# 第5章
# 应用程序相关概念

在 Cocoa 术语里，应用程序是一个进程，它会显示一个图形用户界面（GUI），运行时会与用户互动。Foundation 库里的类看上去对各种程序都有用，而 AppKit 则专注于生成交互式的 GUI。构造应用程序所需的许多核心功能都可以在 Foundation 里找到，而 AppKit 又对它们进行了扩展。

## 5.1 Run Loop

每个 Cocoa 应用程序的核心都是 run loop，它是协同例程的简单实现。Run loop 的每次执行都是由事件触发的，然后事件的响应程序会被触发并执行，直到完成。

在 Cocoa 里很少会显式指定事件及其处理程序之间的映射关系。由用户动作触发的事件，有一套复杂的转发规则确保调用到正确的对象，详细过程会在 5.3 节里讨论。其他事件则需要通过指定一个对象和配对的选择器来手工转发。

前面我们已经见过一个与 run loop 打交道的对象了，NSDistributeNotificationCenter 类会根据收到的远程消息来发送对应的 Objective-C 消息。这些消息是通过分布式

对象机制传递的，当消息出现时，远程对象会把一个事件插进应用程序的事件队列里；然后这个事件会传递给分布式通知中心，由它执行需要的函数调用。

这是 Cocoa 里很常见的抽象，即使不直接跟 run loop 或者事件类打交道，也能写出相当复杂的 Cocoa 应用程序。多数情况下，原始事件会被某个 Cocoa 自带的类处理，然后再转交给那些申请得到通知的对象。大多数时候，你是通过定时器或者通知来跟 run loop 打交道的。

定时器是 NSTimer 类的实例，它会登记一对对象——选择器，它们会在指定的时间间隔之后收到消息。

我们不忙享用 AppKit 提供的这些方便的封装，先来看看怎么在命令行工具里使用 run loop 机制，我们将利用分布式通知来做一个简化版的 UNIX talk 工具。这个例子包括以下三个类：

- LineBuffer NSFileHandle 在有数据可读的时候发通知给它，它每处理完一行也会发出一个通知。
- Sender 它接收行通知，并且发送一个分布式通知，其中包含有用户名及该用户输入的消息。
- Receiver 它侦听提供聊天消息的分布式通知，并将收到的每个消息输出到屏幕上。

这些类并不会互相引用，它们通过通知交流。Run loop 的加入，让这个应用变成完全是事件驱动的，没有哪个组件会在那里轮询，而是全都响应一个事件然后再等待下一个事件。

这个程序的第一部分是 LineBuffer 类，如代码清单 5.1 所示。创建时，它接收一个文件句柄作为参数，侦听这个文件句柄有数据可提供时发出的通知。第 101~104 行发给 NSNotificationCenter 的消息，所有的参数都不是 nil。在这里，该对象只想接收来自特定名字的特定对象发出的通知。在 105 行，它指示文件句柄生成一个新线程并在其中等待，等到有数据可读时再发出通知。

这个类实际的工作是在 -readData: 方法里完成的。当初始化时指定的通知出现时，这个方法就会被调用来响应之。首先，这个方法从文件句柄里读出数据，并以此构造一个字符串（第 110~114 行）。然后，它把该字符串追回到内部缓冲里，并用换行把它拆成字符串数组。

代码清单 5.1：行缓冲类（取自 examples/Notifications/distributedNotify.m）

```
@interface LineBuffer : NSObject {
    NSMutableString *buffer;
}
- (id) initWithFile: (NSFileHandle*)aFileHandle;
@end
@implementation LineBuffer
- (id) initWithFile: (NSFileHandle*)aFileHandle
{
    if (nil == (self = [self init]))
    {
        return nil;
    }
    buffer = [[NSMutableString alloc] init];
    NSNotificationCenter *center =
        [NSNotificationCenter defaultCenter];
    [center addObserver: self
               selector: @selector(readData:)
                   name: NSFileHandleDataAvailableNotification
                 object: aFileHandle];
    [aFileHandle waitForDataInBackgroundAndNotify];
    return self;
}
- (void) readData: (NSNotification*)aNotification
{
    NSFileHandle *handle = [aNotification object];
    NSData *data = [handle availableData];
    NSString *str =
        [[NSString alloc] initWithData: data
                              encoding: NSUTF8StringEncoding];
    [buffer appendString: str];
    [str release];
    NSArray *lines =
        [buffer componentsSeparatedByString: @"\n"];
    NSNotificationCenter *center =
        [NSNotificationCenter defaultCenter];
    // The last object in the array will be the newline string
    // if a new line is found, or the unterminated line
    for (unsigned int i=0 ; i<[lines count] - 1 ; i++)
    {
        NSDictionary *line =
            [NSDictionary dictionaryWithObject: [lines objectAtIndex: i]
                                        forKey: @"Line"];
        [center postNotificationName: LineInputNotification
                              object: self
                            userInfo: line];
    }
    [buffer setString: [lines lastObject]];
    [handle waitForDataInBackgroundAndNotify];
}
- (void) dealloc
{
```

```
139     NSNotificationCenter *center =
140         [NSNotificationCenter defaultCenter];
141     [center removeObserver: self];
142     [buffer release];
143     [super dealloc];
144 }
```

NSString 的 componentsSeparatedByString: 方法会返回一个字符串数组，其中至少会包含一个字符串。如果里面没有分隔符，它就是整个输入字符串。如果以分隔符结尾，那么最后一个字符串会是空串（@""）。如果不是以分隔符结尾，那最后一项就不会是空串了。这也就是说，除了最后一行，数组的各个对象都是一个完整的行，而最后一个对象要么是空串要么是不完整的行。[1] 该对象会为这里的每一行新发一个通知。

这些通知会由 Sender 类接收，如代码清单 5.2 所示。在第 57~60 行，这个类还登记接收了一个通知，该通知由行缓冲发出。这回它却没有指定源。在更复杂的程序里，可能会有不同的对象用这个名字来发通知，不过在这个简单的例子里，我们可以在两个类之间有一点松耦合。

**代码清单 5.2：分布式通知的发送者类（取自 examples/Notifications/distributedNotify.m）**

```
42 @end
43
44 @interface Sender : NSObject {
45     NSString *name;
46 }
47 - (id) initWithName: (NSString*)aName;
48 @end
49 @implementation Sender
50 - (id) initWithName: (NSString*)aName
51 {
52     if (nil == (self = [self init]))
53     {
54         return nil;
55     }
56     name = [aName retain];
57     NSNotificationCenter *center =
58         [NSNotificationCenter defaultCenter];
59     [center addObserver: self
60                selector: @selector(sendMessage:)
61                    name: LineInputNotification
62                  object: nil];
63     return self;
```

