Objective-C

维基百科，自由的百科全书

Objective-C是一种通用、高级、面向对象的编程语言。它扩展了标准的ANSI C编程语言，将Smalltalk式的消息传递机制加入到ANSI C中。目前主要支持的编译器有GCC和LLVM（采用 Clang作为前端）。

Objective-C的商标权属于苹果公司，苹果公司也是这个编程语言的主要开发者。苹果在开发NeXTSTEP操作系统时使用了Objective-C，之后被OS X和iOS继承下来。现在Objective-C是OS X和iOS操作系统、及与其相关的API、Cocoa和Cocoa Touch的主要编程语言。

Objective-C

编程范型	面向对象
发行时间	1983
设计者	布莱德·考克斯和Tom Love
实作者	苹果公司
最新发行时间	Objective-C 2.1[1] / 2007年10月26日
型态系统	静态类型、动态类型、弱类型
主要实作产品	Clang、GCC
启发语言	C、Smalltalk
影响语言	Java、Objective-J、TOM、Nu语言
操作系统	跨平台

目录

历史

1980年代初，Stepstone 公司的 布莱德·考克斯（Brad Cox）与 Tom Love 发明了 Objective-C。 1981年，当他们两人还在ITT公司技术中心任职时，接触到了一个名为 SmallTalk-80 的语言，恰巧 Cox 当时对软件设计中的重用问题非常感兴趣，于是他马上就意识到了 SmallTalk-80 这个语言在软件建构中无法衡量的巨大价值，但同时 Cox 与 Love 也明白，在目前的ITT公司中，引入技术最重要的关键是与 C 语言兼容。

于是 Cox 开始撰写一个 C 语言的预处理器，打算使 C 语言具备些许 Smalltalk 的本领。Cox 很快实现出了一个可用的 C 语言扩展，此即为 Objective-C语言的前身。到了 1983 年，Cox 与 Love 合伙成立了 Productivity Products International（PPI）公司，将 Objective-C 及其相关库商品化贩售，并在之后将公司改名为 StepStone。1986年，Cox 出版了一本关于 Objc 的重要著作《Object-Oriented Programming, An Evolutionary Approach》，书内详述了 Objective-C 的种种设计理念。

1988年，乔布斯(Steve Jobs)离开苹果公司后成立了 NeXT Computer 公司，NeXT 公司买下 Objective-C 语言的授权，并扩展了著名的开源编译器GCC 使之支持 Objective-C 的编译。并基于 Objective-C 开发了 AppKit 与 Foundation Kit 等等库，作为 NeXTSTEP 的的用户接口与开发环境的基础。虽然 NeXT 工作站后来在市场上失败了，但 NeXT 上的软件工具却在业界中被广泛赞扬。这促使 NeXT 公司放弃硬件业务，转型为销售 NeXTStep（以及OpenStep）平台为主的软件公司。

1992年，自由软件基金会的 GNU 开发环境增加了对 Objective-C 的支持。1994年，NeXT Computer公司和Sun Microsystem联合发布了一个针对 NEXTSTEP 系统的标准典范，名为 OPENSTEP。OPENSTEP 在自由软件基金会的实现名称为 GNUstep。1996年12月20日，苹果公司宣布收购 NeXT Software 公司，NEXTSTEP/OPENSTEP环境成为苹果操作系统下一个主要发行版本OS X的基础。这个开发环境的该版本被苹果公司称为Cocoa。

2005年，苹果电脑雇用了克里斯·拉特纳及LLVM开发团队[2]，clang及LLVM成为苹果公司在GCC之外的新编译器选择，在 Xcode 4.0之后均采用 LLVM 作为默认的编译器。最新的 Objective-C 特性也都率先在 Clang 上实现。

语法

Objective-C是C语言的严格母集合，意思是任何原始的C语言程序不经修改就可以直接通过Objective-C编译器，在Objective-C中使用C语言代码也是完全合法的。Objective-C形容自己是*盖在C语言上的薄薄一层*，因为Objective-C的原意就是在C语言主体上加入面向对象的特性。Objective-C的面向对象语法源于Smalltalk消息传递风格。所有其他非面向对象的语法，包括变量类型，预处理器（preprocessing），流程控制，函数声明与调用皆与C语言完全一致。

