，类B继承于类A，但没有实现initialize方法，在B收到第一个消息之前，运行时系统发送initialize消息给它，但是，因为类B没有实现initialize方法，类A的initialize方法将被代替去执行，因此，类A要去确保它的初始逻辑仅被执行一次。

为了避免执行初始逻辑超过一次，当实现initialize方法时，使用下面的模板：

```
+ (void)initialize
{
    static BOOL initialized = NO;
    if (!initialized) {
        // Perform initialization here.
        ...
        initialized = YES;
    }
}
```

根类方法（**Methods of the Root Class**）

所有的对象，类和实例一样，都需要有一个运行时系统的接口，让类对象和实例能够反射它们的能力和报告它们在继承树上的位置，NSObject提供了这些接口。

因此，NSObject方法并不需要去实现两次—一次是提供实例的运行时接口，另一次是为类对象去复制这个接口。类对象被指定为去执行在根类中定义的实例方法。当一个类对象收到一个它不能使用它的类方法去响应的消息时，运行时系统决定它是否能使用跟实例

的方法来响应。如果没有类方法能执行时，类对象可以执行定义在根类中的实例方法。

关于更多的类对象执行跟实例方法的特性，参看在Foundation框架手册中的NSObject类的详细说明。

在源码中的类名

在源代码中，类名被用于两种非常不同的环境，两种环境标明了类的两种角色：作为数据类型、作为对象：

类名作为对象的一种类型名，如：

```
Rectangle  *  anObject;
```

上面的anObject是Rectangle的静态类型指针，编译器知道anObject将有一个Rectangle实例的数据结构，有Rectangle的实例方法，有Rectangle类的继承关系。

静态类型能够让编译器做更好的类型检查，让代码具有更好的可读性。关于更多的静态类型说明请参看“Enabling Static Behavior”。

作为消息表达式的接收者，类名引用的是类对象。这种表示法被上面的好几个例子使用。类对象使用类名来表示仅在作为消息接收者时才可以，在其他任何情况下，你都需要类对象去透漏它的id（通过发送它一个class消息）。下面的例子是把Rectangle类作为一个参数传递给isKindOfClass:消息。

```
if  (  [anObject  isKindOfClass:[Rectangle  class]]  )
    ...
```

上面的例子中，如果简单的使用“Rectangle”作为参数将是不合法的，类名只能在作为消息接收者时才能被作为类对象。

如果你在编译时不知道类名，但是在运行时你知道类名的字符串，可以使用NSClassFromString返回类对象：

```
NSString  *className;
    ...
if  (  [anObject  isKindOfClass:NSClassFromString(className)]  )
    ...
```

如果传递进去的字符串不是一个有效的类名，上面的函数将返回nil。

类名和全局变量、函数名存在同样的命名空间，类和全局变量不能有相同的名字，类名大约是Objective-C中仅有的全局可见的名字。

类定义

面向对象程序的大部分代码都是在构建新对象—定义新类。在Objective-C中，类定义分两部分：

接口部分，声明类的方法和实例变量，以及类的超类

实现部分，实际的对类进行定义，包含它的方法的实现代码

这两部分通常被分在两个文件中。但是，有时类的定义文件可以通过使用category特性分成好几部分，category特性可以区分类的定义部分，和已存在类的扩展部分，category特性的更多描述请参看章节“Categories and Extensions”。

源文件

编译器并没有要求接口部分和实现部分分割在两个不同的文件，但通常我们将它们分割在两个不同的文件中，接口文件必须对每个使用到这个类的人可见。

单个文件中能声明或实现一个和多个类，但是，通常每个类定义在一个独立的接口文件中，即使没有使用独立的实现文件。独立的接口文件有利于表明类的状态和依赖关系。

接口和实现文件通常使用类名来命名。实现文件使用.m作为后缀，表明它包含的是Objective-C源代码文件，接口文件可以使用任意其他的扩展名，因为它被包含在其他源代码文件中，接口文件通常使用.h作为后缀，表明它是一个头文件（header file）。例如，Rectangle类在Rectangle.h中声明，在Rectangle.m中定义。

把对象的接口从它的实现文件中分离出来更符合面向对象的程序设计规范。每个对象都是自包含的实体，从外部可以作为一个“黑盒”来看待。一旦你决定了一个对象与你程序的其他元素的关系—你在接口文件中声明的那样，你就可以自由的修改它的实现，而不用担心会影响到应用的其他部分。

类接口

类接口声明的开始使用编译器指令@interface，截止使用编译器指令@end。（所有Objective-C的编译器指令都使用“@”开始）。

```
@interface ClassName : ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

第一行声明了新类的类名和它的超类名，超类定义了新类在类继承树上的位置，如果冒号和超类被省略，新类将被定义为一个根类，和NSObject处于同一位置。

实例变量的声明包含的大括号之中，实例变量是类的每个实例的数据结构部分，下面的例子说明了实例变量的声明：

```
float width;
float height;
BOOL filled;
NSColor *fillColor;
```

类方法的声明在大括号之后，在类结束符之前。类方法—被类对象使用的方法—的方法名在前面使用“+”号标识：

```
+  alloc;
```

实例方法—被类的实例使用的方法—的方法名在前面使用“-”号标识：

```
-(void)display;
```

虽然不符合习惯，但是你还是能定义同名的类方法和实例方法。方法也可以和实例变量的名同名，特别是在方法返回变量的值时使用和变量名同名的方法名。例如，Circle有一个radius方法，它就和radius实例变量相匹配。

方法返回类型的声明使用标准C中的类型转换语法：

```
-(float)radius;
```

参数类型的声明方式也采用同样的方法：

```
-(void)setRadius:(float)aRadius;
```

如果返回或参数没有指定明确的类型，缺省类型为id，alloc早期就是返回id。

当有多个参数时，参数之间通过空格进行间隔，后续的参数需要使用参数标签和参数名，参数标签和参数名之间使用冒号进行区隔，就跟消息一样，例如：

```
-(void)setWidth:(float)width  height:(float)height;
```

如果方法带有可变数量的参数，声明他们需要用逗号隔开，并用省略号表示，就像下面的例子：

```
-makeGroup:group,  ...;
```

接口导入

接口文件必须包含在任何需要依赖这个类定义的源代码模块，包括任何创建这个类实例的模块，发送消息到这个类声明的方法，或者是使用到在这个类中声明到的实例变量。接口通常使用#import指令进行包含：

```
#import  "Rectangle.h"
```

这个指令和#include指令基本等同，与#include不同的地方是，它能保证同样的文件不会被重复包含，因此，它被作为#include更好的替代，在Objective-C代码中被广泛使用。

为了表明一个类的定义构建在被继承的超类的定义基础上，接口文件需要在开始的地方导入超类的接口：

```
#import  "ItsSuperclass.h"
@interface  ClassName  :  ItsSuperclass
{
    instance  variable  declarations
}
method  declarations
@end
```

每个接口文件包含它的超类的接口作为一种惯例，这样也间接的包含了它的所有祖先类。当一个源模块导入一个类的接口时，它得到构建这个类的全部继承实体。

需要注意的是，如果是要支持超类的*precomp*—一个预编译头文件，你需要去导入预编译作为替代。

引用其他类

接口文件在声明类时，通过导入它的超类，隐含的包含了继承路径上的所有类，从NSObject开始知道它的超类都已经被导入到了类声明文件中，但是，如果接口引用的类不在这个继承路径上，它必须明确的显示使用@class指令导入它们。

```
@class  Rectangle,  Circle;
```

这个指令简单的告诉编译器“Rectangle”和“Circle”是类名，并没导入他们的接口文件。

接口文件中引用到的类名，如静态类型实例变量，返回值，或是参数，都需要显示的导入。如下面的例子：

```
-(void)setPrimaryColor:(NSColor  *)aColor;
```

引用到了类NSColor。

如此简单的声明类名为一个类型，并不依赖任何类接口细节（它的方法和实例变量），@class指令给编译器足够的信息进行预警处理。然而，在类接口被实际使用的地方（实例创建、消息发送），类接口必须被导入。通常是，接口文件中通过@class声明类，在对应的实现文件中导入它们的接口（因为它将创建这些类的实例或发送给他们消息）。

指令@class让编译器和连接器看到最少量的代码，因此，它也是最简单的类名的前向声明方式。因为简单，所以它避免了伴随导入文件引起的循环导入的问题。例如，一个类声明了一个类型为另外一个类的静态类型的实例变量，两个类的接口文件都要彼此导入对方的接口文件，这样编译器将无法正常的进行编译。

接口的角色

接口文件的目的是声明类到其他源模块（或者是源代码），它包含了使用这个类去工作的所需要的全部信息（程序可能还需要少许的文档）。

接口文件告诉用户类是如何连接到继承树上的，类还需要哪些其他的类（在类中继承或是引用到的类）。

接口文件也要让编译器知道对象包含哪些实例变量，告诉程序继承了哪些变量子类。实例变量虽然比它的接口看上去更像是类的实现的部分，但是他们仍然要在接口文件中被声明，这是因为编译器必须要在对象被使用的地方知道它的数据结构，而并不是在它被定义的地方。作为程序员，通常在使用类的时候是不关注类的实例变量，除非是在定义子类的时候。

通过方法声明列表，接口文件让其他模块知道哪些消息可以发送给类对象，哪些消息可以发送给类的实例。每个方法都能用在接口文件声明的类定义之外的地方，方法在类内部的实现并不被关心。

类实现

在实现中，类的定义跟它的声明一样是非常结构化的，它使用@implementation作为开始指令，@end作为结束指令：

