从 C++ 到 Objective-C

2.1 版 中文

Pierre CHATELIER
e-mail: pierre.chaterlier@club-internet.fr

修订版在以下地址下载:
http://pierre.chachatelier.fr/programmation/objective-c.php

此文档还有英语和法语两个版本

特别鸣谢： 我想感谢 Pascal BLEUYARD, Jérôme CORNET, François DELOBEL 和 Jean-Daniel DUPAS，他们试读了本文档并给出很多建议，他们的帮助对于完善此文档相当重要。Jack NUTTING, Ben RIMMINGTON 和 Mattias ARRELID 提供了很多反馈。Jonathon MAH 特别帮助我纠正了很多错误。文档中若有错误，皆与他们无关，而是因本人的疏漏。

目录

概述

这份文档希望能扮演一座 C++ 到 Objective-C 的桥梁。已有很多文本教授 Objective-C 的对象模型。但，就我所知，还没有面向高级 C++ 开发人。他们可以用已拥有的 C++ 知识和 Objective-C 各种概念进行比较学习 。首先，Objective-C 语言看起来不象 Cocoa 的坚实基础而更象它的一个阻碍 (见下文第一章)：它太不同了，难以学习。我花了一些时间才开始喜欢这些挑战，并理解它提供的很多有用的概念。这份文档不是一个教程，而更象一个关于各种概念的便捷手册。我希望这能改善这样一些情况：一些开发人员由于对于 Objective-C 的误解，很快放弃了或者误用了这门语言。这份文档没有希望成为一份完全手册，而是希望成为一份便捷手册。关于概念的详细解释，应当阅读专门的 Objective-C 参考手册。

与 C# 的比较，需要另外一份文档。比起 C++，C# 更为接近 Objective-C。一个 C# 程序员无疑更容易转换到 Objective-C。对于我来说，C# 由于它充斥了大量的高级概念，而远没有 Objective-C 那么有趣，因为它提供的一些功能很难用，在Objective-C 中却相当简单，而且 Cocoa 的质量远胜于 .NET。这些个人观点不是本文要讨论的主题。

1 Objective-C

1.1 Objective-C和 Cocoa

首先需要澄清的是：Objective-C 是一门语言，Cocoa 是一些类的集合，通过它程序员可以做 MacOS X 下开发的。理论上，是即便没有 Cocoa 也可使用 Objective-C：存在一个可以完全编译的gcc 编译器。但在 MacOS X 下，两者 (Objective-C 和 Cocoa) 是密不可分的，因为大部分右语言级别提供的类都是Cocoa 的一部分。

更确切地说，Cocoa 是 Apple 为 MacOS X 而编写的。它遵循 OpenSetp 标准，于 1994 年发布。它是一个基于 Objective-C 的开发框架。GNUsetp 是另外一种开源实现。它的目标尽可能地将Cocoa 的功能移植到 Unix 系统中。本文档编写期间它仍在开发之中。

1.2 Objective-C 简史

由于有起步，发展，发布，标准化等时期存在，很难精确地给出编程语言的诞生日。尽管如此，表一描述了一个粗略的历史，从中可以看到 Objective-C 在它的前辈和“竞争者”中的位置。

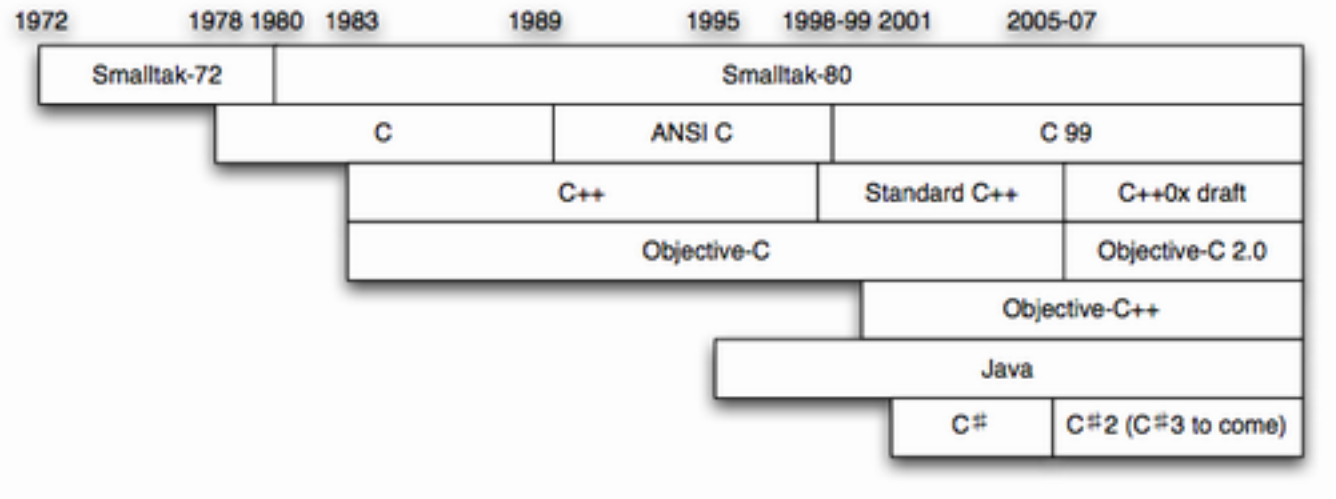

Figure 1 ： Timeline of Java, C, C#, C++ and Objective-C

SmallTalk-80 是最早的真正的面向对象编程语言之一。C++ 和 Objective-C 是以 C 语言为基础创造面向对象语言的两条分支。Objective-C 在多态和语法方面大量借鉴了 Smalltalk，而 C++ 则为了执行性能而转向为了一种更为静态的语言。Java 希望取代 C++，但它在其更为纯粹的面向对象方面同样受到了 Smalltalk 的启发。这也是为什么这本以 Objective-C 标题的文档，引用材料中却有大量 Java 内容。Microsoft 的 C# 是 Objective-C 的直接竞争者。

Objective-C++ 是 Objective-C++ 和 C++ 的聚合。它已经是可用的了，但一些语言行为仍然有缺陷。Objective-C++ 的目标是聚合 Objetive-C 和 C++ 的语法，以获得两者的各自的优势(见 14 章)。

1.3 Objective-C 2.0

本文档已考虑到随同 MacOS X 10.5 一同发布的 Objective-C 2.0 的新特性。虽然有些功能是低层的技术改进，但仍然可以很容易列举一些开发人员可见的高级改进。例如：

- 垃圾收集 ： 见 6.6 节
- 属性 (properties) ： 见 12.2 节
- 快速枚举 ： 见 11.2.2 节
- 协议 (protocole) 的关键词 @optional 和 @required：见 4.4 节
- Objective-C 运行时库 (run-time) 的更新：见 13.2 节

细节在各自章节。

2 语法概览

2.1 关键字

Objective-C 是 C 语言的超集。如同 C++ 一样，书写良好的 C 代码是可以被 Objective-C 编译器编译的，只要不使用一些 C 语言允许的坏的习惯。Objective-C 仅仅加了一些概念和相关的关键字。为了避免冲退，这些关键字以 @ 开头。这有一个简短的详细列表：**`@class`**，**`@interface`**，**`@implementation`**，**`@public`**，**`@private`**，**`@protected`**，**`@try`**，**`@catch`**，**`@throw`**，**`@finally`**，**`@end`**，**`@protocol`**，**`@selector`**，**`@synchronized`**，**`@encode`**，**`@defs`**（文档[4]）。Objective-C 2.0 (见 1.2) 添加了 **`@optional`**，**`@required`**，**`@property`**，**`@dynamic`**，**`@synthesize`**。还有值 nil 和值 Nil，类型 id，类型 SEL 和 类型 BOOL 以及他的值 YES 和 NO。最后，还有一些定义在特定上下文的关键词，它们离开了上下文便失效了：**`in`**，**`out`**，**`bycopy`**，**`byref`**，**`oneway`** (它们被用于定义协议，见 4.4.5)，**`getter`**，**`setter`**，**`readwrite`**，**`readonly`**，**`assign`**，**`retain`**，**`copy`**，**`nonatomic`** (它们被用于定义属性，见 12.2)。

很容易把语言的关键字和继承于根类 NSObject (所有类的父类，见 3.1) 的方法混淆。比如，一些看起来象关键字的内存管理方法：alloc, retain, release, autorelease, 它们实际上是 NSObject 的方法。单词 super 和 self (见 3.3.1) 应该被理解为关键字，但 self 实际上是每个方法的隐藏参数，super 是请求编译器以不同的方式使用 self 的指令。可是，考虑到这些词的不可避免的使用，将这些词视为关键字也并不影响正常使用。

2.2 注释

/* */ 和 // 被用于注释。

2.3 混合实现和声明

如同在 C++ 中一样，是可以在一块指令中插入变量声明语句。

2.4 新类型和值

2.4.1 BOOL，YES，NO

在 C++ 中，布尔类型用 bool 表示。在 Objective-C 中，布尔类型是 BOOL，可被设为 YES 或者 NO。

2.4.2 nil，Nil 和 id

这三个关键词将文档会稍后解释，这里简要介绍：

- 每个对象都是 id 类型，这是个弱类型工具；
- nil 对于对象，相当于 NULL 对于指针。nil 和 NULL 是不能互换的。
- Nil 是 nil 的类的指针。在 Objective-C 中，类也是对象 (这是元类的实例)。

2.4.3 SEL

SEL 是用于存储通过使用 @selector 而获得的选择器的类型。一个选择器表示一个与对象无关的方法：我们能如同使用方法的指针一样使用它，即使技术上它并不是一个正真的指向函数的指针。见 3.3.5 以获得更多的细节。

2.4.4 @encode

基于互操作性的考虑，Objective-C 的数据类型，甚至自定义类型，函数或方法的原型，都可以被 ASCII 编码。调用 @encode(a type) 返回表示这个类型的 C 字符串。

2.5 代码组织：*.h* 和 *.m*，包含

如同在 C++ 中一样，把每个类的接口声明和实现分开是十分实用的。Objective-C 用 .h 文件作为头文件，用 .m 文件作为代码文件；.mm 文件则用于 Objective-C++ (见 14)。Objective-C 引进了 #import 指令代替 #include。实际上，C 头文件必须使用编译检查来避免多次包含。在使用 #import 时这是自动的。下面是一个有典型接口/实现的例子。Objective-C 的语法稍后解释。

C++

```
//In file Foo.h
#ifndef __FOO_H__ //compilation guard
#define __FOO_H__ //
class Foo
{
...
};
#endif
```

```
//In file Foo.cpp
#include "Foo.h"
...
```

Objective-C

```
//In file Foo.h
//class declaration, different from
//the "interface" Java keyword
@interface Foo : NSObject
{
…
}
 @end
```

```
//In file Foo.m
#import "Foo.h"
@implementation Foo
...
@end
```

2.6 类的名字：为什么有 NS?

在这份文档中，几乎所有的类的名字都以 NS 开头，例如 NSObject 或者 NSString。原因很简单：它们都是 Cocoa 的类，大部分 Cocoa 类名都以 NS 开头，因为它们继承于 NeXTStep。
用前缀来标示类的来源是一个很普遍的方法。

2.7 区分函数和方法

Objective-C 不是用方括号标示函数调用的语言。当看到如下代码时，会认为：

[object doSomething];
就是
object.doSomething();
但实际上，Objective-C 是 C 的超集，所以函数 (functions) 在声明，实现，调用的语法和语意上和 C 语言一样。另外一方面，方法 (methods)，不存在于 C 语言中，它有着方括号的特殊的语法。而且，不同之处不仅语法上，也在含义上。这会在 3.2 章详述：这不是方法调用，这是发送了一条消息。这并不简单的学术上的分别；它隐式表示了 Objective-C 的机制。即使代码结构上看上去类似，这样的机制却允许更灵活的动态性。例如，可以在运行时添加一个方法 (见 13.2)。嵌套调用的语法也更为可读 (见 3.3.1)。

3 类和对象

Objective-C 是一个面向对象语言：它管理类和对象。不象 C++ 有很多不符合理想对象模式的地方， Objective-C 使用严格的对象模。例如，在 Objective-C 中，类也是对象可以动态管理。在运行时添加类是可能，基于类的名字创建实例，调用类的方法等等。这比为一个很静态的语言 C++ 实现的部分动态功能 RTTI (见，13.1 节，) 更强大。由于结果取决于编译器并缺乏可移植性，不鼓励使用 RTTI 是很普遍的。

3.1 根类，id 类型，值 nil 和 Nil

在一个面向对象语言中，每个程序都使用一个类集合完成功能。不象 C++，Objective-C 定义乐根类。每个新类都应该是根类的子类。在 Cocoa 中，这个类是 NSObject，它提供了大量的运行时功能。根类概念不是 Objective-C 特有的；它是和对象模型相关的。 Smalltalk 和 Java 使用了根类，C++ 则没有。
严格来说，每个对象都应该是 NSObject 类型，每个指向对象的指针都可以声明为 NSObject*。实际上，我们可以使用 id 类型代替。这是个简短方便的方法来声明一个可指向任何类型对象的指针，并提供了动态类型检查，而不是 C++ 的静态类型检查。这对一些在类似范型机制的方法上使用的弱类型很有用。请注意，指向对象的 null 指针应该被置为 **nil**，而不是 **NULL**。这两个直是不能互换的。一个普通的 C 指针可以被置为 **NULL**，但 **nil** 在 Objective-C 中被引入用于作为对象的指针。在 Objective-C 中，类也是对象 (元类型实例)，声明指向类的指针是可能的，它的 null 值为 **Nil**。

3.2 类声明

很难仅用一个例子来展示 C++ 和 Objective-C 在类声明和实现上的不同。下面将逐步展示这些不同点。

3.2.1 属性和方法

在 Objective-C 中，属性被称为实例数据，成员函数被称为方法。

C++	Objective-C

class Foo
{
double x;
 	
public:
int f(int x);
float g(int x, int y);
};
 	
int Foo::f(int x) {...}
float Foo::g(int x, int y) {...}
	@interface Foo : NSObject
	{
	double x;
	}
	
	-(int) f:(int)x;
	-(float) g:(int)x :(int)y;
	@end
	
	@implementation Foo
	-(int) f:(int)x {...}
	-(float) g:(int)x :(int)y {...}
	@end

在 C++ 中，属性和方法在类的括号内声明。方法实现语法和 C 类似，但是带了指明归属的 (Foo::)。

在 Objective-C 中，属性和方法不能被混合声明。属性声明在花括号内，方法跟随在括号之后。他们的实现在 @implementation block 中。

和 C++ 比较，这是一个很大的不同，因为某些方法可以不在接口中暴露也能够实现。之后会讨论这方面的细节。简短地说，移除不必要地声明 (私有方法，和重写如析构器之类地虚方法) 是一种清理头文件方法。更多解释，请看 4.3.2 节。