Hello World

这里示范了一个基础的Hello World程序。基于Xcode 4.3.1 xcode:

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

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

    @autoreleasepool {
        NSLog(@"Hello World!");
    }

   return 0;
}
```

消息传递

Objective-C最大的特色是承自Smalltalk的消息传递模型（message passing），此机制与今日C++式之主流风格差异甚大。Objective-C里，与其说对象互相**调用方法**，不如说对象之间**互相传递消息**更为精确。此二种风格的主要差异在于*调用方法/消息传递*这个动作。C++里类与方法的关系严格清楚，一个方法必定属于一个类，而且在编译时（compile time）就已经紧密绑定，不可能调用一个不存在类里的方法。但在Objective-C，类与消息的关系比较松散，调用方法视为对对象*发送消息*，所有方法都被视为*对消息的回应*。所有消息处理直到运行时（runtime）才会动态决定，并交由类自行决定如何处理收到的消息。也就是说，一个类不保证一定会回应收到的消息，如果类收到了一个无法处理的消息，程序只会抛出异常，不会出错或崩溃。

C++里，送一个消息给对象（或者说调用一个方法）的语法如下：

```
obj->method(argument);
```

Objective-C则写成：

```
[obj method: argument];
```

此二者并不仅仅是语法上的差异，还有基本行为上的不同。

这里以一个汽车类（car class）的简单例子来解释Objective-C的消息传递特性：

```
[car fly];
```

典型的C++意义解读是“调用car类的fly方法”。若car类里头没有定义fly方法，那编译肯定不会通过。但是Objective-C里，我们应当解读为“发提交一个fly的消息给car对象”，fly是*消息*，而car是消息的*接收者*。car收到消息后会决定如何回应这个消息，若car类内定义有fly方法就运行方法内之代码，若car内不存在fly方法，则程序依旧可以通过编译，运行期则抛出异常。

此二种风格各有优劣。C++强制要求所有的方法都必须有对应的动作，且编译期绑定使得函数调用非常快速。缺点是仅能借由virtual关键字提供有限的动态绑定能力。Objective-C天生即具备鸭子类型之动态绑定能力，因为运行期才处理消息，允许发送未知消息给对象。可以送消息给整个对象集合而不需要一一检查每个对象的型态，也具备消息转送机制。同时空对象nil接受消息后默认为不做事，所以送消息给nil也不用担心程序崩溃。

Objective-C的方法调用因为运行期才动态解析消息，一开始消息比C++ virtual成员函数调用速度慢上三倍。但经由IMP缓存改善，目前已经比C++的virtual function快上50%。

类的定义与实现

Objective-C中强制要求将类的定义（interface）与实现（implementation）分为两个部分。

类的定义文件遵循C语言之惯例以.h为后缀，实现文件以.m为后缀。

Interface

定义部分，清楚定义了类的名称、数据成员和方法。 以关键字@interface作为开始，@end作为结束。

```
@interface MyObject : NSObject {
    int memberVar1; // 实体变量
    id  memberVar2;
}

+(return_type) class_method; // 类方法

-(return_type) instance_method1; // 实例方法
-(return_type) instance_method2: (int) p1;
-(return_type) instance_method3: (int) p1 andPar: (int) p2;
@end
```

方法前面的 +/- 号代表函数的类型：加号（+）代表类方法（class method），不需要实例就可以调用，与C++ 的静态函数（static member function）相似。减号（-）即是一般的实例方法（instance method）。

这里提供了一份意义相近的C++语法对照，如下：

```
class MyObject : public NSObject {
protected:
    int memberVar1;  // 实体变量
    void * memberVar2;

  public:
    static return_type class_method(); // 類方法

    return_type instance_method1();    // 实例方法
    return_type instance_method2( int p1 );
    return_type instance_method3( int p1, int p2 );
}
```

Objective-C定义一个新的方法时，名称内的冒号（:）代表参数传递，不同于C语言以数学函数的括号来传递参数。Objective-C方法使得参数可以夹杂于名称中间，不必全部附缀于方法名称的尾端，可以提高程序可读性。设定颜色RGB值的方法为例：