```
@implementation  ClassName  :  ItsSuperclass
{
    instance  variable  declarations
}
method  definitions
@end
```

然而，每个实现文件都必须导入它自己的接口。例如，Rectangle.m导入Rectangle.h。接口并不需要重复它导入的声明，因此可以简化省略的实现部分：

继承类的名字

实例变量声明

简化的实现可以让它主要集中在方法的定义上：

```
#import  "ClassName.h"
@implementation  ClassName
method  definitions
@end
```

类的方法的定义，跟C函数的定义一样，使用一对大括号。在大括号前，它们的声明跟在接口文件中的一样，只是不要最后面的分号。如：

```
+alloc
{
    ...
}
-(BOOL)isfilled
{
    ...
}
-(void)setFilled:(BOOL)flag
{
    ...
}
```

带可变数量参数的方法就跟函数一样处理：

```
#import  <stdarg.h>
...
-getGroup:group,  ...
{
    va_list  ap;
    va_start(ap,  group);
    ...
}
```

引用实例变量

缺省情况下，实例方法的定义在它的作用域内有对象所有的实例变量，它能简单的通过名字引用它们。虽然编译器创建等同于C的结构来存放实例变量，但是结构的精确的类型被隐藏。你不需要使用结构操作符（.或->）去引用对象的数据。例如，下面的方法定义

引用接收者的filled的实例变量。

```
-(void)setFilled:(BOOL)flag
{
    filled  =  flag;
    ...
}
```

接收对象和它自己都没有把filled声明为方法的参数，实例变量仍然落在它的作用域内，这种简化的方法语法是写Objective-C代码的重大速写法。

当并不是消息的接收对象引用实例变量时，实例变量的对象必须通过声明明确告诉编译器对象的静态类型，在引用一个静态类型对象的实例变量时，需要使用结构操作符（->）。

例如，假定Sibling类声明一个静态类型对象twin作为一个实例变量：

```
@interface  Sibling  :  NSObject
{
    Sibling  *twin;
    int  gender;
    struct  features  *appearance;
}
```

只要静态类型对象的实例变量在类的作用范围之内（因为twin被定义在同样的类中），Sibling方法能直接设置它们：

```
-makeIdenticalTwin
{
    if  (  !twin  )  {
        twin  =  [[Sibling  alloc]  init];
        twin->gender  =  gender;
        twin->appearance  =  appearance;
    }
    return  twin;
}
```

实例变量的作用域

虽然实例变量被声明在类接口中，当更多的是使用在类的实现部分。对象接口依赖它的方法，并不依赖它内部的数据结构。

经常有和实例变量相一致的方法，如下面的例子：

```
-(BOOL)isFilled
{
    return  filled;
}
```

但这并不是一定的，一些方法返回的信息并不是存储在实例变量中的，一些实例变量存储的可能是对象不愿意暴露的信息。

当一个类在不断的修改后，实例变量的选择可能已经改变，即使它的方法名仍然相同。由于是通过消息作为类的实例之间相互交互的手段，所以这些改变并不会影响到他们的接口。

为了迫使对象去隐藏它的数据，编译器限制了实例变量的范围—限制了它们在程序中的可见性，通过设置三个不同的作用域范围级别，提供了弹性，每个级别通过特别的编译器指令来标记：

指令	说明
@private	实例变量只能在声明它的类中被访问
@protected	实例变量可以在声明它的类和它的子类中被访问
@public	实例变量可以在任何地方被访问
@package	在**64**位上，**@package**实例变量在实现这个类的**image**内象**@public**一样，在之外象**@private**一样。 与**private_extern**变量和函数相似，任何在类实现的图像之外的代码试图使用实例变量将得到一个连接错误。对在框架类中的实例变量非常有用，这些地方使用**@private**可能太严格，使用**@protected**或**@public**又太宽泛。

下面是实例变量作用域的图示：

一个指令提供在它之后的所有实例变量的范围，直到下一个范围指令或是变量列表截止。在下面的例子中，age和evaluation实例变量是private的，name、job和wage是protected的，boss是public的。

```
@interface  Worker  :  NSObject
{
    char  *name;
@private
    int  age;
    char  *evaluation;
@protected
    id  job;
    float  wage;
@public
    id  boss;
}
```

缺省情况下，就是没有标记范围的实例变量（像例子中的*name*）是@protected的。

所有的在类定义中声明的实例变量，无论它怎么被标记，都在类定义的作用域内。上例中的类Worker声明了一个job实例变量，在这个类的方法定义中能直接引用它：

```
-  promoteTo:newPosition
{
  id  old  =  job;
  job  =  newPosition;
  return  old;
```

```
}
```

显然，如果一个类不能访问它自己的实例变量，那么这个实例变量将是无用的。

通常，类也能访问被继承类的实例变量，这种能力将随着继承关系一直得到延续，它使类在它们的作用域内拥有完整的数据结构，尤其是你想了解一个你要继承的类的详细情况的时候。上面例子中的*promoteTo:*方法也可以定义在任何继承于Worker的类中，仍然能够正常访问在Worker类中定义的*job*实例变量。

然而，也有一些原因要去限制继承类去直接访问被继承类的实例变量：
声明变量的类可能将变量与它的实现进行了紧密的捆绑，在它的后续的版本中，它可能会去掉这个变量或是修改它的作用，一旦在子类中使用这样的继承变量，将会引起子类的错误。
另外，如果子类访问继承变量，并且修改它，它可能会在声明它的类中引入bug，特别是在这个变量与类的内部实现有复杂的依赖关系的时候。

限制实例变量的作用域在声明它的类范围内，必须要使用@private标记，实例变量被标记为@private后，在子类中也只能通过调用它存在的公共授权方法来进行访问。

另外一个极端是，标记一个变量是@public的将使它完全可见，即使是在声明或继承这个变量之外的类定义中。通常，去获取存储在实例变量中的信息，需要其他对象发送一条消息获取它，然而，公共的实例变量能够在任何地方被访问，就像一个C结构的字段一样。例如：

```
Worker  *ceo  =  [[Worker  alloc]  init];
ceo->boss  =  nil;
```

注意的是，对象必须是静态类型的。

标记实例变量为@public将破坏对象封装数据的能力，它破坏了面向对象程序设计的基本原则—数据封装在对象中，保护数据被直接看见，因无意识的破坏造成的错误。因此，需要尽量避免使用公共实例变量，除非是非常特殊的类。

Categories and Extensions

Category允许向一个存在的类中增加方法—即使你没有这个类的源码。这是一种强大的特性让你能够不需要继承就能扩展已经存在的类的功能。使用Category，你也能分割你自己类的实现到多个文件。类的Extension是类似的，它允许增加在本地已经实现的API的声明。

扩充类的方法

通过Category可以增加类的方法：接口文件中，在Category名字下面增加方法声明它

们，实现文件中在同样的Category名字下面定义它们，Category名字指示这些方法是附加到一个在别处声明的已存在的类上的方法，而不是定义一个新类。但是，不能使用Category增加实例变量到类中。

通过Category增加的方法将成为类的一部分。例如，通过Category给类NSArray增加方法，编译器将认为NSArray实例拥有全部的这些方法（包括通过Category增加的方法），方法增加到NSArray的子类中，NSArray类不会有这个方法（这些都只影响静态类型对象，因为只有静态类型的对象编译器才知道对象的类）。

通过Category增加的方法可以做任何主干类定义的方法做的事情，在运行时，Category增加的方法和主干类定义的方法没不同。通过Category增加的方法也会被所有子类继承，就像其他的方法一样。

Category接口声明和类接口声明非常像，有两处不同，一是在类名后增加Category名，Category名要使用小括号括起来，另一处是去掉了超类。如果你要访问类的任何实例变量，你都需要将类的主干接口文件导入到Category接口文件中：

```
#import  "ClassName.h"
@interface  ClassName  (  CategoryName  )
//  method  declarations
@end
```

Category实现文件通常导入它自己的接口文件。Category的文件命名惯例是在主干类文件名的基础上，通过“+”号与Category名字进行连接后得到新的Category文件名，一般Category的实现文件名为ClassName+CategoryName.m。实现文件的内容看上去像这样：