实例方法以减号符 ”-” 开始，类方法以加号符 “+” 开始 (见 3.9.9 节)；这些符号和 UML 中的 public 或 private 没有联系。参数类型在括号内，参数由符号“：”分隔。请查看 3.3.1 节获得更多方法原型的解释。

在 Objective-C 中，类声明的结尾无需分号，类声明的关键字是 @interface 不是 @class。关键字 @class 只用于前向声明 (见 3.3.2 节)。最后，如果类中没有实例数据，类声明中的花括号内就什么也没有了，这时是可以省去的。

3.2.2 前向声明：@class ，@protocol

为了避免头文件循环依赖，C 语言支持前向声明 (forward declaration)，它允许程序员在仅只需知道这个类存在而无需知道其结构时，声明这个类。在 C++ 中，使用关键字 class；在 Objective-C 中，关键字是 @class。关键字 @protocol 可以被用于协议的前向声明 (见 4.4 节)。

C++

//In file Foo.h
#ifndef __FOO_H__
#define __FOO_H__
 	
class Bar; //forward declaration
class Foo
	//In file Foo.cpp
	
	#include "Foo.h"
	#include "Bar.h"
	void Foo::useBar(void)

```
{
Bar* bar;

public:
    void useBar(void);
};
#endif
```

```
{
…
 }
```

Objective-C

```
//In file Foo.h

@class Bar; //forward declaration
@interface Foo : NSObject
{
  Bar* bar;
}

-(void) useBar;

@end
```

```
//In file Foo.m
#import "Foo.h"
#import "Bar.h"
@implementation Foo
-(void) useBar
{
…
 }

@end
```

3.2.3 public, private, protected

对象模型的一大功能是数据封装。为了封装，需要对于不同部分的代码数据有不同的可见性。

C++	Objective-C

`class Foo`
`{`
` public:`
`    int x;`
`    int apple();`
 	
` protected:`
`    int y;`
`    int pear();`
 	
` private:`
`    int z;`
`   int banana();`
`};`
	`@interface Foo : NSObject`
	`{`
	` @public`
	`    int x;`
	
	` @protected:`
	`    int y;`
	
	` @private:`
	`    int z;`
	`}`
	
	`-(int) apple;`
	`-(int) pear;`
	`-(int) banana;`
	`@end`

在 C++ 中，参数和方法可以是 public，protected 或 privated 可见性。缺省的是 private。
在 Objective-C 中，实例数据可以是 public，protected 或 private，缺省的是 protected。方法只能是 public。不过，是可以通过不在 @interface 中声明只在 @implementation 中实现方法，或

者使用目录 (见 4.5 节)，来模拟私有方法的。这并不能阻止方法被调用，但是它们被尽量少地暴露了。不提前声明而实现方法是 Objective-C 的一个特性，这将在 4.3.2 节解释。
继承不能被标记为 public，protected 或 protected。唯一的继承方式是 public 的。Objective-C 的继承更象 Java 而不是 C++。

3.2.4 静态属性

在 Objective-C 中是不能为声明类的数据变量 (即 C++ 中的 static)。可是，有一些变通的方法：在实现文件中使用一个全局变量 (可以使用 C 的关键字 static 限制其可见域)。类可以使用访问器访问全局变量 (使用类方法或者普通方法)，它的初始化可以在类的 initialize 方法中完成 (见 5.1.10 节)。

3.3 方法

Objective-C 的方法的语法和 C 函数语法相差很大。这一节的目的是描述这些语法，和消息发送的相关原理。

3.3.1 原型和调用，实例方法，类方法

- 实例方法以 “-” 开始，类方法以 “+” 开始 (即 C++ 中的 static 方法)。这两个符号和 UML 中的私有和公有概念没有联系。Objective-C 中的所有方法都是公有的。
- 返回值的类型，参数类型，用括号关闭；
- 参数以符号 “：” 分隔；
- 参数和一个标签关联，这个标签在 “：” 之前；标签是方法名的一部分。它使得方法调用更易读。实际上，标签的使用应该系统化。注意第一个参数没有标签 - 它的标签就是方法的名字。
- 一个方法名可以和属性名相同，这不会有冲突。这使得 getters 方法更容易实现(见 6.4.8 节)。

C++

`//prototype`
`void Array::insertObject(void *anObject, unsigned int atIndex)`
 	
`//使用数组类的实例"shelf"和"book"对象`
`shelf.insertObject(book, 2);`

Objective-C
无标签 (直接翻译自 C++)

```
//原型
//方法名为"insertObject::", 冒用于分隔参数(这和 C++ 的指定作用的冒号符不同)
-(void) insertObject:(id)anObject:(unsigned int)index

//使用数组类的实例"shelf"和"book"对象
[shelf insertObject:book:2];
```

有标签

```
//原型 参数"index"的标签是"atIndex"
//现在方法的名字是"insertObject:atIndex:"
//调用可以很容易的阅读，如同句子一般
-(void) insertObject:(id)anObject atIndex:(unsigned int)index

//使用数组类的实例"shelf"和"book"对象
[shelf insertObject:book:2];          //错误!
[shelf insertObject:book atIndex:2]; //正确
```

请注意，方括号的语法不应该被视为对象 “shelf” *调用了方法* insertObject, 而应该视为向对象 “shelf” 发送消息 insertObject。这表现了 Objective-C 的动态性。我们可以向任何对象发送任何消息。如果它无法处理这个消息，它可以忽略消息(一个异常会被抛出，但程序不会中止)。如果在接收到消息时，对象可以处理它，相关的方法才会被调用。如果向一个没有匹配的方法的类发送消息，编译器会抛出警告(warning)。由于前向编译，这并不没有被认为是错误(error)(见第 3.4.3 节)。如果一个对象是 id 类型，编译期就没有警告，潜在的错误会在运行时出现。

3.3.2 this, self 和 super

对于消息而言，有两个特别的发送对象：**self** 和 **super**。**self** 是指当前对象 (如同 C++ 中 的 **this**)，**super** 是指父类。关键词 **this** 在 Objective-C 中并不存在。它由 **self** 代替。
注意：**self** 并不是一个真正的关键字，它是每个方法的一个隐藏参数，参数值就是当前对象。和 C++ 的 **this** 关键字不同，**self** 的值可以改变。但是，这只在构造器中有用(见 5.1 节)。

3.3.3 在方法内部访问实例变量

如同 C++ 一样，Objective-C 方法可以访问其所在对象的实例变量。C++ 中 this-> 可对应写为 self-> 。

C++	Objective-C

`class Foo {`
`  int x;`
`  int y;`
`  void  f(void);`
	`@interface Foo : NSObject`
	`{`
	`  int x;`
	`  int y;`

```
};

void Foo::f(void)
{
  x = 1;
  int y; //和 this->y 有歧义
  y = 2; //使用了本地变量 y
  this->y = 3; //消除歧义，实例变量赋值
}
```

```
}

-(void) f;

@end


@implementation Foo

-(void) f
{
  x = 1;
  int y; //和 this->y 有歧义
  y = 2; //使用了本地变量
  self->y = 3; //消除了歧义
}

@end
```

3.3.4 原型 id 和 签名，重载

函数是一部分可以被引用的代码，比如以函数指针或者仿函数(C++ 中的 functor 或者称为 函数对象 function object)的形式使用。而且，即使函数名是函数唯一标示的很好的备选对象，但也必须考虑重载的情况。C++ 和 Objective-C 使用不同的方法区别函数原型。C++ 基于参数类型，Objective-C 基于参数标签。

在 C++ 中, 两个不同的函数可以拥有相同的名字，和相同的参数名，而仅仅参数类型不同。函数有无 const 标示也可以用于重载函数。

C++

```
int f(int);
int f(float); //正确，float 区别于 int

class Foo {
 public:
    int g(int);
    int g(float); //正确，float 区别于 int
    int g(float) const; //正确, const 可以区分
};

class Bar {
 public:
    int g(int); //正确, 在 Bar 类中(Bar::), 区别与在 Foo 类中(Foo::)
}
```

在 Objective-C 中, 所有函数都是 C 函数：他们不能被重载(除非编译器被告知使用 C99, 如同 C++ 那样)。可是，Objective-C 方法使用了不同语法，可以由参数标签区分。

Objective-C

```
int f(int);
int f(float); //Error : C functions cannot be overloaded

@interface Foo : NSObject
{
}

-(int) g:(int) x;
-(int) g:(float) x; //错误:Objective-C 认为这个方法和前一个相同(无标签)

-(int) g:(int) x :(int) y;   //正确: 两个匿名标签
-(int) g:(int) x :(float) y; //错误: 和上一个相同

-(int) g:(int) x andY:(int) y;   //正确:第二个标签为 "andY"
-(int) g:(int) x andY:(float) y; //错误:和上一个相同

-(int) g:(int) x andAlsoY:(int) y; //OK:第二个标签为"andAlsoY", 不同于"andY"
@end
```

基于标签标示的方法可以很好的描述函数的正真名字，如下所示。

```
@interface Foo : NSObject {}

//The method name is "g"
-(int) g;

//The method name is "g:"
-(int) g:(float) x;

//The method name is "g::"
-(int) g:(float) x :(float) y;

//The method name is "g:andY:"
-(int) g:(float) x andY:(float) y;

//The method name is "g:andZ:"
-(int) g:(float) x andZ:(float) z

@end
```

很清晰的，两个 Objective-C 方法是用标签(*label*)而不是类型(*type*)区分的。用选择器(*selectors*) 和标签标示方法可以代替“成员函数指针”，这会在 3.3.5 节解释。

3.3.5 成员函数指针：选择器

在 Objective-C 中，方法使用了括号和标签的特别语法。用这种语法是无法声明函数的。函数指针的概念在 C 和 Objective-C 中是相同的。区别在于指向方法的指针。

在 C++ 中，语法是很难懂的，但和 C 语言保持一直：指针基于类型。

C++

```
class Foo {
 public:
    int f(float x) {...}
};

Foo bar
int (Foo::*p_f)(float) = &Foo::f; //Pointer to Foo::f
(bar.*p_f)(1.2345); //calling bar.f(1.2345);
```

在 Objective-C 中，有一个新类型被引入。象“方法指针”一样被称为选择器。它的类型是 SEL ，它的值用 @selector 和方法的全名(函数名和参数标签)计算出来。调用方法可以使用 NSInvocation 类。大部分时候，一系列工具方法 performSelector: (继承自 NSObject) 更为顺手一些，但是也有更多限制。最简单的三个是：

-(id) performSelector:(SEL)aSelector;
-(id) performSelector:(SEL)aSelector withObject:(id)anObjectAsParameter;
-(id) performSelector:(SEL)aSelector withObject:(id)anObjectAsParameter
withObject:(id)anotherObjectAsParameter;

它们的返回值都是被调用的方法。如果方法的参数不是对象，就应该使用如 NSNumber 的包装类，这样就可以传送 float, int, 等等。NSInvocation 类更为强大，可适用的场景也更多(参考文档)。

如前面的章节提到的，没有什么可以阻止一个方法在对象上的调用，即使对象并没有实现该方法。实际上，如果消息被接受了，方法就会有效地触发。在对象不认识这个方法时，一个可以被捕捉地异常被抛出了；应用并没有中断。而且，调用 respondsToSelector: 是可以判断一个对象是否能触发某个方法。

最后，@selector() 的值是在编译器被计算出来的，它并不会减低代码的速度。

Objective-C

```
@interface Slave : NSObject {}
  -(void) readDocumentation:(Document*)document;
@end

//让我们假设数组 array 有10个对象，以及一个文件对象"document"

//普通的方法调用
for(i=0 ; i<10 ; ++i)
 [array[i] readDocumentation:document];
//仅仅作为例子，使用 performSelector: 替代
for(i=0 ; i<10 ; ++i)
 [array[i] performSelector:@selector(readDocumentation:)
                withObject:document];
```

```
//选择器的类型是 SEL
//下面的版本并不比上一个版本更高效，因为 @selector() 是在编译时转化的
SEL methodSelector = @selector(readDocumentation:);
for(i=0 ; i<10 ; ++i)
 [slaves[i] performSelector:methodSelector withObject:document];

//对象"foo"的类型不是(id)
//测试不是强制性的，但是可以阻止异常产生，如果对象没有 readDocumentation: 方法的话。
if ([foo respondsToSelector:@selector(readDocumentation:)])
 [foo performSelector:@selector(readDocumentation:) withObject:document];
```

选择器可以被作为非常简单的函数参数使用。通用的算法中，如排序，可以简单地指定对比函数(见第 11.3 节)。

选择器严格说来不是一个指向函数地指针；它真正的相关类型是 C 语言字符串，在运行时作为方法标示被注册。当类被加载了，它的方法也自动被注册在一个表格里，所以@selector() 可以如预期一样工作。如此，两个选择器相等可以用 == 比较地址而不是比较字符串。

方法的实际地址，如同 C 函数一样，可以使用另外一个类型 IMP 获取。这个类型在 11.3.2 节做了简要介绍，它除了优化之外，很少被使用。虚调用是由选择器处理而不是 IMP 类型。在 Objective-C 中等价于 C++ 的方法指针的是选择器。

最后，你可以记住 Objective-C 中的 self 就如 C++ 中的 this 一样，是每个方法的一个值为当前对象的隐藏参数。第二个隐藏参数 _cmd 其值为当前方法。

Objective-C

```
@implementation Foo

-(void) f:(id)parameter //等价于 C 函数的类型：
                        //"void f(id self, SEL _cmd, id parameter)"
{
  id currentObject = self;
  SEL currentMethod = _cmd;
  [currentObject performSelector:currentMethod
                      withObject:parameter]; //递归调用
  [self performSelector:_cmd withObject:parameter]; //同上
}

@end
```

3.3.6 参数缺省值

Objective-C 不允许为函数或方法参数赋予一个特定的缺省值。若是某些参数只是可选的，我们可以按需多创建几个参数数量不同的方法。在构造器的例子中，我们应该使用缺省构造器 (见 5.1.7 节)。

3.3.7 可变参数

Objective-C 允许可变参数。如同 C，可变参数的语法是以 "..." 为最后一个参数。虽然很多 Cocoa 方法使用了可变参数，但实际编程中，可变参数用得很少。更多细节可以查看 Objective-C 文档。

3.3.8 匿名参数

在 C++ 中，方法原型中的参数是可以没有名字的，因为参数类型对于函数签名来说就足以区别了。这在 Objective-C 中是不可能的。

3.3.9 原型修饰符 (const, static, virtual, "= 0", friend, throw)

在 C++ 中，一些修饰符可加在函数原型中。这些在 Objective-C 中都不存在。见如下列表：

- const : 一个方法不可以被修饰为 const。所以，关键字 mutable 也没有必要存在。

- static : 区分类方法和实例方法是通过在函数原型之前的 "-" 和 "+" 来区分的。

- virtual : 所有的 Objective-C 方法都是虚方法，所以这个关键字是无用的。纯虚方法以协议形式实现 (见 4.4 节)。

- friend : Objective-C 的类或方法没有这个概念。

- throw : 在 C++ 中，我么可以限制一个方法只抛出某几种类型的异常。在 Objective-C 中，我们无法这样做。

3.4 消息和发送

3.4.1 向 nil 发送消息

缺省情况下，向 nil 发送消息是合法的。消息只是被忽略。由于减少了空指针测试，代码能大大简化。GCC 有一个选项可以关闭这个方便的缺省行为，这可以得到额外的优化效果。

