第4章 Foundation: Objective-C基础库

Objective-C 语言核心只定义了两个类：对象（Object）和协议（Protocol）。无论是 GNUstep、Cocoa、libfoundation，或者 Cocotron，都不太可能只使用 Objective-C 而不使用 OpenStep Foundation。可移植对象编译器也提供了一套自己的核心对象，但使用不广泛。

OpenStep Foundation 非常接近 Objective-C 的标准库，它差不多相当于 C 的标准库或者 C++ 的 STL。因为 Objective-C 是 C 的纯超集，所以 C 的标准库也可以使用。准确描述一下最初的想法就是：在 C 的软件基础上，用 Objective-C 来创建组件。

OpenStep 引入基础库，用来隐藏 NeXTSTEP 的以 Mach 为基础的操作系统和 Solaris 之间的差距，并使编写字节序无关（endian-independent）代码更容易。大部分基础库是不依赖字节序的，在 Apple 从大字节序（big-endian）的 PowerPC 移植到小字节序（little-endian）的 x86 架构时，这带来了巨大的好处。

4.1 一般概念

尽管基础库很巨大，但学起来并不难。很多类使用了通用的设计原则，理解了这些概念，就可以很快学会如何使用每一个独立的类。

4.1.1 可变的

Objective-C 没有常量对象的概念。其实这么说也不完全正确，C 的关键字 const 仍然存在，但只用于直接访问实例变量。方法不能被标记为存取器（mutator），所以发送到对象的任何消息都有可能改变它，无论对象指针是否是 const。

所以说，在许多情况下，存在对象的可变和不可变版本是很有用的。在面向对象系统中，一般通过可变和不可变两种类来完成。String 类是一个常见的例子。如果你创建一个内容为 @"like this" 的 Objective-C 字符串，就创建了一个常量字符串。编译器会把字符串直接放进二进制代码的常量区域，如果修改它，就会导致 segmentation fault 错误。如果创建一个新字符串并复制，会让程序变得很慢。这也是公认 Java 代码速度慢的原因之一，Java 的字符串是不可变的，并且它声明为 final，也就没法用 Cocoa 的可变版本子类的方法解决。

NSString 对象是不可变字符串。它有一个子类 NSMutableString。因为是子类，所以任何可以用不可变版本的地方也都可以使用这个可变版本。它实现了全部相同的方法。

可变版本和不可变版本的最大区别，是 -copy 方法的实现。当发送 -copy 消息给不可变对象时，一般得到的返回值是同样的对象（retain 计数会增加）。因为你无法修改它们的任何一个“copy”版本，所以每个对象之间绝对不会有不同。

这种能力是 Objective-C 程序时常比 C++ 快的原因之一，虽然很多微型基准测试会得到相反的结论。在 C++ 程序中，std::string 会真正发生复制。一个 C++ 字符串可能会被复制很多次，但 Cocoa 字符串只是增加和减少引用计数。

4.1.2 Class Cluster

虽然 NSString 是用于不变字符串的类，但你的字符不会真的成为 NSString，而会是 NSConstantString 之类的。这是 NSString 的私有子类，用于特定用途。

在 Cocoa 中这很常见。公共类，比如 NSDictionary，可能会有很多个不同的实现，这些不同的实现为不同的用途优化。在初始化时，会得到一个特定的子类，

而不是抽象超类。

可以使用两种方法。第一个方法是对不同的构造方法或初始化方法返回不同的子类。第二个方法是使用同样的实例变量布局，然后使用一种称为 isa-swizzling 的技巧。isa 指针，是指向对象类的指针，是另外一种实例变量。按照 Objective-C“没有魔法”的哲学，它同样没什么特别的。你可以按自己的希望给它分配值。当新类和旧类在内存中有同样布局时，一切都工作正常。（不然就会遇到一些难于调试的内存问题。）

Class cluster 让子类变得稍有些不同。一般来说，cluser 中每个隐藏的类中实现的方法只是原生方法的一小部分。在 NSString 中，有 -characterAtIndex: 和 -length。所有其他方法都是在超类中实现。如果要创建新的 NSString 子类，必须自己实现这些方法。

当然，如果你愿意实现除这两个原生方法之外的其他方法也没问题。可以对其中的部分实现更高效的版本。

更多关于 isa-swizzling 的内容

除了 class cluster，isa-swizzling 技巧还在很多情况会用到。它可以用于调试 use-after-free 内存问题，具体做法是稍微改变一下类的 -dealloc 方法，让它把所有收到的消息都抛出异常。也可以用它实现状态机，每一个状态都是一个公共类的不同子类。要进入新的状态，只需要简单地把 isa 指针指向另外一个指针的类即可。

你可以自己实现 class cluster，这并不难。典型地，你的 public 类中有好几个不同的初始化方法，每个方法返回的都是不同的子类实例。我们封装一堆值来演示它。接口如代码清单 4.1 所示。注意，这里没有声明实例变量。

代码清单 4.1：pair 类公有接口（取自 examples/ClassCluster/Pair.h）

```
3  @interface Pair : NSObject {}
4  - (Pair*) initWithFloat:(float)a float:(float)b;
5  - (Pair*) initWithInt:(int)a int:(int)b;
6  - (float) firstFloat;
7  - (float) secondFloat;
8  - (int) firstInt;
9  - (int) secondInt;
10 @end
```

在实现文件中，我们定义 Pair 类的两个实体子类，一个用来保存整数，另外一个保存浮点数。如代码清单 4.2 所示。它们都没有定义新的方法。因为接口都是私有的，就算定义了新方法，也没有人知道如何调用。不过它们定义了结构。这样实现出来的 class cluster，可以让不同的实现有不同的数据布局。

代码清单 4.2：pair 实体类的私有接口（取自 examples/ClassCluster/Pair.m）

```
3  @interface IntPair : Pair {
4      int first;
5      int second;
6  }
7  @end
8  @interface FloatPair : Pair {
9      float first;
10     float second;
11 }
12 @end
```

代码清单 4.3 是公有类的实现，非常简单。大部分方法只是简单地返回一个默认值，因为它们不会被调用。更健壮的实现大概应该在这些地方抛出异常。

代码清单 4.3：pair 公有类的实现（取自 examples/ClassCluster/Pair.m）

```
14 @implementation Pair
15 - (Pair*) initWithFloat: (float)a float: (float)b
16 {
17     [self release];
18     return [[FloatPair alloc] initWithFloat: a float: b];
19 }
20 - (Pair*) initWithInt: (int)a int: (int)b
21 {
22     [self release];
23     return [[IntPair alloc] initWithInt: a int: b];
24 }
25 - (float) firstFloat { return 0; }
26 - (float) secondFloat { return 0; }
27 - (int) firstInt { return 0; }
28 - (int) secondInt { return 0; }
29 @end
```

注意，在两个初始化方法中，都有 [self release] 这行。对象创建的典型过程，是先发送 +alloc 消息给 Pair 类，然后对返回值发送初始化消息。因为我们并不需要 +alloc 返回的对象，所以就释放掉它，返回一个新对象。

代码清单 4.4 是私有 pair 类的实现。每个类都单独实现了和数据类型相关的构造方法。访问器方法会返回实例变量的值，或是转换类型之后的实例变量的值，

两种 pair 都可以返回整数或是浮点数。两个类都实现了来自 NSObject 的 -description 方法，这个方法用人类可读的方式返回对象的内容。注意，它们都没有调用父类中指定的初始化方法，这种代码风格是不好的，但这个例子很简单，这样做没问题。

代码清单 4.4：私有 pair 类的实现（取自 examples/ClassCluster/Pair.m）

```
@implementation IntPair
- (Pair*) initWithInt: (int)a int: (int)b
{
    first = a;
    second = b;
    return self;
}
- (NSString*) description
{
    return [NSString stringWithFormat: @"(%d, %d)",
           first, second];
}
- (float) firstFloat { return (float)first; }
- (float) secondFloat { return (float)second; }
- (int) firstInt { return first; }
- (int) secondInt { return second; }
@end
@implementation FloatPair
- (Pair*) initWithFloat: (float)a float: (float)b
{
    first = a;
    second = b;
    return self;
}
- (NSString*) description
{
    return [NSString stringWithFormat: @"(%f, %f)",
           (double)first, (double)second];
}
- (float) firstFloat { return first; }
- (float) secondFloat { return second; }
- (int) firstInt { return (int)first; }
- (int) secondInt { return (int)second; }
@end
```

Pair 类的用户并不需要知道任何一个私有类的存在。用一个简单的测试程序就可以演示这种状况。代码清单 4.5 是一段很短的程序，它只创建两个 pair 对象，并输出记录信息。NSLog 输出的字符串格式是通过每个类中的 -description 方法获得的。

代码清单 4.5：pair 类的演示（取自 examples/ClassCluster/test.m）

```
#import "Pair.h"

int main(void)
{
    [NSAutoreleasePool new];
    Pair *floats = [[Pair alloc] initWithFloat:0.5 float:12.42];
    Pair *ints= [[Pair alloc] initWithInt:1984 int:2001];
    NSLog(@"Two floats: %@", floats);
    NSLog(@"Two ints: %@", ints);
    return 0;
}
```

运行程序，将获得以下输出：

```
2009-01-14 14:27:55.091 a.out[80326:10b] Two floats: (0.500000, 12.420000)
2009-01-14 14:27:55.093 a.out[80326:10b] Two ints: (1984, 2001)
```

更完整实现的 cluster 会提供有名字的构造函数，比如 +pairWithInt:int:，这样就可以避免先分配然后释放 Pair 对象实例。还有其他的方法可以避免这种情况，比如之前提到过的 isa-swizzling。Pair 类有整数和浮点数两个实例变量。实现出来的初始化程序是这样的：

```
- (Pair*) initWithFloat: (float)a float: (float)b
{
    isa = [FloatPair class];
    return [self initWithFloat: a float: b];
}
```

这段代码中，第一行把类指针设置到了子类，第二行再次调用了这个方法。因为类指针被修改了，第二次调用就会调用子类实现的方法，所有子类中的变量都会指向正确的字段。

4.2 Core Foundation类型

Core Fundation（CF）库中包含了一系列 C 的 opaque 类型，它们和很多 Cocoa Foundation 对象接口相似。这种相似并非偶然。Core Foundation 的目标是提供一套丰富通用的基础类型，它们要能用于 Cocoa 和 Carbon 应用程序。后来 Carbon 没有切换到 64 位，所以这个目标不那么重要了，但 Core Foundation 仍然由 OS X 的大量底层部分使用，比如 Launchd。

尽管 C 语言没有类型的继承概念，Core Foundation 类型还是使用了继承。以 CFType 为根，Core Foundation 为 CF 类型实现了基础内存管理。正如 Cocoa 对象

通过 -retain 和 -release 消息进行引用计数，Core Foundation 类型的引用计数通过调用 CFRetain() 和 CFRelease() 并传递参数的方式使用。

很多 Core Foundation 类型通过 toll-free bridging 机制和对应的 Cocoa 对象互操作。任何 CF 结构中的第一个字段都是 isa 指针，和 Objective-C 对象一样。当然，和 Objective-C 不同的是，它的值始终在 0 ～ 216，这是一个没有 Objective-C 类使用的内存区域。当发送消息给 CF 对象时，发送消息的函数会使用一个特殊的转型来对应这个区间的类指针。

同样，当通过 Cocoa 对象调用 Core Foundation 函数时，会检测 isa 指针是否大于 0xFFFF，如果大于，就会调用 Objective-C 来指派方法的运行时函数，跳回 Objective-C 中调用。这样就可以交换 Core Foundation 类型和 Cocoa 对象。

大量 Cocoa Foundation 对象都有类似的 Core Foundation 类型。最常见的大概是 CFString，它等同于 Cocoa 的 NSString。事实上，NSString 和 NSMutableString 都是 Cocoa 中的 class cluster，就是说，它们的实例并不一定真的是这个版本的类。在背后，三个类型都是通过 NSCFString 类型实现的。很多 Cocoa 的 class cluster 都是这样的。

4.3 基本数据类型

基础库提供了很多数据类型，一些是 C 的原生类型，另外一些是对象类型。一些用来表现某种结构数据，比如字符串或日期，另外一些是任意类型的容器类型。

任何重要的 Cocoa 程序都会大量用到它们。有大量的方法可用来操作这些数据类型，在使用这些类型实现新特性之前，要仔细阅读文档。

4.3.1 非对象类型

OpenStep 本来是被设计为在以今天的标准看来很慢的计算机上工作。Smalltalk 最大的性能提升之一，就是明智地使用了非对象类型。这些类型中最常见的就是整数和浮点数变量类型。除此之外，还有一小部分结构体，比如 NSRange，整个 Foundation 框架都在使用它。