```
#import  "ClassName+CategoryName.h"
@implementation  ClassName  (  CategoryName  )
//  method  definitions
@end
```

注意，Category不能声明附加的实例变量，它只能增加方法。所有在类的作用范围内的实例变量也在Category的作用范围之内，即使是标记为@private的实例变量也在Category作用范围之内。

没有限制附加到类的Category的数量，但是每个Category名字都必须要被定义，每个都能声明和定义一组不同的方法。

如何使用**Categories**

使用Category就可以通过其他的实现去扩充类定义—例如，可以增加给Cocoa框架中的类增加方法。新增的方法可以被子类继承，在运行时，并不与原版的方法相区别。

Category作为子类的可替代的选择，相比通过定义子类扩展存在的类来说，通过Category增加方法更直接。例如，可以通过Category向NSArray或其他的Cocoa类增加方法，但是作为子类，你可以不需要原版类的源代码就能扩充它。

通过Category增加方法经常用于扩充类的功能或是覆盖类的继承得来的方法。Category也能够覆盖在类接口中声明的方法，但是，它不能可靠的覆盖在相同类的其他Category中

声明的方法。Category并不是子类的替代，最好不要在Category中试图去重新定义在@interface节中已经明确声明的方法。另一点需要注意的是，一个类不能定义同样的方法超过一次。

当Category覆盖继承得来的方法时，新版本的方法能跟通常一样，通过消息super包含继承版本的方法。但是没有方法可以包含定义在同样名字的类中的同名方法。

也可以通过Category将新类的实现分散到多个源代码文件中—例如，你能组织一个庞大类的方法到多个Category中，放每个Category到一个不同的文件中。当使用这个方式时，可以为程序开发带来不少好处：

提供了组织方法关系的简单方式。不同类的相似方法可以一起放在同一个源代码文件中。

简化了庞大类的管理，特别是有多个开发人员在开发这个类定义时。

能为开发庞大类提供有益的多个编辑对象。

帮助提升本地方法的使用

能配置同一个类在不同的应用中使用，并不必要在同样的代码中维护不同的版本。

Category也经常用于声明informal protocols（参见“Informal Protocols”），以及下面的讨论“Declaring Interfaces for Others to Implement”。

使用Category扩充根类

Category可以给任何类增加方法，也包括根类。增加到NSObject上的方法将让所有的类都连接到你的代码，有时这样是非常有用的，但是也是非常危险的，看上去似乎是对Category很好理解和影响很小的修改，因为继承，影响范围将非常的宽泛，你可能无意识的改变，引起不受注意的类的变化，你不知道你的改变可能会引起什么后果，另一方面，其他人也不知道不知道你的改变会对他们做的有什么影响。

另外需要注意的是，当实现根类的方法时有两个其他特别的地方：

消息发送给super是无效的（它没有超类）

类对象可以执行定义在根类中的实例方法

通常类对象只能执行类方法，但是定义在根类中的实例方法是特例，它们定义了到运行时的接口，所有的对象都继承了它们，类对象是功能完全的对象，就需要共享同样的接口。

这个特性意味当你需要去重视通过Category给NSObject增加的实例方法，它可能不仅能被实例执行，也可以很好的被类对象执行。举个例子，在方法体内，self既表示类对象也表示实例。关于根类的更多信息请参考Foundation框架中关于NSObject类的详细说明。

Extensions

类的Extension就像“匿名”的Category，不同之处在于它的方法必须在原版

@implementation块中实现。

通常类都有一组声明为公共的API，有一组声明为私有的API由类或类所在的框架使用。也可以在私有的头文件或实现文件中通过Category声明API，但是，编译器并不检查所有声明的方法是否都被实现。

例如，编译器在编译下面的声明和实现时并不出错：

```
@interface  MyObject  :  NSObject
{
    NSNumber  *number;
}
-  (NSNumber  *)number;
@end
@interface  MyObject  (Setter)
-  (void)setNumber:(NSNumber  *)newNumber;
@end
@implementation  MyObject
-  (NSNumber  *)number
{
    return  number;
}
@end
```

注意：上面的类并没有实现方法setNumber，如果它在运行时被引用，将产生一个错误。

类的Extension允许声明在本地已经实现的API，例如如下：

```
@interface  MyObject  :  NSObject
{
    NSNumber  *number;
}
-  (NSNumber  *)number;
@end
@interface  MyObject  ()
-  (void)setNumber:(NSNumber  *)newNumber;
@end
@implementation  MyObject
-  (NSNumber  *)number
{
    return  number;
}
-  (void)setNumber(NSNumber  *)newNumber
{
    number  =  newNumber;
}
@end
```

在上面的例子中要注意：

在第二个@interface块中的小括号中没有名字

setNumber方法的实现出现在类的原版@implementation块中

setNumber方法的实现必须出现在类的原版@implementation块中（不能在Category中实现），如果在类的原版@implementation块中没有这个方法的实现，编译器将警告不能找到

setNumber方法的定义。

属性

Objective-C的“声明式属性”特性为对象提供了直接的简单的实现方式。它也为存取对象属性提供了一套简明的语法，而不需要写存取方法。

概述

关于属性有两个方面的说明：声明属性的语法、访问属性的语法。这个章节将给出这两个方面的概要、讨论在这个特性之后的动机。

动机

通常访问实例变量都是通过一对存取方法（getter/setter），可以考虑使用存取器类型的API来管理类的实例的属性，通过使用存取方法，可以维护封装的原则，能够通过getter/setter“方法对”紧密的控制其行为，使实例变量与用户的使用相隔绝。

使用存取器方法有很多好处，但是写存取器方法却是件非常单调的事情—特别是你必须写代码去支持垃圾收集和内存环境管理。另一方面，属性方法的重视程度不够，像存取方法是否是现成安全的、新值在什么时候设置等等

Objective-C的声明式属性特性有下面几项益处：

属性声明提供了清晰、详细的存取方法的行为
编译器能根据提供的特殊的声明自动生成存取器方法，可以写和维护更少的代码
属性使用标识作为静态的方式被描绘和限定，因此编译器能检测到在使用但未声明的属性
运行时系统可以通过类声明的属性进行反射

属性声明

属性声明实际上是通过简单的标识为类的实例的属性声明getter/setter方法，它也可以提供详细的存取器行为。除了声明自己外，实现指令通知编译器去自动生成存取器方法实现，通知编译器在运行时要提供的方法。

使用编译器指令@property去声明一个属性，它可以出现在类、Category和Protocol的方法声明节的任何地方。

通过在@implementation块中的@synthesize和@dynamic指令触发特殊的编译器行为：@synthesize指示编译器去自动生成相关的存取方法实现；@dynamic通知编译器在运行时提供的方法。

下例声明了属性value，用@synthesize指令告诉编译器自动生成与声明给出的说明相匹配的存取方法实现。

```
@interface MyClass : NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end
@implementation MyClass
@synthesize value;
@end
```

属性访问

使用点语法就可以简单的访问属性，就像访问数据结构的元素一样。

```
MyClass *myInstance = [[MyClass alloc] init];
myInstance.value = @"New value";
NSLog(@"myInstance value: %@", myInstance.value);
```

圆点语法是纯的“syntactic sugar”，它有编译器转换为使用存取方法（并不是直接访问实例变量），下面的代码等效于上面的实现：

```
MyClass *myInstance = [[MyClass alloc] init];
[myInstance setValue:@"New value"];
NSLog(@"myInstance value: %@", [myInstance value]);
```

属性声明和实现

属性包含了两部分：属性声明和属性实现。

属性声明

属性声明使用关键字@property开始，@property可以出现在类的@interface块的方法声明的任何地方，@property也可以出现在Category和Protocol中。

```
@property(attributes) type name;
```

@property标明是在声明属性，后面的小括号中提供了参数列表（attributes），参数列表提供了属性附加的关于存取和其他行为的信息，参数的可选值参见章节“属性声明的参数”。象其他的Objective-C类型一样，每个属性都有一个明确的类型和名字。

属性实现指令

通过在@implementation块中的@synthesize和@dynamic指令触发特殊的编译器行为。注意的是两者都不请求任何被给的@property声明。

```
@synthesize
```

使用@synthesize关键字是告诉编译器，如果在@implementation块中不提供属性的getter/setter方法实现，它需要自动为属性生成getter/setter方法的实现。

```
@interface MyClass : NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end
// assume using garbage collection
@implementation MyClass
@synthesize value;
@end
```

可以通过形式property=ivar指示一个特别的实例变量被用于属性，例如：

```
@synthesize firstName, lastName, age = yearsOld;
```

firstName、lastName和age的存取方法的实现将被编译器自动生成，属性age被描述成实例变量yearsOld，其他方面的自动生成方法依赖于参数选项（参见章节“属性声明的参数”）。

行为的不同依赖运行时（参见“运行时差别”）：

对于老的运行时，实例变量必须要已经被声明在@interface块中。如果实例变量和属性同名且类型相容，它是可以用的，否则，将出现一个编译错误。

对于现代的运行时，实例变量在自动生成时被需要，如果存在同名的实例变量，它就可用。