3.4.2 代理向未知对象发生消息

代理是 Cocoa 向用户暴露框架组件的常用方法(如，列表)，这可以充分发挥向未知对象发送消息的优势。例如，一个对象可以将某些任务代理给助手。

```
// 这个函数定义了助手
-(void) setAssistant:(id)slave {
    [assistant autorelease]; //查看关于内存管理的章节
    assistant = [slave retain];
}
```

```
// 方法 performHardWork 可以使用代理
-(void) performHardWork:(id)task {
   //助手未知，我们检查助手是否可以处理这条消息
   if ([assistant respondsToSelector:@selector(performHardWork:)])
     [assistant performHardWork:task];
```

```
   else
     [self findAnotherAssistant];
}
```

3.4.3 转发：处理未知消息

在 C++ 中，如果一个对象调用了一个它没有实现的方法，代码是无法通过编译的。在 Objective-C 中，有些不同：我们总是可以向对象发送消息。如果在运行时，消息无法被处理，它将被忽略(同时 抛出一个异常)；而且，你也可以不忽略消息，而将它转发给另一个对象。

当编译器被告知对象的类型，它就可以测试出消息发送 - 方法调用 - 是否会失败，并抛出警告。但是，这不是错误，因为在这种情况下，是有解决方法存在的。解决方法就是调用**转发调用**(forwardInvocation)：一个最后能被定义为将消息重定向到其它对象的方法。它明显应该是一个 NSObject 类的方法，缺省情况下它什么也不做。下面是另外一种管理助手对象的方法。

```
-(void) forwardInvocation:(NSInvocation*)anInvocation {
    //如果我们在这儿，说明对象不能处理消息调用
    //anInvocation 的 selector 方法可以得到不能处理的消息调用的方法
    if ([anotherObject respondsToSelector:[anInvocation selector]])
        [anInvocation invokeWithTarget:anotherObject];
    else //不要忘记调用父类的本方法
         [super forwardInvocation:anInvocation];
}
```

即使消息最后在 forwardInvocation: 中被处理了，respondsToselector: 检查仍可能返回 NO （仅在这种情况下）。实际上，repondsToSelector：机制不是设计用于猜测 forwardInvocation: 是否可以工作。

使用转发调用被认为是一个坏的实践，因为这是错误发生时触发了一些代码。实际上，好的应用方式如同 Cocoa 的 NSUndoManager 的实现一样。它允许使用更优美的方式：撤销管理可以记录方法的调用，即使它不是那些调用的目标。

3.4.4 向下转型

C++ 中，向下转型是在只有父类对象指针，却需要调用子类方法时使用，通过关键字 dynamic_cast 就可以实现了。在 Objective-C 中，向下转型不是必须的，因为消息可以被发送给任何对象，即使对象无法处理它。

可是，为了避免编译警告，我了可以简单转变对象的类型；Objective-C 中，没有特别的显式的向下转型操作符，可以使用传统的 C 语言的转型语法。

```
//NSMutableString 是 NSString (字符串)的子类，它运行写操作
//方法"appendString:" 只存在于 NSMutableString
NSMutableString* mutableString = ...initializing a mutable string...
NSString* string = mutableString;//保存在 NSString 指针
```

```
//下面不同的调用都是合法的
[string     appendString:@"foo"];//编译器警告
[(NSMutableString*)string appendString:@"foo"];//编译器警告
[(id)string appendString:@"foo"];//编译器警告
```

4 继承

4.1 简单继承

Objective-C 显式实现了继承的概念，但是不支持多重继承，这个局限可以由其他概念(协议，目录)弥补。这些将在后面的章节解析(见 4.4 节)。

C++	Objective-C

`class Foo : public Bar,`
`protected Wiz`
`{`
`}`
	`@interfac Foo : Bar`
	`// single inheritance`
	`// An alternative technique must be used`
	`// to also "inherit" from wiz`
	`{`
	`}`
	`@end`

在 C++ 中，一个类可以继承于一个类或者若干个类，有公开，受保护，私有 继承三种模式。
在方法中，我们可以使用作用域操作符 :: (Biz ::, Wiz ::) 引用父类。
在 Objective-C 中，只能使用公开继承模式从一个类继承。一个方法可使用关键字 super 引用父类，如同 java 一样。

4.2 多继承

Objective-C 没有实现多继承，但是他引入了其它可以实现和多继承效果近似的概念，协议(*protocols* 见 4.4 节) 和 目录 (*categories* 见 4.5 节)。

4.3 虚拟化

4.3.1 虚方法

在 Objective-C 中，所有方法都是虚方法。因此，关键字 *virtual* 并不存在，也没有等价物。

4.3.2 虚方法的无声明重写

在 Objective-C 中，不在接口描述中声明也可以实现一个方法。这个功能不能代替方法的 @private 概念 (即便可以隐藏方法)：它可以被调用；但是它不在接口声明中明示。
这不是一个坏的实践：使用这种技术的方法通常十分“熟悉”父类方法。很多根类 NSObject 的方法都被无声明重写了。一个例子就是构造器 init (见 5.1 节)，析构器 dealloc (见 5.2)，绘制视图类 (view) 的方法 drawRect:，等等。

虽然这样对查看哪些方法可以被重定义造成了麻烦，需要经常阅读父类文档。但接口因此清晰简洁，易读很多。
纯虚方法的概念 (必须在子类中重写的方法)，是为正式协议 (*formal protocols*，见 4.4.1节)准备的。

4.3.3 虚继承

在 Objective-C 中，没有虚继承对应概念。因为没有多继承，所以没有和多继承相关的虚继承的问题。

4.4 协议

Java 和 C# 使用接口(interface) 弥补没有多继承的局限。在 Objective-C 中，有一个相同的概念被使用，但是被称为协议(protocol)。在 C++ 中，相同的概念是抽象类。一个 Objective-C 协议不是一个真正的类：它只能声明方法，但不能拥有任何数据。这有两种类型的协议：正式协议和非正式协议。

4.4.1 正式协议

一个正式协议是一个必须被符合此协议的类实现的方法集合。这可以看作是一个类的证书，保证其能处理某些给定的服务。一个类可以符合不限数量的协议。

C++

```
class MouseListener
{
public :
    virtual bool mousePressed(void) = 0; // pure virtual method
    virtual bool mouseClicked(void) = 0; // pure virtual method
};

class KeyboardListener
{
    public:
        virtual bool keyPressed(void) = 0; // pure virtual method
};

class Foo : public MouseListener, public KeyboardListener {...}

// Foo 必须实现 mousePressed, mouseClicked 和 keyPressed
// 这样它就可以被用作鼠标和键盘事件的监听器
```

Objective-C

```
@protoc0l MouseListener
    - (void) mousePressed;
    - (void) mouseClicked;
@end
```

```
@protocol KeyboardListener
   - (void) keyboardPressed;
   - (void) keyboardClicked;
@end

@interface Foo : NSObject <MouseListener, KeyboardListener>
{
...
}
@end

// Foo 必须实现 mousePressed, mouseClicked 和 keyPressed
// 这样它就可以被用作鼠标和键盘事件的监听器
```

在 C++ 中，协议是由抽象类和纯虚方法实现的。C++ 的抽象方法可以包含数据对象，这比 Objective-C 协议强大。

在 Objective-C 中，协议是一个特殊的概念。语法使用尖括号< … >，这和 C++ 的模板没有联系，Objective-C 中并不存在模板这个概念。

一个类不声明符合协议，也可以实现所有的协议方法。这种情况下，confromsToProtocol: 方法返回 NO。由于效率原因，conformsToProtocol:并不一一检验每个方法是否符合协议，而是依赖于开发者的显式声明。可以，conformsToProtocol: 的否定的返回值并不意味当协议方法被调用时，程序行为一定是错误的。下面是 confromsToProtocol: 的原型：

```
- (BOOL) conformsToProtocol:(Protocol *)protocol
// @protocol (协议名) 会返回一个协议对象
```

符合正式协议的对象的**类型**可以将协议名称添加在一对尖括号之中，加入到类名中。这对于断言是很有用的，例如：

```
// 下面 Cocoa 标准方法可以接受一个任意类型对象，但是必须符合 NSDraggingInfo 协议
- (NSDraOperation) draggingEntered:(id <NSDraggingInfo>)sender;
```

4.4.2 可选方法

也许可以描述这一个符合某协议的类：它可以处理特定的服务，但是并不强制实现所有协议方法。比如，在 Cocoa 中，代理对象的概念就广泛地被使用：一个对象可以辅助处理某些任务，但不是所有的任务。

一个立即的解决方法是将一个正式协议切分成多个，使类符合这些协议的一个子集。这不是很方便。Cocoa 带来了一个叫做非正式协议 (inform protocols) 的解决方案。在 Objective-C 1.0 中，非正式协议被使用到了 (见 4.4.3 节)。在 Objective-C 2.0 中，新关键字 @optional 和 @required 用于区分可选和必须方法。

```
@protocol Slave
@required //必须部分
-(void) makeCoffee;
-(void) duplicateDocument:(Document*)document count:(int)count;
@optional //可选部分
-(void) sweep;
@required //你可以间隔可选和必须部分
-(void) bringCoffee;

@end
```

4.4.3 非正式协议

非正式协议不是一个真正的“协议”：它不是一个代码约束工具。另一方面，它性质上是非正式的，同时是为了使代码自文档化。

一个非正式协议使开发者按需求组织方法，所以可以使类组织更一致。

所以，并不奇怪，非正式协议并没有用某种不严格的正式协议来声明，而是使用了另外一个概念：类的目录 (class category) (见 4.5 节)。

让我们想象一个名为 “文件管理” 的服务。假设这里有绿，蓝和红三种不同的文件。即使只能处理蓝色文件，slave 类也可能需要使用三个正式协议：manageGreenDocuments, manageBlueDocuments 和 manageRedDocuments。对于 Slave 类而言，更好的方法可能是添加一个目录 DocumentsManaging，用于声明它可以完成这项任务。目录的名字指定在括号内 (更多解释见 4.5 节)。

```
@interface Slave (DocumentsManaging)
-(void) manageBlueDocumnets:(BlueDocument*)document;
-(void) trasBlueDocoments:(BlueDocument*)document;
@end
```

任何类都可以使用 DocumentsManaging 目录，去声明与服务相关的方法。

```
@interface PremiumSlave (DocumentsManaging)
-(void) manageBlueDocuments:(BlueDocument*)document;
-(void) manageRedDocuments:(RedDocument*)document;
@end
```

开发者浏览代码看到 DocumentsManaing 目录。于是，他可以假设这个类对于某些任务是有用的，他可以通过查看文档检查任务详情。即使他不查看源代码，在运行时，这样的调用仍然是可能的：

```
if ([mySlave respondsToSelector:@selector(manageBlueDocuments:)])
    [mySlave manageBlueDocumentsdocument];
```

严格地讲，除开认识原型，非正式对象对编译器来说是没有用的，它并没有规范类的使用。但是，他对于自文档化代码很重要，使得 API 更易读。

4.4.4 协议类型对象

在运行时，一个协议代表了一个类的对象，它的类型是 Protocol*。这样一个对象可以作为参数传给方法 conformsToProtocol: (见 13.1.2 节)。

关键字 @protocol，被用于声明一个协议，也被用于创建一个 Protocol * 协议对象：

```
Protocol *myPotocol = @protocol (protocol name)
```

4.4.5 对远程对象的消息筛选

由于 Objective-C 的动态性，远程对象之间的很容易交流。他们可以属于不同机器上的不同程序，但是可以代理某些任务，交换信息。正式协议是一个保证无论来自何处的对象都可以提供某种服务的完美的方法。正式协议的概念提供了一些额外的关键字，可使得远程对象之间的交流更有效率。

这些关键字被插入在正式协议内声明的方法原型之中，用于添加关于它们行为的额外信息。他们可以用于指明哪些参数是输入参数，哪些是输出结果；也可以用于告只哪些可被用于拷贝或者引用；方法是否是同步的。这有一些不同定义：

- **in** 参数是输入变量；
- **out** 参数是输出变量；
- **inout** 参数即可输入，也可输出；
- **bycopy** 参数被拷贝传送；
- **byref** 参数被以引用传送；
- **oneway** 方法是异步的 (结果不能马上得到) - 因此它必须返回 void

例如，这是一个返回一个对象的异步方法：

```
-(oneway void) giveMeAnObjectWhenAvailable:(bycopy out id*)anObject;
```

缺省的，除了 const 指针是 in 的，参数被认为是 inout 的。选择 in 或者 out 代替 inout 是种优化。缺省的传送模式是 byref，方法缺省为同步的 (不声明 oneway)。

对于参数值传送，如非指针变量，out 和 inout 没有意义的，只有 in 是正确的。

4.5 类的目录 (Category)

为类创建目录是一种将类实现切分为几个部分的方法。每一个目录都是类的一部分。一个类可以有任意数目的目录，但是都不能添加实例数据。他带来了如下的好处：

- 对于要求精细的开发者，这可以将方法分类的。对于一个庞大的类，它的不同角色可以很清晰的划分；
- 可以分离地编译它们，这使得合作工作在同一个类是可能的；
- 如果目录的接口描述和实现都在同一个实现文件中(.m file)，那么它可以被定义为只有在显示文件内部才可见的私有方法 (但任何知道原型的人都可以使用它，这里没有调用限制)。对于这种目录有个合适的名字 ***FooPrivateAPI***；

- 一个类可以在不同的应用中通过目录被扩展，而不用拷贝相同的类代码。任何类都可以通过目录被扩展，甚至 Cocoa 中的类。

最后一点是很重要的：每个开发者都可以在标准类上扩展符合他们各自要求的方法。这不是一个真正的问题：继承也可以做到。但是，简单继承可能会导致很复杂的继承谱图。而且，为单个方法而创建一个类，这有点得不偿失。对于这个问题，类的目录是一个优雅的解决方法。

C++	Objective-C

class MyString : public string
{
public:
//元音计数
int vowelCount(void);
 	
};
 	
int MyString::vowelCount(void)
{
...
}
	@interface NSString (VowelsCounting)
	//请注意没有使用括号{}
	-(int) vowelCount; //元音计数
	@end
	
	@implementation NSString
	(VowelsCounting)
	-(int) vowelCount
	{
	...
	}
	@end

在 C++ 中，新类的的使用是没有任何限制的。

在 Objective-C 中，目录给予 NSString 类 (一个标准 Cocoa 类) 一个可以在整个程序中使用的扩展。没有新的类被创建。每个 NSString 类都可以使用目录的扩展 (即使不变字符串，见 9.1 节)。但是不能在目录中添加实例数据，所以没有括号 {} 块。