为什么不用对象，有几个原因。首先是它们的大小。这种结构体大多数是一对值。比如一个范围（range）就是起始位置和长度。增加一个 4 字节的 isa 指针和 4 字节的引用计数，等于让它的大小翻倍了。把它们变成结构体，可以在寄存

器中传递值，这会让使用或返回它们的方法（或函数）调用更快。最后，它们很少用到别名。当在某处设置一个范围、点或矩形的时候，你需要一个副本而不是别名。

Cocoa 中最常用的结构如下：

- NSRange，一对正整数，用来表现序列中从某个偏移位置开始的长度。常用于在 NSString 中定义子串，也可用于 Array 和其他类似的数据结构。
- NSPoint，两个浮点数，在坐标系中表示 x 和 y。
- NSSize，结构和 NSPoint 一样。和 NSPoint 的区别是，NSSize 不会是负数。虽然作为结构体，并没有办法确保这种约束，但给 NSSize 分配了负数，会导致异常或其他的错误。
- NSRect，是 NSPoint 和 NSSize 的组合，用来在 2D 空间中定义一个矩形。

注意，最后三个类型一般用于 AppKit 的绘图函数，尽管它们是在 Foundation 库中定义的。

CGFloat、NSUInteger，以及其他类似类型

在 10.5 之前，大部分结构体使用 int、float 和类似的类型。10.5 中，Apple 重新定义了很多类型，函数开始使用 CGFloat 和 NSUInteger 类型。新的 NSUInteger、NSInteger 类型分别等同于 C99 的 uintptr_t 和 intptr_t。它们提供了和 C89 及更老的方言的兼容性。CGFloat 在 32 位系统中定义为 float，在 64 位系统中定义为 double。

Foundation 库也包含了不少其他原生数据类型，包括很多枚举类型。常见的例子包括 NSComparisonResult，它定义了 NSOrderedAscending、NSOrderedSame 和 NSOrderedDescending，用于定义两个对象如何排序。如果要对一个容器内的 Cocoa 对象排序，则对集合中的对象调用一个函数，函数的返回值是前面提到的三个值中的一个，这个值将确定新的顺序。

4.3.2 字符串

NSString 是 Foundation 库中最常用的类之一。从技术上说，其实用到的是 NSString 的子类，因为它是一个 class cluster，并不会被直接使用。

每个 NSString 的实体子类，必须覆写至少两个方法，这两个方法在 NSString 中定义，分别是 -length 和 -characterAtIndex:。第一个方法返回字符串的长度，第

二个方法返回字符串中指定索引位置的 unicode（32 位）字符。注意字符串的内部格式可以是任意格式的。NSString 的 class cluster 设计为允许 8、16 和 32 位字符串存储在内部，如果给出的字符串中没有集合之外的字符，就可以用它们表示。NSString 的子类会透明地处理任何所需的转换，以致于大部分程序员会忘记这件事。

尽管必须覆写的方法只有这些，但 NSString 中大部分方法都调用了 getCharacters:range: 方法，它会在调用者提供的缓冲区中写入一个子串。父类中的实现方法是重复调用 -characterAtIndex: 完成的，所以在子类中直接实现这个方法，会比调用父类中的方法快很多。

注意，这个方法的前缀是 get。在 Cocoa 中，这是一种习惯用法，用于在调用者提供的空间中返回值。而 length 方法是没有前缀的，就只是返回长度。

尽管可以创建自己的 NSString 子类，但使用无子类的对象会更好。比如 Foundation 库中的 NSAttributedString，它响应 -stringValue 消息时，会返回带有保存的属性的字符串，但不能直接用在使用字符串的地方。第 8 章我们会研究这个类的更多细节。

NSString 有一个公共子类（也是一个 class cluster）NSMutableString，它可以修改展现的字符串。它增加了修改字符的方法。这个类增加了 7 个新方法，其中 6 个是以 replaceCharactersInRange:withString: 方法为基础的。

NSString 类有大量方法，10.5 中又增加了一些新的方法。大部分都是用于路径操作。之前很长时间困扰 OS X 开发者的问题是，MacOS 和 OPENSTEP 用不同的方法来表现路径。MacOS 使用多路径文件层次，每个磁盘是一个文件，路径用冒号分隔。OPENSTEP 用 UNIX 风格的文件层次，是一个单根，路径用斜杠隔开。Mac OS X 应用程序通常要处理这两种情况。

幸好 NeXT 早就遇到了这些问题。OpenStep 应用程序可以运行在 Solaris、OPENSTEP 和 Windows。Windows 文件路径结构和传统的 MacOS 路径结构相似。NSString 有一系列方法来增加 / 删除路径的组成部分，也可以把路径切成几个部分，而不依赖于文件系统用什么方式表现。应该使用这些最佳实践代替用手工构建路径。

最近版本的 OS X 开始摆脱使用文件路径，很多方法使用名字空间为 file:// 的 URL 来替代文件路径。NSString 中没有多少处理 URL 的方法，但 NSURL 类提供了很多。

4.3.3 数字和值的装箱

原生类型的优势是效率，缺点是对于期望对象的容器不够智能。在这种用途上，有三个类可以用来替代，每个类都装箱了一种原生类型。

> **装箱**
>
> 装箱是一个术语，指的是在对象中包裹基础类型。Lisp 和 Smalltalk 之类的高级语言会自动处理装箱，这样在需要对象的时候也可以直接用原生类型。Objective-C 必须手工装箱。

最常用的装箱类是 NSValue，它可以装入任何原生类型。常见用法是用来封装 Foundation 库的结构类型，比如 NSRange，并把它们存储到容器里。这个类有一个子类（确切地说，是一个 class cluster）NSNumber，用来保存单独的数字值。从 char 到 longlong 类型，任何值都可以存储在它里面，并且可以通过使用 -somethingValue 系列方法转换成所需结果。比如，可以创建 NSNumber 来存储一个原生的 unsigned int：

```
[NSNumber numberWithUnsignedInt: myInt];
```

它可以被存储进容器，取回，传递给另一个方法，也可以转换成 64 位值，如下：

```
[aNumber longLongValue];
```

这里要注意，如果你想做反向操作，也就是说，创建一个存有 64 位值的 NSNumber，然后取回 32 位或更小的值，得到的数值会被悄无声息地截断。

十进制运算

除了继承自 C 的标准二进制类型及其装箱后的等效类型，Foundation 库还定义了 NSDecimal 结构和对应的装箱类型 NSDecimalNumber。它们可用来进行浮点数指针运算。一些小数，比如 0.1，不能用有限二进制值表现。这对于金融类应用程序很麻烦，它们必须使用固定长度的小数。NSDecimal 类型可用以解决这个问题。

NSNull 经常被忽视，它是一个单件实例，用来表现 NULL 的装箱值。[1]

1 容器对象中不允许nil值，所以要使用NSNull来表现null值。

> **0 值的多种类型**
>
> 在 C 语言中，NULL 被定义为 (void*)0，一个指向 0 值的指针。因为 void* 类型可以被转换成任何指针，因而 NULL 值可以用于任何指针。Objective-C 增加了两个新的 0 值类型：nil 和 Nil，分别代表 (id)0 和 (Class)0。另外还有装箱版本 NSNull，以及被装箱进 NSValue 和 NSNumber 对象的 0 值。也就是说，在 Cocoa 中，根据用途的不同，0 有很多种表示方法。

和 Foundation 库中其他类不同，没有 NSMutableNumber 和 NSMutableDecimalNumber 这两个类存在。如果需要修改已经装箱的值，将需要先拆箱，然后修改原生类型的值，最后重新装箱。这样做的意义在于，操作原生类型会比发送消息快很多。在 Smalltalk 或 Lisp 这类语言中，编译器会尝试替你把对象转换到原生类型值，让这个操作变成透明的，但 Objective-C 编译器还没有智能到这么做。

4.3.4 数据

在 C 语言中，各种数据通常都用和字符串一样的方法表示，都是 char*s。在 Cocoa 中，字符串对象没法这么用，因为要进行编码转换。可以用 NSData 类封装原数据。可以把这个类看做 void 的装箱版本，尽管它也会保存长度以防止指针悬空覆盖掉其他内存地址造成 bug。

发送 -bytes 消息，可以得到一个指向对象数据的指针。看起来这样更高效，但并不总是这样。比如有时候要表现一系列不连续的内存区域，或是没有被读入内存的数据。当调用对象的 -bytes 时，要确保所有数据都在连续的内存区域，这可能是个比较昂贵的操作。在此之后的操作，不需要再进行交换，会非常快。

简单的文件 I/O 操作，可以使用 NSData 和它的不可变子类 NSMutableData 完成。数据对象可以使用文件内容初始化，无论是通过文件读入操作或是使用 mmap()。使用内存映像（memory-mapped）NSData 对象进行文件的随机存取会更方便。在 32 位平台，这样会很快耗尽地址空间，但在 64 位系统有足够的多余地址用于内存映射文件。

用这种方法访问文件是 VM 友好的。如果把文件内容读入内存，系统可用内存就会降低，然后系统得把副本写入交换文件，甚至你根本没修改过。如果通过 dataWithContentsOfMappedFile: 之类的方法创建 NSData 对象，就会简单地收回内

存页，然后重新从需要的原始文件读回来。

因为 NSData 对象可以从 URL 初始化，因而提供了一种非常简单的办法来访问系统的URL载入服务。OS X对不同的URL类型都有载入数据的方法，包括文件、HTTP 和 FTP。

4.3.5 缓存和丢弃数据

对于大部分现代应用程序来说，内存保护是一个重要问题。这几年，内存价格快速下降，用内存来存储计算结果或通过网络接收到的数据已经成为重要方法。这样的程序要移植到 iPhone 之类的小内存设备时，就会很麻烦。

OS X 10.6 之后，Apple 引入了 NSDiscardableContent 协议。它定义了用于对象的事务 API。在使用一个实现此协议的对象之前，需要先给它发送 -beginContentAccess 消息。

如果返回 YES，就可以使用这个对象，完成使用之后，再发送 -endContentAccess 消息。其他代码可能会发送 -discardContentIfPossible 消息给这个对象，如果在事务之外收到这个消息，接收者会丢弃它的内容。

通过一个具体实现最容易理解它，NSMutableData 有一个叫做 NSPurgeableData 的子类，其行为和 NSMutableData 完全相同，但也实现了 NSDiscardableContent 协议。当收到 -discardContentIfPossible 消息时，它会释放掉所封装的数据，除非数据正在被访问。

如果要在现有代码中结合使用了 NSDiscardableContent 协议的对象，NSObject 定义的 -autoContentAccessingProxy 提供了安全的办法。这个方法会返回一个代理对象给接收者，这个代理对象会在创建时发送 -beginContentAccess 给接收者，在对象销毁的时候发送 -endContentAcces，并把所有其他消息发给原来的对象。在代理存在期间，可以防止对象中的内容被释放掉。

存储缓存对象时，这个方法很有用，比如把应用程序中其他数据展示为图形，在需要的时候可以重新生成图像。对象始终有效，但用它生成的内容不一定始终有效。也就是说，在非垃圾回收环境中，这是一种可以被归零的弱引用。它比弱引用更灵活，当需要被释放时，这种方法可提供足够的控制粒度。

更常见的情况，实现这个协议的对象会和 NSCache 结合使用。这个类和字典类的概念差不多，但它设计为存储可丢弃的内容。使用 -setObject:forKey:cost: 方法可以往缓存中加入对象，第三个参数定义了在缓存中保持这个对象的成本

（cost），当总成本超过 -setTotalCostLimit: 设置的限制时，缓存就会尝试丢弃一些对象的内容（也有可能是这些对象本身）来降低成本。

最常见的成本限制就是内存。当使用 NSPurgeableData 实例时，可以使用它的大小作为限制。你也可能会用缓存来限制一些紧缺资源的对象数量，比如文件句柄，或者其他服务器上的远程资源。

4.3.6 日期和时间

POSIX 系统的时间存储为 time_t 值。传统 UNIX 系统中，它是一个有符号 32 位值，记录从 UNIX 时代（从 1970 年开始）到现在的秒数。也就是说，到 2038 年，值会溢出，产生新的千年虫问题。在 OS X 中，time_t 是 long 类型，所以在 32 位系统上它是 32 位，64 位系统上是 64 位。300 兆年之后，如果人们还在用 OS X，那时候才会导致溢出，大概这个时间足够程序员把自己的程序移植到新系统上了。