```
@dynamic
```

使用@dynamic关键字告诉编译器将实现暗含属性的API契约，而不是直接提供方法的实现，或者在运行时采用其他的机制，例如动态代码加载，或是动态方法方案。下面的例子是直接方法实现，它等效于上面的例子：

```
@interface MyClass : NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end
// assume using garbage collection
@implementation MyClass
@dynamic value;
- (NSString *)value {
    return value;
}
- (void)setValue:(NSString *)newValue {
    if (newValue != value) {
        value = [newValue copy];
```

```
    }
}
@end
```

属性声明的参数

可以通过属性参数修饰属性，使用属性参数的形式为@property(attribute [, attribute2,...])。象方法一样，属性的范围是在接口声明的范围之内。属性声明可以使用逗号分隔属性列表中的不同属性名，属性的参数将修饰所有的这些属性。如果你要使用垃圾收集器，你需要在属性声明中使用存储修饰符__weak和__strong，但它并不是属性参数列表形式上的一部分。

```
getter=getterName, setter=setterName
```

getter=和setter=指定了属性存取器方法的名字，getter不带参数，返回与属性类型相匹配的类型，setter带一个与属性类型相匹配类型的参数，返回void。

默认的名字分别为*propertyName*和*setPropertyName:*，例如，属性foo，存取器的名字分别为foo和setFoo:。指定特别的存取器方法名需要符合值对规则，当属性类型为boolean型时，getter通常指定为*isPropertyName*。

```
readonly
```

指示属性是只读的，默认情况下属性是可读写的。

如果指定属性为readonly，只有getter要求在@implementation块中出现，或是使用@synthesize指令仅生产getter，因此，让你试图用点操作符去给它赋值，将得到一个编译错误。

```
readwrite
```

指示属性可以被读写，这是默认值。

在@implementation块中要求getter、setter方法都要有，或者是使用@synthesize指令生成getter和setter方法。

```
assign
```

指定setter方法是用简单的赋值操作，这是默认的方式。

如果应用使用垃圾收集器，要去使用采用了NSCopying protocol的类（型）的属性时，你最好能明确的使用assign进行说明，而不是简单的使用默认值。（让编译器确认你要使用assign来进行赋值操作，即使这个变量的类型是可拷贝的）。

```
retain
```

指定retain将使用对象自己的赋值方法，默认是使用assign。

这个参数仅对Objective-C对象类型有效（不能使用retain在Core Foundation对象中，参见“Core Foundation”）。

```
copy
```

指定使用对象的拷贝方式进行赋值，默认是使用assign。

赋值时将引用对象的copy方法，这个参数仅对实现了NSCopying protocol的对象类型有效，更多的讨论，请参考“Copy”。

nonatomic

指定存取操作是非原子操作，默认情况下，存取操作是原子操作。

默认情况下的原子操作，为属性的存取提供了在多线程环境下安全、严谨的操作—属性值在进行读取或设置时总是能被完整的提取和进行安全的设置，而不受其他线程的干扰，更多细节，请参考“性能和线程”。

如果你没有指定nonatomic，自动生成的存取器将包含一个被管理的内存环境，返回值的内存能被自动的释放；如果指定了nonatomic，自动生成的对象属性存取器将简单的直接返回变量值。

如果你使用@synthesize指令指示编译器去创建存取方法，它将按给出的关键字生产对应相匹配的代码。如果你自己实现存取器方法，你要担保你的实现是匹配这些说明的（例如，如果指定了copy的赋值方式，你必须确定你在setter方法中是使用copy的方式处理输入变量的）。

你可以使用getter=， setter=， readonly， readwrite这些参数参数在类的接口、Category和Protocol声明中。在参数列表中仅能使用readonly、readwrite中的一个。setter=/getter=是可选项，标识为readonly时最多只能出现一个（getter=），如果你指定了属性为readonly，同时也为setter=提供了方法，编译器将会有警告提醒。

assign、retain和copy是彼此互斥的，是否要明确的标识它们依赖是否使用了垃圾收集器：

如果没有使用垃圾收集器，针对对象属性，必须指定明确的赋值方式：assign、retain和copy，否则编译器将出现警告（编译器强迫你去考虑你要的内存管理方式，并要你明确的去声明它）。

如果使用了垃圾收集器，你可以使用默认的方式（也就是说，你可以不必声明赋值方式为assign、retain还是copy），除非是属性的类型是一个类，并且这个类符合NSCopying。默认值通常能满足你的需要，但是，如果属性的类型是能够拷贝的，为了维护封装性，经常需要制作对象的一个私有拷贝。

Copy

如果在属性参数中指定copy参数，就是指定在赋值过程中使用拷贝方式。如果自动生成相对应的存取器，自动生成的方法中将使用copy方法，这样的方法对象字符串这样的属性是非常有用的，它们往往要求通过setter得到的新值是可以改变（例如，NSMutableString实例），你要确认你的对象有私有的可变（？？原文是不可变immutable）的copy方法。例如，你声明一个属性如下：

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

自动生成的setter方法和下面的代码类似：

```
-(void)setString:(NSString *)newString
{
    if (string != newString)
```

```
        {
            [string  release];
            string  =  [newString  copy];
        }
    }
```

上面的方法针对字符串能更好的施行，但是如果属性是一个聚集，如数组或是集合，可能会有问题，通常的做法是要使聚集有可变的能力，但是copy方法返回一个不可变(immutable)的聚集版本，在这种情况下，必须提供自己的setter方法实现，就像下面的代码一样：

```
    @interface  MyClass  :  NSObject
    {
        NSMutableArray  *myArray;
    }
    @property  (nonatomic,  copy)  NSMutableArray  *myArray;
    @end
    @implementation  MyClass
    @synthesize  myArray;
    -  (void)setMyArray:(NSArray  *)newArray
    {
        if  (myArray  !=  newArray)
        {
            [myArray  release];
            myArray  =  [newArray  mutableCopy];
        }
    }
    @end
```

属性重载

在子类中可以重载属性，但是在子类中必须完整的拷贝先祖的属性参数（readonly和readwrite是个例外，子类属性的这个参数可以和先祖类不一样），在Category和Protocol中以同样的规则来重载属性—当属性在Category和Protocol中重载时，属性参数需要被完整的复制。

如果声明一个类的属性为readonly，可以在类的Extension、Protocol、子类中声明它是readwrite的。对于在类的Extension中重载，被重载的属性要在任何能引起自动生成setter方法的代码的@synthesize语句之前。重载一个只读属性为读写属性，这为类提供了两种实现模式：为一个不可变的类提供一个可变的子类（NSString、NSArray、NSDictionary都是这样的例子）；属性的公共访问API是只读的，私有的能访问类的内部的实现是读写的。下面的例子显示使用类的Extension为在公共头中声明为只读的属性重载为在私有领域声明为读写：

```
    //  public  header  file
    @interface  MyObject  :  NSObject
    {
        NSString  *language;
    }
```

```
@property (readonly, copy) NSString *language;
@end
// private implementation file
@interface MyObject ()
@property (readwrite, copy) NSString *language;
@end
@implementation MyObject
@synthesize language;
@end
```

性能与线程

如果是自己提供属性的方法实现，那么属性的声明对效率和线程安全没有影响。

如果让编译器自动生成属性的方法实现，方法的实现依赖于提供的属性声明，属性参数中影响性能和线程安全的有retain、assign、copy和nonatomic，前三个参数只是影响setter方法的赋值实现部分，就象下面的例子一样（注意实现只是作为示例，并不严谨）：

```
// assign
property = newValue;
// retain
if (property != newValue)
{
    [property release];
    property = [newValue retain];
}
// copy
if (property != newValue)
{
    [property release];
    property = [newValue copy];
}
```

参数nonatomic的影响决定于环境，默认情况下，自动生成的存取器是原子的。在被管理的内存环境下，原子操作都通过锁来进行控制，另外，返回对象被保持，并且能被自动释放。如果这样的存取器被频繁的引用，可能会出现性能方面的问题。在垃圾收集器环境，大部分自动生成的方法都是原子的，并不考虑性能方面的开销。

理解方法的原子实现的目的是为了提供健壮的存取器是非常重要的--代码的正确性无法保证。即使“atomic”表示属性是线程安全的，简单的标识所有的属性为atomic的也不意味这这个类或者对象是线程安全的，线程安全并不仅局限在存取器方法上。关于更多的多线程信息，请参考*Threading Programming Guide*。

标记

属性支持整个范围内的C风格装饰，属性可以通过__attribute__风格的标记标识为支持或不支持，就像下面的例子一样。

```
@property CGFloat x
AVAILABLE_MAC_OS_X_VERSION_10_1_AND_LATER_BUT_DEPRECATED_IN_MAC_OS_X_VERSION_10_4;
```

```
@property  CGFloat  y  __attribute__((...));
```

核心框架

在“属性实现指令”节中，不能为非对象类型指定retain　参数，因此如果要声明类型为CFType 的属性时，自动生成的存取器代码如下面的例子：