```
- (void) setColorToRed: (float)red Green: (float)green Blue:(float)blue; /* 宣告方法*/
```

```
[myColor setColorToRed: 1.0 Green: 0.8 Blue: 0.2]; /* 呼叫方法*/
```

这个方法的签名是setColorToRed:Green:Blue:。每个冒号后面都带着一个float类别的参数，分别代表红，绿，蓝三色。

Implementation

实现区段则包含了公开方法的实现，以及定义私有（private）变量及方法。 以关键字@implementation作为区段起头，@end结尾。

```
@implementation MyObject {
  int memberVar3; //私有實體變數
}

+(return_type) class_method {
    .... //method implementation
}
-(return_type) instance_method1 {
    ....
}
-(return_type) instance_method2: (int) p1 {
    ....
}
-(return_type) instance_method3: (int) p1 andPar: (int) p2 {
    ....
}
@end
```

值得一提的是不只Interface区段可定义实体变量，Implementation区段也可以定义实体变量，两者的差别在于访问权限的不同，Interface区段内的实体变量默认权限为protected，声明于implementation区段的实体变量则默认为private，故在Implementation区段定义私有成员更符合面向对象之封装原则，因为如此类之私有信息就不需曝露于公开interface（.h文件）中。

创建对象

Objective-C创建对象需通过alloc以及init两个消息。alloc的作用是分配内存，init则是初始化对象。 init与alloc都是定义在NSObject里的方法，父对象收到这两个信息并做出正确回应后，新对象才创建完毕。以下为范例：

```
MyObject * my = [[MyObject alloc] init];
```

在Objective-C 2.0里，若创建对象不需要参数，则可直接使用new

```
MyObject * my = [MyObject new];
```

仅仅是语法上的精简，效果完全相同。

若要自己定义初始化的过程，可以重写init方法，来添加额外的工作。（用途类似C++ 的构造函数constructor）

```
- (id) init {
    if ( self=[super init] ) {   // 必須調用父类的init
        // do something here ...
    }
    return self;
}
```

协议（Protocol）

协议是一组没有实现的方法列表，任何的类均可采纳协议并具体实现这组方法。

Objective-C在NeXT时期曾经试图引入多重继承的概念，但由于协议的出现而没有实现之。

协议类似于Java与C#语言中的“接口”。在Objective-C中，有两种定义协议的方式：由编译器保证的“正式协议”，以及为特定目的设定的“非正式协议”。

非正式协议为一个可以选择性实现的一系列方法列表。非正式协议虽名为协议，但实际上是挂于NSObject上的未实现分类（Unimplemented Category）的一种称谓，Objetive-C语言机制上并没有非正式协议这种东西，OSX 10.6版本之后由于引入@optional关键字，使得正式协议已具备同样的能力，所以非正式协议已经被废弃不再使用。

正式协议类似于Java中的"接口"，它是一系列方法的列表，任何类都可以声明自身实现了某个协议。在Objective-C 2.0之前，一个类必须实现它声明符合的协议中的所有方法，否则编译器会报告错误，表明这个类没有实现它声明符合的协议中的全部方法。Objective-C 2.0版本允许标记协议中某些方法为可选的（Optional），这样编译器就不会强制实现这些可选的方法。

协议经常应用于Cocoa中的委托及事件触发。例如文本框类通常会包括一个委托（delegate）对象，该对象可以实现一个协议，该协议中可能包含一个实现文字输入的自动完成方法。若这个委托对象实现了这个方法，那么文本框类就会在适当的时候触发自动完成事件，并调用这个方法用于自动完成功能。

Objective-C中协议的概念与Java中接口的概念并不完全相同，即一个类可以在不声明它符合某个协议的情况下，实现这个协议所包含的方法，也即实质上符合这个协议，而这种差别对外部代码而言是不可见的。正式协议的声明不提供实现，它只是简单地表明符合该协议的类实现了该协议的方法，保证调用端可以安全调用方法。

语法

协议以关键字@protocol作为区段起始，@end退出，中间为方法列表。