1 实际上，在这种用法下，NSFileHandle只会整行发送，尽管这种行为没有保证。

```
64 }
65 - (void) sendMessage: (NSNotification*)aNotificaton
66 {
67     NSNotificationCenter *center =
68         [NSDistributedNotificationCenter defaultCenter];
69     NSString *line =
70         [[aNotificaton userInfo] objectForKey: @"Line"];
71     NSDictionary *message =
72         [NSDictionary dictionaryWithObject: line
73                                     forKey: @"message"];
74     [center postNotificationName: ChatMessageNotification
75                           object: name
76                         userInfo: message];
77 }
78 - (void) dealloc
79 {
80     NSNotificationCenter *center =
81         [NSNotificationCenter defaultCenter];
82     [center removeObserver: self];
83     [name release];
84     [super dealloc];
85 }
86 @end
```

这个类既发送通知，又接收通知。它接收本地的行已读出通知，然后发送包含该行的分布式通知。第 75 行显示了发送分布式通知与本地通知的区别，分布式通知的发送者是一个字符串，本例中是发送聊天消息的人的名字，名字是在对象创建时设置的。

本例子最后一个类是 Receiver，如代码清单 5.3 所示。它也在创建时登记接收一个通知。跟另外两个类不同的地方则在于，它登记接收的是一个分布式通知。每次这个类收到通知，它就会把消息打印到屏幕上。注意，这里它用的是 C 函数 printf()。在 Objective-C 程序里用一些纯 C 没什么不妥。许多 C++ 程序员会觉得用 C 标准库会让他们的代码在某种程度上变得不整洁，但是 Objective-C 是设计用来给 C 增加能力，而不是取而代之，而且有时候用 C 函数正是完成事情最简单的办法。

至此我们有了这么三个类，还需要把它们彼此关联起来。代码清单 5.4 展示了这个小程序的 main() 函数。注意，用户名是从用户默认值系统查出来的。这是获取命令行参数快速而又简单的方法。NSUserDefaults 类允许用命令行选项来覆盖任何默认值，因此我们可以在调用可执行程序时用 -name 参数指定用户名，要是两边都没有指定，那就用 "anon"。

代码清单 5.3：分布式通知的接收者类（取自 examples/Notifications/distributed Notify.m）

```
@interface Receiver : NSObject {}
- (void) receiveNotification: (NSNotification*)aNotification;
@end

@implementation Receiver
- (id) init
{
    if (nil == (self = [super init]))
    {
        return nil;
    }
    // register to receive notifications
    NSNotificationCenter *center =
        [NSDistributedNotificationCenter defaultCenter];
    [center addObserver: self
               selector: @selector(receiveNotification:)
                   name: ChatMessageNotification
                 object: nil];
    return self;
}
- (void) receiveNotification: (NSNotification*)aNotification
{
    printf("%s: %s\n",
        [[aNotification object] UTF8String],
        [[[aNotification userInfo] objectForKey: @"message"] UTF8String]);
}
- (void) dealloc
{
    NSNotificationCenter *center =
        [NSDistributedNotificationCenter defaultCenter];
    [center removeObserver: self];
    [super dealloc];
```

代码清单 5.4：聊天工具的 main() 函数（取自 examples/Notifications/distributed Notify.m）

```
int main(void)
{
    [NSAutoreleasePool new];
    // Set up the receiver
    Receiver *receiver = [Receiver new];
    NSString *name =
        [[NSUserDefaults standardUserDefaults]
            stringForKey: @"name"];
    if (nil == name)
    {
        name = @"anon";
    }
    [[Sender alloc] initWithName: name];
    [[LineBuffer alloc] initWithFile:
```

```
161         [NSFileHandle fileHandleWithStandardInput]];
162     [[NSRunLoop currentRunLoop] run];
163     return 0;
164 }
```

第 162 行是运行循环开始的地方。每一个线程都有自己的运行循环对象，虽然开始的时候它们可能什么都不做。给它发送 -run 消息之后，它会进入睡眠状态，收到事件之后会派发事件，之后再进入睡眠状态。

你可能已经注意到，这个程序包含许多自动释放的对象，如拆分字符串时返回的数组就是一例。我们不用专门释放它们，它们会在当前的自动释放池释放时被释放。而我们只明确创建了一个自动释放池，在第 149 行，并且没有销毁它。不过有 NSRunLoop，字符串数组并不会造成内存泄漏。run-loop 每次循环开始的时候，会创建一个新的自动释放池，并在循环结束时销毁之。这也就是说，如果你在用 run loop，那只要 autorelease 对象就好，完全不需要操心池的管理。

如果用了垃圾回收，那么就更没啥好操心的了。即使启用了垃圾回收，run loop 还是有帮助的，它会在事件之间试着触发回收活动。

编译并运行这个工具，就有了一个简单的聊天系统，我们在这个系统里输入的每一行都会发给该程序的所有实例（以及任何请求接收消息的其他程序）。

```
$ gcc -framework Foundation -std=c99 distributedNotify.m
$ ./a.out  -name David
Thomas: Hello
Hi
David: Hi
How are you?
David: How are you?
Thomas: Fine thanks.
```

同时，在另一个终端里：

```
$ ./a.out -name Thomas
Hello
Thomas: Hello
David: Hi
David: How are you?
Fine thanks.
Thomas: Fine thanks.
```

来自两个实例的消息在两边都会显示出来。Run loop 让我们可以创建简单的事件驱动应用程序，它使用两个事件源——键盘和聊天程序的另一实例，而分布式通知则可以完成简单的点对点通信。

## 5.2 应用程序和委托

目前为止我们用到的所有例子都有一个 C 风格的 main() 函数。所有的 Cocoa 应用程序都要有这个函数，因为 Objective-C 并没有定义其他的程序入口点。在绝大多数 Cocoa 应用程序里，这个函数的主体会由 Xcode 创建并且没有修改，其中只有下面这么一行：

```
return NSApplicationMain(argc, argv);
```