```
@interface  MyClass  :  NSObject
{
    CGImageRef  myImage;
}
@property(readwrite)  CGImageRef  myImage;
@end
@implementation  MyClass
@synthesize  myImage;
@end
```

在被管理的内存环境下，编译器生成的seter存取器将只是简单的分配新值给实例变量（新值并不能被保留，老值没有被释放），这通常是错误的，因此，你不能通过自动方式来实现存取方法，你要自己去实现这些方法。

在垃圾收集环境下，如果变量被声明为__strong：

```
...
__strong  CGImageRef  myImage;
...
@property  CGImageRef  myImage;
```

存取器将被适当的生成—myImage将不能被保留，setter 操作将触发一个写阻碍。

例子

下面的例子阐明了使用属性的几个不同方式：

- *Link* Protocol 声明了一个属性*next*。
- *MyClass*采用了*Link*　Protocol，如是也暗含的声明了属性*next*，*MyClass*也声明了几个其他的属性。
- *creationTimestamp*和*next*通过编译器自动生成存取方法，但是使用与存在的实例变量不一样的名字。
- *name*通过编译器自动生成存取方法，使用实例变量自动生成（在32位运行时系统中，实例变量自动生成不被支持—详细参见“属性实现指令”和“运行时系统差异”）。
- *gratuitousFloat* 指示为dynamic—表示直接使用方法的实现。
- *nameAndAge*虽然并没指示为dynamic，但是默认就是，它要求直接指定明确的方法实现（由于它是只读的，所以只要求getter方法）。

```
@protocol  Link
@property  id  <Link>  next;
@end
```

```
@interface MyClass : NSObject <Link>
{
    NSTimeInterval intervalSinceReferenceDate;
    CGFloat gratuitousFloat;
    id <Link> nextLink;
}
@property(readonly) NSTimeInterval creationTimestamp;
@property(copy) __strong NSString *name;
@property CGFloat gratuitousFloat;
@property(readonly, getter=nameAndAgeAsString) NSString *nameAndAge;
@end
@implementation MyClass
@synthesize creationTimestamp = intervalSinceReferenceDate, name;
// synthesizing 'name' is an error in legacy runtimes
// in modern runtimes, the instance variable is synthesized
@synthesize next = nextLink;
// uses instance variable "nextLink" for storage
@dynamic gratuitousFloat;
// will warn unless -gratuitousFloat and -setGratuitousFloat: occur in
@implementation
- (CGFloat)gratuitousFloat
{
    return gratuitousFloat;
}
- (void)setGratuitousFloat:(CGFloat)aValue
{
    gratuitousFloat = aValue;
}
- (NSString *)nameAndAgeAsString
{
    return [NSString stringWithFormat:@"%@ (%fs)", [self name],
                [NSDate timeIntervalSinceReferenceDate] -
                intervalSinceReferenceDate];
}
- init
{
    if (self = [super init])
    {
        intervalSinceReferenceDate = [NSDate timeIntervalSinceReferenceDate];
    }
    return self;
}
@end
```

点语法

Objective-C为引用存取方法提供了紧凑和便利的点操作，当你要获取或修改另一个对象的属性时，使用它非常的有用。

概述

下面的例子演示了读写属性：

```
Graphic *graphic = [[Graphic alloc] init];
NSColor *color = graphic.color;
CGFloat xLoc = graphic.xLoc;
BOOL hidden = graphic.hidden;
int textCharacterLength = graphic.text.length;
if (graphic.textHidden != YES)
{
    graphic.text = @"Hello";
}
graphic.bounds = NSMakeRect(10.0, 10.0, 20.0, 120.0);
```

获取属性值通过呼叫属性同名的方法（默认情况下，getter方法与属性名同名），设置属性值通过属性的setter方法（默认情况下，方法名是在属性名前加上set，后面加上冒号）。事实上，只要满足getter/ setter方法的规范，即使没有声明它是一个属性，也可以通过点语法引用这些方法（如果没有声明为属性，getter/ setter方法必须被命名为*property*和*setProperty:*（property代表你点后面的名字））。点语法维护了类的封装—你不能直接访问实例变量。

下面的代码等效于上面的那段代码，但使用的是客户化的存取器（你也能实现你自己的方法而不用自动化生成的）。

```
Graphic *graphic = [[Graphic alloc] init];
NSColor *color = [graphic color];
CGFloat xLoc = [graphic xLoc];
BOOL hidden = [graphic hidden];
int textCharacterLength = [[graphic text] length];
if ([graphic isTextHidden] != YES)
{
    [graphic setText:@"Hello"];
}
[graphic setBounds:NSMakeRect(10.0, 10.0, 20.0, 120.0)];
```

虽然属性表达式在功能上等效于消息的发送，但是编译器在检测到向一个只读属性写数据时，会提示错误。在引用不存在的*setProperty:*方法时，消息发送则只会产生一个未定义方法的警告，在运行时会引发失败。

属性类型为C语言类型时，属性支持复合运算。下面的示例就是利用符合运算来更新NSMutableData实例的length属性：

```
NSMutableData *data = [NSMutableData dataWithLength:1024];
data.length += 1024;
data.length *= 2;
data.length /= 4;
```

上面的例子等效于：

```
[data setLength:[data length] + 1024];
[data setLength:[data length] * 2];
[data setLength:[data length] / 4];
```

下面的例子示例了属性不能使用的一种情况：

```
id  y;
x  =  y.z;    //  z  并没有被声明为属性
```

上例中y并没有指明其静态类型，z并没有被声明为属性。处理这种情况有几种方式，如果有歧义，就引发一个未声明的属性错误。如果z被声明，并且是在当前编译单元中唯一被声明的z属性，那它就不是有歧义的。如果有多个z属性声明，并且都有相同的类型（如BOOL），那么它也是合法的。声明为readonly，而另一个没声明，也会引起歧义。

值nil

如果属性被赋nil值，结果和给nil发送消息等效，下面的例子就是等效的：

```
//  each  member  of  the  path  is  an  object
x  =  person.address.street.name;
x  =  [[[person  address]  street]  name];
//  the  path  contains  a  C  struct
//  will  crash  if  window  is  nil  or  -contentView  returns  nil
y  =  window.contentView.bounds.origin.y;
y  =  [[window  contentView]  bounds].origin.y;
//  an  example  of  using  a  setter....
person.address.street.name  =  @"Oxford  Road";
[[[person  address]  street]  setName:  @"Oxford  Road"];
```

self

要去通过存取器访问自己的属性，要明确的使用self标识出来：

```
self.age  =  10;
```

如果没有使用self.，你是在直接访问实例变量，在下面的例子中，age属性的setter方法并没有被引用：

```
age  =  10;
```

性能和线程

点语法产生的代码完全等效于标准方法，执行效率等效于直接引用存取方法，也没有额外的线程依赖。

惯例

```
aVariable  =  anObject.aProperty;
```

引用aProperty方法，返回值给aVariable变量赋值，属性aProperty的类型和aVariable变量的类型必须是相容的，否则编译器会报警告。

```
anObject.name  =  @"New  Name";
```

引用anObject对象上的setName:方法，传入参数为@"New Name"。

如果setName:方法不存在，或是属性不存在，或是setName:的返回值类型不是void，编

译器会报警告。

```
xOrigin = aView.bounds.origin.x;
```

引用bounds方法，给xOrigin赋值，origin是bounds方法返回的NSRect类型的结构，x是返回结构的一个元素值。

```
NSInteger i = 10;
anObject.intProperty = anotherObject.floatProperty = ++i;
```

将11赋值给anObject.intProperty和anotherObject.floatProperty，等式右边的先计算，值通过setFloatProperty:和setIntProperty:进行设置，在每个等式中，都是等式右边的先计算。

不良习惯

下面的样例是非常糟糕的：

```
anObject.retain;
```

会引起一个编译警告（`warning: value returned from property not used.`）

```
/* method declaration */
- (BOOL) setFooIfYouCan: (MyClass *)newFoo;
/* code fragment */
anObject.fooIfYouCan = myInstance;
```

上面的例子将产生一个编译警告，setFooIfYouCan:方法看上去并不像是一个setter方法，因为它的返回值类型不是void。

```
flag = aView.lockFocusIfCanDraw;
```

引用lockFocusIfCanDraw方法，将返回值赋值给flag，但是如果返回值类型和flag类型不一致，将产生一个编译警告。