一个目录甚至可以是匿名的，这是很好实现私有的方法。

```
@interface NSString ()
// 请注意没有括号 {} 被使用
-(int) myPrivateMethod;
@end

@implementation NSString ()
-(int) myPrivateMethod
{
….
}
@end
```

4.6 协议，目录，继承的联合使用

联合使用协议，目录和继承的唯一限制是，子类和目录不能同时声明，必须分为两步。

```
@interface Foo1: SuperClass<Protocol1, Protocol2, ...> //OK
@end

@interface Foo2 (Category) <Protocol1, Protocol2, ...> //OK
@end
```

```
//blew : compiler error
@interface Foo3 (Category) : SuperClass <Protocol1, Protocol2, ...>
@end

// a solution:
@interface Foo3 : SuperClass <Protocol1, Protocol2, ...>  //setp 1
@end

@interface Foo3 (Category)  //setp 2
@end
```

5 实例化

类的实例化引出了两个问题：构造器 / 析构器 / 复制操作的实现，以及如何在内存中管理他们？
首先，很重要的一点：在 C 和 C++ 中，除非变量被声明为 static，否则它们缺省是"自动的" 的，只在它们定义的块存在。只有在动态内存分配被使用的情况下，变量可以一直使用，直到调用了 free() 或 delete 方法。 C++ 遵循这个规则。
然而，在 Objective-C 中，所有对象都是动态分配的。只是相当符合逻辑的，C++ 很静态，而 Objective-C 很动态。如果对象不是在运行时创建，Objective-C 不会有这么丰富的动态性。
请看第 6 章关于 retain 和 release 对象的方法的详细解释。

5.1 构造器，初始化器

5.1.1 区别分配和初始化

在 C++ 中，对象的初始化和分配都是在构造器被调用时进行的。在 Objective-C 中，它们是两个不同的方法。
分配是由**类方法** alloc 完成的，它也初始化所有的实例数据。实例数据都被置为 0，除了 NSObject 的 isa (is-a) 指针外，isa 指针值描述的是在运行时新创建的对象的确切类型。实例数据是否被置为特定的值，取决于构造参数，相关代码代码被置于一个**实例方法**之中。它的名字通常由 init 开始。因此，构造被清晰地分为两个步骤：分配和初始化。alloc 消息被发送至类，init... 消息被发送至有 alloc 实例化新生成的对象。
初始化过程不是可选的，alloc 必须紧随 init；连续调用父类的初始化器，这必会终结于的 NSObject 的 init 方法的调用，NSObject 的 init 方法完成了很多重要工作。
在 C++ 中，构造器的名字是不可选的。在 Objective-C 中，初始化器是一个普通方法，以 init 为前缀只是传统而非强制。但是，强烈建议你遵守这个规则：**初始化器的名字必须以 "init" 开始。**

5.1.2 使用 alloc 和 init

调用 alloc 返回的新对象必须马上被调用 init。init 的调用返回一个对象。大多数时候，这将是一个初始对象。有时，当使用单例的时候 (一个类只有允许有一个实例对象)，init 可被替换返回另外一个值。所以，init 的返回值不应被忽略。通常，alloc 和 init 会被同时调用。

C++

```
Foo* foo = new Foo;
```

Objective-C

```
Foo* foo1 = [Foo alloc];
[foo1 init]; // 错误：返回值应该被用到

Foo* foo2 = [Foo alloc];
foo2 = [foo2 init]; //正确，但不简便

Foo* foo3 = [[Foo alloc] init]; //正确，通常做法
```

为了知道对象是否被创建了，C++ 要求要么捕捉异常或者非0测试 (如果 new(nothrow) 被使用了)。对于 Objective-C，非 nil 测试便够了。

5.1.3 正确的初始化器的例子

正确初始化器的限制有

- 名字以 init 开始；
- 返回一个对象；
- 调用超类的 init 方法，所以 NSObject 的 init 方法最终会被调用；
- 考虑了 [super init...] 的返回值；
- 正确处理了创建错误，无论是本类的还是来源于继承的。

下面给了 C++ 和 Objective-C 的代码示例。错误处理也在其中。

C++

```
class Point2D {
  public:
    Point2D(int x, int y);
  private:
    int x;
    int y;
 };

Point2D::Point2D(int unX, int unY) {x = unX; y = unY;}
...
Point2D  p1(3,4);
Point2D* p2 = new Point2D(5, 6);
```

Objective-C

```
@interface Point2D : NSObject {
   int x;
   int y;
}

//注意 : "id" 在 Objective-C 有点类似于 "void*"。
//(id) 一个对象最泛的类型
-(id) initWithX:(int)unX andY:(int)unY;
@end
@implementation Point2D
-(id) initWithX:(int)unX andY:(int)unY
{
 //父类的 init 方法必须被调用
  if (!(self = [super init])) //如果父类是 NSObject，初始化过程有错误，
   return nil;                //则必须返回 nil。

  self->x = unX;//成功的情况下，做额外的初始化
  self->y = unY;
  return self; //返回对象自己
}
@end
...
Point2D* p1 = [[Point2D alloc] initWithX:3 andY:4];
```

5.1.4 self = [super init …]

最让人惊讶的创建语法是 self = [super init...]。再次强调 self 是传递给所有方法的隐藏参数，表示当前对象。因此，它是一个本地变量，那为什么我们要更改它的值呢？ Will Shipley[9] 试图在一份很有趣的文档里展示这是无用的。他的论点基于一个关于 Objective-C 运行时的假设，可是这个假设是错误的。实际上，self 必须被修改，后面有更多的解释。

修改必须发生是因为，[super init] 返回的对象并不是当前对象。单例就是例子，Will Shipley[9] 的例子是一个：一个单例调用两次 init 是不合逻辑的。这揭示了一个概念上的错误。

可是，API 可以将一个新分配的对象替换为另一对象。Core Data 就是这样处理一些和数据库某些字段相关的实例数据的。当你编写 Cocoa 提供的 NSManagedObject 类的子类时，必须要注意这种替换。

在这种情况下，self 将被连续赋予两个不同的值：第一个值是由 alloc 返回的，第二个是由 [super init...] 返回的。修改 self 的值有负面效益：每次访问成员数据都是隐式地使用它，就像下面代码展示的那样。

```
@interface B : A
{
  int i;
}
```

```
@end

@implementation B

-(id) init {
  //在这一步，self 的值是 alloc 的返回值
  //让我们假设 A 做了替换，返回了一个不同的 “self”
  id newSelf = [super init];
  NSLog(@"%d", i);     //打印 self->i 的值
  self = newSelf;      //可能会想 “i” 没有被修改，但是...
  NSLog(@"%d", i);     //打印 self->i,  显示的是 newSelf->i,
                       //和之前的显示的不同
  return self;
}

@end
...
B* b = [[B alloc] init];
```

self = [super init] 形式简洁，是最简单的避免事后引入 bugs 的方法。可是，应当注意“旧” self 所指向的对象的结果：它应该被释放。

第一条规则很简单：谁替换了 self 的值，谁就负责处理旧的 self。这里，是 [super init...] 必须负责释放。例如，如果你编写 NSManagedObject (一个 Cocoa 类，它会执行替换) 的子类，你无需关系旧的 self。相反，NSManagedObject 的开发者必须正确处理它。

从而，如果你恰好开发一个类会执行替换，你必须知道如何在它的初始化过程中释放一个对象。这个问题和处理错误一样：在创建对象错误时，我们应该做些什么呢 (不合法的参数，不可用的资源...)？这在 5.1.5 得到了回答。

5.1.5 初始化失败

在创建一个对象时，可能会在如下三个地方出现错误：

- 1：调用 [super init...] 之前：如果创建参数被认为是非法的，初始化应该即时中止；
- 2：调用 [super init...] 之时：如果父类就失败了，我们将放弃当前的初始化；
- 3：调用 [super init...] 之后：比如，如果一个附加的资源申请分配失败。

在每个例子中，nil 必须被返回，而且触发错误的类必须负责释放当前对象。这里，我们负责在情况 1 和 3 释放对象，情况 2 无需处理。我们只需要调用 [self release] 以释放当前对象，这是最自然的方法 (第 6 章 介绍内存管理，将解释 release)。

对象的销毁终结于 dealloc 的调用 (见 5.2 节 的析构器)；析构器的实现应该和初始化对象的过程对应。alloc 中将所有实例数据都初始化为 0 是相当有用的。

```
@interface A : NSObject {
  unsigned int n;
}
```

```
-(id) initWithN:(unsigned int)value;
@end

@implementation A

-(id) initWithN:(unsigned int)value
{
  //情况＃1 合法的创建
  if (value == 0) //这里，我要求一个严格的正数值
  {
    [self release];
    return nil;
  }

  //情况 #2 (父类部分是否正确?)
  if (!(self = [super init])) //父类负责替换 self，错误发生，父类负责释放 self
    return nil;

  //情况 #3 (初始化能否完成?)
  n = (int)log(value);
  void* p = malloc(n); //试图申请一个资源
  if (!p)              //在申请不成功的情况下，需要处理错误，释放 self
  {
    [self release];
    return nil;
  }

}
@end
```

5.1.6 将构造拆分为 alloc + init

在某些情况下，连续使用 alloc 和 init 听起来挺累人的。幸运的是，这可以被便捷构造器简化。这种构造器需要牵涉到 Objective-C 的内存管理概念。所以，文档的 6.4.6 节将给出确切的解释。简短地说，这样一个构造器，名字需要由类名开头，行为象一个 init 方法，但需要自己执行 alloc 方法。对象会在 autorelease 池注册 (见 6.4 节)，如果不被发送 retian 消息，它将只有一个有限的生命周期。这是一个例子：

```
//累人的
NSNumber* tmp1 = [[NSNumber alloc] initWithFloat:0.0f];
...
[tmp1 release];

//便捷的
```

```
NSNumber* tmp2 = [NSNumber numberWithFloat:0.0f];
...
//no need to release
```

5.1.7 缺省构造器

缺省构造器在 Objective-C 中没有真正的意义。因为，所有的对象被动态分配，它们的创建总是显示的。可是，一个首选的构造器可以被用于简化代码，正确的构造器大多数时候类似于：

```
if (!(self = [super init])) //"init" 或其他合适的父类的初始化器
  return nil;
//成功的情况下，添加一些代码...
return self;
```

由于代码冗余并不是一个好的实践，在每个可能的初始化器中重复以上代码看起来并不省力。最佳解决方案是将最必须的代码提出来。然后，其他初始化器调用这个首选的初始化器，也称之为缺省初始化器。逻辑上来说，缺省初始化器应该是拥有最多参数的那个初始化器，因为 Objective-C 中是不可一个给予参数缺省值的。

```
-(id) initWithX:(int)x
{
  return [self initWithX:x andY:0 andZ:0];
}
-(id) initWithX:(int)x andY:(int)y
{
  return [self initWithX:x andY:y andZ:0];
}
//缺省初始化器
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
  if (!(self = [super init]))
    return nil;
  self->x = x;
  self->y = y;
  self->z = z;
  return self;
}
```

如果初始化器不是拥有最多参数的那个，它的作用就不大。

```
//the following code is not handy
-(id) initWithX:(int)x //designated initializer
{
  if (!(self = [super init]))
    return nil;
  self->x = x;

   return self;
```

```
}

-(id) initWithX:(int)x andY:(int)y
{
 if (![self initWithX:x])
    return nil;
  self->y = y;

  return self;
}
-(id) initWithX:(int)x andY:(int)y andZ:(int)z
{
 if (![self initWithX:x])
    return nil;
  self->y = y;
  self->z = z;
  return self;
}
```

5.1.8 初始化列表和实例数据的缺省值

C++ 中构造器的初始化列表的概念在 Objective-C 中并不存在。可是，值得提出的是，并不象 C++ 的行为，Objective-C 的 alloc 方法将实例数据的所有位都初始化为 0，所以指针指向 nil。对于对象类型的实例数据这不是问题，因为它们实际上是指针：在 Objective-C 中，对象都是动态分配的，都以指针形式存在 (而 C++ 中，对象类型的实例数据可以不是指针)。

5.1.9 虚构造器

Objective-C 是可能获得虚构造器的。查看 6.4.6 节 获取跟多细节。

5.1.10 类构造器

在 Objective-C 中，因为类本身也是对象，它们提供可被重定义的构造器。这明显是一个继承自 NSObject 的类方法；它的原型是 +(void)initialize;
首次使用这个类或者其某个子类时，这个方法会被自动调用。可是，并不是说，对于给定类该方法只被调用了一次；实际上，如果子类没有重写 +(void)inittialize，Objective-C 的机制将保证从它调用父类的 +(void)initialize。

5.2 析构器

在 C++ 中，析构器如同构造器一样是一个可被重写的特殊的函数。在 Objective-C 中，析构器是一个名为 dealloc 的实例方法。
在 C++ 中，析构器在对象被释放时被自动调用；这和 Objective-C 一样；只是释放对象的方法各有不同 (见 第6章)。
析构器不应该被显式地调用。实际上， C++ 中，唯一应显式调用析构器的例子是：开发者自己管理用作申请分配用的内存池。但在 Objective-C 中，没有任何应显式调用 dealloc 的例子。在 Cocoa 中，我们可以使用自定制内存区域，但这不影响通常的 分配/释放 (见 5.3 节)。

C++

```
class Point2D
{
  public:
    ~Point2D();
}

Point2D::~Point2D() {}
```

Objective-C

```
@interface Point2D : NSObject
-(void) dealloc; //这个方法可被重写
@end
@implementation Point2D
//在这个例子中，析构方法其实无需重写
-(void) dealloc
{
 [super dealloc]; //不要忘记调用父类的析构方法
}
@end
```

5.3 拷贝操作

5.3.1 经典克隆，copy, copyWithZone:, NSCopyObject()

在 C++ 中，正确一致地实现拷贝函数和拷贝运算符（等于号 = ）是很重要的。在 Objective-C 中，运算符重载是不可能；我们只能保证克隆方法是正确。
在 Cocoa 中，克隆要求实现名为 NSCoping 的协议的方法

```
- (id) copyWithZone:(NSZone*)zone;
```

它的参数是一个为克隆而分配的内存区域。Cocoa 允许使用不同的自定义区域：一些方法以这样的区域作为参数。大多数时候，使用缺省区域就可以了，并没有必要每次都特别指定。幸运地是， NSObject 提供了这样的方法

```
- (id) copy;
```

它封装了以一个缺省区域作为参数的 `copyWithZone:` 方法。但是，copyWithZone: 是 NSCoping 协议的一个方法。最后，工具方法 `NSCopyObject(...)` 提供了一个略有不同的方法，它更简单但是需要谨慎使用。下面代码没有考虑 `NSCopyObject(...)` (5.3.2 节将给出解释)。

```
//如果父类没有实现 copyWithZone: ，并且 NSCopyObject()未被使用
-(id) copyWithZone:(NSZone*)zone
{
  //一个新对象被创建
  Foo* clone = [[Foo allocWithZone:zone] init];
  //实例数据必须被手工复制
  clone->integer = self->integer; //"integer" 是整数类型
  //以相同的机制触发对象克隆
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //某些子对象可能无需克隆而需共享
  clone->objectToShare = [self->objectToShare retain]; //见内存管理
  //如果有写方法，以上两种情况都可以使用
  [clone setObject:self->object];
  return clone;
}
```

