Carbon-Cocoa 集成指南概述

无论您在开发应用程序时选择哪一种开发环境——Cocoa 或 Carbon——您可能会发现，另一种开发环境提供了您想要在应用程序中使用的功能。选择 Cocoa 或 Carbon 开发环境来创建新的应用程序并没有限制您必须使用该环境中定义的 API。您可以在 Cocoa 应用程序中使用 Carbon 的 API，也可以在 Carbon 应用程序中使用 Cocoa 的 API。本文档将向您介绍具体的使用方法。

出于许多原因，您可能希望在应用程序中集成 Cocoa 和 Carbon，如：

- 您既想使用现有的代码，又想利用另一框架提供的技术。
- 您正在开发一个对 Carbon 和 Cocoa 都可用的公共服务。
- 对于某些任务来说，在 Cocoa 环境下比在 Carbon 环境下更易于完成；而对于另一些任务来说，情况正相反。
- 您已经在一种环境下创建了一个很棒的用户界面，但您想通过另一种环境访问该界面。
- 您的程序设计团队由具有不同技能——Cocoa 和 Carbon——的工程师组成。

谁应该阅读本文档？

本文档假定您使用Objective-C进行Cocoa编程，并且不讨论有关Java集成的问题。为了充分理解本文档，您最好具备Cocoa或Carbon环境下的编程经验，并且对其它编程环境的基础知识也有所了解。“参考”部分包含了能够帮助您了解这些知识的文档列表。

本文档的组织结构

在程序中集成 Cocoa 和 Carbon 之前，您应该熟悉一些基本概念。这些概念会在以下文章中详细介绍：

- “Carbon和Cocoa用户接口间通信”讨论了Mac OS X如何在Carbon和Cocoa应用程序环境之间传递用户事件。
- “混合语言代码的预处理”列出了当您在一个项目中混合使用不同的编程语言时，可使用的文件扩展名。
- “可交互的数据类型”提供了有关Foundation（Cocoa）和Core Foundation（Carbon）之间可以交换使用的数据类型的信息。
- “在应用程序中同时使用 Carbon 和 Cocoa”提供了在一个应用程序中同时使用 Carbon 和 Cocoa 代码的简要介绍。包括对封装成 C 接口的函数的介绍。

在上述文档中讨论的概念将实际应用于下列文档提供的示例代码中：

- “在Carbon程序中使用Cocoa功能”介绍了如何在Carbon应用程序中使用与用户接口无关的Cocoa功能。

- “在Cocoa程序中使用Carbon功能”介绍了如何在Cocoa应用程序中使用与用户接口无关的Carbon功能。
- “在Carbon程序中使用Cocoa用户接口”介绍了怎样才能使Cocoa用户接口在Carbon中正常工作。
- “在Cocoa程序中使用Carbon用户接口”介绍了怎样才能使Carbon用户接口在Cocoa中正常工作。
- “HICocoaView：在Carbon窗口中使用Cocoa视图”介绍了在Mac OS X v10.5 中引入的HICocoaView是如何使在Carbon窗口中使用Cocoa视图成为可能的。
- “在导航服务对话框中使用Cocoa”介绍了使用Mac OS X v10.5 中“导航服务”的Carbon应用程序是如何能够直接访问Cocoa类`NSOpenPanel`和`NSSavePanel`的功能的。

参考

关于开发 Mac OS X 应用程序，特别是 Cocoa 和 Carbon 应用程序的更多信息，请参考下列文档：

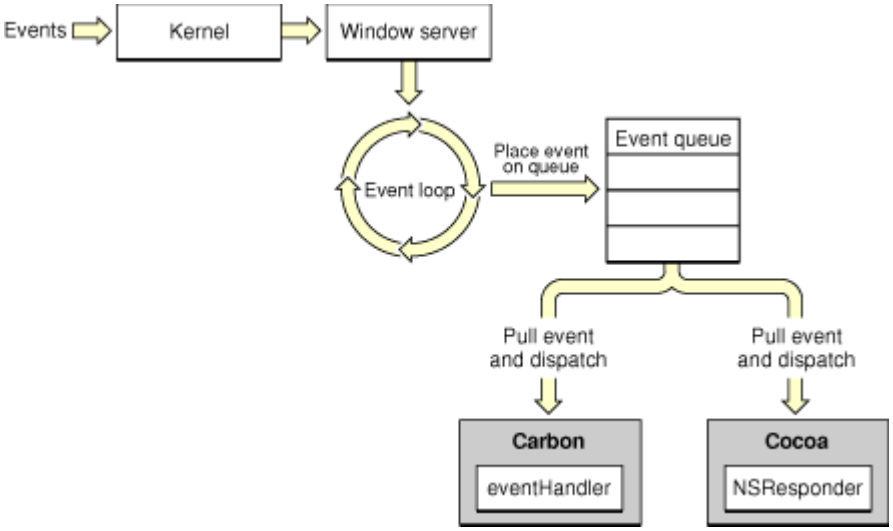
- *Cocoa入门初步*和*Carbon入门初步*分别为不熟悉Cocoa和Carbon的开发人员提供了相关的指导性介绍以及学习途径。
- *工具和语言初步* 为不熟悉 Apple 集成开发环境（IDE）的开发人员提供了相关的指导性介绍以及学习途径。
- *Mac OS X 技术概述*对可用于 Mac OS X 的技术进行了大体上的介绍，并提供了相关文档的链接。附录 A“Mac OS X 框架”列出了对 Mac OS X 开发人员可用的框架。
- *Cocoa 内存管理编程指南*介绍了对象所有权策略及其相关的用于创建、复制和销毁对象的技术。

Carbon 和 Cocoa 用户接口间的通信

在 Mac OS X 的版本 10.2 之前，由于用户事件生成以及传递至应用程序的方式不同，在 Cocoa 应用程序中使用 Carbon 窗口或在 Carbon 应用程序中使用 Cocoa 窗口是一项充满挑战的工作。本文概要地介绍了用户事件的处理过程，并大致描述了 Carbon 和 Cocoa 是如何向彼此传递用户事件的。

图 1 显示了一个用户事件（如单击或按键）通过系统到达 Carbon 和 Cocoa 应用程序环境的路径。当控制输入设备（如鼠标）的驱动程序检测到用户操作，则产生一个用户事件，并将其传递给窗口服务器。

图 1 用户事件在 Mac OS X 中的路径

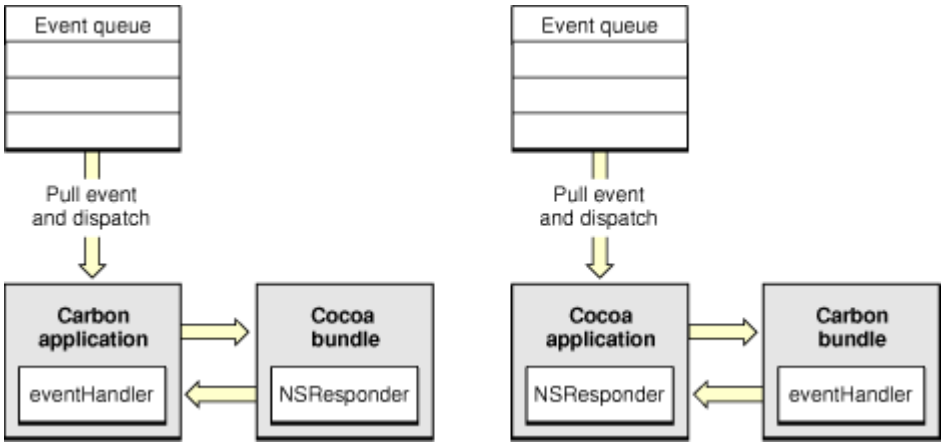


当窗口服务器接收到该事件时，它将查询当前所有打开的窗口。然后，窗口服务器会将事件发送到运行循环的事件端口，该运行循环隶属于事件发生的窗口所在的进程。事件管理器从运行循环端口处获取事件，并以适当的格式将事件打包，再将其传递给进程的应用程序环境所特有的事件处理机制。该机制确保了事件能够被与点击的控件（或按下的按键）相关联的函数或方法所处理。

在 Mac OS X 的版本 10.2 之前，事件被传递给进程的应用程序环境。所以，当一个 Carbon 应用程序试图使用 Cocoa 窗口时，Cocoa 窗口对应的事件会被传递给 Carbon 应用程序。但由于 Carbon 应用程序没有的事件的处理函数，也无法将事件传递给创建窗口的 Cocoa 环境，因此该事件将被丢弃。反过来，对于一个 Cocoa 应用程序中的 Carbon 窗口来说也是这样。Carbon 窗口对应的事件会被传递给 Cocoa 应用程序，但 Cocoa 应用程序没有该事件的处理函数，也无法将该事件传递给 Carbon 环境。

从 Mac OS X 的版本 10.2 开始，系统会自动安装相应的处理函数，从而可以在 Cocoa 和 Carbon 两种环境之间传递事件。图 2 显示了对于使用 Cocoa 资源的 Carbon 应用程序（图中左侧）和使用 Carbon 资源的 Cocoa 应用程序（图中右侧），Carbon 和 Cocoa 环境之间的通信路径。
让我们首先看一看，在用于 Carbon 应用程序的 Cocoa 窗口中，用户事件的传递是如何进行的。系统会自动为 Cocoa 窗口的 `WindowRef` 对象安装 Carbon 事件处理函数。当一个用户事件（例如，点击按钮）在 Cocoa 窗口中发生时，该用户事件被传递给 Carbon 应用程序。Carbon 应用程序将用户事件分派给对应于 Cocoa 窗口的 `WindowRef` 对象的事件处理函数。由系统提供的事件处理函数知道如何将事件打包为一个 Cocoa 的 NSEvent 对象，然后会将该 NSEvent 对象传递 Cocoa 窗口通过正常的 Cocoa 事件处理机制来处理，包括不针对某个具体控件的事件响应链。就点击按钮而言，按钮会接收到点击按钮的事件，并使用正常的 Cocoa 目标动作机制处理该事件。

图 2 Carbon 和 Cocoa 之间的事件传递



反过来，对于用于 Cocoa 应用程序中的 Carbon 窗口，系统会自动创建一个 Cocoa 的 NSWindow 对象来代表 Carbon 窗口。当用户点击 Carbon 窗口中的按钮时，与点击按钮这一动作对应的 NSEvent 对象会传递给 Cocoa 应用程序。Cocoa 的正常事件处理机制会将该事件传递给由系统提供的对应 Carbon 窗口的 NSWindow 对象。NSWindow 对象知道如何创建一个 Carbon 事件，并将其传递给 Carbon 窗口的事件处理函数。从这儿开始，事件通过 Carbon 的事件目标层级体系被处理。