```
/* property declaration */
@property(readonly) NSInteger readonlyProperty;
/* method declaration */
- (void) setReadonlyProperty: (NSInteger)newValue;
/* code fragment */
self.readonlyProperty = 5;
```

上面的代码中，因属性被声明为readonly，将产生一个编译警告（`warning:assignment to readonly property 'readonlyProperty'`），在运行时，它可以执行，但是简单的给属性增加一个setter方法，并不必然标识属性为readwrite。

属性和键值对编码

在Cocoa中，有两种方式可以访问对象的属性：静态方法，通过点语法直接引用存取方法；动态方法，通过使用键值对（key-value coding，简称KVC）。KVC定义了一般的属性存取方法：valueForKey:和setValue:forKey:，通过把属性名作为字符串键值作为参数来操纵属性值。KVC并不是一种通常意义的存取方法，它经常是在没有其他选择的情况下使用（在编译时代码中并不知道属性的确切名字）。

KVC和声明式属性是垂直技术，两种方式可以交叉使用。声明式属性使用点语法，

KVC使用键值路径的方式进行访问。需要注意的是，通过点语法直接访问属性，是引用了接收者的标准存取方法（依此可以推导出：在KVC方法valueForKey:和setValue:forKey:中并没有引用点语法）。

下面用例子举例说明使用KVC方法访问属性：

```
@interface MyClass
@property NSString *stringProperty;
@property NSInteger integerProperty;
@property MyClass *linkedInstance;
@end
```

在下面的实例中使用KVC访问它的属性：

```
MyClass *myInstance = [[MyClass alloc] init];
NSString *string = [myInstance valueForKey:@"stringProperty"];
[myInstance setValue:[NSNumber numberWithInt:2] forKey:@"integerProperty"];
```

下面的例子演示了点语法和通过KVC路径访问属性的不同：

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;
```

等同的KVC语法为：

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:[NSNumber numberWithInt:2]
forKeyPath:@"linkedInstance.integerProperty"];
```

子类的属性

在子类中，可以修改超类的readonly属性为可写的。例如，可以定义MyInteger类的属性value为只读的：

```
@interface MyInteger : NSObject
{
    NSInteger value;
}
@property(readonly) NSInteger value;
@end
@implementation MyInteger
@synthesize value;
@end
```

在子类MyMutableInteger中，可以重置这个属性为可写的：

```
@interface MyMutableInteger : MyInteger
@property(readwrite) NSInteger value;
@end
@implementation MyMutableInteger
@dynamic value;
- (void)setValue:(NSInteger)newX
{
    value = newX;
}
@end
```

属性反射

编译器编译属性声明时，将产生包括类定义、Category定义、Protocol定义的属性描述元数据（metadata）。可以通过函数查看指定类名或Protocol名的元数据，获得以@encode字符串形式表示的属性类型，以C字符串数组方式拷贝属性参数列表，在类或Protocol中，声明式属性都是可见的。

属性结构定义为一个不透明的句柄，对属性结构进行了描述：

```
typedef struct objc_property *Property;
```

函数class_getProperty和protocol_getProperty分别可以通过名字查询类和Protocol的属性：

```
Property class_getProperty(Class cls, const char *name);
Property protocol_getProperty(Protocol *proto, const char *name);
```

函数class_copyPropertyList以动态分配的数组指针方式返回类中的所有属性。protocol_copyPropertyList返回的是Protocol中的所有属性。

```
Property* class_copyPropertyList(Class cls, uint32_t *count);
Property* protocol_copyPropertyList(Protocol *protocol, uint32_t *count);
```

函数property_getInfo返回属性名和属性类型，属性类型是以@encode类型字符串表示的。

```
void property_getInfo(Property *property, const char **name, const char
**type);
```

函数property_copyAttributeList返回属性在编译时的参数列表，列表是动态内存分配的以C字符串指针数组。

```
const char **property_copyAttributeList(Property *property, unint32_t
*count);
```

运行时差异

一般的属性行为在所有的运行时上都是一致的。在现代的运行时（(64-bit）与老运行时（32-bit）有一个关键的差异：现代的运行时支持更强健的实例变量。

在老的运行时中，必须在@interface块声明与属性同名，类型兼容的实例变量，所有的访问这个类的其他文件都要包含这个接口，子类必须知道这个类的全部存储细节，实例变量的自动生成存取代码不被支持（除非你定义了同名的、类型兼容的属性）。

例如：

```
@interface My32BitClass : NSObject
{
    CGFloat gratuitousFloat;
}
@property CGFloat gratuitousFloat;
@end
@implementation My32BitClass
@synthesize gratuitousFloat; // uses the instance variable
"gratuitousFloat" for storage
@end
```

```
    @interface  My64BitClass  :  NSObject
    {  }
    @property  CGFloat  gratuitousFloat;
    @end
    @implementation  My64BitClass
    @synthesize  gratuitousFloat;  //  synthesizes  the  instance  variable
"gratuitousFloat"  for  storage
    @end
```

Protocol

Protocol声明的方法可以被任何类来实现，协议至少常用在下面三种情形下：

声明期待其他类去实现的方法

声明接口到匿名对象

不通过继承关系分类相似类

声明其他类去实现的接口

类和Category接口声明的方法都是和特定的类进行关联的，这是类实现的主要方法。另一方面，正式（formal）和非正式（Informal）Protocol声明的方法并不和类相关，但是一个或多个类都可能实现了Protocol声明的方法。

Protocol是一组简单的方法声明列表，但并不和任何类进行关联。例如，将鼠标的用户行为方法汇集到一个Protocol中：

```
-  (void)mouseDown:(NSEvent  *)theEvent;
-  (void)mouseDragged:(NSEvent  *)theEvent;
-  (void)mouseUp:(NSEvent  *)theEvent;
```

任何要去响应鼠标事件的类都要采用这个Protocol并且实现这些方法。

Protocol将方法声明从类的继承树上分离出来，因此它能使用在很多类和Category不适用的地方。Protocol列出了在其他地方（可能）实现的方法，但是是谁实现的并不感兴趣，无论哪个类感兴趣，都需要遵从（conform）这个Protocol—不管它有没有实现协议声明的方法，对象根据类型来划分，可以基于它们继承自相同的类而具有相似性，也可以基于它们遵从（conform）相同的接口具有相似性。类并不在继承树分支上，但是因为遵从相同的接口而被划为一类。

Protocol在面向对象的设计中扮演重要的角色，特别是一个项目被分割成许多个模块来实现，或是项目包含在其他项目中开发的对象。Cocoa使用它们通过Objective-C消息进行大量的进程间的通信。

然而，Objective-C程序并不一定要使用Protocol，不像类定义和消息表达式，Protocol是可选的，一些Cocoa框架使用了它们，另一些不适用它们，使不使用，完全取决于项目本身。

其他类去实现的方法

如果知道一个对象的类，能够找出接口声明（也能从接口声明中找到它从哪里继承来），找出它能响应哪些消息，声明公开了它能接收的消息，Protocol也提供了一种公开它能发送的消息的方式。

通讯工作的两种方式：发送消息和接收消息。例如，一个对象可能委托一个特定的操作到另一个对象上，或是它询问另一个对象一些信息。在一些案例中，一些对象可能需要向其他对象通告它的行为。

如果消息的发送类和接收类都在同一个项目中（或者是有人提供给你接收者的接口文件），这样的通讯是容易协调的，发送类简单的导入接收类的接口文件就可以了，导入的接口文件已经声明了发送者在消息中使用的方法。

然而，如果开发的对象发送消息到一个还未定义的对象—对象留给其他人去实现—你没有接收者的接口文件，你需要使用使用其他的方式去声明它但是不实现它，Protocol满足了这种意图，它通知编译器类在使用的方法，以及定义这些方法的文件要一起参与。

例如，开发一个对象，要求其他对象通过发送helpOut:或其他消息获得协助，提供一个assistant实例变量去记录消息的输出，定义了一个方法去设置实例变量，该方法让其他对象注册它们自己为你的对象消息的潜在接收者：

```
- setAssistant:anObject
{
    assistant = anObject;
}
```

当一个消息被发送到assistant时，需要检查接收者是否实现了应答的方法：

```
- (BOOL)doWork
{
    ...
    if ( [assistant respondsTo:@selector(helpOut:)] ) {
        [assistant helpOut:self];
        return YES;
    }
    return NO;
}
```

有时在你写代码的时候，你并不知道对象可以注册它自己作为assistant，也不能导入实现它的类的接口文件，仅能为helpOut:方法声明一个Protocol。

为匿名对象声明接口

Protocol可以用于声明匿名对象（anonymous object），不知道类的对象被称为匿名对象。匿名对象可能描绘一项服务，或是提供一组函数，特别是在对象的类型并不需要被知道的时候。（对象在定义应用的基础架构方面提供了基础的角色，对象在使用前必须被初始化对匿名对象并不是一个好的选项）。

对象对开发者来将并不是匿名的，但是当开发者提交它给其他人使用时，它们是匿名的，例如下面几种情况：

框架或是提供现成对象给其他人使用的包中，包含的对象并没有使用类名来标识，或者没有接口文件。缺少类名和类接口，使用者无法创建类的实例，作为替代，提供者必须提供一个已经准备好的实例。典型的是在另一个类的方法中，返回一个可用的对象：