OpenStep 规范里说，这个函数等价于以下代码：

```
void NSApplicationMain(int argc, char *argv[])
{
    [NSApplication sharedApplication];
    [NSBundle loadNibNamed: @"myMain" owner: NSApp];
    [NSApp run];
}
```

这个说明有点过于简化了，主 nib 文件的名字并不是硬编码的，它是从应用程序资源包（bundle）里的属性列表中查出来的。不过除了这一点，基本上就是这个样子了。第一行创建一个共享的应用程序对象。NSApplication 使用单例模式，每个应用程序里它只会有一个实例。在这个函数首次被调用时，它会设置全局变量 NSApp 指向这个共享实例，当应用程序开始运行之后，就会返回这个全局变量了。

Nib 文件加载之后，其 owner 设置为这个应用程序对象。这样，从主 nib 文件里构造的对象，就能轻松地跟应用程序其他部分创建的对象关联起来。应用程序委托通常是在 nib 文件里设置好的，当然也可以在加载 nib 之前通过代码指定。最后，发一个 -run 消息给应用程序对象。

当应用程序运行时，默认的 NSRunLoop 实例会从系统收集消息并负责分发。默认情况下，NSApp 会注册到大部分用户输入消息上，然后发给响应者链（responder chain）。响应者链会在下一节解释。

可以定义 NSApplication 的子类，尽管通常这是个坏主意。通过 -setDelegate: 或者直接在 nib 文件里可以指定一个委托，应用程序会在做几乎每件事情时调用这个委托，这是修改应用程序行为更干净也更安全的方法。

应用程序的主要入口点在应用程序委托里。委托可以选择实现两个方法中的一个作为主入口点：

```
- (void)applicationWillFinishLaunching: (NSNotification*)aNotification;
- (void)applicationDidFinishLaunching: (NSNotification*)aNotification;
```

Cocoa 里有个很常见的模式，事件会有“should”、“will”和“did”三个形式。“should”形式询问委托是否应该处理该事件，“will”形式表示事件将要发生，“did”形式表明事件已经完成。

这里没有“should”形式，因为假如一个应用程序已经开始启动，我们认为它总是应该启动的。

-applicationWillFinishLaunching: 委托方法是在 run loop 初始化完成时调用的。如果拖放文件到上面而导致应用程序开始运行，那么接下来这些文件会被打开，最后 -applicationDidFinishLaunching: 方法会调用，此后会开始正常的事件处理。

注意，这两个方法都接收一个通知作为参数。其他对象也能收到这些通知。如果在给应用程序发送 -run 消息之前创建一个对象，并登记它要接收名为 NSApplicationDidFinishLaunchingNotification 的通知，那么应用程序启动完成时也会通知这个对象。应用程序的很多其他委托方法也是这样，你可以选择是在委托内部处理还是用另外一些代码。例如，-applicationDidHide: 消息对于委托用处不大，很少有应用程序需要在隐藏时做特别的事情，但是个别视图对象可能会，比如在解除隐藏之前停止更新内容。这时候，应该让各个视图登记接收相应的通知，而委托则不用实现这个方法。

通知，尽管是 Foundation 的一部分，对于 AppKit 却重要得多。许多 AppKit 对象至少发送一种通知。大部分遵循与 NSApplication 同样的模式，发送所有的通知给委托的特定方法，同时也把通知发到通知中心。委托机制要简单一些，因为你只需要实现适当的方法，不过也有限制，每个对象只能有一个委托。

还记得吗？ Cocoa 在委托上使用内省（introspection）。你的委托对象可以不实现任何一个 NSApplication 文档描述为委托方法的方法。用户程序对象在调用这些方法之前，会给委托发送 -respondsToSelector: 消息，只有返回 YES 时才发送委托消息。有些支持委托的对象会缓存委托理解的消息集，因此，如果你使用任何运行期技巧来替换委托响应的方法，那么需要再次调用 -setDelegate: 以确保缓存已更新。

## 5.3 响应者链

Cocoa 里有两类事件。我们已经见过高层事件，它们是 NSNotification 类的实例。来自窗口系统的低层事件则封装成 NSEvent。

你极少需要跟事件打交道（相反经常用通知和消息）。AppKit 里有很大一块就是为了隔离它们。例如，当你按一个键时，会生成一个事件，但你只会收

到一个通知说某个文本视图展示的文本变了，或者收到来自按钮或菜单项的 Objective-C 消息说它们已经被点中了。

通常，视图类会处理 NSEvent。然后它们调用委托方法，或者给其控制器里的目标（target）发送活动消息（action message），或者发出一个通知。控制器响应活动消息，并调用其模型对象。模型对象则会发送通知。

### 5.3.1 事件传递

事件传递与通知分发完全不同。一个通知可以被随便多少个对象接收处理，但只能有一个对象能处理事件。事件发给应用程序的 -sendEvent: 方法，从而进入系统。你可以定制一个 NSApplication 子类来覆盖这个方法，从而实施某种事件拦截，虽然罕有人这么干。这个方法接收一个 NSEvent 作为参数，它是 Cocoa 里事件分发的基本形式。这个层次的事件封装诸如按钮或鼠标移动之类的活动。

从 OS X 10.6 开始，Apple 引入了一个机制，可以在事件传递给应用程序之前观察或转换事件。NSEvent 的 -addLocalMonitorForEventsMatchingMask:handler: 方法接收一个代码块作为第二个参数。任何事件被发送之前，都会先经过这个代码块，而实际发给应用程序的事件是这个代码块返回的。举个例子，你可以在应用程序里利用这个来捕获鼠标事件并用键盘事件替换它，从而实现鼠标手势。你可能也会想到这对调试很有用，可以在特定事件发给应用程序时得到通知。

多数事件需要发给指定的窗口。NSApplication 调用活动窗口的 -sendEvent: 方法来完成传递，活动窗口是 NSWindow 的实例。从这里开始，事件传递变得友好一点了。每个窗口有一个首先响应者（first reponder）指针，响应者是能接收键盘事件的视图对象，键盘事件会发送给这个对象。鼠标事件则会发送给事件发生时所在的视图。图 5.1 显示了 Interface Builder 在 nib 文件里创建的默认对象。注意，里面有一个 First Responder 对象。这样首先响应者就可以设置为活动（action）的目标（target）。这种设置允许 target/action 机制把消息发给响应者链。这种用法的例子如剪切、复制、粘贴菜单项发送活动消息给首先响应者。