总之，由于 Mac OS X 在用户接口元素层（而不是在应用程序层）提供 Cocoa 和 Carbon 间自动转换的机制，因此 Carbon 和 Cocoa 可以共享相同的窗口。

关于应用程序环境和系统架构的更多信息，请参考 *Mac OS X 技术概述*。欲了解更多有关 Carbon 事件的信息，请参考 *Carbon 事件管理器编程指南*。欲了解更多有关 Cocoa 事件的信息，请参考 Cocoa 编程主题 —— *Cocoa 事件处理指南*。

混合语言代码的预处理

Carbon 和 Cocoa 各自的 API 使用不同的语言。Carbon 使用 C，而 Cocoa 使用 Objective-C。此外，Carbon 程序员可能更喜欢使用 C++。当您生成一个应用程序，而该程序同时使用了 C、Objective-C、Objective-C++ 和 C++文件时，您必须确保编译器会执行相应的预处理。需要编译的文件的扩展名表明了编译的类型，如表 1 所示。

表 1　文件扩展名和编译

文件扩展名	对编译器的指示
`.c`	必须预处理的 C 源代码
`.m`	Objective-C 源代码
`.mm`	Objective-C++源代码
`.h`	头文件（不会被编译或链接）
`.cc, .cp, .cxx, .cpp, .c++, .C`	必须预处理的 C++源代码

在一个 Xcode 项目中，您可以通过改变文件类型来覆盖源文件的扩展名。文件类型决定了 Xcode 预处理和编译该文件的方式。例如，假定您想要向一个以 .c 为文件扩展名的 C 源文件中添加 Objective-C 代码。您可以按照如下步骤检查和修改此源文件的文件类型：

1. 在项目窗口中选择该 C 源文件。
2. 打开“文件信息”窗口并选择“通用”面板。
3. 找到“文件类型”设置项。
4. 将其值改为“sourcecode.c.objc”。

现在，Xcode 就能够编译文件中的 Objective-C 代码了。

您也可以指示 Xcode 将项目中的所有源文件当做是以同一种语言编写的来进行处理，不管它们的文件扩展名和文件类型是什么。例如，假设您要向多个 C++源文件中添加 Objective-C 代码。为了指明编译器把所有的源文件看作 Objective-C++文件，应进行以下操作：

1. 在项目窗口的“组和文件”面板中选中该项目。
2. 打开“项目信息”窗口，选择“生成”面板。
3. 找到“将源文件编译为”设置项。
4. 将其值“根据文件类型”改为“Objective-C++”。

现在，Xcode 就能够编译您所有 C++文件中的 Objective-C 代码了。您也可以针对每个目标文件分别改变该设置。

关于完整的 Xcode 文档，请参考 ADC 参考库中工具类的文档。Xcode 用户指南也可以从 Xcode 应用程序的“帮助”菜单中获得。

可交互的数据类型

在 Core Foundation 框架（Carbon）和“基础”框架（Cocoa）中存在许多可以交互使用的数据类型。这意味着：您可以使用同样的数据结构作为 Core Foundation 函数调用的参数和 Objective-C 消息调用的接收者。例如，`NSLocale`（见 *NSLocale 类参考*）和 Core Foundation 中与之相对应的 CFLocale（见 *CFLocale 参考*）是可以互换的。 因此，在一个需要 `NSLocale *`参数的方法中，您可以传递一个 `CFLocaleRef` 实例。同样在一个需要 `CFLocaleRef` 参数的函数中，您可以传递一个 `NSLocale` 实例。您可以将一种类型强制转换为另一种类型，使得编译器不发出警告，如下面的例子所示。

```
NSLocale *gbNSLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_GB"];
CFLocaleRef gbCFLocale = (CFLocaleRef) gbNSLocale;
CFStringRef cfIdentifier = CFLocaleGetIdentifier (gbCFLocale);
NSLog(@"cfIdentifier: %@", (NSString *)cfIdentifier);
// logs: "cfIdentifier: en_GB"
CFRelease((CFLocaleRef) gbNSLocale);
```

```
CFLocaleRef myCFLocale = CFLocaleCopyCurrent();
NSLocale * myNSLocale = (NSLocale *) myCFLocale;
[myNSLocale autorelease];
NSString *nsIdentifier = [myNSLocale localeIdentifier];
CFShow((CFStringRef) [@"nsIdentifier: " stringByAppendingString:nsIdentifier]);
// logs identifier for current locale
```

通过该例我们注意到，内存管理的函数和方法也是可互换的——您可以对 Cocoa 对象使用 `CFRelease` 函数，也可以对 Core Foundation 对象使用 `release` 和 `autorelease` 方法。

注意：当使用垃圾收集机制时，对 Cocoa 对象和 Core Foundation 对象的内存管理是有重要区别的。请参考“使用带有垃圾收集机制的 Core Foundation”了解细节。

可交互使用的数据类型也被称为**无缝转换**的数据类型，自 Mac OS X 的 v10.0 版本以来开始提供。表 1 列出了在 Core Foundation 和 Foundation 之间可互换的数据类型的清单。该表还列出了每一对数据类型的可以无缝转换时，对应的 Mac OS X 的版本号。

表 1　Core Foundation 和 Foundation 之间可互换使用的数据类型

Core Foundation 类型	Foundation 类	可用版本
CFArrayRef	NSArray	Mac OS X v10.0
CFAttributedStringRef	NSAttributedString	Mac OS X v10.4
CFCalendarRef	NSCalendar	Mac OS X v10.4
CFCharacterSetRef	NSCharacterSet	Mac OS X v10.0
CFDataRef	NSData	Mac OS X v10.0
CFDateRef	NSDate	Mac OS X v10.0
CFDictionaryRef	NSDictionary	Mac OS X v10.0
CFErrorRef	NSError	Mac OS X v10.5
CFLocaleRef	NSLocale	Mac OS X v10.4
CFMutableArrayRef	NSMutableArray	Mac OS X v10.0
CFMutableAttributedStringRef	NSMutableAttributedString	Mac OS X v10.4
CFMutableCharacterSetRef	NSMutableCharacterSet	Mac OS X v10.0
CFMutableDataRef	NSMutableData	Mac OS X v10.0
CFMutableDictionaryRef	NSMutableDictionary	Mac OS X v10.0

CFMutableSetRef	NSMutableSet	Mac OS X v10.0
CFMutableStringRef	NSMutableString	Mac OS X v10.0
CFNumberRef	NSNumber	Mac OS X v10.0
CFReadStreamRef	NSInputStream	Mac OS X v10.0
CFRunLoopTimerRef	NSTimer	Mac OS X v10.0
CFSetRef	NSSet	Mac OS X v10.0
CFStringRef	NSString	Mac OS X v10.0
CFTimeZoneRef	NSTimeZone	Mac OS X v10.0
CFURLRef	NSURL	Mac OS X v10.0
CFWriteStreamRef	NSOutputStream	Mac OS X v10.0

注意：并非所有的数据类型都是可以无缝转换的，即使它们的名字可能很相似。例如，NSRunLoop 和 CFRunLoop 之间，NSBundle 和 CFBundle 之间，以及 NSDateFormatter 和 CFDateFormatter 之间都不能无缝转换。

在应用程序中同时使用 Carbon 和 Cocoa

我们总是可以在同一应用程序中集成 Cocoa 和 Carbon 的功能，至少包括那些不涉及处理用户接口元素的功能。

在 Cocoa 应用程序中使用 Carbon

如果一个 Cocoa 应用程序想要调用 Carbon 函数，唯一的条件就是，编译器能够访问相应的头文件，并且应用程序链接合适的框架。也就是说，要访问 Carbon 功能，您只需导入 Carbon.h 并链接 Carbon 框架。由于 Objective-C 是 ANSI C 的超集，因此在 Cocoa 应用程序中调用 Carbon 函数是很容易的（尽管在 Mac OS X 的 v10.2 版本之前，所调用的函数不能是用户界面函数）。

在 Carbon 应用程序中使用 Cocoa

通过采取一些额外的步骤，Carbon 应用程序也可以使用许多 Cocoa 和 Objective-C 的技术。要访问 Cocoa 的功能，只需导入 Cocoa.h 并链接 Cocoa 框架。要访问其它 Objective-C 技术，您可能需要导入额外的头文件并链接相应的框架。例如，为了使用 Web 工具包，则需要导入 WebKit.h 并链接 WebKit 框架。

您还需要采取以下步骤：

- 通过调用 NSApplicationLoad 函数，使得 Carbon 应用程序可以使用 Cocoa。通常，您应该在执行任何其它 Cocoa 代码之前，在主函数中调用该函数。
- 在您要使用Cocoa的函数中，分配并初始化一个NSAutoreleasePool对象，并且在不再需要它时释放该对象。请注意，如果您的应用程序运行在Mac OS X的v10.4 或更高版本，则当您的函数被调用时，toolbox已经直接或间接地建立了一个自动释放池。例如，在下列位置或情况中，一个NSAutoReleasePool对象会自动被创建并释放：
  - RunApplicationEventLoop
  - RunAppModalLoopForWindow
  - ModalDialog
  - 对窗口的拖动、大小调整的跟踪
  - 对控件的跟踪和对提示符的拖动
  - Event dispather target 对事件的交叉调度
  - 当窗口的 compositing 视图被重绘时
- 使用 Objective-C 编译器编译您项目中使用 Cocoa 的部分。在 Xcode 中，有几种方法都可以做到这一点：
  - 使用 Objective-C 文件扩展名。
  - 使用“文件信息”面板将源文件的文件类型设置为 sourcecode.c。objc。
  - 将项目或目标生成设置由“将源文件编译为”改为 Objective-C。

封装成 C 接口的函数

当我们要将 Objective-C 代码集成到 Carbon 项目中时，常见的方法是为代码的 Objective-C 部分编写 C 接口的包装函数（或者只是 C 包装函数）。在 Carbon 和 Cocoa 集成的上下文环境中，**C 接口的封装函数**是一个 C 函数，其函数体包含 Objective-C 代码，并且该函数可以向 Cocoa 传递或从 Cocoa 获取数据。这些 C 封装函数可以放在一个单独的 Objective-C 源文件中，并使用 Objective-C 编译器进行编译。

让我们来看一个典型场景，在这个场景中，一个 Carbon 应用程序要访问由 Cocoa 源文件提供的功能。Cocoa 源文件必须所必需的所有类和方法的 Objective-C 代码。同时也必须为每一个会被 Carbon 应用程序所调用的功能封装一个 C 接口函数。例如，对于 changeText——一个接收字符串参数并对其进行某种操作的方法，相应的 C 接口的封装函数类似于下面这样：

```
OSStatus changeText (CFStringRef message)
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
```

```
    [[Controller sharedController] changeText:(NSString *)message];
    [localPool release];
    return noErr;
}
```

