Inside BREW

深入 BREW 开发

作者：焦玉海

导读

这本书适合谁？

在构思这本书的时候，我一直在想它应该针对哪一类读者呢，是有开发经验的程序员，还是初出茅庐的新手？我虽然反复的追问，但是我真的无法完全区分这两类读者，因为任何人的知识都是有局限性的，没有任何人能够什么都懂。所以最终我确定这本书的读者应该是那些想在计算机软件技术上深入钻研的人——用心去读书的人！是的，把这本书献给那些用心去读书的人，这也可以激励我用心去写作！

假如您是一位初学者，您千万不要害怕，在这本书里我只假设您懂得 C 语言，并且用它写了至少一个 100 行以上的程序就足够了。我相信我所讲的内容将有助于提高您在 C 语言开发上的技能，使您能够尽快地跨进软件世界的大门。而且本书专门为您准备了介绍基础知识的章节，夯实您的技术基础。因为在我看来，任何事情要做好都必须有相关坚实的专业知识做为基础，没有基础的上层建筑是不牢固的。

假如您是一位 PC 上的程序员，并且希望学习一些关于嵌入式平台的知识，那么这本书也是适合您的，这本书可以向您展示嵌入式系统的相关知识。从某种程度上来讲，嵌入式系统就是一个小型的 PC。或者由于一个更小的系统也更加容易理解的原因，这本书也可以让您通过对 BREW 平台的深入理解而有助于您理解诸如 Windows 这样的大型平台。再进一步，如果您正在学习有关 COM（Component Object Model）的知识，那么在本书中我将向您展示一个 C 语言版本的 COM 实现，它将进一步加深您对 COM 的理解。

假如您是一位嵌入式系统的程序员，希望深入的了解一个平台并且获得相关的知识，那么，这本书也是适合您的。在这本书里深入的分析了嵌入式系统开发平台 BREW 的实现方法以及组成，通过这些分析以及相关知识（如图形系统、事件驱动机制的实现）的阐述，您将更加深刻的理解嵌入式平台的组成要素。而且通过对 BREW 系统架构的剖析，对您设计自己的开发平台也会提供一个很好的参考。

假如您不但是一位程序员，而且还是一位 BREW 平台上的开发者，那么我不得不说，这本书是您的案头必备！在这本书里将深入的为您分析 BREW 的工作机制，BREW 应用程序的运行原理，以及 BREW 接口的实现方法。与此同时，本书的示例程序中还为您提供了一个 BREW 应用程序的框架，使得无论您开发什么样的 BREW 应用程序都可以得心应手。

如果现在为止，您还是不知道自己应不应该读这本书的话，那么我建议您先看看本书的目录，或许从那里您可以找到让您感兴趣的内容。

内容安排

全书共分三篇，每一篇均有不同的侧重点。

第一篇是勿在浮沙筑高塔，主要是介绍了理解 BREW 和嵌入式系统所需要的基本知识。其中包括硬件、C 语言、编译器和 MakeFile 等专业性较强的内容。虽然本篇的主要目的是给初学者夯实基础，但是，我相信其中所讲的内容对专业人士也会有一定的参考价值。

第二篇是磨刀不误砍柴工，主要介绍 BREW 应用程序的开发过程，让我们能够从开发 BREW 的应用程序的过程中熟悉 BREW 平台。在本篇中，还推荐了一个基于 BREW 事件驱

动机制的应用程序框架，这个框架采用了状态机的方式实现。通过这个框架，我们可以更加轻松的开发出应用程序。

第三篇是一识庐山真面目，在这一部分里我们剖析了 BREW 内部的实现方式，这一部分也是本书的重点。从这一篇的内容中我们将了解到 BREW 是如何使用 C 语言实现 COM 机制，BREW 内核的工作方式，以及 BREW 平台中图形系统的结构和一些高级的 BREW 接口的应用程序。这一篇就是深入 BREW 内部，分析 BREW，为我们的软件系统开发提供参考。

本书的约定

直到我开始写这本书的时候我才发现，如果想描述清楚一个思想是多么的困难，除了需要使用拗口的语言外，还必须要添加一些示意图。因为这些图看起来很直观，可能更容易理解（但是请注意，要结合说明来看，虽然这些说明有些很难懂，但往往本质就在这里面）。虽然我希望我能用最简单的语言来表述我的想法或描述一个原理，但考虑到描述准确性的问题，我不得不使用一些拗口的语言，这的确很矛盾，但是我只能尽力而为。

在这本书中指的嵌入式系统通常指手机系统或者叫做手持设备，因为这个系统更加具有普遍性，并且 BREW 平台本身也是针对手机开发的。

由于这本书的内容是通过对比 PC（Personal Computer）系统与嵌入式系统来各自展现它们的特点，因此在描述他们共性的时候我会使用计算机系统做为两者的统称。有人可能认为计算机系统应该指的是 PC 系统，实际上这是不准确的，因为嵌入式系统也是一种计算机系统，只不过小了那么一点。

在这本书中也频繁的使用了“程序”和“代码”两种称谓，它们之间没有明显的区别，只是根据上下文的不同，我可能会选择不同的称谓。

在本书中，有时候称 BREW 的应用程序为 Applet（小应用），有的时候又会直接称为应用程序，其实这两者都是一样，没有什么不同。之所以会有两种称呼，有的时候只是源于一种习惯——来自 BREW 开发文档本身的一种习惯。

如何联系作者

我尽力让这本书技术精准、可读性高，而且有用，但是我知道一定仍有改善空间。

如果您发现任何错误——技术性的、语言上的、错别字、或任何其他东西——请告诉我。我会试着在本书的新版本中修正它。如果您是第一位告诉我的人，我会很高兴将您的大名登录到本书致谢文内。如果您有改善建议，我也非常欢迎。

我非常乐意和本书的所有读者沟通，接受您对本书以及对我个人的指正和建议。不过我希望能将内容局限在对书籍、对知识的看法，以及对本书谬误的指正或建议上面。如果只是单纯地想和我交个朋友聊聊天，我也会倍感荣幸。

网络时代，唯网独尊，我的 Email 地址是：jiaoyuhai@hotmail.com

目录

前言

没有什么其他技术能够比得上移动通信技术对我们生活的影响了。就在前几年在大学里面读书的时候，我还觉得手机离我等升斗小民还是有距离的。然而就在短短三四年之后，在中国广袤的大地上已经是“处处闻铃声，机站知多少”了。在感受变化之快的同时，对于我这个投身于移动通信软件事业的人来说，更加感受到了手机软件的变化之快。从最开始的黑白“跳梁小丑”到现在真彩界面，从“叮咚”的 MIDI 音到现在的立体声音乐，每一次的硬件升级都带来了用户感官刺激的升级！

在用户享受的时候，我们这些开发人员确不得不想下一步还要升级什么，怎么实现？而正如我们所知的，这些硬件功能的不断升级，也需要可靠的软件来发扬光大。

硬件升级了，软件的开发者们可头痛了——我怎么发扬光大它？我要设计一个什么样的系统来尽可能的减少硬件升级对软件的影响？似乎 PC 机上的东西值得我们借鉴！是的，值得我们借鉴，但是我们并不能照搬照抄，因为手机的嵌入式系统硬件性能和 PC 性能相差甚远。因此，我们要裁减它！于是，各式各样的嵌入式系统出现了。在这本书里我无意于向各位读者介绍很多的嵌入式系统，而是更加希望能够呈现他们的本质——以 BREW 平台来展示。

请允许我的私心，之所以选择 BREW 平台，一是因为我和他一起工作了三年多，从它出生到现在；二是我认为它的实现方式对于我辈软件中人具有很好的参考价值——怎样使用 C 语言来实现面向对象的主要特征（这些特征主要包括简版的封装、继承和多态）；三是我认为它简洁高效。希望各位读者不要怪罪于我！

既然在这里提到了 BREW，那我就先简单的介绍一下吧，毕竟 BREW 还不是广为人知。BREW（Binary Runtime Environment for Wireless）翻译成中文就是无线二进制运行环境，顾名思义，它针对无线通信系统提供了一个二进制级别的运行环境。“二进制运行级别”应该怎么理解呢？通常我们说程序语言经过编译链接后，可以直接在 CPU 上运行的就是二进制级别的语言，因为它生成的目标是针对 CPU 的。使用 BREW 开发的应用程序就是二进制级别的，这样 BREW 的运行效率就可以得到最大的提升。

与此对应的就是解释型的语言，它们并不生成可以直接在 CPU 上运行的二进制代码。它们当中还有些是不需任何编译而直接解释运行的，如 HTML、Script（脚本）语言等等，还有半解释型的语言如 Java，它们部分编译优化了原始代码（编译的是解释器能够识别的而不是 CPU 能够识别的）。不管是解释型语言还是半解释型语言，它们都需要有一个二进制级别的语言解释器。相应语言的应用程序需要经过这个解释器来解释执行。这是可以理解的，因为不管怎么样程序要运行都得通过 CPU 啊，二进制级别的解释器也就是 CPU 可以执行的解释器。

BREW 是由美国 Qualcomm 公司开发，集成在其发布的 CDMA2000 和 WCDMA 芯片的软件版本中，目的是提供一种可以从网络下载应用程序的方法。其核心解决的问题就是如何在手机上支持下载的二进制应用程序的运行。为了解决这个问题，BREW 的开发者们大力的汲取了 COM 的思想，通过最小的 C 语言代码实现了这个功能。这本书将引领我们进入 BREW 的世界，看清它的本质。

第一篇 勿在浮沙筑高塔

对于现在从事软件开发很多人来说，都是直接接触的 Windows 操作系统，一开始学习的东西就是在 Windows 下的图形编程，尤其是可以快速开发的诸如 VB、Delphi 和 C++ Builder 等语言。它们的程序开发方式就像是作画一样，通过将控件放到对应的窗体上，然后设置属性，完成事件处理过程就可以生成一个像模像样的 Windows 程序了。虽然使用这种语言可以很容易的开发出应用程序来，但是却缺乏了对系统的深入了解，只知道怎么做，却不知道为什么这么做以及它是怎么实现的。归根结底，就是缺乏对底层实现原理的知识。正如本章的标题“勿在浮沙筑高塔”一样，缺乏根基总还是底气不足！不过别担心，在这部分里，我将讲述我们所需要的一些“根基”。当然如果您已经对这些知识有所了解，并且只想看看 BREW 是个什么样子，那么可以跳过这一部分，而直接进入到第二部分。

同时，在我看来，不了解硬件结构的程序员不会成为一个最优秀的程序员。因为整个软件和硬件组成了一个系统，如果我们不了解硬件知识，那么我们也不会了解整个软件的来龙去脉，也就不能从整体上把握程序的特点，要写出优秀的程序是很困难的。像 VB、Delphi 和 C++ Builder 等 PME(Property – Method – Event-Driven)语言虽然也给了程序员一定的开发空间，但是缺乏基础的大厦毕竟筑不高！所以我希望每个程序员都能具备一定的硬件知识，这样才能站得高看得远。如果您觉得 PC 太复杂了，那么您可以研究相对简单的嵌入式系统。当然并不是所有的嵌入式系统都简单，比如手机就是一个要比 PC 系统更为复杂的系统，因为它其中包含了无线通信部分的内容，只不过我们现在不管这些内容而只是摘取计算机系统相关的内容而已。

这一篇的主要内容介绍如下：

第一章是硬件基础，任何软件平台都离不开硬件平台的支持，而一些设备的原理却成为了我们学习的障碍。在这一章里，我们将获得一部分硬件的基础知识，虽然它并不全面，但是我想为了阅读本书是足够了。

第二章是软件基础，在这里主要讲述了 C 语言的相关知识。这部分属主要是用我个人的描述方式来讲解 C 语言的细节，希望对我们后续的理解能够有所帮助。

第三章是编译器基础，从中我们可以了解到各种不同类型的编译器以及它们的区别。更为重要的是它介绍了编译器对程序中代码和数据的处理方式，同时使用了一个最小的 ARM 系统的例子，以此来展示 C 语言更加本质的东西。

第四章是工程管理（Make File）基础，在这一章里将主要讲述关于 Make File 的知识，同时给出了一个可以用于大型系统的 Make File 框架。设置这一章的主要考虑是到现在除了比较专业的领域外，大多数人都已经接触不到它了，然而实际上它却很有用。

第一章 硬件基础

硬件是软件的运行平台，没有硬件的支撑软件也将不复存在。您能想象没有显示器软件将如何显示图形，没有 CPU 软件将如何运行吗？反正我想象不到！但是如果把问题反过来问就问到本质了，软件运行需要哪些硬件支持呢？看图 1.1：

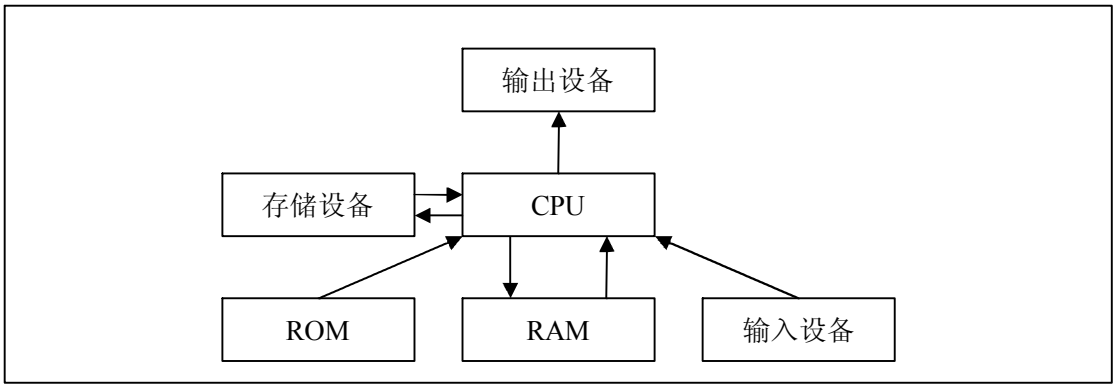

图 1.1 系统结构框图

我们抛开硬件的什么电器特性等等，去芜存菁，就是上面的这个图了。如果程序要运行没有 CPU 是不行的，CPU 要快速的交换数据，没有 RAM 也是不行的。因此无论任何系统，CPU 和 RAM 都是必不可少的。您一定会提醒我 ROM 不也是不变的吗？这种说法不完全对，因为在 PC 系统和嵌入式系统之间 ROM 的作用是不一样的。在 PC 系统中 ROM 就是那个 BIOS 芯片，是用来提供系统的启动代码和基本的输入输出功能的；而在嵌入式系统中，ROM 存储了全部的代码，它已经将 PC 中的 BIOS 和硬盘的与代码相关的功能混合在一起了。

设备	PC 系统典型硬件设备	嵌入式系统典型硬件设备
CPU	任何 CPU	任何 CPU
RAM	任何 RAM	任何 RAM
ROM	BIOS 芯片	Flash 芯片
存储设备	硬盘	Flash 芯片
输入设备	键盘	键盘
输出设备	显示卡+显示器	LCD 显示屏

PC 的 ROM ——BIOS 芯片可以采用 Flash 芯片，在这里之所以不写成 Flash 芯片是因为 BIOS 的作用和嵌入式系统的 Flash 作用不大一样，使用 BIOS 以示区分。

1.1 CPU 和 RAM

从软件观点来讲，任何 CPU 和 RAM 都可以应用于各种系统中，不存在明显的区别，只要 CPU 可以执行指令控制设备就可以了。但是考虑到耗电以及体积（嵌入式设备通常要求耗电低、体积小）等问题，嵌入式系统就发展出了专用的 CPU 芯片。当前应用最广泛的是 ARM CPU。ARM CPU 是由英国的 ARM 公司设计的，由于其执行效率高，体积小，耗电少等特点被广泛应用于嵌入式系统。由于嵌入式系统要求高集成度，通常不会存在单独的 CPU 芯片，而是将 CPU 和很多的外围电路集成到一起，做成一块芯片，因此 ARM 采用授权的方式提供内核芯片设计，以便于使用者进行芯片的集成。

CPU 按照次执行指令的数据带宽可以分为 16 位、32 位、64 位等。32 位 CPU 一次只能

处理 32 位，也就是 4 个字节的数据；而 64 位 CPU 一次就能处理 64 位即 8 个字节的数据。如果我们将总长 128 位的指令分别按照 16 位、32 位、64 位为单位进行编辑的话：旧的 16 位 CPU（如 Intel 80286 CPU）需要 8 个指令，32 位的 CPU 需要 4 个指令，而 64 位 CPU 则只要两个指令。显然，在工作频率相同的情况下，64 位 CPU 的处理速度比 16 位、32 位的更快。

除了运算能力之外，与 32 位 CPU 相比，64 位 CPU 的优势还体现在系统对内存的控制上。由于地址使用的是特殊的整数，而 64 位 CPU 的一个 ALU（算术逻辑运算器）和寄存器可以处理更大的整数，也就是更大的地址。传统 32 位 CPU 的寻址空间最大为 4GB，使得很多需要大容量内存的大规模的数据处理程序在这时都会显得捉襟见肘，形成了运行效率的瓶颈。而 64 位的处理器在理论上则可以达到 1800 万个 TB（1TB=1024GB），将能够彻底解决 32 位计算系统所遇到的瓶颈现象。当然 64 位寻址空间也有一定的缺点：内存地址值随着位数的增加而变为原来的两倍，这样内存地址将在缓存中占用更多的空间，其他有用的数据就无法载入缓存，从而引起了整体性能一定程度的下降。

在进行系统设计时，会根据不同寻址能力的 CPU 来进行寻址空间的分配。由于 CPU 都是通过设备的寄存器（这个寄存器可以理解为设备本身带的 RAM）来控制设备的，因此地址空间的划分就显得十分重要。例如，一个具有 32 位寻址能力的 CPU 不可能讲全部的地址空间都分配给 RAM，好比 PC 系统需要为 BIOS 分配存储空间等。也就是说只要是需要 CPU 直接控制的外部设备都需要为其分配 CPU 地址空间。

RAM 就是可以随机访问，快速读写的存储器。CPU 可以直接从 RAM 中取得数据（CPU 可以从所有分配了地址空间的设备寄存器中取得数据）或代码指令，因此 RAM 的访问速度将直接影响系统的性能。

1.2 ROM 存储芯片

ROM 是每个计算机系统必不可少的，但是其实现的方式却不尽相同。在我们熟悉的 PC 系统中，ROM 是一个称作 BIOS（Base Input & Output System）的芯片。CPU 上电时会从 ROM 中读取指令，因此没有 ROM 的系统是不能够运行的，因为如果没有 ROM，CPU 将无法获得起始的执行指令。在 PC 系统中 BIOS 的作用除了提供起始指令以外，还会扫描硬件设备并初始化主板（Main Board）上的硬件接口。由于 PC 上的接口都遵循着一组通用的协议，因此 BIOS 就可以实现所有硬件接口的驱动（如 USB 接口、显示卡、键盘和鼠标等）和硬件的数据输入输出功能，这也是 BIOS（基本输入输出系统）名称的由来了。在 BIOS 控制的硬件接口中也包含了硬盘的控制接口，在 BIOS 初始化完成后就会到硬盘主分区上查找启动文件（*注 3*），后面的事情就交给 PC 的操作系统了。这个过程请看图 1.2。

通常，在硬盘内有一个主引导记录区，在安装操作系统的时候由操作系统写入 Boot 程序，BIOS 就是取出这段程序然后执行。由于 Boot 程序是由操作系统写入的，因此从这个 Boot 程序开始，系统的运行权限就交由操作系统来控制了。以 Windows2000 操作系统为例，这段代码区域执行时会搜索名叫“NTLDR”的系统文件。之所以分成 Boot 程序和 NTLDR 文件的原因是硬盘的 Boot Sector 很小（只有 466Bytes），不可能容纳全部的启动程序。

图 1.2 只是一个启动的示意图，在这里我并没有详细的列出每一个必须的步骤，因为我的目的只是让我们能够了解 BIOS 硬件芯片在系统中的作用。

对比 PC 的 BIOS，嵌入式系统由于软件规模小，因此将引导代码和操作系统代码全部放到了系统的 Flash 芯片中了。正如我们所知道的，PC 机上大部分的操作系统代码全部放在硬盘上，然后从硬盘上将程序载入内存执行。而嵌入式系统中目前大多数采用直接寻址的方式从 Nor Flash 芯片中读取代码并执行。因此，实际上嵌入式系统简化了 PC 系统的设计，

将 PC 系统中的 BIOS 和硬盘代码全部集中到了一个 Flash 芯片上。因此 BIOS 虽然也可以使用 Flash 芯片，但是相对于嵌入式系统来说，他们的含义和作用却不同。

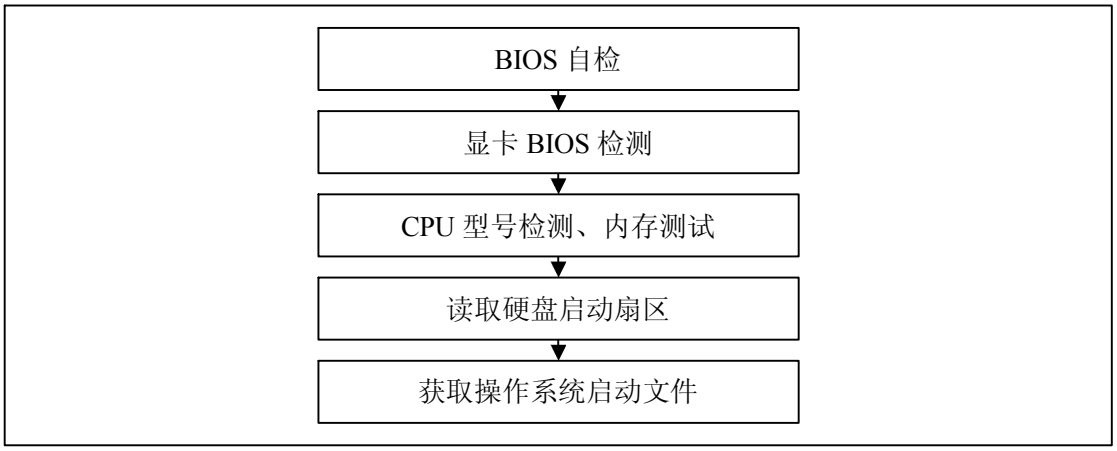

图 1.2 PC 开机流程

Flash 芯片对于我们来说并不陌生，那些可以更新 BIOS 程序的 BIOS 芯片也是使用 Flash 芯片实现的，还有 MP3 用的 SD 卡等等也是 Flash 芯片。Flash 芯片是一种可以多次擦写的存储芯片，广泛的应用于嵌入式系统。Flash 的特点是耗电低，容量大（相对于嵌入式系统而言），写入之前需要先擦除（因为 Flash 芯片的存储单元只允许从 1 变到 0）。当前流行的分为 NOR Flash 和 NAND Flash。

NAND 与 NOR Flash 的区别主要有：

1、NAND Flash 的空间比 Nor Flash 大

2、NAND Flash 的访问速度比 Nor Flash 快

3、NAND Flash 只有 Page 访问模式，Nor Flash 可以进行 Page 和直接地址访问（直接地址访问也就是 CPU 可以直接寻址，或者叫做随机访问）

4、NAND Flash 允许有坏块，但是 Nor Flash 不能有坏块

5、NAND Flash 比 NOR Flash 更加便宜

在嵌入式系统中，NOR 和 NAND 都可以做为代码区和文件系统区来使用。通常情况下 NOR 和 NAND 做为嵌入式文件系统区的时候都使用 Page 模式。Page 模式允许一次读取多个字节，就像硬盘的最小读写单位是扇区一样，只不过 Flash 的最小读写单位叫做 Page。Page 模式下可以加快 Flash 的读写速度。由于 NAND Flash 只支持 Page 读写模式，因此使用 NAND Flash 做为代码区的时候需要外加控制电路。当前使用 NAND 做为代码区正在成为一种流行的趋势（因为 NAND Flash 成本更低），主要的实现方式有两种：一是添加仿真电路使得 NAND Flash 可以支持随机访问；二是增加一个类似硬盘的引导区（通常是第一个 Page），系统启动的时候使用引导区的代码将全部 NAND 中的代码复制到 RAM 中执行。

当然，可以设想随着将来嵌入式操作系统的发展，动态载入内存的形式也许就会出现了，但是目前嵌入式系统仍然没有发展到这个地步。更进一步，当前应用于嵌入式系统的微型硬盘也已经出现了，或许更为复杂的操作系统也可以应用在嵌入式系统上了。

在这里我们主要介绍的 ROM 存储芯片是 Flash，严格意义上说 Flash 并不能称作 ROM，因为 ROM 是只读存储器（Read Only Memory），而 Flash 是一种可读可写的芯片。但是由于 ROM“一次成型、终生不变”的特点，不便于升级换代，现在正逐渐的被 Flash 芯片所取代，但是其功能性的称谓“ROM”还在为大家所用。

在计算机系统中主要存在用户数据、程序数据和代码三种二进制内容。用户数据指用户文件，程序数据是指程序运行时需要修改或使用的非代码内容。下面将就 PC 系统和嵌入式系统中的这三种二进制内容做一个比较，请看下表：

二进制形态	PC 系统	嵌入式系统
用户数据	存储在文件系统中，典型的设备是硬盘	存储在文件系统中，典型的设备是 Flash 存储芯片
程序数据	可读可写的数据存放在 RAM 中；只读数据存放在硬盘中，运行时与代码一起读入 RAM	可读可写的数据存放在 RAM 中；只读数据存放在 Flash 中，与代码存储在同一个区域
代码	存储在文件系统中的文件里，运行时读入 RAM 由 CPU 执行	如果存储在 NOR Flash 等可随机访问的空间中则 CPU 直接在芯片中取指令运行；如果存储在 NAND Flash 等不能随机访问的空间中则需要读入 RAM 中运行

关于程序数据的详细情况将在编译器基础一节详细介绍。

1.3 输出设备

输出设备有很多种，例如显示器、打印机，在这里我们主要讲一下显示设备。

任何显示设备都是点阵式的，至少目前是这样。还记得初次看见电视里显示的人物时内心的惊异，这个世界竟是如此神奇！后来知道了，如果我愿意，我可以买 640*480 个灯泡，组成一个 640*480 的方阵，然后控制每个灯泡的亮灭，我也可以显示一个人物。终于懂了，原来任何的显示设备都是基于这个原理实现的。我想关于这一点我没有必要再多说什么了，毕竟万变不离其宗嘛。

当前流行的显示设备有 CRT 显示器（也就是电脑上很大个头的那种显示器），LCD（液晶）等（虽然我在大学实验室里用的可以显示数字的 LED 灯也是显示设备，但是他太简单了）。CRT 显示器经历了从球形到纯平的物理演变，LCD 则经历了从黑白到彩色的变化。按照显示器每个点能够显示的颜色数目可以分为黑白两色、灰度、8 位色、16 位色、24 位真彩色等显示设备。这个颜色数目就是所谓的色深（Color Depth），色深越大，每个点能够表达的颜色数就越多，这个点就是“象素”。

在嵌入式系统中主要使用 LCD 的显示设备，LCD 会集成一个显示的存储空间，在这个空间中存储了对应的每个象素的值。例如 16 位色的 LCD 每个象素需要由 2 个字节来表示，如果显示屏幕的大小是 100*100，那么就需要 2*100*100 = 20000 个字节的存储空间。程序正是通过更新这个存储空间中的内容来控制 LCD 的显示。

衡量 LCD 显示效果的还有象素间距，象素间距越小，画面就越细腻，显示效果越好。举个极端的例子，如果一个象素是整个 LCD 那么大，那就只能看见一个象素点的“灯泡”了，也就没法显示图像了。通过像素个数和每个点的大小就可以换算出显示屏的大小了，例如常用的 xx 英寸大小的屏幕等等。

1.4 输入设备

输入设备也有很多种，最典型的是键盘和触摸屏。在这一节里，我将简单的介绍一下它们的实现原理。

键盘通常是一个矩阵式的电路，当按键按下的时候，接通电路产生信号，如图 1.3：

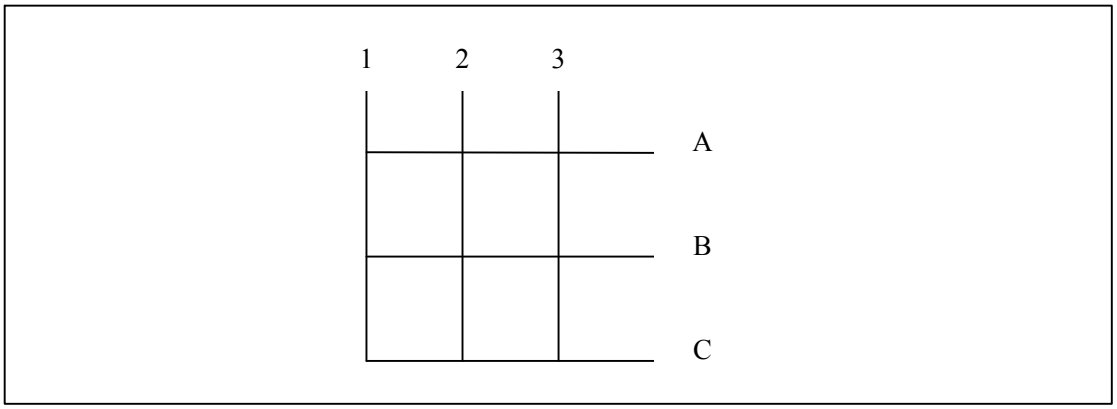

图 1.3 键盘原理

图 1.3 是一个简版的键盘原理图，图中任何交叉点的横向和纵向都未连通，并假设只要交叉点连通，则相应的行和列就可以连通并发生状态改变。其中 1、2、3 和 A、B、C 连接在控制芯片上，通过扫描行和列，确定行的 A、B、C 是否连通，再扫描列 1、2、3 是否连通，这样就可以唯一确定一个点是否按下。千万注意，这只是一个示意图，并不是真正的键盘原理图。

触摸屏是近来应用越来越多的输入器件。典型触摸屏的工作部分一般由三部分组成：两层透明的阻性导体层、两层导体之间的隔离层、电极。阻性导体层选用阻性材料，如铟锡氧化物（ITO）涂在衬底上构成，上层衬底用塑料，下层衬底用玻璃。隔离层为粘性绝缘液体材料，如聚脂薄膜。电极选用导电性能极好的材料（如银粉墨）构成。

触摸屏在工作时，上下导体层相当于电阻网络。当某一层电极加上电压时，会在该网络上形成电压梯度。如有外力使得上下两层在某一点接触，则在电极未加电压的另一层可以测得接触点处的电压，从而知道接触点处的坐标。比如，在顶层的电极(X+,X－)上加上电压，则在顶层导体层上形成电压梯度，当有外力使得上下两层在某一点接触，在底层就可以测得接触点处的电压，再根据该电压与电极(X+)之间的距离关系，知道该处的 X 坐标。然后，将电压切换到底层电极（Y+,Y－）上，并在顶层测量接触点处的电压，从而知道 Y 坐标。通常有专门的控制芯片。很显然，触摸屏的控制芯片要完成两件事情：一是完成电极电压的切换；二是采集接触点处的电压值（即 A/D）。虽然还有其他的实现方法，我就不赘述了，因为本书的目的不是讲解硬件的原理，而仅仅是让我们能够基本了解它们。

1.5 小结

在这一章里介绍了各种硬件的特性和原理，其中包括了最小系统各种组件的介绍，为的是防止我们见到这些东西的时候会一头雾水，不知来由。只要我们见到这些硬件不再感到神秘，那么这个基础就算打好了。

思考题

在 PC 计算机系统中硬盘、BIOS、RAM 和嵌入式系统中的文件 Flash、程序 Flash、RAM 之间有什么区别？它们在系统中的作用分别是什么？

第二章 软件基础

我们正在向我们的软件王国进发，千万别急，在这条路上“枯燥”是我们最大的敌人，不知有多少人在它的面前臣服，但愿您不是其中之一。或许您觉得应该获得一些鼓励，写一些代码，能够看见一些诸如“Hello, World!”之类的信息。非常幸运，从这里开始您将能够看见它们了，我会将部分内容使用源程序的方式向您讲解。在这本书里，我将使用 Visual Studio .Net2003 的开发环境来执行这些测试程序。建立测试工程的步骤如下：

1、首先打开 Visual Studio .Net2003 开发环境，选择新建项目。如图 2.1，选择项目类型 Visual C++项目/Win32 控制台项目，选择路径，填写测试程序名称是 Test1，点击确定按钮。

2、在 Win32 应用程序向导中接受默认设置，点击完成按钮。如图 2.2。

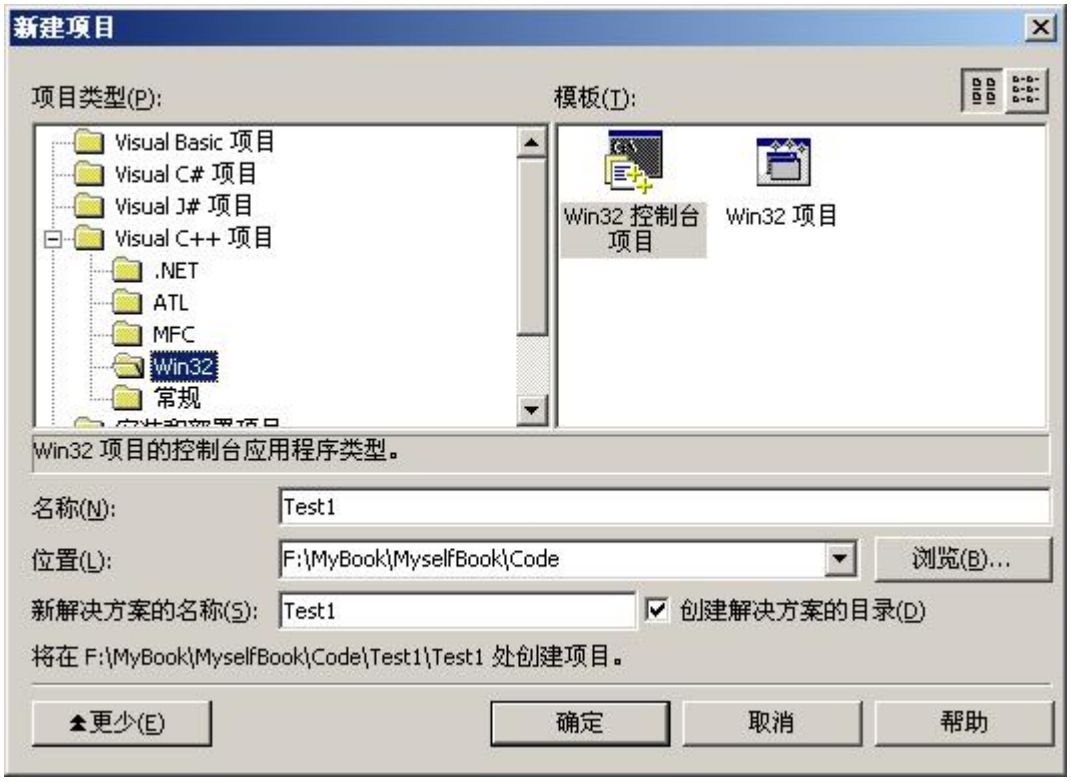

图 2.1 建立测试程序

这样就完成了一个测试用的应用程序，在以后的测试程序中我将不再重复这个步骤。由于我们主要是在 C 语言的基础上讲解，因此在这里创建的是 Win32 控制台应用程序，它使用 Windows 的命令窗口显示输出结果。不过请注意，虽然它使用 Windows 的命令窗口，但是它是一个 Win32 的应用程序，而不是 DOS 应用程序。

在 Windows 环境下，应用程序分为控制台（Console）应用程序和窗口应用程序。控制台应用程序不显示窗口，其表现形式类似于 DOS 环境下的应用程序，但是由于它可以调用 Windows 的 API 来实现，所以它是一个 Windows 的应用程序而不是 DOS 应用程序。同时控制台应用程序可以使用标准 C/C++的库，这样 Windows 下的控制台应用程序就和 DOS 环境下的 C/C++十分的相似了。

在这一部分我们将通过实例来演示一些 C 语言中比较令人“眩晕”的话题，期望能够通过这些实例让您弄明白这些问题的本质。由于本书是建立在您已经有一定的 C 语言基础之上的，因此主要是针对 C 语言中一些较难理解的概念进行讲解，毕竟我们不是一本专门讲解 C 语言的书籍。在本部分主要讲解的内容是指针、结构体、预处理和函数，因为在我

们接下来的行程中必须要充分的理解它们才能继续前进。

图 2.2 Win32 应用程序向导

2.1 重温 C 语言的指针

指针是一个精灵，以至于在我们刚刚接触它的时候有点不知所措，甚至有些人怀着敬畏的心情而决心远离它！不过，当我们掌握了它的时候，就会发现它能让我们随心所欲。尽管我们仍然面临着使用不当所带来的巨大风险，但是我还是会义无反顾的告诉您——一定要使用它。之所以我会在这里向您发出这样的号召，绝对不是因为我对指针的个人情感，我和指针也是非亲非故，只不过是因为透过它我们不但可以编写出灵活的程序，而且可以窥探到程序的真正世界——二进制世界的秘密。这就让我们开始拥抱它吧！

2.1.1 指针的本质

指针的本质是存储它所指向存储空间地址的变量，下面将通过一个测试程序来开始征服它的旅程。打开测试工程 Test1，在 Test1.cpp 中添加如下代码：

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    int *pointer;
    int nNumber = 100;
```

```
    pointer = &nNumber;
    printf("&pointer = 0x%x pointer = 0x%x *pointer = %d \n",&pointer,pointer,*pointer);
    return 0;
}
```

在这段代码中，我们定义了一个 int 型指针 pointer 和一个 int 变量 nNumber，然后让 pointer 指向 nNumber。编译、运行生成可执行文件，可以看到输出的&pointer、pointer 和 *pointer 的值如下：

```
    &pointer = 0x12fed4
    pointer = 0x12fec8
    *pointer = 100
```

*pointer = 100 是我们知道的，也就是指针指向的内存地址的内容，我们在程序中将 pointer 指针指向 nNumber 的地址，那么与 nNumber 的值相等就不足为奇了。pointer = 1244872 这个值就是指针指向的内存地址，是当前函数运行中存储 nNumber 的地址。&pointer 就是这个指针变量的地址，由于我们定义了一个指针变量，因此，在函数运行时将会为这个变量分配一个存储空间，因此这个值是有意义的，而且它与 pointer 的值十分相近。我们可以这样理解指针，它是一个变量（因为指针也需要空间存储它所指向的地址），这个变量的值就是一个内存空间的地址。示意图如下：

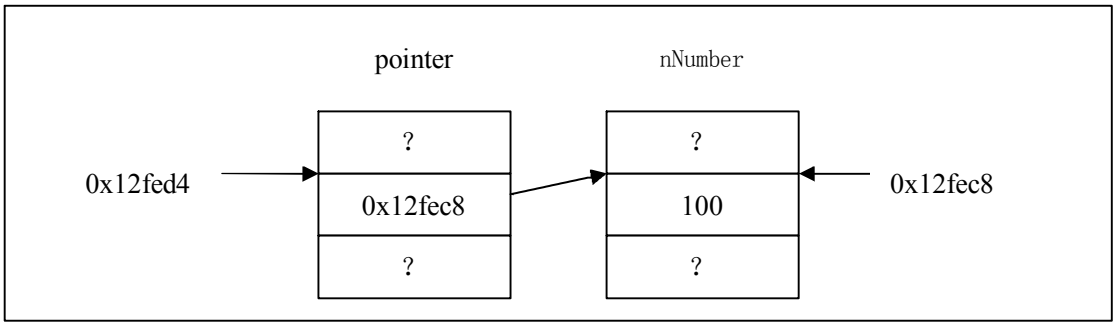

图 2.3 指针示意图

在上面的代码中，&pointer 是一个指向指针的指针，这样的称呼实在是过于繁琐了，我们就称它为“二重指针”吧。顺延的，如果是指向（指针的指针）的指针我们就叫它“三重指针”。在 C 语言中，二重指针是一个非常有用的东西，很多高阶的 C 语言应用都会使用到它。二重指针的主要作用是做为参数为一个指针变量赋值，在后面的章节中我们还会经常使用到它。

对上面的图 2.3 作一个说明，pointer 和 nNumber 分别位于两个不同的内存区域，nNumber 中存储的值是 100，这是程序中指定的；pointer 内存储的值是 0x12fec8，这个值正好是 nNumber 所在的内存地址；图左边的 0x12fed4 是存储 pointer 值的指针变量的地址，我们可以通过定义一个二重指针获得它的内容，在本程序中我们通过&pointer 来获得它的内容。

从本质上来说，指针、二重指针、三重指针等等，都是一样的，从代码的层次（我们将在“编译器基础”部分详细讲解软件的层次问题）来讲都是一个“地址”型的变量，只不过使用的时候会由编译器做一下合法性的检查，目的是为了防止程序出现错误；在二进制层次来说它们就更加没有分别了，都是一个存储内容的容器。打个比方，现在有两个盒子，一个规定放置篮球，另一个规定放置足球。体现在 C 语言中这个“规定”就是定义了两个变量，一个是 int 型的，另一个是指针型的；这两个盒子就是两块存储空间。在现实生活中“规定”是由人来制定的，在程序中就是由 C 语言定义的。那么我违反规定放置篮球的盒子我放置足球，放置足球的盒子我放置篮球。这是没什么问题的，不过会发生错误，因为会让一场足球比赛变成了踢篮球的比赛，让一场篮球比赛变成了足球投篮的比赛。同样的

对于指针和变量的关系也是，它们的内容在二进制层次是可以互换的。但是我们在这里要考虑两件事情，一是这个错误发生的前提条件，这个错误要发生必须是在球的管理者不知道错的情况下。如果球的管理者知道哪个盒子放了篮球哪个盒子放了足球也不会出错。体现在程序上，就是增加强制类型转换来告诉编译器“我知道我要在 int 变量中放置一个指针的值”。否则编译器将检查到这个错误并报错。二是要考虑“盒子”容量的问题，因为足球比篮球小，所以互换的时候不能让篮球撑坏了足球盒子。在程序中也就是 32 位的指针变量不能放到 char 型变量中去存储，因为这样会丢掉地址信息。

综上所述，虽然变量的二进制本质是一样的，但是在代码层次要精确控制变量的类型来避免错误的发生。指针也是一个变量，一个 32 位的指针变量也可以存储在一个 4 字节的无符号整型变量里，前提是我们要知道我们是采用这种方式来存储它们的。

最后使用一个例子来做个演示，新建工程 Test2，输入如下代码：

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    int nNumber = 100;
    int *pointer = &nNumber;
    unsigned int     dwBox;
    unsigned short wBox;

    // 将指针的值分别赋给无符号整型变量
    dwBox =(unsigned int)pointer;
    wBox   =(unsigned short)pointer; // 地址内容将被裁减!!!

    // 输出 Box 变量的值，注意 wBox 与 dwBox 之间的不同
    printf("wBox = 0x%x dwBox = 0x%x\n",wBox,dwBox);

    // 输出 Box 变量指向地址的值，其中*((int *)dwBox)相当于*pointer
    // 此时不能够使用*((int *)wBox)输出值，因为它的地址是经过裁减的
    printf("*dwBox = %d\n",*((int *)dwBox));
    return 0;
}
```

编译后运行，输出如下结果：

```
wBox = 0xfed4 dwBox = 0x12fed4
*dwBox = 100
```

请仔细体会上面的代码，它真正揭示了变量与指针之间微妙的关系，还有类型转换时所发生的数据裁减。

2.1.2 指针的增减

关于指针的增减是一个比较容易让人迷惑的问题，指针本身也是一个变量，它里面存储的是一个地址，那么这个变量的自增是将这个地址值加 1 吗？自增操作和直接加 1 的操作是一样的吗？

为了弄清这个迷惑，这里我们在 Test2 工程中增加一个 Test2-1 的项目（请设置 Test2-1 为启动项目），然后输入如下代码：

```
#include "stdafx.h"

typedef struct _Test2{
	char Test2[100];
}Test2;

int _tmain(int argc, _TCHAR* argv[])
{
	int	nValue = 1;
	char	cValue = 'A';
	Test2 sValue;
	int	*intPtr	= &nValue;
	char	*charPtr	= &cValue;
	Test2 *structPtr= &sValue;

	printf("Int %d Char %d Struct %d\n",(int)intPtr,(int)charPtr,(int)structPtr);

	intPtr++;
	charPtr++;
	structPtr++;
	printf("Int %d Char %d Struct %d\n",(int)intPtr,(int)charPtr,(int)structPtr);

	intPtr	+= 1;
	charPtr	+= 1;
	structPtr+= 1;
	printf("Int %d Char %d Struct %d\n",(int)intPtr,(int)charPtr,(int)structPtr);
	return 0;
}
```

编译运行输出的结果如下：

Int 1244884 Char 1244875 Struct 1244764

Int 1244888 Char 1244876 Struct 1244864

Int 1244892 Char 1244877 Struct 1244964

从这个结果中，我们可以看出，指针的加减是与指针所指的变量类型有关的，它所增加的步数是所指变量的大小，而且指针的+1 和++操作是一样的。那么 void 型的没有类型的指针怎么处理呢？答案是不处理，编译器会通知我们“未知大小”的错误而终止程序的生成。

2.2 重温 C 语言的结构

结构是 C 语言中组织不同数据类型的一种方式，它将不同的数据类型组织到一个相邻的地址空间内。每个定义的结构体变量就是一个多种数据的存储空间。在这里我们主要讲述几个相对“高阶”问题，之所以在这里加上引号，是因为对于某些传说中的高手来说，这不过是小菜一碟。

2.2.1 结构体变量赋值

我们已经习惯了为结构体变量中的每个成员赋值，那么我们可以在两个结构体变更之间直接使用“=”号赋值吗？ 答案是肯定的，因为编译器支持。例如定义一个表示矩形的结构体：

```
typedef struct _Rectangle{
    int x;      // 左上角 x 坐标
    int y;      // 左上角 y 坐标
    int dx;     // 矩形宽度
    int dy;     // 矩形高度
} Rectangle;

定义两个矩形结构体变量并赋值：
Rectangle Rect1, Rect2;
Rect1.x = 100;
Rect1.y = 100;
Rect1.dx = 100;
Rect1.dy =100;

Rect2 = Rect1;
```

上面的赋值在 C 语言中是支持的，编译器会将 Rect2 = Rect1 中的值转化成内存拷贝的 CPU 指令来实现赋值操作。可以想象，对于简单的变量赋值，CPU 只需要执行一个 MOV 指令就可以完成了，因此对于包含多个简单变量的结构体来说，使用多个循环的 MOV 指令就在情理之中了（在早期 16 位 CPU 中，如果对一个 32 位的 int 变量执行赋值操作都需要两条 MOV 指令）。在使用这种赋值方法的时候需要注意的是，在这个结构体变量中最好不要有指针变量，因为指针变量可能在变量 1 中指向一个分配的内存区域，当变量 2 通过赋值操作获得了这个指针值的时候，有可能这个指针已经释放了，这样就导致了空指针情况的发生，后果是使用这个指针的时候将会导致程序崩溃。举例说明如下：

1、定义一个包含指针类型的结构体：

```
typedef struct _TestStruct{
    int nMember;
    int *Ptr;
}TestStruct;
```

2、定义两个这种类型的变量并采用如下使用方法：

```
TestStruct Struct1, Struct2;
Struct1.nMember = 100;
Struct1.Ptr = (int *)malloc(15*sizeof(int)); // 分配 15 个 int 变量的空间

// 结构体赋值
```

```
Struct2 = Struct1; //此时 Struct2.Ptr 与 Struct1.Ptr 的值相等

if(Struct1.Ptr)
{
    free(Struct1.Ptr);
    Strict1.Ptr = NULL;
}

// 这里有很复杂的处理，其中包含了 malloc 等操作

Struct2.Ptr[0] = 2; // 错误的赋值操作，因为此时 Struct2.Ptr 所指向的内容已经被释放了。
```

对于程序来说，修改一个已经释放了空间的内存地址内容是十分危险的。当然，如果程序只有上面那么简单的话也不会出现什么严重的问题，顶多只是非法使用了一块内存区域；但是，如果中间含有复杂的处理，Struct2.Ptr[0] = 2 将修改程序其他部分使用的内存区域，那么这样就可能会有莫名其妙的死机之类的事情发生了。由于其发生问题的时间不固定，因此这类问题调试起来也十分的困难。

2.2.2 结构体嵌套

在一个结构体中可以声明另一个结构体，形成结构体嵌套，如果将内部嵌套的子结构体变量放在父结构体的顶部，那么两个结构体之间还可以进行类型互换。这个特性为实现 C 语言的数据封装提供了一种方法。例如定义如下结构体：

```
typedef struct _Point{
    int x;
    int y;
}Point;

typedef struct _Rectangle{
    Point LeftTop;
    int     dx;
    int     dy;
}Rectangle;
```

由于结构体中 Rectangle 嵌套了结构体 Point，因此如果定义变量 Rectangle Rect1 则 Rect1 可以转化成 Point 使用。例如：

```
Rectangle Rect1 = {{100, 100}, 100, 100};
Rectangle *pRect = &Rect1;
Point *pPoint = (Point *)&Rect1;
```

如果需要访问这个矩形的左上角的 x 坐标值可以有两种方法：pRect->LeftTop.x 或者 pPoint->x。Rectangle 结构体的内存模式如图 2.4 所示：

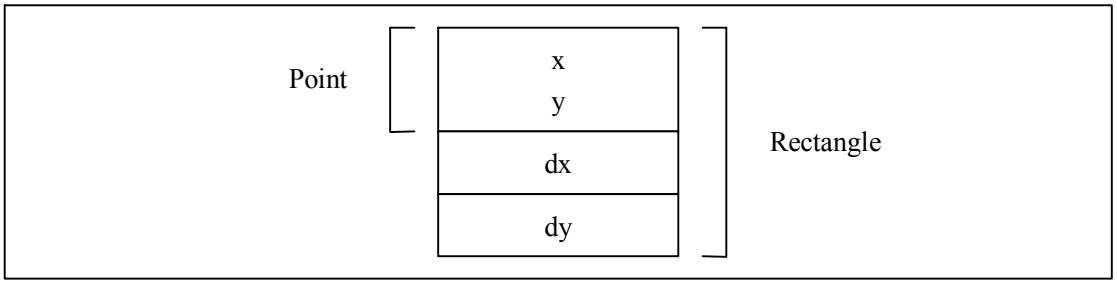

图 2.4 结构体内存模型

从图 2.4 可以看出 Point 是嵌入在 Rectangle 中的，两个结构体的顶端地址是一样的，因此他们之间的指针可以互换，并且可以正常操作。

2.3 重温 C 语言的预处理

在编译器编译源文件之前，会首先通过预处理器来处理源程序中的预处理选项。大名鼎鼎的宏也是预处理的一种，预处理器采用直接替换的方式来处理宏，也就是说将宏定义的内容替换到源文件中之后才开始编译，在每一个调用宏的地方就有一个宏的替换体。例如定义如下宏：

```
#define MAX(a, b) (a > b ? a : b)
```

在程序中使用一次 MAX(a, b)对应的预处理器就把(a > b ? a : b)替换到相应的位置，结果是增加了可执行程序的大小（因为有 n 个重复的代码段）。

如果定义的需要多行的宏，则使用“\”做为行与行之间的连接符，请看下面的宏定义：

```
#define INTI_RECT_VALUE( a, b )        \
{                                      \
    a = 0;                             \
    b = 0;                             \
}
```

注意最后一行就不再使用“\”了。

除了宏之外还有代码的选择编译也是预处理器的主要功能之一。在一个大型的软件项目中，有许多功能需要根据不同的硬件平台或者软件用途来进行选择，例如一个软件的中文版和英文版，在发布的时候就需要使用定义的中文或英文标签来决定。

C 语言的预编译器使用的关键词和功能描述如下表：

关键词	功能描述
#define	用来进行宏和符号或常量的定义。
#undef	取消通过#define 定义过的符号。
#if	用来判断预处理条件，需要#endif 做为结束标记。相对应的#ifdef 和#if defined 用来判断符号是否定义；#ifndef 和#if !defined()是判断符号是否未定义。
#else	#ifdef、#ifndef 的条件分支语句
#endif	#ifdef、#ifndef 的条件结束语句，只要有#if 就需要有#endif
#error	无条件的向预处理器报错。通常用在#if…#endif 之间，用来判断是否符合编译条件。例如： #ifndef ENABLE_COMPILE #error Disable compile #endif

#include	用来包含文件。通常是用来包含头文件，但实际上它什么文件都可以包含。它直接将文件的内容引入到当前的包含文件中，这些包含都是由预处理器完成的。
#pragma	指定编译器的参数，这个和具体的编译器有关。例如有些编译器支持 startup 和 exit pragmas，允许用户指定在程序开始和结束时执行的函数。 #pragma startup load_data #pragma exit close_files
__FILE__	预处理常量，代表当前编译的文件名。例如可以使用如下代码输出当前的文件名：printf(“This file name is %s”,__FILE__);
__LINE__	预处理常量，代表当前编译的行数。例如可以使用如下代码输出当前的行数：printf(“Current line is %d”,__LINE__);
__DATE__	预处理常量，代表当前编译的日期。例如可以使用如下代码输出当前的编译日期：printf(“Current compile date is %s”,__DATE__);
__TIME__	预处理常量，代表当前编译的时间。例如可以使用如下代码输出当前的编译时间：printf(“Current compile time is %s”,__TIME__);

在使用预处理的时候需要注意两件事情：

1、在定义宏或常量时候尽可能的使用括号。这是因为预处理器是将宏和常量采用直接替换的方式，如果不是用括号则有可能产生错误的程序处理。看下面的代码：

```
#define DISPLAY1_HEIGHT 320
#define DISPLAY2_HEIGHT 240
#define DISPLAY_SUM DISPLAY1_HEIGHT+ DISPLAY2_HEIGHT
…
if(DISPLAY_SUM*2 >200)
{
    …
}
…
```

上面的判断语句的真实想法是如果显示高度之和的 2 倍大于 200 则进行相应的处理。而实际上进行预处理后上面的代码变成了 if(DISPLAY1_HEIGHT+ DISPLAY2_HEIGHT*2)也就是 if(320+240*2)，这与我们期望的值相差太远了。因此在定义的时候最好加上括号，这样就不会出问题了。

2、在使用宏的时候必须不能出现参数变化。请看下面的代码：

```
#define SQUARE( a ) ((a) * (a))

int a = 2;
int b;
b = SQUARE( a++ ); // 结果：a = 4，即执行了两次增 1。
```

正确的用法是：

```
b = SQUARE( a );
a++; // 结果：a = 3，即只执行了一次增 1。
```

2.4 重温 C 语言的函数

理论上我们可以只使用一个 main 函数，因为不管多少程序我们都可以就写在一个 main 里面。但是您能设想有一个 10 万行的 main 函数吗？所以我们不得不把程序分成各个模块，不但要分成函数，而且还要将不同类型的函数放在不同的文件中。就我的经验而言，通常程序多于 200 行的时候应该考虑分成函数，而一个文件中的总共代码最好不要超过 5000 行。多了就不利于调试和阅读。当然，这些内容只是为了增强代码的可读性和易管理性的问题，但是您也要知道，软件在达到一定规模后就不单单是技术问题了，还有管理的问题。

2.4.1 形参和实参

直到现在我还我还记得在我初学 C 语言的时候形参和实参给我带来的困惑，我期望能够用非常简单明了的语言来描述它们，让我们来试试吧！

形参：在函数定义的时候使用的参数叫做形参。

实参：在使用函数的时候传递的参数叫做实参。

举例说明：

```
int add(int a, int b) // 这里是函数的定义，a,b 都属于形参
{
    // 这里全部都是函数的实现
    return(a+b);
}
```

上面的函数展示了一个函数的各个部分——定义和实现，在函数定义里的 a 和 b 都是形参，在函数的实现体内，也就是 a+b，中的 a 和 b 是实参的替身。这里是最容易让人困惑的地方，看一个例子吧，假设有一个函数 calc 需要调用 add 函数来实现加法的运算功能：

```
int calc(int calctype, int p1, int p2) // calctype、p1 和 p2 都是形参
{
    if(calctype == 1)
    {
        return add(p1, p2);      // 在这里的 p1 和 p2 是形参还是实参？
    }
    return 0;
}
```

根据我们对形参和实参的定义，p1 和 p2 对于 calc 函数来说是形参，但是对于 add 函数来说是实参。哦，恍然大悟了没有？对于判断到底是实参还是形参是有参考坐标的。同样的此时在函数 add 内部 a+b 等同于 p1+p2，因此说在函数的实现体内形参（形式上的参数）是调用时实参（实际上的参数）的替身。相当于形参为实参占了一个位子，使用函数的时候实参就坐在了这个位子上。

如果您理解了上面的内容，那么在使用函数的时候就再也没有必要纠缠于实参和形参了，因为它们合起来完成了一个参数的传递过程。调用函数的时候，按照形参的类型传递相应的变量就可以了。形参和实参只是一个概念上的区分，在实际的二进制层次并没有这个概念，因此我们不要过分拘泥于此，理解了就好。

2.4.2 宏与函数的比较

对于一个初级的程序员来说，什么时候使用宏，什么时候使用函数还是有些晕的，这里我也顺带的提一下吧。从前面可以知道，当预处理器遇到宏的引用时，都是将它用宏语句进行替换。因此，在每一个宏引用的地方都会增加相应的宏代码，结果就是使得编译后的代码加大。而函数则是直接调用函数的代码，不会增加编译后的代码大小。不过使用函数的缺点是在函数调用的时候会增加一些额外的处理，这使得执行函数的时间比相应的宏代码的执行时间要长一些。因此如果在需要高效率的时候就应该使用宏，如果需要程序变得更小则使用函数。关于函数的额外处理部分的信息将在下一节编译器基础中进行介绍，到那个时候我们就会对函数是如何执行的问题有一个全面的了解了。

2.4.3 函数指针

还记得指针吗？什么，忘了！对您竖起我的大拇指，恭喜您已经修成正果了，因为在您的眼里已经没有了对指针特别的概念。还记得我们在介绍指针的时候说的吗，指针就是一种变量。那么函数呢？其实每个函数也是一样，每个函数名就是一个函数的首地址，因此我们也可以定义一个指针变量来存储它，这个就是函数指针——指向一个函数的指针。这一节将让您从另一个深度上加深对指针的认识。

函数指针在多任务的操作系统环境下是十分有用的。在多任务操作系统环境下，由于牵涉到多个任务之间的交互，因此通常使用函数指针的形式将一个函数做为参数传递给另一个任务，以便于当另一个任务完成操作之后可以调用当前任务的函数通知状态。这个由“另一个任务”调用的函数就是大名鼎鼎的回调（Callback）函数，这个回调函数的传递就是通过函数指针这个载体实现的。回调函数还常用在定时服务的程序里，在设置一个定时器的时候我们会指定一个回调函数，用来在到达定时时间的时候调用以做出相应的处理。

通常情况下使用如下方式定义一个函数指针类型：

```
typedef int (*FunctionType)(int param1, int param2);
FunctionType pFunction;
```

使用的时候可以为 pFunction 赋值，在这里 FunctionType 的函数类型与我们上面的 add 函数的形式相同，因此我们可以改用下面的代码来调用 add 函数：

```
int nResult;
FunctionType pAddFunc = add;

nResult = pAddFunc(1,2);   // 返回的结果是 3
```

在这里之所以使用 typedef 来定义一个类型是为了保持在 C 语言环境下的函数调用一致。为了更好的理解函数指针，我将使用一个 void 型的函数指针来传递这个函数。打开 Visual Studio 2003 .Net，新建 VC 控制台工程 Test3（请参考前面 Test1 的创建过程），输入如下代码：

```
#include "stdafx.h"

typedef int (*FunctionType)(int param1, int param2);
```

```
int add(int a, int b)
{
	return(a+b);
}

int _tmain(int argc, _TCHAR* argv[])
{
	int nResult;
	void *pVoid = add; //  通过 Void 型的指针传递函数指针

	nResult = ((FunctionType)pVoid)(1,2);

	printf("Result is %d\n",nResult,0,0);
	return 0;
}
```

编译链接运行，可以看见输出结果是 3。

从上面的例子可以看出，一个函数名其实就是一个地址，可以直接赋值给指针，并且通过指针类型的转换来实现函数的调用。进一步，如果 FunctionType 类型与函数 add 之间的参数和返回值不一样可以吗？还是让事实说话吧，将上面的程序修改成如下样式：

```
#include "stdafx.h"

typedef int (*FunctionType)(int param1);

int add(int a, int b)
{
	return(a+b);
}

int _tmain(int argc, _TCHAR* argv[])
{
	int nResult;
	void *pVoid = add; //  通过 Void 型的指针传递函数指针

	nResult = ((FunctionType)pVoid)(1);

	printf("Result is %d\n",nResult,0,0);
	return 0;
}
```

编译链接，可以通过。运行，输出结果是 2012749654，好大的数字！

从上面的代码情况看出，程序是可以运行的，只不过输出的结果不对。这是因为 add 需要两个参数，而我们调用的时候却只有一个参数。那么另一个参数是什么呢？答案是一个随机数。前面我们说过，函数中会通过形参为实参留个“位子”，那么现在有两个“位子”却只用了一个，那么可以预见的，这个空的位子中就是一个随机数了。

如果到现在为止，您对这个例子还不是很明白的话，请您一定要再重新体会一下，因

为这部分的内容对于您理解程序会有很大的帮助。

2.4.4 不要使用结构体或数组做为函数的参数

在前面讲解函数参数的时候我们曾经提到过，函数会通过声明的形参为实参留“位子”，那么这是一个多大的“位子”呢？答案是和形参的数据类型的大小一样，而且这个位子是存放在系统的堆栈中的。结果是参数的数据类型越大，需要的堆栈空间就越大，这对于内存空间很小的系统来说是很不利的。而且在进行实参传递的时候还需要将实参的内容拷贝到对应的“位子”中，因此就增加了很多的调用函数时的额外处理。（这里称呼成堆栈，实际上应该是“栈”，因为我们目前习惯于称呼“栈”为堆栈。“堆”通常指的是一个使用 malloc 等函数分配的内存空间。）

基于上述原因，推荐不直接使用大的结构体或数组做为函数的参数，请使用指针进行传递，这样可以省掉很多调用函数的额外处理，提高函数的执行效率。同时占用的堆栈空间比较小，减少了堆栈溢出的可能性，因此使用指针传递参数是一个多赢的方法。千万不要因为害怕指针而不使用它，做为一名 C 语言的程序员是永远也逃脱不了指针的，况且一旦掌握了它的精髓，您就会对它爱不释手了。

2.4.5 使用指针参数传递数据

指针做为 C 语言的灵魂具有很强的技巧性和灵活性，而做为参数传递数据则是它非常重要的应用之一。这里的“传递”有两层含义：传入和传出。

传入就是调用函数的地方将数据传递给函数进行处理，例如一个数组排序的函数，需要通过指针将数组传递给函数使用。这里以一个进行字母排序的函数为例，代码如下：

```
void CharSort(char *pString)
{
    char *pCurrPtr;
    char temp;

    // 判断参数的合法性
    if(pString == NULL)
    {
        return;
    }

    // 排序算法
    for(;*pString!= '\0'; pString++)
    {
        for(pCurrPtr = pString+1; *pCurrPtr != '\0'; pCurrPtr++)
        {
            // 将最小的放在当前位置
            if(*pString > *pCurrPtr)
            {
                // 交换数据
```

```
                    temp = *pString;
                    *pString = *pCurrPtr;
                    *pCurrPtr = temp;
                }
            }
        }
}
```

这个函数是一个简单的冒泡排序算法，使用这个函数需要传递一个 char 类型的数组。此时这个指针参数 pString 就包含了传递参数给函数使用的功能。正如上一节所讲的一样，这里面使用指针来传递数据是十分高效的，只需要在函数栈空间内分配一个指针空间就好了。

指针参数的另一个作用是可以传出参数，返回函数的处理结果。通常情况下我们都是通过函数的返回值来返回处理结果的，但是如果遇到需要返回一个大数组或大数据结构的时候，使用返回值传递参数就显得效率低下了。我们仍旧以上面的 CharSort 函数为例，这个函数的参数其实就具有传入和传出数据两种功能，它首先通过 pString 参数将排序的字符串传递给函数，然后函数的处理结果也是直接通过 pString 来写入原存储空间的。这个过程的示意图如下：

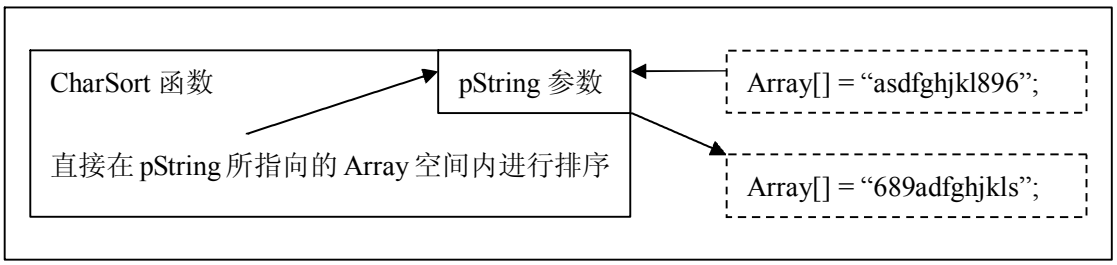

图 2.5 指针参数的传入传出数据功能

使用指针参数传出数据在编写 C 语言程序时是十分普遍的，除了上面的传递数组外，还可以传递单个数据，也可以通过二重指针传递指针值。总之，指针做为参数使用是十分灵活多变的，各位读者可以仔细体会，关键的是理解指针的本质意义——地址类型的变量。

2.5 C 语言中几个特殊的关键词

在这里我只是简单地介绍一下 volatile、__packed 和 const 的作用，省得我们在看到它们的时候不知所措。

2.5.1 volatile 关键词

volatile 的中文意思是“易挥发的”，它主要是给编译器提个醒，告诉编译器对于 volatile 变量不要轻易的进行优化，因为在程序运行过程中这个值会被其他的任务或硬件改变。在编译器中对于语句通常会做一些优化，例如有如下程序：

```
char bExit = 0;
…
for(;;)
```

```
{
    …
    if(bExit)
    {
        break;
    }
}
```

假设现在有另一个任务或线程通过 bExit 来控制程序的退出。如果此时变量不使用 volatile 关键字说明的话，编译时就会对 if(bExit)进行优化，不再在每一次 for 循环中判断 bExit 了，这样就会导致程序运行错误。因此，此时应使用 volatile 关键字说明 bExit 变量，这样编译器就不会做这样的优化了。

2.5.2 __packed 关键词

__packed 用来声明结构体采用单字节偏移。并不是所有的编译器都支持这个选项。使用__packed 声明的结构体会压缩空间。例如有下面一个结构体：

```
struct _Test{
    int a;
    char b;
    char c;
    int d;
}Test;
```

如果不使用__packed 声明，在 ARM 编译器中 sizeof(Test)等于 12（在 ARM 编译器中是 4 字节偏移，int 也是 4 字节变量）。加入__packed 说明后，sizeof(Test)等于 10，编译器会压缩 Test 结构体中 b、c 和 d 变量之间的 padding 字节。对比示意图如下：

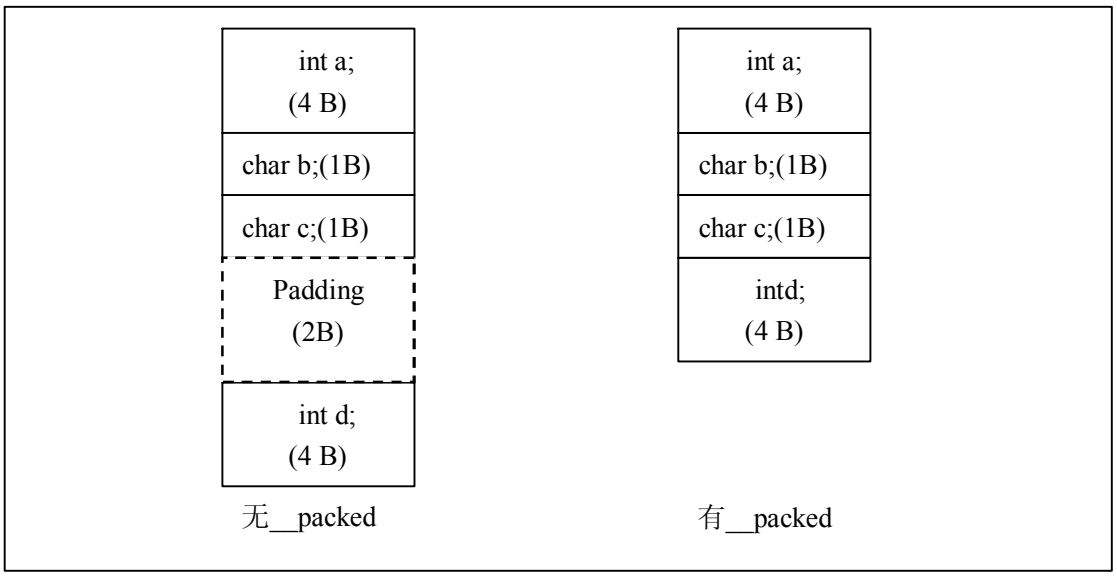

图 2.6 结构体内存映射

从这个图中可以看出，经过__packed 说明之后的结构体，相对于没有使用__packed 说明的节省了 2 字节的 padding 存储空间，实际上这给我们提供了一种紧凑数据的方法。

2.5.3 const 关键词

使用 const 的好处在于它允许指定一种语意上的约束——某种数据不能被修改——编译器具体来实施这种约束。通过 const，我们可以告知编译器和其他程序员某个值要保持不变。只要是这种情况，我们就要明确地使用 const ，因为这样做就可以借助编译器的帮助确保这种约束不被破坏。

对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const，还有，两者都不指定为 const：

```
char *p                  = "hello";      // 非 const 指针, 非 const 数据
const char *p            = "hello";      // 非 const 指针, const 数据
char * const p           = "hello";      // const 指针,非 const 数据
const char * const p     = "hello";      // const 指针, const 数据
```

语法并非看起来那么变化多端。一般来说，我们可以在头脑里画一条垂直线穿过指针声明中的星号（*）位置，如果 const 出现在线的左边，指针指向的数据为常量；如果 const 出现在线的右边，指针本身为常量；如果 const 在线的两边都出现，二者都是常量。标示为 const 的数据，在编译器中当作 RO 只读数据处理。

在指针所指为常量的情况下，有些程序员喜欢把 const 放在类型名之前，有些则喜欢把 const 放在类型名之后、星号之前。所以，下面的函数取的是同种参数类型：

```
void f1(const int *pw);          // f1 取的是指向，widget 常量对象的指针
void f2(int const *pw);          // 同 f2
```

2.6 地址对齐

我们知道，在计算机系统中都是使用地址来管理数据，每一个数据实体都存储在一定的地址空间内，如一个长度为 100 的 char 型数组存储在一个 100 字节的连续空间中。一个连续的地址空间中既有奇数的地址也有偶数的地址，这中间就存在了一个数据以什么样的地址做为起始地址的问题。许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数(通常它为 4 或 8)的倍数，这就是所谓的内存对齐。而这个系数则被称为该数据类型的对齐模数(alignment modulus)。

这种强制的要求一来简化了处理器与内存之间传输系统的设计，二来可以提升读取数据的速度。比如有这样的一种处理器，它每次读写内存的时候都从某个 8 倍数的地址开始，一次读出或写入 8 个字节的数据，假如软件能保证 double 类型的数据都从 8 倍数地址开始，那么读或写一个 double 类型数据就只需要一次内存操作。否则，我们就可能需要两次内存操作才能完成这个动作，因为数据或许恰好横跨在两个符合对齐要求的 8 字节内存块上。某些处理器甚至在数据不满足地址对齐要求的情况还会出错，

地址对齐的问题主要表现在两个方面：一是通过指针类型转换访问数据的情况；二是在结构体内部的数据对齐和访问。接下来我们将就这两个问题做一个阐述。

2.6.1 指针的数据类型转换

C 语言的指针是非常灵活的，我可以将任何一种数据类型的指针转换成另一种数据类型的指针，例如，将一个（char *）转换成一个（int *），或者将一个（int *）转换成（char *）。

这样的指针类型转换在 C 语言中十分的普遍，更有甚者，我们还可以定义一个无类型的指针（void *），可以不受限制的接受任何类型的指针，而编译器也不会提示任何错误。这既是 C 语言灵活的标志，但在其中也蕴藏着杀机！

我们可以假设这样的一种情况，我们从一个串口设备或者网络设备上接收 16 个字节的数据，这些数据都是使用字节方式传送的。此时我们规定，这些数据的第一个字节表示数据的类型，而其后的数据则表示真正的数据内容。此时我们会定义这样的一个数组来接收数据：

```
byte rsv_buf[16];
```

现在，我们假设数据已经存储在这个数组中了，同时假设这个数组在内存地址中的 0x0 至 0xF 的空间内。我们通过对这个数组第一个字节 rsv_buf[0]的判断，知道这里面存储的是一个 word（双字节）型的数据。于是我们使用下面的方式获得了数据：

```
*((word *)&rsv_buf[1]);
```

这里面发生了什么事情呢？我们先对 rsv_buf[1]进行取址操作，然后转换成 word 型的指针，然后通过一个取值运算符“*”来获得这个 word 型指针中的数据。那么此时&rsv_buf[1]到底是一个什么样的值呢？我们知道 rsv_buf 的首地址是 0，那么当然了&rsv_buf[1]的值就是 1 了。我们现在是要从一个奇地址中获得一个双字节的变量。

这个时候地址对齐的问题出现了。当然对于一个支持这种访问方式的 CPU 来说，不会出现任何问题，那么对于不支持这种运行方式的 CPU 呢？一个错误产生了，或许系统崩溃了，或许获取了错误的数据。不管怎样这都将打破系统的正常运行，而且 CPU 不会为我们提供任何可以更正错误的机会。

由于我们并不能准确地知道我们所写的程序将来会运行在什么类型的 CPU 上面，也不知道它是否支持非地址对齐方式的数据访问，因此我们应该尽可能的避免在程序中定义这样的数据结构。即便不可避免的定义了这样的数据结构，也要提供一种转换的机制。比如，在本例中我们可以在定义一个 rsv_data 来转存&rsv_buf[1]地址之后的数据，然后再进行类型转换，这样就不会出现问题了。当然，即便我们清楚的了解我们的程序将要运行的 CPU 支持非对齐方式的数据访问，那么也要尽可能的避免这种情况的发生，因为它将影响我们的程序的执行效率，通常 CPU 执行这些代码需要更多的指令周期。

2.6.2 结构体内存布局

地址对齐的问题也表现在结构体中。C 语言规定一种结构类型的大小是它所有字段的大小以及字段之间或字段尾部的填充区大小之和。填充区？是的，这就是为了使结构体字段满足地址对齐要求而额外分配给结构体的空间。C 语言的标准规定结构体类型的对齐要求不能比它所有字段中要求最严格的那个宽松，可以更严格。我们可以看下面的例子：

```
typedef struct _Example1
{
    byte    a;
    dword   b;
} Example1;
```

我们现在定义了一个结构体 Example1，在其中有两个成员变量 a 和 b。假设这个结构体按照连续的方式布局，那么这个结构体将占用 3 个字节的空间，a 成员在结构体中的偏移是 0，那么结构体成员 b 的偏移则是 1，而成员 b 是一个四字节的变量。这中间就存在了一个地址对齐的问题了。如果假设我们现在是编译器，我们将如何安排这个结构体的内存结构呢？按照 C 语言的标准，这个结构体内成员变量对齐要求取成员地址对齐要求之大者的原

则，这个结构体中的地址对齐的模数应该是 sizeof(word) = 4。实现这种对齐的唯一方式就是在成员 a 和 b 之间增加 3 个字节填充区。增加填充区后的内存模式如下图 2.7 所示。

图 2.7 Example1 的结构体内存模型

这个方案在 a 与 b 之间多分配了 3 个填充(padding)字节，这样当整个结构体首地址满足 4 字节的对齐要求时，b 字段也一定能满足 dword 型的 4 字节对齐规定。那么 sizeof(Example1)显然就应该是 8，而 b 字段相对于结构体首地址的偏移就是 4。下面我们再来看一看将这两个成员变量调换位置之后的情况。

```
typedef struct _Example2
{
    dword   a;
    byte    b;
} Example2;
```

或许您会认为 Example2 的内存布局会比 Example1 的简单，就是一个 4 字节的变量 b 加上一个 1 字节的变量，总共是 5 个字节长度。因为 Example2 结构同样要满足 4 字节对齐规定，而此时 a 的地址与结构体的首地址相等，所以 a 和 b 都一定也是 4 字节对齐。嗯，分析的有道理，可是不全面。让我们来考虑一下定义一个 Example2 类型的数组会出现什么问题。C 标准规定，任何类型(包括自定义结构类型)的数组所占空间的大小一定等于一个单独的该类型数据的大小乘以数组元素的个数。换句话说，数组各元素之间不会有空隙。按照上面的方案，一个 Example2 数组的布局就是如图 2.8 中所示的一样。

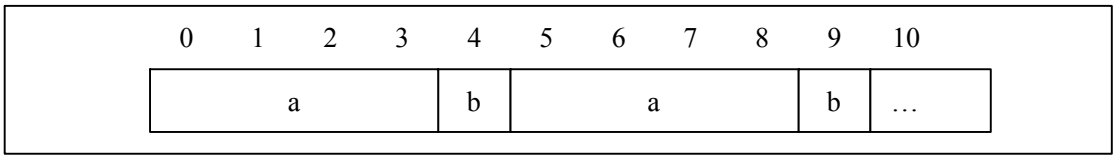

图 2.8 Example2 数组内存模型

我们可以看到，此数组的第一个成员变量已经是四字节对齐了，可是第二个成员变量的起始地址却是 5 开始的。这就不能满足 C 语言的要求，因此在 Example2 类型的数组中，依然要增加填充区，如图 2.9 所示。

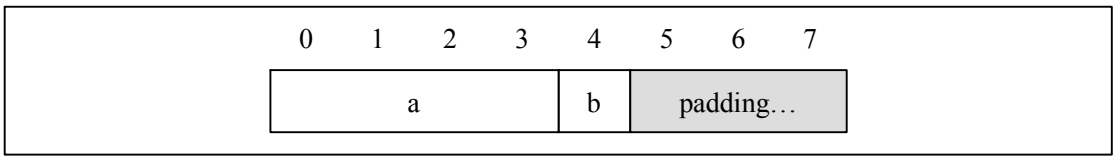

图 2.9 Example2 的结构体内存模型

现在无论是定义一个单独的 Example2 变量还是 Example2 数组，均能保证所有元素的所有字段都满足对齐规定。那么 sizeof(Example2)仍然是 8，而 a 的偏移为 0，b 的偏移是 4。现在我们已经掌握了结构体内存布局的基本准则，尝试分析一个稍微复杂点的类型吧。定义一个 Example3 的结构体：

```
typedef struct _Example3
{
    byte    a;
    word    b;
```

```
    dword    c;
} Example3;
```

这里面有歧义的地方就是 b 这个变量采用什么样的对齐方式。在这个结构体中最大的偏移是 4 字节，那么成员 b 应该是满足 4 个字节的地址对齐方式吗？实际情况是变量 b 只需要满足它自身的对齐方式就可以了，也就是 sizeof(word) = 2。因此现在我们可以得到图 2.10 的内存布局结构。

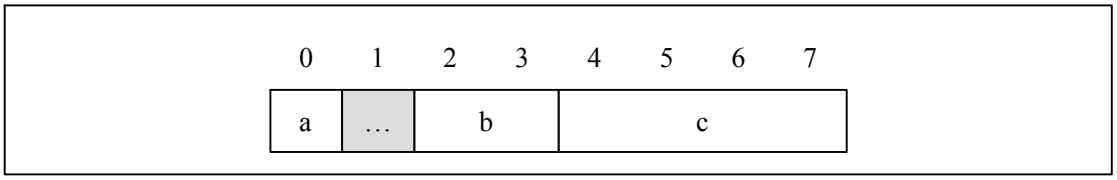

图 2.10 Example3 的结构体内存模型

那么现在我们可以知道，在结构体内部成员变量地址对齐的要求就是满足自身的需求，单字节的变量可以紧接着前面的成员，双字节的要求偶数地址对齐，四字节的要求 4 字节对齐等等，可以依次类推，如果紧接着上一个成员的地址不符合要求就在中间添加填充字节。而对于整个结构体要求的地址对齐方式则取成员变量中要求地址对齐最大的那个。

在实际开发中，我们可以通过指定编译选项来更改编译器的对齐规则（不同的编译器有不同的设置方式，请参考相应的编译器文档）。例如我们可以指定字节对齐的方式是 8，也可以指定是 4，甚至还可以是 1。在设置对齐规则的时候，采用的是参数与默认取二者之小的方式。例如我们通过编译器参数设置结构体偏移量为 2，那么对于 Example1 中的的内存布局就会变成图 2.11 中表示的那样。

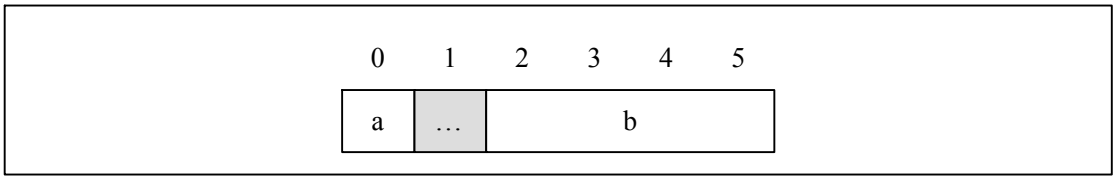

图 2.11 Example1 的 2 字节偏移内存模型

此时仅仅会增加一个字节的填充区。虽然结构体中 b 成员的偏移是 4，但是由于我们设置了编译器的偏移参数为 2，因此将会使用 2 作为此结构体的最大偏移。如果此时我们将这个结构体的偏移设置为 1，那么无论这个结构体的成员排列如何，都不会有任何的填充字节。此时的情况就类似于我们前面所讲的编译器__packed 关键词的意思了。

在这种编译器设置的字节对齐要求比结构体中变量自然要求小的情况下，将会出现访问成员变量时的地址对齐问题。就像图 2.11 中所表示的一样，成员 b 是一个 4 字节的变量，但是它的首地址却是 2，不能被 4 整除的一个地址，我们在使用这样的结构体成员的时候，会同指针的转换的时候一样出现地址对齐的问题吗？

如果我们不做任何事情，肯定会出现问题。不过请您放心，这些事情已经由编译器为我们做了。这是可以理解的，我们使用编译器设置了结构体的字节对齐要求，不过出现问题当然要由编译器负责了。在这种情况下使用结构体成员的时候，编译器在编译相应的代码时，会额外的插入一些 CPU 指令来消除地址对齐问题的影响。典型的，在 ARM C 语言编译器中，如果访问一个使用__packed 关键字声明的结构体成员变量的时候（如使用 pointer->b 语句访问 b 变量），每一个访问成员的 C 语句都会变为 7 个 CPU 指令，同时使用 3 个通用的 CPU 寄存器。这不但会影响程序执行效率，而且还会增加代码的尺寸。所以，不到万不得已，请不要使用这种方式的结构体。

2.7 小结

噢，我要长长的松一口气了，终于将一些 C 语言的基础介绍给完了。在这一章里，主要介绍了 C 语言的一些特性，其中大部分是读者在使用中会遇到的令人迷惑的议题，如果我们能够将这一章的内容透彻的理解了，那么我们就基本上掌握了 C 语言本身的精髓（虽然还有很多高级的应用，不过那都不是 C 语言本身的了）。在这里我要提醒那些初级的程序员朋友们，在掌握一门程序语言技术的同时，千万不要忘记养成一个良好的编码风格。因为如果想成为一名优秀的程序员不但要能够写出复杂的程序，而且要能够将复杂的问题用容易阅读的程序来实现。最好的程序是要大家都能看懂的，而不是只有您一个人能看懂的程序。记住一个程序的 KISS 准则（此 KISS 非彼 Kiss 也）：Keep It Simple and Stupid，也就是保持程序简单性和傻瓜性，别人能容易看懂的程序才是好程序。

思考题

1、指针和变量的相同点和区别的地方在哪里？
2、函数的名称可以赋值给一个 int 型的指针变量吗？
3、如果没有 sizeof 怎样能够获得一个结构体的大小？

第三章 编译器基础

看了这个题目，请不要误会我要告诉您编译器是怎么实现的，我写这节的主要目的是告诉您通常编译器是怎样对待您所写的程序的。大家都知道，程序最终都要在 CPU 上运行，那么像 C 语言这样的高级语言来说，编译器就是联结 C 和 CPU 之间的桥梁了。在这一节里我要告诉您编译器是如何充当这个“桥梁”角色的。

本节首先讲述一下程序员的层次问题，目的是让我们自己知道需要成为一名什么样的程序员；然后分别是编译器的相关介绍以及编译器是如何“对待”程序等内容。

3.1 软件和程序员的层次

如果要了解程序员的层次，那就要先看看程序的层次了。看图 3.1：

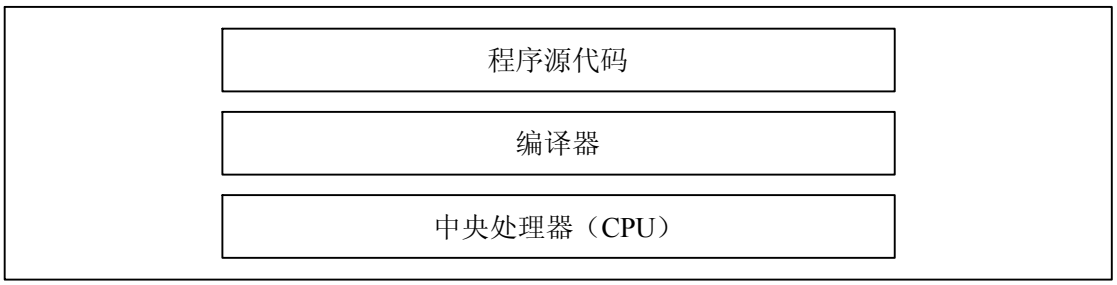

图 3.1 程序的层次

上图的语言描述就是程序源代码经过编译器生成可以在 CPU 运行的二进制代码。由此衍生出了程序员的两个层次——语言层次和二进制层次。

所谓的语言层次的程序员，就是那些可以使用某一个编程语言的程序开发人员。大多数的初学者都属于这个层次，能够遵照一定的规范完成编码工作，但是对下面的事情（编译器和 CPU）就一知半解了。

所谓的二进制层次的程序员，就是那些熟练地掌握某一个编程语言并且知道这个语言的来龙去脉的程序开发人员。他们对程序有着深刻的认识，对于程序中每一部份发生的事情了如指掌。二进制层次的程序员要比语言层次的高，是软件中的高手，因此也比较难得。对于二进制层次的程序员来说，可以非常快速的掌握一门开发语言，因为对于他们来说在二进制层次上只有 CPU 指令的差别而没有语言的差别。不过要想成为一名这样的程序员是需要付出很多努力，并积累足够的编程经验才可以。

程序语言所要解决的问题就是如何更加高效的组织二进制代码。例如，汇编语言是对 CPU 指令的符号化，目的是为了容易使用 CPU 指令从而提高编码效率。由于汇编语言对二进制代码基本没什么组织，因此称它为“低级语言”。C/C++语言对二进制代码的组织有了很大的提高，制定了完全脱离了 CPU 指令的语法规则，大大提高了程序开发效率，因此我们称它为“高级语言”。从这个角度来讲，二进制始终是任何程序开发语言的归宿，因此在二进制层次理解程序是十分必要的。

然而，正是这些高级语言制定了脱离二进制指令的语法规则，将大部分二进制细节都隐藏在了编译器之内，因此加大了程序员与二进制之间的距离，使得成为一名二进制层次的程序员更加困难了。在这一节里我要向您揭示那些隐藏在编译器里的二进制细节，期望能够为您成为一名二进制层次的程序员提供帮助。

3.2 编译器的分类和作用

从不同的角度可以进行不同的编译器分类，因此在这里列举一些典型的编译器并一一进行讲解，这样我们就可以更加直观的看到各个编译器之间的不同部分。这些典型的编译器分别是：汇编语言编译器、Borland C/C++编译器、ARM C/C++编译器、VC++编译器和典型的Java 编译器。由于本书主要使用 C 语言相关的工具，因此这里主要列举的是 C 语言相关编译器。

还有重要的一点是，在本书中大部分的编译器指的是编译器和链接器，除非它们两个摆在一起。

3.2.1 汇编语言编译器

我们知道，汇编语言只是简单的对 CPU 指令进行了封装（封装也就是包装的意思，通常计算机术语里这么称呼），因此汇编语言编译器是与 CPU 相关的。结果是使用汇编语言编写代码的可移植性很差，因为不同的 CPU 可能指令系统和寄存器都不一样。了解计算机历史的人应该知道，最初写的程序都是二进制的，就像是最原始的打孔机之类的东东，需要每个程序员记住每个指令的二进制值（也就是 0 和 1 的组合），编程效率可想而知（向那一代程序员致敬！）。不过汇编语言的出现使得编程效率大大的得到提升，程序员再也不需要记忆指令的二进制值了。也是从这里开始，程序效率提升的步伐大大加快了。紧接着，高级语言和计算机操作系统的飞跃发展更为提高编码效率创造了良好的条件。

3.2.2 Borland C/C++编译器

这个编译器是在 DOS 操作系统时代非常流行的 C 语言编译器。从这句描述可以看出它是与操作系统相关的，当然要完成在 CPU 上运行是一定要和 CPU 相关的。通常对于一个编译器来说会根据不同的 CPU 类型生成不同的二进制代码，这通常通过为编译器传递参数来实现。

现在可能有些人会问一个鸡生蛋还是蛋生鸡的问题：编译器是由谁来编译的？如果一定要追根溯源的话，我会告诉您，世界上第一个汇编语言编译器是用机器码写的，因此不需要编译；世界上第一个高级语言编译器是由汇编语言写的，操作系统也在这个过程中不断的发展。也正是由于不同 CPU 和操作系统平台以及编译器的交互发展，才产生了当今世界如此的软件规模。也可以想象的到英特尔和微软等欧美企业屹立不倒的原因了，因为他们伴随着计算机产业一同成长了那么多年。

3.2.3 ARM C/C++编译器

ARM 是当前嵌入式系统中最为常用的 CPU 内核芯片解决方案。由于 ARM 主要用于嵌入式系统，且嵌入式系统的操作系统千差万别，因此它的目标环境要求与操作系统无关，甚至于操作系统的源代码也是与应用程序代码一起编译的。因此从这个意义上来说，嵌入式系统对于二进制的表现是最完全的，从中我们可以一窥整个系统二进制层次的原貌。在以后的章节中我们也将主要围绕着 ARM 编译器进行讲解。