```
id formatter = [receiver formattingService];
```

通过上面的方法返回的对象是没有类标识的对象，没有任何提供者被显现出来，但是，这样的对象是根本没用的，提供者至少要标识出它能响应哪些消息，可以通过把这个对象与Protocol建立关联，Protocol中声明一组方法的方式达成这一意图。

可以发送Objective-C消息给远程对象（remote object）--远程对象是指在其他应用中的对象（详细参见“远程对象”）。

每个应用有它自己的数据结构、类和内部逻辑，但是你并不需要知道其他应用内部是如何工作的，也不需要知道它内部是如何通讯的。作为一个局外者，你所要知道的是你能给它发送哪些消息（Protocol），发送给谁（Receiver）。

应用公布它的对象中的一个作为远程消息的潜在接收者，这个对象必须公开它能响应的方法（Protocol），它不需要公布其他的任何信息。发送消息的应用不需要知道这个对象的类，也不需要在它自己的设计中使用这个类，所有它要知道的只有Protocol。

Protocol使匿名对象变得可能。没有Protocol，就没有办法可以声明一个对象接口而不标识它的类。

非继承的相似

如果多个类实现同一组方法，这些类经常被组织在一个基类下，基类声明了它们共有的方法。每个子类可能以它自己的方式重新实现这些方法。通过继承和在基类中的公共声明捕获子类的相似性。

但是，有时，汇聚公共方法到基类中，有时是不可能的。类的大部分细节都没有关联，但是却要去实现一些相似的方法，有限的相似性不足以建立继承关系。例如，要支持应用中的对象创建XML的描述和从XML描述初始化对象：

```
- (NSXMLElement *)XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)xmlString;
```

这些方法能被组织到一个Protocol中，类的相似性除了遵从同样的Protocol外，没有其他的任何东西。

对象能通过这些相似性被分类（它们所遵从的Protocol）。例如，一个NSMatrix实例必须要和描绘它的cell对象进行通讯，一个NSMatrix可以要求每一个cell对象都是NSCell类型的（基于基类），所有的从NSCell类继承的对象对能对NSMatrix的消息进行响应。另一方面，NSMatrix对象可以要求每个cell对象都有一组方法能响应特定的消息（基于Protocol），在这种情况下，NSMatrix不必关系这个cell对象属于哪个类，以及它如何去实现这些方法。

正式Protocol

Objective-C语言提供了一种方法正式的声明一组方法为一个Protocol。正式Protocol由语言和运行时支持。例如，编译器能检测基于Protocol的类型，对象在运行时能通过反射报告它们是否遵从了Protocol。

声明Protocol

通过@protocol指令声明正式Protocol：

```
@protocol  ProtocolName
method  declarations
@end
```

例如，可以声明XML描述Protocol如下：

```
@protocol  MyXMLSupport
-  (NSXMLElement  *)XMLRepresentation;
-  initFromXMLRepresentation:(NSXMLElement  *)XMLElement;
@end
```

跟类名不同，Protocol名并不是全局可见的，它们存在于它们自己的命名空间中。

Protocol方法选项

Protocol方法可以通过使用@optional关键字标记方法为可选的，相对应于@optional关键字，@required关键字指示方法是必须的，@required是默认行为。可以通过@optional和@required关键字将Protocol的方法声明分割为两部分。

```
@protocol  MyProtocol
-  (void)requiredMethod;
@optional
-  (void)anOptionalMethod;
-  (void)anotherOptionalMethod;
@required
-  (void)anotherRequiredMethod;
@end
```

非正式Protocol

除正式Protocol外，可以通过在Category中声明非正式Protocol组织方法：

```
@interface NSObject ( MyXMLSupport )
- (NSXMLElement *) XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
@end
```

非正式Protocol通常是作为NSObject类的Category进行声明的，这样任何继承自NSObject类的方法都能产生联系。因为所有的类都从根类继承，方法并没有被限制在任何继承关系上（也可以声明非正式Protocol在其他类的Category上，以此限制它的范围在一个有限的继承分支上，但是很少有原因如此做）。

在声明Protocol时，Category接口并没有相对应的实现，实现这个Protocol的类再一次的在它自己的接口文件中声明了这些方法，和其他的方法一起在它们的实现文件中进行定义。

非正式Protocol集中Category规则去声明一组方法，但并不和任何特定的类联系，也不实现它们。

因为是非正式Protocol，在Category中声明的Protocol并没有收到任何语言上的支持，在编译时没有类型检查，在运行时也没有检查对象是否遵从Protocol。如果要获得这些好处，你需要使用正式Protocol。非正式Protocol在所有的实现方法都是可选的时候可能有用，例如委托，但是（在Mac OS X v10.5或之后的版本）通常使用正式Protocol加上可选选项会更好。

Protocol对象

就像类对象是在运行时描述类的对象，正式Protocol也有一个特殊的数据类型—Protocol类实例来描述。处理Protocol的源代码（更多的用于类型描述）必须提交到Protocol对象。

在许多方面，Protocol和类定义是相似的，它们都声明了方法，在运行时，它们都由对象来表现—类通过类对象，Protocol通过Protocol对象。跟类对象一样，Protocol对象通过运行时系统根据定义和声明它的源代码自动创建，它们在源代码中并不需要分配和初始化。

源代码能够使用@protocol()指令引用到Protocol对象--同样的指令用于声明一个Protocol，不同之处是，这里需要一对圆括号，圆括号中填Protocol名。

```
Protocol *myXMLSupportProtocol = @protocol(MyXMLSupport);
```

这是源代码中能引用Protocol的唯一方式，跟类名不同，Protocol名并不能代表Protocol对象，除非是通过@protocol()指令。

编译器为每个碰到的Protocol创建一个Protocol对象，但要求Protocol满足下面的条件之一：

被一个类采用
源代码中有地方采用（使用@protocol()）

仅被声明没有被使用的Protocol在运行时并没有Protocol对象来描述。

采用Protocol

采用Protocol和声明超类有些相似，都是为类指定方法，超类声明了它继承的方法，Protocol声明了在Protocol列表中的方法。一个类如果在它的Protocol声明列表中包含了某个Protocol，则说明这个类遵守了某个正式Protocol。Protocol声明在超类名后使用尖括号括起来。

```
@interface ClassName : ItsSuperclass < protocol list >
```

Category采用Protocol使用下面的方式：

```
@interface ClassName ( CategoryName ) < protocol list >
```

一个类采用多个Protocol，Protocol名之间通过逗号分隔。

```
@interface Formatter : NSObject < Formatting, Prettifying >
```

类和Category采用一个Protocol，就必须实现所有在Protocol中声明的要求实现的方法，否则，编译器将提示一个警告。类Formatter必须实现在两个Protocol中声明的全部要求实现的方法，除此之外的，就是实现它自己声明的方法。

类或Category采用某个Protocol，就必须导入Protocol声明所在的头文件。声明在Protocol中的方法不需要再在类和Category接口中声明。

简单的采用Protocol而不声明其他的方法是可能的。例如，下面的类声明采用Formatting和Prettifying两个Protocol，但并没有声明自己的实例变量和方法：

```
@interface Formatter : NSObject < Formatting, Prettifying >
@end
```

遵守一个Protocol

如果一个类采用了某个Protocol，或是继承的类采用了这个Protocol，就可以说这个类遵守这个Protocol。类的实例被说成遵守它的类同样遵守的Protocol集。

因为类必须实现所有它采用的Protocol的要求实现的方法，说一个类或一个实例遵守某个协议等价于它拥有这个协议声明的所有方法。

通过发送conformsToProtocol:消息检测对象是否遵守Protocol。

```
if ( ! [receiver conformsToProtocol:@protocol(MyXMLSupport)] ) {
    // Object does not conform to MyXMLSupport protocol
    // If you are expecting receiver to implement methods declared in
the
    //   MyXMLSupport protocol, this is probably an error
}
```

（注意，类也有这样一个同名的方法：conformsToProtocol:）。

conformsToProtocol:检测更像检测单个方法的respondsToSelector:，检测一个Protocol是否被遵守（去检测所有的方法是否被实现）跟一个特定的方法是否被实现是一样的。因为conformsToProtocol:检测在Protocol中的所有的方法，所以它比respondsToSelector:更有效。

conformsToProtocol:检测也跟isKindOfClass:检测相似，除了conformsToProtocol:检测的类型是基于Protocol，而isKindOfClass:检测时基于类的继承关系的。

类型检查

对象类型的声明中包含了正式Protocol，为编译器检查类型是否包含Protocol提供了可能。

在类型声明中，Protocol名在类型名的后面，用尖括号括起来：

```
- (id <Formatting>)formattingService;
id <MyXMLSupport> anObject;
```

作为静态类型，编译器就可以去检查基于类继承的类型，上面的语法可以让编译器去检查类型是否遵守Protocol。

例如，如果Formatter是一个抽象类，下面的声明：

```
Formatter *anObject;
```

分组所有的从Formatter继承的对象为一个类型，可以让编译器去检查按这个类型分配的对象是在这个类的继承关系上的。

类似的，下面的声明：