因为 time_t 依赖于具体实现，所以不是很适合 Cocoa。在一些平台上它是整数，另外一些平台上它是浮点数。Cocoa 定义了双精度（double）的 NSTimeInterval 类型。NSTimeInterval 是浮点数类型，其精度依赖于值的大小。双精度类型中 53 位是尾数，10 位是指数。如果尾数的最低有效位为毫秒，则它的值可以存储 9×10^{12} 秒，约为 285 427 年。如果存储范围是 10 万年，就可以按照半毫秒存储，依此类推。对于 32 位 time_t 存储的值，可以精确到 1 微秒以下，这精度远超过一般需要。大部分 UNIX 类系统的时间片约为 10ms，所以很难获得低于 10ms 的计时器事件。

和其他原生类型一样，Foundation 库定义了原生类型和大量类，使和这些类型的交互更友好。并且用 2001 年（这是 OS X 发布的年份）作为参考日期的起始，以使之更精确。

日期操作比时间操作更复杂。使用 NSTimeInterval 可以轻松表达 400 年之前的时间，但获得对应的日历就要复杂很多。格列高利历是 1582 年引入的，但英国在 1752 年之前没有使用这种日历，俄国在 1918 年之前没使用这种日历。闰年和闰秒的存在让这个问题进一步复杂。就是说，NSTimeInterval 可能要在不同的地点表现为不同的日期。这些就是关于时区的麻烦问题。

NSDate 类清晰简单地包装了距离参考日期的时间间隔（默认是 2001 年，尽管 UNIX 时代和当前时间戳未必是这样）。NSCalendarDate 子类提供了一个版本的格列高利历，虽然并不推荐使用它。

从 10.4 开始，Apple 引入了 NSCalendar 类，它封装了日历。日历是解决内部时间和日期映射的机制。早期的日历只简单解决了固定日期的映射，比如夏至、

冬至和季节。现代日历映射了内部时间和更复杂的日期。Cocoa 能理解大量不同的日历，比如格列高利历、佛历、中国农历、伊斯兰日历。

如果通过 +autoupdatingCurrentCalendar 创建 NSCalendar，日历会自动根据当前指定的地区更新。因为未来日历有可能会发生变化，所以不要缓存日历返回的值。

NSCalendar 可以把 NSDate 转换到 NSDateComponents 对象，它和 POSIX 的结构 tm 大致等价。它允许年、月、日、星期几，并且可以依据 NSDate 在指定日历中的翻译抽取出来。

通常，应该永远把日期保存在 NSDate 对象中，只在需要显示在用户界面上时转换为所需的日历。这就是不推荐使用 NSCalendarDate 的一个原因，虽然它作为 NSData 的子类，看起来很适合长期使用。另外一个原因是它被限制在格列高利历，所以在中国、日本以及美国和欧洲之外的其他多数地区都不适用。

4.4 容器

大部分语言的标准库都提供了容器，Foundation 库也不例外。它包括了不少定义为不透明 C 类型的原生容器类型，然后通过它们创建了更复杂的 Objective-C 类型。

对比 C++ 标准模板库（STL），Cocoa 容器是异构的，可以容纳任何种类的对象。所有对象都通过指针引用，所以存储任意两个对象的指针永远是一样的，都是一个字的大小。

4.4.1 比较和排序

对于排序容器，对象要实现自己的比较方法。几乎任何对象都可以保存在数组中，而存储在 set 中或是用 key 存储在字典中，要求会更严格一些。用这种方法存储的对象需要实现两个方法：-hash 和 -isEqual:。它们有复杂的关系。

1. 任意两个对象通过 isEqual: 比较时，只要它们是相等的，必须返回 YES。

2. 任意两个相等的对象对于 -hash 必须返回同样的值。

3. 存储在容器中的任意对象的 hash 必须是常量。

有时候第一条原则只靠本身是不好实现的。第一条原则的意思是下面的表达

式总是为真：

```
[a isEqual: b] == [b isEqual: a]
```

如果这个不是永远为真，就有可能发生意料之外的奇怪问题。看起来这容易做对，但当你比较一个对象及其子类，或者另外一个类对象时会发生什么事呢？一些类可能允许和其他类比较，比如说，一个封装了数字的对象和另外的对象，如果它们的 intValue 相同，就有可能会被判断为相同。

如果使用这些对象作为字典的 key，就会出问题。当你设置某个 key 在字典中的值的时候，字典首先检查是否已经存在了这个 key。如果存在，就替换现有值，如果没有，就插入一个新的值。

如果 [a isEqual: b] 返回 YES 但 [b isEqual: a] 返回 NO，那么你用 a 做 key 和用 b 做 key 会得到不同的字典。所以，一般来说，使用容器的最佳实践是用同样的对象类型做 key（最常用 NSString）。

代码清单 4.6 是一个简单的例子。它定义了三个新类。第一个，A，是另外两个类的简单父类，它的 hash 是一个常量。它用简单的方式实现了 copyWithZone:。因为这个对象是不可变的（它没有实例变量，所以没有状态，也就没有可变状态），所以复制时只要增加引用计数，返回原来的对象就可以了。需要这个方法是因为字典会尝试复制 key，要确保在这个容器之外也不会改变它们（后面会详细讲到）。

代码清单 4.6：一个 isEqual: 的无效实现（取自 examples/isEqualFailure/dict.m）

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

@interface A : NSObject {}
@end
@interface B : A {}
@end
@interface C : A {}
@end
@implementation A
- (id) copyWithZone: (NSZone*)aZone { return [self retain]; }
- (NSString*)description { return [self className]; }
- (NSUInteger)hash { return 0; }
@end
@implementation B
- (BOOL) isEqual: (id)other { return YES; }
@end
@implementation C
- (BOOL) isEqual: (id)other { return NO; }
@end

```

```
21 int main(void)
22 {
23     id pool = [NSAutoreleasePool new];
24     NSObject *anObject = [NSObject new];
25     NSMutableDictionary *d1 = [NSMutableDictionary new];
26     [d1 setObject: anObject forKey: [B new]];
27     [d1 setObject: anObject forKey: [C new]];
28     NSMutableDictionary *d2 = [NSMutableDictionary new];
29     [d2 setObject: anObject forKey: [C new]];
30     [d2 setObject: anObject forKey: [B new]];
31     NSLog(@"d1: %@", d1);
32     NSLog(@"d2: %@", d2);
33     return 0;
34 }
```

两个子类，B 和 C，实现的 -isEqual: 方法也类似。一个永远返回 YES，另外一个永远返回 NO。在 main() 函数中，我们创建两个可变字典，给它们设置两个对象，分别用 B 和 C 的实例做 key。

运行该程序，结果如下：

```
$ gcc -framework Foundation dict.m &&./a.out
2009-01-07 16:54:15.735 a.out[28893:10b] d1: {
    B = <NSObject: 0x1003270>;
}
2009-01-07 16:54:15.737 a.out[28893:10b] d2: {
    B = <NSObject: 0x1003270>;
    C = <NSObject: 0x1003270>;
}
```

第一个字典只包含了一个对象，第二个包含了两个对象。这就是问题。在更复杂的程序中，一些 key 是来自外部的，你会花很多时间疑惑为什么一些实例会得到一样的 key，另外一些会不一样。

判断不同类的对象是否相同，用 hash 值会更灵敏，两个对象如果相等，就必须有一样的 hash 值。这两个类必须使用同样的 hash 函数，一个类有一些状态是另外一些没有的，就不能用来计算 hash。除此之外，对于所有对象，这两个方法都应该返回同样的常量值。这很简单，但是按照这一逻辑推论，所有的对象的 hash 必须返回 0，这距离理想状态还差得远。

第三个原则理论上最难满足，但实践中反而最容易。一个对象无法知道它是否在容器中。如果你用一个对象作为字典中的 key，或者插入 set，然后改变它，之后它的 hash 就有可能改变。如果 hash 没有改变，它可能会破坏第二个条件。

实践中，你只要避免修改容器中的对象，就可以排除这些问题。

4.4.2 原生容器

之前提到过，Foundation 库提供了一些 C 不透明类型的原生容器。在 10.5 中，它们也有了 isa 指针，这样通过 C 和 Objective-C 接口都可以使用它们。它们的最大优势是，可以直接储存进其他容器，而无须用 NSValue 实例包裹。大部分时候，如果要使用这些容器，一般会通过其 C 接口使用。因为效率更好，功能也更多。对象接口足够支持容器能够包含垃圾回收环境中的弱引用。如果没有使用垃圾回收或原生容器来存储其他值类型，C 接口会更加有用。

Foundation 库中定义的最简单的容器类型，除了数组和结构之类原生 C 类型之外，就是 NSHashTable 了。这是一个简单的 hash 表实现。其中存储一系列唯一的指针值。hash 表通过 NSHashTableCallBacks 结构创建，它定义了 5 个函数，用于和容器内的对象交互：

- hash 定义了一个函数，用来返回指针的 hash 值。
- isEqual 提供了比较函数，用来检查两个指针是否指向相等的值。
- retain 会在每个指针被插入 hash 表的时候被调用。
- release 是 retain 的相反操作，在对象从 hash 表中删除时被调用。
- describe 返回一个 NSString 来描述对象，调试时常用。

这些都符合 NSObject 声明的方法，可以通过预定义的 callback 集合调用 NSObjectHashCallBack 或 NSNonRetainedObjectHashCallBacks，把他们保存在 hash 表中。调用哪个方法，是根据插入对象到 hash 表时是否 retain 对象决定的。

Hash 表的模型是从 NSMapTable 扩展来的。NSMapTable 比 hash 表效率更高，它储存一对值，只用第一个元素来比较。Foundation 库为它定义了两类 callback：一类用于 key，另外一类用于值。

和其他 Cocoa 容器不同，这两个容器都可以用来存储非对象类型，包括适合存放指针的整型或指向 C 结构体以及数组的指针。

4.4.3 数组

Objective-C 是 C 的纯超集，可以访问标准 C 数组，但这种数组只是指向内存中 blob 的指针，用起来并不友好。OpenStep 定义了两种新的数组：mutable（可变）和 immutable（不可变）。NSArray 类实现了不可变的类型，它的子类 NSMutableArray 实现了可变的版本。

和 C 指针不同，它只用来存储 Objective-C 对象。如果需要一个其他对象的数组，可以直接使用 C 数组，也可以创建一个新的 Objective-C 类，包含一个所

需类型的数组。

NSArray 也是一个使用 class cluster 的例子。它的两个原生方法是：-count 和 -objectAtIndex:。它们和 NSString 中的同名方法大体相同，只是返回对象，而不是返回 unicode 字符。

和字符串一样，不可变的数组在存储方面比 C 数组更有效率。比如说，当你用一个数组中的某个范围创建另外一个数组时，（因为原始数组不会发生变化）只需存储这段范围的指针，不需要真正复制大量的元素。

Cocoa 数组也是对象，它可以做一些 C 中纯数据数组所不能完成的。最好的例子就是 -makeObjectsPerformSelector: 方法，它给数组中的每一个元素发送 selector。用这个方法可以写出非常简洁的代码。

在 10.5 中，Apple 增加了 NSPointerArray。其中可以存储任意指针（但不能是非指针类型）。和 NSArray 不同，它可以存储 NULL 值，并且在有垃圾回收时，可以被配置为允许弱引用。这样，NULL 值可以被用于任何在数组中销毁掉的对象。

可变参数初始化器

大部分 Cocoa 容器都有可变参数的构造方法和初始化方法。比如，+arrayWithObjects: 和 +dictionaryWithObjectsAndKeys:。它们的参数都是可变数量的，用 nil 作为结尾，返回实体数组或带有元素名称的字典。要快速创建在编译期间就知道元素数量的集合，这个方法很有用。

Cocoa 的数组非常灵活。这种数组可以用一个单独的方法在头部或尾部插入元素，所以不用修改就可以作为堆栈和队列。用数组做堆栈是非常有效率的。堆栈有三种操作：push，pop 和 top。其中 push 是把一个新对象增加到堆栈的顶部。NSMutableArray 的 -addObject: 就是这个操作。pop 操作是移除堆栈中最后加入的对象，就是 -removeLastObject 所做的。top 用来获得堆栈顶部的对象（对于数组来说是底部），NSArray 中的 -lastObject 方法就是这个操作。

用数组做队列的效率略差。队列中的对象，从一端插入，从另外一端移除。从数组尾部插入对象时成本很低，但从头部插入的时候代价昂贵。同样，从数组底部移除对象效率非常高，但从顶部移除效率就差很多。removeObjectAtIndex: 方法在删除第一个元素时，可能不会真的把数组中所有元素都往前移动。不过，NSMutableArray 是一个 class cluster，它的某些实现在删除第一个元素时效率更好，但这并没有办法保证。

4.4.4 字典

字典，有时候也叫关联数组，由 NSDictionary 实现。字典用于建立从一个对象到另外一个对象的映射，NSMapTable 更为友好，但只能用于对象。

字典中一般使用字符串作为 key，因为字符串符合 key 的所有需求。很多 Cocoa 代码中，用于字典 key 的字符串被定义为常量。在头文件中可以找到这样的代码：