3.2.4 VC++编译器

VC++编译器是微软的基于 Windows 操作系统的 C/C++编译器，它是紧密的与 Windows 操作系统相关的。也正是由于它和 Windows 操作系统联系之紧密，导致它与跨平台没有了任何的关系。如果说对于普通的编译器，如 ARM C/C++编译器、Borland C/C++，由于其需要实现的平台相关代码较少，我们可以比较容易的实现跨平台无关的代码开发，那么，VC++的编译器就十分的困难了，基本上不可能。

3.2.5 Java 编译器

Java 是号称跨平台的，没错，确实是这样子。Java 能够实现跨平台的前提条件是需要在这个 Java 平台上实现 Java 虚拟机，这个虚拟机可就不是跨平台的啦，而是与平台紧密相关的，每一种平台要实现各自的 Java 虚拟机。由此理解，Java 编译器编译的目标代码不是针对 CPU 和操作系统的，而是真对 Java 虚拟机的。所以也可以称 Java 语言是一种半解释型的语言。从这里还可以看出，Java 程序的执行效率比直接编译生成二进制的程序执行效率要低。

接下来看看上面的各种编译器与 CPU 和操作系统的关系。看下表：

名称	与 CPU 相关性	与操作系统相关性
汇编语言编译器	强相关	无关
Borland C/C++编译器	强相关	相关
ARM C/C++编译器	强相关	无关
VC++编译器	强相关	强相关
Java 编译器	无关	无关

为了我们知识的完整性，我想还是有必要提及一下脚本（Script）语言。脚本语言广泛的应用于网页设计中，嵌入到网页内部实现动态或交互的 Web 应用。典型的是 JavaScript 和 VBScript。脚本语言属于完全的解释型语言，不需要编译器，直接由脚本的处理程序生成结果。

3.3 编译器的数据处理方式

编译器的作用是将源代码编译成可以在目标平台运行的程序，核心要处理的就是程序员所写的程序。因此，对于编译器要处理的主要程序元素有：代码、局部变量、全局变量、静态变量以及常量，概括起来就是代码和数据。如何在二进制层面组织代码和数据就是编译器需要完成的工作。通常在二进制层面存在以下三种类型的二进制数据：

1、只读数据（RO Read Only）：程序和常量

2、可读可写数据（RW Read &Write）：有初始化值的全局变量和静态变量

3、零初始化数据（ZI Zero Initialize）：无初始化值的全局变量和静态变量

最终这些数据将会通过编译连接生成一个二进制的可执行文件，并将这些数据分别放在不同的可执行文件段（Section）中：只读数据放在 text 段中；可读可写数据放在 data 段中；零初始化数据放在 bss（Block Started by Symbols）段中。接下来我们将看一看这些数据究竟是如何组织的。

3.3.1 只读数据

只读数据是我们的程序在执行的过程中不能修改的数据，开动脑筋，仔细地想一想在我们的程序中那些数据是不能修改的呢？

第一个出现在我们脑海中的应该是那些常量。在 C 语言中，我们可以使用 const 关键字来声明一个常量。我们知道，一个变量按照其作用域可以分为全局变量和局部变量，对于一个全局的 const 变量来说，属于只读的数据是没有任何疑义的。但是对于一个使用 const 关键字的局部变量来说就有问题了，这些变量由于属于局部变量，因此在运行的时候仍然是放在堆栈中的。如果希望它变成无可争议的只读数据的话，那么我们需要同时声明这个变量为 static 类型的，因为在 C 语言中，静态变量采用了与全局变量一样的处理方式。

除了常量之外，第二个属于只读数据的就是代码本身。代码是数据吗，有没有搞错？不要心存疑问，事实的确如此！在编译器看来，不论时变量还是代码，都属于数据。恰恰是程序中的代码是只读数据的主力军。因为在运行的时候，代码是不可改变的，所以对编译器来说，代码属于只读数据来说就不奇怪了。

3.3.2 可读可写数据

可读可写数据是指那些在代码中指定了初始化变量的全局变量和静态变量。在 C 语言中，全局变量属于固定分配内存的方式，需要在链接的时候就为其分配固定的存储空间。这些存储空间属于这个变量私有，从系统启动到关闭为止都只能由这个变量使用。如果一个非常量的全局变量含有初始值，那么我们就需要首先存储这些初始值，并把它们保存在我们生成的二进制可执行文件中，同时为它分配一个 RAM 中固定的存储空间。当我们开始执行这个可执行文件的时候，需要将这些初始值从可执行文件中复制到它所对应那段固定存储空间中。

从某种程度上来说，可读可写的数据中的一步分需要存储在可执行文件的 text 区中，因为可读可写变量的初始值要存储在一个只读的空间中，在运行的时候才会复制到该变量对应的空间中。而这部分的初始值也就是只读数据的一步分了。这一点在不支持虚拟内存管理的嵌入式系统中表现得尤为突出。在采用这种方式的系统中是十分占用系统存储空间的，因为它既要占用代码段来存储初始值，而且也要占用同等的 RAM 空间来存储数据。

3.3.3 零初始化数据

零初始化数据就是在程序中定义的那些没有初始值的全局变量和静态变量。这些变量的特点就是固定分配存储空间，使用 0 做为初始化的值。这样，整个程序中零初始化变量就可以连接成为一个整片的内存区域，只需要知道地址区间，然后在初始化的时候都赋值为 0 就可以了。在执行程序的时候，展开 ZI 数据的信息，根据地址区间全部进行零初始化赋值操作。因此，零初始化数据之需要占用很少的空间来记录起始地址和结束地址就可以了。

在嵌入式系统中，由于操作系统的代码和数据也包含在这三种类型的数据里面，因此对这三种数据的处理方式就有了很大的不同。从前面的章节中我们知道，嵌入式系统主要有 CPU+ RAM+ Flash 的结构，因此对于系统中的只读数据是存储在 Flash 中的，而 RW 和 ZI 数据则是存储在 RAM 中的。Flash 芯片中存储的就相当于在 Windows 操作系统下的可执行文件（.exe），在系统启动的时候，将存储在 Flash 中的 RW 数据复制到对应的内存（RAM）

区域中，将 ZI 数据按照起始和结束地址零初始化相应的区域。

到现在为止，细心的读者可能会发现还没有提及局部变量的处理过程，他们是存储在什么地方的？首先说明这里所说的局部变量不包含局部的静态变量，因为不管是全局静态变量还是局部静态变量全部按照全局变量方式处理。局部变量产生于函数的内部，因此它的产生和消亡也都在函数里面。在程序开始执行某一函数的时候，会为这个函数内声明的局部变量在栈内分配空间，在函数结束的时候将这些空间弹出。由此可以看出，在使用局部变量的时候要考虑栈的空间大小。由于系统中的栈主要由操作系统来管理，并且通常分配的空间是有限的（尤其是在嵌入式系统中），因此不要在函数体内使用较大的数组。

3.4 编译和链接

当我们行进到这里的时候，我想我们不得不先澄清一下编译器和链接器的概念。在前面讨论的编译器中是包含了编译器和链接器两个部分的，用它来代表从代码到可执行程序的全部处理过程。在实际的编译生成可执行程序中实际上包含了两个步骤——编译和链接。编译的作用是执行预编译操作、检查源程序中的错误以及生成中间目标码，在 Windows 下也就是 .obj 文件，UNIX 下是 .o 文件，即 Object File；链接的作用是将编译过程产生的中间目标码（Object File）组合生成最终的二进制可执行文件。这主要是考虑到有多个源文件的情况下可以单独的编译每一个源文件（或者可以称作模块），这样就为大工程的源代码编译管理带来了极大的方便，关于工程管理的知识我将在工程管理基础部分讲解。

编译时，编译器需要的是语法的正确、函数与变量的声明的正确、编译器头文件的所在位置的正确，只要这些内容都正确，编译器就可以编译出中间目标文件。一般来说，每个源文件（.c/.cpp）都应该对应于一个中间目标文件（O 文件或是 OBJ 文件）。

链接时，主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件（O 文件或是 OBJ 文件）来链接我们的应用程序。链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File）。链接完成后就生成了可以运行的二进制可执行文件。

3.5 控制可执行程序的生成

在完成了编译和链接生成可执行程序之后，我们或许要问怎么样控制这个可执行程序的生成呢？是编译器全部都安排好了吗？如果能够控制可执行程序的生成，那么采用什么方式控制呢？

这一切的控制都是由链接器完成的。任何一个链接器都会有相应的方式控制程序的入口点（Entry）、程序基址（也就是程序在内存中的第一个位置）等内容。在 VC 等 IDE 开发环境中，已经通过“工程属性”的链接器设置选项来控制这些内容。典型的嵌入式系统中 ARM 编译器是通过一个叫做 Scatter-Loading 的.scl 描述文件来控制整个二进制可执行程序的存储器安排。在我们前面的程序世界中我们都是使用 main()函数做为 C 语言的程序入口，但实际上我们可以通过编译器的选项来控制这个入口函数。比如，我们可以在 VC 工程中的工程属性的编译器选项中指定入口点是 mymain()函数，而不是 main()函数。在 ARM 链接器中通过-first 来指定入口函数的值。

为了能够让我们更加深入的理解二进制可执行文件，在这里介绍一下 ARM 编译器使用的 Scatter-Loading 机制。首先我们先来看一个真实的 Scatter-Loading 描述文件的例子，这个例子的全部文件在 Test4 文件夹的 system.scl 文件内：

```
CODE_ROM 0 0xFFFFFF
```

```
{
    # Boot Block 区域
    BB_ROM +0x0
    {
        bootblock.o (BOOTBLOCK_IVT, +FIRST)
        bootblock.o (BOOTBLOCK_CODE)
        bootblock.o (BOOTBLOCK_DATA)
    }

    # 主程序代码和常量
    MAIN_APP +0x0
    {
        * (+RO)
    }

    # RAM 数据
    APP_RAM 0x10000000
    {
        * (+RW, +ZI)
    }

    # 用户自定义的保存区域
    RESERVED_RAM 0x10700000
    {
        mainapp.o (reserved)
    }
}
```

在这个文件中，CODE_ROM 0 0xFFFFFF 指定了这个最终生成的二进制文件地址空间空间在 0-0xFFFFFF 内，也就是 16M 字节空间。在这个内部，依次规划了 Boot Block 区域的代码和常量数据段（RO 数据）、主应用程序的代码和常量数据段（RO 数据）、应用程序 RAM 数据（RW 和 ZI 数据）以及用户特殊用途的保留区域的数据。

BB_ROM +0x0 中的+0x0 表示紧接着上面分配的空间，这里就是 0。MAIN_APP +0x0 就表示了接着 BB_ROM 之后的空间进行分配。APP_RAM 0x10000000 表示 RAM 的地址空间从 0x10000000 地址开始，依次类推，这个内存影射的示意图如下：

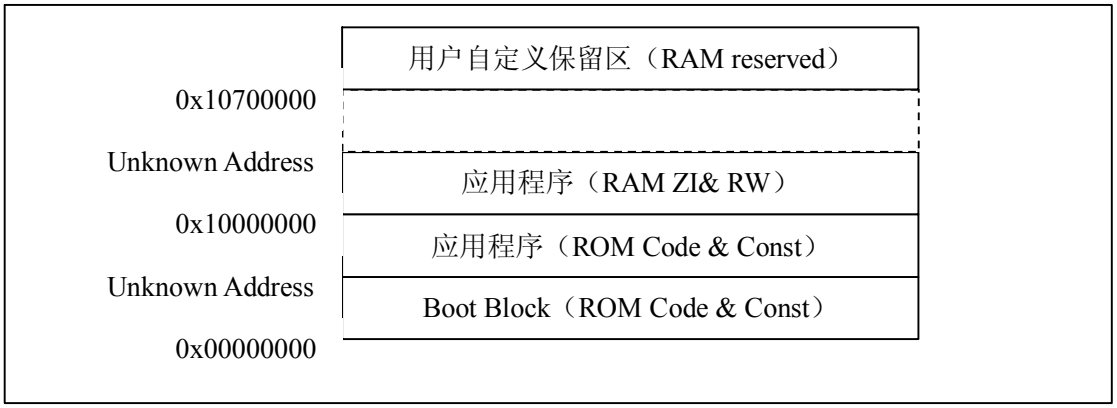

图 3.2 存储器映射图

用户保留区域是由用户根据需要通过#pragma arm section 指定的数据区域类型，可以参看 Test4 中的 mainapp.c 的定义。

bootblock.o (BOOTBLOCK_IVT, +FIRST)是非常关键的一句，它描述了将 bootblock.o 中的 BOOTBLOCK_IVT 区域的数据做为最前端数据，+FIRST 表示放在这个地址段的最前面，在这里也就是 ARM CPU 的中断向量表区域。

bootblock.o (BOOTBLOCK_CODE)和 bootblock.o (BOOTBLOCK_DATA)中说明的是使用 BOOTBLOCK_CODE 和 BOOTBLOCK_DATA 说明的区域，可以参看 bootblock.s 中的相关内容。

* (+RO)表示将所有尚未匹配的只读数据放在这个区域，* (+RW, +ZI)表示将尚未匹配的 RW 和 ZI 数据放在这个空间中。

mainapp.o (reserved)表示将 mainapp.o 中的使用“reserved”包含的数据放在这个空间中。

整个 Test4 工程展示了一个完整的 ARM 系统的构建要素，请认真地研究这个例子，它一定能够让您更加深刻的认识一个嵌入式系统。并且也有助于我们理解其他的系统，毕竟所有的系统原理都是相通的。在工程管理（Make File）一节中还将详细的展示怎样使用这个.scl 文件。

3.6 可执行程序的启动过程

对于一直生活在 Windows 世界的人们来说，最熟悉的可执行程序莫过于.exe 文件了。每当打开这些文件的时候，都会启动一个应用程序。对于写程序的人来说，还知道程序中的 main()函数或者由链接器指定的入口函数是首先被执行的。这种说法是正确的，只不过忽略了执行入口函数前的处理。最简单的，具有初始化值的全局变量是优先于 main 函数被赋值的。为了说明这个问题我们还是来看看 Test4 中的原文件吧。这个例子里面购建了一个相对完整的从 ARM 启动到执行 main 函数的过程，相信会对您理解其他的程序也会有所帮助。如果您有 make 运行环境和 ADS1.2 环境的话，这些代码是可以编译链接的。

与前面简短的例子不同，这个例子中包含了三个原文件和一个头文件，总共约 300 行程序。我相信，对于一个程序员来说，300 行程序也不算什么。在这里我们占用了 8 页多的篇幅，相信这里是本书中最长的一段代码了，不过我觉得有必要讲解这段程序，好让我们看一看以前从未重点关心过的部分。

system.h – 系统头文件

```
#ifndef SYSTEM_H
#define SYSTEM_H
/*=============================================================================
                                ARM 处理器常量定义
文件名：system.h
描述：
    这个文件内定义了 ARM 处理器的常量和栈尺寸，以及一个汇编的调用函数的宏
=============================================================================*/
#ifndef _ARM_ASM_
typedef   unsigned char       byte;          /* Unsigned 8   bit value type. */
typedef   unsigned short      word;          /* Unsinged 16 bit value type. */
typedef   unsigned long       dword;         /* Unsigned 32 bit value type. */
```

```
#endif //_ARM_ASM_

/*    CPSR Control Masks                */
#define PSR_Fiq_Mask                0x40
#define PSR_Irq_Mask                0x80

/*    Processor mode definitions */
#define PSR_Supervisor              0x13
#define PSR_System                  0x1f

/*    Stack sizes.                      */
#define SVC_Stack_Size              0xd0
#define Sys_Stack_Size              0x400

/*    用户数据区大小                    */
#define USER_HEAP_SIZE              0x1000

#if defined(_ARM_ASM_)
/*=============================================================================
   名称: blatox
   描述:
        不必理会是在 ARM 或 THUMB 模式下调用函数的方法
   Arguments:
      destreg -  包含被叫函数地址的寄存器
   被修改的寄存器: lr
=============================================================================*/
            MACRO
            blatox      $destreg
            ROUT

            tst         $destreg, #0x01             /* Test for thumb mode call.    */

            ldrne       lr, =%1
            ldreq       lr, =%2
            bx          $destreg
1
            CODE16
            bx          pc
            ALIGN
            CODE32
2
            MEND
#endif // _ARM_ASM_
#endif //SYSTEM_H
```

bootblock.s – 系统入口文件

```
;*====*====*====*====*====*====*====*====*====*====*====*====*====*====*====*
;                          System Boot Block
;   文件名:        bootblock.s
;   描述:
;   此模块中定义了系统的中断向量表和 Reset 时的处理函数
;*====*====*====*====*====*====*====*====*====*====*====*====*====*====*====*
;============================================================================
;                                MODULE INCLUDE FILES
;============================================================================
#include "system.h"

;============================================================================
;                                MODULE IMPORTS
;============================================================================
            IMPORT    ram_init
            IMPORT    __rt_entry

            ; 导入系统栈的地址
            IMPORT    svc_stack
            IMPORT    sys_stack

            ; 导入由链接器根据 Scatter-Loading 描述文件生成的数据区域 RAM 地址
            ; 和数据区大小

            ; 应用程序 RAM
            IMPORT |Load$$APP_RAM$$Base|
            IMPORT |Image$$APP_RAM$$Base|
            IMPORT |Image$$APP_RAM$$Length|
            IMPORT |Image$$APP_RAM$$ZI$$Base|
            IMPORT |Image$$APP_RAM$$ZI$$Length|

            ; 用户保留区 RAM
            IMPORT |Load$$RESERVED_RAM$$Base|
            IMPORT |Image$$RESERVED_RAM$$Base|
            IMPORT |Image$$RESERVED_RAM$$Length|
            IMPORT |Image$$RESERVED_RAM$$ZI$$Base|
            IMPORT |Image$$RESERVED_RAM$$ZI$$Length|

;============================================================================
;                                MODULE EXPORTS
;============================================================================
            ; 导出重新命名过的链接器符号，以便于其他模块使用
```

```
        ; 应用程序 RAM
        EXPORT Load__APP_RAM__Base
        EXPORT Image__APP_RAM__Base
        EXPORT Image__APP_RAM__Length
        EXPORT Image__APP_RAM__ZI__Base
        EXPORT Image__APP_RAM__ZI__Length

        ; 用户保留区 RAM
        EXPORT Load__RESERVED_RAM__Base
        EXPORT Image__RESERVED_RAM__Base
        EXPORT Image__RESERVED_RAM__Length
        EXPORT Image__RESERVED_RAM__ZI__Base
        EXPORT Image__RESERVED_RAM__ZI__Length

        ; 输出__main 和 _main 符号避免链接器包含标准 C 运行库的初始化处理

        EXPORT __main
        EXPORT _main
;====================================================================
;                          处理器中断向量表
;  ARM 处理器的中断向量表开始于 0x00000000H 地址，这里也是系统重置时的入口点
;
;  除了系统重置以外，其余所有的中断都应该引入到异常处理程序中，但是这里没
;  有这样处理，仅提供了简单的示例，并统一调用 boot_reset_handler 进入重置程
;  序
;====================================================================
        AREA    BOOTBLOCK_IVT, CODE, READONLY
        CODE32                          ; 32 bit ARM instruction set.
__main
_main
        ENTRY                           ; Entry point for boot image.

        ;ARM CPU 中断向量表
        b       boot_reset_handler      ; ARM reset
        b       boot_reset_handler      ; ARM undefined instruction interrupt
        b       boot_reset_handler      ; ARM software interrupt
        b       boot_reset_handler      ; ARM prefetch abort interrupt
        b       boot_reset_handler      ; ARM data abort interrupt
        b       boot_reset_handler      ; Reserved by ARM Ltd.
        b       boot_reset_handler      ; ARM IRQ interrupt
        b       boot_reset_handler      ; ARM FIQ interrupt

;====================================================================
```

```
;   函数名：boot_reset_handler
;   描述：
;   系统初始化函数，进行如下处理：
;       1. 初始化系统栈
;       2. 初始化 RAM
;       3. 调用__rt_entry 初始化 C 语言库，并调用 C 语言的 main 函数
;===============================================================================
            AREA BOOTBLOCK_CODE, CODE
            CODE32                          ; 32 bit ARM instruction set

boot_reset_handler
            ; 进入超级用户模式并安装超级用户模式下的栈
            msr         CPSR_c, #PSR_Supervisor:OR:PSR_Irq_Mask:OR:PSR_Fiq_Mask
            ldr         r13, =svc_stack+SVC_Stack_Size

            msr         CPSR_c, #PSR_System:OR:PSR_Fiq_Mask:OR:PSR_Irq_Mask
            ldr         r13, =sys_stack+Sys_Stack_Size

            // 返回超级用户模式
            msr         CPSR_c, #PSR_Supervisor:OR:PSR_Irq_Mask:OR:PSR_Fiq_Mask

            ; 初始化 RAM
            ldr         r4, =ram_init
            blatox      r4

            ldr         a3, =__rt_entry
            bx           a3

            AREA        BOOTBLOCK_DATA, DATA, READONLY

;===============================================================================
;                                   BOOT BLOCK 数据地址
;   根据导入的链接器 RAM 数据区地址，重新生成新的符号给 Boot RAM 初始化程序使用。
;   由于 RAM 初始化程序使用 C 语言完成，而 C 编译器需要-pcc 参数才能使用$符号，因此
;   这里做一个符号的变换使得 C 语言可以直接使用
;
;===============================================================================
; 应用程序 RAM
Load__APP_RAM__Base
            DCD |Load$$APP_RAM$$Base|

Image__APP_RAM__Base
            DCD |Image$$APP_RAM$$Base|
```

```
Image__APP_RAM__Length
        DCD |Image$$APP_RAM$$Length|

Image__APP_RAM__ZI__Base
        DCD |Image$$APP_RAM$$ZI$$Base|

Image__APP_RAM__ZI__Length
        DCD |Image$$APP_RAM$$ZI$$Length|

; 用户保留区 RAM
Load__RESERVED_RAM__Base
        DCD |Load$$RESERVED_RAM$$Base|

Image__RESERVED_RAM__Base
        DCD |Image$$RESERVED_RAM$$Base|

Image__RESERVED_RAM__Length
        DCD |Image$$RESERVED_RAM$$Length|

Image__RESERVED_RAM__ZI__Base
        DCD |Image$$RESERVED_RAM$$ZI$$Base|

Image__RESERVED_RAM__ZI__Length
        DCD |Image$$RESERVED_RAM$$ZI$$Length|

        END
```

raminit.c – RAM 初始化和 C 运行库替换函数

```
/*====*====*====*====*====*====*====*====*====*====*====*====*====*====*====*
                                RAM 初始化文件

文件名：raminit.c
描述：
    此模块内包含了初始化 RAM 的函数以及链接时需要替换的 C 运行时库函数

*====*====*====*====*====*====*====*====*====*====*====*====*====*====*====*/
/*============================================================================
                                Include Files
============================================================================*/
#include "system.h"

/*============================================================================
                                Global Data
============================================================================*/
```

```
/*   由于在 RAM 初始化之前还不能够使用 RAM，
     因此这里使用 CPU 寄存器做为变量的存储空间*/
__global_reg(1) dword *dst32;
__global_reg(2) dword *src32;
__global_reg(3) dword *stop_point;

/* 应用程序 RAM */
extern byte * Load__APP_RAM__Base;
extern byte * Image__APP_RAM__Base;
extern byte * Image__APP_RAM__Length;
extern byte * Image__APP_RAM__ZI__Base;
extern byte * Image__APP_RAM__ZI__Length;

/* 用户保留区 RAM */
extern byte * Load__RESERVED_RAM__Base;
extern byte * Image__RESERVED_RAM__Base;
extern byte * Image__RESERVED_RAM__Length;
extern byte * Image__RESERVED_RAM__ZI__Base;
extern byte * Image__RESERVED_RAM__ZI__Length;

/*============================================================================
                                Function Declare
============================================================================*/
/*CRT 库函数替换*/
void __user_initial_stackheap ( ) { return; }

/*============================================================================
函数名：ram_init

描述：
  初始化 RAM
依赖文件
  None
返回值
  None
============================================================================*/
void ram_init(void)
{
  /* 复制应用程序区的已初始化数据 */
  stop_point = (dword *) ( (dword) Image__APP_RAM__Base +
                           (dword) Image__APP_RAM__Length);
  for( src32 = (dword *) Load__APP_RAM__Base,
       dst32 = (dword *) Image__APP_RAM__Base;
```

```
            dst32 < stop_point;
            src32++, dst32++ )
        {
          *dst32 = *src32;
        }

    /* 初始化应用程序区的零初始化数据 */
    stop_point = (dword *) ( (dword) Image__APP_RAM__ZI__Base +
                                     (dword) Image__APP_RAM__ZI__Length);
    for( dst32=(dword *) Image__APP_RAM__ZI__Base;
            dst32 < stop_point;
            dst32++ )
        {
          *dst32 = 0;
        }

    /* 复制用户保留区的已初始化数据 */
    stop_point = (dword *) ( (dword) Image__RESERVED_RAM__Base +
                                     (dword) Image__RESERVED_RAM__Length);
    for( src32 = (dword *) Load__RESERVED_RAM__Base,
              dst32 = (dword *) Image__RESERVED_RAM__Base;
            dst32 < stop_point;
            src32++, dst32++ )
        {
          *dst32 = *src32;
        }

    /* 初始化用户保留区的零初始化数据 */
    stop_point = (dword *) ( (dword) Image__RESERVED_RAM__ZI__Base +
                                     (dword) Image__RESERVED_RAM__ZI__Length);
    for( dst32=(dword *) Image__RESERVED_RAM__ZI__Base;
            dst32 < stop_point;
            dst32++ )
        {
          *dst32 = 0;
        }
}
```

mainapp.c – 程序主函数

```
/*====*====*====*====*====*====*====*====*====*====*====*====*====*====*
                                主应用程序文件

文件名：mainapp.c
描述：
```

```
    此模块内包含了 Main 主函数

*====*====*====*====*====*====*====*====*====*====*====*====*====*====*====*/
/*============================================================================
                                Include Files
============================================================================*/
#include "system.h"

/*============================================================================
                                Global Data
============================================================================*/
#pragma arm section zidata = "reserved"
static byte gUserHeap[USER_HEAP_SIZE];
#pragma arm section zidata

/*----------------------------------------------------------------------------
  超级用户模式下的堆栈值
----------------------------------------------------------------------------*/
byte svc_stack[SVC_Stack_Size];
byte sys_stack[Sys_Stack_Size];

/*============================================================================
                                Function Declare
============================================================================*/
int mymain( void );

/*============================================================================
函数名：main
描述：
  程序运行的入口点
依赖文件
  None
返回值
  None, this routine does not return
============================================================================*/
int main( void )
{
    return mymain();
}

/*============================================================================
函数名：main
描述：
  程序运行的入口点
```

```
依赖文件
   None
返回值
   None, this routine does not return
==========================================================================*/
int mymain( void )
{
    int i;

    // 初始化用户 Heap 区域
    for(i=0; i<USER_HEAP_SIZE; i++)
    {
        gUserHeap[i] = 0xFF;
    }

    for(;;){}// 一直在此处循环执行
}
```

在 system.h 文件中定义了一些常量，和一个汇编宏 blatox，这个宏的用途是无缝的在汇编和 C 语言中进行调用。之所以有这个宏是因为 C 语言可以使用 ARM 的 ARM 指令集或 THUMB 精简指令集，前者是 32 位的指令长度，后者是 16 位指令长度。16 位指令集比 32 位指令集更加节省二进制代码空间。由于在系统启动的时候 CPU 使用的全部是 ARM 指令，因此，如果当前 C 语言编译的代码是 THUMB 指令的话就需要不同的处理方式，函数 blatox 就可以通过判断寄存器的值来定位即将调用的代码是 THUMB 还是 ARM 的。

bootblock.s 是一个汇编语言文件，我们从中可以看见它使用了#include 来包含 system.h 文件。如果我们直接使用汇编编译器 armasm 是不能够识别的，因此要首先使用 armcc 的 C 语言编译器处理一下。指令如下：

armcc -ansic -E -cpu ARM7TDMI -apcs /noswst/interwork -D_ARM_ASM_ bootblock.s > bootblock..i

生成处理之后的 bootblock.i 纯汇编文件然后才能使用 armasm 编译。如果想了解其中每个参数的意义，可以查看 ARM 编译器的帮助文档。这里仅说明-D 的作用与#define 相同。

下面的代码是 ARM CPU 的中断向量表，根据上一节的 Scatter-Loading 描述文件可以知道，这个表的起始地址是 0x00000000，也就是 CPU 一上电就会在此处取得指令执行。0x00000000 地址是一个跳转指令，转到 boot_reset_handler 代码段执行。

```
        AREA    BOOTBLOCK_IVT, CODE, READONLY
        CODE32                          ; 32 bit ARM instruction set.
__main
_main
        ENTRY                           ; Entry point for boot image.

        ;ARM CPU 中断向量表
        b       boot_reset_handler      ; ARM reset
        b       boot_reset_handler      ; ARM undefined instruction interrupt
        b       boot_reset_handler      ; ARM software interrupt
        b       boot_reset_handler      ; ARM prefetch abort interrupt
```

```
        b       boot_reset_handler      ; ARM data abort interrupt
        b       boot_reset_handler      ; Reserved by ARM Ltd.
        b       boot_reset_handler      ; ARM IRQ interrupt
        b       boot_reset_handler      ; ARM FIQ interrupt
```

这段代码中的__main 和_main 是为了替换掉链接时 C 运行库的相关内容的，各位读者可以试着在 Test4 种去掉这段代码，链接时将看到“Image does not have an entry point”的警告信息。

紧接着程序跳转到 boot_reset_handler 代码段：

```
        AREA BOOTBLOCK_CODE, CODE
        CODE32                          ; 32 bit ARM instruction set

boot_reset_handler
        ; 进入超级用户模式并安装超级用户模式下的栈
        msr     CPSR_c, #PSR_Supervisor:OR:PSR_Irq_Mask:OR:PSR_Fiq_Mask
        ldr     r13, =svc_stack+SVC_Stack_Size

        msr     CPSR_c, #PSR_System:OR:PSR_Fiq_Mask:OR:PSR_Irq_Mask
        ldr     r13, =sys_stack+Sys_Stack_Size

        // 返回超级用户模式
        msr     CPSR_c, #PSR_Supervisor:OR:PSR_Irq_Mask:OR:PSR_Fiq_Mask

        ; 初始化 RAM
        ldr     r4, =ram_init
        blatox  r4

        ldr     a3, =__rt_entry
        bx      a3
```

在这段代码中首先设置了系统的堆栈空间，然后是调用了 ram_init 函数执行 RAM 的初始化。再之后就调用 C 语言运行库的入口__rt_entry 函数，从这里就正式进入了 C 的世界，当然在这里如果我们不想使用任何 C 语言库函数的话，我们可以指定一个这个值是 mymain 并且屏蔽掉 mainapp.c 中的 main()函数。具体替换方法如下：

1、注释掉 mainapp.c 中的 main()函数

2、注释掉 bootblock.s 文件 24 行的 IMPORT __rt_entry 语句，替换成 IMPORT mymain 语句

3、将 bootblock.s 文件 129 行的__rt_entry 替换成 mymain，并使用 blatox a3 替换掉 bx a3 语句

这个时候编译链接的语句就是没有任何 C 语言库函数影响的纯粹 C 语言版本的程序了。产生这种想法的动机是，在一个大型系统中希望能够自己实现相关的库函数或禁用编译器的库函数。例如，printf 是输出一个语句，但是不同平台的输出方式是不同的，在 DOS 操作系统中是输出在显示器上，而在手机上可能显示在 LCD 屏幕上，或者进一步说，有些系统根本就没有输出。

接下来看看 ram_init 中都作了些什么：

```
void ram_init(void)
{
  /* 复制应用程序区的已初始化数据 */
  stop_point = (dword *) ( (dword) Image__APP_RAM__Base +
                           (dword) Image__APP_RAM__Length);
  for( src32 = (dword *) Load__APP_RAM__Base,
       dst32 = (dword *) Image__APP_RAM__Base;
       dst32 < stop_point;
       src32++, dst32++ )
    {
      *dst32 = *src32;
    }

  /* 初始化应用程序区的零初始化数据 */
  stop_point = (dword *) ( (dword) Image__APP_RAM__ZI__Base +
                           (dword) Image__APP_RAM__ZI__Length);
  for( dst32=(dword *) Image__APP_RAM__ZI__Base;
       dst32 < stop_point;
       dst32++ )
    {
      *dst32 = 0;
    }

  /* 复制用户保留区的已初始化数据 */
  stop_point = (dword *) ( (dword) Image__RESERVED_RAM__Base +
                           (dword) Image__RESERVED_RAM__Length);
  for( src32 = (dword *) Load__RESERVED_RAM__Base,
       dst32 = (dword *) Image__RESERVED_RAM__Base;
       dst32 < stop_point;
       src32++, dst32++ )
    {
      *dst32 = *src32;
    }

  /* 初始化用户保留区的零初始化数据 */
  stop_point = (dword *) ( (dword) Image__RESERVED_RAM__ZI__Base +
                           (dword) Image__RESERVED_RAM__ZI__Length);
  for( dst32=(dword *) Image__RESERVED_RAM__ZI__Base;
       dst32 < stop_point;
       dst32++ )
    {
      *dst32 = 0;
    }
}
```

根据我们上面的 Scatter-Loading 描述文件，有两段内存区域，一段是 APP_RAM 对应的链接器会给初始化程序提供如下几个变量：

Load$$APP_RAM$$Base	初始化数据存储的地址
Image$$APP_RAM$$Base	初始化数据 RAM 的基地址
Image$$APP_RAM$$Length	初始化数据的长度
Image$$APP_RAM$$ZI$$Base	零初始化数据 RAM 的基地址
Image$$APP_RAM$$ZI$$Length	零初始化数据的长度

再看这段程序初始化 APP_RAM 的过程，现在您应该可以看见 Scatter-Loading 描述文件与程序之间的关系了吧。这里要说明的是，由于在 ARM C 编译器中不能使用“$”符号，因此对于这些根据 Scatter-Loading 描述文件生成的链接符号我们在 bootblock.s 文件中进行了重新定义，使用“_”符号代替了“$”符号。

在这里所初始化的内容是程序运行时的空间，这些空间里的数据在关机之后是没有任何信息保存下来的，不管是 ZI 还是 RW 的。也就是说，在每一次开机的过程中，所执行的操作是完全相同的。或许有人会问，如果我需要记录一些变量的内容，让它能够在断电后仍旧能够保存下来，该怎么做呢？非常不幸的告诉您，这里不是实现这个功能的地方。这个功能需要文件系统或者其他非易失性（Non-Volatile NV）设备的支持，这通常就是一个系统存储参数的地方，要区分系统中不同设备的不同作用。对于 RESERVED_RAM 的处理就和 APP_RAM 一样了，就不再赘述了。这里就是一个完整的系统启动过程，进入了 main 函数之后，如何处理就是我们每个程序员自己的任务了。

在了解了 ARM 的启动过程之后，您应该也能够猜出在 Windows 下一个.exe 可执行文件的执行过程了，虽然细节不同，但是思想是一样的，我在这里就不再赘述了。

3.7 函数的调用和返回

在软件基础一节讲解函数的时候，我们知道了每个函数在运行的时候会给参数们留“位子”，以便于在使用函数的时候可以通过这些“位子”让函数输出不同的处理结果。但是我们还可能对这些“位子”的组织方式和返回存在着疑问。这一节我将结合 Windows 下的 VC 编译器和 ARM 编译器来进行讲解。当然不了解这些我们依然可以写出程序，但是，既然我们要追根溯源，那么还是了解一下吧。同时掌握这些东西还有利于优化程序。下表列出了几种典型的函数调用约定：

调用方式	参数传递方式
VC cdecl	参数经堆栈进行传递，由调用者负责清除堆栈
VC stdcall	参数经堆栈进行传递，由函数本身负责清除堆栈
VC fastcall	2 个 dword 或更小的参数通过寄存器传递，其余经由堆栈传递。
ARM Call	4 个 dword 或更小的参数通过寄存器传递，其余经由堆栈传递。

从上面的约定可以看出，不同的编译器函数的调用约定细节是不一样的，这也决定了不同调用约定的函数行为的不同。

cdecl 调用约定由于由调用者负责堆栈的清除，因此可以实现可变参数的函数（如 printf 函数）。但是这样也会增加程序的代码空间，因为每一个调用函数的地方都需要相关的堆栈处理代码。

stdcall 调用约定是 Windows 操作系统 API 函数的默认调用方式，没有进行什么特别的处理。

fastcall 调用约定相当于 stdcall 约定的优化版本，它通过两个寄存器来实现参数的传递，

这样就减少了堆栈的处理，因此可以加快函数的运行。同理可以推理出，如果 fastcall 调用约定的函数参数个数小于或等于 2 个的话，会取得最快的执行速度。

ARM 编译器的调用约定是使用 4 个寄存器来传递参数，因此对于 ARM 平台的程序来说，在小于等于 4 个参数的情况下可以得到最优化的函数调用效率。对于空间较大的参数（如一个大的结构体）ARM 编译器将通过堆栈进行传递。

对于函数的返回值，如果需要的传递空间比较小则通过寄存器传递，如果需要的空间较大，则通过内存进行间接的传递。上述几种方式的返回值传递都是一样的。

3.8 开发中与运行时

我一直希望能够将您从 C 语言的世界中引入程序运行的二进制世界之中，并一直试图将二者的内在联系展示给您。虽然我并不能准确地知道是否达到了目的，但是我还是想在这里总结一下它们之间的关系。

C 语言与二进制分别对应了开发中与运行时两个程序的过程概念。对于很多的初学者会问一些诸如 CPU 怎么识别我定义的变量这类问题，回答这类问题是很棘手的。如果我告诉他 CPU 不认识我们定义的变量时，他一定会问 CPU 不认识那么程序怎么运行啊？这就像是一个在迷宫里的人，迷失在开发中与运行时两个世界组成的迷宫里。

变量是开发中的概念，经过编译器的处理，会将变量的外衣去掉，送进赤裸裸二进制数据的二进制世界中。编译器就是这两个世界的桥梁。在二进制世界里，CPU 只认识二进制的数据和指令，没有数据类型的概念。例如，对于开发过程中的一个数组，到了二进制层面就变成了一个固定大小的二进制空间，不管这个数据类型是 char 的还是 int 的，CPU 只是根据相应的操作指令来执行对这块空间的操作。因此，对于在 C 语言中的诸如变量、函数、结构体以及数组等等，统统是开发过程中的概念，只有编译器会接受这些概念。到了运行的时候 CPU 一概不认，CPU 就是一个非常简单的家伙，只知道根据编译器生成的二进制指令执行，“只要符合规范，我就一言不发”。

只有透彻的理解了开发中与运行时之间的关系，才能从深层次理解程序的运行。因此在某种程度上来说，理解一下编译器本身的“内幕”是很有必要的。只有二进制世界，才是程序真正的世界。

3.9 小结

这一章中主要介绍了编译器的分类以及编译器在整个系统中的作用，同时也包含了一些程序运行的知识，在这里着重说明了开发中和运行时两个程序的阶段，而编译器则成为了这两个阶段的桥梁。通过编译器对开发中的代码进行编译链接，才生成了可以运行的代码。编译器的作用就是将源代码转换成可以在目标机器上运行的机器码，这样可以通过高级语言的开发大大提高机器码的开发速度，可以说编译器是一个提高开发效率的工具。

思考题

编译器将程序分成几种类型的数据？不同类型之间的区别是什么？

第四章 工程管理（Make File）基础

什么是 Make File？很多 Windows 的程序员都不知道这个东西，因为那些 Windows 的 IDE 都为我们做了这个工作，但我觉得要成为一名专业的程序员 Make File 还是要懂。这就好像是我现在懂了 C 语言，但是我还要去了解编译器的“内幕”一样。Make File 关系到了整个工程的编译规则。对于一个大型的工程来说，其中的源文件不计其数，并分别按类型、功能、模块分别放在若干个目录中，这就需要我们能够有一套方便好用的工具来管理这些源文件的编译和链接。因为，如果每次都将全部的文件编译一遍，可能会大大的浪费开发时间，所以能够识别哪些文件需要重新编译的功能就很有必要了。幸好我们有 Make File 来帮助我们做这些，不然真是麻烦大了。

Make File 定义了一系列的规则来指定哪些文件需要重新编译，甚至于进行更复杂的功能操作。Make File 带来的好处就是——“自动化编译”，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 Make File 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 make，Visual C++的 nmake，Linux 下的 GNU make 等。可以看出，Make File 已经是大多数工程管理的标准工具了。

4.1 Make File 核心原理

要掌握一个东西，就应该首先掌握它的核心思想，因此在这里我们将首先来看看 Make File 的核心原理——依赖规则。看下面：

target ... : prerequisites ...
    command
    ...

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”中会有叙述。

prerequisites 就是，要生成那个 target 所需要的文件或是目标。或者理解为，生成 target 需要 prerequisites。

command 也就是 make 需要执行的命令。

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件更新的话，command 所定义的命令就会被执行。这就是 Make File 的规则，也就是 Make File 中最为核心的原理。

完成一个简单的演示例子，在这个例子中有 main.c 和 command.c 两个源文件，一个 defs.h 头文件，main.c 和 command.c 中都包含 defs.h 头文件。要求能够输出 main.o 和 command.o 两个目标文件。定义的 Make 规则如下：

```
edit : main.o command.o
    cc –o edit main.o command.o

main.o : main.c defs.h
    cc –c main.c -o main.o
```

```
command.o : command.c defs.h
    cc –c command.c –o command.o

clean :
    rm main.o \
        command.o
```

反斜杠（\）是换行的意思，告诉 Make 程序下面一行于这一行紧接着。clean 是用来清除生成的 main.o 和 command.o 文件。请注意，在上面的每一行缩进中，Make 要求必须使用 Tab 键，不能使用空格做为缩进量。

在这个例子中，目标文件（target）包含文件 edit 和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是目标文件 edit 的依赖文件。依赖关系本质上就是说明了目标文件是由哪些文件生成的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键做为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的标签（Label）一样。clean 的冒号后什么也没有，因此 make 不会自动去查找 clean 的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个标签（Label）的名字。这样的方法非常有用，我们可以在一个 Make File 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

4.2 Make 的工作方式

在默认的方式下，我们只输入 make 命令，或者使用-f 参数指定需要执行的 Make File 文件，然后 make 程序会进行如下操作：

1、make 会在当前目录下找名字叫“Makefile”或“makefile”的文件（注意没有扩展名）或通过-f 指定的 Make File 文件。

2、如果找到 Make File 文件，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件做为最终的目标文件。

3、如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。

4、如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。（这有点像一个堆栈的过程）

5、最后当然是 C 文件和 H 文件都存在的啦，于是 make 会生成 .o 文件，然后再用.o 文件生成 make 的终极任务——目标文件 edit 了。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。如果在找寻的过程中出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出并报错；而对于所定义命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要求 make 执行。即命令—

—“make clean”，以此来清除所有的目标文件，以便于重新编译。于是在上面的例子中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 main.c，那么根据我们的依赖性，我们的目标 main.o 会被重编译（也就是在这个依性关系后面所定义的命令），于是 main.o 的文件也是最新的啦，于是 main.o 的文件修改时间要比 edit 要新，所以 edit 也会被重新生成了（详见 edit 目标文件后定义的命令）。而如果我们改变了“defs.h”，那么，main.o 和 command.o 都会被重新编译，那么理所当然的，edit 也会被重生成。

为了更加清晰的理清 Make File 的工作方式，现在我们总结一下 Make File 的各个工作阶段及其执行的内容：

1、读入所有的 Makefile
2、读入被 include 的其它 Makefile
3、初始化文件中的变量
4、分析所有规则
5、为所有的目标文件创建依赖关系链
6、根据依赖关系，决定哪些目标要重新生成
7、执行生成命令

1-5 步为第一个阶段，6-7 为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make 会把其展开在使用的位置。但 make 并不会完全马上展开，make 使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

4.3 Make File 实例

或许您会觉得上面的例子也太简单了，根本不值一提。那么现在就来一个高阶的应用来看看。还记得我们在编译器基础一节所作的 Test4 例子吗，我们就在这个例子中添加一个 makefile 文件（注意没有扩展名）。

makefile – 一个简单却齐全的 Make File 实例

```
SHELL = sh
#============================================================================
#   Name:
#       makefile
#
#   Description:
#       Makefile to build the $(TARGET) module.
#
#     The following nmake targets are available in this makefile:
#
#         all                 - make .elf and .mod image files (default)
#         clean               - delete object directory and image files
#         filename.o          - make object file
#         filename.mix        - make C and ASM mix file
#
#     The above targets can be made with the following command:
#
```

```
#         make    [target]
#
#    Assumptions:
#        1. The ARM ADS 1.0.1 or higher tools are installed
#        2. The run environment of make toos is exist in this OS
#
#-------------------------------------------------------------------------
#=========================================================================
TARGET              = Test4#
SCLFILE             = system.scl#
C_OBJS              = mainapp.o raminit.o
A_OBJS              = bootblock.o
OBJS                = $(C_OBJS) $(A_OBJS)

APP_PATH            = .#

# set the search path
vpath %.c $(APP_PATH)
vpath %.s $(APP_PATH)

#-------------------------------------------------------------------------
# Target file name and type definitions
#-------------------------------------------------------------------------

EXETYPE       = elf#                        # Target image file format
MODULE         = bin#                         # Binary module extension

#-------------------------------------------------------------------------
# Target compile time symbol definitions
#-------------------------------------------------------------------------

ARMASM         = -D_ARM_ASM_#

#-------------------------------------------------------------------------
# Software tool and environment definitions
#-------------------------------------------------------------------------
ARMBIN    = $(ARMHOME)/Bin#
ARMBIN := $(subst \,/,$(ARMBIN))#

ARMCC      = $(ARMBIN)/armcc#             # ARM ADS ARM 32-bit inst. set ANSI C compiler
ASM          = $(ARMBIN)/armasm#          # ARM ADS assembler
LD           = $(ARMBIN)/armlink#         # ARM ADS linker
HEXTOOL = $(ARMBIN)/fromelf#         # ARM ADS utility to create hex file from image
```

```
OBJ_CMD       = -o#                    # Command line option to specify output filename

#-------------------------------------------------------------------------------
# Processor architecture options
#-------------------------------------------------------------------------------

CPU = -cpu ARM7TDMI#                   # ARM7TDMI target processor

#-------------------------------------------------------------------------------
# ARM Procedure Call Standard (APCS) options
#-------------------------------------------------------------------------------
APCS = -apcs /noswst/interwork

#-------------------------------------------------------------------------------
# Additional compile time error checking options
#-------------------------------------------------------------------------------

CHK = -fa#                             # Check for data flow anomolies

#-------------------------------------------------------------------------------
# Compiler output options
#-------------------------------------------------------------------------------

OUT = -c#                              # Object file output only

#-------------------------------------------------------------------------------
# Compiler/assembler debug options
#-------------------------------------------------------------------------------

DBG = -g#                              # Enable debug

#-------------------------------------------------------------------------------
# Compiler optimization options
#-------------------------------------------------------------------------------

OPT = -Ospace -O2#                     # Full compiler optimization for space

#-------------------------------------------------------------------------------
# Compiler code generation options
#-------------------------------------------------------------------------------

END = -littleend#                      # Compile for little endian memory architecture
ZA    = -zo#                           # LDR may only access 32-bit aligned addresses
```

```
CODE = $(END) $(ZA)#

#----------------------------------------------------------------------------
# Include file search path options
#----------------------------------------------------------------------------

INC = -I.#

#----------------------------------------------------------------------------
# Linker options
#----------------------------------------------------------------------------

LINK_CMD = -o#                              # Command line option to specify output file
                                            # on linking
LIST     = -list $(TARGET).map#     # Direct map and info output to file
INFO     = -elf -map -info sizes,totals,veneers,unused
LINK_OPT = -scatter $(SCLFILE)#     # Use scatter load description file
#----------------------------------------------------------------------------
# HEXTOOL options
#----------------------------------------------------------------------------

BINFORMAT = -bin#
OUTPUT    = -output#

#----------------------------------------------------------------------------
# Compiler flag definitions
#----------------------------------------------------------------------------

CFLAGS   = $(OUT) $(INC) $(CPU) $(APCS) $(CODE) $(CHK) $(DBG) $(OPT)
AFLAGS   = $(CPU) $(APCS) $(INC)

#----------------------------------------------------------------------------
# Default target
#----------------------------------------------------------------------------
.PHONY : all
all :   startup $(TARGET).$(MODULE) complete

.PHONY : startup
startup:
    @echo ----------------------------------------------------------
    @echo Compile startup
    @echo ----------------------------------------------------------

.PHONY : complete
```

```
complete:
	@echo ----------------------------------------------------------
	@echo All have been done
	@echo ----------------------------------------------------------

#----------------------------------------------------------------------
# Clean target
#----------------------------------------------------------------------

# The object subdirectory, target image file, and target hex file are deleted.
.PHONY : clean
clean :
	@echo ----------------------------------------------------------
	@echo CLEAN
	-rm -f *.o
	-rm -f *.i
	-rm -f *.dep
	-rm -f $(TARGET).$(EXETYPE)
	-rm $(TARGET).$(MODULE)
	-rm $(TARGET).map
	@echo ----------------------------------------------------------

#======================================================================
#                                DEFAULT SUFFIX RULES
#======================================================================
SRC_FILE = $(@F:.o=.c)#              # Input source file specification
OBJ_FILE = $(OBJ_CMD) $(@F)#      # Output object file specification

.SUFFIXES :
.SUFFIXES : .o .c .s .mix .dep

#----------------------------------------------------------------------
# C code inference rules
#----------------------------------------------------------------------
%.o:%.c
	@echo ----------------------------------------------------------
	@echo OBJECT $(@F)
	$(ARMCC) $(CFLAGS) $(OBJ_FILE) $(SRC_FILE)
	@echo ----------------------------------------------------------

%.mix:%.c
	@echo ----------------------------------------------------------
	@echo OBJECT $(@F)
	$(ARMCC) -S -fs $(CFLAGS) $(INC) $(OBJ_FILE) $<
```

```
	@echo ----------------------------------------------------------

#----------------------------------------------------------------------------
# Assembly code inference rules
#----------------------------------------------------------------------------
%.o:%.s
	@echo ----------------------------------------------------------
	@echo OBJECT $(@F)
	$(ARMCC) -ansic -E $(AFLAGS) $(ARMASM) $< > $*.i
	$(ASM) $(AFLAGS) $(OBJ_FILE) $*.i
	@echo ----------------------------------------------------------

#----------------------------------------------------------------------------
# Depend file inference rules
#----------------------------------------------------------------------------
%.dep:%.c
	@echo ----------------------------------------------------------
	$(ARMCC) -ansic -M $< > $*.dep
	@echo ----------------------------------------------------------

#============================================================================
#                                MODULE SPECIFIC RULES
#============================================================================

#----------------------------------------------------------------------------
# Lib file targets
#----------------------------------------------------------------------------

$(TARGET).$(MODULE) : $(TARGET).$(EXETYPE)
	@echo ----------------------------------------------------------
	@echo TARGET $@
	$(HEXTOOL)  $(TARGET).$(EXETYPE)  $(BINFORMAT)  $(OUTPUT)
$(TARGET).$(MODULE)

$(TARGET).$(EXETYPE) : $(OBJS)
	@echo ----------------------------------------------------------
	@echo TARGET $@
	$(LD) $(INFO) $(LINK_OPT) $(LIST) $(LINK_CMD) $(TARGET).$(EXETYPE) $(OBJS)

# ------------------------------------------
# C file dependency list
# ------------------------------------------
ifeq ($(MAKECMDGOALS),)
-include $(C_OBJS:.o=.dep)
```

```
else
ifeq ($(filter all,$(MAKECMDGOALS)),all)
-include $(C_OBJS:.o=.dep)
endif
endif

# ----------------------------------------
# ASM file dependency list
# ----------------------------------------
bootblock.o : bootblock.s
bootblock.o : system.h
```

可不要小看这短短的 200 多行 Make File 啊，正所谓“麻雀虽小，五脏俱全”，它可是该有的都有了。我们可以对它进行扩展，生成一个非常自动化的 Make 系统来。接下来我将逐行解释这个文件中的内容。

最顶行的 SHELL = sh 是告诉 make 程序，使用的 SHELL 程序是 sh.exe。在本例中使用的是 cygwin 环境下的 shell 程序和命令的。cygwin 是用来在 windows 操作系统下模拟 unix 命令和操作方式的实用程序，通过 Internet 您可以很容易的找到这个程序以及 GNU Make 程序，不过 ARM 编译器就需要我们自己购买了。这是我很对不起各位读者的地方，虽然这个例子是在我的机器上测试通过的，但是各位读者朋友们要搭建这个环境就没那么容易了。这个环境中主要有三个要素：cygwin 应用、GNU Make 以及 ARM 编译器。

接下来是注释并说明这个文件的，在 Make File 中“#”号表示后面紧接的是注释，作用于从“#”号之后的一行，千万不要幻想使用“#”注释多行语句，在 Make File 中没有可以注释多行的语句。接下来就进入了正式的 Make File 执行语句了。

4.3.1 变量的定义

```
TARGET          = Test4#
SCLFILE         = system.scl#
C_OBJS          = mainapp.o raminit.o
A_OBJS          = bootblock.o
OBJS            = $(C_OBJS) $(A_OBJS)

APP_PATH        = .#
```

这里面定义了 5 个 Make File 变量，为了能够容易区分变量，通常在 Make File 中都使用全大写字符串做为变量名，当然这不是必须的，只是一个约定俗成的习惯而已。如果需要使用定义的变量则使用$()来饮用这个变量的值，如上面的 OBJS = $(C_OBJS) $(A_OBJS)一句中就使用了 C_OBJS 和 A_OBJS 两个变量。如果我们要使用真实的“$”字符，那么我们需要用“$$”来表示。如果引用的变量没有定义，不要害怕，Make 不会报错，Make 会认为这个变量是空的。这里也提醒各位读者注意，要认真的使用变量名，不要弄错了，因为 Make 不会报告未定义变量的任何错误。

Make File 中的变量定义是不分先后的，比如上面的 C_OBJS 和 A_OBJS 可以定义在 OBJS 变量之后，所得到的结果是一样的。这是一个十分具有迷惑性的方式，我不建议使用

这种方式，虽然它可能让我们觉得很“自由”。为了避免这种情况，可以使用“:=”号来为变量赋值，这种方式只能使用已经定义好的变量，如果变量在前面没有定义，则使用空值来代替。例如：

A = $(B)

B = debug

此时，A 的值是 debug。如果使用

A := $(B)

B = debug

则此时 A 的值是空，因为在 A 变量之前 B 还没有定义。与之对应的赋值操作符还有“?=”，它的作用是首先判断这个变量有没有在前面被定义过，如果没有定义则给这个变量赋值，否则使用已经定义的值。

在这里，OBJS 变量也可以采用如下的方式实现：

OBJS = $(C_OBJS)

OBJS += $(A_OBJS)

其中使用“+=”来进行变量的连接。

这六个变量的意义依次是：目标文件的名称、链接器使用的 Scatter-Loading 描述文件名、C 语言的.o 文件、汇编语言的.o 文件、全部.o 文件和.c 文件的路径。

4.3.2 搜索路径

```
# set the search path
vpath %.c $(APP_PATH)
vpath %.s $(APP_PATH)
```

vpath 是 Make 用来设置指定文件类型搜索路径的，上面两句分别设置了.c 和.s 的源文件搜索路径。在大型项目中，由于多个路径都存在源文件（.c），因此可以设置多个同扩展名的 vpath，并按照所设置路径的先后顺序进行源文件的搜索。注意，这里可给我们提供了一种设置路径优先级的方法。例如，在 A 路径中有一个 main.c，在 B 路径中也有一个 main.c，那么我们就可以通过优先设置那一个路径的 vpath %.c，来决定优先编译哪个路径的 mian.c 了。这种设置方法有的时候是很有用的，各位读者可以根据系统各自不同的需求来使用。

4.3.3 编译器变量的定义

```
#-------------------------------------------------------------------------
# Target file name and type definitions
#-------------------------------------------------------------------------

EXETYPE       = elf#                        # Target image file format
MODULE        = bin#                         # Binary module extension

#-------------------------------------------------------------------------
# Target compile time symbol definitions
#-------------------------------------------------------------------------
```

```
ARMASM          = -D_ARM_ASM_#

#-------------------------------------------------------------------------
# Software tool and environment definitions
#-------------------------------------------------------------------------
ARMBIN    = $(ARMHOME)/Bin#
ARMBIN := $(subst \,/,$(ARMBIN))#

ARMCC      = $(ARMBIN)/armcc#             # ARM ADS ARM 32-bit inst. set ANSI C compiler
ASM          = $(ARMBIN)/armasm#          # ARM ADS assembler
LD           = $(ARMBIN)/armlink#         # ARM ADS linker
HEXTOOL = $(ARMBIN)/fromelf#         # ARM ADS utility to create hex file from image

OBJ_CMD       = -o#                          # Command line option to specify output filename
```

在这段代码中主要定义了各种使用的变量，EXETYPE 和 MODULE 分别是链接生成的文件的扩展名；ARMASM 是给编译器使用的符号定义变量；ARMBIN 是 ARM 编译器所在文件路径；接下来的是使用的 ARM 编译器可执行程序名。这里主要说明一下“ARMBIN := $(subst \,/,$(ARMBIN))#”语句，subst 是 Make 字符串替换函数，原形是：

$(subst <from>,<to>,<text>)

意义是将<text>字符串中的<from>字符串替换成<to>字符串。在这里是将“\”替换成“/”来表示路径分割符，在 Make File 中将“/”做为路径的分割符的。

在紧接着的代码都是定义编译器和链接器命令的变量，这里就不一一的介绍了，有兴趣的读者可以参考 ARM 编译器的帮助文档。

4.3.4 依赖规则

```
#-------------------------------------------------------------------------
# Default target
#-------------------------------------------------------------------------
.PHONY : all
all :    startup $(TARGET).$(MODULE) complete

.PHONY : startup
startup:
    @echo ----------------------------------------------------------------
    @echo Compile startup
    @echo ----------------------------------------------------------------

.PHONY : complete
complete:
    @echo ----------------------------------------------------------------
    @echo All have been done
```

```
    @echo ----------------------------------------------------------------
```

这里的 all 是整个 Make File 中的第一个依赖关系，根据前面的叙述我们知道，make 会从这个依赖关系开始执行。all 是 make 的一个目标，由于 all 只是一个标签，并没有与之相关的文件，因此我们称 all 为一个“伪目标”。“.PHONY”用来显式地指明一个目标是“伪目标”，目的是向 make 说明不管是否有这个文件，这个目标都是“伪目标”。通常在没有和“伪目标”同名的文件情况下，是不需要使用“.PHONY”来显式声明一个“伪目标”的。

我们可以看到 all 目标有三个子目标：startup、$(TARGET).$(MODULE)和 complete。make 将按照这个顺序执行三个子目标。startup 和 complete 两个目标显示了开始和结束的提示信息，echo 是在屏幕上输出信息的命令。默认情况下 make 将显示其执行的命令，在命令前面加上“@”符号将不会输出命令信息。如 startup 部分如果不加“@”则会有如下结果输出：

```
    echo ----------------------------------------------------------------
    ----------------------------------------------------------------
    echo Compile startup
    Compile startup
    echo ----------------------------------------------------------------
    ----------------------------------------------------------------
```

接下来让我们看看$(TARGET).$(MODULE)规则：

```
#-------------------------------------------------------------------------
# Lib file targets
#-------------------------------------------------------------------------

$(TARGET).$(MODULE) : $(TARGET).$(EXETYPE)
    @echo ----------------------------------------------------------------
    @echo TARGET $@
    $(HEXTOOL) $(TARGET).$(EXETYPE) $(BINFORMAT) $(OUTPUT) $(TARGET).$(MODULE)

$(TARGET).$(EXETYPE) : $(OBJS)
    @echo ----------------------------------------------------------------
    @echo TARGET $@
    $(LD) $(INFO) $(LINK_OPT) $(LIST) $(LINK_CMD) $(TARGET).$(EXETYPE) $(OBJS)
```

$(TARGET).$(MODULE)依赖于$(TARGET).$(EXETYPE)，当$(TARGET).$(EXETYPE)文件比$(TARGET).$(MODULE)文件要新的时候，就会执行$(TARGET).$(MODULE)命令，执行生成$(TARGET).$(MODULE)文件的命令。

$(TARGET).$(EXETYPE)依赖于$(OBJS)中包含的各个.o 文件。如果有任何一个.o 文件比$(TARGET).$(EXETYPE)文件要新，则执行链接命令。$@表示目标文件集，在本例中也就是$(TARGET).$(EXETYPE) : $(OBJS)中的$(TARGET).$(EXETYPE)。

接下来就到了.o 文件的依赖关系了，这部分是一个 Make File 的主要处理的部分，在正式介绍它们之前我们先看下面的定义：

```
#=========================================================================
#                              DEFAULT SUFFIX RULES
#=========================================================================
```

```
SRC_FILE = $(@F:.o=.c)#              # Input source file specification
OBJ_FILE = $(OBJ_CMD) $(@F)#     # Output object file specification

.SUFFIXES :
.SUFFIXES : .o .c .s .mix .dep
```

SRC_FILE = $(@F:.o=.c)定义了输入的源文件文件名，$(@F)表示$@中的文件名部分，$(@F:.o=.c)就是将$(@F)中的.o 替换成.c，从而得到了源文件名。注意，这一句需要在执行命令的时候才会进行真正的展开，因此在这里只是定义了一个变量。OBJ_FILE 定义了一个输出目标.o 文件的变量。

.SUFFIXES 是用来通知 make 新的扩展名。.SUFFIXES:语句是清空默认的文件扩展名识别。.SUFFIXES : .o .c .s .mix .dep 声明了五个文件扩展名：

.o —— C 编译器生成的文件

.c —— C 语言源文件

.s —— 汇编语言源文件

.dep —— .c 和.s 文件的头文件依赖关系

.mix —— 由.c 生成 C 和汇编语句混合文件的扩展名

```
#-------------------------------------------------------------------------
# C code inference rules
#-------------------------------------------------------------------------
%.o:%.c
    @echo ----------------------------------------------------------------
    @echo OBJECT $(@F)
    $(ARMCC) $(CFLAGS) $(OBJ_FILE) $(SRC_FILE)
    @echo ----------------------------------------------------------------

%.mix:%.c
    @echo ----------------------------------------------------------------
    @echo OBJECT $(@F)
    $(ARMCC) -S -fs $(CFLAGS) $(INC) $(OBJ_FILE) $<
    @echo ----------------------------------------------------------------
```

上面定义了.o 和.mix 文件的默认依赖关系，在这里固定了它们和源文件.c 之间的关系。“%”号是通配符，%.o 代表任何一个.o 文件。%.o:%.c 指定相应的.o 文件依赖于.c 文件。$<表示依赖的文件名，这里是指%.c 所匹配的文件。.mix 文件是将 C 语句“翻译”成对应的汇编语句，这个文件可以用来分析 C 源文件的效率。

```
#-------------------------------------------------------------------------
# Assembly code inference rules
#-------------------------------------------------------------------------
%.o:%.s
    @echo ----------------------------------------------------------------
    @echo OBJECT $(@F)
    $(ARMCC) -ansic -E $(AFLAGS) $(ARMASM) $< > $*.i
    $(ASM) $(AFLAGS) $(OBJ_FILE) $*.i
    @echo ----------------------------------------------------------------
```

上面定义了.o 文件与.s 文件的依赖关系。$(ARMCC) -ansic -E $(AFLAGS) $(ARMASM)

$< > $*.i 语句的意思是将汇编语句首先使用编译器进行预编译处理，这样处理的好处是我们可以在汇编文件中使用 C 的预编译指令了，比如#define 和#include 等。处理之后的文件通过“>”重定位符号输出到$*.i 文件中。$*表示目标的没有扩展名部分，比如 dir/main.o :mian.c 则$*等于 dir/main 部分。

4.3.5 .Dep 文件

```
#---------------------------------------------------------------------------
# Depend file inference rules
#---------------------------------------------------------------------------
%.dep:%.c
	@echo ----------------------------------------------------------------
	$(ARMCC) -ansic -M $< > $*.dep
	@echo ----------------------------------------------------------------
```

上面定义了.dep 文件的依赖关系。.dep 文件是由编译器输出的符合 make 依赖规则的.o 文件与.h 文件的依赖性规则定义的文本文件。这里就是生成.dep 文件的命令部分。在 Make File 中会通过 include 命令包含.dep 文件（include 还可以包含其他的 Make File 文件），从而引入目标文件与头文件之间的依赖关系，典型的.dep 文件内容如下：

```
mainapp.o:    mainapp.c
mainapp.o:    system.h
```

下面来看看怎样包含.dep 文件：

```
# ----------------------------------------
# C file dependency list
# ----------------------------------------
ifeq ($(MAKECMDGOALS),)
-include $(C_OBJS:.o=.dep)
else
ifeq ($(filter all,$(MAKECMDGOALS)),all)
-include $(C_OBJS:.o=.dep)
endif
endif
```

ifneq（ifeq）是 Make File 的条件判断语句，在 Make File 中有六个条件判断的关键字：ifeq、ifneq、ifdef、ifndef、else 和 endif。ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else 表示条件表达式为假的情况。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。ifeq 是判断条件相等，ifneq 是判断条件不等。ifdef/ifndef 是判断变量是/否被定义。

$(filter all,$(MAKECMDGOALS))是从变量 MAKECMDGOALS 中过滤出 all 字符串。MAKECMDGOALS 变量中会存放我们所指定的终极目标的列表，如果在命令行上，我们没有指定目标，那么这个变量是空值。这个终极目标是指我们在调用 make 命令时输入的，例如执行 make clean，则 MAKECMDGOALS 变量值就是 clean。filter 是 make 的过滤函数，与之对应的还有 filter –out 的反过滤函数，它们的函数原型如下：

$(filter <pattern...>,<text>)

$(filter-out <pattern...>,<text>)

filter 函数以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。filter-out 函数以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。

这里 ifeq ($(MAKECMDGOALS),)判断用户指定的目标是空时则执行 include 命令。否则在目标是 all 的时候也执行 include 命令。如果 include 的文件不存在，make 会查找该文件的依赖关系，在这里则产生了.dep 的依赖关系并执行生成.dep 文件的命令。在命令前面加“-”表示忽略执行命令时产生的错误。

这部分就完成了一个由 include 触发的.dep 的生成.dep 到包含.o 与.h 依赖关系的整个过程，具有很强的参考价值。

4.3.6 清除规则

```
#-------------------------------------------------------------------------
# Clean target
#-------------------------------------------------------------------------

# The object subdirectory, target image file, and target hex file are deleted.
.PHONY : clean
clean :
	@echo ------------------------------------------------------------
	@echo CLEAN
	-rm -f *.o
	-rm -f *.i
	-rm -f *.dep
	-rm -f $(TARGET).$(EXETYPE)
	-rm $(TARGET).$(MODULE)
	-rm $(TARGET).map
	@echo ------------------------------------------------------------
```

这里就是调用了一些前面讲过的命令，清除生成的文件。rm 是一个 Shell 程序，不同的 Shell 会有不同的命令，各位读者可以根据自己不同的运行环境进行相应的修改。

4.4 Make File 符号说明

在上一节中介绍了大部分 Make File 中的符号的意义和用法，这里为了方便各位查找，将最常用的一些 Make File 符号列举出来。

4.4.1 关键词

关键词	用途
define	定义一个“数据包”，是用 enddef 做结尾，可以包含多行的命令。
ifeq/ifneq	条件判断，可以搭配 else 使用，endif 结尾。原型：ifeq(Arg1,Arg2)。
ifdef/ifndef	变量是否定义的条件判断，可以搭配 else 使用，endif 结尾。原型：ifdef Var。

=	变量赋值语句。如果右值包含另一个变量，则可以在后面定义这个变量。
:=	变量赋值语句。如果右值包含另一个变量，则只能引用已定义的变量。
?=	条件赋值语句。如果此变量未定义才重新赋值。
+=	为当前变量追加内容。
%	通配符
vpath	设置搜索路径，原型 vpath %.x <path>，x 表示文件扩展名
\	换行符
@	放在命令前面隐藏命令输出
-	放在命令前面忽略命令错误
:	依赖规则定义符，使用方式：目标:依赖
override	用来指示即便此变量是由 make 的命令行参数设置的，也使用新的赋值。因为默认情况下 Makefile 中对这个变量的赋值会被忽略。
.PHONY	显式声明伪目标
.SUFFIXES	声明扩展名

4.4.2 Make 函数

按功能字符串、函数名排序：

函数原型	描述
$(subst <from>,<to>,<text>)	把字串<text>中的<from>字符串替换成<to>。
$(patsubst <pattern>,<replacement>,<text>)	查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）
$(strip <string>)	去掉<string>字串中开头和结尾的空字符。
$(findstring <find>,<in>)	在字串<in>中查找<find>字串。
$(filter <pattern...>,<text>)	以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。
$(filter-out <pattern...>,<text>)	以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。
$(sort <list>)	给字符串<list>中的单词排序（升序）。
$(word <n>,<text>)	取字符串<text>中第<n>个单词。（从一开始）
$(wordlist <s>,<e>,<text>)	从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。
$(words <text>)	统计<text>中字符串中的单词个数。
$(firstword <text>)	取字符串<text>中的第一个单词。
$(dir <names...>)	从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。

$(notdir <names...>)	从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“/”）之后的部分。
$(suffix <names...>)	从文件名序列<names>中取出各个文件名的后缀。
$(basename <names...>)	从文件名序列<names>中取出各个文件名的前缀部分。
$(addsuffix <suffix>,<names...>)	把后缀<suffix>加到<names>中的每个单词后面。
$(addprefix <prefix>,<names...>)	把前缀<prefix>加到<names>中的每个单词后面。
$(join <list1>,<list2>)	把<list2>中的单词对应地加到<list1>的单词后面。如果<list2>的单词个数要比<list1>的多，那么，<list2>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list1>中。
$(foreach <var>,<list>,<text>)	把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach 函数的返回值。
$(if <condition>,<then-part>)	$(if <condition>,<then-part>,<else-part>)，<condition>参数是 if 的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part>会被计算。
$(call <expression>,<parm1>,<parm2>,<parm3>...)	call 函数是唯一一个可以用来创建新的参数化的函数。我们可以写一个非常复杂的表达式，这个表达式中，我们可以定义许多参数，然后我们可以用 call 函数来向这个表达式传递参数。当 make 执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。
$(origin <variable>)	origin 函数不像其它的函数，他并不操作变量的值，他只是告诉我们这个变量是哪里来的。<variable>是变量的名字，不应该是引用。所以我们最好不要在<variable>中使用“$”字符。Origin 函数会以其返回值来告诉我们这个变量的“出生情况”，下面，是 origin 函数的返回值: “undefined” 如果<variable>从来没有定义过，origin 函数返回这个值“undefined”。 default” 如果<variable>是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。 “environment” 如果<variable>是一个环境变量，并且当 Makefile 被执行时，“-e”参数没有被打开。 file” 如果<variable>这个变量被定义在 Makefile 中。

	“command line” 如果<variable>这个变量是被命令行定义的。 “override” 如果<variable>是被 override 指示符重新定义的。 “automatic” 如果<variable>是一个命令运行中的自动化变量。
$(error <text ...>)	产生一个致命的错误，<text ...>是错误信息。退出 Make 执行。
$(warning <text ...>)	输出一段警告信息，而 make 继续执行。
$(shell <command>)	使用 Shell 执行<command>命令

4.4.3 自动化变量

自动化变量通常用来在依赖规则的命令行中表示规则的一部分。通常依赖规则是如下定义方式：Target : Prerequisites。

$@	表示规则中的目标文件集 Target。在模式（即"%"）规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。
$%	仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix 下是[.a]，Windows 下是[.lib]），其值为空。
$<	依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。
$?	所有同目标相比更新的依赖目标的集合。以空格分隔。
$^	所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。
$+	这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。
$*	这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名是比较有效。如果目标中没有模式的定义，那么"$*"也就不能被推导出，但是，如果目标文件的后缀是 make 所识别的，那么"$*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是 make 所能识别的后缀名，所以，"$*"的值就是"foo"。这个特性是 GNU make 的，很有可能不兼容于其它版本的 make，所以，我们应该尽量避免使用"$*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是 make 所不能识别的，那么"$*"就是空值。

上面七个自动化变量可以加上 D（Directory）或 F（File Name）来分别表示路径和文件名部分。例如，$(@F)表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"；$(@D) 表示"$@"的目录部分（不以斜杠做为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

4.5 小结

Make 是用来进行工程管理的一个非常有效的机制，它核心的思想是通过定义不同的依赖关系，实现工程编译的管理，不必每次编译的时候都将全部的源文件重新编译，而只需要编译从上次编译时直到现在已经修改的源文件。虽然现代的集成开发环境（IDE）已经不再需要我们直接修改 Makefile，但是在一些没有 IDE 开发环境领域（如嵌入式系统或操作系统的开发）还是需要直接使用 Make 进行工程管理的。况且，通过对 Make 的了解可以贯通整个程序开发的过程，这对于希望掌握程序开发来龙去脉的相关人员来说是十分有必要的。还有一点是现在关于 Make 的文档和书籍基本上没在市面上见到，就算是有提到也不是十分的详细，因此希望这里可以提供给各位读者一个相对系统和详细的介绍。

思考题

Make 与编译器之间有什么样的关系？

第二篇 磨刀不误砍柴工

紧接着上一篇的基础知识，在这一篇里，我们将首先来看看 BREW 到底是个什么样的东西，它有哪些东西组成以及我们可以用它来做些什么。本篇的名称叫做“磨刀不误砍柴工”，原因就是因为在这一篇中介绍的都是一些 BREW 的基础知识，并没有分析它内部的实现机制和原理，因为我想，如果要深入的研究一个系统，就要先仔细的看看它的外表是什么样子的，只有这样，我们才可以在分析晦涩的实现原理的时候，与 BREW 的外在联系起来，我非超人，我无法凭空的想象一个系统的实现，因此只有将它和我们可以看得到的联系起来了。这种联系将有助于我们对它本质的理解，可以达到事半功倍的效果。

在我看来，其实技术只不过是一层窗户纸，捅破了就可以一通百通。就拿软件开发来说，如果我们可以对一门语言（如 C 语言）了如指掌，那么，我们在学习其他语言的时候自然就会轻松许多。当然这种了解包括语言的应用以及语言的本身。远的不说，就说摆在我们眼前的这本书，如果您可以将上一篇中的内容轻松读懂，并且可以回答的出每一章后面的思考题，那么，其实您已经是一位上了层次的程序员了。因为您起码已经对 C 语言的应用和内部实现机理了解的很清楚了。处于这个层次的 C 语言程序员，可以利用语言本身的特性编写出高效的代码，而且解决问题的能力也很强。理所当然的，这样的人在开发团队里起码也是核心骨干了。之所以在这里说的这么多，是为了提醒各位读者：在我们还没有对一样技术了然于胸的时候，请不要求全责备！知识广博自然好，但那应该是在我们已经拥有一门深入的技术之后的事情。而且没有一门深入的学问，也会反过来影响我们扩充知识的速度，降低我们的学习的效率，得不偿失啊。

如果您现在还没有这样的一门深入技术，那就赶快选一门吧。如果您选择 BREW，那么恭喜您，您不但可以获得 C 语言的深入技能，同时您也选择了一种相对简单和易于理解的系统。如果拥有了对这个相对简单系统的了解，那么对于您了解诸如 Windows 这样的大型系统来说，无疑已经获得了一个很好的基础，因为 BREW 展示给您的是从上到下的二进制环境。

闲话少说，篇如其名，这一篇中的主要内容如下：

第五章是 BREW 简介，主要是介绍 BREW 产生的背景以及 BREW 系统的组成以及 SDK 中的内容和目录结构。通过这一章的学习我们将能够了解 BREW SDK 的状态和组成，对 BREW 有一个初步的认识。

第六章是使用 Applet 和模块，主要介绍了使用 BREW 编程的相关内容，诸如 MIF 文件，Class ID 以及处理 BREW 事件等基础的介绍。通过这一章我们将了解构成 BREW 应用程序的各个关键要素，以及它们的作用。

第七章是创建新的 BREW 应用程序，这一章里我们将见识到本书中第一个真正的 BREW Applet，以及这个 Applet 中各个要素的讲解。从这一章开始，我们将正式的进入 BREW 开发的世界。

第八章是 BREW 的事件处理，这一章将介绍 BREW 的事件处理机制，同时根据这个机制开发一个应用程序的框架。这个框架使用一个有限状态机的形式来实现，这对于我们开发 BREW 应用程序来说是十分有用的，希望各位读者可以仔细研读本章中的内容。

在进行本篇内容准备的时候，我参考了目前市面上已有的 BREW 的文档及其结构，因为，我并不认为我在讲述这些问题的时候会比他们更加高明，在这里我们要感谢他们的辛勤劳动。当然内容仍然会经过精心的编排，同时仍然是本书风格的叙述。

第五章 BREW 简介

在移动通讯领域，不可变化的应用程序抑制了消费者的使用兴趣，亟需一种可以实现类似 Windows 的图形化系统，在这个系统上，可以容易的开发应用程序，同时支持应用程序的安装和管理等功能。当然，少不了的要求是系统不能过于庞大，小巧似乎是移动通讯设备永恒的主题，对于软件也不例外。在这样的背景下，美国高通公司凭借其在 CDMA2000 平台上的霸主地位，开发出了 BREW，并率先在 CDMA2000 的平台上使用。随着 CDMA 在中国的使用，BREW 也走进了国人的视线，于 2002 年进入中国。在 3G 时代即将到来之时，BREW 也被应用到了高通的 WCDMA 系统中，可以说 BREW 已经成为了高通平台图形化界面开发的主力军，不但现在再用，将来也还会用。

使用 BREW 官方的说法：BREW 是由美国高通公司开发的、应用于无线通讯设备的、进行无线动态应用程序的下载和管理的跨平台的集成开发环境。这句话充分的说明了 BREW 的定位、功能和特性。BREW 定位于无线通讯设备，简洁高效是无线通讯设备的必备特点；功能就是实现无线动态应用程序的下载和管理，这是它的主要功能，当然现在 BREW 的一个用途就是做为用户界面开发的平台，而且这个功能大有愈发流行之趋势；特性就是跨平台，由于 BREW 是使用 C 语言开发的，因此理论上在任何平台上都可以运行。最后的，BREW 是一个集成开发环境，其中包含了 PK（Porting Kit，BREW 移植包）和 SDK 两部分，PK 是集成在移动通讯设备代码映像里的，SDK 是基于 Windows 的开发工具包。

在这一章里，将主要介绍 BREW SDK 的安装及其使用等内容，关于 PK 部分在后续的章节中会有所介绍。

5.1 BREW 是什么？

BREW 是 Binary Runtime Environment for Wireless（无线二进制运行时环境）几个英文单词的缩写，从这几个单词中我们可以看到 BREW 的基本特性：

1、二进制（Binary）：BREW 的 API 是一组二进制的库和组件的集合，生成的目标程序是可以直接在二进制环境下执行的 CPU 指令。这代表了 BREW 的应用程序是高效的，无须经过任何中间层的转换。

2、运行时（Runtime）：BREW 的应用程序及扩展是在运行时发现，并根据需要载入运行的，区别于传统的“大映像”模式的程序方法。这不但代表提供了一个运行时的支持，同时也说明 BREW 采用了按需载入的方式，最节约系统资源。

3、环境（Environment）：一个开放灵活的针对无线通讯设备的客户/服务器环境。正是这样的一个结构，才彻底改变了手持设备上单一应用程序的尴尬局面。同时也正是因为 BREW 提供了一整套的解决方案，才培育出 BREW 应用程序的生存环境。

4、无线（Wireless）：特别针对无线通讯设备而设计的。这代表了他的要求是简洁高效，而且可以随时随地与网络联系，方便快捷。

5.1.1 BREW 系统的组成

整个 BREW 系统由开发平台、运行平台和服务器三个要素组成。开发平台就是我们所使用的 BREW SDK，用来在 PC 端开发可以在运行平台运行的程序。运行平台就是指可以

运行 BREW 应用程序的移动通讯设备上的 BREW 运行环境，它的核心是 BREW 的 Porting Kit。服务器是连接开发和运行平台的一个“连接器”，开发平台所开发出的 BREW 应用程序放在服务器上，以便于运行平台的用户通过无线通讯网络下载应用程序，其核心是 ADS（Application Download Server）服务器，ADS 和其他的辅助工具合起来统称为 BDS（BREW Distribution System）系统。在整个 BREW 系统中这三个要素缺一不可，没有了哪一个都会让 BREW 显得不完整。它们之间的关系如下图：

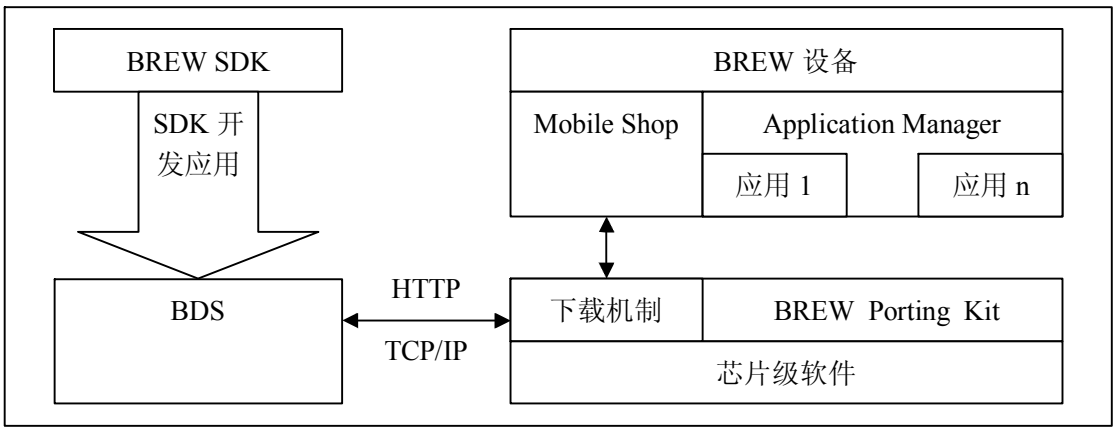

图 5.1 BREW 三要素之间的关系

在图 5.1 中，Mobile Shop 是 BREW 设备上管理应用程序下载部分的程序，意思是像一个商店一样可以买程序。Application Manager 是管理下载应用的程序。这两个部分都属于 BREW 运行平台的一部分，而且是两个特殊的 BREW 应用程序。基本的流程就是使用 BREW SDK 开发的程序交给 BDS，BDS 经过内部处理后，会根据应用程序所支持的 BREW 设备，放在该设备的可下载程序列表中。BREW 设备的使用者通过 Mobile Shop 来获得可下载列表，通过网络下载感兴趣的应用程序，同时支付相关的费用。

BREW 的三要素分别对应了 BREW 产业链里的三个主体，开发平台对应了 BREW 应用程序的开发者，服务器代表了运营商，运行平台代表了 BREW 设备的使用者也就是用户。用户通过 BREW 下载自己喜欢的程序，获得使用应用程序的乐趣，同时支付报酬。运营商通过对用户应用程序的下载获得收入，而开发者则从运营商的用户下载收入中获得报酬。

BREW 设备制造商也是这个产业链中的的重要一环，为什么我的设备上要支持 BREW 呢？其实，原因就是我可以通过支持 BREW 提升设备的价值，从而提升利润空间。而且 BREW 还可以担当一个设备的用户界面开发平台的角色，可以简化设备用户界面的开发时间，从而减少开发成本。这是一个多赢的产业链，以此可以获得各方的支持。

5.1.2 BDS 系统

BREW 三要素之间，BDS 处于中心地位，而且也承担了许多关于 BREW 的幕后工作。BDS 主要是由运营商主导的一个 BREW 下载的控制中心，其主要的任务是维持整个 BREW 产业生态链的生存环境。虽然 BDS 对于 BREW 技术本身没有多大的意义，但是，它确是 BREW 产业生存的核心内容。通过对它的了解，我们可以知道 BREW 的地位和用途，这样也可以反过来影响我们对 BREW 的理解。现在我们就来看看 BDS 系统的内部结构，如图 5.2：

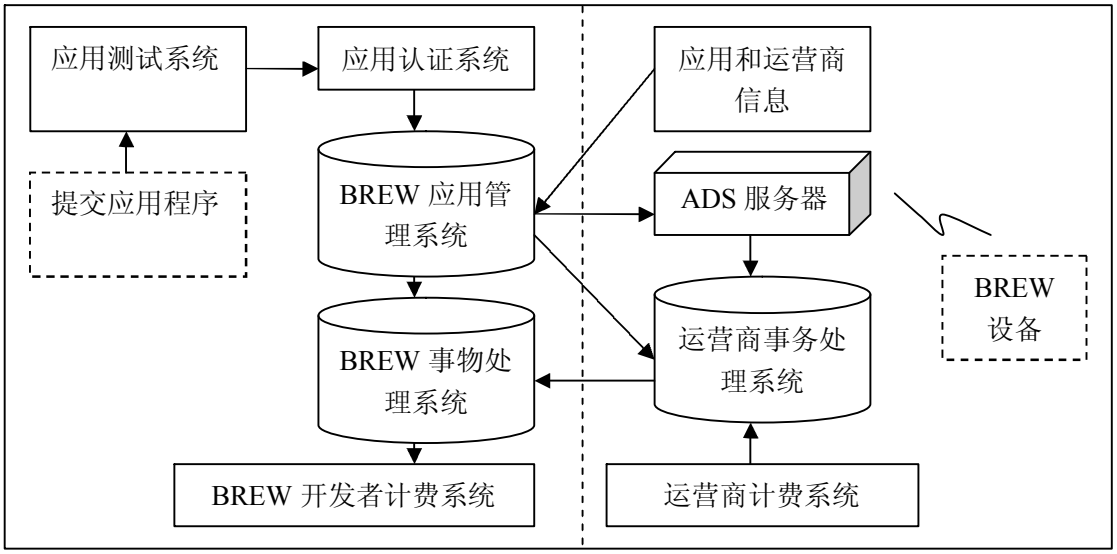

图 5.2 BDS 系统

图中虚线左半部分属于由高通公司统一控制的部分，右半部分属于运营商控制部分。通常应用程序的开发者提交应用程序给高通公司，然后进入整个 BDS 系统的处理流程。首先应用程序进入测试系统，需要通过一个叫做 True BREW 的测试，之后进入认证系统进行应用程序的注册，然后将应用程序放置到 BREW 应用程序的管理系统中。最后，应用程序的开发者将通过 BREW 的计费系统，根据应用程序的下载次数获得报酬。在 BREW 设备的用户端，通过与运营商的 ADS 服务器联系，获得应用程序，同时支付相关的下载费用。系统会通过运营商事务处理系统和 BREW 事务处理系统的联系支付相关的下载费用给应用程序的开发者。这样就完成了整个 BREW 价值链之间的连接，BREW 的整个生态环境通过 BDS 系统得以维持。

5.1.3 BREW 设备系统架构

BREW 设备是整个 BREW 产业链中的终点，为整个产业输送资金血液，整个产业的生存都取决于BREW 设备。BREW 设备是BREW 的运行平台，这个平台的核心是BREW Porting Kit，也就是支持 BREW 应用程序运行的软件库和资源的集合，是 BREW 赖以生存的土壤。BREW 设备的系统架构如图 5.3 所示：

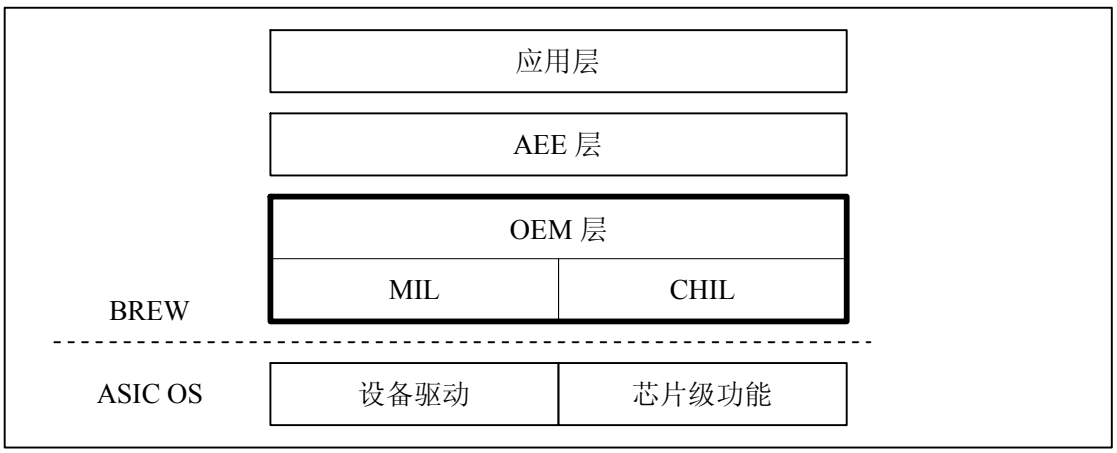

图 5.3 BREW 设备系统架构

在 BREW 内部，分为 OEM 和 AEE 层。AEE 层是 BREW 的接口层和内核所在的位置，应用程序就是通过一定的方法来调用 AEE 层的方法，来调用 BREW 函数库中的函数。OEM 层是提供给 BREW 设备制造商用来实现 BREW 底层接口的层次，也就是说，一个平台上如果需要支持 BREW，那么就需要将在这个平台上实现 BREW OEM 层的函数，通常这个平台与某种专用集成芯片极其操作系统有关，例如基于 X86 芯片架构的 Windows 操作系统和基于高通公司 CDMA 系列芯片的 RTOS 实时操作系统。OEM 层的接口函数分为两种，一种是 MIL（Mobile Interface Layer），另一种是 CHIL（Chip Interface Layer）。MIL 层对应的是专用芯片组外围设备的驱动，典型的设备是 LCD 显示屏。CHIL 层对应的是芯片组的功能函数，典型的功能如 TAPI（呼叫处理）、SMS、Socket 网络接口等。