```
@protocol Locking
- (void)lock;
- (void)unlock;
@end
```

这是一个协议的例子，多线程编程中经常要确保一份共享资源同时只有一个线程可以使用，会在使用前给该资源挂上锁，以上即为一个表明有“锁”的概念的协议，协议中有两个方法，只有名称但尚未实现。

下面的SomeClass宣称他采纳了Locking协议：

```
@interface SomeClass : SomeSuperClass <Locking>
@end
```

一旦SomeClass表明他采纳了Locking协议，SomeClass就有义务实现Locking协议中的两个方法。

```
@implementation SomeClass
- (void)lock {
  // 實現lock方法...
}
- (void)unlock {
  // 實現unlock方法...
}
@end
```

由于SomeClass已经确实遵从了Locking协议，故调用端可以安全的发送lock或unlock消息给SomeClass实体变量，不需担心他没有办法回应消息。

插件是另一个使用抽象定义的例子，可以在不关心插件的实现的情况下定义其希望的行为。

动态类型

类似于Smalltalk，Objective-C具备动态类型：即消息可以发送给任何对象实体，无论该对象实体的公开接口中有没有对应的方法。对比于C++这种静态类型的语言，编译器会挡下对（void*）指针调用方法的行为。但在Objective-C中，你可以对id发送任何消息（id很像void*，但是被严格限制只能使用在对象上），编译器仅会发出“该对象可能无法回应消息”的警告，程序可以通过编译，而实际发生的事则取决于运行期该对象的真正形态，若该对象的确可以回应消息，则依旧运行对应的方法。

一个对象收到消息之后，他有三种处理消息的可能手段，第一是回应该消息并运行方法，若无法回应，则可以转发消息给其他对象，若以上两者均无，就要处理无法回应而抛出的例外。只要进行三者之其一，该消息就算完成任务而被丢弃。若对“nil”（空对象指针）发送消息，该消息通常会被忽略，取决于编译器选项可能会抛出例外。

虽然Objective-C具备动态类型的能力，但编译期的静态类型检查依旧可以应用到变量上。以下三种声明在运行时效力是完全相同的，但是三种声明提供了一个比一个更明显的类型信息，附加的类型信息让编译器在编译时可以检查变量类型，并对类型不符的变量提出警告。

下面三个方法，差异仅在于参数的形态：

```
- setMyValue:(id) foo;
```

id形态表示参数“foo”可以是任何类的实例。

```
- setMyValue:(id <aProtocol>) foo;
```

id<aProtocol>表示“foo”可以是任何类的实例，但必须采纳“aProtocol”协议。

```
- setMyValue:(NSNumber*) foo;
```

该声明表示“foo”必须是“NSNumber”的实例。

动态类型是一种强大的特性。在缺少泛型的静态类型语言（如Java 5以前的版本）中实现容器类时，程序员需要写一种针对通用类型对象的容器类，然后在通用类型和实际类型中不停的强制类型转换。无论如何，类型转换会破坏静态类型，例如写入一个“整数”而将其读取为“字符串”会产生运行时错误。这样的问题被泛型解决，但容器类需要其内容对象的类型一致，而对于动态类型语言则完全没有这方面的问题。

转发

Objective-C允许对一个对象发送消息，不管它是否能够响应之。除了响应或丢弃消息以外，对象也可以将消息转发到可以响应该消息的对象。转发可以用于简化特定的设计模式，例如观测器模式或代理模式。

Objective-C运行时在`Object`中定义了一对方法：

- 转发方法：

```
- (retval_t) forward:(SEL) sel :(arglist_t) args; // with GCC
- (id) forward:(SEL) sel :(marg_list) args; // with NeXT/Apple systems
```

- 响应方法：

```
- (retval_t) performv:(SEL) sel :(arglist_t) args;  // with GCC
- (id) performv:(SEL) sel :(marg_list) args; // with NeXT/Apple systems
```

希望实现转发的对象只需用新的方法覆盖以上方法来定义其转发行为。无需重写响应方法`performv::`，由于该方法只是单纯的对响应对象发送消息并传递参数。其中，`SEL`类型是Objective-C中消息的类型。

例子

这里包括了一个演示转发的基本概念的程序示例。

Forwarder.h