注意：C 封装函数通常分配并初始化一个 NSAutoreleasePool 对象，然后，当不再需要它时释放该对象，如 changeText 函数所示。对于由 Carbon 应用程序直接调用的 C 封装函数来说，这是它们能够实现正确的内存分配的一个必要条件。

最后，总结一下如何使用 C 接口的封装函数以允许 Carbon 应用程序访问 Cocoa 功能：

1. Objective-C 源文件提供了对 Cocoa 功能的访问。该文件针对每一个 Carbon 应用程序所需的 Cococa 功能，封装了相应的 C 接口函数。
2. Carbon 应用程序根据需要调用 C 封装函数。

在 Carbon 应用程序中使用 Cocoa 功能

本文介绍 Carbon 应用程序如何使用与用户界面无关的 Cocoa 功能。在 Mac OS X 的 v10.1 及更高版本中，您可以在 Carbon 应用程序中访问 Cocoa 的功能。为此，您只需要做两件事情：

- 对于每一项您想要在Carbon应用程序中使用的Cocoa方法，都要为该Cocoa方法编写一个封装的C接口函数。更多细节请参考“编写Cocoa源代码”部分。
- 编写调用C接口封装函数的Carbon代码，该封装函数负责初始化Cocoa。更多细节请参考“在Carbon应用程序中调用C封装函数”部分。

以下各节所介绍的任务都用示例代码加以说明，这些示例代码取自一个叫做“拼写检查器”的工作程序。该示例程序使用了Cocoa的拼写检查功能。“关于拼写检查器应用程序”部分对该应用程序作了详细介绍。您可以下载*SpellingChecker-CarbonCocoa*的代码。

虽然在本文的清单中显示了拼写检查器程序的大量代码，但并未包含和解释所有的程序代码。例如，完成处理 Carbon 窗口功能的代码并没有在文中列出。要确切地了解 Carbon 和 Cocoa 的各部分是如何组合在一起的，您应该下载完整的项目。

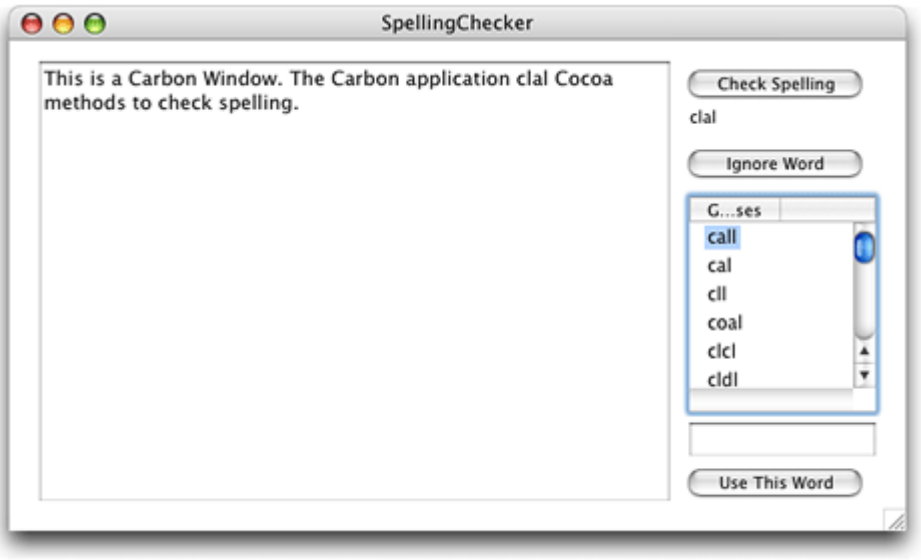
关于拼写检查器应用程序

拼写检查器是一个Carbon示例程序，为窗口中输入的文本提供了拼写检查功能。其用户界面如图 1所示。拼写检查器的窗口是一个由Interface Builder创建的Carbon窗口。用户可以在窗口左侧的大文本框中输入文本。

如果需要检查拼写，用户应点击“拼写检查”按钮。第一个拼错的单词会在按钮的下方显示出来，如图 1 所示（“clal”）。并且程序会在“猜测”列表中给出相应的替换单词的建议。此时，用户可以选择：

- 点击“忽略单词”按钮，忽略拼错的单词。
- 在猜测列表中选中并双击一个单词，来替换拼错的单词。
- 输入一个单词并点击“使用该单词”按钮，来指定使用另一个单词。

图 1 拼写检查器程序的用户界面



拼写检查功能由 Cocoa 框架提供，通过封装的 C 接口函数进行访问，但这一功能是在 Carbon 应用程序中被调用的。

注意：示例程序中的文本框是一个 Unicode 文本编辑控件。对于更复杂的应用程序，最好使用由多语言文本引擎（MLTE）API 提供的文本编辑能力。

编写 Cocoa 源代码

编写 Cocoa 源代码需要执行以下各节所介绍的任务：

- “使用Xcode创建新的Cocoa源文件”
- “标识Cocoa方法”
- “编写C可调用的包装函数”

使用 Xcode 创建新的 Cocoa 源文件

想要使用 Xcode 创建一个 Cocoa 源文件，请执行下列操作：

1. 在 Xcode 中打开您的 Carbon 项目。
2. 选择“文件” > “新建文件”。
3. 在“新建文件”窗口中选择“项目中的空文件”，然后单击“下一步”按钮。
4. 为该文件命名，使其扩展名为 `.m`。示例代码的文件名是 `SpellCheck.m`。

   在“混合语言代码的预处理”中介绍过，`.m`扩展名向编译器表明这是Objective-C代码。
5. 将以下语句添加到您的新文件当中：

```
#include <Carbon/Carbon.h>
#include <Cocoa/Cocoa.h>
```

只要您使用 Xcode 创建源文件，您就不需要修改生成设置和属性列表的值了。

标识 Cocoa 方法

您需要标识出哪些 Cocoa 方法提供了您的 Carbon 应用程序所需要的功能。对于每一个标识出的方法，您将需要编写一个封装的 C 接口函数。

拼写检查器程序需要下列方法所提供的功能：

- `uniqueSpellDocumentTag` 返回一个文档标记。该标记应保证是唯一的。为每个文档使用一个标记来确保对该文档的拼写检查操作是唯一的。
- `checkSpellingOfString:startingAt:`从指定的位置开始，在字符串中查找拼错的单词。此方法返回第一个拼错单词的范围。
- `checkSpellingOfString:startingAt:language:wrap:inSpellDocumentWithTag:wordCount:`指定起始位置和若干其它选项，在字符串中查找拼错的单词。此方法返回第一个拼错单词的范围。
- `ignoreWord:inSpellDocumentWithTag:`向一个单词列表中添加单词，该列表中的单词在进行文档的拼写检查时都被忽略。
- `setIgnoredWords:inSpellDocumentWithTag:`设定一个文档中被忽略单词的列表。
- `ignoredWordsInSpellDocumentWithTag:`返回文档中被忽略单词的数组。
- `guessesForWord:`返回对拼错单词的拼写建议的数组。
- `closeSpellDocumentWithTag:`关闭文档时调用该方法，以确保该文档的被忽略单词的列表被清空。

更多相关信息请参考 *NSSpellChecker 类参考*。

您还需要标识出实现 Cocoa 功能所需的其它方法。例如，类方法 `sharedSpellChecker` 返回一个 NSSpellChecker 实例。

编写封装的 C 接口函数

在标识出能够提供您所需功能的 Cocoa 方法之后，您需要为这些方法编写封装的 C 接口函数。

对于拼写检查器程序，有 8 个Cocoa方法（请参考"标识Cocoa方法"部分）提供了管理和检查文档中拼写的功能。为了使应用程序中的Carbon部分能够访问Cocoa方法，您需要编写封装的C接口函数并将它们放在Cocoa源文件中。同时，您也需要在一个共享的头文件中声明这些函数。表 1 列出了拼写检查器程序中封装的C接口的函数的名称。

表 1　与 Cocoa 方法对应的封装的 C 接口函数

封装的 C 接口函数	Cocoa 方法
UniqueSpellDocumentTag	uniqueSpellDocumentTag
CloseSpellDocumentWithTag	closeSpellDocumentWithTag:
CheckSpellingOfString	checkSpellingOfString:startingAt:
CheckSpellingOfStringWithOptions	checkSpellingOfString:startingAt: language:wrap:inSpellDocumentWithTag: wordCount:
IgnoreWord	ignoreWord:inSpellDocumentWithTag:
SetIgnoredWords	setIgnoredWords:inSpellDocumentWithTag:
CopyIgnoredWordsInSpellDocumentWithTag	ignoredWordsInSpellDocumentWithTag:
GuessesForWords	guessesForWord:

程序清单 1 显示了封装的 C 接口函数 UniqueSpellDocumentTag。请注意自动释放池部分的代码。对于一个被 Carbon 应用程序使用的 Cocoa 方法，您必须在每次使用它时建立一个自动释放池。

程序清单 1 uniqueSpellDocumentTag：方法的 C 可调用的包装函数

```
int     UniqueSpellDocumentTag ()
{
    int  tag;

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    tag  = [NSSpellChecker uniqueSpellDocumentTag];
    [pool release];

    return (tag);
}
```

拼写检查器程序的封装的其他的 C 接口函数均按照清单 1 所示的方式编写，并使用以下准则：

- C 接口函数的参数必须与 Cocoa 方法所需的参数匹配。例如，在程序清单 2 中，C 接口函数的参数 stringToCheck 和 startingOffset，与 checkSpellingOfString:startingAt:方法所需的两个参数相匹配。
- C 接口函数必须分配和初始化一个 NSAutoreleasePool 对象，然后在不再需要它时释放该对象。这是在 Carbon 程序使用 Cocoa 方法的要求。您可以在程序清单 1 和程序清单 2 中看到这方面的例子。
- C 接口函数必须返回被它封装的 Cocoa 方法所返回的数据。例如，程序清单 1 中的 UniqueSpellDocumentTag 函数返回从 uniqueSpellDocumentTag 方法获得的标记值；程序清单

2 中的 CheckSpellingOfString 函数返回从 checkSpellingOfString:startingAt:方法获得的范围。

- 在适当情况下，C 接口函数还可以使用无缝转换的数据类型。例如，程序清单 2 中的 C 接口函数将一个 CFStringRef 类型的值作为参数，但当该函数向 Cocoa 方法传递字符串时，就将其强制转换为 NSString *类型。

程序清单 2 封装 checkSpellingOfString:startingAt:方法的 C 接口函数

```
CFRange  CheckSpellingOfString (CFStringRef stringToCheck,
                    int startingOffset)
{
    NSRange range = {0,0};

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    range = [[NSSpellChecker sharedSpellChecker]
                checkSpellingOfString:(NSString *) stringToCheck
                startingAt:startingOffset];
    [pool release];
    return ( *(CFRange*)&range );
}
```