BREW 的软件架构就是在这样的分层结构中，一层一层的封装，完成了 AEE 层提供给应用程序的一组 API 函数，应用程序位于这个层次的最顶层。如果您问我为什么 BREW 要分层的话，那么我会告诉您，这个原因和您写程序时要分函数的道理是一样的，就是因为程序变得多了，我们才将它们横向分成不同的层次，纵向分成不同的模块，每个模块又分成了好多的函数。软件的难点就在于这些层次的划分，模块的划分，函数和 API 的定义和划分。当然，现在我这样的介绍 BREW 的架构，目的是让您能够有一个总体的概念。您不必现在就细细的研究，可以在您已经对 BREW 有了一定的认识之后再体会这些架构的意义。

5.2 BREW SDK 的安装

BREW SDK 运行于 Windows2000/XP 及其以后的操作系统中，BREWSDK 的安装十分简单，就像是安装其他的 Windows 应用程序一样。我们可以免费的从高通公司的网站上获得 BREW SDK，它目前采用的是在线安装方式，需要我们连接到高通的网站上才能下载安装，要求使用 Internet Explorer 5.5 及其以上的版本。高通公司的网址如下：

https://brewx.qualcomm.com/brew/sdk/download.jsp?page=dx/zh/

如果您不能进入这个页面，证明您还没有注册。这个注册十分的简单，只需要输入您的 E-mail 地址就可以了。注册页面可以通过如下连接方式进入：

http://www.qualcomm.com

安装完成后会增加 BREW 的环境变量和安装程序组，在这个程序组里面就是 BREW SDK 中的内容了，接下来我们就来看看 BREW SDK 的组成。

5.3 BREW SDK 的组成

BREW SDK 中包含了一组工具和组件，应用程序开发者可以通过这些工具和组件高效、快速的开发出多种多样的应用程序，这些工具和组件包括：

1、BREW AEE。BREW AEE 是一个可扩展的面向对象的应用程序开发和执行环境，它提供了一个使用 C/C++开发应用程序和共享模块的平台，同时它采用了类似 Windows 等操作系统的事件驱动程序运行方式。

2、BREW 模拟器（Emulator）。BREW 模拟器是一个用来载入并测试我们所开发应用程序的前端图形用户界面（GUI）。模拟器中可以预载入很多设备的模拟文件，同时也可以自己定制相应的模拟设备，如屏幕大小，按键数量等。模拟器是对 BREW AEE 在设备上运行环境的一个模拟。

3、设备配置器（Device Configurator）。从 BREW3.0 开始设备配置器从 BREW 标准的 SDK 中提取出来了，变成了一个独立的单元，如果需要使用这个功能的话，必须从高通网

站上下载安装。通过它我们可以创建在 BREW 模拟器中使用的设备文件。

4、资源文件编辑器（Resource Editor）。BREW 应用程序中使用的字符串、图片等数据都是从资源文件中获取的，这个工具就是用来完成 BREW 资源文件编辑的。而且，由这个工具生成的资源文件，在 Windows 模拟器中与在 BREW 设备中的事一样的。

5、MIF 文件编辑器（MIF Editor）。BREW 的每一个 Applet 都需要有一个 MIF（Module Information File）文件，这个文件中包含了载入当前模块的必要信息。

6、BREW 头文件。这些头文件中定义了 BREW 的接口、使用的常量以及结构体类型等。这些头文件与在 BREW 设备上运行的 PK 中的头文件是一致的，否则在 SDK 中开发的应用程序就不能在 BREW 设备上运行了。

7、Visual Studio 插件。这些插件是用来建立 BREW 应用程序项目时，在 Visual Studio 中的向导，通过这些向导可以很容易的构建 BREW 应用程序的框架。从 BREW3.0 开始，Visual Studio 向导已经不是 BREW SDK 安装包中的一部分，需要单独下载安装。在安装之前需要在系统中装有 Visual Studio6.0 或更新的系统。

通过上面的这些工具之间交互使用，我们就可以方便的构建一个 BREW 的应用程序。它们之间的交互关系请看图 5.4：

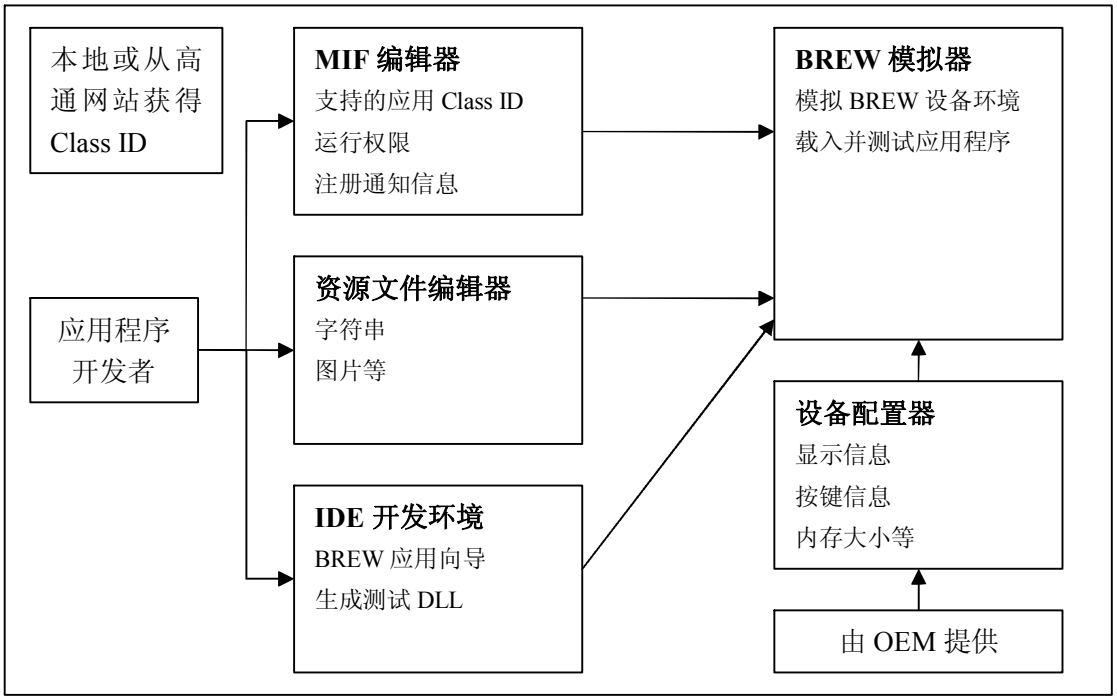

图 5.4 BREW 组件之间的交互

从图中可以看出，BREW SDK 中处于核心地位的是 BREW 模拟器，其他工具都是为它服务的。在这里需要特别说明的是，在 BREW3.0 之后的 BREW SDK 版本的软件中，设备配置器单独的从 SDK 中提取出来了，成为了一个叫做 BREW Device Configurator 的单独需要安装的应用程序。资源文件编辑器、MIF 文件编辑器等实用工具也从 BREW SDK 中提取出来了，变为了一个叫做 BREW SDK Tools 的一个独立的安装包。除了 BREW 模拟器仍然在 SDK 中的安装包之内以外，其他的都需要独立的下载安装。这些内容在 BREW2.x 时代是全部都集成在 BREW SDK 安装包之内的。

除此之外，还有一个叫做 BREW Tools Suite 的一个 BREW 工具集，这些工具属于 BREW 设备上的 BREW 辅助工具。如 BREW Logger 是与设备连接获取应用程序调试信息的，BREW Application Loader 是用来将 BREW 应用程序从 PC 端下载到 BREW 设备上的。这些工具是在使用 BREW 设备时必不可少的。这个工具集也有一个单独的安装包，不过只有 BREW 授

权用户才可以下载安装。

5.4 BREW SDK 的目录结构

安装完成后的 BREW 目录根目录 SDK 下主要包含以下几个文件夹：

1、bin。在此目录下包含了 BREW 在 Windows 环境下运行的可执行文件和数据文件。在子目录“bin/DataFiles”下，包含了 BREW 模拟器使用的声音文件和声调数据库。在“bin/en”目录下包含了 BREW 系统本身在英文模式下所使用的资源文件，我们还可以指定其他语言的资源文件。这些资源文件根据所模拟的实际 BREW 设备的不同，而有不同的图片颜色，这就是目录下的“256Color”、“4Grey”和“mono”的含义。在“bin/Modules”目录下，包含了几个 BREW 模拟器使用的扩展 DLL 文件。在“bin/priv”目录下，存储了 BREW 模拟器运行时，由每个应用程序产生的配置信息。根目录下就是 BREW 模拟器及其相关文件的所在了。

2、bitmaps。在此目录中包含了 BREW 模拟器中所使用的图片。

3、Devices。此目录中包含了 BREW 模拟器使用的设备模拟文件，其中可以包含多种设备文件。通常这些设备文件由各个 BREW 设备的 OEM 厂商提供，也可以自己使用设备文件编辑器自己创建，并在 BREW 模拟器中指定需要使用的设备文件。通常在 BREW 模拟器中运行的设备文件效果如图 5.5。

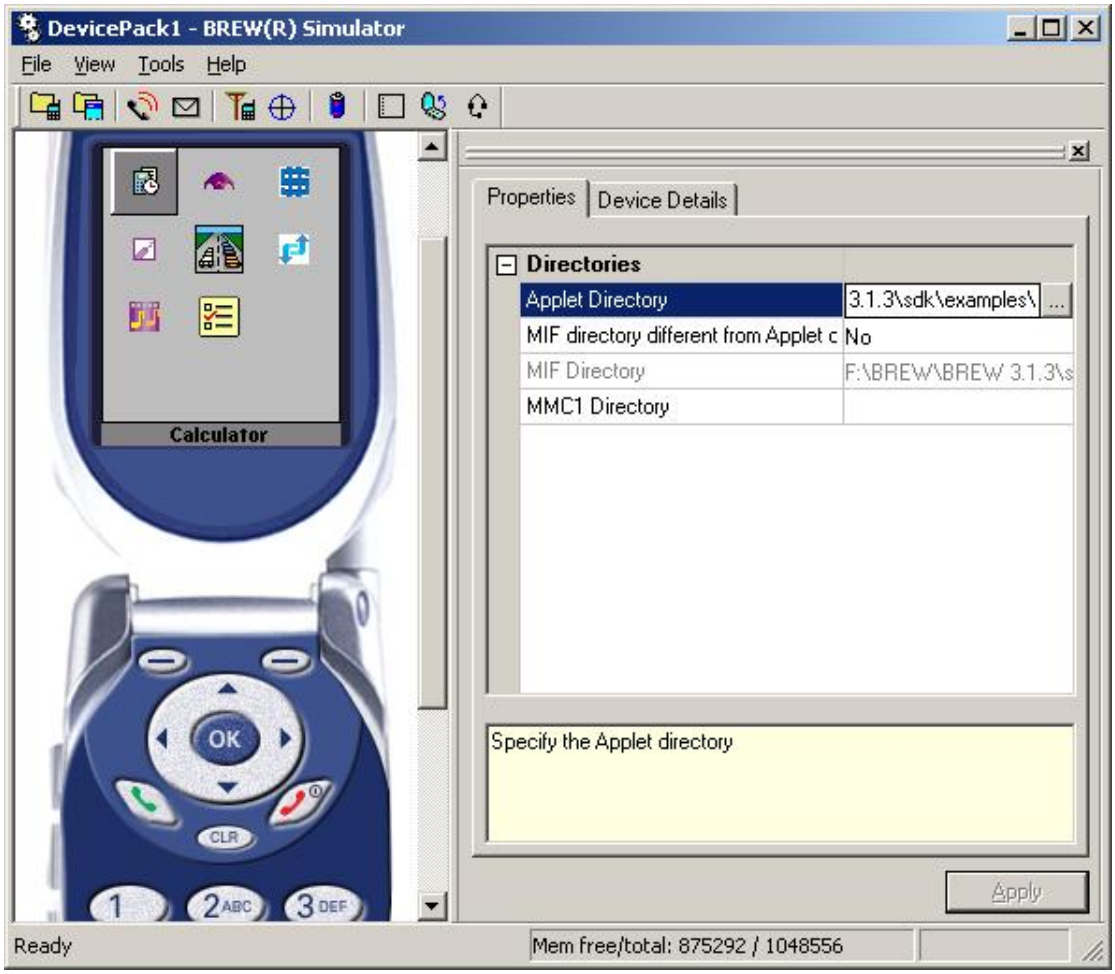

图 5.5 BREW 模拟器中设备文件效果图

4、Docs。此目录中包含了 BREW SDK 的文档，我们可以从这里获得详细的 BREW API

的说明，基本的编程方法等内容。

5、examples。此目录下包含了 SDK 中附带的几个事例应用程序供开发者参考，其中涉及了通常开发 BREW 应用程序时的大部分问题，各位读者可以仔细研究一下每一个事例。

6、inc。此目录中包含了 BREW 的头文件，这些头文件中包含了 BREW 的接口定义，数据类型定义等。BREW 应用程序就是通过使用这些接口定义来使用接口的。

7、src。此目录下包含了 BREW 应用程序通用的 C 语言函数，这些函数与模块的创建和应用程序的入口有关。在里面还有一些包含实用助手函数的源文件，通过这些函数可以简化一些接口的使用。

在 BREW 模拟器中运行时的目录结构与在实际 BREW 设备上的相同，而且对于文件名和路径的使用规则也相同。但是路径加文件名的长度则不同，在我们需要在实际设备上测试应用程序的时候，请查看 BREW 设备的数据文件（此文件随同 BREW 设备一起提供），确定长度是否符合规格的要求。在 BREW 环境中，使用的都是小写的字母做为路径和文件名，不能够使用大写或大小写混合的模式。

5.5 BREW 环境

BREW 是一个可扩展的、面向对象的应用程序开发和执行环境，它使用 C/C++语言进行开发。同时，BREW 还被设计成了一个所见即所得开发系统。BREW 支持应用程序和共享模块，应用程序和模块做为独立的 DLL 开发，运行时由模拟器载入。共享模块中可以包含功能模块，例如开发游戏时，可以将增强功能的图形接口进行重新封装，做为应用开发中的接口使用，这些扩展接口可以同 BREW 标准接口一样在应用程序中使用。

BREW 提供了多组不同功能对象（接口）和功能，这些对象和功能统称为服务，这些服务包括：

1、提供支持简单的基于事件的处理方式的应用程序服务

2、提供访问文件系统、网络服务、内存和显示的核心服务

3、提供增强的图形功能、多媒体、声音和 GPS 定位等增强服务

4、提供诸如支持浏览器开发的综合服务

在这些不同功能和级别的服务中包含了 BREW 的多种类型的接口，如 IShell 接口负责应用程序的控制和管理等功能；IDisplay 负责显示的控制和屏幕刷新等功能。BREW 的每一个接口使用一个 32 位的接口 ID 表示，这个接口 ID 叫做 Class ID。这些 Class ID 都是由高通公司统一管理的，如果需要公开自己开发的接口，那么必须使用高通的注册网页申请一个 ID，申请后这个 ID 就会保留下来给我们自己，不会再分配给其他的人。如果我们的接口只是做为本地测试使用，那么我们可以使用自己定义的接口 Class ID，需要注意的就是不要与现有接口的 ID 相同就可以了。每一个 BREW 接口都是继承自一个叫做 IBase 的接口。

在使用一个接口之前，必须首先使用 ISHELL_CreateInstance()接口函数创建接口实例。IShell 的接口指针是通过调用 ISHELL_CreateInstance()接口的当前应用程序，或接口的入口函数传递进来的，不需要创建。创建实例后，这个接口实例的指针将通过 ISHELL_CreateInstance()方法传递回来。对于一些特定的接口不是通过这个方法创建的，而是通过其他方法直接返回的，如 IImage 接口就是通过 ISHELL_LoadImage()方法获得的。

BREW 的应用程序也是使用一个 32 位的 ID 来表示的，这个 ID 也叫做 Class ID，它与接口的 Class ID 没有什么区别，而且是统一分配的。如果希望我们的应用程序能够分发给用户下载，除了要经过测试以外，您还必须申请全球唯一的 Class ID，就像公开接口一样。要获得这样的 ID，我们必须成为一个授权的开发者。关于如何成为一个授权的开发者请到高通的网站上察看详细的内容。

5.6 BREW 的优缺点

BREW 平台从问世之初，就由于它是由国际技术巨头高通公司的推出而备受瞩目，那么，是不是这种“血缘”关系才让它如此的引人注意呢？答案当然是否定的，除了血缘关系之外，BREW 也有很多自身的优点：

1、BREW 提供了一种高效的 RAM 内存和固定存储器的管理方式，采用了按需载入的方式，所需资源最少。

2、BREW 应用程序的开发支持类似 Windows 的基于事件的处理机制，使得应用程序的开发更加容易。

3、BREW 提供了资源文件的方式，可以容易的进行多语言版本应用程序的开发。

4、所有的文本采用宽字符方式，每一个字符占用两个字节。同时还支持由 OEM 指定的文本格式。

5、由于在 BREW 和底层设备驱动和数据结构之间进行了良好的封装，因此，BREW 应用程序的开发者不必关心底层平台的数据结构和设备驱动。

6、BREW 应用程序的每个模块可以独立开发，最小化了开发时间并且避免了集成问题。

7、模块在二进制层次继承，简化了增加模块到 BREW 平台的任务。

8、BREW 应用程序支持直接接收特定短消息，利用它可以开发出交互性的应用程序。

9、BREW 应用程序可以控制全部核心接口和资源，如显示、声音等。

当然了，任何系统有优点，同时也会有缺点，BREW 平台也是一样的。

1、BREW 平台的接口实现方式决定了它处理异常的能力先天不足，如果使用了一个已经被释放的接口指针，则系统很容易崩溃。

2、当前的 BREW 接口封装的不够完善，有些功能使用现有的 BREW 接口无法进行控制，有些状态也无法获取。例如，对于具有翻盖外观的通信设备来讲，BREW 的 API 中没有可以获得当前状态的方法。

3、部分数据类型封装不够严谨。例如当前关于通话事件中各种通话装态仍然在使用底层的定义，而没有在 OEM 层封装成 BREW 所有的 IPhone 事件。

人无完人，期望 BREW 平台可以在今后的版本中更加完善。

5.7 小结

在本章中概述了 BREW 系统的组成以及 BREW 产业链的三要素：开发平台、运行平台、服务器。以及它们所对应的关切者开发者、用户和运营商。接下来介绍了 BREW 的基本层次结构，这样的层次结构使得 BREW 出色的完成了对底层平台的封装，并且最小化了移植 BREW 平台的工作量。还介绍了 BREW SDK 的目录结构，运行环境等内容。同时，也在本章中第一次见识了 BREW 模拟器的样子。最后介绍了 BREW 的优缺点。总之，本章的目的就是让我们能够对 BREW 有一个初始的概念。

思考题

1、BREW 三要素是什么？哪一个处于中心地位？

2、BREW SDK 有哪几个组件？它们都有什么作用？

第六章 使用 Applet 和模块

在 BREW SDK 中，每一个应用程序模块做为一个独立的 Windows DLL 文件开发。每一个模块中可以包含一个或多个 Applet，并且必须有一个与此应用程序模块对应的 MIF 文件。通过 BREW MIF Editor（MIF 文件编辑器）创建的这个 MIF 文件中，包含了关于这个模块信息，例如支持的类、支持的 Applet、Applet 的权限和 Applet 信息等。在 MIF 文件中还包含了模块中每一个类和指定给其他应用程序使用的类的唯一 Class ID。我们这里所说的类，包含了 Applet 和扩展接口。

一个模块可以从 BREW 的资源文件中读取数据，使得应用程序中可以使用字符串、图片和对话框等资源。通过在资源文件中存储指定的语言数据，使得针对不同国家开发不同版本的应用程序成为可能。我们可以使用 BREW Resource Editor（资源文件编辑器）来为应用程序创建资源文件，同时生成资源文件中关于资源定义的头文件。

一个已经开发的 BREW 应用程序可以运行在模拟器上（DLL 文件），也可以运行在指定的设备上（MOD 文件）。如果需要生成 MOD 文件，必须包含所运行设备 CPU 类型的专用编译器，如 ARM CPU 的 C/C++编译器，不过对于普通的开发者来说，获得 ARM 编译器需要从 ARM 公司购买软件，这就需要一笔小投资了。建立应用程序的基本的头文件和源文件已经在 BREW SDK 中提供了，通过这些文件可以创建一个应用程序和资源文件。BREW 应用程序使用的资源文件和二进制资源文件，无论应用程序运行在模拟器环境下，还是在设备的 BREW 环境下，都是使用相同的文件格式，无需在设备和模拟器之间进行不同的处理。

下面就列举出了开发一个 BREW 应用程序所需的组件：

1、BREW AEE 随 SDK 提供的头文件（在 SDK 中的 inc 目录下的.h 文件）

2、BREW 模块创建所需的助手源文件（AEEAppGen.c 和 AEEModGen.c）

3、Applet 源文件和头文件

4、使用 MIF 文件编辑器创建 MIF 文件

5、Applet 资源文件和相应的资源文件头文件，这些文件使用 BREW 资源文件编辑器创建

注意，应用程序中使用的源文件，对于 Windows 环境和指定设备环境下是相同的，使用同样的源文件去建立 Windows 的 DLL 二进制文件和设备指定的 MOD 二进制文件。

在这一章中，我们将主要根据上面列举出来的，构成 BREW 应用程序的要素进行一一讲解，期望能够让您对 BREW 应用程序有一个更加详细的了解。

6.1 MIF 文件

MIF 文件是每一个 BREW 应用程序必不可少的，其中存储了该模块的详细信息，这些信息包括支持的接口类、支持的 Applet 类以及 Applet 的标题图标等内容。在系统启动的时候（模拟器或 BREW 设备），BREW 枚举所有的 MIF 文件。对于每一个 MIF 文件，BREW 会获取其中全部支持的类。可以通过 BREW 的 API（如 ISHELL_EnumApplet 等）使这些信息在应用程序中使用，应用程序管理器（Application Manager）可以通过这些 API 列举出当前系统中的全部 BREW Applet。

在开发过程中，BREW MIF 文件的命名有严格的要求，主要在以下两个方面：

1、MIF 文件名必须使用小写字母做为开头。

2、MIF 文件名中至少要包含一个字母，不能全部是数字命名的。

下面的表格中包含了有效和无效的 MIF 文件名的事例：

有效 MIF 文件名	无效 MIF 文件名	无效原因
abc.mif	Abc.mif	文件名中使用了大写字母
a2c.mif	123.mif	全数字文件名
a23.mif	1ab.mif	第一个不是字母

不能够使用大写字母的原因是，在 BREW 平台上，规定了文件名不支持大小写混合。不过在当前 BREW3.x 的模拟器上，检测到大小写混合的文件名的时候，只是显示一个警告对话框，但是应用程序是可以在模拟其中运行的。但这不能够保证在实际的 BREW 设备中也可以运行，因此我们禁止这样的命名方法。

对于首字母是数字或全部数字命名的 MIF 文件名，是 BREW 在从 ADS 服务器上下载应用程序是自动替换和命名的，因此是保留的命名方式。而且 BREW 对待全数字命名的应用程序有特殊的方式，因此，如果在开发过程或发布过程中使用这种方式命名的 MIF 文件，可能导致我们的应用程序发生致命的错误。因此，我们应当禁止这种做法。

6.2 BREW 的 Class ID

在 BREW 平台中，每一个接口类或 Applet 都必须有一个唯一的 Class ID。BREW 应用程序开发者可以通过高通网站的https://brewx.qualcomm.com/classid/home.jsp页面获得这些 Class ID。不过，只有注册的授权用户，同时交纳了 Class ID 的购买费用之后才可以申请。如果您当前正在开发应用程序，并且还没有注册成为授权用户，您可以使用临时的 Class ID 进行开发。如果我们手动的为我们的应用程序分配 Class ID，那么我们必须保证它们是唯一的。如果有两个或两个以上的接口 Class ID 或 Applet Class ID 是一样的，那么这些接口和 Applet 将有可能不能运行。例如，考虑下面的目录结构：

abc.mif

test.mif

test\test.dll

如果 abc.mif 和 test.mif 使用了同样的 Class ID，那么 test Applet 将不能正确地运行。因为，BREW 是按照文件名的顺序枚举 MIF 文件的，因此，对于指定的 Class ID，abc.mif 会首先被载入。所以，当我们试图启动 test 应用程序的时候，可能会启动 abc 这个应用程序，或者什么应用程序也启动不了（如果 abc.mif 中仅仅输入了接口类的话）。

6.3 创建一个接口的实例

在一个模块或 Applet 中可以使用任何一个 BREW 的接口，不过在使用这些接口方法之前，需要先创建接口的实例，也就是获得接口的指针。BREW 创建一个接口实例的方法是通过调用 ISHELL_CreateInstance(IShell * pIShell, AEECLSID cls, void * * ppobj)成员函数，这是一个在 BREW 中十分有用的 API 接口函数。在调用这个函数的时候，需要指定创建接口的 Class ID，以及接收接口指针数据的接口指针，这个接口指针通过**ppobj 的二重指针传入。在这个成员函数执行的时候，BREW 会根据指定的 Class ID 搜索支持类的列表，这个列表由两部分组成：一种是内嵌在 BREW 平台中的接口，如 BREW 的标准接口；另一种是系统启动时枚举的 MIF 文件中支持的类。BREW 中调用 ISHELL_CreateInstance 成员函数的具体执行内容如下：

1、查找支持当前类的模块

2、将这个模块载入内存（如果当前没有载入的话）

3、调用 IMOUDULE_CreateInstance()成员函数创建接口实例

如果成功创建了这个类的实例，那么，将会通过参数 ppobj 返回这个接口实例的指针。在这个接口的使用者不再需要这个接口的时候，必须调用该类的 Release 方法释放接口实例，同时将接口指针置空。在编写 BREW 应用程序的时候，创建实例的应用程序一定要负责释放这个实例。

所有的 BREW 类都是从 IBase 继承而来的。IBase 接口中有两个方法，IBASE_AddRef 和 IBASE_Release，由于全部的类都是从 IBase 继承而来，那么，全部 BREW 类都支持 AddRef 和 Release 方法。这两个方法是用来控制接口实例引用计数的。引用计数也就是指当前的类实例有多少个指针指向它（也就是引用它）。当增加一个引用的时候，必须调用 AddRef 方法增加类实例内部的引用计数，当释放一个引用的时候必须调用 Release 方法减少引用计数。当内部引用计数为 0 的时候，就会释放这个类的实例。我们必须严格的遵守这个规则，否则将不能够正确地释放接口实例所占用的系统资源，如内存等，会引起系统资源的耗尽，从而导致系统崩溃。

6.4 创建和终止一个 Applet

在 BREW 模式下，Applet 是一个支持 IApplet 接口的类。这使得 BREW 可以通过一个简单的接口控制所有的 Applet。除了标准的 AddRef 和 Release 方法外，IApplet 接口还支持 HandleEvent 方法。

BREW 可以通过两种方式创建一个 Applet：

1、直接调用 ISHELL_StartApplet(IShell * pIShell, AEECLSID cls)成员函数。这个成员函数允许创建 Applet 并同时通过 IApplet 的 HandleEvent 方法发送 EVT_APP_START 事件。EVT_APP_START 事件告知 BREW Applet 现在已经处于激活状态，同时可以刷新屏幕了。

2、注册通知或闹钟事件。在这种情况下，将立刻创建 Applet 并接收指定的通知事件。如果 Applet 需要刷新屏幕或接收按键输入时，可以通过调用 ISHELL_StartApplet 来启动自身。

如果需要终止当前激活的 Applet，可以调用 ISHELL_CloseApplet(IShell * pIShell, boolean bReturnToIdle)成员函数。调用此方法时，将会终止当前处于激活状态的 Applet，同时发送 EVT_APP_STOP 事件告知 Applet：应用程序将被关闭，请释放相关资源。

6.5 处理 Applet 的事件

当一个 BREW Applet 正在运行的时候（出于激活状态），它将通过 HandleEvent 函数接收事件。这些事件包括键盘、对话框和控件更改事件。下面是一个 Applet 的示例事件流程：

EVT_APP_START

…

其他 BREW 事件（EVT_KEY 等）

…

EVT_APP_SUSPEND（可选）

EVT_APP_RESUME（可选）

…

其他 BREW 事件（EVT_KEY 等）

…

EVT_APP_STOP

通常情况下，一个 Applet 值需要处理几种事件就可以了，包括 EVT_APP_START、EVT_APP_STOP、EVT_KEY 和 EVT_COMMAND。如果在我们的 Applet 中没有处理一个事件的话，我们应该在应用程序的 HandleEvent 函数中返回 FALSE。这将让 BREW 采用默认的方式处理这个事件。任何一个 BREW Applet 的核心函数都是 HandleEvent，因为 BREW 基于事件驱动的机制就是通过这个函数体现出来的，它是 BREW Applet 的一个事件入口。Applet 的运行也是通过这个函数接收事件而运转的。

6.6 挂起和恢复 Applet

在 BREW 环境下，同一时刻仅能有一个激活的、顶层可见的 Applet。但是，在同一时刻可以有多个 Applet 处于运行状态。这个顶层可见的意思是这个 Applet 控制着主显示屏以及接收键盘事件。除了这个顶层可见的应用程序外，其他处于运行状态的应用程序统统处于一个叫做挂起（Suspend）的状态。处于挂起状态的 Applet 除了主屏显示和接收键盘事件外，其他的操作都可以正常进行。如果需要将处于挂起状态的应用程序激活，处于顶层可见，那么需要调用 ISHELL_StartApplet 函数，或通过调用 ISHELL_CloseApplet 函数关闭当前活动的 Applet 直到需要激活的应用程序为止。

当 App1 处于顶层可见时，如果 App2 通过调用 ISHELL_StartApplet 变为顶层可见，那么，App1 将被挂起。当 App2 关闭时，App1 被恢复。挂起和恢复是 BREW 描述一个 Applet 是否具有主显示屏和键盘事件控制权的一个机制。

当 BREW 挂起一个 Applet 的时候，会向这个 Applet 发送 EVT_APP_SUSPEND 事件。如果这个 Applet 将这个事件的处理结果返回 TRUE，则表示告知 BREW 已经处理了挂起事件，但是，此时的 Applet 不会从内存中卸载。如果当前 Applet 不希望处理挂起事件，它可以返回 FALSE，这样，BREW 将会终止当前的应用程序，并发送 EVT_APP_STOP 事件，同时在内存中卸载这个应用程序。任何在 EVT_APP_START 事件中分配的内存，都应改在 EVT_APP_STOP 事件中释放。任何在 AEEClsCreateInstance()函数中分配的内存，需要在应用程序的 APPFreeData()函数中释放。如果在内存是在 AEEClsCreateInstance()中分配的话，那么在 EVT_APP_STOP 事件中释放将会是十分危险的。举个例子，如果在 EVT_APP_START 事件处理中返回了 FALSE，那么 EVT_APP_STOP 事件将不再发送，这样就导致了内存泄露。

在模拟器中，可以通过选区菜单的方式来模拟 EVT_APP_SUSPEND 事件的发送。例如，选取菜单“工具-〉设置”，则模拟器会向当前的应用程序发送 EVT_APP_SUSPEND 事件，在关闭这个设置对话框之后，会发送 EVT_APP_RESUME 事件。

6.7 应用程序堆栈和 IAppHistory 接口

BREW 允许有多个 Applet 同时运行，但是只有一个 Applet 是激活的。这个时候就牵扯到了 BREW Applet 管理的问题，这就是 BREW 应用程序堆栈。通过应用程序的堆栈我们可以：

1、在应用程序历史数据中，允许一个应用程序出现多次

2、即便应用程序处于背景运行状态下（Suspend），也可以使应用程序出现在应用程序历史数据中。

BREW 通过 IAppHistory 接口管理应用程序的历史数据列表，同时允许一个应用程序在

堆栈中存在多个历史数据条目。例如，在当前的 BREW 历史数据中，可以存在如下的应用程序条目序列：

App A -> App B -> App C -> App A -> App B

此时应用程序 A 和 B 都在历史列表中有两个条目。此时 App B 处于激活状态，如果关闭 App B，那么，上面的序列将变成下面这样：

App A -> App B -> App C -> App A

这样的处理方式可以使得整个的应用程序序列按原样返回。并不是每一个应用程序每次启动时都会增加历史数据，但是可以保证的是当前每一个正在运行的应用程序都会有历史数据，无论它是处于激活或挂起状态。如果当前使用 ISHELL_StartApplet 启动的应用程序，没有一个相关的历史数据条目相对应的话，则 BREW 会在历史数据列表中增加条目。

如果历史列表中该应用程序相关的历史数据已经存在的话，是增加新的历史数据条目，还是仅仅改变当前历史数据的顺序，将遵循如下的规则：

1、如果这个应用程序已经在前面挂起的过程中，通过调用 IAPPHISTORY_SetResumeData()设置了恢复数据的话，那么，将会在历史数据列表中新增历史数据条目。此时就是同一个应用程序对应多个历史数据条目的情况。

2、如果这个应用程序已经在前面挂起的过程中，没有调用 IAPPHISTORY_SetResumeData()设置恢复数据，那么，将不增加历史数据条目，而只是改变已有历史数据的顺序。由于在 BREW3.0 版本之前，BREW 并没有提供 IAppHistory 接口，因此这些版本全部采用的是这种处理方式。

在使用非 IAppHistory 接口的方式关闭应用程序的时候，将会删除与全部已关闭应用程序的历史数据条目。使用 IAPPHISTORY_Stop()关闭一个应用程序的时候，不会从应用程序堆栈中移除历史数据，除非这个应用程序处于激活状态。

BREW 的 IAppHistory 接口提供给我们控制 BREW 应用程序的高级方法，使得我们对 BREW 应用程序的控制能力进一步加强了（BREW3.0 之前的版本是没有这个接口的）。例如，当系统的内存不足时，我们可以通过使用 IAPPHISTORY_Stop()方法卸载在内存中的应用程序，而不影响当前的应用程序堆栈。

6.8 创建自定义通知

通知事件是 BREW 提供的一种常用的高级功能。通过这个通知机制，我们可以接收到来自网络、闹钟等模块的事件，而不必采用轮询的方式进行监测，这大大提高了程序的效率。最常见的几种通知类型是通话、短信息、闹钟（闹钟使用的是 EVT_ALARM 事件，但它与通知事件原理相同）事件。一个应用程序如果要获得这样的通知事件，需要使用 ISHELL_RegisterNotify()函数进行事件注册。注册后，当事件产生的条件满足时，BREW 将发送 EVT_NOTIFY 事件给注册的应用程序。不管这个应用程序是否处于激活状态，都能够接收到这个事件。

除了 BREW 系统的通知事件以外，我们还可以创建自己的通知类。基本步骤如下：

1、编写一个从 INotifier 接口继承而来的 BREW 接口类（非 Applet），同时实现 INotifier 接口成员函数。

2、定义接口类所需的通知掩码。

3、实现这个新的接口类。

一旦这个接口类被实现了，那么任何一个 BREW 的应用程序都可以注册这个接口的通知事件。在这里，我们并没有详细的讲解如何的实现这个接口类，因为这里的目的是让您了解 BREW 有这样的一个能力，在后面的章节中，我们会详细的讨论如何创建自定义的通知

类。

6.9 小结

在本章中，主要介绍了使用 BREW Applet 和模块的方法，首先讲述了 BREW 应用程序的组成部分 MIF 文件和 Class ID 的作用，以及他们的特性。接下来介绍了使用 BREW 的 ISHELL_CreateInstance()函数创建接口实例的方法，让您掌握在应用程序中使用接口的方法。再接下来介绍了创建 Applet 和处理 BREW 事件的方法，让您懂得 IApplet 接口中 HandleEvent 成员函数的作用。重点介绍了 BREW 应用程序堆栈的管理方式以及 IAppHistroy 接口的高级特性。最后告诉您，BREW 可以创建自己的通知类。总体上来讲，这章的内容从总体上介绍了一下 BREW 编程的特性，在下一章里，我们将通过实际的例子来讲解创建 BREW Applet 的过程。

思考题

1、Class ID 有什么作用？Applet 和接口的 Class ID 有什么区别吗？
2、MIF 文件有什么作用？
3、应用程序堆栈管理中，增加历史数据条目的规则是什么？

第七章 创建新的 BREW 应用程序

在第六章中，我们介绍了 Applet 和模块的相关内容，并且熟悉了 BREW 开发环境，那么现在是开始创建一个属于我们自己的应用程序的时候了。在 BREW 中创建应用程序最简单的方式是，通过 BREW 在 Visual Studio C++环境中的应用程序向导。通过向导，可以一步一步的让我们构建成功开发 BREW 应用程序所需要的组件。接下来我们就详细的介绍一下 BREW 应用程序的开发方法。

7.1 写在开发前面的话

在进行真正的 BREW 应用程序开发之前，有一些开发的注意事项需要事先讲明。这样可以让我们避免一些经常性的错误，从而减少开发调试的时间。这些注意事项主要包含如下几个方面：

1、从 Windows 的模拟器移植到 BREW 设备的问题

2、在 BREW 设备上会出现问题，而在 BREW 模拟器中不必检测的

3、良好的 BREW 编程习惯

通过对这些注意事项的检查，可减少从 Wondows 到 BREW 设备上的移植任务，同时让程序可以在 BREW 设备上正确运行。详细内容如下表：

注意事项	详细描述
执行空指针检查	关于这一条有如下两个方面： 1、在使用 ISHELL_CreateInstance()或其他接口函数创建接口实例后，一定要进行接口指针有效性的检查，如果指针异常，则不能够使用这个接口 2、检查全部的指针以确认指针有效性，包括传入的、使用 BREW 方法创建的和分配内存后的。不可用的指针将引起系统的致命错误。
避免堆栈溢出	由于系统分配给 BREW 运行环境的堆栈空间是有限的，因此不要在堆栈中分配大数组，也就是说，不要再一个函数的局部变量中声明大数组。如果确实在程序中需要这样的数组，那么，请使用分配内存的方式。
不要使用大的循环	当一个循环运行的时间足够长时，将引起 BREW 设备的复位。基于这样的原因，不能在一个 BREW 事件里进行耗时很长的处理。
尽可能的使应用程序成为设备独立的	为了使我们的应用程序可以独立于设备的内存大小、键盘数量、颜色深度和屏幕尺寸等因素而独立运行，请使用 ISHELL_GetDeviceInfo()方法来获得设备的规格，从而根据不同的规格处理不同的应用程序。
使用资源文件	使用资源文件存储指定语言的字符串、对话框和图片。这样可以方便的进行不同语言版本的开发，而不影响代码本身。
释放内存	由于在 BREW 设备上的可用内存是有限的，因此清理不使用的内存就十分重要了。有如下两种情况： 1、必须释放全部创建的实例 2、应用程序终止时必须释放全部分配的内存

	推荐在内存不再使用的时候就释放掉。
不要使用全局或静态变量	由于全局和局部变量在 BREW 动态应用程序中无法处理，因此基于这样的 BREW 架构不支持全局和静态变量，而且全局和静态变量也会引起 BREW 设备环境下的链接错误。请在应用程序的结构体中声明这些数据。
不要在定义的时候初始化一个结构体	与上面不能使用全局和静态变量一样，在定义的时候初始化一个结构体也会引起 BREW 设备环境下的一个链接错误。
不要直接使用 C 语言的浮点运算	BREW 平台不支持浮点运算。虽然这些运算在模拟器中运行良好，那时因为在 Windows 环境下，在 BREW 设备环境下，通常不支持浮点运算。如果确实需要使用的话，请用 BREW 提供的助手函数进行运算。
使用 BREW 支持的标准库函数	为了可以让我们的程序所占的空间最小，请不要使用目标设备的库函数，而是使用 BREW 提供相应的助手函数。
避免类型转换错误	由于目标设备的编译器对于类型的检查要严格，因此，请明确声明变量转换的类型，以防止从 Windows 环境移植到 BREW 设备环境下产生错误。
检查返回值	如果我们调用的 BREW API 含有返回值，建议处理这些返回值，这样可以让我们所写的程序更加健壮。

7.2 创建一个 BREW 应用程序

在这一节里，我们将首先建立一个叫做 HelloWorld 的应用程序，然后分析这个程序当中的各个要素。也正是从这个例子开始，将正式的带您进入 BREW 的开发世界。在这本书里，我们使用的环境是 Visual Studio .Net 开发环境，当然我们也可以使用 Visual Studio6.0 来创建，并且他们之间看上去没什么太多的不同。

7.2.1 建立应用程序

可以采用如下的步骤建立 HelloWorld 应用程序：

1、我们可以到高通公司的网站上去获取一个名叫 HelloWorld 的应用程序的 Class ID，如果您还没有成为一个授权的 BREW 应用程序开发者，那么您可以将 BREW SDK 样例程序 HelloWorld 中的 HelloWorld.bid 复制过来。

2、使用画图工具或者其他的图片编辑工具为这个应用程序创建图标。图像分为大、中、小三种，我们可以指定 bmp、png、bci 或 jpg 图片，图片的大小没有特殊的要求，我们只需要根据您的喜好选定就可以了。不过当程序运行在设备上时，不同的设备由不同的要求，您可以查看该设备的规格说明。

3、打开 Visual Studio2003。

4、打开 BREW 应用程序向导。通过选择文件->新建->项目菜单，打开创建项目对话框，选择 Visual C++项目下的 BREWAppWizard 项目。在对话框的下部输入应用程序名和应用程序存储路径。点击确定按钮，此时 BREW 的应用程序向导启动。

5、由于我们的应用程序仅需要刷新屏幕，因此保留全部的设置直接点击完成。

6、载入 MIF 文件编辑器，为应用程序创建 MIF 文件。

7、在 MIF 文件编辑器的 Applets 选项卡中，新建 Applet，并选择我们已经获取的

helloworld.bid 文件作为次应用程序的 Class ID。

8、选择应用程序的类型为“Tools”。

9、设置我们在第二步创建的三个图标做为应用程序的图标。

10、保存文件为 helloworld.mfx 文件，同时选择“Build->Compile MIF Script”菜单或者按“F5”键，编译二进制文件。由于此命令输出的文件名采用的是 DOS 短文件名方式，因此请手动修改文件名。

11、关闭 MIF 文件编辑器，返回 Visual Studio 界面。

12、在“项目->helloworld 项目属性”对话框的常规选项中，将输出目录置为空，以便于生成的 DLL 文件可以在根目录中生成。

13、在“解决方案资源管理器”中，打开 helloworld.c 文件。同时输入如下内容：

helloworld.c

```
/*=====================================================================
FILE: helloworld.c
=====================================================================*/

/*=====================================================================
INCLUDES AND VARIABLE DEFINITIONS
=====================================================================*/
#include "AEEModGen.h"          // Module interface definitions
#include "AEEAppGen.h"          // Applet interface definitions
#include "AEEShell.h"           // Shell interface definitions

#include "helloworld.bid"

/*---------------------------------------------------------------------
Applet structure. All variables in here are reference via "pMe->"
---------------------------------------------------------------------*/
// create an applet structure that's passed around. All variables in
// here will be able to be referenced as static.
typedef struct _helloworld {
    AEEApplet       a ;         // First element of this structure must be AEEApplet
    AEEDeviceInfo   DeviceInfo; // always have access to the hardware device information

    // add your own variables here...
} helloworld;

/*---------------------------------------------------------------------
Function Prototypes
---------------------------------------------------------------------*/
static  boolean helloworld_HandleEvent( helloworld* pMe,
                                        AEEEvent eCode, uint16 wParam,
                                        uint32 dwParam);
boolean helloworld_InitAppData(helloworld* pMe);
void    helloworld_FreeAppData(helloworld* pMe);
```

```
/*=============================================================================
FUNCTION DEFINITIONS
=============================================================================*/

/*=============================================================================
FUNCTION: AEEClsCreateInstance

DESCRIPTION
    This function is invoked while the app is being loaded. All Modules must provide this
    function. Ensure to retain the same name and parameters for this function.
    In here, the module must verify the ClassID and then invoke the AEEApplet_New() function
    that has been provided in AEEAppGen.c.

  After invoking AEEApplet_New(), this function can do app specific initialization. In this
  example, a generic structure is provided so that app developers need not change app specific
  initialization section every time except for a call to IDisplay_InitAppData().
  This is done as follows: InitAppData() is called to initialize AppletData
  instance. It is app developers responsibility to fill-in app data initialization
  code of InitAppData(). App developer is also responsible to release memory
  allocated for data contained in AppletData -- this can be done in
  IDisplay_FreeAppData().

PROTOTYPE:
  int AEEClsCreateInstance(AEECLSID ClsId,IShell * pIShell,IModule * po,void ** ppObj)

PARAMETERS:
    clsID: [in]: Specifies the ClassID of the applet which is being loaded

    pIShell: [in]: Contains pointer to the IShell object.

    pIModule: pin]: Contains pointer to the IModule object to the current module to which
    this app belongs

    ppObj: [out]: On return, *ppObj must point to a valid IApplet structure. Allocation
    of memory for this structure and initializing the base data members is done by
AEEApplet_New().

DEPENDENCIES
  none

RETURN VALUE
  AEE_SUCCESS: If the app needs to be loaded and if AEEApplet_New() invocation was
      successful
```

```
   EFAILED: If the app does not need to be loaded or if errors occurred in
       AEEApplet_New(). If this function returns FALSE, the app will not be loaded.

SIDE EFFECTS
   none
=============================================================================*/
int AEEClsCreateInstance(AEECLSID ClsId, IShell *pIShell, IModule *po, void **ppObj)
{
      *ppObj = NULL;

      if( ClsId == AEECLSID_HELLOWORLD )
      {
            // Create the applet and make room for the applet structure
            if( AEEApplet_New(sizeof(helloworld),
                                    ClsId,
                                    pIShell,
                                    po,
                                    (IApplet**)ppObj,
                                    (AEEHANDLER)helloworld_HandleEvent,
                                    (PFNFREEAPPDATA)helloworld_FreeAppData) )
            {
                  //Initialize applet data, this is called before sending EVT_APP_START
                  // to the HandleEvent function
                  if(helloworld_InitAppData((helloworld*)*ppObj))
                  {
                        //Data initialized successfully
                        return(AEE_SUCCESS);
                  }
                  else
                  {
                        //Release the applet. This will free the memory allocated for the applet when
                        // AEEApplet_New was called.
                        IAPPLET_Release((IApplet*)*ppObj);
                        return EFAILED;
                  }
            } // end AEEApplet_New
      }
      return(EFAILED);
}

/*=============================================================================
FUNCTION SampleAppWizard_HandleEvent

DESCRIPTION
```

```
        This is the EventHandler for this app. All events to this app are handled in this
        function. All APPs must supply an Event Handler.

PROTOTYPE:
        boolean SampleAppWizard_HandleEvent(IApplet * pi, AEEEvent eCode, uint16 wParam,
uint32 dwParam)

PARAMETERS:
        pi: Pointer to the AEEApplet structure. This structure contains information specific
        to this applet. It was initialized during the AEEClsCreateInstance() function.

        ecode: Specifies the Event sent to this applet

       wParam, dwParam: Event specific data.

DEPENDENCIES
    none

RETURN VALUE
    TRUE: If the app has processed the event
    FALSE: If the app did not process the event

SIDE EFFECTS
    none
=============================================================================*/
static boolean helloworld_HandleEvent(helloworld* pMe, AEEEvent eCode, uint16 wParam,
uint32 dwParam)
{
      AECHAR szBuf[] = {'H','e','l','l','o',' ',
                                'W','o','r','l','d','\0'};

      switch (eCode)
      {
            // App is told it is starting up
            case EVT_APP_START:
                  // Add your code here...
                  // Clear the display.
                  IDISPLAY_ClearScreen( pMe->a.m_pIDisplay );
                  // Display string on the screen
                  IDISPLAY_DrawText( pMe->a.m_pIDisplay, // What
                                          AEE_FONT_BOLD,                  // What font
                                          szBuf,                          // How many chars
                                          -1, 0, 0, 0,                    // Where & clip
                                          IDF_ALIGN_CENTER | IDF_ALIGN_MIDDLE );
```

```
                // Redraw the display to show the drawn text
                IDISPLAY_Update (pMe->a.m_pIDisplay);
                return(TRUE);

            // App is told it is exiting
            case EVT_APP_STOP:
                // Add your code here...
                return(TRUE);

            // App is being suspended
            case EVT_APP_SUSPEND:
                // Add your code here...
                return(TRUE);

            // App is being resumed
            case EVT_APP_RESUME:
                // Add your code here...

                return(TRUE);

            // A key was pressed. Look at the wParam above to see which key was pressed. The key
            // codes are in AEEVCodes.h. Example "AVK_1" means that the "1" key was pressed.
            case EVT_KEY:
                // Add your code here...
                return(TRUE);

            // If nothing fits up to this point then we'll just break out
            default:
                break;
    }

    return FALSE;
}

// this function is called when your application is starting up
boolean helloworld_InitAppData(helloworld* pMe)
{
      // Get the device information for this handset.
      // Reference all the data by looking at the pMe->DeviceInfo structure
      // Check the API reference guide for all the handy device info you can get
      pMe->DeviceInfo.wStructSize = sizeof(pMe->DeviceInfo);
      ISHELL_GetDeviceInfo(pMe->a.m_pIShell,&pMe->DeviceInfo);

      // Insert your code here for initializing or allocating resources...
```

```
    // if there have been no failures up to this point then return success
    return TRUE;
}

// this function is called when your application is exiting
void helloworld_FreeAppData(helloworld* pMe)
{
    // insert your code here for freeing any resources you have allocated...
}
```

这个文件当中的代码大部分是经过 BREW 向导产生的，为了方便排版，我对里面的代码进行了一些小改动。由于这是第一次展示 BREW 应用程序，因此保留了向导生成的注释等内容，目的是让您能够看到 BREW 应用的全貌。今后的应用程序展示，将只使用骨干程序，以节省版面空间并减少对您钱财的浪费。

正如前面我们所见的，在这个文件中共有四个 BREW 骨干函数存在，它们分别是：AEEClsCreateInstance 、 helloworld_HandleEvent 、 helloworld_InitAppData 以及 helloworld_FreeAppData。它们在整个 BREW 应用程序中扮演着不同的角色，而且 AEEClsCreateInstance 和***_HandleEvent 函数还是每一个动态应用程序必不可少的。接下来让我们来看看它们各自的作用以及相互之间的关系。

AEEClsCreateInstance(AEECLSID ClsId, IShell *pIShell, IModule *po, void **ppObj)函数是整个 BREW 应用程序的一个入口点，在其内部调用了 AEEApplet_New 函数用来创建应用程序实例。AEEApplet_New 函数在 AEEAppGen.c 文件中，这个函数可以指定一个事件处理函数和一个资源释放函数，在这个程序中指定的分别是 helloworld_HandleEvent 和 helloworld_FreeAppData 函数。在 AEEClsCreateInstance 函数中将几个重要的参数传进了 BREW 应用程序中，如 IShell 指针 pIShell、要创建的 Class ID ClsId、模块指针 po 以及应用程序实例返回指针 ppObj 等。我们还可以看到在 AEEApplet_New 函数调用成功后，将执行 helloworld_InitAppData 函数，用来初始化数据，这就是这个初始化函数得到执行的地方。

在 helloworld.c 文件中，还定义了一个 helloworld 结构体。由于 BREW 实现原理上的特点，这个结构体就成为了 BREW 应用程序存储全局变量的地方。如果我们需要在我们的应用程序运行期间保存数据，请在这个结构体中增加变量，而不要使用全局函数。在运行时，AEEApplet_New 函数会根据这个结构体的大小分配存储空间。这个结构体中默认声明的第一个变量是 AEEApplet 结构，这个声明的顺序不可以改变。AEEApplet 的定义如下（参考 AEEAppGen.h 文件）：

```
struct _AEEApplet
{
    //
    // NOTE:    These 3 fields must be declared in this order!
    //
    DECLARE_VTBL(IApplet)
    AEECLSID            clsID;

    uint32              m_nRefs;                // Applet reference counter
    IShell      *       m_pIShell;          // pointer to IShell
    IModule     *       m_pIModule;             // pointer to IModule
```

```
    IDisplay    *       m_pIDisplay;      // pointer to IDisplay

    //Pointer to Handle Event Function
    AEEHANDLER          pAppHandleEvent;

    // Pointer to FreeAppData function. This will be invoked when the
    // reference count of the App goes to zero. This function is supplied
    // by the app developer.
    // NOTE: Apps should NOT directly call their FreeAppData function.
    PFNFREEAPPDATA pFreeAppData;
};
```

这个结构体中的成员变量都是在 AEEApplet_New 函数中进行填充的，我们可以看到里面有 IDisplay 接口指针定义，由于对于每一个 BREW 应用程序来说，IDisplay 接口是必备的，因此 AEEApplet_New 函数里面会创建这个接口的实例，这样我们就不必在我们的应用程序中再次创建这些接口了。在这个结构体里面还有一个 AEEHANDLER 类型的变量，这其实是一个函数指针，用来保存我们通过 AEEApplet_New 函数传递进来的应用程序中捕获事件的函数，在这里面指定的是 helloworld_HandleEvent 函数。还有一个 PFNFREEAPPDATA 类型的变量，这也是一个回调函数的变量，用来保存释放资源的函数，在应用程序退出时将会调用这个函数执行指定的资源释放函数，在本例中这个函数就是 helloworld_FreeAppData 函数。m_pIShell 和 m_pIModule 变量分别存储了 BREW Shell 和模块接口的指针，我们可以在调用 BREW Shell 接口函数（定义在 AEEShell.h 中的 ISHELL_开头的接口）的时候可以使用这个指针，因为 IShell 指针是通过应用程序的入口函数传递进来的，而不需要创建（IShell 是不能够自己创建自己的）。关于 IModule 指针我们会在第三部分的 Shell 内幕中有详细的介绍，这里就不多说了。其他的变量是 BREW 规定的，在这里面我们也不多说了（我们才刚刚开始啊）。

在 BREW 应用程序启动后，会首先向应用程序中发送 EVT_APP_START 事件，这个事件是应用程序收到的第一个 BREW 事件。由于我们指定的应用程序事件捕获函数是 helloworld_HandleEvent，因此在这里将由此函数处理这个事件。在这个事件里我们可以进行应用程序的初始化操作，例如，显示一个界面、创建一个接口实例等等。在本例中我们调用了三个 IDisplay 接口的方法：IDISPLAY_ClearScreen 和 IDISPLAY_ DrawText 以及 IDISPLAY_Update 方法，它们的作用依次是清除屏幕显示、在屏幕上绘制文字以及刷新屏幕。运行后，这段代码会在屏幕的中央显示“Hello World”字样。在处理进行完成之后，需要使用“return TRUE”来告知 BREW 应用程序已经启动成功了，可以接收其他事件了。否则表示应用程序启动失败了，程序将不能正常运行。

在应用程序启动成功之后，可以接收 BREW 传递给应用程序的其他事件了，例如用户点击了键盘，则会有 EVT_KEY 事件发送。当应用程序结束的时候，BREW 会向应用程序的事件捕获函数发送 EVT_APP_STOP 事件。如果我们在 EVT_APP_START 事件中创建了接口实例，那么与此相对应的，我们应该在 EVT_APP_STOP 事件中释放这些接口实例。这个事件也要求捕获者返回 TRUE，以表示应用程序已经成功停止运行了。注意，BREW 内部支持的 Unicode 的字符编码，Unicode 编码是国际标准的双字节字符编码，越来越多的系统都开始支持 Unicode 编码了。双字节宽度的 Unicode 字符与单字节的 ASCII 编码相比，Unicode 可以支持最多 65536 个字符（单字节最多 256 个），几乎可以囊括全世界不同语言的全部字符，中文从 Unicode 编码的 0x4E00 编号开始。

现在我们只需要简单的几步就已经创建了一个 BREW 应用程序了，不要惊奇，一切就

是这么简单！

7.2.2 测试应用程序

在创建完成一个 BREW 应用程序之后，我们的第一个任务就是要在 BREW 模拟其中运行我们的应用程序。首先我们需要在 Visual Studio 程序中生成应用程序。在 Visual Studio 中生成应用程序的时候，会创建一个.dll 的动态链接库文件，BREW 模拟器通过载入这个文件而执行我们的应用程序。使用 Visual Studio 生成应用程序是十分容易的，我们只需要选择“生成->生成解决方案”就可以了。

选择“调试->启动”菜单项编译连接程序，则可以进入调试运行模式，在这种模式下我们可以使用 Visual Studio 的调试功能跟踪我们的应用程序执行。由于我们生成的 BREW 应用程序是一个单独的 DLL 文件，因此，在我们第一次运行调试程序的时候，需要在调试运行程序的对话框中输入模拟器文件 BREW_Simulator.exe 做为运行 DLL 文件的可执行文件，设置对话框如图 7.1 所示。

图 7.1 选择运行应用程序的模拟器程序

一旦我们生成了我们的应用程序之后，我们就需要配置模拟器，用来运行我们的应用程序了。我们需要指定我们的应用程序的 MIF 文件路径和应用程序所在的路径，而且应用程序的.dll 文件必须在与 MIF 文件名同名的文件夹下面。就以本例为例，我们指定的 MIF 文件路径是 Code\hellowrold，而应用程序的路径则是 Code\，因为本例中 MIF 文件在 helloworld 的路径下面。当然我们也可以把 MIF 文件复制到其他的地方，然后指定这个路径作为 MIF 路径。

对于初学者来说可能遇到不能启动应用程序并提示“为了节省空间模块已经被移除”的提示，其实这是由于在我们指定的应用程序路径下面没有找到相关的 DLL 文件所致。通常的原因是我们将 DLL 的输出路径设置在了默认的 helloworld\debug 下面了。解决这个问题有两种方法，一种是将这个文件手动的转移到 helloworld 路径下，另外就是更改项目的输出路径。更改的方法请参照上一节中建立应用程序的第 12 步进行设置。

可以参照如下的步骤来测试我们的应用程序，这样的分步方式会让我们觉得使用模拟器更加简单一些：

1、使用 Visual Studio 中的“调试->启动”或者按 F5 键运行应用程序。

2、当 Visual Studio 提示选择执行 DLL 的应用程序时，请选择 BREW 模拟器所在的文件。

3、打开模拟器的属性页面。如果没有属性页面可以通过 View -> Properties 打开。

4、在属性页面的“Properties”选项卡中，设置“MIF directory different from Applet Dirctory”项目为“Yes”。

5、设置 MIF 文件路径为我们的应用程序 MIF 文件所在的位置。

6、设置应用程序路径为我们的应用程序所在文件夹的上一层路径。注意应用程序所在的路径必须与 MIF 文件名相同。

7、使用模拟器中的设备按键，选择我们的应用程序开始运行。

在本例中，我们启动 HelloWorld 应用程序之后，将会在模拟器的设备屏幕的中间看见“Hello World”字样，此时证明我们的应用程序已经运行成功了。我们可以点击模拟器上的 End 键退出当前的应用程序。

7.2.3 在设备上运行应用程序

乍一看来，在一个 BREW 设备上运行我们的应用程序是一件十分浪费时间的事情，但是在一个真实的硬件环境中运行我们的应用程序是十分重要的，因为这样做可以让我们检测到应用程序在模拟器中没有遇到的错误，尤其是在我们自然不自然的使用 Visual Studio 库函数的时候。不过不幸的是，让我们的应用程序运行在一个 BREW 设备上，并不是一件轻松的事情，因为我们必须联系高通公司的工程师去获得我们所需要的 BREW 设备。如果我们需要在一个还不能从网络下载应用程序的 BREW 设备（通常指手机）上运行我们的应用程序，我们必须首先按照如下步骤进行操作：

1、从高通公司那里获得我们所需要的手机及其规格说明和模拟器设备文件。

2、针对该设备的规格，使用其模拟器设备文件进行模拟开发。

3、生成 BREW 设备上可以运行的二进制.mod 文件，生成本地的 Class ID 和 MIF 文件等。参照上一节的相关描述。

4、从设备的规格说明上获得设备 BREW 菜单的使用方法，打开 BREW 设备的 BREW 测试模式（通常在一个叫做 BREW Flags 的菜单内，选中 BREW_TEST_ENABLE 项目）。

5、从高通的测试网页上获取测试用的签名文件，这个签名文件通常与 BREW 设备的 ESN 有关，这个 ESN 通常使用一个贴纸贴在设备的某个地方。

6、使用 BREW Apploader 应用程序将应用程序载入 BREW 设备中。

7、在 BREW 设备的一个叫做“应用程序管理器”的应用程序中打开我们的应用程序，此时我们就可以在 BREW 设备上测试应用程序了。

由于要在一个 BREW 设备上测试应用程序，必须首先成为一个授权的 BREW 开发用户，这样我们才能够获得 BREW Apploader 和测试用的签名文件。因此我们需要访问高通的 BREW 网站去查看详细的如何成为一个授权用户信息，并按照要求进行注册。成为一名授权的 BREW 应用程序开发者之后，我们不但可以获得测试等相关的 BREW 工具，而且我们还可以购买一定数量的 Class ID。关于如何使用 BREW Apploader 应用程序以及如何为设备生成应用程序的问题，您可以查看后面章节的介绍。

要想做到上面的这些事情，实在是一个让人头痛的事情，或许这是高通公司基于商业和应用程序安全的角度的考虑。不过，如果 BREW 平台可以变得更为开放一些，或许将会在嵌入式开发平台领域产生更大的影响力。

7.3 BREW 开发初步

在我们使用 BREW 进行开发的时候，我们必须彻底的了解两个概念：应用程序流程和在我们的应用程序中可以使用的接口。因为这些主题对于 BREW 开发来说是至关重要的，所以在这一节里将介绍这些内容。如果说前几章让我们窥见了 BREW 的全貌，那么从这一

节开始将正式带您进入 BREW 的应用程序开发。

7.3.1 理解 BREW 应用程序流程

正如我们前面所见到的一样，我们的应用程序是基于事件驱动机制的，也就是说，我们的应用程序通过 BREW 平台发送过来的事件开始运行的。这些事件不但包含了诸如控件发送过来的用户接口事件，而且还包含了描述应用程序诸如启动和停止等外部行为的事件，例如接收一条短消息或者启动一个应用程序。因此，我们的应用程序的中心是一个叫做_HandleEvent 的事件捕获函数。应用程序通过这个事件捕获函数获得系统中的事件（更为准确的说法应该是 BREW 平台通过这个函数将事件传递给应用程序），并通过这些事件检测系统的运行状态（如按键、启动应用程序等等）。进一步的，我们的应用程序判断这些传进来的事件，并决定如何处理这些事件（捕获则返回 TRUE，不捕获则返回 FALSE）。如果我们的应用程序在某个事件中并不希望进行任何处理，通常情况下应首先将它传递给应用程序正在使用的接口（如控件），然后再返回处理结果。这样可以让应用程序中使用的接口获得处理事件的机会。

在我们的应用程序可以捕获事件之前，我们首先需要在系统中注册我们的事件捕获函数。由于我们的应用程序实际上是一个 BREW 接口的实例，并且事件捕获函数不过是这个接口中的一个方法而已，因此我们必须在应用程序接口中增加一个指向我们的应用程序事件捕获函数的引用。虽然这并不是一件困难的事，而且是每一个 BREW 应用程序都必须去做的事情，但是 BREW 还是提供了一个助手函数用来帮助我们实现这个应用程序接口，包括注册一个事件捕获函数。

所有的这一切都发生在我们的应用程序载入的时候。当 BREW 试图载入一个应用程序的时候，它将执行每一个应用程序的 AEEClsCreateInstance 函数。在我们的应用程序的这个函数中，将判断传入的 Class ID 是否与我们应用程序的 Class ID 相同，如果相同则创建一个自身的实例。典型的我们可以通过调用 BREW 的助手函数 AEEApplet_New 函数来实现这些功能。这个 AEEApplet_New 函数将在幕后为我们的应用程序分配存储空间、实现应用程序接口并返回一个我们的应用程序实例。下面就是一个简单的 AEEClsCreateInstance 函数：

```
int AEEClsCreateInstance( AEECLSID clsID,
                          IShell * pIShell,
                          IModule * po,
                          void ** ppObj )
{
    boolean result;
    *ppObj = NULL;

    // 如果 Class ID 符合我的应用程序...
    if( clsID == AEECLSID_MYCLASSID )
    {
        // 使用 BREW 助手函数创建应用程序实例
        result = AEEApplet_New( sizeof( AEEApplet ),
                                clsID,
                                pIShell,
                                po,
```

```
                                        (IApplet**)ppObj,
                                        (AEEHANDLER) HandleEvent,
                                        NULL );
    }
    return result ? AEE_SUCCESS : EFAILED;
}
```

以面向对象的思想去考虑，我们可以将 AEEClsCreateInstance 函数理解为一个创建对象实例的对象工厂，它负责创建一个指定类的实例。在我们的应用程序中实现的 AEEClsCreateInstance 函数仅仅是一个系统调用的类方法，在其他的应用程序请求启动我们的应用程序时，系统将会调用它。注意，无论我们是在创建一个应用程序或者一个扩展接口（与其它应用程序共享的一个 BREW 接口），这些启动的处理都是一样的需要通过 AEEClsCreateInstance 函数，只不过扩展接口不需要注册事件捕获函数，因此不必调用 AEEApplet_New 函数了。在下一部分“一识庐山真面目”里，将会详细的剖析这一启动过程，届时将进一步增加我们对 BREW 平台的理解。

一旦我们已经创建了我们应用程序的实例，BREW 系统将会调用我们应用程序的事件捕获函数，传递各种 BREW 事件。通常，在我们的应用程序中必须处理以下的几个事件：

1、EVT_APP_START 事件。在应用程序启动时，我们在应用程序中注册的事件捕获函数将会接收到这个事件，这表示我们的应用程序已经开始运行了。在我们的应用程序中，可以在这个事件中进行创建接口，或者分配内存空间等操作。

2、EVT_APP_STOP 事件。在我们的应用程序结束时将接收到这个事件，表示应用程序已经停止运行了。我们应该在应用程序收到这个事件的时候，释放全部分配的内存和和创建的接口实例等资源。

3、EVT_APP_SUSPEND 事件。在我们的应用程序接收到这个事件的时候，它表示应用程序需要中断执行。这种情况通常发生在我们在当前的应用程序中启动了另一个应用程序，或者在我们的应用程序运行过程中收到了一个电话等需要打断当前应用程序运行的情况下。在这个事件中，我们需要保存应用程序中的相关状态数据，用于在应用程序恢复执行时恢复程序的状态。此事件过后，应用程序进入挂起状态。

4、EVT_APP_RESUME 事件。在我们的应用程序从中断执行（挂起）状态返回到运行状态时，将会收到这个事件。在这个事件中我们需要根据在 EVT_APP_SUSPEND 事件中保存的状态数据恢复应用程序的执行状态。此事件之后，应用程序就处于正常的活动状态了。

下面将介绍一下其他相关的 BREW 系统事件，对于每一个 BREW 事件，如果有 wParam 和 dwParam 参数，将被传递给给定的小程序和控件。如果我们处理这个事件，需要在事件处理函数中返回 TRUE，否则返回 FALSE。这些事件如下所示：

事件名称	所属类型	描述
EVT_APP_START	系统事件	启动应用程序的事件，dwParam = (AEEAppStart *)
EVT_APP_STOP	系统事件	应用程序停止，无参数
EVT_APP_SUSPEND	系统事件	应用程序挂起，无参数
EVT_APP_RESUME	系统事件	应用程序恢复，dwParam = (AEEAppStart *)
EVT_APP_CONFIG	系统事件	切换应用程序，显示配置界面
EVT_APP_HIDDEN_CONFIG	系统事件	切换应用程序，显示隐藏配置界面
EVT_APP_BROWSE_URL	系统事件	在 EVT_APP_START 之后调用，dwParam = (const AECHAR * pURL)
EVT_APP_BROWSE_FILE	系统事件	在 EVT_APP_START 之后调用，dwParam =

		(const AECHAR * pszFileName)
EVT_APP_MESSAGE	系统事件	文本消息，wParam = AEESMSEncoding，dwParam 取决于 wParam 值的字符串格式
EVT_APP_TERMINATE	系统事件	EVT_APP_STOP 的强制版本。小程序将被释放。
EVT_EXIT	系统事件	在 BREW 终止时发送给所有已加载的小程序
EVT_APP_NO_CLOSE	系统事件	应用程序不应关闭
EVT_APP_NO_SLEEP	系统事件	应用程序正在运行 - 运行应用程序很长时间之后调用
EVT_KEY	按键事件	按键事件，wParam= 按键代码
EVT_KEY_PRESS	按键事件	按键按下事件，wParam = 按键代码，发生在 EVT_KEY 事件之前
EVT_KEY_RELEASE	按键事件	按键抬起事件，wParam = 按键代码，发生在 EVT_KEY 事件之后
EVT_CHAR	按键事件	字符事件，wParam 表示发生事件的宽字符
EVT_UPDATECHAR	按键事件	字符更新事件，wParam 表示发生事件的宽字符
EVT_COMMAND	控件事件	自定义控件的事件，wParam 表示发生事件的 Item ID 值。如菜单按下时发生此事件。
EVT_CTL_TAB	控件事件	控件焦点切换事件，dwParam= 控件；wParam = 0-向左顺序切换，1-向右顺序切换
EVT_CTL_SET_TITLE	控件事件	设置标题的消息接口，wParam = ID；dwParam = 资源文件（如果 ID != 0）或文本
EVT_CTL_SET_TEXT	控件事件	设置文本的消息接口，wParam = ID；dwParam = 资源文件（如果 ID != 0）或文本
EVT_DIALOG_INIT	对话框事件	对话框初始化事件，创建控件，预初始化值、标记和其它项，wParam = ID；dwParam = IDialog *
EVT_DIALOG_START	对话框事件	对话框打开事件，wParam = ID；dwParam = IDialog *
EVT_DIALOG_END	对话框事件	对话框正常结束事件，wParam = ID；dwParam = IDialog *
EVT_COPYRIGHT_END	对话框事件	版权对话框结束事件
EVT_ALARM	Shell 事件	闹钟事件，wParam 表示闹钟序号
EVT_NOTIFY	Shell 事件	通知事件，dwParam = AEENotify *
EVT_BUSY	Shell 事件	发送到应用程序以确定是否可以中止或停止该应用程序。
EVT_FLIP	设备事件	设备翻盖事件，wParam = TRUE（打开）；FALSE（关闭）
EVT_KEYGUARD	设备事件	键盘锁事件，wParam = TRUE（键盘锁定打开）
EVT_HEADSET	设备事件	耳机事件，wParam == TRUE（已插入耳机，否则为 FALSE）
EVT_PEN_DOWN	设备事件	触摸笔按下事件，dwParam= 点笔的位置： 高

		16 位表示 x 坐标，低 16 位表示 y 坐标
EVT_PEN_MOVE	设备事件	触摸笔移动事件，dwParam = 点笔的位置： 高 16 位表示 x 坐标，低 16 位表示 y 坐标
EVT_PEN_UP	设备事件	触摸笔抬起事件，dwParam = 点笔的位置： 高 16 位表示 x 坐标，低 16 位表示 y 坐标
EVT_PEN_STALE_MOVE	设备事件	触摸笔事件过多时发送，应用程序通常忽略这个事件。dwParam = 点笔的位置： 高 16 位表示 x 坐标，低 16 位表示 y 坐标
EVT_USER	用户事件	用户定义事件（应用程序私有）

除了上面的这些 BREW 系统预定义事件外，在 BREW SDK 中 AEE.h 文件中还定义了其他的一些事件，有兴趣的读者可以详细的查看这个文件。我们也可以定义属于我们自己应用程序独有的事件，这些事件通常以 EVT_USER 事件作为起始序号，这样可以避免与系统事件之间的冲突。

通常，一个事件处理函数的形式如下面的代码所示：

```
static boolean HandleEvent( IApplet *pi,
                            AEEEvent eCode,
                            uint16 wParam,
                            uint32 dwParam )
{
    MyAppTyp * pMe = (AEEApplet*)pi;

    // 决定如何处理收到的事件
    switch (eCode){
    // 应用程序启动
    case EVT_APP_START:
        // 进行相关的初始化操作
        return TRUE;

    // 挂起应用程序
    case EVT_APP_SUSPEND:
        // 保存数据用于恢复应用程序
        return TRUE;

    // 恢复应用程序到运行状态
    case EVT_APP_RESUME:
        // 恢复挂起时的应用程序状态
        return TRUE;

    // 应用程序关闭
    case EVT_APP_STOP:
        // 释放资源
        return TRUE;

    default:
```

```
            break;
        }
        return FALSE;
}
```

在事件处理函数中，使用一个 switch 语句来分发不同的事件，这样的程序既运行高效又划分的清楚。最简单的一个 BREW 应用程序是由两个函数组成的：AEEClsCreateInstance 和一个对应的事件捕获函数。这两个函数共同组成了应用程序的可执行区域，但是，应用程序的数据在什么地方呢？

BREW 平台与其它很多轻量级的应用程序平台一样，都不支持全局变量。这既是一个祝福也是一个诅咒。没有全局变量，我们的应用程序将易于调试和维护；然而，不幸的问题是如果没有全局变量，我们如何存放那些不需要通过函数堆栈传递的数据呢？（不支持全局变量也会引起其他的一些问题，如对于静态变量的管理，C 编译器将它同全局变量一样对待，因此也不能够在函数中使用静态的变量。因此我们必需注意在应用程序的任何部分都不能使用静态变量，否则即便可以在模拟器中正常运行，但是也不能在 BREW 设备上正常运行。）

做为一种替代全局变量的方法，我们可以定义一个包含应用程序变量的结构体，在这个结构体中我们可以保存应用程序的状态数据以及创建的接口实例指针等全局内容。在运行时为我们的这个结构体分配存储空间，这样在我们的应用程序看来，它就有了存储全局数据的地方了。BREW 平台就是这样做的，我们可以在 AEEClsCreateInstance 函数中调用 AEEApplet_New 的地方，指定我们自己的结构体来做为创建应用程序实例时的数据结构。为了能够这样做，在我们的应用程序接口体中定义的第一个成员变量必须是 AEEApplet 结构体，这样就相当于我们的结构体是 AEEApplet 结构的一个超集，只有这样才不会影响到正常的应用程序实例的创建。一旦 AEEApplet_New 执行完毕之后，就已经为我们的应用程序结构体分配了存储空间，并为第一个成员变量 AEEApplet 填充了数据。这样，我们就可以得到如下的一个简单的应用程序代码了：

```
typedef struct _MyAppTyp
{
    AEEApplet a;                              // 应用程序信息结构体
    uint32 m_launchTime, m_nEvents; //  应用程序指定的数据
} MyAppTyp;

int AEEClsCreateInstance( AEECLSID clsID,
                          IShell * pIShell,
                          IModule * po,
                          void ** ppObj )
{
    boolean result;
    *ppObj = NULL;

    //  如果 Class ID 符合我的应用程序...
    if( clsID == AEECLSID_MYCLASSID )
    {
        //  使用 BREW 助手函数创建应用程序实例
        result = AEEApplet_New( sizeof(MyAppTyp),
                                clsID,
```

```
                                        pIShell,
                                        po,
                                        (IApplet**)ppObj,
                                        (AEEHANDLER) HandleEvent,
                                        NULL );
    }
    return result ? AEE_SUCCESS : EFAILED;
}

static boolean HandleEvent( IApplet *pi,
                            AEEEvent eCode,
                            uint16 wParam,
                            uint32 dwParam )
{
    MyAppTyp *pMe = (MyAppTyp )pi;
    pMe->m_nEvents++;

    // 决定如何处理收到的事件
    switch (eCode){
    // 应用程序启动
    case EVT_APP_START:
        pMe->m_launchTime = GETTIMESECONDS();
        pMe->m_nEvents = 1;
        return TRUE;

    // 挂起应用程序
    case EVT_APP_SUSPEND:
        return TRUE;

    // 恢复应用程序到运行状态
    case EVT_APP_RESUME:
        return TRUE;

    // 应用程序关闭
    case EVT_APP_STOP:
        DBGPRINTF( "Application ran for %ld seconds.",
                   GETTIMESECONDS() - pMe->m_launchTime );
        DBGPRINTF( "Application received %ld events.",
                   pMe->m_nEvents );
        return TRUE;

    default:
        break;
    }
```

```
    return FALSE;
}
```

在这个例子里面，应用程序的结构体中，除了 AEEApplet 结构外，就包含了两个变量，用来记录起始运行时间和收到的事件数量。这个应用程序记录了接收的事件数量以及运行的时间，在应用程序退出之后通过 DBGPRINTF 函数在调试窗口中输出这些信息。与前面提供的函数不同的地方在于调用 AEEApplet_New 函数时分配空间的结构体从 AEEApplet 变为了 MyAppTyp 结构体。与此相对应的，HandleEvent 函数接收的结构体指针就属于 MyAppTyp 结构体的指针了。由于 AEEApplet 结构体定义在 MyAppTyp 结构体的顶端，因此 MyAppTyp 结构体指针可以安全的转换为 AEEApplet 结构体的指针。在应用程序中，由于实际上创建的是一个 MyAppTyp 的存储空间，因此通过 HandleEvent 函数第一个参数（IApplet *）传递进来的指针可以转换为 MyAppTyp 类型，这样我们就可以在应用程序中使用 MyAppTyp 中的数据成员了。

经过这种方式处理的应用程序中，既包含了程序的运行代码，又包含了程序的数据存储空间，由此组成了一个完整的可执行应用程序。

7.3.2 理解 BREW 接口

与大多数面向对象的平台一样，BREW 平台中的各种接口均继承自一个通用的接口。图 7.2 列举出了一部分 BREW 接口的继承关系：

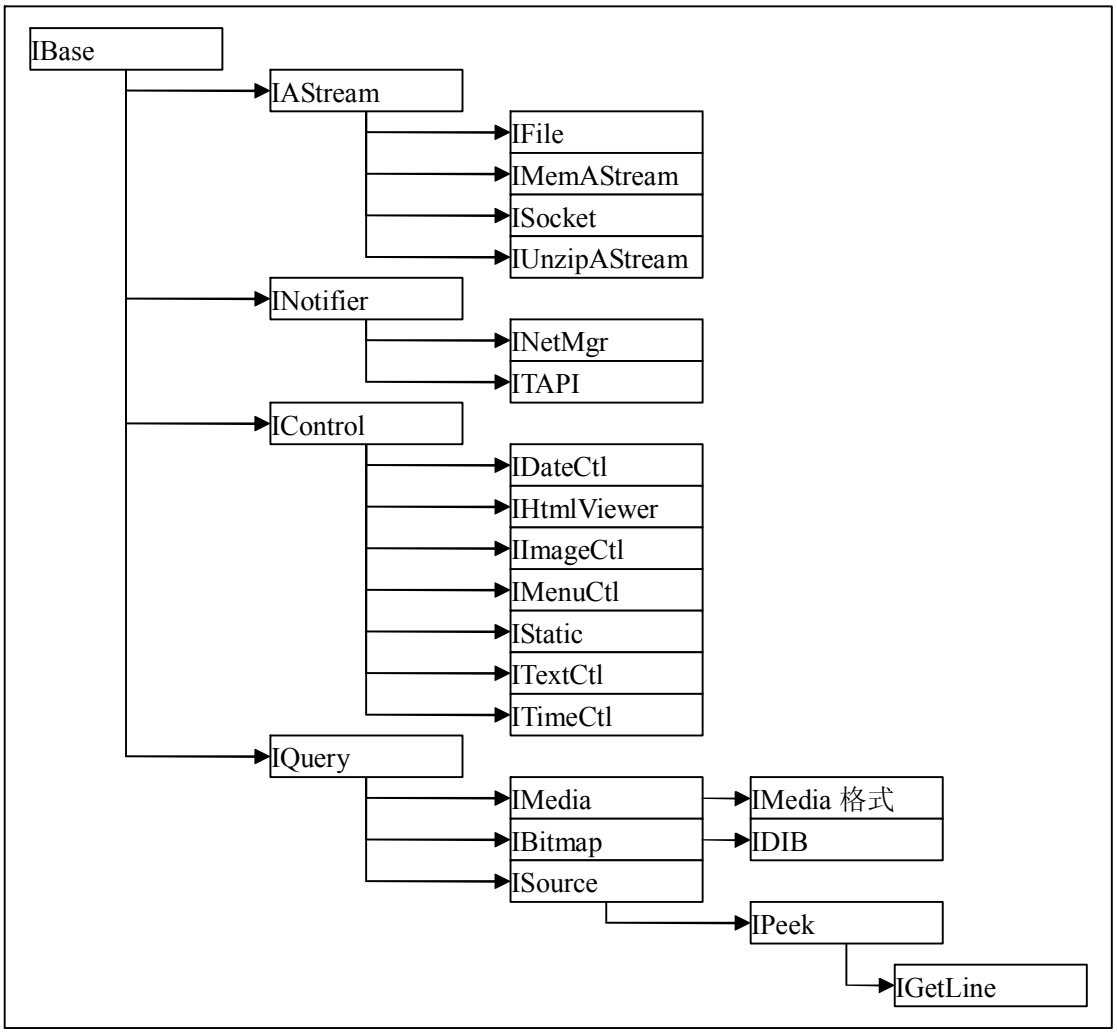

图 7.2 BREW 接口继承关系

图 7.2 中列出了 BREW 接口中一部分具有继承关系的接口，这些列出来的接口只是 BREW 众多接口中的一小部分，不过确是最常用的一部分接口。从图的左边到右边，按照箭头方向依次是从基类接口到派生接口。派生接口的实例可以调用其基类接口定义的方法进行调用。所有的 BREW 接口均继承自 IBase 接口，用来进行最为基本的资源管理。IBase 接口提供了两个方法 IBASE_AddRef 和 IBASE_Release。对于任何一个 BREW 接口的指针，我们都可以使用 IBASE_AddRef 接口来增加接口指针的引用计数；使用 IBASE_Release 接口减少接口指针引用计数，当接口的引用计数变为 0 之后，将释放这个接口所占用的资源。

通过这种接口之间的继承关系，我们可以编写一些针对某基类接口的控制函数，从而可以实现对多种派生接口的控制。例如我们可以编写一个使用数据流接口 IAStream 的函数，这样我们为这个函数传入 IFile 或者 ISocket 接口都是合法的。我们还可以编写一个所有控件的设置矩形显示框的公用函数，在这个函数中我们可以传入任何一个继承自 IControl 的接口指针，然后调用 ICONTROL_SetTect 方法来设置控件的显示区域。

最常用的一个 BREW 接口是 IShell 接口，它提供了一组系统级的方法，包括创建一个接口的实例等功能。例如，我们可以使用 ISHELL_CreateInstance 接口创建一个 IDisplay 接口指针的实例：

```
ISHELL_CreateInstance( pIShell,
                       AEECLSID_DISPLAY,
                       (void **) &pme->m_pIDisplay);
```

如果我们花一些时间去浏览每一个 BREW 头文件的话，可以发现 pIShell 代表了一个 IShell 接口的指针，而 pme->m_pIDisplay 则是一个 IDisplay 类型的接口指针，通过在这个变量前面加上取址运算符“&”作为参数传入，我们就可以获得一个 IDisplay 型的接口指针了。最后，我们就可以通过这个接口指针使用 IDisplay 接口中提供的各种方法了。

除了 ISHELL_CreateInstance 接口外，IShell 接口中还提供了各种不同作用的接口，如设置闹钟和定时器的接口，启动和关闭应用程序的接口，创建和关闭对话框的接口等等。您可以查看 BREW 的 API 文档查看其中的每一个接口，或者在本书的下一篇中您也能看见这些不同种类 API 的作用以及它们可能的内部机理（尽我所能的为您剖析）。

除了 IShell 等接口之外，BREW 还提供了一组助手函数，它们中有很多的实现是为了替换 C 标准库的函数。幸运的是，我们不需要为如何使用它们而感到担心，因为通常它们的形式与标准的 C 语言库相比就是大写的而已。一些常用的函数对比如下：

BREW 助手函数	C 语言标准库
ATOI	atoi
DBGPRINTF	printf
FREE	free
MALLOC	malloc
MEMCMP	memcmp
MEMCPY	memcpy
MEMMOVE	memmove
MEMSET	memset
REALLOC	realloc
SPRINTF	sprintf
STRCAT	strcat
STRCHR	strchr
STRCMP	strcmp

STRCPY	strcpy
STRDUP	strdup
STRLEN	strlen
STRNCPY	strncpy
STRSTR	strstr

还有很多属于 BREW 专有的一些助手函数，不过这里没有列举出来，因为它们实在是太多了。由于 BREW 的字符串处理是以 Unicode 为基础的，因此除了 ASCII 字符串助手函数之外，它还提供了一组处理宽字节的字符串处理函数。对应于单字节的助手函数，这些宽字节的助手函数只需要增加一个 W 就可以了，如 WSTRCPY 代表宽字节字符的复制函数。在编写程序时，请确认我们的应用程序中使用的是 BREW 助手函数，而不是 C 语言的库函数，否则可能引起应用程序在 BREW 设备上的运行错误或者无端的增加程序的大小。

对于那些已经学习过面向对象（如 SmallTalk，C++等）的人来说，需要注意的是 BREW 没有那些语言的一些特征。例如，BREW 中所有的数据类型仅仅是数据类型而已，而不是一个对象。请不要用我们固有的面向对象的思维方式去理解 BREW，否则会给我们的编码带来一些麻烦，尤其是在继承这个问题上，BREW 的继承不是在语法上的，而是在二进制层面的一种继承。关于这一点在后面的章节中将会有详细的叙述，现在唯一需要注意的是不要用我们以前的知识去理解 BREW 的面向对象方式。

7.4 指定语言的资源文件

无论是 BREW 的系统资源文件，还是 BREW 应用程序的资源文件，都根据不同的语言放置在不同的目录下面。例如，英文的资源文件放在一个叫做 en 的目录下，简体中文的资源文件放在一个叫做“zhcn”的目录下面。这些目录的命名遵从 ISO639 的语言名称定义。在 BREW SDK 的 AEELngCode.h 头文件中，定义了全部的语言目录名称，有兴趣的读者可以看看这个文件。

在 BREW 运行设备上，BREW 会根据设备指定的不同语言，在运行应用程序时到不同的目录下载入资源文件。正是 BREW 的这种处理方式，使得 BREW 可以轻松的开发多语言的不同语言版本。同时，这也是 BREW 的一大特色。

资源文件中可以添加字符串等资源，可以使用 IShell 接口的资源文件相关函数载入资源文件，例如可以使用 ISHELL_LoadResString(IShell * pIShell, const char * pszResFile, int16 nResID, AECHAR * pBuff, int nSize) 接口函数从指定的资源文件中载入字符串。具体各种接口函数的使用方法请参考 BREW API 的说明文档。

7.5 为 BREW 设备生成应用程序

如果需要使我们的应用程序可以在 BREW 设备上运行，那么，就必须有相应设备上的 C/C++编译器。当前的 BREW 设备都是运行在 ARM CPU 上的，因此这就需要 ARM 编译器 ADS（ARM Development Suite）1.0.1、ADS1.2 或者是 BREW Builder。还可以使用 GNU 编译器（GCC）2.95。

7.5.1 ARM 开发工具集（ADS）

ADS 是 ARM 公司针对 ARM 内核 CPU 的 C/C++语言编译链接的工具集和，目前 BREW 平台都是运行在 ARM 内核的 CPU 上的，因此 BREW 的动态应用程序也是使用 ARM 编译器编译的。当前 ADS 的最高版本是 ADS1.2。在 BREW 的开发向导中，提供了对 ARM 编译环境的支持，可以直接在 Visual Studio 中生成目标二进制文件（.mod）。要获得此工具，需要从 ARM 公司购买。关于 ADS 的详细信息请参考网站http://www.arm.com上的信息。

7.5.2 BREW Builder

通过高通公司与 ARM 公司联系，ARM 公司为开发者提供了一套专门针对 BREW 的 ARM 编译器，它的软件购买费用大约是 ADS 的三分之一。在这个软件中包含了编译器、链接器和汇编工具，其中不包括 ADS 的集成开发环境和库文件管理工具，也不包括浮点运算的功能，同时提供的帮助文档也是 PDF 格式的，而不是 ARM 的 DynaText 格式。它使用的编译环境与 ADS 是一样的，可以在 Visual Studio 中直接使用。当前 BREW Builder 的名称是 Realview Compilation Tools for BREW。

7.5.3 GNU 编译器

除了 ADS 和 BREW Builder 之外，我们还可以使用 GCC 编译器来编译 BREW 应用程序，而且这个软件是免费的。虽然使用 GCC 编译工具不是 BREW 官方推荐的做法（不推荐大多数是因为商业利益的关系），但是高通公司还是提供了相关的工具，它们主要有：

1、make 工具，用来管理源代码工程。

2、Cygwin 工具，用来在 Windows 环境下模拟 Unix 命令。

3、GCC 编译器，用来编译和链接源代码。

4、其他，主要是用来解决 GCC 编译中库文件链接问题的相关文件。

关于这些工具的详细信息和使用方法，可以到高通的网站上去查看详细的信息。如果使用 GNU 编译器，那么相关的编译的 Make File 就需要自己动手编写了，它没有集成到 Visual Studio 的环境中去。当然，高通公司也提供了示例文件可以供您参考。

7.5.4 各种编译工具的选择

前面介绍的三种工具都可以作为 BREW 目标设备的编译工具，我们可以结合自己的实际情况进行安装。ADS 适合于那些开发 BREW 扩展功能和进行 BREW 移植的厂商，通过它我们可以使用全部 ARM 编译器的功能。如果我们只是作为动态应用程序的开发者，那么我们使用 BREW Builder 或者 GCC 就足够了，它们的功能已经可以满足要求了。GCC 属于开放源码组织的工具，因此是免费的，对于不希望为此付费的开发者来说最适合不过了，只不过需要自己搭建编译环境，并书写 Make File。

选择完工具之后，我们就可以根据所选择的方式生成可以在 BREW 设备上安装并运行的应用程序了。

7.5.5 BREW 文件类型和动态应用程序的安装

BREW 应用程序由 MIF 文件(.mif)、资源文件(.bar)、签名文件(.sig)和模块文件(.mod)组成。签名文件是 BREW 系统用来验证当前应用程序是否具有在本机运行权限的文件，在通过空中接口的网络下载安装时，签名文件是由 ADS 服务器提供下载的。但是在应用程序开发测试过程中，BREW 提供了一种测试模式。在这种测试模式下，应用程序可以根据 BREW 设备的 ESN 号码，从高通网站上申请一个签名文件。同时打开 BREW 设备的测试模式（通常在 BREW 设备中都会提供一个隐藏菜单，在此菜单中可以进行相关功能的设置），就可以使用这个签名文件在 BREW 设备中运行程序了。模拟器中不需要使用签名文件。

在安装 BREW 动态应用程序时，将这些文件安装在 BREW 设备的不同目录下。BREW 对安装应用程序的路径有明确的要求，从 BREW3.0 开始的 BREW 设备目录结构如下：