```
extern NSString *kAKeyForSomeProperty;
```

之后在私有的实现文件中会有类似这样的代码：

```
NSString *kAKeyForSomeProperty = @"kAKeyForSomeProperty";
```

这个模式在 Cocoa 各处都能看到，同样也出现在各种第三方框架中。有时候也可以使用字面量，而不是 key，这样会在二进制下占用更多的空间，也会有点慢，所以这样做没什么优势。

如你所想，字典的可变版本是 NSMutableDictionary，它增加了原生方法 -setObject:forKey: 和 -removeObjectForKey:，以及一些方便方法。

字典时常被用来替代创建新的类。如果你需要存储一些结构数据，数据中不包含方法，那么用字典来存储又快又节约资源。可以调用一个方法就创建出字典，如下：

```
[NSDictionary dictionaryWithObjectsAndKeys:
    image, @"image",
    string, @"caption", nil];
```

这是一个可变参数构造函数，参数是带有一个 nil 作为结尾的对象列表。插入字典中的是一对对的 key 和对象。得到创建返回的字典之后，可以通过发送 -objectForKey: 消息来访问它们。

Cocoa 在很多地方使用到了它们。比如通知（Notification）也存储在字典中，通知的类型是已经定义好的特殊的 key 集合。这样未来要增加数据就很容易。

4.4.5 集合

就像 NSDictionary 是建立于原生类型 NSMapTable 之上的对象，NSSet 是建立在原生类型 NSHashTable 之上的对象。和数学中的概念一样，Cocoa 中的集合也是唯一对象们的无顺序容器。和数组不同的是，一个对象只能在集合中出现一次。

判断两个对象是否相等的规则非常简单。集合中的对象首先根据其 hash 值，散列在不同的桶中，对于小集合，就是它们 hash 值的某几位散列。当一个新对象插入时，集合首先根据它的 hash 值来找到正确的桶。然后通过 -isEqual:，判断桶中是否有和这个新对象相同的。如果没有相同的对象，就把这个新对象插入进去。

对于 NSSet 来说，这项工作只在用一个对象列表当做参数初始化的时候才会进行。NSMutableSet 允许在现有集合中加入对象，所以每次都会进行检查。如你所想，如果所有对象的 hash 值一样，就会非常慢（O(n)）。

除了基本的集合，OpenStep 还提供了 NSCountedSet。它是 NSMutableSet 的子类，也是可变的。和普通的集合不同，计数集合（也叫做背包）允许对象在容器中存在多个。和集合相同，它们也是无顺序的。也可以把它想象成没有顺序的数组，数组还允许“不同但是相等”的对象存在于同一容器中，计数集合只是给对象保持一个计数器。

从 10.3 开始加入了 NSIndexSet。它是一个整数集合，可以用于数组的索引或是其他用整数做索引的数据结构。在 NSIndexSet 内部，保存了一系列不重叠的范围。所以如果用这个类存储连续范围，效率就会非常好。

NSIndexSet 本身用处并不大。但通过 NSArray 的一些方法，比如 -objectsAtIndexes:，就会很有用，它返回一个包含了指定的索引元素的数组。用索引集合操作 NSArray 只需检查一次边界，比一个个查找速度快很多。

4.5 枚举类型

枚举 Foundation 库容器的传统办法，是通过 NSEnumerator。这个对象很简单，它响应 -nextObject 消息，返回下一个对象，如果没有下一个对象，就返回 nil。要用枚举器枚举一个容器，很简单，只需调用容器的 -objectEnumerator，然后对返回的枚举器循环发送 -nextObject，直到它返回 nil。

10.5 中，Apple 增加了一个快速枚举系统。它使用了新的 for 循环结构来控制容器，这是 Objective-C 2.0 的一部分。这时候甚至完全不需要直接使用枚举器。可以直接使用 NSArray 的 -makeObjectsPerformSelector: 之类的方法。代码清单 4.7 列出了给一个数组中所有对象发送一条消息的三种办法。

代码清单 4.7：给 Cocoa 中一个对象发送消息的三种办法（取自 examples/Enumeration/enum.m）

```
1  #import <Foundation/Foundation.h>
2
3  @interface NSString (printing)
4  - (void) print;
5  @end
6  @implementation NSString (printing)
7  - (void) print
8  {
9      fprintf(stderr, "%s\n", [self UTF8String]);
10 }
11 @end
12
13 int main(void)
14 {
15     [NSAutoreleasePool new];
16     NSArray* a =
17         [NSArray arrayWithObjects: @"this", @"is", @"an", @"array", nil];
18
19     NSLog(@"The Objective-C 1 way:");
20     NSEnumerator *e=[a objectEnumerator];
21     for (id obj=[e nextObject]; nil!=obj ; obj=[e nextObject])
22     {
23         [obj print];
24     }
25     NSLog(@"The Leopard way:");
26     for (id obj in a)
27     {
28         [obj print];
29     }
30     NSLog(@"The simplest way:");
31     [a makeObjectsPerformSelector: @selector(print)];
32     return 0;
33 }
```

第 20 ～ 24 行，是使用枚举器的方法。这方法很复杂，也容易出错，所以在 Étoilé 中，我们用 FOREACH 宏隐藏了这种模式（这也会让速度稍微快一点）。第 26 ～ 29 行，是一个简单点的方法，使用了快速枚举模式。这种方法代码简单，速度也快，确实很好。第 31 行是最后一种方法，更加简单，只有一行。如果你要发送的消息不止一个，或者消息中有超过一个的参数，那就不能用这个机制了。

运行代码，可以得到：

```
$ gcc -std=c99 -framework Foundation enum.m && ./a.out
2009-01-07 18:06:41.014 a.out[30527:10b] The Objective-C 1 way:
this
is
an
array
2009-01-07 18:06:41.020 a.out[30527:10b] The Leopard way:
this
is
an
array
2009-01-07 18:06:41.021 a.out[30527:10b] The simplest way:
this
is
an
array
```

4.5.1 利用高级消息枚举

进行枚举还有另外一种方法，是由 Marcel Weiher 提出的。这个机制叫做高阶消息（HOM），它使用了 Objective-C 的代理能力。它给容器类增加了 -map 这样的方法。当这些方法被调用时，会返回一个代理对象，把发送给它们的所有消息反弹给数组中的每个对象。

代码清单 4.8 是作为 catagory 加入 NSArray 的 -map 方法。代码来自 EtoileFoundation 框架，但删掉了 Étoilé 特殊的宏。

这个框架采用 BSD 许可证，可以按你的希望用于自己的项目。

代码清单 4.8: 使用高级消息实现的 map 方法实例（取自 examples/HOM/NSArray+map.m）

```
3  @interface NSArrayMapProxy : NSProxy {
4      NSArray * array;
5  }
6  - (id) initWithArray:(NSArray*)anArray;
7  @end
8
9  @implementation NSArrayMapProxy
10 - (id) initWithArray:(NSArray*)anArray
11 {
12     if (nil == (self = [self init])) { return nil; }
13     array = [anArray retain];
14     return self;
15 }
16 - (id) methodSignatureForSelector:(SEL)aSelector
17 {
18     for (object in array)
```

```
    {
        if([object respondsToSelector:aSelector])
        {
            return [object methodSignatureForSelector:aSelector];
        }
    }
    return [super methodSignatureForSelector:aSelector];
}
- (void) forwardInvocation:(NSInvocation*)anInvocation
{
    SEL selector = [anInvocation selector];
    NSMutableArray * mappedArray =
        [NSMutableArray arrayWithCapacity:[array count]];
    for (object in array)
    {
        if([object respondsToSelector:selector])
        {
            [anInvocation invokeWithTarget:object];
            id mapped;
            [anInvocation getReturnValue:&mapped];
            [mappedArray addObject:mapped];
        }
    }
    [anInvocation setReturnValue:mappedArray];
}
- (void) dealloc
{
    [array release];
    [super dealloc];
}
@end

@implementation NSArray (AllElements)
- (id) map
{
    return [[[NSArrayMapProxy alloc] initWithArray:self] autorelease];
}
@end
```

-map 方法本身相对比较简单，只是创建一个代理的实例，关联到数组上，返回。可以这样使用这个 category：

```
[[array map] stringValue];
```

这条语句会对数组中每个元素都发送 -stringValue 消息，然后返回包含结果的数组。当发送 -stringValue 消息给代理时，运行库会调用 -methodSignatureForSelector: 方法。这是为了找到方法的类型。这个方法的实现很简单，只是对数组中所有对象调用同名方法，直到找到有返回值的一个。

然后，-forwardInvocation: 方法被调用，它的参数是一个封装过的消息。这个

方法把消息发送给数组中的所有对象，然后再把结果加入到一个新的数组中。

和 -makeObjectsPerformSelector: 不同，使用高阶消息来给对象发送消息，可以有任意数量的参数。同样的机制也完全可用来实现容器的各种其他高级操作，比如归并或选择。

对比其他枚举机制，这种转发机制确实会慢一些，但它让源码表达出了更多的信息，这就很有吸引力。这种方法减少了重复代码，写起来程序更简单。HOM 在现代的 Smalltalk 实现中也有用到，但它最早的实现是在 Objective-C 中出现的。

高阶消息的应用不止枚举器这一种。它还有很多其他用途，包括在线程之间发送消息。我们会在第 23 章中看到如何在异步消息中使用它。

4.5.2 利用 block 枚举

OS X 10.6 为 C 语言家族增加了 block，上一章我们看过这个概念。Block 本身是很有用的，但它的真正威力是在被 Foundation 库整合之后。整合来自很多新的方法，比如 NSArray 中的

```
- (void)enumerateObjectsUsingBlock:
        (void (^)(id obj, NSUInteger idx, BOOL *stop))block;
```

这个 block 需要三个参数：一个对象，对象在数组中的索引位置，还有一个指向 boolean 值的指针，用来设置枚举是否应该停止。我们可以用 block 来重写同样的枚举例子，如下：

```
[a enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) { [obj print]; } ];
```

这里在 block 参数类型中内联了代码，可读性很差，把声明 block 和调用换行写会稍微好一些。在这个例子中，block 没有引用参数之外的任何东西，这样看来就和用函数指针差不多，除了 block 可以被声明为内联。

刚才看到的是一个简单的版本。更复杂的版本中有一个可选参数 NSEnumerationOptions。这是一个枚举类型，用来表示枚举如何处理转发，是反转进行还是并行进行。如果指定了 NSEnumerationConcurrent，这个数组就会产生一个新的线程，或通过线程池把枚举器访问分配到多个处理器上。对于大的数组或是长时间运行的 block，这是个好办法。

Foundation 库定义了另外两种 block 用于容器：测试器 block 和比较器 block。测试器 block 返回 BOOL 值，而比较器 block 是用 NSComparator typedef 得到的：

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

比较器 block 在任何要处理排序的地方都会被用到。可变的和不可变的数组、集合和字典都可以用比较器排序。它包括了一些十分复杂的方法，比如 NSDictionary 中的这个：

```
- (NSArray*)keysSortedByValueUsingComparator: (NSComparator)cmptr;
```

这个方法的参数是一个比较器 block，它用来确定两个对象的顺序。字典中的所有值都会调用这个方法，这个方法会根据值排序，然后返回一个排序之后的 key 数组。可以通过字典的 -valueForKey: 消息获得值。

测试器 block 一般用于过滤。和比较器不同，它不和一个具体的类型关联，因为每个类都定义了可以被 block 使用的参数。比如 NSIndexSet 的测试器 block 用 NSUInteger 做参数，而 NSDictionary 测试器使用 key 和 value 作为参数。

大部分容器类，包括 Foundation 库之外的，比如被 Core Data 框架管理的，支持 NSPredicate 作为一种过滤框架。在 OS X 10.6 中，可以从测试器 block 创建 NSPredicate。也可以创建 NSSortDescriptor 实例，它常用于结合比较器 block 和 Cocoa 绑定，或者用 -comparator 方法，把 NSSortDescriptor 对象转换成比较器 block。

4.5.3 支持快速枚举

你时常会需要实现自己的容器类，并把它们用于 for...in 循环。如果你的容器可以创建枚举器，就可以使用枚举器作为快速枚举的支持。但这样做稍微有点笨重，速度也慢。完全支持快速枚举，需要容器符合 NSFastEnumeration 协议，并实现以下方法：

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

要理解这个方法，必须理解 NSFastEnumerationState 结构。这个结构在 Foundation 库头文件 NSEnumerator.h 中定义，如代码清单 4.9 所示。

代码清单 4.9：快速枚举状态结构（取自 NSEnumerator.h）