```
#import <objc/Object.h>

@interface Forwarder : Object
{
    id recipient; //该对象是我们希望转发到的对象。
}

@property (assign, nonatomic) id recipient;

@end
```

Forwarder.m

```
#import "Forwarder.h"

@implementation Forwarder

@synthesize recipient;

- (retval_t) forward: (SEL) sel : (arglist_t) args
{
    /*
     *检查转发对象是否响应该消息。
     *若转发对象不响应该消息，则不会转发，而产生一个错误。
     */
    if([recipient respondsTo:sel])
       return [recipient performv: sel : args];
    else
       return [self error:"Recipient does not respond"];
}
```

Recipient.h

```
#import <objc/Object.h>

// A simple Recipient object.
@interface Recipient : Object
- (id) hello;
@end
```

Recipient.m

```
#import "Recipient.h"

@implementation Recipient

- (id) hello
{
    printf("Recipient says hello!\n");

    return self;
}

@end
```

main.m

```
#import "Forwarder.h"
#import "Recipient.h"

int main(void)
{
    Forwarder *forwarder = [Forwarder new];
    Recipient *recipient = [Recipient new];

    forwarder.recipient = recipient; //Set the recipient.
    /*
     *转发者不响应hello消息！该消息将被转发到转发对象。
     *（若转发对象响应该消息）
     */
    [forwarder hello];

    return 0;
}
```

脚注

利用GCC编译时，编译器报告：

```
$ gcc -x objective-c -Wno-import Forwarder.m Recipient.m main.m -lobjc
main.m: In function `main':
main.m:12: warning: `Forwarder' does not respond to `hello'
$
```

如前文所提到的，编译器报告Forwarder类不响应hello消息。在这种情况下，由于实现了转发，可以忽略这个警告。 运行该程序产生如下输出：

```
$ ./a.out
Recipient says hello!
```

类别 (Category)

在Objective-C的设计中，一个主要的考虑即为大型代码框架的维护。结构化编程的经验显示，改进代码的一种主要方法即为将其分解为更小的片段。Objective-C借用并扩展了Smalltalk实现中的“分类”概念，用以帮助达到分解代码的目的。[3]

一个分类可以将方法的实现分解进一系列分离的文件。程序员可以将一组相关的方法放进一个分类，使程序更具可读性。举例来讲，可以在字符串类中增加一个名为“拼写检查”的分类，并将拼写检查的相关代码放进这个分类中。

进一步的，分类中的方法是在运行时被加入类中的，这一特性允许程序员向现存的类中增加方法，而无需持有原有的代码，或是重新编译原有的类。例如若系统提供的字符串类的实现中不包含拼写检查的功能，可以增加这样的功能而无需更改原有的字符串类的代码。

在运行时，分类中的方法与类原有的方法并无区别，其代码可以访问包括私有类成员变量在内的所有成员变量。

若分类声明了与类中原有方法同名的函数，则分类中的方法会被调用。因此分类不仅可以增加类的方法，也可以代替原有的方法。这个特性可以用于修正原有代码中的错误，更可以从根本上改变程序中原有类的行为。若两个分类中的方法同名，则被调用的方法是不可预测的。

其它语言也尝试了通过不同方法增加这一语言特性。TOM在这方面走的更远，不仅允许增加方法，更允许增加成员变量。也有其它语言使用面向声明的解决方案，其中最值得注意的是Self语言。

C#与Visual Basic.NET语言以扩展函数的与不完全类的方式实现了类似的功能。Ruby与一些动态语言则以"monkey patch"的名字称呼这种技术。

使用分类的例子

这个例子创建了Integer类，其本身只定义了integer属性，然后增加了两个分类Arithmetic与Display以扩展类的功能。虽然分类可以访问类的私有成员，但通常利用属性的访问方法来访问是一种更好的做法，可以使得分类与原有类更加独立。这是分类的一种典型应用—另外的应用是利用分类来替换原有类中的方法，虽然用分类而不是继承来替换方法不被认为是一种好的做法。

Integer.h