响应者链负责决定哪个对象应该处理这样发过来的事件。响应者链从首先响应者开始，然后沿着视图层次向上遍历。视图层次可能相当复杂，图 5.2 展示了一个中等复杂度的窗口。它包含一个文本视图（text view）、一个浏览器视图，以及一个按钮。尽管看上去简单，视图层次可不是这么简单的。我们可以用 Interface Builder 的大纲视图（outline view）看到这个层次。图 5.3 展示的就是这个窗口里全部的视图。

在最深的地方，共有六个嵌套的视图。如果文本视图是首先响应者，那么它

们每个都会有机会响应事件。多数情况下，文本视图会自己处理事件，如果它没有，就会委托给带边框的滚动视图（bordered scroll view），它会再委托给分隔条视图（split view），如此这般，直到找到窗口对象。

如果窗口对象处理不了，那么它会先传给它的委托，再给它的控制器。最后到达应用程序对象，然后会给应用程序对象委托一个机会来处理。在文档驱动的应用程序里，还有 NSDocument 和 NSDcoumentController 对象会有机会处理事件。

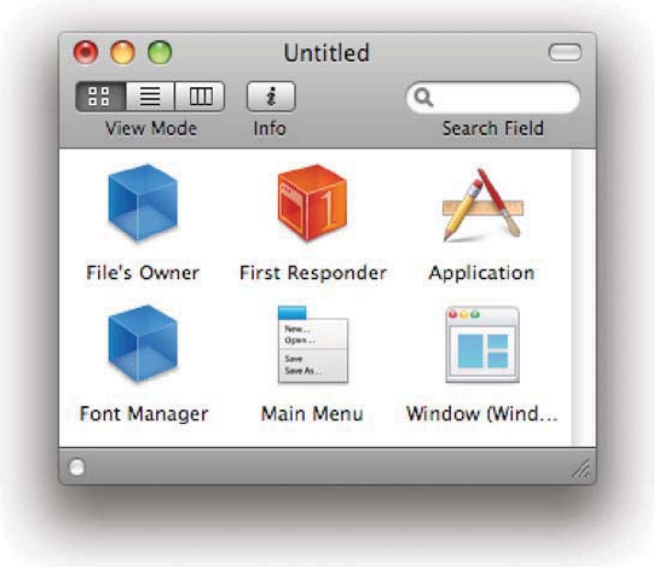


图 5.1：应用程序 nib 里的默认对象

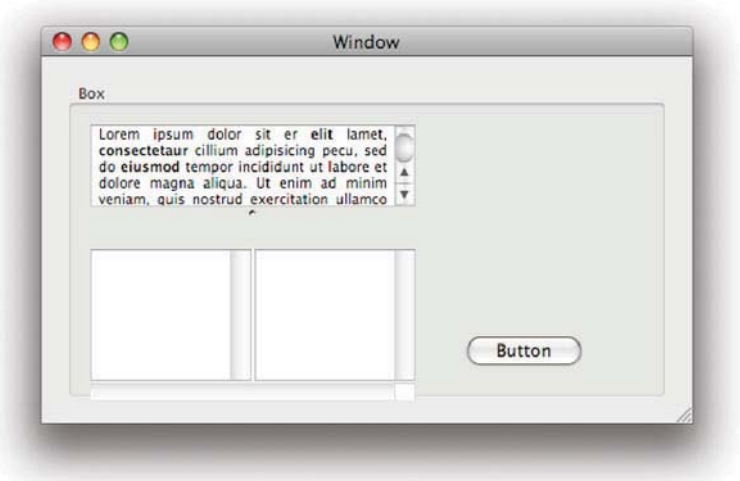


图 5.2：内有视图的窗口

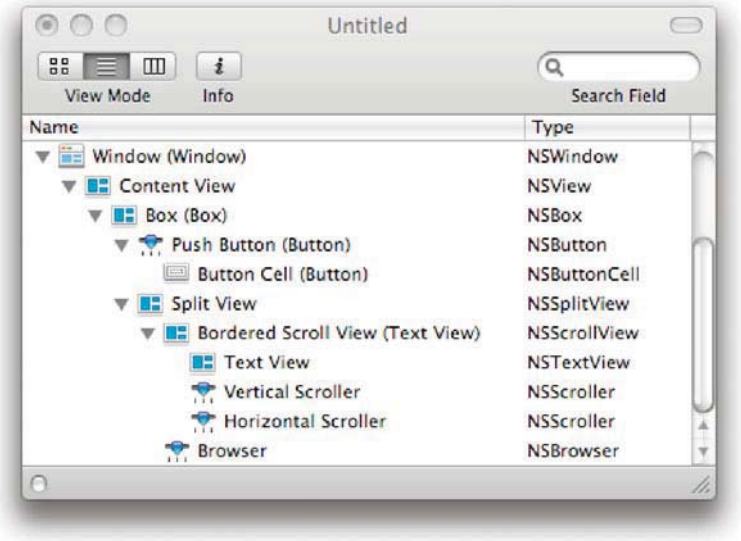


图 5.3：图 5.2 中窗口的视图层次

从概念上讲，你可以认为从视图层次看，窗口是在应用程序里。视图层次上的每个对象会先试着自行处理事件，然后传给其委托对象，再传给控制器（如果跟委托不是同一个对象），然后传给视图层次的上一级。这是实际过程的简化描述，不过多数情况下已经足够接近实际情况了。

你有两个办法来打破响应者链。如果你制作一个自定义视图，就可以不把事件转给视图层次的上层，当然你得慎重考虑要把事件发到哪里去。另外，你也可以在 NSApplication 或者 NSWindow 的子类里覆盖 -sendEvent: 方法，从而在事件进入响应者链之前就抓住它们，就像这样：

```
- (void)sendEvent:(NSEvent *)event
{
    if ([event type] == NSKeyDown)
    {
        // Intercept events
        return;
    }
    [super sendEvent: event];
}
```

这个简化的版本忽略了所有传给窗口的按钮事件。你也可以这样只拦截某些事件，可以特别处理或干脆不让它们进入响应者链。像 Keynote 的演示模式就会用到类似这样的东西，拦截所有按钮事件，不再发给用来绘制显示内容的对象，而是用来控制幻灯片播放或者悄悄地忽略掉。

### 5.3.2 目标与活动

许多视图使用目标 - 活动（target-action）模式来通知其控制器某件事件发生了。活动是一种特殊的消息，它有一个 sender 参数。有两个办法告知 Interface Builder 目标和活动，首先是通过检视器，其次是用头文件。在 Cocoa 头文件里，你可以找到这样的宏定义：

```
#define IBAction void
```