```
19 typedef struct {
20     unsigned long state;
21     id *itemsPtr;
22     unsigned long *mutationsPtr;
23     unsigned long extra[5];
24 } NSFastEnumerationState;
```

第一次调用这一方法时，结构中的 mutationsPtr 字段需要被初始化。它必须是一个指向有效内容的指针，长度至少也要是 long 的长度。第一次调用时，调用者会自动缓存它指向的值，之后循环中每次指示器的动作都会比较它。如果这个值改变了，就会抛出一个异常。与此相反，NSEnumerator 没办法知道它正在枚举的集合是否发生了变化。

第二个参数指向调用者分配的缓冲区，第三个参数是缓冲区的大小。如果容器内部是用 C 数组存储对象的，可以通过把 state->itemsPtr 指向数组，直接得到元素的数量。否则，就要复制元素长度到 stackbuf，然后返回它复制的数字。编译器当前设置长度为 16，所以每次枚举 16 个元素，就发送一次消息。对比看来，如果使用枚举器，长度至少要 32（一个用于枚举器，一个是枚举器到容器）。这样就可以明白为什么 Apple 把它叫做“快速枚举”系统。

来看看如何才能在你自己的容器中支持快速枚举，我们创建两个新的类，如代码清单 4.10 所示。这两个类都符合 NSFastEnumeration 协议。一个是可变的，另外一个是不可变的。可变和不可变对象对快速枚举的支持稍微有一点区别。

代码清单 4.10：整型数组接口（取自 examples/FastEnumeration/IntegerArray.h）

```
1  #import <Foundation/Foundation.h>
2
3  @interface IntegerArray : NSObject<NSFastEnumeration> {
4      NSUInteger count;
5      NSInteger *values;
6  }
7  - (id)initWithValues: (NSInteger*)array count: (NSUInteger)size;
8  - (NSInteger)integerAtIndex: (NSUInteger)index;
9  @end
10
11 @interface MutableIntegerArray : IntegerArray {
12     unsigned long version;
13 }
14 - (void)setInteger: (NSInteger)newValue atIndex: (NSUInteger)index;
15 @end
```

这两个接口最应该注意到的区别是，可变版本中有一个实例变量 version。这个变量用来追踪在枚举过程中对象是否改变过。

代码清单 4.11 是不可变版本。前两个方法非常简单，只是初始化数组，允许访问值。这里的数组是一个简单的 C 缓冲区，通过 malloc() 创建。对象被销毁时，通过 -dealloc 方法释放。

第 28 行，这里的快速枚举器向实例变量返回了一个指针。这段代码支持局部枚举，只需调用者从全部容器中请求子集即可。现在编译器还不支持这个功能，

但这只是一个 Objective-C 方法，如果其他代码想要得到某些元素的值，你无法保证它不被这些代码直接调用。state 字段会被设置为调用者需要的第一个元素。通常使用时，这个值不是 0 就是具体的数量。项目的指针通过一些简单的指针算法，在实例变量数组中设置成正确的偏移量。

代码清单 4.11：一个简单的支持快速枚举的不可变整型数组（取自 examples/FastEnumeration/IntegerArray.m）

```
@implementation IntegerArray
- (id)initWithValues: (NSInteger*)array count: (NSUInteger)size
{
    if (nil == (self = [self init])) { return nil; }
    count = size;
    NSInteger arraySize = size * sizeof(NSInteger);
    values = malloc(arraySize);
    memcpy(values, array, arraySize);
    return self;
}
- (NSInteger)integerAtIndex: (NSUInteger)index
{
    if (index >= count)
    {
        [NSException raise: NSRangeException
                    format: @"Invalid index"];
    }
    return values[index];
}
- (NSUInteger)countByEnumeratingWithState: (NSFastEnumerationState*)state
                                  objects: (id*)stackbuf
                                    count: (NSUInteger)len
{
    NSUInteger n = count - state->state;
    state->mutationsPtr = (unsigned long *)self;
    state->itemsPtr = (id*)(values + state->state);
    state->state += n;
    return n;
}
- (void)dealloc
{
    free(values);
    [super dealloc];
}
@end
```

数组返回的最后一个值的索引位置被更新到 state 字段中。并且，for...in 循环会两次调用这个方法。第一次调用之后，state 字段会被设置成数量。第二次调用，n 的值会被设置为 0，循环中止。

注意，mutation 指针被设置为 self。把它解引用，会获得 isa 指针。这个类不

支持修改值，但一些其他的代码可能会把这个对象的类修改成可以修改的子类。在这个案例中，mutation 指针会被修改。这是非常不应该的，大部分情况，self 指针只是一个便捷的值，它是一个在循环期间保持有效和恒定的指针。

可变版本稍微有点复杂，如代码清单 4.12 所示。这个类增加了一个方法，允许设置数组中的值。注意第 42 行，version 是自增的，用来在数组变化时停止枚举。

这个类中的枚举方法把 mutationsPtr 指针设置为实例变量 version 的地址。循环结构中的这个值在代码生成时被缓存，每一个循环迭代都要和当前值比较，来检测是否有变化发生。

代码清单 4.12：一个简单的支持快速枚举的可变整型数组（取自 examples/FastEnumeration/IntegerArray.m）

```
39 @implementation MutableIntegerArray
40 - (void)setInteger: (NSInteger)newValue atIndex: (NSUInteger)index
41 {
42     version++;
43     if (index >= count)
44     {
45         values = realloc(values, (index+1) * sizeof(NSInteger));
46         count = index + 1;
47     }
48     values[index] = newValue;
49 }
50 - (NSUInteger)countByEnumeratingWithState: (NSFastEnumerationState*)state
51                                   objects: (id*)stackbuf
52                                     count: (NSUInteger)len
53 {
54     NSInteger n;
55     state->mutationsPtr = &version;
56     n = MIN(len, count - state->state);
57     if (n >= 0)
58     {
59         memcpy(stackbuf, values + state->state, n * sizeof(NSInteger));
60         state->state += n;
61     }
62     else
63     {
64         n = 0;
65     }
66     state->itemsPtr = stackbuf;
67     return n;
68 }
69 @end
```

因为可变数组的内部数组可能会被重新分配导致无效，所以我们把值复制出来，放到栈缓冲区中。这并非技术所需，容器不是线程安全的，所以不能通过可

能导致问题的方式访问数组，但在这里可以这样用，这也是一个展示如何使用栈缓冲区的例子。

栈缓冲区有固定的大小，通常是 16。第 56 行，我们找到在栈缓冲区数字比较小的位置，并且返回元素的数量。第 59 行，我们复制这些元素。之后 itemsPtr 指针被设置为栈缓冲区的地址。

使用栈缓冲区并不是必要的。不可变版本的数组不需要用到它。对于在内部没有使用数组的类，这是一个简单的办法，可以提供临时区域存放元素。

代码清单 4.13 很简单，是用来测试这两个类的。这段代码创建了两个整型数组，一个可变，一个不可变，通过 for...in 循环结构来使用快速枚举。

代码清单 4.13：测试快速枚举实现（取自 examples/FastEnumeration/test.m）

```
1  #import "IntegerArray.h"
2
3  int main(void)
4  {
5      [NSAutoreleasePool new];
6      NSInteger cArray[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
           15, 16, 17, 18, 19, 20};
7      IntegerArray *array = [[IntegerArray alloc] initWithValues: cArray
8                                                           count: 20];
9      NSInteger total = 0;
10     for (id i in array)
11     {
12         total += (NSInteger)i;
13     }
14     printf("total: %d\n", (int)total);
15     MutableIntegerArray *mutablearray =
16         [[MutableIntegerArray alloc] initWithValues: cArray
17                                               count: 20];
18     [mutablearray setInteger: 21 atIndex: 20];
19     for (id i in mutablearray)
20     {
21         total += (NSInteger)i;
22     }
23     printf("total: %d\n", (int)total);
24     for (id i in mutablearray)
25     {
26         total += (NSInteger)i;
27         printf("value: %d\n", (int)(NSInteger)i);
28         [mutablearray setInteger: 22 atIndex: 21];
29     }
30     return 0;
31 }
```

注意，在这种循环中的元素类型必须是 id。这是类型检查器唯一能接受的。编译器不会发送任何消息到返回的对象，所以它们的大小就和返回的 id 相同。

第 28 行，我们在循环中修改了容器中的内容。这种情况正是快速枚举结构中 mutationsPtr 指针需要检测的。如果我们的实现正确，就会抛出异常。运行程序，会看到以下情况：

```
$ gcc -framework Foundation *.m && ./a.out
total: 210
total: 441
value: 1
2009-02-21 16:24:56.278 a.out[4506:10b] *** Terminating app
due to uncaught exception 'NSGenericException', reason:
'*** Collection <MutableIntegerArray: 0x1004bb0> was mutated
while being enumerated.'
```

我们在这里不需要明确抛出异常。只要修改 version 实例变量就会这样。注意，这个异常是在第一个循环迭代之后出现的，就是说，前 16 个值一次返回了。这就是为什么 mutationsPtr 是一个指针，而不是一个值。如果它只是一个简单的值，那么每次调用时都需要设置 version 实例变量的值，循环在下一次调用之前，也无法检测 mutation 的值。因为它是指针，它就可以在循环的每一次迭代中进行解引用，测试成本也很低，所以数组被改变之后，会在调用者试图载入值的时候发现。

因为在修改数组的内容之前，我们先修改 version 的计数，这就保证了 mutation 始终会被捕捉到。如果在类中增加任何其他的 mutation 方法，只需记得在开始的时候增加 version++，就可以让它工作正常。

4.6 属性列表

属性列表（plist）是 OPENSTEP 和 OS X 的常见功能。由于表单形式的简单和工具的简便，很多系统都实现了对属性列表的支持。

原始的 NeXT 属性列表，是简单的 ASCII 格式，它用单个字符来代表不同容器类型的边界。这有很多局限。它很难扩展，相对也难于嵌入其他格式中。为了解决这些问题，OS X 增加了一种 XML 格式的属性列表。它可以比过去的格式存储更多的类型，它具有属性的名称空间，可以被嵌入进任何 XML 格式中。

不幸的是，XML 是非常罗嗦的格式，解析 XML 成本很高，存储 XML 也需要很多空间。所以，Apple 又加入了第三种格式，是一种专有的（非文档的）二进制编码。这种格式解析速度快，占用空间也小。

NeXT 的格式在 OS X 中已经基本停止使用了，不过还有不少命令行工具仍然使用它，因为这种格式更适合人类阅读。XML 格式主要用于交换，二进制格式用于本地存储。表 4.1 对比了 XML 格式和 OpenStep 属性表格式。你可以看到在新 XML 格式中数组和字典的表现方式是多么冗长。

另外，GNUstep 扩展了 OpenStep 格式，令其可以存储 NSValue 和 NSDate 对象，所以，虽然缺少了一些交互性，但 GNUstep 的 OpenStep 式的属性表和 XML 属性表具有一样的表现力。

plutil 工具可以在 XML 格式和二进制格式之间进行转换。还可以从原始 NeXT 格式转换出数据，但无法转换回去。因为这个格式表现力比新格式差——例如，它无法保存 NSDate 对象——所以，工具无法保证这样的属性表会被安全转换。转换时，-lint 选项会强制检查 plist 文件并报告错误。在需要手工编辑属性表的时候这非常有用。

4.6.1 序列化

属性列表的主要用途，就是提供一种存储容器中数据的格式，这种格式能方便地读 / 写。表 4.1 是 Cocoa 对象和属性列表中各种元素的对应关系，但是，因为只支持了相对较少的通用数据类型，所以 Cocoa 不是唯一能操作 plist 的系统。

表 4.1 存储在 OpenStep（NeXT）和 XML（Apple）属性表中的数据类型

Type	Cocoa Class	NeXT	XML
String	NSString	"a string"	<string>a string</string>
Boolean	NSNumber	N/A	<true /> or <false />
Integer	NSNumber	12	<integer>12</integer>
Floating Point	NSNumber	12.42	<real>12.42</real>
Date	NSDate	N/A	<date>2009-01-07T13:39Z</date>
Binary data	NSData	<666f6f>	<data>fooZm9v</data>
Arrays	NSArray	("a")	<array> <string>a</string> </array>
Dictionaries	NSDictionary	{"key" = "value";}	<dict> <key> <string>key</string> </key> <value> <string>value</string> </value> </dict>

Core Foundation 库支持所有数据类型，CFLite 也是。就是说，属性列表可以用于没有使用 Cocoa 的简单 C 程序，甚至连 Core Foundation 也不需要。还有一些其他的库也可以用来解析属性列表。NetBSD 的 proplib 是一个 BSD 许可的 C 库，它可以用来操作 XML 属性列表，无须任何 Apple 的代码。

就是说，Cocoa 可以用这种格式来创建配置文件或是简单的数据文件，非 Cocoa 的应用程序可以通过各种工具包很容易解析和使用这些文件。很多 OS X 的核心配置信息是存储在属性列表中的，它们在任何 Cocoa 应用程序启动之前就要被用到了。

Cocoa 容器可以写入属性列表文件，并且通过一次调用来读出它们。代码清单 4.14 是一个例子。这个程序创建了一个简单的数组，将其写入到文件中，然后重新读回到另外一个数组中。通过记录两个数组，发现它们是相等的。

代码清单 4.14：保存数组到属性列表（取自 examples/PropertyList/plist.m）

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

int main(void)
{
    [NSAutoreleasePool new];
    NSArray *a = [NSArray arrayWithObjects:@"this", @"is", @"an", @"array",
         nil];
    [a writeToFile:@"array.plist" atomically:NO];
    NSArray *b = [NSArray arrayWithContentsOfFile:@"array.plist"];
    NSLog(@"a: %@", a);
    NSLog(@"b: %@", b);
    return 0;
}
```