注意 allocwithZone: 替代了 alloc，以处理 zone 参数。alloc 封装了以缺省区域为参数对 allocWithZone: 的调用。若需了解更多关于 Cocoa 的自定义内存区域的管理，请查询 Cocoa 文档。

我们必须注意父类可能已经实现了 copyWithZone: 。

```
//如果父类实现了 copyWithZone:，而且 NSCopyObject() 没有被使用
-(id) copyWithZone:(NSZone*)zone
{
  // 创建新对象
  Foo* clone = [super copyWithZone:zone];
  //你必须克隆一些特殊实际数据。
  clone->integer = self->integer; //"integer" 是整数类型
  //以相同的机制触发对象克隆
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //某些子对象可能无需克隆而需共享
  clone->objectToShare = [self->objectToShare retain]; //见内存管理
  //如果有写方法，以上两种情况都可以使用
  [clone setObject:self->object];
  return clone;
}
```

5.3.2 NSCopyObject()

NSObject 类没有实现 NSCopying 协议，这也是为什么其子类无法直接调用 [super copy...]，必须使用形如 [[... alloc] init] 标准的初始化器 。

工具方法 NSCopyObject() 可以被用于编写更简单的代码，但是要谨慎对待指针成员 (包括对象)。这个函数创建一个对象的二进制拷贝，它的原型是：

```
//extra bytes 通常是 0，索引实例数据的额外空间
id <NSObject> NSCopyObject(id <NSObject> anObject,
                           unsigned int extraBytes, NSZone *zone)
```

二进制拷贝可以自治地复制那些不是指针地实例数据；但是对于指针成员 (包括对象)，必须考虑它会创建一个指向数据的指针的额外引用。最普遍地做法是在克隆之后，重置指针。

```
//如果父类没有实现 copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Foo* clone = NSCopyObject(self, 0, zone);//数据的二进制拷贝
  //clone->integer = self->integer; //无用：二进制数据拷贝已经拷贝了实例数据对象应该
                                    //实现的拷贝
  clone->objectToClone = [self->objectToClone copyWithZone:zone];
  //共享子对象必须注册新的引用
  [clone->objectToShare retain]; //见内存管理部分
  //因为指针值的二进制拷贝，读写器类似于释放 clone->object 这是不必要的，
  //因此，使用读写器之前，指针应该被重置
  clone->object = nil;
  [clone setObject:self->object];
  return clone;
}
```

```
//如果父类实现了 copyWithZone:
-(id) copyWithZone:(NSZone*)zone
{
  Foo* clone = [super copyWithZone:zone];
  //父类是否实现了 NSCopyObject() ? 这对于剩下要做什么很重要
  clone->integer = self->integer;//只在 NSCopyObject() 没有被使用的情况下

  //如有疑问，一个代克隆的子对象必须真正地被克隆
  clone->objectToClone = [self->objectToClone copyWithZone:zone];

  //无论是否使用 NSCopyObject() , retain 都必须使用 (见 内存管理)
  clone->objectToShare = [self->objectToShare retain];
  clone->object = nil; //如有疑问，最好重置
  [clone setObject:self->object];
  return clone;
}
```

5.3.3 伪克隆，可变性，mutableCopy 和 mutableCopyWithZone:

当克隆对象是不可变的，那么可以作出一个根本性优化：我们可以假装它已经被克隆了；无需真正地复制它，只要返回一个引用就可以了。重这里，我们可以区别可变对象和非可变对象。

一个非可变对象，其任何实例数据都不能有改变；只有初始化器可以给予其一个合理的状态。在这个例子中，只返回被克隆对象的引用这样的伪克隆是安全的。因为无论是对象本身还是其克隆都不能被改变。一个有高效的 copyWithZone: 的实现如下：

```
-(id) copyWithZone:(NSZone*)zone
{
 //对象返回自己，引用计数器增加
 return [self retain]; //查看内存管理
}
```

retian 的使用可查看 Objective-C 的内存管理 (见 6 章)。克隆的存在使得引用计数增加1，即使删除克隆也不会销毁原对象。

“伪克隆”不是可有可无的优化。创建对象要申请分配内存，这是一个很长的过程，若是可能应该尽量避免。这也是为什么要区分两种类型的对象：不可变对象，克隆可以简化；可变对象，则不能。区别可变对象和不可变对象，可以先创建不可变对象的类，再继承这个类并选择性添加可变性，比如添加方法改变它们的实例数据。例如，再添加改变实例数据的方法。例如，在 Cocoa 中，NSMutableString 是 NSString 的子类，NSMutableArray 是 NSArray 的子类，NSMutableData 是 NSData 的子类，等等。

然而，根据这里呈现的技术，看起来不可能获得一个真正的克隆：通过“伪克隆”，从一个不可变对象那安全地获到一个可变对象。这样的限制将大大减弱不可变对象的可用性。

除了 NSCopying 协议，这里有另外一个名为 NSMutableCopying 的协议 (见 4.4 节)，要求实现

```
- (id) mutableCopyWithZone:(NSZone *)zone;
```

mutableCopyWithZone: 必须返回一个可变克隆，克隆的改变不会影响到原对象。类似于 copy 方法，mutableCopy 自动以缺省内存区为参数调用 **mutableCopyWithZone:** 方法。**mutableCopyWithZone:** 的实现看起来如下，和之前展示的典型 copy 方法类似：

```
//如果父类没有实现 mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Foo* clone = [[Foo allocWithZone:zone] init];//或，如可能使用 NSCopyObject()
  clone->integer = self->integer;
  //如同 copyWithZone:, 某些实例对象会被克隆，某些会被共享
  //一个可变实例对象数据可以调用 mutableCopyWithZone: 完成复制

  //...
  return clone;
```

```
}
```

不要忘记父类可能已经实现了 mutableCopyWithZone: 。

```
//如果父类实现了 mutableCopyWithZone:
-(id) mutableCopyWithZone:(NSZone*)zone
{
  Foo* clone = [super mutableCopyWithZone:zone];
  //...
  return clone;
}
```

6 内存管理

6.1 new 和 delete

Objective-C 中没有 C++ 的关键字 new 和 delete (new 作为方法存在，是 alloc + init ,但被弃用了)。它们分别被调用 alloc (见 5.1) 和 release (见 6.2) 替代。

6.2 引用计数器

Objective-C 的内存管理是语言中最重要的部分之一。在 C 或 C++ 中，一片内存区域申请一次就要释放一次。它可以被任意多的指针引用，但只有一个指针能被使用 delete 调用。

另外，Objective-C 实现了一个引用计数表格。一个对象知道有多少个引用指向它。这可以用狗和狗链的比喻来解释 (*Cocoa Programming for MacOS X[7]* 中的比喻) 。如果一个对象是狗，每个人都要求用狗链套住它。如果不再关注狗，就松开狗链。当狗至少有一条狗链套住它时，它就必须待在那儿。但是一旦狗链的数字降到 0，这条狗就自由了！

更技术一点，一个新创建的对象的引用计数为 1。如果一部分代码需要引用这个对象，它可以向对象发送 retain 消息，这将使计数器加一。当一部分代码不再需要这个对象时，它可以向对象发送 release 消息，这将使计数器减一。

只要计数器的值为正值，对象就能够按需要收到 retain 和 release 消息。一旦计数值降到 0，析构器 dealloc 就自动被调用。之后再向对象的地址发送 release 消息将是不合法的，会引起内存错误。

这项技术不是 C++ STL 的 auto_ptr 的等价物。另一方面，Boost 库[5] 提供一个指针的封装类 shared_ptr 类，它实现了引用计数器表。但它不是标准库的一部分。

6.3 alloc, copy, mutableCopy, retain, release

理解了内存管理的原理并不意味着就知道如何使用它。这一节的目标是给出一些规则。关键字 autorelease 先放在一边，因为它比较难于理解。

基本规则是：**任何东西被调用 alloc，[mutable]copy[WithZone:] 或 retain 时自增引用计数值，同时需要调用同样多的 [auto]release。**实际上，有三种方法使引用计数器增加。这也意味着你必须注意释放对象的有限几种情况：

- 当你显式地用 alloc 实例化一个对象时；
- 当你显式地用 copy[WithZone:] 或者 mutableCopy[WithZone:] 拷贝对象时 (无论是否为伪拷贝。见 5.3.3)；
- 当你显式地使用 retain 时。

请记住默认情况下，向 nil 发送消息 (如 release) 是合法，不会引起任何后果 (见 3.4.1)。

6.4 autorelease

6.4.1 珍贵的 autorelease

前一节描述的规则很重要，它值得重复一次：**任何东西被调用 alloc，[mutable] copy[WithZone:] 或 retain 时自增引用计数值，同时需要调用同样多的 [auto]release。**

实际上，仅仅只有 alloc, retain 和 release，这个规则是无效的。的确存在一些方法不是构造器，但它却被用于创建对象：例如，C++ 中的二进制加法 (obj3 operator+(obj1, obj2))。在 C++ 中，返回的对象将被在栈 (stack) 上分配空间，离开作用域时被自动销毁。但在 Objective-C 中，这样的对象不存在。方法必须使用 alloc，但是不能在返回它之前将其在栈上释放！这里给出了一些错误示例：

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  return result;
}
//ERROR : the function performs "alloc", so, it is creating
//an object with a reference counter of 1. According
//to the rule, it should destroy the object.
//This can lead to a memory leak when summing three points :
[calculator add:[calculator add:p1 and:p2] and:p3];

//The result of the first addition is anonymous
//and nobody can release it. It is a memory leak.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
   return [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                andY:([p1 getY] + [p2 getY])];
}
//ERROR : This is exactly the same code as above. The fact that
//no intermediate variable is used does not change anything.
```

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
```

```
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result release];
  return result;
}
//ERROR : obviously, it is nonsense to destroy the object after creating it
```

问题似乎很棘手。若是没有 autorelease 的话，确实如此。为了简便起见，我们可以视向对象发送 autorelease 消息为一个会“稍晚”起效的 releasse 消息。但，“稍晚”并不意味着“任何时候”。这在 6.4.2 节详细描述了。这里有一个唯一的解决方法：

```
-(Point2D*) add:(Point2D*)p1 and:(Point2D*)p2
{
  Point2D* result = [[Point2D alloc] initWithX:([p1 getX] + [p2 getX])
                                          andY:([p1 getY] + [p2 getY])];
  [result autorelease];
  return result;        //a shorter writing is "return [result autorelease]"
}
//CORRECT : "result" will be automatically released later,
//after being used in the calling code
```

6.4.2 autorelease 池

在前一节，autorelease 被作为一种魔幻类型的 release 做了介绍，它能在正确的时间自动启用。但是，让编译器猜测什么是正确的时间没有任何意义。这种情况下，垃圾收集会更有用。为了解释它如何工作，更多的关于 autorelease 的细节将被给出。

对象每次收到 autorelease 消息，只是在一个 “autorelease 池”注册。当池被销毁了，对象将收到一个真正的 release 消息。那么问题就变为：如何管理池呢？

这没有单一的答案：如果你使用 Cocoa 制作一个有图形界面的应用程序，大部分时间你什么都不用做了。否则，你需要自己创建和销毁池。

一个有图形界面的应用普遍会使用事件循环。这样的循环等待用户事件，然后唤醒程序响应事件，接着继续休眠直到下一次事件。当你用 Cocoa 创建了一个带图形界面的应用的时候，一个 autorelease 池在事件循环的开始被自动创建了，在事件循环结尾自动销毁了。这是符合逻辑的：一般而言，一个用户动作触发一系列任务。暂时对象，因其无需被保留到下一个事件，被创建，之后又然后被销毁。如果其中一些需要保留更长一点，开发必须按需要使用 retain。

另外一方面，当没有图形界面时，你必须在代码需要的地方创建 autorelease 池。当一个对象接收到 autorelease 消息时，他知道如何找到最近的 autorelease 池。然后到了需要清空池的时候，你只许用 release 销毁池即可。一个典型的 Cocoa 命令行程序包含如下代码：

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

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

请注意，MacOS X10.5 在 NSAutoreleasePool 类添加了 drain 方法。如果垃圾收集启用了，这个方法等效于 release，否则，就会触发垃圾回收启动 (见 6.6 节) 。这对于编写可运行于两种环境 (有无垃圾收集) 的代码很有用。

6.4.3 使用多个 autorelease 池

在代码中使用多个 autorelesae 池是可能的，有时还很有用。一个收到 autorelease 消息的对象会在最近的池注册。因此，如果函数创建和使用了大量临时对象，可以通过创建本地 autorelease 池提高性能。这样，大量临时对象将被尽快销毁，在函数返回后也不会占用内存。

6.4.4 关于 autorelease 的警告

autorelease 很方便，但并不要误用。

- 首先，发送多余必要的 autorelease 就如同发送过多的 release，这将在清空池的时候导致内存错误；
- 然后，即使任何 release 消息都可以被 autorelease 代替，这也会影响性能，因为 autorelease 池比普通的 release 而言，需要做更多工作。而且，推迟所有的内存释放将引起无用无关的内存消耗高峰。

6.4.5 autorelease 和 retain

由于 autorelease，方法可以创建能自动释放的对象。可是，长期持有一个对象是很普遍的需求。这种情况下，我们发送 retain ，并且计划 release。然后，关于对象的生命周期可从两个角度去看：

- 从函数的开发者的角度来看，对象被创建了，它的释放也安排好了会稍后进行；
- 从函数的调用者的角度来看，对象的生命周期通过 retain 延长了 (函数中使用的 autorelease 不会将引用计数器减到 0)，但是由于有计数器的加一，所以调用者要负责稍后的释放。

6.4.6 便捷构造器，虚构造器

alloc 和 init 的连续使用在某些情况下，看起来有些累人。幸运的是，可以使用便捷构造器。便捷构造器的名字由类的名字作为前缀，行为如同一个 init 方法，但是它自己会执行 alloc 方法。可是，返回对象被会注册在 autorelease 池，若是不被发送 retain 消息，它将成为暂时对象。例如：

```
//累人的
NSNumber* zero_a = [[NSNumber alloc] initWithFloat:0.0f];
```

```
...
[zero_a release];

...
//简便的
NSNumber* zero_b = [NSNumber numberWithFloat:0.0f];
...
//no need of release
```

参照内存管理的章节 (见 6 章)，这样的构造器和 autorelease 有关系。由于需要正确使用 self，相关代码并不是显而易见的。便捷构造器的确是一个*类方法*，所以 self 在类方法中，是一个元类对象，指向类的对象。而在普通构造器中，由于其是一个实例方法，self 是类的实例，指向一个普通的对象。