```
id <Formatting> anObject;
```

分组所有的遵守Formatting Protocol的对象为一个类型，不管它们在类的继承树上的哪个位置，编译器确认按这个类型分配的对象是遵循这个Protocol。

在每种情况下，类型总是分组相似的对象—无论是它们要共享一个公共的继承，还是因为它们汇聚是因为一组公共的方法。

这两个类型可以合并在一个单个声明中：

```
Formatter <Formatting> *anObject;
```

Protocol不能作为类对象的类型，仅实例的静态类型可以被标明为Protocol的，也只有实例的静态类型能被标明为类。（然而，在运行时，类和实例都能响应conformsToProtocol:消息）。

Protocol包含（继承）

一个Protocol能包含另一个Protocol，使用类似于类采用Protocol的语法：

```
@protocol ProtocolName < protocol list >
```

所有列在尖括号中的Protocol是ProtocolName Protocol的一部分。例如，如果Paging Protocol包含了Formatting Protocol：

```
@protocol Paging < Formatting >
```

任何遵循Paging Protocol的对象也将遵循Formatting，类型声明

```
id <Paging> someObject;
```

和conformsToProtocol:消息

```
if ( [anotherObject conformsToProtocol:@protocol(Paging)] )
    …
```

只需要检测是否遵循Paging Protocol也就已经检测了是否遵循Formatting。

当类采用了一个Protocol，它必须实现Protocol要求实现的全部方法，需要注意的是，如果这个Protocol还包含了其他的Protocol，这个类必须实现它包含的所有的Protocol要求实现的方法，一个类需要遵循一个被包含的Protocol：

需要实现Protocol声明的方法，或者

从一个采用这个Protocol并实现了它的方法的类继承

例如，Pager类采用了Paging Protocol，如果Pager是NSObject的子类：

```
@interface Pager : NSObject < Paging >
```

它必须实现所有Paging的方法，包括声明在包含Formatting Protocol中的方法，它采用了Paging Protocol外还采用了Formatting Protocol。

在其他方面，如果Pager是Formatter的子类，Formatter是采用Formatting Protocol的单独的类：

```
@interface Pager : Formatter < Paging >
```

它必须实现所有在Paging Protocol中声明的方法，Formatting中声明的方法不必再次实现。Pager继承了Formatter，也同样遵循了Formatting Protocol。

注意，类采用协议可以不用正式的采用它通过实现在Protocol中声明的方法。

循环引用

在一些复杂的应用中，可能会出现循环包含的情况：

```
#import "B.h"
@protocol A
- foo:(id <B>)anObject;
@end
```

在B Protocol中的声明如下：

```
#import "A.h"
@protocol B
- bar:(id <A>)anObject;
@end
```

在这样的循环结构下，没有文件能被正确的编译，去打破这种递归循环，需要在Protocol定义的地方使用@protocol指令指示向前引用那些需要的Protocol去替代导入接口文件，下面的代码片段示例了应该如何去做：

```
@protocol B;
@protocol A
- foo:(id <B>)anObject;
@end
```

注意，使用这种方式的@protocol指令只是简单的告诉编译器“B”是一个Protocol，它将会在后面才进行定义，它并不会导入定义B的接口文件。

Fast Enumeration

Objective-C 2.0提供了一个语言特性，可以使用一种简明的语法快速安全的轮询集合的内容。

for…in

Objective-C 2.0提供了一个语言特性，允许轮询集合的内容，语法的定义如下：

```
for ( Type newVariable in expression ) { stmts }
```

或

```
Type existingItem;
for ( existingItem in expression ) { stmts }
```

在上面的两种语法中，expression代表一个遵循NSFastEnumeration Protocol的对象，通常为数组或枚举（Cocoa的集合类：NSArray、NSDictionary和NSSet—都采用这个Protocol，类似NSEnumerator）。任何集合类型都可以采用NSFastEnumeration Protocol（详细参见“实现NSFastEnumeration Protocol”）。NSArray和NSSet轮询的是它们的内容对象，其他类需要清楚的说明其轮询迭代的对象—例如，NSDictionary和核心数据类NSManagedObjectModel提供了对NSFastEnumeration的支持，NSDictionary轮询的是它的键值，而NSManagedObjectModel迭代的是它的实体。

使用Fast Enumeration有下面几种好处：

使用轮询更有效率，比如，相比直接使用NSEnumerator，Fast Enumeration效率更高。

语法简明

Fast Enumeration是“安全的“，Fast Enumeration提供了轮询期间的保护机制，当你试图去修改在轮询期间的集合时，将会有异常抛出。

因为同一个对象在迭代期间不允许修改，所以可以同时进行多个轮询。

使用Fast Enumeration

下面的代码示例了在NSArray和NSDictionary对象上使用fast enumeration。

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];
for (NSString *element in array) {
    NSLog(@"element: %@", element);
}
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
        @"quattuor", @"four", @"quinque", @"five", @"sex", @"six", nil];
NSString *key;
for (key in dictionary) {
    NSLog(@"English: %@, Latin: %@", key, [dictionary valueForKey:key]);
}
```

也可以使用NSEnumerator代替fast enumeration，下面的代码就是示例使用NSEnumerator。

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];
NSEnumerator *enumerator = [array reverseObjectEnumerator];
for (NSString *element in enumerator) {
    if ([element isEqualToString:@"Three"]) {
        break;
    }
}
NSString *next = [enumerator nextObject];
// next = "Four"
```

集合和枚举都是有序的—例如，NSArray和NSEnumerator都源于数组，轮询依照这个次序，因此可以简单的依据这个次序统计迭代的次数。

```
NSArray *array = /* assume this exists */;
NSInteger index = 0;
BOOL ok = NO;
for (id element in array) {
    if (/* some test for element */) {
        ok = YES;
        break;
    }
    index++;
}
if (ok) {
    NSLog(@"Test passed by element at index %d", index");
}
```

实现NSFastEnumeration Protocol

自己定义类要支持fast enumeration，必须实现NSFastEnumeration Protocol，这个Protocol只包括一个方法-- countByEnumeratingWithState:objects:count:。

```
@protocol NSFastEnumeration
- (NSUInteger)countByEnumeratingWithState:(NSFastEnumerationState *)state
objects:(id *)stackbuf count:(NSUInteger)len;
@end
```

第一个参数是一个枚举结构（NSFastEnumerationState）：

```
typedef struct {
    unsigned long state;
    id *itemsPtr;
    unsigned long *mutationsPtr;
    unsigned long extra[5];
} NSFastEnumerationState;
```

下面的规则必须在任何实现中被遵照：

在开始进入时，字段“state”被置为0，除此外，都不允许也不要求修改state。State必须在计数器返回非0前置为非0（否则，将为期待返回0进入无限循环，对象将一直开启一个新的enumeration）。对于复杂的enumeration，“extra”字段绑定到附加的状态。

在退出时，计数器0表示enumeration中已经不存在对象，这也可以表示一个坏的或不稳定的内部状态，但通常只是简单的表示没有更多的对象。这个值为0，并不能推断出state的内容。

计数器返回非0值表示itemsPtr指向第一个非0计数器对象的指针，

注意：本文档是基于Mac OS X v10.5的Objective-C的2.0版本，包括了这个版本中的的一些新特性，如属性特性（参见“属性”节）、更快的enumeration（参见“Fast Enumeration”）、optional protocols可配置协议、以及（现代平台）non-fragile实例变量。这些新特性在Mac OS X v10.5以前的版本都不被支持，如果你使用的是Mac OS X v10.5以前的版本，可以学习Objective-C Programming Language 1.0。

注意：给nil发送消息的行为效果在Mac OS X v10.5中进行了改变，对于以前版本的行为效果，请参看以前版本的文档。

注意：提供一个新的根类是一项需要相当的技巧并有巨大风险的工作，这个类需要复制大量NSObject类做的工作，比方像实例的分配、关联它们到他们的类、在运行时中标识它们。因为这个原因，在Cocoa中，你一般使用NSObject作为根类。关于更多的信息，请查看Foundation框架文档中关于NSObject类和NSObject协议的部分。

注意：编译器还为每个类创建一个““metaclass”对象，它就像类对象描述类的实例一样描述类对象，你可以发送消息给类对象和实例，但是metaclass对象只由运行时在内部使用。

注意：使用未声明为static的外部变量也是允许的，但是使用静态变量更有利的表明封包到独立对象中的意图。

注意：记住，运行时系统逐个为每个类发送initialize消息，因此在类的initialize方法实现中，不允许发送initialize消息给它的超类。

重点：缺省值是@dynamic，因此，可以不对特定的属性指定是@synthesize还是@dynamic，必定提供一个getter/setter（仅提供getter表示是一个只读属性的实现）。

注意：即使匿名对象的提供者也不用知道它的类，对象自己在运行时知道自己的类。有一个类的消息可以返回匿名对象的类。在Protocol中的信息已经足够了。