运行程序时，如果工作正常，就会看到下面的情况：

```
$ gcc -framework Foundation plist.m && ./a.out
2009-01-07 19:13:15.299 a.out[34155:10b] a: (
    this,
    is,
    an,
    array
)
2009-01-07 19:13:15.300 a.out[34155:10b] b: (
    this,
    is,
    an,
    array
)
```

这是一个只包含了字符串的简单例子，但同样的代码可以在包含了任何类型的数组上使用，把它们存入属性列表。

这样的基本功能已经足够大部分用户需要了，至于更多高级的需求，就需要用到 NSPropertyListSerialization 了。这个类可以用来校验属性列表，还可以不用文件，而是使用内存中的 NSData 对象来保存和读入。之前提到过的 plutil 工具就是这个类的简单包装。

要用 NSPropertyListSerialization 创建属性列表，就要用到下面这个方法：

```
+ (NSData *)dataFromPropertyList: (id)plist
                          format: (NSPropertyListFormat)format
                errorDescription: (NSString**)errorString;
```

plist 对象是要被转换成属性列表的对象。format 可以是表示 XML 格式的 NSPropertyListXMLFormat_v1_0，或者表示二进制属性列表的 NSPropertyListBinaryFormat_v1_0。NSPropertyListFormat 枚举中，也定义了 NSPropertyListOpenStepFormat，但只有在读 OpenStep 属性列表时才有用。OS X 不再写出这种格式了。最后一个参数是指向字符串的指针，用于返回错误信息。

这种要求一个指向字符串的指针作为参数来返回错误的方法并不常见。这主要取决于它的年代。这种方法是 OS X 10.2 引入的。在此之前，异常始终用于返回错误，或者是产生软错误返回 BOOL 的方法。在 10.2.7（或更早的版本但安装了 Safari）中 Apple 引入了 NSError 类。一个指向这个类实例的指针的指针，它经常在最后一个参数中传递，并且在返回时设置为非空，但这个方法还很新，用它为时过早。

要反序列化属性列表，可以用下面这个便利方法：

```
+ (id)propertyListFromData: (NSData*)data
          mutabilityOption: (NSPropertyListMutabilityOptions)opt
                    format: (NSPropertyListFormat*)format
          errorDescription: (NSString**)errorString
```

这里大部分参数都是一样的。现在 format 是输出参数了，用来设置属性列表的格式，它会自动检测。新的参数 opt，定义了要被反序列化的对象是否可变。属性列表不存储是否可变的选项（NSString 和 NSMutableString 是通过同样的方式保存的），所以需要在反序列化时指定。你可以指定是否所有元素都可变（NSPropertyListMutableContainersAndLeaves），或者只有容器可变（NSPropertyListMutableContainers），或者都不可变（NSPropertyListImmutable）。

4.6.2 用户默认值

所有系统都会碰上如何存储应用程序的偏好设置这个问题。这个问题有众

多解决方案，从独立的配置文件到集中的注册表。OS X 采用了一条中间道路。每个应用程序都和一个属性列表关联，这个属性列表中存储着一个字典。通过 NSUserDefaults 类可以访问它们，在应用程序中由于这些值改变之后的通知部分也由这个类负责。

这种做法最大程度结合了两种方法的优点。你可以把它当做一个系统维护的用户默认数据库。默认的命令行工具可以浏览和修改任何应用程序的默认值。因为它们只是属性列表，你可以在默认系统之外修改它们，或将其从不必要应用程序中删除。

要从应用程序中获得用户默认值，可以这样做：

```
[NSUserDefaults standardUserDefaults];
```

这个方法返回一个单件对象。你可以从应用程序中多次调用这个方法，但始终会使用同一个对象。它会在内存中维护一个默认值的副本，以后会和硬盘存储同步。也可以通过发送 -synchronize 消息来强制同步。

共享的默认值对象和 NSMutableDictionary 结构相似。它有 setObject:forKey: 和 objectForKey: 方法，分别用来设置和获得对象。另外还有一些便利方法，比如 boolForKey: 用来获得装箱过的值，进行拆箱并且返回一个 bool 值。

因为用户默认值还支持 key-value coding（KVC），所以可以通过标准的 KVC 方法来访问默认值。特别还有一个 valueForKeyPath: 方法，很适合用于默认值中存放了字典的情况。可以通过一次调用来获得内嵌的键：

```
[[NSUserDefaults standardUserDefaults] valueForKeyPath: @"dict.key"];
```

只要用户的默认值系统中包含名为“dict”的字典，字典中有叫做“key”的键，就可以得到对应的值。你可以用这种方法，通过更长的键路径来获得更深层嵌套的字典值。不过不幸的是，对应的 setValue:forKeyPath: 方法会导致运行时异常。

默认值和命令行

用户默认值系统在一系列默认的域中搜索值。NSArgumentDomain 是一个非常容易被忽视的类。对于从命令行获得值，这是一个非常有用的方式。如果要在命令行启动一个 Cocoa 应用程序，可以通过命令行中得到的值来覆盖用户的 -default 值。也可以通过这种方式快速方便地给使用 Foundation 库的命令行工具定义参数选项，只需在默认值中加载时，把命令行选项的名称作为 key 传入就可以了。

默认值中使用可变容器有一些困难。所以没有直接支持，需要的时候可以从容器创建一个副本，然后重新插入回容器。一般是这样几个步骤：

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
NSMutableDictionary *d = [[defaults dictionaryForKey: aKey] mutableCopy];
// Some operations on d
[defaults setObject: d forKey: aKey];
```

这有点笨拙，所以一般会包装成方法。特别要记住，category 允许你给 NSUserDefaults 增加方法。如果应用程序存储了一个叫做 sound 的字典，你可能会考虑增加一个 -setSound:forAction: 之类的方法，用来设置默认值中 sound 方法的条目。

用户默认值只能支持那些可以被属性列表保存的对象，而 NSColor 对象是一个明显的例外。Apple 建议在 NSUserDefaults 增加一个 category，通过 NSArchiver 机制保存其他对象。

NSArchiver 可以对实现了 NSCoding 机制的对象进行序列化。如果你要保存的对象是默认实现，NSArchiver 可以把它转换为 NSData 实例，并且可以用 NSUnarchiver 恢复它。因为默认值系统可以操作 NSData 实例，这样就提供了一个把对象存入默认值的机制。

通常还有一些更轻量级的方法。一些对象，比如 NSURL，容易当做字符串存储。代码期待 4.15 是一个简单的 catagory，可以把 URL 当做字符串存储到默认值。如果你的对象已经实现了 -stringValue 和 -initWithString: 方法，那么使用这个机制会比 NSCoding 更简单。

代码清单 4.15：一个可以把 URL 存入默认值的 category（取自 examples/URLDefaults/urldefaults.m）

```
3  @implementation NSUserDefaults (NSURL)
4  - (void) setURL: (NSURL*)aURL forKey: (NSString*)aKey
5  {
6      [self setObject: [aURL absoluteString] forKey: aKey];
7  }
8  - (NSURL*) URLForKey: (NSString*)aKey
9  {
10     return [NSURL URLWithString: [self stringForKey: aKey]];
11 }
12 @end
```

你的对象可以采用任意一种机制，区别只是复杂度不同。

4.7 和文件系统交互

某种意义上，和文件系统交互是最重要的问题。在大部分 UNIX 类系统中，包括 OS X，只有文件系统能提供持久化存储。用户默认值，只是文件系统中很小一部分的高级接口，提供了通过字典形式接口访问特定文件的方法。

如何和文件系统交互，取决于你手边的任务。Cocoa 提供了一些方法把文件暴露为 UNIX 风格的字节流，或是某些结构数据。可以根据需求选择所需。

4.7.1 Bundle

Bundle 是 OS X 中的重要部分。它过去用在 NeXT 系统中，现在逐渐替代了早期 Mac OS 系统中使用过的资源分叉结构。Bundle 不需要特殊的文件系统支持，这是很大优势。

OS X 中应用程序的 bundle 中，可以有其他资源或代码。在 NeXT 系统中，应用程序 bundle 通常会保存在不同平台上运行的不同版本，你可以把一个 .app 共享在 NFS 上，然后从 OPENSTEP、Solaris 或其他能支持的系统中运行它。在现在的 OS X 中还能看到遗留的痕迹，应用程序 bundle 中，二进制文件放在 Contents/MacOS 目录下。理论上，你可以增加其他平台的二进制文件，尽管 Apple 的工具现在已经不支持这样做了。

在 OS X 发布（同时也是被命名）之前，开发中的传统 MacOS 继任者被叫做 Rhapsody。Apple 当时宣布了三个“盒子”，其中两个最终成了 OS X 的一部分。蓝盒子是 MacOS 的虚拟化兼容层，在早期版本的 OS X 上被叫做 Classic，它没有出现在 Intel Mac 上。黄盒子是 OpenStep 环境，后来更名为 Cocoa。最后一个红盒子，和 WINE 类似，是 OS X 上的 Wiindows 环境，不过最后没有出现在最终产品中。同时规划了一个 Windows 版本的黄盒子，基于 NeXT 的 OPENSTEP Enterprise（OSE），其中包含 Project Builder 和 Interface Builder，让开发者可以在 Windows 上开发 Cocoa 应用程序。

看起来，Apple 仍然在维护着一个继承自 Windows 黄盒子的版本，并用来移植 Safari 之类的程序到 Windows，不过 Apple 并没有让这些应用程序使用 bundle 架构。虽然红盒子没有发布，但可以认为是 OS X 保留了未来可以运行不同格式可执行文件的可能性。

OS X 和 OPENSTEP 一样，使用 Mach-O 二进制格式，这种格式可以在同一个二进制文件内支持多种执行格式（当字节序相同时，共享常量和数据）。这样比给每个版本一个独立文件更有效率，可以在一个文件中支持 Intel 和 PowerPC，以及 32 位和 64 位可执行文件。NeXT 把这个叫做胖二进制（fat binary），Apple 改称为通用二进制（universal binary）。

因为应用程序是打包的，每一个应用程序至少有一个包可以用来读入资源。在一个非常简单的应用程序中这是自动完成的。应用程序启动和连接到应用程序的 delegate 和其他对象时，main nib 会被载入。

其他资源可以在应用程序中通过 NSBundle 类载入。通常，要用来交互的每一个 bundle 都会有一个 NSBundle 类实例。应用程序的 bundle 可以这样获得：

```
[NSBundle mainBundle];
```

这里要小心。在将来的某些时候，你可能会确定你的类非常有用，打算重用它。这时候，你会把这个类移出来，变成一个框架，其中包含了所有可能要载入的资源（框架是另外一种 bundle，包含了可以载入的库、头文件和资源）。当从你的类中获取 main bundle 时，你会得到链接框架的应用程序 bundle，而不是框架的 bundle。如果要获取类的 bundle 中的资源，这就不是你想要的。你应该使用：

```
[NSBundle bundleForClass: [self class]];
```

这样相对慢一些，所以最好在类的 +initialize 方法中完成它，然后缓存在静态变量中，像这样：

```
static NSBundle *frameworkBundle;
+ (void) initialize
{
   frameworkBundle = [[NSBundle bundleForClass: self] retain];
}
```

在真实代码中，还需要再做一个包装，进行一个检查，确保只被正确的类调用，和在第 3 章中提到过的方法一样。因为这是类方法，只需传递 self 作为参数，而不是 [self class]。也可以用 +bundleWithIdentifier 方法，它一般会更快。这个方法需要提供一个 identifier 参数，用来载入 bundle。Bundle identifier 是 bundle 属性列表中的 CFBundleIdentifier 键。

得到了 bundle 之后，就可以从中载入资源了。这是一个两步的过程。首先找到资源的路径，用下面的方法：