在代码中可以通过像下面这样定义活动方法来使用它：

```
- (IBAction)doSomething: (id)sender;
```

编译时，预处理器会把 IBAction 简单替换成 void，因此这个方法应该没有返回值。如果你在 Interface Builder 里加载这个头文件，它就会知道类的实例希望收到这类活动。

在活动上能做的最简单的事件，就是把它们连接到特定目标上，不过这不是最灵活的用法。它们也可以连接到响应者链上。如果你把目标设置成首先响应者代理，那么消息就会沿着修改过的响应者链传递。

传递活动消息要比一般的消息稍复杂一点。NSResponder 里定义的每个事件消息都有一个默认的实现，它只是调用链条里的下一个响应者。而如果你发送一个 doSomething: 消息给首先响应者，会收到一个异常报告，说首先响应者不理解这类消息。

活动传递实际上是由 NSApplications 的 -sendAction:to:from 方法处理的。它从键盘窗口（key window）的首先响应者开始，沿响应者链向上直到找到键盘窗口。然后它去找主窗口（main window）的首先响应者（如果主窗口不是键盘窗口），然后再向上遍历它的响应者链。

对于活动的每个可能的目标，它会发送 -respondsToSelector: 消息看看目标是否理解该消息。如果理解，就发送活动；否则去响应者链的上层。跟普通的响应者链一样，如果没有视图能处理，NSApp 及其委托会试着处理消息。

这个机制用于自动启用和禁用菜单项。你可以发送 -targetForAction: 消息给 NSApp，参数是活动的选择器。如果当前响应者链的某个对象认识这个选择器，就返回这个对象；否则返回 nil。如果你把菜单项的目标设置为首先响应者，那么这个机制就会在菜单项显示时用来决定该项是否可用。如果响应者链里没有对象响应这个活动，那么它会变灰掉。

### 5.3.3 变成首先响应者

正在响应鼠标事件的那个视图会自动取得首先响应者状态。如果你实现 NSView 的子类，只要覆盖 -accetsFirstResponder 方法并返回 YES 就能自动获得这个行为。这个方法会被调用，以确定是否应该调用方法询问是不是要变成首先响应者。这里通常应该返回一个常量，要是某个视图在特定时刻不想成为首先响应者，那稍后还有别的机会拒绝。

当一个视图被选中成为新的首先响应者，它会收到 -becomeFirstResponder 消息。它应该在这里完成所有必要的准备工作，诸如绘制环绕自己的边框，如果接受首先响应者状态就返回 YES。

在另一个视图变成首先响应者之前，当前这个必须先放弃这个状态。这要调用 -resignFirstResponder 方法。某个对象可以在这个方法里返回 NO 以拒绝失去首先响应者状态，不过这很罕见。

永远也不要自己调用这些方法。窗口对象会跟踪首先响应者，有些对象会向窗口要首先响应者，试着手工设置一个可能会让它们无法工作，并且给事件传递制造混乱。相反地，应该用 NSWindow 的 -makeFirstResponder: 方法，它会以适当的顺序调用这些方法并正确跟踪变化。

## 5.4 应用程序里的Run Loop

本章开头我们讲过 run loop 的基本概念。NSRunLoop 等待事件，并在收到事件时发送消息。任何使用 Foundation 的项目都可以使用 run loop，包括命令行和 Web 应用程序，不过对于用 AppKit 写的程序，它要重要得多。

Foundation 里的 run-loop 类跟分布式对象紧密集成在一起。最早的 run-loop 模型来自 NeXTSTEP 的 Mach 内核架构。run loop 只理解一个事件源，就是 Mach 端口。这些端口是 Mach 应用程序与进程外的部分通信的基本机制。

作为一个微内核，Mach 自己只做很少一点事情。它提供两个抽象：内存对象和端口。内存对象就是带有某些权限的一块内存，而端口则是可用于传递消息的链接。在这之上，用到两个模型来构造复杂的操作系统。纯微内核方法，也称为多服务器微内核，用到大量进程，这些进程提供传统操作系统的各项服务。每个设备驱动会用自己的进程运行，并为设备的 I/O 范围分配一些内存对象。网络栈里 TCP、IP、以太网和网卡驱动可能会分别有自己的进程，彼此通过端口通信。

虽然这个设计很容易扩展到多个处理器上，但也有一个明显的不足，通过 Mach 端口发送消息比进行系统调用大约慢 10 倍。这使得多服务器模型在单处理器机器上非常慢。替代方案是单服务器微内核。它以 Mach 进程的形式运行一个现成的内核，通常是 4BSD。内核的底层部分重写成调用 Mach 原语，而不是使用平台特定的汇编，然后绝大多数用户空间进程只跟这个服务器交流。

OS X 和 NeXTSTEP 都是单进程方法的例子。不过在 NeXTSTEP 里，Mach 端还是广泛用于进程间通信（IPC）。因为微内核设计，Mach 端口既可以用来接收其他进程的消息，也可以接收操作系统的，或者是来自那个实现了绝大部分操作系统功能的 BSD 服务器。

设计 run-loop 模型的指导思想就是：所有事件会从外部进程发出、以 Mach 消息的形式到达。当 OpenStep 在其他平台之上实现时，这个说法稍微有点泛化，不过基本思想还是一样的。事件源被封装成 NSPort 对象，其中某些对程序员隐藏了，特别是连接到窗口服务器（window server）来发送键盘和鼠标事件的那些。来自其他进程的使用分布式对象的连接，则通过 NSPort 子类处理，它们是对 Mach 端口或者网络套接字的封装。

run loop 有两个方法管理事件源：

```
- (void)addPort: (NSPort*)aPort forMode: (NSString*)mode;
- (void)removePort: (NSPort*)aPort forMode: (NSString*)mode;
```

第一个参数都是端口，端口是封装通信渠道的抽象类。第二个参数则是 run loop 模式。传统 Foundation 上只定义了两个模式：NSDefaultRunLoopMode 和 NSConnectionReplyMode。第二个是给作为分布式对象机制一部分的端口准备的，而第一个则可用于其他所有的事件源。只有 run loop 的模式与端口加进来的模式一致时，端口才会被用做事件。