很容易写出不好的便捷构造器。让我们假设一个类 Vehicle，它拥有颜色这个属性，并提供了一个便捷构造器。

```
//The Vehicle class
@interface Vehicle : NSObject
{
   NSColor* color;
}
-(void) setColor:(NSColor*)color;
//convenience constructor
+(id) vehicleWithColor:(NSColor*)color;
@end
```

便捷构造器的实现有些困难。

```
// bad convenience constructor
+ (Vehicle *)vehicleWithColor:(NSColor *)color
{
  // 这里 "self" 的值不应该改变
  self = [[self alloc] init]; // 错误！
  [self setColor:color];
  return [self autorelease];
}
```

在类方法中的 self 指向类对象。 它不应该被当作类的实例使用。

```
// 几乎完美的构造器
+ (id) vehicleWithColor:(NSColor *)color
{
  id newInstance = [[Vehicle alloc] init]; // 正确，但忽视了可能有的子类调用。
  [newInstance setColor:color];
```

```
  return [newInstance autorelease];
}
```

我们仍然可以改进这个构造器。在 Objective-C 中，可以表达出虚构造器的行为。构造器仅需要自我检查执行本方法的对象的真正的类型。然后，它便可以直接创建正确子类型的对象。伪关键字 class 可以被使用；这个 NSObject 方法返回当前对象的类对象 (元类对象)。

```
@implementation Vehicle
+(id) vehicleWithColor:(NSColor *)color
{
  id newInstance = [[[self class] alloc] init];
  [newInstance setColor:color];
  return [newInstance autorelease];
}
@end

@interface Car : Vehicle {...}
@end

…

// 制造了 (red) car
id car = [Car vehicleWithColor:[NSColor redColor]];
```

类似于构造器以 init 开头的规则，强烈建议你使用类名作为前缀。只有很少的例子不符合这个规则，例如前面代码中的 [NSColor redColor]，其实最好应该写为 [NSColor colorRed]。
最终，让我们重复一下规则：**任何东西被调用 alloc，[mutable]copy[WithZone:] 或 retain 时自增引用计数值，同时需要调用同样多的 [auto]release。**当调用一个便捷构造器时，你并没有显式地调用 alloc，所以你没有责任负责 release。然而，当创建这样一个构造器时，你使用了 alloc，所以不能忘记使用 autorelease。

6.4.7 赋值器 (Setter)

赋值器是一个典型的例子，如果不知道 Objective-C 内存管理的知识就能难写好赋值器。让我们假设有一个封装了名为 title 的 NSString 类型数据的类，假设我们要改变这个字符串的值。这会是一个能引出赋值器主要问题的简单例子：参数如何被使用。不象 C++，Objective-C 中只有一个原型是合法的 (对象只能通过指针被使用)，但却能有若干种实现：它可以是简单赋值 (assign)，retain 赋值，或者复制 (copy)，其中每一个都有关于数据模型的特别含义。并且，在每个例子中，旧资源应该首先被释放以避免内存泄漏。

赋值 (不完全代码)

外部对象作为参数传递进如，仅被弱引用，不使用 retain 关键字。如果外部对象被修改了，对当前类来说，修改是可见的。如果外部对象恰好被释放了，当前引用又没有被置为 nil，它将成为一个非法的引用。

```
-(void) setString:(NSString*)newString
{
  ... memory management to be detailed later
  self->string = newString; //assignation
}
```

retain 赋值 (不完全代码)

外部对象被引用，同时由于使用了 retain 关键字引用计数器加一。若果外部对象被改变了，对当前类来说，修改是可见的。当前引用被释放了，外部对象不会被释放。

```
-(void) setString:(NSString*)newString
{
  ... memory management to be detailed later
  self-> string = [newString retain]; //assignation with retain
}
```

复制 (不完整代码)

外部对象没有被引用：一个克隆被创建了。如果一个外部对象被修改了，对于克隆是不可见的。逻辑上，克隆是由当前对象自己管理，克隆的生命周期不会比对象自己长。

```
-(void) setString:(NSString*)newString
{
 ... memory management to be detailed later
 self->string = [newString copy];//cloning;
                                  //the NSCopying protocol is used
}
```

为了补全代码，对象之前的状态应该被考虑：在每种情况下，赋值器在置新值之前，需要释放旧引用。这部分代码是微妙的。

简单赋值 (完全代码)

最简单的例子。旧引用被复写。

```
-(void) setString:(NSString*)newString
{
 //no strong link : the old reference can be overwritten
  self->string = newString; //assignation
}
```

retain 赋值 (完全代码)

在这种情况下，除非新引用和旧引用是同一个，否则旧引用应该被释放。

```
//Bad codes
                                                                    -
```

```
(void) setString:(NSString*)newString
{
 self->string = [newString retain];
 //ERROR : memoy leak : the old "string" is no more referenced
}

-(void) setString:(NSString*)newString
{
 [self->string release];
  self->string = [newString retain];
  //ERROR : if newString == string,  (it can happen)
  //and that the reference counter of newString was 1,
  //the it is invalid to use newString (string) after
  //[self->string release], because it has been dealocated at this point
}

-(void) setString:(NSString*)newString
{
 if (self->string != newString)
   [self->string release]; //ok: it is safe to send release even to nil
   self->string = [newString retain];  //ERROR : should be in the "if";
  //because if string == newString,
  //the counter should not be incremented
}
```

```
//Correct codes
//Practice "Check before change"
//the most intuitive for a C++ developer
-(void) setString:(NSString*)newString
{
 //avoid degenerated case where there is nothing to do
  if (self->string != newString)
  {
   [self->string release]; //release the old one
   self->string = [newString retain]; //retain the new one
  }
}

//Practice "Autorelease the old value"
-(void) setString:(NSString*)newString
{
 [self->string autorelease]; //even if string == newString,
                             //it's correct, because release is delayed

 self->string = [newString retain]; //...and thus this retain happens before
}

//Practice "retain, then release"
-(void) setString:(NSString*)newString
```

```
{
 [self->newString retain]; //the reference counter is increased by 1(except
on
  [self->string release]; //...so that is does not reach 0 here
  self->string = newString; //but no "retain" is added here !
}
```

复制 (完全代码)

关于复制的典型错误或者好的解决方案，和 retain 复制一样，只是 retian 关键字需被 copy 替换。

伪克隆

请注意复制可以是“伪克隆” (见 5.3.3)，不过对结果没有影响。

6.4.8 取值器 (Getter)

Objective-C 中，所有对象都被动态分配。他们被引用并封装为指针。通常，取值器只返回指针值，而不应复制对象。取值器的名字通常和数据成员的名字一样，这在 Objective-C 中是可以的，并不会产生冲突。在布尔值的情况下，名字可能以 is 开始，以使其读起来如同一个断言。

```
@interface Button
{
   NSString* label;
   BOOL      pressed;
}
-(NSString*) label;
-(void) setLabel:(NSString*)newLabel;
-(BOOL) isPressed;
@end


@implementation Button
-(NSString*) label
{
  return label;
}
-(BOOL) isPressed
{
  return pressed;
}

-(void) setLabel:(NSString*)newLabel {...}
@end
```

当返回实例数据的指针时，如果它们是可变的，修改它们就很容易了。不鼓励在对象外部修改对象，因为这样破坏了数据封装。

```
@interface Button
```

```
{
 NSMutableString* label;
}

-(NSString*) label;
@end
@implementation Button
-(NSString*) label
{
 return label; //OK, but a well-informed user could downcast
 //the result to NSMutableString, and thus modify the string
}

-(NSString*) label
{
 //solution 1 :
  return [NSString stringWithString:label];
  //OK : a new, immutable string, is returned

 //solution 2 :
  return [[label copy] autorelease];
  //OK, using copy (and not mutableCopy) on an NSMutableString will return
  //an NSString
}
@end
```

6.5 Retain 循环

retian 循环是应该被避免地。如果对象 A retain 了对象 B，B 和 C 相互 retain 了对方，B 和 C 就形成了一个 retain 循环。

A -> B <- -> C

如果 A release B, B将不会被释放，因为它还被 C retain 住着。C 不能被释放，因为它还被 B retain 住着。可是，A 是这个循环的唯一引用，所以，这个循环成了一个不可达的循环。Retain 循环常常导致内存泄漏。这就是为什么，在树结构中，一个节点通常会 retain 其子节点，但子节点通常不会 retian 它的父节点。

6.6 垃圾收集

Objective-C 2.0 (见 1.2) 实现了垃圾收集。换句话说，你可以将所有内存管理的工作交给垃圾收集而不再关注 retain 和 release。Objective-C 的一个很棒的选择在于使垃圾收集成为一个可选功能：你可以决定是要精确地控制对象的生命周期，或者想要 bug 更少的代码。垃圾收集需以整个代码单位，或整个开启或整个禁用。

如果垃圾收集开启了，retain, release 和 autorelease 将被重定义为什么都不做。因此，为无垃圾收集而编写的代码在理论上可以很容易重新编译为有垃圾收集的版本。“理论上”意味着还有很多

涉及资源释放的微妙的问题需要处理。所以，为两种情况编写统一的代码并不简单，很多开发者都会重新开发一遍。这些细微的问题在 Apple 的文档 [2] 中有详细叙述。下面 4 节描述了其中的一些难点，以说明这些都是需要认真研究的。

6.6.1 finalize

在垃圾收集环境下，对象的析构顺序是不确定的，它并不遵循使用 dealloc 的顺序。一个 finalize 方法被加到 NSObject 中以将析构过程切分成两个步骤：资源释放和实际的释放。但是编写一个好的 finalize 方法是很精妙的。需要考虑很多限制。

6.6.2 weak, strong

__weak 和 __strong 被显式地使用在声明中并不常见。可以，了解它们有助于理解关于垃圾收集的一些新的难题。

一个对象指针缺省是使用 __strong 参数的：这是一个强引用。这意味着只要引用存在对象就不能被销毁。这样的行为是可以预见的：当所有(强)引用消失了，对象才能被回收和释放。在一些例子中，禁用这种行为是很有用的：垃圾回收不应该增加它所持有的对象的生命周期，因为这会阻止对象被销毁。在这种情况下，这些回收使用的是通过 __weak 关键字实现的弱引用。NSHashtable 是一个例子 (见 11.1)。当被引用对象消失后， _weak 引用是会自动结束其生命周期的。

一个很相关的例子是 Cocoa 的 Notification Center，这已经超出了纯 Objective-C 的范畴了，这部分将不会在本文档中详述。

6.6.3 NSMakeCollectable()

Cocoa 不是 MacOS X 的唯一 API。Core Foundation 是另外一个；它们是兼容的，可以共享数据和对象，但是 Core Foundation 纯 C 编写的 API。首先，垃圾收集无法和 Core Foundation 的指针一起工作。这个问题已经接近，可以在垃圾收集环境中使用 Core Foundation 而不引起内存泄漏。NSMakeCollectable 文档是理解这种技术的很好起点。

6.6.4 AutoZone

Apple 开发的 Objective-C 的垃圾收集器被命名为 AutoZone。它被开源了[1]。MacOS X10.6 中又有了一些演进。

7 异常

Objective-C 的异常处理较 C++ 更接近于 Java，主要缘于 @finally 关键字。finally 存在于 Java，C++ 则没有。它是 try() … catch() 块的附加 (可选的) 部分，其中的代码总是会执行的，无论是否有异常被捕捉到了。这在编写短小干净的释放资源代码时很有用。除此之外，Objective-C 中 @try … @catch … @finally 的行为是非常经典的；但，只有对象可以被抛出 (不象 C++)。一个简单有 @finally 和没有 @finally 的例子如下.

Without finally	With finally

BOOL problem = YES;
@try{
  dangerousAction();
  problem = NO;
}
@catch (MyException* e){
  doSomething();
  cleanup();
}
@catch (NSException* e){
  doSomethingElse();
  cleanup();
  //here is the exception re-thrown
  @throw
}
if (!problem)
  cleanup();
	try{
	  dangerousAction();
	}
	@catch (MyException* e){
	  doSomething();
	}
	@catch (NSException* e){
	  doSomethingElse();
	  //here is the exception re-thrown
	  @throw
	}
	@finally{
	  cleanup();
	}

严格地说，@finally 并不是必须的，但它在处理异常时确实是一个有用的工具。如前面例子所示，它也能很好处理在 @catch 块 重新抛出异常的情况。实际上，@finally 在离开 @try 块时被触发。下面给出一个例证。

```
int f(void)
{
  printf("f: 1-you see me\n");
  //See the section about strings to understand the "@" syntax
  @throw [NSException exceptionWithName:@"panic"
                                 reason:@"you don't really want to known"
                               userInfo:nil];
  printf("f: 2-you never see me\n");
}

int g(void)
{
  printf("g: 1-you see me\n");
  @try {
    f();
    printf("g: 2-you do not see me (in this example)\n");
  }
  @catch(NSException* e) {
    printf("g: 3-you see me\n");
    @throw;
    printf("g: 4-you never see me\n");
```

```
  }
  @finally {
    printf("g: 5-you see me\n");
  }
  printf("g: 6-you do not see me (in this example)\n");
}
```

最后一点：C++ 中的 catch (和断点)，可以捕捉任意东西，这在 Objective-C 不可能。实际上，因为只有对象可以被抛出，所以总是能通过 id 类型 (见 3.1) 捕捉它们。
请注意 NSException 类存在于 Cocoa 中，鼓励用它作为所有抛出对象的父类。所以，catch(NSException *e) 同等于 catch(...) 。

8 多线程

8.1 线程安全

在 Objective-C 中，使用 POSIX APIs[2] 实现多线程的程序是非常易读的。Cocoa 提供了它自己的类来管理竞争线程。两种方法都需注意：代码不同区域起的多个线程同时访问相同的内存区域会引起不可预料的结果。
POSIX APIs，和 Cocoa 一样，实现了锁和互斥对象。Objective-C 提供了关键字 @synchronized，同等于 Java 的同名关键字。

8.2 @synchronized

一块被 @synchronized(...) 包围的代码段会被自动上锁，同一时间只有一个线程可访问执行这段代码。这并不总是最好的解决竞争访问的方法，但对大多数关键块来说，它却是简洁的方法。
@synchronized 要求一个对象作为参数 (任何对象，比如 self) 作为锁。

```
@implementation MyClass
-(void) criticalMethod:(id) anObject
{
  @synchronized(self)  {
   //this part of the code is exclusive to any block @synchronized(self)
   //(with the same "self"...)
  }

 @synchronized(anObject)
  {
  //this part of the code is exclusive to any block @synchronized(anObject)
  //(with the same "anObject"...)
  }
}
```

```
@end
```

9 Objective-C 中的字符串

9.1 Objective-C 中唯一的静态对象

C 语言中，字符串是字符数组，或者 char* 的指针。处理这种数据类型很困难且很容易出错。C++ 的 string 类则好多了。在 Objective-C 中，第5章已经介绍，所有对象不是自动分配的，而是需要在运行时申请。唯一不同的是静态字符串的使用。这使得我们可以使用 C 的静态字符串作为构造参数来创建 NSString 对象；但这并不方便，而且浪费内存。