路径名	描述
fs:/mif	存储 MIF 文件的文件夹
fs:/sys	存储 BREW 系统文件的地方，如系统配置文件、系统资源文件等
fs:/mod	存储应用程序文件夹及文件的地方。在这个文件夹中，使以各个 MIF 文件名相同名称的文件夹以及 MOD 文件和签名文件等
fs:/shared	存储公共文件的地方

假设现在有应用程序 TestA，那么它的组成文件可能会是如下的结构：

fs:/mif/testa.mif

fs:/mod/testa/testa.mod

fs:/mod/testa/testa.sig

fs:/mod/testa/testa.bar

fs:/mod/testa/public.bar

在 BREW3.0 版本之前，这个目录结构如下：

/testa.mif

/testa/testa.mod

/testa/testa.sig

/testa/testa.bar

/testa/public.bar

也就是说，在 BREW3.0 版本之前，所有的 MIF 文件都是放在 BREW 文件系统的根目录下面的，各个应用程序所在的路径也是在根目录下的。注意这里面的 public.bar 文件，为了避免进入“资源文件名称也与 MIF 文件名相同的误区”，我特别的加入这个文件来告诉您：资源文件的命名没有特殊限制（除了要要遵守文件系统的约定），而且同一个应用程序中也可以有多个资源文件。

7.6 使用 BREW 工具

BREW 给应用程序开发者提供了多种不同的工具，用来编辑或创建不同种类的 BREW 要素。通过这些工具的协同使用，使得我们可以很容易的开发出 BREW 的应用程序。这一节我们就来详细的介绍每种工具的使用方法。

7.6.1 使用 MIF 编辑器

每一个 BREW 应用程序（准确地说是每一个模块）都有对应的 MIF 文件，用来描述该模块中每一个应用程序的信息，如 Class ID、图标、名称等等，MIF 编辑器就是用来编辑 MIF 文件中这些内容的。在 BREW MIF 编辑器中可以创建 MFX 和 MIF 文件，MFX 是开发 MIF 过程中使用的 XML 格式的中间文件。MIF 是一种从 MFX 文件编译而成的特殊类型的 BREW 资源文件，其中包含有关 BREW 模块 (MOD)文件内容的信息。MIF 创建之后，将以二进制形式提交并加载到目标设备。在模拟器上运行应用程序时，也可以使用 MIF。大多数情况下，我们只需要使用 Applets 和 General 两个选项卡。Applets 选项卡如图 7.3 所示。

通过这个 Applets 选项卡，我们可以在模块中增加应用程序，包括设置通知信息、设置标记（Flags）、以及显示信息等。添加一个 Applet 信息时，需要指定 Class ID 所在的 BID 文件。BID 文件可以本地生成或者从高通的网站上获得。由于每个模块中可以包含多个应用程序，因此需要多个 BID 文件，并且这些 BID 文件中包含的 Class ID 不能够相同。一旦 Applet 的入口被创建，那我们就可以编辑这个 Applet 的信息了。我们可以设置这个应用的图标，图标分为大、中、小三种，分别对应了在应用程序管理器中的不同菜单模式下的显示图标。BREW 模拟器默认的界面就是应用程序管理器了，当我们设置了这些图标之后我们可以通过改变设置而看到不同的画面。指定的图片格式可以示 BMP、PNG 或 JPEG。我们还可以指定应用程序的类型，如果我们选择的类型是“Hidden”，那么这个应用程序的图标将不会出现在应用程序管理器中。如果我们使用 BREW 开发设备的用户界面的话，那么这些应用程序的类型都应改选择“Hidden”，以使他们不会在设备上的应用程序管理器中显示出来。

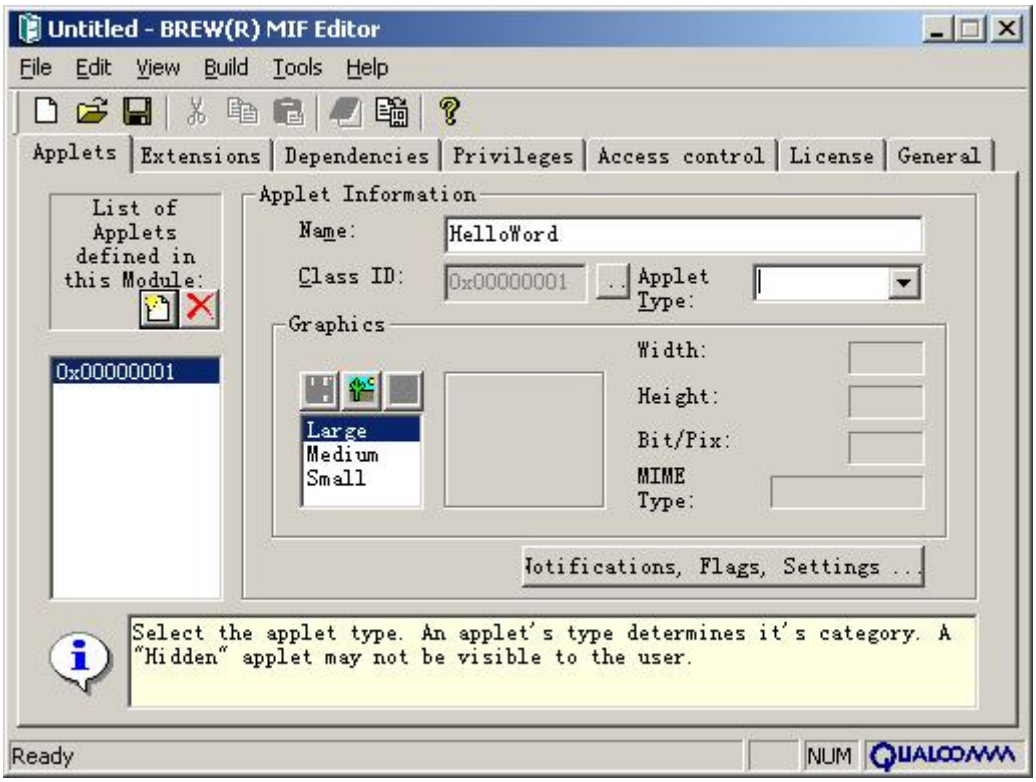

图 7.3Applets 选项卡

我们还可以选择高级设置按钮，进入 Applet 高级设置对话框。在这个对话框中我们可以设置应用程序的标志（Flags）、接收的通知事件和显示信息等。其中各种标志的描述如下：

标志	描述

Popup	此项标志描述当载入应用程序时不清空设备屏幕。如果没有选中此选项，那么当载入应用程序时将首先清空当前的屏幕。如果我们的应用程序在启动时出现白屏，那么可以选择此项解决问题。
Screensaver	此项表示当前的应用程序属于一个屏幕保护的应用程序，这将导致此应用程序出现在屏幕保护程序的列表中。
Phone	此标志表示将 BREW 设定的关闭全部应用程序的按键事件（通常是 End 键）像其他正常的按键一样传递给此应用程序（仅在此应用程序处于激活状态时）。此标志只有在应用程序具有系统级别的访问权限时才有效。
Show On Config Menu	此选项表示是否将此应用程序在 BREW 设置菜单中显示，显示的内容是在下面的“Menu string”输入框中输入的内容。
Show On Hidden Config Menu	此选项表示是否将此应用程序在 BREW 隐藏设置菜单中显示，显示的内容是在下面的“Menu string”输入框中输入的内容。

有些BREW 设备支持配置或设置菜单，允许设备用户为每个 BREW 小程序设置首选项。例如，游戏用户可以从首选菜单中选择游戏难度等级。我们可以在 MIF 中指定配置菜单上是否显示小程序。如果我们的小程序包括在这些菜单中，从配置菜单或隐藏配置菜单中选择时，就向它发送 EVT_APP_CONFIG 或 EVT_APP_HIDDEN_CONFIG 事件。小程序收到这些事件后，应该显示一个设置菜单以便于用户设置参数。模拟器不支持这些配置菜单。这些配置菜单目前很少使用，因此很少有应用程序会选择这个选项。

使用高级设置对话框还可以设置应用程序接收的通知事件。通知的来源有 AEECLSID_NET、AEECLSID_SHELL 和 AEECLSID_TAPI 三个类。在“掩码”字段的下拉列表中选择掩码，可以确定给类发送哪些通知。此下拉列表中的选项将随上面所选的系统类而变化。如果通知是由非 BREW 标准类提供的，则可以选择自定义，然后输入 Class ID。在“掩码”字段中，输入掩码的编号，可以确定给类发送哪些通知。掩码 0xFFFFFFFF 指定发送全部通知；掩码 0x00000000 则指定不发送通知。通知及可以在 MIF 文件中指定，也可以使用 BREW 接口来指定，在后面的 Shell 功能介绍里会有介绍。

每一个应用程序还可以设置显示的信息，通常不必使用这些内容，因此这里就不做详细的介绍了，有兴趣的读者可以参考 BREW 的帮助文档。

另一个常用的选项卡就是 General 选项卡了，General 选项卡如图 7.4 所示。

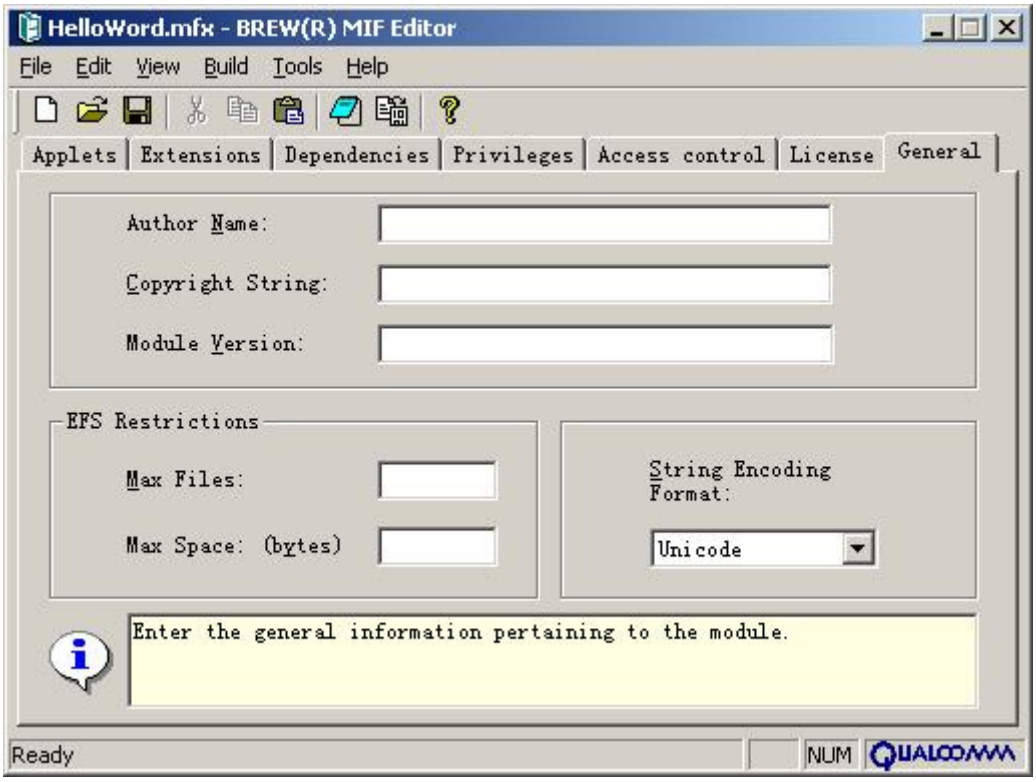

图 7.4 General 选项卡

General 选项卡中可以添加模块的版权和版本信息，以及访问文件系统的限制和 MIF 文件中的字符串格式。这些内容都是模块的常规信息。最大文件数可以输入从 7 到 65535 之间的数字，代表指定模块可以创建文件的总数， 它包括在模块目录以及共享目录中创建的文件和目录；最大空间中可以指定的最小空间为 20,480 字节，最大为 4294967294 字节，代表模块可以占用的文件空间总量，它包括模块目录和共享目录中的文件空间。字符串格式是指存储在此 MIF 文件中的字符串（如此选项卡中的作者名称和 Applets 选项卡中的 Applet 名称等）编码格式。关于编码格式我们会在后面的章节中有详细说明。

Extensions 选项卡中是设置此模块中导出类的地方。导出类是模块中所有非 Applet 的 Class ID。同 BREW MIF 编辑器窗口中的“Applets”选项卡一样，我们可以从 BREW Class ID 网页，或通过选择计算机上现有的 BID 文件获取非 Applet 类的 ClassID。MIF 可以使用两种类型的导出类：非保护和受保护。非保护导出类可供给任何应用程序随时使用。受保护导出类则必须明确声明为该 MIF 的外部依存（也就是在该 MIF 文件的 Dependencies 选项卡中增加使用该受保护导出类的类 Class ID）。如果将受保护类指定给 MIF 而没有将其声明为外部依存，则将返回 EPRIVLEVEL 错误。虽然非保护类没有被明确声明为类的外部依存也可以使用，但这会导致无法跟踪 BREW 中的“使用模块”，引起此导出类的使用问题。我们可以考虑如下情况，如果使用了此非保护导出类的类，没有声明为此模块的外部依存，那么将有可能发生在使用此非保护导出类的类不知情的情况下删除了导出类所在的模块，从而导致使用此导出类失败。因此，模块必须始终声明受保护类和非保护类的外部依存。

Extensions 选项卡还可以设置导出的 MIME 类型。导出 MIME 类型是模块中的非 Applet 类，执行该类可以处理特定 MIME 类型的文件（例如 SND 声音文件和 MIDI 文件）。对于 MIME 处理程序类，我们可以指定该类所处理的 MIME 类型。其它类可以使用 IShell 接口的 GetHandler 函数获取处理程序类的实例，该处理程序类以它处理的 MIME 类型命名。这个功能就类似于我们在 Windows 中使用特定的应用程序处理特定文件一样。

Dependencies 选项卡用来设置模块的外部依存。外部依存的含义是依赖或使用此模块的

外部 Class ID。这里面我们容易进入的一个误区是，认为外部依存是此模块所依赖或使用的其它模块的类，而事实上却是恰恰相反的，因此请各位读者提起十二分精神注意此事。在此选项卡左边是“Available”的 Class ID 列表，也就是可用的 Class ID，但是在这个列表中的 Class ID 还没有成为此模块的外部依存，我们需要使用“Add”按钮将我们希望作为外部依存的 Class ID 转移到右边的“Used”列表中。

Privileges 选项卡用来设置模块可以使用各种系统资源的权限，这些权限包括：

名称	描述
File	IFile、IDBMgr、IDatabase 和 IDBRecord 接口的文件和数据库函数。
Network	INetwork 和 ISocket 接口的网络和套接字函数，用以设置 TCP 和 UDP 套接字。
Web Access	访问 IWeb 接口。
TAPI	ITAPI 接口中的 TAPI 函数。
Position Location	IPOSDET_SetGPSConfig、IPOSDET_GetGPSConfig 和 IPOSDET_GetGPSInfo 函数，提供基于 GPS 的定位信息。ISHELL_GetPosition 函数，提供对设备定位功能的访问。
Access To Ringer Directory	BREW SDK 和手持设备上存储音质文件的目录的写入权限。 在运行 IRINGER_Create() 函数时，将自动创建该目录。
Write Access To Shared Directory	BREW SDK 和手持设备上 <BREW\sdk\examples\Shared> 目录的写入权限。
Access To Sector Information	访问从 IPOSDET_GetSectorInfo API 获取的扇区信息。
Access To Address Book	访问 IAddrBook 接口

在“Advanced Privileges”对话框中，可以确定模块是否可以访问某些 BREW 系统函数，它们仅限运营商和设备制造商使用，BREW 应用程序开发者不能使用。Download，指示可以访问 IDownload 接口中的函数，这些函数用于从运营商网站下载 BREW 应用程序。All/System，指示可以访问所有 BREW 系统函数。

Access Control 选项卡用来设置模块的 ACL（Access Control Level），关于 ACL 的详细信息可以参考第十章数据存储功能中关于 ACL 的描述。License 选项卡用来设置该模块的许可证选项，此选项仅在进行应用程序测试的时候才有效，因此大多数情况下我们可以不予理会。

上面的各个选项卡填写完毕之后，我们就可以保存这个 MIF 文件了。保存时可以选择保存的类型：v1、v2、MFX 文件。v1 版本用于 BREW3.0 之前的 BREW 平台，v2 版本用于 BREW3.0 版本及其之后的版本，我们可以根据实际情况保存不同的格式。MFX 文件最终也需要编译成上面的两种二进制形式才能使用。

7.6.2 使用资源文件编辑器

BREW 资源编辑器允许我们创建应用程序中使用的对话框、字符串、二进制数据以及文件对象。我们还可以使用资源编辑器创建控件，如：菜单、列表、日期选择器以及计时器等。 如果创建的应用程序需要在不同语言的 BREW 设备上运行，这种资源文件的管理方式将十分有用。资源文件编辑器的操作界面如图 7.5 所示。

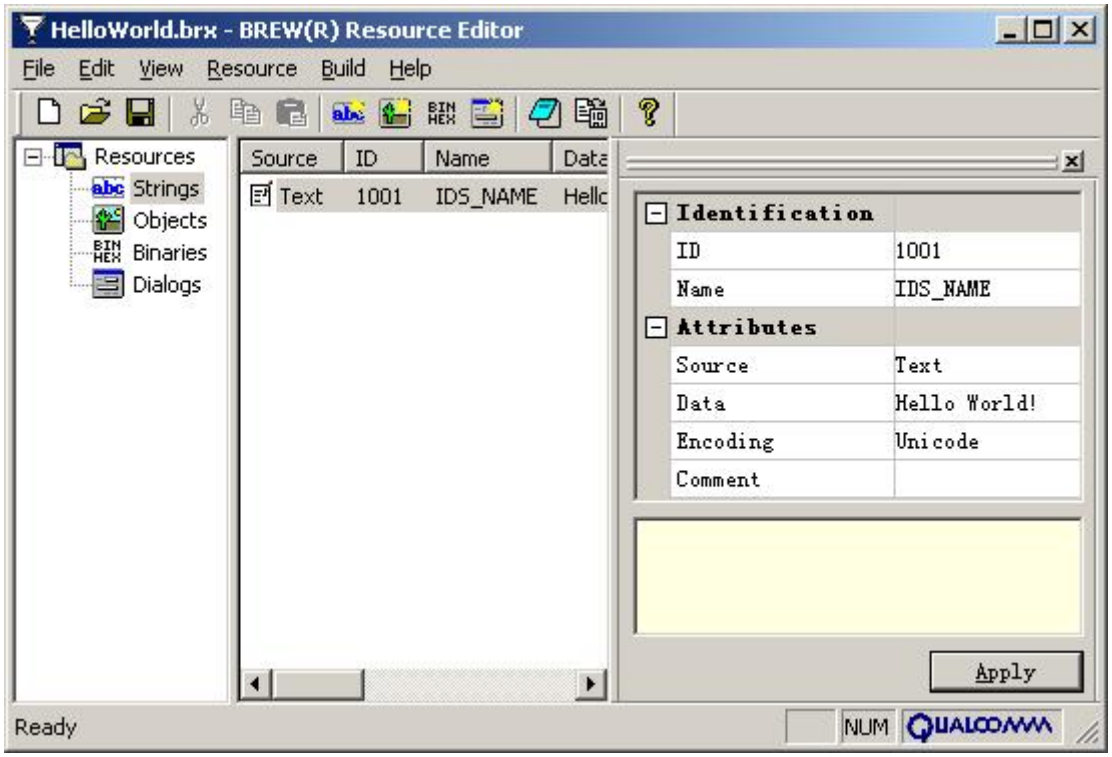

图 7.5 资源文件编辑器

字符串资源元素是一个字符数组，这些字符可以是 Unicode、ISOLATIN 1、KSC5601、S-JIS 或 GB2312。默认类型为 Unicode。通过将小程序使用的全部字符串保存在资源文件中，我们可以轻松地针对不同国家/地区本地化小程序。对象资源可以是各种不同的格式或类型，但一定具有 MIME 类型。对象资源通常为图形图像。对象资源的另一个常见用法是在应用程序资源中嵌入 HTML 文件。使用这种用法时，数据应采用 ASCII 格式且 MIME 类型应设置为"text/html"。当然我们还可以直接存储二进制数据。

对话框资源由设备屏幕上显示的一个或多个 BREW 控件组成。应用程序可以定义多个对话框接口，引导用户在一系列要求输入信息的对话框中完成输入。BREW 应用程序使用 IShell 接口的 CreateDialog 函数从资源文件中加载对话框，并在屏幕上显示其控件。加载对话框之后，可使用 IDateCtl、IMenuCtl、ITextCtl 和 ITimeCtl 接口函数修改其控件的外观和行为、获取设备用户在各个控件中输入或选择的数据。IShell 的 EndDialog 函数可终止对话框，并在设备屏幕上显示之前激活的对话框（如果有）。

在我们建立了资源文件之后，我们需要将资源文件编译成二进制的（.bar）文件，同时还会生成资源文件 ID 定义的一个头文件。通过这个头文件中的 ID，使得我们可以在应用程序中使用相应资源。与 MIF 文件不同的是，只要我们喜欢，一个应用程序中可以使用多个资源文件。而且命名也没有特殊的要求，可以使用任意我们自己喜欢的名称。

7.6.3 使用 BREW 模拟器

BREW 模拟器用于模拟选定的 BREW 设备，使得我们可以加载 BREW 环境下开发的测试小程序和类。模拟的 BREW 设备可以使用各种屏幕、字体、键盘、可用内存量、支持的语言和其它参数。在模拟过程中，模拟器将在 PC 显示器上打开设备的图像。通过点击对应设备按键的图像区域，可以对要模拟的 Applet 提供按键输入，同时 Applet 生成屏幕输出显示在设备图像的屏幕区域。BREW 模拟器还可以通过鼠标事件模拟触摸屏设备所产生的

EVT_PEN_DOWN、EVT_PEN_MOVE 和 EVT_PEN_UP 事件。通过这些事件可以实现应用程序的屏幕点击操作。BREW 模拟器的效果如图 5.5 所示（在第五章）。

正是如图 5.5 所展示的一样，BREW 模拟器为我们提供了一个图形化的模拟界面。我们可以通过鼠标点击模拟设备上的按键与应用程序之间交互，也可以通过鼠标点击屏幕来模拟触摸屏事件。BREW 模拟器可以通过不同的皮肤模拟不同的设备，在这里我们就称之为模拟设备吧。模拟设备文件使用 BREW 设备文件编辑器创建和编辑的，可以通过模拟器“File->Load Device”菜单来载入不同的模拟设备。

应用程序管理器会在小程序目录中搜索模拟器中显示的小程序。 默认情况下，模拟器在 <BREW\sdk\examples> 目录下查找小程序，但是我们可以通过“File->Change Applet Dir…”菜单选项更改该目录。默认情况下，MIF 文件目录和 Applet 目录是相同的。如果我们希望设置一个单独的 MIF 文件目录，我们可以通过“Tools->Setting”菜单来进行设置。使用该菜单选项还可以指定是否激活堆验证以及默认的 DNS（Domain Name Server）服务器。堆验证（Heap Validation）的目的是检测在应用程序退出之后是否存在未被释放的已分配内存空间，这样的机制可以保证我们的小程序中不会存在内存泄露。DNS 服务器主要是提供给模拟器一个单独的 DNS 服务器选项，如果没有设置 DNS，将使用当前 PC 机的设置。当打开 Setting 对话框的时候，模拟器会向当前正在运行的 Applet 发送 EVT_APP_SUSPEND 事件，关闭对话框的时候会发送 EVT_APP_RESUME 事件，因此通过这个对话框可以模拟应用程序的挂起和恢复的操作。

除了上面的功能之外，BREW 模拟器还可以模拟 TAPI、SMS、GPS 功能的数据输入，使得我们可以通过模拟器开发这些功能。当前模拟器的缺点是不能够模拟 UIM 卡部分的处理和操作。关于模拟器其他的一些功能和设置请参考相应的帮助文档。

7.6.4 使用设备文件编辑器

BREW 模拟器通过模拟设备文件来模拟真实的应用程序运行的硬件平台和软件平台环境，之所以能够实现这种模拟方式的关键在于设备文件的存在。而编辑设备文件的工具就是 BREW Device Configuratior，也就是设备文件编辑器。通常情况下设备文件编辑器只对 OEM 用户提供安装和下载，设备文件则是通过 OEM 设备生产厂商提供给 BREW 应用程序开发者。这样可以保证每个模拟设备的信息是准确的。

通过 BREW 设备文件编辑器我们可以模拟设备的按键、屏幕大小等许多信息。为了能够构建一个模拟设备文件，需要提供两张设备的图片，一张做为前景设备图片，另一张做为按键按下时的效果图。通常这两张的图片是同一张图片，但是有不同的偏移，这样就会有一种按下去的效果。可以在前景图上标记不同的区域，按键或者屏幕。然后通过对不同区域的属性设置，设备模拟工作就算初步完成了。然后，需要填写设备相信信息的属性表格，其中包括设备制造厂商，显示屏信息，内存信息等多种设备规格，通过这些信息我们可以了解整个设备的情况。设备文件编辑器的效果如图 7.6 所示。

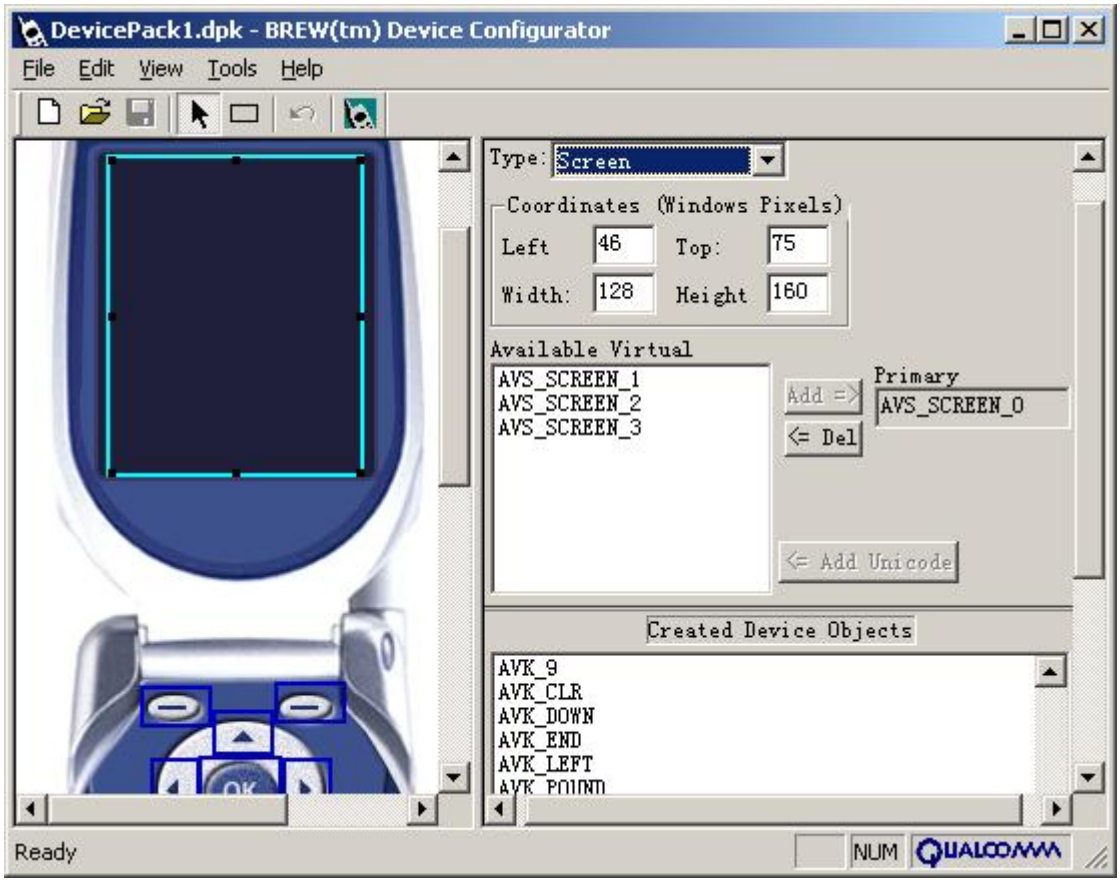

图 7.6 设备文件编辑器

7.6.5 使用应用程序下载器

在模拟器中开发完成应用程序并生成可以在设备上运行的二进制（.mod）文件之后，我们需要一种能够将应用程序下载到 BREW 设备中去测试的工具。这个工具就是 BREW Application Loader，也就是应用程序下载器。

为了使用 BREW 应用程序下载器，我们需要首先从高通网站获取一个签名文件，这个签名文件通常以（.sig）作为扩展名。当我们安装小程序的时候，我们将需要将 MIF 文件、签名文件、资源文件和模块二进制文件复制到 BREW 设备中去。可以按照如下步骤进行：

1、使用 BREW 设备的数据线连接到 PC 机的一个未使用的端口上。这个端口可能是 COM 口，也可以是 USB 口，这与 BREW 设备有关。

2、载入 BREW Application Loader 应用程序。

3、选择在步骤一中指定的串口连接到 BREW 设备。

BREW 设备和 PC 机之间将通过特定的通信协议进行通信，然后我们将看到 BREW 设备内部的目录结构，我们可以通过 BREW 应用程序下载器打开、创建目录，而且可以将设备中的文件复制出来。在下载我们的应用程序的时候，我们需要将应用程序的 MIF 文件复制到文件系统的根目录下，然后将其他文件复制到与 MIF 文件名同名的目录下面，如果这个目录不存在，我们需要去创建它。我们也可以指定一次性的下载模块的全部内容，这样就不需要我们手动的复制文件了。由于这个工具的通信协议属于高通特有的格式，因此目前只有基于高通的 BREW 设备平台才能使用此工具。如果我们的 BREW 设备不是使用高通芯片的，那么我们可能需要从该 BREW 设备的制造商那里获得相应的应用程序下载工具。

7.6.6 使用记录器（Logger）获取调试信息

与 BREW 应用程序下载工具一样，BREW 记录器也是一个 BREW 设备与 PC 连接的工具。BREW 记录器的作用是获取在 BREW 设备中运行的应用程序的调试信息，这些调试信息是通过 BREW 助手函数 DBGPRINTF 输出的。通过 BREW 记录器我们可以捕获应用程序在运行时所输出的调试信息，这些信息在模拟器环境下开发时，也会从 Visual Studio 的输出窗口中输出。这些输出的信息会以记录的形式出现在屏幕上，您可以保存这些信息以便于分析问题。

通过在应用程序中添加适当的调试信息，可以很方便的跟踪处理应用程序在 BREW 设备中遇到的情况，记录遇到的问题，使得我们可以更加容易的分析解决问题。

使用 BREW Logger 的方法与使用 BREW AppLoader 的方法几乎相同，也是需要选择设备所在的端口，打开之后开始调试。由于这些工具需要获得高通公司的授权之后才能够下载使用，因此，这里就不详细的介绍他们的使用方法了。添加这些说明是为了让您了解 BREW 提供了这些方便的工具，有需要的时候您可以去获得这些工具。

7.7 小结

在本章中，我们建立了第一个 BREW 应用程序，从中我们了解到了建立 BREW 应用程序的方法和注意事项，随后我们分析了 BREW 应用程序的组成部件，最关键的两个函数是入口函数 AEEClsCreateInstance 和事件捕获函数。与此同时我们分析了这些关键点之间的关系，并演示了如何生成一个拥有自己数据类型的 BREW 应用程序的方法。接下来，我们讲述了为 BREW 设备生成应用程序的方法，以及 BREW 提供的各种工具的使用。通过对这些 BREW 工具的讲述，我们可以初步的使用这些工具进行 BREW 开发了。

思考题

1、BREW 事件捕获函数 HandleEvent 的第一个参数是（IApplet *）类型的，我们可否将其改为（MyAppTyp *）类型的？为什么？

2、BREW MIF 编辑器中，输出 Class ID 表示什么意思？

第八章 BREW 的事件处理

在上一章里，我们创建了一个叫做 HelloWorld 的应用程序，是的，Hello BREW，我们已经进入到了你的世界！在领略了基本的 BREW 应用程序之后，我们将继续进发，去仔细的看一看应用程序的每一个核心的东西。

BREW 应用程序最为基本的内容就是它的事件驱动和处理的机制，通过这样的机制，我们可以使用相对简单的思路开发出我们所需要的应用程序。基于事件驱动机制开发的好处就是我们可以把复杂的程序容易的分割成各个小模块程序，同时针对事件驱动机制的特点我们还可以实现一个状态机的结构，这样的程序框架对于开发大型的应用程序来说是十分的方便的。

在这一章里面，我们将详细的剖析一下 BREW 的事件处理的方式，以及我们的应用程序是如何捕获（Handle）和处理它们的。同时，根据这种方式的特点我们还将构建一个状态机的应用程序框架，而且这个框架将有利于我们今后应用程序的开发。

8.1 理解事件驱动模型

做为 BREW 平台最为核心的一块，BREW 的事件驱动模型是我们开发 BREW 应用程序必须要提前理解和消化的。这也是我们开发应用程序的基本的一步。

8.1.1 处理一个事件

对于一个 BREW 应用程序来说，不管愿不愿意处理这个事件，都将收到来自系统或使用者的事件，并且将这些事件传递到应用程序中的控件或者接口当中。正如前面章节所描述的一样，BREW 通过应用程序的 HandleEvent 函数将事件传递给应用程序，应用程序处理这些事件并且通过函数的返回值告知系统这个事件的处理情况。

与大多数的操作系统不同，我们的应用程序不需要采用的轮询的方式来查看是否有事件传递过来，BREW 系统将会根据相关的事件主动的调用应用程序的 HandleEvent 函数将事件传递给我们的应用程序。这个区别是至关重要的，因为我们的应用程序在捕获一个事件的时候，必须进行尽可能简单的处理。不像一些大型的操作系统平台，如 Windows CE 和其他的嵌入式系统操作系统，BREW 运行在一个单线程的环境，因此，我们必须尽可能快地处理每一个事件，这样才不至于影响系统的正常运行。否则，应用程序的表象可能就会是响应用户非常迟钝，这样的应用程序是用户不会喜欢的。

基于这样的一个 BREW 事件驱动环境，要求我们的应用程序及时处理事件。这意味着应用程序应该迅速处理事件并立即返回。BREW 将事件传递给应用程序时，应用程序会通过返回 TRUE（表示已处理）或 FALSE（表示未处理）来指示应用程序是否已处理事件。如果小程序必须将事件传递给其它事件处理程序（如控件或接口），它只需返回调用这个接口 HandleEvent 函数的结果。

BREW 应用程序的事件处理（HandleEvent）函数接受三种与事件相关参数的输入，它们分别作为 HandleEvent 函数的第二、第三和第四个参数传递。下面是一个事件处理程序函数的示例：

```
boolean HandleEvent(MyApp* pMe, AEEEvent eCode, uint16 wParam, uint32 dwParam);
```

第二个参数为 AEEEvent 类型，用于指定应用程序接收的主要事件。第三个和第四个参数是根据接收事件来定义的短数据和长数据。 这些值取决于事件本身，并根据事件的定义来定义。

典型的，我们的应用程序的事件处理函数是一个巨大的 switch 语句的程序结构，在这个结构中我们可以看到我们所感兴趣的事件，或者直接将某些事件传递到一些二级事件处理函数之中，这些二级事件处理函数既可以是我们为了程序结构的清晰而编写的函数，也可以是某个控件或接口的事件处理函数。关于这个 switch 语句的结构，我们可以在上一章创建的应用程序中看到。

执行应用程序时，我们只需考虑和处理应用程序可能需要处理的事件。一般情况下，我们可以忽略很多事件。例如，如果应用程序执行一个只需使用上下左右箭头键作为输入的游戏，并收到与 0-9 按键对应的事件，那么它将返回 FALSE 来表示应用程序不处理这些事件。体现在代码上就是我们的 switch 语句中不必“case”这些事件。

8.1.2 捕获系统事件

在我们的应用程序当中，最起码需要处理 EVT_APP_START、EVT_APP_STOP、EVT_APP_SUSPEND 和 EVT_APP_SUSPEND 事件，上一章里面我们已经了解了它们的作用。而除了这些事件之外，应用程序中还可能涉及到的系统事件有：

1、EVT_APP_MESSAGE 事件。此事件表示当前系统发送了一个文本短消息到我们的应用程序中。

2、EVT_ALARM 事件。此事件表示系统发送了一个闹钟事件到我们的应用程序之中。

3、EVT_NOTIFY 事件。此事件表示有一个系统的通知事件到达。发送通知事件的通常包括呼叫等事件。我们的应用程序需要注册相应的通知事件才能接收到这些通知。

这些系统事件是属于 BREW 的系统级别的事件，无论当前我们的应用程序处在什么样的状态下，都会处理这些系统事件。也就是说即便当前我们的应用程序处于关闭状态，也同样的可以接收到这些系统事件。

我们还可以定义大于 EVT_USER 的事件代码，通过 ISHELL_PostEvent 接口来向特定的应用程序发送自定义事件。例如：

```
bRet = ISHELL_PostEvent( pIShell, classid, eCode, wParam, dwParam );
```

其中 pIShell 是 IShell 接口的指针，classid 表示事件发送到哪一个应用程序，eCode 就是我们要发送的事件。通过这样的方式，就像系统事件一样，无论目标应用处于什么样的状态，都可以收到这个事件进行处理。bRet 代表事件是否发送成功，注意，这里面返回值代表的是事件是否发送成功，而不是事件的处理结果。

8.1.3 捕获用户接口事件

在我们的应用程序中处理的大多数事件是键盘事件或者菜单选择事件，这些事件的特点是只有在应用程序处于活动状态的时候才能处理，这些事件就是典型的用户接口事件。这些事件的处理也是我们应用程序的核心内容，因为这些事件描述了当前用户的输入信息。

最简单的用户输入就是用户按一下按键，然后释放按键。这其中 BREW 当前活动的应用程序将收到三个事件：EVT_KEY_PRESS、EVT_KEY 和 EVT_KEY_RELEASE。EVT_KEY_PRESS 描述用户已经按下了一个按键，EVT_KEY_RELEASE 表示用户已经释放了这个按键，EVT_KEY 则是介于两者之间的一种按键装态的描述。通常情况下我们的应用

程序只需要处理 EVT_KEY 事件就好了。

对于每一个按键事件，系统通常使用一个 16 位的按键代码来表示当前按下的是哪一个按键。这些按键代码定义在 AEEVCode.h 中，最为常用的按键代码如下：

按键	按键代码
0	AVK_0
1	AVK_1
2	AVK_2
3	AVK_3
4	AVK_4
5	AVK_5
6	AVK_6
7	AVK_7
8	AVK_8
9	AVK_9
*	AVK_STAR
#	AVK_POUND
清除键（返回键）	AVK_CLR
上方向键	AVK_UP
下方向键	AVK_DOWN
左方向键	AVK_LEFT
右方向键	AVK_RIGHT
选择键（确认键）	AVK_SELECT
呼叫键	AVK_SEND
结束键（关机键）	AVK_END

通过上面的表格我们可以发现，与其他系统相比 BREW 缺少了一些独立的字符按键。这是因为通常一个 BREW 设备都是只有一个数字键盘和少数的几个控制按键，而字符的输入则是通过文件控件来完成。文本控件通过用户输入方式和所选择的不同输入法，如数字、字母以及拼音或笔画等，来实现对应的文本输入。事实上，要做到这些我们需要将应用程序捕获到的用户输入事件传入文本事件中去，传入的方式是通过文本控件的 HandlEvent 方法。

在我们的应用程序中通常使用控件来处理一些用户的输入事件，而且在应用程序中通过控件的 HandleEvent 函数将这些事件传入，而控件在处理了相关事件之后，也同样会以某种方式来保存结果或者通知应用程序。例如，文本控件会保存并显示用户输入的文本，而菜单控件则会通过发送 EVT_COMMAND 事件来告知应用程序用户的选择项目是什么。在后面的章节中我们将逐渐的看到这些应用。

有一个特殊的按键事件需要我们特别的注意，这个事件就是带有 AVK_CLR 参数的 EVT_KEY 事件。在默认情况下，当 BREW 内核捕获到这个事件的时候所执行的动作是关闭当前的应用程序。因此，如果我们不想 BREW 在收到这个事件的时候关闭我们的应用程序的话，我们需要在应用程序中处理这个事件，也就是在我们的应用程序接收到这个事件的时候返回 TRUE，以使得 BREW 内核不会再处理这个事件。

8.2 构建应用程序框架

由于在每一个 BREW 应用程序中很多事件的处理方式都基本相似，因此，如果我们可

以把这些基本的事件处理采用统一的方式来处理，那么将极大地降低我们构建新应用程序的工作量。在这一部分中，我们将来构建这样的一个应用程序框架，在这个框架中可以自动处理诸如 EVT_APP_START 和 EVT_APP_SUSPEND 等事件。

在本章中，这个应用程序框架将从一个叫做文件浏览器的应用程序中进行构建。在这个程序中，所实现的功能就是将 BREW Shared 文件夹下面的文件枚举出来，选中相应的文件后可以显示文件的基本信息，如大小等。

8.2.1 创建文件浏览器应用程序

由于我们现在才开始构建第一个真正包含了一定功能的应用程序，因此，在这里将一步一步的讲解如何构建这个应用程序。同时，还可以参考上面一章里面关于创建应用程序的步骤说明，在后面的示例中我们将省略构建应用程序的步骤。好，我们现在打开 MicroSoft Visual Studio .Net2003，选择新建项目后如下图：

图 8.1 生成 FileExplorer 应用程序

选择 BREW 向导，输入项目名称为 FileExplorer。如果现在您看不到这个 BREW 向导的话，说明还没有安装 BREW Addins。这个插件现在包含在 BREW SDK Tools 中，和 MIFEditor 和 ResourceEditor 一起。我们可以从高通网站上下载并安装 BREW SDK Tools 应用程序。

确定后出现如图 8.2 所示的下对话框，其中我们要选上 File，因为我们的应用程序中将使用文件处理的接口。然后我们选择 Options 选项，看到如图 8.3 种所示，在是否输出注释的部分选择“No”，因为这些注释将会占领很大的文档篇幅。然后选择 MIF Editor 按钮，建立此应用程序的模块信息文件，打开 MIF Editor 应用程序。如图 8.4 种所示一样。最后，点击“Finish”，建立应用程序。

在 MIF 文件编辑器的 Applets 选项卡中，我们选择“List of Applets defined in this module”中的增加 Applet 按钮，显示如图 8.5 所示。我们选择从本地生成 Class ID，并输入 Class ID 的值为 0x99999999，输入 Applet 的名称为 FileExplorer，然后点击确定，保存生成.bid 文件。

这样，一个 Applet 就加入到一个模块中去了。

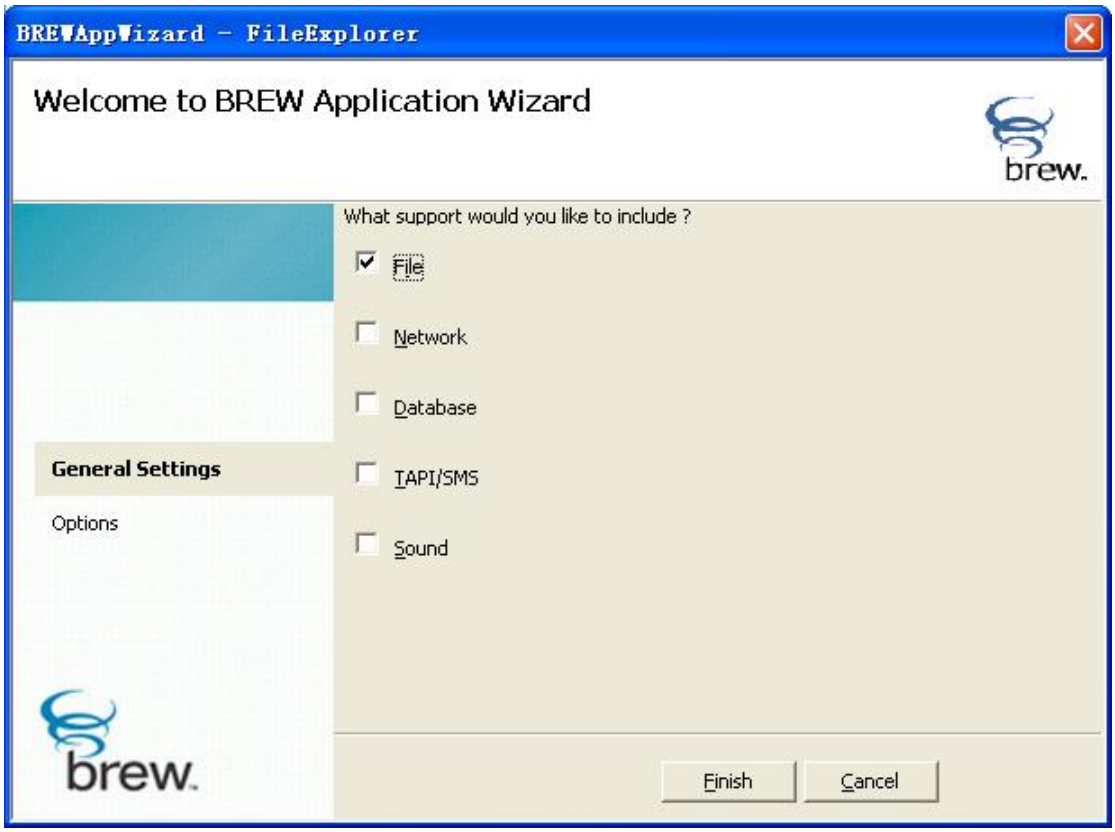

图 8.2 设置 FileExplorer 应用程序的包含头文件

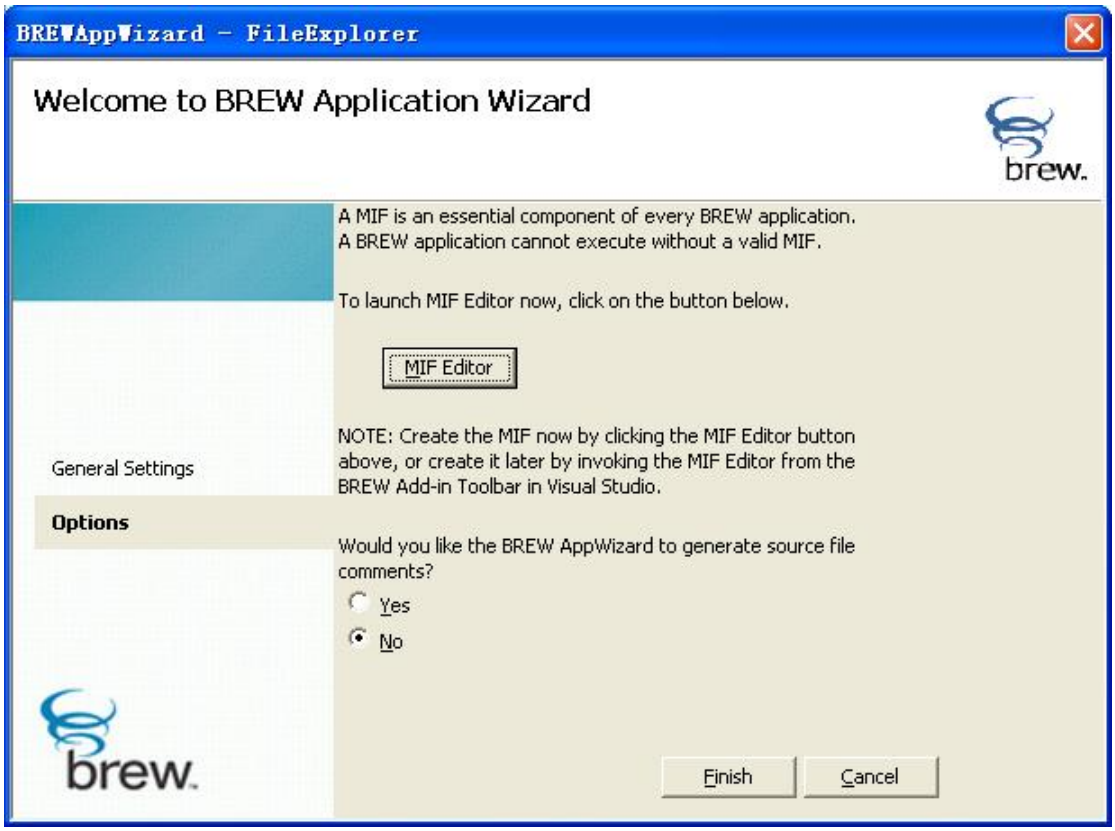

图 8.3 设置 FileExplorer 应用程序的选项

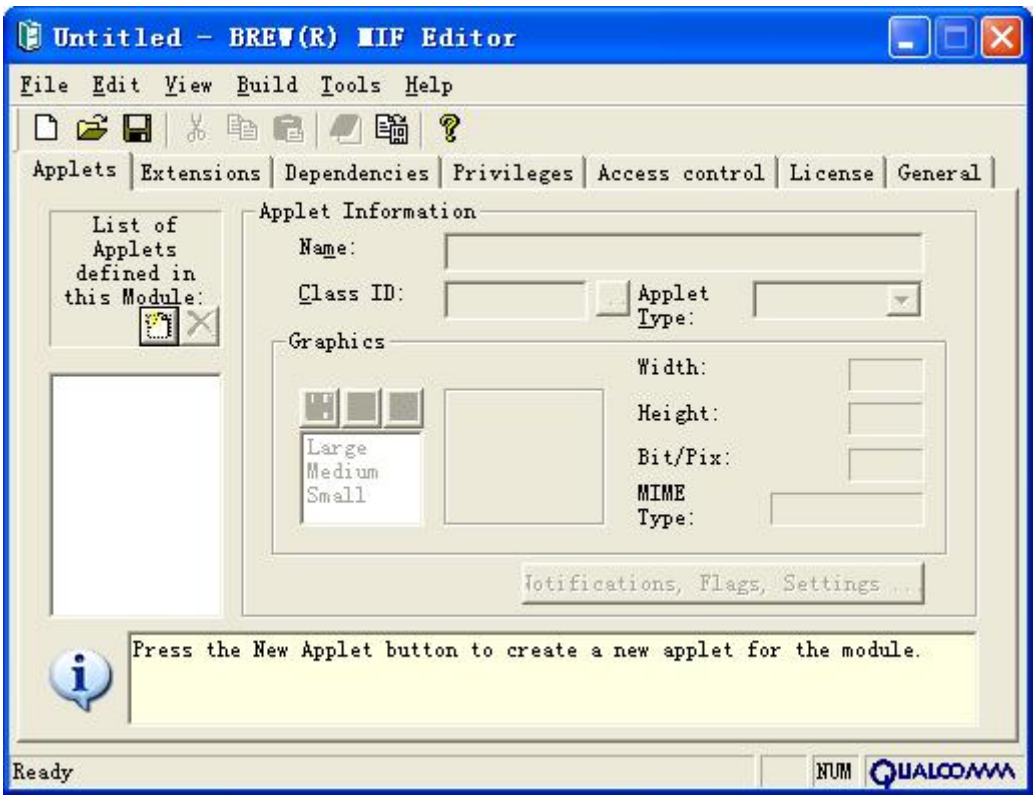

图 8.4 MIF Editor

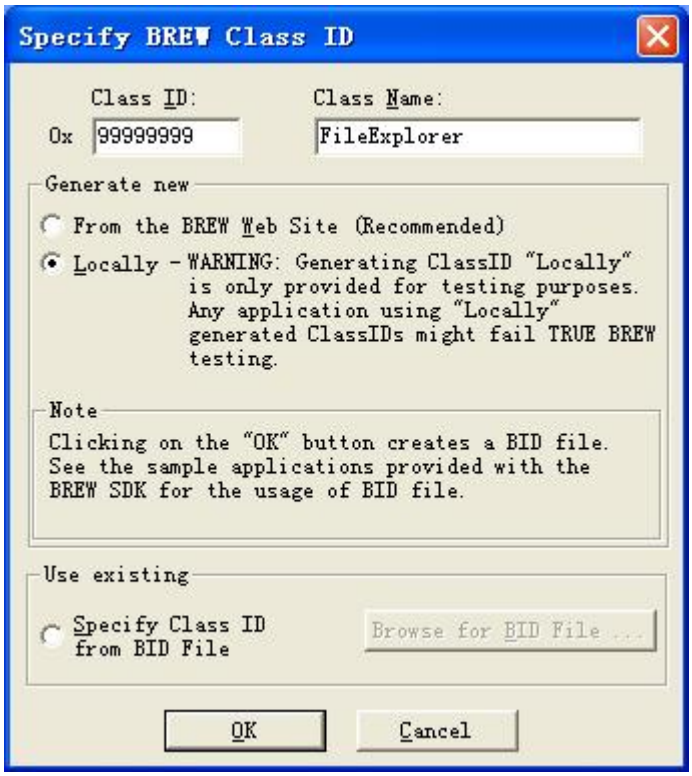

图 8.5 在模块中增加 Applet

增加完 Applet 之后，还需要到 Privileges 中增加 File 的访问权限，否则在我们的应用程序中将不能使用文件系统的接口。最后保存生成的文件为 FileExplorer.mfx 文件，同时，选择 build 菜单，生成 mif 文件。

至此，一个由向导生成的 FileExplorer 应用程序就声称完毕了，我们接下来要做的就是在这块“白纸”上勾勒出我们的应用程序框架。

8.2.2 使用状态表示应用程序

如果我们写一个应用程序只需要几百行的代码的话，那么这个世界对于程序员来说简直是太美妙了。但是，这仅仅是一个愿望，因为实际的程序代码通常需要成千上万行。就算是用 BREW 这样高度集成的平台来开发一个稍微复杂的应用程序，都需要上万行的代码才能完成。而且对于程序员来说，每增加一行代码，就相当于打开了一个潘多拉盒子，不知道这一行将来会带来什么样的后果，这种情况在代码较多的时候尤为明显。把这个潘多拉的盒子关上是所有程序员一直的的梦想，于是，各种各样的程序结构诞生了，函数分割、文件分割、以及状态分割等形式相继出现了，又于是 C 语言出现了、C++语言出现了、可视化编程出现了。这些都是程序员们为了关闭这个潘多拉盒子所做的努力，事实证明这些努力没有白费。这些编程语言以及对应的代码组织方式极大的方便的程序的管理，使得开发大型程序不再那么复杂了。

当然，凡事总有穷尽，无论怎样的努力，程序中间总会有缺陷。或者这是人类思维方式的一种缺陷，或者受限于编程者的水平，这个潘多拉盒子还是时常会打开。“当我们改变不了环境的时候就改变我们自己吧”，对于程序缺陷，既然我们无法避免那就容忍它的存在吧！我们唯一能够做的就是尽可能的减少它们的存在。

编写简单的程序可以在很大程度上减少程序的缺陷，而让程序变得简单的方法就是将程序都分割成一小块一小块的，然后通过某种方式将这些小块的程序连接起来，共同构成一个大型的应用程序。一种较好的方式就是我们将要介绍的状态机，我们可以把应用程序分割成不同的状态。例如，在 HelloWorld 应用程序中，显示“Hello World”的界面就可以看成一个显示字符串的状态。为了在 BREW 应用程序开发的过程中能够拥有更加清晰的程序流程，我们特意设计了这样的一个状态机的应用程序框架。

我们的基本思路是这样的，划分应用程序状态的方法是每一个显示界面都作为一个状态，在进入状态的时候创建界面，同时状态机停止运转，此时等待用户在当前的界面进行操作，用户操作完成后关闭当前的界面，并启动状态机继续运行。由于 BREW 的每一个应用程序都是通过 HandleEvent 方法来获得事件的，在每一个界面等待用户输入的时候，每一个界面都对应一个界面的 HandleEvent 函数，用来捕获用户在这个界面的输入。这样，我们的应用程序将被分成状态和界面两种单元。

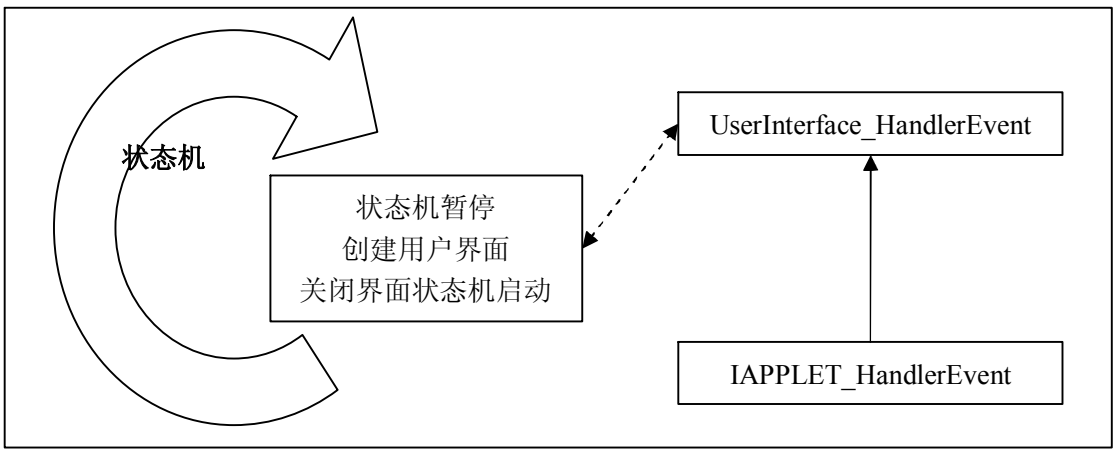

图 8.6 状态机基本原理

状态机在代码上讲就是一个 for(;;)循环，只有在创建一个用户界面之后才会暂时终止运行，可以实现这样功能的代码如下（可以在本书所附的示例代码 Test5/FileExplorer 应用目录下的 AppStateMachine.c 中找到这个函数）：

```
/*==========================================================================
函数       : CStateMachine_RunFSM
说明       : 有限状态机引擎。
参数       : pMe [in]：指向 CStateMachine 对象结构的指针
返回值     : 无。
备注       : 无
==========================================================================*/
static void CStateMachine_RunFSM(CStateMachine *pMe)
{
    NextFSMAction nextFSMAction = NFSMACTION_WAIT;

    for(;;)
    {
        nextFSMAction = pMe->m_pfnState((IStateMachine *)pMe,
                                        pMe->m_pUserState);

        pMe->m_eWndRet      = WNDRET_CREATE;
        pMe->m_wWndParam    = 0;
        pMe->m_dwWndParam   = 0;

        if (nextFSMAction == NFSMACTION_WAIT)
        {
            break;
        }
    }
} // End CStateMachine_RunFSM
```

pMe->m_pfnState 就是我们在应用程序中的状态函数，状态机根据这个函数的返回值 nextFSMAction 来判断是否暂停状态机的运行。从代码中可以看到当 nextFSMAction 为 NFSMACTION_WAIT 的时候这个 for 循环将被 break 语句打破。

我们定义的状态机结构体如下所示（下面的代码综合了 AppStateMachine.h 和.c 的内容）：

```
// 状态处理函数返回给状态处理主函数的值类型
typedef enum _NextFSMAction{
    NFSMACTION_WAIT,
    NFSMACTION_CONTINUE
} NextFSMAction;

typedef NextFSMAction (*PFNSTATE)(IStateMachine *po, void *pUser);

typedef struct _CStateMachine{
    AEEVTBL(IStateMachine) *pvt;
    uint32                 m_nRefs;
```

```
    // IShell interface
    IShell              *m_pShell;

    PFNSTATE        m_pfnState;        // 状态处理函数
    void *              m_pUserState;     // 用户数据
    PFNAEEEVENT m_pfnEvent;        // 事件处理函数
    void *              m_pUserEvent;     // 用户数据
    int                 m_eWndRet;          // Windows 返回值
    uint16              m_wWndParam;       // Windows 返回值 word 参数
    uint32              m_dwWndParam;     // Windows 返回值 dword 参数

    AEECLSID        m_dwAppClsID;     // 当前使用此接口的应用程序 Class ID
    boolean             m_bSuspending;    // 当前状态机是否被挂起
}CStateMachine;
```

应用程序中将通过一个指针变量来保存一个这样的结构体。我们将创建的用户界面叫做一个 Window。状态机使用 m_pfnState 来获得状态，使用 m_pfnEvent 来处理用户 Window 的事件，在 m_pfnState 状态函数执行的时候创建 Window。

8.2.3 实现状态机管理的接口

为了能够管理状态机，我们必须提供一些管理状态机的接口：

1、启动状态机，通过这个接口应用程序可以选择在何时启动这个状态机。

2、终止状态机，通过这个接口应用程序可以选择在何时终止状态机的运行。

3、挂起状态机，通过这个接口应用程序可以暂停状态机的运行，同时保存相关的状态数据，以便于恢复状态机的运行。

4、恢复状态机，通过这个接口应用程序可以恢复一个被挂起的状态机继续运行。

5、转移状态，通过这个接口应用程序可以转移到下一个状态去处理。

6、创建一个窗口（Window），通过这个接口应用程序可以创建一个显示界面。

7、关闭一个窗口（Window），通过这个接口应用程序可以关闭一个显示界面，同时设置相应的 Window 返回值给状态处理函数。

8、Window 事件捕获函数，应用程序将通过这个接口将事件传递给 Window 的事件捕获函数。

9、辅助函数，如获得当前的状态，以及获得当前窗口的返回值等操作。

与上面描述功能基本相近的各个函数已经定义在了 FileExplorer 应用程序中的 AppStateMachine.c 的文件中。他们的形式如下：

```
/*==========================================================================
函数      ：IStateMachine_HandleEvent
描述      ：状态机的事件捕获函数，在此函数中将事件传递给在当前状态中打开的窗口
参数      ：po[in] 指向当前状态机结构体的指针
            eCode[in] 当前的事件代码
            wParam[in] 事件的 16 位参数
            dwParam[in] 事件的 32 位参数
```

```
返回值    ：如果捕获事件则返回 TRUE，否则返回 FALSE
=============================================================================*/
static boolean IStateMachine_HandleEvent(IStateMachine *po,
                                         AEEEvent    eCode,
                                         uint16      wParam,
                                         uint32      dwParam)
{
    CStateMachine *pMe = (CStateMachine*)po;
    return
pMe->m_pfnEvent?pMe->m_pfnEvent(pMe->m_pUserEvent,eCode,wParam,dwParam):FALSE;
}// End IStateMachine_HandleEvent

/*=============================================================================
函数      ：IStateMachine_Start
描述      ：开始状态机的运行，状态机开始之后可以调用 ISTATEMACHINE_Stop 终止状态
机
            的运行，或者 ISTATEMACHINE_Stop 挂起状态机
参数      ：po[in] 指向当前状态机结构体的指针
            pfnState[in] 启动时的状态函数，同时也是状态 ID
            pUser[in] 传递给状态函数的用户数据
返回值    ：无
=============================================================================*/
static void IStateMachine_Start(IStateMachine *po,
                                PFNSTATE      pfnState,
                                void          *pUser)
{
    CStateMachine *pMe = (CStateMachine*)po;
    pMe->m_pfnState     = pfnState;
    pMe->m_pUserState   = pUser;

    if(pMe->m_pfnState){
        CStateMachine_RunFSM(pMe);
    }
}// End IStateMachine_Start

/*=============================================================================
函数      ：IStateMachine_Stop
描述      ：终止状态机的运行，终止之后只能使用 ISTATEMACHINE_Start 重新启动状态机
参数      ：po[in] 指向当前状态机结构体的指针
返回值    ：无
=============================================================================*/
static void IStateMachine_Stop(IStateMachine *po)
{
    CStateMachine *pMe = (CStateMachine*)po;
```

```
    IStateMachine_Suspend(po);
    pMe->m_pfnState = NULL;
    pMe->m_pfnEvent = NULL;
}// End IStateMachine_Stop

/*============================================================================
函数      : IStateMachine_Suspend
描述      : 挂起当前正在运行的状态机，挂起之后使用 ISTATEMACHINE_Start 重新启动
            状态机，或使用 ISTATEMACHINE_Resume 恢复当前的状态机运行
参数      : po[in] 指向当前状态机结构体的指针
返回值    : 无
============================================================================*/
static void IStateMachine_Suspend(IStateMachine *po)
{
    CStateMachine *pMe = (CStateMachine*)po;
    if(pMe->m_pfnState){
        pMe->m_bSuspending = TRUE;
        IStateMachine_CloseWnd(po,0,0,0);
    }
}// End IStateMachine_Suspend

/*============================================================================
函数      : IStateMachine_Resume
描述      : 恢复被挂起的状态机
参数      : po[in] 指向当前状态机结构体的指针
返回值    : 无
============================================================================*/
static void IStateMachine_Resume(IStateMachine *po)
{
    CStateMachine *pMe = (CStateMachine*)po;

    if(pMe->m_pfnState){
        pMe->m_bSuspending = FALSE;
        CStateMachine_RunFSM(pMe);
    }
}// IStateMachine_Resume

/*============================================================================
函数      : IStateMachine_GetState
描述      : 获取状态机当前的状态
参数      : po[in] 指向当前状态机结构体的指针
返回值    : 无
============================================================================*/
```

```
static void* IStateMachine_GetState(IStateMachine *po)
{
    CStateMachine *pMe = (CStateMachine*)po;
    return pMe->m_pfnState;
}// IStateMachine_GetState

/*=============================================================================
函数      ：IStateMachine_MoveTo
描述      ：转移当前的状态到下一个状态
参数      ：po[in] 指向当前状态机结构体的指针
            pfnState[in] 状态处理函数，也是状态的 ID
            pUser[in] 状态处理函数使用的用户数据
返回值    ：无
=============================================================================*/
static void IStateMachine_MoveTo(IStateMachine *po,
                                 PFNSTATE      pfnState,
                                 void          *pUser)
{
    CStateMachine *pMe = (CStateMachine*)po;
    pMe->m_pfnState    = pfnState;
    pMe->m_pUserState  = pUser;
}// End IStateMachine_MoveTo

/*=============================================================================
函数      ：IStateMachine_OpenWnd
描述      ：打开一个窗口，用来显示用户界面和处理用户事件
参数      ：po[in] 指向当前状态机结构体的指针
            HandleEvent[in] 窗口事件处理函数，也是窗口的 ID
            pUser[in] 窗口事件处理函数使用的用户数据
返回值    ：无
=============================================================================*/
static void IStateMachine_OpenWnd(IStateMachine *po,
                                  PFNAEEEVENT   HandleEvent,
                                  void          *pUser)
{
    CStateMachine *pMe = (CStateMachine*)po;
    pMe->m_pfnEvent    = HandleEvent;
    pMe->m_pUserEvent  = pUser;
    (void)ISHELL_PostEvent( pMe->m_pShell,
                            pMe->m_dwAppClsID,
                            EVT_WND_OPEN,
                            0, 0);
}// End IStateMachine_OpenWnd
```

```
/*==========================================================================
函数      : IStateMachine_CloseWnd
描述      : 关闭当前的窗口
参数      : po[in] 指向当前状态机结构体的指针
            eWndRet[in] 窗口关闭时的返回值，状态处理函数通过这个值来决定下一步
                        怎么做
            wParam[in] 窗口返回值的 16 位参数
            dwParam[in] 窗口返回值的 32 位参数
返回值    : 无
==========================================================================*/
static void IStateMachine_CloseWnd(IStateMachine *po,
                                   int          eWndRet,
                                   uint16       wParam,
                                   uint32       dwParam)
{
    CStateMachine *pMe = (CStateMachine*)po;

    if(pMe->m_pfnEvent){
        (void)ISHELL_HandleEvent(pMe->m_pShell, EVT_WND_CLOSE, 0, 0);
        pMe->m_pfnEvent      = NULL;
        pMe->m_pUserEvent    = NULL;
        pMe->m_eWndRet       = eWndRet;
        pMe->m_wWndParam     = wParam;
        pMe->m_dwWndParam    = dwParam;

        // 状态机被挂起，等待状态机恢复或重新 Start
        if(FALSE == pMe->m_bSuspending){
            CStateMachine_RunFSM(pMe);
        }
    }
}// End IStateMachine_CloseWnd

/*==========================================================================
函数      : IStateMachine_GetWndRet
描述      : 获得最近关闭窗口的返回值
参数      : po[in] 指向当前状态机结构体的指针
            *pwParam[in/out] 窗口返回值的 16 位参数
            *pdwParam[in/out] 窗口返回值的 32 位参数
返回值    : int 窗口的返回值
==========================================================================*/
static int IStateMachine_GetWndRet(IStateMachine *po,
                                   uint16       *pwParam,
                                   uint32       *pdwParam)
{
```

```
    CStateMachine *pMe = (CStateMachine*)po;
    if(pwParam){
        *pwParam = pMe->m_wWndParam;
    }

    if(pdwParam){
        *pdwParam = pMe->m_dwWndParam;
    }
    return pMe->m_eWndRet;
}// End IStateMachine_GetWndRet
```

如果我们完全使用状态机来构建一个应用程序的话，那么，我们可以在在应用程序的EVT_APP_START 事件中直接调用 IStateMachine_Start 函数来启动状态机，在调用这个函数时需要指定初始状态；在 EVT_APP_STOP 事件中调用 IStateMachine_Stop 来终止状态机的运行；在 EVT_APP_SUSPEND 事件中调用 IStateMachine_Suspend 来挂起状态机；在EVT_APP_RESUME 事件中调用 IStateMachiine_Resume 来恢复状态机的运行。在状态机运行的过程中，使用 IStateMachine_MoveTo 来转移到下一个状态，使用 IStateMachine_GetState 来获得当前的状态。

在应用程序中，表示状态的是一个如下类型的函数：

```
typedef NextFSMAction (*PFNSTATE)(IStateMachine *po, void *pUser);
```

其中，po 就是传递了 CStateMachine 结构体的一个指针，pUser 是用户在使用IStateMachine_MoveTo 中指定的状态处理函数所需要传入的用户数据。在这个状态处理函数中通常判断当前的窗口（Window）返回值是什么来决定如何处理状态，这些返回值预定在下面的枚举变量中（AppStateMachine.h）：

```
// 窗口返回值
enum{
    WNDRET_CREATE = 0,
    WNDRET_OK,
    WNDRET_CANCELED,
    WNDRET_YES,
    WNDRET_NO,
    WNDRET_USER1,
    WNDRET_USER2,
    WNDRET_USER3,
    WNDRET_USER4,
    WNDRET_USER5,
    WNDRET_MAX
};
```

WNDRET_CREATE 表示用户要在这个状态中创建一个窗口（Window），此时应该调用IStateMachine_OpenWnd 接口打开一个窗口（Window）。调用此接口的时候，窗口（Window）的事件处理函数将接收到一个 EVT_WND_OPEN 事件。关闭窗口（Window）的动作是由窗口（Window）本身通过调用 IStateMachine_Close 来完成的，在这个接口执行的过程中将向窗口（Window）事件处理函数发送 EVT_WND_CLOSE 事件，同时重新启动状态机。启动状态机的时候将再次执行当前的状态处理函数，所不同的是此时执行状态处理函数的时候，窗口（Window）的返回值不同了（在关闭窗口的时候重新替换成了上面的一些定义）。此时

根据窗口的返回值决定如何操作。

到此为止您可能有这样的一个疑问，除了 CStateMachine 外，里面还有一个 IStateMachine 类型是做什么的呢？在这里呢，我先不说，具体的情况将会在第三篇扩展 BREW 接口中谈到。当然，有兴趣的读者可以自己试着将 AppStateMachine.c 和 AppStateMachine.h 两个文件通读一遍，提前研究一下有好处。在这里唯一要说明的是，要想使用 StateMachine 的接口函数，需要调用 AppStateMachine_New 函数来获得 IStateMachine 的接口指针。

8.3 使用应用程序框架构建应用

在上面一节当中，我们已经介绍了状态机相关的管理接口，现在我们就要用上面的接口来构建 FileExplorer 应用程序的内容了。本章第一节当中我们已经建立了这个应用程序，这一节将主要讲解如何使用应用程序框架来构建我们的应用程序。

8.3.1 文件浏览器应用程序的启动代码

文件浏览器要完成的主要功能是使用菜单控件枚举出在 fs:\shared 路径下的文件，当选择这些文件的时候可以查看文件的信息，包括大小和创建时间。为此，我们在这个应用程序中使用了一个 BREW 接口 IFileMgr 和一个控件 IMenuCtl。首先，我们要做的事情就是在应用程序的结构体中增加相应的变量。完整的应用程序的结构体变成如下样式：

```
typedef struct _FileExplorer {
    AEEApplet          a ;               // First element must be AEEApplet
    AEEDeviceInfo    DeviceInfo;    // Hardware device information
    AEERect             m_myRC;          // Applet 的矩形框

    IStateMachine *m_pStateMachine; //  状态机实例指针
    IFileMgr          *m_pFileMgr;          //  文件管理接口实例指针
    IMenuCtl          *m_pMenuList;        //  菜单控件接口实例指针

    FileInfo           m_pFileInfo;     //  保存所选文件信息
} FileExplorer;
```

然后在 FileExplorer_InitAppData 函数中初始化这些变量：

```
static boolean FileExplorer_InitAppData(FileExplorer* pMe)
{
    int nRet;

    pMe->DeviceInfo.wStructSize = sizeof(pMe->DeviceInfo);
    ISHELL_GetDeviceInfo(pMe->a.m_pIShell,&pMe->DeviceInfo);
    pMe->m_myRC.dx = pMe->DeviceInfo.cxScreen;
    pMe->m_myRC.dy = pMe->DeviceInfo.cyScreen;

    //  获得状态机实例
    nRet = AppStateMachine_New(pMe->a.m_pIShell,&pMe->m_pStateMachine);
    if(nRet != SUCCESS){
```

```
        return FALSE;
    }

    // 创建文件管理接口
    nRet = ISHELL_CreateInstance( pMe->a.m_pIShell,
                                  AEECLSID_FILEMGR,
                                  &pMe->m_pFileMgr);
    if(nRet != SUCCESS){
        return FALSE;
    }

    // 创建一个菜单控件
    nRet = ISHELL_CreateInstance( pMe->a.m_pIShell,
                                  AEECLSID_MENUCTL,
                                  &pMe->m_pMenuList);
    if(nRet != SUCCESS){
        return FALSE;
    }
    return TRUE;
}// End FileExplorer_InitAppData
```

为了保证分配和释放接口的对等关系，我们需要在 FileExplorer_FreeAppData 函数中释放创建的接口实例：

```
void FileExplorer_FreeAppData(FileExplorer* pMe)
{
    if(pMe->m_pStateMachine){
        ISTATEMACHINE_Release(pMe->m_pStateMachine);
    }

    if(pMe->m_pFileMgr){
        IFILEMGR_Release(pMe->m_pFileMgr);
    }

    if(pMe->m_pMenuList){
        IMENUCTL_Release(pMe->m_pMenuList);
    }
}// End FileExplorer_FreeAppData
```

之后的主要任务是修改应用程序的事件捕获函数为如下样式：

```
static boolean FileExplorer_HandleEvent(FileExplorer *pMe,
                                        AEEEvent     eCode,
                                        uint16       wParam,
                                        uint32       dwParam)
{
    // 首先将事件传递给状态机
    if(ISTATEMACHINE_HandleEvent(pMe->m_pStateMachine,eCode,wParam,dwParam)){
```

```
        // 状态机捕获了事件，直接返回
        return TRUE;
    }

    switch (eCode)
    {
        case EVT_APP_START:
            // 启动状态机
            ISTATEMACHINE_Start(pMe->m_pStateMachine, FE_STATE_INIT, pMe);
            return(TRUE);

        case EVT_APP_STOP:
            // 停止状态机
            ISTATEMACHINE_Stop(pMe->m_pStateMachine);
            return(TRUE);

        case EVT_APP_SUSPEND:
            // 挂起状态机
            ISTATEMACHINE_Suspend(pMe->m_pStateMachine);
            return(TRUE);

        case EVT_APP_RESUME:
            // 恢复状态机运行
            ISTATEMACHINE_Resume(pMe->m_pStateMachine);
            return(TRUE);

        default:
            break;
    }

    return FALSE;
}// End FileExplorer_HandleEvent
```

从这个函数中我们可以看到事件将首先传递给应用程序框架。而且在 EVT_APP_START 事件中，我们启动的状态是 FE_STATE_INIT。这个状态处理函数如下：

```
static NextFSMAction FE_STATE_INIT(IStateMachine *po, FileExplorer* pMe)
{
    ISTATEMACHINE_MoveTo(po,FE_STATE_SHOWLIST,pMe);
    return NFSMACTION_CONTINUE;
}// End FE_STATE_INIT
```

从这个状态开始，我们就开始正式的进入了状态机运行阶段。我们也可以看到，目前这个状态所做的就是转移到 FE_STATE_SHOWLIST 状态。我们可以在这个状态中做一些使用状态机所必需的初始化操作。当然，当前我们的应用程序很简单，因此没有作什么具体的处理。

8.3.2 文件浏览器应用程序的文件列表状态

状态机运行之后，就会从初始化状态（FE_STATE_INIT）转移到文件列表状态（FE_STATE_SHOWLIST）。文件列表状态的处理内容如下：

```
static NextFSMAction FE_STATE_SHOWLIST(IStateMachine *po, FileExplorer* pMe)
{
    switch(ISTATEMACHINE_GetWndRet(po, NULL, NULL)){
    case WNDRET_CREATE:
        // 打开 WndShowList_HandleEvent 窗口，显示界面，状态机进入等待状态
        ISTATEMACHINE_OpenWnd(po,WndShowList_HandleEvent,pMe);
        return NFSMACTION_WAIT;

    case WNDRET_OK:
        // 转移状态到 FE_STATE_DRAWINFO
        ISTATEMACHINE_MoveTo(po,FE_STATE_DRAWINFO,pMe);
        break;

    case WNDRET_CANCELED:
        // 转移状态到 FE_STATE_EXIT
        ISTATEMACHINE_MoveTo(po,FE_STATE_EXIT,pMe);
        break;

    default:
        break;
    }

    return NFSMACTION_CONTINUE;
}// End FE_STATE_SHOWLIST
```

在处理状态的 WNDRET_CREATE 窗口（Window）返回值时，我们创建了一个窗口（Window）WndShowList_HandleEvent，同时返回 NFSMACTION_WAIT 告诉应用程序框架状态机应处于等待状态直至窗口（Window）关闭。由于 WNDRET_CREATE 是进入状态时默认的值，因此进入这个状态之后就会创建文件类表窗口。在 WndShowList_HandleEvent 函数中将进行具体的窗口事件处理：

```
static boolean WndShowList_HandleEvent(FileExplorer *pMe,
                                       AEEEvent      eCode,
                                       uint16        wParam,
                                       uint32        dwParam)
{
    // 将事件传递给菜单控件
    if(IMENUCTL_HandleEvent(pMe->m_pMenuList, eCode, wParam, dwParam)){
        // 菜单控件捕获了事件，直接返回
        return TRUE;
    }
```

```
switch (eCode){
case EVT_WND_OPEN:   // 打开窗口事件
{
    int       nItemID   = 1; // 菜单的起始 ItemID
    // 菜单标题“Shared Files”
    AECHAR wTitle[] = {'S','h','a','r','e','d',
                       ' ','F','i','l','e','s','\0'};

    // 设置菜单控件的显示方式
    IMENUCTL_SetRect(pMe->m_pMenuList, &pMe->m_myRC);
    IMENUCTL_SetTitle(pMe->m_pMenuList, NULL, NULL, wTitle);
    IMENUCTL_SetProperties(pMe->m_pMenuList,MP_UNDERLINE_TITLE);

    // 枚举文件夹中的文件
    (void)IFILEMGR_EnumInit(pMe->m_pFileMgr, AEEFS_SHARED_DIR, FALSE);
    for(;;){
        FileInfo myFileInfo;
        AECHAR     pszFile[AEE_MAX_FILE_NAME];

        // 枚举一个文件
        if(!IFILEMGR_EnumNext(pMe->m_pFileMgr, &myFileInfo)){
            // 枚举失败，证明已经枚举完成，跳出循环
            break;
        }

        // 将文件名加入菜单控件
        STRTOWSTR(myFileInfo.szName, pszFile, sizeof(pszFile));
        IMENUCTL_AddItem( pMe->m_pMenuList,
                          NULL, NULL, nItemID++,
                          pszFile, 0);
    }

    // 激活并重绘菜单控件
    IMENUCTL_SetActive(pMe->m_pMenuList, TRUE);
    IMENUCTL_Redraw(pMe->m_pMenuList);
    return(TRUE);
}

case EVT_WND_CLOSE: // 关闭窗口事件
    // 删除菜单控件中的全部条目并停用菜单控件
    IMENUCTL_DeleteAll(pMe->m_pMenuList);
    IMENUCTL_SetActive(pMe->m_pMenuList, FALSE);
    return(TRUE);
```

```
    case EVT_KEY:
        switch(wParam){
        case AVK_CLR:
        case AVK_SOFT2:
            // 关闭当前窗口并返回取消状态给状态机处理
            ISTATEMACHINE_CloseWnd(pMe->m_pStateMachine,
WNDRET_CANCELED,0,0);
            return TRUE;
        default:
            break;
        }
        break;

    case EVT_COMMAND:
    {
        CtlAddItem itemInfo;
        char     pszFile[AEE_MAX_FILE_NAME];

        // 获得当前选中文件信息，并存储到 pMe->m_pFileInfo 中去
        IMENUCTL_GetItem(pMe->m_pMenuList, wParam, &itemInfo);
        WSTRTOSTR(itemInfo.pText, pszFile, sizeof(pszFile));
        IFILEMGR_GetInfo(pMe->m_pFileMgr, pszFile, &pMe->m_pFileInfo);

        // 关闭当前窗口，同时返回 OK 状态给状态机处理
        ISTATEMACHINE_CloseWnd(pMe->m_pStateMachine, WNDRET_OK, 0, 0);
        return TRUE;
    }

    default:
        break;
    }

    return FALSE;
}// End WndShowList_HandleEvent
```

EVT_WND_OPEN 事件时应用程序框架打开一个窗口时发送给应用程序的，应用程序将把这个事件传递给应用程序框架，然后，应用程序框架再将事件传递给 WndShowList_HandleEvent 函数。我们可以看见，在这个事件中，应用程序设置了菜单控件的属性，同时将枚举出来的文件添加到菜单控件中去。然后刷新界面，显示出文件列表。

现在，应用程序将停留在这个界面上等待用户的输入。具体的，当我们按返回键或者软键 2 的时候将关闭当前的窗口，并将窗口的返回值指定为 WNDRET_CANCELED。同时应用程序将接收到 EVT_WND_CLOSE 事件，在这个事件中我们清理了菜单控件的内容。关闭窗口之后将重新启动状态机运行。此时，我们可以看到 FE_STATE_SHOWLIST 状态中处理 WNDRET_CANCELED 的方法是进入 FE_STATE_EXIT 状态：

```
static NextFSMAction FE_STATE_EXIT(IStateMachine *po, FileExplorer* pMe)
{
    // 关闭应用程序，同时状态机进入等待状态
    ISHELL_CloseApplet(pMe->a.m_pIShell, FALSE);
    return NFSMACTION_WAIT;
}// End FE_STATE_EXIT
```

这个状态就是简单的关闭当前应用程序就完事了。

当我们在 WndShowList_HandleEvent 选择一个文件的时候，菜单控件将发送一个 EVT_COMMAND 事件给应用程序，这个事件的 wParam 参数代表了选中的条目，我们根据这个信息获得了我们所选的文件名。同时，根据这个文件名，我们将获得所选的文件信息，并保存在应用程序结构体的 m_pFileInfo 变量中。调用 ISTATEMACHINE_CloseWnd 关闭当前窗口，并设置返回状态为 WNDRET_OK，在 FE_STATE_SHOWLIST 状态中，处理这个返回值的方式是进入 FE_STATE_DRAWINFO 状态。

8.3.3 文件浏览器应用程序的文件信息状态

FE_STATE_DRAWINFO 状态的处理函数如下：

```
static NextFSMAction FE_STATE_DRAWINFO(IStateMachine *po, FileExplorer* pMe)
{
    switch(ISTATEMACHINE_GetWndRet(po, NULL, NULL)){
    case WNDRET_CREATE:
        // 打开 WndDrawInfo_HandleEvent 窗口，显示界面，状态机进入等待状态
        ISTATEMACHINE_OpenWnd(po,WndDrawInfo_HandleEvent,pMe);
        return NFSMACTION_WAIT;

    case WNDRET_OK:
        // 转移状态到 FE_STATE_SHOWLIST
        ISTATEMACHINE_MoveTo(po,FE_STATE_SHOWLIST,pMe);
        break;

    default:
        break;
    }

    return NFSMACTION_CONTINUE;
}// End FE_STATE_DRAWINFO
```

在处理 WNDRET_CREATE 返回值的时候，创建了 WndDrawInfo_HandleEvent 窗口：

```
static boolean WndDrawInfo_HandleEvent(FileExplorer *pMe,
                                       AEEEvent      eCode,
                                       uint16        wParam,
                                       uint32        dwParam)
{
    switch (eCode){
```

```
case EVT_WND_OPEN:   // 打开窗口事件，使用 pMe->m_pFileInfo 中的信息进行显示
{
    char FileName[] = "Name: %s";
    char FileSize[] = "Size: %d Bytes";
    char FIleDate[] = "Date: %04d.%02d.%02d %02d:%02d:%02d";
    char DestBuf[AEE_MAX_FILE_NAME+10];
    AECHAR StrBuf[AEE_MAX_FILE_NAME+10];
    JulianType myFileDate;

    IDISPLAY_ClearScreen(pMe->a.m_pIDisplay);

    // Draw file name
    SPRINTF(DestBuf, FileName, pMe->m_pFileInfo.szName);
    STRTOWSTR(DestBuf, StrBuf, sizeof(StrBuf));
    IDISPLAY_DrawText(pMe->a.m_pIDisplay ,0,StrBuf,-1,0,0,NULL,0);

    // Draw file size
    SPRINTF(DestBuf, FileSize, pMe->m_pFileInfo.dwSize);
    STRTOWSTR(DestBuf, StrBuf, sizeof(StrBuf));
    IDISPLAY_DrawText(pMe->a.m_pIDisplay ,0,StrBuf,-1,0,20,NULL,0);

    // Draw date
    GETJULIANDATE(pMe->m_pFileInfo.dwCreationDate, &myFileDate);
    SPRINTF(DestBuf, FIleDate,
            myFileDate.wYear, myFileDate.wMonth, myFileDate.wDay,
            myFileDate.wHour, myFileDate.wMinute, myFileDate.wSecond);
    STRTOWSTR(DestBuf, StrBuf, sizeof(StrBuf));
    IDISPLAY_DrawText(pMe->a.m_pIDisplay ,0,StrBuf,-1,0,40,NULL,0);
    IDISPLAY_Update(pMe->a.m_pIDisplay);
    return(TRUE);
}

case EVT_WND_CLOSE: // 关闭窗口事件
    return(TRUE);

case EVT_KEY:
    // 按任意键返回 OK 给状态机处理
    ISTATEMACHINE_CloseWnd(pMe->m_pStateMachine, WNDRET_OK, 0, 0);
    return(TRUE);

default:
    break;
}
```

```
    return FALSE;
}// End WndDrawInfo_HandleEvent
```

在 EVT_WND_OPEN 事件中，我们在屏幕上依次描绘了文件名、尺寸和创建日期。无论接收到任何按键事件都将关闭当前的窗口，并设置返回值为 WNDRET_OK。在 FE_STATE_DRAWINFO 状态中处理 WNDRET_OK 的方式是返回 FE_STATE_SHOWLIST 状态。

8.4 小结

在这一章里，我们熟悉了 BREW 处理事件的方式，以及主要需要处理哪些事件。我们了解了系统事件和用户事件之间的区别，初步理解了 BREW 的事件驱动模型。更为重要的是，根据这个模型，我们构建了一个通用的应用程序框架，通过这个框架，我们可以简化复杂应用程序的开发。需要各位读者了解的是，这个应用程序框架采用了一个有限循环状态机。通过对这个状态机的理解，将有助于我们加深对 BREW 的认识。

思考题

1、在一个状态中创建了一个窗口，然后在窗口事件处理函数中关闭了这个窗口，并返回了 WNDRET_OK，问，在这个返回值的处理中还可以创建窗口吗？也就是一个状态处理函数之中可以多次创建窗口吗？

第三篇 一识庐山真面目

我们可以很容易的使用向导在 BREW SDK 中开发一个应用程序，而且可以在模拟器中看到应用程序的效果。然而，BREW 的价值体现在什么地方，为什么我们需要 BREW？不回答这些问题就不能够真正的懂得 BREW 存在的必要性，也不能清楚地知道 BREW 的来龙去脉。更进一步的，对于一个程序开发者来说，如果不知道所开发应用平台的特性，那么也很难能够开发出变化自如、随心所至的应用程序来，毕竟所有的平台都是不完美的。而且，由于程序开发者对于接口的理解可能也会与接口的开发者之间有差别，因此，如何理解诸如接口实现的原理（如，通过理解 Bitmap 概念而认识 IBitmap 接口）就显得十分重要。一个平台当中主要有些什么，它们是怎么实现的，回答这些问题就是我们这一部分的主要内容。

学习一门新技术，总会涉及到很多相关的知识，这些相关的知识用软件系统中的说法叫做“上下文”。我们在前两部分已经介绍了关于掌握 BREW 的一些“上下文”知识，目的是让您能够更快更好的掌握我们将要学习的内容。只有连同上下文一同掌握了，我们才能够完整的把握一门技术，甚至某些时候连这门技术的历史也要掌握。这就是要有“吾将上下而求索”的精神。惟有如此，我们才能够清楚的认识它、消化它，然后将这些技术为我所用。这部分的内容是本书的精华所在，可能囿于笔者的水平，未能将一些内容阐释清楚，希望各位读者能够对这些内容发扬“上下求索”的精神主动搞定它！

本部分的章节安排如下：

第九章是 BREW 原理，在这一章里将一一的展示 BREW 所面临的问题，以及采用什么样的方式来解决这些问题，同时也介绍了一些 BREW 实现方式的特性。这一章是本篇的基础。

第十章是探秘 BREW 接口，在 BREW 原理一章的基础之上，这一章将介绍 BREW 接口实现的细节。

第十一章是 Shell 内幕，这一章将介绍 Shell 的作用，以及实现的方式。这里还将介绍 BREW 应用程序与接口之间的关系。

第十二章是图形与多媒体系统，这一章将介绍图形系统中的 Bitmap、DIB 和 Font 等实现的原理，同时还将介绍字符编码等相关内容。

第九章 BREW 原理

我们 PC 的 Windows 操作系统功能是如此的强大，以至于我们可以获得任何一个我们想要的程序，任何一个人都可以为 Windows 开发应用程序，或者给自己，或者为他人。而这一切源于操作系统的开放性和硬件平台的不断发展，尤其是存储器的发展，使得我们在编写程序的时候不必在意需要多大的存储空间了。