这又是一个 AppKit 建立在 Foundation 功能之上的好例子。使用 Foundation 编程时，你可以定义自己的 run-loop 模式，以限制程序在某个状态下会收到的消息。当使用 AppKit 时，你会找到两个已经定义好的模式，分别是 NSModalPanelRunLoopMode 和 NSEventTrackingRunLoopMode。如果你运行一个模态对话框，那么只有目的地是该窗口的消息会在第一个模式下传递。第二个则允许同步跟踪鼠标的移动，例如在一个盒子里绘制形状或者选择某块文本。用纯异步事件来实现这些会相当困难。

当应用程序启动时，NSApp 会用默认模式启动主线程的 run loop，并登记接收来自窗口服务器的消息。图 5.4 显示了一个 Cocoa 应用程序事件消息流的例子。

窗口服务器会给进程发送 Mach 消息[1]，其中封装有一个端口消息对象。这个消息传给 run loop，在其中转换成 NSEvent 对象，再传递给 NSApp 的 -sendEvent: 方法。应用程序对象会把它发给适当的窗口，它再根据事件类别用适当的方法转给首先响应者。在这个例子里，首先响应者没有处理，而是委托给响应者链。此后，上级视图把事件转给其委托，该委托对象会响应这个事件。

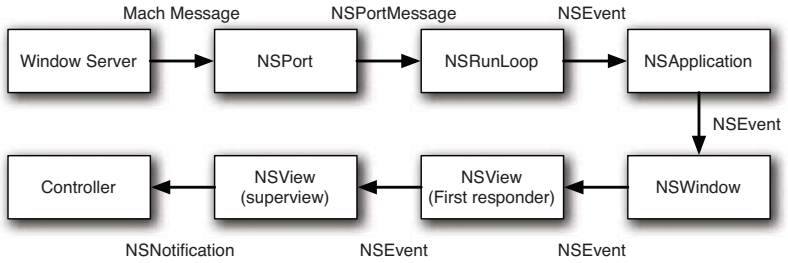


图 5.4：Cocao 应用程序消息流的例子

以这种方式传递事件过程非常复杂，多数情况下，你可以忽略之。如果你想要实现一个视图对象，只要创建 NSResponder 的子类，或者更多情况下是它某个现成子类的子类。如果你想要实现一个控制器，那只要把你的对象设置为委托或者注册到某个通知上，两种情况下，你都只要为特定的高层活动实现几个 Objective-C 方法。

许多 Cocoa 类隐藏了 run loop 模式的细节。最简单的例子是 NSAlert。它会显示一个模态对话框，可以是整个应用程序模态或者模态附着在某个窗口上。两者的实现方法差异很大。第一个 run loop 会用队列缓存所有事件，除非其目的地就是显示警告的那个窗口；第二个则会正常传递事件，不过会指示窗口在对话框退出之前不要响应其他任何事件。

其中第一个方法支持简单的同步式编程风格（尽管用户体验很糟糕）。你可以简单地调用一下 alert 对象上的 -runModal 方法，然后通过返回值获得被按下的按钮。第二个稍微复杂一点，需要调用 alert 上的这个方法：

```
- (void)beginSheetModalForWindow: (NSWindow*)window
                   modalDelegate: (id)modalDelegate
                  didEndSelector: (SEL)alertDidEndSelector
                     contextInfo: (void*)contextInfo
```

这个方法指定了表单将要附着的窗口，以及一些回调信息。委托和选择器参数符合标准的目标 - 活动模式，当对话框结束时选择器指定的方法会被调用。跟

1 或者，也可能是其他形式的IPC。

一般的活动不同的是，这个选择器必须接收三个参数：刚结束的 alert 对象、标记按钮的返回值，以及传入的 contextInfo。

别的很多面板也遵循这个模式。如果你的应用想要显示一个“打开”对话框，那么 NSOpenPanel 对象既有方法可以作为模态对话框执行，也有按照非模态对话框执行的方法。通常，应该只在做原型或快速 Hack 时使用模态形式，真正的应用程序应该使用非模态的版本。

还有一个事件源我们还没有仔细看过，NSTimer 类提供了一个简单的定时器接口，并且与 run-loop 类密切相关。当定时器被加到 run loop 里时，会保存在一个队列里，队列的第一个定时器规定了 run loop 等待事件时会阻塞多久。通常，可以用下面这个方法创建定时器：

```
+ (NSTimer*)scheduledTimerWithTimeInterval: (NSTimeInterval)seconds
                                    target: (id)target
                                  selector: (SEL)aSelector
                                  userInfo: (id)userInfo
                                   repeats: (BOOL)repeats;
```

第一个参数规定未来多少秒之后定时器被触发。如果 repeats 参数是 YES，那么定时器会以固定的时间间隔反复触发。回想一下，NSTimeInterval 是一个 double（双精度浮点数），因此它可以接收分数秒，每秒触发多次的定时器可用来实现动画。剩下三个参数，跟 Cocoa 里其他用到回调的方法是一样的。选择器必须接收一个参数，就是 NSTimer 对象自己。作为通知，它响应 -userInfo 消息并返回第四个参数传进来的对象。

## 5.5 委托与通知

Cocoa 里绝大部分对象都有委托对象，可以调用它们的 -setDelegate: 方法来指定，不过更多情况下，你会在 Interface Builder 里指定它们。在 Interface Builder 里，如果你按住 Control 键在两个对象间拖动，就会看到图 5.5 里的那种线出现。线的两端都高亮了，这在试图把一个对象连接到子视图时非常有用。在图上的例子里，一个对象是在窗口里，另一个则是在项目窗口（project window）里用图标表示的。如果在项目窗口里点一下大纲视图图标，则会看到视图层次的大纲，有时候用大纲视图连接嵌套层次比较深的对象会更方便一些。