您还需要创建一个包含封装的 C 接口函数的声明的头文件，并且该头文件可以被包含在适当的源文件中。

拼写检查器程序所需的 C 接口函数的其余代码在一个名为 SpellCheck.m 的 Cocoa 源文件中。您可以下载代码 *SpellingChecker-CarbonCocoa*。

在 Carbon 应用程序中调用 C 接口函数

您可以根据需要，在 Carbon 应用程序中使用封装的 C 接口函数。程序清单 3 显示了如何通过您的 Carbon 应用程序的事件处理程序调用 C 接口函数（CheckSpellingOfString）。（您可以从开发人员示例代码的网站下载拼写检查器程序，结合上下文阅读这段代码。）程序清单后面对有编号的代码行进行了详细解释。

程序清单 3 在您的 Carbon 应用程序中调用 C 接口函数

```
if (command.commandID == 'Spel') // 1
{
    GetControlByID (window, &controlID, &control); // 2
    err = GetControlData (control, 0,
              kControlStaticTextCFStringTag,
              sizeof(CFStringRef),
              &stringToSpellCheck,
```

```
            &count); // 3
    if (err == noErr)
    {
        windowInfo->range = CheckSpellingOfString (stringToSpellCheck, 0); // 4
        if (windowInfo->range.length > 0) // 5
            SetMisspelledWord (window, stringToSpellCheck, &windowInfo->range);
        else
            windowInfo->range.location = 0;
    }
}
```

下面解释一下代码的作用：

1. 检查用户点击“拼写检查”按钮时所发出的 Command ID 是否是事先声称的 ID。
2. 调用控件管理函数`GetControlByID`以获得Unicode文本编辑控件的`ControlRef`。它是一个文本框，包含由用户输入且需要进行拼写检查的文本。见图 1。
3. 调用控件管理函数 `GetControlData` 以获得用户在文本框中输入的字符串。
4. 调用 C 接口函数 `CheckSpellingOfString`。还记得吗，这个 C 接口函数包装了 Cocoa 方法 `checkSpellingOfString:startingAt:`。
5. 如果发现拼错的单词，就使用从 C 接口函数返回的位置信息来设置拼错单词的位置。

在 Cocoa 程序中使用 Carbon 功能

由于 Objective-C 是 ANSI C 的一个超集，所以在 Cocoa 应用程序中调用不涉及用户接口的 Carbon 函数是一件轻而易举的事。一个 Cocoa 程序总是可以调用底层的 Carbon 函数，因为 Cocoa 已经链接了 Application Services 框架。要使用高层的 Carbon 函数，Cocoa 应用程序必须导入 `Carbon.h` 并链接 Carbon 框架。

以下各节介绍了几种可以在 Cocoa 应用程序中使用 Carbon 函数（非用户接口函数）的情况：

- “处理QuickTime影片”
- “在Cocoa中访问Resource Fork”
- “使用FSRef数据类型”
- “在Cocoa中管理Core Foundation对象”

在Cocoa程序中可以调用Carbon函数的情况并不仅限于本文中所介绍的例子。这些例子只是为您介绍了几种可在Cocoa程序中使用非UI的Carbon函数的方法。如果您想要在Cocoa程序中使用Carbon接口，请阅读“在Cocoa程序中使用Carbon用户接口”部分。

处理 QuickTime 影片

Cocoa 的 NSMovieView 类能够在框架中显示一个 NSMovie 对象（对 QuickTime 影片的包装），并提供了播放和编辑影片的方法。虽然已经有了编辑、设置视图大小、设置控制器、设置播放模式及处理声音的方法，但 NSMovieView 类中的方法不能提供可用于处理 QuickTime 影片的全部功能。例如，假如有多个语言轨道可用，没有设置 QuickTime 影片语言的方法。然而，在 Cocoa 应用程序中，您可以调用任意 QuickTime 的函数，比如 `SetMovieLanguage`。您需要做的只是调用 QuickTime 函数 `EnterMovies` 来初始化影片工具箱。

关于程序中可调用的 QuickTime 函数的更多信息请参考 *QuickTime 框架参考*。

程序清单 1 显示了在 Cocoa 方法中调用 QuickTime 函数的一个例子。清单中的代码为 QuickTime 影片生成了一个轨道媒体类型的数组。轨道媒体类型被显示在影片属性窗口的 NSTableView 控件中。清单后面对有编号的代码行进行了详细解释。

清单 1 为 QuickTime 影片生成轨道媒体类型的数组

```
// Before you call QuickTime functions you must initialize the
// Movie Toolbox by calling the function EnterMovies();


- (void) myBuildTrackMediaTypesArray:(NSMovie *) movie
{
    short i;
    Movie qtmovie = [movie QTMovie];

    for (i = 0; i < GetMovieTrackCount (qtmovie); ++i)                // 1
    {
        Str255  mediaName;
        OSErr   myErr;
        Track   movieTrack = GetMovieIndTrack (qtmovie, i+1);          // 2
        Media   trackMedia = GetTrackMedia (movieTrack);              // 3
        MediaHandler trackMediaHandler = GetMediaHandler(trackMedia);

        myErr = MediaGetName (trackMediaHandler, mediaName, 0, NULL);   // 4
        [myMovieTrackMediaTypesArray insertObject:[
                    NSString stringWithCString:&mediaName[1]
                    length:mediaName[0]]
                atIndex:i];                                       // 5
    }
}
```

下面解释一下代码的作用：

1. 调用 QuickTime 函数 `GetMovieTrackCount` 来获得影片中轨道的数量。
2. 调用 QuickTime 函数 `GetMovieIndTrack` 来确定某一轨道的轨道标识符。请注意，轨道的编号从给定值 1 开始。
3. 调用 QuickTime 函数 `GetTrackMedia` 来获得媒体结构，其中包含了该轨道的样本数据。
4. 调用 QuickTime 函数 `MediaGetName` 来获得媒体类型的名称（`Str255`）。
5. 将媒体名称作为 NSString 添加到 NSArray 数组中。

更多相关信息，请参考 *NSMovieView 类参考*。

在 Cocoa 中访问 Resource Fork

要和遗留文件打交道的 Cocoa 程序可能需要读入 Mac OS 9 文件的 Resource Fork，并解析资源数据。资源管理器是一个 Carbon API，因此您可以在您的 Cocoa 代码中调用合适的函数，如程序清单 2 所示。一旦读取了数据，您就可以调用 Cocoa 方法 `stringWithCString:length:`来获得一个 NSString 变量，然后对其进行解析。

清单 2 在 Cocoa 应用程序中调用资源管理函数

```
FSRef ref;
NSString* theFilePath;    // the full path of the resources file
if (FSPathMakeRef ([theFilePath fileSystemRepresentation], &ref, NULL)
    == noErr)
{
    short res = FSOpenResFile (&ref, fsRdPerm);
    if (ResError() == noErr)
    {
        // Code that calls Resource Manager functions to read resources
        // goes here.
        CloseResFile(res);
    }
}
```

使用 FSRef 数据类型

Carbon 中的文件管理器使用 `FSRef` 数据类型来指定一个文件或者一个目录的名称和位置。当您在 Cocoa 应用程序中调用 Carbon 函数时，您可能需要将 `FSRef` 变量作为参数传递给其中的一个函数，或者您可能会接收到作为返回值而返回的 `FSRef` 变量。例如，如果您调用别名管理函数 `FSResolveAliasFile`，您

必须提供一个标识别名的 FSRef 变量。FSRef 是一个不透明的 80 字节的结构，所以通常采用指针来传递 FSRef 参数。

如果已经给定一个已存在文件的路径，您可以使用程序清单 3 中的代码来获得标识该文件的 FSRef 变量。程序清单后面对有编号的代码行进行了详细解释。

清单 3 将路径转换为 FSRef 的 Cocoa 方法

```
- (BOOL) myMakeFSRef:(FSRef *) outFSRef fromPath:(NSString *)inPath
{
    OSStatus status = noErr;
    status = FSPathMakeRef ([inPath fileSystemRepresentation],
                            outFSRef,
                            NULL); // 1
    return status == noErr; // 2
}
```

下面解释一下代码的作用：

1. 调用文件管理函数 FSPathMakeRef 将某个路径转换为 FSRef 类型。outFSRef 参数必须指向一个实际的 FSRef 结构。
2. 如果转换成功，则返回 YES。

如果已经给定一个已存在文件的 FSRef 变量，您可以使用程序清单 4 中的代码来获得一个相应的 URL。程序清单后面对有编号的代码行进行了详细解释。

清单 4 将 FSRef 转换为 URL 的 Cocoa 方法

```
- (NSURL *) myCreateURLFromFSRef:(FSRef *)inFSRef
{
    NSURL* url = nil;
    UInt8 path[PATH_MAX];
    OSStatus status = noErr;

    status = FSRefMakePath (inFSRef, (UInt8*)path, sizeof(path)); // 1
    if (status == noErr) {
        url = [NSURL fileURLWithPath: [NSString stringWithUTF8String:path]]; // 2
    }
    return url; // 3
}
```

下面解释一下代码的作用：

1. 调用文件管理函数 FSRefMakePath 将 FSRef 变量转换为一个路径。
2. 使用该路径创建一个新的 NSURL 对象。

3. 如果转换成功，则返回一个 NSURL 对象。

关于文件管理的完整文档，请参考*文件管理器参考*。

在 Cocoa 中管理 Core Foundation 对象

Cocoa与Core Foundation使用类似的内存分配规则来分配、保留和释放对象。一般来说，在名称中含有Copy或Create字样的Core Foundation函数的返回值必须由调用者释放，而其它函数的返回值则无须由调用者释放。采用alloc、copy或new方法创建的Cocoa对象必须由调用者释放，其它方法返回的对象则不应由调用者释放。此外，有一些数据类型可以在Carbon和Cocoa中交互使用；它们被称为无缝转换数据类型。（参考"可交互的数据类型"部分，查看可互换的数据类型的列表。）因此，您可以在同一程序的两种环境中都可以使用这些函数和方法。

下面的代码调用了 Cocoa 方法 initWithCharacters 来初始化一个新分配的 NSString 变量。在使用该字符串的代码被执行之后，您需要释放该字符串。

```
NSString *str = [[NSString alloc] initWithCharacters: ...];
// Your code that uses the string goes here.
[str release];
```

您可以通过调用下面的 Carbon 代码，获得相同的结果。这段代码使用了 Core Foundation 函数 CFStringCreateWithCharacters。

```
CFStringRef str = CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
CFRelease (str);
```