```
- (NSString*)pathForResource: (NSString*)name
                      ofType: (NSString*)extension
                 inDirectory: (NSString*)subpath
             forLocalization: (NSString*)localizationName
```

这个方法有两个包装过的版本，其中最后的参数填充了默认值。最简单的一个只需要前两个参数，它使用用户偏好的本地化语言，在 bundle 中的顶级资源目录中读取。

如果要读入 bundle 中指定类型的全部资源，可以用下面这个方法，它返回一个数组：

```
- (NSArray*)pathsForResourcesOfType: (NSString*)extension
                        inDirectory: (NSString*)subpath
```

这个方法用指定的位置，查找某个类型的全部资源，比如，在 bundle 的 Resources 目录下的 theme 目录中的所有 png 文件。

除了资源，代码也可以从 bundle 中读入。代码清单 4.16 是一个简单的框架载入器。因为框架是另外一种布局明确的 bundle，标准的 bundle 载入代码也可以用于它们。

这个例子来自 Étoilé 的语言包（LangaugeKit），它用来加载脚本，编译所需程序到指定的运行中的框架，而无须把每个脚本所需的框架都链接进来。

这个例子展示了不少 Cocoa 特性。首先是文件管理器，下一节我们会看到它。第 24 行用到了它，用来检测给出的路径下是否存在框架。如果存在，第 27 和 28 行就加载其中的代码。

第 11 行的函数很有用，但常常被忽视，它可以避免很多实例中的路径硬编码问题。第 19 行展示了如何结合 NSString 的路径使用，通过在正确的路径后追加 Framework 目录，组合框架名称作为路径，然后是 .framework 扩展名。在 OS X 上也可以用 -stringWithFormat: 完成这个工作，但移植到其他系统时，路径的格式未必会一样。

代码清单 4.16：简单的框架载入器（取自 examples/Loader/simpleLoader.m）

```
7  @implementation SimpleLoader
8  + (BOOL) loadFramework: (NSString*)framework
9  {
10     NSFileManager *fm = [NSFileManager defaultManager];
11     NSArray *dirs =
12         NSSearchPathForDirectoriesInDomains(
13             NSLibraryDirectory,
14             NSAllDomainsMask,
15             YES);
```

```
16      FOREACH(dirs, dir, NSString*)
17      {
18          NSString *f =
19              [[[dir stringByAppendingPathComponent: @"Frameworks"]
20                  stringByAppendingPathComponent: framework]
21                      stringByAppendingPathExtension: @"framework"];
22          // Check that the framework exists and is a directory.
23          BOOL isDir = NO;
24          if ([fm fileExistsAtPath: f isDirectory: &isDir]
25              && isDir)
26          {
27              NSBundle *bundle = [NSBundle bundleWithPath: f];
28              if ([bundle load])
29              {
30                  NSLog(@"Loaded bundle %@", f);
31                  return YES;
32              }
33          }
34      }
35      return NO;
36  }
37  @end
```

4.7.2 工作区和文件管理

Cocoa 提供了两种和文件系统交互的方法：NSFileManager 和 NSWorkspace。后者是 AppKit 的一部分，提供了更高级的接口。NSWorkspace 类在后台进行文件操作，完成时发送通知，而 NSFileManager 是同步工作的。两个类都是单件模式的，在一个应用程序中只有一个实例。

我们在代码清单 4.16 中看到了文件管理器的一个功能，用 -fileExistsAtPath:isDirectory: 方法，可以检查文件是否存在。这里的第二个参数是 BOOL 指针，用来设置文件是否在目录中被发现。

文件管理器支持大部分普通文件操作，比如复制、移动，文件和目录的链接。它也可以用来枚举目录内容和比较文件。大多数 NSFileManager 的函数在 NSWorkspace 里面以单独的方法形式暴露出来：

```
- (BOOL)performFileOperation: (NSString*)operation
                      source: (NSString*)source
                 destination: (NSString*)destination
                       files: (NSArray*)files
                         tag: (NSInteger)tag
```

这个方法以 source 和 destination 目录作为参数，files 参数是一个数组。它可以处理移动、复制、链接、删除、移动到垃圾箱这几种操作，并且设置整数 tag

来说明操作是否成功。

大部分 NSWorkspace 函数处理更高级的文件操作。NSFileManager 使用 UNIX 风格的字节流处理文件，而 NSWorkspace 以用户级别的抽象来处理文件，展示文档或应用程序。比如 openFile: 方法，这个方法用默认应用程序打开特定文件，一般用于开发命令行打开工具。

低级文件管理器的方法非常容易使用。代码清单 4.17 是一个简单的文件复制工具。它通过用户默认系统来读入命令行参数，然后用文件管理器来复制指定文件。

代码清单 4.17：简单的文件复制工具（取自 examples/FileCopy/FileCopy.m）

```
1  #import <Foundation/Foundation.h>
2
3  int main(void)
4  {
5      [NSAutoreleasePool new];
6      NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
7      NSString *source = [defaults stringForKey: @"in"];
8      NSString *destination  = [defaults stringForKey: @"out"];
9      NSFileManager *fm = [NSFileManager defaultManager];
10     [fm copyPath: source
11           toPath: destination
12          handler: nil];
13     return 0;
14 }
```

注意，演示代码中没有检查输入文件是否存在，也没有检查输出是否是有效位置。文件管理器会调用 delegate 方法来处理错误，但是第 12 行中没有设置 handler，所以也就没办法进行错误检查了。handler 不是必须实现的，它只是允许你追踪操作过程和确定处理过程中是否出错。这个方法的返回值是布尔值，用来判断复制是否成功。这个工具这样运行：

```
$ gcc -framework Foundation FileCopy.m -o FileCopy
$ ./FileCopy -in FileCopy -out CopyOfFileCopy
$ ls
CopyOfFileCopy FileCopy        FileCopy.m
```

注意文件管理器会自动处理相对路径。无论文件管理器的 -currentDirectoryPath 返回什么，都会被视为相对路径。可以通过发送 -changeCurrentDirectoryPath: 消息给文件管理器，来切换运行中程序的工作目录。对于命令行程序来说，工作目录比对于图形界面程序更重要。命令行程序从 shell 继承当前工作目录。当前目录这个概念对于从 Finder 或 Dock 中调用的应用程序是没有意义的。

从 10.5 开始，Apple 开始使用统一类型标识（UTI）系统来确定文件类型。UTI 是分层排列的类型。NSWorkspace 可用来映射文件扩展名和 UTI。

4.7.3 使用路径工作

NSString 提供了大量有用的方法用于文件系统路径。这些方法可以把路径分解开，不管路径的表现形式如何，最后都能拆开成不同部分。

在 UNIX 平台上，一个波浪号通常是用户主目录的快捷方式。对于这个符号，可以调用 NSHomeDirectory()，但用户通常会在这个符号之后连接其他字符串，并期望能工作。在 Finder 的 Go 菜单中，选择 Go to Folder，然后输入“~/Documents”可以打开一个新窗口，进入主目录下的 Documents 文件夹。

NSString 类为包含波浪号的字符串提供了便利方法。如果给一个字符串发送 -stringByExpandingTildeInPath 消息，之后将会获得包含绝对路径的字符串，其中没有波浪号。另外一个方法用到的比较少，就是给包含完整路径的字符串发送 -stringByAbbreviatingWithTildeInPath 消息，如果这个路径是在用户主目录下，它就会被折叠成用波浪号开头的路径形式。

和文件系统交互时，经常需要把路径分解成三个部分：文件名，文件扩展名，文件所在路径。用 NSString 都可以做到，像下面这样：

```
NSString *fullPath = @"/tmp/folder/file.extension";
// ext = @"extension";
NSString *ext = [fullPath pathExtension];
// file = @"file";
NSString *file = [[fullPath lastPathComponent]
    stringByDeletingPathExtension];
// dir = @"/tmp/folder";
NSString *dir = [fullPath stringByDeletingLastPathComponent];
```

同样也有方法把几个单独的部分重新组成路径，包括合并各部分和设置扩展名。在开始自己写代码做这类事情之前，首先应该确认一下 NSString 是不是已经提供了你所需要的。

4.7.4 文件访问

通过 NSFileManager 可以和文件交互，通过 NSWorkspace 可以在其他应用程序中打开文件，但都不是访问文件内容。

你可以使用 C 库和 POSIX 函数来完成，但都不是很方便。Cocoa 把它们包装成了 NSFileHandle 类。这个类包装了 open() 函数返回的文件句柄。它有四个单件实例，分别代表 C 的标准输入、输出、错误流和一个丢弃所有写入数据的占位符。文件句柄可以通过读、写、更新这几个构造方法来创建。

NSFileHandle 可以在任何获得 C 文件句柄的地方使用，像下面这样初始化：

```
- (id)initWithFileDescriptor: (int)fileDescriptor
              closeOnDealloc: (BOOL)flag
```

其中 fileDescriptor 可以是任意 C 文件描述符，比如用 open() 或 socket() 返回的。最后的对象支持读或是写，是依赖于底层的文件描述符的。对于程序中有很多地方用到这个文件的情况，把 flag 参数设置为 YES 是个简单的办法，可以确保在正确的位置关闭文件。之后只要程序中还有某些部分通过对象使用这个文件描述符，它就会一直保持打开，直到不再用了才关闭。

如果只是要从文件中读入数据，可以从文件创建 NSData 对象，用如下方法：

```
+ (id)dataWithContentsOfFile: (NSString*)path
                     options: (NSUInteger)mask
                       error: (NSError**)errorPtr
```

这个方法会用所指向文件的内容创建一个新的 NSData 对象，如果失败，就设置指向 NSError 实例的 errorPtr。其中 mask 参数可以设置两个选项：NSMappedRead 和 NSUncachedRead。前一个用 mmap() 替代读文件操作。如前文讨论过的，如果你知道将来这个文件不会被修改，这就是个好办法，比如对于 bundle 中的只读资源就很合适。如果系统的内存不足，成本比较低的方式是释放内存数据，之后需要时再从原始文件中读入，数据读入之后就算没有变化，也会被写入交换文件。第二个选项，是让数据不经过操作系统缓存。如果你知道只需读入数据一次，之后就丢弃掉，这样可以提高性能。这样做用的内存更少，但会造成更多的磁盘访问。

NSData 也可以用来把数据写入文件。对于小文件的输出，用 NSMutableData 在内存中创建文件，然后用 writeToFile:atomically: 或 writeToURL:atomically: 方法，很容易输出到文件。这两个方法的第二个参数都是 BOOL 类型，如果设置为 YES，就会先写到临时文件，然后把这个临时文件改名到正式文件，以确保磁盘一致性。

4.8 通知

通知（Notification）是 Cocoa 中松耦合的另外一个例子。它在事件和事件句柄之间提供了一个中间层。每个对象不再保存一个需要接受到事件的对象列表，所有事件都通过通知中心（NSNotificationCenter 类）。对象可以通过指定名称或指定对象（或者同时使用）请求通知。当指定对象发送命名通知，observer 会收到带有 NSNotification 参数的消息。

使用这个机制，很容易就可以把应用程序的几个部分联系起来，而且不需要硬编码。Foundation 和 AppKit 是这种分层的一个很好的例子。Foundation 定义了通知，在 AppKit 中大量使用。许多 view 对象在发生互动时都会发送通知，还要传递消息到它们的 delegate。用户交互的一部分变化时，可以给多个对象发送通知，只需写很少的代码。

默认情况下，通知是同步发送的。一个对象发送通知到通知中心，通知中心把它传递给所有 observer。有时候这不是所期望的行为。对于异步通知传递，集成在运行循环中的 NSNotificationQueue 类，允许通知被延迟发送，直到运行循环空闲或是运行循环的下一次迭代。

4.8.1 请求通知

一般意义上，对于通知你会做两件事：发送和接收。接收一个通知分两步，第一步，告诉通知中心，你想收到什么，然后等着就行了。

代码清单 4.18 是一个用来接收通知的简单类。它实现了一个方法 receiveNotification:，这个方法接收通知对象，同时会记录下通知的名称和 userinfo 字典中的值，是 userinfo 字典让通知变得如此灵活。因为它是字典，接收者甚至不需要知道其中的键。可以通过枚举每个键，对字典中所有对象做一些事情，如果是不知道的键，简单忽略掉即可。

对象创建之后，默认的通知中心将来自任何对象的名为“ExampleNotification”的通知投递给 receiveNotification: 方法。这里也可以指定只接受来自特定对象的消息。在这个案例中，还可以把 name: 参数设置成 nil，这样可以得到来自这个对象的所有消息。

代码清单 4.18：接收通知的对象（取自 examples/Notifications/notify.m）

```
3  @interface Receiver : NSObject {}
4  - (void) receiveNotification: (NSNotification*)aNotification;
5  @end
6
7  @implementation Receiver
8  - (id) init
9  {
10     if (nil == (self = [super init]))
11     {
12         return nil;
13     }
14     // register to receive notifications
```