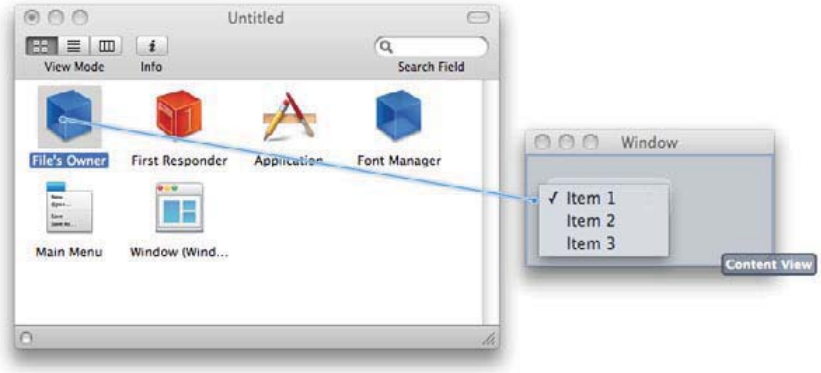


图 5.5：在 Interface Builder 里连接插点和活动

当放开鼠标时，就会看到所有可供连接的活动和插点（outlet）。每个对象最多可以发送一个活动（不过想接收多少个都可以），而它定义的插点每个只能设置一个对象。图 5.6 显示了下拉框所定义的插点：delegate。如果有对象连接到这个插点，那么定义好的所有委托消息都会发给它。

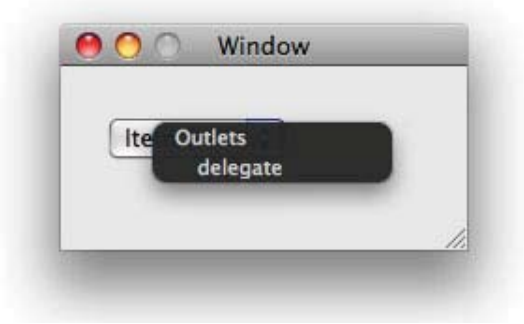


图 5.6：在 Interface Builder 里设置一个委托

委托可以收到两种消息，简单一点的等价于通知，通常有一个参数接收通知，另一种则需要委托一些活动。 NSControl 里两个都多次用到，这个类是 Cocoa 里所有简单控件的父类。这个对象支持的委托方法包括以下针对编辑文本的：

```
-        (BOOL)control: (NSControl*)control
textShouldBeginEditing: (NSText*)fieldEditor;
- (void)controlTextDidBeginEditing: (NSNotification*)aNotification;
```

第一个返回一个布尔值表明控件此时是否允许修改文本。如果委托决定实现这个方法，那么它就能控制视图的行为。相反，第二个方法只是简单地通知一下某事已经发生，其返回值是 void，因为这里不需要委托做决策。在调用这个方法的同时，控件还会发送一个 NSControlTextDidBeginEditingNotification。其他任意

多个对象也可以侦听这个通知并按自己的方式处理之。需要回应的委托方法则不是这样，它们只能由单个对象处理。

### 控件与视图

某些 GUI 工具包把所有的可视组件都称为控件，Cocoa 则不然，控件是视图的一类，视图这个术语用来指代所有用户界面组件。控件是一类简单的视图，它们只封装特定类格子（cell）的行为。通常它们是非常简单的组件，用于表现少量非结构化的数据。

许多更加复杂的对象还有另一类委托，称为它们的数据源。当委托控制视图的用户交互行为时，数据源控制视图与模型对象的交互。数据源经常需要至少实现某几个方法，这跟一般的委托不同，一般的委托通常都可以忽略任何不感兴趣的消息。数据源得要用数据填充视图，因此格子（cell）返回给它的值就很重要了，不能用默认值替代。

NSTableView 类是使用数据源的好例子。以下必须实现的方法是由 NSTableDataSource 这个非正式协议规定的。这个协议里只有两个方法是数据源必须实现的：

```
- (NSInteger)numberOfRowsInTableView: (NSTableView*)aTableView;
-           (id)tableView: (NSTableView*)aTableView
objectValueForTableColumn: (NSTableColumn*)aTableColumn
                      row: (NSInteger)rowIndex;
```

头一个返回数据的行数，第二个为特定行列组合返回一个对象。注意，这些方法都有一个视图对象参数，也就是说同一个数据源对象可以用到多个视图上，并且可以为视图返回不同的数据。

这个数据源协议里最常用的可选方法是跟前面第二个方法配套的，用来设置值：

```
- (void)tableView: (NSTableView*)aTableView
   setObjectValue: (id)anObject
   forTableColumn: (NSTableColumn*)aTableColumn
              row: (NSInteger)rowIndex;
```

如果没有实现这个方法，那么表格视图就不支持把数据写回模型对象，即使它被设置成可编辑状态。该协议里的其他方法是关于拖放处理的。有时候，数据源方法可以用来提供更好的用户交互。NSComboBox 的数据源协议有一个可选方法来实现自动补全（auto-completion）。如果数据源对象实现了这个方法，组合框

就会试着自动补全用户输入的文字。

我们会在第 11 章更详细地介绍数据源。

## 5.6 视图层次

视图层次用来传递事件，但也提供一种相关视图对象的语义分组，以及一套增量坐标变换。每个视图都有自己的坐标系，定义上级坐标的坐标变换。

### 5.6.1 窗口

视图层次的最顶上是窗口。从概念上讲，窗口就是一个虚拟的显示器。操作系统的工作就是让不同的程序共享硬件资源，同时无须了解彼此的存在。它会对所有程序的声音流进行混音以共享扬声器，它根据用户的选择让应用程序中的每一个独占键盘来共享键盘，它为每个应用程序分配时间片来共享 CPU，它还通过分区来共享内存。它让每个应用程序创建虚拟屏幕，并且把它们的内容显示到真正的屏幕上，从而实现屏幕共享。

在 Cocoa 里，屏幕由 NSScreen 对象表示。它不是视图层次的一部分，虽然你可能会希望它们是视图层次的顶层。屏幕是非常简单的对象，提供显示设备的信息，但不处理实际在屏幕上画东西的工作，这个是由窗口服务器完成的。当你在一个窗口里画图时，就是在向窗口服务器发送命令。根据所用 OS X 版本及硬件能力的不同，这些命令会在 CPU 或 GPU 上运行代码实际执行画图操作，结果存进一个离线缓冲区。然后，窗口服务器负责把所有的缓冲区拼到一起，形成屏幕上的内容。

Cocoa 里的窗口是 NSWindows 类的实例。这个类负责传递事件给视图，以及处理在其范围内的图形绘制。每个窗口都只有一个 NSView 对象代表其内容。这个视图可以包含其他视图，通常也确实包含一些。

任何 Cocoa 应用程序，都有两个重要的窗口，主窗口和键盘窗口。主窗口是代表用户注意力的窗口，通常是当前文档的窗口或非文档驱动应用的主要用户界面。键盘窗口则是接收键盘事件的窗口。通常两者是同一个窗口，但并不总是这样的。文档经常会有许多辅助窗口浮在周围，如面板或检视器，它们可以变成键盘窗口但不是主窗口，例如用户想要在其中某窗口里的文本框里输入时。