然而，嵌入式系统可就没那么幸运了。至今为止，在嵌入式系统里仍然没有一个能够像 Windows 这样应用如此广泛的操作系统，也没有可以不考虑存储空间的硬件平台。在数以亿计的嵌入式设备使用者中，都还在用着一成不变的应用程序，单调同时也令人乏味。我们能不能也像在 Windows 下面一样，在嵌入式系统中可以安装应用程序呢？应该怎样克服嵌入式系统的限制而实现这个功能呢？

有梦想才会不断的追求！我们知道，在 Windows 中程序都是以文件的形式存储在文件系统中的，然后通过操作系统控制这些程序的运行，我们可以说它的程序是“分散式”的。而在嵌入式系统中通常是将程序烧录在一个 Flash 芯片中，文件系统在另一个 Flash 芯片中（也可以二者在同一个芯片中），CPU 是直接从程序 Flash 芯片中读取指令执行的，没有经过文件系统，我们可以叫这种程序是“一体式”的。Windows 的“分散式”程序体通过文件的形式存在，可以把程序的不同部分分割成不同的文件，当我们只需要更新一个模块内容的时候，只更新这个文件就可以了。熟悉它的朋友们可能已经知道了，这个文件就是在 Windows 操作系统中的 DLL 文件。这样的方式可以很容易的实现程序分发，这给了我们一个很好的启示：嵌入式系统中也有文件系统，把程序放在文件系统里不就可以了吗？真是个好主意！

在我们庆幸找到了好方法的时候，问题不偏不倚地出现了：系统如何运行文件系统中的程序，文件中的程序又如何调用平台中的函数？要实现“分散式”的程序运行，这两个问题是必须要解决的，而其中第二个问题就更为重要了。或许您现在还不是十分的明白这些问题的意义，不要着急，这一章里我会逐一的向您讲解如何理解并解决这两个问题。当然，现在我们知道 BREW 已经在嵌入式系统中解决了这两个问题，从现在开始就让我们沿着开发者的足迹去追寻 BREW 的本质吧。

9.1 平台的作用

如果想要清楚的了解我们在嵌入式系统中所面临的问题，那么我们就首先需要了解“分散式”系统的结构。一个“分散式”系统需要有三个部分组成：平台、软件开发工具包（SDK：Software Development Kit）和应用程序。“分散式”应用程序的运行需要平台的支持，就像是 DLL 文件只有在 Windows 操作系统平台下才有作用，而到了 Linux 平台则不起任何作用一样；应用程序则通过 SDK 进行开发，开发出的源程序经过编译之后可以运行在运行平台之上。平台又分为开发平台和运行平台，开发平台是 SDK 运行的平台，用来开发可以在运行平台上运行的应用程序，对于一些系统还会提供模拟运行平台的模拟器，以便于在没有显而易见的运行设备的时候也可以看到开发的效果；运行平台是应用程序运行的平台，它提供应用程序运行的环境，同时肩负着控制应用程序的作用。开发平台和运行平台可以是同一个平台，也可以两个不同的平台，比如现在的 Winows 平台的应用就可以使用 VC 等工具开发基于 Windows 的应用程序，而 BREW SDK 则是运行在 Windows 环境下，但应用程序却在嵌入式系统中运行。它们之间的关系如下图：

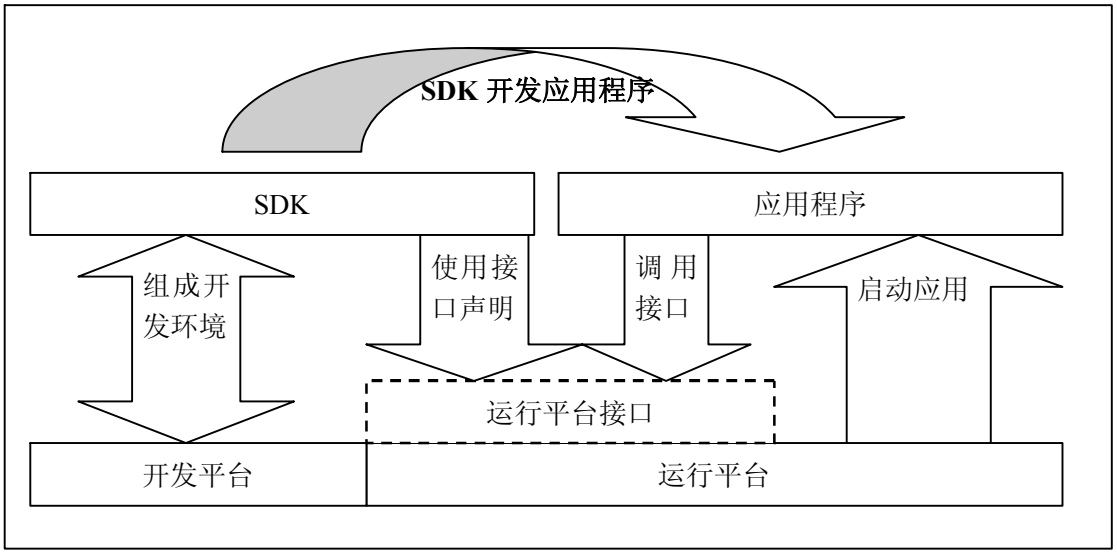

图 9.1 分散式系统结构图

从这个图中我们可以看出，SDK 需要使用运行平台的接口声明来开发应用程序，运行平台负责根据用户的输入启动应用程序，而应用程序则通过运行平台的接口调用运行平台的函数库来实现功能，我们的问题主要集中在应用程序和运行平台的互动关系上。

从前面的编译器基础一章中我们可以知道，在固定链接的模式下，各个函数的地址是固定的，我们可以在同一个映像文件中调用任何函数。而存储在文件系统中的程序就不一样了，文件系统中的程序没有固定的位置并且地址也不连续，我们该怎样实现应用程序的启动呢？可选的方案就是将应用程序复制到一个连续的内存块中去，然后在内存中执行程序。在这里需要特别的说明一下，在 PC 的 Windows 操作系统中，Windows 将全部的程序载入内存中运行，并且其中包含了复杂的内存管理功能，但是在嵌入式系统中通常程序是在 Flash 芯片中运行的，只有可读写的数据是放在 RAM 中的，具体的细节可以参考编译器基础一章。BREW 主要是应用程序在嵌入式系统中的，因此将程序复制到内存中执行是需要特殊处理的。这个特殊的处理就是我们所面临的第一个问题了——系统如何运行文件系统中的程序。

在我一开始理解 BREW 的时候，我曾经认为系统如何运行存在于文件系统中的程序是我们所面临的主要问题，但当深入 BREW 内部的时候发现根本不是这么回事。现在我们可以假设运行平台分配了一个足够大的内存，这块内存地址是已知的，可以想象的到我们可以从这个地址开始执行程序。现在我们暂时忽略那些特殊的处理，文件系统中的这个程序现在正在运行，就像在 Flash 中的程序一样的在运行。从理论上来讲这个是行得通的，因此系统如何运行文件系统中的程序的问题并不是我们所面临的难题。实际上 BREW 也是按照这个思路做的，只是实现时还有细节的东西在，关于这些细节我们将在 Shell 内幕一章进行详细的介绍。

进一步的，假设现在程序运行到了需要调用平台函数的时候了，问题就来了，由于当前的应用程序是开发者使用 SDK 开发的，就像平台不知道应用程序的地址一样，应用程序也不知道平台的函数地址，因此，我们现在面临的问题是怎么能够知道应用程序中所调用的平台函数的地址。SDK 可以提供运行平台中的每个函数的地址吗？可以提供，但是行不通。因为平台是会经常升级的，导致每个函数的链接地址不固定，如果由 SDK 提供所有函数的地址带来的问题是，只要运行平台一升级，那么 SDK 和应用程序都需要同时升级。如果这样的话我们的分散式程序就不能实现“分散式”的升级了，这种程序的运行方式也就没有任何意义了。看来我们还要寻找更为高级的方法，这种方法要能够提供一种应用程序与运行平台之间无关的机制。我们现在所需要的这个“机制”就是第二个问题了——文件系统中的程

序如何调用平台的函数。

从分散式系统结构图中我们可以看到，SDK 使用的是运行平台的接口声明，应用程序调用真正的运行平台接口。或者换句话说开发过程中使用运行平台的接口声明，而在运行时应用程序使用真正的二进制接口，并且在二进制层面调用接口函数。那么，现在无论是 SDK 还是应用程序都与接口相关，那么，可以想象的到的解决第二个问题的方式就是让接口和接口实现之间分离。接口与实现间分离的方法就是 BREW 的核心，也是接下来我们主要阐述的议题。

13.2 软件分发和 C 语言

为了更好的理解“实现接口和实现分离”所面临的问题，让我们先来看看通常的 C 语言软件库是如何分发的，这对于我们的理解非常有用。为了能够更加清楚地理解问题，我们在接下来几节的论述中不会仅仅局限于嵌入式系统，因为同样的问题也存在于 PC 系统中，更为重要的是 PC 系统所面临的问题更加典型，并且这些问题在软件系统中是具有普遍性的。

想象现在有一个 C 语言库的开发厂商，它开发了一个算法，可以在 O(1)时间效率内实现子字符串的搜索，O(1)时间效率的意思是指搜索时间为常数，与目标字符串的长度没有关系。为了实现这个功能，软件厂商生成了一个头文件 FastString.h，包含如下内容：

FastString.h – 接口声明文件第一版

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

// 要求使用者不能直接使用此结构体中的内容
typedef struct _IFastString {
    char *m_pString;    // 指向字符串的指针
} IFastString;

// 创建目标字符串对象
void IFastString_CreateObject(IFastString * pIFastString, char *pStr);

// 释放目标字符串对象
void IFastString_Release(IFastString *pIFastString);

// 获取目标字符串长度
int IFastString_GetLength(IFastString *pIFastString);

// 查找字符串，返回偏移量
int IFastString_Find(IFastString *pIFastString, char *pFindStr);
#endif // FASTSTRING_H_
```

除了这个头文件之外软件厂商还提供了接口的实现文件 FastString.c

FastString.c – 接口实现文件第一版

```
#include ”FastString.h”
#include <string.h>
```

```
// 创建目标字符串对象
void IFastString_ CreateObject (IFastString * pIFastString, char *pStr)
{
	IFastString *pMe = pIFastString;

	if(pMe == NULL || pStr == NULL)
	{
		return;
	}

	pMe->m_pString = malloc(strlen(pStr) +1);
	strcpy(pMe->m_pString,pStr);
}

// 释放目标字符串对象
void IFastString_Release(IFastString *pIFastString)
{
	IFastString *pMe = pIFastString;

	if(pMe == NULL)
	{
		return;
	}

	if(pMe->m_pString)
	{
		free(pMe->m_pString);
	}
}

// 获取目标字符串长度
int IFastString_GetLength(IFastString *pIFastString)
{
	IFastString *pMe = pIFastString;

	if(pMe == NULL)
	{
		return;
	}

	return strlen(pMe->m_pString);
}

// 查找字符串，返回偏移量
```

```
int IFastString_Find(IFastString *pIFastString, char *pFindStr)
{
    IFastString *pMe = pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    // 搜索算法省略，因为这里仅仅假设存在这样的一个搜索算法
}
```

这个接口总共有四个接口：CreateObject、Release、GetLength 和 Find。CreateObject 用来创建 IFastString 接口，从实现中我们可以看到它构建了 IFastString 结构体的内容。Release 用来释放由 CreatObject 分配的内存。GetLength 获得字符串的长度。Find 用来在目标字符串中查找指定的字符串。在这个接口的实现中，使用了一个初始化的技巧（CreateObject 和 Release），目的是为了再使用前构建 IFastString 结构体，使用后可以通过 Release 释放构建时分配的内存。

一般来讲这个库的使用者会将.lib 库链接到自己的工程中，通过接口声明的头文件使用库中的函数，这是一个非常可行的做法。这样做带来的结果是库的可执行代码将成为客户应应用程序中不可分割的一部分。

现在假设对于 FastString 库文件占用了大约 16M 的代码空间（这里假设为了完成 O(1)算法可能需要十分复杂的程序，并且采取了一些空间换时间的策略等，这可能占用大量的存储空间）。如图 9.2 所示，如果现在有三个应用程序都在使用 FastString 库，那么每一个应用程序都将使用 16M 的空间来存储这些代码，总共花费了 48M 的空间。如果一个用户安装了这三个程序，也就是说有 32M 的空间浪费了，去存储了同样的 FastString.lib 中的代码。

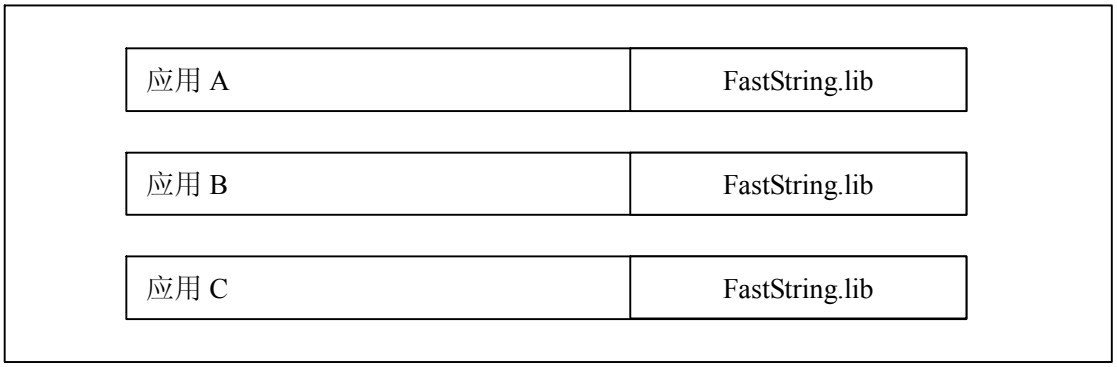

图 9.2 多个应用程序使用 FastString 库

在这种情况下的另一个问题是，如果当前 FastString 库的厂商发现了程序中的缺陷，那么就没有任何办法可以替换已经存在的缺陷代码。一旦 FastString 的代码链接到了应用程序中，我们就不可能在用户的设备上替换这部分的代码。因此，库厂商不得不重新为每个应用程序的开发者广播发布新的库文件，并且希望他们能够重新的编译他们的应用程序，以便能够使用新的代码。很显然，这可真是一件麻烦的事情，一旦应用程序开发者链接了这个库文件，FastString 便失去了模块化的特征。跟进一步说，这在嵌入式系统中是完全不可能的，在这里 FastString 的角色就是运行平台，我们总不能每一个应用程序都包含一个运行平台去啊。

13.3 动态链接

解决上面问题的一种技术是使用动态链接技术（Dynamic Link）将 FastString 包含起来，这种技术的典型应用是 Windows 操作系统中的动态链接库（DLL 文件）。这种方法是将 FastString 源文件编译成特殊的独立的二进制文件，并强迫 FastString 将所有的接口从二进制文件中引用出去，建立相应的引出表，以便于在运行时把每个接口的名字映射到对应的二进制接口地址上。与此同时还需要为使用者生成相应的引入库，通过这个引入库 FastString 的使用者可以获得 FastString 中每个接口的符号。引入库中并没有包含实际的代码，它只是简单的包含 FastString 引出的符号名字。当客户链接引入库的时候，这些符号信息会加入到当前的可执行文件中，运行时动态的装载 FastString 二进制库文件，并在执行时调用相应的程序。当然这些需要一些辅助工具的支持，例如编译器的支持等。此时应用程序的结构如图 9.3：

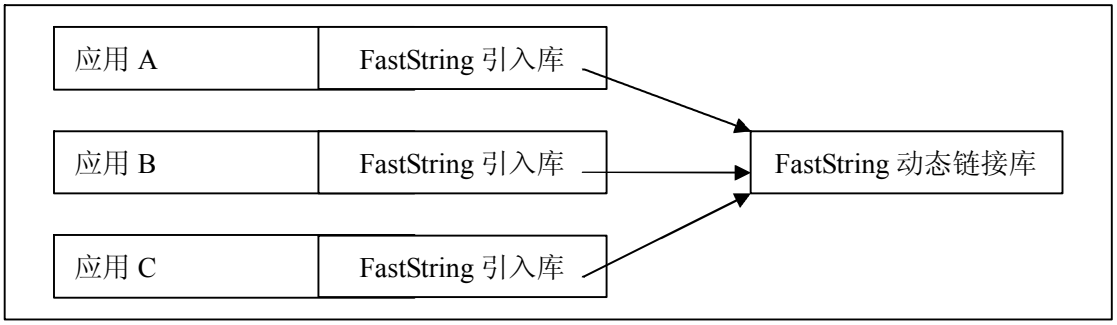

图 9.3 动态链接示意图

上图就是 FastString 在动态链接方式下的运行模式，这里面的引入库非常的小，所以可以忽略它占用的空间。在这种情况下，FastString 的代码库就只需要一份了。运行时所有的应用程序调用同一个库中的内容，理论上当发现 FastString 中有缺陷的时候，我们可以更新 FastString 二进制组件而不影响应用程序。可以看到，我们已经迈出了重要一步，不过还没有完全解决我们所面临的问题。

13.4 封装性

我们现在已经找到了一种可以实现动态链接的方法，那么下一个问题则与封装有关。考虑这样的情形：一个组织使用了 FastString，同时希望能够在 2 个月内完成开发和测试。假设在这两个月中，某些具有特殊的怀疑精神的开发人员打算在他们的应用程序上测试一下 FastString 的性能，以便于测试 O(1)时间效率的搜索算法。令人惊讶的是 Find 方法的搜索速度很快，并且与字符串的长度无关，但是 GetLength 方法的速度不是很理想，因为在 GetLength 方法中使用的是 strlen 计算字符串的长度，它查找字符串中的 NULL 结束符，它的算法需要遍历正个字符串的内容，因此它的执行效率是 O(n)，当字符串很长，而且调用次数很多的时候，执行的速度很慢。于是开发人员要求厂商提高 GetLength 操作的执行速度，使它在常数时间内完成。但是现在有一个问题，开发人员已经开发完成了他们的应用程序，他们不希望由于使用新的 GetLength 方法而更改任何现有的程序。而且其他的厂商可能已经发布了使用这个现有版本的基于 FastString 的产品，从任何方面将库厂商都不应该影响这些已经面世的产品。

这个时候我们要查看我们的 FastString 的实现，以便确定哪些可以改变，哪些不可以改变。幸运的是，我们已经要求使用者不能直接使用 IFastString 结构体中的内容，假设所有的

使用者都遵循了这一约定，于是我们很快的修改了 GetLength 的方法，将头文件改成了如下的方式（未修改部分未列出）：

FastString.h – 接口声明文件第二版

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

// 要求使用者不能直接使用此结构体中的内容
typedef struct _IFastString {
    char *m_pString;    // 指向字符串的指针
    int m_nLen;         // 存储字符串的长度
} IFastString;

#endif // FASTSTRING_H_
```

将实现文件改成了如下方式（未修改部分未列出）：

FastString.c – 接口实现文件第二版

```
// 创建目标字符串对象
void IFastString_ CreateObject (IFastString * pIFastString, char *pStr)
{
    IFastString *pMe = pIFastString;

    if(pMe == NULL || pStr == NULL)
    {
        return;
    }

    pMe->m_nLen = strlen(pStr);
    pMe->m_pString = malloc(pMe->m_nLen +1);
    strcpy(pMe->m_pString,pStr);
}

// 获取目标字符串长度
int IFastString_GetLength(IFastString *pIFastString)
{
    IFastString *pMe = pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    return pMe->m_nLen;
}
```

很快的修改后重新发布了 FastString 的第二个版本。在这里一个显著的改进是在 CreateObject 时将字符串长度存储起来了，当用户调用 GetLength 方法时直接返回存储的长

度。这样没有修改任何接口的内容，因此应用程序也就不需要修改了。

客户收到了第二版的 FastString 后，替换了 FastString 的动态链接库，重新编译链接全部的应用程序，测试后发现不但原始代码不需要任何修改，而且 GetLength 方法的速度也大大加快了。最终这个第二版的 FastString 会随着这个产品而发布到用户手中。在安装应用程序的时候，第二版的 FastString 会将第一版的替换掉。这通常不会有问题，因为修改并没有影响公开的接口，因该只会增强原先已经安装的使用 FastString 的应用程序的功能而以。

请想象这样的情形，当用户更新了第二版的 FastString 后，运行新版的应用程序，用户惊喜的发现程序运行的速度提高了。然后用户关闭了新的应用程序，而打开了另一个以前安装的使用旧版本 FastSting 的应用程序。现在的 FastString 已经替换成了第二版，因此用户发现这个应用程序的性能也增强了。然而不久异常出现了，系统出现了未知的错误。不过没关系，对于已经习惯了现代商业软件的人士来说，这不算什么问题，于是重新卸载并重新安装了两个应用程序，还是不起作用啊，异常依然发生！到底是怎么回事？

原因在于 IFastString 结构体的修改。在未修改前 sizeof(IFastString) == 4，因为只有一个 char *m_ pString 变量（假设系统是 32 位的）。修改后 sizeof(IFastString) == 8，增加了 4 个字节。新版本的软件已经重新编译了，因此相应的 IFastString 结构体已经增加了空间。但是对于使用第一版 FastString 编译的应用程序来说，此时在应用程序里面分配的 IFastString 的空间依然是 4，于是当第一版的应用程序调用第二版 FastString 的时候，将本该属于其他用途的 4 个字节用作了 m_nLen 的区间，这是十分粗暴的，产生异常也就不足为奇了。

还记得前面的约定吗？我们要求 FastString 的使用者不可以直接对 IFastString 结构体中的数据进行直接操作，以达到应用程序与数据结构间的无关性。但实际上这样的约定基本上不可能被遵守，因为在实际中不管出于什么样的目的，总会有些开发者直接使用结构体中的内容（这些开发者使用的使 FastString 的 C 语言库文件形式，没有使用动态链接技术）。加上前面的异常，这一切的根源是我们没有一个实现二进制数据封装的方式。如果现在能够有一种可以将全部的数据结构封装在 FastString 内部的方法就好了。C 语言是灵活的，只要我们找到了问题，我们就可以实现它。于是第三版的 FastString 新鲜出炉了，首先是头文件：

FastString.h – **接口声明文件第三版**

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

typedef void IFastString;

// 创建目标字符串对象
void IFastString_CreateObject(IFastString ** ppIFastString, char *pStr);

// 释放目标字符串对象
void IFastString_Release(IFastString *pIFastString);

// 获取目标字符串长度
int IFastString_GetLength(IFastString *pIFastString);

// 查找字符串，返回偏移量
int IFastString_Find(IFastString *pIFastString, char *pFindStr);
#endif // FASTSTRING_H_
```

接下来是实现的源文件：

FastString.c – 接口实现文件第三版

```
#include ”FastString.h”
#include <string.h>

typedef struct _CFastString {
	char *m_pString;	// 指向字符串的指针
	int m_nLen;		// 存储字符串的长度
} CFastString;

// 创建目标字符串对象
void IFastString_CreateObject (IFastString **ppIFastString, char *pStr)
{
	CFastString *pMe = malloc(sizeof(CFastString));

	if(pMe == NULL || pStr == NULL)
	{
		return;
	}

	pMe->m_nLen = strlen(pStr);
	pMe->m_pString = malloc(pMe->m_nLen +1);
	strcpy(pMe->m_pString,pStr);

	* ppIFastString = (IFastString *)pMe;
}

// 释放目标字符串对象
void IFastString_Release(IFastString *pIFastString)
{
	CFastString *pMe = (CFastString *)pIFastString;

	if(pMe == NULL)
	{
		return;
	}

	if(pMe->m_pString)
	{
		free(pMe->m_pString);
	}

	free(pMe)
}
```

```
// 获取目标字符串长度
int IFastString_GetLength(IFastString *pIFastString)
{
    CFastString *pMe = (CFastString *)pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    return strlen(pMe->m_pString);
}

// 查找字符串，返回偏移量
int IFastString_Find(IFastString *pIFastString, char *pFindStr)
{
    CFastString *pMe = (CFastString *)pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    // 搜索算法省略，因为这里仅仅假设存在这样的一个搜索算法
}
```

在这个实现中，我们使用了 CFastString 做为 FastString 的内部数据结构，定义了 void 型的 IFastString 类型做为接口指针传递，还有重要的一条是通过 CreateObject 获得数据结构的存储空间。通过这样的实现方法，我们将全部的数据类型封装在了 FastString 库中，这样，无论新的还是老的应用程序，使用的都是统一的 IFastString 指针，真正的数据是在 CreateObject 中进行创建的，也就不会出现上面的两种情况了。对于 FastString 的客户来说，他们所能看到的就是 IFastString 的 void 类型和四个接口函数，内部的 CFastString 的结构已经被隐藏起来了。不过这个第三版的 FastString 修改了接口函数 CreateObject，因此不能够与前两版兼容。不过不要紧，我们现在只是在说明一个更好的方法而已。

13.5 虚拟函数表

封装性的本质是实现了接口与实现之间在二进制层次的分离，第三版的 FastString 似乎已经解决了我们所面临的第二个问题。不过现在我们的接口仍然在使用着动态链接用的引入库文件，而且相应的动态链接库也需要提供由符号名到二进制函数地址映射的内容，为了支持这些特性，我们必须要修改相应的编译器才行。修改编译器，很复杂不是吗？

让我们再回到嵌入式系统上来吧，ARM CPU 是在嵌入式系统中使用最广泛的 CPU，因此相应的 ARM 编译器也是应用最广泛的，基本上成为了一种通用的编译器。我们怎么修改这个编译器呢？似乎难度有点大。更近一步的，嵌入式系统中大大小小的 CPU 有好多种，我们也不可能把所有的这些编译器都修改了啊，看来修改编译器的可能性不大。

又一次让我们体验到了理想与现实之间的差距！不过别灰心，看看我们现在的接口，它是二进制层面上的，我们能不能把这个引入库变成标准的 C 语言的头文件同时又能够实现接口与实现间的分离呢？如果真的能够实现这样的方法，那么将意味着我们可以通过通用编译器来实现动态链接的技术。这可真是令人兴奋，这个方法要比动态链接技术还要好。

按照这个思路进一步分析，如果将现在的第三版 FastString 使用的 C 头文件做为标准的接口文件，那么将意味着各个接口需要静态的链接到应用程序中，接口和实现之间还是没有实现分离，难道我们转了一圈又回到原点了？真的又回来了，不过，不同的是我们现在已经得到了第三版的 FastString。现在我们已经知道了通过 CreateObject 来获得内部的空间，那么我们是否也可以通过一个函数来获得接口呢？可以，这个技术就是虚拟函数表（VTBL）。

这可真是“山重水复疑无路，柳暗花明又一村”啊，先看看我们这个第四版的 FastString 的头文件吧：

FastString.h – **接口声明文件第四版**

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

typedef struct _IFastString IFastString;
typedef struct _IFastStringVtbl IFastStringVtbl;
typedef void (*PFNCreateObject)( IFastString **ppIFastString, char *pStr);

struct _IFastString
{
   struct IFastStringVtbl    *pvt;
};

struct _IFastStringVtbl
{
      void (*Release) (IFastString *pIFastString);
      int (*GetLength) (IFastString *pIFastString);
      int (*Find) (IFastString *pIFastString, char *pFindStr);
};

// 释放目标字符串对象
#define IFASTSTRING_Release(p)    ((IFastString*)p->pvt)->Release(p)

// 获取目标字符串长度
#define IFASTSTRING_GetLength(p)    ((IFastString*)p->pvt)->GetLength(p)

// 查找字符串，返回偏移量
#define IFASTSTRING_Find(p,s)    ((IFastString*)p->pvt)->Find(p,s)
#endif // FASTSTRING_H_
```

接下来是 C 语言的源文件：

FastString.c – **接口实现文件第四版**

```
#include ”FastString.h”
#include <string.h>
```

```
typedef struct _CFastString {
	IFastStringVtbl *pvt; // 指向虚拟函数表的指针
	char *m_pString;	// 指向字符串的指针
	int m_nLen;		// 存储字符串的长度
} CFastString;

// 函数声明
static void IFastString_Release(IFastString *pIFastString);
static int IFastString_GetLength(IFastString *pIFastString)
static int IFastString_Find(IFastString *pIFastString, char *pFindStr);

IFastStringVtbl gvtFastString = { IFastString_Release,
                                  IFastString_GetLength,
                                  IFastString_Find
                                 };

// 创建目标字符串对象
void IFastString_CreateObject (IFastString **ppIFastString, char *pStr)
{
	CFastString *pMe = malloc(sizeof(CFastString));

	if(pMe == NULL || pStr == NULL)
	{
		return;
	}

	pMe->pvt = &gvtFastString;
	pMe->m_nLen = strlen(pStr);
	pMe->m_pString = malloc(pMe->m_nLen +1);
	strcpy(pMe->m_pString,pStr);

	* ppIFastString = (IFastString *)pMe;
}

// 释放目标字符串对象
static void IFastString_Release(IFastString *pIFastString)
{
	CFastString *pMe = (CFastString *)pIFastString;

	if(pMe == NULL)
	{
		return;
	}
```

```
    if(pMe->m_pString)
    {
        free(pMe->m_pString);
    }

    free(pMe)
}

// 获取目标字符串长度
static int IFastString_GetLength(IFastString *pIFastString)
{
    CFastString *pMe = (CFastString *)pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    return strlen(pMe->m_pString);
}

// 查找字符串，返回偏移量
static int IFastString_Find(IFastString *pIFastString, char *pFindStr)
{
    CFastString *pMe = (CFastString *)pIFastString;

    if(pMe == NULL)
    {
        return;
    }

    // 搜索算法省略，因为这里仅仅假设存在这样的一个搜索算法
}
```

首先对这个第四版的 FastString 程序做一个说明。在 FastString 头文件中我们定义了两个类型：IFastString 和 IFastStringVtbl。IFastStringVtbl 类型是虚拟函数表的类型，IFastString 中包含了指向虚拟函数表的指针。在接口定义的时候，我们使用了((IFastString*)p->pvt)来调用虚拟函数表中的函数指针，这也说明了如果要使用接口，必须要提供 IFastString 的指针类型。可以看出 Release、GetLength 和 Find 已经实现了在 C 语言定义的接口与实现函数之间的分离。

接着看一下源文件中的情况。CFastString 结构体与第三版的 FastString 不同的是增加了一个 IFastStringVtbl 类型的指针，而且这个指针在结构体的最顶部，如果将 IFastString 和 CFastString 对比一下我们可以发现他们都在顶部包含了 IFastStringVtbl 的指针，这就意味着 CFastString 是 IFastString 的超集。这一点是很重要的，我们可以看见在 CreateObject 函数中

返回的 IFastString 指针其实是指向 CFastString 的指针的，只有 CFastString 是 IFastString 的超集的时候我们才可以这么做。在这个源文件中还定义了一个 IFastStringVtbl 的变量 gvtFastString，并为这个变量初始化成了各个对应的函数，这个变量就使我们的虚拟函数表。虚拟函数表的示意图如下：

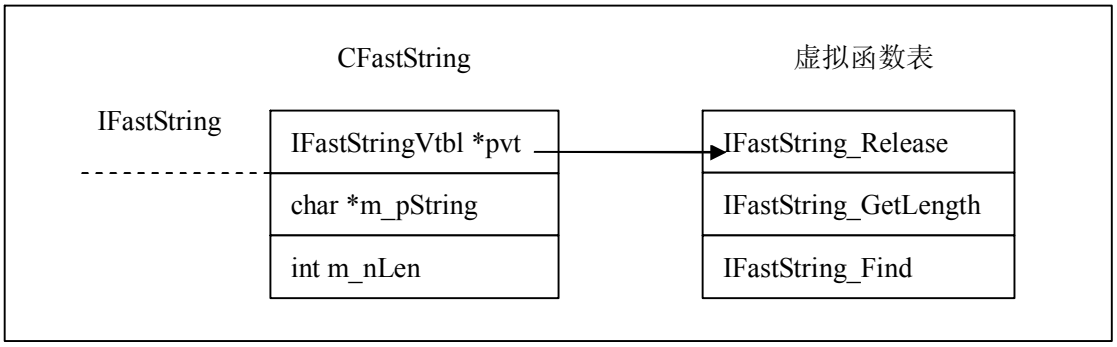

图 9.4 虚拟函数表

在这个第四版的 FastString 中，我们可以看到，除了 CreateObject 成员之外，其余的三个成员函数都已经添加到了虚拟函数表中，而且这个虚拟函数表还可以随着需求的增加而进行无限的扩大，我们只付出了一个函数 CreateObject 的代价就实现了无限多个接口和实现之间的分离了。

由于用户需要使用 CreateObject 来获得 IFastString 的指针，因此我们没有办法将其与实现分离开，怎么办？现在只有这一根“线”还在困扰着我们，我们难道要功败垂成了吗？当然不能了。开动脑筋，回到我们应用程序的启动过程，对于一个程序，不管是由 main 函数或者其他的什么函数做为启动函数，都允许启动的时候传递参数，可能您已经想到了吧，我们把这个 CreateObject 函数做为参数传递给应用程序不就可以了吗？恍然大悟！这也就是为什么我们将 CreateObject 函数定义成了一个 PFNCreateObject 的函数类型的原因了，目的就是为了让使用者可以在应用程序中定义这种类型的函数指针。

现在我们的应用程序、接口和实现之间已经分离了，中间使用了 CreateObject 这根细线连接了起来，只要接口不变，应用程序和实现之间就不会有任何的联系，包括二进制层面和 C 语言层面的。只不过这要求我们应用程序和接口的实现之间需要使用同一种编译器，或许这就叫做有得有失吧，不过对于嵌入式系统来说这是必须的，因为没有哪一种编译器可以支持全部的嵌入式 CPU。

13.6 支持多个接口

到现在为止所展示的技术已经解决了我们所面临的问题，不过对于一个平台来说不可能只有一个 FastString 接口，可能还有诸如 FastNumber 的接口。我们总不能把 FastString 和 FastNumber 两个接口的 CreateObject 都做为参数传递给应用程序的启动函数吧？看来我们现有的 FastString 需要进行一些扩展，来实现只传递一个参数给应用程序就可以创建多个接口的功能。在这里我们将增加一个叫做 Shell 的接口来管理其他的接口，相关的代码如下：

shell.h – 接口声明文件

```
#ifndef SHELL_H_
#define SHELL_H_

#define CLASSID_FASTSTRING 0x00000001
```

```
#define CLASSID_FASTNUMBER 0x00000002

typedef struct _IShell IShell;
typedef struct _IShellVtbl IShellVtbl;

struct _IShell
{
   struct IShellVtbl    *pvt;
};

struct _IShellVtbl
{
      void (*CreateInstance) (IShell *pIShell,
                              int nClassID,
                              void **ppObj,
                              unsigned int nUserData);
};

// 释放目标字符串对象
#define ISHELL_CreateInstance(p,c,pp,u)    ((IShell*)p->pvt)->CreateInstance(p,c,pp,u)
#endif // SHELL_H_
```

源文件如下：

Shell.c – 接口实现文件

```
#include ”Shell.h”

typedef struct _CShell {
      IShellVtbl *pvt; // 指向虚拟函数表的指针
} CShell;

// 函数声明
static void IShell_CreateInstance(IShell *pIShell,
                                  int nClassID,
                                  void **ppObj,
                                  unsigned int nUserData);

IShellVtbl gvtShell = { IShell_CreateInstance };

// 创建 Shell 对象
void IShell_CreateObject (void**ppObj, unsigned int nUserData)
{
      CShell *pMe = malloc(sizeof(CShell));

      if(pMe == NULL)
      {
```

```
        return;
    }

    pMe->pvt = &gvtShell;

    * ppObj = (void*)pMe;
}

// 创建由 ClsID 指定的
static void IShell_CreateInstance(IShell *pIShell,
                                  int nClassID,
                                  void **ppObj,
                                  unsigned int nUserData)
{
    CShell *pMe = (CShell *)pIShell;

    if(pMe == NULL)
    {
        return;
    }

    switch(nClassID){
        case CLASSID_FASTSTRING:
            IFastString_CreateObject((IFastString **)ppObj, (char *)nUserData);
            break;

        case CLASSID_FASTNUMBER:
            IFastNumber_CreateObject((IFastNumber **)ppObj, nUserData);
            break;

        default:
            break;
    }
}
```

在这个 Shell 接口中，我们定义了一个接口函数 CreateInstance。它的作用是通过参数 nClassID 来创建指定的接口实例。IShell_CreateObject 函数用来创建 Shell 接口本身，在使用的时候必须由系统直接调用 IShell_CreateObject 来产生 Shell 对象，然后再通过 Shell 接口来创建其他的接口（如 FastString）。同时在这里包含了一个在本书中尚未实现的接口 FastNumber，使用它仅仅是为了方便举例而已，因此有兴趣的读者可以仿照 FastString 接口实现 FastNumber 接口。

从 shell.h 和 shell.c 文件中可以看到，Shell 接口的实现方式与第四版的 FastString 实现方式是相同的。更进一步的，从 CreateInstance 接口函数的内部实现我们可以知道，它使用了一个 Class ID 来识别用户创建的是哪一个接口，并通过 switch 语句实现相关接口的 CreateObject 函数的调用。通过这样的 Shell 管理，应用程序只需要知道一个 Shell 接口的指

针就可以创建其他的接口了。换句话说，在启动应用程序的时候，我们先调用IShell_CreateObject 函数创建一个 Shell 指针，并将这个 Shell 指针做为参数传递给应用程序的启动函数，那么理所当然的，我们就可以在应用程序中使用 Shell 的接口ISHELL_CreateInstance 来创建其他的接口了。通过这样的方式，我们不但可以实现接口的管理工作，而且同时也为接口的扩展性提供了足够的灵活性。

13.7 接口的扩展性

到现在为止所展示的技术使得用户可以通过调用统一的 C 语言声明的接口，实现动态的二进制库的装载，这样可以无限制的升级库程序而不影响已有的应用程序，并且客户也不需要重新编译他们基于当前库文件所开发的应用程序，这对于创建一个复杂的运行平台来说是非常有用的。然而，这个接口却不能够随着时间而变化。这是因为客户在编译的时候需要有精确的接口定义，对接口的任何变化都需要客户重新编译他们的应用程序，以适应这种变化。更为糟糕的是，改变接口的定义完全违背了我们对接口封装性的要求。即便是最无伤大雅的变化，比如修改了接口的用途但是保留接口函数的原形不变，也会导致应用程序不再发生作用。这意味着接口的定义绝对不能改变，它既是语义上的约定，同时也是二进制层次上的约定。为了拥有一个稳定的，行为可预测性的运行时环境，接口的不变性这一要求是十分重要的。

尽管接口具有不变性的原则，但是我们通常需要在一个接口定义好之后，希望能够加入原先设计时没有预测到的新功能。此时我们可以利用对虚拟函数表布局结构的知识，只是简单的把新的方法追加到现有接口的底部，就可以实现对接口的扩展。考虑下面的 FastString 接口声明：

```
typedef struct _IFastString IFastString;
typedef struct _IFastStringVtbl IFastStringVtbl;

struct _IFastString
{
   struct IFastStringVtbl    *pvt;
};

struct _IFastStringVtbl
{
      void (*Release) (IFastString *pIFastString);
      int (*GetLength) (IFastString *pIFastString);
      int (*Find) (IFastString *pIFastString, char *pFindStr);
};
```

简单的修改接口 vtbl 的声明，在现有的结构体内增加新的接口函数的类型声明，这样得到的二进制声明结构是原有的声明的超集，因为新的方法总是出现在旧版本方法之后。在针对新方法的接口实现中，我们可以填充在这个结构体中的函数指针：

```
typedef struct _IFastString IFastString;
typedef struct _IFastStringVtbl IFastStringVtbl;

struct _IFastString
```

```
{
   struct IFastStringVtbl    *pvt;
};

struct _IFastStringVtbl
{
      // 第一版接口
      void (*Release) (IFastString *pIFastString);
      int (*GetLength) (IFastString *pIFastString);
      int (*Find) (IFastString *pIFastString, char *pFindStr);
      // 第二版接口
      int (*FindN)( IFastString *pIFastString, char *pFindStr, int n);
};
```

这种方式完全可以正常工作。在第一版接口上开发的应用程序将忽略前三个接口之外的其他接口的信息。当老的应用程序使用第二版接口实现的二进制程序的时候，它仍然可以正常的工作。然而，新的客户总是希望可以使用新的方法 FindN，以便于能够获得子字符串第 N 次出现的位置。如果此时应用程序的运行平台依然使用的是第一个版本的实现，那么不幸的，问题发生了。当调用一个未曾实现的接口时，显而易见的，程序崩溃了。

这项技术的问题是修改了公开的接口，从而影响了接口的封装性。就像是只修改了 C 语言函数的声明会产生编译错误一样，改变了二进制的接口结构也会引起运行时的代码错误。这意味着接口必须是不可改变的，一旦公开后就不能再变化。解决这个问题有两种方法：一是允许接口的实现暴露多个接口，或者换句话说就是如果需要扩展老的接口，那么就重新定义一组新的接口，这样的话就可以在不支持新接口的旧平台上运行的时候判断接口的有效性，当然，这要求应用程序要对创建接口时的异常进行处理；二是支持在运行时对接口版本的判断或者在应用程序和运行平台之间有一种版本的比较机制，比如通过提供一个可以获得接口版本的 API 来进行版本的比较工作。不管是使用哪一种方法，无疑都会增加开发的负担，包括接口的开发者和用户的开发者，因此最好的方法是尽可能的不要去修改已经定义好的接口。

13.8 资源管理

当我们在使用我们的接口的时候，我们还会遇到另外的一个问题，请看下面的代码：

```
IFastString *pFastString;
char TargetStr[] = “This is a test example only!”’

ISHELL_CreateInstance(pShell, CLASSID_FASTSTRING, (void **)&pFastString, (unsigned int)TargetStr);

(void)FastString_Test(pFastString);
IFASTSTRING_Release(pFastString);
```

相应的 FastString_Test 函数如下：

```
int FastString_Test(IFastString *pFastString)
{
```

```
    int nOffset;
    nOffset = IFASTSTRING_Find(pFastString, ”test”);

    IFASTSTRING_Release(pFastString);
    return nOffset;
}
```

在这个例子里，pFastString 在主函数中创建，同时做为参数传递给了 FastString_Test 函数，然后释放 pFastString，这很正常，没有什么问题。在 FastString_Test 函数中调用了 Find 方法，并且在使用函数退出的时候释放了 pFastString 指针，这很正常，也没有什么问题。然而，当我们将两者结合起来的时候，问题出现了：主函数中释放了一次接口指针，而在 FastString_Test 函数中也释放了一次同样的接口指针。换句话说，在这个程序中由于开发者的疏忽，对同一个接口释放了两次，这将导致不可预测的异常发生。这里面的问题是在增加了对接口指针的引用的时候，没有相应的处理机制来记录当前实例引用的次数，也就是当前接口实例的一个资源管理的问题。

或许您可以说这个问题可以通过使用者的细心来避免，那么我们再来看另外一个问题：如果当前的接口是可以共用的一些函数，比如这个接口中的方法全部是诸如 STRLEN 之类的重定义助手函数接口，并且在创建接口的时候需要分配一些公用的内存空间，那么，我们在每一次创建这个接口实例的时候，都必须分配不一样的存储空间吗？如果我们这样做了，实际上不会有什么问题，但是会浪费存储空间，因为他们本来是可以全部共用的。况且，如果我们全部都使用同一个指针在各个函数之间调用，那么对于这个指针来说使用起来会很危险的，因为我们不知道每一个函数是怎样处理这个指针的，这可真是太糟糕了。

为了解决这个问题，一个可行的做法是为每一个接口增加一个引用计数的管理机制。在增加了这个机制之后的第五版 FastString 实现如下：

FastString.h – **接口声明文件第五版**

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

typedef struct _IFastString IFastString;
typedef struct _IFastStringVtbl IFastStringVtbl;

struct _IFastString
{
   struct IFastStringVtbl    *pvt;
};

struct _IFastStringVtbl
{
    int (*AddRef) (IFastString *pIFastString);
    int (*Release) (IFastString *pIFastString);
    int (*GetLength) (IFastString *pIFastString);
    int (*Find) (IFastString *pIFastString, char *pFindStr);
};

// 增加接口指针的引用计数
```

```
#define IFASTSTRING_AddRef(p)    ((IFastString*)p->pvt)->AddRef(p)

// 释放目标字符串对象
#define IFASTSTRING_Release(p)    ((IFastString*)p->pvt)->Release(p)

// 获取目标字符串长度
#define IFASTSTRING_GetLength(p)    ((IFastString*)p->pvt)->GetLength(p)

// 查找字符串，返回偏移量
#define IFASTSTRING_Find(p,s)    ((IFastString*)p->pvt)->Find(p,s)
#endif // FASTSTRING_H_
```

相应的实现文件如下：

FastString.c – 接口实现文件第五版

```
#include ”FastString.h”
#include <string.h>

typedef struct _CFastString {
    IFastStringVtbl *pvt; // 指向虚拟函数表的指针
    int m_nRef;
    char *m_pString;      // 指向字符串的指针
    int m_nLen;           // 存储字符串的长度
} CFastString;

// 函数声明
static int IFastString_AddRef(IFastString *pIFastString);
static int IFastString_Release(IFastString *pIFastString);
static int IFastString_GetLength(IFastString *pIFastString)
static int IFastString_Find(IFastString *pIFastString, char *pFindStr);

IFastStringVtbl gvtFastString = { IFastString_AddRef,
                                  IFastString_Release,
                                  IFastString_GetLength,
                                  IFastString_Find
                                 };

// 创建目标字符串对象
void IFastString_CreateObject (IFastString **ppIFastString, unsigned int nUserData)
{
    CFastString *pMe = malloc(sizeof(CFastString));
    char *pStr = (char *)nUserData;

    if(pMe == NULL || pStr == NULL)
    {
        return;
```

```
        }

        pMe->pvt = &gvtFastString;
        pMe->m_nLen = strlen(pStr);
        pMe->m_pString = malloc(pMe->m_nLen +1);
        strcpy(pMe->m_pString,pStr);

        pMe->m_nRef = 1;
        * ppIFastString = (IFastString *)pMe;

}

// 增加接口指针的引用计数
static int IFastString_AddRef(IFastString *pIFastString)
{
        CFastString *pMe = (CFastString *)pIFastString;

        return (++pMe->m_nRef);
}

// 释放目标字符串对象
static int IFastString_Release(IFastString *pIFastString)
{
       CFastString *pMe = (CFastString *)pIFastString;

        if(--pMe->m_nRef >0)
        {
              return pMe->m_nRef;
        }

        if(pMe->m_pString)
        {
              free(pMe->m_pString);
        }

        free(pMe)
}

// 获取目标字符串长度
static int IFastString_GetLength(IFastString *pIFastString)
{
        CFastString *pMe = (CFastString *)pIFastString;

       return strlen(pMe->m_pString);
```

```
}

// 查找字符串，返回偏移量
static int IFastString_Find(IFastString *pIFastString, char *pFindStr)
{
    CFastString *pMe = (CFastString *)pIFastString;

    // 搜索算法省略，因为这里仅仅假设存在这样的一个搜索算法
}
```

在第五版的 FastString 中，我们主要是增加了接口 AddRef。这个接口是在增加指针引用的时候调用的，它仅仅增加了接口内部变量 m_nRef。同时为了实现相应的机制，在 Release 方法内增加了对引用计数的条件判断：如果当前的引用计数不为零，则直接返回引用计数的值，否则释放创建接口实例时所分配的内存。这样，当我们在使用上面的 FastString_Test 函数之前，调用 IFASTSTRING_AddRef 方法，就解决了资源管理的问题了。

增加这个 AddRef 方法还有一定的人为因素，因为只要程序员能够足够的注意，那么就不会存在资源管理的问题。但是，人生不如意十有八九，我们不能将全部的希望都寄托在程序员的身上，谁都有犯错误的时候。因此增加了 AddRef 的约定，它与 Release 方法相对应，形成了一种对称的编程“美感”，约定了如果要增加对指针的引用就调用 AddRef，对应的应该在适当的位置使用 Release 释放指针引用。

细心的读者可能还会发现第五版的 FastString 的实现中有两处不一样：一是 IFastString_CreateObject 函数的参数变化了，由原来的 char *类型成了现在的 unsigned int 类型，这样做的目的是为了统一在 Shell 程序中 CreateObject 的形式；二是 FastString 接口函数中 if(pMe == NULL)的判断去掉了，我们知道，在第一版的 FastString 中这句判断是很有必要的，它可以检测当前指针的有效性，那么想象一下对于我们的虚拟函数表 NULL 指针意味着什么？根据我们已有的虚拟函数表的知识，调用的接口是基于 IFastString 指针的相对偏移量，例如使用 NULL 指针来调用 Release 接口，那么实际上调用的是基于 0 地址的 4 字节偏移位置的函数，只有天知道这个地址中存储的是什么东西！因此，为接口传递空指针的时候会不可避免的发生异常，根本就不可能执行到接口函数，所以相应的判断是没有任何意义的。相应的示意图如下：

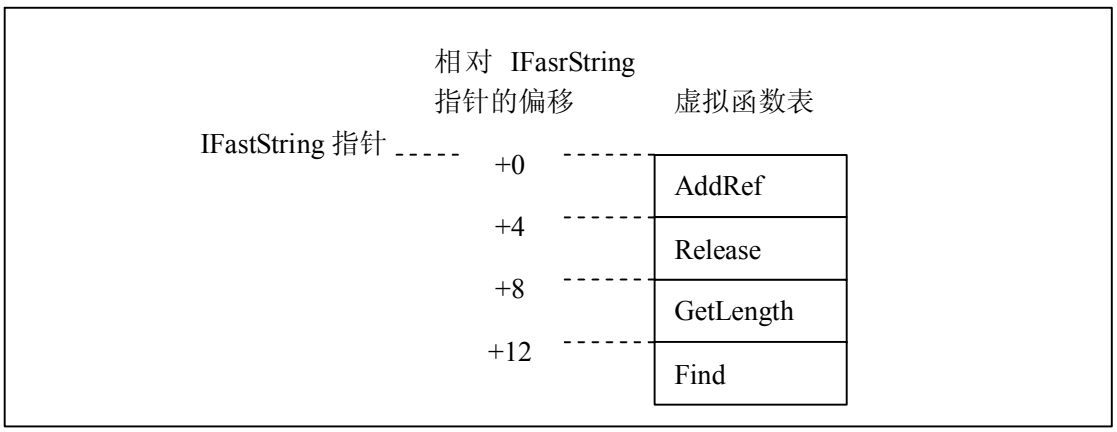

图 9.5 接口的偏移量

最后，再让我们清楚地看一下这种虚拟函数表所实现的总体框图吧，看看应用程序、接口定义以及接口实现之间的关系：

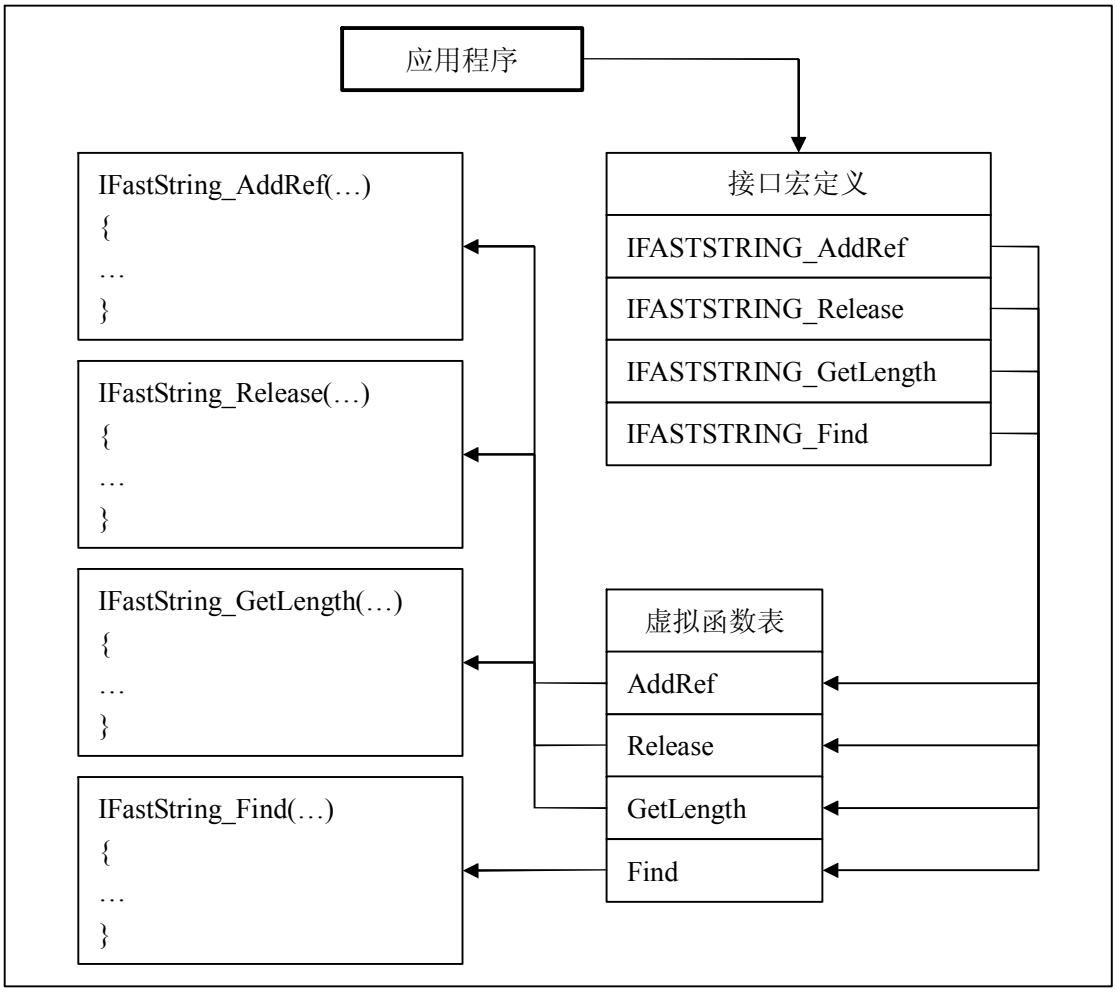

图 9.6 应用程序、接口定义以及接口实现的关系

至此，我们已经完成了全部需要解决的问题，最终我们实现的第五版 FastString 就是 BREW 接口的实现方法，只不过在实现的细节上有所不同。同时，我也相信各位在阅读了第二部分之后，一定会对这一部分的介绍颇有感触。接下来我们就趁热打铁，介绍 BREW 实现方法的一些高级特性，其中涉及了面向对象和 COM 组件的相关知识，对于没有接触过这两部分内容的读者来说，阅读可能会有一定困难。不过我会尽我最大的努力，争取用最容易理解的方式来阐述这些特性。

13.9 面向对象的特性

C 语言本身没有面向对象的特性，但是我们使用 C 语言开发的 BREW 就具有了面向对象的特性了。面向对象的主要特征是：数据抽象、继承和多态。数据抽象指的是使用一组数据和方法描述一个我们要表达的内容（对象），它的关键点在于将方法和数据结合也叫做封装；继承是指一个对象可以通过某种方式使用另一个对象中的方法和数据，它包含了语法上的继承和二进制层次的继承；多态是指通过虚拟函数实现的成员函数晚捆绑的特性，其核心的特征是成员函数的晚捆绑。

BREW 具有良好的数据抽象特性，它将全部的数据封装在了接口实现的内部，只将接口函数暴露给外部使用。这种方式是实现数据封装的理想方式，我想这一点可以从前面的“封装性”一节看出来。因此，BREW 已经具有了面向对象的第一个特性。

在这个最好的封装实现的基础上，根本没有任何数据直接暴露出来，因此，继承性就体

现在接口的继承上面了。由于 BREW 是使用 C 语言实现的，因此没有办法实现语法上的直接继承，但是它实现了二进制层次上的继承。继承在二进制上的表现就是在本数据结构中兼容另一个数据结构。例如，假设结构体 A 中包含了成员 int i，现在有结构体 B 包含了同样的 int i 类型的成员，并且后面紧跟了 int j 成员，此时 B 结构是 A 结构的超集，也可以说 B 结构继承了 A 结构。现在我们已经知道 BREW 接口的二进制结构，如果需要实现二进制的 BREW 继承，我们就可以通过定义一个接口 A 的超集来实现另一个接口 B，此时我们就可以说这个接口 B 是从接口 A 继承来的。我们可以通过接口 A 的定义使用接口 B 中的方法。因此说，BREW 具有面向对象的继承的特性，只不过不是在语法上，而是在二进制的数据结构层面。

面向对象的第三个特性就是多态，可以说这个特性是 BREW 天生的。我们知道 BREW 一开始就是通过 VTBL 来实现的，可以实现运行时的接口函数绑定（也就是迟绑定）。关于这里点我们可以从五个版本的 FastString 的实现中发现这一过程。晚捆绑与早捆绑的区别在于早捆绑在使用接口的时候再编译链接的时候就已经确切的知道每个接口函数的地址，这就类似于使用一个 C 语言的库；而晚捆绑则是在运行时才知道每个接口函数的确切地址，因为虚拟函数表是在运行时才赋值给相应的接口的。使用这种技术的一个动机是我们可以在运行的时候控制使用一个接口中的多个实现中的哪一个实现。如果当前 FastString 接口已经公开了，那么可能会有第三方的厂家按照这个接口规范，来实现一个更好的 FastString 接口。与最原始的 FastString 接口布局一样，新的实现的接口函数布局与老的一样，那么此时仅仅需要更新一个 Class ID 我们就可以切换到新的接口了。使用晚捆绑的另一个动机是，应用程序可以检测到当前的运行平台是否已经实现了此接口，并给予相应的处理，这样可以避免在接口为实现的平台上运行时引起致命的错误。

综上所述，BREW 具有了面向对象的三个主要特性，因此说他是具有面向对象性质的一种开发平台。

13.10 与 COM 的比较

COM 是 Windows 平台上实现的一种跨语言的开发机制，目前在 Windows 平台的底层，许多功能都是通过 COM 机制来实现的。COM 通过统一的、独立的接口定义语言（IDL：Interfase Definition Language）来定义统一的接口，并规定了相应的接口二进制规范，这样就可以按照这个二进制规范，通过各种不同的开发语言来实现 COM 程序的开发，而实现这种接口与实现之间完全分离的技术就是虚拟函数表（VTBL）。

熟悉 COM 开发的读者对 BREW 应该有一种似曾相识的感觉，没错，在我的定义中，BREW 就是一个简单版本的 COM。BREW 与 COM 相比，它们的核心思想是十分一致的，都具有接口与实现分离的特性，都使用了 VTBL 的技术等等。它们的不同点主要表现在以下几个方面：

第一，它们的接口与实现间的分离程度不同。COM 是接口和实现的完全分离，为此专门规范了统一的接口定义语言，因此而接口的实现可以采用任何一种开发语言，如 C/C++ 和 Java 等。而 BREW 则为了简化开发，使用了 C 语言形式的接口定义，这样就使得 BREW 的应用程序和实现都需要基于同样规则的编译器。或者换句话说，COM 实现是与开发语言无关的，而 BREW 的实现则是与语言相关的。

第二，它们的接口创建方式不同。COM 通过 Windows 注册表，使用文本的名字寻找相应的 Class ID（这个 Class ID 需要通过注册程序进行注册），例如 FastString 接口可以通过传递字符串“FastString”最为参数从而创建 FastString 接口，实现这个功能的基础是运行时的类型识别技术（RTTI，相关的内容可以参考 VC 的书籍）。而 BREW 则为了简便起见，仅仅

通过 Class ID 来创建接口。

第三，它们的规范层次不同。COM 是二进制层次的规范，只要符合 COM 的二件制规范，我们可以使用任何一种语言进行开发。而 BREW 则是一种开发语言上的规范（使用了统一的 C 语言接口定义）。当然，我们也可以把 BREW 做为一种二进制层次的规范，但是似乎这并不大适合于在嵌入式系统中应用程序，因为当前大部分嵌入式系统都是使用 C 语言来开发的。不过对于 BREW 应用程序来说，只要遵循 BREW 的调用约定，是可以使用其它语言开发的。这里就不详细的讨论这个问题了。

13.11 小结

本章介绍了 BREW 实现的来龙去脉，展示了 BREW 所需要解决的两个主要问题：如何启动程序以及如何调用平台中的函数。最终，我们通过五个版本的 FastString 事例解决了全部的问题。BREW 的本质是通过虚拟函数表技术实现了接口定义与接口实现之间的分离，这样 BERW 应用程序就可以存储在文件系统中，并在运行的时候调用接口函数。可以这样说，BREW 已经将 C 语言运用到了极至，理解了 BREW 的原理可以对程序的开发有更加深入的认识。

思考题

1、BREW 要解决的主要是什么问题？
2、如果为 BREW 接口传递空指针会发生什么状况？

第十章 探秘 BREW 接口

BREW 做为一个运行平台，接口实现一直是他的主要开发工作之一，从 BREW 第一个版本开始，它就公开了几十个接口，而且随着 BREW 版本的不断升级，BREW 接口的种类和个数也在不断的增多。归纳起来可以分为基础类接口、网络类接口、图形多媒体类接口以及工具类接口等，正可谓五花八门。不过，虽然 BREW 接口的种类众多，但是它们在 BREW 平台上定义接口的方式是一样的。从 BREW 原理一章我们已经得知了 BREW 接口实现的原理，在这一章里，我们将展开讨论 BREW 接口的具体实现，同时我们还将讨论 BREW 中几个特殊的接口定义。当然，本章的主要目的不是为了要介绍 BREW 每个接口的用法是什么样子的，比如 IDislay 接口中有哪些 API，它们是怎么用的之类的，而是介绍接口是如何实现的。

10.1 BREW 接口的定义

BREW 提供了一组用来实现现接口的定义宏，通过这些宏可以根据接口的名称生成相应的接口定义的结构。由于我们在 BREW 原理一章中已经定义了一个类似 BREW 接口的 FastString 接口，实际的 BREW 接口定义形式与 FastString 的基本相同，只不过没有使用这些宏而已。下面我们就来看看与 BREW 接口定义相关的一些宏（请参看 AEE.h 文件）：

```
#define VTBL(iname)             iname##Vtbl

#define QINTERFACE(iname) struct _##iname {\
                                    struct VTBL(iname)    *pvt;\
                                };\
                                typedef struct VTBL(iname) VTBL(iname);\
                                struct VTBL(iname)

#define GET_PVTBL(p,iname)          ((iname*)p)->pvt
```

在这里我们列举了定义一个 BREW 接口所需要的三个宏，VTBL(iname)的作用是生成 VTBL 名称，其中的“##”是 C 语言预定义时表示将 iname 与 Vtbl 连接起来；QINTERFACE(iname)是生成结构结构的宏；GET_PVTBL(p,iname)是用来根据接口指针获得接口虚拟函数表指针。现在我们仍然以 FastString 为例，使用 BREW 的这三个宏定义一个 BREW 接口：

FastString.h – 接口声明文件

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

#include ”AEE.h”

typedef struct _IFastString IFastString;

QINTERFACE(IFastString)
{
```

```
    int (*AddRef) (IFastString *pIFastString);
    int (*Release) (IFastString *pIFastString);
    int (*GetLength) (IFastString *pIFastString);
    int (*Find) (IFastString *pIFastString, char *pFindStr);
};

// 增加接口指针的引用计数
#define IFASTSTRING_AddRef(p)       GET_PVTBL(p, IFastString)->AddRef(p)

// 释放目标字符串对象
#define IFASTSTRING_Release(p)      GET_PVTBL(p, IFastString)->Release(p)

// 获取目标字符串长度
#define IFASTSTRING_GetLength(p)    GET_PVTBL(p, IFastString)->GetLength(p)

// 查找字符串，返回偏移量
#define IFASTSTRING_Find(p,s)        GET_PVTBL(p, IFastString)->Find(p,s)
#endif // FASTSTRING_H_
```

将这个文件中的宏定义展开得到如下内容：

FastString.h – **接口声明文件**

```
#ifndef FASTSTRING_H_
#define FASTSTRING_H_

#include ”AEE.h”

typedef struct _IFastString IFastString;

struct _IFastString {
    struct IFastStringVtbl    *pvt;
};
typedef struct IFastStringVtbl IFastStringVtbl;
struct IFastStringVtbl
{
    int (*AddRef) (IFastString *pIFastString);
    int (*Release) (IFastString *pIFastString);
    int (*GetLength) (IFastString *pIFastString);
    int (*Find) (IFastString *pIFastString, char *pFindStr);
};

// 增加接口指针的引用计数
#define IFASTSTRING_AddRef(p)       ((IFastString*)p->pvt)->AddRef(p)

// 释放目标字符串对象
#define IFASTSTRING_Release(p)      ((IFastString*)p->pvt)->Release(p)
```

```
// 获取目标字符串长度
#define IFASTSTRING_GetLength(p)    ((IFastString*)p->pvt)->GetLength(p)

// 查找字符串，返回偏移量
#define IFASTSTRING_Find(p,s)          ((IFastString*)p->pvt)->Find(p,s)
#endif // FASTSTRING_H_
```

从这个宏定义展开的结果来看，使用 BREW 宏定义的接口与我们在 BREW 原理一章得到的结果是一样的，不同的仅仅在于_IFastStringVtbl 定义直接换成了 IFastStringVtbl。不过此时的实现中仍然有一个_IFastString 存在，这浪费了变量的命名空间，于是，BREW 便废除了 QINTERFACE 的宏定义，转而生成了下面的三个宏定义（请参考 AEEInterface.h）：

```
#define AEEVTBL(iname) iname##Vtbl

#define AEEINTERFACE(iname) \
        typedef struct AEEVTBL(iname) AEEVTBL(iname); \
        struct AEEVTBL(iname)

#define AEEGETPVTBL(p,iname)    (*((AEEVTBL(iname) **)((void *)p)))
```

使用 AEEINTERFACE 声明的接口定义形式，省去了_IFastString 结构体，同时也简化了定义。之所以能够使用这个宏定义来实现接口的定义，主要的技巧在于 AEEGETPVTBL 宏定义。从 BREW 原理一章我们可以知道，在创建接口的时候返回的接口指针是指向内部结构体 CFastString 的，同时由于 CFastString 的第一个成员便是 IFastStringVtbl 变量，因此在 IFastString 和 CFastString 结构体之间是可以进行类型转换的。同时，由于 IFastStringVtbl 指针 pvt 位于结构体 CFastString 的顶部，这样我们根据指针的特性可以知道，如果我们对 CFastString 指针进行取值操作的话，我们实际获得的是 IFastStringVtbl 的指针值。因此，实际上此时的 IFastString 指针是指向 IFastStringVtbl 指针的，也就是说 IFastString 指针是一个 IFastStringVtbl 类型的一个双重指针。对应的二进制结构如下图：

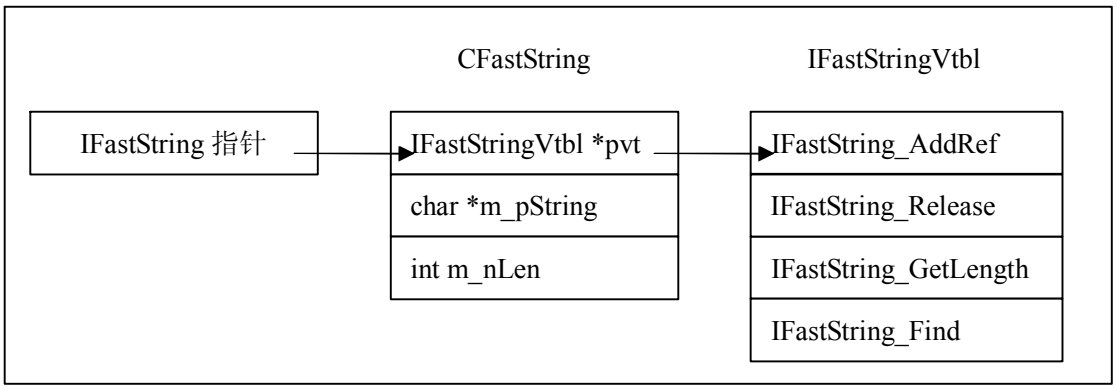

图 10.1 接口指针层次图

进一步的，从分解 AEEGETPVTBL 宏定义可以知道，这个宏将 IFastString 指针 p 强制转换成了 IFastStringVtbl 类型的双重指针，然后再通过取值符号“*”获得 IFastStringVtbl 的指针，这样我们就可以通过“->”符号取得相应的接口函数了。使用 AEEINTERFACE 实现的 IFastString 接口定义形式如下：

FastString.h – **接口声明文件**

```
#ifndef FASTSTRING_H_
```

```
#define FASTSTRING_H_

#include ”AEE.h”

typedef struct IFastString IFastString;

AEEINTERFACE(IFastString)
{
      int (*AddRef) (IFastString *pIFastString);
      int (*Release) (IFastString *pIFastString);
      int (*GetLength) (IFastString *pIFastString);
      int (*Find) (IFastString *pIFastString, char *pFindStr);
};

// 增加接口指针的引用计数
#define IFASTSTRING_AddRef(p)       AEEGETPVTBL(p, IFastString)->AddRef(p)

// 释放目标字符串对象
#define IFASTSTRING_Release(p)      AEEGETPVTBL(p, IFastString)->Release(p)

// 获取目标字符串长度
#define IFASTSTRING_GetLength(p)    AEEGETPVTBL(p, IFastString)->GetLength(p)

// 查找字符串，返回偏移量
#define IFASTSTRING_Find(p,s)         AEEGETPVTBL(p, IFastString)->Find(p,s)
#endif // FASTSTRING_H_
```

注意，在接口定义中 QINTERFACE 和 GET_PVTBL 宏定义以及 AEEINTERFACE 和 AEEGETPVTBL 宏定义要配对使用（两组之间不能交叉），否则会出现编译错误。正是因为 BREW 接口的定义采用了如上的一些技巧，所以 BREW 才实现了接口定义与接口实现之间的分离，带来了令我们兴奋的功能。

10.2 基类接口 IBase

由于通过虚拟函数表形式实现的 BREW 接口具有良好的面向对象的特性，特别是具有接口的二进制层面的继承机制，因此 BREW 将标准的以及每个接口都需要的接口定义在了一个叫做 IBase 的虚拟接口里面，同时提供了 INHERIT_IBase 宏定义，用来让全部其他的 BREW 接口方便的继承。

IBase 接口包含两个方法，分别是 AddRef 和 Release，主要是每个接口的资源管理所使用的。BREW 中 IBase 相关的定义如下（它们分别在 AEE.h 和 AEEInterface.h 文件中）：

```
#define INHERIT_IBase(iname) \
   uint32   (*AddRef)          (iname*);\
   uint32   (*Release)         (iname*)
```

```
QINTERFACE(IBase)
{
    INHERIT_IBase(IBase);
};

#define IBASE_AddRef(p)                    GET_PVTBL(p,IBase)->AddRef(p)
#define IBASE_Release(p)                   GET_PVTBL(p,IBase)->Release(p)

（以上代码摘自 BREW SDK 中 AEE.h 和 AEEInterface.h 文件）
```

在这里 IBase 的定义方法仍然使用的是 QINTERFACE，这主要是由于历史原因造成的，相信在后续发布的 BREW 版本里会逐渐的使用 AEEINTERFACE 来替换。相应的，存在 INHERIT_IBase 宏定义的情况下，接口定义形式的如下（仍旧以 IFastString 为例）：

```
AEEINTERFACE(IFastString)
{
    INHERIT_IBase(IBase);
    int (*GetLength) (IFastString *pIFastString);
    int (*Find) (IFastString *pIFastString, char *pFindStr);
};
```

BREW 约定每一个接口都需要采用这样的形式定义，这里之所以说是约定而不是规定，是因为这些都是实现的源代码，我们当然可以自己按照这个形式定义接口，而不使用 INHERIT_IBase。但是，任何一个接口中包含 AddRef 和 Release 两个方法都是必须的。此时，我们可以使用 IBASE_AddRef 和 IBASE_Release 两个方法来操作任何一个接口，例如存在接口 IFastString 的实例 pIFastString，此时我们可以调用 IBASE_AddRef(pIFastString)来增加 FastString 的引用计数，使用 IBASE_Release(pIFastString)来释放一个指针引用。为什么可以这样请各位读者根据虚拟函数表的相关知识自行理解。

IBase 接口的特殊性就在于，它是每个接口必须在二进制层面继承的基类接口，这一点是实现一个 BREW 接口所必须的。

10.3 应用接口 IApplet

与 IBase 接口相对应的，在实现 BREW 应用程序的时候，必须要遵守 IApplet 的规范。从第二部分我们知道，在 BREW 应用程序中最为关键的一个函数是_HandleEvent 函数，BREW 应用程序的任何一个事件都是通过这个函数传递进来的。而实质上，BREW 的 Applet 和接口之间基本的实现原理都是一样的，做为应用程序也需要实现下面定义的接口：

```
#define INHERIT_IApplet(iname) \
    INHERIT_IBase(iname); \
    boolean    (*HandleEvent)(iname * po, AEEEvent evt, uint16 wp, uint32 dwp)

QINTERFACE(IApplet)
{
    INHERIT_IApplet(IApplet);
};
```

```
#define IAPPLET_AddRef(p)                    GET_PVTBL(p,IApplet)->AddRef(p)
#define IAPPLET_Release(p)                   GET_PVTBL(p,IApplet)->Release(p)
```
（以上代码摘自 BREW SDK 中 AEE.h 文件）

对于使用 BREW SDK 向导生成的应用程序框架来说，不需要开发者去实现这个接口中的内容，向导会帮助我们完成这个工作，而且我们将在 Shell 内幕一章为您解剖整个应用程序实现的框架。之所以在这里罗列出 IApplet 接口，是希望传递一个信息，BREW Applet 和接口之间是有关系的，同时也因为 IApplet 也是一个比较特殊的基类接口。

10.4 控件接口 IControl

在 BREW 中还有另外一个比较特殊的基类接口，那就是 IControl。IControl 的定义形式如下：

```
#define INHERIT_IControl(iname) \
    INHERIT_IBase(iname);\
    boolean        (*HandleEvent)    (iname *, AEEEvent evt, uint16 wParam, uint32 dwParam);\
    boolean        (*Redraw)           (iname *);\
    void           (*SetActive)        (iname *, boolean);\
    boolean        (*IsActive)         (iname *);\
    void           (*SetRect)          (iname *, const AEERect *);\
    void           (*GetRect)          (iname *, AEERect *); \
    void           (*SetProperties)    (iname *, uint32);\
    uint32         (*GetProperties)    (iname *); \
    void           (*Reset)            (iname *)

QINTERFACE(IControl)
{
    INHERIT_IControl(IControl);
};

#define ICONTROL_AddRef(p)                    GET_PVTBL(p,IControl)->AddRef(p)
#define ICONTROL_Release(p)                   GET_PVTBL(p,IControl)->Release(p)
#define   ICONTROL_HandleEvent(p,ec,wp,dw)    GET_PVTBL(p,IControl)->HandleEvent(p,ec,
wp, dw)
#define ICONTROL_Redraw(p)                    GET_PVTBL(p,IControl)->Redraw(p)
#define ICONTROL_SetActive(p,a)               GET_PVTBL(p,IControl)->SetActive(p,a)
#define ICONTROL_IsActive(p)                  GET_PVTBL(p,IControl)->IsActive(p)
#define ICONTROL_SetRect(p,prc)               GET_PVTBL(p,IControl)->SetRect(p,prc)
#define ICONTROL_GetRect(p,prc)               GET_PVTBL(p,IControl)->GetRect(p,prc)
#define   ICONTROL_SetProperties(p,props)       GET_PVTBL(p,IControl)->SetProperties(p,
props)
#define ICONTROL_GetProperties(p)             GET_PVTBL(p,IControl)->GetProperties(p)
#define ICONTROL_Reset(p)                     GET_PVTBL(p,IControl)->Reset(p)
```

（以上代码摘自 BREW SDK 中 AEE.h 文件）

BREW 中的所有控件都必须使用 INHERIT_IControl 来定义接口，这样做的目的是提供一组可以控制任何控件公共特性的方法。例如，程序中有两个控件 IStatic 和 IMenuCtl，这两个控件在两个页面中显示，那么，为了统一显示的风格，现在需要在程序中编写一个统一的设置矩形显示区域的函数，名曰 MyApp_SetRect：

```
void MyApp_SetRect(void *pControl)
{
    AEERect ctlRect;

    SETAEERECT(&ctlRect, 0, 0, 100, 100);
    ICONTROL_SetRect((IControl*) pControl, &ctlRect);
}
```

通过这样的函数，可以很容易的实现对应用程序中控件行为的统一控制，这也是 IControl 接口存在的主要目的。在这些强制接口中，ICONTROL_HandleEvent 是用来向控件传送事件的函数，控件通过这个接口获得 BREW 应用程序或 BREW 对话框传递过来的事件；ICONTROL_Redraw 用来重新在屏幕上刷新控件；ICONTROL_SetActive 用来设置控件在不在活动状态，只有处于活动状态的控件才能接收事件；ICONTROL_IsActive 是用来判断控件是否在活动状态；ICONTROL_SetRect 和 ICONTROL_GetRect 分别用来设置/获得当前控件显示的矩形区域；ICONTROL_SetProperties 和 ICONTROL_GetProperties 分别用来设置/获得当前控件的属性，每一个控件都有自己的一些属性，它们采用 32 位掩码来进行使能的设置，因此每个控件最多有 32 个属性；ICONTROL_Reset 用来清空控件的所有设置，恢复到最初的状态。通过这些方法基本上可以控制控件的大多数公共行为了。

BREW 通过 IBase、IApplet 和 IControl 这些接口二进制层次的“强制性”继承方式，提供了控制不同接口实例的方法。这种二进制层次的继承，是 BREW 超越 C 语言本身特性的一个主要方面。

10.5 接口的实现

在 BREW 原理一章我们知道，为了支持多个 BREW 接口，提供了一个 Shell 的接口用来管理其余多个接口，创建接口的函数就是 ISHELL_CreateInstance。每一个接口都有一个唯一的 Class ID 来标示。然后通过应用程序的启动函数将 Shell 接口的指针传递进去，这样应用程序就可以通过 Shell 接口创建所需要的接口了。为了更好的描述整个过程，请看下面的示意图：

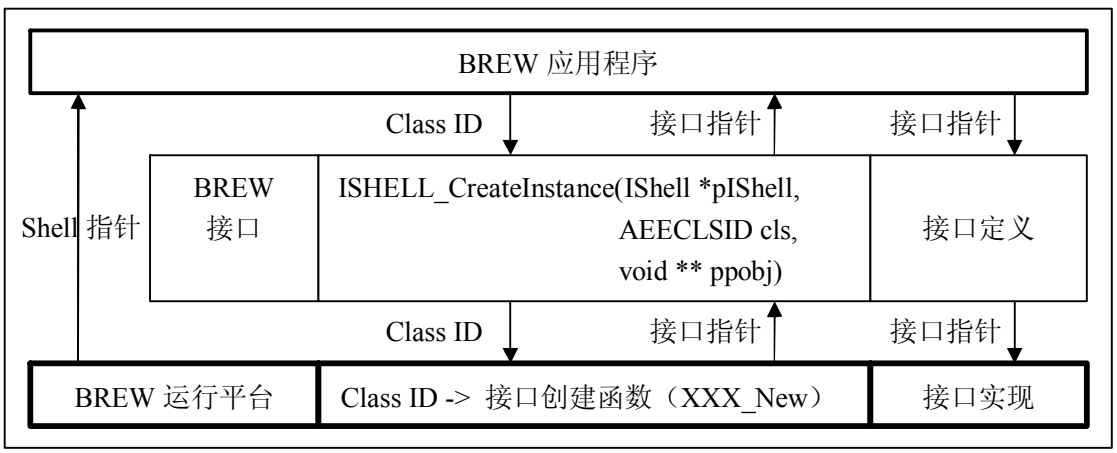

图 10.1 BREW 接口的生存过程

从图中我们可以看出（由左至右跟随箭头的方向），BREW 运行平台在启动应用程序的时候将 Shell 指针传递给 BREW 应用程序（具体参数如何传递将在 Shell 内幕一章中进行讲述），BREW 应用程序然后根据自身的使用情况创建所需使用的接口，同时将此接口的 Class ID 和 Shell 指针做为参数传递给 ISHELL_CreateInstance，用来创建相应的接口。在 ISHELL_CreateInstance 函数中，将根据 Class ID 来查找所对应接口命名为 XXX_New 的接口创建函数（XXX 指不同的接口命名），创建并初始化接口，然后通过 ISHELL_CreateInstance 接口中的 ppobj 参数将此接口的指针传递出去，此时在应用程序中就可以使用相应接口中的方法了。ppobj 是一个 void 型的二重指针，从前面的基础部分我们可以知道，二重指针可以做为传出数据的参数来传递一个指针的值。

在整个过程中有两个非常关键的点，一个是接口的 ID，另一个就是接口的创建函数 XXX_New。

10.5.1 接口的 ID（Class ID）

Class ID 就是接口在 BREW 平台上的身份证，从 BREW 原理一章中我们可以知道，BREW 使用了一种方式管理 Class ID 与接口创建函数的对应关系。在我们的例子中采用了一个 switch 语句，而实际上 BREW 采用的是一组结构体（AEEStaticClass）数组来管理了一个 Class ID 的表。AEEStaticClass 的定义如下：

```
typedef struct AEEStaticClass
{
    AEECLSID       cls;
    uint16         wFlags;
    uint16         wMinPriv;
    void           (*pfnInit)(IShell * piShell);
    int            (*pfnNew)(IShell *piShell,AEECLSID cls,void **ppif);
} AEEStaticClass;
```

在这个结构体中包含了 Class ID（cls）、对应的属性（wFlags）、私有设置（wMinPriv）、初始化函数指针（*pfnInit）以及接口创建函数指针（*pfnNew）。BREW 通过查找这个结构体所组成的链表，来调用对应的接口创建函数，从而创建对应的接口实例。这种方式使用在 BREW 的静态方式中，对于 BREW 动态应用程序的开发者来说，这些内容是不可见的，即便是在动态应用程序中扩展 BREW 接口时使用的也不是这种方式。

做为一个开放式的应用程序开发平台，必须有统一的底层接口来支持应用程序。也就是说，对于应用程序来说，底层接口应该都是采用同样的方式创建的，否则开发的应用程序将没有了可移植的特性，也就失去了其开放性的目的。在 BREW 中区分各个接口的就是接口的 ID，这个 ID 是一个 32 位的值。为了实现平台的开放目的，必须保证在任何平台上这些 Class ID 与相应接口的对应都是一致的。例如 AEECLSID_SHELL 的值是 0x01001000，就必须保证在任何硬件平台上 BREW 的 Shell 接口对应的 Class ID 都是 0x01001000。为此 BREW 采取了全球统一的 Class ID 管理方式，即通过其网站申请应用程序或扩展接口的 Class ID 以便于可以公开扩展的接口。如果接口的 Class ID 没有经过这样的全球申请方式，那么这个 Class ID 所对应的内容将可能与其它已经申请过的产生冲突，从而导致应用程序不能正常运行。当然，对于 BREW 设备开发者来说，可以使用 BREW 预留给 OEM 的 Class ID 空间来扩展接口，不过这些接口只能在特定的设备上使用，且通常不公开。

这里希望让您了解的就是 BREW 的全球 Class ID 管理的方式，这个 Class ID 是一个 32 位的数值。有时候我在想，是不是会有 Class ID 用完的时候，就像是电脑的千年虫一样，BREW 是否也会有一个“32 位虫”出现？这可能取决于 BREW 的发展速度了，又或者 BREW 的创造者们早就有了高明的对策，不过这些只有天知道！

10.5.2 接口的创建函数（XXX_New）

从上面的 AEEStaticClass 的结构体中我们可以看见这个 XXX_New 的原型：

```
int (*pfnNew)(IShell *piShell,AEECLSID cls,void **ppif);
```

这个就是联系 Class ID 和真正的接口内容的纽带，也是因为有了这个函数，才实现了 BREW 的 Class ID 管理方式。这个函数有三个参数和一个 int 型的返回值，这个返回值是表示接口是否创建成功，如果成功则返回 SUCCESS，否则返回 EFAILED 或者 ENOMEMORY 等错误值（这些值定义在 AEEError.h 文件中），使用者可以根据不同的返回值进行不同的处理。三个参数中，piShell 是传递了 BREW 的 Shell 指针，从前面可以知道，只要拥有了这个 Shell 指针，我们就可以使用 BREW 的接口管理函数了。第二个参数就是 AEECLSID 类型的一个值，表示我们要创建的是哪一个接口。第三个参数是一个二重指针，还记得多重指针的作用吗，他可以做为传出型的参数，将创建好的接口指针传递出来。

从本书前面的扩展 BREW 接口一章我们可以知道，在这个函数的内部实现里，主要完成的任务是为接口分配运行时的内存空间，其中包括分配内存空间，初始化虚拟函数表等内容。在这个函数执行成功之后，一个可以使用的接口就产生了，并通过 ppif 参数传递出去。虽然对于动态扩展和静态扩展这个函数的形式不一样，但是基本的原理还是相同的。

10.6 资源管理和 IBase

IBase 接口的两个方法 AddRef 和 Release 是每一个接口必备的功能，它们有着一些简单的约定，接口指针的所有使用者都必须遵循这些约定。当多个接口指针可能（也可能不是）指向同一个接口实例的时候，这些规则可以使接口的使用者从繁重的接口生命周期的管理中解脱出来。使用者只需要对每一个接口指针，都遵循简单的 AddRef/Release 规则，接口本身自然会管理自己的生命周期。这些规则可以概括为以下几个原则：

1、接口指针的创建者，在接口创建成功的情况下（非空接口指针），应该在离开接口指针作用域之前调用 Release 方法。

2、当一个非空接口指针，从一个内存位置复制到另一个内存位置的时候，要求调用 AddRef 方法，用来通知接口实例又有新的附加引用产生了。

3、对于已经包含非空指针的内存位置来说，如果需要重写该内存位置（如替换或清空），则需要首先调用 Release 方法，用来通知接口实例这个引用已经被销毁了。

4、如果我们对两个或多个内存位置之间的关系有特殊理解的话，那么多余的 AddRef/Release 方法可以被优化掉。

这 4 个原则包含了一个接口实例的全部生命周期和使用规则。其中第 4 条原则是建立在对整个接口指针的使用非常了解的情况下的，但是这种了解有的时候是十分困难的。我们认可任何的程序都可以优化，但是如果让我们花上几天的时间来优化掉几个简单的 AddRef/Release 方法的话，是不值得的，除非这个人已经疯了。所以，建议您按照前三个规则使用 AddRef/Release 方法，这样既安全又方便。

根据以上几个原则，我们可以把这些指导性的原则，变成编写程序时候的指导，以便于

确定什么时候应该调用 AddRef 和 Release 方法，什么时候不应该调用。下面是比较通用的，应该调用 AddRef 的情形：

A1. 当把一个非空接口指针写入一个局部变量的时候。

A2. 当被调用方把一个非空接口指针写入到[out]参数或[in,out]类型参数中的时候。

A3. 当被调用方返回一个非空接口指针做为一个返回结果的时候。

A4. 当把一个非空接口指针写入一个成员变量的时候。

在这些规则中，A3 是需要特别注意的，如果要通过一个函数的返回值来传递一个接口指针，那么这个函数的返回值必须被处理，否则将引起内存泄露。

下面是比较通用的，应该调用 Release 的情形：

R1. 在改写一个非空的局部变量或成员变量之前。

R2. 在离开一个非空局部变量的作用域（Scope）之前。

R3. 当被调用方要改写一个[in,out]参数，并且参数的初始值不为空的情况下。注意，[out]参数往往被认为初始值为空，因此对于[out]参数永远都不必释放。

还有一种很常见的特殊情况是，当把一个接口指针做为[in]参数传给函数时，可以适用前面的给出的第 4 条原则的特殊情况：

S1. 当调用方把一个非空接口指针通过[in]参数传递给函数的时候，既不需要调用 AddRef 方法，也不需要调用 Release 方法。因为在函数的调用堆栈中，临时变量的生命周期只在这个函数的运行期间内有效，且只是调用方生命周期的一个子集。

这几条规则基本上涵盖了 BREW 接口使用过程中全部的与生命周期有关的情况，值得仔细品读。

为了更加直观的了解这些规则，我们假设现在有一个全局函数 GetObject，它返回某个 BREW 接口的指针：

```
void GetObject([out]IBase **ppObj);
```

同时有一个使用这个接口指针的函数 UseObject：

```
void UseObject([in]IBase *pObj);
```

下面的代码使用这两个函数来操控某个接口实例，并返回一个接口指针给它的调用者。代码的注释部分给出了相应适用的指导规则。代码如下：

```
void GetAndUse(/*[out]*/ IBase **ppObj)
{
    IBase *pIBase1 = NULL, *pIBase2 = NULL;

    *ppObj = NULL;                                  // R3

    // 获取两个接口指针
    GetObject(&pIBase1);                            // A2
    GetObject(&pIBase2);                            // A2

    // 将 pIBase2 指向第一个实例
    if(pIBase2)
    {
        IBASE_Release(pIBase2);                     // R1
    }

    pIBase2 = pIBase1;
```

```
    IBASE_AddRef(pIBase2);                          // A1

    // 将 pIBase2 传递给其他函数使用
    UseObject(pIBase2);                             // S1

    // 将 pIBase2 做为 ppObj 返回值
    *ppObj = pIBase2;
    IBASE_AddRef(*ppObj);                           // A2

    // 超出作用域，释放接口
    if(pIBase1)
    {
        IBASE_Release(pIBase1);                     // R2
    }

    if(pIBase2)
    {
        IBASE_Release(pIBase2);                     // R2
    }
}
```

很重要的一点是，上述代码中 A2 规则使用了两次，但是却是从两个完全不同的角度来使用这条规则的。当调用 GetObject 的时候，这部分代码做为调用方，而 GetObject 函数的实现做为被调用方，也就是说，做为被调用方的 GetObject 函数有责任对从其[out]参数返回的 IBase 指针调用 AddRef 方法。当改写 ppObj 指针的内存的时候，这段代码做为被调用方，因此必须在把接口指针返回给调用方之前，应该正确的调用 AddRef 方法。