下面的代码调用了 Core Foundation 函数 CFStringCreateWithCharacters，将返回的字符串强制转换为 Cocoa 的 NSString 类型，并使用 Cocoa 的方法 release 释放了该字符串。

```
NSString *str = (NSString *) CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
[str release];
```

同样，下面的代码混合使用了 Core Foundation 和 Cocoa，但调用了 Cocoa 的方法 autorelease 来释放字符串。

```
NSString *str = (NSString *) CFStringCreateWithCharacters(...);
// Your code that uses the string goes here.
[str autorelease];
```

注意：由于在 Core Foundation 中没有自动释放的概念，相比于使用 alloc、copy 或 new 的 Cocoa 方法，Core Foundation 函数中的大部分都是 Copy 或 Create 函数。因此必须确保您的 Cocoa 代码在适当的时候，会使用 release 或 autorelease 方法，来释放 Core Foundation 对象。

更多相关信息，请参考 Cocoa 编程主题 *Cocoa 内存管理编程指南*，或 Core Foundation 编程主题 *Core Foundation 内存管理编程指南*。

在 Carbon 程序中使用 Cocoa 用户接口

从Mac OS X的 10.2 版本开始，您可以在Carbon程序中使用Cocoa用户接口。系统提供了可令Cocoa和Carbon互相传递用户事件的代码，所以您只需进行少量的操作，就能够使Cocoa用户接口在Carbon中正常工作。其中大多数操作类似于您在Carbon程序中使用非UI的Cocoa功能时所做的工作——编写C接口的封装函数并调用它们。（请参考"在Carbon程序中使用Cocoa功能"部分。）

在Mac OS X的 10.4 及更早的版本中，不支持在Carbon窗口中嵌入Cocoa的NSView类，牢记这一点是非常重要的。更多相关信息，请参考"HICocoaView：在Carbon窗口中使用Cocoa视图"。

要在 Carbon 程序中使用 Cocoa 用户界面，您需要进行两个主要操作：

- 编写一个Cocoa源文件，其中包括：您想要使用的界面、支持该界面的Cocoa方法，以及用来访问所需Cocoa功能的封装的C接口函数。"编写Cocoa源文件"小节详细介绍了这一部分的内容。
- 编写为接口提供handler的Carbon代码。详细信息请参考"设置Carbon程序以使用Cocoa接口"小节的内容。

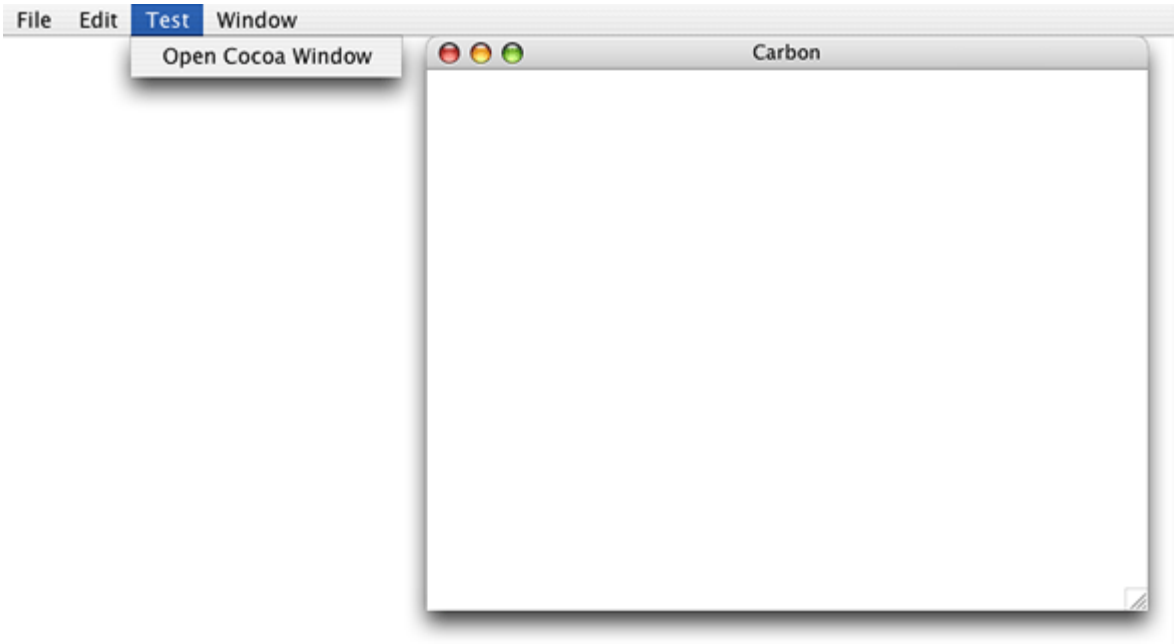
下面各节所介绍的操作都用示例代码加以说明，这些代码取自一个叫做"CocoaInCarbon"的工作程序。关于该程序的说明见"关于CocoaInCarbon程序"小节。您可以下载*CocoaInCarbon*的代码。

请记住，CocoaInCarbon 程序的许多部分是针对该示例程序的；您需要根据自己的目的重新编写这些部分的代码。虽然本文的程序清单中显示了 CocoaInCarbon 程序的大部分代码，但并非该程序的所有代码都包括在其中。

关于 CocoaInCarbon 程序

当您在阅读后面小节中出现的代码时，事先了解一下 CocoaInCarbon 程序如何工作和它的用户接口是什么样子，可能会对您有所帮助。当程序启动时，会出现一个空的窗口。如图 1 所示，该窗口是一个在 Interface Builder 创建的 nib 文件中定义的 Carbon 窗口。该程序提供了一个"测试"菜单，菜单中包含一个"打开 Cocoa 窗口"命令。

图 1 CocoaInCarbon 程序的 Carbon 用户界面



当用户在“测试”菜单中选择“打开 Cocoa 窗口”时，程序调用封装的 C 接口函数打开一个 Cocoa 窗口（如图 2 所示）并使其成为活动窗口。

图 2 CocoaInCarbon 应用程序中的 Cocoa 窗口

当用户点击图 2 中的按钮时，Carbon 应用程序就接收到了这一鼠标事件，并自动将其调度给 Cocoa。具体来说，按钮接收鼠标事件并发送它的动作方法。该动作方法会触发由 Carbon 程序提供的按钮命令处理程序。按钮命令处理程序调用封装的 C 接口函数，将文本“button pressed！”发送到按钮下面的文本框中，如图 2 所示。

尽管这个示例程序显然是一个人为的例子，但它为我们展示了 Carbon 和 Cocoa 可以在何种程度上互相通信。

编写 Cocoa 源文件

编写 Cocoa 源文件需要执行以下各节所描述的操作：

- “使用Xcode创建Cocoa源文件”
- “编写公共头文件”
- “实现控制器”
- “声明控制器接口”
- “编写C接口的封装函数”
- “在Interface Builder中创建Cocoa窗口”

使用 Xcode 创建 Cocoa 源文件

您需要编写一个 Cocoa 源文件，其中包含您的 Carbon 应用程序所需的所有 Cocoa 功能。请按照下列步骤创建一个 Cocoa 源文件：

1. 在 Xcode 中打开您的 Carbon 项目。
2. 选择“文件” > “新建文件”。
3. 在“新建文件”窗口中选择“在项目中的空文件”，点击“下一步”按钮。
4. 为该文件命名，使其扩展名为 .m。示例代码的文件名为 Controller.m。

   在“混合语言代码的预处理”中介绍过，.m扩展名向编译器表明这是Objective-C代码。
5. 创建其它任何您的应用程序所需的文件。例如，CocoaInCarbon 程序的 Cocoa 源文件有一个接口文件 Controller.h。您必须创建该文件，并通过向 Controller.m 文件中添加导入声明将其导入到 Controller.m 文件中。

只要您使用 Xcode 创建源文件，您就不必修改生成设置和属性列表的值了。

编写公共头文件

Cocoa 源文件和 Carbon 源文件都需要包含一份头文件的副本，在该头文件中声明了用于两者之间传递事件的常量。定义 CocoaInCarbon 程序中按钮点击事件的常量就是这样的常量之一。封装的 C 接口函数的声明也可以在此共享文件当中。程序清单 1 显示了头文件（CocoaStuff.h）的内容，CocoaInCarbon 项目的 Cocoa 源文件和 Carbon 源文件都必须包含该头文件。

清单 1 Cocoa 和 Carbon 公共头文件的内容

```
enum {
    kEventButtonPressed = 1
};

//Declare the wrapper functions
OSStatus initializeCocoa(OSStatus (*callBack)(int));
OSStatus orderWindowFront(void);
OSStatus changeText(CFStringRef message);
```

实现控制器

程序清单 2 显示的是 CocoaInCarbon 程序中的代码，这部分代码实现了程序中 Cocoa 源文件的控制器。其中的关键项是 NSApplicationLoad 函数（行编号为 2）的使用。Carbon 应用程序无法访问 Cocoa 界面，除非在应用程序中包含对 NSApplicationLoad 的调用。该函数并不是 Cocoa 程序所需要的，但对于

使用 Cocoa 的 Carbon 应用程序来说是必须要使用的。NSApplicationLoad 函数自 Mac OS X 的 10.2 版本开始可用。NSApplicationLoad 应该在 Carbon 初始化之后被调用。
程序清单后面对清单中有编号的代码行进行了详细解释。

清单 2 实现 Cocoa 源文件中的控制器

```
#import "Controller.h"

static Controller *sharedController;

@implementation Controller

+ (Controller *) sharedController
{
    return sharedController;
}

- (id) init                                                    // 1
{
    self = [super init];
    NSApplicationLoad();                                        // 2
    if (![NSBundle loadNibNamed:@"MyWindow" owner:self]) {
        NSLog(@"failed to load MyWindow nib");
    }
    sharedController = self;
    return self;
}

- (void) setCallBack:(CallBackType) callBack                          // 3
{
    _callBack = callBack;
}

- (void) showWindow                                             // 4
{
    [window makeKeyAndOrderFront:nil];
}

- (void) changeText:(NSString *)text                                // 5
```

```
{
    [textField setStringValue:text];
}

- (IBAction) buttonPressed:(id)sender                                    // 6
{
    (*_callBack) (kEventButtonPressed);
}

@end
```