```
#import <objc/Object.h>

@interface Integer : Object
{
@private
    int integer;
}

@property (assign, nonatomic) integer;

@end
```

Integer.m

```
#import "Integer.h"

@implementation Integer

@synthesize integer;

@end
```

Arithmetic.h

```
#import "Integer.h"

@interface Integer(Arithmetic)
- (id) add: (Integer *) addend;
- (id) sub: (Integer *) subtrahend;
@end
```

Arithmetic.m

```
#import "Arithmetic.h"

@implementation Integer(Arithmetic)
- (id) add: (Integer *) addend
{
    self.integer = self.integer + addend.integer;
    return self;
}

- (id) sub: (Integer *) subtrahend
{
    self.integer = self.integer - subtrahend.integer;
    return self;
```

```
}
@end
```

Display.h

```
#import "Integer.h"

@interface Integer(Display)
- (id) showstars;
- (id) showint;
@end
```

Display.m

```
#import "Display.h"

@implementation Integer(Display)
- (id) showstars
{
    int i, x = self.integer;
    for(i=0; i < x; i++)
       printf("*");
    printf("\n");

    return self;
}

- (id) showint
{
    printf("%d\n", self.integer);

    return self;
}
@end
```

main.m

```
#import "Integer.h"
#import "Arithmetic.h"
#import "Display.h"

int
main(void)
{
    Integer *num1 = [Integer new], *num2 = [Integer new];
    int x;

    printf("Enter an integer: ");
    scanf("%d", &x);

    num1.integer = x;
    [num1 showstars];

    printf("Enter an integer: ");
    scanf("%d", &x);

    num2.integer = x;
    [num2 showstars];

    [num1 add:num2];
    [num1 showint];

    return 0;
}
```

注释

可以利用以下命令来编译：

```
gcc -x objective-c main.m Integer.m Arithmetic.m Display.m -lobjc
```

在编译时间，可以利用省略`#import "Arithmetic.h"`与`[num1 add:num2]`命令，以及`Arithmetic.m`文件来实验。程序仍然可以运行，这表明了允许动态的、按需的加载分类；若不需要某一分类提供的功能，可以简单的不编译之。

扮演

Objective-C允许一个类在程序中完全取代另一个类，这种行为称为前者“扮演”目标类。

注意：类的扮演在Mac OS X v10.5中被废弃，在64位运行时中不可用。

#import

在C语言中，`#include`预处理指令总是使被包含的文件内容被插入指令点。在Objective-C中，类似的指令`#import`保证一个文件只会被包含一次，类似于一般头文件中的

```
#ifndef XXX
#define XXX ...
#endif
```

惯用法，或MSVC中的

```
#pragma once
```

语言变化

Objective-C++

Objective-C++是GCC的一个前端，它可以编译混合了C++与Objective-C语法的源文件。Objective-C++是C++的扩展，类似于Objective-C是C的扩展。由于在融合C++与Objective-C两种语言的特性方面没有做特别的工作，因此有以下限制：

- C++类不能从Objective-C类继承，反之亦然。
- Objective-C定义内部不能定义C++命名空间。
- Objective-C类的成员变量不能包括不含默认构造函数和/或含有虚方法的C++类对象，但使用C++类指针并无如此限制（可以在 -init方法中对之进行初始化）。
- C++“传递值”的特性不能用在Objective-C对象上，而只能传递其指针。
- Objective-C声明不能存在在C++模板声明中，反之亦然。但Objective-C类型可以用在C++模板的参数中。
- Objective-C和C++的错误处理语句不同，各自的语句只能处理各自的错误。
- Objective-C错误使得C++对象被退出时，C++析构函数不会被调用。新的64位运行时解决了这个问题。[4]

Objective-C 2.0

在2006年7月苹果全球开发者会议中，Apple宣布了“Objective-C 2.0”的发布，其增加了“现代的垃圾收集，语法改进[5]，运行时性能改进[6]，以及64位支持”。2007年10月发布的Mac OS X v10.5中包含了Objective-C 2.0的编译器。

垃圾收集

Objective-C 2.0提供了一个可选的垃圾收集器。在向后兼容模式中，Objective-C运行时会将引用计数操作，例如“retain”与“release”变为无操作。当垃圾收集启用时，所有的对象都是收集器的工作对象。普通的C指针可以以“__strong”修饰，标记指针指向的对象仍在使用中。被标记为“__weak”的指针不被计入收集器的计数中，并在对象被回收时改写为“nil”。iOS上的Objective-C 2.0实现中不包含垃圾收集器。垃圾收集器运行在一个低优先级的后台线程中，并可以在用户动作时暂停，从而保持良好的用户体验。[7]

属性

Objective-C 2.0引入了新的语法以声明变量为属性，并包含一可选定义以配置访问方法的生成。属性总是为公共的，其目的为提供外部类访问（也可能为只读）类的内部变量的方法。属性可以被声明为“readonly”，即只读的，也可以提供储存方法包括“assign”，“copy”或“retain”（简单的赋值、复制或增加1引用计数）。默认的属性是原子的，即在访问时会加锁以避免多线程同时访问同一对象，也可以将属性声明为“nonatomic”（非原子的），避免产生锁。