AppKit 里，NSWindow 有一个子类是广泛使用的，那就是 NSPanel。它代表浮动面板。Cocoa 的标准对话框就用到了它，它不能变成主窗口。面板主

要的工作就是覆盖 NSWindow 里的返回某值的方法，让它返回另一个值。例如 -hidesOnDeactive 方法，在两个类中这个方法都返回一个可设置的标识，不过在窗口里默认值是 NO，而面板里则默认是 YES。这就意味着面板在应用程序不再活动时会被隐藏，而窗口则不会。这有助于 OS X 减少屏幕凌乱，应用程序通常不应该覆写这个设置。

通常每个窗口都关联有一个控制器，是 NSWindowController 子类的实例。它比其他控制器做的事情要多一点，并且在 nib 加载过程中会用到。通常，你会向 NSWindowController 对象发送 -initWithWindowNibName: 消息来创建窗口。这个方法从应用程序资源包里加载指定名字的 nib，并把自己设置成所有者（owner）。典型情况下，nib 里的主窗口会把其窗口控制器设为文件的所有者（File's owner）。

### 5.6.2 视图

Cocoa 视图一般是 NSView 的子类。因为 Objective-C 类型系统和消息派遣机制并不依赖于类继承关系，也可以用其他类的子类作为视图对象，不过这需要大量的重复劳动，通常不是什么好主意。

视图是响应者链的一部分，从 NSResponder 那里继承了许多这方面的行为，NSResponder 是 NSView 的父类。视图增加的行为绝大部分是跟画图相关的。

与绘制相关的主要方法是 -drawRect:。它接收一个 NSRect 作为参数，并且绘制视图在这个矩形区域内的部分。这个矩形用的是接收者的坐标系统。每个视图都有自己的局部坐标系统。视图的左下角坐标是 (0, 0)——原点——坐标值向右上角增大。

在子类里调用 -drawRect: 之前，父视图会修改绘图状态，以便让子视图的坐标系统生效，并在调用后恢复老的坐标系统。这就是说，在视图里绘制是非常简单的。视图很大程度上可以忽略其父视图和相关的坐标系统。

Cocoa 包含有大量的标准视图类。如果可能，请尽量使用它们而不是自己定义一个。Mac 应用程序的一致性，有一部分就源于多数应用程序都使用精心挑选出来的用户界面元素。如果你定义一个视图对象，实现的是跟现存视图近似的功能，那么你可能会把用户搞糊涂。最常用的视图包括：

- NSButton，封装了一个简单的按钮，提供有大量选项。
- NSImageView，用于绘制图片。
- NSTextField 和 NSTextView，分别提供简单和功能丰富的文本绘制功能。
- NSMovieView，QuickTime 的封装，可用来在窗口里显示视频。

- NSOpenGLView，处理 OpenGL 环境的设置，支持调用 OpenGL API 在 Cocoa 窗口里画图。

还有些别的，大部分都可以在 Interface Builder 里使用和设置，无须写代码。注意，Interface Builder 组件盘（palette）可选条目跟视图类并非一一对应。有些类，像 NSButton，会以不同的默认选项出现多次。

### 5.6.3 格子

许多视图，像表格视图，包含大量小组件。一个小表格可能会有十行十列，在每一个行列交点，都有些东西需要画，可能还会有列头，所以总计 100 个子视图。这可能会吃掉很多内存。在 10.5 里，按 32 位编译的话 NSView 是 80 字节，64 位模式下是 152 字节。对于表格视图，我们需要 16KB 的内存来放视图对象。如果放大到 100×100 的表格，那视图实例就要占掉约 1.5MB 内存。在实际程序里，你多半会使用 NSView 的子类，这样开销就更大了。

这个问题的解决方案是使用格子——NSCell 的子类——它比完整的视图要轻量级得多。在 Leopard 上 NSCell 是 20 或 32 字节，分别对应 32 位和 64 位目标。对于 100×100 的例子，用格子取代每个行列点上的视图，可以节省约 1MB 内存。格子跟视图不一样，它不维护自己的的坐标系统，也不会用代码完成相关设置；相反，它们画到一个特别的视图里。这样的一个衍生效果是，格子可以多次重用以在视图上进行绘制。

> **像素和点**
>
> Cocoa 里所有的绘制都使用 PostScript 显示模式。Cocoa 里的距离单元是 PostScript 点。每英寸有 72 个 PostScript 点。老一点的 OS X 会假设显示总是 72dpi 的，因此有些像素总是一个点。这不是什么好假设，现代显示器可以超过 150dpi。代码不应该对点会占多少个像素做假设。

表格视图通常每列只有一个格式，在每个可见行上都用它。通常它会调用格子的 -setObjectValue: 方法，设的值是数据源返回的，然后用 -drawWithFrame:inView: 方法告诉格子在视图的什么位置画自己。

格子可以想象成是墨水图章。墨水图章让你可以在纸上用不同的颜色画同一个形状，只要蘸上不同颜色的墨水再盖章就行了。

许多视图对象都使用格子。最简单的是 NSControl。它是一个非常简单的

类，只用一个格子进行绘制。Cocoa 里许多基本用户界面元素都是它的子类并带有专用的格子。比较好的例子是 NSButton，它是 NSControl 的子类，使用 NSButtonCell 完成绘制。稍微复杂一点的例子是 NSMatrix，它包含一个两维的格子矩阵。

如果你创建新的视图，那么你可能得考虑一下用格子抽象并把你的绘图代码放到一个 NSCell 子类里，然后把它们封装到一个 NSControl 的子类里。这样你就可以在表格视图、大纲视图、浏览器视图、矩阵里使用新的 UI 组件了，也可以直接把它插到视图层次里。

## 5.7 小结

本章对 AppKit 概念做了一个简要介绍。我们看到了 Cocoa 里分发的事件类型，以及是怎么分发的。我们也看到了 AppKit 是怎么以多种方式扩展 Foundation 的，以及视图层次是怎么设计的。

本章探索的所有这些概念，是 Cocoa 应用设计的中心思想。我们会在后面的章节更加详细地了解如何运用它们。我们已经了解了事件是怎么发送给屏幕上的视图，然后再到模型对象，以及这又是怎么触发图形绘制的。