下面解释一下代码的作用：

1. 定义初始化Cocoa的方法。稍后您将为该方法编写一个C接口的封装函数。请参考"编写C接口的封装函数"部分。
2. 根据需要调用 NSApplicationLoad。这一入口点是使用 Cocoa API 的 Carbon 应用程序所必须的，而对于 Cocoa 程序则无必要。
3. 为控制器设置一个回调函数。该回调函数（正如您在后面所看到的那样）由 Carbon 应用程序提供，用于处理按钮点击事件。
4. 显示Cocoa窗口，使其处于活动状态并在所有窗口中处于最前面。稍后您将为该方法编写一个C接口的封装函数。请参考"编写C接口的封装函数"小节。
5. 在Cocoa窗口的文本框中显示一个字符串。稍后您将为该方法编写一个C接口的封装函数。请参考"编写C接口的封装函数"小节。
6. 定义与Cocoa窗口中的按钮关联的Interface Builder动作。它会调用回调函数（由Carbon提供）来处理按钮点击事件。接下来您需要将该动作方法链接到它的目标按钮。请参考"在Interface Builder中创建Cocoa窗口"小节。

   注意：Interface Builder 动作只是一个例子，它展示了 Cocoa 中的动作方法是如何控制 Carbon 应用程序中的东西的。该例子表明这种控制是可能的；然而，示例代码所执行的操作并不需要 Carbon。也就是说，Cocoa 本身就能够更新文本框。

声明控制器接口

程序清单 3 中显示了声明控制器接口的代码（Controller.h）。在该清单中值得注意的是_callBack 实例变量。Carbon 应用程序提供了该变量所代表的回调函数。

清单 3 声明控制器的接口

```
#import <Cocoa/Cocoa.h>
#import "CocoaStuff.h"

typedef OSStatus (*CallBackType)(int);
```

```
@interface Controller : NSObject
{
    id window;
    id textField;
    CallBackType _callBack;
}

- (void)setCallBack:(CallBackType)callBack;
- (void)showWindow;
+ (id)sharedController;
@end
```

编写 C 接口的封装函数

程序清单 4 显示了C接口的封装函数，它们属于Cocoa源文件的一部分。这些函数位于CocoaInCarbon项目的Controller.m文件中。程序清单 4 中的每个函数封装了程序清单 2中的一个方法。
代码中的关键部分之一是 NSAutoreleasePool 对象的使用。对于 Carbon 应用程序所使用的 Cocoa API，您必须根据需要建立自动释放池。程序清单后面对有编号的代码行进行了详细解释。

清单 4 Cocoa API 的 C 可调用的包装函数

```
OSStatus initializeCocoa (OSStatus (*callBack)(int))                  // 1
{
    Controller *controller;
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];                      // 2
    controller = [[Controller alloc] init];                            // 3
    [controller setCallBack:callBack];                                 // 4
    [localPool release];                                               // 5
    return noErr;
}

OSStatus orderWindowFront(void)                                        // 6
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
    [[Controller sharedController] showWindow];
```

```
    [localPool release];
    return noErr;
}

OSStatus changeText (CFStringRef message)                              // 7
{
    NSAutoreleasePool *localPool;

    localPool = [[NSAutoreleasePool alloc] init];
    [[Controller sharedController] changeText:(NSString *)message];     // 8
    [localPool release];
    return noErr;
}
```

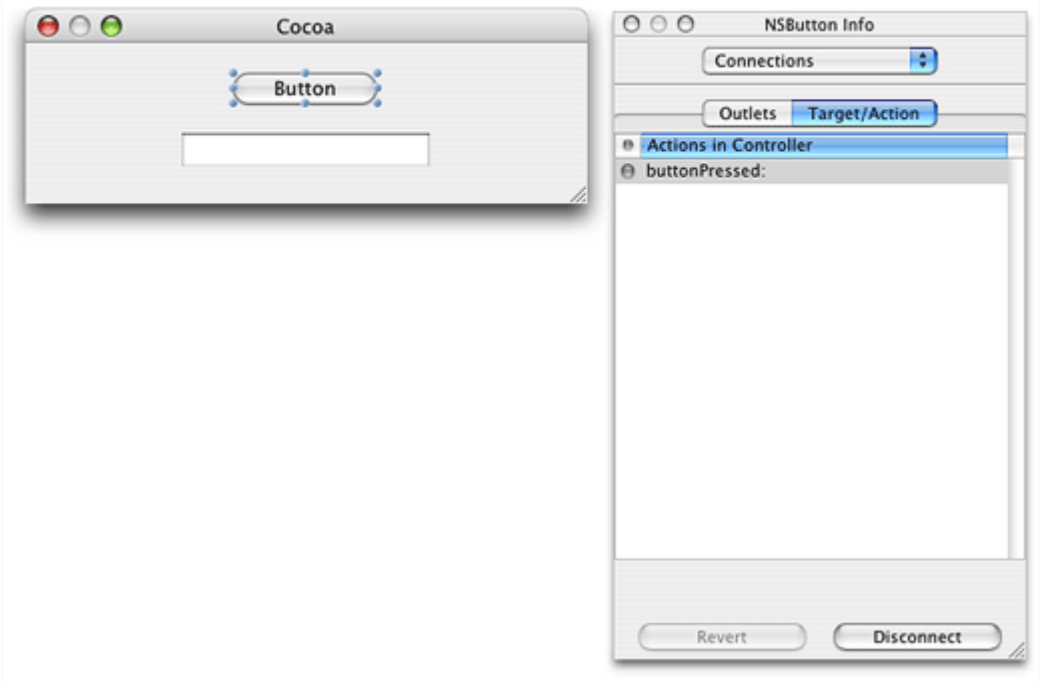
下面解释一下代码的作用：

1. 定义一个 C 接口的封装函数，Carbon 应用程序必须调用该函数来初始化 Cocoa。
2. 根据需要分配并初始化一个自动释放池。您必须为每一个封装的C接口函数执行该操作。更多相关信息，请参考“在应用程序中同时使用Carbon和Cocoa”部分。
3. 调用 `init` 方法。回想一下，该方法调用 `NSApplicationLoad`。
4. 将 Carbon 应用程序提供的回调函数赋给控制器的回调实例变量。
5. 释放本地的自动释放池。
6. 为 `showWindow` 方法定义一个 C 接口的封装函数。
7. 为 `changeText:`方法定义一个 C 接口的封装函数。
8. 将一个`CFStringRef`值强制转换为NSString *类型。在“可交互的数据类型”部分介绍过，`CFString`数据类型和NSString类是一对无缝转换类型。

在 Interface Builder 中创建 Cocoa 窗口

您必须使用Interface Builder将Cocoa窗口添加到适当的nib文件中，并为目标控件链接适当的动作。nib文件包含一组用户接口元素的定义。当示例程序中的按钮被点击时，会触发`buttonPressed`动作方法（见图3）。回顾程序清单 2可知，`buttonPressed`方法会调用指定给控制器的回调函数。在CocoaInCarbon程序中，回调函数是由Carbon程序提供的。

图 3 链接按钮与点击按钮的动作



关于在 Interface Builder 中创建 Cocoa 窗口的更多信息，请参考 *Cocoa Objective-C 应用程序开发教程*。

为使用 Cocoa 界面设置 Carbon 程序

您的 Carbon 程序必须执行一些操作才能使用由 Cocoa 提供的接口。这些操作将在以下小节介绍：

- “包含公共头文件”
- “编写命令处理程序”

包含公共头文件

无论是Cocoa源文件还是Carbon源文件，都必须包含一份公共头文件的副本，在该头文件中声明了用于传递事件的常量。程序清单 1显示了CocoaInCarbon项目中Cocoa源文件和Carbon源文件都需要包含的头文件（`CocoaStuff.h`）。详细信息请参考“编写公共头文件”部分。

编写命令处理程序

CocoaInCarbon 程序中的 Carbon 代码提供了一个函数（`handleCommand`），该函数作为参数被传递给初始化 Cocoa 的 C 接口函数。当用户点击 Cocoa 窗口中的按钮时，链接到该按钮的动作方法就会调用 `handleCommand` 函数。在 CocoaInCarbon 程序中，该方法只是向 Cocoa 方法返回一个字符串，因此实际上没必要这样做。但是，`handleCommand` 函数作为一个例子，向我们说明了一个更复杂的应用程序如何获取 Cocoa 源文件中的用户动作。

程序清单 5 显示了 `handleCommand` 函数。对已编号代码行的详细解释位于清单的下面。

清单 5 处理 Cocoa 按钮点击事件的 Carbon 函数

```
static OSStatus handleCommand (int commandID)
{
    OSStatus osStatus = noErr;
    if (commandID == kEventButtonPressed)                              // 1
    {
        osStatus = changeText (CFSTR("button pressed!"));              // 2
        require_noerr (osStatus, CantCallFunction);
    }
CantCallFunction:
    return osStatus;
}
```

下面解释一下代码的作用：

1. 检查事件以确保该事件是一个按钮点击事件，即由此函数负责处理的唯一事件。
2. 以字符串“button pressed！”为参数，调用 C 接口函数 `changeText`。

在 Cocoa 程序中使用 Carbon 用户接口

本文向您介绍如何在一个 Cocoa 程序中使用 Carbon 用户接口。在 Mac OS X 的 10.2 及更高版本中，系统提供了可令 Cocoa 和 Carbon 互相传递用户事件的代码。两种应用程序环境之间的通信使得 Cocoa 应用程序可以使用 Carbon 用户接口。

系统不支持在 Cocoa 窗口中嵌入 Carbon 控件，牢记这一点是非常重要的。

要在 Cocoa 应用程序中使用 Carbon 接口，您需要执行以下主要操作：

- “创建Carbon用户接口。”您需要使用Interface Builder创建一个Carbon窗口，并向窗口中添加控件和其它项。
- “设置Cocoa程序以使用Carbon用户接口。”您需要加载描述Carbon窗口的nib文件，创建一个NSWindow对象以便使用Cocoa方法管理Carbon窗口，并显示该窗口。

以下各节所介绍的操作都会用示例代码加以说明，这些代码取自一个叫做“CarbonInCocoa”的工作程序。关于该应用程序的说明见“关于CarbonInCocoa程序”。您可以下载*CarbonInCocoa*的代码。

虽然本文的程序清单中显示了 CocoaInCarbon 程序的大部分代码，但并非该程序的所有代码都包括在其中。您应该下载 CarbonInCocoa 的 Xcode 项目，以便更完整地了解 Cocoa 和 Carbon 的各部分是如何组合在一起的。

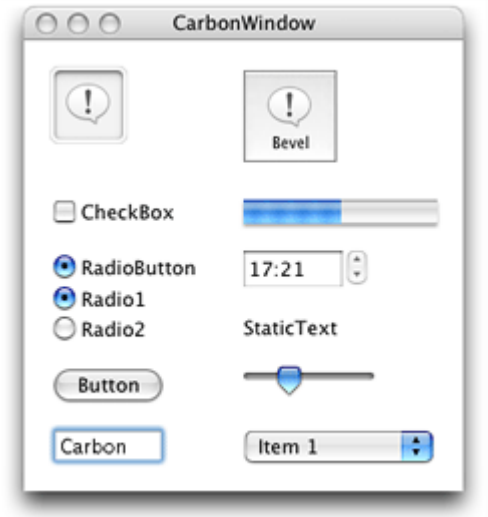
关于 CarbonInCocoa 程序

当您在阅读后面小节的内容时，事先了解一下 CocoaInCarbon 程序如何工作和它的用户接口是什么样子，可能会对您有所帮助。当程序启动时，会出现如图 1 所示的窗口。该窗口是一个由 Interface Builder 创建的 Cocoa 窗口。