关于 AddRef 和 Release 之间还有一些微妙的情况值得讨论。AddRef 和 Release 的函数原型都返回一个 32 位的无符号整数。这个整数反映了当前接口指针的引用计数，这个计数只有是 0 的情况下才有意义，因为，只有在引用计数为零的情况下才会释放接口实例，宣布接口指针无效。然而反过来就不一定成立，也就是说即便当前返回的引用计数不为 0，那么，也不表示当前的接口指针是有效的。事实上，一旦当前接口指针（也就是初始接口指针的一个复制）调用的 AddRef 和 Release 方法次数相同，那么，对应的这个接口指针就是无效的。虽然可能由于其他的引用关系，这个接口实例依然存在，但是这只是一种特殊的情况，极有可能在某种紧急的情况下情况改变了。因此，要确保“接口指针在释放后不再被使用”策略的一个可行做法是，在使用了 Release 方法后将指针置为空（NULL）。如以下代码所示：

```
    if(pIBase1)
    {
        IBASE_Release(pIBase1);
        pIBase1 = NULL;
    }
```

采用这种方法之后，如果继续使用接口被释放的接口，会产生一个内存访问错误，这样的错误可以很容易的复现，因此，我们就可以很容易的查找到错误的所在。

与 AddRef 和 Release 的另外一个微妙之处发生在退出代码这一块。如前面我们定义的函数 GetAndUse，这个函数只有一个退出点，在退出之前会释放所有的接口指针。但是，如果这个函数中间有多个 return 语句，也就是有多个退出点，那么意味着如果函数是在这些退

出点中退出的，这些接口的资源将得不到释放，永远的的留在了系统之中，这就是我们所说的“内存泄露”，如果这个函数执行的非常频繁，那么最终会将系统资源耗尽，导致系统崩溃。因此，小心的处理每一个退出点是十分必要的。

10.7 小结

本章剖析了 BREW 接口定义的方式，最为重要的是应该理解 BREW 接口是相对于接口指针的偏移的这个接口本质。正是基于这个本质，BREW 具有了二进制层次的继承以及多态的面向对象特性。与此同时，我们还介绍了几个常用的公共接口的定义形式，不期望您可以记住它们，但是您一定要知道他们只不过是一些偏移量固定了的 BREW 接口的宏定义。接下来就是介绍了 BREW 接口的工作流程，让我们可以了解它的来龙去脉。最后介绍了重要的 AddRef 和 Release 规则，掌握这些规则对于我们编写 BREW 程序来说有很实际的意义，基本的原则就是被调用方负责增加引用计数，调用方负责释放接口的引用计数，还有就是释放了的接口指针不能再次使用的原则。

思考题

假设现在创建了接口 IBase *pIBase1，如果对这个接口指针连续使用 IBASE_Release 方法多次会发生什么情况，为什么？如果在调用 IBASE_Release 方法的中间有一些内存分配的动作（也就是接口指针 pIBase1 所指向的内存被重新分配过），又会怎么样，为什么？

第十一章 Shell（外壳）内幕

BREW 的最为重要的意义在于，它以较小的代价实现了嵌入式设备中应用程序的动态装载，就像是我们在 Windows 中使用安装程序一样，可以将程序存储在文件系统中。因此，如何控制和管理这些应用程序就是重中之重的事情了，而这一切都依赖于 BREW 的 Shell。之所以我们将它称之为 Shell，这源于在人们大脑中对程序功能的一种浪漫比喻——它的功能就像一个外壳，任何应用程序都在其保护之下，并且心肝情愿的受其控制。Shell 是一切 BREW 应用程序赖以生存的环境，BREW 应用程序正是通过和 Shell 的交互来完成它的生命周期。交互，是的，应用程序正是通过与 Shell 的交互才实现了它的事件驱动的机制，完成它生存的使命。Shell 可以说是任何一个系统中最为重要的部分，因为 Shell 就是系统内核的一个外在的表现。内核是负责系统运行、调度的指挥机构，一个内核设计的好坏往往就决定了一个系统设计的是否成功。

在这一章里，您将看到 BREW 的 Shell 主要有哪些功能，以及在这些功能背后 BREW 内核是如何处理的，更为重要的您将看到对 BREW 应用程序（Applet）的剖析。

11.1 Shell 的功能

Shell 的功能可以用四个字来形容，那就是“杂七杂八”。因为所有系统相关的、不能归类的、而且每个应用程序都可能使用的功能全部放在这里面了。这一节将介绍 BREW Shell 都有哪些功能以及这些功能的具体含义。不过这一部分没有介绍全部 Shell 接口的使用，紧紧介绍了一些常用的接口。

11.1.1 应用程序管理

BREW Shell 的应用程序管理函数可提供以下功能：

1、创建、启动和终止 BREW 类和应用程序

2、获取设备上的模块和类的信息

3、允许应用程序互相发送事件

4、允许运行 BREW 应用程序，而不干扰设备必须执行的其它活动

ISHELL_CreateInstance 用于创建设备模块支持的 BREW 类实例和用户自定义类实例。

ISHELL_StartApplet 函数允许运行指定的小程序。如果必要，该函数可以创建一个小程序实例，中止当前正在运行的小程序（如果有），然后以 EVT_APP_START 事件为参数调用指定小程序的 IAPPLET_HandleEvent 函数，从而运行该小程序。ISHELL_CloseApplet 将 EVT_APP_STOP 事件发送给当前正在运行的小程序，并调用其释放函数。ISHELL_CloseApplet 主要用于 AEE 外壳自身，因为不可能使用一个小程序终止另一个小程序。我们还可以使用 ISHELL_StartAppletArgs(IShell *pIShell, AEECLSID cls, const char * pszArgs)来启动应用程序，此方法允许在启动 BREW 应用程序向它传递命令行参数。它向 BREW 应用程序传递参数，指示 BREW 启动指定的 32 位 Class ID 所关联的小程序。 如果可以支持并启动所请求的类，BREW 将加载并启动小程序。请注意，启动小程序之前，该调用会立即返回到调用程序，小程序将异步启动 因此，如果找不到指定的小程序类 ID，函数将返回 SUCCESS 但不启动小程序。如果启动或恢复了小程序，则将清除显示并向

ISHELL_HandleEvent 发送 EVT_APP_START 或 EVT_APP_RESUME 以及 AEEAppStart 参数块。

ISHELL_ActiveApplet 用于获取当前正在运行的小程序的 Class ID。ISHELL_EnumAppletInit 和 ISHELL_EnumNextApple 可用于枚举设备模块中的小程序。枚举小程序时，ISHELL_EnumNextApplet 将返回指向小程序 AEEAppInfo 数据结构的指针。该结构可以标识小程序的 MIF 文件、标题、图标和类型（游戏、工具、PIM 和其它类型）。ISHELL_QueryClass 函数用于确定设备上是否存在某个类。如果该类为小程序，则将提供指向 AEEAppInfo 数据结构的指针，其中填充了相关的信息（如果小程序可用）。

ISHELL_SendEvent 用于向指定的类发送事件。如果当前没有该类的实例，BREW Shell 将创建一个实例，然后使用指定的事件代码和数据参数调用 IAPPLET_HandleEvent 函数。除非应用程序选择自动启动，否则应用程序将在完成事件处理后终止。

除了不会立即调用目标类的 IAPPLET_HandleEvent 函数之外，ISHELL_PostEvent 与之类似。事件放置在队列中稍后发送，这允许调用的程序继续执行，而不会中断。如果接收事件的类标识未知，则可以使用 ISHELL_HandleEvent。这时，BREW Shell 将把事件发送给当前运行的小程序或它的活动对话框（如果有）。

BREW Shell 包含了若干函数，使得 BREW 应用程序可与设备上的其它活动共存。BREW 应用程序模型基于协作多任务处理设计，这意味着每个应用程序在处理事件时只能运行尽可能短的时间，然后退出以便运行设备上的其它活动。

ISHELL_Busy 使 BREW 应用程序可以确定设备上是否存在其它正在进行的活动要求它立即退出。ISHELL_ForceExit 的作用与 ISHELL_Busy 相同。ISHELL_Resume 允许应用程序将耗时的任务分解为更小的可中断程序块。每个程序块由回调函数及关联的数据指针表示。ISHELL_Resume 用于调度以后要调用的回调函数（数据指针是其唯一参数）。ISHELL_CanStartApplet 用于检查能否启动 BREW 应用程序，如果设备上正在运行优先级较高的活动，则可能返回非真值。

11.1.2 闹钟功能

闹钟功能是 BREW 提供给用户使用的定时报警的功能，如果当前时间到达指定报警时间时，AEE Shell 将通知相应的应用程序，就像我们平常使用的闹钟一样。与只在应用程序运行时才被激活的计时器不同，即使应用程序不在运行也可以接收闹钟到期的通知。通常情况下，当所设置的报警时间距当前时间比较远时使用闹钟。例如，日程约会时间将要来临时，日程应用程序将使用闹钟提醒用户。

BREW Shell 中关于闹钟的方法主要有：ISHELL_SetAlarm、ISHELL_CancelAlarm 和 ISHELL_AlarmsActive。

设置闹钟调用 ISHELL_SetAlarm(IShell * pIShell, AEECLSID cls, uint16 nUserCode, uint32 nMins) 函数，指定从当前时间算起距离闹钟通知发生时间的分钟数 nMins、16 位闹钟代码 nUserCode、以及到达闹钟时间时接收通知的应用程序（我们的或他人的）的 BREW ClassID cls。通知发生时，使用 EVT_ALARM 事件和 16 位闹钟代码 nUserCode 做为参数，调用被通知应用程序的 IAPPLET_HandleEvent() 函数。nUserCode 参数使得应用程序能够区别多个同时激活的闹钟。 如果被通知的应用程序当前未在运行，AEE Shell 将创建它的一个实例来处理通知事件，完成后即终止该实例。 如果有必要，应用程序可以选择激活自身。BREW Shell 将闹钟保存在 BREW 数据库中，并在启动 BREW 设备时不断检查闹钟的到期时间。如果闹钟到期时设备是关闭的，则 BREW 外壳将在下次启动设备时生成通知。这

句话可能会让您产生误解，以为即使是在设备关闭的情况下也可以检测到闹钟是否到期，而实际情况是只有在设备开机，BREW 初始化完成之后，BREW 会根据当前的时间，以及已经设置的闹钟的信息来判断是否需要发送闹钟到期的事件。

ISHELL_CancelAlarm(IShell * pIShell, AEECLSID cls, uint16 nUserCode) 用于取消当前激活的闹钟。ISHELL_AlarmsActive(IShell * pIShell) 用于检查任一 BREW 内置报警器（闹钟、倒数计时器或秒表）当前是否处于激活状态。

11.1.3 定时器功能

当 BREW 应用程序中需要使用诸如定时刷新等类似的定时功能时，需要使用 BREW 的定时器（Timer）功能。当前已经运行（不管是否在 Active 状态）的应用程序（其引用计数非零）可以使用 AEE Shell 的定时器，在超过指定的时间段后执行某项操作。这些时间段一般很短（以秒或毫秒计算）。如果需要经过较长的时间段，则可以使用 BREW 外壳的警报函数获取通知（即使应用程序当前没有运行）。

BREW Shell 中有关定时器的方法主要有：ISHELL_SetTimer、ISHELL_SetTimerEx、ISHELL_CancelTimer 和 ISHELL_GetTimerExpiration。

如果要设置定时器，可以使用 ISHELL_SetTimer(IShell * pIShell, int32 dwMSecs, PFNNOTIFY pfn, void * pUser)函数，指定定时时间长度 dwMSecs（毫秒级），定时器到期时调用的回调函数 pfn，以及用户数据指针 pUser。当定时器到期时会调用 PFNNOTIFY 类型的一个回调函数，这个函数的类型定义如下：

```
typedef void (* PFNNOTIFY)(void * pData);
```

BREW Shell 在定时器到期调用此回调函数时，将用户数据指针 pUser 做为参数传递给 pData，这样，回调函数中就可以处理用户的数据了。通常情况下，这个 pUser 指针指向当前应用程序的内部结构指针，这使得回调函数可以像当前应用程序中其他内部函数一样的使用当前应用程序中的数据结构。使用定时器时需要注意以下几点：

1、计时器将在当前时间 +<指定的毫秒数> 时到期。

2、所有正常的处理都可以在回调中进行，其中包括绘制屏幕、写入文件及其它项目。

3、计时器不会重复。如果用户希望重复计时器，则必须重置计时器。

4、指定相同的回调函数和数据指针的定时器将自动覆盖具有相同回调和数据指针的待用计时器。这意味着定时器是自动覆盖型的，而且区分的标准就是回调函数 pfn 和用户数据指针 pUser，两者如果都相同则认为是同一个计时器。

5、终止当前激活的应用程序时，BREW Shell 将扫描计时器列表，如果应用程序终止的结果是被删除（例如，引用计数为 0），并使用指向 IApplet 的数据指针找到了相关计时器，则将删除该计时器。这里注意，只有使用了 IApplet 指针做为 pUser 的时候，才会在退出应用程序的时候自动取消，其他的情况下不会被自动取消。

6、负超时值被处理为 0 超时值。

7、不要寄希望于 BREW 系统会在应用程序退出时取消定时器，请主动地在 EVT_APP_END 事件中取消全部的定时器。

ISHELL_SetTimerEx(IShell * pIShell, int32 dwMSecs, AEECallback * pcb)方法与 ISHELL_SetTimer 方法一样，可以设置 BREW 定时器，唯一不同的是 ISHELL_SetTimerEx 使用了 AEECallBack 结构体来传递参数。AEECallBack 结构体定义如下：

```
structure _AEECallback
{
```

```
    AEECallback * pNext;
    void * pmc;
    PFNCBCANCEL pfnCancel;
    void * pCancelData;
    PFNNOTIFY pfnNotify;
    void * pNotifyData;
    void * pReserved;
}
```

在这个结构体中，pfnNotify 和 pNotifyData 分别对应于 ISHELL_SetTimer 中的 pfn 和 pUser 参数，作用亦相同。其他的成员变量 pNext/pmc/pReserved 属于系统保留成员，应用程序开发者（也就是我们）不要修改这些变量。pfnCancel 是用来指定取消回调的函数，pCancelData 就是为这个函数传递的数据了，一般情况下这个回调是不会使用的，尤其是在设置定时器这一块，它的存在只是增加一种取消回调的方法。关于 ISHELL_SetTimerEx 还有一点可能让各位迷惑的，看下面的两个定义（摘自 AEE.h 文件）：

```
#define ISHELL_SetTimer(p,s,pfn,pu)
                GET_PVTBL(p,IShell)->SetTimer(p,s, pfn,pu)
#define ISHELL_SetTimerEx(p,s,pcb)
                GET_PVTBL(p,IShell)->SetTimer (p,s, (PFNNOTIFY)pcb, (void *)pcb)
```

从中我们可以看出，ISHELL_SetTimer 和 ISHELL_SetTimerEx 采用了在虚拟函数表中同样的 SetTimer 函数，也就是说最终它们采用了同样的一个函数，那么，AEECallBack 结构指针又怎能转换成两种不同的数据呢（(PFNNOTIFY)pcb,(void *)pcb）？这是一种怎样的处理方法？从这里我们可以看到，在 ISHELL_SetTimerEx 定义中，第三个参数和第四个参数是同一个值，这在使用 ISHELL_SetTimer 中是不可能出现的。因此，如果在内部函数中增加第三个和第四个参数值是否相等的判断，那么我们就知道传进来参数的意义了，到底是 AEECallback 的结构还是其他就很清楚了。读者们千万不要认为类型转换之后就可以获得相应的数据了，实际上是函数内部有不同的处理。

ISHELL_GetTimerExpiration(IShell * pIShell, PFNNOTIFY pfn, void * pUser) 用于确定特定定时器到期之前剩余的毫秒数。定时器由回调函数和创建定时器时提供的数据结构地址来标识。ISHELL_CancelTimer(IShell * pIShell, PFNNOTIFY pfn, void * pUser) 用于取消正在运行的定时器。如果将回调函数的参数设置为 NULL，则将取消与指定数据结构地址相关联的所有定时器。如果应用程序实例的引用计数减为 0（零），则将取消与该应用程序相关联的所有定时器（关于这一点，请看上面注意事项的第五条）。

定时器功能在很多的 BREW 应用程序中都会使用到，而且使用的时候也很容易出现问题，尤其在应用程序退出时定时器的释放方面。希望各位读者可以记住一条，在应用程序退出之前要将应用程序中的所有定时器都释放掉，否则很容易由于回调函数中引用的资源已经被释放，而引起系统的崩溃。

11.1.4 资源文件和文件处理功能

应用程序可以使用 BREW Shell 提供的许多函数从文件中读取各类数据。这些文件可以是使用 BREW 资源编辑器创建的 BREW 资源（.bar）文件，也可以是内容与 MIME 类型相关或由文件扩展名标识的文件。我们还可以通过定义自己的处理程序类并使用它们操作特定 MIME 类型的文件，来扩展 BREW 可以识别的文件类型集。

请使用以下函数访问 BREW 资源文件。每个函数的参数包括资源文件名称和整型资源ID：

ISHELL_LoadResData 用于加载除字符串和位图图像之外的资源。 目前，这些资源包括与对话框有关的若干资源类型和以二进制模式存储到资源文件中的数据。 调用ISHELL_FreeResData 可以释放存储资源信息的内存。

ISHELL_LoadResImage 用于从指定的资源文件中加载位图图像，并返回包含该位图的IImage 接口实例指针。IIMAGE_Release() 用于释放存储位图的数据。

ISHELL_LoadResObject 用于实现声音和图像加载的函数。ISHELL_LoadImage、ISHELL_LoadResImage、ISHELL_LoadSound() 和 ISHELL_LoadResSound() 都是使用不同参数调用该函数的宏指令。

ISHELL_LoadResSound 用于从指定的资源文件中加载声音资源，并返回包含该声音文件的 ISoundPlayer 接口实例指针。ISOUNDPLAYER_Release() 用于释放存储声音信息的数据。

ISHELL_LoadResString 用于将字符串资源读入字符缓冲区，指向该字符缓冲区的指针是该函数的一个参数。

ISHELL_LoadImage 和 ISHELL_LoadSound 可用于直接加载图像和声音文件，而不必首先将它们的内容放入 BREW 资源文件。该文件必须包含 BREW 支持的一种内部 MIME 类型，如 Windows 位图 (.BMP) 或设备特定的图像原位图格式，以及 MIDI（.midi）或 CMX（.cmx）声音文件。

在这里顺带介绍一些关于 CMX 的背景知识。CMX（Compact Multimedia eXtension Services）是美国高通公司的无线平台中实现多媒体功能软件集合的统称。CMX 中可以支持 MIDI 音频、MP3 解码、AAC 音频解码、QCP 录音文件以及 JPEG 和 PNG 的图片解码等。BREW 中的 Media 接口就是在他的基础上建立起来的。部分 IMage 解码也是通过 CMX 实现的。当然，BREW 中的这些功能到底是否基于 CMX，最终取决于 BREW 所运行的底层平台是否支持 CMX 功能。

ISHELL_RegisterHandler 用于将 MIME 文件类型与为处理该类型文件而实现的 BREW 处理程序类的 Class ID 相关联。ISHELL_GetHandler 用于返回与给定 MIME 类型相关联的处理程序类的 Class ID，包括上文提到的 BREW 内置类型。关于这两个 Shell 方法，将在后面的多媒体接口的实现中详细的介绍，届时我们将可以了解到 BREW 实现多媒体接口的方法了。

关于资源文件的实现内幕，我们会在后面的章节中有详细的描述，在这里为了保持知识的完整性，我们仅仅介绍相关接口的功能。

11.1.5 通知（Notify）

除了前面提到的闹钟功能可以根据时间向应用程序发送事件之外，我们还可以通过一种叫做“通知”的方式来向应用程序发送特定的事件。BREW Shell 的通知机制允许 BREW 类相互通知发生的特定事件。要接收通知，已激活的类必须向 BREW Shell 注册，指定通知类的 Class ID 和通知的事件。当发生需要通知的事件时，通知程序类将调用 ISHELL_Notify，向每一个已经注册接收该事件通知的类发送通知。这里需要特别注意的是，BREW 的通知机制允许多个应用程序同时注册一个通知事件，而 BREW 会为每一个注册的应用程序都发送通知事件，优先的顺序是先看当前应用程序是否出于激活状态，如果是则激活应用程序优先通知，否则按照注册的先后顺序通知。

BREW Shell 为类提供了两个注册方法以接收事件通知：

1、通过 MIF 文件编辑器指定有关应用程序 MIF 文件中的通知信息进行注册。 即使未在运行也必须接收事件通知的应用程序可以使用这种注册方法。例如接收每个进入和外发呼叫通知的呼叫日志应用程序，即使设备用户没有运行该应用程序显示呼叫日志，应用程序也必须处理通知。

2、如果仅在应用程序运行的特定时间内需要进行通知，则可以调用一个叫做 ISHELL_RegisterNotify 的 BREW Shell 方法启动事件通知。例如，游戏应用程序在进入呼叫到达时可能显示一条消息，允许设备用户接受呼叫或继续游戏。应用程序仅要求在设备用户正在游戏时通知进入的呼叫，因此设备用户开始游戏时将调用 ISHELL_RegisterNotify。

在 ISHELL_RegisterNotify(IShell * pIShell, AEECLSID clsNotify, AEECLSID clsType, uint32 dwMask)中，clsType 是发出通知的 Class ID，也叫做通知程序类。通知程序类用于提供事件通知，该事件由 32 位变量 dwMask 表示。变量的低 16 位包含通知掩码（这里先介绍一下什么是掩码，因为这个名词曾经让我晕头转向。我们知道一个变量最终是由二进制位表示的，比如 uint32 类型的数据就是由 32 位二进制位组成的，其中的每一个二进制位都可以由 0 或 1 表示，也就是我们可以通过简单的“与或”操作取得或设置每一位的状态，那么，这每一个可以表示 0 和 1 两种状态的二进制位就叫做掩码。从程序中可以看到，每一个掩码定义的值用 16 进制表示都是 0x1、0x2、0x4、0x8 等形式，因为这些值就代表了一个二进制位），其中每一位都对应一个事件。我们知道变量的高 16 位包含通知程序匹配值，可用于事件所关联的数据。例如，表示 UDP 套接字活动的事件使用发生该活动的端口匹配值。clsNotify 是注册接收通知的 Class ID，当注册接收通知时，需要提供掩码变量值，将其中每个相关事件的位都设置为 1。如果类不再需要某个事件通知，可以将掩码变量中该事件的位设置为 0（零）调用 ISHELL_RegisterNotify。例如应用程序终止时，可能将掩码变量设置为 0（零），然后调用 ISHELL_RegisterNotify 方法将取消所有已经注册的通知事件。

通知程序类调用 ISHELL_Notify(IShell * pIShell, AEECLSID clsType, uint32 dwMask, void * pData)时，BREW Shell 将 EVT_NOTIFY 类型事件发送给每个注册接收事件通知的应用程序。这将使应用程序调用其 IAPPLET_HandleEvent 函数。 如果应用程序当前没在运行，BREW Shell 将创建它的实例来处理事件，完成后即终止该应用程序。如果应用程序想继续运行，也可以给自身发送启动事件。 调用 IAPPLET_HandleEvent 时，还将向应用程序传递一个 AEENotify 类型的结构指针，该指针用于标识发生的事件、生成通知的类及特定于通知的一些附加数据。clsType 是发送通知事件的类，dwMask 是发送当前通知的掩码，pData 是通知事件的附加数据。具体的通知流程如下图所示：

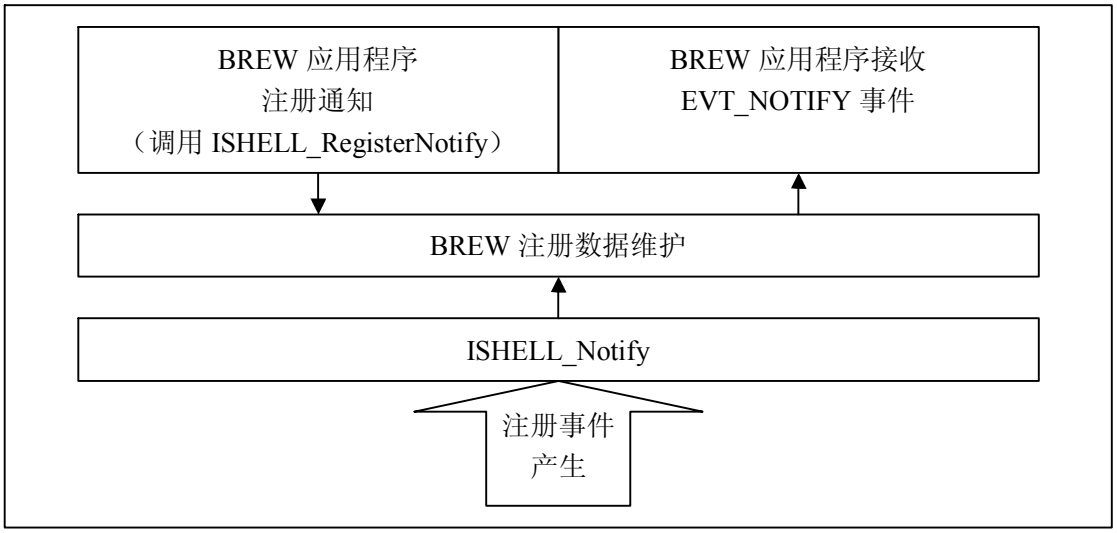

图 11.1 通知事件流程

在这幅图中，BREW 注册数据维护的部分包括了注册信息的管理，以及根据注册信息分别发送 EVT_NOTIFY 事件的处理。也就是说，当 BREW Shell 收到通知类的事件时，就会根据注册通知事件类的信息，依次向每一个应用程序发送通知，在调用 ISHELL_Notify 部分的代码中不必完成通知过程。

当前，INetMgr 类可以向其它相关类发送网络连接状态更改和特定 UDP 端口到达数据的通知。此外，我们还可以实现自己的通知程序类，只需要实现该类的 INotifier 接口。INotifier 接口要求我们实现类的 INOTIFIER_SetMask() 函数。只要其它类注册接收该类发出的事件通知，AEE 外壳即会调用此函数。INOTIFIER_SetMask() 仅包含一个 32 位的变量参数，是每个注册接收该类通知的类使用的掩码变量的逻辑“或”。只要至少一个类注册了接收对应的事件通知，该变量即会有一位的值为 1。当类首次注册要接收事件通知时，可使用该变量执行任何所需的初始化，如果没有任何类需要特定事件的通知时，可使用该变量完成结束操作。

11.1.6 对话框、消息框和提示

对话框由包含一个或多个 BREW 控件的屏幕组成，允许设备用户输入数据或选择菜单项目。尽管可以使用 BREW 控件接口（IDateCtl 接口、IMenuCtl 接口、ITextCtl 接口 和 ITimeCtl 接口）创建屏幕，但 IDialog 接口可以大大简化该任务：使用 BREW 资源编辑器创建对话框，包括对话框中每个控件的规格。BREW 可维护与当前运行的程序相关联的对话框栈。创建对话框时，会将对话框置于栈顶；结束对话框时，则将其从栈内删除，并恢复其下面的对话框。这样可以轻松实现能够引导设备用户贯穿屏幕序列（例如，菜单分级体系）的应用程序。

处于激活状态的对话框将接收所有事件，并将事件分配给当前激活的控件。对话框也处理控件切换，允许设备用户在多控件对话框中的控件之间移动焦点。这样我们便无需实现分配和处理这些事件的代码了。

ISHELL_CreateDialog (IShell * pIShell, const char * pszResFile, uint16 wID, DialogInfo * pInfo)用于创建对话框。 此函数可接受 BREW 资源编辑器所创建对话框的标识符 wID，或数据结构指针 pInfo（其中填充了指定对话框控件的代码）做为输入。 如果成功，函数将在屏幕上显示对话框控件，并将对话框推入对话框堆栈。要结束对话框，可以调用 ISHELL_EndDialog 终止栈顶对话框，并立即显示其下面的对话框。此函数还可以向应用程序发送 EVT_DIALOG_END 事件，允许执行与终止对话框相关的所有处理，例如获取设备用户在对话框控件中输入的值。ISHELL_EndDialog 还可释放对话框使用的所有资源。

ISHELL_CreateDialog 不返回指向所创建对话框的接口指针。ISHELL_GetActiveDialog 用于获取栈顶对话框的 IDialog 接口指针。使用此指针可调用组成 IDialog 接口的函数：IDIALOG_SetFocus 用于返回任一对话框控件的接口指针，IDIALOG_SetFocus() 用于指定多控件对话框中接收设备用户输入的控件。

BREW Shell 还提供了一些通过一次调用即可创建简单常用对话框的函数。ISHELL_Prompt 用于显示带软键控制菜单的对话框，提示设备用户作出选择。设备用户作出选择后，使用 EVT_COMMAND 将选择返回到应用程序并自动终止该对话框。还有两个函数可创建显示只读文本消息和标题的对话框。ISHELL_MessageBox 用于从 BREW 资源文件中读取标题和消息文本，ISHELL_MessageBoxText 则用于接受指向代码中指定的标题和消息文本字符串的指针。设备用户按下按键时，将结束这些函数创建的对话框。

11.1.7 设备和应用程序配置信息

这些函数允许获取关于设备及特定应用程序的配置信息。ISHELL_GetDeviceInfo() 可返回指向设备的 AEEDeviceInfo 结构的指针，其中包括它的屏幕大小、支持的颜色、可用内存量、字符编码以及其它项目的相关信息。这些信息通常是由 BREW 设备制造商根据设备的实际情况填充的，我们不能改变这些信息。

ISHELL_GetPrefs 和 ISHELL_SetPrefs 提供了应用程序注册配置信息的通用机制。ISHELL_SetPrefs (IShell * pIShell, uint16 wVer, void * pCfg, uint16 nSize)函数用于将配置数据指针和该数据的版本号与应用程序的 Class ID 关联。其它应用程序可使用指定的 Class ID 和版本号做为参数调用 ISHELL_GetPrefs 来获取这些数据。我们还必须提供结构声明，定义各版应用程序配置数据的内容。虽然在这两个函数的参数中，并没有指定相关的 Class ID，但是在这个函数的执行内部将会根据 Class ID 来区分不同应用程序的配置信息。

11.1.8 其它

除了前面提到的各种功能之外，BREW 还提供了支持其它功能的多个接口，例如 ISHELL_Beep(IShell * pIShell, BeepType nBeep, boolean bLoud)允许应用程序为设备用户提供多种声音信号或振动信号，根据给定设备支持的对象和类型 nBeep，可以为关闭设备、警告、提醒、消息到达和错误事件设置声音信号，为警告和提醒设置振动信号，我们也可以指定布尔响度参数 bLoud，来决定是否使用更高音量的声音等。在这里面就不多作介绍了，有兴趣的读者可以查看 BREW 的 API 帮助文档，那里面有权威而详细的说明。

11.2 BREW Shell 的初始化和终止

如果需要使用 BREW，那么就必须在运行 BREW 的设备启动时初始化 BREW Shell。在 BREW 的 OEM Porting Kit 中，负责初始化的函数是 IShell *AEE_Init(void)类型的函数。OEM Porting Kit 是 BREW 运行平台，需要在支持 BREW 的设备上完成对 OEM 层的支持。AEE 是 Application Execution Environment 意思，也就是实现 BREW 接口和内核的地方，在 AEE 层次的下方就是 OEM 层，也就是移植 BREW 需要实现的底层接口函数定义。在调用了 AEE_Init 并且初始化成功之后，将创建第一个 IShell 实例，同时 AEE_Init 返回 IShell 指针。这个 IShell 指针就是整个系统中使用的，在整个 BREW 的生命周期中它将永远存在。

在初始化过程中，BREW 还将根据 OEM 层指定的默认应用程序。我们知道，在 BREW 环境中，如果按下设备的 End 键，应该返回默认的应用程序中去。这里需要注意的是，这里的默认程序与初始化自动启动的应用程序不同，按下 End 键之后，在 OEM 层会启动由 OEM 指定的应用程序，而且这部分代码需要 BREW 的移植者自行实现，它与自动启动的默认应用程序不同。

BREW 的运行需要一个异步信令循环的机制来支持，这种机制通常采用如下的表达形式：

```
int n_Sig;

if(NULL == AEE_Init())
{
```

```
    // 初始化失败
    return;
}

for(;;)
{
    n_Sig = OS_Wait(BREW_SIG| BREW_EXIT_SIG| ...);
    if(n_Sig & BREW_SIG)
    {
        // 处理 BREW 信令
        AEE_Dispatch();
    }

    if(n_Sig & …)
    {
        …
    }

    if(n_Sig & BREW_EXIT_SIG)
    {
        // 释放全部 BREW 资源，退出 BREW
        AEE_Exit();

        // 退出任务
        break;
    }
}
```

在这段代码中，OS_Wait 是根据不同的操作系统而采用不同的任务等待函数，当程序执行到这个函数的时候，会切换到其他的任务去执行，直到 BREW_SIG 的信令传过来之后才接着处理程序。当收到 BREW 的信令时，通常调用 AEE_Dispatch 函数来分发处理在 BREW 事件队列中的事件。处理完成后，重新开始下一个信令循环。就是通过不断触发这样的事件循环，才能够让 BREW 生生不息循环不止。

如果当前运行 BREW 的任务，接收到退出 BREW 的信令时，BREW 将调用 AEE_Exit 函数退出 BREW，同时释放 BREW 占用的资源。当然，上面所提到的仅仅与 BREW 底层有关，对于 BREW 应用程序开发者来说是不可见的。在这里面说明一下是为了能够让我们更加清楚地了解 BREW 的运行方式。

11.3 应用程序在 Shell 中的实现

应用程序和 Shell 之间就像是一对孪生兄弟，谁也离不开谁。没有 Shell 的应用程序无法运行，而没有应用程序的 Shell 也毫无意义。BREW Shell 提供了一组方法来控制应用程序，应用程序也通过这些方法与 BREW Shell 之间进行交互。BREW 的应用程序机制最大的优点，在于它的模块化功能做得比较好。试想一下，对于移动通信设备来说，其面临着随时随地都有可能来一个电话，这个时候正在运行的应用程序就需要处理是否接听，或者采取什么样的

方式接听这个电话。如果模块化做的不好，那就需要在每一个功能项目里处理这种问题，比如说我要在电话本的应用程序中处理，也要在通话列表的应用程序中处理，这样无疑增加了程序的负担和不必要的工作量。如果对于模块化比较好的系统来说，这个问题就不存在了，管理通话的应用程序全权负责这个模块中的内容，就不会浪费人力物力了。

如果要实现这种模块化的功能，那么就需要有一个良好的应用程序框架。我们的 BREW 就提供了这样的框架，使得我们可开发出高模块化的应用程序，这个框架的秘密就在 BREW Shell 和应用程序之间的交互上。下面就让我们逐一揭开它们的神秘面纱。

11.3.1 动态和静态应用程序

BREW 的应用程序分为动态应用程序和静态应用程序，动态应用程序就是我们平常可以使用 BREW 模拟器开发的，可以使用 ARM 编译器生成.mod 二进制文件的应用程序。关于动态应用程序的开发，我们已经在第二篇中详细的介绍了，BREW 存在的最有魅力的意义，就在于使用较小的系统资源实现了分散式的程序安装和运行。因此可以说动态应用程序是 BREW 的灵魂。

与动态应用程序相对应的就是静态应用程序。静态应用程序是指那些移动设备生产厂家，在使用 BREW 开发应用程序时直接与 BREW 编译到一个映像文件中的应用程序。静态应用程序之所以存在的一个理由就是，当使用 BREW 开发设备的用户界面（UI：User Interface 或 MMI：Man Machine Interface）时需要将应用程序和系统集成在同一个映像里面。由于静态应用程序只有设备生产厂家能够开发，因此，静态应用程序的 Class ID 可以使用 BREW 保留给 OEM 区间内的 ID。

在 BREW3.x 版本的时代，BREW 动态应用程序和静态应用程序没有什么明显的区别，都需要使用 MIF 文件。然而在 BREW2.x 时代，动态和静态应用程序的实现是有很大区别的。通常，静态应用程序没有 MIF 文件，模块信息通过一个叫做 AEEAppInfo 的结构体来指定，这个结构体的定义如下：

```
typedef struct
{
    AEECLSID          cls;
    char    *         pszMIF;              // App Resource file
    uint16            wIDBase;             // Base ID for locating title, icon, etc
    uint16            wAppType;            // Extended OEM/Carrier App Type
    uint16            wPad2;
    uint16            wPad3;
    uint16            wPad4;
    uint16            wFlags;              // Applet Flags (AFLAG_...)
} AEEAppInfo;
```

在这个结构体中包含了应用程序的 Class ID 信息 cls，应用程序资源文件的名称 pszMIF，定位小程序标题、图标等资源文件的基本 ID wIDBase，扩展的 OEM/运营商类型的 wAppType，以及标示应用程序属于游戏、工具等属性的 wFlags。相信各位读者对于这些内容都有一些似曾相识的感觉，没错，这些内容就是在我们应用程序中 MIF 文件的部分内容。

这个结构体并非只存在于静态应用程序中，在 BREW Shell 的 ISHELL_EnumNextApplet (IShell * pIShell, AEEAppInfo * pai)方法中使用了这个结构体来获得应用程序的信息。从中我们也可以看出，AEEAppInfo 结构体就是 BREW 管理应用程序列表的一个内部结构。不要

认为 pszMIF 是指当前应用程序的资源文件名称，其实它指的是对应的 MIF 文件的名称，在获取应用程序信息时，可以通过这个名称和 wIDBase 来获得应用程序的名称以及图标等信息，让我们来参考一下 AEEShell.h 中的定义吧：

```
//
// Standard Applet Resource Offsets - These offsets begin at AEEAppInfo.wIDBase.
//
#define IDR_NAME_OFFSET              0
#define IDR_ICON_OFFSET              1
#define IDR_IMAGE_OFFSET             2
#define IDR_THUMB_OFFSET             3
#define IDR_SETTINGS_OFFSET          4
#define IDR_VERSION_OFFSET           5
#define IDR_ENVIRONMENT_OFFSET       6
#define IDR_OFFSET_STEP             20

//
// Applet Resource Macros
//
// 应用程序名称字符串 ID
#define APPR_NAME(ai)            (uint16)(((ai).wIDBase + IDR_NAME_OFFSET))

// 正常大小的应用程序图标 ID
#define APPR_ICON(ai)            (uint16)(((ai).wIDBase + IDR_ICON_OFFSET))

// 应用程序大图标 ID
#define APPR_IMAGE(ai)           (uint16)(((ai).wIDBase + IDR_IMAGE_OFFSET))

// 应用程序小图标 ID
#define APPR_THUMB(ai)           (uint16)(((ai).wIDBase + IDR_THUMB_OFFSET))

// 应用程序设置数据 ID
#define APPR_SETTINGS(ai)        (uint16)(((ai).wIDBase + IDR_SETTINGS_OFFSET))

// 应用程序版本数据 ID
#define APPR_VERSION(ai)         (uint16)(((ai).wIDBase + IDR_VERSION_OFFSET))

// 应用程序环境数据 ID
#define APPR_ENVIRONMENT(ai)
                        (uint16)(((ai).wIDBase + IDR_ENVIRONMENT_OFFSET))
```

上面的定义就是我们获取应用程序信息数据的来源，它们是不同类型数据的不同 ID。在应用程序中，我们可以通过使用 AEEAppInfo 结构体中的 pszMIF、wIDBase，以及这些 ID 定义，使用 BREW Shell 的各种资源文件相关方法，来获得存储在 MIF 或资源文件中的应用程序信息。既然已经分析到这里了，那就让我们揭开 MIF 文件的神秘面纱吧——MIF 文件其实就是固定资源文件 ID 的一个已经编译过的二进制资源文件，其二进制格式与已经编译

的资源文件（.bar）是一样的！在本书后面将专门有一章来分析 BREW 资源文件的二进制结构，届时我们也将分析一下 MIF 文件的结构，从而推导出二进制资源文件的数据结构。

在 BREW2.x 版本的静态应用程序中，由于没有使用 MIF 文件，因此，AEEAppInfo 结构体中的 pszMIF 和 wIDBase 项就可以置空。当然我们可以选择指定一个资源文件，起码可以获得应用程序的名称，虽然没有图标等其他信息。

11.3.2 从 IModule 到应用程序

我们知道，每一个 BREW 应用程序都有一个 MIF 文件，在这个 MIF 文件中包含了整个模块的信息，如 Applet 信息、扩展接口和外部依赖等。这个文件在 BREW 进行初始化的时候将 MIF 信息读取到 BREW 内存信息结构列表中，以便于在创建应用程序或接口时查询这个表中的信息。而与每一个 MIF 文件相对应的，都有一个 IModule 接口。在载入模块时（也就是创建模块中第一个接口或第一个 Applet 的时候），MIF 文件寻找与之对应的 Load 函数，这个函数的定义形式如下：

```
int XXX_Load(IShell *pIShell, void *ph, IModule **ppMod);
```

其中“XXX”代表模块的名称，在静态模块的情况下，“XXX”是根据不同的模块有不同的名称。例如电话簿模块的函数名可能是 PhoneBook_Load 函数。在动态应用程序中这个函数名称固定是 AEEMod_Load，函数的实体在 AEEModGen.c 文件中。

AEEMod_Load 函数是动态应用程序模块的入口函数，BREW Shell 就是通过对这个函数的调用启动模块的。关于这一点我们可以从 ARM 编译的指令中获得佐证，在 Visual Studio 中打开任何一个 BREW 应用程序的工程，选择输出 ARM Make 选项（可以通过工具菜单或工具条），就会输出对应的.mak 文件。从这个文件中我们可以看到如下的链接指令：

```
#----------------------------------------------------------------------------
# Linker options
#----------------------------------------------------------------------------

LINK_CMD = -o                          #Command line option to specify output file
                                       #on linking

ROPILINK = -ropi                       #Link image as Read-Only Position Independent

LINK_ORDER = -first AEEMod_Load
```

“-first”参数就是指定链接时放在二进制映像文件起始位置的符号，这里面指定的就是 AEEMod_Load 函数（在编译器里，所有的变量和函数都是使用不同的符号（Symbols）来表示的）。同时在这里面指定了“-ropi”参数，意思是将链接的二进制映像文件链接成地址独立的程序。也就是说，在这个二进制映像文件的内部，仅仅存在着只读的代码数据，没有全局变量和静态变量，所有函数的地址都是使用与 CPU 的 PC（Program counter 程序寄存器）寄存器地址相关的形式编译。这样，在这个映像文件内部的全部程序跳转都相当于只与入口点地址有关。正是因为有了 ARM 链接器的这一个特性，BREW 动态应用程序才得以实现。

运行时，BREW Shell 将全部.mod 文件中的指令复制到已经分配的一段连续的 RAM 空间中，同时在这段 RAM 空间的起始位置中以 AEEMod_Load 函数的形式，执行这段二进制文件中的代码。此时应用程序就通过这个入口点开始执行了。下面是 AEEMod_Load 函数内部执行的代码（参考 AEEModGen.c 文件）：

```
int AEEMod_Load(IShell *pIShell, void *ph, IModule **ppMod)
{
    // Invoke helper function to do the actual loading.
    return AEEStaticMod_New(sizeof(AEEMod),pIShell,ph,ppMod,NULL,NULL);
}
```

AEEStaticMod_New 函数的内部代码如下：

```
int AEEStaticMod_New(int16 nSize, IShell *pIShell, void *ph, IModule **ppMod,
                     PFNMODCREATEINST pfnMC,PFNFREEMODDATA pfnMF)
{
    AEEMod *pMe = NULL;     // 声明 AEEMod 结构体指针变量
    VTBL(IModule) *modFuncs;// 声明 IModule 虚拟函数表指针变量

    // 检查参数的有效性
    if (!ppMod || !pIShell) {
        return EFAILED;
    }

    if (nSize < 0) {
        return EBADPARM;
    }
    *ppMod = NULL;

    // 为 AEEMod 实例分配存储空间（AEEMod 结构体+虚拟函数表空间）
    if (nSize < sizeof(AEEMod)) {
        nSize += sizeof(AEEMod);
    }

    if (NULL == (pMe = (AEEMod *)MALLOC(nSize + sizeof(IModuleVtbl)))) {
        return ENOMEMORY;
    }
    // 分配虚拟函数表空间并初始化虚拟函数表
    // 注意，由于在动态应用程序中不能存在任何的静态数据，
    // 因此我们采用这种方式为虚拟函数表分配空间


    modFuncs = (IModuleVtbl *)((byte *)pMe + nSize);

    // 初始化虚拟函数表中的函数成员
    modFuncs->AddRef          = AEEMod_AddRef;
    modFuncs->Release         = AEEMod_Release;
    modFuncs->CreateInstance = AEEMod_CreateInstance;
    modFuncs->FreeResources   = AEEMod_FreeResources;

    // 初始化虚拟函数表指针指向前面分配的 modFuncs 所指向的存储空间
```

```
    INIT_VTBL(pMe, IModule, *modFuncs);

    // 初始化成员变量

    // 存储模块的实例创建函数 CreateInstance 的地址
    pMe->pfnModCrInst = pfnMC;

    // 存储模块数据释放函数的地址
    pMe->pfnModFreeData = pfnMF;

    pMe->m_nRefs = 1;
    pMe->m_pIShell = pIShell;
    ISHELL_AddRef(pIShell);

    // 将 IModule 接口实例指针指向已经分配的空间中
    *ppMod = (IModule*)pMe;

    return SUCCESS;
}
```

在这个函数中，主要是创建了一个 IModule 接口。每一部份代码的作用已经在上面进行了说明。函数调用完成后返回 AEEMod_Load 函数，同时将 IModule 接口实例指针通过 ppMod 传递给 AEEMod_Load。AEEMod_Load 函数则直接返回 AEEStaticMod_New 的处理结果。

在 IModule 接口创建成功之后，BREW 将调用 IMODULE_CreateInstance 接口函数创建模块中的 Applet 或接口实例了。由上面的虚拟函数表初始化部分可以看出，这个接口对应的函数是 AEEMod_CreateInstance。

```
modFuncs->CreateInstance = AEEMod_CreateInstance;
```

AEEMod_CreateInstance 函数的实现如下（参考 AEEModGen.c）：

```
static int AEEMod_CreateInstance(IModule *pIModule,IShell *pIShell,
                                 AEECLSID ClsId,void **ppObj)
{
    AEEMod      *pme = (AEEMod *)pIModule;
    int         nErr = EFAILED;

    // 对于动态应用程序必须支持 AEEClsCreateInstance 函数以便于调用创建实例的函数
    // 对于静态应用程序必须通过 AEEStaticMod_New 函数注册一个实例创建函数
    if (pme->pfnModCrInst) {
        nErr = pme->pfnModCrInst(ClsId, pIShell, pIModule, ppObj);
#if !defined(AEE_STATIC)
    } else {
        nErr = AEEClsCreateInstance(ClsId, pIShell, pIModule, ppObj);
#endif
    }

    return nErr;
```

```
}
```

AEEClsCreateInstance 函数？好熟悉啊，不是吗？在我们创建的动态应用程序中都有这个函数的存在啊。现在我们就来看看这个函数的实现吧（您可以打开任何一个 BREW 示例应用程序的.c 文件，如 helloworld.c 文件，来查看这个函数）：

```
int AEEClsCreateInstance(AEECLSID ClsId,IShell * pIShell,IModule * pMod,void ** ppObj)
{
    *ppObj = NULL;
     IApplet *pMe = NULL;

     // 判断创建应用程序或接口的 Class ID
     if(ClsId == AEECLSID_HELLOWORLD)
     {
          // 创建 IApplet 接口实例
          if(AEEApplet_New( sizeof(helloworld),
                            ClsId,
                            pIShell,
                            pMod,
                            (IApplet**)ppObj,
                            (AEEHANDLER)helloworld_HandleEvent,
                            (PFNFREEAPPDATA) helloworld_FreeAppData)
          {
               pMe = (IApplet *)(* ppObj);

               // 初始化应用程序数据
               if(helloworld_InitAppData((helloworld *)pMe))
               {
                    return(AEE_SUCCESS);
               }
               else
               {
                    IAPPLET_Release(pMe);
               }
          }
     }
     return (EFAILED);
}
```

在这个函数里面增加了“if(ClsId == AEECLSID_HELLOWORLD)”的判断，有什么作用呢？通常我们创建的模块中仅仅包含一个 Applet，因此不必增加这个条件判断。然而，如果当前的模块中包含多个 Applet 或者多个输出接口的时候，增加这个 Class ID 的判断就是必须的了，否则我们将不能够区分将要启动哪一个 Applet 或创建哪一个接口实例。

接下来 AEEClsCreateInstance 函数中调用了 AEEApplet_New 函数。请注意调用这个函数时的第一个参数 sizeof(helloworld)，这里是通知 AEEApplet_New 为当前应用程序分配这么大的空间。helloworld 是当前应用程序的私有数据结构，在这个结构体中我们可以声明在应用程序中使用的全局变量。此时我们进入 AEEApplet_New 函数的内部看一看（参考

AEEAppGen.c）：

```
boolean AEEApplet_New(int16 nIn,
                      AEECLSID clsID,
                      IShell * pIShell,
                      IModule * pIModule,
                      IApplet **ppobj,
                      AEEHANDLER pAppHandleEvent,
                      PFNFREEAPPDATA pFreeAppData)
{
    AEEApplet *          pme = NULL;
    VTBL(IApplet) *     appFuncs;
    int                  nSize;

    if(nIn < 0)
        return(FALSE);

    nSize = (int)nIn;

    if(!ppobj)
        return(FALSE);

    *ppobj = NULL;

    //参数检查
    if (!pIShell || !pIModule)
        return FALSE;

    if(nSize < sizeof(AEEApplet))
        nSize += sizeof(AEEApplet);

    // 创建应用程序实例
    if (NULL == (pme = (AEEApplet*)MALLOC(nSize + sizeof(IAppletVtbl))))
        return FALSE;

    appFuncs = (IAppletVtbl *)((byte *)pme + nSize);

    //初始化 IApplet 接口的虚拟函数表成员项目
    appFuncs->AddRef      = AEEApplet_AddRef;
    appFuncs->Release     = AEEApplet_Release;
    appFuncs->HandleEvent = AEEApplet_HandleEvent;

    INIT_VTBL(pme, IApplet, *appFuncs);    // 初始化虚拟函数表

    //初始化数据成员
```

```
    pme->m_nRefs            = 1;
    pme->m_pIShell          = pIShell;
    pme->m_pIModule         = pIModule;
    pme->m_pIDisplay        = NULL;

    pme->clsID              = clsID;

    // 存储事件处理函数和资源释放函数指针
    pme->pAppHandleEvent = pAppHandleEvent;
    pme->pFreeAppData = pFreeAppData;

    // 创建显示接口
    ISHELL_CreateInstance(pIShell, AEECLSID_DISPLAY,
                                    (void **)&pme->m_pIDisplay);

    if (!pme->m_pIDisplay) {
        //Cleanup
        FREE_VTBL(pme, IApplet);
        FREE(pme);
        return FALSE;
    }

    // 增加 IShell 和 IModule 接口的引用
    ISHELL_AddRef(pIShell);
    IMODULE_AddRef(pIModule);

    // 返回 IApplet 接口实例指针
    *ppobj = (IApplet*)pme;

    return TRUE;
}
```

从此函数的内部程序来看，它创建了一个 IApplet 的实例，同时将这个实例指针通过 ppobj 参数传递出去。同时在内部还创建了每一个 Applet 都会使用的 IDisplay 接口。应用程序实例创建成功之后，BREW Shell 将通过 IApplet 接口来控制应用程序：使用 IApplet 的 IAPPLET_HandleEvent 接口传递事件给应用程序，使用 IAPPLET_Release 方法释放应用。IApplet 接口的 HandleEvent 和 Release 方法实现如下（参考 AEEAppGen.c）：

```
boolean    AEEApplet_HandleEvent(IApplet * po, AEEEvent evt, uint16 wParam,
                                                uint32 dwParam)
{
    return ((AEEApplet *)po)->pAppHandleEvent(po, evt,wParam,dwParam);
}

static uint32 AEEApplet_Release(IApplet * po)
{
```

```
    IShell * pIShell = NULL;

    AEEApplet * pme = (AEEApplet *)po;

    if (--pme->m_nRefs)
        return(pme->m_nRefs);

    // 调用应用程序注册的资源释放函数

    if(pme->pFreeAppData)
        pme->pFreeAppData(po);

    // 释放在 AEEApplet_New()函数中创建的 IDisplay 接口
    if (pme->m_pIDisplay)
        IDISPLAY_Release(pme->m_pIDisplay);

    // 释放对 IModule 接口的引用
    IMODULE_Release(pme->m_pIModule);

    pIShell = pme->m_pIShell;

    // 释放 IApplet 实例
    FREE_VTBL(pme, IApplet);
    FREE(pme);

    ISHELL_Release(pIShell);     // 释放对 IShell 接口的引用

    return 0;
}
```

从中我们可以看出，IApplet 接口 HandleEvent 方法的实现仅仅是调用了使用 AEEApplet_New 函数注册的事件处理函数而已，Release 方法中则释放了在 AEEApplet_New 函数中创建的资源。至此，我们可以理出从 IModule 接口的创建到应用程序执行的过程了，这个过程如图 15.2 IModule 和动态应用程序启动流程所示。

从图中我们可以看出，对于应用程序的运行，BREW Shell 总共有三个动作：一是从 BREW 内部的模块列表中获得应用程序所在模块的入口函数（这里是 AEEMod_Load）并执行此函数同时获得 IModule 接口实例指针；二是在成功获得 IModule 接口实例之后，会调用模块的创建实例的方法 IMODULE_CreateInstance 创建应用程序的实例，同时获得兼容 IApplet 接口的应用程序实例；三是在应用程序创建成功之后将通过 IAPPLET_HandleEvent 函数向应用程序发送事件，发送的第一个事件就是 EVT_APP_START，如果这个事件应用程序返回 TRUE 则表示应用程序已经启动成功了。这三个动作是 BREW Shell 在启动应用程序时的连续动作，严格的按照顺序执行，首先获得了 AEEMod_Load 入口函数，然后通过这个函数获得 IModule 接口实例指针，最后再通过 IModule 的方法获得 IApplet 兼容接口实例的指针。

注意，在上面获得 IApplet 接口实例的时候，使用的说法是“IApplet 兼容接口”，这个

说法是有原因的。我们知道，IApplet 接口有三个方法 AddRef、Release 和 HandleEvent，且 IApplet 接口是一个虚接口，在 BREW 内部没有实现，需要在应用程序中实现。为了广大程序员编写应用程序的方便（每一个 Applet 都需要实现 IModule 和 Applet 接口），BREW 提供了两个现成的 IModule 和 IApplet 接口的实现，分别放在了 AEEModGen.c 和 AEEAppGen.c 文件中。这里面由于 IModule 接口与应用程序内部的函数没有任何关系，因此我们也没有什么可以优化的，老老实实的使用 AEEModGen.c 中的实现就好了。但是对于在 AEEAppGen.c 中的 IApplet 接口的实现我们可不是一定要使用的。

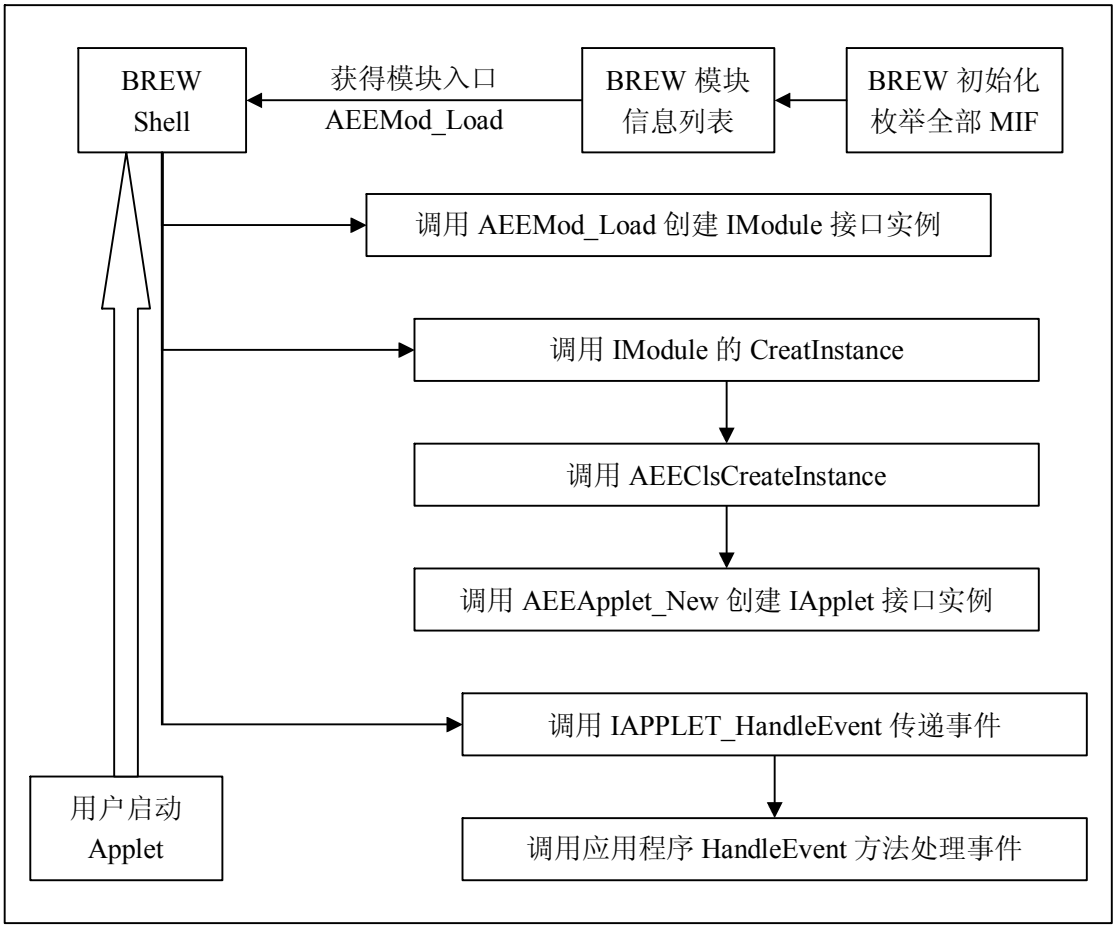

图 11.2 从 IModule 到动态应用程序启动流程

我们知道，BREW Shell 通过 IAPPLET_HandleEvent 方法向应用程序发送事件，在文件 AEEAppGen.c 中关于 IApplet 的实现中，与这个方法相对应的是 AEEApplet_HandleEvent 函数。在上面我们看到了它调用的是我们注册的事件捕获函数（如：helloworld_HandleEvent 函数）。毫无疑问的，这中间增加了一层函数调用，从而增加了程序执行的负担。因此我们可以考虑跳过这层函数的调用进行优化，优化的方法就是在工程中删除 AEEAppGen.c 文件，然后参照此文件中的方法，在主体应用程序所在的文件中实现 IApplet 兼容接口。由于这个自己实现的接口中并不完全仅仅拥有 IApplet 的三个方法，可能基于某种特殊的目的我在这个实现当中又增加了一个叫做诸如 GetRef 的方法之类的功能。但是，无论怎样实现，这个接口的虚拟函数表中前三个方法都必须是 AddRef、Release 和 HandleEvent，这就是“兼容”的含义了。

各位读者们可以试着自己完成这个任务，如果现在还没有任何思路的话，不要急，我们会在后面的扩展 BREW 接口一章中详细的讲解接口的实现方法，届时再返回来实现它也是可以的。或许有人要问，这样的实现方式很影响效率吗？我的回答是可能对于那些追求完美

的程序员来说，这样的问题是不可原谅的，但实际情况是这个中间层的函数十分简单，只有一个 return 的调用（参看 AEEApplet_HandleEvent 函数的实现），因此编译器会将代码进行优化，从而不会有中间函数的复杂调用（或许编译器已经优化到了一条 CPU 指令，不过这只有编译器知道），也就不会有那么大的系统开销。如果应用程序不是非常非常严格的要求效率并且是动态应用程序，那么就不需要修改了。在这里提出这样的议题，目的是为了增加您对动态应用程序启动的理解，仅此而已。

不过，对于静态应用程序，我们就需要修改许多的内容了，目的是为了避免函数的命名冲突。首先，在我们的应用程序文件（如 helloworld.c）中的 AEEClsCreateInstance 函数就需要改名，不能同时在两个静态应用程序的文件中使用同一个函数名；其次就是 AEEMod_Load 函数也需要更改名称，虽然我们仍然可以使用 AEEStaticMod_New 方法。也许顺便的，我们可以一起修改了 IModule 和 IApplet 接口的实现，这就取决于我们的爱好了。不过静态应用程序仅仅对 BREW 设备的 OEM 厂商有意义而已，动态应用程序的开发就接触不到了。

11.3.3 助手（Helper）函数的实现原理

在 BREW 动态应用程序中，由于 BREW 实现原理的原因，每一个接口都需要通过 BREW Shell 进行创建，创建成功之后才能使用接口的功能函数。但是在这里面有一个例外，那就是 BREW 的助手函数如 MALLOC 等，我们可以直接在应用程序中使用这些函数（需要包含 AEEStdlib.h 文件），而不需要任何的接口创建，它们的实现有什么特别的吗？这一节里将揭开 BREW 助手函数实现的神秘面纱。

我们知道，使用这些助手函数唯一要做的就是在源文件中包含 AEEStdLib.h 文件，那么我们就看看这个文件中的如下内容吧：

```
#if defined(AEE_SIMULATOR)     // 在模拟器环境下
/* Simulator case */

#ifdef __cplusplus
extern "C" {
#endif

extern AEEHelperFuncs *g_pvtAEEStdLibEntry;

#ifdef __cplusplus
}
#endif

#define GET_HELPER()         g_pvtAEEStdLibEntry
#define GET_HELPER_VER() 0 /* soon to be gone */

#else /* #if defined(AEE_SIMULATOR) */ // 在设备环境下

#ifdef __cplusplus
extern "C" {
#endif
```

```
extern int AEEMod_Load(IShell *ps, void * pszRes, IModule ** pm);

#ifdef __cplusplus
}
#endif

#define GET_HELPER() (*(((AEEHelperFuncs **)AEEMod_Load) - 1))
#define GET_HELPER_VER() (*((uint32 *)(((byte *)AEEMod_Load) - sizeof(AEEHelperFuncs
*) - sizeof(uint32))))

#endif /* #if defined(AEE_SIMULATOR) */
```

在这部分代码中，GET_HELPER 宏就是用来获得助手函数指针的。在这段代码的前半部分定义了在模拟器环境下获取助手指针的方法，也就是在 Visual Studio 环境下编译时使用的方式，这段代码在“#if defined(AEE_SIMULATOR)”条件编译内。我们可以看到，在模拟器环境下，使用 g_pvtAEEStdLibEntry 做为助手函数的指针，那么，这个变量又是从什么地方来的呢？使用 Visual Studio 的“到达变量定义处”的功能将很容易找到这个变量的位置。我们可以选中这个变量，同时单击鼠标右键，就可以看到这个菜单了。原来这个变量是在 AEEModGen.c 文件中的：

```
#ifdef AEE_SIMULATOR     // 在模拟器环境下
    // IMPORTANT NOTE: g_pvtAEEStdLibEntry global variable is defined for
    //     SDK ONLY! This variable should NOT BE:
    //
    //          (1) overwritten
    //          (2) USED DIRECTLY by BREW SDK users.
    //
    //   g_pvtAEEStdLibEntry is used as an entry point to AEEStdLib,
AEEHelperFuncs *g_pvtAEEStdLibEntry;
#endif
```

从对这个变量的注释中我们可以了解到，这个变量不能够在应用程序中直接使用，并且这个变量是仅仅针对在模拟器上运行的应用程序的，不能用在生成设备上运行的应用程序中。相对应的，这个变量在 AEEMod_Load 函数中进行赋值，我们可以在 AEEModGen.c 中看见如下代码：

AEEStaticMod_New 函数内部

```
#ifdef AEE_SIMULATOR    // 在模拟器环境下
    if (!ph) {
        return EFAILED;
    } else {
        g_pvtAEEStdLibEntry = (AEEHelperFuncs *)ph;
    }
#endif
```

从这段代码中可以看出，这个 g_pvtAEEStdLibEntry 变量是通过 AEEStaticMod_New 函数的 ph 参数传递进来的。就像是 IShell 接口的指针一样，这个参数同样是经过模块入口函数 AEEMod_Load 传递进来的。不过这只是在模拟器的环境下，下面我们就来看看在设备环

境下的实现方法。

我们可以从代码中看到，设备环境下 GET_HELPER 和 GET_HELPER_VER 的定义如下：

```
#define GET_HELPER() (*(((AEEHelperFuncs **)AEEMod_Load) - 1))
#define GET_HELPER_VER() (*((uint32 *)(((byte *)AEEMod_Load) - sizeof(AEEHelperFuncs *) - sizeof(uint32))))
```

十分奇怪的定义，不过不要因为复杂就放弃分析了，请您耐心的看，这只不过是 C 语言的高级应用且多了几个括号而已。AEEMod_Load 符号是 BREW 二进制映像文件的入口地址所在，既然是一个地址，那么它实际上也就是一个指针（关于 C 语言函数做为指针使用的情况，我们已经在本书的第一部分入软件基础一章有所介绍了）。那么，AEEMod_Load -1 则代表是指针减 1。我们也看到了这里首先将 AEEMod_Load 转换成了 AEEHelperFuncs 结构体的二重指针，因此指针地址增减的步长就是一个指针的长度（一重指针），对于 32 位系统来说就是 4 个字节。而对于 GET_HELPER_VER 则减了两个步长（sizeof(AEEHelperFuncs *)+ sizeof(uint32)）。这样处理之后的内存结构如图 11.3 助手函数实现的内存模型所示。

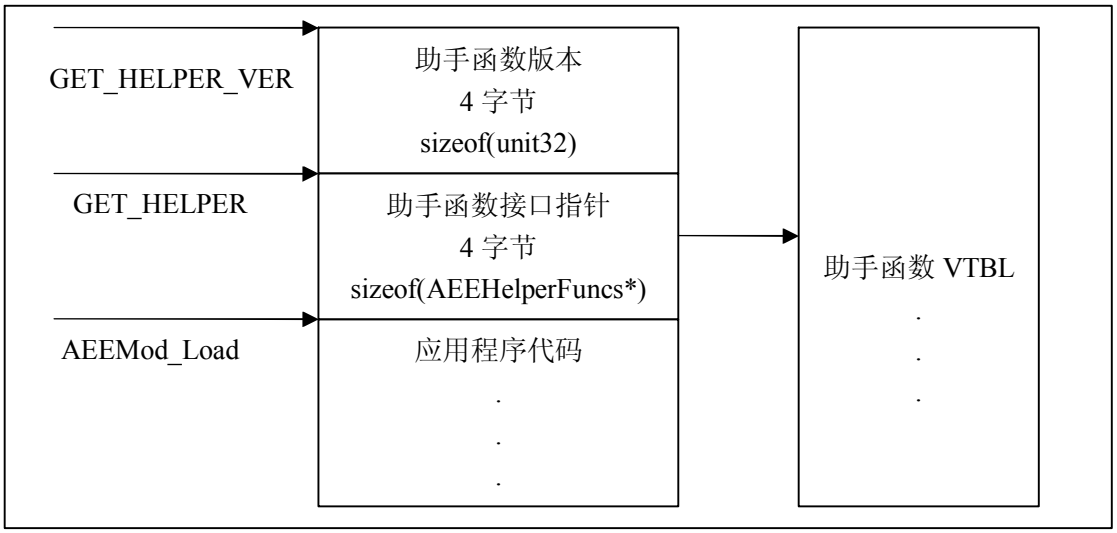

图 11.3 助手函数实现的内存模型

我想通过这个图我们就可以清楚地了解到 BREW 处理助手函数的方法。可以推测得到（我也没有具体实现的源代码，因此只能是推测了），BREW 在载入应用程序的二进制.mod 文件的时候，先分配两个 32 位的数据区域，这两个区域中第一个放置助手函数的版本号，第二个放置助手函数的接口实例指针，然后紧跟着将.mod 二进制文件复制到余下的内存中，然后执行 AEEMod_Load 函数载入模块。通过前两个数据区域传递助手函数接口信息，通过 AEEMod_Load 的参数传递 IShell 接口指针，这样就可以在内存中执行应用程序的代码了。或许现在您会问，为什么模拟器环境下与设备环境下助手函数的实现方法不一样啊？我想原因就出现在获取助手函数的版本上。最初的 AEEMod_Load 函数的设计并没有设计助手函数的版本获取方法，后来需要增加就选择了这样的一种方式，而且模拟器中现在仍然不支持获取助手函数的 GET_HELPER_VER，在模拟器中它始终是 0。这应该属于 BREW 设计上的一个小缺陷，不过没关系，它依然掩盖不住 BREW 的其他优点。

最后需要注意在静态 BREW 应用程序中助手函数的使用方法，由于静态应用程序可能同时有多个，因此如果使用动态应用程序中的助手函数的方法就行不通了，将会发生函数重名的冲突。因此，如果需要在静态应用程序中使用助手函数请包含 AEEStdLib_static.h 文件，并将动态应用程序中包含 AEEStdLib.h 文件的语句删除。如果我们打开 AEEStdLib_static.h 文件，可以看到这个文件中助手函数的实现是与助手函数的函数名直接对应的。

11.3.4 向应用程序发送事件

BREW Shell 提供了几种方法可以用来向指定的应用程序发送事件，它们是：

1、ISHELL_PostEventEx(IShell * pIShell, uint16 wFlags, AEECLSID clsApp, AEEEvent evt, uint16 wp, uint32 dwp)

2、ISHELL_PostEvent(IShell * pIShell, AEECLSID clsApp, AEEEvent evt, uint16 wp, uint32 dwp)

3、ISHELL_SendEvent(IShell * pIShell, AEECLSID clsApp, AEEEvent evt, uint16 wp, uint32 dwp)

4、ISHELL_HandleEvent(IShell * pIShell, AEEEvent evt, uint16 wp, uint32 dwp)

这四个方法可以用来向指定的 BREW 应用程序发送 BREW 事件或用户自定义事件。在 AEEShell.h 文件中查看这些接口的定义如下：

```
#define ISHELL_SendEvent(p,cls,ec,wp,dw)
   GET_PVTBL(p,IShell)->SendEvent(p,0,cls,ec,wp, dw)
#define ISHELL_PostEvent(p,cls,ec,wp,dw)
   GET_PVTBL(p,IShell)->SendEvent(p,(EVTFLG_ASYNC|EVTFLG_UNIQUE),cls,ec,wp,dw)
#define ISHELL_PostEventEx(p,flgs,cls,ec,wp,dw)
   GET_PVTBL(p,IShell)->SendEvent(p,((EVTFLG_ASYNC)|(flgs)),cls,ec,wp,dw)
#define ISHELL_HandleEvent(p,ec,wp,dw)
   GET_PVTBL(p,IShell)->SendEvent(p,0,(AEECLSID)0,ec,wp,dw)
```

从中我们可以看到，它们都使用了 IShell 接口虚拟函数表中的同一个函数 SendEvent，这个 SendEvent 的函数原形如下：

```
boolean (*SendEvent)( IShell * po,              // IShell 接口指针
                      uint16 wFlags,            // 事件属性标记
                      AEECLSID clsApp,          // 接收此事件的应用程序 Class ID
                      AEEEvent evt,             // 发送的事件
                      uint16 wParam,            // 事件的 16 位附加数据
                      uint32 dwParam);          // 事件的 32 位附加数据
```

可以看出 SendEvent 函数的原形与 ISHELL_PostEventEx 是一致的，现在就让我们来解释一下这个函数中各个参数的意义。po 是指向 IShell 接口实例的指针，是使用 IShell 接口必须的，不再多说；clsApp 指定接收此事件的应用程序 Class ID，如果为 0 则表示发送到当前活动的应用程序中；evt、wParam 和 dwParam 分别是事件代码以及此事件的附加数据。

我们主要来看看第二个参数 wFlags。wFlags 是发送事件的属性标记，它表示了传递事件的方式。在当前的 BREW 版本中，它有两个可以使用的标记值（定义在 AEEShell.h 中）：

```
#define EVTFLG_UNIQUE    0x0001     // Only one of this type should be posted...
#define EVTFLG_ASYNC     0x0002     // Event is asynchronous...
```

EVTFLG_ASYNC 标记表示事件采用异步方式处理。异步方式的意义是将事件送入 BREW Shell 内部的事件队列中去，不管事件处理的结果，只要事件加入队列成功函数就返回 TRUE。异步方式发送的事件将在下一个 BREW 信令（参看 15.2 一节）处理是发送给应用程序去处理。如果没有指定此标记，那么事件将立即发送到应用程序的事件捕获函数中进行处理，同时返回事件的处理结果，这与直接调用应用程序的事件处理函数是一样的。

EVTFLG_UNIQUE 标记表示任何时刻这种事件类型只有一个可以等候处理，此标记只有在设置了 EVTFLG_ASYNC 标记时才有效。如果置位此标记，则不会有多个具有相同事

件代码的事件等候应用程序处理。要允许传递多个具有相同事件代码的事件，请不要置位此标记。

对比 SendEvent 参数的讲解与各个事件发送方法的定义，可以得到这些接口的作用了。ISHELL_PostEventEx 方法的参数与 SendEvent 函数的一样，因此它就是 SendEvent 函数的调用。ISHELL_PostEvent 函数设置了 EVTFLG_ASYNC 和 EVTFLG_UNIQUE 标记，因此它的作用是向应用程序中异步的发送事件。ISHELL_SendEvent 函数没有指定任何的标记，因此它的作用是直接向应用程序发送事件，并返回事件处理结果。ISHELL_HandleEvent 方法没有设置任何属性标记，同时将接收事件的 Class ID 定义成了 0，因此它的作用是向当前活动的应用程序发送一个同步处理的事件。

看了上面的方法，我想我们还可以定义一个向当前活动的应用程序发送异步事件的方法了，如下：

```
#define ISHELL_SyncHandleEvent(p,ec,wp,dw)
  GET_PVTBL(p,IShell)->SendEvent(p,(EVTFLG_ASYNC|EVTFLG_UNIQUE),0,ec,wp,dw)
```

这样我们就可以方便的为当前活动的应用程序发送异步事件了。不过不幸的是 BREW 没有将这样的方法做为 BREW Shell 接口的标准功能。

11.3.5 应用程序与接口的区别

到现在为止，我们已经见识了 BREW 应用程序的全过程，结合前面关于 BREW 接口实现的方式，我们是否会在心中有所疑问——接口和应用程序之间有什么样的区别呢？

在回答这个问题之前，我想我们应该先看一看接口和应用程序之间的相同点：

1、接口和应用程序都有 Class ID 与其唯一的对应。每一个接口和应用程序都必须拥有一个唯一的 Class ID 来标识。

2、接口和应用程序都是通过 Class ID 来创建实例的。在 BREW Shell 实现内部，接口的创建函数和应用程序的创建函数都是通过 Class ID 实现对应的。

我们暂时不作任何分析，接着看看显性的接口与实现之间的区别：

1、接口是为应用程序服务的。每个接口存在的意义就在于被应用程序使用，从而完成接口的功能。

2、接口的实现继承自 IBase 接口，而应用程序的实现是继承自 IApplet 接口。

3、接口与应用程序的创建方式不同。接口使用 ISHELL_CreateInstance 方法创建，而应用程序则通过 ISHELL_StartApplet 来创建。

对于不同点的第一条来说，这是接口和应用程序的理所当然的生存意义，就好比饿了要吃东西一样，没什么理由可讲了。对于第二条接口必须继承 IBase、应用程序必须继承于 IApplet 来说，其中存在着某种相同——IApplet 也同样的继承自 IBase 接口。这暗示着应用程序只不过是接口的一个扩展。

而这第三条不同看似理所当然，使用 ISHELL_CreateInstance 接口来创建接口实例，使用 ISHELL_StartApplet 启动应用程序。现在来看两条相同点，同样有 Class ID，同样是通过 Class ID 创建实例，那么，我们可不可以使用 ISHELL_CreateInstance 接口来创建应用程序，使用 ISHELL_StartApplet 来创建接口呢？

当然不行！不要胡思乱想，我问这个问题只是想勾起您的想象力而已，不然还造两个接口干什么。不过，虽然不行，但是我们却可以从中嗅出某种关联的味道来，也正是从接口与应用程序之间的这种关联我们才能真正的了解它们之间的本质区别。现在假设我们已经编写了一个应用程序例如 HelloWorld，如果我们在另外一个应用程序中的某个事件（如

EVT_KEY）中输入如下代码：

```
{
    IApplet *pIApplet;
    int nRet;

    nRet = ISHELL_CreateInstance( pme->a.m_pIShell,
                                  AEECLSID_HELLOWORLD,
                                  (void **)&pIApplet);

    if(SUCCESS == nRet)
    {
        DBGPRINTF("SUCCESS");
    }
    else
    {
        DBGPRINTF("EFAILED %d",nRet);
    }
}
```

我们可以将这些代码输入到 BREW 事例应用程序的 MediaPlayer 中，在 mediaplayer.c 文件中 CMediaPlayer_HandleEvent 函数的 case EVT_KEY 事件中加入上面的代码，然后在调试环境下启动模拟器，打开 MediaPlayer 应用程序，然后按任意的一个数字键，这时输出窗口中将显示 EFAILED 21，打开 AEEError.h 文件，我们可以看到 21 号错误表示权限不够。此时我们需要修改 helloworld.mif 文件和 mediaplayer.mif 文件，在 helloworld.mif 文件的 Extensions 选项卡中，将 helloworld 的 Class ID 做为导出类；在 mediaplayer.mif 文件中将 Privileges 选项卡中的高级部分打开，然后选择 Download 和 All 使得应用程序拥有全部的权限，然后在 Dependencies 选项卡中设置 helloworld 的 Class ID 为导入类 ID。此时再次执行上面的操作，我们可以看到输出了 SUCCESS，这就证明了一个应用程序可以做为输出的接口并创建接口的实例。对于静态应用程序实现这个功能就更加容易了，没有必要设置如此繁琐的 MIF 文件。

从这一点上足以证明，在 BREW 内部处理接口和应用程序的核心方法是一样的，都是采用同样的数据结构在进行 Class ID 的管理。之所以接口不能做为应用程序启动，其主要的原因是：一、通常接口不兼容 IApplet 接口，因此如果使用 ISHELL_StartApplet 将会引起程序的异常；二、即便接口兼容了 IApplet 接口（如控件），也由于通常接口中不会处理应用程序的 EVT_APP_START 等事件而不能正常启动。或者反过来说，如果我们的接口既兼容 IApplet 接口，又处理了应用程序的几个必备的事件，那么我们的接口与应用程序之间就没有任何区别了。

综上所述，接口和应用程序之间通过 BREW Shell 的内部管理机制有机的结合到了一起，虽然他们之间不尽相同，但它们之间的内在联系要比它们的外在表现要紧密地多。掌握了 BREW 接口和应用程序之间的这些关系，就可以更加清楚地把握 BREW 内在本质。

11.3.6 使用应用程序的启动参数

除了简单的使用 ISHELL_StartApplet 方法启动应用程序之外，BREW Shell 还提供了

ISHELL_StartAppletArg 方法来启动应用程序，这个接口的原形如下：

```
int ISHELL_StartAppletArgs (IShell * pIShell, AEECLSID cls, const char * pszArgs )
```

这个方法允许在启动应用程序的时候为应用程序指定命令行参数，这个命令行参数类似于在 DOS 环境下启动应用程序时指定打开的文件一样。所指定的命令行参数会通过应用程序的 EVT_APP_START 事件的 dwParam 的 32 位数据传入。传入时，dwParam 中存储的是一个 AEEAppStart 结构体的指针，这个结构体中的内容由 BREW Shell 指定。结构体的定义如下（参考 AEEShell.h 文件）：

```
typedef struct
{
    int             error;          // 由应用程序填充的错误代码（如果启动时发生错误）
    AEECLSID    clsApp;             // 启动应用程序的 Class ID
    IDisplay *      pDisplay;       // 显示接口
    AEERect     rc;                 // 应用程序的显示区间
    const char  *pszArgs;           // 命令行参数
} AEEAppStart;
```

应用程序每一次启动并收到 EVT_APP_START 和 EVT_APP_RESUME 事件时，都会同时收到这个结构体。不过这个结构体中的 pszArgs 成员只有 EVT_APP_START 事件，并且调用 ISHEL_StartAppletArgs 方法启动应用时才会填充，其余情况这个指针的内容为空。在这个结构体中指针成员不需要应用程序释放资源，BREW Shell 会自动管理这些资源。

通常情况下，命令行参数可以用来指示启动到应用程序的某个状态中去，而不是简单的进入应用程序的初始状态。例如我们可以通过命令行参数决定是直接进入电话簿应用程序的记录列表界面，还是直接进入添加号码界面。具体的使用方法就要看我们怎样的发挥想象力了。这里面提醒各位读者的是，请把命令行参数看作一个存储空间，而不是一个字符串。因为一个存储空间中可以存储不同类型的数据。当然在这个存储空间中的数据不可以有 0，否则就被截断了。如果我们需要通过这个参数传递数字或者是地址，可以使用格式化输出函数 SPRINTF 来将其转换成字符串。

11.4 模拟多线程

BREW 并不支持真正意义上的多线程，因为 BREW 通常运行在一个实时多任务操作系统中的一个任务上。这样的操作系统特点是按照任务优先级的高低，依次的执行。如果一个具有较高优先级的任务一直在运行，那么导致的结果将是低优先级的任务无法执行。在这样的环境下，BREW 中所有的任务都在同一个任务中执行，这将直接导致同一时刻只能有一个 BREW 任务执行。为了能够更加清楚地讨论 BREW 多线程的问题，我们有必要先介绍一下关于线程的东西。

11.4.1 关于多线程

多线程是在一个应用程序内部实现多任务处理的能力。程序可以把它自己分隔为各自独立的线程，或者叫做执行绪，这些线程似乎同时在执行着。这一概念初看起来似乎没有什么用处，但是它可以让程序使用多线程在背景执行冗长作业，从而让使用者不必长时间地无法使用其设备进行其它工作，即使在设备繁忙的时候，使用者也应该能够使用它。

例如在手机设备中，如果当前用户正在执行电话簿中记录的复制过程，这些记录很多，

假设有 1000 条记录，那么这个复制的过程是很长的。如果我们使用一个循环来处理这些记录，那么结果可能是在程序执行的期间内，我们将不能够使用任何其它的应用程序，比如浏览一条短消息等，这对用户来讲是很不方便的。如果现在我们能够有一种方法，使得在复制记录的同时依然可以浏览短消息，那不是很好吗。这就是多线程所要解决的问题。

在通常 PC 的操作系统中，按照线程将 CPU 的执行时间分成若干片段，每一个时间片断都交给一个线程来执行，这样只要时间片断足够的小，那么使用者将不会有任何被中断的感觉。例如在 Windows 操作系统中，如果一个窗口正在处理内容，我们仍然可以终止处理或者进行其他的菜单操作等。

为了能够实现多线程，还要避免多线程的一些陷阱，这些陷阱通常十分的隐蔽，而且难于理解。典型的我们要面临的问题就是线程之间的“争吵”，这发生在程序写作者假设一个线程在另一个线程需要某资料之前已经完成了某些处理（如准备数据）的时候。为了帮助协调线程的活动，操作系统要求各种形式的同步，如果没有这些同步，那么线程之间将会由于公共数据的处理问题而执行错误，且这种错误往往很难查找。同步的方法一种是同步信号（semaphore），它允许程序写作者在程序代码中的某一点阻止一个线程的执行，直到另一个执行绪发信号让它继续为止。类似于同步信号的是临界区域（critical section），它是程序代码中不可中断的部分。

但是同步信号还可能产生称为死锁（deadlock）的常见线程错误，这发生在两个线程互相阻止了另一个的执行，而继续执行的唯一办法又是它们继续向前执行。我们可以列举一个在 16 位系统中经常会发生的情况，假定一个线程执行下面的简单叙述：

```
lCount++ ;
```

其中 lCount 是由其它线程使用的一个 32 位的 long 型态变量，C 中的这个叙述在 16 位编译环境下被编译为两条二进制 CPU 指令，第一条将变量的低 16 位加 1，而第二条指令将任何可能的进位加到高 16 位上。假定操作系统在这两个机械码指令之间中断了线程。如果 lCount 在第一条机械码指令之前是 0x0000FFFF，那么 lCount 在线程被中断时为 0，而这正是另一个线程将看到的值。只有当线程继续执行时，lCount 才会增加到正确的值 0x00010000。

这是那些偶尔会导致操作问题的错误之一。在 16 位程序中，解决此问题正确的方法是将叙述包含在一个临界区域中，在这期间线程不会被中断。然而，在一个 32 位程序中，该叙述是正确的，因为它被编译为一条二进制 CPU 指令。

看来在多线程系统中临界区成为了一个十分关键的因素了。现在我们就来看看通用的临界区是什么样子，之后我们再看 BREW 中的线程是什么样子的。

在单工操作系统中，传统的计算机程序不需要红绿灯来帮助协调它们之间的行为。它们在执行时似乎独占了整条路，而且也确实是这样，没有什么会干扰它们的工作。即使在多任务操作系统中，大多数的程序也似乎各自独立地在执行，但是可能会发生一些问题。例如，两个程序可能会需要同时从同一个文件中读或者对同一文件进行写。在这种情况下，操作系统提供了一种共享文件和记录上锁的技术来帮助解决这个问题。然而，在支持多线程的操作系统中，情况会变得混乱而且存在潜在的危险。两个或多个线程共享某些数据的情况并不罕见。例如，一个线程可以更新一个或者多个变量，而另一个线程可以使用这些变量。有时这会引发一个问题，有时又不会（记住操作系统将控制权从一个线程切换到另一个线程的操作，只能在机器码指令之间发生。如果只是一个整数被线程共享，那么对这个变量的改变通常发生在单个指令中，因此潜在的问题被最小化了）。

然而，假设线程共享几个变量或者数据结构。通常，这么多个变量或者结构的字段在它们之间必须是一致的。操作系统可以在更新这些变量的程序中间中断一个线程，那么使用这些变量的线程得到的将是不一致的数据。结果是冲突发生了，并且通常不难想象这样的错误将对程序造成怎样的破坏。我们所需要的是类似于红绿灯的程序写作技术，以帮助我们对线

程交通进行协调和同步，这就是临界区域。大体上，一个临界区域就是一块不可中断的程序代码。

在 BREW 中，由于应用程序运行在操作系统的一个任务之中，因此并不存在真正意义上的多线程，每一个处理序列都会在另一个处理序列完成之后再执行，因此通常并不需要使用临界区来控制程序的执行。不过在 BREW Shell 的的内部实现里，通常需要有临界区的控制，主要是为了 BREW 所在的任务与系统中其它任务之间的交互而使用。虽然在 BREW 中每个执行序列的单元不会被中断，但是我们还是需要小心的处理多个执行序列之间共享数据的问题。就如同上面介绍的关于多线程的相关知识一样，多一些这样的理解对于编写 BREW 应用程序也是很有好处的。通常大家都会认为 BREW 不支持多线程，准确地应该说 BREW 不支持真正的多线程，但是支持程序中的多个执行序列，我们就算做部分支持多线程吧，这也是本节标题“模拟”的意义所在。

11.4.2 使用 BREW 回调（CallBack）模拟多线程

BREW Shell 为我们提供了一种回调函数的机制，通过回调函数我们可以利用 BREW 的信令循环分批次的完成应用程序中进行的大量重复性的操作。这样可以防止在一个 BREW 事件处理过程中占用过多的 CPU 运行时间而导致问题的发生（发生问题的原因在于影响了操作系统处理任务的实时性）。与回调函数有关的接口和数据结构主要有：

1、ISHELL_Resume 方法

2、AEECallback 结构

3、CALLBACK_Cancel、CALLBACK_Init 和 CALLBACK_IsQueued

首先让我们来看看 ISHELL_Resume 方法，这个方法的函数原形如下：

```
void ISHELL_Resume (IShell * pIShell, AEECallback * pcb );
```

这个函数使用了一个 AEECallBack 结构体作为参数，执行后将 AEECallback 结构体中的数据加入 BREW 内部的数据处理队列中去，在下一个 BREW 信令循环中将执行在 pcb 参数中指定的回调函数以，使应用程序或对象协同处理多任务。如果已注册回调，则会先将其注销，然后再重新注册。CALLBACK_Cancel、CALLBACK_Init 和 CALLBACK_IsQueued 分别是进行回调设置时的辅助宏定义，它们与 AEECallback 结构体的定义如下（请参考 AEE.h 文件）：

```
typedef struct _AEECallback AEECallback;

typedef void (*PFNCBCANCEL)(AEECallback * pcb);

struct _AEECallback
{
    AEECallback *pNext;                 // RESERVED
    void        *pmc;                   // RESERVED
    PFNCBCANCEL  pfnCancel;             // Filled by callback handler
    void        *pCancelData;           // Filled by callback handler
    PFNNOTIFY    pfnNotify;             // Filled by caller
    void        *pNotifyData;           // Filled by caller
    void        *pReserved;             // RESERVED - Used by handler
};
```

```
#define CALLBACK_Init(pcb,pcf,pcx) {(pcb)->pfnNotify = (PFNNOTIFY)(pcf); \
                                    (pcb)->pNotifyData = (pcx);}
#define CALLBACK_Cancel(pcb)       if (0 != (pcb)->pfnCancel) (pcb)->pfnCancel(pcb)
#define CALLBACK_IsQueued(pcb)     (0 != (pcb)->pfnCancel)
```

现在我们就来解释一下 AEECallback 结构体中各个成员的意义。pNext 和 pmc 为 BREW Shell 保留使用的数据成员，调用 ISHELL_Resume 方法的程序不得修改此成员。它们用来管理回调函数时建立一个回调的数据链表时使用，如果我们修改了此数据，可能会引起系统回调函数机制的失灵。

pfnCancel 是一个函数指针，用来取消一个尚未执行的回调。此数据也由 BREW Shell 指定，用户不可以修改。通过这个函数可以将当前的回调从回调数据链表中取消掉，这样在下一次 BREW 信令循环时就不会执行这个回调了。调用程序必须将此指针设置为 NULL。pCancelData 是传递给 pfnCancel 所指定函数的数据，调用程序不得修改此成员。

pfnNotify 是 BREW Shell 调用的回调函数。使用者必须将此指针设置为指向回调处理程序所调用的函数，以便于执行应用程序中的处理。pNotifyData 是传递给 pfnNotify 函数的数据，调用程序必须将此指针设为指向须传递给 pfnNotify 函数的数据。通常情况下，这个数据指向应用程序的私有结构体，这样就可以在回调函数中获得全部应用程序的数据了。