幸运的是，Objective-C 中也存在静态字符串，引号标记的 C 语言字符串之前加 @ 前缀，就构成了 Objective-C 字符串。

```
NSString* notHandy = [[NSString alloc] initWithUTF8String:"helloWorld"];
NSString* stillNotHandy = //initWithFormat is a kind of sprintf()
[[NSString alloc] initWithFormat:@"%s", "helloWorld"];
NSString* handy = @"hello world";
```

并且，静态字符串可如同常规对象一样称为消息发送对象。

```
int size = [@"hello" length];
NSString* uppercaseHello = [@"hello" uppercaseString];
```

9.2 NSString 和 编码

NSString 对象很有用，因为除了有大量的方便的方法之外，它也支持不同的编码 (ASCII, Unicode, ISO Latin 1...)。应用的翻译和本地化使用这类字符串是很容易的。

9.3 对象描述，%@ 格式扩展，NSString 到 C 字符串

在 Java 中，每个对象都继承自 Object，而且都有可用于将对象描述为字符串的 **toString** 方法，这在调试时很有用。在 Objective-C 中，这个方法叫做 **description**，它返回一个 NSString 对象。

NSString 对象不支持 C 语言的 printf 方法。NSLog 可用于替代它。有很多其他方法接受格式化字符串参数。对于 NSString 对象，格式化符为不是 “%s” 而是 “%@”。更为普遍的，“%@” 可被用于任何对象，因为实际上会调用 - (NSString *)description 而返回 NSString 值。

而且，一个 NSString 对象可以通过 UTF8String 方法 (之前为 cString) 被转化为 C 语言字符串。

```
char* name = "Spot";
NSString* action1 = @"running";
printf("My name is %s, I like %s, and %s...\n",
        name, [action1 UTF8String], [@"running again" UTF8String]);
NSLog(@"My name is %s, I like %@ and %@\n",
```

```
        name, action1, @"running again");
```

10 C++ 特有特性

目前为止，你已经看到了 C++ 的面向对象概念是如何在 Objective-C 中表达的。然而，其他一些 C++ 功能并未涉及。它并不涉及面向对象概念，而是一些关于代码如何编写的问题。

10.1 引用

Objective-C 中没有引用 (&)。内存管理使用引用计数器和 `autorelease` 使引用失去了作用。因为对象总是动态创建的，它们仅以指针形式被引用。

10.2 内联

Objective-C 没有实现内联。对方法而言，这是合理的，由于 Objective-C 的动态性，它很难“冻结”某些代码。尽管如此，内联对于一些 C 语言中如 `max()`, `min()` 等工具方法是很有用的。这一问题在 Obejctive-C++ 中会得到解决。

然而，注意 GCC 编译器提供非标准的关键字 __inline 或者 __inline__ 用于 C 中的内联，同时也能使用在 Objective-C 中。并且，GCC 也可以编译 C99 代码 (一个用关键字 inline 支持内联的 C 语言版本)。因此，基于 C99 的 Objective-C 也能使用内联。
若不是为了内联而内联，而是由于性能问题，你可以考虑使用 IMP 缓存 (见 11.3.2)。

10.3 模板

模板是继承和虚函数的一种替代机制，它被设计得更为有效率，但明显远离了纯面向对象原则。(你知道巧妙地使用模板可以公开地访问私有成员吗？)。Objective-C 没有实现模板，由于方法名重载机制和选择器 (selector) 模板实现起来很困难。

10.4 操作符重载

Objective-C 不能实现操作符重载。

10.5 友元

Objective-C 没有友元得概念。实际上，C++ 中，它对于提高操作符重载的效率异常有用，而 Objective-C 中没有操作符重载这个概念。Java 中包的概念某种上近似于友元，在 Objective-C 中，这大部分时候可以使用目录 (Category) (见 4.5) 解决。

10.6 const 方法

Objetive-C 中方法不能被声明为 **const**。因此，关键词 **mutable** 也不存在。

10.7 构造器的初始化列表

Objective-C 中不存在构造器的初始化列表。

11 STL 和 Cocoa

标准 C++ 类库是其强大的方面之一。即使它有一些不足 (现在在函数对象这点上，不足可以由 SGI STL [8] 弥补)，但它仍然是很完备的。STL 不是 C++ 语言的一部分，因为它只是一个扩展。我们也容易想到在其他语言中寻找类似的扩展。在 Objective-C 中，你必须在 Cocoa 中寻找容器，迭代器和其他可用的算法。

11.1 容器

很明显，Cocoa 用面向对象的方式做了这件事：对象不是模板，它只包含对象。在写这份文档的时候，可用的容器有：

- NSArray 和 NSMutableArray ，有序集合；
- NSSet 和 NSMutableSet，无序集合；
- NSDictionary 和 NSMutableDictionary，键值关联对集合；
- NSHastable，使用弱引用的哈希表 (仅 Objetive-C 2.0 存在，见后)；

你可能注意到缺少了 NSList 和 NSQueue。实际上，这两个容器不过是由 NSArray 实现的。

不象 C++ 中的 vector<T> ，Objective-C 的 NSArray 真正隐藏了其内部实现而只通过访问器暴露了其内容。因此，NSArray 没有义务维护一系列物理上连续内存单元。NSArray 的实现做了一些妥协，使 NSArray 如同数组或列表那样有效地被使用。因为在 Objective-C 中容器只保存对象的指针，维护这些单元就变得很有效率。

NSHashTable 等价于 NSSet，但使用了弱引用 (见 6.6.2)，在垃圾回收环境下这是很有用的。

11.2 迭代器

11.2.1 典型枚举

Objective-C 的纯面向对象实现，使其较 C++ 更容易实现迭代器。NSEnumerator 就是如此设计的：

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSEnumerator* enumerator = [array objectEnumerator];
NSString* aString = @"foo";
id anObject = [enumerator nextObject];
while (anObject != nil)
{
```

```
   [anObject doSomethingWithString:aString];
   anObject = [enumerator nextObject];
}
```

容器的方法 objectEnumerator 返回一个能自移动 (nextObject) 的迭代器。较 C++ 这个行为更接近于 Java。当迭代器移到容器结尾处时，nextObject 返回 nil。

这有一个更为普遍的使用迭代器的形式，基于 C 的简洁风格。

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSEnumerator* enumerator = [array objectEnumerator];
NSString* aString = @"foo";
id anObject = nil;
while ((anObject = [enumerator nextObject])) {
  [anObject doSomethingWithString:aString];
}//double parenthesis suppress a gcc warning here
```

11.2.2 快速枚举

Objective-C 2.0 (见 1.2) 已经介绍遍历一个容器的新语法，这时 NSEnumerator 是隐式的 (其实，和 NSEnumerator 是一样)。语法形式：

```
NSArray* someContainer = ...;
for(id object in someContainer) { //here, each object is typed "id"
    ...
}
for(NSString* object in someContainer) {//here, each object is
typed "NSString*
  ...//if an object is not an NSString*, it's up to the developer to handle
that
}
```

11.3 函数对象

11.3.1 使用选择器

Objective-C 中的 *selector* 使函数对象并不那么有用了。实际上，弱类型允许用户在不关系接受者是否有能力处理消息的情况下发送消息。例如，这有一个等价于前面使用迭代器的例子：

```
NSArray* array = [NSArray arrayWithObjects:object1, object2, object3, nil];
NSString* aString = @"foo";
[array makeObjectsPerformSelector:@selector(doSomethingWithString:)
                       withObject:aString];
```

在这个例子中，对象并不必是同种类型，它们甚至无需实现 **doSomethingWithString:** 方法 (这将抛出一个 "选择器不被识别" 的异常)!

11.3.2 IMP 缓存

这里将不详细叙述，获取方法的内存地址是可能的。通过集中方法的查找，可以用于优化对于一堆对象的相同选择器 (selector) 的多次调用。这被称为 “IMP 缓存”，因为 Objective-C 中实现函数这一数据类型的就是 IMP。

调用 **`class_getMethodImplementation()`**(见 13.2) 可以被用于获得这样的 IMP 指针。请注意，虽然它是一个指向被实现的函数的真正的指针：不能有虚调用。它大多数时候被用于做优化相关的事情，使用时也应该谨慎。

11.4 算法

STL 中大量的通用算法在 Cocoa 中找不到对应。作为代替，你可以看看每个容器提供的方法。

12 隐式代码

这章汇集了两种减少代码量的功能。他们的目的各不相同：键直对编码 (Key-value ccoding) 可以通过找到首个合法的实现来解决间接方法调用的问题 (见 12.1) ，而属性(properties) 可以让编译器生成某些写起来很烦人的方法。

12.1 键值对编码 (Key-value coding)

12.1.1 原则

键值对编码是一种实用方法的称谓：通过数据成员的名字访问数据成员。这类似于使用关联数组 (NSDictionary， 见 11.1)，数据成员的名字相当于键。类 NSObject 有方法 valueForKey: 和 setValue:forKey:。如果数据成员本身是对象，便需更深地探寻，键会形成一条路径，由点分隔。相关方法为 valueForKeyPath: 和 setValue:forKeyPath:。

```
@interface A {
  NSString* foo;
}
... //some methods must be implemented for that code to be complete
@end

@interface B {
  NSString* bar;
  A* myA;
}
... //some methods must be implemented for that code to be complete
@end
```

```
@implementation B
...
//Let us assume an object a of type A and an object b of type B
B* a = ...;
B* b = ...;
NSString* s1 = [a valueForKey:@"foo"]; //ok
NSString* s2 = [b valueForKey:@"bar"]; //ok
NSString* s3 = [b valueForKey:@"myA"]; //ok
NSString* s4 = [b valueForKeyPath:@"myA.foo"]; //ok
NSString* s5 = [b valueForKey:@"myA.foo"]; //erreur !
NSString* s6 = [b valueForKeyPath:@"bar"]; //ok, why not
...
@end
```

由于这些语法，我们可能使用相同的代码管理有相同实例数据名的不同类的对象。

最好的使用用例将数据(其名字) 绑定到一些触发器(尤其是一些方法调用)上，例如键值对观察者模式 (Key-value Observing, KVO) ，这里就不详述了。

12.1.2 拦截

通过 valueForKey: 或者 setValue:forKey: 访问数据并不是原子操作。它遵守一套方法调用约定。实际上，访问只有在一些特定的方法被实现的情况下才可能完成(在使用属性 (properties) 的情况下会被隐式地生成 ，见 2.2)，或者在直接访问实例数据被显式允许的情况下。

苹果的文档精确描述了 valueForKey: 和 setValue:forKey: 的行为。
例如，调用 valueForKey:@”foo”

- 若方法 getFoo: 存在，则调用它；
- 否则，若方法 foo 存在，则调用它 (最常见的情况)；
- 否则，若方法 isFoo 存在，则调用它 (对于布尔数据很常见)；
- 否则，若方法 accssInstanceVariablesDirectly 返回 YES，则尝试读取成员数据(若其存在的话) _foo，若无尝试读 _isFoo，之后尝试 foo，接着尝试 isFoo；
- 在上面这些步骤成功的情况下，放回匹配的值；
- 若上面这些步骤都失败了，调用方法 valueForUndefineKey:；在 NSObject 缺省实现了这个方法，它抛出了一个异常。

例如，调用 setValue:..forKey:@”foo”

- 若方法 setFoo: 存在，则调用它；
- 否则，若方法 accssInstanceVariablesDirectly 返回 YES，则尝试写成员数据(若其存在的话) _foo，若无尝试写 _isFoo，之后尝试 foo，接着尝试 isFoo；

- 若上面的步骤都失败了，调用方法 setValue:forUnderfineKey:；在 NSObject 缺省实现了这个方法，它抛出了一个异常。

请注意：**“一个对方法 valueForKey: 或者 setValue:forKey: 的调用可能会触发任何合适的方法；若是没有相关的成员数据存在，这个调用就是一个假的。”**例如，调用 valueForKey:@”length” 在语义上同等于直接调用方法 length，因为，它是 KVC 匹配时，第一个被找到的方法。
但是，KVC 的性能明显没有直接调用方法好，使用时请注意。

12.1.3 原型

使用 KVC 需要方法原型符合一定要求：读方法 (getter) 没有参数，返回一个对象；写方法 (setter) 有一个对象为参数，没有返回值。原型的参数的类型并不重要，因为它是 id 类型。
请注意，结构体和原生类型 (int, float ...) 是支持的：Objective-C 运行时可以把它们自动包装为 NSNumber 或者 NSValue 对象。所以，valueForKey: 总是会返回一个对象。

有种特殊情况：将 nil 传入 setValue:forKey 方法，将被方法 setNilValueForKey: 处理。

12.1.4 高级功能

有一些细节值得注意，虽然它们在这里不会被详述。

- 第一点是关于键路径 (keypaths) 可以包含一些特殊处理，如加法，平均值，最小值，最大值等，这些使用 @ 区分。
- 第二点是关于两类方法的一致性：KVC 中的 valueForKey: 和 setValue:forKey:，关联数组这样的集合数据类型提供了 objectForKey: 和 setObject:forKey: (见 11.1 章)。这里还是使用 @ 进行区分。

12.2 属性 (Properties)

12.2.1 属性的使用

类声明时，有属性这样一种概念。我们使用关键词 @property (以及一些参数，见 12.2.3 章) 来标记一个属性，将成员数据和访问器 (可由编译器自动生成) 关联起来。其目标是减少代码量，节约开发时间。
而且，访问属性的语法比调用方法要简单。即使我们自己编写访问器，也值得使用属性。属性的性能和方法调用是一样的，因为编译器在编译期将属性换成了相应的方法。
大多数时候，一个属性是和一个成员数据相关联的。但是，如果访问器被重写了，我们也能提供一个看似为“假”的属性，换句话说，从类外部来看，它就是一个属性值，但在内部其行为比简单的值读写可能要复杂得多。

12.2.2 属性的描述

描述一个属性意味着告诉编译器访问器应该如何被实现：

- 从外部看来，它是否是只读的？
- 如果一个成员数据是原生类型，那么选择会少一些；如果是对象，那它应该被封装成 复制型，强应用的，还是弱应用的？(这关系到内存管理，见 6.4.7 章)
- 它应该是线程安全的吗 (见，8.1 章) ？
- 访问器的名字是什么？
- 需要绑定到哪个成员数据？
- 访问器是自动生成，还是由开发人员编写？

分两步回答这些问题：

- 在类声明 @interface 块中，属性应该结合合适的参数被声明 (见 12.2.3 章)；
- 在类实现 @implement 块中，访问其被表明为隐式的，或给予一个实现 (见 12.2.4)。

访问器的原型是严格的，对于读方法 (getter)，返回类型应该是期望的类型 (或兼容类型)，对于写方法 (setter)，返回 void，而且只能有一个参数，类型应该是期望的类型 (或兼容类型)。