```
15      NSNotificationCenter *center =
16          [NSNotificationCenter defaultCenter];
17      [center addObserver: self
18                 selector: @selector(receiveNotification:)
19                     name: @"ExampleNotification"
20                   object: nil];
21      return self;
22  }
23  - (void) receiveNotification: (NSNotification*)aNotification
24  {
25      printf("Received notification: %s",
26              [[aNotification name] UTFString]);
27      printf("Received notification: %s",
28          [[[aNotification userInfo] objectForKey: @"message"] UTFString]);
29  }
30  - (void) dealloc
31  {
32      NSNotificationCenter *center =
33          [NSNotificationCenter defaultCenter];
34      [super dealloc];
35  }
36  @end
```

关于处理通知，还有个非常重要的部分没有提到。必须要记得在对象销毁时，发送 removeObserver: 消息到消息中心。所以，最佳实践是确保对象注册时候的 observer 始终为 self。这样就容易在正确的时间调用 removeObserver:，只需把调用放进 -dealloc 方法。在垃圾回收环境，通知中心应该保持和 observer 的弱引用，当它们不再有效的时候，就会自动被移除。

在这个简单的例子中，通知是用字符串确定的。创建公开通知的更通用做法是用共享的全局对象，它们在一个文件中初始化，并被定义在头文件中。

4.8.2 发送通知

发送消息更是简单。代码清单 4.19 展示了一个简单的对象，它响应 sendMessage: 消息，发送通知。其参数中的字符串会被插入 userInfo: 字典，用通知投递出去。

它和代码清单 4.18 中的 Receiver 类是对照的。这两个类可以互相通信，但没有任何直接引用。

代码清单 4.19：发送通知（取自 examples/Notifications/notify.m）

```
38 @interface Sender : NSObject {}
39 - (void) sendMessage: (NSString*)aMessage;
40 @end
41 @implementation Sender
42 - (void) sendMessage: (NSString*)aMessage
43 {
44     NSNotificationCenter *center =
45         [NSNotificationCenter defaultCenter];
46
47     NSDictionary *message =
48         [NSDictionary dictionaryWithObject: aMessage
49                                     forKey: @"message"];
50     [center postNotificationName: @"ExampleNotification"
51                           object: self
52                         userInfo: message];
53 }
54 @end
55
56 int main(void)
57 {
58     [NSAutoreleasePool new];
59     // Set up the receiver
60     Receiver *receiver = [Receiver new];
61     // Send the notification
62     [[Sender new] sendMessage: @"A short message"];
63     return 0;
64 }
```

在 main() 方法中创建类的实例，然后调用 sender 的 sendMessage: 方法。发送出去的通知会被 receiver 收到：

```
$ gcc -framework Foundation notify.m && ./a.out
Received notification: ExampleNotification
Message is: A short message
```

4.8.3 发送异步通知

一般情况，发送通知是同步操作的。发送 -postNotification: 消息到通知中心，它会迭代所有注册为接收这种通知的对象，向其发送通知，然后返回。

每一个线程都有自己的通知中心，分别都有通知队列，通知队列是 NSNotificationQueue 的实例。它和通知中心的工作方式类似，但会延缓投递通知，等到将来运行循环迭代时再进行。

如果要不断生成同样的通知，通知队列就尤其有用。Observer 不是经常需要立刻运行。想想文本框里面的拼写检查器，每次用户输入一个字符，就需要发送一个通知。拼写检查器可以注册通知，接收通知，看输入的词是否有效，并更新显示状态。这样做有两个缺点，首先，一个单词在输入时要被检查几次，这些检查通常不是必需的。然后，拼写检查会影响到输入。

更好的设计，是在用户输入内容时，表示文本框需要拼写检查，但把真正的拼写检查延缓到以后做。通知队列提供了两个帮助，第一，它把通知聚合起来，把多个相同的通知转换成一个。就是说，每次按键都可以发送通知，但拼写检查器最后只收到一个。第二个有用的特性是延缓投递通知，当通过队列发送通知时，你可以决定是要立刻发送，还是过一会儿发送，或是等到线程没有其他要做的事情时再发送。一般程序总有大量时间是不做事的，通知队列可以等到这个时候再发送通知，比如说，当用户停止输入的时候再进行拼写检查。在实践中，没有什么用户打字速度能让现代的 CPU 满负载，但这同样适合于其他数据源，比如磁盘和网络，它们提供数据可以足够快，快到让 CPU 忙碌一段时间。

通知队列是消息中心的前端。你可以往队列中插入通知，之后它会再发送到通知中心，通知中心再发给 observer。这个组合的意义是，等待通知的对象没必要知道 sender 使用的是通知队列还是同步发送通知。

有两个办法可以获得通知队列的指针。最常用的是给类发送消息 +defaultQueue。它会返回和线程默认通知中心连接的通知队列。发送到这个队列的通知会被投递到其线程中。

另外，你可以显式地在获得的通知中心上创建队列。如图 4.1 所示，你可以有不同的队列。每个对象可以给一个或多个通知队列发送通知，或者直接发送到通知中心。通知队列会合并通知，并把它们传给通知中心，然后分发给所有注册的 observer。

如果有多个通知队列，就可以控制如何合并通知。你可能会把来自某一个类所有实例的通知都合并起来，但是其他类的同样通知不能合并，要分开发送，这样的情况可以给这个类创建一个新的通知队列。然后在初始化的时候，发送 -initWithNotificationCenter: 消息，把队列附加在通知中心上。

通过队列发送通知，可以用下面的消息，或省略掉一些参数用更简单的格式。

```
- (void)enqueueNotification: (NSNotification*)notification
               postingStyle: (NSPostingStyle)postingStyle
               coalesceMask: (NSUInteger)coalesceMask
                   forModes: (NSArray*)modes;
```

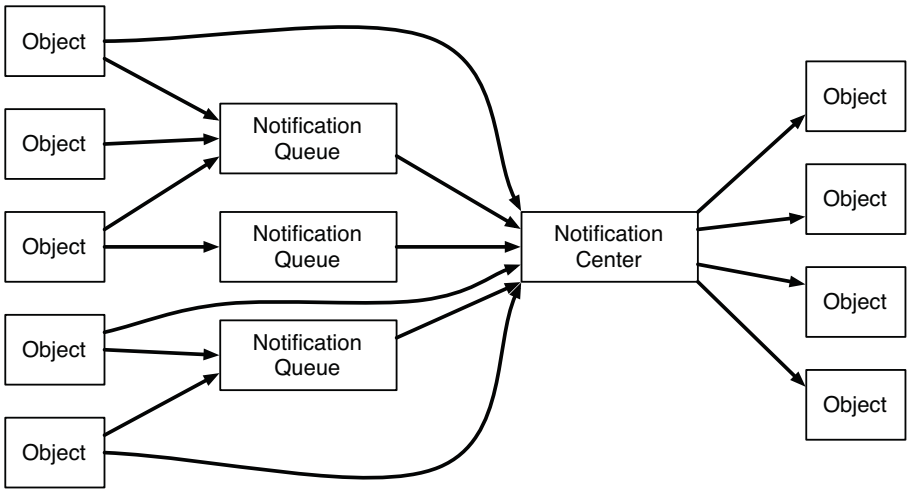

图 4.1：Cocoa 程序中的通知流程

第一个参数是通知，可以自己创建，和使用通知中心一样，有一些便利方法可以用来创建它们。常用的方法是给通知类发送 +notificationWithName:object:user Info: 消息。

第二个参数定义如何发送通知。这里有三个选项。如果指定 NSPostNow，它的行为就和直接发送到通知中心差不多。通知会同步发送，但首先会被合并。如果有一组通知，可以用这个方法来清理。如果有一组操作，它们都要触发某一类通知，可以把这些通知都发送到一个队列中，然后用 NSPostNow 方式发送，这样可以确保只有一个通知被发送出去。

其他的选项都会延缓通知发送，直到将来的运行循环迭代到。如果指定了 NSPostASAP，通知会被尽快发送。但就算这样，可能也不会在下一次运行循环迭代一开始就发送，因为可能还有使用同样优先级的其他通知已经在等待投递了，但也不会等待太久。如果通知不是非常重要，可以把发送方式设置为 NSPostWhenIdle。这会等到运行循环中没有更紧急的事件要处理的时候才投递。

可以在队列中停留更长时间的通知更容易被合并。如果所有通知都用 NSPostNow 方式投递，队列是没有机会合并它们的。如果所有通知都用 NSPostWhenIdle 方式投递，通知会在队列中停留比较长的时间，就会有更多的机会被合并。

合并行为是用第三个参数配置的。这个参数用掩码的方式组合一些标识，这些标识指定了什么样的通知要被合并，比如，它们来自同样的 sender 或是同样的

类型。一般这两个标识会被同时用到，或者都不使用。合并来自同样 sender 但类型不同的通知可能会导致非常奇怪的行为。你也可以合并来自不同 sender 类型相同的通知，但这也不是值得推荐的。

最后一个参数指定通知被投递时运行循环的模式。通过这个，可以避免通知被使用 certain 模式操作，或者定义新的模式用来操作 certain 类别的通知，并且保持它们在队列中，直到你明确通知运行循环进入这些模式。这样就提供了一个简单的方式延缓通知投递，直到你明确表示要操作它们。

通知队列非常强大，但并不是经常用到。一般把它们看做是优化技术。如果你在评估代码的时候发现操作重复的通知浪费了很多时间，就可以考虑增加通知队列了。

4.8.4 分布式通知

通知不仅可以用于单独进程。NSDistributedNotificationCenter 类是 NSNotificationCenter 的子类，它提供了分布式对象（DO）机制。这样在 OS X 中进程间通信（IPC）就变得很简单。

> **通知和信号**
>
> 如果有 UNIX 编程背景，你会发现通知和 UNIX 的信号概念很相似。当用通知队列组合时，工作方式类似使用信号支持的 POSIX 实时扩展发送信号。它们是异步投递的，可以排队，并且可以携带一个指针大小的扩展数据。

分布式通知不如直接使用分布式对象（DO）那么灵活，但提供了一个非常简单的进程间通信方式。从原理上说，分布式通知可以统一为 Bonjour，这样可以本地网段发送通知。构造函数 -notificationCenterForType: 提示了未来分布式通知的可能性，但现在 OS X 只支持 NSLocalNotificationCenterType。GNUstep 实现了 GSNetworkNotificationCenterType，可以使用分布式对象在网络之间投递通知，但目前 Apple 还没有提供同样的方法。

注册接收分布式通知，和注册接收普通通知差不多。主要区别在最后一个参数。注意分布式通知中心和通知中心中用来注册 observer 的方法原型是不同的：

```
// NSNotificationCenter
- (void)addObserver: (id)notificationObserver
           selector: (SEL)notificationSelector
               name: (NSString*)notificationName
             object: (id)notificationSender;
// NSDistributedNotificationCenter
- (void)addObserver: (id)notificationObserver
           selector: (SEL)notificationSelector
               name: (NSString*)notificationName
             object: (NSString*)notificationSender;
```

在通知中心中，最后一个参数是指向需要接收通知对象的指针。在分布式通知中心中，sender 是用来识别的名称，而不是指针。原因很显然，分布式通知来自其他进程，指针只在当前进程的地址空间有效。

发送分布式通知也有同样的变化。使用的方法是一样的，但 sender 是一个字符串，而不是对象。一般这是发送通知的应用程序名称。

发送分布式通知的 userinfo 字典也有一些同样的约束。因为通知是通过分布式对象（DO）发送的，所以字典中的全部对象必须符合 NSCoding 协议，这样它们可以被序列化，然后再被远程进程反序列化。大量的监听进程都能进行反序列化，这会保持通知很小。

4.9 小结

在本章中，我们看到了 Foundation 框架中最重要的几个方面。这个框架覆盖了 Cocoa 开发环境的核心函数，甚至提供了一些通常被认为是语言特性的功能，比如引用计数和消息转发。

我们花时间检验了 Foundation 库的核心概念，在下面的章节，你将会看到我们在这里讨论过的概念的例子。

我们看过了 Foundation 库提供的容器类——集合，数组，字典——以及如何迭代这些类型。我们也看到了如何在容器中存储非对象的基础类型。

本章覆盖了 Foundation 库框架中最重要的几个方面。但这不应该被当做详尽的参考。框架中包含了大量的类和函数。把 Foundation 库的类参考打印出来，尽管其中很多方法只有单行的注释，那也会比这本书还厚。

深入理解 Foundation 库的全部细节是不可能的。本章的目的是强调最重要的部分。要做一个好的 Cocoa 程序员，就算精通了本章中讨论的类，也仍然有很长的路要走。