为了简化操作，BREW 预先定义了 CALLBACK_Cancel 和 CALLBACK_Init 宏用来取消和初始化一个回调的数据结构。通常使用回调函数的方法如下：

```
… 某使用回调的函数内部
{
    AEECallBack myCB;

    CALLBACK_Init(&myCB, MyFuncsCB, pMe);

    if(!CALLBACK_IsQueued(&myCB))
    {
        ISHELL_Resume(pMe->m_pIShell, &myCB);
    }
}
… 某使用回调的函数内部

// 回调处理函数
void MyFuncsCB(MyApp *pMe)
{
… 用户数据处理
   // 如果仍然需要继续处理回调，可以在这里再次调用 ISHELL_Resume 方法
}
```

在这段示例代码中，CALLBACK_IsQueued 用来判断是否此回调函数已经加入了回调列表中。如果已经加入了，将不再次加入了。通常情况下，一旦调用 ISHELL_Resume 成功，pfnCancel 数据成员会立即由 BREW Shell 设置一个指定的函数。在应用程序中加入这样的处理之后，就仿佛增加了一个线程一样，只要不停的循环，这个线程就会不停的运行。如果同时设置多个回调函数，就仿佛有多个线程在运行一样。只不过由于 BREW 的特点，这些回调在执行时会阻止 BREW 其他事件的处理，在用户角度来看还是会影响应用程序的运行。

不过由于将应用程序循环执行的代码分成了在多个事件循环中完成，从而避免了过长的单个事件处理时间。通常情况下使用 BREW 的回调函数机制进行数据的复制或用户文件的转储等操作。

11.4.3 使用 BREW 事件模拟多线程

从上面的回调函数的处理机制中我们可以知道，BREW 中实现多线程的方式就是利用 BREW 的信令循环机制，分批地处理数据。进一步的我们能否通过一个事件的循环来实现这样的功能呢？

如果要实现这样的功能，我们需要这样的一个事件：这个事件可以通过 BREW 的信令循环方式发送，而不是直接执行。想一想我们前面介绍的关于向 BREW 应用程序发送事件的部分，什么样的事件发送方式是异步的而不是同步的？那就是 ISHELL_PostEvent 啊。异步事件的方式不就是通过 BREW 的信令循环发送到应用程序的吗？因此我们也可以同样的使用异步事件处理的方式模拟多线程的处理。在应用程序收到这个事件的时候，调用一个事件处理函数，这个函数就如同前面的回调函数一样。或许由于在自己的应用里面指定处理函数，我们可以采用比回调函数更加灵活的处理方式（起码我们不用固定于 PFNNOTIFY 函数类型了）。

在这里提出这样的方式是希望拓宽各位读者的思路，了解 BREW 内部处理的一些本质的东西，因此就不再费力的提供事例代码了。

11.4.5 使用 IThread 接口模拟多线程

IThread 是一个基于回调的 BREW API，通过它可以模拟线程处理。它提供了启动和停止线程的方法。IThread 不可重复使用，即线程从其启动函数返回后，ITHREAD_Start 只能调用一次；调用 ITHREAD_Exit 或 ITHREAD _Stop 之后，不能再次调用 ITHREAD_Start。通常使用 IThread 接口函数的顺序如下面的代码所示：

```
… 某个使用线程的函数并已经提供应用程序数据结构指针 pMe 和 IThread 接口 m_pThrd
{
    // 首先调用 ITHREAD_HoldRsc 将在线程中使用的接口与 IThread 的生命周期关联
    // 这里面我们假设了一个 IAddrBook 接口和一个 IFileMgr 接口
    (void)ITHREAD_HoldRsc(pMe->m_pThrd, (IBase *)pMe->m_pAddrBook);
    (void)ITHREAD_HoldRsc(pMe->m_pThrd, (IBase *)pMe->m_pFileMgr);

    // 调用 ITHREAD_Start 启动线程
    if(SUCCESS != ITHREAD_Start( pMe->m_pThrd,
                                 512,
                                 ThrdFuncs_StartCB,
                                 (void *)pMe))
    {
        return EFAILED;
    }
}
… 某个使用线程的函数
```

```
int ThrdFuncs_StartCB(IThread *piThread, MyApp *pMe)
{
    // 调用 ITHREAD_Join 设置回调函数，可以设置多个
    AEECallback myCB;
    int nRet;

    CALLBACK_Init(&myCB, MyFuncsCB, pMe);
    ITHREAD_Join(piThread, &myCB, &nRet);

    // 调用 ITHREAD_Malloc 为线程内函数分配内存
    // 调用 ITHREAD_Free 释放已经分配的内存
    //
    return SUCCESS;
}

// 回调处理函数
void MyFuncsCB(MyApp *pMe)
{
   … 用户数据处理
   // 调用 ITHREAD_GetResumeCBK 获得当前的回调的指针
   // 调用 ITHREAD_Suspend 挂起线程
   // 如果需要增加回调函数需要再次调用 ITHREAD_Join
}
```

通过上面的方法就可以使用 BREW 的 IThread 接口模拟线程的执行了。在 IThread 接口的实现中，它首先使用 ITHREAD_Start 方法调用线程的启动函数，通过这层的封装，我们可以切换线程的堆栈（栈的大小可以根据我们线程的规模在 ITHREAD_Start 的 nStackSz 参数中定，一般情况下 512 个字节应该够用了）。使用这个方法创建的线程也就拥有了自己的堆栈管理，从调用线程的启动函数（本例中是 ThrdFuncs_StartCB 函数）开始，就进入了线程的执行空间。为了保证在线程中使用的资源能够与线程的生命周期相关，IThread 接口还提供了 ITHREAD_HoldRsc 方法存储一个继承自 IBase 的接口（全部 BREW 接口均继承自 IBSE，因此可以关联全部的接口），这些关联的接口可以在 IThread 释放时跟着自动释放。我们也可以通过 ITHREAD_ReleaseRsc 方法取消这种关联。在线程的处理过程中，推荐使用 ITHREAD_Malloc 和 ITHREAD_Free 接口分配和释放内存，通过这些接口管理的内存也可以和 IThread 的生命周期关联起来。

ITHREAD_Join 函数用来连接一个新的回调函数到线程中去，通过在回调函数中循环的调用 ITHREAD_Join 方法，可以保证线程一直运行下去。在 IThread 的内部，管理了一个回调的链表，IThread 正是通过调度这个链表中的回调来完成线程处理的。有的时候可能要求线程暂停运行，暂停线程的方法是调用 ITHREAD_Suspend 接口，使用这个接口之前需要使用 ITHREAD_GetResumeCBK 接口获得恢复时的回调，当恢复回调时将此事获得的回调指针传递给 ISHELL_Resume 接口，就可以恢复线程的运行了。结束线程时可以调用 ITHREAD_Stop 或者 ITHREAD_Exit 接口。线程终止后需要释放线程接口，不可以再次调用 ITHREAD_Start 方法重新启动线程。

从调用 ISHELL_Resume 接口来恢复线程运行这一点来看，在 BREW Shell 的内部，线

程接口 IThread 所管理的回调列表与 BREW 系统回调之间存在着某种的联系。不过由于关于 IThread 接口的说明太少，具体它们之间的关联是怎样的也无从猜测了。不过可以确信的一点就是 IThread 接口的运行方式与我们前面提到的两种模拟线程的方式核心是一样的。

11.5 闹钟与定时器

在嵌入式系统中，闹钟和定时器的功能有时是必不可少的，例如在手机中他们都是常用的功能。闹钟通常用来设置一个闹铃，在一天的某个时间提醒设置者做某件事情，还可以使用闹钟功能开发类似日程表的应用程序。定时器通常用来进行计时，将计时的结果显示在屏幕上，可以使用它来开发类似于秒表的应用程序。闹钟用于长时间的等待过程中，定时器用于短时间的定时钟。在一个系统的实现里，定时器的应用往往比闹钟的应用更加广泛，因为在开发的过程中使用短时间定时的几率要比使用闹钟的高的多。现在我们就来看看在 BREW 平台中关于定时和闹钟的相关内容。

15.5.1 定时功能的实现

对于一个系统来说，从 CPU 的时钟频率开始，都离不开对时钟的依赖。芯片（包括 CPU）的运行速度是以时钟频率来衡量的，系统地反应时间也是与时钟频率相关联的。可以说时钟频率是一个平台得以成功的关键所在，是数字电路系统中的“脉搏”。正是这个脉搏的不断冲击，才一次又一次的激发了芯片的运行，才有了现在丰富多彩的数字世界。

对于软件来说，看上去与硬件的时钟频率之间没有明显的联系，不过请不要忘记，软件赖以生存的环境——CPU 就是一个基于时钟频率的芯片。CPU 要使用时钟频率的激发来运行程序，同时在 CPU 的外围电路中，还会提供一个定时的硬件中断，每隔一定的时间，如 1ms、2ms、5ms、10ms 等等，产生一个 CPU 中断，为这个系统提供一种定时的方法。这个底层的定时方法就是在软件上层实现定时器的基础。通常基于一个 CPU 的时钟系统的简图如下：

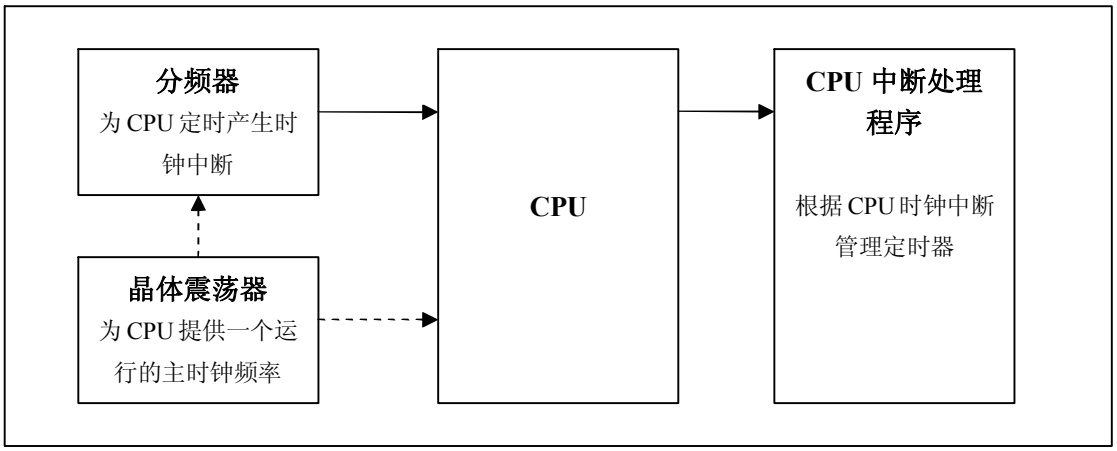

图 11.4 时钟系统结构图

图中的晶体震荡器产生 CPU 运行时所需的时钟频率，同时为 CPU 时钟中断提供频率。分频器将晶体震荡器中的时钟频率进行分频后提供一个定时的时钟中断。当然分频器中输入的频率可以采用与 CPU 不同的频率输入源，在这里只是提供一种方式而已。这个定时中断产生的时间间隔可以是 2.5ms、5ms、10ms 等等，这与我们系统的处理能力有关。典型的例子是在一个 CPU 主频为 40MHz 的 ARM7TDMI CPU 系统中，时钟中断的时间间隔是 5ms。

也许有人会问，5ms 一个中断是不是太短了？这就要看在 5ms 的时间里 CPU 可以做多少事情了，我们假设这个 CPU 的一个指令周期是 4 个时钟频率（对于支持流水线的 CPU 来说平均也就是 3 个左右），那么执行每条指令的时间就是 1/(40M/4) = 0.1us。如此我们可以计算出 5ms 执行的指令是 5ms/0.1us = 50000 条指令。这 5 万条指令已经可以做很多事情了，因此 5ms 的时钟中断时间是足够系统运行的。不过这个系统的定时精度也只能是 5ms 了。

在芯片集成度日益提高的今天，通常会将 CPU 和一些外围电路集成到一起，在手机中这种芯片叫做基带芯片（与射频芯片相对应）。基带芯片中集成了很多的功能，包括上面提到的时钟中断的分频器。而且还可以通过基带芯片中的寄存器来配置分频器的分频倍数，这样我们就可以根据需要配置我们的定时精度了。

有了这样的一个定时中断，我们就可以在中断处理程序中添加定时管理的功能了。可以想象的到，我们可以管理一个定时器的数据链表，将这个链表中的定时时间每 5ms 减少一次，当定时时间到 0 的时侯触发定时器的回调函数。通过这样的方式我们就可以实现系统的定时功能了。这个数据结构可以类似如下的形式：

```
typedef struct timer_struct {
    struct timer_struct *prev_ptr;        /* 连接到前一个定时结构                */
    struct timer_struct *next_ptr;        /* 连接到下一个定时结构                */
    int    cntdown;                       /* 定时剩余毫秒数                      */
    void (*routine_ptr)(int,void*);       /* 定时到达时的处理函数                */
    void *usr_data_ptr;                   /* 用户自定义数据指针，用于回调函数    */
} timer_type;
```

这个结构体中的前两个数据是用来管理双向链表时所使用的，cntdown 是用来记录剩余的毫秒数的，routine_ptr 是一个定时器到达时调用的处理函数，usr_data_ptr 用来给回调函数作为参数的用户数据指针。通过 timer_struct 结构体的一个双向链表，我们就可以管理系统的多个定时器了。这就是基于 CPU 的时钟服务程序的思想。通过对这些内容的了解，我相信您也可以理解 BREW 定时器的实现了。

由于 BREW 是系统中的一个上层应用程序开发的平台，在软件的结构上，它是用 OEM 曾调用底层的时钟服务，这个底层的时钟服务就类似于我们上面介绍的基于 CPU 中断的程序了。有了这样的一个时钟服务，要实现 BREW 的定时功能就容易了，起码我们的实现思路已经有了。为了减少 BREW 移植的难度，在 BREW 内部也管理了一个定时器链表。通过对内部这个链表的管理，BREW 只需要底层的时钟服务程序的一个定时器就够了，因为每一次我们只需要将定时时间最短的一个定时器，设置到底层的定时器链表中就可以了。这个 BREW 内部的链表管理方式与底层平台的管理方式大同小异，所以就不再详细介绍了，相信您已经可以理解它的实现方式了。

15.5.2 闹钟功能的实现

与定时器毫秒级别的时间设置精度不同，闹钟可以设置一日或多日的某一分钟进行提醒。从本质上来讲，闹钟只是定时器功能的一个应用。我们可以首先查看一下设置闹钟和取消闹钟的两个街口的函数原形：

```
int ISHELL_SetAlarm (IShell * pIShell, AEECLSID cls, uint16 nUserCode, uint32 nMins );
int ISHELL_CancelAlarm (IShell * pIShell, AEECLSID cls, uint16 nUserCode ) ;
```

从这两个函数的说明中我们可以知道，在设置闹钟时，需要指定闹钟事件 EVT_ALARM 发送到的应用程序 Class ID，用户定义的 16 位闹钟代码 nUserCode 以及 32 位的从设置闹钟

时刻开始定时的分钟数。取消的时侯是通过 Class ID 和用户定义的闹钟号码来决定。由此可以看出，在 BREW 的内部，闹钟是通过 Class ID 和用户代码两个要素进行区分的。在 BREW 内部存储了一个闹钟结构的数据空间，这些数据同时还会存储到 BREW 系统的配置文件中，以便于关机之后仍然可以保存这些数据。由区别闹钟的两个要素我们可以推测，在这个结构体中，至少需要包含定时时间、Class ID 和用户数据三个成员变量。相应的可能数据结构如下：

```
typedef struct _AlarmInfo{
    AEECLSID cls;            // 接收 EVT_ALARM 事件的应用程序 Class ID
    uint32 nExpireMin;       // 定时时间
    uint16 wUserCode;        // 闹钟代码
}AlarmInfo;
```

在设置的闹钟重复时，将会覆盖原有的闹钟数据，而设置成新的闹钟参数。判断重复的标准就是查看 cls 和 wUserCode 成员是否全部相同。每当存储这个结构体的数据被修改时，都将所有的数据重新写入文件系统中，以便于保存修改的内容。对于一个需要等待较长时间的闹钟来说，会通过设定一个较长时间的定时器来实现，而定时器的回调函数则是通过 BREW Shell 的闹钟管理部分指定的，在这个回调函数被调用的时侯将会根据闹钟数据结构向相应的应用程序中发送 EVT_ALARM 事件。从 BREW 定时器管理的方式我们可以知道，这种长时间的定时器几乎不会占用过多的系统资源，因此并不会影响系统的运行速度。

BREW 通过对闹钟数据的维护，完成了闹钟所需要的功能，其中包括通过文件系统存储闹钟数据的方式，这样就可以在下次开机的时候依然能够获得闹钟的数据。不过此时您可能有这样的疑问：由于设置闹钟的时候使用的时间长度是从当前时间开始的分钟数，那么这个分钟数是否也是使用定时检查的方式呢？如果采用这种方式，那么在关机之后如何实现定时递减呢？答案是闹钟并没有采用定时递减的方式，而是在存储的数据上做了一些手脚。设想一下，如果将我们设置的闹钟时间长度加上当前的时间（比如说 2005 年 12 月 25 日 13：00），那么这个闹钟到达的时间不久变成了一个具体的时间值了吗（如 2005 年 12 月 26 日 13：00）？这样存储的数据就不需要定时检查了，只需要在系统启动的时候设置一个定时器就可以了。既然现在已经不使用定时检查的方式了，那么第二个问题就不复存在了。这些就是 BREW 闹钟功能实现的原理，了解了这些，您也可以编写自己的闹钟处理机制了。

15.5.3 为什么定时器变慢了？

当我们编写一个连续定时应用程序的时侯，如一个秒表，在这个程序运行一段时间之后，我们会发现定时器变慢了！？是我们编写的应用程序出问题了吗？其实这源于 BREW 设备的一种节能方式。

我们知道，从 CPU 上电的那一刻起，CPU 就会马不停蹄的以白分百的速率运行。不要感到惊奇，事实如此！即便是所有的任务都已经完成了，操作系统也会运行一个 for(;;)的死循环程序，CPU 永远不会空闲。原因是这样的，我们的程序之所以能够运行，这在于 CPU 上电的时候会从某个地址开始取指令去执行（通常是 0 地址），从这一刻开始 CPU 运行在我们程序的控制之下了。如果此时 CPU 停止运行了，直接导致的结果是 CPU 不会去取指令执行，那我们的程序又如何控制 CPU 呢？这就是让您理解，程序执行的主旨是 CPU 取指令运行，而不是指令让 CPU 运行。

或许有人会问，为什么系统中还需要有计算程序执行效率的工作呢，照这里的说法执行效率不适 100%了吗？CPU 永远都是在使用同样的效率在运行——100%的在运行，不过程

序的执行效率与 CPU 的运行效率不一样。我们通常会计算一个模块占用了多少的 CPU 运行时间，我们称之为 MIPS（百万指令每秒）。通常每一个 CPU 根据它的主频可以计算出它的总共 MIPS，然后通过记录一定时间内每个任务占用了多少的 CPU 时间而计算出这个任务的 MIPS 数值。通过对系统中多个任务的计算，从而计算出系统程序的执行是否超出了系统的处理能力。如果各个任务 MIPS 总和超出了 CPU 总共的 MIPS，那么证明这个系统的 CPU 将不能在指定的时间内处理完所有的任务。这是十分危险的，因为我们将不能实现系统所需要的功能，并且由于系统的任务不能及时地处理而导致系统崩溃。但是，这些问题与 CPU 的执行效率没有关系，而只与 CPU 的处理速度有关，因为任何时刻 CPU 的效率都是白分百。系统崩溃的原因不是因为 CPU 运行了太多的程序，而是因为系统中各个任务不能及时地配合而引起的。

百分百运行，这很浪费，不是吗？在根本不需要执行程序的时候，却偏偏要去执行。这个问题在使用电池供电的嵌入式系统中尤为突出，比如在手机中。由于电池的供电能力有限，而用户们却希望手机的待机时间可以更长，这中间就涉及到了一个如何节能的问题。CPU 这样的一直高速运行是十分浪费的，因此我们必须找出一种方法能够让 CPU 停止运行或者也不要这样的全速运行。前面已经讲过了，如果 CPU 停止运行了我们的程序也就不能运行了，这样也就不能控制系统了。我们也知道，CPU 的运行效率是一直不变的，不过，CPU 运行的速度是可以改变的啊！我们可不可以在系统任务繁忙的状态下让 CPU 在最高速度下运行，而在空闲的时候让他在最低的速度上运行呢？当然可以了，这正是大多数系统采用的节能方式。这种 CPU 低速运行的状态叫做系统的睡眠状态。通常操作系统会指定一个最低优先级的任务来处理睡眠状态，在这个任务中除了降低 CPU 的执行速度外，还可以关闭在睡眠状态中不必使用的设备，真是一举多得啊！进入睡眠模式后，通常使用按键来唤醒系统退出睡眠模式。

此时降低系统运行速度方式就是降低 CPU 的时钟频率，这样 CPU 执行指令的时间延长了，自然的也就节能了。CPU 的时钟频率被降低了，那么表示与 CPU 时钟频率相关的设备运行频率也会降低。当然了，定时器中断的时间间隔自然而然的也就延长了，在这种情况下，BREW 的每一个定义器也延长了（变慢了），这就是为什么我们的秒表应用程序会在运行一段时间之后变慢的原因了。为了避免这个问题的发生，我们需要在我们的应用程序中捕获 EVT_APP_NO_SLEEP 事件。因为在系统进入睡眠模式之前，BREW 会首先发送这个事件到当前正在运行的应用程序中，如果这个事件返回了 TRUE，则表示应用程序正在运行，不允许系统进入睡眠模式，否则进入睡眠模式。因此只需要在我们的应用程序收到这个事件的时候返回 TRUE，就可以避免定时器变慢的问题了。

11.6 对话框、消息框和提示框

在本章的第一节中，您已经看到了关于对话框的一些介绍，我们这里就不再重复那些基本的问题了。这一部分的主要任务是让您了解 BREW 对话框的内部工作方式，使得您可以更加灵活的在您的应用程序中使用对话框。

11.6.1 对话框的创建和终止

我们前面讲过，使用 ISHELL_CreateDialog 来创建一个对话框，这个接口函数的原形如下：

```
int ISHELL_CreateDialog ( IShell * pIShell,
```

```
                    const char * pszResFile,
                    uint16 wID,
                    DialogInfo * pInfo ) ;
```

除了需要的 IShell 接口指针外，这个接口的参数还有资源文件名以及在资源文件中对话框资源的 ID，最后一个参数是 DialogInfo 类型的一个结构体。在大多数情况下，我们使用在资源文件中已经创建好的对话框，因此最后一个 pInfo 参数设置为 NULL 就可以了。当我们指定了资源文件以及对话框的 ID 后，调用此接口，如果一切正常，一个我们在资源文件编辑器中设置好的对话框就会呈现在面前。在这个过程中会有两个事件发送到创建对话框的应用程序中：EVT_DIALOG_INIT 和 EVT_DIALOG_START，这两个事件是以异步的方式传递到应用程序中的（相对于调用 ISHELL_CreateDilaog 接口的时候）。EVT_DIALOG_INIT 事件可以用来控制当前对话框中的控件，如在菜单控件中增加项目、设置控件的属性等，还可以用来阻止对话框的创建（对这个时间返回 FALSE）。不过由于在这个事件之后，Dialog 内部仍然会做一些处理，因此，请不要在 EVT_DIALOG_INIT 事件中进行如下处理：一是设置控件的显示区域，因为在 EVT_DIALOG_INIT 事件之后，对话框会根据当前对话框内的控件进行显示区域的调整，如将 SoftKey 控件设置到显示区的底部等；二是设置控件的标题，因为在 EVT_DIALOG_INIT 事件之后，对话框会设置标题为资源文件中指定的标题。

在 EVT_DIALOG_START 事件中，我们可以任意的控制在此对话框中的控件，因为一旦这个事件返回了 TRUE，那么对话框就会立即连同其中的控件显示出来。因此，这里建议在使用对话框的时候将大部分的处理全部放在 EVT_DIALOG_START 事件中进行，这样既不会影响我们的应用程序执行，还可以任意的控制其中的控件。不过，在 EVT_DIALOG_START 事件中处理屏幕描绘是无效的，因为在此事件返回 TRUE 之后，对话框会调用清空屏幕的操作然后刷新其中的控件，在此之前的所有对屏幕刷新操作都是无效的。

除了创建在资源文件中的对话框之外，我们还可以通过指定 DialogInfo 结构体来创建一个对话框。DialogInfo 结构体及其相关的数据结构的定义如下（请参考 AEEShell.h 文件）：

```
// 控件列表项目的数据结构
typedef struct _DListItem
{
    uint16      wID;            // ID of the item
    uint16      wTextID;        // Text
    uint16      wIconID;        // Icon ID
    uint16      pad;
    uint32      dwData;         // 32-bit data
} DListItem;

// 列表项目的信息头结构体
typedef struct
{
    AEECLSID    cls;            // class of the control
    uint16      wID;            // ID of the control
    uint16      nItems;         // Item count
    uint32      dwProps;        // See IControl (or specific control)
    uint16      wTextID;        // ID of initial text
    uint16      wTitleID;       // ID of initial title
```

```
    AEERect         rc;                 // Rect - Relative to Dialog (-1, -1, -1, -1) is default
} DialogItemHead;

// 列表结构体
typedef struct
{
    DialogItemHead h;
    DListItem       items[1];
} DialogItem;

// 对话框信息头结构体
typedef struct
{
    uint16          wID;                // Dialog ID
    uint16          nControls;          // Number of controls
    AEERect         rc;                 // Rect (-1, -1, -1, -1) is default
    uint32          dwProps;            // See IControl
    uint16          wTitle;             // Title
    uint16          wFocusID;           // ID of focus control
} DialogInfoHead;

// 对话框信息结构体
typedef struct
{
    DialogInfoHead h;
    DialogItem      controls[1];
} DialogInfo;
```

一个对话框的完整描述由 DialogInfo 数据结构描述，这个结构中主要的部分是一个对话框的信息头 DialogInfoHead，在这个信息头中包含了对话框 ID wID、对话框的个数 nControls、对话框的显示区域 rc、对话框的属性 dwProps、对话框的标题 wTitle 以及对话框的焦点控件 ID wFocusID 的信息。在这些信息里面，还没有支持对话框的标题，因此可以不予理会。

对于对话框中的每一个控件来说，也有控件头信息结构 DialogItemHead，里面包含了控件的 Class ID cls、控件的 ID wID、控件中包含的项目数量 nItems、控件的属性 dwProps、控件的初始文本 ID wTextID、控件的初始标题 ID wTitleID 以及控件的显示区域 rc。正是在这些结构体的包含于嵌套之中才描绘了一个完整的对话框结构，为了更为直观，请看图 11.5：

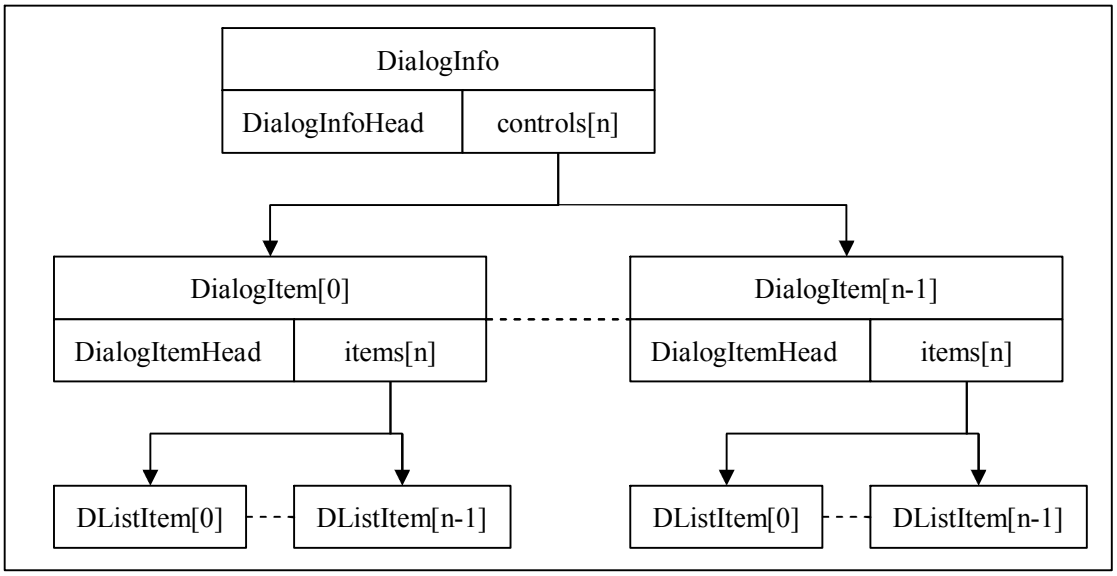

图 11.5 对话框的数据结构

相信从这幅图中您可以了解到对话框数据结构的组成了，在实际使用的过程中，这个数据结构采用的是尽可能紧凑原则。例如，假如一个控件中没有包含任何的项目，那么这个结构体中只能包含一个头信息，也就意味着，在我们定义的数据结构中将不能够直接使用 DialogItem 的类型定义（因为这个结构体类型中包含了一个项目 items[1]的定义）。因此，如果我们需要在实际的开发过程中通过 DialogInfo 创建对话框的话，最好创建自己的 DialogInfo 数据类型。假设现在我们需要创建一个包含三个控件的对话框，其中一个对话框没有项目、一个有一个项目、另一个有两个项目，那么可能的数据类型定义如下：

```
typedef struct _MyDialogInfo{
    DialogInfoHead dh;          // 对话框头信息，必须放在第一位
    DialogItemHead dih1;        // 第一个控件的头信息，由于没有子项目，只能有信息头
    DialogItem     di2;         // 第二个控件，包含一个子项目，可直接使用 DialogItem
    DialogItemHead dih3;        // 第三个控件的头信息
    DListItem      dli31;       // 第三个控件中第一个子项目
    DListItem      dli32;       // 第三个控件中第二个子项目
}MyDialogInfo;
```

在这个结构体的定义中，第三个控件的定义也可以采用如下的形式：

```
    DialogItem     di3;         // 第三个控件的头信息及其第一个子项目
    DListItem      dli32;       // 第三个控件中第二个子项目
```

定义这样的结构体之后，我们可以定义一个这种类型结构体的变量，然后将内容依次进行填充就可以了。这种使用方法是十分麻烦的，而且也不好用，因此很少会使用到。不过从中我们可以窥探出对话框的内部数据处理是按照这种紧凑的数据进行排列的，在使用对话框的时候要千万注意这个问题。

一个创建成功的对话框可以通过 ISHELL_EndDialog 来关闭，同时，在创建对话框的应用程序被关闭或者挂起的时候也会关闭对话框。对话框还支持嵌套，目前最多可以有 5 级的对话框重叠，处于最顶层的是当前活动的对话框，当使用 ISHELL_EndDialog 关闭对话框时，将关闭处于最顶层的对话框并返回下一级的对话框。

11.6.2 使用对话框管理控件

使用对话框的一个理由就是它可以方便的管理其中的控件。对话框可以自动地管理控件的焦点切换，和事件捕获。还可以根据传递进来的对话框数据结构（无论是通过资源文件还是通过 DialogInfo 指针）自动创建并设置控件的属性、标题和调整显示区域等。为了实现这些管理，在对话框的内部管理了一个全部控件的数据链表，通过这个数据链表来控制全部的控件。

我们知道，全部的控件都继承自 IControl 接口，因此这是对话框能够管理控件的关键所在。因为对于任何一个控件，对话框都可以通过 IControl 接口对其进行控制。IControl 接口提供了如下的一些方法：

```
ICONTROL_AddRef()          // 继承自 IBase，管理资源
ICONTROL_GetProperties()   // 获得控件属性
ICONTROL_GetRect()         // 获得控件显示区域
ICONTROL_HandleEvent()     // 控件事件捕获函数
ICONTROL_IsActive()        // 判断控件是否处于活动状态
ICONTROL_Redraw()          // 刷新控件的显示
ICONTROL_Release()         // 继承自 IBase，管理资源
ICONTROL_Reset()           // 重置控件为最初的状态
ICONTROL_SetActive()       // 设置控件是否处于活动状态
ICONTROL_SetProperties()   // 设置控件属性
ICONTROL_SetRect()         // 设置控件显示区域
```

在对话框内部，正是通过这些接口控制控件的。切换焦点可以通过设置控件是否激活来控制，传递事件的时候传递给当前活动的控件，还可以通过设置属性的 API 来设置控件的属性等等，IControl 提供给我们足够的能力去控制控件的行为。

现在我们还有两个问题没有搞清楚：一是 IControl 控件没有提供设置标题的方法，对话框如何设置控件的标题呢？二是对话框中的控件是如何进行焦点切换的，仅仅使用 IControl 提供的 API 够吗？

要回答这两个问题，需要从控件的内部讲起，因为它们的实现都与控件的处理方式密不可分。对于支持设置标题的控件，在内部的内部都会处理一个叫做 EVT_CTL_SET_TITLE 的控件内部事件，通过这个事件，我们可以把标题的在资源文件中的字符串 ID 以及资源文件名称指针分别通过 wParam 和 dwParam 传递给控件，控件在接收到这个事件的时候会使用这两个参数来获得标题。对话框设置控件标题的实现，正是基于这个事件。

通常我们使用 BREW 设备的四个方向键进行对话框的焦点切换，但是这里面有的时候存在冲突，例如一个菜单控件会处理上、下方向键的事件，如果在对话框中处理了，那么菜单控件将不能够正常上下滚动了。因此，在进行焦点切换的时候又增加了 EVT_CTL_TAB 事件。对话框仍然原封不动的将四个方向键的事件传递给对话框中的控件，然后由控件决定是否应该进行控件的切换。当控件不处理这四个事件认为应该进行焦点切换的时候，将会向外部发送 EVT_CTL_TAB 事件，wParam 为 0 表示逆序切换，为 1 表示顺序切换。在对话框中，如果收到了这个事件，将会执行焦点切换的动作。

了解了这些内容，我们就可以在我们的应用程序中构建多个控件的焦点切换的架构了，同时也为编写与 BREW 完全兼容的控件提供了参考。

11.6.3 使用对话框处理事件

对话框可以自动将事件传递给它包含的控件，因而这些控件的事件处理就不需要应用程序操心了。而且，对话框捕获事件是优先于应用程序的，BREW Shell 会优先将事件传递给对话框，而不是应用程序。与此同时，对话框还提供了一个 DLG_HANDLE_ALL_EVENTS 属性，使得指定的事件处理函数可以完全捕获传递给应用程序的事件（应用程序启动、停止、挂起、恢复事件除外），包括对话框中控件已经处理的事件。这样的方式给我们一个捕获全部 BREW 事件的机会，增加了开发 BREW 应用程序的灵活度。

要使用 BREW 对话框的这一特性，我们首先需要使用 IDIALOG_SetProperties 接口设置属性 DLG_HANDLE_ALL_EVENTS，然后调用 IDIALOG_SetEventHandler 接口指定处理事件的函数，这个函数必须为 PFNAEEEVENT 类型。这样，在这个 PFNAEEEVENT 类型的函数中我们将接收到全部对话框接收的事件。PFNAEEEVENT 的函数参数形式与应用程序的事件捕获函数是一样的：

```
typedef boolean (* PFNAEEEVENT)(void * pUser, AEEEvent evt, uint16 w, uint32 dw);
```

如果在我们调用了 IDIALOG_SetEventHandler 后，但是没有设置捕获全部事件的对话框属性 DLG_HANDLE_ALL_EVENTS，那么，这个回调函数中将只能接收到对话框事件（也就是 EVT_DIALOG_INIT、EVT_DIALOG_START、EVT_DIALOG_END 事件）。

11.6.4 特殊对话框之消息框和提示框

除了标准的 BREW 对话框之外，BREW 还提供了两种基于对话框的特殊形式：消息框和提示框。在应用程序中可以使用 ISHELL_MessageBox、ISHELL_MessageBoxText 以及 ISHELL_Prompt 接口直接创建一个提示框。这几个对话框的创建就是通过了 DialogInfo 指针进行的，属于在 BREW 内部对 ISHELL_CreateDialog 又进行了一层封装，同时设置了事件处理函数，这样就可以在捕获到按键事件的时候关闭对话框了。

ISHELL_Prompt 的功能要比 ISHELL_MessageBox 和 ISHELL_MessageBoxText 更加强大一些。通过 AEEPromptInfo 结构体，我们可以自定义提示框的外观，AEEPromptInfo 结构体的定义如下（可以参考 AEEShell.h 文件）：

```
typedef struct
{
    const char    *    pszRes;          // 与此提示关联的资源文件
    const AECHAR * pTitle;          // 提示框的标题文本（如果未指定 pszRes）
    const AECHAR * pText;           // 提示框提示信息（如果未指定 pszRes）
    uint16              wTitleID;     // 提示框标题文本在资源文件中的 ID
    uint16              wTextID;      // 提示框提示信息在资源文件中的 ID
    uint16              wDefBtn;      // 默认（或已选择按钮）的 ID
    const uint16 * pBtnIDs;          // 指向按钮 ID 数组的指针
    uint32              dwProps;      // 提示的属性
    AEEFont             fntTitle;     // 标题字体
    AEEFont             fntText;      // 文本字体
    uint32              dwTimeout;    // 提示的超时。 如果设置为 0（零），则提示不设超时。
} AEEPromptInfo;
```

在这里面需要特别说明的是按钮 ID 的数组指针 pBtnIDs，由于按钮的内容需要经过资源文件指定。因此，如果 pszRes 为空的话此成员将没有任何作用，在这个数组中放置的就是每个按钮文本对应的资源文件字符串 ID。在创建对话框的时候将这些 ID 与对话框软键（Soft Key）控件的每一个项目对应起来。相应的，可以触发这些按钮关闭对话框，并发送对应资源 ID 值相同的 EVT_COMMAND 事件。当应用程序收到这些按钮的命令事件后，可以在这个事件中进行指定的处理。

由于我们在关闭这些对话框之后需要返回应用程序中，那么将要面临重新刷新屏幕以便显示应用程序内容的问题（对话框不会记得在它创建之前显示的内容）。为了能够正确地显示应用程序的内容，提示框和消息框关闭的时候都会发送 EVT_DIALOG_END 事件到应用程序，应用程序在收到这个事件的时候需要执行刷新屏幕的任务，这样就可以正确地显示应用程序信息了。

11.6.5 对话框的优缺点

在 BREW 平台中，对话框既是一个神仙，也是一个魔鬼。这源于它双重的性格。使用对话框的好处是：

1、可以方便的管理控件，无需应用程序过多的介入，包括向控件传递事件。

2、在资源文件中可以指定对话框的属性，这样可以做到显示内容与程序无关，而只与资源文件相关。

3、对话框可以优先于应用程序捕获事件。

对话框的这些优点都减少了我们开发过程中的工作量，但是与此同时对话框也有着让人心烦的缺点。

首先，由于对话框采用的是全屏幕刷新的方式，因此如果我们需要在对话框的基础上增加自己的刷新请求（如增加提示信息等）将变的十分麻烦。要实现这种自定义刷新的功能，只有在对话框的 EVT_DIALOG_START 事件或 EVT_DIALOG_INIT 事件中采用 Post Event 的方式，异步的发送自定义刷新事件。毫无疑问的，这增加了代码执行的负担，降低了程序的执行效率。因此在那些 CPU 运算能力不高的平台上，会显得我们的应用程序运行很慢。与此同时这也限制了我们开发程序的自由，使得基于对话框开发自定义界面的应用程序变得麻烦。

其次，离开了资源文件的对话框的创建将变得十分的麻烦。使用 DialogInfo 指针的方式需要自己去构建结构体，而且还要小心翼翼的去设置这些数据，比较容易出现问题。

因此，带着这些问题的对话框在实际的 BREW 开发过程中使用的频率并不高，大多数的开发者宁愿构建自己的对话框系统也不使用 BREW 系统提供的对话框机制。

11.7 探秘 BREW 内部事件处理流程

到现在为止，我们所知道 BREW 中可以捕获事件的有三种：控件、对话框和应用程序。其中控件属于被动的事件接收者，BREW Shell 与其没有任何直接关联。剩下的对话框可以直接接收 BREW Shell 的事件，并且还优先于应用程序接收事件。我想应该到了揭开 BREW 内部事件传递流程神秘面纱的时候了，请看图 11.6。

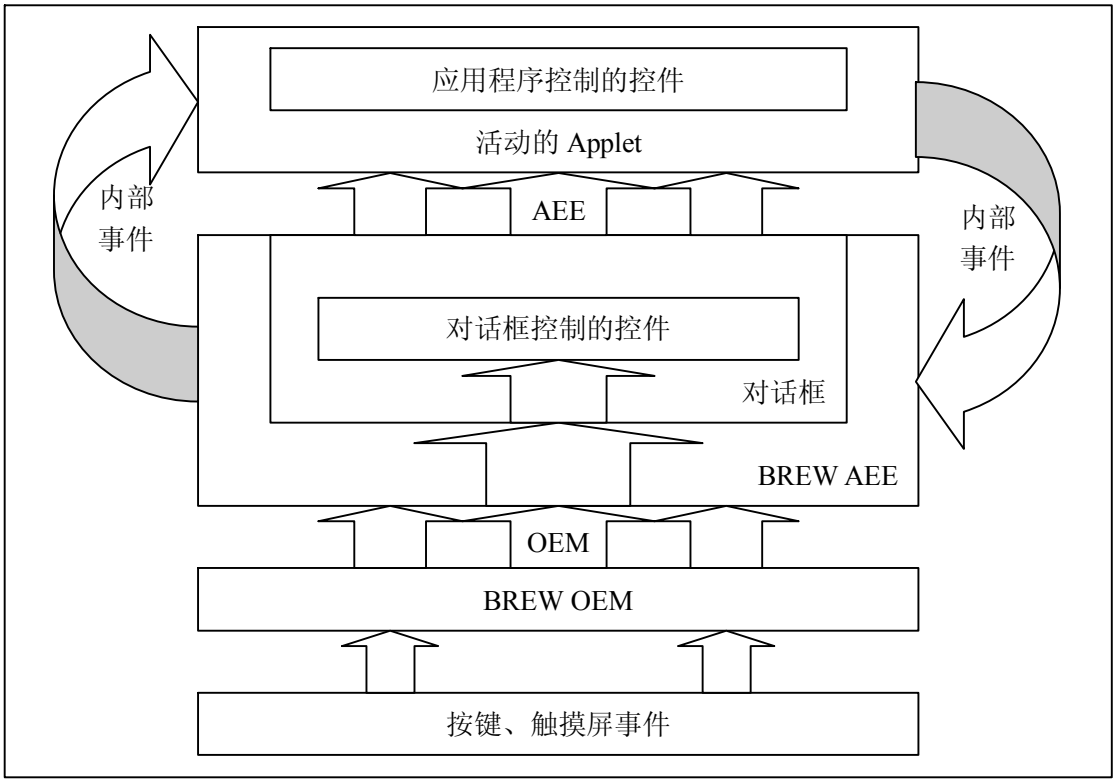

图 11.6 BREW 事件处理流程

我想对于这幅图，我不必讲的太多，这是整个 BREW Shell 内部的事件处理过程。其中应用程序处理事件的函数就是应用程序的 IAPPLET_HandleEvent 函数，对话框处理事件的函数就是对话框的 IDIALOG_HandleEvent 以及通过 IDIALOG_SetEventHandler 接口所指定的事件捕获函数。控件则使用 ICONTROL_HandleEvent 接口来捕获事件。就是这样循环不息的事件传递，构成了 BREW 的事件驱动的机制，理解了它，也就掌握了 BREW 事件处理的神髓，开发应用程序就会更加得心应手。

11.8 小结

在本章中，我们首先介绍了 BREW Shell 的基本功能，初步让您了解了 BREW Shell。紧接着介绍了 BREW Shell 通用的启动过程，然后带您进入了 BREW 应用程序的内部，深入地剖析了 AEEModGen.c 和 AEEAppGen.c 两个文件中的函数在应用程序中的作用，从而了解了 BREW 从 AEEMod_Load 函数作为入口开始执行的方式，更进一步的引出了 BREW 应用程序的本质就是实现 IApplet 接口的关键点。 接下来我们讲解了关于多线程的相关知识，以及如何在 BREW 中模拟多线程。从中我们知道了为什么 BREW 不能够支持真正的多线程。后续依次讲解了系统中的定时器实现原理、对话框的实现和特点。在讲解定时器的时候，我们从硬件的时钟中断讲起，一直到 BREW 定时器的实现，完整地展示了一个系统中时钟的实现方式，最后还涉及了系统如何通过降低 CPU 时钟频率，从而实现节能的问题。在本章的最后，我们揭开了 BREW 内部事件的处理流程，展示了一个完整的 BREW 事件处理模型给您。

思考题

1、系统的时钟中断都有什么作用？对系统的性能影响如何？
2、动态应用程序入口函数 AEEMod_Load 函数名可以替换吗？为什么？
3、为什么助手函数不需要创建接口指针就可以直接使用？

第十二章 扩展 BREW 接口

讨厌使用那些不好用的接口吗？是的，很讨厌。

希望为自己的系列应用程序提供独有的接口吗？是的，需要。

需要将几个接口封装成一个接口吗？是的，需要。

……

不管基于怎样的理由，我们想必都想拥有自己的接口吧。BREW 可以让我们如愿以偿，因为 BREW 的接口是可扩展的。就像一个 BREW 应用一样，如果您是一位负责 BREW 移植工作的工程师，那么将可以在移植 BREW 的时候开发静态的扩展接口；如果您是一位 BREW 应用程序的开发工程师，那么您也可以选择使用动态的扩展接口。

动态和静态扩展接口之间没有明显的区别，只不过是由不同的人在不同的环境下完成的而已。它们都需要完成接口的定义、声明、实现接口的内部数据结构和实现接口函数。而且接口和 BREW Applet 之间也没有什么太大的区别，包括实现方式和运行原理，只不过接口是直接继承自 IBase，而 Applet 是经过 IApplet 接口之后间接的继承自 IBase。

动态和静态扩展接口之间最大的区别在于入口方式不同。静态接口是由 BREW 的 OEM 层直接将入口函数链接进 BREW 的接口列表的，而且接口的 Class ID 可以采取 OEM 层指定或像动态应用程序一样到高通公司申请的两种方式。动态应用则需要使用模块的 MIF 文件来声明接口为外部扩展接口（还记得 MIF 文件编辑器中的 Extensions 选项卡吗？不记得就到第二部分去看一看吧）。而且，如果需要公开发布我们的接口，那么动态扩展接口只能采用从高通公司获得 Class ID 一种方式了（与静态扩展接口不一样，BREW 的移植者可以在设备的说明文档中为应用程序的开发者公布他们扩展接口的 Class ID）。

在这一章里，我们将主要针对动态扩展接口进行讲解。通过这一章您将能够看到在使用 BREW 进行接口扩展时所需要的程序实现，以及为何这样实现。在 BREW 接口探秘一章中我们已经详细的分析了 BREW 接口的实现方式，因此本章将侧重于实际创建一个扩展接口时需要进行的工作。

12.1 理解模块的生命周期

万物皆有生有灭，软件模块也同样逃脱不了。需要使用这个模块的时候就会加载它，并且让它执行相关的功能（诞生到存活），功能完成后就要释放掉这些模块（消亡）。这样的一个过程就是软件模块的生命过程，也就是这个模块的生命周期。

前面我们已经讲过，每一个模块都有 AddRef 和 Release 两个模块资源管理的接口，当增加当前模块的引用计数的时候，需要调用 AddRef 方法，释放一个模块的引用的时候，调用 Release 方法。当一个模块使用 ISHELL_StartApplet 或者 ISHELL_CreateInstance 的时候，它的实例被创建并且存活于系统的内存之中。由于 BREW 模块使用指针的方式来引用模块的实例，因此在这个模块存活的过程中可使用多个指针来引用这个模块的实例。此时的引用需要调用模块的 AddRef 方法。在一个引用指针使用完毕后将调用模块的 Release 方法，如果当前释放的已经是这个模块的最后一个引用，那么将释放这个模块所占用的全部系统资源，这个模块的生命周期宣告完结。

如果我们希望一个模块能够在 BREW 平台的运行期间内一直存在，或者叫做 BREW 内存驻留模块，那么，我们可以有两种选择：一是只需要利用这个模块生命周期的规则，调用一次模块的 AddRef 方法，而不再调用模块的 Release 方法，这仿佛模拟了一次人为的内存

泄露；二是在开发接口的内部实现上做文章，例如将 Release 方法无效化（使 Release 方法为空，什么都不做）。当然，这么做需要有一个前提，那就是这个模块中所分配的内存都是使用 sys_malloc 而不是 BREW 的助手函数 MALLOC。这对于动态接口开发者来说是一个噩耗，因为这将导致无法实现动态的内存驻留接口。

为什么？这需要从 BREW 的内存管理方式讲起。在 BREW 中使用 MALLOC 或者 IHeap 接口分配的内存都是与应用程序相关的，目的是为了避免内存泄露而影响整个系统的运行。每一段分配的内存与应用程序相关之后，可以在应用程序退出的时候将所有相关的内存释放掉，同时也会给应用程序管理器发送一个“模块未释放全部分配的内存”的通知。这样的内存管理方式也同时禁止跨应用程序的内存调用，也就是在 App1 种分配的内存，不能传递给 App2 使用，否则将会产生严重的错误。

看来这样的内存驻留接口只能留给静态接口的开发者了。如果我们现在正希望开发一个这样的静态接口，那么可以采用第二种方法来实现这种内存驻留的模块。只不过在当前 BREW 的开发环境内，没有为我们提供更大的发挥空间，使用内存驻留模块也不能实现一些高级的功能。这其中一个主要的问题就是，在 BREW 平台和应用程序之间我们没有任何一种方法可以在不影响应用的情况下捕获事件。

在这里需要知道的是一个模块的生命周期管理是模块实现中最为基本也最为重要的部分。体现在 BREW 中就是入口函数、AddRef 方法和 Release 方法的实现。

12.2 声明接口

既然要扩展一个接口，那么声明这个接口就是必不可少的了。BREW 接口声明的本质是将虚拟函数表中的定义使用 C 语言宏定义的形式让它看起来更加标准化。在前面的章节中我们已经查看过了 AEEGETPVTBL 宏的定义，它展开之后的形式如下：

```
#define AEEGETPVTBL(p,iname)    (*((AEEVTBL(iname) **)((void *)p)))
```

实际上它就是一个指针类型的强制转换，如果您在程序中直接使用后面的部分也是没有任何问题的，只不过这样会看起来有点让人讨厌而已。

还记得我们在前面介绍的应用程序框架吗，这是我们在“BREW 的事件处理”一章中通过 FileExplorer 介绍的。在这一章扩展接口中，我们将使用这个例子中的 IStateMachine 构建一个扩展接口。关于这些定义接口的功能我们就不再一一的介绍了，我们更加关注的是如何将这个事例程序中的代码变成扩展接口所需要的样式。修改之后的状态机接口存储在 Test10\IStateMachine 文件夹中，为此我们也特意构筑一个应用程序 IStateMachine。这个接口的声明如下（摘自 IStateMachine.h）：

```
typedef struct IStateMachine IStateMachine;

// 状态处理函数返回给状态处理主函数的值类型
typedef enum _NextFSMAction{
    NFSMACTION_WAIT,
    NFSMACTION_CONTINUE
} NextFSMAction;

typedef NextFSMAction (*PFNSTATE)(IStateMachine *po, void *pUser);

// 窗口事件
```

```
enum{
    EVT_WND_OPEN = (EVT_APP_LAST_EVENT+1),
    EVT_WND_CLOSE
};

// 窗口返回值
enum{
    WNDRET_CREATE = 0,
    WNDRET_OK,
    WNDRET_CANCELED,
    WNDRET_YES,
    WNDRET_NO,
    WNDRET_USER1,
    WNDRET_USER2,
    WNDRET_USER3,
    WNDRET_USER4,
    WNDRET_USER5,
    WNDRET_MAX
};

// 状态机接口定义
#define INHERIT_IStateMachine(IStateMachine)                                \
    INHERIT_IApplet(IStateMachine);                                         \
    void (*Start)          (IStateMachine *po,                              \
                            PFNSTATE       pfnState,                        \
                            void           *pUser);                         \
    void (*Stop)           (IStateMachine *po);                             \
    void (*Suspend)        (IStateMachine *po);                             \
    void (*Resume)         (IStateMachine *po);                             \
    void*(*GetState)       (IStateMachine *po);                             \
    void (*MoveTo)         (IStateMachine *po,                              \
                            PFNSTATE       pfnState,                        \
                            void           *pUser);                         \
    void (*OpenWnd)        (IStateMachine *po,                              \
                            PFNAEEEVENT    HandleEvent,                     \
                            void           *pUser);                         \
    void (*CloseWnd)       (IStateMachine *po,                              \
                            int            eWndRet,                         \
                            uint16         wParam,                          \
                            uint32         dwParam);                        \
    int    (*GetWndRet)    (IStateMachine *po,                              \
                            uint16         *pwParam,                        \
                            uint32         *pdwParam)
```

```
AEEINTERFACE(IStateMachine) {
    INHERIT_IStateMachine(IStateMachine);
};

#define ISTATEMACHINE_AddRef(p)         AEEGETPVTBL(p, IStateMachine)->AddRef(p)
#define ISTATEMACHINE_Release(p)        AEEGETPVTBL(p, IStateMachine)->Release(p)
#define ISTATEMACHINE_HandleEvent(p,e,w,dw)\
                                AEEGETPVTBL(p, IStateMachine)->HandleEvent(p,e,w,dw)
#define ISTATEMACHINE_Start(p,s,pu) AEEGETPVTBL(p, IStateMachine)->Start(p,s,pu)
#define ISTATEMACHINE_Stop(p)           AEEGETPVTBL(p, IStateMachine)->Stop(p)
#define ISTATEMACHINE_Suspend(p)        AEEGETPVTBL(p, IStateMachine)->Suspend(p)
#define ISTATEMACHINE_Resume(p)         AEEGETPVTBL(p, IStateMachine)->Resume(p)
#define ISTATEMACHINE_GetState(p)      AEEGETPVTBL(p, IStateMachine)->GetState(p)
#define ISTATEMACHINE_MoveTo(p,s,pu)\
                                AEEGETPVTBL(p, IStateMachine)->MoveTo(p,s,pu)
#define ISTATEMACHINE_OpenWnd(p,h,pu)\
                                AEEGETPVTBL(p, IStateMachine)->OpenWnd(p,h,pu)
#define ISTATEMACHINE_CloseWnd(p,e,w,dw)\
                                AEEGETPVTBL(p, IStateMachine)->CloseWnd(p,e,w,dw)
#define ISTATEMACHINE_GetWndRet(p,pw,pdw)\
                                AEEGETPVTBL(p, IStateMachine)->GetWndRet(p,pw,pdw)
```

从这些代码中我们可以看到，我们首先定义了一个 IStateMachine 的类型，而实际上我们在整个头文件中再也找不到一个 IStateMahine 的结构体定义了，因此 IStateMachine 纯粹属于为了定义而定义的类型了，我们就将它看成一个“虚”类型吧。定义这个类型的目的是为了更好的进行数据封装，并没有多大的实际意义，我们可以将之忽略。在这里特别的提到它，目的是为了避免各位读者在分析程序的时候遇到困难。

接下来我们使用 AEEINTERFACE(IStateMachine)来定义了接口函数列表的结构体，这些函数统统定义在宏 INHERIT_IStateMachine(IStateMachine)中。最后，我们通过 AEEGETPVTBL 来将这些接口暴露出来。关于这些宏定义我们在探秘 BREW 接口一章中已经有了详细的分析，在这里就不再赘述了。

12.3 实现接口内部数据结构

上一节当中已经定义了相应的接口，那些接口的结构体只不过是暴露给接口使用者使用的而已。而本着 BREW 的 COM 精神，内部数据是需要隐藏在接口内部的，在本例中这个内部数据的结构体就是 CStateMachine 结构体（摘自 IStateMachine.c）：

```
typedef struct _CStateMachine{
    AEEVTBL(IStateMachine) *pvt;
    uint32              m_nRefs;
    IModule             *m_pMod;

    // IShell interface
    IShell              *m_pShell;
```

```
    PFNSTATE            m_pfnState;       // 状态处理函数
    void *              m_pUserState;     // 用户数据
    PFNAEEEVENT         m_pfnEvent;       // 事件处理函数
    void *              m_pUserEvent;     // 用户数据
    int                 m_eWndRet;        // Windows 返回值
    uint16              m_wWndParam;      // Windows 返回值 word 参数
    uint32              m_dwWndParam;     // Windows 返回值 dword 参数

    AEECLSID            m_dwAppClsID;     // 当前使用此接口的应用程序 Class ID
    boolean             m_bSuspending;    // 当前状态机是否被挂起
}CStateMachine;
```

结构体中的第一个变量是 AEEVTBL(IStateMachine) *pvt，这个变量就是暴露给外部使用的接口虚拟函数表（VTBL）指针。这个变量一定要在结构体中的第一个位置，就像是动态应用程序中的 IApplet a 一定要在应用程序结构体成员的第一位一样。紧接着的成员变量是在实现接口函数的时候所使用的，这些变量没有次序上的要求。

12.4 接口初始化

接口的初始化是实现接口最为关键的一步，在这一步中将完成分配内存、初始化虚拟函数表、初始化成员变量等内容。接下来我们将沿着应用程序的入口，逐渐的分析接口初始化的过程。

我们知道，对于一个模块来说，模块的入口函数时 AEEMod_Load 函数，这个函数我们可以从 AEEModGen.c 文件中找到。这个文件中实现了一个叫做 IModule 的接口，这个接口是与模块的 MIF 文件相对应的，每一个 MIF 文件将对应一个这样的入口函数，函数的实现如下（摘自 AEEModGen.c 文件）：

```
#ifdef AEE_LOAD_DLL
__declspec(dllexport) int AEEMod_Load(IShell *pIShell, void *ph, IModule **ppMod)
#else
#if defined(BREW_MODULE) || defined(FLAT_BREW)
extern int module_main(IShell *pIShell, void *ph, IModule **ppMod);
int module_main(IShell *pIShell, void *ph, IModule **ppMod)
#else
int AEEMod_Load(IShell *pIShell, void *ph, IModule **ppMod)
#endif
#endif
{
    // Invoke helper function to do the actual loading.
    return AEEStaticMod_New(sizeof(AEEMod),pIShell,ph,ppMod,NULL,NULL);
}
```

在这个函数中，调用了 AEEStaticMod_New 函数来创建一个 IModule 的实例，IModule 的接口中包含的最主要的一个接口函数是 IMODULE_CreateInstance。BREW 在获得 IModule 接口实例之后，在创建模块中的接口或应用的实例的时候，调用的就是这个接口来实现的。关于这一点我们在 Shell 内幕一章已经有所介绍，尚未熟悉的读者可以回过头看看。对应的

IMODULE_CreateInstance 接口的实现函数如下（摘自 AEEModGen.c 文件）：

```
static int AEEMod_CreateInstance(IModule *pIModule,IShell *pIShell,
                                 AEECLSID ClsId,void **ppObj)
{
    AEEMod      *pme = (AEEMod *)pIModule;
    int             nErr = EFAILED;

    // For a dynamic module, they must supply the AEEClsCreateInstance()
    //     function. Hence, invoke it. For a static app, they will have
    //     registered the create Instance function. Invoke it.
    if (pme->pfnModCrInst) {
        nErr = pme->pfnModCrInst(ClsId, pIShell, pIModule, ppObj);
#if !defined(AEE_STATIC)
    } else {
        nErr = AEEClsCreateInstance(ClsId, pIShell, pIModule, ppObj);
#endif
    }

    return nErr;
}
```

从中我们可以看到，这里面调用到了我们所熟知的 AEEClsCreateInstance 函数。通常，我们使用 BREW 向导在 Visual Studio 中建立应用程序的时候，每一个应用程序都会有一个这样的函数。而实际上，扩展接口也使用了这样的一个函数（摘自 IStateMachine.c）：

```
int AEEClsCreateInstance(AEECLSID ClsId, IShell *pIShell,
                         IModule *po,       void    **ppObj)
{
    *ppObj = NULL;

    if( ClsId == AEECLSID_STATEMACHINE ){
        if(SUCCESS == AppStateMachine_New(pIShell, po, ppObj)){
            // 数据初始化成功，返回 SUCCESS
            return(SUCCESS);
        }
    }

    return(EFAILED);
}// End AEEClsCreateInstance
```

从函数的实现当中我们可以看出，与实现应用程序不同的是，我们现在不是调用 AEEApplet_New 函数了，而是调用 AppStateMachine_New 函数。因此，我们的 IStateMachine 的工程当中只包含了 AEEModGen.c 文件，而没有包含 AEEAppGen.c 文件。这个 AppStateMachine_New 函数函数就是我们整个应用程序的初始化函数了。接下来我们将来看一看这个函数内部实现都作了些什么。

12.4.1 分配内存

为接口的实现和内部变量分配内存是初始化函数最先完成的工作之一。请看下面的代码（摘自 IStateMachine.c）：

```
CStateMachine          *pMe = NULL;
VTBL(IStateMachine) *appFuncs;

if(!pShell || !ppobj || !po)
{
    return(EBADPARM);
}

*ppobj = NULL;

// 为状态机结构体和函数表分配空间
pMe = (CStateMachine*)MALLOC(sizeof(CStateMachine) + sizeof(VTBL(IStateMachine)));
if (NULL == pMe)
{
    return ENOMEMORY;
}
```

首先我们定义了扩展接口内部结构体 CStateMachine 的指针变量 pMe，以及虚拟函数表指针变量 appFuncs。然后判断了参数的有效性，接着就是分配空间了。我们看到为结构体分配的空间大小是 sizeof(CStateMachine) + sizeof(VTBL(IStateMachine))，sizeof(CStateMachine)是接口内部结构体的尺寸，这个没有任何疑义了，关键是 sizeof(VTBL(IStateMachine))是做什么的？这个东西是接口的虚拟函数表的尺寸。由于内部结构体中定义的只是虚拟函数表的指针，因此在给结构体分配空间的时候并没有包含虚拟函数表的大小，这里就直接为其分配了空间。

12.4.2 初始化虚拟函数表

上面已经为虚拟函数表分配了空间，那么接下来要做的事情就是初始化虚拟函数表（摘自 IStateMachine.c）：

```
appFuncs = (VTBL(IStateMachine) *)((byte *)pMe + sizeof(CStateMachine));

// 初始化函数表的内容
appFuncs->AddRef          = IStateMachine_AddRef;
appFuncs->Release         = IStateMachine_Release;
appFuncs->HandleEvent     = IStateMachine_HandleEvent;
appFuncs->Start           = IStateMachine_Start;
appFuncs->Stop            = IStateMachine_Stop;
appFuncs->Suspend         = IStateMachine_Suspend;
```

```
    appFuncs->Resume            = IStateMachine_Resume;
    appFuncs->GetState          = IStateMachine_GetState;
    appFuncs->MoveTo            = IStateMachine_MoveTo;
    appFuncs->OpenWnd           = IStateMachine_OpenWnd;
    appFuncs->CloseWnd          = IStateMachine_CloseWnd;
    appFuncs->GetWndRet         = IStateMachine_GetWndRet;

    AEEINITVTBL(pMe, IStateMachine, *appFuncs);     // Initialize the VTBL
```

虚拟函数表的空间紧接在内部结构体之后，因此 appFuncs 的值使用了第一行作示的方法。紧接着为虚拟函数表中的函数指针赋予了相应的函数地址。最后一句话是调用了 AEEINITVTBL 将 pvt 的值等于了 appFuncs。这样虚拟函数表就算初始化完成了，此时的接口函数已经开始生效了。

12.4.3 初始化成员变量

初始化的最后一步就是初始化其中的成员变量。从本质上来讲，初始化虚拟函数表也算作初始化成员变量的一部分，也就相当于在初始化 pvt 变量，不过，由于这个 pvt 变量太特殊了，与其他的成员变量还有本质的不同。下面是初始化成员变量的代码（摘自 IStateMachine.c）：

```
    // 增加 IShell 指针的引用计数
    ISHELL_AddRef(pShell);
    IMODULE_AddRef(po);

    // 初始化状态机内部数据
    pMe->m_pShell        = pShell;
    pMe->m_pMod          = po;
    pMe->m_dwAppClsID = ISHELL_ActiveApplet(pMe->m_pShell);
    pMe->m_nRefs         = 1;

    *ppobj = (IStateMachine*)pMe;
```

这里面比较特殊的是要保存一个 IShell 和 IModule 指针，并且增加这两个指针的引用计数，如果不这么做的话将无法使用 IShell 指针，而对于 IModule 来说则产生更加严重的后果。因为 BREW 创建完 IModule 接口之后会在使用完毕之后释放掉，而我们的接口实例还在，那么将导致这个模块无效，也就是我们的接口不能使用了。在模拟器上，这个现象将表现为一个奇怪的内存访问错误，有兴趣的读者可以试一下。

12.5 实现接口函数

接口初始化完毕之后，就是实现接口的函数了。在我们的例子中，由于我们采用了与 FileExplorer 应用程序相同的功能，因此就不一一介绍每个接口的实现了。重点的来看看每一个接口都需要使用的 AddRef 和 Release 接口的实现（摘自 IStateMachine.c）：

```
/*=============================================================================
函数       ：IStateMachine_AddRef
```

```
描述      ：增加状态机实例的引用计数
参数      ：po[in]  指向状态机实例的指针
返回值    ：返回增加之后的引用计数
=============================================================================*/
static uint32    IStateMachine_AddRef(IStateMachine * po)
{
    return(++((CStateMachine*)po)->m_nRefs);
}// End IStateMachine_AddRef

/*=============================================================================
函数      ：IStateMachine_Release
描述      ：释放当前状态机实例，在此函数中将判断引用计数的个数，如果为 0 则释放
参数      ：po[in]  指向状态机实例的指针
返回值    ：返回减少之后的引用计数
=============================================================================*/
static uint32    IStateMachine_Release(IStateMachine * po)
{
    CStateMachine *pMe = (CStateMachine*)po;

    if(pMe->m_nRefs){
        if(--pMe->m_nRefs)
        {
            return(pMe->m_nRefs);
        }

        ISHELL_Release(pMe->m_pShell);
        IMODULE_Release(pMe->m_pMod);

        // Free object
        FREE(pMe);
    }

    return(0);
}// End IStateMachine_Release
```

IStateMachine_AddRef 函数只是简单的将引用计数加 1，然后返回相应的值就完成了。IStateMachine_Release 中则先减少引用计数，如果引用计数变成了 0，则释放使用的接口和初始化时分配的程序空间。所有的 BREW 接口都是采用这种方式来进行引用计数管理的，是每一个接口必需实现的。

12.6 测试扩展接口

前面我们已经扩展的一个动态的 IStateMachine 接口，那么我们将如何使用这个接口呢？这就是本节要解决的问题。要想测试这个新扩展的动态接口，需要完成两个主要的部分，其一是创建扩展接口所在的 MIF 文件，其二是建立测试应用程序。

12.6.1 MIF 文件

动态的扩展应用也需要通过 MIF 文件来提供输出功能。在此之前我们需要做的就是为这个接口分配一个 Class ID，由于我们只是在本机进行测试，因此我们为这个接口创建了一个如下的 bid 文件：

```
#ifndef STATEMACHINE_BID
#define STATEMACHINE_BID

#define AEECLSID_STATEMACHINE              0x99999998

#endif //STATEMACHINE_BID
```

AEECLSID_STATEMACHINE 定义已经在前面的 AEEClsCreateInstance 函数中见识过了，就是通过对这个的条件判断才获得 IStateMachine 接口初始化函数的。同理，如果我们在这个函数中增加多个 Class ID 的话，我们就可以在一个模块中支持多个扩展接口了。

选定了 Class ID 后，我们需要将这个 Class ID 的值放在 MIF 文件的 Extensions 选项卡中：

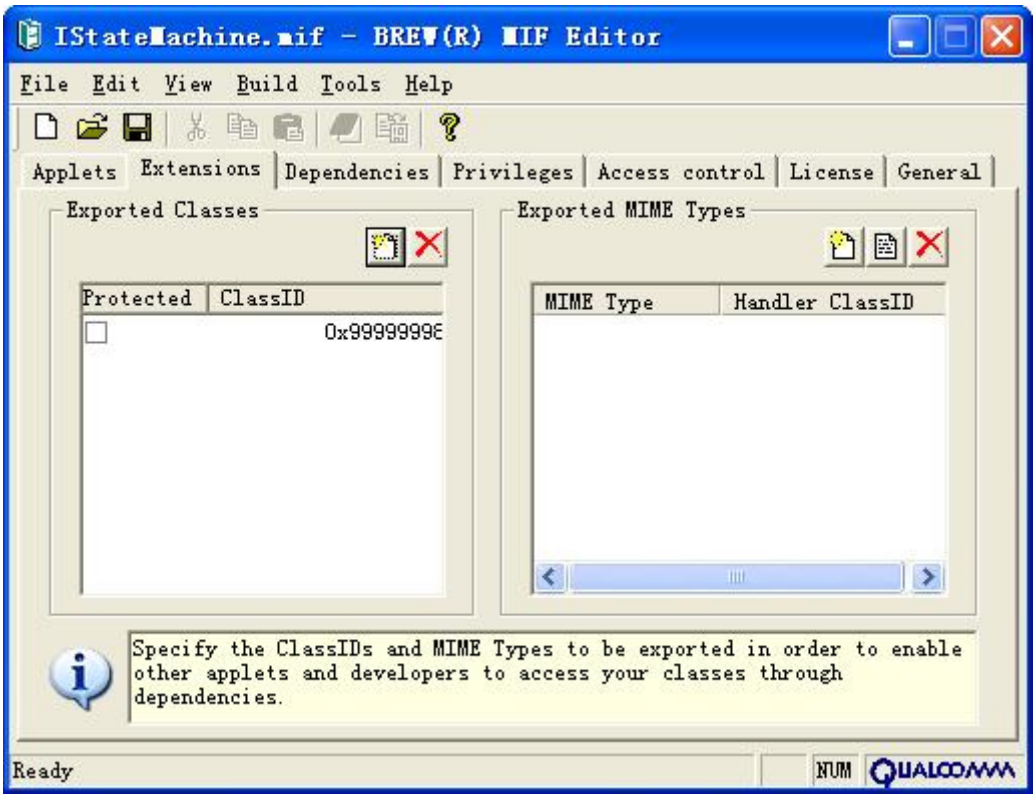

图 16.1 添加扩展接口的 Class ID 到 MIF

12.6.2 建立测试应用程序

为了测试这个扩展接口，我们在 FileExplorer 的基础之上，重新建立了 StateMachineTest 应用程序。应用程序与 FileExplorer 之间没有进行大的修改。测试需要注意的是：

1、在 StateMachine.mif 文件中，需要在 Dependencies 选项卡中添加

AEECLSID_STATEMACHINE 的 Class ID。

2、需要将扩接口的 IStateMachine.mif 和测试应用程序的 StateMachineTest.mif 文件放在同一个路径内，以便于模拟器可以找得到这个扩展接口的 mif。同样的扩展接口的文件夹和应用程序的文件夹也要集中在一起。

运行我们的 StateMachineTest 应用程序，可以看到相应的效果，我们就不再介绍了，请各位读者自行在事例应用程序中进行测试。

12.7 在 BREW 中使用 C++

C++、面向对象，多么时髦而又让人向往的名字啊，不需要其他理由，单单这两个名字就足以说明我们为什么想在 BREW 中使用 C++了。不过，不过要过于乐观，在 BREW 上使用 C++不但要面临很多问题，而且还要受到很多的限制。这主要是因为 BREW 的实现原理与 C++上的不同，以及 BREW 使用 C 语言实现等原因。在这一节当中，我们将逐一的剖析这些问题。本节中的完整示例在 IStateMachineCPP 中。

12.7.1 面临的问题

使用 C++开发 BREW 接口的第一个问题是如何声明这个接口。这是因为要实现一个可以给任何一个 BREW 应用使用的公开接口，必须保证这个接口的定义能够被 C 语言接受。因为我们知道，BREW 使用 C 语言定义的接口。不幸的是，C 语言不能接受 C++类方式的接口，所以我们不能使用 C++的类方式来声明和定义我们的接口。

使用 C++扩展 BREW 接口的另一个问题是如何实现接口的虚拟函数表（VTBL）。这源于 C 和 C++在函数调用约定上的不同，C 语言采用 C 函数调用约定，而 C++则采用了 thiscall 调用约定。这两种约定采用不同的堆栈使用方式和参数的传递顺序，更要命的是 C++中会给类成员函数“偷偷”传递一个叫做 this 的指针。也就是说，C++完全的改变了函数的执行方式。而 BREW 使用 C 语言方式定义的接口，因此，这直接导致了在 BREW 和 C++之间进行接口映射的困难。

new 和 delete 的功能缺失是我们所面临的另一个问题。BREW 应用程序中不允许链接 C 或者 C++的库函数，因为这些库函数可能使用了不同于 BREW 的内存和堆栈管理方式，这将导致程序运行时不可预知的错误，最为典型的就是涉及了内存操作的 new 和 delete 函数。这两个函数在使用 C++开发的过程中是被禁止使用了，因为他们使用的是 C++库函数中的内存管理方式，而这些方式将不能和 BREW 的兼容，明显的错误就是在使用这些功能的应用程序中会产生一个链接错误（当然，这仅仅发生在生成 BREW 设备执行程序的时候）。

12.7.2 接口的定义和实现

既然由于 C 不能兼容 C++导致接口的定义不能采用 C++的类方式来定义，那么，我们可以反过来想，C++是兼容 C 语言的，因此我们保持使用 C 语言来定义接口。这也就是说，我们仍然使用 C 语言来定义和声明接口。从前面章节中对 BREW 接口定义的方式我们可以发现，BREW 接口的定义实际上是定义一个包含多个函数指针的虚拟函数表的定义。因此，只要我们能够在扩展接口的内部数据结构中，同样的实现这样一个虚拟函数表就可以了。我们面临的第一个问题以戏剧性的方式解决了，那就是什么都不变！

现在我们面临的主要问题是如何解决由于函数调用约定的不同而遇见的虚拟函数表（VTBL）填充问题。既然接口的使用方式不能改变，那么我们还是只能从 C++的角度来看看怎么解决这个问题。幸运的是，C++给我们提供了这样的一种方式，那就是使用 extern “C” 来在 C++文件中声明或者使用 static 方式生明一个类成员函数，这两种方式都将使用 C 约定调用函数，通过这样方式声明的函数就可以作为 BREW 接口虚拟函数表的填充函数了。由于使用 extern “C”方式声明的函数与 C++类本身没有任何的相关性，因此我们决定采用 static 声明成员函数的方式来实现接口函数（Test11\IStateMachineCPP 应用下的 IStateMachine.cpp 文件）：

```
class CStateMachine
{
private:
    AEEVTBL(IStateMachine) *pvt;
    uint32              m_nRefs;
    IModule             *m_pMod;

    // IShell interface
    IShell              *m_pShell;

    PFNSTATE            m_pfnState;       // 状态处理函数
    void *              m_pUserState;     // 用户数据
    PFNAEEEVENT         m_pfnEvent;       // 事件处理函数
    void *              m_pUserEvent;     // 用户数据
    int                 m_eWndRet;        // Windows 返回值
    uint16              m_wWndParam;      // Windows 返回值 word 参数
    uint32              m_dwWndParam;     // Windows 返回值 dword 参数

    AEECLSID            m_dwAppClsID;     // 当前使用此接口的应用程序 Class ID
    boolean             m_bSuspending;    // 当前状态机是否被挂起
public:
    static int New(IShell *pShell, IModule *po, void** ppobj);
    static uint32   AddRef(void * po);
    static uint32   Release(void * po);
    static boolean HandleEvent(void *po, AEEEvent eCode, uint16 wParam, uint32 dwParam);
    static void Start(void *po, PFNSTATE pfnState, void *pUser);
    static void Stop(void *po);
    static void Suspend(void *po);
    static void Resume(void *po);
    static void*GetState(void *po);
    static void MoveTo(void *po, PFNSTATE pfnState, void *pUser);
    static void OpenWnd(void *po, PFNAEEEVENT HandleEvent, void *pUser);
    static void CloseWnd(void *po, int eWndRet, uint16 wParam, uint32 dwParam);
    static int   GetWndRet(void *po, uint16 *pwParam, uint32 *pdwParam);
};
```

与 C 语言版本实现相的是，pvt 也必须是类的第一个成员变量，这样才能兼容 BREW 接

口定义和引用的方式。

现在我们又面临了另一个问题，那就是在这些 static 成员函数里面实现 BREW 接口仍然不能够体现出 C++语言的面向对象和封装等特性，不能充分的发挥出 C++语言本身的优势。解决这个问题的方法是声明这些函数的 C++“内部版本”：

```
private:
    int OnNew(IShell *pShell, IModule *po);
    uint32 OnAddRef(void);
    uint32 OnRelease(void);
    boolean OnHandleEvent(AEEEvent eCode, uint16 wParam, uint32 dwParam);
    void OnStart(PFNSTATE pfnState, void *pUser);
    void OnStop(void);
    void OnSuspend(void);
    void OnResume(void);
    void* OnGetState(void);
    void OnMoveTo(PFNSTATE pfnState, void *pUser);
    void OnOpenWnd(PFNAEEEVENT HandleEvent, void *pUser);
    void OnCloseWnd(int eWndRet,uint16 wParam, uint32 dwParam);
    int   OnGetWndRet(uint16 *pwParam, uint32 *pdwParam);
    void OnRunFSM(void);
```

static 方式声明的接口函数通过调用内部对应的类函数而实现相应的功能。当然，这种实现方式的代价是以增加了一个调用函数的运行时间换来的，如果从 C++开发效率上来看还是值得的。由此看来程序开发也是一种“妥协的艺术”。我们看看 Start 接口的实现就能明白这一切了：

```
void CStateMachine::Start(void        *po,
                          PFNSTATE    pfnState,
                          void        *pUser)
{
    CStateMachine *pMe = (CStateMachine*)po;
    pMe->OnStart(pfnState, pUser);
}// End IStateMachine_Start

void CStateMachine::OnStart(PFNSTATE pfnState, void *pUser)
{
    m_pfnState        = pfnState;
    m_pUserState      = pUser;

    if(m_pfnState){
        OnRunFSM();
    }
}// End OnStart
```

定义和实现了这些接口函数之后，我们的另一个问题就是扩展接口的入口函数和类初始化函数。其中 AEEClsCreateInstance 函数需要使用 extern “C”来说明，因为这个函数将给在 AEEModGen.c 中的函数调用，而由于 C++编译器只对类内部的函数执行 thiscall 调用，因此这个函数与 C 中的并没有任何的不同，只不过修改了调用了 New 函数名称：

```
int AEEClsCreateInstance(AEECLSID ClsId, IShell *pIShell,
                         IModule *po,       void    **ppObj)
{
    *ppObj = NULL;

    if( ClsId == AEECLSID_STATEMACHINECPP ){
        if(SUCCESS == CStateMachine::New(pIShell, po, ppObj)){
            // 数据初始化成功，返回 SUCCESS
            return(SUCCESS);
        }
    }

    return(EFAILED);
}// End AEEClsCreateInstance
```

CStateMachine::New 的实现如下：

```
int CStateMachine::New(IShell *pShell, IModule *po, void** ppobj)
{
    CStateMachine           *pMe = NULL;

    if(!pShell || !ppobj || !po)
    {
        return(EBADPARM);
    }

    *ppobj = NULL;

    // 为状态机结构体和函数表分配空间
    pMe          =          (CStateMachine*)MALLOC(sizeof(CStateMachine)          +
sizeof(VTBL(IStateMachine)));
    if (NULL == pMe)
    {
         return ENOMEMORY;
    }

    if(SUCCESS != pMe->OnNew(pShell, po)){
        return EFAILED;
    }

    *ppobj = pMe;
    return(SUCCESS);
}// End AppStateMachine_New
```

这里面最为值得注意的就是 pMe 依然使用了 MALLOC 来分配存储空间，这样的方式将让我们失去类的构造和析构函数的功能。关于这一点将在下一小节中有详细的介绍。剩余的初始化工作完全交给了 OnNew 函数处理，在这个函数中将为虚拟函数表填充函数：

```
int   CStateMachine::OnNew(IShell *pShell, IModule *po)
{
      VTBL(IStateMachine) *appFuncs;

      appFuncs = (VTBL(IStateMachine) *)((byte *)this + sizeof(CStateMachine));

      // 初始化函数表的内容
      appFuncs->AddRef            = AddRef;
      appFuncs->Release           = Release;
      appFuncs->HandleEvent       = HandleEvent;
      appFuncs->Start             = Start;
      appFuncs->Stop              = Stop;
      appFuncs->Suspend           = Suspend;
      appFuncs->Resume            = Resume;
      appFuncs->GetState          = GetState;
      appFuncs->MoveTo            = MoveTo;
      appFuncs->OpenWnd           = OpenWnd;
      appFuncs->CloseWnd          = CloseWnd;
      appFuncs->GetWndRet         = GetWndRet;

      AEEINITVTBL(this, IStateMachine, *appFuncs);      // Initialize the VTBL

      // 增加 IShell 指针的引用计数
      ISHELL_AddRef(pShell);
      IMODULE_AddRef(po);

      // 初始化状态机内部数据
      m_pShell         = pShell;
      m_pMod           = po;
      m_dwAppClsID = ISHELL_ActiveApplet(m_pShell);
      m_nRefs          = 1;
      m_pfnState     = NULL;              // 状态处理函数
      m_pUserState = NULL;              // 用户数据
      m_pfnEvent      = NULL;             // 事件处理函数
      m_pUserEvent = NULL;              // 用户数据
      m_eWndRet       = 0;                // Windows 返回值
      m_wWndParam    = 0;                 // Windows 返回值 word 参数
      m_dwWndParam = 0;                  // Windows 返回值 dword 参数
      m_bSuspending= FALSE;             // 当前状态机是否被挂起
      return SUCCESS;
}// End OnNew
```

从这段程序中我们可以看到，其内部的实现除了使用了“this”这个 C++特色的指针之外，与 C 语言相比就是少了对于 pMe 的调用。内部的实现和初始化的内容都没有什么明显的不同。

12.7.3 new 和 delete

上一节当中我们讲到了在为类分配存储空间的时候，我们使用了 BREW 的 MALLOC，而没有使用 C++的 new 来分配类空间，这样做的原因就是因为 BREW 不允许链接任何 C++的库函数。我们知道，在 C++中 new 操作符不但为类分配存储空间，而且也会调用类的构造函数来初始化类的成员变量。使用 MALLOC 导致了在为类分配空间的时候将无法调用类的构造函数，同样的，也就无法调用类的析构函数。这说明了使用 C++开发 BREW 接口是以牺牲 C++的部分特性作为代价的。

当然，对于当前的类，我们可以重载相应的 new 和 delete 操作符，从而达到在形式上与 C++靠拢的目的。对应的我们可以在类定义中增加如下内联函数：

```
public:
    void *operator new(size_t size){return MALLOC (size);}
    void operator delete(void * ptr){FREE(ptr);}
    void *operator new[](size_t size){return MALLOC (size);}
    void operator delete[](void * ptr){FREE(ptr);}
```

这里面分别重载了 new、new[]、delete 和 delete[]操作符，这样做分别实现了重载分配单一类空间和分配类数组的功能。作为一条准则，我们希望在开发 BREW 接口的过程中，每一个相关的类都重载 new 和 delete 运算符。

通常，在模拟器上面，直接使用 new 和 delete 所带来的问题并不明显，而且程序依然可以正常的运行。但是，当我们要为目标设备生成可执行程序的时候，一个链接错误就将产生了：

```
Error: L6265E: Non-RWPI Section libspace.o(.bss) cannot be assigned to PI Exec region ER_ZI.
Error: L6248E: libspace.o(.text) in PI region 'ER_RO' cannot have address type relocation to __libspace_start in PI region 'ER_ZI'.
```

12.7.4 使用 C++开发应用程序

前面讲了如何使用 C++扩展 BREW 接口的问题，同样的，我们也可以使用 C++开发 BREW 应用程序。在开发应用程序的时候，需要进行如下的修改：

1、定义应用程序类 MyApp 继承 AEEApplet 结构体：

```
//类定义
class MyApp : public AEEApplet
{
}
```

2、在 MyApp 类中封装 HandleEvent、InitAppData 和 FreeAppData 三个函数：

```
{
public:
   static boolean    HandleEvent(void * po, AEEEvent eCode,uint16 wParam, uint32 dwParam);
   static void FreeAppData(CPPApp * pCPPApp);
   static boolean    InitAppData(IApplet *pIApplet);
private:
   boolean    OnAppInitData();
```

```
    void        OnAppfreeData();
    boolean    OnEvent(AEEEvent eCode, uint16 wParam, uint32 dwParam);
}
```

3、仿照扩展接口中的方法，使得外控函数 HandleEvent 等直接调用 OnEvent 等内部函数实现功能：

```
boolean CPPApp::HandleEvent(void *po, AEEEvent eCode, uint16 wParam, uint32 dwParam)
{
    MyApp *pMe = (MyApp *)po;
    return pMe->OnEvent(eCode, wParam, dwParam);
}
```

4、改写 AEEClsCreateInstance 函数为类似如下形式：

```
int AEEClsCreateInstance(AEECLSID clsID, IShell* pIShell, IModule* pIModule, void **ppobj)
{
    if(clsID == AEECLSID_MYAPP){
        if(!AEEApplet_New(sizeof(MyApp),
                          clsID,
                          pIShell,
                          pIModule,
                          (IApplet**)ppobj,
                          (AEEHANDLER)MyApp::HandleEvent,
                          (PFNFREEAPPDATA)MyApp::FreeAppData)){
            //注册 c++中实现的事件处理和程序数据释放函数
            return EFAILED;
        }

        if (!MyApp::InitAppData((IApplet *)*ppobj)){
            //初始化程序数据
            return EFAILED;
        }

        return SUCCESS;
    }
    return EFAILED;
}
```

经过上面的改写之后，我们就可以结合 BREW 接口和 C++来共同开发 BREW 应用程序了。在 C++的使用上面，同样存在着与扩展接口一样的限制。

12.8 小结

在本章里面，我们介绍了扩展 BREW 接口的方法，并且以 IStateMachine 接口为例，实际的展示了如何扩展一个接口。从这个过程中我们看到了从接口定义、接口初始化到接口实现的全过程。在本章中还讨论了如何使用 C++来扩展接口的问题，我们分析了面临的问题，最后给出了解决方案。希望各位读者注意的是，在 BREW 中使用 C++时对 new 和 delete 的

使用的限制。当然从本质上来讲 BREW 的应用和接口之间并没有太大的区别，因此，我们还介绍了关于使用 C++编写 BREW 应用程序的方法。这一章的主要目的是通过实际的示例，使得我们能够更加深入的了解 BREW，并掌握扩展 BREW 接口的方法。

思考题

1、在接口的初始化过程中，我们的示例程序里面使用了虚拟函数表和内部结构体一起分配内存的方式，问，除了这种方式之外还能采用什么方式？需要做哪些修改？

第十三章 BREW 图形系统

图形化是当今电子应用领域正在实现主题，小到一个电子表，大道一个计算机，各种消费类和应用类的电子产品无不在开发图形化的用户界面来取悦用户。基于此，一个软件开发平台的图形系统就十分重要了，必须有一个好的图形系统才能符合发展的潮流。自然的，BREW 做为一个嵌入式系统的开发平台，图形系统的搭建也显得尤为重要，尤其是在开发各种应用方面。

通常一个图形系统中包含文字、图形两部分。文字从本质上说也是一种图形，只不过是固定的一种图形输出。图形之中包含了位图、设备无关位图、各种图形格式的支持等内容，可以说是涉猎面十分广博。一个平台提供的这些功能多寡就成为衡量平台图形系统好坏的标准之一了，比如这个图形系统是否支持不规则图形的描绘，是否支持 3D 图形的描绘、是否支持图像的拉伸等等。这些功能存在与否直接决定了基于这个图形系统的图形开发能力。

当然，对于嵌入式系统来讲，图形系统的执行效率也是衡量图形系统的重要指标。有的时候，甚至要不惜牺牲功能来提高图形系统的效率，效率是嵌入式系统永恒的主题。

在 BREW 平台中，支持了常用的图形系统的功能，如文字的输出、位图的描绘、基本的图像解码等功能，这些功能共同组成了 BREW 的基本图形系统。在此基础之上，BREW 开发出了各种各样的接口，以便于开发者能够方便的使用这些功能。在这一章里，我们将就 BREW 图形系统中实现的功能来对图形系统有一个了解，着重于介绍基本图形系统的概念和原理。通过对这些概念和原理的理解，我们才能够更加灵活的使用这些功能。就是要让您不但知其然，而且要知其所以然。

13.1 位图（Bitmap）

位图是一个二维的数据位数组，它与图像的像素一一对应。当现实世界的图像被扫描成位图以后，图像被分割成网格，并以像素作为取样单位。在位图中的每个像素值指明了一个单位网格内图像的平均颜色。单色位图每个像素只需要一位二进制数表示，灰色或彩色位图中每个像素需要多个二进制位表示。

位图和矢量图在计算机图形处理世界中都占有一席之地。位图经常用来表示来自真实世界的复杂图像，例如数字化的照片或者视讯图像。矢量图更适合于描述由人或者机器产生的图像，比如建筑蓝图。位图和矢量图的区别在于位映像图像和向量图像之间的差别。位映像图像用离散的像素来处理输出设备；而向量图像用笛卡尔坐标系统来处理输出设备，其线条和填充对象能被个别拖移。现在大多数的图像输出设备是位映像设备，这包括视讯显示、点阵打印机、激光打印机和喷墨打印机。而笔式绘图机则是向量输出设备。

位图有两个主要的缺点。第一个问题是容易受设备依赖性的影响。最明显的就是对颜色的依赖性，在单色设备上显示彩色位图的效果总是不能令人满意的。另一个问题是位图经常暗示了特定的显示分辨率和图像纵横比。尽管位图能被拉伸和缩小，但是这样的处理通常包括复制或删除像素的某些行和列，这样会破坏图像的大小。而矢量图在放大缩小后仍然能保持图形样貌不受破坏。

位图的第二个缺点是需要很大的储存空间。例如，描述完整的 640×480 像素，16 色的视频图形数组（VGA：Video Graphics Array）屏幕的一幅位图需要大于 150 KB 的空间；一幅 1024×768，并且每个像素为 24 位颜色的图像则需要大于 2 MB 的空间。位图的储存空间由图像的大小及其包含的颜色决定，而矢量图的储存空间则由图像的复杂程度决定。