```
@interface Person : NSObject {
    @public
        NSString *name;
    @private
        int age;
}

@property(copy) NSString *name;
@property(readonly) int age;

-(id)initWithAge:(int)age;
@end
```

属性的访问方法由@synthesize关键字来实现，它由属性的声明自动的产生一对访问方法。另外，也可以选择使用@dynamic关键字表明访问方法会由程序员手工提供。

```
@implementation Person
@synthesize name;
@dynamic age;

-(id)initWithAge:(int)initAge
{
    age = initAge; // 注意：直接赋给成员变量，而非属性
    return self;
}

-(int)age
{
    return 29; // 注意：并非返回真正的年龄
}
@end
```

属性可以利用传统的消息表达式、点表达式或"valueForKey:"/"setValue:forKey:"方法对来访问。

```
Person *aPerson = [[Person alloc] initWithAge: 53];
aPerson.name = @"Steve"; // 注意：点表达式，等于[aPerson setName: @"Steve"];
NSLog(@"Access by message (%@), dot notation(%@), property name(%@) and direct instance variable access (%@)",
      [aPerson name], aPerson.name, [aPerson valueForKey:@"name"], aPerson->name);
```

为了利用点表达式来访问实例的属性，需要使用“self”关键字：

```
-(void) introduceMyselfWithProperties:(BOOL)useGetter
{
    NSLog(@"Hi, my name is %@.", (useGetter ? self.name : name)); // NOTE: getter vs. ivar access
}
```

类或协议的属性可以被动态的读取。

```
int i;
int propertyCount = 0;
objc_property_t *propertyList = class_copyPropertyList([aPerson class], &propertyCount);

for ( i=0; i < propertyCount; i++ ) {
    objc_property_t *thisProperty = propertyList + i;
    const char* propertyName = property_getName(*thisProperty);
    NSLog(@"Person has a property: '%s'", propertyName);
}
```

快速枚举

比起利用NSEnumerator对象或在集合中依次枚举，Objective-C 2.0提供了快速枚举的语法。在Objective-C 2.0中，以下循环的功能是相等的，但性能特性不同。

```
// 使用NSEnumerator
NSEnumerator *enumerator = [thePeople objectEnumerator];
Person *p;

while ( (p = [enumerator nextObject]) != nil ) {
    NSLog(@"%@ is %i years old.", [p name], [p age]);
}
```

```
// 使用依次枚举
for ( int i = 0; i < [thePeople count]; i++ ) {
    Person *p = [thePeople objectAtIndex:i];
    NSLog(@"%@ is %i years old.", [p name], [p age]);
}
```