图 1 CarbonInCocoa 应用程序的 Cocoa 用户界面

当用户点击“显示 Carbon 窗口”按钮时，会打开一个如图 2 所示的窗口，并试该窗口在所有窗口中处于最前面并处于活动状态。图 2 中的窗口是采用 Interface Builder 创建的 Carbon 窗口。当用户在其中的一个窗口中点击鼠标时，该窗口会变成活动窗口。用户可以在任一窗口的文本框内输入文本，可以将文本从一个窗口复制到另一个窗口，也可以从任一窗口剪切文本。

图 2 CarbonInCocoa 程序的 Carbon 用户界面



CarbonInCocoa 程序很简单。使用该程序的目的在于说明：您只需提供很少的代码，就可以在一个 Cocoa 应用程序中使用 Carbon 用户接口。

创建 Carbon 用户接口

您应该使用 Interface Builder 来创建 Carbon 用户接口。请按照下列步骤创建一个 Carbon 窗口：

1. 打开 Interface Builder。
2. 在“起点”对话框中的 Carbon 下面，选择“窗口”并单击“新建”。
3. 当窗口出现后，将 Carbon 面板中的项拖进窗口中来创建界面。

关于使用 Interface Builder 的详细信息，请参考 *Interface Builder*，或查阅 Interface Builder 中的帮助菜单。

关于如何制作遵从 Aqua 标准的界面，请参考 *Apple 人机接口指南*。

4. 保存文件。

Interface Builder将界面保存在一个nib文件中。（“nib”中的“ib”代表Interface Builder。）nib文件中包含接口信息一个归档。当您要显示接口时，您需要解归档该nib文件。你可以在“加载Nib文件”小节中了解到如何完成这项工作。

设置 Cocoa 应用程序以使用 Carbon 用户接口

为了使您的 Cocoa 程序能够使用 Carbon 用户接口，您必须执行以下操作：

- “添加描述Carbon接口的Nib文件”
- “声明控制器接口”
- “加载Nib文件”
- “为Carbon窗口创建NSWindow对象”
- “显示Carbon窗口”

添加描述 Carbon 接口的 Nib 文件

要添加详细说明 Carbon 接口的 nib 文件，请执行下列操作：

1. 打开您的 Cocoa 程序的 Xcode 项目。
2. 选择“项目” > “添加文件”。
3. 找到您刚刚创建的 nib 文件并双击它的文件名。
4. 在弹出的对话框中点击“添加”按钮。

如果您的 Cocoa 应用程序具有多个目标接口，您需要先选择相应的目标接口，再点击“添加”按钮。

声明控制器接口

CarbonInCocoa程序的控制器具有两个实例变量：一个是对应Carbon窗口的`WindowRef`结构，一个是NSWindow对象。其中，NSWindow对象允许我们使用Cocoa方法管理Carbon窗口。（请参考“显示Carbon窗口”部分。）
该示例程序的控制器没有其它的实例变量，但您的应用程序在适当的情况下可能需要声明其它的实例变量。程序清单 1 显示了 CarbonInCocoa 程序中对控制器的声明。

清单 1 控制器的声明

```
@interface MyController : NSObject
```

```
{
    WindowRef   window;
    NSWindow    *cocoaFromCarbonWin;
}
@end
```

加载 Nib 文件

当 Cocoa 程序启动时，Cocoa 的 nib 文件会被自动加载；但是，您必须显式加载包含 Carbon 接口信息的 nib 文件。对于 CarbonInCocoa 程序来说，当用户点击 Cocoa 窗口中的“显示 Carbon 窗口”按钮时，nib 文件就会被加载。

程序清单 2 显示了运行时刻加载 Carbon 的 nib 文件所需的代码。程序清单后面对有编号的代码行进行了详细解释。

清单 2 加载 Carbon 窗口的 nib 文件

```
    CFBundleRef bundleRef;
    IBNibRef    nibRef;
    OSStatus    err;

    bundleRef = CFBundleGetMainBundle();                            // 1
    err = CreateNibReferenceWithCFBundle (bundleRef,
                CFSTR("SampleWindow"),
                &nibRef);                                       // 2
    if (err!=noErr)
              NSLog(@"failed to create carbon nib reference");

    err = CreateWindowFromNib (nibRef,
                        CFSTR("CarbonWindow"),
                        &window);                               // 3
    if (err!=noErr)
              NSLog(@"failed to create carbon window from nib");

    DisposeNibReference (nibRef);                                   // 4
```

下面解释一下代码的作用：

1. 调用 Core Foundation 不透明类型的 CFBundle 的函数 `CFBundleGetMainBundle` 来获得应用程序 main bundle 的一个实例。您在下一函数调用时需要用到该实例的引用。

2. 调用 Interface Builder 服务函数 `CreateNibReferenceWithCFBundle` 来创建一个对 Carbon 窗口的 nib 文件的引用。您提供的 Core Foundation 字符串必须是该 nib 文件的文件名，但没有 `.nib` 扩展名。

3. 调用Interface Builder服务函数`CreateWindowFromNib`从nib文件中解归档Carbon窗口信息。该调用的返回值`window`是一个对应于Carbon窗口的`WindowRef`对象。在"声明控制器接口"小节介绍过，`window`是一个控制器实例变量。

4. 调用 Interface Builder 服务函数 `DisposeNibReference` 来销毁对 nib 文件的引用。您应该在完成对一个对象的解归档后立即调用该函数。

为 Carbon 窗口创建 NSWindow 对象

您不必为 Carbon 窗口创建一个 NSWindow 对象；但是，这样做可以让您使用 Cocoa 方法——而不是 Carbon 函数——来管理 Carbon 窗口。在某些应用程序中更倾向于这样做，而实际上 CarbonInCocoa 程序也正是这样做的。

您可以通过分配一个 NSWindow 对象并使用 `initWithWindowRef:`方法初始化该对象，从而为 Carbon 窗口创建一个 NSWindow 对象，如下面的代码所示。

```
cocoaFromCarbonWin = [[NSWindow alloc] initWithWindowRef:window];
```

在"加载Nib文件"部分曾介绍过，`window`是对Carbon窗口的一个引用。
您可以在 *NSWindow 类参考*中找到更多关于 NSWindow 对象和 `initWithWindowRef:`方法的信息。

显示 Carbon 窗口

由于您为 Carbon 窗口创建了一个 NSWindow 对象，因此您可以调用 `makeKeyAndOrderFront:`方法来显示该窗口，而不用调用 Carbon 函数 `ShowWindow`。您可以通过下面的代码显示 Carbon 窗口。

```
[cocoaFromCarbonWin makeKeyAndOrderFront:nil];
```

HICocoaView：在 Carbon 窗口中使用 Cocoa 视图

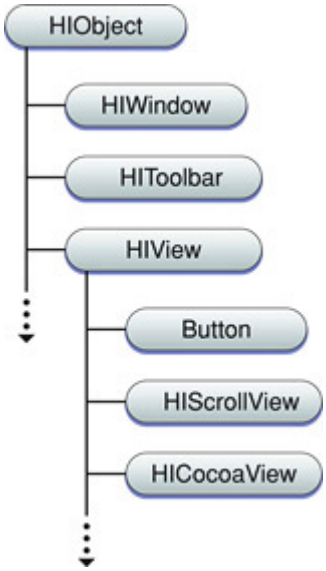
Cocoa 所提供的视图要么在目前的 HIToolbox 中不可用，要么虽然可用却没有提供充分的支持。这些视图包括 `WebView`、`PDFView`、`QTMovieView` 和 `NSTokenField` 等。此外，Cocoa 和 Carbon 控件的层次结构是不相容的，因此一直难以或根本无法在同一窗口内嵌入两个来自于不同框架的视图。

一种新的叫做 HICocoaView 的 HIView 类型解决了上述问题。在 Mac OS X 的 v10.5 及更高版本中，您可以在 Carbon 窗口中的 HIView 层次结构内嵌入一个 Cocoa 视图(`NSView` 的任意子类)。这是通过将 Cocoa 视图和一个叫做 HICocoaView 的 Carbon 封装视图——HIView 的一个子类——相关联来实现的。您可以使用标准 HIView 函数来处理该封装视图，同时您也可以使用 Cocoa 方法来处理相关的 Cocoa 视图。

注意：只有 composite 的 Carbon 窗口才支持 HICocoaView。关于 composite 窗口的更多信息，请参考 *HIView 编程指南*。

图 1 显示了 HICocoaView 类与 HIObject 类的继承关系。

图 1 HICocoaView 类的层次结构



由于 HICocoaView 是 HIView 的一个子类，因此您可以使用 HIView 函数对封装视图进行操作。例如，您可以使用 `HIViewFindByID` 在一个窗口的视图层次结构中查找视图。如果需要视图可见，您可以调用 `HIViewSetVisible`。如果需要重绘视图，您可以调用 `HIViewSetNeedsDisplay`。如果您需要对视图进行更多的控制，您也可以拦截任何 Carbon 控件事件并自己执行相应的操作。请注意，您不需要处理 `kEventControlDraw` 事件；封装视图会负责绘制它所封装的 Cocoa 视图。

对于您可以封装的 Cocoa 视图的类型是没有限制的。一个被封装的 Cocoa 视图可能还包含其它的 Cocoa 视图。要访问被封装的 Cocoa 视图中的功能，您可以使用该视图的任何方法。要调用这些方法，您需要使用 Objective-C 来创建并发送消息给 Cocoa 视图。例如，如果您想要让一个与 Carbon 封装视图相关联的 `PDFView` 对象前进到下一页，您需要向该对象发送 `goToNextPage:`消息。

当您修改一个 Carbon 封装视图的状态时，相关的 Cocoa 视图的状态会自动作出调整。这是一个单向的过程——也就是说，向 Cocoa 视图发送消息并不一定会改变封装视图的状态。例如，向 Cocoa 视图发送 `setFrame:`消息并不会改变封装视图在其父视图中的位置。

当您在一个窗口中嵌入封装视图时，也会有以下限制：

- 该封装视图必须被嵌入窗口的内容视图中。
- 该封装视图不能与其它封装视图相重叠。
- 该封装视图不能包含任何其它的 Carbon 视图。（这一限制不适用于被封装的 Cocoa 视图，被封装的 Cocoa 视图可以包含其它的 Cocoa 视图。）