然而，位图优于矢量图之处在于速度。将位图复制给视讯显示器通常比复制基本图形文件的速度要快。最近几年，压缩技术允许压缩位图的文件大小，以使它能有效地传输并广泛地用于 Internet 的网页上。由于矢量图形在描绘是通常需要占用更多的处理时间，因此，在资源需求紧张的嵌入式系统中很少使用。BREW 平台也是一样，只是实现了位图的功能，而没有关于矢量图的功能，因此我们在这里不会再介绍关于矢量图的内容了，而以位图作为主要的阐述对象。

13.1.1 位图的来源

位图可以手工建立，例如，使用 Windows 附带的画图程序。一些人宁愿使用位映像绘图软件也不使用向量绘图软件。他们假定：图形最后一定会复杂到不能用线条跟填充区域来表达。位图图像也能由计算机程序计算生成。尽管大多数计算生成的图像能按向量图形储存，但是高清晰度的画面或碎形图样通常还是需要位图。

现在，位图通常用于描述真实世界的图像，并且有许多硬设备能让我们把现实世界的图像输入到计算机。这类硬件通常使用电荷耦合设备（CCD：charge-coupled device），这种设备接触到光就释放电荷。有时这些 CCD 单元能排列成一组，一个像素对应一个 CCD；为节约设备成本，通常只用一行 CCD 扫描图像。

在这些计算机 CCD 设备中，扫描仪是最古老的。它用一行 CCD 沿着纸上图像（例如照片）的表面扫描。CCD 根据光的强度产生电荷。模拟数字转换器（ADC：Analog-to-digital converters）把电荷转换为数字讯号，然后排列成位图。

现在，数字照相机和便携式摄像机的价格对于个人使用者来说开始变得负担的起了。它看起来很像普通照相机。但是数字照相机不使用底片，而用一组 CCD 来拦截图像，并且在 ADC 内部把数字图像直接储存在照相机内的内存中。通常，数字照相机与计算机的接口要通过串行或者 USB 端口。

在计算机系统的用户界面开发过程中，取得图片的来源需要开发者自行根据需求进行创作。有的时候需要寻求专业的图片制作人来制作位图，从而获得更好的效果。随着数码相机的普及，通过数码相机来获得位图已经越来越普遍了，即便您对编辑位图没有任何概念，只要您懂得使用相机就能够获得质量上乘的位图了。获得位图已经越来越容易了。

13.1.2 位图尺寸

位图呈矩形，并有空间尺寸，图像的高度和宽度都以像素为单位。例如，图 13.1 中的网格可描述一个很小的位图：宽度为 10 像素，高度为 6 像素，或者更简单地计为 10×6：

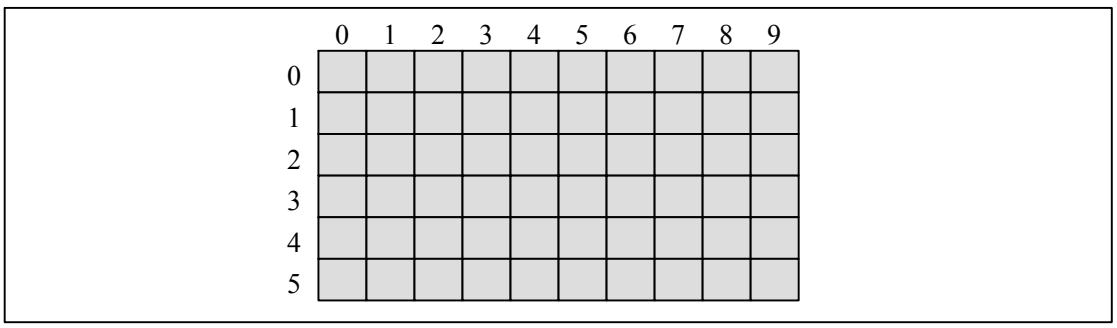

图 13.1 简单的位图

习惯上，位图尺寸是先给出宽度，再给出高度。上面的位图总数为 10×6 或者 60 像素。

在实际的编程开发过程中，将经常使用符号 cx 和 cy 来表示位图的宽度和高度。c 表示计数，因此 cx 和 cy 是沿着 x 轴（水平）和 y 轴（垂直）的像素数。

我们能根据 x 和 y 坐标来描述位图上具体的像素。一般（并不都是这样）在网格内计算像素时，位图开始于图像的左上角，也就是说位图的左上角是坐标系的起点(0, 0)。这样，在此位图右下角的像素坐标就是(8, 5)。因为从 0 开始计数，所以此值比图像的宽度和高度小 1。

位图的空间尺寸通常也指定了分辨率，但这是一个有争议的词。我们说我们的电脑显示有 1024×768 的分辨率，但是可能激光打印机的分辨率只有每英寸 300 点（300dpi）。现在，我们通常喜欢使用分辨率来描述一个图形设备（如显示器）在它的显示范围内，所能显示的像素数量。然而从“分辨率”这个词的意义上来说，它应该是指一个图形设备的分辨能力。例如我们在一个 1024 米×768 米的墙面上（当然没有这样的一个墙存在，仅仅假设而已）显示一个 1024×768 个像素的位图，那么，也就是说每个像素的大小是 1 米×1 米，巨大的一个马赛克啊！所以，分辨率更为准确的说法应该是单位尺寸内可以显示的像素。按照这个说法，分辨率越高所显示的图像就越细腻。当然对于同一个图形设备来说，设置不同的分辨率是有意义的，因为它的尺寸是固定的。不管怎样，当我们在提到分辨率的时候，应该是有明确的意义的。

位图是矩形的，但是计算机内存空间是线性的。通常（但并不都是这样）位图按列储存在内存中，且从顶列像素开始到底列结束。（DIB 是此规则的一个主要例外）。每一列，像素都从最左边的像素开始依次向右储存。这就好像储存几列文字中的各个字符一样。

13.1.3 颜色和位图

除空间尺寸以外，位图还有颜色尺寸。这里指的是每个像素所需要的位数，有时也称为位图的颜色深度（color depth）、位数（bit-count）或者位/像素（bpp：bits per pixel）数。位图中的每个像素都有相同数量的颜色位。

每像素 1 位的位图称为二阶（bilevel）、二色（bicolor）或者单色（monochrome）位图。每像素可以是 0 或 1，0 表示黑色，1 可以表示白色，但并不总是这样。对于其它颜色，一个像素就需要有多个位。可能的颜色值等于 2 位数值。用 2 位可以得到 4 种颜色，用 4 位可以得 16 种颜色，8 位可得到 256 种颜色，16 位可得到 65,536 种颜色，而 24 位可得到 16,777,216 种颜色。

每一个像素可以表示颜色的数目越多（也就是色深越大），那么表示这个像素所描绘的颜色就会与现实中的颜色更加接近。目前流行的 24 位颜色所显示的颜色数目已经足够的多了，已经达到了我们人眼所能分辨颜色差别的极限了。即便再增加色深，意义也不是很大了。

在这里介绍一下图形系统发展的历史，这些内容将有助于我们对图形系统的理解。这些内容必须从我们使用的 PC 机的常用 Windows 操作系统说起，因为正是这些 PC 操作系统带动了整个电子领域图形系统的发展。即便是现在最好的嵌入式系统的图形处理能力也不能和一个中档的 PC 机相比。

位图可按其颜色位数来分类；在 Windows 的发展过程中，不同的位图颜色格式取决于常用视频显卡的功能。实际上，我们可把视频显卡内存看作是一幅巨大的位图——我们从显示器上就可以看见。Windows 1.0 多数采用的显示卡是 IBM 的彩色图像适配器（CGA：Color Graphics Adapter）和单色图形卡（HGC：Hercules Graphics Card）。HGC 是单色设备，而 CGA 也只能在 Windows 以单色图形模式使用。单色位图现在还很常用（例如，鼠标的光标一般为单色），而且单色位图除显示图像以外还有其它用途。

随着增强型图形显示卡（EGA：Enhanced Graphics Adapter）的出现，Windows 使用者开始接触 16 色的图形。每个像素需要 4 个颜色位。（实际上，EGA 比这里所讲的更复杂，它还包括一个 64 种颜色的调色盘，应用程序可以从中选择任意的 16 种颜色，但 Windows 只按较简单的方法使用 EGA）。在 EGA 中使用的 16 种颜色是黑、白、两种灰色、高低亮度的红色、绿和蓝（三原色）、青色（蓝和绿组合的颜色）。现在认为这 16 种颜色是 Windows 的最低颜色标准。同样，其它 16 色位图也可以在 Windows 中显示。大多数的图示都是 16 色的位图。通常，简单的卡通图像也可以用这 16 种颜色制作。

在 16 色位图中的颜色编码有时称为 IRGB（高亮红绿蓝：Intensity-Red-Green-Blue），并且实际上是源自 IBM CGA 文字模式下最初使用的十六种颜色。每个像素所用的 4 个 IRGB 颜色位都映像为下表所示的 Windows 十六进制 RGB 颜色。

IRGB	RGB 颜色	颜色名称
0	00-00-00	黑
1	00-00-80	暗蓝
10	00-80-00	暗绿
11	00-80-80	暗青
100	80-00-00	暗红
101	80-00-80	暗洋红
110	80-80-00	暗黄
111	C0-C0-C0	亮灰
1000	80-80-80	暗灰
1001	00-00-FF	蓝
1010	00-FF-00	绿
1011	00-FF-FF	青
1100	FF-00-00	红
1101	FF-00-FF	洋红
1110	FF-FF-00	黄
1111	FF-FF-FF	白

接下来，IBM 最先发布了视频图像数组（Video Graphics Array：VGA）以及 PS/2 系列的个人计算机。它提供了许多不同的显示模式，但最好的图像模式（Windows 也使用其中之一）是水平显示 640 个像素，垂直显示 480 个像素，带有 16 种颜色。要显示 256 种颜色，最初的 VGA 必须切换到 320×240 的图形模式，这种像素数不适合 Windows 的正常工作。显示 256 种颜色的显卡模式采用每像素 8 位。不过，这些 8 位值都不必与实际的颜色相符。事实上，显示卡提供了“调色盘对照表（palette lookup table）”，该表允许软件指定这 8 位的颜色值，以便与实际颜色相符合。在 Windows 中，应用程序不能直接存取调色盘对照表。实际上，Windows 储存了 256 种颜色中的 20 种，而应用程序可以通过“Windows 调色盘管理器”来自订其余的 236 种颜色。关于调色盘的内容，将会在本章的第三节有所介绍。调色盘管理器允许应用程序在 256 色显示器上显示实际位图。Windows 所储存的 20 种颜色如表下所示。

IRGB	RGB 颜色	颜色名称
0	00-00-00	黑
1	80-00-00	暗红
10	00-80-00	暗绿

11	80-80-00	暗黄
100	00-00-80	暗蓝
101	80-00-80	暗洋红
110	00-80-80	暗青
111	C0-C0-C0	亮灰
1000	C0-DC-C0	美元绿
1001	A6-CA-F0	天蓝
11110110	FF-FB-F0	乳白
11110111	A0-A0-A4	中性灰
11111000	80-80-80	暗灰
11111001	FF-00-00	红
11111010	00-FF-00	绿
11111011	FF-FF-00	黄

最近几年，True-Color 显卡很普遍，它们在每像素使用 16 位或 24 位。有时每像素虽然用了 16 位，其中有 1 位不用，而其它 15 位则分别表示红、绿、蓝三原色。这样红、绿、蓝每种都有 32 色阶，组合起来就可以达到 32,768 种颜色。更普遍的是，6 位用于绿色（人类对此颜色最敏感），这样就可得到 65,536 种颜色。对于非技术性的 PC 使用者来说，他们并不喜欢看到诸如 32,768 或 65,536 之类的数字，因此通常将这种视讯显示卡称为 Hi-Color 显示卡，它能提供数以千计的颜色。

到了每个像素 24 位时，我们总共有了 16,777,216 种颜色（或者 True Color、数百万的颜色），每个像素使用 3 字节。这与今后的标准很相似，因为它大致代表了人类感官的极限而且也很方便。

13.1.4 嵌入式系统显示设备

嵌入式系统的显示设备就是随着 LCD 液晶显示器的发展而发展的。LCD 为英文 Liquid Crystal Display 的缩写，即液晶显示器，是一种数字显示技术，可以通过液晶和彩色过滤器过滤光源，在平面面板上产生图象。与传统的阴极射线管（CRT）相比，LCD 占用空间小，低功耗，低辐射，无闪烁，降低视觉疲劳。缺点是与同大小的 CRT 相比，价格更加昂贵。

早在 1888 年，人们就发现液晶这一呈液体状的化学物质，象磁场中的金属一样，当受到外界电场影响时，其分子会产生精确的有序排列。如果对分子的排列加以适当的控制，液晶分子将会允许光线穿越。LCD 显示屏都是由不同部分组成的分层结构。位于最后面的一层是由荧光物质组成的可以发射光线的背光层。背光层发出的光线在穿过第一层偏振过滤层之后进入包含成千上万水晶液滴的液晶层。液晶层中的水晶液滴都被包含在细小的单元格结构中，一个或多个单元格构成屏幕上的一个像素。当 LCD 中的电极产生电场时，液晶分子就会产生扭曲，从而将穿越其中的光线进行有规则的折射，然后经过第二层过滤层的过滤在屏幕上显示出来。

最初采用这种结构的液晶显示器只能够支持单色的显示，多应用在医疗器械等专用领域。后来随着 PC 系统的发展以及液晶显示技术的提高，出现了应用在笔记本电脑上的液晶显示屏。在彩色 LCD 面板中，每一个像素都是由三个液晶单元格构成，其中每一个单元格前面都分别有红色，绿色，或兰色的过滤器。这样，通过不同单元格的光线就可以在屏幕上显示出不同的颜色。彩色的 LCD 显示屏要求有强烈的背景灯光作为支撑，否则不能显示出

图像来，这就带来了彩色显视屏在阳光下显示不清的问题。

在 20 世纪 90 年代，随着移动通信设备的不断普及，手机逐渐成为了市场规模庞大的电子消费品。最初的手机都使用了单色的 LCD 显示屏，特点是耗电低，待机时间长。然而随着消费者个性化需求的增加，LCD 显示也从单色向彩色转变，而且屏幕的尺寸也越来越大。随之而来的是耗电大、待机时间变短。由于这种嵌入式设备的发展要比 PC 系统的晚，而且嵌入式系统的市场是一种零散的非垄断式的，因此导致了在嵌入式系统中没有统一的显示设备接口，这就为开发通用的操作系统平台带来了困难。由此而来的，各种各样的嵌入式图形界面开发平台就有了生存的空间。

在当今社会，互联网已经十分普及，新一代的移动通信系统也都增加了接入互联网的功能，因此嵌入式系统中的图形界面就变得更为重要了。在没有统一标准的情况下，各种平台都参考 PC 系统的方式，经过优化转移到了嵌入式系统中来。各种图形格式也都遵循着互联网的标准，而且由于 LCD 的显示设备是属于点阵形式，与位图更为接近，因此大多数的嵌入式系统都支持位图。

从总体上来说，嵌入式系统显示设备经历了同 PC 一样的发展历程，而且最初的 LCD 也是用在笔记本电脑上的。因此我们可以说嵌入式显示系统的发展只不过是 PC 发展的一个缩影。现在我们应该知道的是，嵌入式系统显示设备的色彩深度已经和传统的 PC 协调一致了，都已经支持到了 24 位色深。不过将彩色 LCD 应用于嵌入式系统设备还有耗电大、需要强背光支持等问题。

由于在嵌入式系统中显示设备通常使用的是 LCD 液晶显示器，而液晶显示器又是属于点阵形的显示工具，因此在这种设备上使用位图就再也平常不过了。同时，由于嵌入式系统中没有一个显示设备的标准，因此每一个在嵌入式系统中的平台不得不根据所使用的显示设备来重新提供设备驱动程序，进而在这个驱动的基础之上实现该系统的位图功能。例如，在 BREW 平台设备移植过程中，需要根据设备显示器件的颜色深度和大小实现在该设备上的位图功能。此时实现的位图功能与设备密切相关，因此我们叫做 DDB（设备相关位图：Device Dependent Bitmap）。与此对应的就是我们在下一节中要讲的 DIB（设备无关位图：Device Independent Bitmap），DIB 与 DDB 都属于位图，我们也可以把 DDB 理解成一种特殊的与设备兼容的 DIB。我们通常使用一个 DDB 与设备的显示内存关联起来，这样就可以把设备模拟成一个位图了。

13.1.5 位图操作

构建一个位图系统，需要支持一些基本的位图操作功能：位块传输、拉伸、位映像等功能。下面我们就来看看这些位图操作的含义。

位块传输是位图操作过程中最为基本的操作，用来进行位图的复制——将一个位图中的数据复制到另一个位图中，但不是简单的复制。在传统的 Windows 操作系统中，实现这个功能的函数叫做 Bitblt。Bitblt（读作“bit blit”）代表位块传输（bit-block transfer）。BLT 起源于一条汇编语言指令，该指令在 DEC PDP-10 上用来传输内存块。术语“bitblt”第一次用在图像上与 Xerox Palo Alto Research Center（PARC）设计的 SmallTalk 系统有关。在 SmallTalk 中，所有的图形输出操作都使用 bitblt。程序写作者有时将 blt 用作动词，例如：「Then I wrote some code to blt the happy face to the screen and play a wave file.」。BitBlt 函数移动的是像素，或者（更明确地）是一个位映像图块，但是传输（transfer）与 BitBlt 的意义不尽相同。此功能实际上对象素执行了一次位操作，并且可以产生一些特殊的效果，这些内容就属于位映像操作的范畴了。

在很多系统中（如 Windows 操作系统），在 BitBlt 内的最大限制是两个设备内容必须是兼容的。这意味着或者其中之一必须是单色的，或者两者的每个像素都相同的位数。而在 BREW 的 Bitmap 实现中，则完全去除了这些限制。相关的位块传输操作有两个接口函数：

```
int IBITMAP_BltIn(IBitmap * po, int xDst, int yDst, int dx, int dy, IBitmap *pSrc, int xSrc,
                  int ySrc, AEERasterOp rop ) ;
int IBITMAP_BltOut (IBitmap * po, int xDst, int yDst, int dx, int dy, IBitmap *pDst, int xSrc,
                    int ySrc, AEERasterOp rop ) ;
```

IBITMAP_BltOut 可用于辅助 IBITMAP_BltIn。如果目标位图不支持 IBITMAP_BltIn 操作，并且位图属于不同的类，则 IBITMAP_BltIn 的操作将由源位图（这里面指的是调用 IBITMAP_BltIn 时传递的 pSrc 参数）的 IBITMAP_BltOut 代为实现。通过这种方式，两类位图都有可能执行该操作，并且只要其中一类支持即可成功。在使用 IBITMAP_BltIn 时，源位图的格式与目标位图的格式可能相同，也可能不相同，但并不一定支持所有的源格式。例如，如果源位图 pSrc 是属于 17 位色深的位图（当然不存在这样奇怪的位图），而在目标位图中没有提供关于 17 位色深位图到自身位图（如 16 位色深）的转换功能，此时调用目标位图的 IBITMAP_BltIn 将不能够实现转换。当然，我们可以调用源位图的 IBITMAP_BltOut 功能来实现位块传输，因为可能源位图中提供了自身到目的位图的转换，不过这种调用通常是在 IBITMAP_BltIn 函数中完成的。在 IBITMAP_BltIn 中，一旦转换失败，则会调用源位图的 IBITMAP_BltOut 来实现位块传输。可以说，BREW 平台在这方面的设计是十分巧妙的。

位图操作的另一个常用功能就是位图的拉伸（Stretch）。使用拉伸会碰到一些与位图大小缩放相关的一些根本问题。在扩展一个位图时，必须复制像素的行或列。如果放大倍数不是原图的整数倍，那么此操作会造成产生的图像有些失真。如果目的矩形比来源矩形小，那么在缩小图像时就必须把两行（或列）或者多行（或列）的像素合并到一行（或列）。由于在进行拉伸操作时需要进行大量的运算，以保证最小的图像失真，因此在 CPU 运算能力有限的嵌入式系统中，通常不会提供这样的功能。BREW 也不例外，在 BREW 的位图操作中，也没有提供位图的拉伸功能。

我们现在已经知道在进行位块传输的时候，会进行一些格式匹配的转换，这属于位映像操作的一部分。除此之外，在位块传输的过程中我们还可以进行一些其他的位映像操作，如透明传输、色彩和成等功能，这些功能通常也叫做光栅操作。这些操作也是位图操作中需要进行支持的，这些功能的实现通常是在进行位块传输的过程中，通过指定参数的形式实现的。例如上面的两个 BREW 接口中，就使用最后一个参数 AEERasterOp 来指定需要进行的位映像操作。

除了上面的操作之外，在 BREW 平台中还提供了修改位图中某个点的功能，这些 API 函数可以让我们更加方便的操作位图。

13.2 设备无关位图（DIB）

因为 DDB 位图的位格式相当依赖于设备，所以 DDB 不适用于图像交换。DDB 内没有色彩对照表来指定位图的位与色彩之间的联系。DDB 只有在系统开机到关机的生命期内被建立和清除时才有意义。为了实现在各种系统中交换数据的需要，我们需要一种与设备无关的位图，这种位图可以不依赖于显示设备而拥有自己独立的数据格式，这种位图就是我们将要讲到的 DIB。

正如我们所知的，像.GIF 或.JPEG 之类的其它图像文件格式在 Internet 上比 DIB 文件更常见。这主要是因为.GIF 和.JPEG 格式进行了压缩，明显地减少了下载的时间。尽管有一个用于 DIB 的压缩方案，但极少使用。DIB 内的位图几乎都没有被压缩。如果我们想在程序

中操作位图，这实际上是一个优点。如果在内存中有 DIB，我们就可以提供指向该 DIB 的指标作为某些函数的参数，来把 DIB 转化成 DDB 在设备上显示。

13.2.1 DIB 文件格式

DIB 是一种文件格式，它的扩展名为.BMP，在极少情况下为.DIB。典型的在 Windows 操作系统中应用程序使用的位图图像被当做 DIB 文件建立，并作为只读资源储存在程序的可执行文件中。图标和鼠标光标也是形式稍有不同的 DIB 文件。

程序能将 DIB 文件减去前 14 个字节加载连续的内存块中。这时就可以称它为“packed DIB（packed-DIB）格式的位图”。程序也可以完全存取 DIB 的内容并以任意方式修改 DIB。程序也能在内存中建立自己的 DIB 然后把它们存入文件。

DIB 文件有四个主要部分：

1、文件表头

2、信息表头

3、RGB 调色盘对照表（不一定有）

4、位图像素位

我们可以把前两部分看成是 C 的数据结构，把第三部分看成是数据结构的数组。在内存中的 packed DIB 格式内有三个部分：

1、信息表头

2、RGB 调色盘对照表（不一定有）

3、位图像素位

除了没有文件表头外，其它部分与储存在文件内的 DIB 相同。DIB 文件（不是内存中的 packed DIB）以定义为如下结构的 14 个字节的文件表头开始：

```
typedef __packed struct _DIBFileHeader                  // DIB 文件头
{
    WORD            bfType ;            // 标记为"BM"或 0x4D42
    DWORD           bfSize ;            // 整个文件的尺寸
    WORD            bfReserved1 ;       // 保留，为 0
    WORD            bfReserved2 ;       // 保留，为 0
    DWORD           bfOffsetBits ;      // DIB 像素在文件中的偏移
} DIBFileHeader;
```

我们在这里使用了__packed 关键字来说明这个结构体，以表示这个结构体是紧凑型的，中间不能插入填充字节。经过这样声明的结构长度为 14 字节，它以两个字母“BM”开头以指明是位图文件。这是一个 WORD 值 0x4D42。紧跟在“BM”后的 DWORD 以字节为单位指出了包括文件表头在内的文件大小。下两个 WORD 字段设定为 0。结构还包含一个 DWORD 字段，它指出了文件中像素位开始位置的字节偏移量。此数值来自 DIB 信息表头中的信息，为了使用的方便提供在这里。

13.2.2 DIB 信息表头和调色盘对照表

最早的 DIB 文件起源于 OS/2 操作系统，在 DIBFileHeader 结构后紧跟了 DIBCoreHeader 结构，它提供了关于 DIB 图像的基本信息。紧缩的 DIB（Packed DIB）开始于 DIBCoreHeader：

```
typedef struct _DIBCoreHeader
{
    DWORD       bcSize ;          // 此结构体的尺寸 = 12
    WORD        bcWidth ;         // 图片的宽度（像素）
    WORD        bcHeight ;        // 图片的高度（像素）
    WORD        bcPlanes ;        // = 1
    WORD        bcBitCount ;      // 色深
}DIBCoreHeader;
```

“Core（核心）”用在这里看起来有点奇特，它是指这种格式是其它由它所衍生的位图格式的基础。DIBCoreHeader 结构中的 bcSize 字段指出了数据结构的大小，在这种情况下是 12 字节。bcWidth 和 bcHeight 字段包含了以像素为单位的位图大小。尽管这些字段使用 WORD 意味着一个 DIB 可能为 65,535 像素高和宽，但是我们几乎不会用到那么大的单位。bcPlanes 字段的值始终是 1。这个字段是早期 Windows 位图的残留物。

bcBitCount 字段指出了每像素的位数。对于 OS/2 样式的 DIB，这可能是 1、4、8 或 24。这样，bcBitCount 字段等于 1 代表 2 色 DIB、等于 4 代表 16 色 DIB、等于 8 代表 256 色 DIB 等等，依次类推。

对于前三种情况（也就是位数为 1、4 和 8 时），DIBCoreHeader 后紧跟调色盘对照表，16 和 24 位 DIB 没有调色盘对照表。调色盘对照表是一个 3 字节 RGBTriple 结构的数组，数组中的每个元素代表图像中的每种 RGB 颜色：

```
typedef struct _RGBTriple// rgbt
{
    BYTE rgbtBlue ;       // blue level
    BYTE rgbtGreen ;      // green level
    BYTE rgbtRed ;        // red level
}RGBTriple;
```

根据每个像素的位数，调色盘对照表的大小始终是 2、16 或 256 个 RGBTriple 结构。因为 RGBTriple 结构的长度是 3 字节，许多 RGBTriple 结构可能在 DIB 中以奇数地址开始。然而，因为在 DIB 文件内始终有偶数个的 RGBTriple 结构，所以紧跟在调色盘对照表数组后的数据块总是以 WORD 地址边界开始。紧跟在调色盘对照表（24 位 DIB 中是信息表头）后的数据是像素位本身。

现在我们掌握了 OS/2 兼容的早期 DIB 版本，同时也看一看 Windows 中 DIB 的扩展版本，因为这个版本目前应用的最为广泛。这种 DIB 形式跟前面的格式一样，以 DIBFileHeader 结构开始，但是接着是 DIBInfoHeader 结构，而不是 DIBCoreHeader 结构：

```
typedef __packed struct _DIBInfoHeader
{
    DWORD biSize ;              // 结构体的大小 = 40
    LONG    biWidth ;           // 图片的宽度（像素）
    LONG    biHeight ;          // 图片的高度（像素）
    WORD    biPlanes ;          // = 1
    WORD    biBitCount ;        // 色深(1, 4, 8, 16, 24, or 32)
    DWORD biCompression ;       // 压缩码
    DWORD biSizeImage ;         // 图像的字节数
    LONG    biXPelsPerMeter ;   // horizontal resolution
    LONG    biYPelsPerMeter ;   // vertical resolution
```

```
    DWORD biClrUsed ;                // 使用的颜色数量
    DWORD biClrImportant ;           // 重要的颜色数量
}DIBInfoHeader;
```

我们可以通过检查结构的第一字段区分与OS/2兼容的DIB和Windows DIB，前者为12，后者为40。我们将注意到，在这个结构内有六个附加的字段，但是DIBInfoHeader不是简单地由DIBCoreHeader加上一些新字段而成。仔细看一下：在DIBCoreHeader结构中，bcWidth和bcHeight字段是16位WORD值；而在DIBInfoHeader结构中它们是32位LONG值。这是一个令人讨厌的小变化，当心它会给我们带来麻烦。

另一个变化是：对于使用DIBInfoHeader结构的1位、4位和8位DIB，调色盘对照表不是RGBTriple结构的数组。相反，DIBInfoHeader结构紧跟着一个RGBQuad结构的数组：

```
typedef struct _RGBQuad
{
    BYTE rgbBlue ;        // blue level
    BYTE rgbGreen ;       // green level
    BYTE rgbRed ;         // red level
    BYTE rgbReserved ; // = 0
}RGBQuad ;
```

除了包括总是设定为0的第四个字段外，与RGBTriple结构相同。rgbReserved成员通常用在图像处理的软件中存储一些额外的信息。注意，如果DIBInfoHeader结构以32位的地址边界开始，因为DIBInfoHeader结构的长度是40字节，所以RGBQuad数组内的每一个项目也以32位边界开始。这样就确保通过32位微处理器能更有效地对调色盘对照表数据寻址。

biPlanes字段始终是1，但biBitCount字段现在可以是16或32以及1、4、8或24。

biCompression字段有两个用途：对于4位和8位DIB，它指出像素位被用一种运行长度编码方式压缩了；对于16位和32位DIB，它指出了颜色屏蔽是否用于对像素位进行编码。无论biCompression字段是什么，biSizeImage字段都指出了字节内DIB图素数据的大小。如果biCompression字段是0，则biSizeImage也通常为0。运行长度编码（RLE）是一种最简单的数据压缩形式，它是根据DIB映射在一列内经常有相同的图素字符串这个事实进行的。RLE通过对重复图素的值及重复的次数编码来节省空间，而用于DIB的RLE方案只定义了很少的矩形DIB图像，也就是说，矩形的某些区域是未定义的，这能被用于表示非矩形图像。

biXPelsPerMeter和biYPelsPerMeter字段以每公尺多少像素这种笨拙的单位指出图像的实际尺寸。应用程序能够利用它以准确的大小显示DIB。在大多数DIB内，这些字段设定为0，这表示没有建议的实际大小。

biClrUsed是非常重要的字段，因为它影响色彩对照表中项目的数量。对于4位和8位DIB，它能分别指出色彩对照表中包含了小于16或256个项目。虽然并不常用，但这是一种缩小DIB大小的方法。例如，假设DIB图像仅包括64个灰阶，biClrUsed字段设定为64，并且色彩对照表为256个字节大小的色彩对照表包含了64个RGBQuad结构。像素值的范围从0x00到0x3F。DIB仍然每像素需要1字节，但每个像素字节的高2位为零。如果biClrUsed字段设定为0，意味着色彩对照表包含了由biBitCount字段表示的全部项目数。biClrUsed字段对于16位、24位或32位DIB也可以为非零。在这些情况下，不使用色彩对照表解释像素位。相反地，它指出DIB中色彩对照表的大小，程序使用该信息来设定调色盘在256色视讯显示器上显示DIB。我们可能想起在OS/2兼容格式中，24位DIB没有色彩对照表，而在Windows中，24位DIB有色彩对照表，biClrUsed字段指出了它的大小。

总结如下：

1、对于 1 位 DIB，biClrUsed 始终是 0 或 2。色彩对照表始终有两个项目。

2、对于 4 位 DIB，如果 biClrUsed 字段是 0 或 16，则色彩对照表有 16 个项目。如果是从 2 到 15 的数，则指的是色彩对照表中的项目数。每个像素的最大值是小于该数的 1。

3、对于 8 位 DIB，如果 biClrUsed 字段是 0 或 256，则色彩对照表有 256 个项目。如果是从 2 到 225 的数，则指的是色彩对照表中的项目数。每个像素的最大值是小于该数的 1。

4、对于 16 位、24 位或 32 位 DIB，biClrUsed 字段通常为 0。如果它不为 0，则指的是色彩对照表中的项目数。执行于 256 色显示卡的应用程序能使用这些项目来为 DIB 设定调色盘。

另一个警告：原先使用早期 DIB 文件编写的程序不支持 24 位 DIB 中的色彩对照表，如果在程序使用 24 位 DIB 的色彩对照表的话，就要冒一定的风险。

biClrImportant 字段实际上没有 biClrUsed 字段重要，它通常被设定为 0 以指出色彩对照表中所有的颜色都是重要的，或者它与 biClrUsed 有相同的值。如果它被设定为 0 与 biClrUsed 之间的值，就意味着 DIB 图像能仅仅通过色彩对照表中第一个 biClrImportant 项目合理地取得。当在 256 色显示卡上并排显示两个或更多 8 位 DIB 时，这是很有用的。

13.2.3 DIB 像素存储

像大多数位图格式一样，DIB 中的图素位是以水平行组织的，用视讯显示器硬件的术语称作“扫描线”。行数等于 DIBCoreHeader 结构的 bcHeight 字段。然而，与大多数位图格式不同的是，DIB 从图像的底行开始，往上表示图像。

在此应定义一些术语，当我们说“顶行”和“底行”时，指的是当其正确显示在显示器或打印机的页面上时出现在虚拟图像的顶部和底部。就好像肖像的顶行是头发，底行是下巴，在 DIB 文件中的“第一行”指的是 DIB 文件的色彩对照表后的图素行，“最后行”指的是文件最末端的图素行。

因此，在 DIB 中，图像的底行是文件的第一行，图像的顶行是文件的最后一行。这称之为由下而上的组织。因为这种组织和直觉相反，您可能会问：为什么要这么做？DIB 创始人认为系统内的坐标系统——包括窗口、图形和位图——应该是一致的。这引起了争论：大多数人，包括在全画面文字方式下编程和窗口环境下工作的程序写作者认为应使用垂直坐标在屏幕上向下增加的坐标。然而，计算机图形程序写作者认为应使用解析几何的数学方法进行视频显示，这是一个垂直坐标在空间中向上增加的直角（或笛卡尔）坐标系。

简而言之，数学方法赢了。位图内的所有事物都以左下角为原点（包括窗口坐标），因此 DIB 也就有了这种由下而上的方式。

DIB 文件的最后部分（在大多数情况下是 DIB 文件的主体）由实际的 DIB 的图素字节成。图素位是由从图像的底行开始并沿着图像向上增长的水平行组织的。DIB 中的行数等于 DIBCoreHeader 结构的 bcHeight 字段。每一行的图素数等于该结构的 bcWidth 字段。每一行从最左边的图素开始，直到图像的右边。每个图素的位数可以从 bcBitCount 字段取得，为 1、4、8、16、24 或 32。以字节为单位的每行长度始终是 4 的倍数。行的长度可以计算为：

```
RowLength = 4 * ((bcWidth * bcBitCount + 31) / 32) ;
```

或者在 C 内用更有效的方法：

```
RowLength = ((bmch.bcWidth * bmch.bcBitCount + 31) & ~31) >> 3 ;
```

如果需要，可通过在右边补充行（通常是用零）来完成长度。图素数据的总字节数等于 RowLength 和 bcHeight 的乘积。

13.2.4 BREW DIB

在 BREW 内部，提供了对上面所讲述的 DIB 格式文件的解析支持，也就是说，我们可以通过 ISHELL_LoadBitmap 接口来载入.bmp 文件。载入后，BREW 会将其转换成 BREW 内部的 IDIB 结构：

```
struct IDIB {
    AEEVTBL(IBitmap) *pvt;              // 虚拟函数表指针
    IQueryInterface * pPaletteMap;      // cache for computed palette mapping info
    byte *              pBmp;             // 像素存储空间
    uint32 *            pRGB;             // 颜色表
    NativeColor         ncTransparent;  // 透明色
    uint16              cx;               // 位图的宽度（像素）
    uint16              cy;               // 位图的高度（像素）
    int16               nPitch;           // 每行之间的偏移（byte）
    uint16              cntRGB;           // 色彩对照表中颜色的数量
    uint8               nDepth;           // 色深
    uint8               nColorScheme;     // 颜色设置
    uint8               reserved[6];      // 保留字段
};
```

前两个成员我们不用去管它，就看后面的。pBmp 存储了经过转换之后的位图像素数据存储空间，采用的是以左上角为坐标原点的方式（与 Windows 中的不同）。pRGB 指向存储颜色对照表的存储空间。ncTransparent 是当前这个位图中的透明色，在描绘位图时如果指定了透明方式，就使用这个成员的值作为透明色，BREW 中默认的透明色是紫色。cx 和 cy 分别是位图的宽度和高度。nPitch 指定的是位图中每一行之间的偏移（按字节），这样可以减少位图操作中的运算，进而提高运行速度。cntRGB 表示当前色彩对照表中的颜色数量。nColorScheme 表示颜色配置，主要用于 8、12、16 和 24 位的位图（在 BREW 中可以支持多种位图色深）中，用来表示每个像素位中颜色的配置，比如 16 位色中使用的是 565 格式，还是 555 的 RGB 配色格式。

由于在 BREW 平台中，DIB 和 DDB 都属于位图的范畴，因此，这些接口都可以使用 IBitmap 接口进行操作。

13.3 调色板对照表

如果我们不考虑过去，那么本节就没有存在的必要。调色板对照表仅仅使用在 8 位色深以下的位图中，这些位图最多只能显示 256 种颜色。我们用 256 种颜色能做什么呢？很明显，要显示真实世界的图像，仅 16 种颜色是不够的，至少要使用数千或数百万种颜色，256 种颜色是位于中间状态。用 256 种颜色来显示真实世界的图像虽然不是十分的够用，但如果我们能够根据特定的图像来指定这些颜色，那么将足可以显示很多的图像。这意味着显示系统不能简单地选择标准系列的 256 种颜色，而希望它们对每个图像都使用理想的颜色。

这就是调色板对照表所要涉及的全部内容。它用于指定图像在 8 位或 8 位以下的位图格式中，可以在高位显示模式（如在真彩色的显示系统中）下显示更多的中间颜色。我们现在已经知道，对于一个 8 位色深以下的，位图文件来说都留有一个调色板对照表，在这个调色板对照表中存储的是一个 RGB 结构数组。我们可以通过在这个数组中指定独立于显示设备

的标准颜色值，从而可以在显示设备上显示尽可能多的颜色。

例如，在一个 1 位色深的位图文件中，每一个像素只有一个二进制位表示，那么它只可能是 0 或者 1。在一个同样是 1 位色深的显示设备中，只能显示黑或白两种颜色。那么我可以让这个位图在一个真彩色的显示设备上，显示红和蓝两种颜色吗？这种功能的实现就依赖于调色板对照表了。我们知道在单色的位图文件中，调色板对照表的长度是 2，也就是说我们可以在调色板对照表中存储两个 RGB 颜色信息。现在，我们就可以在这两个信息中，指定一个是蓝色，一个是红色，那么，我们现在就可以通过这个调色板对照表在可以显示红和蓝的显示设备上显示颜色了，这个时候，这个图可以叫做双色位图，而不是简单的黑白位图了。

同理，在 2、4 和 8 位色深的位图上，我们分别可以指定 4、16 和 256 种颜色信息。由于调色板对照表中指定的是真彩色的 RGB 值，因此，即便是这些低色深的位图仍然可以指定真彩颜色中的颜色值。例如，我们可以在颜色表中，指定一个颜色值 RGB(200,0,0)，在 256 色的显示设备中，这个颜色只能被显示为红色，但是如果在一个真彩色的显示设备中，这个颜色将是一个淡红色。这也是我们前面说“显示尽可能多颜色”的含义了。由于这种功能就像是画家在画画时使用的调色板，因此我们将位图文件中的这个表叫做调色板对照表。如果一个位图使用了调色板对照表，那么这些位图的像素区域存储的就是调色板对照表中颜色的索引值，而不是真正的颜色信息。通常一个系统中的调色板对照表使用方式如图 13.2 所示。

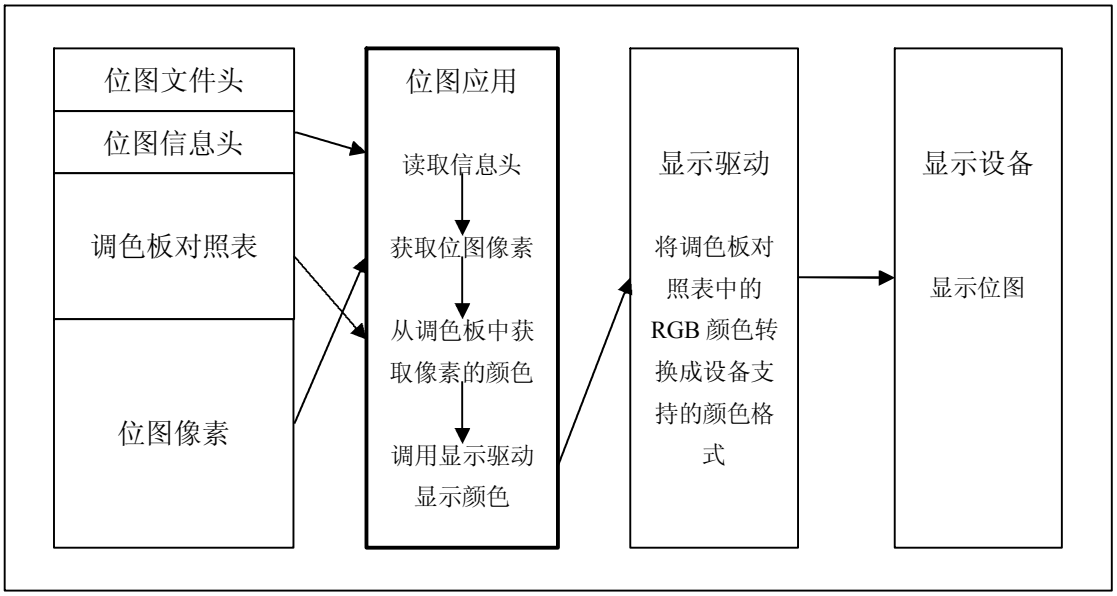

图 13.2 调色板对照表使用方式

处理位图显示的应用程序是位图处理的核心，它需要从位图文件中获取位图信息。首先获取位图文件的信息头部，从中可以计算出位图文件中位图像素数据存储的位置。对于使用调色板对照表的位图文件来说，这些像素数据中存储的是对应调色板数组中的颜色索引值。应用程序通过对应像素的索引值，找出这个像素对应的 RGB 颜色信息。在调色板对照表中存储的是一种标准的 RGB 颜色的数组，位图处理程序将这些信息传递给显示设备的驱动程序，驱动程序会将这些标准的颜色信息转换成设备所使用的颜色格式，这样，位图就可以显示在设备上了。

当然，与此位图文件相对应的位图编辑（或者创建）应用，也将遵循位图的这些要求，创建合理的位图文件。

13.4 文字（Font）

文字系统是一个图形显示系统的核心组建之一，由于文字不像图片那样的千变万化，因此，在一个平台中通常会提供固定的文字库，使得各个应用程序之间可以共享这些文字的显示图形。在这一节中您将会看到文字处理的来龙去脉。

13.4.1 字符集简史

虽然不能确定人类开始讲话的时间，但书写已有大约 6000 年的历史了。实际上，早期书写的内容是象形文字。每个字符都对应于发声的字母表则出现于大约 3000 年前。虽然人们过去使用的多种书写语言都用得好好的，但 19 世纪的几个发明者还是看到了更多的需求。Samuel F. B. Morse 在 1838 年到 1854 年间发明了电报，当时他还发明了一种电报上使用的代码。字母表中的每个字符对应于一系列短的和长的脉冲。虽然其中大小写字母之间没有区别，但数字和标点符号都有了自己的代码。

Morse 码并不是以其它图画的或印刷的象形文字来代表书写语言的第一个例子。1821 年到 1824 年之间，年轻的 Louis Braille 受到在夜间读写信息的军用系统的启发，发明了一种代码，它用纸上突起的点作为代码来帮助盲人阅读。Braille 代码实际上是一种 6 位代码，它把字符、常用字母组合、常用单字和标点进行编码。一个特殊的 escape 代码表示后续的字符代码应解释为大写。一个特殊的 shift 代码允许后续代码被解释为数字。

这些字符编码方式，为计算机系统提供了良好的借鉴。早期计算机的字符码是从 Hollerith 卡片（号称不能被折迭、卷曲或毁伤）发展而来的，该卡片由 Herman Hollerith 发明并首次在 1890 年的美国人口普查中使用。6 位字符码系统 BCDIC（Binary-Coded Decimal Interchange Code：二进制编码十进制交换编码）源自 Hollerith 代码，在 60 年代逐步扩展为 8 位 EBCDIC，并一直是 IBM 大型主机的标准，但没使用在其它地方。

美国信息交换标准码（ASCII：American Standard Code for Information Interchange）起始于 20 世纪 50 年代后期，最后完成于 1967 年。开发 ASCII 的过程中，在字符长度是 6 位、7 位还是 8 位的问题上产生了很大的争议。从可靠性的观点来看不应使用交叉替换字符（字符数太少了），因此 ASCII 不能是 6 位编码，但由于费用的原因也排除了 8 位版本的方案（当时每位的储存空间成本仍很昂贵）。这样，最终的字符码就有 26 个小写字母、26 个大写字母、10 个数字、32 个符号、33 个句柄和一个空格，总共 128 个字符码。ASCII 现在记录在 ANSI X3.4-1986 字符集－用于信息交换的 7 位美国国家标准码（7-Bit ASCII：7-Bit American National Standard Code for Information Interchange），由美国国家标准协会（American National Standards Institute）发布。

ASCII 码有许多优点。例如，26 个字母代码是连续的（在 EBCDIC 代码中就不是这样的）；大写字母和小写字母可通过改变一位数据而相互转化；10 个数字的代码可从数值本身方便地得到（在 BCDIC 代码中，字符「0」的编码在字符「9」的后面！）。最棒的是，ASCII 是一个非常可靠的标准。在键盘、视讯显示卡、系统硬件、打印机、字体文件、操作系统和 Internet 上，其它标准都不如 ASCII 码流行而且根深蒂固。

ASCII 的最大问题就是它只是一个真正的美国标准，所以它不能良好满足其它国家或者民族字符编码的需要。例如我们应该怎样表示一个中文字符？英语使用拉丁（或罗马）字母表。在使用拉丁语字母表的书写语言中，英语中的单词通常很少需要重音符号（或读音符号）。即使那些传统惯例加上读音符号也无不当的英语单字，例如 résumé，拼写中没有读音符号也会被完全接受。

但在美国以外，在语言中使用读音符号很普遍。这些重音符号最初是为使拉丁字母表适合这些语言读音不同的需要。在远东或西欧的南部，会遇到根本不使用拉丁字母的语言，例如希腊语、希伯来语、阿拉伯语和俄语（使用斯拉夫字母表）。更有甚者，汉字系统中的文字是非常多的，根本就与拉丁文字不兼容。

ASCII 的历史开始于 1967 年，此后它主要致力于克服其自身限制以更适合于非美国英语的其它语言。例如，1967 年，国际标准化组织（ISO：International Standards Organization）推荐一个 ASCII 的变种，代码 0x40、0x5B、0x5C、0x5D、0x7B、0x7C 和 0x7D 为“国家使用保留”，而代码 0x5E、0x60 和 0x7E 标为“当国内要求的特殊字符需要 8、9 或 10 个空间位置时，可用于其它图形符号”。这显然不是一个最佳的国际解决方案，因为这并不能保证一致性。但这却显示了人们如何想尽办法为不同的语言来编码的。

在小型计算机开发的初期，就已经严格地建立了 8 位字节。因此，如果使用一个字节来保存字符，则需要 128 个附加的字符来补充 ASCII。1981 年，当最初的 IBM PC 推出时，视频显卡的 ROM 中烧有一个提供 256 个字符的字符集，这也成为 IBM 标准的一个重要组成部分。最初的 IBM 扩展字符集包括某些带重音的字符和一个小写希腊字母表（在数学符号中非常有用），还包括一些块型和线状图形字符。附加的字符也被添加到 ASCII 控制字符的编码位置，这是因为大多数控制字符都不是拿来显示用的。

该 IBM 扩展字符集被烧进无数显示卡和打印机的 ROM 中，并被许多应用程序用于修饰其文字模式的显示方式。不过，该字符集并没有为所有使用拉丁字母表的西欧语言提供足够多的带重音字符，而且也不适用于 Windows。Windows 不需要图形字符，因为它有一个完全图形化的系统。

在 Windows 1.0（1985 年 11 月发行）中，Microsoft 没有完全放弃 IBM 扩展字符集，但它已退居第二重要位置。因为遵循了 ANSI 草案和 ISO 标准，纯 Windows 字符集被称作「ANSI 字符集」。ANSI 草案和 ISO 标准最终成为 ANSI/ISO 8859-1-1987，即“American National Standard for Information Processing-8-Bit Single-Byte Coded Graphic Character Sets-Part 1: Latin Alphabet No 1”，通常也简写为“Latin 1”。

MS-DOS 3.3（1987 年 4 月发行）向 IBM PC 用户引进了代码页（code page）的概念，Windows 也使用此概念。代码页定义了字符的映像代码。最初的 IBM 字符集被称作代码页 437，或者“MS-DOS Latin US”。代码页 850 就是“MS-DOS Latin 1”，它用附加的带重音字母代替了一些线形字符。其它代码页被其它语言定义。最低的 128 个代码总是相同的；较高的 128 个代码取决于定义代码页的语言。

在 MS-DOS 中，如果用户为 PC 的键盘、显卡和打印机指定了一个代码页，然后在 PC 上创建、编辑和打印文件，一切都很正常，每件事都会保持一致。然而，如果用户试图与使用不同代码页的用户交换文件，或者在机器上改变代码页，就会产生问题。字符码与错误的字符相关联。应用程序能够将代码页信息与文件一起保存来试图减少问题的产生，但该策略包括了某些在代码页间转换的工作。

虽然代码页最初仅提供了不包括带重音符号字母的附加拉丁字符集，但最终代码页较高的 128 个字符还是包括了完整的非拉丁字母，例如希伯来语、希腊语和斯拉夫语。自然，如此多样会导致代码页变得混乱；如果少数带重音的字母未正确显示，那么整个文字便会混乱不堪而不可阅读。

代码页的扩展正是基于所有这些原因，但是还不够。斯拉夫语的 MS-DOS 代码页 855 与斯拉夫语的 Windows 代码页 1251 以及斯拉夫语的 Macintosh 代码页 10007 不同。每个环境下的代码页都是对该环境所作的标准字符集修正。但中国、日本和韩国的象形文字符号有大约 21,000 个。如何在容纳这些语言的同时仍然保持和 ASCII 的某种兼容性呢？

解决的方法就是是双字节字符集（DBCS：double-byte character set）。DBCS 从 256 代码

开始，就像 ASCII 一样。与任何行为良好的代码页一样，最初的 128 个代码是 ASCII。然而，较高的 128 个代码中的某些总是跟随着第二个字节。这两个字节一起（称作首字节和跟随字节）定义一个字符，通常是一个复杂的象形文字。

虽然中文、日文和韩文共享一些相同的象形文字，但显然这三种语言是不同的，而且经常是同一个象形文字在三种不同的语言中代表三件不同的事。Windows 支持四个不同的双字节字符集：代码页 932（日文）、936（简体中文）、949（韩语）和 950（繁体汉字）。只有为这些国家（地区）生产的 Windows 版本才支持 DBCS。

双字符集问题并不是说字符由两个字节代表。问题在于一些字符（特别是 ASCII 字符）由 1 个字节表示。这会引起附加的程序设计问题。例如，字符串中的字符数不能由字符串的字节数决定。必须剖析字符串来决定其长度，而且必须检查每个字节以确定它是否为双字节字符的首字节。如果有一个指向 DBCS 字符串中间的指针，那么该字符串前一个字符的地址是什么呢？惯用的解决方案是从开始的指针分析该字符串！

我们面临的基本问题是世界上的书写语言不能简单地用 256 个 8 位代码表示。以前的解决方案包括代码页和 DBCS 已被证明是不能满足需要的，而且也是笨拙的。那什么才是真正的解决方案呢？

身为程序写作者，我们经历过这类问题。如果事情太多，用 8 位数值已经不能表示，那么我们就试更宽的值，例如 16 位值。而且这很有趣的，正是 Unicode 被制定的原因。与混乱的 256 个字符代码映像，以及含有一些 1 字节代码和一些 2 字节代码的双字节字符集不同，Unicode 是统一的 16 位系统，这样就允许表示 65,536 个字符。这对表示所有字符及世界上使用象形文字的语言，包括一系列的数学、符号和货币单位符号的集合来说是充裕的。

明白 Unicode 和 DBCS 之间的区别很重要。Unicode 使用（特别在 C 程序设计语言环境里）“宽字符集”。Unicode 中的每个字符都是 16 位宽而不是 8 位宽。在 Unicode 中，没有单单使用 8 位数值的意义存在。相比之下，在双字节字符集中我们仍然处理 8 位数值。有些字节自身定义字符，而某些字节则显示需要和另一个字节共同定义一个字符。

处理 DBCS 字符串非常杂乱，但是处理 Unicode 文字则像处理有秩序的文字。您也许会高兴地知道前 128 个 Unicode 字符（16 位代码从 0x0000 到 0x007F）就是 ASCII 字符，而接下来的 128 个 Unicode 字符（代码从 0x0080 到 0x00FF）是 ISO 8859-1 对 ASCII 的扩展。Unicode 中不同部分的字符都同样基于现有的标准。这是为了便于转换。希腊字母表使用从 0x0370 到 0x03FF 的代码，斯拉夫语使用从 0x0400 到 0x04FF 的代码，美国使用从 0x0530 到 0x058F 的代码，希伯来语使用从 0x0590 到 0x05FF 的代码。中国、日本和韩国的象形文字（总称为 CJK）占用了从 0x3000 到 0x9FFF 的代码。

Unicode 的最大好处是这里只有一个字符集，没有一点含糊。Unicode 实际上是个人计算机行业中几乎每个重要公司共同合作的结果，并且它与 ISO 10646-1 标准中的代码是一一对应的。Unicode 的重要参考文献是《The Unicode Standard，Version 2.0》（Addison-Wesley 出版社，1996 年）。这是一本特别的书，它以其它文档少有的方式显示了世界上书写语言的丰富性和多样性。此外，该书还提供了开发 Unicode 的基本原理和细节。

Unicode 有缺点吗？当然有。Unicode 字符串占用的内存是 ASCII 字符串的两倍。（然而压缩文件有助于极大地减少文件所占的磁盘空间。）但也许最糟的缺点是：人们相对来说还不习惯使用 Unicode。单就现有的平台来说，由于历史沿袭的关系，大多数是构建在 ASCII 和 DBCS 的基础之上的，这其中就包含 C 语言的开发环境。大车总是很难调头，因此，想要在计算机领域内统一使用 Unicode 编码还需要时间，不过在当前嵌入式系统中大部分的都已经使用了 Unicode 编码。

13.4.2 字符编码冲突

还是让我们从 ASCII 码说起。除了前面提到的标准 7 位 ASCII 码之外，为了表示更多的欧洲常用字符对 ASCII 进行了扩展，ASCII 扩展字符集使用 8 位（bits）表示一个字符，共 256 字符。ASCII 扩展字符集比 ASCII 字符集扩充出来的符号包括表格符号、计算符号、希腊字母和特殊的拉丁符号。

这些扩展的 ASCII 码带来了中、日、韩文的识别问题。例如，中文的文字编码规范叫做“GB2312-80”，它是和 ASCII 兼容的一种编码规范，其实就是利用扩展 ASCII 没有真正标准化这一点，把一个中文字符用两个扩展 ASCII 字符来表示。这种方法最大的问题就是，中文文字没有真正属于自己的编码，因为扩展 ASCII 码虽然没有真正的标准化，但是 PC 里的 ASCII 码还是有一个事实标准的（存放着英文制表符），所以很多软件利用这些符号来画表格。这样的软件用到中文系统中，这些表格符就会被误认作中文字，破坏版面。而且，统计中英文混合字符串中的字数，也是比较复杂的，我们必须判断一个 ASCII 码是否扩展，以及它的下一个 ASCII 是否扩展，然后才“猜”那可能是一个中文字。

于此同时，文化的差异导致了不同的语言文字的出现。对于西方文字，由于字符集较小，所以一般采用单字节编码（每个文字的编码长度为 8 位），而东方一些国家语言中文字数量较多，达数千个，通常称为大字符集文字，这些文字只有通过一个以上的字节才能实现其在计算机中的完全表示。这些大字符集文字主要是中、日、韩三国的文字。而且这些标准之间有很多重复的编码，即中、韩编码之间存在着全部冲突，中、日之间和日、韩之间存在着部分冲突。这样就在计算机对文字处理的时候面临着一个难题：当有中、日、韩三种字符串资源时，如何能在一台计算机上分辨出其所属的编码并将它们正确的显示出来。

这时候，我们就知道，要真正解决问题，不能从扩展 ASCII 的角度入手，也不能仅靠一个国家来解决。而必须有一个全新的编码系统，这个系统要可以将中文、英文、法文、德文……等等所有的文字统一起来考虑，为每个文字都分配一个单独的编码，这样才不会有上面那种现象出现。于是，Unicode 诞生了。Unicode 有两套标准，一套叫 UCS-2，用 2 个字节为字符编码，另一套叫 UCS-4，用 4 个字节为字符编码。 以目前常用的 UCS-2 为例，它可以表示的字符数为 2^16=65535，基本上可以容纳所有的欧美字符和绝大部分的亚洲字符 。

虽然我们现在已经有了 Unicode 编码，但是不可能在一夜之间所有的系统都使用 Unicode 来处理字符，所以 Unicode 在诞生之日，就必须考虑一个严峻的问题：和 ASCII 字符集之间的不兼容问题。我们知道，ASCII 字符是单个字节的，比如“A”的 ASCII 是 0x41。而 Unicode 是双字节的，比如“A”的 Unicode 是 0x0041，这就造成了一个非常大的问题：对于单字节字符串处理的语言使用'\0'作为字符串结尾（如 C 语言），而 Unicode 里恰恰有很多字符都有一个字节为 0，这样一来，这些字符串函数将无法正常处理 Unicode，除非把世界上所有的程序以及他们所用的函数库全部换掉。你我都知道，这是不可能的。

于是，比 Unicode 更伟大的编码方式诞生了，那就是：UTF（UCS Transformation Format UCS 转换格式）。它是将 Unicode 编码规则和计算机的实际编码对应起来的一个规则。现在流行的 UTF 有 2 种：UTF-8 和 UTF-16 。其中 UTF-16 和上面提到的 Unicode 本身的编码规范是一致的，这里不多说了。而 UTF-8 不同，它定义了一种“区间规则”，这种规则可以和 ASCII 编码保持最大程度的兼容。

UTF-8 有点类似于 Haffman 编码，它将 Unicode 编码为 00000000-0000007F 的字符，用单个字节来表示；00000080-000007FF 的字符用两个字节表示； 00000800-0000FFFF 的字符用 3 字节表示。因为 Unicode-16 编码中不会有 0xFFFF 以上的字符，所以 UTF-8 最多是使

用 3 个字节来表示一个字符。在 UTF-8 里，英文字符仍然跟 ASCII 编码一样，因此原先的函数库可以继续使用。在 Unicode 编码中，中文字符使用 0x4E00 开始到 0x9FFF 的区间范围，因此一个中文字符需要使用 3 个字节来表示。这样我们就可以解决 Unicode 编码与 ASCII 处理不兼容的问题了。

使用 Unicode 和 UTF 编码可以解决各国文字间的编码冲突和兼容 ASCII 编码的问题，但它不能解决各国文字的兼容处理问题。Unicode 在一定程度上改进了多国语言文字的处理能力，但是这种编码方案打乱了原来各国文字的编码，这种不兼容就是一个很难解决的问题，因为如果按照这个标准，那么原则上几乎所有与其不兼容的应用软件都应重写以适应固定大字符集编码方案的要求。这种重新编码而产生的不兼容性影响了其方案的推广应用。或许从某种角度来讲，这种不兼容的矛盾是无法调和的，我们只能够期望有一天全世界的人们都使用同一种字符编码了。

13.4.3 宽字符和 BREW

虽然在计算机系统中，需要考虑向后兼容性，应该支持单字节文字编码，尤其是在有着“悠久”历史的大型计算机系统中。但是，在后生小辈的嵌入式系统中，情况则完全不同了。由于大多数的嵌入式系统规模较小，因此支持什么样的字符编码将完全取决于系统应用环境。例如，在一个不需要显示文字的单片机系统中就根本不需要文字编码，而对于像手机这样公开的平台里则需要支持多种文字编码。

BREW 是构建在手机这样的大型嵌入式系统中的，它的文字编码可以通过在 BREW 移植的过程中进行定制，这也符合嵌入式系统的可定制原则。不过，在 BREW 内核部分，各种字符串大多是做为宽字符串来处理的。在 BREW 中，大部分的接口也都是支持款字符的。除了 BREW 的文件系统和应用程序启动参数，仍然使用单字节的字符来表示文件名和命令行参数外，其它部分主要都使用宽字符。

在 BREW 的开发环境内，定义了 AEECHAR 类型做位宽字符的类型定义。在 BREW 应用程序的资源文件处理过程中，BREW 完全使用了宽字符串做为内部字符串的交换格式。从这一点上说，BREW 的应用程序是属于构建在宽字符基础之上的。与此同时，在 BREW 的助手函数中，也增加了对宽字符串的处理，这位在应用程序开发过程中使用宽字符提供了极大的方便。具体的相关函数可以参考 BREW 的 API 参考文档。总之，在 BREW 中使用宽字符是十分方便的。

与此同时，BREW 还提供了对 UTF-8 编码的支持，这样可以做到与 ASCII 编码的兼容性。在 BREW 的助手函数中提供了 Unicode 到 UTF-8 编码的转换函数 WSTRTOUTF8，以及将 UTF-8 转换成 Unicode 的 UTF8TOWSTR 函数。我们可以在我们的应用程序中直接使用它们。

13.4.4 字形输出

在确定了字符编码之后，我们所要做的事情就是根据编码在显示设备上显示字形。显示字形的方式是从一个叫做字库的数据库中，查找所需字符编码的字形，然后输出到屏幕上显示。在嵌入式系统中通常使用点阵形式的字库，来做为字形输出的数据库。在整个字符显示的过程中，有字符码、字库索引、字库数据和显示设备等几个要素构成，它们之间的关系如图 13.3 所示。

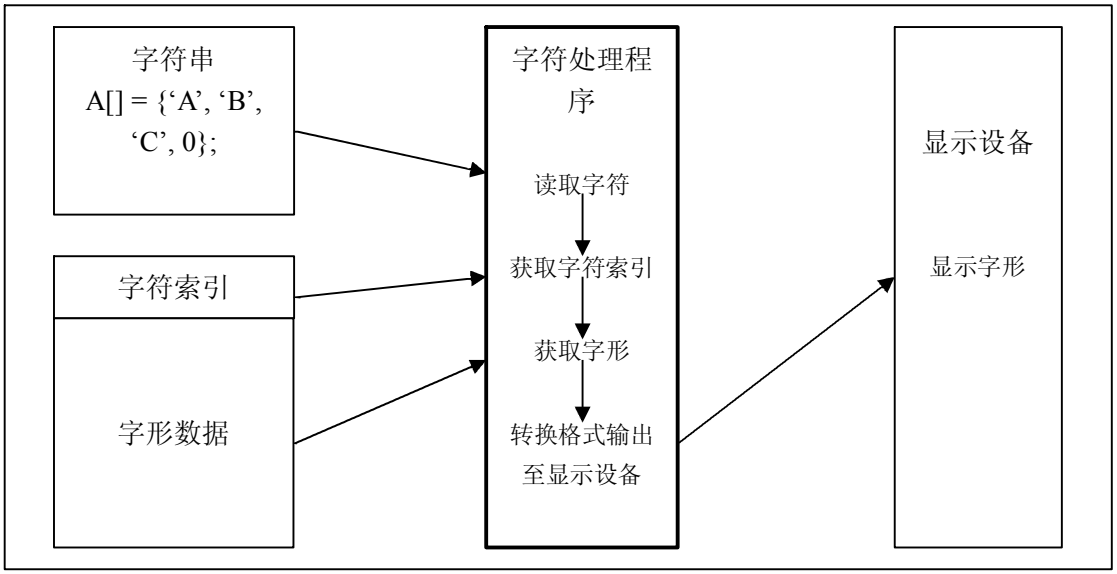

图 13.3 字符输出过程

字符处理程序通常由平台软件提供，例如，在 BREW 中就提供了 IDISPLAY_DrawText 方法和 IFont 接口来显示字符。字符处理程序首先获得显示字符的编码，如字符 A 对应的二进制编码值就是 0x41（在我们定义的字符串数组中仅仅存储了字符的编码值，而不是字形本身，虽然在源程序的文本编辑软件中我们输入了可见的字形，但那只不过是字符编辑软件根据字符编码显示的字形而已），字符处理程序根据这个数值查找字形数据库的索引列表，找到该字符在字形数据库中的位置，然后，再根据这个位置获取字形数据。

对于点阵型的字库，字形使用每一个二进制位表示字符中的一个显示点。这就类似一个单色的位图一样，只不过这个位图之中没有位图的信息头等内容。例如，对于一个 16*16 点阵的中文字符来说，这个字将被看作由 16*16 = 256bits 的数据位构成，并且按照 16 行和 16 列连续的排列，有形状的点用 1 表示，否则用 0 表示。

在获取了字形数据之后，需要将这些数据转换成显示设备可以使用的显示格式。由于字形数据仅相当于单色位图的像素存储区，因此我们可以借鉴位图操作的方法将字形转换成可以在设备上显示的设备位图格式，这样转换后的数据就可以在显示设备中显示了。

上面的整个过程是大多数点阵型字符显示方式所采用的，字形数据库文件就是每个字符字形数据的一个连接整体。或许您现在会问，如何获得这些字形的数据呢？有多种方式，其中最方便的一种是花钱购买，要么我们也可以自己画出这些字形。自己画？！您一定会感觉非常的沮丧。是啊，对于英文字库还好说，可是中文字库呢、韩文字库呢？太多的字形了，要数以万计，怎么画的出来！虽然我现在不能够保证您是否能够画出来，但是我可以向您保证的是第一个字库可定是人们手动一个一个画出来的。既然现在我们已经了解了它的困难，那么请怀有感激之心去使用那些现成的字库吧！

在这里我还可以向您提示一点，就在我们面前的 Windows 操作系统就是一个具有广阔字形资源的系统。我们可以通过一些 Windows 的函数来获得显示在屏幕上的那些字形的位图，这样我们就可以将这些字形的位图转化成单色的字库了。例如，我们可以创建一个窗口应用程序，然后在这个窗口中输出希望的文字，最后通过获得这个窗口的位图来获得字形的位图。还是那句话，请怀有感激之心来使用这些字形数据。

13.5 图像解码

在前面的章节中，我们已经介绍了关于位图的内容，其中着重讲述了原始的 DIB 位图

文件格式。然而随着网络的普及，由于 BMP 占用空间过大因而不适合于在网络中进行传输。为了解决这个问题，就出现了各种各样的位图压缩的方法，由此也产生了多种研所位图的格式，这其中包括 JPEG、GIF 等。

13.5.1 位图压缩算法

位图压缩算法主要有下面的几种：

（1）、行程长度压缩（RLE 算法）

RLE（Run-Length Encode）算法的原理是将一扫描行中的颜色值相同的相邻像素用一个计数值和那些像素的颜色值来代替。例如:aaabccccccddeee，则可用 3a1b6c2d3e 来代替。对于拥有大面积，相同颜色区域的图像，用 RLE 压缩方法非常有效。由 RLE 原理派生出许多具体行程压缩方法:

1、PCX 行程压缩方法：该算法实际上是位映射格式到压缩格式的转换算法，该算法对于连续出现 1 次的字节 V，若 V>0xC0 则压缩时在该字节前加上 0xC1，否则直接输出 V。对于连续出现 N 次的字节 V，则压缩（成 0xC0＋N, V）这两个字节，因而 N 最大只能为 0xff – 0xC0 = 0x3F(十进制为 63)。当 N 大于 63 时， 则需分多次压缩。

2、BI_RLE8 压缩方法：在 WINDOWS 的位图文件中采用了这种压缩方法。该压缩方法编码也是以两个字节为基本单位。其中第一个字节规定了用第二个字节指定的颜色重复次数，例如，编码 0302 表示从当前位置开始连续显示 3 个颜色值为 02 的像素。当第 1 个字节为零时第 2 个字节有特殊含义：0 表示行末；1 表示图末；2 转义后面 2 个字节，这两个字节分别表示下一像素相对于当前位置的水平位移和垂直位移。这种压缩方法所能压缩的图像像素位数最大为 8 位(256 色)图像。

3、BI_RLE 压缩方法：该方法也用于 WINDOWS 位图文件中，它与 BI_RLE8 编码类似，唯一不同是 BI_RLE4 的一个字节包含了两个像素的颜色，因此，它只能压缩的颜色数不超过 16 的图像。因而这种压缩应用范围有限。

4、紧缩位压缩方法：该方法是用于 Apple 公司的 Macintosh 机上的位图数据压缩方法，TIFF 规范中使用了这种方法，这种压缩方法与 BI_RLE8 压缩方法相似，如 1c1c1c2132325648 压缩为:83 1c 21 81 32 56 48，显而易见，这种压缩方法最好情况是每连续 128 个字节相同，这 128 个字节可压缩为一个数值 7f。这种方法还是非常有效的。

（2）、霍夫曼编码压缩：

霍夫曼编码也是一种常用的压缩方法。是 1952 年为压缩文本文件建立的，其基本原理是频繁使用的数据用较短的代码代替，很少使用的数据用较长的代码代替，每个数据的代码各不相同。这些代码都是二进制码，且码的长度是可变的。如：有一个原始数据序列，ABACCDAA 则编码为 A(0)、B(10)、C(110)、(D111)，压缩后为 010011011011100。产生霍夫曼编码需要对原始数据扫描两遍，第一遍扫描要精确地统计出原始数据中的每个值出现的频率，第二遍是建立霍夫曼树并进行编码，由于需要建立二叉树并遍历二叉树生成编码，因此数据压缩和还原速度都较慢，但简单有效，因而得到广泛的应用。

（3）、LZW 压缩方法：

LZW 压缩技术比其它大多数压缩技术都复杂，压缩效率也较高。其基本原理是把每一个第一次出现的字符串用一个数值来编码，在还原程序中再将这个数值还成原来的字符 串，如用数值 0x100 代替字符串"abccddeee"这样每当出现该字符串时，都用 0x100 代替，起到了压缩的作用。至于 0x100 与字符串的对应关系则是在压缩过程中动态生成的，而且这种对应关系是隐含在压缩数据中，随着解压缩的进行这张编码表会从压缩数据中逐步得到恢复，后

面的压缩数据再根据前面数据产生的对应关系产生更多的对应关系。直到压缩文件结束为止。LZW 是可逆的， 所有信息全部保留。

（4）、算术压缩方法：

算术压缩与霍夫曼编码压缩方法类似，只不过它比霍夫曼编码更加有效。算术压缩适合于由相同的重复序列组成的文件，算术压缩接近压缩的理论极限。这种方法，是将不同的序列映像到 0 到 1 之间的区域内，该区域表示成可变精度(位数)的二进制小数，越不常见的数据要的精度越高(更多的位数)，这种方法比较复杂，因而不太常用。

（5）JPEG（联合图片专家组 Joint Photographic Exprerts Group）：

JPEG 标准与其它的标准不同，它定义了不兼容的编码方法，在它最常用的模式中，它是带失真的，一个从 JPEG 文件恢复出来的图像与原始图像总是不同的，是属于有损压缩算法的一种。JPEG 的另一个显著的特点是它的压缩比例相当高，原图像大小与压缩后的图像大小相比，比例可以从 1%到 80～90%不等。这种方法效果也好，在多媒体系统系统中应用广泛。

13.5.2 压缩位图的格式

所谓文件格式一词简单地说就是保存计算机数据的方式。现在流行着许多不同的图像格式，它们都有独特的存储数据的方法。典型的有 JPEG、GIF 和 TIFF 等，下面我们就来看看这些压缩位图格式的特点。

（1）、JPEG：

JPEG 格式是现在使用最为广泛的格式之一，Mac 机和 Windows 系统的几乎所有程序都可以打开和保存 JPEG 图像。JPEG 还是互联网中图像处理时使用的主要两种文件格式之一。

JPEG 格式的优点之一是可以压缩图像数据，保证图像文件比较小。图像文件越小越节省磁盘空间，从网上下载时也就越节省时间。问题是 JPEG 使用的是有损压缩方案。这就是说有些图像数据在压缩过程中丢失了。你每打开、编辑和再保存图像一次，图像就重复被压缩一次，损失也就更多。

如果你需要使用 JPEG 格式保存图像，最好等到图像最后编辑完成再进行保存。在你对图像进行处理时，使用本机程序格式或 TIFF 格式（将在下一节中讨论）保存图像，这些格式能够保留所有重要的图像数据。只有在你完全结束图像编辑之后再保存成 JPEG。这种做法你只需要压缩图像一次，保证数据丢失限制到最小范围。

（2）、TIFF：

TIFF 的缩写全意为带标记的图像文件格式（Tagged Image File Format）。TIFF 也是互联网上最流行的一种图像文件格式。TIFF 图像可以在大多数 Macintosh 和 Window 程序中打开和使用。典型的，适用 TIFF 图像作为传真时所使用的图像格式。

TIFF 有时候使用 LZW 压缩方法。LZW 是一种无损压缩方式，在图像压缩时这种方式只废除一些无用的图像数据，所以它不会损失图像质量。然而，这种方式的图像压缩不能像用 JPEG 压缩方式那样大大降低图像文件的大小。这也是为什么 Tiff 格式并不能被认为是互联网上使用的最佳图像格式的原因。尽管大多数程序支持 LZW 压缩方式，但仍然有一小部分应用程序不支持。因此，如果你打开 TIFF 图像时存在问题，可能就出在压缩上面。这时可以试一下用其他程序打开再保存图像，以取消压缩。

（3）、GIF：

这种格式是 CompuServe 公司的公告牌服务机构开发并推广的一种图像传输格式。现在 GIF（图形交换格式，Graphics InterChange Format 的缩写）和 JPEG 已是万维网中应用最为

广泛的两种格式。GIF 格式的一个变种（正式叫 GIF89a），其特点之一是可以让你把图像的部分区域设置成透明的，这样可以透过图像看到 Web 网页的背景。

GIF 和 TIFF 一样都使用 LZW 压缩，这样可以减小文件尺寸，而不会丢失任何重要图像数据。GIF 的缺点是它局限于保存 8 位图像（256 色或更低），8 位图像让人们感到它比较粗糙、有斑点。

（4）、PNG：

PNG 的全称是便携网络图形（Portable Network Graphics）格式。它是一种很新的文件格式，由 Web Graphics 公司开发。这种格式和 GIF 格式不同，它可以不受 256 色的局限；和 JPEG 格式不同，它不使用有损压缩。这种格式的优点是使你得到高质量的图像；缺点是你的文件尺寸更大，这就是说在网上为了看到你的图像需要更长的下载时间。PNG 还处于发展期，网页制作和浏览器方面的支持软件还不多。现在对于在网页上使用的图像来说 GIF 和 JPEG 仍然还是比较好的格式，不过 PNG 图像的应用也正越来越广泛。

13.5.3 BREW IImage 接口

在 BREW 平台中，内嵌了对 BMP 和 PNG 图像的支持。而对于 JPEG 和 GIF 等图形格式来说，BREW 平台是否支持就需要看移植 BREW 的时候，是否添加了这些图形格式的解码支持。我们可以使用 ISHELL_LoadResImage 函数从一个资源文件中获取一个有效的 IImage 接口指针，或者使用 ISHELL_LoadImage 直接从文件加载图像，同时也获得一个 IImage 指针。当 BREW 支持我们所指定的图形格式时，我们将获得一个有效的 IImage 接口指针并在 BREW 内部调用该图形格式的解码器。如果获得了一个无效的空指针，那么表明 BREW 不支持我们所指定的图形格式或者我们传递的参数有错误。

IImage 接口是 BREW 当中的一个虚接口，就类似于 IApplet 接口一样，任何图形格式解码的实现都需要实现这个 IImage 接口中所定义的各种 API，或者使用更为“专业”的说法叫做继承 IImage 接口。在每种图形格式解码实现的内部，都需要创建一个 DDB 形式的位图，用来存储解码之后的数据。从前面的介绍我们可以知道，DDB 形式的位图可以直接输出到显示设备上显示图形，因此解码之后的图形可以很方便的输出到设备上进行显示。

或许现在已经有人在想了，既然在实现的内部已经获得了一个 DDB 形式的位图，那么，我们是否可以对这个位图显示之前进行一些特殊的处理呢？答案是不行。因为在 IImage 接口的内部，并没有给我们提供一个获得 IBitmap 接口的方法，因此我们没有办法直接对其进行操作。不过，幸运的是在 BREW3.0 的平台上 IImage 接口增加了一个 IIMAGE_SetDisplay 的接口。我们可以使用 IBITMAP_CreateCompatibleBitmap 接口自己创建一个设备兼容的 IBitmap 实例，然后将 IImage 刷新到这个 IBitmap 之中，这样我们就可以在刷新之前编辑这个位图了。具体实现的代码如下：

```
// 已经获得图片接口 IImage *pImage
IBitmap *pBmp1, *pBmp2;
AEEImageInfo imgInfo = {0};
IDIB * pDecodedDIB = NULL;

IIMAGE_GetInfo(pImage, &imgInfo);
IDISPLAY_GetDeviceBitmap(pme->a.m_pIDisplay, &pBmp1);

// 创建一个设备兼容的位图
```

```
IBITMAP_CreateCompatibleBitmap(pBmp1, &pBmp2, imgInfo.cx, imgInfo.cy);

// 保留原有的 IDisplay 位图
pBmp1 = IDISPLAY_GetDestination(pme->a.m_pIDisplay);

// 设置 IDisplay 的位图区域为新创建的
IDISPLAY_SetDestination(pme->a.m_pIDisplay, pBmp2);
IDISPLAY_ClearScreen(pme->a.m_pIDisplay);

// 向新位图中写入数据
IIMAGE_Draw(pImage, 0, 0);
IDISPLAY_UpdateEx(pme->a.m_pIDisplay, FALSE);

// 获取 IDIB 的图形数据
IBITMAP_QueryInterface(pBmp2, AEECLSID_DIB, (void **)&pDecodedDIB);

// 恢复 IDisplay 的原有位图
IDISPLAY_SetDestination(pme->a.m_pIDisplay, pBmp1);
IDISPLAY_UpdateEx(pme->a.m_pIDisplay, FALSE);
```

仔细阅读这段代码，或许它将给您带来惊喜，注释和功能我已经添加在代码中了。通过这样的操作之后，我们就在 pBmp2 种获取了该 IImage 接口的 DDB 位图了。在此之后，我们就可以使用 IDIB 接口中的 API 来实现修改这个图像内容的想法了。

13.6 BREW 图形系统结构

前面我们已经介绍了有关显示和字符的相关内容了，在这一节里面我们将剖析一下 BREW 的图形系统到底是一个怎样的结构。

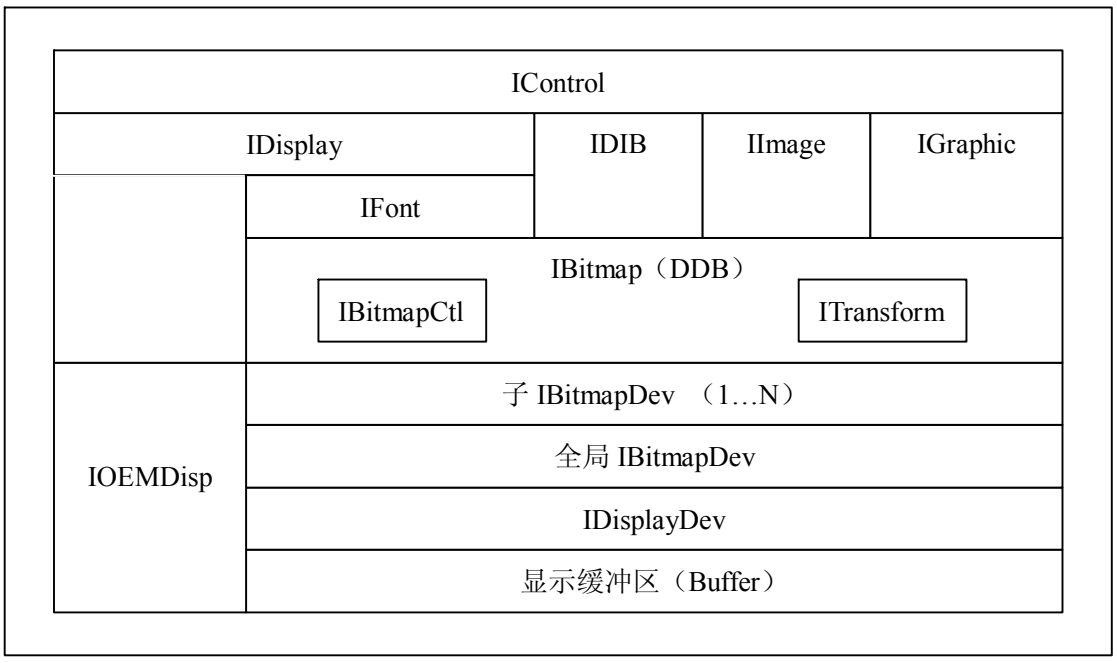

图 13.4 BREW 显示系统结构图

在 BREW 平台中，对于每一个显示屏，OEM 层应该实现一个 IDisplayDev 的接口，提供底层显示平台的设备分离，以便于对多个显示屏进行控制。IDisplayDev 不会对 BREW 应用程序公开。在 IDisplayDev 接口的实现中，除了继承自 IBase 的接口之外，还包含一个接口 IDISPLAYDEV_Update (IDisplayDev *po, IBitmap *pbmSrc, AEERect *prc)用来向指定的显示屏幕的缓冲区写入数据。其中 pbmSrc 是需要刷新设备的 IBitmap（DDB），prc 是需要刷新的显示区域。通过在 prc 中所指定的矩形刷新区域，可以使得底层的显示驱动不必每次刷新整个屏幕，只刷新这个区域就好了，这样可以提高屏幕刷新的速度。有的时候也管这个矩形叫做 Dirty-Rect，也就是脏矩形，意思是已经有程序在这个矩形区域内进行绘画了（也就是脏了）。

与 IDisplayDev 接口对应的，每个显示屏都会有一个 IBitmapDev 的全局实现以支持多设备显示。全局 IDisplayDev 的作用是创建并维护一个 IDisplayDev 所需要使用的 IBitmap 实例，今后全部针对此显示设备的操作都将使用这个 IBitmap 实例中的显示缓冲区。这个实例是一个典型的 DDB，是与现实设备完全相关的。在 BREW 中，DDB 与 DIB 之间没有明显的分别，DDB 通常只做为 DIB 的一个特殊子集而已，这样的结构可以大大简化系统的复杂程度。

子 IBitmapDev 的作用是创建一个与全局 IBitmapDev 中维护的 IBitmap 实例中共享显示缓冲区的新的 IBitmap。也就是说除了显示缓冲区相同之外，子 IBitmapDev 就是全局 IBitmapDev 的一个初始的结构。通过这些子的 IBitmapDev 接口，仍然可以刷新屏幕。这实际上是创建了多个 DDB 类型的位图结构。

BREW DDB 在经过封装之后，以 IBitmap 接口的形式共享给 BREW 上层。请注意，在 BREW 中通常 IDIB 和 IBitmap 接口之间是可以相互转化的，也就是说，实际上他们采用了相同的接口，只不过内部实现不一样而已。在 IBitmap 接口中，还包含了 IBitmapCtl 和 ITransform 两个接口。IBitmapCtl 用来控制当前的 IBitmap 实例能否被刷新。ITransform 用来进行图形变换处理，通过对专有显示设备（也就是本设备的显示器）的优化实现，可以提高 IBitmap 的变换速度，提高运行效率。例如我们可以专门针对 16 位色深的位图进行透明操作的优化等，只不过这些工作是 BREW 的 OEM 厂商完成的。

基于 BREW DDB，BREW 分别实现了 IDIB、IFont、IDisplay 和 IGraphic 等接口。在这些接口之上又构建了控件系统。这些就是 BREW 全部图形系统的组成结构。

13.7 小结

在本章中，我们首先介绍了位图的基本概念，以及位图的两种形式 DDB 和 DDB，并着重阐述了 DDB 位图的文件结构。接着介绍了文字系统的构成和通常使用的方法。最后介绍了位图的压缩格式以及 BREW 图形系统的结构。本章中介绍的这些内容是多数基于位图的图形系统都需要支持的，通过对这些相关知识的了解，将有助于我们了解其它的图形系统。

思考题

1、在我们的 C 语言中存储的字符是字形吗？为什么？
2、UTF-8 编码格式的优点是什么？

附录 A 缩略语

ADS	Application Download Server	应用程序下载服务器
ADS	ARM Development Suite	ARM 开发工具集
AEE	Application Execution Environment	应用程序执行环境
API	Application Programming Interface	应用程序编程接口
ASIC	Application Specific Integrated Circuit	专用集成电路
BAR	BREW Applet Resource File	BREW 应用程序资源文件的扩展名
BBF	BREW Binary Font	BREW 字体文件扩展名
BDS	BREW Distribution System	BREW 分发系统
BIOS	Basic Input Output System	基本输入输出系统
BREW	Binary Runtime Environment for Wireless	无线二进制运行环境
BRIDLE	BREW Isolated Domain for Legitimate Execution	BREW 一种基于 CPU 的安全程序执行方法，通过给不同层次代码不同的资源访问权限而实现
BRX	BREW Resource Intermediate in XML format	XML 格式的 BREW 资源文件
BST	BREWStone reference file	BREWStone 测试的参考文件扩展名
BWT	BREWStone weight file	BREWStone 测试权重文件
CB	Callback	回调函数
CDMA	Code Division Multiple Access	码分多址,基于扩频技术的一种崭新而成熟的无线通信技术
CGI	Common Gateway Interface	通用网关接口
CHIL	Chip Interface Layer	芯片接口层
CMX	Compact Multimedia eXtension Services	压缩多媒体扩展服务，美国高通公司的无线平台中实现多媒体功能软件集合的统称
COM	Component Object Model	组件对象模型
CPU	Central Processing Unit	中央处理器
DDF	Device data form	设备数据表
DDS	Device data sheet	设备数据薄
DIB	Device Independent Bitmap	设备无关位图
DLL	Dynamic Link Library	动态链接库，Windows 中存储和使用程序或资源的一种方式

EFS	Embedded File System	嵌入式文件系统
ESN	Electric Serial Number	电子序列号
GCC	GNU Cross Compiler	GNU 综合编译器
GUI		
ID	Identification	身份
IDE	Integrated Development Environment	集成开发环境
MD5	Message digest algorithm 5	一种加密算法
MIF	Module Information File	模块信息文件
MIL	Mobile Interface Layer	移动接口层
MO-SMS	Mobile originated SMS	发送短消息
OAT	Operational Acceptance Test	BREW OEM 层移植有效性测试
OEM	Original equipment manufacturer	原始设备制造商
OS	Operating System	操作系统
PEK	Porting Evaluation Kit	移植评估包，指 BREW 移植后测试用的软件包
PK	Porting Kit	BREW 的移植软件包
PME	Property－Method－Event-Driven	属性、方法和事件驱动模型
PST	Product Support Tool	产品支持工具
RAM	Random Access Memory	随机存取存储器
RO	Read Only	只读
ROM	Read Only Memory	只读存储器
RTOS	Real Time Operating System	实时操作系统
R-UIM	Removable user identity module	可移动用户身份模块
RW	Read Write	可读可写
SDK	Software Development Kit	软件开发包
SIG	Signature file	BREW 应用程序的签名文件扩展名
SMS	Short Message Service	短信息服务
TAPI	Telephone Application Programming Interface	电话应用程序接口
UCS	Unicode Sets	Unicode 字符集
UTF8	UCS Transformation Format-8	将 Unicode 转换成单字节编码的一种编码格式
VTBL	Virtual Table	虚拟函数表
ZI	Zero Initialize	零初始化

参考文献

在本书的写作过程中阅读和参考了大量的相关文章和文献，这些文献资料相信对于广大读者来说也有一定的参考价值，同时在这里列出来也是为了感谢这些文献的作者，是他们的辛勤劳动让我们有了通向知识之路的桥梁。

1、《COM 本质论》

【原 书 名】 Essential COM

【作 者】（美）Box，D. [同作者作品]

【译 者】 潘爱民[同译者作品]

【丛 书 名】开发大师系列

【出 版 社】中国电力出版社 【书 号】 7508306112

【出版日期】2001 年 8 月 【页 码】 358 【版 次】1-1

说明：这本书给作者理解 BREW 带来了很大的启发，对于理解 BREW 还是其他的程序都有很好的参考价值。

2、《Windows 程序设计》

【原 书 名】 Programming Windows (Fifth Edition)

【原出版社】 Microsoft Press

【作 者】（美）Charles Petzold [同作者作品] [作译者介绍]

【译 者】 北京博彦科技发展有限责任公司[同译者作品]

【丛 书 名】 Microsoft 程序设计系列

【出 版 社】 北京大学出版社 【书 号】 730104187X

【出版日期】 2004 年 9 月 【页 码】 1376 【版 次】1-8

说明：这本书介绍了很多计算机的基础知识，尤其是图形系统的说明部分，在本书中也参考了他的一些思路和内容。

3、《Apress Developing Software for the QUALCOMM BREW Platform》

说明：网络外文书籍，没有中译版本，是我认为比较好的 BREW 入门书。

后记与鸣谢

本书终于在匆忙之中简单的收尾了，由于笔者时间的关系，其中很多计划完成的章节都已经删减了比如国际化开发、电信接口的使用、BUIW、PEK 测试等等很多内容，我们只能期待在本书的下一版本中完善了。

从 2005 年 10 月起笔开始到现在这本书完成的过程中 BREW 也发展了很多，现在已经从 BREW3.0 发展到了 BREW4.0 甚至 BREW5.0 也已经有了时间表，但是正所谓万变不离其宗，不管 BREW 如何升级它的核心实现方式是永远不会过时的。对于技术学习我给各位读者的建议就是多读书、读好书，多读了才知道什么书是好书，然后在深入挖掘好书的内容，这样你的知识水平就可以得到真正的提高。

正所谓众口难调，笔者不期望这本书能够得到大家一致的认同，觉得好的留在床头经常看看，就是对笔者最大的支持；觉得不好的就多提建设性意见，也算是对笔者的一大支持。

由于本书的部分章节已经在我个人 Blog 上发表，其中不少网友提出了宝贵的意见，在此特别感谢 nntt、cx、thieven、ytree、hotechboy66 等网友。

在本书的结尾还要感谢我的妻子马雪，是她放弃了部分家庭时间，容忍我在业余时间完成这本书，同时也对本书的部分章节贡献了智慧和汗水，这本书的完成有她不少的功劳。