```
// 使用快速枚举
for (Person *p in thePeople) {
    NSLog(@"%@ is %i years old.", [p name], [p age]);
}
```

快速枚举可以比标准枚举产生更有效的代码，由于枚举所调用的方法被使用NSFastEnumeration协议提供的指针算术运算所代替了。[8]

语言分析

Objective-C是非常“实际”的语言。它用一个很小的、用C写成的运行库，使得应用程序的大小增加很少，与此相比，大部分OO系统需要极大的运行时虚拟机来执行。ObjC写成的程序通常不会比其源代码和库（通常无需包含在软件发布版本中）大太多，不会像Smalltalk系统，即使只是打开一个窗口也需要大量的容量。由于Obj-C的动态类型特征，Obj-C不能对方法进行内联（inline）一类的优化，使得Obj-C的应用程序一般比类似的C或C++程序更小。

Obj-C可以在现存C编译器基础上实现（在GCC中，Obj-C最初作为预处理器引入，后来作为模块存在），而不需要编写一个全新的编译器。这个特性使得Obj-C能利用大量现存的C代码、库、工具和编程思想等资源。现存C库可以用Obj-C包装器来提供一个Obj-C使用的OO风格界面包装。

以上这些特性极大地降低了进入Obj-C的门槛，这是1980年代Smalltalk在推广中遇到的最大问题。

Objective-C的最初版本并不支持垃圾回收（garbage collection）。在当时这是争论的焦点之一，很多人考虑到Smalltalk回收时有漫长的“死亡时间”，令整个系统失去功用，Objective-C为避免此问题才不拥有这个功能。某些第三方版本加入了这个功能（尤是GNUstep），苹果公司也在其Mac OS X 10.5中提供了实现。

另一个广受批评的问题是ObjC不包括名字空间机制（namespace mechanism）。取而代之的是程序员必须在其类名称加上前缀，由于前缀往往较短（相比命名空间），这时常引致冲突。在2007年，在Cocoa编程环境中，所有Mac OS X类和函数均有“NS”作为前缀，例如NSObject或NSButton来清楚分辨它们属于Mac OS X核心；使用“NS”是由于这些类的名称在NeXTSTEP开发时定下。

虽然Objective-C是C的严格超集，但它也不视C的基本类型为第一级的对象。

和C++不同，Objective-C不支持运算符重载（它不支持ad-hoc多态）。亦与C++不同，但和Java相同，Objective-C只容许对象继承一个类（不设多重继承）。Categories和protocols不但可以提供很多多重继承的好处，而且没有很多缺点，例如额外运行时间过重和二进制不兼容。

由于Obj-C使用动态运行时类型，而且所有的方法都是函数调用（有时甚至连系统调用（syscalls）也如此），很多常见的编译时性能优化方法都不能应用于Obj-C（例如：内联函数、常数传播、交互式优化、纯量取代与聚集等）。这使得Obj-C性能劣于类似的对象抽象语言（如C++）。不过Obj-C拥护者认为Obj-C本就不应应用于C++或Java常见的底层抽象，Obj-C的应用方向是对性能要求不大的应用

参考文献

1. ^ Mac OS X 10.6 Snow Leopard: the Ars Technica review, page 5 (http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/5)
2. ^ Adam Treat, *mkspecs and patches for LLVM compile of Qt4 (http://lists.trolltech.com/qt4-preview-feedback/2005-02/msg00691.html)*
3. ^ Example of *categories* concept (http://video.google.com/videoplay?docid=-7466310348707586940&ei=0dr7Sle6L46qrgLk7dHsDg&q=Smalltalk-80)
4. ^ Using C++ With Objective-C (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocCPlusPlus.html#//apple_ref/doc/uid/TP30001 CH10-SW1) in Mac OS X Reference Library, last retrieved in 2010-02-10.
5. ^ Objective-C 2.0: more clues (http://lists.apple.com/archives/Objc-language/2006/Aug/msg00039.html). Lists.apple.com. 2006-08-10 [2010-05-30].
6. ^ Re: Objective-C 2.0 (http://lists.apple.com/archives/Objc-language/2006/Aug/msg00018.html). Lists.apple.com. [2010-05-30].
7. ^ Apple Computer, Inc. Leopard Technology Series for Developers: Objective-C 2.0 Overview (http://developer.apple.com/leopard/overview/objectivec2.html). Developer.apple.com. 2007-11-06 [2010-05-30].
8. ^ Apple, Inc. Fast Enumeration (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocFastEnumeration.html). apple.com. 2009 [2009-12-31].

外部链接

- 苹果官方Objective-C开发说明文档 (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html)