访问器的名字是可以编写的：对于一个名为 foo 的数据，其读方法的名字应为 foo，写方法的名字应为 setFoo:。也允许自定义这些名字。 但不同于 KVC (见 12.1.2 章)，这些名字必须在编译器确定，因为属性被设计成和直接方法调用一样快的功能。所以，也没有提供对于不兼容类型参数的封装。

这里有一个带少许解释的例子，但也能演示其总体行为。如果希望得到全面的理解，后面的子章节会有详细解释。

```
@interface class Car : NSObject
{
  NSString* registration;
  Person* driver;
}
//registration 只读，复制型
@property NSString*  (readonly, copy) registration;

//driver 弱应用(no retain)，并可写
@property Person* (assign) driver;

@end

...

@implementation
//如果开发人员不自己编写"registration"的访问器，编译器将自动生成
@synthesize registration;
```

```
//开发人员将实现"driver"的访问器
@dynamic driver;
//this method will match the getter for @dynamic driver
-(Person*) driver  {
  ...
}

//方法匹配 @dynamic driver 的写方法
-(void) setDriver:(Person*)value {
  ...
}

@end
```

12.2.3 属性的参数

属性参数根据以下模板声明：
@property *type name*;
或者
@property *(attributes) type name*;

如果参数没有给出，它们将有缺省值；否则，需要给出具体值。这些值的含义如下：

- readwrite (缺省值) 或 readonly 确定属性有读写方法还是仅有读方法；
- assign (缺省值), retain 或 copy 确定内部如何存储值；
- nonatomic 不生成线程安全守护。默认是会产生的。(没有关键词 atomic)；
- getter=...，setter=... 用于更改访问器的名字。

在读方法中，assign，retain 或 copy 会影响成员数据改写的方式。

在一个 -(void) setFoo:(Foo*)value 方法中，这三种方式为：
self->foo = value; //简单赋值
self->foo = [value retain]; //赋值，同时应用计数器增1
self->foo = [vlaue copu]; //对象被拷贝，必须符合协议 MSCopying (见 5.3.1 章)
在有垃圾收集的环境里 (见，6.6)，retain 和 assign 是相同的。但这种情况下，参数 __weak 和 __strong 可以被加上。

```
@property(copy,getters=getS,setter=setF:) __weak NSString* s;
```
//复杂的声明 (注意，"setF:" 带冒号)

12.2.4 属性的自定义实现

12.2.2 的代码片段描述了与两个关键字 @synthesize 和 @dynamic 相关的代码实现。

@dynamic 意思是开发者将提供期望的实现 (仅提供写方法的实现，若声明时指明了只读；否则写方法和读方法都需实现)。

@synthesize 意思是，除非开发者自己实现了，编译器将自动生成符合属性声明中种种约束的访问器。在前面给的例子中，如果开发者已经实现了 -(NSString*)registration 方法，编译器将使用开发者的实现，而不是自己生成一个。于是，我们可以可以少写一个方法，它由编译器自动生成，另外一个则由开发者提供。

最后，不同访问器没有在编译时被找到，因为没有指明 @sythesize，编译器又没有自动生成，我们可以在运行时添加访问器 (见 13.2)。这样访问属性是合法的。但是，访问器的名字需要在编译期就被确定。

如果，在运行时，没有任何访问器被发现，那么一个异常将被抛出，但是程序不会被停止；它的行为和缺少了一个方法是一样的。

当使用 @synthesize 时，可以要求编译器将属性绑定到一个特别的成员数据上，成员数据无需有和属性相同的名字。

```
@interface A : NSObject {
  int _foo;
}
@property int foo;
@end

@implementation A
@synthesize foo=_foo; //bind to "_foo" rather than "foo"
                      //(which does not even exist here)
@end
```

12.2.5 访问属性的语法

读写属性值，语法为点操作符：这和 C 的结构体的语法相同，也和键路径 (keypaths) 原则相同 (见 12.1.1)。其性能和直接调用相关方法是一样的。

```
@interface A : NSObject {
  int i;
}
@property int i;
@end
@interface B : NSObject {
  A* myA;
}
```

```
@property(retain) A* a;
@end

...
A* a = ...
B* b = ...;
a.i = 1; //equivalent to [a setI:1];
b.myA.i = 1;//equivalent to [[b myA] setI:1];
```

请注意，在上例类 A 中，self->i 和 self.i 的差别是巨大的。其实，self->i 是直接访问成员数据本身，而 self.i 触发了属性机制，相关方法被调用了。

12.2.6 高级功能

关于属性的文档 [4] 里说，对于64位编译器，Objective-C 运行时有一些和 32位模式不同的地方。关联到 @property 声明的实例数据可能会被忽略，因为它们是隐式的。苹果的文档里有相关内容可以获得全部信息。

13 动态性

13.1 RTTI (Run-Time Type Information)

C++ 也许应该被认为是一个伪的面向对象语言。比较起 Objective-C，它很静态。这是一个为了得到最好的运行时性能而刻意为之的选择。C++ 中，我们能通过过 typeinfo 这个库获得程序运行时的信息。但由于要依赖编译器，它并不安全。在强类型语言中，查询对象的类型并不是一个常用的需求，但在遍历容器时会被用到。dyanmic_cast 操作符，有时 typeid 也会被用到，但在和程序交互时会受到限制。如何在只知道类的名字（字符串）时，测试一个对象是否是一个类的实例呢？

Objective-C 能很自然地解决上述问题。类也是是对象，它们继承它们父类的行为。

13.1.1 class, superclass, isMemberOfClass, isKindOfClass

一个对象在运行时获知自己的类型的能力叫作 内省 (introspection)，它可由一些函数提供。

i**sMemberOfClass :** 是一个方法用于回答这样的问题：“我是给定类的实例（不包括子类实例的情况）吗？”而 **isKindOfClass:** 则回答“我是给定类或者其子类的实例吗？”

这些方法需要使用一个伪的关键词： class (并非用于前向声明的 @class)。实际上，class 是 NSObject 的一个方法，返回一个 Class 对象，它元类的实例。请注意，nil 值是 Nil 类的实例。

```
BOOL test = [self isKindOfClass:[Foo class]];
if (test)
  printf("I am an instance of the Foo class\n");
```

同时注意，你可以通过 superclass 得到其父类。

13.1.2 conformToProtocol

这个方法在关于协议(protocol) (见 4.4)。它允许你知道一个对象是否符合某个协议。但他不是动态的，因为编译器只检查显式的一致性而不是去检查每个方法。如果一个对象实现实现了给定协议的所有方法，却未显式地声明它符合某个协议。程序中，对象符合协议，但 **conformsToProtocol:** 却返回 NO。

13.1.3 resondsToSelector, instancesRespondToSelector

responsToSelector: 是一个继承于 NSObject 的实例方法。它能判断一个对象是否实现了给定的方法。选择器 (selector) 的概念被使用到 (见 3.3.4)。举例：

```
if ( [self respondsToSelector:@selector(work)] )
{
  printf("I am not lazy.\n");
  [self work];
}
```

为了判断一个类是否实现了给定方法，但不检查其继承的方法，我们可以使用 class方法 **instancesRespondToSelector:** 。

```
if ([[self class] instancesRespondToSelector:@selector(findWork)])
{
   printf("I can find a job without the help of my mother\n");
}
```

请注意，**responsToSelector:** 不能确定前向声明 (forwarding) 的类的对象是否能处理一条消息。

13.1.4 强类型和 id 弱类型

C++ 使用强类型：我们只能根据对象的类型使用对象，否则无法通过编译。在 Objective-C 中，如果一个对象的类型是显式知道的，限制会更灵活。编译器只会产生一个警告 (warning)，程序还是能够运行。消息将会简单地被丢弃(引发一个异常)，除非消息转发机制被触发了(见 3.4.3) 。如果这是开发人员期望地，那么警告就是多余的；这种情况下，使用弱类型替代真实类型可以消除警告。实际上，任何对象都是 id 类型，id 类型能成为任何消息的发送对象。

在使用代理时，弱类型属性是必须。代理对象不必知道自己被使用了，如：

```
-(void) setAssistant:(id)anObject
{
  [assistant autorelease];
  assistant = [anObject retain];
}

-(void) manageDocument:(Document*)document
{
  if ([assistant respondToSelector:@(manageDocument:)])
    [assistant manageDocument:document];
  else
    printf("Did you fill the blue form ?\n");
}
```

在 Cocoa 的图像界面编程中，代理使用得非常普普遍。所有的控制通过将用户对象的动作转移到作为代理的业务对象。

13.2 运行时操作 Objective-C 的类

通过包含头文件 <objc/objc-runtime.h>，我们能调用若干工具函数，在运行时改变类的信息，诸如方法和实例变量.

Objective-C 2.0 引进了一些更实用的新方法 (如，用于替代class_addMethods 的 class_addMethod(…))，废止了 1.0 版很多的函数。

在运行时，改变一个类明显变得更为容易。即使运行时改变类的需求不是那么普遍，但在很多情况下，这是一种很有价值的功能。

14 Objective-C++

Objective-C++ 仍在发展之中。它是可用的，虽然仍然有一些问题有待突破。Objective-C++ 使得我们可以同时使用 Objective-C 和 C++，以获得两者的不同的优势。

混用 C++ 和 Objective-C 的异常时，将 C++ 对象作为 Objective-C 对象的实例数据时，会有一些意料之外的行为，理解这些需要查询相关的文档。尽管如此，使用 Objective-C++ 做开发无疑是可能，同时已有部分程序是由 Objective-C++ 编写的。这些文件以 .mm 为后缀名。

从 MacOS X 10.4 开始， C++ 对象可以作为 Objective-C 类的实例数据，也能被自动被创建和被析构[3]。

15 Objective-C 的未来

我没有诸如 Objective-C 3.0 或其他的关于 Objective-C 未来发展的信息。 但是，Objective-C 自然受益于 C 语言的发展进步。GCC[4] 和 LLVM[5] 项目便很有趣。

GCC, 按照标准，将持续增强它对 C++ 的支持。Objective-C 无疑将受益于此。此外，LLVM 最近通过引进 块(block) 的概念增加了对 闭包(Closure) 的支持。这是一个 C 语言的重大改进，而 Objective-C 也可以使用这项改进。

15.1 块(block)

15.1.1 支持和使用案例

从MacOS X 10.6 开始，原生 API，如 GCD(Grand Central Dispatch)，就能使用块。它的语法类似于函数指针，只是星号被重音符代替了。一个例子：

```
void dispatch_apply(size_t iterations, dispatch_queue_t queue, void (^block)
(size_t));
```

最后一个参数不是函数指针，而是块。

块是一个与 Cocoa 没有关系的概念，它其实是 Objective-C 语言本身的特性。块来源于 C 语言，目前还不是标准，不被所有的编译器支持。在MacOS X 10.6 中发布的 LLVM 编译器提供了对块的支持。

对于已经了解块的人来说，块是已在 Javascript 中被使用的闭包的一个 C 语言实现。简便起见，它可以被视为匿名函数，使用时才被创建，同时能记录创建时的环境(各变量的值)。

块被用于替代回调是很实用的，因为它能减少为实现异步或延迟调用所书写的代码量。

15.1.2 语法

一个块可被如下创建

^{... du code ...}

或者

^(parameter1,[other parameters...]){...du code...}

它能被作为一个变量储存，将重音符号替换成星号后也可以作为一个函数指针被调用。

```
Typedef void (^b1_t)(void);
b1_t block1 = ^{printf("coucou\n");};
block1(); // 显示 coucou

typedef void (^b2_t)(int i);
b2_t block2 = ^(int i){printf("%d\n", i);};
block2(3); //显示 3
```

15.1.3 记录环境

块能引用其创建时的外部变量。它记录这些变量的状态，并能实用它们，但只能读。在块中更改不属于这个块的变量需要一些特殊的指令(见 15.1.4)。

```
// example 1
int i = 1;
b1_t block1 = ^{ printf("%d", i); };
block1(); // 显示 1
i = 2
block1(); // 仍显示 1 ！值为块创建时被记录的值

// example 2
int i = 1;
b1_t block1 = ^{ i=2; } // 非法 ： 无法编译。外部变量不能被修改

// example 3
typedef void (^b1_t)(void);
b1_t getBlock()  // 此函数返回一个块
{
      int i = 1; // 此变量为函数的本地变量
      b1_t block1 = ^{ printf("%d", i); }
      return block1;
}

b1_t block1 = getBlock();
block1(); // 显示 1 ： 块记录了其创建时的本地变量 的值
```

15.1.4 __block 变量

块可以使用一个已不在其当前作用域的变量，这件事情看起来有点诡异：块只是做了一份拷贝。这就是为什么，你不能在块内改变一个外部变量：为了避免混淆。

但也可突破这个限制。如果一个变量被声明为 __block 属性，块内便能修改它。这很微妙，因为被记录的值在块被调用时，必须仍是可被修改的。

值得一提的是，extern 或 static 变量 可以被块修改，因为他们不是 auto 的(C 语言的缺省方式)。

```
//exemple 1
__block int i = 1;
b1_t block1 = ^{ printf("%d", i); };
block1();//display 1
i = 2;
block1();//display 2

//exemple 2
__block int i = 1;
b1_t block1 = ^{printf("++i = %d", ++i);};
block1(); //显示 2
block1(); //显示 3

//exemple 3
typedef void (^b1_t)(void);
b1_t getBlock() //这个函数返回块
{
  __block int i = 1; // 这个变量是函数的本地变量，被返回的块在函数 getBlock() 外，将不在合法

  b1_t block1 = ^{ printf("%d", i); };
  return block1;
}

b1_t block1 = getBlock();
printf("Caution: the following call is invalid\n");
block1(); // 由于变量 "i" 的缘故， 可能引发段错误 Segmentation Faults
```

结语

这份文档仅提供了一个 Objective-C 与 C++ 各个方面的概念的粗略比较。因此，它作为那些希望学习 Objective-C 的高级程序员的一个便捷的参考手册是有用的。希望这个目标已经达到了，我仍欢迎读者的反馈，帮助我提高这份文档的水平。

引用

[1] Apple Computer, Inc. Autozone. http://www.opensource.apple.com/darwinsource/10.5. 5/autozone-77.1/README.html.

[2] Apple Computer, Inc., Developer documentation. Garbage Collection Programming Guide. http://developer.apple.com/documentation/Cocoa/Conceptual/GarbageCollection. pdf.

[3] Apple Computer, Inc., Developer documentation. Key-Value Coding Programming Guide. http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding/ KeyValueCoding.pdf.

[4] Apple Computer, Inc., Developer documentation. The Objective-C 2.0 Programming Language. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf.

[5] Boost. http://www.boost.org/.

[6] GNUstep. http://www.gnustep.org/.

[7] Aaron Hillegass. *Cocoa Programming for MacOS X, 2nd edition*. Addison-Wesley, 2004.

[8] SGI. Standard template library programmer's guide. http://www.sgi.com/tech/stl.

[9] Will Shipley. self = [supid init]. http://wilshipley.com/blog/2005/07/ self-stupid-init.html.