下一节将介绍如何将 HICocoaView 纳入您的应用程序。

使用 HICocoaView

HICocoaView 的 API 易于理解和使用。它有三个函数：

`HICocoaViewCreate`	创建一个 Carbon 视图，用于封装 Cocoa 视图。
`HICocoaViewSetView`	将一个 Cocoa 视图和一个 Carbon 封装视图相关联。
`HICocoaViewGetView`	返回与 Carbon 封装视图相关联的 Cocoa 视图。

本节说明何时以及如何使用这些函数。

为您的 Carbon 项目使用 Cocoa 做准备

在您能够使用 HICocoaView 的功能之前，您需要按照以下步骤为您的 Carbon 项目使用 Objective-C 和 Cocoa 做好准备：

- 向您的项目目标中添加适当的 Cocoa 框架。例如，如果您打算使用 Cocoa 的 web 视图，就需要向待链接框架的列表中添加 `Cocoa.framework` 和 `WebKit.framework`。
- 导入必要的 Cocoa 头文件。例如，如果您打算使用 Cocoa 的 web 视图，就需要向您的源文件中添加下列代码：

```
#import <Cocoa/Cocoa.h>
#import <WebKit/WebKit.h>
```

- 在您使用 Cocoa 的函数中，分配并初始化一个 `NSAutoreleasePool` 对象，并在不需要它的时候释放该对象。（如果您的应用程序运行在 Mac OS X 的 v10.4 或更高版本，你就不需要在工具箱直接或间接调用的函数中使用自动释放池了。）
- 通过调用 `NSApplicationLoad` 函数，为您的 Carbon 应用程序使用 Cocoa 做好准备。通常，您应该在执行任何其它 Cocoa 代码之前，在主函数中调用该函数。
- 使用Objective-C或Objective-C++编译器来生成您项目中使用Cocoa的部分。"混合语言代码的预处理"部分介绍了如何配置Xcode项目以便使用适当的编译器。

创建封装视图

要创建一个Carbon封装视图并将其添加到Carbon窗口的视图层次结构中，您可以使用以下两种方法之一：

- 调用 `HICocoaViewCreate` 函数在运行时刻创建封装视图，并指定您想要封装的 Cocoa 视图。然后将该封装视图嵌入到窗口的视图层次结构中。关于在视图层次结构中嵌入和定位视图的信息，请参考 *HIView 编程指南*。
- 使用 Interface Builder 来设计一个基于 nib 的 Carbon 窗口，该窗口包含一个封装视图的占位符。Interface Builder 并不提供关联 Cocoa 视图与封装视图的方法，所以您需要在运行时刻完成关联操作。当您实例化窗口时，系统会为您创建一个空的封装视图，并将其添加到窗口的视图层次结构中。

使用 HICocoaViewCreate

下面的代码示例演示了如何使用HICocoaViewCreate创建一个被封装的Cocoa视图，该视图可被嵌入Carbon窗口的内容视图中。

```
NSView *myCocoaView = [[SomeNSView alloc] init];
HIViewRef myHICocoaView;
HICocoaViewCreate (myCocoaView, 0, &myHICocoaView);
[myCocoaView release];
```

使用 Interface Builder

如果您要使用 Interface Builder，第一步是向您的 Carbon 窗口中添加一个 HIView 对象。具体操作是：从“Carbon 对象”面板上选中一个 HIView 对象并将其拖进窗口中。如果您喜欢，您可以调整视图的大小使其填满窗口的更大区域。然后选择该视图并使用“检查器”窗口将"com.apple.HICocoaView"指定为它的类 ID。您还需要指定一个控件签名和 ID；您会在运行时刻使用这些值来找到该视图。

图 2 显示了一个基于 nib 并具有封装视图的 Carbon 窗口在 Interface Builder 中可能的样子。

图 2 Interface Builder 中的 Carbon 封装视图

关联 Cocoa 视图与封装视图

要将一个Cocoa视图关联到一个现有的Carbon封装视图，您需要使用HICocoaViewSetView函数。在下面两种情况下使用该函数：

- 您有一个空的封装视图，并准备用它来封装一个 Cocoa 视图。通常，当您使用 Interface Builder 创建了一个封装视图作为占位符，等待一个 Cocoa 视图在运行时刻与之相关联时，会出现这种情况。
- 您有一个已关联到某 Cocoa 视图的封装视图，而您想用一个新的 Cocoa 视图替换原有的 Cocoa 视图。

下面的代码示例演示了如何在窗口的内容视图层次结构中找到一个包装视图，并将一个 Cocoa 的 web 视图与其相关联。

```
const HIViewID kMyHICocoaViewID = { 'Test', 1 };
HIViewRef myHICocoaView = NULL;
HIViewFindByID (HIViewGetRoot(myWindow), kMyHICocoaViewID, &myHICocoaView);
if (myHICocoaView != NULL) {
   WebView *myWebView = [[WebView alloc] init];
   HICocoaViewSetView (myHICocoaView, myWebView);
   [myWebView release];
}
```

从封装视图获取 Cocoa 视图

如果您有一个与Cocoa视图相关联的Carbon封装视图，您可以使用HICocoaViewGetView函数得到一个指向该Cocoa视图的指针。通常，当您想要向Cocoa视图发送消息时会使用此函数。

下面的代码示例演示了如何从一个现有的封装视图中获取 Cocoa 的 web 视图，以及如何加载网页。

```
NSString *urlText = @"http://developer.apple.com/referencelibrary/";
WebView *myWebView = (WebView*) HICocoaViewGetView (myHICocoaView);
if (myWebView != NULL)
   [[myWebView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:urlText]]];
```

使用基于 Nib 的 Cocoa 用户接口

如果您使用 Cocoa 的 nib 文件来指定一个更为复杂的用户接口，则为了在 HICocoaView 中嵌入 Cocoa 用户接口，您的 Carbon 应用程序需要在运行时刻加载该 nib 文件。一种方法是使用自定义控制器对象来装载 nib 和访问用户接口。您可以使用 NSViewController 类来简化这一任务。NSViewController 可以轻松地加载 nib 文件并访问基于 NSView 的用户接口内部的情况。这里介绍的方法是根据一个示例应用程序 *HIView-NSView* 改编的。该示例程序使用 NSViewController 的一个子类 WebViewController 来实现用户接口中的某些特性。

程序清单 1 显示了如何实现一个封装函数，该函数可以创建一个基于 nib 的 Carbon 窗口，可以创建基于 nib 的、包含一个简单 web 浏览器用户接口的 Cocoa 窗口，还可以在 HICocoaView 中嵌入该用户接口。程序清单后面对有编号的代码行进行了详细解释。

程序清单 1 在 Carbon 窗口中使用基于 nib 的 Cocoa 用户界面

```
static OSStatus MyNewWindow (void)
{
    OSStatus status = noErr;

    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init]; // 1

    status = CreateWindowFromNib (gMainNibRef, CFSTR("MainWindow"), &gWindow); //
2
    require_noerr(status, CantCreateWindow);

    WebViewController* controller =
        [[WebViewController alloc] initWithNibName:@"WebView" bundle:nil]; // 3
    SetWRefCon(gWindow, (SRefCon)controller); // 4
```

```
    HIViewRef carbonView;
    status = HIViewFindByID (HIViewGetRoot(window), kMyHICocoaViewID, &carbonView);
// 5
    require_noerr(status, CantFindHICocoaView);

    NSView* cocoaView = [controller view]; // 6
    if (cocoaView != nil)
        status = HICocoaViewSetView (carbonView, cocoaView); // 7

    ShowWindow(gWindow); // 8

CantCreateWindow:
CantFindHICocoaView:

    [pool release]; // 9
    return status;
}
```

下面解释一下代码的作用：

1. 创建一个本地自动释放池。这一步骤是必需的，因为该函数不会被工具箱调用。
2. 创建一个基于 nib 的 Carbon 窗口。在这个例子中，主 nib 对象和新建的窗口对象都是全局变量。
3. 创建一个 Cocoa 视图控制器，来访问基于 nib 的 Cocoa 视图和实现 Cocoa 视图的用户接口。
4. 将 Cocoa 视图控制器作为窗口数据存储起来。当关闭窗口时，这些数据信息用于释放控制器。
5. 在 Carbon 窗口中查找 HICocoaView 封装视图。
6. 从视图控制器中检索 Cocoa 视图。
7. 在 HICocoaView 包装视图中嵌入 Cocoa 视图。
8. 使 Carbon 窗口可见。
9. 释放本地自动释放池。

图 3 显示了 Carbon 窗口内的一个简单的 Cocoa web 浏览器视图

图 3 Carbon 窗口内的 Cocoa 用户接口

想要学习如何编写一个实现如图 3 所示的用户界面的 `NSViewController` 子类，请参考示例程序 *HIView-NSView*。

在导航服务对话框中使用 Cocoa

从 Mac OS X 的 v10.5 版本开始，“导航服务”对话框是使用 Cocoa 的 NSOpenPanel 类和 NSSavePanel 类的自定义子类来实现的。这确保了使用“导航服务”的 Carbon 应用程序会得到与 Cocoa 应用程序相同的对用户界面的改进。此更改也适用于表单对话框。

这种变化的结果是，Carbon应用程序可以直接访问这两个Cocoa类的功能。“导航服务”NavDialogRef对象可以根据对话框的类型，转换为一个指向NSOpenPanel或NSSavePanel实例的指针。通过将您的NavDialogRef对象转换为相应的Cocoa对象，您可以使用Cocoa对象类中的任何Objective-C访问方法。这类似于无缝转换（见“可交互的数据类型”部分），但它只能进行单向的转换。
“导航服务”与 Cocoa 之间的桥接也存在一些局限。由“导航服务”创建的对话框必须由 NavDialogRun 方法调用，而不能是 NSOpenPanel 或 NSSavePanel 方法。也就是说，由 NSOpenPanel 或 NSSavePanel 的构造函数创建的对话框无法转换为 NavDialogRef 对象。在 Carbon 模式和 Cocoa 模式中有一些相互重叠的功能，例如文件过滤、设置附件视图和设置委托对象。在这些情况下，为避免冲突，您不应该混用这两种模式。

下面是一个例子，该例使用“导航服务”创建了一个“打开文件”对话框，并将其转换为一个 NSOpenPanel 对象来设置对话框中的标题和消息文本。

```
// navOptions structure has been created and initialized for text documents
NavDialogRef theDialog = NULL;
NavCreateChooseFileDialog(&navOptions, NULL, NULL, NULL, Handle_NavFilter, NULL,
&theDialog);
[(NSOpenPanel*)theDialog setTitle:@"Open Document"];
[(NSOpenPanel*)theDialog setMessage:@"Choose a text document to open:"];
NavDialogRun(theDialog);
```